Will Spaeth
2-23-20

Homework 3

```
##################
# supercomputer.py #
##################

#!/usr/bin/env python3

from collections import deque
import pickle
import os
import sys
import json
import subprocess
import itertools

import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.callbacks import EarlyStopping

from models import dnn
from symbiotic_metrics import FractionOfVarianceAccountedFor

def main():
    """Spits out training jobs for each configuration"""

    # Create error file
    with open("error/err.txt", "w") as f:
        pass

    # Create option dictionary
    options = {
        "rotation": list(range(20)),
        "n_train_folds": [1, 2, 3, 5, 10, 18],
        "dropout": [0, .3, .6, .8]
        #"l2": [0, .001, .01, .1]
    }

    option_combinations = create_combinations(options)

    option_combinations = create_index_log(options, option_combinations)
```

```
    # Start a job for each hyperparameter
    for option_combo in option_combinations:
        start_training_job(**option_combo)

def create_combinations(option_dictionaries):
    '''
    Used to create a list of dictionaries containing all possible combinations
        of input dictionary arguments
    Found on tutorial website: https://riptutorial.com/python/example/10160/all-combinations-
of-dictionary-values
    '''
    keys = option_dictionaries.keys()
    values = (option_dictionaries[key] for key in keys)
    combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]


    for i in range(len(combinations)):
        combinations[i]["experiment_num"] = i

    return combinations

def create_index_log(options, option_combinations):
    """Write index to file that describes experiment hyperparameters"""

    fbase = "results/"
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    batch_num = 0
    while ( os.path.exists("{}batch_{}/".format(fbase, batch_num)) ):
        batch_num += 1

    fbase = "{}batch_{}/".format(fbase, batch_num)
    os.mkdir(fbase)

    with open('{}index.txt'.format(fbase), 'w') as f:
        f.write("Number of experiments: {}\n".format(len(option_combinations)))
        json.dump(options, f)
        f.write("\n")

        for i in range(len(option_combinations)):
            option_combinations[i]["batch_num"] = batch_num
            json.dump(option_combinations[i], f)
            f.write("\n")
```

```
    return option_combinations

def start_training_job(**kwargs):
    """
    Starts a job for the fed arguments. This takes the form of a subprocess,
    whether on a normal computer or supercomputer
    """

    print("Starting job:\n\t{}".format(kwargs))

    # Decide which script to run
    if "-s" in sys.argv:
        script_to_run = ["sbatch", "supercomputer_job.sh", "-s"]
    else:
        script_to_run = ["./standard_job.sh"]

    # Build script with hyperparameters
    full_command = [
        *script_to_run,
        "-job"
    ]
    for key, value in kwargs.items():
        full_command.append("--{}={}".format(key, value))

    # Run chosen script with correct arguments
    process = subprocess.Popen(full_command)

    # Wait if not parallel
    if "-p" not in sys.argv:
        process.wait()

def parse_args():

    # Parse the hyperparameter arguments
    kwargs = {}
    for arg in sys.argv:
        if "--" in arg:
            arg = arg.replace("--", "")

            key, value = arg.split("=")
            kwargs[key] = value

    return kwargs
```

```
def train(**kwargs):

    print("PARAMETERS: {}".format(kwargs))

    # Unpack relevant kwargs
    rotation = int(kwargs["rotation"])
    n_train_folds = int(kwargs["n_train_folds"])
    experiment_num = int(kwargs["experiment_num"])
    batch_num = int(kwargs["batch_num"])

    if "dropout" in kwargs.keys():
        dropout = float(kwargs["dropout"])
    else:
        dropout = 0

    if "l2" in kwargs.keys():
        l2 = float(kwargs["l2"])
    else:
        l2 = 0

    # Rotate indices based on current rotation
    rotation_indices = get_rotation_indices(n_folds=20, rotation=rotation)

    # Get the training, validation, and test fold indices
    fold_inds = get_set_indices(rotation_indices=rotation_indices, n_train_folds=n_train_folds)

    ''' Load data
    Key MI, Length 20, Shape (1193, 960)
    Key theta, Length 20, Shape (1193, 2)
    Key dtheta, Length 20, Shape (1193, 2)
    Key ddtheta, Length 20, Shape (1193, 2)
    Key torque, Length 20, Shape (1193, 2)
    Key time, Length 20, Shape (1193, 1)
    '''
    if "-s" in sys.argv:
        data_path = "/home/fagg/ml_datasets/bmi/bmi_dataset.pkl"
    else:
        data_path = "../homework_2/bmi_dataset.pkl"
    with open(data_path, "rb") as fp:
        hw2_dataset = pickle.load(fp)

    # Splits the data into its respective train, validation, and test sets / ins and outs
    processed_data = process_dataset(hw2_dataset, fold_inds)
```

```
    # Build model
    model = dnn(
        input_size=(processed_data["train"]["ins"].shape[1],),
        hidden_sizes=[300, 150, 100, 50, 10],
        output_size=processed_data["train"]["outs"].shape[1],
        hidden_act="elu",
        output_act="linear",
        dropout=dropout,
        l2=l2)

    # Compile model with fvaf metric
    fvaf = FractionOfVarianceAccountedFor(processed_data["test"]["outs"].shape[1])
    model.compile(optimizer="adam", loss="mse", metrics=[fvaf], verbose=2)
    model.summary()

    # Callbacks
    es_callback = EarlyStopping(
                        monitor="val_loss",
                        patience=20,
                        restore_best_weights=True,
                        min_delta=.0001)

    # Train model
    history = model.fit(
            x=processed_data["train"]["ins"],
            y=processed_data["train"]["outs"],
            validation_data = (processed_data["val"]["ins"], processed_data["val"]["outs"]),
            epochs=10000,
            batch_size=32,
            callbacks=[es_callback]
            )

    # Log results
    log(model, processed_data, kwargs)

    # Plot the torque and save figure
    if experiment_num == 0:
        plot_shoulder_orientation(model, processed_data, kwargs)

def plot_shoulder_orientation(model, data, kwarg_dict):
    """Plots the torque graph"""

    # Create results directory
```

```
    save_path = "results/"
    if not os.path.exists(save_path):
        os.mkdir(save_path)

    save_path += "batch_{}/".format(kwarg_dict["batch_num"])
    if not os.path.exists(save_path):
        os.mkdir(save_path)

    # Create specific experiment directory
    save_path += "experiment_{}/".format(kwarg_dict["experiment_num"])
    if not os.path.exists(save_path):
        os.mkdir(save_path)

    true_orientation = data["test"]["outs"][:, 0]
    predicted_orientation = model.predict(data["test"]["ins"])[:, 0]

    # Create and configure plot
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.plot(data["test"]["time"], true_orientation, label="True Orientation")
    ax.plot(data["test"]["time"], predicted_orientation, label="Predicted Orientation")
    ax.legend()
    plt.ylabel("Orientation")
    plt.xlabel("Time")

    # Save plot
    fig.savefig(save_path + f"orientation_plot.png", dpi=fig.dpi)

def log(model, data, kwarg_dict):
    """Log results to file"""

    print("Logging results")

    # Generate results
    results = {}
    results['predict_train'] = model.predict(data["train"]["ins"])
    results['eval_train'] = model.evaluate(data["train"]["ins"], data["train"]["outs"])
    results['predict_val'] = model.predict(data["val"]["ins"])
    results['eval_val'] = model.evaluate(data["val"]["ins"], data["val"]["outs"])
    results['predict_test'] = model.predict(data["test"]["ins"])
    results['eval_test'] = model.evaluate(data["test"]["ins"], data["test"]["outs"])

    for key, value in kwarg_dict.items():
        results[key] = value
```

```
    # Create results directory
    fbase = "results/"
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    fbase += "batch_{}/".format(kwarg_dict["batch_num"])
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    fbase += "experiment_{}/".format(kwarg_dict["experiment_num"])
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    # Save results
    with open("{}results_dict.pkl".format(fbase), "wb") as fp:
        pickle.dump(results, fp)
        fp.close()

    # Create model directory
    if not os.path.exists("{}/model/".format(fbase)):
        os.mkdir("{}/model/".format(fbase))

    # Save model
    model.save("{}/model/".format(fbase))

def process_dataset(dataset, fold_inds):
    """
    Process the dataset into the train, validation, and test folds;
    Also split into ins & out sets
    """

    processed_data = {}
    for key in fold_inds.keys():
        processed_data[key] = split_dataset(dataset, fold_inds[key])

    return processed_data

def split_dataset(dataset, inds):

    # Placeholder for data splits
    processed_data = {
        "ins": None,
        "outs": [],
```

```
        "time": None
    }

    for key in dataset.keys():
        # Get folds for this key
        folds = [dataset[key][ind] for ind in inds]

        # Join the folds
        joined = np.concatenate((folds), axis=0)

        # See if the key is for the ins or outs of the dataset
        if key == "MI":
            processed_data["ins"] = joined
        elif key == "time":
            processed_data["time"] = joined
        elif key == "theta":
            processed_data["outs"] = np.expand_dims(joined[:, 0], axis=1)

    return processed_data

def get_set_indices(rotation_indices, n_train_folds):
    """Get the fold indices for each set"""

    inds = {}
    inds["train"] = [rotation_indices[i] for i in range(n_train_folds)]
    inds["val"] = [rotation_indices[len(rotation_indices)-2]]
    inds["test"] = [rotation_indices[len(rotation_indices)-1]]

    return inds

def get_rotation_indices(n_folds, rotation=0):
    """Rotate folds to get the right indices"""

    fold_list = list(range(n_folds))
    fold_list = deque(fold_list)
    fold_list.rotate(rotation)
    fold_list = list(fold_list)

    return fold_list

if __name__ == "__main__":

    # If this is a subprocess, run the training program
    if "-job" in sys.argv:
```

```
    kwargs = parse_args()

    try:
        train(**kwargs)

    # If any exception occurs, write to error folder to differentiate between all the job outputs
    except Exception as e:
        fbase = "error/"
        if not os.path.exists(fbase):
            os.mkdir(fbase)

        with open("{}err.txt".format(fbase), "a") as f:
            err_str = "Error: {}\n".format(e)
            f.write(err_str)

else:
    main()
```

```
#########
# local.py #
#########

#!/usr/bin/env python3

import sys
import json
import pickle
import os
import subprocess

import matplotlib.pyplot as plt
import numpy as np

def main():

   # Use -s argument to scp results from supercomputer before continuing
   if "-s" in sys.argv:
      script_to_run = [
         "scp",
         "-r",
         "jwspaeth@schooner.oscer.ou.edu:/home/jwspaeth/workspaces/advanced-
ml/homework_3/results",
         "./"]
      process = subprocess.Popen([*script_to_run])
      process.wait()

   # Load dropout experiments and results
   dropout_options, dropout_experiments = load_index_log(batch_num=0)
   dropout_results = [load_result_from_experiment(experiment) for experiment in
dropout_experiments]

   # Load l2 experiments and results
   l2_options, l2_experiments = load_index_log(batch_num=1)
   l2_results = [load_result_from_experiment(experiment) for experiment in l2_experiments]

   # Compute fvaf curves for each key value
   #   (training folds x hyperparameter value), so average across rotations
   dropout_avg_fvafs = compute_avg_fvaf_curves(dropout_results,
                                    dropout_options["n_train_folds"],
                                    dropout_options["dropout"],
                                    "dropout",
                                    dropout_options["rotation"])
```

```
    l2_avg_fvafs = compute_avg_fvaf_curves(l2_results,
                                l2_options["n_train_folds"],
                                l2_options["l2"],
                                "l2",
                                l2_options["rotation"])

    # Plot fvaf curves for each key value
    plot_fvaf_curves(dropout_options["n_train_folds"],
                dropout_options["dropout"],
                dropout_avg_fvafs["val"],
                "Validation Dropout")
    plot_fvaf_curves(l2_options["n_train_folds"],
                l2_options["l2"],
                l2_avg_fvafs["val"],
                "Validation L2")

    # Get argmax for the best hyperparameter values
    dropout_argmax_fvafs = np.argmax(dropout_avg_fvafs["val"], axis=1)
    l2_argmax_fvafs = np.argmax(l2_avg_fvafs["val"], axis=1)

    # Build arrays containing test values for best validation models
    test_dropout = []
    test_l2 = []
    for i in range(dropout_argmax_fvafs.shape[0]):
        test_dropout.append(dropout_avg_fvafs["test"][i, dropout_argmax_fvafs[i]])
        test_l2.append(l2_avg_fvafs["test"][i, l2_argmax_fvafs[i]])
    test_dropout = np.expand_dims(np.concatenate(test_dropout), axis=1)
    test_l2 = np.expand_dims(np.concatenate(test_l2), axis=1)
    test = np.concatenate((test_dropout, test_l2), axis=1)

    # Plot the test set fvaf for the argmaxes
    plot_fvaf_curves(dropout_options["n_train_folds"],
                ["Dropout", "L2"],
                test,
                "Test Curves")

def load_result_from_experiment(experiment):
    """Load result of given experiment"""

    file_str =
"results/batch_{}/experiment_{}/results_dict.pkl".format(experiment["batch_num"],
experiment["experiment_num"])
    with open(file_str, "rb") as fp:
        return pickle.load(fp)
```

```
def load_index_log(batch_num):
    """Load the index log of the batch"""

    experiments = []
    with open("results/batch_{}/index.txt".format(batch_num), "r") as f:
        contents = f.read().split("\n")

        options = json.loads(contents[1])
        for i, experiment_str in enumerate(contents[2:len(contents)-1]):
            experiments.append(json.loads(experiment_str))

        return options, experiments

def plot_fvaf_curves(n_train_folds_list, key_list, curves, plot_name):
    """Plot fvaf based on the given curves"""

    # Create results directory
    save_path = "results/"
    if not os.path.exists(save_path):
        os.mkdir(save_path)

    # Create plots directory
    save_path += "fvaf_plots/"
    if not os.path.exists(save_path):
        os.mkdir(save_path)

    # Create and configure plot
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    for i in range(curves.shape[1]):
        ax.plot(n_train_folds_list, curves[:, i], label=str(key_list[i]))
    plt.legend()
    plt.ylabel("Average FVAF")
    plt.xlabel("Number of Training Folds")

    plt.title(plot_name)

    # Save
    fig.savefig("{}{}_fvaf_plot.png".format(save_path, plot_name), dpi=fig.dpi)

def get_matching_result(results, n_train_fold, key_val, key_name, rotation_val):
    """Find the result that matches the given values"""
```

```
    for i, result in enumerate(results):

        if (int(result["n_train_folds"]) == n_train_fold) and (float(result[key_name]) == key_val) and
(int(result["rotation"]) == rotation_val):
            return result

def compute_avg_fvaf_curves(results, n_train_fold_list, key_list, key_name, rotation_list):
    """Gets the average fvaf across rotations for both the validation and test sets"""

    # Create 3d array of results
    fvaf_curves = {
        "val": np.zeros(shape=(len(n_train_fold_list), len(key_list), len(rotation_list)),
dtype=object),
        "test": np.zeros(shape=(len(n_train_fold_list), len(key_list), len(rotation_list)),
dtype=object)
    }
    for i, n_train_fold in enumerate(n_train_fold_list):
        for j, key_val in enumerate(key_list):
            for k, rotation_val in enumerate(rotation_list):

                # Get result with the matching parameters
                result = get_matching_result(results, n_train_fold, key_val, key_name, rotation_val)

                # Store the validation or test fvaf
                fvaf_curves["val"][i, j, k] = result["eval_val"][1]
                fvaf_curves["test"][i, j, k] = result["eval_test"][1]

    # Average across rotations
    fvaf_curves["val"] = np.average(fvaf_curves["val"], axis=2)
    fvaf_curves["test"] = np.average(fvaf_curves["test"], axis=2)

    return fvaf_curves

if __name__ == "__main__":
    main()
```

```
############
# models.py #
############

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras import regularizers

def pipe_model(inputs, layers):
    """Pipes an input through a model to obtain the output hook"""

    for i in range(len(layers)):
        if i == 0:
            carry_out = layers[i](inputs)
        else:
            carry_out = layers[i](carry_out)

    return carry_out

def dnn(input_size, hidden_sizes, output_size, hidden_act="sigmoid", output_act="tanh",
dropout=0, l2=0):
    """Construct a simple deep neural network"""

    layers = []

    # Dropout layer if applicable
    if dropout > 0:
        layers.append(Dropout(rate=dropout))

    # Add hidden layers with respective dropout and l2 values
    layers.append(
        hidden_stack(hidden_sizes, hidden_act, dropout=dropout, l2=l2)
        )

    # l2 regularization if applicable
    if l2 > 0:
        layers.append(
            Dense(
                output_size,
                activation=output_act,
                kernel_regularizer=regularizers.l2(l2)
                )
            )
    else:
```

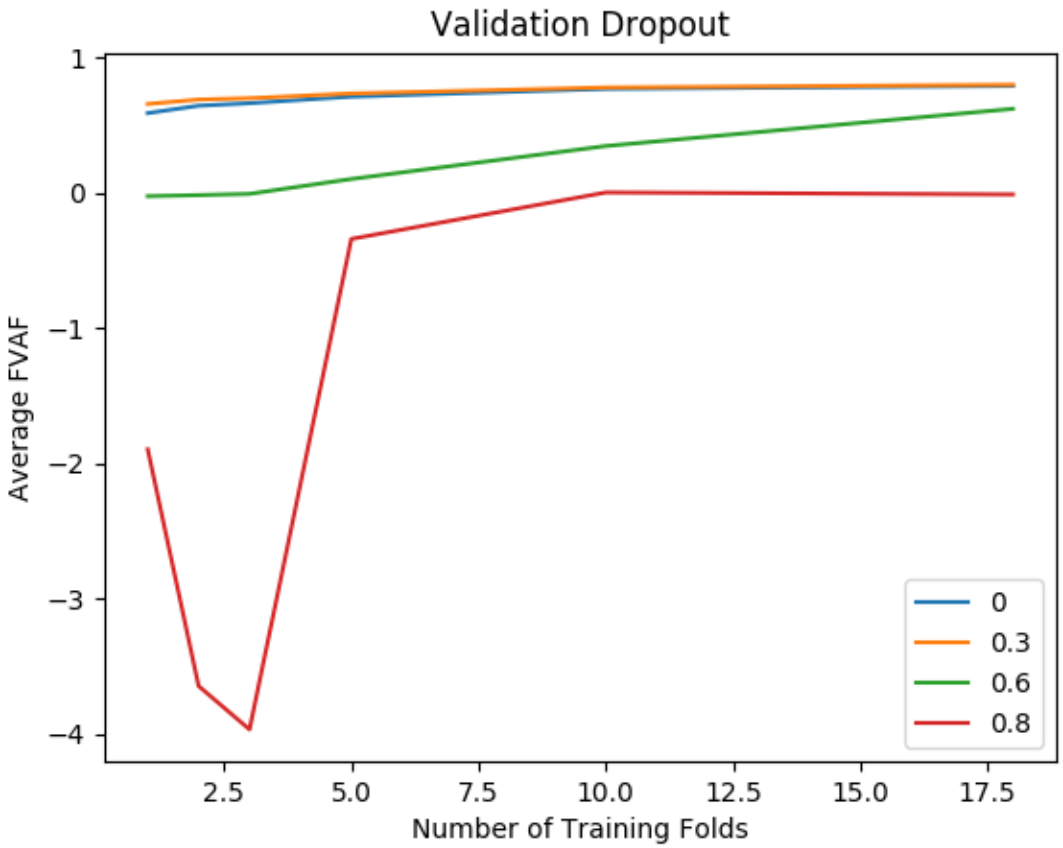
```
        layers.append(
            Dense(
                output_size,
                activation=output_act
                )
            )

    # Pipe model by feeding through input placeholder
    inputs = Input(shape=input_size)
    outputs = pipe_model(inputs, layers)

    return Model(inputs=inputs, outputs=outputs)

def hidden_stack(hidden_sizes, hidden_act="sigmoid", dropout=0, l2=0):
    """Represents a stack of neural layers"""

    layers = []
    for size in hidden_sizes:

        # Apply l2 if applicable
        if l2 > 0:
            layers.append(Dense(
                    size,
                    activation=hidden_act,
                    kernel_regularizer=regularizers.l2(l2)
                    )
                )
        else:
            layers.append(Dense(
                    size,
                    activation=hidden_act,
                    )
                )

        # Apply dropout if applicable
        if dropout > 0:
            layers.append(Dropout(rate=dropout))

    def hidden_stack_layer(inputs):
        """Layer hook for stack"""

        for i in range(len(layers)):
            if i == 0:
                carry_out = layers[i](inputs)
```

```
        else:
            carry_out = layers[i](carry_out)

    return carry_out

return hidden_stack_layer
```

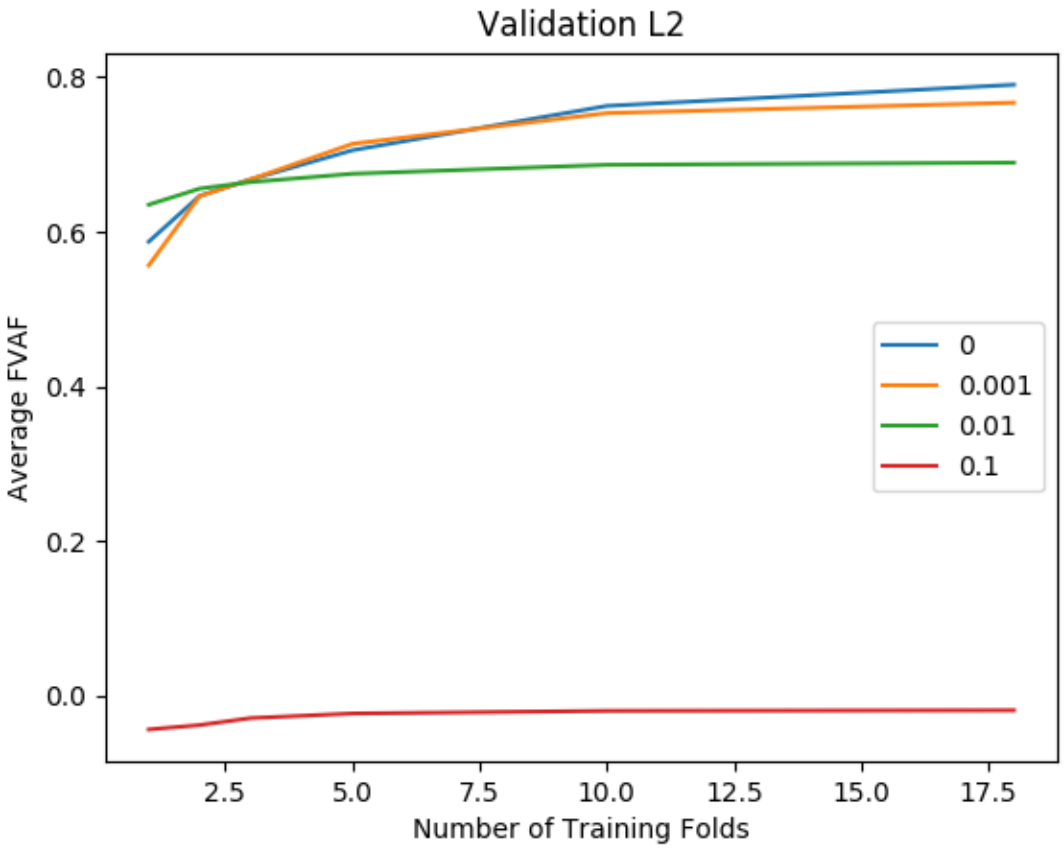
```
######################
# supercomputer_job.sh #
######################

#!/bin/bash

#SBATCH --partition=normal
#SBATCH --ntasks=1
#SBATCH --mem=2000
#SBATCH --output=job-output/subprocess-%j-stdout.txt
#SBATCH --error=job-output/subprocess--%j-stderr.txt
#SBATCH --time=7:00:00
#SBATCH --job-name=subprocess-%j
#SBATCH --mail-user=john.w.spaeth-1@ou.edu
#SBATCH --mail-type=ALL
#SBATCH --chdir=/home/jwspaeth/workspaces/advanced-ml/homework_3/
#SBATCH --wait

python3 supercomputer.py $@
```

```
#################
# standard_job.sh #
#################

#!/bin/bash

python3 supercomputer.py $@
```

##########################
Dropout Validation Curves
##########################



####################
L2 Validation Curves
####################



############################
Dropout and L2 Test Curves
############################