karpathy / char-rnn

Multi-layer Recurrent Neural Networks (LSTM, GRU, RNN) for character-level language models in Torch

88 commits 1 branch 0 releases 18 contributors

karpathy Merge pull request #164 from gdb/master ... Latest commit 6f9487a on Apr 30 2016

data/tinyshakespeare	first commit	2 years ago
model	changing default LSTM initialization to use biases of 1.0 for the for...	2 years ago
util	Fix unclear errors	2 years ago
.gitignore	Add t7 files to .gitignore	2 years ago
Readme.md	Update Readme.md	a year ago
convert_gpu_cpu_checkpoint.lua	fixing a bug introduced in previous commit. We have to use doubles no...	2 years ago
inspect_checkpoint.lua	add opencl to sample.lua and inspect_checkpoint.lua, add link to clto...	2 years ago
sample.lua	fixing a bug introduced in previous commit. We have to use doubles no...	2 years ago
train.lua	fix message	2 years ago

Readme.md

char-rnn

This code implements **multi-layer Recurrent Neural Network** (RNN, LSTM, and GRU) for training/sampling from character-level language models. In other words the model takes one text file as input and trains a Recurrent Neural Network that learns to predict the next character in a sequence. The RNN can then be used to generate text character by character that will look like the original training data. The context of this code base is described in detail in my blog post.

If you are new to Torch/Lua/Neural Nets, it might be helpful to know that this code is really just a slightly more fancy version of this 100-line gist that I wrote in Python/numpy. The code in this repo additionally: allows for multiple layers, uses an LSTM instead of a vanilla RNN, has more supporting code for model checkpointing, and is of course much more efficient since it uses mini-batches and can run on a GPU.

Update: torch-rnn

Justin Johnson (@jcjohnson) recently re-implemented char-rnn from scratch with a much nicer/smaller/cleaner/faster Torch code base. It's under the name torch-rnn. It uses Adam for optimization and hard-codes the RNN/LSTM forward/backward passes for space/time efficiency. This also avoids headaches with cloning models in this repo. In other words, torch-rnn should be the default char-rnn implemention to use now instead of the one in this code base.

Requirements

This code is written in Lua and requires Torch. If you're on Ubuntu, installing Torch in your home directory may look something like:

```
$ curl -s https://raw.githubusercontent.com/torch/ezinstall/master/install-deps | bash
$ git clone https://github.com/torch/distro.git ~/torch --recursive
$ cd ~/torch;
$ ./install.sh      # and enter "yes" at the end to modify your bashrc
$ source ~/.bashrc
```

See the Torch installation documentation for more details. After Torch is installed we need to get a few more packages using LuaRocks (which already came with the Torch install). In particular:

```
$ luarocks install nngraph
$ luarocks install optim
$ luarocks install nn
```

If you'd like to train on an NVIDIA GPU using CUDA (this can be to about 15x faster), you'll of course need the GPU, and you will have to install the CUDA Toolkit. Then get the `cutorch` and `cunn` packages:

```
$ luarocks install cutorch
$ luarocks install cunn
```

If you'd like to use OpenCL GPU instead (e.g. ATI cards), you will instead need to install the `cltorch` and `clnn` packages, and then use the option `-opencl 1` during training (cltorch issues):

```
$ luarocks install cltorch
$ luarocks install clnn
```

Usage

Data

All input data is stored inside the `data/` directory. You'll notice that there is an example dataset included in the repo (in folder `data/tinyshakespeare`) which consists of a subset of works of Shakespeare. I'm providing a few more datasets on this page.

Your own data: If you'd like to use your own data then create a single file `input.txt` and place it into a folder in the `data/` directory. For example, `data/some_folder/input.txt`. The first time you run the training script it will do some preprocessing and write two more convenience cache files into `data/some_folder`.

Dataset sizes: Note that if your data is too small (1MB is already considered very small) the RNN won't learn very effectively. Remember that it has to learn everything completely from scratch. Conversely if your data is large (more than about 2MB), feel confident to increase `rnn_size` and train a bigger model (see details of training below). It will work *significantly better*. For example with 6MB you can easily go up to `rnn_size` 300 or even more. The biggest that fits on my GPU and that I've trained with this code is `rnn_size` 700 with `num_layers` 3 (2 is default).

Training

Start training the model using `train.lua`. As a sanity check, to run on the included example dataset simply try:

```
$ th train.lua -gpuid -1
```

Notice that here we are setting the flag `gpuid` to -1, which tells the code to train using CPU, otherwise it defaults to GPU 0. There are many other flags for various options. Consult `$ th train.lua -help` for comprehensive settings. Here's another example that trains a bigger network and also shows how you can run on your own custom dataset (this already assumes that `data/some_folder/input.txt` exists):

```
$ th train.lua -data_dir data/some_folder -rnn_size 512 -num_layers 2 -dropout 0.5
```

Checkpoints. While the model is training it will periodically write checkpoint files to the `cv` folder. The frequency with which these checkpoints are written is controlled with number of iterations, as specified with the `eval_val_every` option (e.g. if this is 1 then a checkpoint is written every iteration). The filename of these checkpoints contains a very important number: the **loss**. For example, a checkpoint with filename `lm_lstm_epoch0.95_2.0681.t7` indicates that at this point the model was on epoch 0.95 (i.e. it has almost done one full pass over the training data), and the loss on validation data was 2.0681. This number is very important because the lower it is, the better the checkpoint works. Once you start to generate data (discussed below), you will want to use the model checkpoint that reports the lowest validation loss. Notice that this might not necessarily be the last checkpoint at the end of training (due to possible overfitting).

Another important quantities to be aware of are `batch_size` (call it B), `seq_length` (call it S), and the `train_frac` and `val_frac` settings. The batch size specifies how many streams of data are processed in parallel at one time. The sequence length specifies the length of each stream, which is also the limit at which the gradients can propagate backwards in time. For example, if `seq_length` is 20, then the gradient signal will never backpropagate more than 20 time steps, and the model might not *find* dependencies longer than this length in number of characters. Thus, if you have a very difficult dataset where there are a lot of long-term dependencies you will want to increase this setting. Now, if at runtime your input text file has N characters, these first all get split into chunks of size `BxS` . These chunks then get allocated across three splits: train/val/test according to the `frac` settings. By default `train_frac` is 0.95 and `val_frac` is 0.05, which means that 95% of our data chunks will be trained on and 5% of the chunks will be used to estimate the validation loss (and hence the generalization). If your data is small, it's possible that with the default settings you'll only have very few chunks in total (for example 100). This is bad: In these cases you may want to decrease batch size or sequence length.

Note that you can also initialize parameters from a previously saved checkpoint using `init_from` .

Sampling

Given a checkpoint file (such as those written to `cv`) we can generate new text. For example:

```
$ th sample.lua cv/some_checkpoint.t7 -gpuid -1
```

Make sure that if your checkpoint was trained with GPU it is also sampled from with GPU, or vice versa. Otherwise the code will (currently) complain. As with the train script, see `$ th sample.lua -help` for full options. One important one is (for example) `-length 10000` which would generate 10,000 characters (default = 2000).

Temperature. An important parameter you may want to play with is `-temperature` , which takes a number in range (0, 1] (0 not included), default = 1. The temperature is dividing the predicted log probabilities before the Softmax, so lower temperature will cause the model to make more likely, but also more boring and conservative predictions. Higher temperatures cause the model to take more chances and increase diversity of results, but at a cost of more mistakes.

Priming. It's also possible to prime the model with some starting text using `-primetext` . This starts out the RNN with some hardcoded characters to *warm* it up with some context before it starts generating text. E.g. a fun primetext might be `-primetext "the meaning of life is "` .

Training with GPU but sampling on CPU. Right now the solution is to use the `convert_gpu_cpu_checkpoint.lua` script to convert your GPU checkpoint to a CPU checkpoint. In near future you will not have to do this explicitly. E.g.:

```
$ th convert_gpu_cpu_checkpoint.lua cv/lm_lstm_epoch30.00_1.3950.t7
```

will create a new file `cv/lm_lstm_epoch30.00_1.3950.t7_cpu.t7` that you can use with the sample script and with `-gpuid -1` for CPU mode.

Happy sampling!

Tips and Tricks

Monitoring Validation Loss vs. Training Loss

If you're somewhat new to Machine Learning or Neural Networks it can take a bit of expertise to get good models. The most important quantity to keep track of is the difference between your training loss (printed during training) and the validation loss (printed once in a while when the RNN is run on the validation data (by default every 1000 iterations)). In particular:

- If your training loss is much lower than validation loss then this means the network might be **overfitting**. Solutions to this are to decrease your network size, or to increase dropout. For example you could try dropout of 0.5 and so on.
- If your training/validation loss are about equal then your model is **underfitting**. Increase the size of your model (either number of layers or the raw number of neurons per layer)

Approximate number of parameters

The two most important parameters that control the model are `rnn_size` and `num_layers`. I would advise that you always use `num_layers` of either 2/3. The `rnn_size` can be adjusted based on how much data you have. The two important quantities to keep track of here are:

- The number of parameters in your model. This is printed when you start training.
- The size of your dataset. 1MB file is approximately 1 million characters.

These two should be about the same order of magnitude. It's a little tricky to tell. Here are some examples:

- I have a 100MB dataset and I'm using the default parameter settings (which currently print 150K parameters). My data size is significantly larger (100 mil >> 0.15 mil), so I expect to heavily underfit. I am thinking I can comfortably afford to make `rnn_size` larger.
- I have a 10MB dataset and running a 10 million parameter model. I'm slightly nervous and I'm carefully monitoring my validation loss. If it's larger than my training loss then I may want to try to increase dropout a bit and see if that heps the validation loss.

Best models strategy

The winning strategy to obtaining very good models (if you have the compute time) is to always err on making the network larger (as large as you're willing to wait for it to compute) and then try different dropout values (between 0,1). Whatever model has the best validation performance (the loss, written in the checkpoint filename, low is good) is the one you should use in the end.

It is very common in deep learning to run many different models with many different hyperparameter settings, and in the end take whatever checkpoint gave the best validation performance.

By the way, the size of your training and validation splits are also parameters. Make sure you have a decent amount of data in your validation set or otherwise the validation performance will be noisy and not very informative.

Additional Pointers and Acknowledgements

This code was originally based on Oxford University Machine Learning class practical 6, which is in turn based on learning to execute code from Wojciech Zaremba. Chunks of it were also developed in collaboration with my labmate Justin Johnson.

To learn more about RNN language models I recommend looking at:

- My recent talk on char-rnn
- Generating Sequences With Recurrent Neural Networks by Alex Graves
- Generating Text with Recurrent Neural Networks by Ilya Sutskever
- Tomas Mikolov's Thesis

License

MIT