Image captioning

1 Introduction

Image captioning is a a term used in machine learning for the task of giving a textual desription of an image. In this assigmnent we will see how we can use RNNs for this task. In addition we will use a convolutional neural network to extract features from the image. The convolutional neural network will extract a feature map of size $7 \times 7 \times 1024$ for a an image of shape $224 \times 224 \times 3$. This feature extraction will be fixed throughout. We will then learn a projection of the 1024 - dimensional feature-vectors down to 256 - dimensional vectors, mainly due to computational considerations. Furthermore we will use *content-based soft attention* so that at each time step we transform the $7 \times 7 \times 256$ feature map into a 256 - dimensional vector that is a weighted average of the 49 feature vectors. Note that when using content-based attention, the vectors are just treated as a flat array without considerations to their spaital positions or relaitionships. If we want to use that kind of information we should use some spatial encoding scheme to embed that information into the feature vectors themselves. The idea is that at each time step we can attend to the location in the image that contains the relevant information for what we are currently describing.

There are **two** files included:

- **image_captioning.py** - Python script
- **image_captioning.ipynb** - Ipython notebook

The notebook may be opened locally using Jupyter or in the cloud with Google Colab (navigate to File –> Upload notebook). The notebook format may sometimes be easier to read, and work with interactively, so give it a try if you like. If you use Colab you may also use a GPU for free! Navigate to 'Edit –> Notebook settings" and then choose GPU as hardware accelrator.

Note that some summaries are already being logged into TensorBoard, so you probably want to check those out, and maybe add your own.

2 Implementation tasks

Our baseline attention mechansim (implemented in the class **UniformAttention** attentions equally to all parts of the image at every time step, i.e.

$$p_{t,j} = \frac{1}{49} \tag{1}$$

$$z^t = \sum_{j=1}^{49} p_{t,j} x_j \tag{2}$$

where z^t is the context vector given as input to the RNN at timestep t.

Your task is to improve upon this by implementing two other *trainable* attention schemes that estimates the attention weights $p_{t,j}$ in a way that we can pay more attention to relevant parts of the image. More specifically we shall try to learn a function f so that

$$\alpha_{t,j} = f(s^t, x_j) \tag{3}$$

$$p_{t,j} = \frac{e^{\alpha_{t,j}}}{\sum_k e^{\alpha_{t,k}}} \tag{4}$$

f should thus hopefully be able to figure out how relevant location j is at time step t. Note that the attention weights are just the softmax activation functions applied to the logits $\alpha_{t,j}$ at each time step.

2.1 Additive attention

The first method is an example of what is sometimes called *additive* attention. Here we replace f above with a feedforward neural network. In this problem you will implement a very simple feedforward network, following Neural Machine Translation by Jointly Learning to Align and Translate, which we shall denote *Bahdanau*-attention. The logits $\alpha_{t,j}$ are calculated as

$$\alpha_{t,j} = W \tanh(U s^t + V x_j + b) + c \tag{5}$$

for learnable matrices U, V, W and bias vectors b and c. Implement the attention mechanism given in (5) through a class, e.g. called `BahdanauAttention`, that inherits from `tf.keras.layers.Layer`. Train a model and try the `evaluate` function for different images and see if the attention weights seem to make sense. The `call` method of the class should have the same signature as that of `UniformAttention`, it should take two inputs:

- feature_vectors: tensor of shape [batch_size, 49, 256]

- state_output: tensor of shape [batch_size, 512]

and return

- context_vector : tensor of shape [batch_size, 256], which is the weighted average of the feature vectors.
- attention_weights: tensor of shape [batch_size, 49], which are the weights used in the averaging. These are returned for visualization purposes only.

2.1.1 Solution

```
class BahdanauAttention(layers.Layer):
  def __init__(self, units):
    super(BahdanauAttention, self).__init__()
    self.W1 = layers.Dense(units)
    self.W2 = layers.Dense(units)
    self.V = layers.Dense(1)

  def call(self, features, state_out):
    # features(CNN_encoder output) shape == (batch_size, 64, embedding_dim)

    # state_out shape == (batch_size, state_out_size)
    # state_out_with_spatial_axis shape == (batch_size, 1, state_out_size)
    state_out_with_spatial_axis = tf.expand_dims(state_out, 1)

    # score shape == (batch_size, 64, state_out_size)
    score = activations.tanh(self.W1(features) +
    ↪  self.W2(state_out_with_spatial_axis))

    # attention_weights shape == (batch_size, 64, 1)
    # you get 1 at the last axis because you are applying score to self.V
    attention_weights = activations.softmax(self.V(score), axis=1)

    # context_vector shape after sum == (batch_size, embedding_dim)
    context_vector = attention_weights * features
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights
```

2.2 Multiplicative/dot-product - attention

The second approch uses a more tailored to the estimation of attention weights. We learn a query matrix Q and key matrix K so that

$$\alpha_{t,j} = (Qs^t)^T K x_j \tag{6}$$

Implement the attention mechanism in (6) and try to train a model with it.

2.2.1 Solution

```
class DotproductAttention(layers.Layer):
  def __init__(self, embedding_dim):
    super(DotproductAttention, self).__init__()

    self.Q = layers.Dense(embedding_dim)
    self.K = layers.Dense(embedding_dim)

  def call(self, feature_vectors, state):
    """feature_vectors: [batch_size, num_feature_vectors, feature_units]"""

    # [batch_size, embedding_dim]
    q = self.Q(state)
    # [batch_size, num_feature_vectors, embedding_dim]
    k = self.K(feature_vectors)
    # [batch_size, embedding_dim] ==> [batch_size, 1, embedding_dim]
    q = tf.expand_dims(q, axis=1)
    # ==> [batch_size, feature_vectors]
    alpha = tf.reduce_sum(q*k, axis=-1)
    attention_weights = activations.softmax(alpha, axis=-1)

    # ==> [batch_size, feature_units]
    context_vector = tf.reduce_sum(tf.expand_dims(attention_weights,
    ↪  axis=-1)* feature_vectors, axis=1)

    return context_vector, attention_weights
```

3 Questions

3.1 Pretrained Convnet

We are using a convolutional neural network trained on imagenet to extract features, and their parameters are not adjusted during the optimization. What are some advantages and disadvantages to this approach compared to also updating these weights during optimization (or even starting with random weights)?

3.1.1 Solution

Starting with a pretrained model helps us immediately get a more semantically meaninful feature vector representing the information in the image.

Starting with random weights would probably require a lot more training data as we are vastly increasing the number of parameters we are optimizing over. Learning good image features starting from random weights may also be challenging in this situation, as the learning signal is much more noise (or less direct, if you will) compared to training the convnet on an image classification task. Finetuning the model could be an option, it might give better results given enough training data. The downside is however that it will be require much more computation and memory, as we can't extract the features once and cache them.

3.2 Pretrained word embedding

The embedding layer is initialized with random weights. Would it be possible to somehow use a pretrained embedding layer?

3.2.1 Solution

- Possible but some caveats: many existing embeddings may be trained on different tokenization, e.g. instead of splitting a sentence into words, may even have been split into subwords.
- See e.g. `https://keras.io/examples/nlp/pretrained_word_embeddings/`
- Also see `https://tfhub.dev/s?module-type=text-embedding`

3.3 Supervised learning

One can not in general say that there is a *correct* caption for a given image. Is *supervised* learning still an appropriate framework?

3.3.1 Solution

Yes, in general Y can be a stochastic function of X.

3.4 Baseline loss

Having a simple baseline loss during training is often useful for a sanity check that things behave as expected. Can you think of any models you should be able to beat?

3.4.1 Solution

- Probabilities according to word frequency.
- Probabilities according to word frequency by time step.
- Model trained on only captions (independent of images). In this way you can check if your model is not just learning to generate typical sentences.

3.5 Validation loss

What are some possible issues with our validation loss? Why do this validation loss not tell the whole story?

3.5.1 Solution

- We do teacher forcing during validation. When generating captions we may go outside the distribution we have trained on. The validation loss does not measure such effect.
- In general difficult to find any good automatic measures for test performance.

3.6 RNN cell vs layer

In TensorFlow, what is the difference between the `LSTM` and `LSTMCell` layers (or in general betwenn `RNN` and `RNNCell` layers?

3.6.1 Solution

The `RNN` layer is basically the application of an `RNNCell` to the whole time sequence. It is very convenient to use when

1. We are presented the whole input sequence at once.
2. We do not feed the outputs of the model as input at the next time step.

If either of the two conditions above are not satisfied we will usually have to create a custom loop where we iterate over the sequence and this is where the `RNNCell` us useful. Note that if you create your own custom RNN cell, you may create an `RNN` layer by giving your cell as input to the `RNN` class.