Gradual_shift_better.py

```
import utils
import models
import datasets
import numpy as np
import tensorflow as tf
from tensorflow.keras import metrics
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import pickle

def compile_model(model, loss='ce'):
  loss = models.get_loss(loss, model.output_shape[1])
  model.compile(optimizer='adam',
          loss=[loss],
          metrics=[metrics.sparse_categorical_accuracy])

def train_model_source(model, split_data, epochs=1000):
  model.fit(split_data.src_train_x, split_data.src_train_y, epochs=epochs,
verbose=False)
  print("Source accuracy:")
  _, src_acc = model.evaluate(split_data.src_val_x, split_data.src_val_y)
  print("Target accuracy:")
  _, target_acc = model.evaluate(split_data.target_val_x, split_data.target_val_y)
  return src_acc, target_acc

def run_experiment(
  dataset_func, n_classes, input_shape, save_file,
model_func=models.simple_softmax_conv_model,
  interval=2000, epochs=10, loss='ce', soft=False, conf_q=0.1, num_runs=20,
num_repeats=None):
  (src_tr_x, src_tr_y, src_val_x, src_val_y, inter_x, inter_y, dir_inter_x, dir_inter_y,
    trg_val_x, trg_val_y, trg_test_x, trg_test_y) = dataset_func()
  if soft:
    src_tr_y = to_categorical(src_tr_y)
    src_val_y = to_categorical(src_val_y)
    trg_eval_y = to_categorical(trg_eval_y)
    dir_inter_y = to_categorical(dir_inter_y)
    inter_y = to_categorical(inter_y)
    trg_test_y = to_categorical(trg_test_y)
  if num_repeats is None:
    num_repeats = int(inter_x.shape[0] / interval)
  def new_model():
    model = model_func(n_classes, input_shape=input_shape)
    compile_model(model, loss)
    return model
  def student_func(teacher):
    return teacher
  def run(seed):
    utils.rand_seed(seed)
    trg_eval_x = trg_val_x
    trg_eval_y = trg_val_y
    # Train source model.
    source_model = new_model()
    source_model.fit(src_tr_x, src_tr_y, epochs=epochs, verbose=False)
    _, src_acc = source_model.evaluate(src_val_x, src_val_y)
    _, target_acc = source_model.evaluate(trg_eval_x, trg_eval_y)
    # Gradual self-training.
    print("\n\n Gradual self-training:")
    teacher = new_model()
    teacher.set_weights(source_model.get_weights())
    gradual_accuracies, student = utils.gradual_self_train(
      student_func, teacher, inter_x, inter_y, interval, epochs=epochs, soft=soft,
      confidence_q=conf_q)
    _, acc = student.evaluate(trg_eval_x, trg_eval_y)
    gradual_accuracies.append(acc)
    # Train to target.
    print("\n\n Direct boostrap to target:")
    teacher = new_model()
    teacher.set_weights(source_model.get_weights())
    target_accuracies, _ = utils.self_train(
      student_func, teacher, dir_inter_x, epochs=epochs, target_x=trg_eval_x,
      target_y=trg_eval_y, repeats=num_repeats, soft=soft, confidence_q=conf_q)
    print("\n\n Direct boostrap to all unsup data:")
    teacher = new_model()
    teacher.set_weights(source_model.get_weights())
    all_accuracies, _ = utils.self_train(
      student_func, teacher, inter_x, epochs=epochs, target_x=trg_eval_x,
      target_y=trg_eval_y, repeats=num_repeats, soft=soft, confidence_q=conf_q)
    return src_acc, target_acc, gradual_accuracies, target_accuracies, all_accuracies
  results = []
  for i in range(num_runs):
    results.append(run(i))
  print('Saving to ' + save_file)
  pickle.dump(results, open(save_file, "wb"))

def experiment_results(save_name):
  results = pickle.load(open(save_name, "rb"))
  src_accs, target_accs = [], []
  final_graduals, final_targets, final_alls = [], [], []
  best_targets, best_alls = [], []
  for src_acc, target_acc, gradual_accuracies, target_accuracies, all_accuracies in
results:
    src_accs.append(100 * src_acc)
    target_accs.append(100 * target_acc)
    final_graduals.append(100 * gradual_accuracies[-1])
    final_targets.append(100 * target_accuracies[-1])
    final_alls.append(100 * all_accuracies[-1])
    best_targets.append(100 * np.max(target_accuracies))
    best_alls.append(100 * np.max(all_accuracies))
  num_runs = len(src_accs)
  mult = 1.645  # For 90% confidence intervals
  print("\nNon-adaptive accuracy on source (%): ", np.mean(src_accs),
      mult * np.std(src_accs) / np.sqrt(num_runs))
  print("Non-adaptive accuracy on target (%): ", np.mean(target_accs),
      mult * np.std(target_accs) / np.sqrt(num_runs))
  print("Gradual self-train accuracy (%): ", np.mean(final_graduals),
      mult * np.std(final_graduals) / np.sqrt(num_runs))
  print("Target self-train accuracy (%): ", np.mean(final_targets),
      mult * np.std(final_targets) / np.sqrt(num_runs))
  print("All self-train accuracy (%): ", np.mean(final_alls),
      mult * np.std(final_alls) / np.sqrt(num_runs))
  print("Best of Target self-train accuracies (%): ", np.mean(best_targets),
      mult * np.std(best_targets) / np.sqrt(num_runs))
  print("Best of All self-train accuracies (%): ", np.mean(best_alls),
      mult * np.std(best_alls) / np.sqrt(num_runs))

def rotated_mnist_60_conv_experiment():
  run_experiment(
    dataset_func=datasets.rotated_mnist_60_data_func, n_classes=10,
input_shape=(28, 28, 1),
    save_file='saved_files/rot_mnist_60_conv.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=10,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

def portraits_conv_experiment():
  run_experiment(
    dataset_func=datasets.portraits_data_func, n_classes=2, input_shape=(32, 32, 1),
    save_file='saved_files/portraits.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=20,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

def gaussian_linear_experiment():
  d = 100
  run_experiment(
    dataset_func=lambda: datasets.gaussian_data_func(d), n_classes=2,
input_shape=(d,),
    save_file='saved_files/gaussian.dat',
    model_func=models.linear_softmax_model, interval=500, epochs=100, loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

# Ablations below.
def rotated_mnist_60_conv_experiment_noconf():
  run_experiment(
    dataset_func=datasets.rotated_mnist_60_data_func, n_classes=10,
input_shape=(28, 28, 1),
    save_file='saved_files/rot_mnist_60_conv_noconf.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=10,
loss='ce',
    soft=False, conf_q=0.0, num_runs=5)

def portraits_conv_experiment_noconf():
  run_experiment(
    dataset_func=datasets.portraits_data_func, n_classes=2, input_shape=(32, 32, 1),
    save_file='saved_files/portraits_noconf.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=20,
loss='ce',
    soft=False, conf_q=0.0, num_runs=5)

def gaussian_linear_experiment_noconf():
  d = 100
  run_experiment(
    dataset_func=lambda: datasets.gaussian_data_func(d), n_classes=2,
input_shape=(d,),
    save_file='saved_files/gaussian_noconf.dat',
    model_func=models.linear_softmax_model, interval=500, epochs=100, loss='ce',
    soft=False, conf_q=0.0, num_runs=5)

def portraits_64_conv_experiment():
  run_experiment(
```

```
    dataset_func=datasets.portraits_64_data_func, n_classes=2, input_shape=(64, 64,
1),
    save_file='saved_files/portraits_64.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=20,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

def dialing_ratios_mnist_experiment():
  run_experiment(
    dataset_func=datasets.rotated_mnist_60_dialing_ratios_data_func,
    n_classes=10, input_shape=(28, 28, 1),
    save_file='saved_files/dialing_rot_mnist_60_conv.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=10,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

def portraits_conv_experiment_more():
  run_experiment(
    dataset_func=datasets.portraits_data_func_more, n_classes=2, input_shape=(32,
32, 1),
    save_file='saved_files/portraits_more.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=20,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

def rotated_mnist_60_conv_experiment_smaller_interval():
  run_experiment(
    dataset_func=datasets.rotated_mnist_60_data_func, n_classes=10,
input_shape=(28, 28, 1),
    save_file='saved_files/rot_mnist_60_conv_smaller_interval.dat',
    model_func=models.simple_softmax_conv_model, interval=1000, epochs=10,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5, num_repeats=7)

def portraits_conv_experiment_smaller_interval():
  run_experiment(
    dataset_func=datasets.portraits_data_func, n_classes=2, input_shape=(32, 32, 1),
    save_file='saved_files/portraits_smaller_interval.dat',
    model_func=models.simple_softmax_conv_model, interval=1000, epochs=20,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5, num_repeats=7)

def gaussian_linear_experiment_smaller_interval():
  d = 100
  run_experiment(
    dataset_func=lambda: datasets.gaussian_data_func(d), n_classes=2,
input_shape=(d,),
    save_file='saved_files/gaussian_smaller_interval.dat',
    model_func=models.linear_softmax_model, interval=250, epochs=100, loss='ce',
    soft=False, conf_q=0.1, num_runs=5, num_repeats=7)


def rotated_mnist_60_conv_experiment_more_epochs():
  run_experiment(
    dataset_func=datasets.rotated_mnist_60_data_func, n_classes=10,
input_shape=(28, 28, 1),
    save_file='saved_files/rot_mnist_60_conv_more_epochs.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=15,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

def portraits_conv_experiment_more_epochs():
  run_experiment(
    dataset_func=datasets.portraits_data_func, n_classes=2, input_shape=(32, 32, 1),
    save_file='saved_files/portraits_more_epochs.dat',
    model_func=models.simple_softmax_conv_model, interval=2000, epochs=30,
loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

def gaussian_linear_experiment_more_epochs():
  d = 100
  run_experiment(
    dataset_func=lambda: datasets.gaussian_data_func(d), n_classes=2,
input_shape=(d,),
    save_file='saved_files/gaussian_more_epochs.dat',
    model_func=models.linear_softmax_model, interval=500, epochs=150, loss='ce',
    soft=False, conf_q=0.1, num_runs=5)

if __name__ == "__main__":
  # Main paper experiments.
  portraits_conv_experiment()
  print("Portraits conv experiment")
  experiment_results('saved_files/portraits.dat')
  rotated_mnist_60_conv_experiment()
  print("Rot MNIST conv experiment")
  experiment_results('saved_files/rot_mnist_60_conv.dat')
  gaussian_linear_experiment()
  print("Gaussian linear experiment")
  experiment_results('saved_files/gaussian.dat')
  print("Dialing MNIST ratios conv experiment")
  dialing_ratios_mnist_experiment()
  experiment_results('saved_files/dialing_rot_mnist_60_conv.dat')

  # Without confidence thresholding.
  portraits_conv_experiment_noconf()
  print("Portraits conv experiment no confidence thresholding")
  experiment_results('saved_files/portraits_noconf.dat')
  rotated_mnist_60_conv_experiment_noconf()
  print("Rot MNIST conv experiment no confidence thresholding")
  experiment_results('saved_files/rot_mnist_60_conv_noconf.dat')
  gaussian_linear_experiment_noconf()
  print("Gaussian linear experiment no confidence thresholding")
  experiment_results('saved_files/gaussian_noconf.dat')

  # Try predicting for next set of data points on portraits.
  portraits_conv_experiment_more()
  print("Portraits next datapoints conv experiment")
  experiment_results('saved_files/portraits_more.dat')

  # Try smaller window sizes.
  portraits_conv_experiment_smaller_interval()
  print("Portraits conv experiment smaller window")
  experiment_results('saved_files/portraits_smaller_interval.dat')
  rotated_mnist_60_conv_experiment_smaller_interval()
  print("Rot MNIST conv experiment smaller window")
  experiment_results('saved_files/rot_mnist_60_conv_smaller_interval.dat')
  gaussian_linear_experiment_smaller_interval()
  print("Gaussian linear experiment smaller window")
  experiment_results('saved_files/gaussian_smaller_interval.dat')

  # Try training more epochs.
  portraits_conv_experiment_more_epochs()
  print("Portraits conv experiment train longer")
  experiment_results('saved_files/portraits_more_epochs.dat')
  rotated_mnist_60_conv_experiment_more_epochs()
  print("Rot MNIST conv experiment train longer")
  experiment_results('saved_files/rot_mnist_60_conv_more_epochs.dat')
  gaussian_linear_experiment_more_epochs()
  print("Gaussian linear experiment train longer")
  experiment_results('saved_files/gaussian_more_epochs.dat')
```

Model.py

```
import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras import regularizers
from tensorflow.keras import losses

# Models.

def linear_model(num_labels, input_shape, l2_reg=0.02):
    linear_model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=input_shape),
    keras.layers.Dense(num_labels, activation=None, name='out',
        kernel_regularizer=regularizers.l2(l2_reg))
    ])
    return linear_model

def linear_softmax_model(num_labels, input_shape, l2_reg=0.02):
    linear_model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=input_shape),
    keras.layers.Dense(num_labels, activation=tf.nn.softmax, name='out',
        kernel_regularizer=regularizers.l2(l2_reg))
    ])
    return linear_model

def mlp_softmax_model(num_labels, input_shape, l2_reg=0.02):
    linear_model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=input_shape),
    keras.layers.Dense(32, activation=tf.nn.relu,
        kernel_regularizer=regularizers.l2(0.0)),
    keras.layers.Dense(32, activation=tf.nn.relu,
        kernel_regularizer=regularizers.l2(0.0)),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(num_labels, activation=tf.nn.softmax, name='out',
        kernel_regularizer=regularizers.l2(l2_reg))
    ])
    return linear_model

def simple_softmax_conv_model(num_labels, hidden_nodes=32,
input_shape=(28,28,1), l2_reg=0.0):
    return keras.models.Sequential([
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same', input_shape=input_shape),
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same'),
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same'),
    keras.layers.Dropout(0.5),
    keras.layers.BatchNormalization(),
    keras.layers.Flatten(name='after_flatten'),
    # keras.layers.Dense(64, activation=tf.nn.relu),
    keras.layers.Dense(num_labels, activation=tf.nn.softmax, name='out')
    ])

def deeper_softmax_conv_model(num_labels, hidden_nodes=32,
input_shape=(28,28,1), l2_reg=0.0):
    return keras.models.Sequential([
    keras.layers.Conv2D(hidden_nodes, (5,5), (1, 1), activation=tf.nn.relu,
                        padding='same', input_shape=input_shape),
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same', input_shape=input_shape),
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same'),
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same'),
    keras.layers.Dropout(0.5),
    keras.layers.BatchNormalization(),
    keras.layers.Flatten(name='after_flatten'),
    # keras.layers.Dense(64, activation=tf.nn.relu),
    keras.layers.Dense(num_labels, activation=tf.nn.softmax, name='out')
    ])

def unregularized_softmax_conv_model(num_labels, hidden_nodes=32,
input_shape=(28,28,1), l2_reg=0.0):
    return keras.models.Sequential([
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same', input_shape=input_shape),
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same'),
    keras.layers.Conv2D(hidden_nodes, (5,5), (2, 2), activation=tf.nn.relu,
                        padding='same'),
    keras.layers.Flatten(name='after_flatten'),
    # keras.layers.Dense(64, activation=tf.nn.relu),
    keras.layers.Dense(num_labels, activation=tf.nn.softmax, name='out')
    ])

def keras_mnist_model(num_labels, input_shape=(28,28,1)):
    model = keras.models.Sequential()
    model.add(keras.layers.Conv2D(32, kernel_size=(3, 3),
                    activation='relu',
                    input_shape=input_shape))
    model.add(keras.layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(keras.layers.Dropout(0.25))
    model.add(keras.layers.Flatten())
    model.add(keras.layers.Dense(128, activation='relu'))
    model.add(keras.layers.Dropout(0.5))
    model.add(keras.layers.Dense(num_labels, activation='softmax'))
    return model

def unregularized_keras_mnist_model(num_labels, input_shape=(28,28,1)):
    model = keras.models.Sequential()
    model.add(keras.layers.Conv2D(32, kernel_size=(3, 3),
                    activation='relu',
                    input_shape=input_shape))
    model.add(keras.layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(keras.layers.Flatten())
    model.add(keras.layers.Dense(128, activation='relu'))
    model.add(keras.layers.Dense(num_labels, activation='softmax'))
    return model

def papernot_softmax_model(num_labels, input_shape=(28,28,1), l2_reg=0.0):
    papernot_conv_model = keras.models.Sequential([
     keras.layers.Conv2D(64, (8, 8), (2,2), activation=tf.nn.relu,
                    padding='same', input_shape=input_shape),
     keras.layers.Conv2D(128, (6,6), (2,2), activation=tf.nn.relu,
                    padding='valid'),
     keras.layers.Conv2D(128, (5,5), (1,1), activation=tf.nn.relu,
                    padding='valid'),
     keras.layers.BatchNormalization(),
     keras.layers.Flatten(name='after_flatten'),
     keras.layers.Dense(num_labels, activation=tf.nn.softmax, name='out')
    ])
    return papernot_conv_model

# Losses.

def sparse_categorical_hinge(num_classes):
    def loss(y_true,y_pred):
        y_true = tf.reduce_mean(y_true, axis=1)
        y_true = tf.one_hot(tf.cast(y_true, dtype=tf.int32), depth=num_classes)
        return losses.categorical_hinge(y_true, y_pred)
    return loss

def sparse_categorical_ramp(num_classes):
    def loss(y_true,y_pred):
        y_true = tf.reduce_mean(y_true, axis=1)
        y_true = tf.one_hot(tf.cast(y_true, dtype=tf.int32), depth=num_classes)
        return tf.sqrt(losses.categorical_hinge(y_true, y_pred))
    return loss

def get_loss(loss_name, num_classes):
    if loss_name == 'hinge':
        loss = sparse_categorical_hinge(num_classes)
    elif loss_name == 'ramp':
        loss = sparse_categorical_ramp(num_classes)
    elif loss_name == 'ce':
        loss = losses.sparse_categorical_crossentropy
    elif loss_name == 'categorical_ce':
        loss = losses.categorical_crossentropy
    else:
        raise ValueError("Cannot parse loss %s", loss_name)
    return loss
```

Utils.py

```
import numpy as np
from tensorflow.keras.models import load_model
import tensorflow as tf

def rand_seed(seed):
    np.random.seed(seed)
    tf.compat.v1.set_random_seed(seed)

def self_train_once(student, teacher, unsup_x, confidence_q=0.1, epochs=20):
    # Do one bootstrapping step on unsup_x, where pred_model is used to make
predictions,
    # and we use these predictions to update model.
    logits = teacher.predict(np.concatenate([unsup_x]))
    confidence = np.amax(logits, axis=1) - np.amin(logits, axis=1)
    alpha = np.quantile(confidence, confidence_q)
    indices = np.argwhere(confidence >= alpha)[:, 0]
    preds = np.argmax(logits, axis=1)
    student.fit(unsup_x[indices], preds[indices], epochs=epochs, verbose=False)

def soft_self_train_once(student, teacher, unsup_x, epochs=20):
    probs = teacher.predict(np.concatenate([unsup_x]))
    student.fit(unsup_x, probs, epochs=epochs, verbose=False)

def self_train(student_func, teacher, unsup_x, confidence_q=0.1, epochs=20,
repeats=1,
               target_x=None, target_y=None, soft=False):
    accuracies = []
    for i in range(repeats):
        student = student_func(teacher)
        if soft:
            soft_self_train_once(student, teacher, unsup_x, epochs)
        else:
            self_train_once(student, teacher, unsup_x, confidence_q, epochs)
        if target_x is not None and target_y is not None:
            _, accuracy = student.evaluate(target_x, target_y, verbose=True)
            accuracies.append(accuracy)
        teacher = student
    return accuracies, student

def gradual_self_train(student_func, teacher, unsup_x, debug_y, interval,
confidence_q=0.1,
                       epochs=20, soft=False):
    upper_idx = int(unsup_x.shape[0] / interval)
    accuracies = []
    for i in range(upper_idx):
        student = student_func(teacher)
        cur_xs = unsup_x[interval*i:interval*(i+1)]
        cur_ys = debug_y[interval*i:interval*(i+1)]
        # _, student = self_train(
        #     student_func, teacher, unsup_x, confidence_q, epochs, repeats=2)
        if soft:
            soft_self_train_once(student, teacher, cur_xs, epochs)
        else:
            self_train_once(student, teacher, cur_xs, confidence_q, epochs)
        _, accuracy = student.evaluate(cur_xs, cur_ys)
        accuracies.append(accuracy)
        teacher = student
    return accuracies, student

def split_data(xs, ys, splits):
    return np.split(xs, splits), np.split(ys, splits)

def train_to_acc(model, acc, train_x, train_y, val_x, val_y):
    # Modify steps per epoch to be around dataset size / 10
    # Keep training until accuracy
    batch_size = 32
    data_size = train_x.shape[0]
    steps_per_epoch = int(data_size / 50.0 / batch_size)
    logger.info("train_xs size is %s", str(train_x.shape))
    while True:
        model.fit(train_x, train_y, batch_size=batch_size,
steps_per_epoch=steps_per_epoch, verbose=False)
        val_accuracy = model.evaluate(val_x, val_y, verbose=False)[1]
        logger.info("validation accuracy is %f", val_accuracy)
        if val_accuracy >= acc:
            break
    return model

def save_model(model, filename):
    model.save(filename)

def load_model(filename):
    model = load_model(filename)

def rolling_average(sequence, r):
    N = sequence.shape[0]
    assert r < N
    assert r > 1
    rolling_sums = []
    cur_sum = sum(sequence[:r])
    rolling_sums.append(cur_sum)
    for i in range(r, N):
        cur_sum = cur_sum + sequence[i] - sequence[i-r]
        rolling_sums.append(cur_sum)
    return np.array(rolling_sums) * 1.0 / r
```

Dataset_test.py

```
from scipy.io import loadmat, savemat
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from skimage.transform import resize

def shuffle(xs, ys):
    indices = list(range(len(xs)))
    np.random.shuffle(indices)
    return xs[indices], ys[indices]

def mnist_resize(x):
    H, W, C = 32, 32, 3
    x = x.reshape(-1, 28, 28)
    resized_x = np.empty((len(x), H, W), dtype='float32')
    for i, img in enumerate(x):
        if i % 1000 == 0:
            print(i)
        # resize returns [0, 1]
        resized_x[i] = resize(img, (H, W), mode='reflect')

    # Retile to make RGB
    resized_x = resized_x.reshape(-1, H, W, 1)
    resized_x = np.tile(resized_x, (1, 1, 1, C))
    return resized_x

def save_mnist_32():
    (mnist_x, mnist_y), (_, _) = mnist.load_data()
    mnist_x = mnist_resize(mnist_x / 255.0)
    savemat('mnist32_train.mat', {'X': mnist_x, 'y': mnist_y})

def make_mnist_svhn_dataset(num_examples, mnist_start_prob, mnist_end_prob):
    data = loadmat('mnist32_train.mat')
    mnist_x = data['X']
    mnist_y = data['y']
    mnist_y = np.squeeze(mnist_y)
    mnist_x, mnist_y = shuffle(mnist_x, mnist_y)
    print(np.min(mnist_x), np.max(mnist_x))

    data = loadmat('svhn_train_32x32.mat')
    svhn_x = data['X']
    svhn_x = svhn_x / 255.0
    svhn_x = np.transpose(svhn_x, [3, 0, 1, 2])
    svhn_y = data['y']
    svhn_y = np.squeeze(svhn_y)
    svhn_y[(svhn_y == 10)] = 0
    svhn_x, svhn_y = shuffle(svhn_x, svhn_y)
    print(svhn_x.shape, svhn_y.shape)
    print(np.min(svhn_x), np.max(svhn_x))

    delta = float(mnist_end_prob - mnist_start_prob) / (num_examples - 1)
    mnist_probs = np.array([mnist_start_prob + delta * i for i in range(num_examples)])
    # assert((np.all(mnist_end_prob >= mnist_probs) and np.all(mnist_probs >=
mnist_start_prob)) or
    #     (np.all(mnist_start_prob >= mnist_probs) and np.all(mnist_probs >=
mnist_end_prob)))
    domains = np.random.binomial(n=1, p=mnist_probs)
    assert(domains.shape == (num_examples,))
    mnist_indices = np.arange(num_examples)[domains == 1]
    svhn_indices = np.arange(num_examples)[domains == 0]
    print(svhn_x.shape, mnist_x.shape)
    assert(svhn_x.shape[1:] == mnist_x.shape[1:])
    print(mnist_indices[:10], svhn_indices[:10], svhn_indices[-10:])
    xs = np.empty((num_examples,) + tuple(svhn_x.shape[1:]), dtype='float32')
    ys = np.empty((num_examples,), dtype='int32')
    xs[mnist_indices] = mnist_x[:mnist_indices.size]
    xs[svhn_indices] = svhn_x[:svhn_indices.size]
    ys[mnist_indices] = mnist_y[:mnist_indices.size]
    ys[svhn_indices] = svhn_y[:svhn_indices.size]
    return xs, ys


save_mnist_32()
# xs, ys = make_mnist_svhn_dataset(10000, 0.9, 0.1)
# print(xs.shape, ys.shape)
# ex_0 = xs[ys == 0][0]
# plt.imshow(ex_0)
# plt.show()
# ex_0 = xs[ys == 0][-1]
# plt.imshow(ex_0)
# plt.show()

    # Read and process MNIST images
    # Read and process SVHN images
    # Shuffle MNIST and SVHN images
    # First generate an array of datasets
        # Interpolate between start and end prob
        # Use that to pick and index from a Bernoulli
    # Then select the images (could do some fancy indexing, or just do it manually,
append images)
    # Should be fast enough, did that in rotated dataset
    # Return this

# data = loadmat('svhn_train_32x32.mat')
# Xs = data['X']
# Xs = np.transpose(Xs, [3, 0, 1, 2])
# Ys = data['y']
# Ys = np.squeeze(Ys)
# print(Xs.shape, Ys.shape)
# print(np.min(Ys), np.max(Ys))
# Ys[(Ys == 10)] = 0
# print(np.min(Ys), np.max(Ys))
# print(np.min(Xs), np.max(Xs))

# Resize MNIST images to make them colored. Just add extra channels.
# Need to check min and max for MNIST and SVHN, preprocess them as needed
# Could add instance normalization as well, if we think that helps.
# (train_x, train_y), (_, _) = mnist.load_data()
# print(np.min(train_x), np.max(train_x))
# train_x = np.tile(np.expand_dims(train_x, axis=-1), (1, 1, 1, 3))
# print(np.min(train_y), np.max(train_y))
# print(train_y.shape)
# ex_0 = train_x[train_y == 0][0]
# plt.imshow(ex_0)
# plt.show()
# print(train_x.shape)

# X10s = Xs[y10s]
# print(X10s.shape)
# plt.imshow(X10s[0])
# plt.show()
```

Datasets.py

```
import collections
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.datasets import mnist
from tensorflow.keras.datasets import cifar10
import scipy.io
from scipy import ndimage
from scipy.stats import ortho_group
import sklearn.preprocessing
import pickle
import utils

Dataset = collections.namedtuple('Dataset',
    'get_data n_src_train n_src_valid n_target_unsup n_target_val n_target_test
target_end '
    'n_classes input_shape')

SplitData = collections.namedtuple('SplitData',
    ('src_train_x src_val_x src_train_y src_val_y target_unsup_x target_val_x
final_target_test_x '
     'debug_target_unsup_y target_val_y final_target_test_y inter_x inter_y'))

image_options = {
    'batch_size': 100,
    'class_mode': 'binary',
    'color_mode': 'grayscale',
}

def split_sizes(array, sizes):
    indices = np.cumsum(sizes)
    return np.split(array, indices)

def shuffle(xs, ys):
    indices = list(range(len(xs)))
    np.random.shuffle(indices)
    return xs[indices], ys[indices]

def get_split_data(dataset):
    Xs, Ys = dataset.get_data()
    n_src = dataset.n_src_train + dataset.n_src_valid
    n_target = dataset.n_target_unsup + dataset.n_target_val + dataset.n_target_test
    src_x, src_y = shuffle(Xs[:n_src], Ys[:n_src])
    target_x, target_y = shuffle(
        Xs[dataset.target_end-n_target:dataset.target_end],
        Ys[dataset.target_end-n_target:dataset.target_end])
    [src_train_x, src_val_x] = split_sizes(src_x, [dataset.n_src_train])
    [src_train_y, src_val_y] = split_sizes(src_y, [dataset.n_src_train])
    [target_unsup_x, target_val_x, final_target_test_x] = split_sizes(
        target_x, [dataset.n_target_unsup, dataset.n_target_val])
    [debug_target_unsup_y, target_val_y, final_target_test_y] = split_sizes(
        target_y, [dataset.n_target_unsup, dataset.n_target_val])
    inter_x, inter_y = Xs[n_src:dataset.target_end-n_target],
Ys[n_src:dataset.target_end-n_target]
    return SplitData(
        src_train_x=src_train_x,
        src_val_x=src_val_x,
        src_train_y=src_train_y,
        src_val_y=src_val_y,
        target_unsup_x=target_unsup_x,
        target_val_x=target_val_x,
        final_target_test_x=final_target_test_x,
        debug_target_unsup_y=debug_target_unsup_y,
        target_val_y=target_val_y,
        final_target_test_y=final_target_test_y,
        inter_x=inter_x,
        inter_y=inter_y,
    )

# Gaussian dataset.

def shape_means(means):
        means = np.array(means)
        if len(means.shape) == 1:
            means = np.expand_dims(means, axis=-1)
        else:
            assert(len(means.shape) == 2)
        return means

def shape_sigmas(sigmas, means):
    sigmas = np.array(sigmas)
    shape_len = len(sigmas.shape)
    assert(shape_len == 1 or shape_len == 3)
    new_sigmas = sigmas
    if shape_len == 1:
        c = np.expand_dims(np.expand_dims(sigmas, axis=-1), axis=-1)
        d = means.shape[1]
        new_sigmas = c * np.eye(d)
        assert(new_sigmas.shape == (sigmas.shape[0], d, d))
    return new_sigmas

def get_gaussian_at_alpha(source_means, source_sigmas, target_means,
target_sigmas, alpha):
    num_classes = source_means.shape[0]
    class_prob = 1.0 / num_classes
    y = np.argmax(np.random.multinomial(1, [class_prob] * num_classes))
    mean = source_means[y] * (1 - alpha) + target_means[y] * alpha
    sigma = source_sigmas[y] * (1 - alpha) + target_sigmas[y] * alpha
    x = np.random.multivariate_normal(mean, sigma)
    return x, y

def sample_gaussian_alpha(source_means, source_sigmas, target_means,
target_sigmas,
                          alpha_low, alpha_high, N):
    source_means = shape_means(source_means)
    target_means = shape_means(target_means)
    source_sigmas = shape_sigmas(source_sigmas, source_means)
    target_sigmas = shape_sigmas(target_sigmas, target_means)
    xs, ys = [], []
    for i in range(N):
        if alpha_low == alpha_high:
            alpha = alpha_low
        else:
            alpha = np.random.uniform(low=alpha_low, high=alpha_high)
        x, y = get_gaussian_at_alpha(
            source_means, source_sigmas, target_means, target_sigmas, alpha)
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)

def continual_gaussian_alpha(source_means, source_sigmas, target_means,
target_sigmas,
                             alpha_low, alpha_high, N):
    source_means = shape_means(source_means)
    target_means = shape_means(target_means)
    source_sigmas = shape_sigmas(source_sigmas, source_means)
    target_sigmas = shape_sigmas(target_sigmas, target_means)
    xs, ys = [], []
    for i in range(N):
        alpha = float(alpha_high - alpha_low) / N * i + alpha_low
        x, y = get_gaussian_at_alpha(
            source_means, source_sigmas, target_means, target_sigmas, alpha)
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)

def make_moving_gaussian_data(
    source_means, source_sigmas, target_means, target_sigmas,
    source_alphas, inter_alphas, target_alphas,
    n_src_tr, n_src_val, n_inter, n_trg_val, n_trg_tst):
    src_tr_x, src_tr_y = sample_gaussian_alpha(
        source_means, source_sigmas, target_means, target_sigmas,
        source_alphas[0], source_alphas[1], N=n_src_tr)
    src_val_x, src_val_y = sample_gaussian_alpha(
        source_means, source_sigmas, target_means, target_sigmas,
        source_alphas[0], source_alphas[1], N=n_src_val)
    inter_x, inter_y = continual_gaussian_alpha(
        source_means, source_sigmas, target_means, target_sigmas,
        inter_alphas[0], inter_alphas[1], N=n_inter)
    dir_inter_x, dir_inter_y = sample_gaussian_alpha(
        source_means, source_sigmas, target_means, target_sigmas,
        target_alphas[0], target_alphas[1], N=n_inter)
    trg_val_x, trg_val_y = sample_gaussian_alpha(
        source_means, source_sigmas, target_means, target_sigmas,
        target_alphas[0], target_alphas[1], N=n_trg_val)
    trg_test_x, trg_test_y = sample_gaussian_alpha(
        source_means, source_sigmas, target_means, target_sigmas,
        target_alphas[0], target_alphas[1], N=n_trg_tst)
    return (src_tr_x, src_tr_y, src_val_x, src_val_y, inter_x, inter_y,
            dir_inter_x, dir_inter_y, trg_val_x, trg_val_y, trg_test_x, trg_test_y)

def make_high_d_gaussian_data(
    d, min_var, max_var, source_alphas, inter_alphas, target_alphas,
    n_src_tr, n_src_val, n_inter, n_trg_val, n_trg_tst):
    assert(min_var > 0)
    means, var_list = [], []
    for i in range(4):
        means.append(np.random.multivariate_normal(np.zeros(d), np.eye(d)))
        means[i] = means[i] / np.linalg.norm(means[i])
        # Generate diagonal.
        diag = np.diag(np.random.uniform(min_var, max_var, size=d))
        rot = ortho_group.rvs(d)
```

```
    var = np.matmul(rot, np.matmul(diag, np.linalg.inv(rot)))
    var_list.append(var)
  return make_moving_gaussian_data(
    source_means=[means[0], means[1]], source_sigmas=[var_list[0], var_list[1]],
    target_means=[means[2], means[3]], target_sigmas=[var_list[2], var_list[3]],
    source_alphas=source_alphas, inter_alphas=inter_alphas,
target_alphas=target_alphas,
    n_src_tr=n_src_tr, n_src_val=n_src_val, n_inter=n_inter,
    n_trg_val=n_trg_val, n_trg_tst=n_trg_tst)

def make_moving_gaussians(source_means, source_sigmas, target_means,
target_sigmas, steps):
  source_means = shape_means(source_means)
  target_means = shape_means(target_means)
  source_sigmas = shape_sigmas(source_sigmas, source_means)
  target_sigmas = shape_sigmas(target_sigmas, target_means)
  for i in range(steps):
    alpha = float(i) / (steps - 1)
    mean = source_means[y] * (1 - alpha) + target_means[y] * alpha
    sigma = source_sigmas[y] * (1 - alpha) + target_sigmas[y] * alpha
    x, y = get_gaussian_at_alpha()
    xs.append(x)
    ys.append(y)
  return np.array(xs), np.array(ys)

def high_d_gaussians(d, var, n):
  # Choose random direction.
  v = np.random.multivariate_normal(np.zeros(d), np.eye(d))
  v = v / np.linalg.norm(v)
  # Choose random perpendicular direction.
  perp = np.random.multivariate_normal(np.zeros(d), np.eye(d))
  perp = perp - np.dot(perp, v) * v
  perp = perp / np.linalg.norm(perp)
  assert(abs(np.dot(perp, v)) < 1e-8)
  assert(abs(np.linalg.norm(v) - 1) < 1e-8)
  assert(abs(np.linalg.norm(perp) - 1) < 1e-8)
  s_a = 2 * perp - v
  s_b = -2 * perp + v
  t_a = -2 * perp - v
  t_b = 2 * perp + v
  return lambda: make_moving_gaussians([s_a, s_b], [var, var], [t_a, t_b], [var, var], n)

# MNIST datasets.

def get_preprocessed_mnist():
  (train_x, train_y), (test_x, test_y) = mnist.load_data()
  train_x, test_x = train_x / 255.0, test_x / 255.0
  train_x, train_y = shuffle(train_x, train_y)
  train_x = np.expand_dims(np.array(train_x), axis=-1)
  test_x = np.expand_dims(np.array(test_x), axis=-1)
  return (train_x, train_y), (test_x, test_y)

def sample_rotate_images(xs, start_angle, end_angle):
  new_xs = []
  num_points = xs.shape[0]
  for i in range(num_points):
    if start_angle == end_angle:
      angle = start_angle
    else:
      angle = np.random.uniform(low=start_angle, high=end_angle)
    img = ndimage.rotate(xs[i], angle, reshape=False)
    new_xs.append(img)
  return np.array(new_xs)

def continually_rotate_images(xs, start_angle, end_angle):
  new_xs = []
  num_points = xs.shape[0]
  for i in range(num_points):
    angle = float(end_angle - start_angle) / num_points * i + start_angle
    img = ndimage.rotate(xs[i], angle, reshape=False)
    new_xs.append(img)
  return np.array(new_xs)

def _transition_rotation_dataset(train_x, train_y, test_x, test_y,
                 source_angles, target_angles, inter_func,
                 src_train_end, src_val_end, inter_end, target_end):
  assert(target_end <= train_x.shape[0])
  assert(train_x.shape[0] == train_y.shape[0])
  src_tr_x, src_tr_y = train_x[:src_train_end], train_y[:src_train_end]
  src_tr_x = sample_rotate_images(src_tr_x, source_angles[0], source_angles[1])
  src_val_x, src_val_y = train_x[src_train_end:src_val_end],
train_y[src_train_end:src_val_end]
  src_val_x = sample_rotate_images(src_val_x, source_angles[0], source_angles[1])
  tmp_inter_x, inter_y = train_x[src_val_end:inter_end],
train_y[src_val_end:inter_end]
  inter_x = inter_func(tmp_inter_x)
  dir_inter_x = sample_rotate_images(tmp_inter_x, target_angles[0],
target_angles[1])
  dir_inter_y = np.array(inter_y)
  assert(inter_x.shape == dir_inter_x.shape)
  trg_val_x, trg_val_y = train_x[inter_end:target_end], train_y[inter_end:target_end]
  trg_val_x = sample_rotate_images(trg_val_x, target_angles[0], target_angles[1])
  trg_test_x, trg_test_y = test_x, test_y
  trg_test_x = sample_rotate_images(trg_test_x, target_angles[0], target_angles[1])
  return (src_tr_x, src_tr_y, src_val_x, src_val_y, inter_x, inter_y,
      dir_inter_x, dir_inter_y, trg_val_x, trg_val_y, trg_test_x, trg_test_y)

def dial_rotation_proportions(xs, source_angles, target_angles):
  N = xs.shape[0]
  new_xs = []
  rotate_ps = np.arange(N) / float(N - 1)
  is_target = np.random.binomial(n=1, p=rotate_ps)
  assert(is_target.shape == (N,))
  for i in range(N):
    if is_target[i]:
      angle = np.random.uniform(low=target_angles[0], high=target_angles[1])
    else:
      angle = np.random.uniform(low=source_angles[0], high=source_angles[1])
    cur_x = ndimage.rotate(xs[i], angle, reshape=False)
    new_xs.append(cur_x)
  return np.array(new_xs)

def dial_proportions_rotated_dataset(train_x, train_y, test_x, test_y,
                   source_angles, target_angles,
                   src_train_end, src_val_end, inter_end, target_end):
  inter_func = lambda x: dial_rotation_proportions(
    x, source_angles, target_angles)
  return _transition_rotation_dataset(
    train_x, train_y, test_x, test_y, source_angles, target_angles,
    inter_func, src_train_end, src_val_end, inter_end, target_end)

def make_rotated_dataset(train_x, train_y, test_x, test_y,
            source_angles, inter_angles, target_angles,
            src_train_end, src_val_end, inter_end, target_end):
  inter_func = lambda x: continually_rotate_images(x, inter_angles[0],
inter_angles[1])
  return _transition_rotation_dataset(
    train_x, train_y, test_x, test_y, source_angles, target_angles,
    inter_func, src_train_end, src_val_end, inter_end, target_end)

def make_population_rotated_dataset(xs, ys, delta_angle, num_angles):
  images, labels = [], []
  for i in range(num_angles):
    cur_angle = i * delta_angle
    cur_images = sample_rotate_images(xs, cur_angle, cur_angle)
    images.append(cur_images)
    labels.append(ys)
  images = np.concatenate(images, axis=0)
  labels = np.concatenate(labels, axis=0)
  assert images.shape[1:] == xs.shape[1:]
  assert labels.shape[1:] == ys.shape[1:]
  return images, labels

def make_rotated_dataset_continuous(dataset, start_angle, end_angle, num_points):
  images, labels = [], []
  (train_x, train_y), (_, _) = dataset.load_data()
  train_x, train_y = shuffle(train_x, train_y)
  train_x = train_x / 255.0
  assert(num_points < train_x.shape[0])
  indices = np.random.choice(train_x.shape[0], size=num_points, replace=False)
  for i in range(num_points):
    angle = float(end_angle - start_angle) / num_points * i + start_angle
    idx = indices[i]
    img = ndimage.rotate(train_x[idx], angle, reshape=False)
    images.append(img)
    labels.append(train_y[idx])
  return np.array(images), np.array(labels)

def make_rotated_mnist(start_angle, end_angle, num_points, normalize=False):
  Xs, Ys = make_rotated_dataset(mnist, start_angle, end_angle, num_points)
  if normalize:
    Xs = np.reshape(Xs, (Xs.shape[0], -1))
    old_mean = np.mean(Xs)
    Xs = sklearn.preprocessing.normalize(Xs, norm='l2')
    new_mean = np.mean(Xs)
    Xs = Xs * (old_mean / new_mean)
  return np.expand_dims(np.array(Xs), axis=-1), Ys

def make_rotated_cifar10(start_angle, end_angle, num_points):
  return make_rotated_dataset(cifar10, start_angle, end_angle, num_points)

def make_mnist():
  (train_x, train_y), (_, _) = mnist.load_data()
```

```
    train_x = train_x / 255.0
    return np.expand_dims(train_x, axis=-1), train_y

def make_mnist_svhn_dataset(num_examples, mnist_start_prob, mnist_end_prob):
    data = scipy.io.loadmat('mnist32_train.mat')
    mnist_x = data['X']
    mnist_y = data['y']
    mnist_y = np.squeeze(mnist_y)
    mnist_x, mnist_y = shuffle(mnist_x, mnist_y)

    data = scipy.io.loadmat('svhn_train_32x32.mat')
    svhn_x = data['X']
    svhn_x = svhn_x / 255.0
    svhn_x = np.transpose(svhn_x, [3, 0, 1, 2])
    svhn_y = data['y']
    svhn_y = np.squeeze(svhn_y)
    svhn_y[(svhn_y == 10)] = 0
    svhn_x, svhn_y = shuffle(svhn_x, svhn_y)

    delta = float(mnist_end_prob - mnist_start_prob) / (num_examples - 1)
    mnist_probs = np.array([mnist_start_prob + delta * i for i in range(num_examples)])
    # assert((np.all(mnist_end_prob >= mnist_probs) and np.all(mnist_probs >=
mnist_start_prob)) or
    #     (np.all(mnist_start_prob >= mnist_probs) and np.all(mnist_probs >=
mnist_end_prob)))
    domains = np.random.binomial(n=1, p=mnist_probs)
    assert(domains.shape == (num_examples,))
    mnist_indices = np.arange(num_examples)[domains == 1]
    svhn_indices = np.arange(num_examples)[domains == 0]
    assert(svhn_x.shape[1:] == mnist_x.shape[1:])
    xs = np.empty((num_examples,) + tuple(svhn_x.shape[1:]), dtype='float32')
    ys = np.empty((num_examples,), dtype='int32')
    xs[mnist_indices] = mnist_x[:mnist_indices.size]
    xs[svhn_indices] = svhn_x[:svhn_indices.size]
    ys[mnist_indices] = mnist_y[:mnist_indices.size]
    ys[svhn_indices] = svhn_y[:svhn_indices.size]
    return xs, ys

# Portraits dataset.

def save_data(data_dir='dataset_32x32', save_file='dataset_32x32.mat',
target_size=(32, 32)):
    Xs, Ys = [], []
    datagen = ImageDataGenerator(rescale=1./255)
    data_generator = datagen.flow_from_directory(
        data_dir, shuffle=False, target_size=target_size, **image_options)
    while True:
        next_x, next_y = data_generator.next()
        Xs.append(next_x)
        Ys.append(next_y)
        if data_generator.batch_index == 0:
            break
    Xs = np.concatenate(Xs)
    Ys = np.concatenate(Ys)
    filenames = [f[2:] for f in data_generator.filenames]
    assert(len(set(filenames)) == len(filenames))
    filenames_idx = list(zip(filenames, range(len(filenames))))
    filenames_idx = [(f, i) for f, i in zip(filenames, range(len(filenames)))]
                # if f[5:8] == 'Cal' or f[5:8] == 'cal']
    indices = [i for f, i in sorted(filenames_idx)]
    genders = np.array([f[:1] for f in data_generator.filenames])[indices]
    binary_genders = (genders == 'F')
    pickle.dump(binary_genders, open('portraits_gender_stats', "wb"))
    print("computed gender stats")
    # gender_stats = utils.rolling_average(binary_genders, 500)
    # print(filenames)
    # sort_indices = np.argsort(filenames)
    # We need to sort only by year, and not have correlation with state.
    # print state stats? print gender stats? print school stats?
    # E.g. if this changes a lot by year, then we might want to do some grouping.
    # Maybe print out number per year, and then we can decide on a grouping? Or
algorithmically decide?
    Xs = Xs[indices]
    Ys = Ys[indices]
    scipy.io.savemat('./' + save_file, mdict={'Xs': Xs, 'Ys': Ys})

def load_portraits_data(load_file='dataset_32x32.mat'):
    data = scipy.io.loadmat('./' + load_file)
    return data['Xs'], data['Ys'][0]

def make_portraits_data(n_src_tr, n_src_val, n_inter, n_target_unsup, n_trg_val,
n_trg_tst,
                        load_file='dataset_32x32.mat'):
    xs, ys = load_portraits_data(load_file)
    src_end = n_src_tr + n_src_val
    inter_end = src_end + n_inter
    trg_end = inter_end + n_trg_val + n_trg_tst
    src_x, src_y = shuffle(xs[:src_end], ys[:src_end])
    trg_x, trg_y = shuffle(xs[inter_end:trg_end], ys[inter_end:trg_end])
    [src_tr_x, src_val_x] = split_sizes(src_x, [n_src_tr])
    [src_tr_y, src_val_y] = split_sizes(src_y, [n_src_tr])
    [trg_val_x, trg_test_x] = split_sizes(trg_x, [n_trg_val])
    [trg_val_y, trg_test_y] = split_sizes(trg_y, [n_trg_val])
    inter_x, inter_y = xs[src_end:inter_end], ys[src_end:inter_end]
    dir_inter_x, dir_inter_y = inter_x[-n_target_unsup:], inter_y[-n_target_unsup:]
    return (src_tr_x, src_tr_y, src_val_x, src_val_y, inter_x, inter_y,
            dir_inter_x, dir_inter_y, trg_val_x, trg_val_y, trg_test_x, trg_test_y)

def rotated_mnist_60_data_func():
    (train_x, train_y), (test_x, test_y) = get_preprocessed_mnist()
    return make_rotated_dataset(
        train_x, train_y, test_x, test_y, [0.0, 5.0], [5.0, 60.0], [55.0, 60.0],
        5000, 6000, 48000, 50000)

def rotated_mnist_60_dialing_ratios_data_func():
    (train_x, train_y), (test_x, test_y) = get_preprocessed_mnist()
    return dial_proportions_rotated_dataset(
        train_x, train_y, test_x, test_y, [0.0, 5.0], [55.0, 60.0],
        5000, 6000, 48000, 50000)

def portraits_data_func():
    return make_portraits_data(1000, 1000, 14000, 2000, 1000, 1000)

def portraits_data_func_more():
    return make_portraits_data(1000, 1000, 20000, 2000, 1000, 1000)

def portraits_64_data_func():
    return make_portraits_data(1000, 1000, 14000, 2000, 1000, 1000,
load_file='dataset_64x64.mat')

def gaussian_data_func(d):
    return make_high_d_gaussian_data(
        d=d, min_var=0.05, max_var=0.1,
        source_alphas=[0.0, 0.0], inter_alphas=[0.0, 1.0], target_alphas=[1.0, 1.0],
        n_src_tr=500, n_src_val=1000, n_inter=5000, n_trg_val=1000, n_trg_tst=1000)
```