Managing Machine Learning Experiments

Seb Arnold - May 23, 2018

Who Am I ?

- PhD Student in Reinforcement Learning and Optimization.
- Contributor to PyTorch, TensorFlow, neon, Keras.
- Maintainer of Randopt.

With the support of **Fund3**

The Problem

1. Code your experiment. (10%)

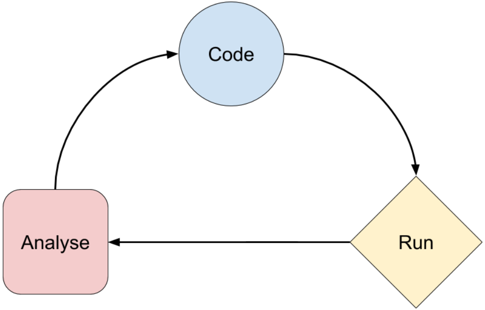


Figure 1: The Typical ML Loop

The Problem

1. Code your experiment. (10%)
2. Search for hyperparams. (70%)

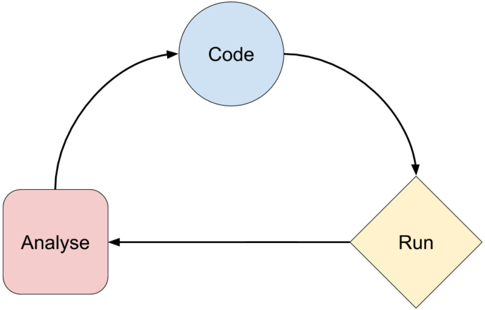


Figure 1: The Typical ML Loop

The Problem

1. Code your experiment. (10%)
2. Search for hyperparams. (70%)
3. Analyze the results. (20%)

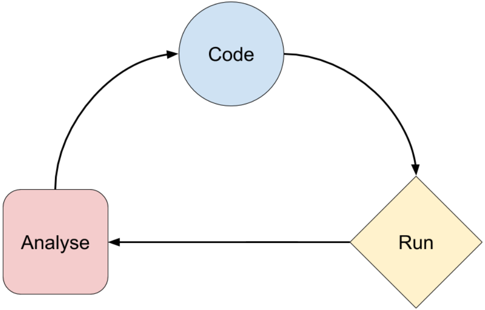


Figure 1: The Typical ML Loop

The Problem

1. Code your experiment. (10%)
2. Search for hyperparams. (70%)
3. Analyze the results. (20%)
4. Repeat. (∞)

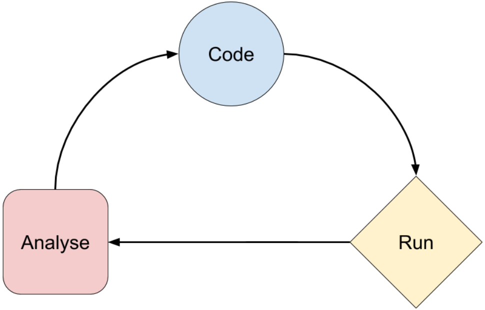
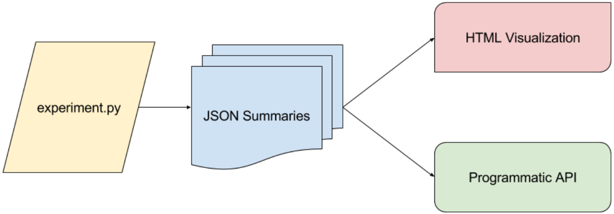


Figure 1: The Typical ML Loop

Randopt Overview



Features

- Human-readable format
- Support for parallelism / distributed / asynchronous experiments
- Command-line and Programmatic API
- Shareable Web Visualization
- Automatic Hyperparameter Search

Randopt 101

```
import randopt as ro
```

Randopt 101

```
import randopt as ro

exp = ro.Experiment(name='quadratic',
                    directory='mydir')
```

Randopt 101

```
import randopt as ro

exp = ro.Experiment(name='quadratic',
                    directory='mydir')

x, y = 3, 4

loss = lambda a, b: a**2 + b**2
```

Randopt 101

```
import randopt as ro

exp = ro.Experiment(name='quadratic',
                    directory='mydir')

x, y = 3, 4

loss = lambda a, b: a**2 + b**2

result = loss(x, y)
```

Randopt 101

```python
import randopt as ro

exp = ro.Experiment(name='quadratic',
                    directory='mydir')

x, y = 3, 4

loss = lambda a, b: a**2 + b**2

result = loss(x, y)

exp.add_result(result, data={
    'x': x,
    'y': y,
})
```

Randopt 101

```python
import randopt as ro

exp = ro.Experiment(name='quadratic',
                    directory='myresults')

x, y = 3, 4

loss = lambda a, b: a**2 + b**2

result = loss(x, y)

exp.add_result(result, data={
    'x': x,
    'y': y,
})
```

Directory structure

- experiment.py
- myresults/
  - 1519732... .json

1519732... .json

```
{
    'x': 3,
    'y': 4,
    'result': 25
}
```

Searching for Hyperparameters

The Problem: Finding good hyperparameters is akin to black magic.

Searching for Hyperparameters

The Problem: Finding good hyperparameters is akin to black magic.

- Requires familiarity with each model and each hyperparam.
- The relationship between hyperparameters is non-linear.
- Is task *AND* data dependent.
- A long and tedious task when obtaining a single result takes *weeks*.

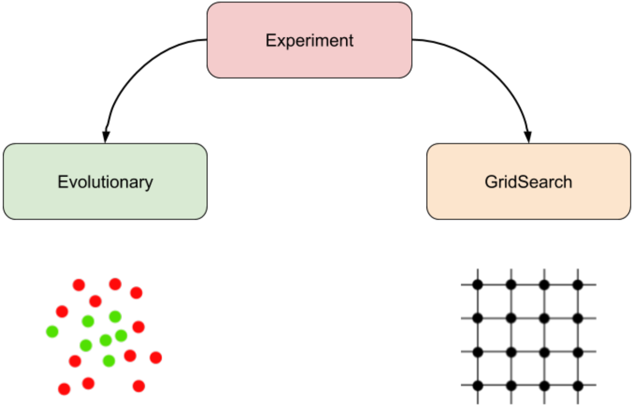
Searching for Hyperparameters

The Problem: Finding good hyperparameters is akin to black magic.

- Requires familiarity with each model and each hyperparam.
- The relationship between hyperparameters is non-linear.
- Is task *AND* data dependent.
- A long and tedious task when obtaining a single result takes *weeks*.

Note Automatic hyperparameter tuning is not optimal, but decent.

Search Algorithms



Random Search

Let’s modify our previous example.

```
exp = ro.Experiment(name='quadratic', directory='mydir', params={
    'x': ro.Uniform(-0.5, 0.1),
    'y': ro.Truncated(ro.Gaussian(), min=-0.5, max=0.5)
})
```

Random Search

Let's modify our previous example.

```
exp = ro.Experiment(name='quadratic', directory='mydir', params={
    'x': ro.Uniform(-0.5, 0.1),
    'y': ro.Truncated(ro.Gaussian(), min=-0.5, max=0.5)
})
```

Then we can record 100 results.

```
for i in range(100):
    exp.sample_all_params()
    result = loss(exp.x, exp.y)
    exp.add_result(result)
```

Or set values manually.

```
exp.x = 0.01
exp.y = 0.001
result = loss(exp.x, exp.y)
exp.add_result(result)
```

Grid Search

Let's use `GridSearch` instead of `Experiment`.

```
exp = ro.GridSearch(name='quadratic', directory='mydir', params={
    'x': ro.Choice([-0.5, -0.1, 0.1, 0.5]),
    'y': ro.Choice([-0.1, -0.001, 0.1, 0.3])
})
```

Grid Search

Let’s use `GridSearch` instead of `Experiment`.

```
exp = ro.GridSearch(name='quadratic', directory='mydir', params={
    'x': ro.Choice([-0.5, -0.1, 0.1, 0.5]),
    'y': ro.Choice([-0.1, -0.001, 0.1, 0.3])
})
```

Every call to `exp.sample_all_params()` does:

1. Open all saved JSONSummaries in `mydir/quadratic/`
2. Count the number of runs for each configuration defined in the grid.
3. Set parameters to least ran configuration.

Evolutionary Search

Let's use `Evolutionary` instead of `Experiment`.

```
exp = ro.GridSearch(name='quadratic', directory='mydir', params={
    'x': ro.Gaussian(0.0, 0.01),
    'y': ro.Choice([-0.1, 0, 0.1])
})
```

Evolutionary Search

Let's use `Evolutionary` instead of `Experiment`.

```
exp = ro.GridSearch(name='quadratic', directory='mydir', params={
    'x': ro.Gaussian(0.0, 0.01),
    'y': ro.Choice([-0.1, 0, 0.1])
})
```

Every call to `exp.sample_all_params()` does:

1. Select 10 best config from saved JSONSummary in `mydir/quadratic/`.
2. Uniformly at random, choose parent from the 10 best.
3. Sample perturbations from given samplers and apply them to parent.
4. Set parameters to perturbed parent.

Managing Experiments

The Problem: Keeping track of results is a pain.

Small scale

- For short runs, often rely on memory or napkin.
- For long runs, often rely on spreadsheet or notebook.

Large scale

- Database of results.

More problems

- What about collaboration ?
- What about different machines / drivers / tiny code changes ?
- What about **human-friendliness** ?

Exploring Results

1. Programmatic API

```
exp.count()  # 10
exp.all()  # Generator over all JSONSummaries

best = exp.top(10, fn=lambda a, b: a.result < b.result)  # Sort + Select
best.mean('x')
best.std('x')
best_of_best = best[:5]
best_of_best_of_best = best_of_best.filter(lambda a: a.result < 0.1)
```

Exploring Results

1. Programmatic API

```
exp.count()  # 10
exp.all()  # Generator over all JSONSummaries

best = exp.top(10, fn=lambda a, b: a.result < b.result)  # Sort + Select
best.mean('x')
best.std('x')
best_of_best = best[:5]
best_of_best_of_best = best_of_best.filter(lambda a: a.result < 0.1)
```

2. Filesystem

Edit / Copy / Remove summaries from the command line, file explorer, or your favorite editor (vim).

Visualizing Results

The Problem: Creating visualizations is tedious and often redundant.

In fact, you either want to plot the same old quantities or need something you've never done before.

Web Visualization

By calling

```
roviz.py mydir/quadratic
```

we obtain

Demo Time !

Custom Visualizations

Computing result statistics is easy.

```
import randopt as ro

exp = ro.Experiment(name='quadratic', directory='mydir')
results = list(exp.all())
xs = [r.x for r in results]
ys = [r.y for r in results]
zs = [r.result for r in results]
```

Which we plot with our favorite package.

```
from plotify import Plot3D
p = Plot3D('Quadratic Plot')
p.plot(xs, ys, zs, label='Result')
p.show()
```

Custom Visualizations

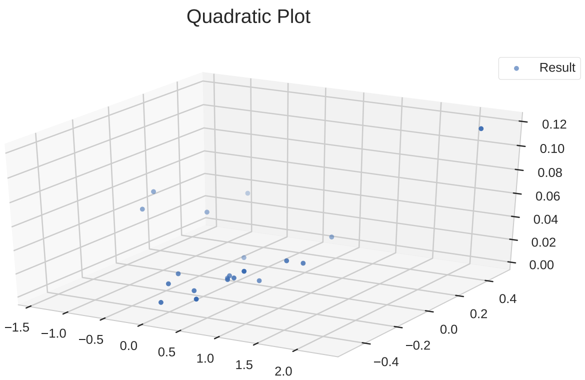


Figure 2: Custom 3D Plot

Advanced Features

ro.cli

- Python utility to create command-line interfaces.

ropt.py

- Command-line helper for hyperparameter search.

attachments

- Handling large data results.

parallel experiments

- Tapping into the super cluster you have.

Command-Line Interface

The Problem: CLIs are great, but so painful to write.

Command-Line Interface

The Problem: CLIs are great, but so painful to write.

```
@ro.cli
def run_experiment(x=23, y=12.0, dataset='mnist'):
    pass # Heavy computation

if __name__ == '__main__':
    ro.parse()
```

Command-Line Interface

The Problem: CLIs are great, but so painful to write.

```
@ro.cli
def run_experiment(x=23, y=12.0, dataset='mnist'):
    pass # Heavy computation

if __name__ == '__main__':
    ro.parse()
```

```
python experiment.py run_experiment --x 12 --y 0.1 --dataset 'cifar'
```

Command-Line Interface

The Problem: CLIs are great, but so painful to write.

```
@ro.cli
def run_experiment(x=23, y=12.0, dataset='mnist'):
    pass # Heavy computation

if __name__ == '__main__':
    ro.parse()
```

```
python experiment.py run_experiment --x 12 --y 0.1 --dataset 'cifar'
```

```
@ro.experiment
def run_experiment(x=23, y=12.0, dataset='mnist'):
    # Heavy computation
    return result, data, attachments
```

Command-Line Interface

The Problem: CLIs are great, but so painful to write.

```
@ro.cli
def run_experiment(x=23, y=12.0, dataset='mnist'):
    pass # Heavy computation

if __name__ == '__main__':
    ro.parse()
```

```
python experiment.py run_experiment --x 12 --y 0.1 --dataset 'cifar'
```

```
@ro.experiment
def run_experiment(x=23, y=12.0, dataset='mnist'):
    # Heavy computation
    return result, data, attachments
```

Thanks commandr for inspiration !

CLI Hyperparams Search

The Problem: 1 script for experiment, 1 for hyperparameters search.

CLI Hyperparams Search

The Problem: 1 script for experiment, 1 for hyperparameters search.

Using `ropt.py` we can generate commands for hyperparameter search.

```
python run_experiment \
        --x=12 \
        --y=23 \
        --dataset='mnist'
```

CLI Hyperparams Search

The Problem: 1 script for experiment, 1 for hyperparameters search.

Using `ropt.py` we can generate commands for hyperparameter search.

```
ropt.py python run_experiment \
                --x=12 \
                --y=23 \
                --dataset='mnist'
```

CLI Hyperparams Search

The Problem: 1 script for experiment, 1 for hyperparameters search.

Using `ropt.py` we can generate commands for hyperparameter search.

```
ropt.py python run_experiment \
                --x='Gaussian(0.0,0.1)' \
                --y='Choice([-0.1,0.0,0.1])' \
                --dataset='mnist'
```

CLI Hyperparams Search

The Problem: 1 script for experiment, 1 for hyperparameters search.

Using `ropt.py` we can generate commands for hyperparameter search.

```
ROPT_NSEARCH=120 ROPT_TYPE=Evolutionary ROPT_NAME='quadratic' ROPT_DIR='mydir'\
ropt.py python run_experiment \
                --x='Gaussian(0.0,0.1)' \
                --y='Choice([-0.1,0.0,0.1])' \
                --dataset='mnist'
```

CLI Hyperparams Search

The Problem: 1 script for experiment, 1 for hyperparameters search.

Using `ropt.py` we can generate commands for hyperparameter search.

```
ROPT_NSEARCH=120 ROPT_TYPE=Evolutionary ROPT_NAME='quadratic' ROPT_DIR='mydir'\
ropt.py python run_experiment \
                --x='Gaussian(0.0,0.1)' \
                --y='Choice([-0.1,0.0,0.1])' \
                --dataset='mnist'
```

```
python run_experiment --x 0.00213 --y -0.1 --dataset='mnist'
python run_experiment --x 0.0361 --y -0.1 --dataset='mnist'
python run_experiment --x -0.00887 --y 0.0 --dataset='mnist'
...
...
```

Attachments

The Problem: JSON Summaries aren't suited for large data results.

Attachments

The Problem: JSON Summaries aren't suited for large data results.

```
exp.add_result(result,
               data={'convergence': mylist},  # Data for Web / quick analysis
               attachment={'images': large_image_list})  # Larger result data
```

Attachments

The Problem: JSON Summaries aren't suited for large data results.

```
exp.add_result(result,
               data={'convergence': mylist},   # Data for Web / quick analysis
               attachment={'images': large_image_list})  # Larger result data
```

```
result = next(exp.all())
result.attachment['images']  # Lazy loaded
```

Attachments

The Problem: JSON Summaries aren't suited for large data results.

```
exp.add_result(result,
               data={'convergence': mylist},   # Data for Web / quick analysis
               attachment={'images': large_image_list})  # Larger result data
```

```
result = next(exp.all())
result.attachment['images']  # Lazy loaded
```

Attachments are

- for anything that is not human readable,
- linked to a particular JSON Summary,
- serialized via `cPickle`,
- lazy-loaded upon first access.

Parallel Experiments

The Problem: My compute can handle more than 1 experiment at a time.

Parallel Experiments

The Problem: My compute can handle more than 1 experiment at a time.

Solution: your favorite way of syncing a directory among computing nodes.

Some examples:

- single desktop machine: use multiple processes.
- compute cluster: use a shared-memory node.
- collaborators: use git/Dropbox synced folder.

Randopt does not impose constraint on the sharing strategy !

Even More Features

Features Not Covered

- Multi-Objective Optimization (`ro.objectives`)
- Plugins and Extensions (BayesOpt, Live Plotting, HO Monitoring)

Even More Features

Features Not Covered

- Multi-Objective Optimization (`ro.objectives`)
- Plugins and Extensions (BayesOpt, Live Plotting, HO Monitoring)

Future Features

- Performance Improvements
- Debugging ML Models
- Fancier Built-in Visualizations
- Your biggest ML hurdle ?

Fin

Thank you !

Fin

Thank you !

Learn more at: github.com/seba-1511/randopt

/seba-1511

seba-1511.com

/seba-1511