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 code has been designed to be as flexible as possible provided you follow a few guidelines.

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:
  - –

3 Datasets

- As mentioned above, this code 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 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 dpeth (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 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.

[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.

5 Training

-