slurmjobs

Release 1.0.0

Bea Steers

May 18, 2022

CONTENTS

Automating Slurm job generation.

Generate a set of *.sbatch* files over a grid of parameters. A run script is created which will submit all generated jobs as once.

Now with *slurmjobs.Singularity()* support!

You can also use *Shell* which excludes the slurm/module references so you can test & run on your local machine or test server.

CHAPTER

ONE

INSTALLATION

```
pip install slurmjobs
```

CHAPTER

TWO

USAGE

```python
import slurmjobs

jobs = slurmjobs.Singularity(
   'python train.py', email='me@nyu.edu', n_gpus=2)

# generate jobs across parameter grid
run_script, job_paths = jobs.generate([
   # two different model types
   ('model', ['AVE', 'AVOL']),
   # try mono and stereo audio
   ('audio_channels', [1, 2]),
], epochs=500)

# NOTE:
#       extra keywords (e.g. ``epochs``) are also passed as arguments
#       across all scripts and are not included in the job name.

# ******************************
# ** everything was generated ** - now let's see the outputted paths.

slurmjobs.util.summary(run_script, job_paths)
```

Output:

```
Generated 4 job scripts:
         jobs/train/train,model-AVE,audio_channels-1.sbatch
         jobs/train/train,model-AVE,audio_channels-2.sbatch
         jobs/train/train,model-AVOL,audio_channels-1.sbatch
         jobs/train/train,model-AVOL,audio_channels-2.sbatch

To submit all jobs, run:
. jobs/train/run.sh
```

2.1 Tutorial

Let's go through the process of running your code in a singularity container.

2.1.1 Setting up Singularity

First we need to create the overlay that will contain our anaconda distribution along with any packages we want to install to the environment.

`slurmjobs` provides a script that will do that for you! Just type:

```
singuconda
```

And it will ask you for some values. All of which you can mash enter through if you don't care about them (which will use their defaults). The defaults will give you a singularity overlay with an empty conda environment. The prompts give you the opportunity to change the overlay and sif file paths, as well as specify which packages you'd like installed, specified in the form of:

- **a pip requirements file:** i.e. `pip install -r ./your/requirements.txt` - you provide: `./your/requirements.txt`
- **a local python package: a repo directory containing a `setup.py` file where your project source lives.** i.e. `pip install -e ./your/package` - you provide: `./your/package`
- **any conda packages that you want to install: conda packages** i.e. `conda install numpy librosa` - you provide: `numpy librosa`

Once we have that information the script will:

1. Check if the overlay file already exists. If not, then we copy the overlay file to your current directory.
2. We then enter the singularity container and install miniconda
3. Then we install all of the pip/conda stuff you requested to conda

Then you're all set! :) You have a singularity overlay with anaconda and any packages you asked it to install.

Note: If people would prefer, I can also refactor `singuconda` into a Python class to allow you to configure it with the rest of your slurmjobs code.

2.1.2 Generating Jobs

Now let's generate some job scripts!

The simplest configuration you can give is:

```
import slurmjobs

def generate():
    jobs = slurmjobs.Singularity('python project/train.py')

    # generate the jobs, providing a grid of
    # parameters to generate over.
    run_script, job_paths = jobs.generate([
        # two different model types
```

(continues on next page)

(continued from previous page)

```
        ('model', ['AVE', 'AVOL']),
        # try mono and stereo audio
        ('audio_channels', [1, 2]),
    ], epochs=500)

    # print out a summary of the scripts generated
    slurmjobs.util.summary(run_script, job_paths)

if __name__ == '__main__':
    import fire
    fire.Fire()
```

Then you can just generate the jobs by doing:

```
python jobs.py generate
```

And your files will be found in: `./jobs/project.train` (name automatically derived from the command name.)

But you can do a lot more:

```
import os
import slurmjobs

def generate():
    jobs = slurmjobs.Singularity(
        'python train.py',
        # give the job batch a name
        name='my-train-script',
        # say your script uses hydra, so tell slurmjobs how to format your arguments
        cli='hydra',
        # set the working directory for your script
        root_dir='/scratch/myuser/myproject',
        # disable job backups. By default it'll save them as `~{name}_01` etc.
        backup=False,
        # set the email to whoever generates the jobs (nyu uses your netid
        # for both email and greene, so you can use $USER to make it easier
        # with multiple people)
        email=f'{os.getenv("USER")}@nyu.edu',
        # set the number of cpus and gpus to request per job
        n_cpus=2, n_gpus=2,
        # set the requested time (e.g. 2 days)
        time='2-0',
        # disable passing job_id to your script (if your script doesn't accept one)
        job_id=None,  # or change the key: job_id='my_job_id',
        # pass any arbitrary sbatch flags
        # see: https://slurm.schedmd.com/sbatch.html
        sbatch={
            ...
        },
        # you can also pass anything else here and it'll be changeable
        # in __init__ and available in the templates (e.g. if you extend the templates)
    )
```

(continues on next page)

(continued from previous page)

```
    run_script, job_paths = jobs.generate([
        ('model', ['AVE', 'AVOL']),
        ('audio_channels', [1, 2]),
    ], epochs=500)

    slurmjobs.util.summary(run_script, job_paths)

if __name__ == '__main__':
    import fire
    fire.Fire()
```

To add initialization and cleanup code around your command, see *Customizing your Template* which will tell you how to add custom code.

2.1.3 Parameter Grids

If you have a simple parameter grid, then you don't really have to think about this, and you can keep passing your grid like the previous example.

But you may also have a slightly more complex grid you want to try. If that is the case, then you can do:

```
from slurmjobs import Grid, LiteralGrid

g = Grid([
    ('a', [1, 2]),
    ('b', [1, 2]),
], name='train')

# append two configurations
g = g + LiteralGrid([{'a': 5, 'b': 5}, {'a': 10, 'b': 10}])

# create a bigger grid from the product of another grid
g = g * Grid([
    ('c', [5, 6])
], name='dataset')

# omit a configuration from the grid
g = g - [{'a': 2, 'b': 1, 'c': 5}]

# then
assert list(g) == [
    {'a': 1, 'b': 1, 'c': 5},
    {'a': 1, 'b': 1, 'c': 6},
    {'a': 1, 'b': 2, 'c': 5},
    {'a': 1, 'b': 2, 'c': 6},
    # {'a': 2, 'b': 1, 'c': 5},  omitted
    {'a': 2, 'b': 1, 'c': 6},
    {'a': 2, 'b': 2, 'c': 5},
    {'a': 2, 'b': 2, 'c': 6},
    {'a': 5, 'b': 5, 'c': 5},
    {'a': 5, 'b': 5, 'c': 6},
    {'a': 10, 'b': 10, 'c': 5},
```

(continues on next page)

(continued from previous page)

```
    {'a': 10, 'b': 10, 'c': 6},
]

# Then you can pass the grid like normal

jobs = slurmjobs.Singularity('python train.py')
run_script, job_paths = jobs.generate(g, epochs=500)
```

2.1.4 Breaking Arguments across multiple functions

Sometimes you may have a situation where your arguments need to be broken out across multiple functions. You can do this by naming your grids.

```
.. code-block:: python

class Singularity(slurmjobs.Singularity):
    # remember to extend a base template
    template = '''{% extends 'job.singularity.j2' %}

{% block command %}
{{ command }} {{ cli(args.main, indent=4) }}

python my_other_script.py {{ cli(args.other, indent=4) }}
{% endblock %}
    '''

g = slurmjobs.Grid([
    ('a', [1, 2]),
    ('b', [1, 2]),
], name='main')

g * slurmjobs.Grid([
    ('c', [1, 2]),
    ('d', [1, 2]),
], name='other')
```

2.1.5 Customizing Job IDs

The purpose of a job ID is to give your jobs a pretty name that is descriptive of its configuration so you can use it in filenames and log files while also being unique between job instances so that they're not both trying to write to the same place.

Canonically, job IDs are created using this pattern: `{key}-{value},{key2}-{value2},...`. This is meant as a naive but somewhat effective way of encoding the parameters.

But things aren't always that simple. The main cases that I see you needing to changing the job ID formatter are:

- your grid contains a long value like a list or long string
- your grid contains long float values that you want to shorten
- your grid contains objects that don't have a nice string representation (this would most likely lead to an issue with CLI formatting too, but I digress)

- your grid contains too many keys and you'd like to abbreviate them to avoid hitting filename length limits.

Out of the box we include some levers that you can pull to tweak your job ID.

```
class Singularity(slurmjobs.Singularity):
    # a dict of abbreviations (full_key -> abbreviated_key)
    key_abbreviations = {}
    # a length to clip the keys to (e.g. 2 would turn model into mo)
    abbreviate_length = None
    # limit the precision on float values (e.g. 3 means 3 decimal places)
    float_precision = None
```

But of course, I'm sure you may have other ways you want to do things, so you have full liberty to change the job ID generation. Just make sure that your job IDs are still unique between jobs!!

```
class Singularity(slurmjobs.Singularity):
    # formatting for a single key-value pair.
    # you can return a tuple, string, or None (to exclude)
    def format_id_item(self, k, v):
        # I'm wacky and like my keys backwards
        k = k[::-1]
        # e.g. special formatting for booleans
        if v is True:
            return k
        if v is False:
            return f'not-{k}'
        return k, v

    # formatting the entire job ID. Do what you like!
    def format_job_id(self, args, keys=None, name=None):
        return ','.join(
            [name]*bool(name),
            *[f'{k}-{self.format_id_item(args[k])}' for k in keys or args]
        )
```

2.1.6 Customizing your Template

Another thing that you'll probably want to end up doing at some point is to add some customization, initialization, or cleanup to your scripts.

Often I will personally add most of that internally in my scripts, but you may also prefer to add it with bash.

Here's the block structure in each of the templates:

- **base (`job.base.j2`)**
  - `header`: This has a description of the job and arguments
  - `body`: This is the main body of the script and wraps around everything. You can use this for top-level initialization / cleanup.
    - `environment`: load anaconda and your conda environment.
    - `main`: An empty wrapper around the `command` block where you can add script initialization / cleanup
      - `command`: The heart of the script. This is where your script gets passed its arguments.

- **shell (`job.shell.j2` extends `job.base.j2`)**
  - `body` > `main`: Calls the main block using `nohup` so that your shell jobs can survive dropped ssh connections.
- **sbatch (`job.sbatch.j2` extends `job.base.j2`)**
  - `header` > `arguments`: added sbatch arguments at the beginning of the file.
  - `body` > `modules`: This is where we do `module purge` and `module load cuda` etc.
- **singularity (`job.singularity.j2` extends `job.sbatch.j2`)**
  - `body`: wraps with a singularity call. All of the body is run inside the singularity container

Here's how you can add initialization / cleanup code around your command. Use `{{ super() }}` to add back in the parent template's code.

```
class Singularity(slurmjobs.Singularity):
    # remember to extend a base template
    template = '''{% extends 'job.singularity.j2' %}

{% block main %}

# initialization
export SOMETHING=asdfasdfasdf
mkdir -p my-directory

{# call the rest of the main block #}
{{ super() }}

# some cleanup
rm -r my-directory

{% endblock %}
    '''
```

If you want to add code outside the singularity container, you just need to do:

```
class Singularity(slurmjobs.Singularity):
    # remember to extend a base template
    template = '''{% extends 'job.singularity.j2' %}

{% block body %}
export SOMETHING=asdfasdfasdf
mkdir -p my-directory
{{ super() }}
rm -r my-directory
{% endblock %}
    '''
```

And if you want to delete a parent's section, just do:

```
class Slurm(slurmjobs.Slurm):
    # remember to extend a base template
    template = '''{% extends 'job.sbatch.j2' %}
```

(continues on next page)

(continued from previous page)

```
{% block modules %}{% endblock %}
    '''
```

2.1.7 Customizing Argument Formatting

You can define your own formatter by subclassing *slurmjobs.args.Argument()*. If your class name ends with `'Argument'`, you can omit that when passing the cli name. This works by gathering subclasses matching the passed string against their names. If the class ends with 'Argument', the suffix will be removed. If you don't want a class to be available, prefix the name with an underscore.

Example:

```
import slurmjobs

class MyCustomArgument(slurmjobs.args.Argument):
    @classmethod
    def format_arg(cls, k, v=None):
        if v is None:
            return
        # idk do something fancy
        return '..{}@{}'.format(k, cls.format_value(v)) # ..size@10

batch = slurmjobs.SBatch('echo', cli='mycustom')
print(batch.command, batch.cli(size=10, blah='blorp'))
# echo ..size@10 ..blah@blorp
```

2.2 Sample Files

This is a quick sample of the generated files.

2.2.1 Singularity

We have:

- `jobs/{name}/{job_id}.sbatch`: The sbatch file for each job.
- `jobs/{name}/run.sh`: A script that submits all jobs at once.

```
import slurmjobs

jobs = slurmjobs.Singularity(
   'python train.py', name='sing', backup=False,
   email='me@nyu.edu', n_gpus=2)

run_script, job_paths = jobs.generate([
   ('model', ['AVE', 'AVOL']),
   ('audio_channels', [1, 2]),
], epochs=500)

slurmjobs.util.summary(run_script, job_paths)
```

Output:

```
Generated 4 job scripts:
         jobs/sing/sing,model-AVE,audio_channels-1.sbatch
         jobs/sing/sing,model-AVE,audio_channels-2.sbatch
         jobs/sing/sing,model-AVOL,audio_channels-1.sbatch
         jobs/sing/sing,model-AVOL,audio_channels-2.sbatch

To submit all jobs, run:
. jobs/sing/run.sh
```

jobs/sing/sing,model-AVE,audio_channels-1.sbatch

```
#!/bin/bash
#SBATCH --job-name=sing,model-AVE,audio_channels-1
#SBATCH --mail-type=ALL
#SBATCH --mail-user=me@nyu.edu
#SBATCH --output=jobs/sing/slurm/slurm_%j__sing,model-AVE,audio_channels-1.log
#SBATCH --time=3-0
#SBATCH --mem=48GB
#SBATCH --gres=gpu:2


##########################
#
# Job: sing,model-AVE,audio_channels-1
# Args:
# {'audio_channels': 1,
#  'epochs': 500,
#  'job_id': 'sing,model-AVE,audio_channels-1',
#  'model': 'AVE'}
#
##########################




###### ( hop into the singularity o_O )
singularity exec  \
    --nv \
    --overlay overlay-5GB-200K.ext3:ro \
    /scratch/work/public/singularity/cuda11.0-cudnn8-devel-ubuntu18.04.sif \
    /bin/bash << EOF
echo "@: entered singularity container"
source /ext3/env.sh







python train.py \
    --model=AVE \
```

(continues on next page)

(continued from previous page)

```
    --audio_channels=1 \
    --epochs=500 \
    --job_id=sing,model-AVE,audio_channels-1



###### (escape from the singularity @o@ )
echo "@: exiting singularity container"
exit 0;
EOF
```

jobs/sing/sing,model-AVE,audio_channels-2.sbatch

```
#!/bin/bash
#SBATCH --job-name=sing,model-AVE,audio_channels-2
#SBATCH --mail-type=ALL
#SBATCH --mail-user=me@nyu.edu
#SBATCH --output=jobs/sing/slurm/slurm_%j__sing,model-AVE,audio_channels-2.log
#SBATCH --time=3-0
#SBATCH --mem=48GB
#SBATCH --gres=gpu:2


##########################
#
# Job: sing,model-AVE,audio_channels-2
# Args:
# {'audio_channels': 2,
#  'epochs': 500,
#  'job_id': 'sing,model-AVE,audio_channels-2',
#  'model': 'AVE'}
#
##########################




###### ( hop into the singularity o_O )
singularity exec  \
    --nv \
    --overlay overlay-5GB-200K.ext3:ro \
    /scratch/work/public/singularity/cuda11.0-cudnn8-devel-ubuntu18.04.sif \
    /bin/bash << EOF
echo "@: entered singularity container"
source /ext3/env.sh
```

(continues on next page)

(continued from previous page)

```
python train.py \
    --model=AVE \
    --audio_channels=2 \
    --epochs=500 \
    --job_id=sing,model-AVE,audio_channels-2



###### (escape from the singularity @o@ )
echo "@: exiting singularity container"
exit 0;
EOF
```

jobs/sing/sing,model-AVOL,audio_channels-1.sbatch

```
#!/bin/bash
#SBATCH --job-name=sing,model-AVOL,audio_channels-1
#SBATCH --mail-type=ALL
#SBATCH --mail-user=me@nyu.edu
#SBATCH --output=jobs/sing/slurm/slurm_%j__sing,model-AVOL,audio_channels-1.log
#SBATCH --time=3-0
#SBATCH --mem=48GB
#SBATCH --gres=gpu:2


##########################
#
# Job: sing,model-AVOL,audio_channels-1
# Args:
# {'audio_channels': 1,
#  'epochs': 500,
#  'job_id': 'sing,model-AVOL,audio_channels-1',
#  'model': 'AVOL'}
#
##########################




###### ( hop into the singularity o_O )
singularity exec  \
    --nv \
    --overlay overlay-5GB-200K.ext3:ro \
    /scratch/work/public/singularity/cuda11.0-cudnn8-devel-ubuntu18.04.sif \
    /bin/bash << EOF
echo "@: entered singularity container"
source /ext3/env.sh
```

(continues on next page)

(continued from previous page)

```
python train.py \
    --model=AVOL \
    --audio_channels=1 \
    --epochs=500 \
    --job_id=sing,model-AVOL,audio_channels-1



###### (escape from the singularity @o@ )
echo "@: exiting singularity container"
exit 0;
EOF
```

jobs/sing/sing,model-AVOL,audio_channels-2.sbatch

```
#!/bin/bash
#SBATCH --job-name=sing,model-AVOL,audio_channels-2
#SBATCH --mail-type=ALL
#SBATCH --mail-user=me@nyu.edu
#SBATCH --output=jobs/sing/slurm/slurm_%j__sing,model-AVOL,audio_channels-2.log
#SBATCH --time=3-0
#SBATCH --mem=48GB
#SBATCH --gres=gpu:2


##########################
#
# Job: sing,model-AVOL,audio_channels-2
# Args:
# {'audio_channels': 2,
#  'epochs': 500,
#  'job_id': 'sing,model-AVOL,audio_channels-2',
#  'model': 'AVOL'}
#
##########################




###### ( hop into the singularity o_O )
singularity exec  \
    --nv \
    --overlay overlay-5GB-200K.ext3:ro \
    /scratch/work/public/singularity/cuda11.0-cudnn8-devel-ubuntu18.04.sif \
    /bin/bash << EOF
echo "@: entered singularity container"
source /ext3/env.sh
```

(continues on next page)

(continued from previous page)

```
python train.py \
    --model=AVOL \
    --audio_channels=2 \
    --epochs=500 \
    --job_id=sing,model-AVOL,audio_channels-2



###### (escape from the singularity @o@ )
echo "@: exiting singularity container"
exit 0;
EOF
```

jobs/sing/run.sh

```
#########################
#
# Job Batch: sing
# Params:
# [
#   ('model', ['AVE', 'AVOL']),
#   ('audio_channels', [1, 2]),
# ]
#
#########################



sbatch "jobs/sing/sing,model-AVE,audio_channels-1.sbatch"
sbatch "jobs/sing/sing,model-AVE,audio_channels-2.sbatch"
sbatch "jobs/sing/sing,model-AVOL,audio_channels-1.sbatch"
sbatch "jobs/sing/sing,model-AVOL,audio_channels-2.sbatch"
```

2.2.2 Slurm

We have:

- `jobs/{name}/{job_id}.sbatch`: The sbatch file for each job.
- `jobs/{name}/run.sh`: A script that submits all jobs at once.

```
import slurmjobs

jobs = slurmjobs.Slurm(
   'python train.py', name='slurm', backup=False,
```

(continues on next page)

(continued from previous page)

```
   email='me@nyu.edu', n_gpus=2)

run_script, job_paths = jobs.generate([
   ('model', ['AVE', 'AVOL']),
   ('audio_channels', [1, 2]),
], epochs=500)

slurmjobs.util.summary(run_script, job_paths)
```

Output:

```
Generated 4 job scripts:
         jobs/slurm/slurm,model-AVE,audio_channels-1.sbatch
         jobs/slurm/slurm,model-AVE,audio_channels-2.sbatch
         jobs/slurm/slurm,model-AVOL,audio_channels-1.sbatch
         jobs/slurm/slurm,model-AVOL,audio_channels-2.sbatch

To submit all jobs, run:
. jobs/slurm/run.sh
```

jobs/slurm/slurm,model-AVE,audio_channels-1.sbatch

```
#!/bin/bash
#SBATCH --job-name=slurm,model-AVE,audio_channels-1
#SBATCH --mail-type=ALL
#SBATCH --mail-user=me@nyu.edu
#SBATCH --output=jobs/slurm/slurm/slurm_%j__slurm,model-AVE,audio_channels-1.log
#SBATCH --time=3-0
#SBATCH --mem=48GB
#SBATCH --gres=gpu:2


##########################
#
# Job: slurm,model-AVE,audio_channels-1
# Args:
# {'audio_channels': 1,
#  'epochs': 500,
#  'job_id': 'slurm,model-AVE,audio_channels-1',
#  'model': 'AVE'}
#
##########################


##### Load Modules
    module purge
```

(continues on next page)

(continued from previous page)

```
python train.py \
    --model=AVE \
    --audio_channels=1 \
    --epochs=500 \
    --job_id=slurm,model-AVE,audio_channels-1
```

jobs/slurm/slurm,model-AVE,audio_channels-2.sbatch

```
#!/bin/bash
#SBATCH --job-name=slurm,model-AVE,audio_channels-2
#SBATCH --mail-type=ALL
#SBATCH --mail-user=me@nyu.edu
#SBATCH --output=jobs/slurm/slurm/slurm_%j__slurm,model-AVE,audio_channels-2.log
#SBATCH --time=3-0
#SBATCH --mem=48GB
#SBATCH --gres=gpu:2


##########################
#
# Job: slurm,model-AVE,audio_channels-2
# Args:
# {'audio_channels': 2,
#  'epochs': 500,
#  'job_id': 'slurm,model-AVE,audio_channels-2',
#  'model': 'AVE'}
#
##########################


##### Load Modules
    module purge




python train.py \
    --model=AVE \
    --audio_channels=2 \
    --epochs=500 \
    --job_id=slurm,model-AVE,audio_channels-2
```

jobs/slurm/slurm,model-AVOL,audio_channels-1.sbatch

```
#!/bin/bash
#SBATCH --job-name=slurm,model-AVOL,audio_channels-1
#SBATCH --mail-type=ALL
```

(continues on next page)

(continued from previous page)

```
#SBATCH --mail-user=me@nyu.edu
#SBATCH --output=jobs/slurm/slurm/slurm_%j__slurm,model-AVOL,audio_channels-1.log
#SBATCH --time=3-0
#SBATCH --mem=48GB
#SBATCH --gres=gpu:2


##########################
#
# Job: slurm,model-AVOL,audio_channels-1
# Args:
# {'audio_channels': 1,
#  'epochs': 500,
#  'job_id': 'slurm,model-AVOL,audio_channels-1',
#  'model': 'AVOL'}
#
##########################


##### Load Modules
    module purge




python train.py \
    --model=AVOL \
    --audio_channels=1 \
    --epochs=500 \
    --job_id=slurm,model-AVOL,audio_channels-1
```

jobs/slurm/slurm,model-AVOL,audio_channels-2.sbatch

```
#!/bin/bash
#SBATCH --job-name=slurm,model-AVOL,audio_channels-2
#SBATCH --mail-type=ALL
#SBATCH --mail-user=me@nyu.edu
#SBATCH --output=jobs/slurm/slurm/slurm_%j__slurm,model-AVOL,audio_channels-2.log
#SBATCH --time=3-0
#SBATCH --mem=48GB
#SBATCH --gres=gpu:2


##########################
#
# Job: slurm,model-AVOL,audio_channels-2
# Args:
# {'audio_channels': 2,
#  'epochs': 500,
```

(continues on next page)

(continued from previous page)

```
#  'job_id': 'slurm,model-AVOL,audio_channels-2',
#  'model': 'AVOL'}
#
##########################


##### Load Modules
    module purge




python train.py \
    --model=AVOL \
    --audio_channels=2 \
    --epochs=500 \
    --job_id=slurm,model-AVOL,audio_channels-2
```

jobs/slurm/run.sh

```
###########################
#
# Job Batch: slurm
# Params:
# [
#   ('model', ['AVE', 'AVOL']),
#   ('audio_channels', [1, 2]),
# ]
#
###########################



sbatch "jobs/slurm/slurm,model-AVE,audio_channels-1.sbatch"
sbatch "jobs/slurm/slurm,model-AVE,audio_channels-2.sbatch"
sbatch "jobs/slurm/slurm,model-AVOL,audio_channels-1.sbatch"
sbatch "jobs/slurm/slurm,model-AVOL,audio_channels-2.sbatch"
```

2.2.3 Shell

We have:

- `jobs/{name}/{job_id}.job.sh`: The shell file for each job.
- `jobs/{name}/run.sh`: A script that submits all jobs at once.

```
import slurmjobs

jobs = slurmjobs.Shell(
   'python train.py', name='shell', backup=False,
   email='me@nyu.edu', n_gpus=2)

run_script, job_paths = jobs.generate([
   ('model', ['AVE', 'AVOL']),
   ('audio_channels', [1, 2]),
], epochs=500)

slurmjobs.util.summary(run_script, job_paths)
```

Error:

Error:

```
Traceback (most recent call last):
  File "/home/docs/checkouts/readthedocs.org/user_builds/slurmjobs/checkouts/latest/
↪docs/_tmp.py", line 5, in <module>
    email='me@nyu.edu', n_gpus=2)
  File "/home/docs/checkouts/readthedocs.org/user_builds/slurmjobs/envs/latest/lib/
↪python3.7/site-packages/slurmjobs-1.0.0-py3.7.egg/slurmjobs/core.py", line 121, in _
↪__init__
    f"Unrecognized options: {wrong_options}. If you extended your template to use␣
↪additional options, "
TypeError: Unrecognized options: {'n_gpus'}. If you extended your template to use␣
↪additional options, please give them default values in your Class.options␣
↪dictionary.
```

2.3 Job Generation

class slurmjobs.**Singularity**(*command*, *overlay=None*, *sif=None*, **a*, ***kw*)

Generate jobs for sbatch + singularity. Use this for HPC Greene.

This is functionally equivalent to:

```
{# sbatch_args #}

# execute it inside the container
singularity exec --overlay ... ... /bin/bash << EOF

. /ext3/env.sh
```

(continues on next page)

(continued from previous page)

```
{# shell_code_body #}

EOF
```

To setup a singularity overlay with a anaconda python environment, please feel free to use https://gist.github.com/beasteers/84cf1eb2e5cc7bc4cb2429ef2fe5209e I'm in the process of integrating it with `slurmjobs`.

Source Tutorial: https://sites.google.com/a/nyu.edu/nyu-hpc/services/Training-and-Workshops/tutorials/singularity-on-greene

class slurmjobs.**Slurm**(**a*, *sbatch=None*, *modules=None*, *n_gpus=None*, *n_cpus=None*, *nv=None*, ***kw*)

Generate jobs for sbatch. This was used for HPC Prince. For HPC Greene, please use `Singularity`.

This is functionally equivalent to:

```
# sbatch_args
#SBATCH --nodes=...
#SBATCH --cpus-per-task=...
#SBATCH --gres=gpu:...
#SBATCH --mail-user=...
...

{# shell_code_body #}
```

class slurmjobs.**Shell**(*command*, *name=None*, *cli=None*, *root_dir=None*, *backup=True*, *job_id=True*, *template=None*, *run_template=None*, ***options*)

Generate batch jobs to be run without a task scheduler. This will create a grid of scripts that will be run using nohup, a program that will keep running your script even if the parent process (e.g. your ssh session) dies.

This is mainly just for doing small tests and whatnot.

```
import slurmjobs

jobs = Shell('python myscript.py')
jobs.generate([
    ('a', [1, 2]),
    ('b', [1, 2]),
])
```

class slurmjobs.**Jobs**(*command*, *name=None*, *cli=None*, *root_dir=None*, *backup=True*, *job_id=True*, *template=None*, *run_template=None*, ***options*)

The base class for Job generation. Sub-class this if you want to provide your own

```
import slurmjobs

batch = slurmjobs.SlurmBatch(
    'python train.py',
    email='mynetid@nyu.edu',
    conda_env='my_env')

# generate jobs across parameter grid
run_script, job_paths = batch.generate([
    ('kernel_size', [2, 3, 5]),
    ('nb_stacks', [1, 2]),
```

(continues on next page)

(continued from previous page)

```
    ('lr', [1e-4, 1e-3]),
], receptive_field=6)
```

We use Jinja2 for our script templating. See https://jinja.palletsprojects.com/en/3.0.x/templates/ for documentation about its syntax and structure.

options

Template options. Anything here is made available to the template.

Type dict

template

The Jinja2 template for the sbatch script.

Type str

job_template

The Jinja2 template for the jobs script that launches the sbatch files.

Type str

cli

The command line argument format your script uses. By default, this uses Fire. See `args.FireArgument`.

Type str, *slurmjobs.args.Argument*

job_id_arg

The argument name to use for passing the job ID. Set to None to ignore omit the job ID.

Type str

job_id_key_sep

The separator between key and value used in the job ID.

Type str

job_id_item_sep

The separator between key-value items in the job ID.

Type str

allowed_job_id_chars

Characters (other than alphanumeric) allowed in the job ID. Other characters are removed.

Type str, list

special_character_replacement

A string that can be used in place of special characters not found in `allowed_job_id_chars`. Default is empty.

Type str

key_abbreviations

A mapping from full name to abbreviation for keys in the job ID.

Type dict

abbreviate_length

The length to abbreviate keys in the job ID.

Type int

`float_precision`

The number of decimals to limit float values in the job ID.

Type int

`format_id_item`(*k*, *v*)

Formats a key-value pair for the job ID.

You can override this to change how each key-value pair in a job ID is formatted.

Parameters

- **k** (*str*) – The argument key.
- **value** (*any*) – The argument value.

Returns Return a tuple if you want it to be joined using `job_id_key_sep`. Return a string to be used as is. Return None to omit it from the job ID.

Be warned that omitting key/value pairs runs the risk of filename collisions which will mean multiple jobs overwritting each other in the same file.

`format_job_id`(*args*, *keys=None*, *name=None*, *ignore_keys=None*)

Convert a dictionary to a job ID.

Parameters

- **args** (*GridItem*) – The job dictionary.
- **keys** (*list, tuple*) – The keys that we want to include in the job ID.
- **name** (*str*) – The job name to include in the job ID. By default this will be the name associated with the Jobs object. Pass `False` to use no name.

`generate`(*grid_=None*, **a*, *ignore_job_id_keys=None*, ***kw*)

Generate slurm jobs for every combination of parameters.

Parameters

- **grid** (*Grid, list*) – The parameter grid.
- ***a** – Positional arguments to add to the command.
- ****kw** – Additional keyword arguments to pass to the command.

Returns

str: The path to the run script.

list[str]: The list of paths for each job file.

Return type tuple

`generate_job`(*job_id*, **a*, *_args=None*, *_grid=None*, ***params*)

Generate a single slurm job file

`generate_run_script`(*_job_paths*, *_grid=None*, ***kw*)

Generate a job run script that will submit all jobs.

2.4 Parameter Grids

Parameter grids!

This lets you do basic grid expansion and grid arithmetic.

```
g = Grid([
    ('a', [1, 2]),
    ('b', [1, 2]),
], name='train')

# append two configurations
g = g + LiteralGrid([{'a': 5, 'b': 5}, {'a': 10, 'b': 10}])

# create a bigger grid from the product of another grid
g = g * Grid([
    ('c', [5, 6])
], name='dataset')

assert list(g) == [
    {'a': 1, 'b': 1, 'c': 5},
    {'a': 1, 'b': 1, 'c': 6},
    {'a': 1, 'b': 2, 'c': 5},
    {'a': 1, 'b': 2, 'c': 6},
    {'a': 2, 'b': 1, 'c': 5},
    {'a': 2, 'b': 1, 'c': 6},
    {'a': 2, 'b': 2, 'c': 5},
    {'a': 2, 'b': 2, 'c': 6},
    {'a': 5, 'b': 5, 'c': 5},
    {'a': 5, 'b': 5, 'c': 6},
    {'a': 10, 'b': 10, 'c': 5},
    {'a': 10, 'b': 10, 'c': 6},
]
```

class slurmjobs.grid.**Grid**(*_Grid__grid*, *name=None*, ***constants*)

A parameter grid! To get all combinations from the grid, just do `list(Grid(...))`.

Parameters

- **grid** (*list, dict*) – The parameter grid. Should be either a dict or a list of key values, where the values are a list of values to use in the grid. Examples of valid inputs:

```
# simple grid
Grid([ ('a', [1, 2]), ('b', [1, 2]) ])
Grid({ 'a': [1, 2], 'b': [1, 2] })

# paired parameters
Grid([
    ('a', [1, 2]),
    (('b', 'c'), ([1, 2], [1, 2]))
])
Grid([
    ('a', [1, 2]),
    [{'b': 1, 'c': 1}, {'b': 2, 'c': 2}],
```

(continues on next page)

(continued from previous page)

```
])

# any of these are valid grid specs
g = slurmjobs.Grid([
    # basic
    ('a', [1, 2]),
    # paired
    (('b', 'c'), ([1, 1, 2, 2], [1, 2, 1, 2])),

    # literal list of dicts
    [{'d': i} for i in [1, 2]],
    # dict generator
    ({'e': i} for i in [1, 2]),
    # function
    lambda: [{'f': i} for i in [1, 2]],
    # function that returns a generator
    lambda: ({'g': i} for i in [1, 2]),

    # basic generator
    ('h', (x for x in [1, 2])),
    # basic function
    ('i', lambda: [x for x in [1, 2]]),
])
keys = 'abcdefghi'
assert list(g) == [
    dict(zip(keys, vals)) for vals in
    itertools.product(*([ [1, 2] ]*len(keys)))
]
```

- **name** (*str*) – The name of this grid. Can be used to search for the parameters from this grid.
- ****constants** – Extra parameters to add to the grid that don't vary. These will not be included in the job_id name.

Just a heads up, there is nothing stopping you from passing an infinite generator, meaning that you can make some fancy sampling grid generators, but `slurmjobs` will take that and not know when to stop. If you want to use an infinite generator, just wrap it in `itertools.islice` which will let you provide a limit.

Obviously, `slurmjobs` doesn't operate anywhere near the memory scale where you'd need to even use generators in the first place, but I figured why limit the implementation if it can be used for other things too.

__len__()

Get the number of iterations in the grid.

Note that any use of generators or functions without a length will cause this to raise a TypeError.

__getitem__(*index*)

Get the series for a variable name.

__iter__()

Yield all combinations from the parameter grid.

class slurmjobs.grid.**LiteralGrid**(*_LiteralGrid__grid*, *name=None*, ***constants*)

A parameter grid, specified as a flattened list. This doesn't do any grid expansion, it lets you specify the grid as you want.

Parameters

- **grid** (*list, dict*) – The parameter list. Should be a list of dicts, each corresponding to a parameter config.
- **name** (*str*) – The name of this grid. Can be used to search for the parameters from this grid.

```
g = LiteralGrid([
    {'a': 1, 'b': 1},
    {'a': 1, 'b': 2},
    {'a': 2, 'b': 2},
])
assert list(g) == [
    {'a': 1, 'b': 1},
    {'a': 1, 'b': 2},
    {'a': 2, 'b': 2},
]
```

__len__()

Get the number of iterations in the grid.

Note that any use of generators or functions without a length will cause this to raise a TypeError.

__iter__()

Yield all combinations from the parameter grid.

2.4.1 Grid Operations

class slurmjobs.grid.**GridCombo**(**grids*, *name=None*)

This handles the multiplication of two grids (combinations). It will create a grid as a product of all provided grids.

You can create this doing `grid_a * grid_b`. The only reason to use this directly is if you want to give it a name or if you want to make a grid product of 3 or more grids.

```
a = Grid(('a', [1, 2]), ('b', [1, 2]))
b = Grid(('c', [1, 2]))

# functionally equivalent
c = a * b
c = GridCombo(a, b, name='my-a-b-combo-grid')
c_items = [
    dict(da, **db)
    for da, db in itertools.product(a, b)
]
```

__len__()

Get the number of iterations in the grid.

Note that any use of generators or functions without a length will cause this to raise a TypeError.

__iter__()

Yield all combinations from the parameter grid.

class slurmjobs.grid.**GridChain**(**grids*, *name=None*)

This handles the addition of two grids (one after the other).

You can create this doing `grid_a + grid_b`. The only reason to use this directly is if you want to give it a name.

```
a = Grid(('a', [1, 2]), ('b', [1, 2]))
b = Grid(('c', [1, 2]))

# functionally equivalent
c = a + b
c = GridChain(a, b, name='my-a-then-b-grid')
c_items = list(a) + list(b)
```

__len__()

Get the number of iterations in the grid.

Note that any use of generators or functions without a length will cause this to raise a TypeError.

__iter__()

Yield all combinations from the parameter grid.

class slurmjobs.grid.**GridOmission**(*grid*, *omission*, *name=None*)

This handles the subtraction of two grids (combinations). It will yield only dicts from `grid_a` that don't appear in `grid_b`.

You can create this doing `grid_a - grid_b`. The only reason to use this directly is if you want to give it a name.

```
a = Grid(('a', [1, 2]), ('b', [1, 2]))
b = Grid(('a', [2]), ('b', [1]))

# functionally equivalent
c = a - b
c = GridOmission(a, b, name='my-a-minus-b-grid')
omit = list(b)
c_items = [da for da in a if da not in omit]
```

__len__()

Get the number of iterations in the grid.

Note that any use of generators or functions without a length will cause this to raise a TypeError.

__iter__()

Yield all combinations from the parameter grid.

2.4.2 Extending Grids

class slurmjobs.grid.**BaseGrid**(*name=None*, *ignore_job_id_keys=None*, ***constants*)

The base class for all grids. Use this if you want to extend another grid.

You just need to implement:

- `__iter__`: This should yield all items generated by the grid
- `__len__`: This should tell you the number of items in the grid. If you cannot determine the length of a grid, then raise a TypeError (as the other grids do).

- `__repr__`: A nice string representation of the grid

__repr__()

A nice string representation of the grid.

__len__()

Get the number of iterations in the grid.

Note that any use of generators or functions without a length will cause this to raise a TypeError.

__iter__()

Yield all combinations from the parameter grid.

__add__(*other*)

Combine two parameter grids sequentially.

__mul__(*other*)

Create a grid as the combination of two grids.

classmethod as_grid(*grid*)

Ensure that a value is a grid.

2.5 Argument Formatting

Argument Formatters

class slurmjobs.args.**Argument**

The base-class for all argument formatters. If you want to provide a new argument formatter, just override this class and change either the:

- `format_arg` method that is used to format positional and key-value pairs, or
- `format_value` (which is called in `format_arg`) to format python values as a string. This is often some form of quoted repr or json formatting.

classmethod get(*key='fire'*, **a*, ***kw*)

Return an instance of an argument formatter, based on its name.

format_arg(*k*, *v=Ellipsis*)

Format a key-value pair for the command-line.

If it is a positional argument, the value will be in key and the value will be `...`.

Parameters

- **k** – The key (or value, for positional arguments).
- **v** – The value. If it's a positional argument, it will be `v=...`

format_value(*v*)

Format a value for the command-line.

class slurmjobs.args.**FireArgument**

Argument formatting for Python Fire.

Example: `python script.py a b --arg1 c --arg2 d`

- Docs: https://google.github.io/python-fire/guide/
- Github: https://github.com/google/python-fire

format_arg(*k*, *v=Ellipsis*)

Format a key-value pair for the command-line.

If it is a positional argument, the value will be in key and the value will be `...`.

Parameters

- **k** – The key (or value, for positional arguments).
- **v** – The value. If it's a positional argument, it will be `v=...`

class slurmjobs.args.**ArgparseArgument**

Argument formatting for Python's builtin argparse.

Example: `python script.py a b --arg1 c --arg2 d`

- Docs: https://docs.python.org/3/library/argparse.html

format_arg(*k*, *v=Ellipsis*)

Format a key-value pair for the command-line.

If it is a positional argument, the value will be in key and the value will be `...`.

Parameters

- **k** – The key (or value, for positional arguments).
- **v** – The value. If it's a positional argument, it will be `v=...`

class slurmjobs.args.**SacredArgument**

Formatting for sacred.

Example: `python script.py with arg1=a arg2=b`

- Docs: https://sacred.readthedocs.io/en/stable/
- Github: https://github.com/IDSIA/sacred

class slurmjobs.args.**HydraArgument**(*default_prefix='+'*, ***kw*)

Formatting for hydra.

Example: `python script.py db.user=root db.pass=1234`

- Docs: https://hydra.cc/docs/intro/
- Github: https://github.com/facebookresearch/hydra

format_arg(*k*, *v=Ellipsis*)

Format a key-value pair for the command-line.

If it is a positional argument, the value will be in key and the value will be `...`.

Parameters

- **k** – The key (or value, for positional arguments).
- **v** – The value. If it's a positional argument, it will be `v=...`

2.6 Function Receipts

class slurmjobs.receipt.**Receipt**(*name=''*, **a*, *receipt_id=None*, *__dir__=None*, ***kw*)

Make a receipt for a function call. This allows you skip over a function if it was successfully ran. This is useful if a long script fails in the middle and you want to re-run it, but you don't need to re-run the first part.

This will cache the function execution history using the string representation of the function's arguments. This works well as 99% of things have some sort of string representation, however, if you have a string representation that doesn't stay consistent, like an object that prints out its ID, then the receipt won't work.

To resolve this, you can either:

- subclass this method and override `Receipt.hash_args(*a, **kw)` to return something invariant for your
- Provide your own `receipt_id` to the function call.
- submit a PR for a better hash function!

```
receipt = Receipt(func.__name__, *a, **kw)

if not receipt.exists():
    try:
        do_something()
        receipt.make()
        # yay we can skip it next time!
    except Exception:
        """Oh well. It failed, but we'll just try again next time."""

else:
    """Oh good it ran completely last time so we can skip it and move to the next␣
↪one."""
```

hash_args(**a*, ***kw*)

Take the function arguments and return a hash string for them.

slurmjobs.receipt.**use_receipt**(*func*, *receipt_dir=None*, *test=None*)

Use a receipt for a function call, which lets us skip a result if the function completed successfully the last run. This is just a wrapper around `Receipt` that handles the receipt checking/making logic for you.

```
# do step 1. If it already ran successfully it will skip and do nothing.
use_receipt(my_step1_function)(**step1_kwargs)

# do step 2. here we're passing a custom receipt ID
custom_receipt_id = ...
use_receipt(my_step2_function)(**step2_kwargs, receipt_id=custom_receipt_id)

# do step 3
use_receipt(my_step3_function)(**step3_kwargs)
```

2.7 Changes

2.7.1 Unreleased

- Rewrote like 70% of the code, including: jinja templates, jobs base class and subclasses
- added a *slurmjobs.Singularity()* class and template
- added parameter grid classes!! Lets you do grid arithmetic :)
- Renamed:
  - `slurmjobs.Batch` => *slurmjobs.Jobs()*
  - `slurmjobs.SlurmBatch` => *slurmjobs.Slurm()*
  - `slurmjobs.Jobs.default_options` => `slurmjobs.Jobs.options`
  - `slurmjobs.Jobs.JOB_ID_KEY` => `slurmjobs.Jobs.job_id_arg`
- Changed signatures:
  - removed parameters from `slurmjobs.Jobs.__init__()`: `multicmd, cmd_wrapper, init_script, run_init_script, post_script, paths`
  - added `Receipt(receipt_id='a-str-used-instead-of-hash_args()')`
- removed utilities:
  - `get_job_name`, `make_job_name_tpl`, `format_value_for_name`, `_encode_special`, `_decode_special`, `_obliterate_special`: replaced by *slurmjobs.Jobs.format_id_item()* and *slurmjobs.Jobs.format_job_id()*
  - `expand_grid`, `expand_paired_params`, `unpack`, `split_cond` replaced by `slurmjobs.Grid()`
  - `singularity_command` replaced by *slurmjobs.Singularity()*
  - `Factory` replaced by `slurmjobs.util.subclass_lookup()`
- `slurmjobs.util.flatten()` now returns a generator rather than a list.
- `slurmjobs.Jobs.template` and `slurmjobs.Jobs.run_template` both expect a template string (not a path). to specify a path, either have a way to read it from file or just extend the template
- simplified argument formatting (removed special namedtuple classes)
  - changed `NoArgVal` to just `...`
  - changed `Argument.build` to just `Argument.__call__`
- improved how command, cli, and args are rendered (all done in jinja now)
- added overridable method `Receipt.hash_args` to allow for custom hashing
- added proper docs
- added tensorflow `2.7` to `cuda_versions.csv`
- added `scripts/singuconda` and `slurmjobs.singuconda` as a WIP rewrite.

2.7.2 0.2.2

- *cmd_wrapper* can also be a function
- added *util.find_tensorflow_cuda_version* to lookup cuda versions
- *util.singularity_command* returns a function that will now escape quotes in the passed command

2.7.3 0.2.1

- Now you can pass in a list of dicts and it will use each dictionary as a job. This works along side the parameter grid expansion so you can do: `` `python jobs.generate([{'param1': 5}, ('param1', [100, 150]), ('param2', [200, 250])]) ` ``
- Added *cmd_wrapper* argument to *SlurmBatch('python myscript.py', cmd_wrapper=slurmjobs.util.singularity_command(overlay, sif))* for easier command formatting (no longer need to use *multicmd=True* and *{__all__}*). It expects a string with one positional format arg, e.g. ('sudo {}')

2.7.4 0.2.0

Oops - TODO - fill in. This was changes to adapt to NYU Greene and Singularity containers.

2.7.5 0.1.7

- added JSON metadata that can be stored in the receipt. Currently, adds *duration_secs* and *time*
- added more receipt logging (on successful write, on skip, on error)
- Set *slurmjobs.Receipt.TEST* instead of *slurmjobs.use_receipt.TEST*

2.7.6 0.1.2

- Added a receipt utility to avoid re-running functions *slurmjobs.use_receipt(func)(*a, **kw)*
- Added *Batch().generate(expand_grid=False)* option to avoid expanding parameters and passing explicit grids
- fixed json encoding error in run templates
-

2.7.7 0.1.1

- **commands can now access the original values (without command line flag attached) by using the variable name preceded v**
  - *'{year}' -> '–year 2016'*
  - *'{_year}' -> '2016'*
- specifying *cli=False* will disable any formatting and will just pass them sequentially.
- weird things were happening when *shlex.quote*-ing *repr* so changed to *json.dumps*

2.7.8 0.1.0

- expanded support to handle multi-line commands.
- added more tests
- moved *init_script* so it happens after activating conda
- added *source ~/.bashrc* to job
- added *run_init_script* so scripts can run code before you submit the jobs
- removed hard coded *nodes* sbatch arg. It is now changeable (not sure why it was hardcoded..)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

F

G

H

J

K

L

M

O

R

S

T

U