Practical Exercises and Code for PGM Class Tutorial:

Combining Variational Inference and Sequential Monte Carlo

Exercises 1 & 2 were in the theory part of this tutorial, and now we move on to the practical part. In this part we will use the code from https://github.com/tensorflow/models/tree/master/research/fivo and will aim to reproduce a part of the results reported by the authors of "Filtering Variational Objectives" paper that we discussed in the theory part (the paper is available at https://arxiv.org/pdf/1705.09279.pdf ; we will refer to it here as [1]).

The goal of this part of the tutorial is to provide an introductory practical experience with state-of-the-art coding frameworks, datasets and infrastructure. This part of the tutorial will not involve coding from scratch, but will require ability to look into and comprehend existing code and training scripts.

Exercise 3: FIVO Python Code and Data Setup

Install tensorflow & related python dependencies for FIVO.

Unzip `fivo.zip` with the FIVO-related code, install Tensorflow and a few python dependencies:

```
pip install --upgrade tensorflow
pip install scipy dm-sonnet
```

More information for Tennsorflow on various platforms: https://www.tensorflow.org/install/

Download and pre-process pianoroll dataset

[1] demonstrated FIVO benefits on polyphonic music datasets:

- Piano-midi.de is a classical piano MIDI archive
- Nottingham is a collection of 1200 folk tunes
- MuseData is an electronic library of orchestral and piano classical music
- JSB chorales refers to the corpus of 382 four-part harmonized chorales by J. S. Bach

Extra note, in case you are interested: each dataset has at a least 7 hours of polyphonic music, the total duration is ~67 hours; the number of simultaneous notes (polyphony) varies from 0 to 15, with average of ~3.9; piano keys from A0 to C8 are used (so input is represented by 88 binary values, one for each piano key: https://en.wikipedia.org/wiki/Piano_key_frequencies).

To download and preprocess JSB dataset, assuming your FIVO code is in directory `fivo`, do:

```
cd fivo
export PIANOROLL_DIR=~/pianorolls
mkdir $PIANOROLL_DIR
./bin/download_pianorolls.sh $PIANOROLL_DIR
python data/calculate_pianoroll_mean.py --in_file=$PIANOROLL_DIR/jsb.pkl
```

Optional Extra Setup

If you would like to isolate this project from other python setups you have on your machine, you could install everything in virtualenv, for example run:

```
pip install virtualenv
virtualenv vismc_env
source vismc_env/bin/activate
```

Then install all the packages in this isolated environment.
Note: to create virtualenv with a specific python version use:

```
virtualenv --python=/usr/bin/python2.7 vismc_env
```

Exercise 4: Reproducing FIVO Benefits

We will first try to reproduce results reported in [1] on the JSB dataset using the code provided by the authors (in the `fivo`directory). The authors report training VRNN cells with latent size of 32, with N=4 particles. They report doing grid search for the best learning rate and early stopping using validation set to determine the overall number of gradient updates. We will try learning rate of 3×10^{-4}, and will first run for 1K gradient updates.

Run the training:

```
python fivo.py \
  --mode=train \
  --logdir=/tmp/fivo-jsb \
  --model=vrnn \
  --latent_size=32 \
  --bound=fivo \
  --summarize_every=50 \
  --batch_size=4 \
  --num_samples=4 \
  --learning_rate=0.0003 \
  --max_steps=1000 \
  --dataset_path="$PIANOROLL_DIR/jsb.pkl" \
  --dataset_type="pianoroll"
```

Then get evaluation results:

```
python fivo.py \
  --mode=eval \
  --split=test \
  --bound=fivo \
  --alsologtostderr \
  --logdir=/tmp/fivo-jsb \
  --model=vrnn \
  --latent_size=32 \
  --batch_size=4 \
  --num_samples=4 \
  --dataset_path="$PIANOROLL_DIR/jsb.pkl" \
  --dataset_type="pianoroll"
```

Evaluation results should look similar to:

```
INFO:tensorflow:Restoring parameters from /tmp/fivo/model.ckpt-2557
INFO:tensorflow:Model restored from step 2557, evaluating.
INFO:tensorflow:test elbo ll/t: -12.725472, iwae ll/t: -12.675230 fivo ll/t: -12.232440
INFO:tensorflow:test elbo ll/seq: -9689.683047, iwae ll/seq: -9651.427070 fivo ll/seq: -931
4.269062
```

From Figure 3 of [1] we see that the ultimate benefit of using FIVO bound could manifest after a very large number of gradient updates. Since you are probably running on only one or a few CPUs, we will just try to reproduce the initial stages of the training and confirm that we can see the benefit of using FIVO.

Read through `fivo/fivo.py` to see how you can specify arguments for the training and evaluation process. Launch training for 2K steps using ELBO, IWAE and FIVO bounds. Plot the per-step train log likelihood (`ll/t`) vs the number of training steps. Run evaluation and report test log likelihood after 2K steps. (Hint: if you have used tensorflow/tensorboard before, you can get the plots from tensorboard; otherwise you can just use any other way to plot a few results that are printed on the command line when you launch test and training scripts).

After you get your results

Note that your results after 2K gradient updates would not yet come close to numbers reported in [1], where training log likelihood is reported to be -4.59 for FIVO bound. Getting such results would require a bit more gradient updates (perhaps ~200K), grid search over learning rate and early stopping using validation set. Given how long 2K steps took on your machine, you can now appreciate the amount of computation required to demonstrate the benefits. It is common for algorithms using deep / recurrent neural networks to require a significant number of compute time to demonstrate interesting results that you might find in the literature.

Exercise 5:

Take a look at `fivo/bounds.py`. This file contains code for computing IWAE and FIVO bounds. Read through the comments for functions `iwae()` and `fivo()`. These should give you the overall idea of how the bound-related computations are implemented.

It would be a bit challenging to uderstand all the code in-depth if you have not worked with Tensorflow before (or other graph computation frameworks, e.g. Theano). So for this exercise we will just focus on modifying one aspect of the code: the resampling criterion.

"A Tutorial on Particle Filtering and Smoothing" suggests that alternative criteria can be used, such as the entropy of the weights. (https://www.stats.ox.ac.uk/~doucet/doucet_johansen_tutorialPF2011.pdf)

In practice, most computations are done in log space, so when you read through the code you will notice that logs of unnormalized weights are computed. Computing in log space is a common practice that prevents underflow (probabilities or weights being too small to represent accurately for computations). Luckily, Section 2.3 of "On adaptive resampling strategies for sequential Monte Carlo methods" suggests a more convenient formulation when working in log space for resampling criterion: the relative entropy of the empirical particle measure w.r.t. its weighted version: $RE = -\frac{1}{N}\sum_{i=1}^{N}\log w_i > a_n$, where w_i are unnormalized weights and $a_n$is some threshold value. We resample if RE is above a_n. (https://arxiv.org/pdf/1203.0464.pdf)

You will need to look through the code in `bounds.py` and locate the place where you can insert the code to compute and use this new resampling criterion.

Hint1: A useful note on Effective Sample Size computations: http://www.nowozin.net/sebastian/blog/effective-sample-size-in-importance-sampling.html

Hint2: Some relevant Tensorflow functions: https://www.tensorflow.org/api_docs/python/tf/reduce_logsumexphttps://www.tensorflow.org/api_docs/python/tf/reduce_sum

Hint3: In the code, `log_weights_acc` is a Tensorflow tensor, not a regular python object. But you can think of as a $B \times N$ array, where B is the batch size and N is the number of samples. `tf.reduce_sum(x, axis=0)` means summing over axis 0 of tensor/array x, like in numpy. If this seems obscure - don't worry about it, just use axis 0 for this exercise and it should work out.

Hint4: Printing the values of Tensorflow tensors is a bit tricky, because the usual python print statements in the code would be executed during graph construction (when the values have not been populated yet). To print the values at the time when the graph computations are done, you can use:

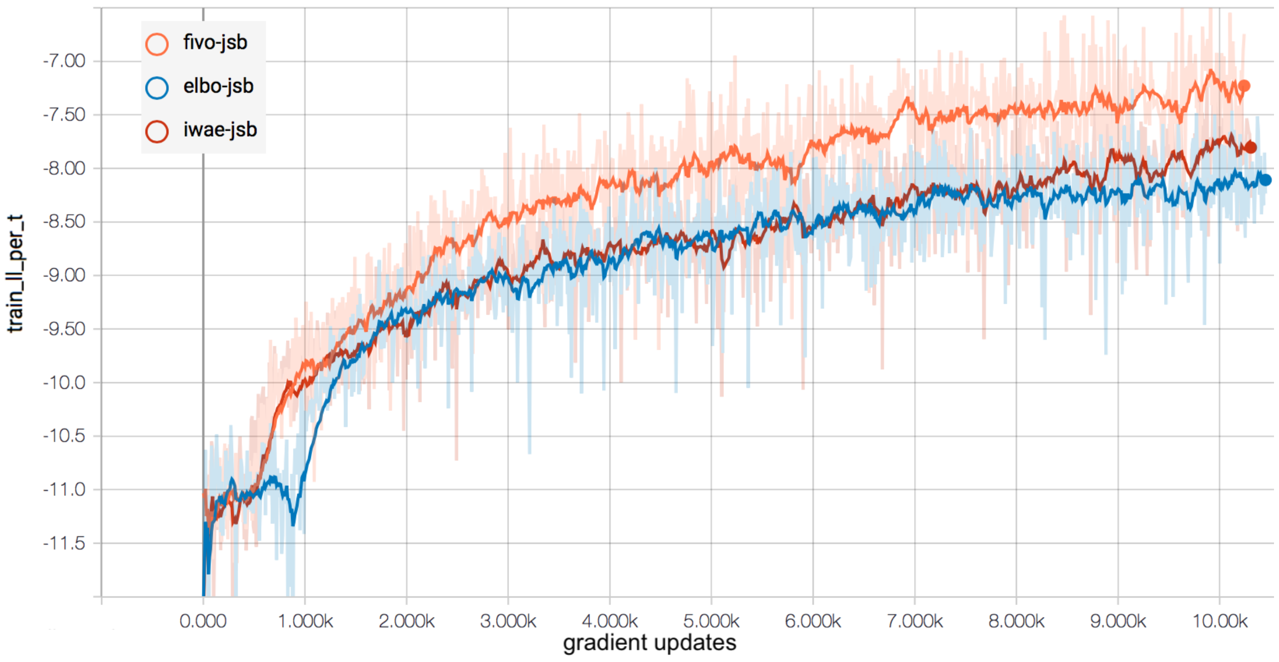
```
log_weights_acc = tf.Print(log_weights_acc,[log_weights_acc],
                           message="log_weights_acc", summarize=100)
```

Documentation for tf.Print(): https://www.tensorflow.org/api_docs/python/tf/Print

Answers to Exercises

Answer to Exercise 4

After looking at `fivo.py` it should become clear that the type of the bound can be selected using `--bound` argument, to train for 1K steps we can set `--max_steps=2000`. The plot results can be for any number of gradient update steps, depending on students' compute power access. Below is a plot where we trained with ELBO, IWAE, FIVO bounds on JSB pianorolls dataset and plotted train log likelihood per timestep for up to 10K gradient updates. Note that on each individual run the performance might vary, so this is just one example.


fivo-jsb
elbo-jsb
iwae-jsb
train_ll_per_t
1.000k
2.000k
3.000k
4.000k
5.000k
6.000k
7.000k
8.000k
9.000k
10.00k
gradient updates