Lee Lab Cluster/Modeling Overview

Hal Rockwell 6/29/21

June 30, 2021

Abstract

A guide to using the Mind computing cluster for machine learning purposes, particularly neural response modeling.

1 Cluster usage

If you're using deep learning in the lab, unless you're working with tiny networks or have a GPU machine of your own, you're going to end up using the CNBC computing cluster, currently known as Mind (previously Psych-o). For most information about the cluster, you should refer to its website,[1] but in this document I'll go over the basics of everyday usage.

When you log into Mind, you are on the "head node." You should not run anything computationally intensive on this node, as it's shared by all Mind users and you could slow them down (quick, low-scale file/data processing, like something you'd casually do on your laptop, is okay). You log into your home folder, for example `/home/hrockwel/`, but each user also has a data folder like `/user_data/hrockwel/`. Your disk space quota, displayed on login, is substantially larger for the `/user_data/` folder, so if you're downloading or generating large amounts of saved data, you should try to do it in that folder. I usually work entirely out of `/user_data/`, since there's little reason I'm aware of not to.

To actually run compute-heavy programs, you need to access the cluster's compute nodes through the SLURM scheduler, which controls and balances access to them. The overview of SLURM on the cluster wiki[2] is excellent, and I highly recommend reading it. The most vital SLURM commands to know are `squeue`, `srun`, and `scancel`. All of these are just command-line programs you run by typing into the prompt when you're logged in to the cluster. The first, `squeue`, prints a list of currently-running jobs (sessions on compute nodes) and some information on each one. The other two start and stop jobs as specified by the arguments you pass them.

An example `srun` command, broken down:

```
srun -N1 -p gpu -c 4 --mem=8GB --gres=gpu:1 --time=2:00:00 --out=log.txt python test.py &
```

- `-N1` specifies that you just need one node. It's not always necessary to include, but I'm not sure when it isn't, so I always include it to be safe. If you're running a truly massive job, you might use more than one node, I guess.
- `-p gpu` specifies that you're using the `gpu` partition. The other option is `cpu`, and you always need to specify which. If your job requires a GPU, you need to use the `gpu` partition, otherwise don't.
- `-c 4` specifies the number of CPU cores you're requesting, always necessary. You should keep this small (I usually use 2 but one might be enough) unless you're doing CPU-heavy processing, in which case you'll obviously want more.
- `--mem=8GB` specifies the amount of RAM you're requesting, always necessary. If you run out of memory, your job will be automatically canceled, so use a little more than you think you'll need. But not too much—you don't want to waste resources.

[1] https://ni.cmu.edu/computing/article-category/cluster/
[2] https://ni.cmu.edu/computing/knowledge-base/slurm-scheduler/

- `--gres=gpu:1` specifies that you want one GPU. Your code will usually only use more than one if you specifically design it to, so if you haven't, don't ask for more than one. I think most of the machines have 4 GPUs, so that's the maximum you can request on a single node (but for up-to-date info, check the wiki).

- `--time=2:00:00` specifies the maximum amount of time (format `D-HH:MM:SS`) the job will run for. If you don't include this, it will use a default value (at the time of writing, 4 hours). When the time runs out, your job will automatically end, but longer times will result in a lower queue priority, so treat this like RAM—as much as you need and no more.

- `--out=log.txt` Anything printed to `stdout` or `stderr` will direct to the specified file. If not included, they will go to `stdout` of the session that ran the job instead, which you might end up missing. It's useful to have a log file for non-interactive jobs to see what went wrong if something crashed (and to save whatever you're printing).

- `python test.py &` tells slurm what to run on the requested node. In this case, it will run the specified python file (in the python environment you launched the command from). For a non-interactive job like this, it will end the job as soon as the program ends, so if you requested more time than needed, you won't be taking up cluster space unnecessarily. The `&` at the end is not strictly necessary, but just forks after the `srun` command, allowing you to keep using the session you ran it from (without it, the session will be tied up waiting for the compute job to finish). Replacing this whole final part with `--pty bash` will enter you into an interactive session on the requested node instead, which is preferable for certain kinds of work.

A scattering of generally useful tips about using SLURM and the cluster in general, overlapping with the wiki's advice:

- The only way to use a GPU is to be running on a GPU compute node (with a GPU requested). Otherwise, your code will not even detect the presence of a GPU (for example `torch.cuda.is_available()` will return false on CPU nodes and the head node).

- If you want to use a graphical program from the cluster (which is generally a bad idea due to horrible input lag), you can add the flag `--x11` to the `srun` command. I'm not sure if this will work on Windows computers. You'll also need to configure SSH to allow it (I think `-Y` does this on Linux).

- When copying large numbers of files to and from the cluster, `rsync` works much faster than `scp`.

- Since there are multiple ways for compute jobs to abruptly halt, whether running out of time or memory, it's a very good idea to frequently save intermediate results (e.g. partially trained networks), if possible, rather than just saving everything at the end.

- If you must run many (dozens or even hundreds) of jobs at once, try to make them as short as possible (preferably below four hours), and as resource-light as possible too. The cluster is often less busy overnight and on weekends, too.

- I always run the `squeue` command before running any jobs to get an idea of how busy the cluster is at the moment. It's also nice, when checking on your own jobs, to pipe its output to `grep <username>` to filter out everyone else's—I have a short alias for this in my `.bashrc`.

- Writing a bash script that uses `SBATCH`, as described in the wiki page, to run jobs is a more careful way of doing it than the `srun` commands I described, and preferable if you're doing something more complicated (or want better logging, even emails when the job finishes/fails!).

- This has been happening less often lately, but occasionally the disk speed of the cluster will grind to a halt, and even basic tasks like searching a directory will take several seconds or even longer. This can make your scripts run unbearable slowly, just because loading in packages (or saving results) takes forever. I don't know why this happens and have no solution to it other than to wait a few hours until it stops.

- The two most useful functions of `scancel` are cancelling single jobs by ID and cancelling all of your jobs by username. An example of the first case is `scancel 123456`, where the number given is the ID of the job you want to cancel, visible in the results of `squeue`. An example of the second case is `scancel -u hrockwel`, which cancels all jobs by user `hrockwel` (if that's you). I don't know of a good way to cancel many, but not all, jobs at once—there's probably one somewhere in the man page of `scancel`.

2 The Conda package manager

Conda[3] is an indispensable package and environment manager for Python. It's the first thing to install on the cluster (following the installation instructions on its website) if you're working with Python. Having a dedicated Conda environment for a project allows you to easily identify and download which packages and versions of packages you need for it, and allows others to easily replicate your environment to run your code without any problems. (The downside is that installing packages with Conda is often painfully slow for unclear reasons, but it's well worth the extra couple minutes up-front).

There isn't much to say about Conda other than that it's worth using, especially on the cluster. It's fairly straightforward once you get it installed. Below is a bash script that gives an example of how to create and install a set of useful packages into a Conda environment. One additional thing to note is that not every package can be installed with Conda—it focuses primarily on packages relevant to scientific computing—however, you can install `pip` with it, and then install a wider range of packages into your Conda environment using `pip`.

```
# setting up an example miniconda environment
# after installing it according to website instructions
# the installed packages will take a fair amount of disk space
# maybe up to a gigabyte or so

# make and enter env -- good to have one of these for each project
# so you don't run into package version problems
# but it can be helpful to have a base package like this with general things
conda create -n basic_ml
source ~/.bashrc # not necessary outside of a script
conda activate basic_ml

# basics
conda install -c conda-forge numpy scipy matplotlib jupyterlab scikit-image
# ML stuff
conda install scikit-learn pytorch torchvision tensorflow

# and you're good to go
# can save the env to a file for easy replication
# though this has problems transferring between PC and Mac/Linux
conda env export > basic_ml.yml

# anyone with the file and conda can use the following line to install:
# conda env create -f basic_ml.yml
# although this may not transfer properly between mac/linux and windows
```

Conda has some weird issues about running in scripts, which aren't worth getting into in detail (and I don't fully understand), so if you try to just run this file with `bash`, it might not work correctly. However, running each of the commands in your normal shell should be fine. It's mainly meant as a reference for how to do the basics in Conda—creating environments, installing packages, and exporting/loading them from files.

[3] https://docs.conda.io/en/latest/miniconda.html

3 Using Torchvision models

The example Conda environment from the previous section installs four "machine learning" packages: Scikit-Learn, PyTorch, Torchvision, and TensorFlow. Scikit-Learn is handy for everything but deep learning—if you're doing linear or logistic regression, PCA, clustering or SVMs, it's the go-to. It also has very thorough documentation[4] and an intuitive API. In this guide, I don't have anything to say about TensorFlow, since I don't know how to use it. Avoid it if possible—people familiar with both prefer PyTorch for research work. Finally, PyTorch (and Torchvision, which is just a sub-package with common computer vision models) is the automatic differentiation platform of choice for most work in the lab.

If you've worked with NumPy arrays, PyTorch tensors are very similar, with many of their methods having the same names between packages. The key extensions of PyTorch are that the tensors and the operations performed on them can be moved to the GPU, vastly speeding them up when tensors are large, and automatic differentation, which automatically computes the gradients of tensors with respect to any indicated outputs, making deep learning a lot easier to implement. When dealing with neural networks in PyTorch, as you usually are, the central class is the `nn.Module`—networks are `nn.Module`s, usually made up of many simpler `nn.Module`s.

The intention of this section isn't to give a general guide of PyTorch or Torchvision, which you can find online[5] (or learn by Googling "pytorch how to do [thing you want to do]" over and over again—a foolproof strategy). Rather, it's meant to show you how to do some basic work with computer vision models that's fairly common in the lab—running trained models on images and extracting intermediate-level responses. This might not make much sense if you don't already have some basic familiarity with the framework.

On the next two pages is a Python script that loads in a Torchvision network pretrained on Imagenet and an image, preprocesses the image, then runs the network on the image, checking the class it predicts and extracting the responses of units in a certain intermediate convolutional layer. The whole script takes place on the CPU, since it's quite fast to run a model on a single image. It could be moved to the GPU (if one is available) by adding ".cuda()" to the end of every time a Torch object is created (just lines 14 and 33, I think). You can run it and follow along with the printed outputs (and displayed figure), seeing what they are in the comments.[6]

The central points are correctly reshaping and normalizing the input image for a Torchvision model, and extracting responses from a particular intermediate layer by locating that layer in the object's structure and creating a hook for it.

[4] https://scikit-learn.org/stable/

[5] https://pytorch.org/tutorials/. I don't know if these are any good.

[6] You'll need the image, of course, which should be available in the same place you found this PDF. Or you can use your own.

```
# example using a torchvision network
# extracting some of its activations
import numpy as np
import torch
import matplotlib.pyplot as plt

from skimage.io import imread
from torchvision import transforms
from torchvision.models import resnet34

# download a small network for the example
# only downloads the weights the first time, only if you set pretrained=True
# these are trained on Imagenet, basically trustworthy that they're good
network = resnet34(pretrained=True)

# get an idea of what the object looks like
print(network)

# load in an example image
# diagram of an octopus brain -- they look quite weird
img = imread('octopus.jpg')
# the 224x224 size is necessary for most Imagenet-trained networks
print(img.shape)

# process the image before passing into the network
img = img[np.newaxis, ...].transpose(0, 3, 1, 2) # NCHW format
img = img / 255. # convert to 0-1 range
# these are the mean and standard deviation of Imagenet for each color
# always necessary to normalize by them for Torchvision models
# since they were trained on images with that normalization
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
img = normalize(torch.tensor(img).float())

# now, say we want to get intermediate responses from some middle conv layer
# setup a hook to do this
# the hook runs the function on arguments (module, input, output)
# each time the layer processes an input
int_responses = []
hook = network.layer3[1].conv2.register_forward_hook(
        lambda m, i, o: int_responses.append(o.data.cpu().numpy()))

# run the image through
# always use no_grad when not training to speed things up
with torch.no_grad():
    predictions = network(img)

# what does the network think this is?
print(predictions.argmax())
# 600 -- "hook, claw" -- well it wasn't trained on octopus brains

# remove the hook for cleanliness
hook.remove()

# and examine our activations
# not much spatial resolution at this stage in the resnet
```

```
int_responses = int_responses[0]
print(int_responses.shape)

# can look at one channel's activation as an image
# though it's not really enlightening
plt.figure()
plt.imshow(int_responses[0, 0])
plt.title('Intermediate conv layer response to octopus brain image')
plt.show()
```

4 Neural response modeling

The other main thing we tend to do with neural networks in the lab is neural response modeling, predicting the recorded responses of real neurons in the brain from the images that they were shown, using an artificial neural network. This section will touch on training networks in PyTorch as well, which is applicable outside of neural response modeling, of course. Since the involved code tends to get long pretty quickly, the document doesn't have it embedded, but just points to the right spots in the lab Github.

4.1 General framing of the problem

In the simple, standard case of neural response modeling, you have a population of N recorded neurons. The monkey (or mouse) was presented with M images (tensors of grayscale pixel values with shape $H \times W$), and you have the corresponding average firing rates of each neuron in response to each image. You are trying to estimate the function:

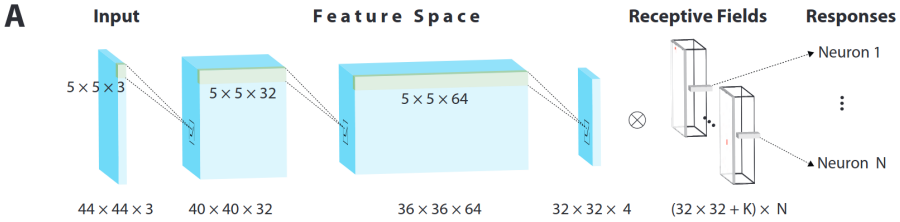
$$f : \mathbb{R}^{H \times W} \to \mathbb{R}^N$$

This function is the mapping between images and neural population responses. You have M examples to use for your estimation. There are infinitely many ways to parameterize this function. If you try to use linear regression, the simplest possible method, and learn a $HW \times N$ weight matrix, your predictive performance will be abysmal. So, you will use a neural network, as (at the time of writing) those are empirically the best method of parameterization.

Of course, there are infinitely many different neural networks you could use. One common method is to use a pretrained neural network, like the Torchvision model covered in the previous section, as a frontend, doing linear regression on the output of some intermediate layer. This is what we call "transfer learning," since the features used are learned in a different task, like object recognition (one with much more data than you can show to a monkey). This effectively replaces the problem with a new mapping to learn, still to $\mathbb{R}^N$, but from $\mathbb{R}^{H' \times W' \times C}$ instead (where H' and W' are the spatial dimensions of images at that intermediate layer, and C is the number of channels it has). I won't cover this in any further detail—the focus of this section is the "data-driven" approach, where the network mapping from the raw image space is trained from scratch to predict neural responses.

4.2 Network structure

The basic architecture of the standard data-driven network for neural response modeling is essentially that of the 2017 paper "Neural system identification for large populations separating 'what' and 'where'" by Klindt et. al.[7]. A single network is used to model the whole population of recorded neurons. This is outlined in Figure 5 of that paper, copied below:



The first part of the network is the shared convolutional core, or "feature space," as the figure calls it. This is just a normal convolutional neural network, with some number of channels. It's "shared" because the features it learns are used to predict all of the recorded neurons (as opposed to learning a separate network for each cell, which doesn't work as well). Typically, it has 2-3 layers, with the first layer's kernel being fairly large, and subsequent ones smaller, and channel counts the same for each layer, in the ballpark of 16-32 channels. Normalization layers like batch or instance normalization can be alternated with the convolutions. Empirically, non-saturating nonlinearities tend to work best; we usually use the Softplus nonlinearity, which is a smooth approximation to a ReLU, but I don't think it makes much difference. Sometimes, there is a max- or mean-pooling layer after the final convolutional layer.

After the final layer of the convolutional core, there is the readout. In the simplest case, this is just a fully connected layer, so if the dimensions of the final convolutional layer are $C \times H \times W$, a weight of size CHW is learned for each recorded neuron to predict its response. However, the 2017 paper introduced a factorized readout to reduce parameters, taking advantage of the fact that early visual neurons have receptive fields that are spatially localized, so they won't look at one feature in one part of the image and another feature in a different part. This means we instead learn an HW-size spatial receptive field for each output neuron, followed by a C-size feature receptive field, for a total of $HW + C$ parameters—substantially less than CHW. This allows for significant performance improvements. Applying these weights to the output of the convolutional layer leads to a single number for each recorded neuron, which is passed through a final nonlinearity to produce the predicted spike count for the cells.

Before moving on to training, I should note that all of these details are subject to changes and potential improvements. Channel counts and kernel sizes can be varied, recurrent layers can be added to predict a time-varying response (or better predict a static one), the parameter count of the readout can be further reduced with a clever reparameterization, etc. The architecture described is a baseline, at this point, and there's much interest in improving it.

Code implementing this model, hopefully in a fairly straightforward fashion, can be found on the lab Github, in the `gaya-data` repository.[8] The class in that file that implements the standard architecture is `BethgeModel` (the last author on the 2017 paper), and it uses the factorized readout in the `FactorizedLinear` class.[9]

[7] https://proceedings.neurips.cc/paper/2017/file/8c249675aea6c3cbd91661bbae767ff1-Paper.pdf

[8] https://github.com/leelabcnbc/gaya-data/blob/master/modeling/models/cnns/bethge.py

[9] There's a slight complication to this—at the time of writing, that code actually uses the `MultiFactorizedLinear` class, since I was experimenting with having multiple spatial receptive fields for each neuron. But with default parameters, it's just like using a single `FactorizedLinear` layer.

4.3 Training

Training neural networks is a very complicated matter, with too many little details to cover in a single document. I recommend referring to the simple training loop[10] and script utilizing it[11] in the lab Github repository on modeling the Gaya data. Reading through those concrete examples should make the overview here make more sense.

The loss function used in neural response prediction is composed of a predictive loss and several regularization terms. The predictive loss is typically a Poisson loss[12] instead of the more standard-for-regression MSE loss, as this empirically tends to work better (and matches the idea that the predicted spike counts are averages of samples from a Poisson distribution). There are usually two regularization terms: a measure of smoothness of the first-layer kernels, and a measure of sparsity (L1 norm) of the spatial and feature readouts. The first term pushes the network towards learning smooth, coherent features in the first layer (as opposed to noisy kernels), and doesn't seem to be as important in practice. The second term pushes the network to learn a localized receptive field for each recorded neuron, and also a sparse weighting of the shared convolutional core's features (the former is more intuitive, and possibly more important). Typical weightings of these regularization terms are included in the example files linked above, though it's good practice to perform a grid search over them for your problem.

For training, the data is always split into three groups: training, validation, and test sets, roughly 60/20/20. The test set, as always, is only used for reporting final performance values. The validation set is used to halt the training process, once the predictive loss over it stops decreasing, and for hyperparameter selection. The optimization method used is typically Adam, with a learning rate of around 10^{-3}, and the training is split into three stages of equal length, with the learning rate decreasing by a factor of 3 between each stage.

The simplest way to report final performance values is the average (across neurons) of the Pearson correlation between predicted and real neural responses. Depending on the dataset and methods used, this tends to be in the ballpark of 0.4.-0.7 if things are working correctly. However, a better metric to use might be CCNorm, which divides the squared correlation coefficient by a term CCMax, something like the maximum correlation coefficient that can be obtained given the trial-by-trial noise in the data. I don't typically use this, but code for computing CCMax can be found in the lab Github,[13] and it's good to at least be familiar with the concept. This improves comparisons of performance between neurons and datasets, which may have different levels of noise (with different numbers of trials, etc.).

Like the network architecture, these details of training are open to modification, but are a reasonable place to start.

[10] https://github.com/leelabcnbc/gaya-data/blob/master/modeling/train_utils.py
[11] https://github.com/leelabcnbc/gaya-data/blob/master/modeling/scripts/train_data_driven_cnn.py
[12] https://github.com/leelabcnbc/gaya-data/blob/master/modeling/losses.py
[13] https://github.com/leelabcnbc/strflab-python/blob/master/strflab/stats.py

4.4 Yimeng's modeling framework

The previous section link to the code I wrote for constructing and training models on the Gaya data. However, for his final thesis chapter, Yimeng (a PhD student in the lab who worked on neural response modeling and graduated Spring 2021) wrote a much more complex and flexible framework for constructing and training models. Conceptually, the actual networks and training are similar to what my code does, but the way his repository[14] is structured allows for much more flexibility and modularity. This comes at the cost of being difficult to understand at first. The point of this section is to reduce that cost slightly by giving a brief overview of his code's structure. Yimeng's thesis addressed the question of whether recurrent models could improve over the standard feedforward one (described above) for neural response prediction, so there is code for both, but for simplicity I will only address his feedforward models.

Yimeng's models are first defined architecturally by a JSON structure in `thesis_v2/models/maskcnn_polished/builder.py`. The main function in this file takes in necessary parameters and outputs a JSON specification of the network structure. It does this by putting together JSON specifications of each block, the basic ones taken from the separate `pytorch-module-in-json`[15] repository, loaded in with `load_modules`[16]. The factorized readout layer is not one of these basic blocks, and so its JSON structure is defined in `thesis_v2/blocks_json/maskcnn.py`, while its actual PyTorch code is in `thesis_v2/blocks/maskcnn/nn_modules.py`. Eventually, this JSON specification of the network is turned into a PyTorch `nn.Module` by the `build_net` function from the separate repository—the details of this process are only relevant if you're trying to expand the JSONnet framework (and I don't know them).

The more complicated part is Yimeng's training code. The outward-facing part of it, a function you could call a single time in a script to train a network with your desired parameters, is in `scripts/training/yuanyuan_8k_a_3day/maskcnn_polished/master.py` [17]. There are 7 layers of nested training functions between this and the actual updating of the model's weights. I will briefly step through those.

1. `thesis_v2/training_extra/maskcnn_like/training.py:train_one` is the outermost training function called. It sets the random seed for the model weights and calls:
2. `thesis_v2/training_extra/training.py:train_one_wrapper`, which creates the actual `nn.Module` object of the network with `build_net` (and does the same for the loss function and optimizer from their own specifications), and calls:
3. `train_one_inner` from the same file, which sets the random seed for the training process, converts the data into a useful class, and calls;
4. `thesis_v2/training/training_aux.py:training_wrapper`, which saves the statistics and other recorded data after training, but first calls:
5. `thesis_v2/training/training.py:train`, which loops over stages of training (for example, the learning rate is reduced between each stage) and calls:
6. `train_one_phase` from the same file, which loops over epochs, and calls (in addition to similar functions on validation set, and controlling early stopping):
7. `_train_one_epoch` from the same file, which loops over batches in a single epoch, and does the actual updating of the model weights with the loss function and optimizer.

The advantage of using Yimeng's code is that the JSON specification of networks, and his nested training loops, allow easy and reproducible experimentation with almost any feature of the process imaginable (once you understand all the parts). This allows you to vary things in your experiments by just changing arguments to functions, instead of rewriting chunks of code, which is a much better way to do things. For example, if you dig around in my `gaya-data` repository, you'll find a lot of

[14] https://github.com/leelabcnbc/thesis-yimeng-v2

[15] https://github.com/leelabcnbc/pytorch-module-in-json

[16] https://github.com/leelabcnbc/thesis-yimeng-v2/blob/master/thesis_v2/blocks/__init__.py

[17] https://github.com/leelabcnbc/thesis-yimeng-v2/blob/master/scripts/training/yuanyuan_8k_a_3day/maskcnn_polished/master.py

similar training scripts and model definitions, each updated a lot of times, with lots of duplicated code, making it hard to keep track of what's going on. Properly used, Yimeng's framework should let you avoid that and allow for much cleaner work.