How to Use This Library

Patrick Sullivan
psullivan@wustl.edu

June 25, 2017

1 Introduction

- This source code is intended to be used to build, train, and analyze neural network models using Tensorflow. The focus is primarily on autoencoders and generative models using convolutional nets, though the library has been designed to be as flexible as possible provided you follow a few guidelines.
- This library requires a very recent version of Tensorflow to work out of the box, specifically as of commit `g2336cdf7f`; this is a few commits ahead of the main 1.2 release. The easiest way to do this is simply build TF from source from the `master` branch (instructions are on the TensorFlow website). This sounds daunting but it is actually fairly straightforward (at least on Linux) and I would recommend giving it a try. Just follow the instructions on the TF website.
- Alternatively, you can use the main 1.2 release if you remove the caching suppport from `data.py`, just comment out line 47 or any references to `cache`. You may also be able to get away with any main release from 1.0 onwards, but no guarantees.

2 Running the Code

- All code is executed through calls to `train.py` by passing in command line arguments. There are numerous arguments available for different scenarios depending on what you are trying to accomplish; one of these is an option to pass in a file that contains arguments itself (for more complex cases).
- In general the bare minimum of arguments is `--model` and `--dir`, which will train the specific model you pass in with all of the default options, and store the summaries, checkpoints, and other results in the given directory. You may also use just `--config` to specify a file containing all of your options; any parameters specified on both the command line and

in the config file will give precedence to the command line (thus you can use a common config file and tailor it per run, if desired, without creating a new config file for each run).

- Run `python train.py --help` for a list of available command line arguments along with some (hopefully) descriptive text for each.
- Some examples:

```
python train.py --model iwgan
                --dir workspace/tests/new-iwgan
                --epochs 100
                --n_gpus 2
                --batch_size 256
                --optimizer adam
                --lr 1e-4
                --beta1 0.5
                --beta2 0.9
```

Figure 1: Trains the Improved Wasserstein GAN model over 100 epochs using 2 GPUs. Batch size is 256 and the optimizer has been set to ADAM using $\alpha =$ 1e-4, $\beta_1 = 0.5$ and $\beta_2 = 0.9$.

```
python train.py --config examples/vae.config
                --dir workspace/vae
```

Figure 2: Trains a Variational Autoencoder using the options specified in the config file and store the results in the given directory.

```
python train.py --config examples/vae.config
                --dir workspace/vae
                --epochs 5
```

Figure 3: Same as above, but overrides the `epochs` setting in the config file.

3 Datasets

- As mentioned above, this library is primarily focused on autoencoders and generative (unsupervised) convolutional nets. This means the datasets are assumed to be compsed solely of 2D images. It's easy

```
> python train.py --config examples/cnn.config --dir workspace/baseline/cnn
  Parsing options...
        config = None
        seed = b's\xb8\xb0\xa2'
        n_gpus = 2
        profile = False
        epochs = 100
        batch_size = 512
        epoch_size = -1
        examples = 64
        dir = workspace/baseline/cnn
        n_disc_train = 5
        optimizer = rmsprop
        lr = 0.0001
        loss = l1
        momentum = 0.01
        decay = 0.9
        centered = False
        beta1 = 0.9
        beta2 = 0.999
        model = cnn
        latent_size = 200
        dataset = floorplans
        resize = None
        shuffle = True
        buffer_size = 10000
        grayscale = False
        cache_dir = None
  Initializing input pipeline...
  Initializing model...
  Initializing supervisor...
  Generating baseline summaries and checkpoint...
  Starting training...
  Epoch   1: 100%|##############| 157/157 [09:28<00:00,  3.49batch/s, loss=0.948669(-)]
  Epoch   2: 100%|##############| 157/157 [00:49<00:00,  3.69batch/s, loss=0.295768(-)]
  Epoch   3: 100%|##############| 157/157 [00:49<00:00,  3.69batch/s, loss=0.300179(-)]
```

Figure 4: Example output of a training session through 3 epochs. Note that the first epoch took nearly 10 minutes but subsequent ones only 49 seconds due to caching effects.

enough to add support for different types of datasets, but this is not supported directly.

- Datasets should be stored in a TFRecord format. Documentation is available online about the details of this, and examples of generating TFRecord datasets from the filesystem and from NumPy data are available in the `data` directory. In short, each image needs to include its data in the `image` key, along with its width, height, and depth (number of channels) in the `width`, `height`, and `channels` keys, respectively.

- In general, effort has been made to allow you to use a dataset without only minor and unimpactful changes from their origional form:

  - Images do not have to be stored in a dense array format. This code leverages `tf.images.decode_image` to read data in `jpg`, `gif`, and `png` formats (and any more that are added as TF grows). Thus you can be sure that you are using the actual dataset "directly" without any artifacts due to conversion issues.

  - In addition, images can be of any size, and need not all be of uniform dimensions; there is a resize argument you can pass in if you need your dataset to be of a uniform dimension (as you likely will).

  - Before you freak out, don't worry, this not actually that inefficient. During training, the dataset will be resized, preprocessed, etc., fully during the first pass over the dataset, but all results will be cached so that this only happens once. Caches can exist in memory (the default) or on disk (using the `--cache_dir` option).[1] If you can fit the entire dataset in RAM, do it, otherwise cache it to disk; the input pipeline will automatically take advantage of multiple threads and other techniques to try to keep the GPUs as busy as possible at all times.

4 Models

- Each model is responsible for building itself and returning a function that describes one training step (i.e., training for one batch). Otherwise, the convention for accessing any information you need from a model (loss nodes, sampling nodes, etc.) is to include them in some sort of collection which is then accessed later. In other words, instead of maintaining state in variables or classes or convoluted functional params, *mark up your*

[1] You can always resize your dataset in advance if you really want to, but trust me when I say that after many tests with my own datasets, this is really not necessary, and it hinders your flexibility significantly. For example, the `floorplans` dataset only adds around 10 minutes to resize and preprocess during the first epoch, which is only 0.1% of the time taken during an 18-hour GAN training run.

graph to include what you need, then reference those features in the graph by traversing it in some way.

- Models are not inherently multi-GPU, and are responsible for handling this themselves, however, a few utility functions are provided to make this process relatively painless. Each of the provided models `gan.py`, `vae.py`, and `cnn.py` are multi-GPU and show the basic outline of how to structure your code into towers across GPUs, how to ensure variables are pinned to the right device, how to average the gradients during training, etc. While there are many things to consider here, it is not quite as complicated as it sounds; look at the provided models' source.

5 Training

- During training, the library will execute the training function returned by the model. This function is responsible for executing any graph nodes needed for a single training step. For example, the VAE model simply runs the optimizer's minimizer over the single loss function, while the IWGAN model runs the discriminator node n times before running the generator.
- No input (placehodler) data is passed into the training function. Rather, the input nodes are built into the graph by the input pipeline and will provide batches automatically. This means each call to `sess.run` consumes one batch!
- Summaries are generated at the middle and the end of each epoch. In addition, during the first 3 epochs, the summaries are run 10 times (spaced evenly) during each epoch. The purpose of this is to allow more insight into the early training steps. Note that *all* summary nodes are run each time; I would recommend not worrying about efficiency here as your training time is still going to massively dominate all other considerations, even if you are generating a ridiculous number of statistics/summaries.
- Some useful summary operations are provided in `ops summaries.py` for generating image montages of activations, weights, samples, etc., along with other useful statistics.
- Checkpoints are saved at the beginning (after initialization but before any training) and at the end of each checkpoint.
- If you wish to resume training, simply use the same directory as a previous one; the `tf.Supervisor` will automatically resume from the most recent checkpoint in that directory, and TensorBoard will in turn automatically combine events files from multiple runs.

- This that if you run into crashing issues (usually due to `NaN` or `Inf` being present in the gradients, i.e., vanishing and exploding gradients), you can simply run the same command again and it will resume and attempt to complete the remaining epochs. If you do not set the seed during this, it will generate a new one randomly, and you may avoid running into the same problem.[2]
- The `--epochs` argument sets the *maximum* number of epochs to train for if you pass it a number, but if you pass it `+`n (e.g., `+10`), it will train for n additional epochs from the last saved checkpoint.

6 Adding Your Own Models

- Each model should return a training function that accepts a session and the arguments and runs what it considers to be "one batch."
- It is strongly recommended to read through one of the provided models to get an idea of how to structure your code, as many models can be structured the same way as the provided examples. In fact, you will likely want to use a copy of one as your starting point.
- The example models will also give you a good idea of the utility functions provided, how to handle scoping, how to add in summaries, how to use multiple GPUs, etc.
- You will need to adjust `train.py` to allow use of your model.[3]

7 Visualization

- There is some additional visualization code provided in `visualization.py` that can work with saved checkpoints. The idea here is that, after training, you can point this code to the training directory (`python visualization.py --dir some_dir`) along with some options and it will generate all kinds of graphs and images of your model's results over time.
- *However* as of today this code hasn't been updated in a couple months and it may no longer work out of the box. It will be cleaned up in a future release, and using Tensorboard summaries is still your best bet for visualizing and analyzing your models for the time being.

[2] Of course, if you consistently run into issues with exploding or vanishing gradients, you should make sure you model does not have any bugs in it. If it appears bug-free, then you should adjust your optimizer's hyperparameters to encourage convergence.

[3] In the future this will automatically be done dynamically for any models added to the `models` directory.