karpathy / **neuraltalk2**

Watch 229 Star 3,340 Fork 700

Code Issues 104 Pull requests 9 Projects 0 Pulse Graphs

Efficient Image Captioning code in Torch, runs on GPU

53 commits **1** branch **0** releases **9** contributors

Branch: master New pull request Find file Clone or download

karpathy update Latest commit `bd8c9d8` on Sep 23 2016

coco-caption	first commit	2 years ago
coco	first commit	2 years ago
cv	adding my crossvalidation scripts to give people idea about how i use…	2 years ago
misc	Add support for .jpeg files	a year ago
vis	Tweak colors a bit	2 years ago
.gitignore	first commit	2 years ago
README.md	update	8 months ago
convert_checkpoint_gpu_to_cpu.lua	adding beam search, helps quite a bit	2 years ago
eval.lua	added dump_path option	2 years ago
prepro.py	Add helpful error if image is corrupted	2 years ago
test_language_model.lua	adding beam search, helps quite a bit	2 years ago
train.lua	resolving merge conflicts	2 years ago
videocaptioning.lua	videocaptioning	a year ago

README.md

NeuralTalk2

Update (September 22, 2016): The Google Brain team has released the image captioning model of Vinyals et al. (2015). The core model is very similar to NeuralTalk2 (a CNN followed by RNN), but the Google release should work significantly better as a result of better CNN, some tricks, and more careful engineering. Find it under im2txt repo in tensorflow. I'll leave this code base up for educational purposes and as a Torch implementation.

Recurrent Neural Network captions your images. Now much faster and better than the original NeuralTalk. Compared to the original NeuralTalk this implementation is **batched, uses Torch, runs on a GPU, and supports CNN finetuning**. All of these together result in quite a large increase in training speed for the Language Model (~100x), but overall not as much because we also have to forward a VGGNet. However, overall very good models can be trained in 2-3 days, and they show a much better performance.

This is an early code release that works great but is slightly hastily released and probably requires some code reading of inline comments (which I tried to be quite good with in general). I will be improving it over time but wanted to push the code out there because I promised it to too many people.

This current code (and the pretrained model) gets ~0.9 CIDEr, which would place it around spot #8 on the codalab leaderboard. I will submit the actual result soon.

a man is playing tennis on a tennis court

a train is traveling down the tracks at a train station

a cake with a slice cut out of it

a bench sitting on a patch of grass next to a sidewalk

You can find a few more example results on the demo page. These results will improve a bit more once the last few bells and whistles are in place (e.g. beam search, ensembling, reranking).

There's also a fun video by @kcimc, where he runs a neuraltalk2 pretrained model in real time on his laptop during a walk in Amsterdam.

Requirements

For evaluation only

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 nn
$ luarocks install nngraph
$ luarocks install image
```

We're also going to need the cjson library so that we can load/save json files. Follow their download link and then look under their section 2.4 for easy luarocks install.

If you'd like to run on an NVIDIA GPU using CUDA (which you really, really want to if you're training a model, since we're using a VGGNet), you'll of course need a 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 the cudnn backend (the pretrained checkpoint does), you also have to install cudnn. First follow the link to NVIDIA website, register with them and download the cudnn library. Then make sure you adjust your `LD_LIBRARY_PATH` to point to the `lib64` folder that contains the library (e.g. `libcudnn.so.7.0.64`). Then git clone the `cudnn.torch` repo, `cd` inside and do `luarocks make cudnn-scm-1.rockspec` to build the Torch bindings.

For training

If you'd like to train your models you will need loadcaffe, since we are using the VGGNet. First, make sure you follow their instructions to install `protobuf` and everything else (e.g. `sudo apt-get install libprotobuf-dev protobuf-compiler`), and then install via luarocks:

```
luarocks install loadcaffe
```

Finally, you will also need to install torch-hdf5, and h5py, since we will be using hdf5 files to store the preprocessed data.

Phew! Quite a few dependencies, sorry no easy way around it :\

I just want to caption images

In this case you want to run the evaluation script on a pretrained model checkpoint. I trained a decent one on the MS COCO dataset that you can run on your images. The pretrained checkpoint can be downloaded here: pretrained checkpoint link (600MB). It's large because it contains the weights of a finetuned VGGNet. Now place all your images of interest into a folder, e.g. `blah` , and run the eval script:

```
$ th eval.lua -model /path/to/model -image_folder /path/to/image/directory -num_images 10
```

This tells the `eval` script to run up to 10 images from the given folder. If you have a big GPU you can speed up the evaluation by increasing `batch_size` (default = 1). Use `-num_images -1` to process all images. The eval script will create an `vis.json` file inside the `vis` folder, which can then be visualized with the provided HTML interface:

```
$ cd vis
$ python -m SimpleHTTPServer
```

Now visit `localhost:8000` in your browser and you should see your predicted captions.

You can see an example visualization demo page here.

Running in Docker. If you'd like to avoid dependency nightmares, running the codebase from Docker might be a good option. There is one (third-party) docker repo here.

"I only have CPU". Okay, in that case download the cpu model checkpoint. Make sure you run the eval script with `-gpuid -1` to tell the script to run on CPU. On my machine it takes a bit less than 1 second per image to caption in CPU mode.

Beam Search. Beam search is enabled by default because it increases the performance of the search for argmax decoding sequence. However, this is a little more expensive, so if you'd like to evaluate images faster, but at a cost of performance, use `-beam_size 1` . For example, in one of my experiments beam size 2 gives CIDEr 0.922, and beam size 1 gives CIDEr 0.886.

Running on MSCOCO images. If you train on MSCOCO (see how below), you will have generated preprocessed MSCOCO images, which you can use directly in the eval script. In this case simply leave out the `image_folder` option and the eval script and instead pass in the `input_h5` , `input_json` to your preprocessed files. This will make more sense once you read the section below :)

Running a live demo. With OpenCV 3 installed you can caption video stream from camera in real time. Follow the instructions in torch-opencv to install it and run `videocaptioning.lua` similar to `eval.lua` . Note that only central crop will be captioned.

I'd like to train my own network on MS COCO

Great, first we need to some preprocessing. Head over to the `coco/` folder and run the IPython notebook to download the dataset and do some very simple preprocessing. The notebook will combine the train/val data together and create a very simple and small json file that contains a large list of image paths, and raw captions for each image, of the form:

```
[{ "file_path": "path/img.jpg", "captions": ["a caption", "a second caption of i"tgit ...] }, ...]
```

Once we have this, we're ready to invoke the `prepro.py` script, which will read all of this in and create a dataset (an hdf5 file and a json file) ready for consumption in the Lua code. For example, for MS COCO we can run the prepro file as follows:

```
$ python prepro.py --input_json coco/coco_raw.json --num_val 5000 --num_test 5000 --images_root coco/images --word_cc
```

This is telling the script to read in all the data (the images and the captions), allocate 5000 images for val/test splits respectively, and map all words that occur <= 5 times to a special `UNK` token. The resulting `json` and `h5` files are about 30GB and contain everything we want to know about the dataset.

Warning: the prepro script will fail with the default MSCOCO data because one of their images is corrupted. See this issue for the fix, it involves manually replacing one image in the dataset.

The last thing we need is the VGG-16 Caffe checkpoint, (under Models section, "16-layer model" bullet point). Put the two files (the prototxt configuration file and the proto binary of weights) somewhere (e.g. a `model` directory), and we're ready to train!

```
$ th train.lua -input_h5 coco/cocotalk.h5 -input_json coco/cocotalk.json
```

The train script will take over, and start dumping checkpoints into the folder specified by `checkpoint_path` (default = current folder). You also have to point the train script to the VGGNet protos (see the options inside `train.lua`).

If you'd like to evaluate BLEU/METEOR/CIDEr scores during training in addition to validation cross entropy loss, use `-language_eval 1` option, but don't forget to download the coco-caption code into `coco-caption` directory.

A few notes on training. To give you an idea, with the default settings one epoch of MS COCO images is about 7500 iterations. 1 epoch of training (with no finetuning - notice this is the default) takes about 1 hour and results in validation loss ~2.7 and CIDEr score of ~0.4. By iteration 70,000 CIDEr climbs up to about 0.6 (validation loss at about 2.5) and then will top out at a bit below 0.7 CIDEr. After that additional improvements are only possible by turning on CNN finetuning. I like to do the training in stages, where I first train with no finetuning, and then restart the train script with `-finetune_cnn_after 0` to start finetuning right away, and using `-start_from` flag to continue from the previous model checkpoint. You'll see your score rise up to about 0.9 CIDEr over ~2 days or so (on MS COCO).

I'd like to train on my own data

No problem, create a json file in the exact same form as before, describing your JPG files:

```
[{ "file_path": "path/img.jpg", "captions": ["a caption", "a similar caption" ...] }, ...]
```

and invoke the `prepro.py` script to preprocess all the images and data into and hdf5 file and json file. Then invoke `train.lua` (see detailed options inside code).

I'd like to distribute my GPU trained checkpoints for CPU

Use the script `convert_checkpoint_gpu_to_cpu.lua` to convert your GPU checkpoints to be usable on CPU. See inline documentation for why this separate script is needed. For example:

```
th convert_checkpoint_gpu_to_cpu.lua gpu_checkpoint.t7
```

write the file `gpu_checkpoint.t7_cpu.t7` , which you can now run with `-gpuid -1` in the eval script.

License

BSD License.

Acknowledgements

Parts of this code were written in collaboration with my labmate Justin Johnson.

I'm very grateful for NVIDIA's support in providing GPUs that made this work possible.

I'm also very grateful to the maintainers of Torch for maintaining a wonderful deep learning library.