Meets Specifications

Congratulations on finishing this project, you did great. The final captions are qualitatively very nice. In your answers, I see that you do not have any potential confusion and in fact, you seem to have a clear understanding of the project. It is also clear that you were in complete control of your experiments. Well done!

If you are interested in knowing more about this task like where it can be useful, you can go through these resources:

- Visual Question Answering
- Rich Image Captioning in the Wild
- Image Captioning and Visual Question Answering Based on Attributes and External Knowledge
- Intention Oriented Image Captions with Guiding Objects
- Object Counts! Bringing Explicit Detections Back into Image Captioning
- A Multi-task Learning Approach for Image Captioning
- Counter Factual Visual Explanations

Since you successfully solved this task, now is the perfect time to make an attempt at writing code for the natural extension of this problem where you not only make the model generate captions but also see what parts of the image its looking at to make predictions, a concept called "attention". If you have not already, please refer to the popular, one of the already suggested papers - Show, Attend and Tell to know more. You can seek inspiration from these works on GitHub - One and Two.

Good luck with the rest of the nanodegree.

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 the required files have been submitted, all visualization cells have been executed and all questions have been answered. Good job!

model.py

The chosen CNN architecture in the `CNNEncoder` class in model.py makes sense as an encoder for the image captioning task.
A wise move to go along with the `ResNet50` model as it is proven to have given very good results for this kind of problem.

If interested, you can try replacing it with Inception model as explained in Google's blog post on this task.
The chosen RNN architecture in the `RNNDecoder` class in model.py makes sense as a decoder for the image captioning task.
Your `RNNDecoder` looks great. Well done!

If interested, check out this nice paper to know more about learning CNN-LSTM architectures for image caption generating use cases.

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 arguments have been set reasonably well, good job on the reasoning as to how certain values have been chosen.

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

Your reasoning in the answers is on point. It is clear to me that you made well-informed decisions in coming up with the decoder architecture and also in selecting the hyperparameters, it is also clear that you are comfortable with the overall pipeline of the project. `512` is a great and mostly safe choice for both `embed_size` and `hidden_size`. Overall, great!

Additional comments for real-time usage of these trained models:

Most of the students often follow the suggested papers and it is probably the right thing to do when you are dealing with a new problem provided you are not sure where to begin. But what is also important is to understand the extreme cases like how small a network can we use to make the model predict decent quality captions. If you have the time and a good GPU to work with (I understand this is a big "if"), you should always be able to answer (to yourself at the very least) some questions like:

- Why `ResNet50`?
- Why not `ResNet18` or other smaller but powerful network architectures like `MobileNet`, `ShuffleNet` and `EfficientNet`?
- Can this problem be solved with smaller `embed_size` and `hidden_size`, maybe as small as `128` and even `64`? Where is the breaking point?
- What other data transforms can help with the task at hand?
- How important is it to tune the `vocab_threshold` value?
- How can I quantitatively evaluate my models? (read about BLEU score here)

Experimenting with different configurations such as the ones I mentioned can be extremely boring and would require a GPU of your own but the insights can really help us when we're dealing with real-time memory/computational constraints (often faced when deploying trained models on edge devices). Anyways, you did the right thing no doubt but please try and make sure you can answer these kinds of questions to yourself whenever you are on to new problems. Overall, a great job!

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 is in accordance with the choice of CNN architecture.

However, it would have been great if you had explained the chosen transforms by at least describing how they are helpful in this setup. You might have a decent understanding of what is going on but it is important for us to know why you think the chosen transform operations are relevant. Since you did great in all other rubrics, we can overlook this one but in your next project(s) please address the questions with relevant (sometimes even the obvious stuff) explanations.

For now, make sure you answer these questions to yourself:

- Why were the images resized to `224x224` particularly? Why not `256x256` or `100x100`?
- Why were these exact values used? Why are they special?

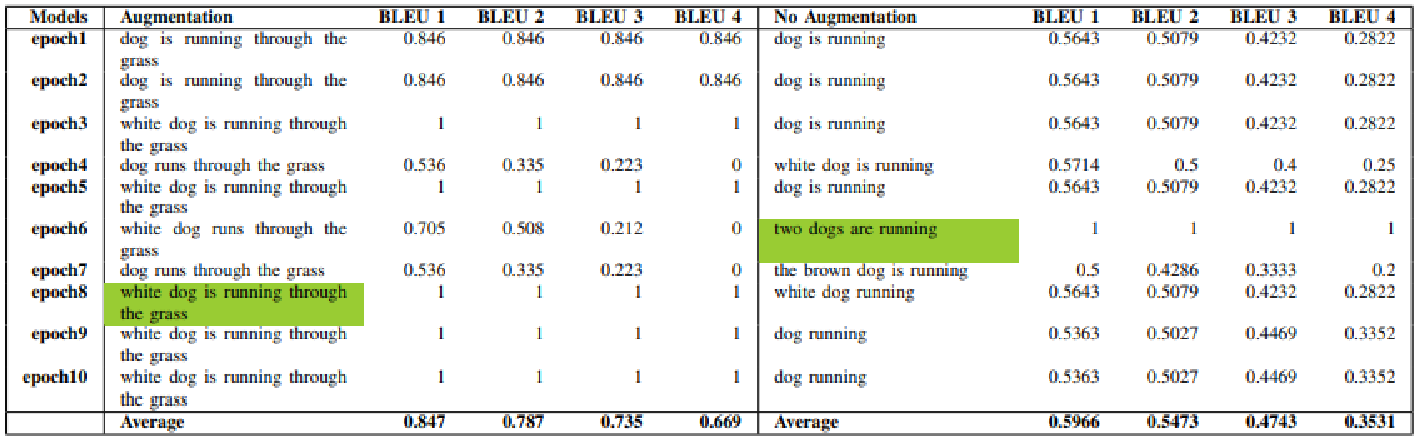
```
• transforms.Normalize((0.485, 0.456, 0.406),

•                      (0.229, 0.224, 0.225))]
```

- What is the significance of `RandomCrop` and `RandomHorizontalFlip` operations?
- Is the `RandomVerticalFlip` operation advisable for this task? Why/Why not?

Please understand that I am not feeding you the answers on purpose. A self-search would only help you remember for a longer time.

Also, I want to emphasize that data augmentation makes a lot of difference when it comes to the performance of the model for these tasks. This 2019 paper discusses this in detail. You can see the BLEU score comparison between two networks trained on Flickr8k dataset (not MS COCO). The paper also goes on to show how data augmentation stabilizes (i.e., consistent model performance) the training process overall across epochs.

TABLE I. BLEU SCORES FOR THE FIRST GROUP, WHERE THE IMAGE IN FIG. 5 IS A REFERENCE IMAGE (AUGMENTED VS. NON-AUGMENTED).



Models	Augmentation	BLEU 1	BLEU 2	BLEU 3	BLEU 4	No Augmentation	BLEU 1	BLEU 2	BLEU 3	BLEU 4
epoch1	dog is running through the grass	0.846	0.846	0.846	0.846	dog is running	0.5643	0.5079	0.4232	0.2822
epoch2	dog is running through the grass	0.846	0.846	0.846	0.846	dog is running	0.5643	0.5079	0.4232	0.2822
epoch3	white dog is running through the grass	1	1	1	1	dog is running	0.5643	0.5079	0.4232	0.2822
epoch4	dog runs through the grass	0.536	0.335	0.223	0	white dog is running	0.5714	0.5	0.4	0.25
epoch5	white dog is running through the grass	1	1	1	1	dog is running	0.5643	0.5079	0.4232	0.2822
epoch6	white dog runs through the grass	0.705	0.508	0.212	0	two dogs are running	1	1	1	1
epoch7	dog runs through the grass	0.536	0.335	0.223	0	the brown dog is running	0.5	0.4286	0.3333	0.2
epoch8	white dog is running through the grass	1	1	1	1	white dog running	0.5643	0.5079	0.4232	0.2822
epoch9	white dog is running through the grass	1	1	1	1	dog running	0.5363	0.5027	0.4469	0.3352
epoch10	white dog is running through the grass	1	1	1	1	dog running	0.5363	0.5027	0.4469	0.3352
	Average	**0.847**	**0.787**	**0.735**	**0.669**	**Average**	**0.5966**	**0.5473**	**0.4743**	**0.3531**

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.
Your understanding of what parameters have been updated during the training (and why) is absolutely correct.

Also, one very simple way of understanding model complexity is to see how many trainable parameters a network has. This would also help us compare two models. You could count the trainable (emphasis on trainable) parameters in many ways but this is perhaps a good way to do it:

```
def count_parameters(model):

    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

The `if p.requires_grad` check makes sure you are only counting the trainable ones. Take the condition off and you'd be counting all parameters of the network at hand.

The submission describes how the optimizer was selected.

Yes, `Adam` is often the safest (says the latest study, click on the link to see the paper) choice to go with while solving these kinds of problems. Here is another interesting paper that summarizes all the existing optimizers in one place.

Here is a quick snapshot of the findings of the paper:

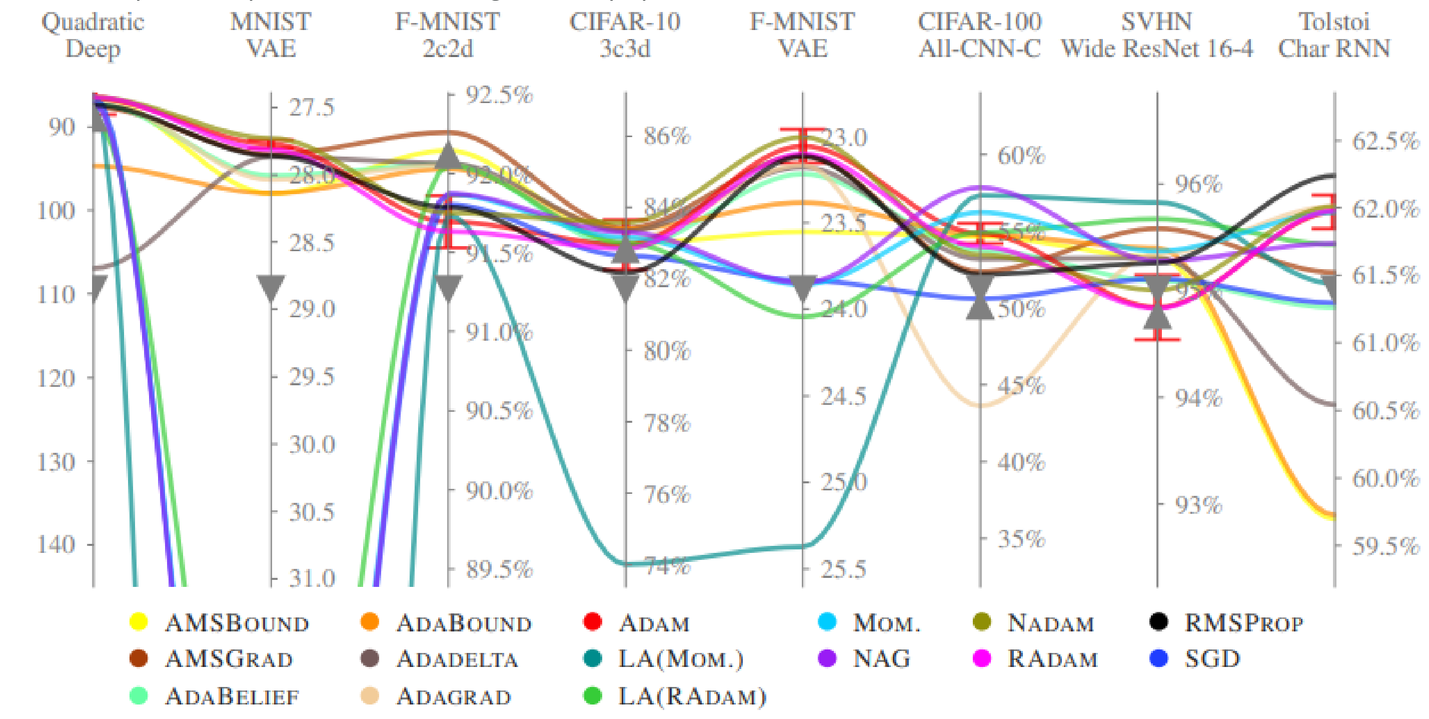


Figure 4: Mean test set performance over 10 random seeds of all tested optimizers on all eight optimization problems using the *large budget* for tuning and *no learning rate schedule*. One standard deviation for the *tuned* Adam optimizer is shown with a red error bar (I; error bars for other methods omitted for legibility). The performance of *untuned* Adam (▼) and AdaBound (▲) are marked for reference. The upper bound of each axis represents the best performance achieved in the benchmark, while the lower bound is chosen in relation to the performance of Adam with default parameters.

Also, `Adam` actually adjusts the learning rate by itself while training but that is not possible when we train the model for just 1-3 epochs, this means (from my experience with this project) any other learning rate but `0.001` would have yielded different (potentially bad) results.

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 code in Step 2 is well written with comments.

You could try validating your models which would help you a great deal in avoiding overfitting and also in tuning your hyperparameters appropriately. I understand it is not entirely clear how to validate your model in this setting. There are many ways to do this but you can follow my "quick" tutorial below to pull it off. You essentially need the following two things:

- A dataset and a dataloader to load COCO's validation set
- An evaluation code for comparing model output and actually results

Firstly, the dataloader code in `data_load.py` doesn't load the validation set. If you check the file, you will notice that the `get_loader` code has `if` conditions for `train` and `test` sets but not `val`. All you have to do is to add another `if` condition to load the validation files. Perhaps this way:

```
    if mode == 'val':

        if vocab_from_file==True:

            assert os.path.exists(vocab_file), "vocab_file does not exist.  Change vocab_from_file
to False"

        img_folder = os.path.join(cocoapi_loc, 'cocoapi/images/val2014/')

        annotations_file = os.path.join(cocoapi_loc, 'cocoapi/annotations/captions_val2014.json')
```

For clarity, you could simply clone the existing `CoCoDataset` and create a new dataset, say `CoCoValDataset` and repalce `"train"` with `"val"`. Then, in notebook 2, you can initialize your validataset set loader just the way you initialized trainset loader.

```
val_data_loader = get_loader(transform=transform_val, # You can remove augmentation operations fro
m the transform_train

                               mode='val',

                               batch_size=batch_size,

                               vocab_threshold=vocab_threshold,

                               vocab_from_file=True)
```

You can now use this dataloader and the trained model to get the model captions. Feel free to copy-paste the training loop (of course you remove the lines with the training parts) to get the final captions.

Finally, you now need code than can compare the model output and the actual caption results. Thankfully, `pycocoevalcap` is a brilliant tool that comes in handy for this task. Feel free to go through the source files of the tool but all you need now is to write a couple of lines as shown here in the example. This tool provides several metrics (BLEU, Metero, Rouge-L, CIDEr, SPICE) for you to keep track off. Metrics you can use to compare your models during the experimental phase. Check the repo's README for more details.

There you go. This is how you validate your model. If you have the GPU time left, I'd urge you to try it out. You could always seek for help on Knowledge if you run into any issues. Good luck!

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.

The chosen transform is efficient and it is nice to see that you did not include the data augmentation operation (`RandomHorizontalFlip`) in the test transform, something many students blindly include without putting much thought into it.

```
transform_test = transforms.Compose([

    transforms.Resize((224, 224)),                                 # smaller edge of image resized to 256

    transforms.ToTensor(),                                  # convert the PIL Image to a tensor

    transforms.Normalize((0.485, 0.456, 0.406),      # normalize image for pre-trained model

                        (0.229, 0.224, 0.225))])
```

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

Perfect, the entries from the output of the `sample` method do leverage the LSTM architecture to generate valid token indices. Each entry in the output corresponds to an integer that indicates a token in the vocabulary. RNN has worked correctly.

While this sampling method works well, there are actually slightly better ways to do the same. The current method you implemented is called "Greedy Search" method where you always consider the tokens with the highest probability only. The beam search algorithm, however, selects multiple alternatives for an input sequence at each timestep. The number of multiple alternatives depends on a parameter called beam width. At each time step, the beam search selects beam width number of best alternatives with the highest probability as the most likely possible choices for the time step. The algorithm (consider beam width 3) is as follows:

- Step 1: Find the top 3 words with the highest probability given the input sentence.
- Step 2: Find the three best pairs for the first and second words based on conditional probability
- Step 3: Find the three best pairs for the first, second and third word based on the input sentence and the chosen first and the second word

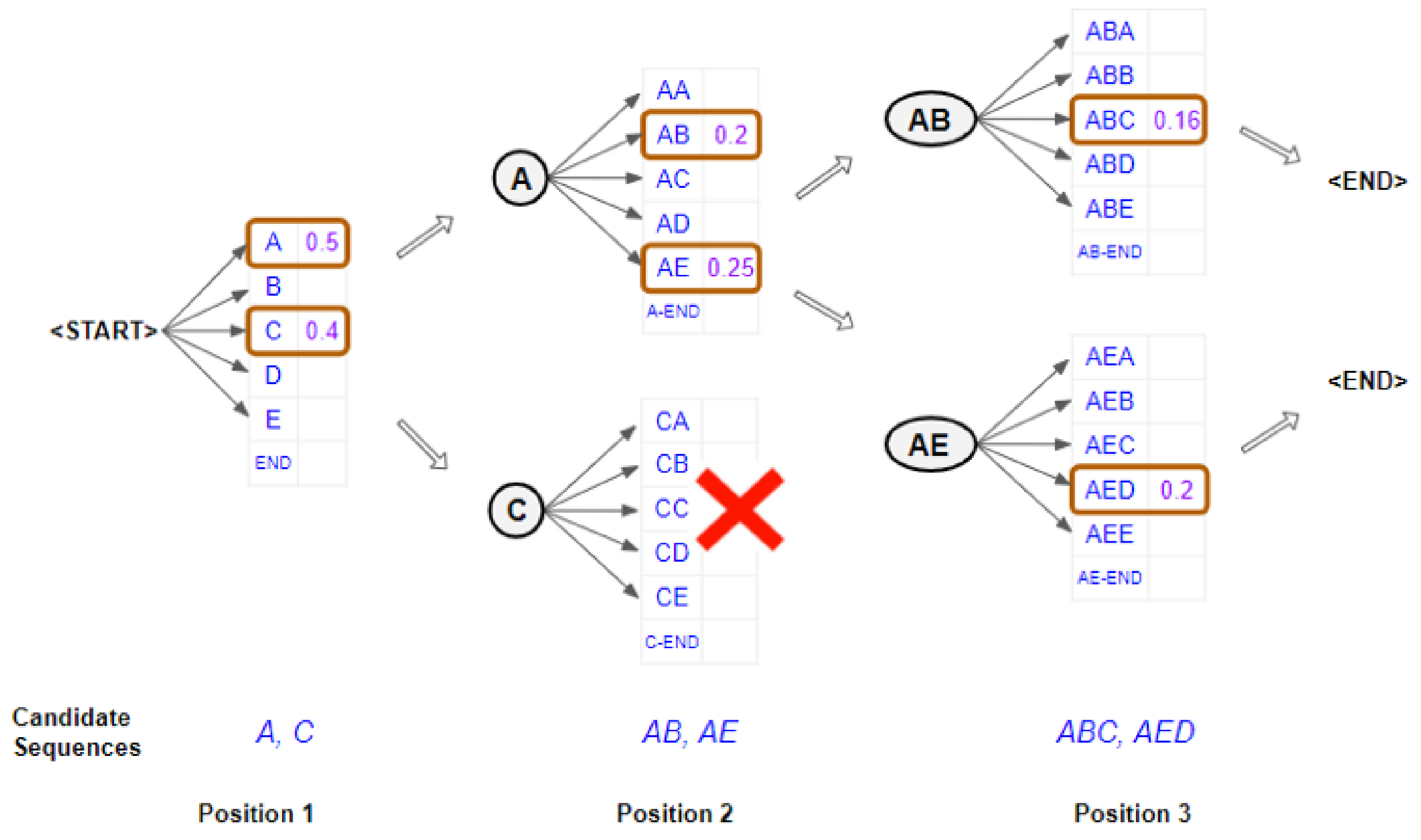


Figure: The process of beam search (beam width: 2, max length of an output sequence: 3)

You can check this file and the repository overall for the detailed implementation of beam search.

Note: A higher beam width will give a better translation but would use a lot of memory and computational power.

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 assertion holds and the sentences are reasonably clean, and any `<start>` and `<end>` tokens have been removed. Nice way of cleaning the sentences.

List comprehensions are always both computationally and aesthetically better choices over looping. Here's a more "pythonic" way of cleaning the sentences:

```
def clean_sentence(output):

    sentence = ''

    filtered_output = [index for index in output if (index != 0 and index != 1)]

    tokens = [data_loader.dataset.vocab.idx2word[index]

                            for index in filtered_output]

    ## Or you could do it all with one line of code
    tokens = [data_loader.dataset.vocab.idx2word[x] for x in output if x not in [0, 1]]



    sentence = ' '.join(tokens)

    return sentence
```

*Note: This works well only in conjunction with the useful practice I mentioned within the `sample` method in the **Code Review** section.*

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 predictions look very cool. The caption quality is great. You pass this rubric for correctly providing two pairs of examples, one where the model performed well and another where the model did not perform well. Good job!