UDACITY

< Back to Computer Vision Nanodegree

Image Captioning

REVIEW

CODE REVIEW 2

HISTORY

Meets Specifications

You've done a **fantastic job** completing the Image Captioning Project!

You might want to check out the resources below to **further improve your model** and **solidify your understanding of the material**:

- Rich Image Captioning in the Wild
- Power of Combining CNN-RNN architectures
- Image Captioning and Visual Question Answering Based on Attributes and External Knowledge

Keep up the great work

Files Submitted

The submission includes model.py and the following Jupyter notebooks, where all questions have been answered:

2_Training.ipynb, and
3_Inference.ipynb.

Well done submitting all of the required files and answering all of the questions in the notebook!

model.py

The chosen CNN architecture in the `CNNEncoder` class in model.py makes sense as an encoder for the image captioning task.

🎉 Great choice selecting the `ResNet` architecture as a CNN Encoder!

Suggestion: you could also try out the Inception model for the CNN as implemented in the open source Show and Tell Image Captioning by Google

The chosen RNN architecture in the `RNNDecoder` class in model.py makes sense as a decoder for the image captioning task.

🎉 Well done crafting a LSTM model with an embedding layer!

2_Training.ipynb

When using the `get_loader` function in data_loader.py to train the model, most arguments are left at their default values, as outlined in Step 1 of 1_Preliminaries.ipynb. In particular, the submission only (optionally) changes the values of the following arguments: `transform`, `mode`, `batch_size`, `vocab_threshold`, `vocab_from_file`.

🎉 Nicely done correctly setting the arguments with reasonable values!

The submission describes the chosen CNN-RNN architecture and details how the hyperparameters were selected.

Great job describing the chosen CNN-RNN architecture!

Tips for further improvement:

- Consider using weight initialization and Batch Normalization for the CNN

```
class EncoderCNN(nn.Module):
    def __init__(self, embed_size):
        """Load the pretrained ResNet-152 and replace top fc layer."""
```

```
        super(EncoderCNN, self).__init__()
        resnet = models.resnet152(pretrained=True)
        modules = list(resnet.children())[:-1]      # delete the last fc layer.
        self.resnet = nn.Sequential(*modules)
        self.linear = nn.Linear(resnet.fc.in_features, embed_size)
        self.bn = nn.BatchNorm1d(embed_size, momentum=0.01)
        self.init_weights()

    def init_weights(self):
        """Initialize the weights."""
        self.linear.weight.data.normal_(0.0, 0.02)
        self.linear.bias.data.fill_(0)
```

The transform is congruent with the choice of CNN architecture. If the transform has been modified, the submission describes how the transform used to pre-process the training images was selected.

🎉 The transform implementation is congruent with the CNN architecture!

The submission describes how the trainable parameters were selected and has made a well-informed choice when deciding which parameters in the model should be trainable.

🎉 Nicely done describing the trainable and untrainable parameters in your answer! You have made a well-informed choice to stick with the default values and tweak if necessary

The submission describes how the optimizer was selected.

Great job choosing the Adam algorithm as an optimizer! Here are some of the benefits of the Adam algorithm:

- Straightforward to implement
- Computationally efficient
- Little memory requirements
- Invariant to diagonal rescale of the gradients
- Well suited for problems that are large in terms of data and/or parameters
- Appropriate for non-stationary objectives

- Appropriate for problems with very noisy/or sparse gradients
- Hyper-parameters have intuitive interpretation and typically require little tuning

The code cell in Step 2 details all code used to train the model from scratch. The output of the code cell shows exactly what is printed when running the code cell. If the submission has amended the code used for training the model, it is well-organized and includes comments.

🎉 Great job training your model in **Step 2!** The cell does have output of running the code and the code is well-organized!

3_Inference.ipynb

The transform used to pre-process the test images is congruent with the choice of CNN architecture. It is also consistent with the transform specified in `transform_train` in 2_Training.ipynb.

🎉 Superb! The entries from the output of the `sample` method do leverage the LSTM architecture to generate valid token indices

The implementation of the `sample` method in the `RNNDecoder` class correctly leverages the RNN to generate predicted token indices.

🎉 Superb! The entries from the output of the `sample` method do leverage the LSTM architecture to generate valid token indices

The `clean_sentence` function passes the test in Step 4. The sentence is reasonably clean, where any `<start>` and `<end>` tokens have been removed.

The `clean_sentence` function has been correctly implemented to remove the `<start>` and `<end>` tokens making the sentence reasonably clean.

The submission shows two image-caption pairs where the model performed well, and two image-caption pairs where the model did not perform well.

Yep, the submission does show 2 image-caption pairs for both `model performed well` and `model did not perform well` sections.

Suggestion: consider taking a look at the SPICE framework for automatically evaluating image captions

DOWNLOAD PROJECT

2 CODE REVIEW COMMENTS ›

RETURN TO PATH

Rate this review