```
import sys
import glob

import imageio
import numpy as np

class Core50Dataset:
    """Handler for Core50 dataset used in this class"""

    # Data path
    if "-s" in sys.argv:
        data_path = "/home/fagg/ml_datasets/core50/core50_128x128/"
    else:
        data_path = "/Users/willspaeth/datasets/core50/core50_128x128/"

    # Declare the various partitions
    s_inds = [1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
    o_scissors = [11, 12, 13, 14, 15]
    o_mugs = [41, 42, 43, 44, 45]
    o_train = [2, 3, 4, 5]
    o_val = [1]

    def __init__(self, exp_cfg=None):
        pass

    def load_data(self):
        """Load data dictionary for training"""

        # Compile data dictionary
        data_dict = {
            "train": {"ins": [], "outs": []},
            "val": {"ins": [], "outs": []}
        }

        # Loop through images and place into appropriate partitions
        image_count = 0
        for s in self.s_inds:
            for o in self.o_scissors+self.o_mugs:
                images = self.load_images(s, o)
                image_count += len(images)

                # Scissors are positive, mugs are negative
                if o in self.o_scissors:
                    label_list = [[1, 0] for i in range(len(images))]
                else:
                    label_list = [[0, 1] for i in range(len(images))]

                # Check if the o is in the train split or val split
                if (o%10) in self.o_train:
                    data_dict["train"]["ins"] = data_dict["train"]["ins"] + images
                    data_dict["train"]["outs"] = data_dict["train"]["outs"] + label_list
                else:
                    data_dict["val"]["ins"] = data_dict["val"]["ins"] + images
                    data_dict["val"]["outs"] = data_dict["val"]["outs"] + label_list

        print("Final image count: {}".format(image_count))

        # Convert data dictionary entries into numpy arrays instead of lists
        data_dict["train"]["ins"] = np.stack(data_dict["train"]["ins"], axis=0)
        data_dict["train"]["outs"] = np.stack(data_dict["train"]["outs"], axis=0)
        data_dict["val"]["ins"] = np.stack(data_dict["val"]["ins"], axis=0)
        data_dict["val"]["outs"] = np.stack(data_dict["val"]["outs"], axis=0)

        return data_dict

    def load_images(self, s, o):
        """Load multiple images based on their s and o conditions"""
        images = []
```

```
        filenames = self._get_matching_filenames(s, o)
        for filename in filenames:
            images.append(imageio.imread(filename).astype(float))


        return images

    def get_input_size(self):
        """Gets the dimensionality of one sample"""

        return (128, 128, 3)

    def _get_matching_filenames(self, s, o):
        """Get matching filenames based on s and o conditions"""

        file_pattern = "{}s{}/o{}/C_{:02d}_{}_*0.png".format(self.data_path, s, o, s, o)
        return glob.glob(file_pattern)
```

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Flatten, BatchNormalization

from custom_layers import pipe_model, hidden_stack, conv_stack_2d

def cnn(input_size, exp_cfg):
    """Construct a simple convolutional neural network"""

    layers = []

    # Input batch normalization
    layers.append(
            BatchNormalization(axis=exp_cfg.model.input_axis_norm)
        )

    # Add conv layers with respective l2 values
    layers.append(
        conv_stack_2d(
                filters=exp_cfg.model.conv.filters,
                kernels=exp_cfg.model.conv.kernels,
                strides=exp_cfg.model.conv.strides,
                max_pool_sizes=exp_cfg.model.conv.max_pool_sizes,
                batch_norms=exp_cfg.model.conv.batch_norms,
                l2=exp_cfg.model.conv.l2
            )
        )

    # Flatten for dnn
    layers.append(
        Flatten()
        )

    # Add dnn
    layers.append(
        hidden_stack(
                hidden_sizes=exp_cfg.model.dense.hidden_sizes,
                batch_norms=exp_cfg.model.dense.batch_norms,
                dropout=exp_cfg.model.dense.dropout
            )
        )

    # Add output layer
    layers.append(
        Dense(
            exp_cfg.model.output.output_size,
            activation=exp_cfg.model.output.activation
            )
        )

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

    return Model(inputs=inputs, outputs=outputs)
```

```
from tensorflow.keras.layers import Dense, Dropout, Conv2D, MaxPool2D, BatchNormalization
from tensorflow.keras import regularizers

from .util import pipe_model

def conv_stack_2d(filters, kernels, strides, max_pool_sizes, batch_norms=0, padding="same
", activation="elu", l2=0):
    """Represents a stack of convolutional layers"""

    # If padding is one word or default, extend into a uniform list
    if type(batch_norms) != list:
        batch_norms = [batch_norms]*len(filters)

    # If padding is one word or default, extend into a uniform list
    if type(padding) != list:
        padding = [padding]*len(filters)

    # If activation is one word or default, extend into a uniform list
    if type(activation) != list:
        activation = [activation]*len(filters)

    # Add convolutions
    layers = []
    for i in range(len(filters)):

        if l2 > 0:
            layers.append(Conv2D(
                            filters=filters[i],
                            kernel_size=kernels[i],
                            strides=strides[i],
                            padding=padding[i],
                            activation=activation[i],
                            kernel_regularizer=regularizers.l2(l2)
                ))
        else:
            layers.append(Conv2D(
                            filters=filters[i],
                            kernel_size=kernels[i],
                            strides=strides[i],
                            padding=padding[i],
                            activation=activation[i]
                ))

        if batch_norms[i] == 1:
            layers.append(BatchNormalization(axis=3))

        layers.append(MaxPool2D(pool_size=max_pool_sizes[i]))

    def conv_stack_2d_layer(inputs):
        """Layer hook for stack"""

        return pipe_model(inputs, layers)

    return conv_stack_2d_layer
```

```
from yacs.config import CfgNode as CN

# Construct root
_D = CN()

# Training or evaluation
_D.mode = "train"

# Save parameters
_D.save = CN()
_D.save.experiment_batch_name = "deep_test_1"

# Dataset parameters
_D.dataset = CN()
_D.dataset.name = "Core50Dataset"

# Model parameters
_D.model = CN()
_D.model.name = "cnn"
_D.model.input_axis_norm = 3
_D.model.conv = CN()
_D.model.conv.filters = [5, 10, 15]
_D.model.conv.kernels = [2, 2, 2]
_D.model.conv.strides = [1, 1, 1]
_D.model.conv.l2 = 0
_D.model.conv.max_pool_sizes = [4, 4, 4]
_D.model.conv.batch_norms = [1, 1, 1]
_D.model.dense = CN()
_D.model.dense.hidden_sizes = [5, 5]
_D.model.dense.dropout = .5
_D.model.dense.batch_norms = [1, 1]
_D.model.output = CN()
_D.model.output.output_size = 2
_D.model.output.activation = "softmax"
_D.model.reload_path = ""

# Training parameters
_D.train = CN()
_D.train.learning_rate = .001
_D.train.epochs = 200
_D.train.batch_size = 32
_D.train.loss = "mse"
_D.train.metrics = ["acc"]
_D.train.verbose = 2

# Callback parameters
_D.callbacks = CN()
_D.callbacks.names = ["EarlyStopping", "FileMetricLogger"]
_D.callbacks.EarlyStopping = CN()
_D.callbacks.EarlyStopping.patience = 50
_D.callbacks.EarlyStopping.min_delta = .0001

# Evaluation parameters
_D.evaluate = CN()

# Misc parameters
_D.misc = CN()
_D.misc.default_duplicate = 1 # Duplicates experiments by this amount. Only activates if
all options are empty

# Construct list of configuration keys and their possible options
# â\200¢ If the key is in the list, the default is overwritten, unless its corresponding
value list is empty
all_options_dict = {}
```

```
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras import regularizers

from .util import pipe_model

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

    if type(batch_norms) != list:
        batch_norms = [batch_norms]*len(hidden_sizes)

    # Add dense layers
    layers = []
    for i, size in enumerate(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,
                    )
                )

        if batch_norms[i] == 1:
            layers.append(BatchNormalization(axis=1))

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

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

        return pipe_model(inputs, layers)

    return hidden_stack_layer
```

```
#!/usr/bin/env python3

"""
Aggregates results in a batch of experiments
"""

import sys
import json
import pickle
import os
import glob

import matplotlib.pyplot as plt
import numpy as np
import sklearn.metrics

from datasets import Core50Dataset

def main():

    # Declare experiment batch
    batch_name = "shallow_test_1"
    fbase = "results/{}/".format(batch_name)

    # Load experiment batch
    results = load_batch_results(fbase)

    # Create list of validation accuracy curves
    argmin_losses = [( np.amin(result["history"]["val_loss"]), np.argmin(result["history"
]["val_loss"]) ) for i, result in enumerate(results)]
    acc_curves = [result["history"]["val_acc"] for i, result in enumerate(results)]

    # Plot learning curves
    plot_learning_curves(fbase, argmin_losses, acc_curves, "Validation")

    # Create list of ROC curves
    all_predictions = [result["predict_val"] for result in results]
    outs = Core50Dataset().load_data()["val"]["outs"]
    roc_curves = [generate_roc_curve(outs, predictions) for predictions in all_prediction
s]

    # Plot ROC curves
    plot_roc_curves(fbase, roc_curves, "Validation")

def load_batch_results(fbase):
    """Load the results for the whole batch"""

    file_pattern = fbase + "/experiment*/"
    filepaths = glob.glob(file_pattern)

    results = []
    for filepath in filepaths:
        results.append( load_result_from_experiment(filepath) )

    return results

def load_result_from_experiment(fbase):
    """Load result of given experiment"""
    filename = fbase + "results_dict.pkl"
    with open(filename, "rb") as fp:
        return pickle.load(fp)

def generate_roc_curve(outs, predictions):
        '''
        Produce a ROC plot given a model, a set of inputs and the true outputs
        Assume that model produces N-class output; we will only look at the class 0 score
s
        '''
        # Compute false positive rate & true positive rate + AUC
        fpr, tpr, thresholds = sklearn.metrics.roc_curve(outs[:,0], predictions[:,0])
```

```
        auc = sklearn.metrics.auc(fpr, tpr)

        curve = {
            "fpr": fpr,
            "tpr": tpr,
            "auc": auc
        }

        return curve

def plot_learning_curves(fbase, argmin_losses, curves, set_name):
    """Plot learning curves for the given curves and losses"""
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    # Accumulate the accuracy values for averaging, and plot curves
    acc_sum = 0
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    for i, curve in enumerate(curves):
        loss = argmin_losses[i][0]
        vmax = curve[argmin_losses[i][1]]
        acc_sum += vmax
        ax.plot(curve, label="Best Loss {:.2f}:  Epoch {}, Accuracy {:.2f}".format(argmin
_losses[i][0], argmin_losses[i][1], vmax))

    # Organize figure
    plt.title("{} Learning Curves -- Shallow CNN".format(set_name))
    ax.legend(loc="upper left", bbox_to_anchor=(1, 1))
    plt.ylabel("Accuracy")
    plt.xlabel("Epochs")

    # Save figure
    fig.savefig(fbase + "learning_curves.png", dpi=fig.dpi, bbox_inches="tight")

    # Save mean accuracies
    with open("{}mean_validation.txt".format(fbase), "w") as f:
        f.write("Average accuracy: {}".format(acc_sum/len(curves)))

def plot_roc_curves(fbase, curves, set_name):
    """Plot roc curves given a set of curves"""

    if not os.path.exists(fbase):
        os.mkdir(fbase)

    # Generate the plot
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    for curve in curves:
        ax.plot(curve["fpr"], curve["tpr"], 'r', label='AUC = {:.3f}'.format(curve["auc"]
))

    # Organize figure
    plt.title("{} ROC Curves -- Shallow CNN".format(set_name))
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

    # Save figure
    fig.savefig(fbase + "roc_curves.png", dpi=fig.dpi)

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

```
#!/usr/bin/env python3

"""
This is built to be a general purpose configuration system for
supervised/unsupervised learning tasks.
Process:
    (1) Read configuration file
    (2) Create batch of experiments based on default configuration and options
    (3) Spit out each experiment as a separate process
    (4) Within each experiment subprocess, run train and logs results
    (5) Evaluate functions available after training is complete
"""

import pickle
import os
import sys
import json
import subprocess
import itertools

import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.optimizers import Adam

from config.config_handler import config_handler
from exceptions import MissingConfigArgException

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

    # Get cfg name
    cfg_name = get_cfg_name()

    # Create configuration handler
    cfg_handler = config_handler(cfg_name)

    if cfg_handler.get_mode() == "train":
        # Create index file
        create_index_log(cfg_handler)

    # Start a job for each hyperparameter
    print("Number of jobs to start: {}".format(cfg_handler.get_num_experiments()))
    for i in range(cfg_handler.get_num_experiments()):
        start_job(cfg_name, i, cfg_handler.get_mode())

def get_cfg_name():
    """Get cfg name from the command line args"""

    for arg in sys.argv:
        if "-cfg_name=" in arg:
            return arg.replace("-cfg_name=", "")

    raise MissingConfigArgException()

def get_exp_num():
    """Get experiment number from the command line args"""

    for arg in sys.argv:
        if "-exp_num=" in arg:
            return int(arg.replace("-exp_num=", ""))

def get_results_from_file(fbase):

    filename = fbase + "results_dict.pkl"
    with open(filename, "rb") as fp:
        return pickle.load(fp)

def create_index_log(cfg_handler):
    """Write index to file that describes experiment hyperparameters"""
```

```
    # Create directories
    fbase = "results/"
    if not os.path.exists(fbase):
        os.mkdir(fbase)
    batch_name = cfg_handler.get_experiment(0).save.experiment_batch_name
    fbase = "{}{}/".format(fbase, batch_name)
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    # Write index log to file
    with open('{}index.txt'.format(fbase), 'w') as f:
        f.write("Number of experiments: {}\n".format(cfg_handler.get_num_experiments()))
        json.dump(cfg_handler.get_options(), f)
        f.write("\n\n")

        for i in range(cfg_handler.get_num_experiments()):
            individual_option = cfg_handler.get_option(i)
            f.write("\tExperiment {}: ".format(i))
            json.dump(individual_option, f)
            f.write("\n")

def start_job(config_name, experiment_num, mode):
    """
    Starts a job for the fed arguments. This takes the form of a subprocess,
    whether on a normal computer or supercomputer
    """

    print("Starting job: {}".format(experiment_num))

    # 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",
        "-cfg_name={}".format(config_name),
        "-exp_num={}".format(experiment_num)
    ]

    if mode == "train":
        full_command.append("-train")
    elif mode == "eval":
        full_command.append("-eval")
    else:
        raise Exception("Error: configuration mode must be either train or eval")


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

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

def evaluate(model=None):
    """
    Evaluation only works on existing models and cached results. The model can be fed as
an argument,
    or reloaded through the configuration file. If reloaded, it will save results into
    the original experiment's directory.
    """

    # Get configuration values
    cfg_name = get_cfg_name()
    experiment_num = get_exp_num()
    cfg_handler = config_handler(cfg_name)
```

```
    # Might want to return evaluation_cfg, then load previous exp_cfg from file. This wor
ks for now though.
    filename, exp_cfg = cfg_handler.get_experiment(experiment_num)

    # Define fbase and create tree
    fbase = "results/"
    if not os.path.exists(fbase):
        os.mkdir(fbase)
    fbase += "{}/".format(exp_cfg.save.experiment_batch_name)
    if not os.path.exists(fbase):
        os.mkdir(fbase)
    fbase += "experiment_{}/".format(experiment_num)
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    # Create print mechanisms
    reset_log_files(fbase, exp_cfg.mode)
    redirect_stdout(fbase, exp_cfg.mode)
    redirect_stderr(fbase, exp_cfg.mode)

    # Load original configuration for this experiment and the associated results director
y
    revived_cfg = cfg_handler.get_cfg_from_file(fbase)
    results = get_results_from_file(fbase)

    # Load dataset
    dataset = cfg_handler.get_dataset(exp_cfg)

    # Load model
    if model is None:
        model = cfg_handler.get_model(input_size=dataset.get_input_size(), exp_cfg=exp_cf
g, filename=filename)
    model.summary()

    # Load evaluation functions
    evaluation_functions = cfg_handler.get_evaluation_functions(exp_cfg)

    # Use evaluation functions
    for evaluation_function in evaluation_functions:
        evaluation_function(dataset, model, exp_cfg, revived_cfg, results, filename)

def train():
    """Trains one ML model"""

    # Get configuration values
    cfg_name = get_cfg_name()
    experiment_num = get_exp_num()
    cfg_handler = config_handler(cfg_name)
    exp_cfg = cfg_handler.get_experiment(experiment_num)

    # Define fbase and create tree
    fbase = "results/"
    if not os.path.exists(fbase):
        os.mkdir(fbase)
    fbase += "{}/".format(exp_cfg.save.experiment_batch_name)
    if not os.path.exists(fbase):
        os.mkdir(fbase)
    fbase += "experiment_{}/".format(experiment_num)
    if not os.path.exists(fbase):
        os.mkdir(fbase)

    # Create print mechanisms
    reset_log_files(fbase, exp_cfg.mode)
    redirect_stdout(fbase, exp_cfg.mode)
    redirect_stderr(fbase, exp_cfg.mode)

    # Cache configuration for future reload
    save_cfg(exp_cfg, fbase)

    # Print info
```

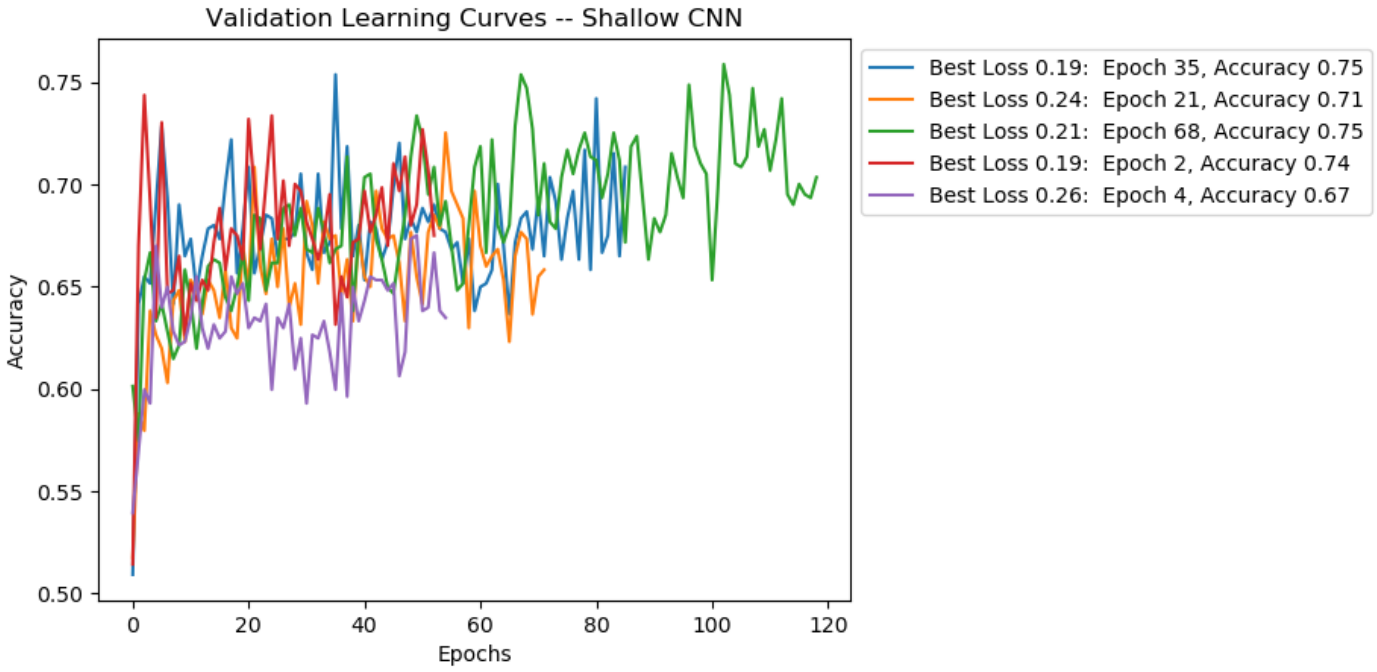
```
    print("Config name: {}".format(cfg_name))
    print("Experiment num: {}".format(experiment_num))
    print()

    # Load data
    dataset = cfg_handler.get_dataset(exp_cfg)
    data_dict = dataset.load_data()

    print("Train ins shape: {}".format(data_dict["train"]["ins"].shape))
    if type(data_dict["train"]["outs"]) == list:
        print("Train outs shape: {}".format(data_dict["train"]["outs"][0].shape))
        print("Train outs shape: {}".format(data_dict["train"]["outs"][1].shape))
    else:
        print("Train outs shape: {}".format(data_dict["train"]["outs"].shape))

    # Build model
    model = cfg_handler.get_model(
        input_size=dataset.get_input_size(),
        exp_cfg=exp_cfg)

    # Compile model
    model.compile(
        optimizer=Adam(learning_rate=exp_cfg.train.learning_rate),
        loss=cfg_handler.get_loss(exp_cfg.train.loss),
        metrics=exp_cfg.train.metrics)
    model.summary()

    # Callbacks
    callbacks = cfg_handler.get_callbacks(fbase, exp_cfg)

    # Future work
    # Check if train key is a dict. If so, do non-generator training. Else, do generator
training.
    # Check if val key exists. If not, don't use.
    #   Otherwise, check if val is a dict or generator and proceed accordingly
    # In total, 6 options.

    # Train model. Requirements for model.fit are liable to expand in the future, and wou
ld
    #   require further development. e.g. using sample or class weights, validation steps
, or validation frequency
    if "val" in data_dict.keys():
        history = model.fit(
                x=data_dict["train"]["ins"],
                y=data_dict["train"]["outs"],
                validation_data = (data_dict["val"]["ins"], data_dict["val"]["outs"]),
                epochs=exp_cfg.train.epochs,
                batch_size=exp_cfg.train.batch_size,
                callbacks=callbacks,
                verbose=exp_cfg.train.verbose
                )
    else:
        history = model.fit(
                x=data_dict["train"]["ins"],
                y=data_dict["train"]["outs"],
                epochs=exp_cfg.train.epochs,
                batch_size=exp_cfg.train.batch_size,
                callbacks=callbacks,
                verbose=exp_cfg.train.verbose
                )

    # Log results
    log_results(data_dict, model, exp_cfg, fbase)

def log_results(data, model, exp_cfg, fbase):
    """Log results to file"""

    print("Logging results")

    # Generate results
```

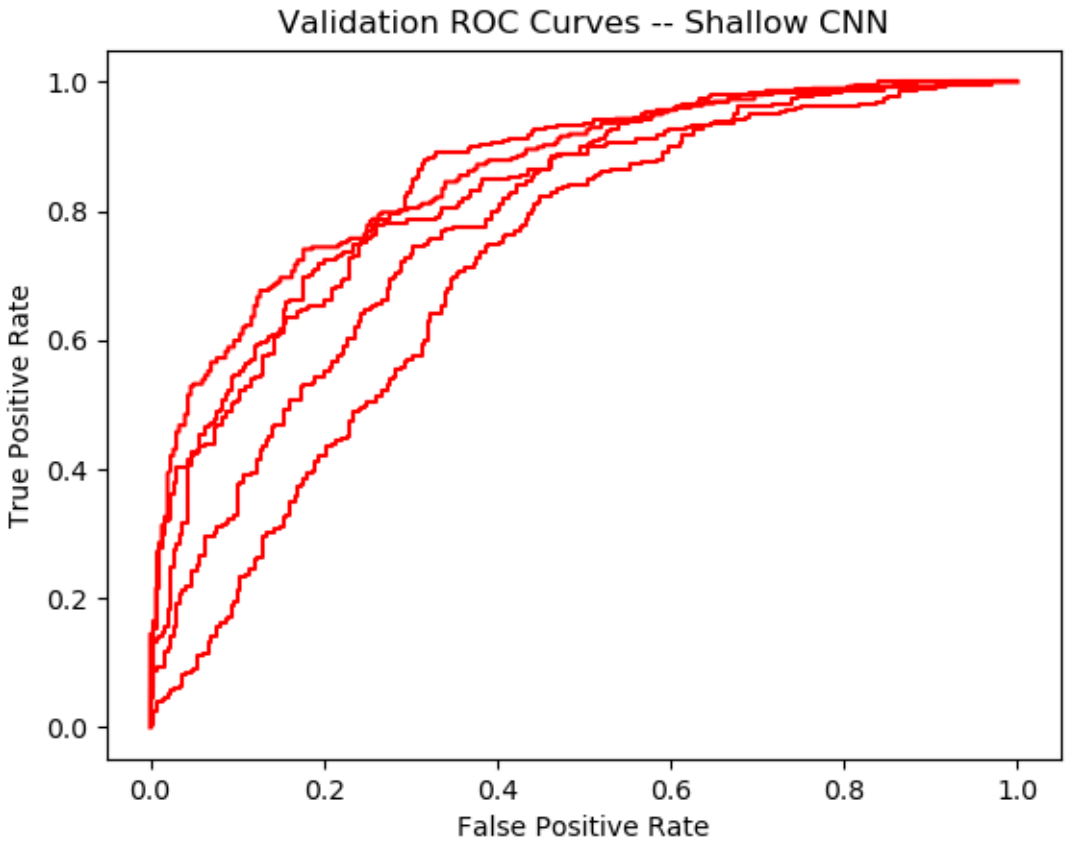
```
    results = {}
    results_brief = {}
    results["history"] = model.history.history
    if "train" in data.keys():
        results['predict_train'] = model.predict(data["train"]["ins"])
        results['eval_train'] = model.evaluate(data["train"]["ins"], data["train"]["outs"
])
        results_brief["eval_train"] = str(results["eval_train"])

    if "val" in data.keys():
        results['predict_val'] = model.predict(data["val"]["ins"])
        results['eval_val'] = model.evaluate(data["val"]["ins"], data["val"]["outs"])
        results_brief["eval_val"] = str(results["eval_val"])

    if "test" in data.keys():
        results['predict_test'] = model.predict(data["test"]["ins"])
        results['eval_test'] = model.evaluate(data["test"]["ins"], data["test"]["outs"])
        results_brief["eval_test"] = str(results["eval_test"])

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

    # Save model
    model.save("{}/model_and_cfg/saved_weights.h5".format(fbase), save_format="tf")

    # Save brief results for human readability
    with open("{}results_brief.txt".format(fbase), "w") as f:
        f.write(json.dumps(results_brief))

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

def save_cfg(exp_cfg, fbase):
    # File path
    filename = fbase + "model_and_cfg/"
    if not os.path.exists(filename):
        os.mkdir(filename)

    filename += "exp_cfg"
    with open(filename, "wt") as f:
        f.write(exp_cfg.dump())

def reset_log_files(fbase, mode):
    """Clears all logs files"""

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

    # File path
    with open("{}{}_out_log.txt".format(fbase, mode), "w") as f:
        pass

    with open("{}{}_err_log.txt".format(fbase, mode), "w") as f:
        pass

def redirect_stdout(fbase, mode):
    """Redirect stdout to file"""

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

    sys.stdout = open("{}{}_out_log.txt".format(fbase, mode), mode="a")

    print("--------Stdout Start--------")

def redirect_stderr(fbase, mode):
```

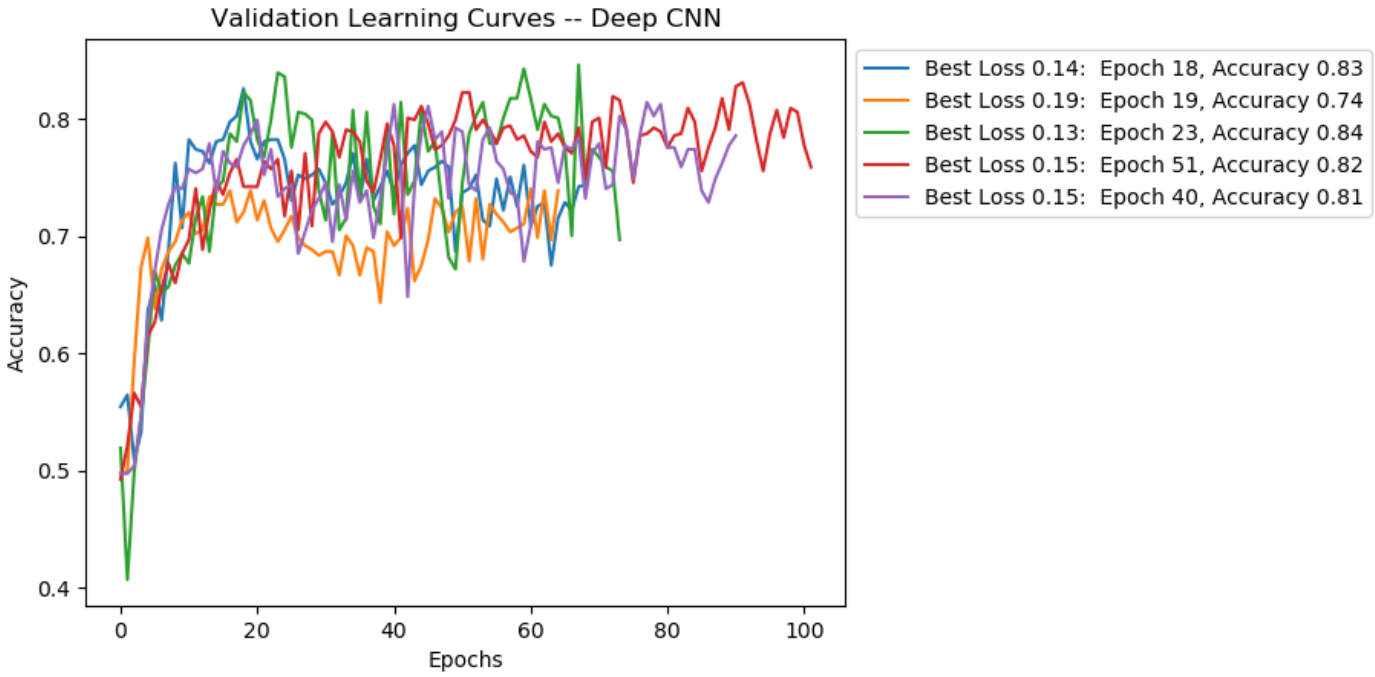
```
    """Redirect stderr to file"""

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

    sys.stderr = open("{}{}_err_log.txt".format(fbase, mode), mode="a")

    print("--------Stderr Start--------", file=sys.stderr)

if __name__ == "__main__":

    # If this is a subprocess, run the training program
    if "-job" in sys.argv:
        if "-train" in sys.argv:
            train()
        elif "-eval" in sys.argv:
            evaluate()
    else:
        main()
```

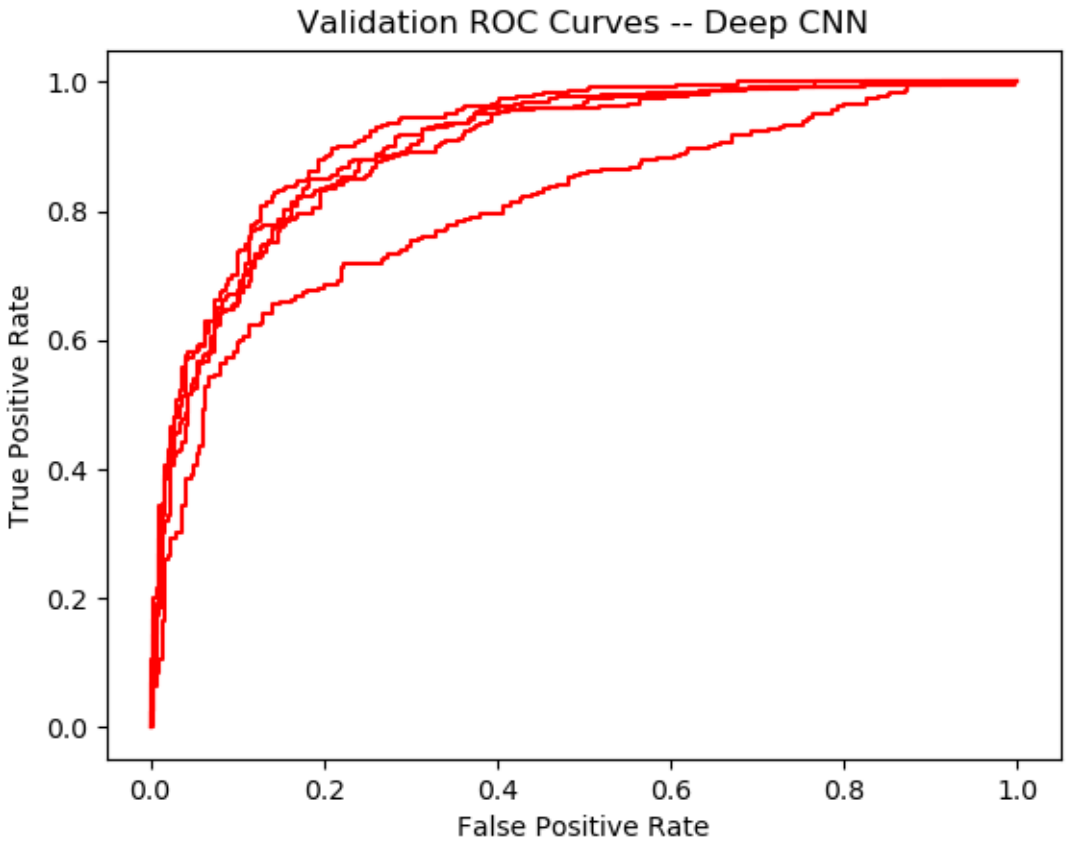
```
from yacs.config import CfgNode as CN

# Construct root
_D = CN()

# Training or evaluation
_D.mode = "train"

# Save parameters
_D.save = CN()
_D.save.experiment_batch_name = "shallow_test_1"

# Dataset parameters
_D.dataset = CN()
_D.dataset.name = "Core50Dataset"

# Model parameters
_D.model = CN()
_D.model.name = "cnn"
_D.model.input_axis_norm = 3
_D.model.conv = CN()
_D.model.conv.filters = [10]
_D.model.conv.kernels = [3]
_D.model.conv.strides = [2]
_D.model.conv.l2 = 0
_D.model.conv.max_pool_sizes = [2]
_D.model.conv.batch_norms = [1]
_D.model.dense = CN()
_D.model.dense.hidden_sizes = [100]
_D.model.dense.dropout = 0
_D.model.dense.batch_norms = [1]
_D.model.output = CN()
_D.model.output.output_size = 2
_D.model.output.activation = "softmax"
_D.model.reload_path = ""

# Training parameters
_D.train = CN()
_D.train.learning_rate = .001
_D.train.epochs = 200
_D.train.batch_size = 32
_D.train.loss = "mse"
_D.train.metrics = ["acc"]
_D.train.verbose = 2

# Callback parameters
_D.callbacks = CN()
_D.callbacks.names = ["EarlyStopping", "FileMetricLogger"]
_D.callbacks.EarlyStopping = CN()
_D.callbacks.EarlyStopping.patience = 50
_D.callbacks.EarlyStopping.min_delta = .0001

# Evaluation parameters
_D.evaluate = CN()

# Misc parameters
_D.misc = CN()
_D.misc.default_duplicate = 5 # Duplicates experiments by this amount. Only activates if
all options are empty

# Construct list of configuration keys and their possible options
# â\200¢ If the key is in the list, the default is overwritten, unless its corresponding
value list is empty
all_options_dict = {}
```

```
#!/bin/bash

python3 main.py $@
```


Validation Learning Curves -- Shallow CNN
Accuracy
Epochs
0.50
0.55
0.60
0.65
0.70
0.75
0
20
40
60
80
100
120
Best Loss 0.19: Epoch 35, Accuracy 0.75
Best Loss 0.24: Epoch 21, Accuracy 0.71
Best Loss 0.21: Epoch 68, Accuracy 0.75
Best Loss 0.19: Epoch 2, Accuracy 0.74
Best Loss 0.26: Epoch 4, Accuracy 0.67


Validation ROC Curves -- Shallow CNN
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


Validation Learning Curves -- Deep CNN
Accuracy
Epochs
0.4
0.5
0.6
0.7
0.8
0
20
40
60
80
100
Best Loss 0.14: Epoch 18, Accuracy 0.83
Best Loss 0.19: Epoch 19, Accuracy 0.74
Best Loss 0.13: Epoch 23, Accuracy 0.84
Best Loss 0.15: Epoch 51, Accuracy 0.82
Best Loss 0.15: Epoch 40, Accuracy 0.81


Validation ROC Curves -- Deep CNN
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

Mean Validation Accuracies:

- Deep CNN: 0.8077051997184753
- Shallow CNN: 0.7246231198310852