Full code for data generation is located at
https://github.com/Kovaleski-Research-Lab/general/_3x3/tree/andy/_branch
Full code for the time series networks is located at
https://github.com/Kovaleski-Research-Lab/meta/_atom/_rnn.git

Here I'll present the code relevant to scaling up to Kubernetes, so I'll be sticking to the meta_atom_rnn repository.

I. All code uses a global config file,
meta_atom_rnn/configs/params.yaml

```
# Purpose: Configuration File

# Create: Expriment Parameter
# - 0 = train network (rnn or lstm – determined by network.arch)
# - 1 = load results (load model checkpoints and loss metrics)
# - 2 = run evaluation (create plots)
# - 3 = preprocess data (from reduced volumes to y-component of 1550
slices plus phases)

experiment: 1
network:
  arch: 0.  # 0 for RNN, 1 for LSTM

# System Parameters

# Commenting these out – Irrelevant to this project


# Visualization Parameters – These are for the evaluation stage. Helps
us organize the way we're loading/evaluation data

visualize:
  all_versions: [0, 1]      #0 for RNN, 1 for LSTM
  # Sequences are the number of image slices the model sees. The
length of this list * 2 is the number of experiments we run – Each
value is put through the RNN and the LSTM.
  sequences : [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60].
```

```
# Kubernetes parameters. pp_job, train_job, load_results_job, and
evaluation_job are used in launch_preprocessing.py,
launch_training.py, launch_load_results.py and launch_evaluation.py,
respectively. Some dictionaries seem incomplete because values are
recycled to avoid repeats.

kube :
  namespace : gpn-mizzou-muem
  image : docker.io/kovaleskilab/meep:v3_lightning
  job_files : /develop/data/kube_jobs  # this is a local directory
  pvc_volumes : dft-volumes # use `kubectl get pvc` to see list of
pvcs
  pvc_preprocessed : preprocessed-data
  pvc_results : training-results

  pp_job:
    num_cpus : 4
    num_mem_lim : 150Gi
    num_mem_req : 100Gi
    kill_tag : preprocess

    paths :

      # interactive pod directories
      data :
        volumes : /develop/results  # points to folder containing
reduced volumes in pvc called dft-volumes
        preprocessed_data : /develop/data/preprocessed_data # points
to folder containing data after it has been preprocessed in pvc called
preprocessed-data
      timing : /develop/data/preprocessed_data/timing # this is where
we dump timing stats for tasks

      # local path where template is located
      template : templates/preprocess_job.txt

  train_job :
    num_cpus : 1
    num_mem_lim : 12Gi
```

```yaml
  num_mem_req : 12Gi
  num_gpus : 1
  kill_tags : [rnn,lstm]
  sequence :
  arch :

  paths :
    data :
      train : /develop/data/preprocessed_data/train
      valid : /develop/data/preprocessed_data/valid
    results :
      # interactive pod directories
      model_results : /develop/results
      model_checkpoints : /develop/results/checkpoints
      analysis : /develop/results/analysis
    logs : /develop/results/checkpoints/current_logs
    # local path where template is located
    template : templates/train_job.txt

load_results_job :
  num_mem_req : 64Gi
  num_mem_lim : 128Gi
  paths :
    template : templates/load_results_job.txt
    params: /develop/code/meta_atom_rnn/configs/params.yaml

evaluation_job :
  paths:
    template : templates/evaluation_job.txt
```

II. For creation of persistent volume claims (PVCs), located at meta_atom_rnn/k8s/storage

```yaml
# This one is for storing the reduced volumes.
# The data reduction pod dumps into this pvc,and gets mounted to develop/results.

# dft_volumes.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dft-volumes
spec:
  storageClassName: rook-cephfs
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1T

# This one is for storing the preprocessed data.
# The data processing pod pulls data from dft-volume and dumps into this pvc. It gets mounted to develop/data/preprocessed_data.

# preprocessed_data.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: preprocessed-data
spec:
  storageClassName: rook-cephfs
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 6G
```

This one holds model checkpoints, plots, and other evaluation items.

```
# The nn-training pod pulls data from preprocessed-data and dumps checkpoints into this pvc at develop/results/checkpoints. The load_results pod and evaluation pod are mounted here to develop/results/analysis.

# training_results.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: training-results
spec:
  storageClassName: rook-cephfs
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 50Gi
```

III. I used interactive pods to observe my data as I moved through the pipeline, located at meta_atom_rnn/k8s/monitor_pods.

For brevity, I’ll just list raw_dataset_mon.yaml:

```
# this is being used as the raw dataset monitor
# run `kubectl apply -f monitor.yaml` to create
# run `kubectl exec -it andy-monitor -- /bin/bash` to enter


apiVersion: v1
kind: Pod
metadata:
  name: monitor-raw-data
spec:
  containers:
```

```yaml
  - name: monitor-raw-data
    image: docker.io/kovaleskilab/meep:v3_lightning
    stdin: True
    tty: True
    resources:
      limits:
        memory: 4Gi
        cpu: 2
      requests:
        memory: 4Gi
        cpu: 2
    volumeMounts:
      - name: meep-dataset-v2
        mountPath: /develop/data
volumes:
  - name: meep-dataset-v2
    persistentVolumeClaim:
      claimName: meep-dataset-v2
```

IV. Jobs are created using
meta_atom_rnn/k8s/launch_eval.py
Meta_atom_rnn/k8s/launch_preprocess.py
Meta_stom_rnn/k8s/launch_training.py
and are supported by meta_atom_rnn/k8s/k8s_support.py

They each have their own templates, located at meta_atom_rnn/k8s/templates and are called evaluation_job.txt, load_results.txt and preprocess_job.txt, train_job.txt. (evaluation_job.txt and load_results.txt are both used by launch_eval.py and controlled by a parameter)

For brevity, I'll just list those associated with training.

```
# train_job.txt. Items in {{ }} are filled in by the
template in launch_training.py
apiVersion: batch/v1
kind: Job
metadata:
  name: {{job_name}}
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: {{job_name}}
          image: {{path_image}}
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8880
          env:
            - name: NCCL_SOCKET_IFNAME
              value: eth0
            - name: NCCL_DEBUG
              value: INFO
          command: ["/bin/sh", "-c"]
          args: ["git clone https://github.com/Kovaleski-
Research-Lab/meta_atom_rnn.git .;
                  echo cloned repo for arch {{arch}}, sequence
{{sequence}};
                  python3 main.py -config configs/params.yaml
-arch {{arch}} -seq_len {{sequence}}"]
          resources:
            limits:
              memory: {{num_mem_lim}}
              cpu: {{num_cpus}}
              nvidia.com/gpu: {{num_gpus}}
            requests:
              memory: {{num_mem_req}}
              cpu: {{num_cpus}}
              nvidia.com/gpu: {{num_gpus}}
          volumeMounts:
            - name: {{pvc_preprocessed}}
```

```
              mountPath: {{pp_data_path}}
            - name: {{pvc_results}}
              mountPath: {{results_path}}
            - name: shm
              mountPath: /dev/shm

      volumes:
        - name: {{pvc_preprocessed}}
          persistentVolumeClaim:
            claimName: {{pvc_preprocessed}}
        - name: {{pvc_results}}
          persistentVolumeClaim:
            claimName: {{pvc_results}}
        - name: shm
          emptyDir:
            medium: Memory
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: nvidia.com/gpu.product
                    operator: In
                    values:
                      - NVIDIA-GeForce-RTX-3090
```

```
# launch_training.py
# This is the script that creates job templates using the
jinja2 library. It loops through each experiment (sequence
length) and creates a job file which launches a pod for each
GPU job. For experiments 5, 10, 15…60, one RNN model is
launched and one LSTM model is launched, resulting in 24 GPU
jobs. This script also has the option to kill jobs during
debugging or once they are complete. Helper functions are
supplied by k8s_support.py which is provided next.

# This script gets called by main.py, which is called from
within a container.
```

```
# Import Python functions
import os
import sys
import yaml
import time
import subprocess
from IPython import embed

# Import functions specific to Kubernetes tasks
from dateutil.tz import tzutc
from kubernetes import client, config
from jinja2 import Environment, FileSystemLoader

# Import custom functions
from k8s_support import exit_handler, load_file, save_file,
parse_args, load_config

sys.path.append("../")
from utils.general import create_folder

def run(params):

   # Create template for Kubernetes job
    template =
load_file(params['kube']['train_job']['paths']['template'])

   # This just gives us an identifier for convenience
    tag =
params['kube']['train_job']['paths']['template'].split("/")[-
1]

   # The folder where our template (job.txt) is located
    folder =
params['kube']['train_job']['paths']['template'].replace("/%s"
% tag, "")

   # load job template into jinja environment
    environment = Environment(loader =
FileSystemLoader(folder))
```

```
   # create template
template = environment.get_template(tag)

   # Need a local folder to put the job file into
    create_folder(params['kube']['job_files'])

   # this is the list of models we'll train according to
sequence length
    sequences = params['visualize']['sequences']

   # Network architectures: 0 for RNN, 1 for LSTM.
    arches = [0, 1]

   # all sequences will be used to train both RNN and LSTM
    for sequence in sequences:

        params['dataset']['seq_len'] = sequence

        # both architectures will be trained on each sequence
length
        for arch in arches:

            params['network']['arch'] = arch

            arch_str = 'rnn' if arch == 0 else 'lstm'

        # assign job name
            job_name = "%s-%s" % (arch_str, str(sequence))

        # fill in template info from global params
            template_info = {'job_name': job_name,
                             'num_cpus':
str(params['kube']['train_job']['num_cpus']),
                             'num_gpus':
str(params['kube']['train_job']['num_gpus']),
                             'num_mem_req':
str(params['kube']['train_job']['num_mem_req']),
                             'num_mem_lim':
str(params['kube']['train_job']['num_mem_lim']),
```

```
                              'pvc_preprocessed':
params['kube']['pvc_preprocessed'],
                              'pp_data_path':
params['kube']['pp_job']['paths']['data']['preprocessed_data']
,
                              'pvc_results':
params['kube']['pvc_results'],
                              'results_path':
params['kube']['train_job']['paths']['results']['model_results
'],
                              'ckpt_path':
params['kube']['train_job']['paths']['results']['model_checkpo
ints'],
                              'path_image':
params['kube']['image'],
                              #'path_logs':
params['kube']['path_logs'],
                              'sequence': sequence,
                              'arch': arch,
                             }

        # fill in the template with new params
            filled_template = template.render(template_info)

            path_job =
os.path.join(params['kube']['job_files'], job_name.zfill(2) +
".yaml")

            save_file(path_job, filled_template)

        # use kubectl command to create job. Once this is
done, pods will be created and can be observed with `kubectl
get pods`
            subprocess.run(['kubectl', 'apply', '-f',
path_job])
            print(f"launching job for {arch}, {sequence}")
```

```
if __name__=="__main__":

   # If `kill` is True, we’ll go through and find all the open
jobs and kill them. Otherwise, we’ll run the run() function.
    kill = False
    #kill = True

    args = parse_args(sys.argv)

    params = load_config(args['config'])

    if kill == False:

        run(params)

    elif kill == True:
        kill_tags = params['kube']['train_job']['kill_tags']
        for kill_tag in kill_tags:
            exit_handler(params,kill_tag)
```

k8s_support.py

```
# Some helper functions to assist with launch_{task}.py


# import Python functions
import os
import sys
import yaml
import subprocess

# import Kubernetes functions
from kubernetes import client, config
```

```
# This function goes through and finds all our jobs and kills
them.
def exit_handler(params,kill_tag):

    config.load_kube_config()   # python can see the kube
config now. Lets us run API commands.

    v1 = client.CoreV1Api()   # initializing a tool to do kube
stuff.

    pod_list = v1.list_namespaced_pod(namespace =
params["kube"]["namespace"])    # get all pods currently
running

    current_group = [ele.metadata.owner_references[0].name for
ele in pod_list.items if(kill_tag in ele.metadata.name)]    #
getting the names of the pods

    current_group = list(set(current_group))    # remove any
duplicates

    for job_name in current_group:
        subprocess.run(["kubectl", "delete", "job", job_name])
# delete the kube job (a.k.a. pod)

    print("\nCleaned up any jobs that include tag : %s\n" %
kill_tag)

def load_file(path):

    data_file = open(path, "r")

    info = ""

    for line in data_file:
        info += line

    data_file.close()
```

```
    return info

def save_file(path, data):

    data_file = open(path, "w")

    data_file.write(data)

    data_file.close()

# custom argparser
def parse_args(all_args, tags = ["--", "-"]):

    all_args = all_args[1:]

    if(len(all_args) % 2 != 0):
        print("Argument '%s' not defined" % all_args[-1])
        exit()

    results = {}

    i = 0
    while(i < len(all_args) - 1):
        arg = all_args[i].lower()
        for current_tag in tags:
            if(current_tag in arg):
                arg = arg.replace(current_tag, "")
        results[arg] = all_args[i + 1]
        i += 2

    return results

def load_config(argument):

    try:
        return yaml.load(open(argument), Loader =
yaml.FullLoader)

    except Exception as e:
```

```
        print("\nError: Loading YAML Configuration File")
        print("\nSuggestion: Using YAML file? Check File For
Errors\n")
        print(e)
        exit()
```

All other launch_{task}.py methods follow the same templating logic but do not require a for loop because they only launch one job.