CS354 Deep Learning: Assignment 1

Mako Bates

2021-03-23

Model

I pulled the 2012 AlexNet architecture from PyTorch hub. The model comes with a tidy divide between the feature-extraction convolutional layers and the fully-connected classifier layers. The function `fetch_model` takes a settings object that instructs it to use a pre-trained or un-trained feature-extractor and classifier. The settings also say which of the layers to lock and which to expose to further training/fine-tuning.

I ran my job in tripliciate, using the same approach for each part of the assignment. For **part 1** of the assignment I used a fully un-trained model, and exposed all layers to training. The settings file is copied here with some stuff elided.

```
{ ...
  "num_models": 5,
  "pretrained_features": false,
  "pretrained_classifier": false,
  "trainable_features": [
    true, true, true, true, true, true, true, true, true, true, true, true, true
  ],
  "trainable_classifier": [
    true, true, true, true, true, true, true
  ],
  "training_profiles": [
    { "name": "alex",
      "learning_rate": 0.01,
      "momentum": 0.9,
      "weight_decay": 0.0005 },
    ...
    { "name": "walter",
      "learning_rate": 1e-06,
      "momentum": 0.1,
      "weight_decay": 0.0005 }
  ],
  "max_batch_size": 100,
  "total_lifetime": 259200,
  ... }
```

For **part 2** I used a fixed pre-trained feature extractor, and just trained the classifier.

```
{ ...
  "num_models": 5,
  "pretrained_features": true,
  "pretrained_classifier": false,
  "trainable_features": [
    false, false, false, false, false, false, false, false, false, false, false, false, false
  ],
  "trainable_classifier": [
    true, true, true, true, true, true, true
  ],
  "training_profiles": [
    { "name": "alex",
      "learning_rate": 0.01,
      "momentum": 0.9,
      "weight_decay": 0.0005 },
    ...
    { "name": "walter",
      "learning_rate": 1e-06,
      "momentum": 0.1,
      "weight_decay": 0.0005 }
  ],
  "max_batch_size": 100,
  "total_lifetime": 259200,
  "slurm_file": "bates_cs354_a1_submit2.sh",
  ... }
```

For **part three** I again used a pre-trained feature-extractor and an untrained classifier, but I exposed all but the first of the convolutional layers to further training.

```
{ ...
  "num_models": 5,
  "pretrained_features": true,
  "pretrained_classifier": false,
  "trainable_features": [
    false, true, true, true, true, true, true, true, true, true, true, true, true
  ],
  "trainable_classifier": [
    true, true, true, true, true, true, true
  ],
  "training_profiles": [
    { "name": "alex",
      "learning_rate": 0.01,
      "momentum": 0.9,
      "weight_decay": 0.0005 },
    ...
    { "name": "walter",
      "learning_rate": 1e-06,
      "momentum": 0.1,
      "weight_decay": 0.0005 }
  ],
  "max_batch_size": 100,
  "total_lifetime": 259200,
  "slurm_file": "bates_cs354_a1_submit3.sh",
  "eon_lifetime": 2400
  ... }
```

Training

Not everything I tried worked, and it's unclear which of the decisions I made may have reduced or advanced my actual results.

- I tried a couple of things to achieve training parallelism within Slurm; so far as I can tell none of them worked so I removed them.
- I did my best to ensure that the model itself and the training were running on the GPU, but I had no way of validating that the job was actually using resources appropreatly.
- I used a leader-board with five slots as part of a ratchet approach to training. After a process finishes an epoch it runs it against the test data to get an accuracy score. This score is compared against the existing leaderboard. If it beats any of them, then the model weights are exported to disk, overwriting the prior bottom-place weights. Otherwise the new weights are discarded. In either case the process will load from disk the saved weights that have been least-worked-on for training in its next epoch. I don't know if this is a good idea or not.
- I planned a complex system of alternating hyper-parameters, as reflected in the long list of `training_profiles` in each of the settings files. Regardless of whether this was a good idea or not, I implemented the state-managment wrong, so in practice they were used in linear order.
- Having failed to explain to Slurm that I wanted my job to be run in parallele on multiple nodes, I resorted to just adding each job to the queue multiple times. To avoide blocking everyone else from using DeepGreen while my jobs were running, I set them up to recursively add themselves back to the end of the job queue every fourty minutes. This part worked fine.
- I had expected to be able to run many jobs simultainously, and designed/calibrated the leaderboard system under that assumption. In practice I rarely had more than two processes running simultaiously. This slowed down training, and there were some bugs in the system for transitioning between learning-hyper-parameters, so I had to manually intervene (stop all jobs and restart them) after the first day.

For actual training details:

- Loss was calculated using PyTorch's built-in `CrossEntropyLoss`.
- Weights were updated using PyTorch's built-in `SGD` optimizer.
- I followed the original AlexNet paper (instead of the relevant tutorial I found) and used `0.0005` weight-decay.
- The learning rate started at `10^-2`, and progressed to `10^-6`.
- The momentum switched between `0.9`, `0.5`, and `0.1`.

Results

All three attempts eventually reached models that the system was unable to improve on. (Attempt 1 may have still had room for improvement, I killed it early to ensure more resources would go to the other two.) I had 8-18 items in the Slurm queue for over 48 hours, but limitations in the way I set up logging make it difficult to say how much compute-time was used on any of the three attempts, or even how many epochs happened.

The top of each attempt's leaderboard when shut down all the jobs is as follows:

- Attempt 1 (train from scratch): **57.78%**
- Attempt 2 (pre-trained feature extractor): **65.56%**
- Attempt 3 (fine-tuned feature extractor): **71.92%**

References

In addition to resources provided in class, I found these pages and tutorials helpful:

- https://pytorch.org/hub/pytorch_vision_alexnet/
- https://kushaj.medium.com/training-alexnet-with-tips-and-checks-on-how-to-train-cnns-practical-cnns-in-pytorch-1-61daa679c74a
- https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf