UDACITY

Return to Classroom

Image Captioning

REVIEW

CODE REVIEW 5

HISTORY

Meets Specifications

Great submission 🎉 🎉 ! Now you can think about taking this to the next step. Suggestions:
In addition to the suggestions given in the review comments here are some more:

1. Fine-tune the hyperparameters for the given dataset and the model for better accuracy and performance. Start with standard values and use grid search for fine-tuning. https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/
2. Experiment with transfer learning for the embedded layer. You can use one from here: https://nlp.stanford.edu/projects/glove/ check out the section "Download pre-trained word vectors".
3. Experiment with transfer learning: using different models, fine-tuning different layers and how the hyper-parameters are adjusted accordingly.
4. Explore what each layer is doing, http://yosinski.com/deepvis
5. Think about how to increase accuracy? How to regularize the network? Think about how to make the training faster and improve model performance?
6. Try the model on a different dataset.
7. Read more related research papers and implement variations and solutions.
   Example: https://cs224d.stanford.edu/reports/msoh.pdf
   https://arxiv.org/abs/1505.00468v7
8. Create an app.
   All the best!! 👍 👍

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.

All required files are submitted. Thank you. 👍

model.py

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

Resnet 50 is a good choice. Read about the power of residual learning here
✅ The pretrained Resnet model is used correctly. The new trainable FC layer replaces the last layer to create the feature vector. Well done.

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

The batch_first is set to True. well done!
✅ Good work with removing <end> tag and embedding of captions and concatenation with Encoder embed vector.
✅ The Embedding layer, the LSTM and the fully-connected layers are implemented correctly in the Decoder.

Read suggestion for LSTM parameter: https://discuss.pytorch.org/t/could-someone-explain-batch-first-true-in-lstm/15402

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`.

The hyper parameters are given standard values. Good choice.

- batch_size = 512. Mostly the batch size depends on the GPU resources available. Other than that if you choose a batch size too small then the gradients will become more unstable and would need to reduce the learning rate. So batch size and learning rate are linked. Also if one use a batch size too big then the gradients will become less noisy but it will take longer to converge.
- vocab_threshold = 5. This is helpful to focus mostly on important words.
- vocab_from_file = True Great! this is time saving

- vocab_from_file = True Great! this is time saving
- embed_size = 512. Optimum number - The dimensionality of word embeddings represent how many features of the words we can extract. It does have some correlation with the size of the vocabulary. If the number is small we may not extract enough features, if the number is too large it may be unnecessary and even reduce performance. Check out this interesting visualization of projections in embedded dimensions https://projector.tensorflow.org/
- hidden_size = 512. Optimum number. If the number of hidden dim is too small it has less accuracy, if it is too large it affects the performance.
- num_epochs = 5. The number of epochs should be just enough to reach the expected loss values. It is recommended to stop the training early when the validation loss stops improving.

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

Answer:
For the encoder, it has a CNN structure, the architecture of which includes the ResNet-50 pretrained on the ImageNet dataset. From the pretrained network, the image features are extracted and passed through the last fully connected layer with its output having the equivalent size as the word embeddings.
As for the decoder, it has a RNN structure consisting of one embedding layer(transforming the input image features and captions into embeddings), one LSTM layer with one hiddenlayer, and one fully connected layer(mapping the hidden state's output into the vocab_size).
(Reference: Oriol Vinyals et al., Show and Tell: A Neural Image Caption Generator, CVPR 2015.)

It is wise to follow a research paper that has already done thorough work on the model and then refine it with finer details for the problem being solved. Well done 👍. I advise you to read and implement more papers for a comparative study.
You have a good idea about Resnet architecture used in Encoder.
The design of Decoder is simple and sufficient.

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.

Answer:
The provided values were used in transfrom_train since a enoughly large number of images were available online, images.cocodataset.org. Although data augmentations are used in many cases to artificially make more data available, to improve the model performances, and to reduce the overfitting, it is not particularly advised to utilize some aggresive augmentation strategies for the cases that a very large amount of data is available. It should also be noted that data augmentation can decrease the model performance in that distorted data by augmentations can greatly differ from real data that is what we really want to train. In this regard, more aggresive augmentations are not additionally implemented here, just nearly staying with the default transformations.

Great insights! 👏👏
Yes, aggresive augmentation strategies can decrease the model performance and even be counter-productive.
Hence the existing transforms used are good enough to use as it is. Well done.

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.

Using the encoder pretrained on the ImageNet dataset, the transfer learning was applied to train the encoder - only the last fully connected layer was directly trained. Since we deal with simliar kinds of images to the images in the ImageNet dataset, it is natural to go on the transfer learning approach(can expect a better convergence within much less time). For the decoder, it would also be possible to apply a similar approach to train the model; however, I did not especially apply the transfer learning for the decoder but train all the parameters of the decoder. Actually, the decoder has a RNN structure, not having a CNN structure as the encoder. Also, the number of the trainable parameters of the decoder is relatively much smaller than the trainable parameters of the encoder. For these reasons, it was concluded that there's no need to put much effort into transfer learning to train the decoder.

Good understanding of the concepts. Well done. 👍👍
Yes, we re-train only the classifier layer of Encoder since it uses pre-trained network plus we train all the layers of decoder from scratch.
You said *"the number of the trainable parameters of the decoder is relatively much smaller than the trainable parameters of the encoder."*
Not really CNN needs fewer trainable parameters that RNN. RNN implements fully connected layer within which uses many trainable parameters. The only reason we can speed up training here is by using pre-trained network. Just like CNN we can use pre-trained layers for embedding layers.
Please check this link for more information:
https://datascience.stackexchange.com/questions/10615/number-of-parameters-in-an-lstm-model

The submission describes how the optimizer was selected.

Answer:
The reason why Adam is simply chosen as the optimizer here is that Adam has performed reasonably well in almost all cases in my experiences(often considered as a standard choice and as one of the most efficient optimizers). Adam optimizer adaptively tweaks the learning rate with a less chance to be stuck in a local minima than would be the case if SGD were used as an optimiezer. Of course, it is still possible to try several other optimizers for training to improve the performance to some extent.

Adam optimizer is a good choice. You have a good understanding of the same. Check out the comparative study on optimizers. link.

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.

The model has trained correctly. 👍.
Perplexity helps the model learn and generate meaningful sentences. Read more about it here

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.

Transformations for Test dataset is chosen correctly and in sync with the training notebook.
EncoderCNN and DecoderRNN models are correctly used.

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

`sample` method is implemented correctly. LSTM() and embed() layers used correctly. The model is called sequentially to get the sequence of numbers representing words that are concatenated and returned as output caption. Well done! 👏 👏

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

Well done. The `clean_sentence` removes the start and the end token. Try to use python comprehensions for better performance. https://www.geeksforgeeks.org/comprehensions-in-python/.

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

The prediction is shown for both performing and not performing examples correctly. Well done. 👍 👍

DOWNLOAD PROJECT

5 CODE REVIEW COMMENTS ›

RETURN TO PATH

UDACITY

Return to Classroom

Image Captioning

REVIEW

CODE REVIEW 5

HISTORY

model.py 5

```
import torch
import torch.nn as nn
import torchvision.models as models


class EncoderCNN(nn.Module):
    def __init__(self, embed_size):
        super(EncoderCNN, self).__init__()
        resnet = models.resnet50(pretrained=True)
```

AWESOME

Resnet 50 is a good choice. Read about the power of residual learning here

```
        for param in resnet.parameters():
            param.requires_grad_(False)

        modules = list(resnet.children())[:-1]
        self.resnet = nn.Sequential(*modules)
        self.embed = nn.Linear(resnet.fc.in_features, embed_size)

    def forward(self, images):
        features = self.resnet(images)
        features = features.view(features.size(0), -1)
        features = self.embed(features)
        return features


class DecoderRNN(nn.Module):
```

```
    def __init__(self, embed_size, hidden_size, vocab_size, num_layers=1):
        super(DecoderRNN, self).__init__()
        self.embed_size = embed_size
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size
        self.num_layers = num_layers
        self.word_embeddings = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(input_size=embed_size, hidden_size=hidden_size, num_layers=num
```

AWESOME

The batch_first is set to True. well done!

```
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, features, captions):
        embeds = self.word_embeddings(captions[:, :-1])
```

AWESOME

✅ Good work with removing <end> tag and embedding of captions and concatenation with Encoder

```
        embeddings = torch.cat((features.unsqueeze(1), embeds), 1)
        lstm_out, self.hidden = self.lstm(embeddings)
        features = self.fc(lstm_out)
        return features

    def sample(self, inputs, states=None, max_len=20):
        " accepts pre-processed image tensor (inputs) and returns predicted sentence (list
        hidden = (torch.randn(self.num_layers, 1, self.hidden_size).to(inputs.device),
                  torch.randn(self.num_layers, 1, self.hidden_size).to(inputs.device))

        list_predictions = []
        for i in range(max_len):
            lstm_out, hidden = self.lstm(inputs, hidden)
```

AWESOME

LSTM() and embed() layers used correctly. Well done!👍

```
            out = self.fc(lstm_out).squeeze(1)
            out_argmax = out.argmax(dim=1)
            list_predictions.append(out_argmax.item())
```

AWESOME

The model is called sequentially to get the sequence of numbers representing words that are concate

```
            inputs = self.word_embeddings(out_argmax).unsqueeze(1)

        return list_predictions
```

RETURN TO PATH