Deep learning for NLP

Joint text and image representations

Benoit Favre

24 Feb 2017

1 Introduction

For this tutorial, we will work with the MSCOCO image captioning dataset[1], in particular with the val2014 data which contains a set of 40k images annotated with five captions each. For example, one of the pictures is annotated with the following captions:

a male skier leaning as he goes down a snowy hill
a man in the snow on some skis
a man is skiing on the snow in the winter
a man riding skis down a snow covered slope
a skier skiing down a snowy ski slope

We intend to build a model that projects captions and images in the same representation space, so that an image is close to its captions in that space, and far away from dissimilar captions and dissimilar images. This space will be really neat because we will be able to use it to label unseen images by searching for the most similar caption to its representation, and we will also be able to search for the closest images given a new description.

Instead of using the 200,000 captions available in MSCOCO, we will use only 10,000 images and one caption per image.

To create image representations, we will start with a state-of-the-art image processing model which is able to recognize 1000 Imagenet[2] concepts in a picture with very good accuracy. We chop the last layer of this model and use the activations as input representation. We will call this processing feature extraction as it builds features for the images.

[1] `http://mscoco.org/`
[2] http://www.image-net.org/

Normally, we would include the whole image model as part as our neural network, initialize it with weights trained on the concept detection task, and fine-tune it as we train the other components of our model. But this part is very expensive to train, so we will use precomputed features for the training data.

For text representation, we will use a recurrent neural network with GRU units, and in particular the value of the last hidden state after reading the whole caption. The embedding layer will be initialized with GloVe embeddings.

Then we need a loss to tell our classifier to learn to transform the image representation and the text representation so that they end-up at the same location. It is a case of metric learning: We want the dot product between the image and text representations to be high when the caption and image match, and low when they don't. So our model will compute the dot product of the representations of two pairs, a matched (or positive) pair and a mismatched (or negative) pair. Then, we will be trained with a maximum margin loss which makes sure the positive pair has a (dot product) value higher than one plus the value of the negative pair.

Note that using a mismatched pair which has a dot product lower than the positive pair will not modify the model, so we need a good strategy to select the pairs. The best strategy would be to try them all and use the highest-scoring one, but that's too expensive to compute, so here we will just associate a random caption with the same image which was used in the positive pair.

This kind of loss is very similar to triplet ranking, a famous technique for learning how to rank examples given representations.

2 Image representation

In the following, we will use ResNet50 model implemented in Keras[3]. Once its decision layer has been removed, it generates a feature vector of size 2048. Since passing images through this 50-layer neural network is slow without a GPU, we provide you with a numpy matrix containing already computed features for the whole set of 10,000 images.

You can load that matrix with:

```
features = np.load('resnet50-features.10k.npy')
print(features.shape)
```

This should show a shape of (10000, 2048). The features for the i-th image can be accessed in `features[i]`.

Once the model will be trained, you will probably want to extract features for new images. You can do it with the following code which uses the modules available in `resnet50.py` and `imagenet_utils.py`:

```
from resnet50 import ResNet50
from keras.preprocessing import image
```

[3] https://github.com/fchollet/deep-learning-models

```
from imagenet_utils import preprocess_input

resnet_model = ResNet50(weights='imagenet', include_top=False)

def extract_features(img_path):
    img = image.load_img(img_path, target_size=(224, 224))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    features = resnet_model.predict(x)
    return np.expand_dims(features.flatten(), axis=0)

features = extract_features('some_image.png')
```

On first run, the `resnet50` module will load the weights for the model (about 100MB) and store them in `~/.keras/models`.

To compare text and image representations, they need to have the same size, so we will add a `Dense` layer which reduces the size from 2048 to 256. This layer will be trained as part of our model and generate image representations from ResNet50 features.

3 Text representation

The captions are stored in the `annotations.10k.txt` file. On each line, it contains an image name followed by one caption for that image. The text has already been tokenized, lower-cased and stripped of its punctuation (using NLTK). The lines of that file correspond to the rows of the matrix of precomputed image features. To load both, you could use the following function:

```
def load(captions_filename, features_filename):
    features = np.load(features_filename)
    images = []
    texts = []
    with open(captions_filename) as fp:
        for line in fp:
            tokens = line.strip().split()
            images.append(tokens[0])
            texts.append(' '.join(tokens[1:]))
    return features, images, texts
```

It returns a tuple containing the image features, the image names, and the caption texts.

We need to convert the texts to integer sequences of the same size. This can be performed with the `Tokenizer` from Keras, and the `pad_sequences` function. Most captions are short, so specifying a `maxlen` of 16 is reasonable.

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
captions = pad_sequences(sequences, maxlen=16)
```

It's the right time to save the vocabulary so that we can reload it later.

```
vocab = tokenizer.word_index
vocab['<eos>'] = 0 # add word with id 0

import json
with open('vocab.json', 'w') as fp: # save the vocab
    fp.write(json.dumps(vocab))
```

Our text representation model will input word sequences as arrays of integers, then project them to a 100-dimensional space with embeddings, and feed those embeddings to gated recurrent units (GRU). We will not use the output of the GRU to predict a word-level label, but rather only use the last hidden state as representation for the whole text. Before putting together the model, we can load a set of pretrained embeddings with the `embedding` module. These embeddings have been filtered to only contain the words used in the captions.

```
import embedding
embedding_weights = embedding.load(vocab, 100,
↪  'glove.twitter.27B.100d.filtered.txt')
```

4 Putting together the model

For that part, we need to use the functional api from Keras since it allows sharing weights between parts of a model. Here, we will build two pairs of representations that share some weights. At this point, it is a good idea to read about the functional api in the Keras documentation[4].

Our model will contain three inputs: an image features input (the output of the ResNet50 model), a correct caption input, and a noise caption input.

```
from keras.layers import Input, Dense, Embedding, GRU

image_input = Input(shape=(2048,))
caption_input = Input(shape=(16,))
noise_input = Input(shape=(16,))
```

[4] https://keras.io/getting-started/functional-api-guide/

Next we can setup the individual layers that are going to be used at several place in the model. We have an embedding layer which is initialized with GloVe embeddings, a GRU layer which creates the text representation and a dense layer which creates the image representation.

```
caption_embedding = Embedding(len(vocab), 100, input_length=16,
↪  weights=[embedding_weights])
caption_rnn = GRU(256)
image_dense = Dense(256, activation='tanh')
```

Now, we need to create the pipelines for each input by just calling the layers on their respective inputs. Note that this way the weights of the correct caption and noise pipelines are shared.

```
image_pipeline = image_dense(image_input)
caption_pipeline = caption_rnn(caption_embedding(caption_input))
noise_pipeline = caption_rnn(caption_embedding(noise_input))
```

To compute the dot product between the image and caption representations we need to use the `merge` function from Keras. In 'dot' mode, it computes the dot product between its arguments and returns a scalar. We then concatenate the results for both pairs and that's the output of the model which will be used by the loss function.

```
from keras.utils.layer_utils import merge

positive_pair = merge([image_pipeline, caption_pipeline],
↪  mode='dot')
negative_pair = merge([image_pipeline, noise_pipeline],
↪  mode='dot')
output = merge([positive_pair, negative_pair], mode='concat')
```

From the inputs and this output, we can create multiple models. First a model for training which outputs the concatenation of the result of the positive and negative pairs, then a model which only does the image pipeline and one which only does the caption pipeline. Those two models will be used at prediction time when we have novel images or captions and we want to compute their representations.

```
training_model = Model(input=[image_input, caption_input,
↪  noise_input], output=output)
image_model = Model(input=image_input, output=image_pipeline)
caption_model = Model(input=caption_input,
↪  output=caption_pipeline)
```

5 Custom loss

Keras does not provide a loss for maximizing the margin between a positive and negative example. So we have to write it. If we call p_i the score of the positive pair of the i-th example, and n_i the score of the negative pair of that example, the loss is

$$loss = \sum_i max(0, 1 - p_i + n_i) \tag{1}$$

It is very straightforward to implement in Keras thanks to its backend functions (functions from Theano and Tensorflow). The loss gets two arguments: a tensor containing the true label for the minibatch, and another tensor containing the model output. Our loss does not use the true label, only the model output, which corresponds to the concatenation of the positive score and the negative score.

```
from keras import backend as K
def custom_loss(y_true, y_pred):
    positive = y_pred[:,0]
    negative = y_pred[:,1]
    return K.sum(K.maximum(0., 1. - positive + negative))
```

While we are creating custom Keras functions, we can also compute an accuracy value: How many times did the positive pair effectively get a higher value than the negative pair? It's as easy as:

```
def accuracy(y_true, y_pred):
    positive = y_pred[:,0]
    negative = y_pred[:,1]
    return K.mean(positive > negative)
```

Once the `custom_loss` and `accuracy` functions are defined, they can be used to compile the model. Note that we only compile the `training_model`, the other models will not be manipulated again, they just represent a subset of the whole model.

```
training_model.compile(loss=custom_loss, optimizer='adam',
↪  metrics=[accuracy])
```

6 Training

For training, the model inputs three components: the ResNet50 features for the image (`features` hereafter), the correct captions (`captions`) and the mismatched captions (`noise`). It also expects labels, so we will create an array of zeros which is never used by the loss function (`fake_labels`). The `noise` matrix needs to be shuffled before each epoch so that the identity of the mismatched pair is not learned by the model.

To verify that the model is not overfitting, we will split the training data and create a validation set. The first 9,000 examples are used for training, and the last 1,000 for validation.

```
noise = np.copy(captions)
fake_labels = np.zeros((len(features), 1))

X_train = [features[:9000], captions[:9000], noise[:9000]]
Y_train = fake_labels[:9000]
X_valid = [features[-1000:], captions[-1000:], noise[-1000:]]
Y_valid = fake_labels[-1000:]

# actual training
for epoch in range(10):
    np.random.shuffle(noise) # don't forget to shuffle mismatched
↪   captions
    training_model.fit(X_train, Y_train,
↪   validation_data=[X_valid, Y_valid], nb_epoch=1,
↪   batch_size=64)
```

After a few epochs, an accuracy of more than 90% on the validation set should be obtained. This means that for 9 out of 10 images, the model outputs a higher score for the correct caption than for a random caption.

Once training is finished, we need to save the weights of the models used to create representations. The rest of the network does not contain any trainable weight so it can be thrown away. We shall also compute and save the representations of all images and captions for using them at prediction time.

```
# save models
image_model.save('model.image')
caption_model.save('model.caption')

# save representations
np.save('caption-representations',
↪   caption_model.predict(captions))
np.save('image-representations', image_model.predict(features))
```

7 Captioning novel images

Since training can be a bit long, code for the following sections can be written in a different python script. Note that some functions should be copied from above, and that models and other parameters (such as vocab and maxlen) should be loaded from the files they are saved in. In addition of models, we can load the representations we have computed for the whole dataset.

```
# load models
from keras.models import load_model
```

```
image_model = load_model('model.image')
caption_model = load_model('model.caption')

# load representations (you could as well recompute them)
import numpy as np
caption_representations = np.load('caption-representations.npy')
image_representations = np.load('image-representations.npy')
```

Instead of using the Tokenizer, we can write our own text to integer routine which uses the vocabulary we had saved in a json file.

```
from keras.preprocessing.sequence import pad_sequences
import json

vocab = json.loads(open('vocab.json').read())
def preprocess_texts(texts):
    output = []
    for text in texts:
        output.append([vocab[word] if word in vocab else 0 for
↪   word in text.split()])
    return pad_sequences(output, maxlen=16)
```

To generate a caption for a novel image, we need to load that image, extract features with ResNet50, pass them through the image part of our model to get a 256-value representation. Then, this representation can be compared to all the caption representations we have computed for the training data by computing the dot product between the matrix containing all caption representations and the image representation. This should result in a vector of scores in which indices correspond to captions. We can use numpy magic (`argpartition` and `argsort`) to retrieve the indices of the n highest scoring captions and display them to the user.

The following function assumes that you use the `extract_features` function to compute ResNet50 image features, that caption representations are loaded in the `caption_representations` matrix, and that `texts` contains the corresponding caption texts (use the `load` function from above).

```
def generate_caption(image_filename, n=10):
    # generate image representation for new image
    image_representation =
↪   image_model.predict(extract_features(image_filename))
    # compute score of all captions in the dataset
    scores = np.dot(caption_representations,
↪   image_representation.T).flatten()
    # compute indices of n best captions
    indices = np.argpartition(scores, -n)[-n:]
    indices = indices[np.argsort(scores[indices])]
    # display them
```

```
    for i in [int(x) for x in reversed(indices)]:
        print(scores[i], texts[i])
```

You can now generate a caption for one of the images in the validation set (the training set images would work as well, but what's the point generating a caption for an image we already have a caption for?).

```
generate_caption('images/COCO_val2014_000000301581.jpg')
```

```
9.35361 a shadowy skier skiing down a snowy mountain
9.00662 a person is snowboarding down a snowy slope
8.98038 a downhill skier shredding the slopes of snow
8.93038 a person in black jacket skiing on a slope
8.92909 a woman skiing on the snowy slopes
```

Not so bad, isn't it? If you try other images, you might not get as lucky, because our training data is not very diverse and the model is not very large. In addition, the model is limited to the list of existing captions. If you try to caption scenes which are not depicted in the training data you will get surprises.

8 Searching for images

In the same way you compared an image representation to all caption representations, you can input a novel caption, and ask what are the images which match the most this caption. Again, you will compute the representation for that caption, and take its dot product with the matrix containing representations for all images of the corpus.

The function is very similar as for generating a caption, except that it assumes that you can turn a textual caption to an array of integers of the correct shape to be input to the caption model with `preproces_texts`, and that `image_representations` contain the matrix of image representations, and `images` contains the name of the image files.

```
def search_image(caption, n=10):
    caption_representation =
↪   caption_model.predict(preprocess_texts([caption]))
    scores = np.dot(image_representations,
↪   caption_representation.T).flatten()
    indices = np.argpartition(scores, -n)[-n:]
    indices = indices[np.argsort(scores[indices])]
    for i in [int(x) for x in reversed(indices)]:
        print(scores[i], images[i])
```

To make it fancier, you could display the retrieved images.

```
search_image('a man in the snow on some skis')
```

9 Going further

The representations work for images and captions, so if you explore captions which are similar to a given caption, they should have the same meaning. Same applies for images. A nice extension could be to plot the t-SNE representation of both images and captions, for a sample of the data.

You can download the whole image set and train a model for the 200,000 captions (on GPU). At this point, you can refine the ResNet50 weights to better fit the problem. This was a key factor in the `NeuralTalk2` project to generate good captions with its conditioned language model.

Since the model quickly reaches an accuracy of 95%, this means that only 5% of the training instances actually modify the model. So a good improvement could be to find strategies for building negative pairs that are guaranteed to have a better score than the positive pair in order to improve the model every time. Note that it is easy to do this at a very high cost, but what about keeping training time low?