Patterns for research in machine learning

18 Jul 2012

Here I list a handful of code patterns that I wish I was more aware of when I started my PhD. Each on its own may seem pointless, but collectively they go a long way towards making the typical research workflow more efficient. And an efficient workflow makes it just that little bit easier to ask the research questions that matter.

My guess is that these patterns will not only be useful for machine learning, but also any other computational work that involves either a) processing large amounts of data, or b) algorithms that take a significant amount of time to execute.

Disclaimer: The ideas below have resulted from my experiences working with bash. Other IDEs, languages or frameworks may have better solutions for the kinds of problems that I'm trying to address.

Always, always use version control.

This is useful because:

- It makes it easier to collaborate with others.
- It makes it easier to replicate experimental results.
- It makes it easier to work on your code-base from multiple computers.

Separate code from data.

Create spatial separation between your source code and your data files:

```
project_name/code/
project_name/data/
```

This is useful because:

- It makes it easier to share your code with others.

ali@arkitus.com (mailto:ali@arkitus.com)
(https://twitter.com/arkitus)
(http://www.linkedin.com/in/smalieslami)
(https://www.facebook.com/alieslami)

- It makes it easier to swap between datasets.

Even better would be to always assume that your code and your data are located independently:

```
/path/to/code/
/other/path/to/data/
```

This is useful because:

- It reinforces the separation.
- It's an easy way of hiding your data from revision control.

The data folder will often be too large to store on your machine, and you will have no choice but to separate the two.

Separate input data, working data and output data.

It can be useful to think of data as belonging to three distinct categories:

- **Input** files come with the problem. They never change.
- **Working** files are generated by your algorithms as they work. They always change.
- **Output** files are generated by your algorithms when they finish work successfully. They rarely change.

The directory structure will now look something like this:

```
/path/to/code/
/other/path/to/data/input/
/other/path/to/data/working/
/other/path/to/data/output/
```

This is useful because:

- You know that you can safely delete files from working/ (e.g. to save space). These files can, in theory, be regenerated simply by running your code again.
- It makes it easy to share the results in output/ with others (e.g. in presentations, or as input to LaTeX documents).

Modify input data with care.

- Always keep the raw data as distributed. Keep a note of where you got it from, together with any

read-me or licensing files that came with it.
- Write a one-touch script to convert the raw data into whatever format you use for your own code.
- Don't ever clean data by hand, and if you do, document it thoroughly, change by change.

Save everything to disk frequently.

- Save the model parameters to disk at suitable intervals.
- If a figure is useful for run-time diagnosis, then it should probably be saved to disk too.
- When you run your algorithm on different datasets, store the output in separate folders.
- Store the output of each day's work in a separate folder.

This is what the working folder might look like:

```
working/18_07_2012/dataset_1/
working/18_07_2012/dataset_2/
working/19_07_2012/dataset_1/
working/19_07_2012/dataset_2/
```

Inside the 18_07_2012/dataset_1 folder you might find:

```
dataset_1/likelihood_curve_iteration_100.eps
dataset_1/likelihood_curve_iteration_200.eps
dataset_1/likelihood_curve_iteration_300.eps
dataset_1/model_parameters_iteration_100.dat
dataset_1/model_parameters_iteration_200.dat
dataset_1/model_parameters_iteration_300.dat
```

Separate options from parameters.

I often see code that stores algorithm parameters and model parameters in the same data structure. In my experience things work best when the two are separated.

- Options specify how your algorithm should run.
- Parameters specify the model, and are usually an output of your algorithm.

```
% set the options
options.run_name = '18_07_2012/dataset_1/';
options.dataset_path = '/other/path/to/data/input/da
taset_1.dat';
options.working_path = ['/other/path/to/data/working
/' options.run_name];
options.output_path = ['/other/path/to/data/output/'
options.run_name];
options.learning_rate = 0.1;
options.num_iterations = 300;

% load the data
data = deserialise(options.dataset_path);

% learn the parameters
parameters = train_model(options, data);
```

Some parameters will not be affected by the algorithm's execution, e.g. model size parameters or the model's hyper-parameters. I store these as parameters, but use values specified in options to initialise them.

Do not use global variables.

Whenever possible, communicate through function arguments:

```
% set the options
options = ...

% load the data
data = ...

% learn the parameters
parameters = train_model(options, data);
```

and not through global variables:

```
global options, data;

% set the options
options = ...

% load the data
data = ...

% learn the parameters
parameters = train_model(); % assumes options and da
ta have been set globally
```

This is useful because:

- It makes it much easier to debug your code.
- It makes it easier to parallelise your code.

Record the options used to generate each run of the algorithm.

```
% set the options
options = ...

% load the data
data = ...

% learn the parameters
parameters = train_model(options, data);

% store the results
serialise(options, 'options.dat', options.working_pa
th);
serialise(parameters, 'parameters.dat', options.work
ing_path);
```

This is useful because it makes it easier to reproduce results. For completeness you may also want to:

- Consider setting the random number generator seed to a value specified in options.
- Consider saving a copy of the code used to execute each run.

Make it easy to sweep options.

```
% set the options
options.learning_rate = 0.1;
options.latent_dimensions = {10, 20};
options.num_iterations = {300, 600};

% load the data
data = ...

% sweep the options
for options_configuration in get_configurations(opti
ons)

    % learn the parameters
    parameters = train_model(options, data);

    % store the results
    serialise(parameters, 'parameters.dat'], ...
              [options.working_path '_' options_conf
iguration.name]);

end
```

The function get_configurations() can be written in such a way to make the above code segment train 4 different models, one for each valid combination of variables that are being swept over, and store the results in separate directories:

```
working/latent_dimensions_10_num_iterations_300/
working/latent_dimensions_20_num_iterations_300/
working/latent_dimensions_10_num_iterations_600/
working/latent_dimensions_20_num_iterations_600/
```

This is useful because it makes it easier to try out different algorithm options. If you have access to a cluster, you can easily use this to distribute each run to a different computer.

Make it easy to execute only portions of the code.

If your code can conceptually be thought of as some sort of pipeline where computations are made sequentially:

Write your main script in such a way that you can specify which computations you want to execute. Store the results of each part of the computation to disk. For

example, the following command runs the preprocess_data, initialise_model and train_model scripts.

```
>> run_experiment('dataset_1_options', '|preprocess_
data|initialise_model|train_model|');
```

And this command runs the only the train_model script but also evaluates its performance:

```
>> run_experiment('dataset_1_options', '|train_model
|evaluate_model|');
```

loading the preprocessed data and initialised model from disk.

Since the run_experiment() function might potentially be performing complex tasks such as: loading options from disk, sweeping parameters, communicating with a cluster of computers or managing the storing of results, you do not want to have to run the script manually for every run.

In my experience, commenting out segments of code to simulate this behaviour is a waste of time in the long-run. For complex projects you will constantly be switching between work on different parts of the pipeline.

Use checkpointing.

Your experiments will occasionally fail during execution. This is particularly true when many are run in parallel.

- Store the entire state (counters and so on) to disk at suitable intervals.
- Write code that, once activated, continues running the algorithm from the latest saved state.
- Make sure it is clearly made visible that the algorithm is starting from a saved state.

```
% set the options
options = ...

% load the data
data = ...

if saved_state_exists(options)

    % load from disk
    [parameters, state] = deserialize_latest_params_
state(options.working_path);

    % command line output
    disp(['Starting from iteration ' state.iteration
]);

else

    % initialize
    parameters = init_parameters();
    state = init_state();

end

% learn the parameters
parameters = train_model(options, data, parameters,
state);
```

Write demos and tests.

Your project structure will now look like:

```
project/
    code/
    data/
    demos/
    tests/
```

This is useful for reasons which should hopefully be relatively clear!

Other thoughts.

- Read The Pragmatic Programmer (http://www.amazon.co.uk/gp/product/020161622X/ref=as_li_ss_tl?ie=UTF8&camp=1634&creative=19450&creativeASIN=020161622X&linkCode=as2&tag=booka0c-21) by Hunt and Thomas.
- Always estimate how long you expect an experiment to take to run.
- Keep a journal that explains why you ran each

experiment and what your findings were.
- Do most of your coding and debugging with datasets that take ~10 seconds or less to process.
- Make it easy to swap in and out different models and fitness measures.
- Also read Charles Sutton (http://www.theexclusive.org/2012/08/principles-of-research-code.html)'s follow-up post and the HackerNews (http://news.ycombinator.com/item?id=4384317) discussion on this same topic.

Thanks to Iain Murray (http://homepages.inf.ed.ac.uk/imurray2/), Nicolas Heess (http://homepages.inf.ed.ac.uk/s0677090/), Jono Millin (http://www.jonomillin.com/), Sebastian Bitzer (http://www.cbs.mpg.de/staff/bitzer-11348), Isomorphismes (http://isomorphismes.tumblr.com/), John Langford (http://hunch.net/%7Ejl/), Yuxi Luo (http://havef.github.com/), Bob Carpenter (http://alias-i.com/) and Rob Lang (http://robsneuron.blogspot.co.uk/) for comments and suggestions.