Only-label 成员推理复现

目标数据集：CIFAR10

代码理解

主要有模型代码 model.py、训练代码 training.py、攻击代码 attacks.py 组成。

1. 模型代码 model.py

（1）CNN make-conv()函数
创建一个卷积神经网络模型 CNN。

a.

```
if len(input_shape) == 2:
  data_format = 'channels_first'
elif len(input_shape) == 3 and (input_shape[0] == 1 or input_shape[0] == 3):
  data_format = 'channels_first'
else:
  data_format = 'channels_last'
conv_model = tf.keras.models.Sequential()
conv_model.add(tf.keras.Input(input_shape))
```

根据输入的张量类型，选择图像数据格式。channel_first 表示将通道数放在了第一个位置，channel_last 就表示将通道数放在了最后位置。

b.

```
for _ in range(depth):
  conv_model.add(tf.keras.layers.Conv2D(32, (3, 3), padding='same',
                                        data_format=data_format))
  conv_model.add(tf.keras.layers.Activation('relu'))
  conv_model.add(tf.keras.layers.Conv2D(32, (3, 3), data_format=data_format))
  conv_model.add(tf.keras.layers.Activation('relu'))
conv_model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
for _ in range(depth):
  conv_model.add(tf.keras.layers.Conv2D(64, (3, 3), padding='same',
                                        data_format=data_format))
  conv_model.add(tf.keras.layers.Activation('relu'))
  conv_model.add(tf.keras.layers.Conv2D(64, (3, 3), data_format=data_format))
  conv_model.add(tf.keras.layers.Activation('relu'))
conv_model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
```

创建模型，共 8 个卷积层、2 个池化层，分为两部分。

c.

```
if regularization == 'l1':
  k_reg = tf.keras.regularizers.L1L2(l1=reg_constant)
elif regularization == 'l2':
  k_reg = tf.keras.regularizers.L1L2(l2=reg_constant)
else:
  k_reg = None
```

正则化处理，有 l1、l2 或无范数。

d.

```
if regularization == 'dropout':
  conv_model.add(tf.keras.layers.Dropout(reg_constant))
else:
  conv_model.add(tf.keras.layers.Dropout(0.0))
```

Dropout 层，是用来防止过拟合的。通过随机丢弃一些神经元（将输出设置为 0）来减少对模型过分的依赖。Else 部分表示丢弃率为 0，即无该层。

e

```
conv_model.add(tf.keras.layers.Flatten())
conv_model.add(tf.keras.layers.Dense(512, kernel_regularizer=k_reg))
conv_model.add(tf.keras.layers.Activation('relu'))
```

全连接层。Flatten（）将卷积层的多维输出转换成一维。后添加一个有 512 个神经元的全连接层，并应用正则化，类型就是 k_reg，激活函数还是常见的 relu。

（2）make_fc.()

该部分和上面函数的功能差不多，只是神经元数量和激活函数（用的 tanh）略微不同。

2. 训练代码 training.py

（1）train_step() 每一步的训练过程。

a.

```
with backprop.GradientTape(persistent=True) as tape:
  tape.watch(x)
  y_pred = self(x, training=True)
  loss = self.compiled_loss(
    y, y_pred, sample_weight, regularization_losses=self.losses)
  if self.optimizer._num_microbatches is None:
    self.optimizer._num_microbatches = tf.shape(input=loss)[0]
  losses = [tf.reduce_mean(input_tensor=tf.gather(loss, [idx])) for idx in range(self.optimizer
final_grads = [tape.gradient(losses[i], self.trainable_variables) for i in range(self.optimizer
sample_params = (
  self.optimizer._dp_sum_query.derive_sample_params(self.optimizer._global_state))
```

计算训练过程中的 loss。使用 GradientTape 来计算梯度，结合了微批次（microbatch）的处理方式。y_pred 这一部分应该是在前向传播处理，得到预测值。后面就可以根据这个预测值和实际值进行 loss 的计算。

采用微批次的处理方式。将整体的 loss 分解成微批次的 losses，由其计算微批次梯度，再来优化。这样可以有利于梯度稳定性和优化。

b.

```
var_list = self.trainable_variables
sample_state = self.optimizer._dp_sum_query.initial_sample_state(var_list)
for grads in final_grads:
  grads_list = [g if g is not None else tf.zeros_like(v) for (g, v) in zip(list(grads), var_lis
  sample_state = self.optimizer._dp_sum_query.accumulate_record(
    sample_params, sample_state, grads_list)

grad_sums, self.optimizer._global_state = (
  self.optimizer._dp_sum_query.get_noised_result(
    sample_state, self.optimizer._global_state))
```

引入差分隐私，用于在梯度更新前对梯度进行噪声注入，起到数据保护的作用。
Var_list 包含模型中所有可训练的变量，即需要在反向传播中更新的权重和偏置等参数。根据这个，可以初始化模型的样本状态。
在 for 循环里，处理每个微批次的梯度，并将相关数据、信息保存。
最后，根据刚刚算出来的梯度，添加噪声。这样就可以保护训练时的相关数据。

C.

```
def normalize(v):
  return tf.truediv(v, tf.cast(self.optimizer._num_microbatches, tf.float32))

final_grads = tf.nest.map_structure(normalize, grad_sums)

self.optimizer._was_compute_gradients_called = True
self.optimizer.apply_gradients(zip(final_grads, self.trainable_variables))
self.compiled_metrics.update_state(y, y_pred, sample_weight)
del tape  # since persistent, we need to garbage collect.
return {m.name: m.result() for m in self.metrics}
```

标准化操作。return 部分是将输入 v 除以微批次数量，得到一个平均梯度。
tf.nest.map_structure 用于对 grad_sums 中的每个元素应用 normalize 函数。所以 final_grads 就是每个变量对应的平均梯度。
后面就是 trian_step()的关键点，根据梯度来启用梯度，更新模型权重。

（2）trian_model（对一组模型训练的过程）
训练 models 列表里面的所有模型，但如果没有对应的训练样本就跳过。
A.

```
history = models[i].fit(train_set[0][train_lbl_sel],
                        train_set[2][train_lbl_sel],
                        batch_size=kwargs.get('batch_size', 1),
                        callbacks=[tf.keras.callbacks.EarlyStopping(
                          monitor='val_loss', mode='min', patience=20)],
                        # validation_data=(val_set[0][val_lbl_sel],
                        #                  val_set[2][val_lbl_sel]),
                        epochs=kwargs.get('epochs', 1),
                        shuffle=True, verbose=0,
                        )
```

用 fit（）进行训练。Trian_lbl__sel 是用来提供样本的特征 label 的，EarlyStopping 是用来监控模型 loss，当 loss 不再减少（有预期减少值），就直接停止训练。

B.

```
  hist = history.history
  keys = hist.keys()
  hist = {key: hist[key][-1] for key in keys if key == 'loss' or key == 'sparse_categorical_acc
  epochs.append(len(history.history['loss']))
  losses.append(hist['loss'])
  # print(f"label: '{i}' has {hist}")
return models, np.mean(losses)
```

这部分就是保存相关模型数据了，如 loss、epoch 等。

（3） EpsilonTracker（）

```
class EpsilonTracker(tf.keras.callbacks.Callback):
  def __init__(self, noise_multiplier, *args, **kwargs):
    super().__init__(*args, **kwargs)
    self.eps = []
    self.noise_multiplier = noise_multiplier

  def on_epoch_end(self, epoch, logs={}):
    # eps, _ = compute_eps_poisson(flags_obj.class_size*data.N_CLASSES,
    #                              flags_obj.train_batch_size,
    #                              self.noise_multiplier, epoch,
    #                              (1./flags_obj.class_size*data.N_CLASSES)/10)
    eps = compute_eps_poisson(epoch, self.noise_multiplier,
                              args.ndata,
                              args.batch_size,
                              1. / args.ndata)
    logs['eps'] = eps
    # if epoch % 10 == 0:
    #   print(f"current eps value is: {eps}")
    self.eps.append(eps)
```

跟踪并记录每个训练轮次结束时的隐私损失参数，并更新相关数据。

（4）EarlyStopping（）

提前停止训练的监控函数，当满足 loss 的降低已经到了一定标准后，就停止训练。

A

```
self.monitor = monitor
self.patience = patience
self.verbose = verbose
self.baseline = baseline
self.min_delta = abs(min_delta)
self.wait = 0
self.stopped_epoch = 0
self.restore_best_weights = restore_best_weights
self.best_weights = None
```

定义这个监控过程的系列参数。Monitor 表示要监控的指标，本项目在默认下的监控量就是 loss，如果想引进差分隐私保护的话，通过更改命令可以改变监控量为 eps；patience 表示耐心值，即如果在 patience 轮数下，监控量的改变值一直小于<min_delta（规定的最小改变值），就会停止训练；baseline 表示基线值，monitor 的值大于这个值，符合其他条件就可以停止训练；wait 作为变量，用于记录已经有多少轮 monitor 变化<min_delta；restore_best_weights 表示是否更新该轮的权重为最好权重，这得根据 best_weights 和 loss 来看。

B

```
if mode == 'min':
  self.monitor_op = np.less
elif mode == 'max':
  self.monitor_op = np.greater
else:
  if 'acc' in self.monitor:
    self.monitor_op = np.greater
  else:
    self.monitor_op = np.less

if self.monitor_op == np.greater:
  self.min_delta *= 1
else:
  self.min_delta *= -1
```

根据不同的任务需要，选择不同 min_delta。如果涉及到差分隐私，那么其 eps 在训练过程中肯定是组件增大的，所以 min_delta 就要是正值。Loss 同理。

C

```
def on_train_begin(self, logs=None):
  # Allow instances to be re-used
  self.wait = 0
  self.stopped_epoch = 0
  if self.baseline is not None:
    self.best = self.baseline
  else:
    self.best = np.Inf if self.monitor_op == np.less else -np.Inf
```

初始化。在训练开始前，将各种变量的值进行初始化，选择相应的 min_delta 等。

D

```
def on_epoch_end(self, epoch, logs=None):
  current = self.get_monitor_value(logs)
  if current is None:
    return
  if self.monitor_op(current - self.min_delta, self.best):
    self.wait = 0
    if self.restore_best_weights:
      self.best_weights = self.model.get_weights()
  else:
    print(f"stopping eps of: {current}")
    self.wait += 1
    if self.wait >= self.patience:
      self.stopped_epoch = epoch
      self.model.stop_training = True
      if self.restore_best_weights:
        if self.verbose > 0:
          print('Restoring model weights from the end of the best epoch.')
        self.model.set_weights(self.best_weights)
```

在每一 epoch 结束后，都要进行检查，判断十分需要立即停止训练，判断思路就是 A 中所说的。这个就是根据思路的代码思想，计算出各种变量的值，然后进行比较。如果符合，就把 stop_training 定义为 True，停止训练。最后就是是否需要保存本轮的结果为最好结果以及对应的权重。

E

```
  def on_train_end(self, logs=None):
    if self.stopped_epoch > 0 and self.verbose > 0:
      print('Epoch %05d: early stopping' % (self.stopped_epoch + 1))
```

训练结束时，进行打印。但是在默认代码进行下，verbose 的值是不会大于 0 的，本行不会输出。

F

```
def get_monitor_value(self, logs):
  logs = logs or {}
  monitor_value = logs.get(self.monitor)
  return monitor_value
```

从日志中获取 monitor 的值。

（5）main（）函数

A

```
model = getattr(models, args.model)  # get model_fn

# get datasets for target / source models
target_train_set, target_test_set, source_train_set, source_test_set, input_dim, n_classes = utils.get_da
regularization = "none"
if args.defense in ['l1', 'l2', 'dropout']:
  regularization == args.regularization
elif args.defense in ['', 'dp', 'advreg', 'fine-tune', 'whole']:  # last two are transfer learning
  regularization = 'none'  # these train the model differently, which we will do at training time below.
else:
  raise ValueError(f"Defense: {args.defense} not valid")
```

根据预先定义的参数，获取模型、数据集、测试集、正则化形式。

B

```
target_model = model(input_dim, args.model_depth, regularization, args.reg_constant, n_classes)
source_model = model(input_dim, args.model_depth, regularization, args.reg_constant, n_classes)
t_optim = tf.keras.optimizers.Adam()
t_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
t_metrics = [tf.keras.metrics.SparseCategoricalAccuracy()]
s_optim = tf.keras.optimizers.Adam()
s_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
s_metrics = [tf.keras.metrics.SparseCategoricalAccuracy()]
t_cbs = []
s_cbs = []
```

定义模型的各种变量和参数，包括 loss、范数、优化等，后续要进行处理。

C

```
if args.defense == 'dp':
  target_model.train_step = MethodType(train_step, target_model)
  source_model.train_step = MethodType(train_step, source_model)
  t_optim = DPAdamGaussianOptimizer(learning_rate=0.0001,
                                    num_microbatches=args.batch_size,
                                    noise_multiplier=args.noise,
                                    l2_norm_clip=2.)
  s_optim = DPAdamGaussianOptimizer(learning_rate=0.0001,
                                     num_microbatches=args.batch_size,
                                     noise_multiplier=args.noise,
                                     l2_norm_clip=2.)
  t_cbs.extend([EpsilonTracker(args.noise),
   EarlyStopping(mode='min', patience=0,
                 min_delta=0,
                 monitor='eps',
                 baseline=args.reg_constant),
   ])
  s_cbs.extend([EpsilonTracker(args.noise),
                EarlyStopping(mode='min', patience=0,
                              min_delta=0,
                              monitor='eps',
                              baseline=args.reg_constant),
                ])
```

如果防御机制是 dp（事先已经规定好的），就根据 dp 来设置特定的学习率、微批次大小、噪声和范数。

该部分剩下的两个 if 也一样，都是根据不同的防御机制来选择不同的指标数值和相应的 EarlyStopping 机制。

D

```
target_model.compile(optimizer=t_optim, loss=t_loss, metrics=t_metrics)
source_model.compile(optimizer=s_optim, loss=s_loss, metrics=s_metrics)

target_model.fit(*target_train_set, epochs=1000, callbacks=t_cbs, batch_size=args.batch_size, verbose=2)
source_model.fit(*source_train_set, epochs=1000, callbacks=s_cbs, batch_size=args.batch_size, verbose=2)

target_model.save(args.target_model_path)
source_model.save(args.source_model_path)
```

编译模型、训练模型、保存模型。有一个源模型和目标模型，现在源模型上进行相关推断分析，再到目标模型上。

3.攻击代码 attack.py

（1）get_max_accuracy()函数

```
if thresholds is None:
  fpr, tpr, thresholds = roc_curve(y_true, probs)

accuracy_scores = []
precision_scores = []
for thresh in thresholds:
  accuracy_scores.append(accuracy_score(y_true,
                                        [1 if m > thresh else 0 for m in probs]))
  precision_scores.append(precision_score(y_true, [1 if m > thresh else 0 for m in probs]))

accuracies = np.array(accuracy_scores)
precisions = np.array(precision_scores)
max_accuracy = accuracies.max()
max_precision = precisions.max()
max_accuracy_threshold = thresholds[accuracies.argmax()]
max_precision_threshold = thresholds[precisions.argmax()]
return max_accuracy, max_accuracy_threshold, max_precision, max_precision_threshold
```

如果没有提前设定阈值 thresholds（即用来进行判定的临界值），就通过计算 roc_curve 曲线来收拢阈值。
精准度 precision_scores 是根据 accuracy 和阈值之间的比较得出的。
最后返回最大准确率、精准度。

（2）get_thresholds()函数

```
acc_source, t, prec_source, tprec = get_max_accuracy(source_m, source_stats)

# find best accuracy on test data (just to check how much we overfit)
acc_test, _, prec_test, _ = get_max_accuracy(target_m, target_stats)

# get the test accuracy at the threshold selected on the source data
acc_test_t, _, _, _ = get_max_accuracy(target_m, target_stats, thresholds=[t])
_, _, prec_test_t, _ = get_max_accuracy(target_m, target_stats, thresholds=[tprec])
print("acc src: {}, acc test (best thresh): {}, acc test (src thresh): {}, thresh: {}".format(acc_source, acc_test,
                                                                                              acc_test_t, t))
print(
  "prec src: {}, prec test (best thresh): {}, prec test (src thresh): {}, thresh: {}".format(prec_source, prec_test,
                                                                                             prec_test_t, tprec))

return acc_test_t, prec_test_t, t, tprec
```

使用源模型的输出确定最佳决策阈值，然后在目标模型上评估该阈值的有效性。利用 get_max_accuracy()函数完成下述三个任务：
在源数据上找到最佳阈值。
在目标数据上找到最佳准确率。
在源数据上选择的阈值下评估目标模型。

（3）trian_model（）函数
这个和 training.py 里面的 train_model（）是一样的。

（4）calc_confuse（）函数

```
def calc_confuse(preds, labels):
  labels = labels.astype(np.bool).squeeze()
  preds = np.argmax(preds, axis=1).astype(np.bool)
  tp = np.logical_and(np.equal(labels, True), np.equal(preds, True)).astype(
    np.int).sum()
  fp = np.logical_and(np.equal(labels, False), np.equal(preds, True)).astype(
    np.int).sum()
  tn = np.logical_and(np.equal(labels, False), np.equal(preds, False)).astype(
    np.int).sum()
  fn = np.logical_and(np.equal(labels, True), np.equal(preds, False)).astype(
    np.int).sum()
  return tp, fp, tn, fn
```

计算分类模型的混淆矩阵（Confusion Matrix）中的四个核心指标：真阳数（TP）、假阳数（FP）、真阴数（TN）和假阴例（FN）。可以帮助揭示模型是否倾向于某一类别。（限于二分类任务，或者把 n 分类等效转化成二分类再来用）。

（5）test_model()函数

```
    lbl_sel = test_set[1].flatten() == i
    features = test_set[0][lbl_sel]
    membership_labels = test_set[2][lbl_sel]
    lens.append(len(membership_labels))
    loss, acc = models[i].evaluate(features,
                                   membership_labels,
                                   batch_size=1000,
                                   verbose=0)
    pred = models[i].predict(features)
    tp, fp, tn, fn = calc_confuse(pred, membership_labels)
    # calcacc = (tp + tn) / (tp + tn + fp + fn)
    accs.append(acc)
    tps.append(tp)
    fps.append(fp)
    tns.append(tn)
    fns.append(fn)
    preds.extend(pred)
accuracy = (np.sum(tps) + np.sum(tns)) / (
    np.sum(tps) + np.sum(tns) + np.sum(fps) + np.sum(fns)
r = np.sum(tps) / (np.sum(tps) + np.sum(fns))
p = np.sum(tps) / (np.sum(tps) + np.sum(fps))
f1 = (2 * (r * p)) / (r + p)
```

核心部分如上，就是进一步计算并保存一些列评估指标，这些指标数值的初始值在其他函数里算出来了，这里只是用公式算比率之类的。

（6）assign_best()函数

```
def assign_best(best, testbest, metric, val, other_vals, old_models, new_models, testval, testothers):
  if val > best[metric]:
    best[metric] = val
    testbest[metric] = testval
    for (key, v) in other_vals:
      best[key] = v
    for (key, v) in testothers:
      testbest[key] = v
    return new_models
  return old_models
```

根据给定的性能指标 metric 来决定是否更新模型和对应的性能记录。比较当前模型在某个指标上的表现（val）与之前记录的最佳表现值（best[metric]），如果当前模型的表现更好，就会更新相关的记录，并返回新的模型。否则，保留旧模型不变。相关值也是在其他部分已经算出来了。

（6）AttackModel()类

A

```
def __init__(self, aug_type='n'):
  """ Sample Attack Model.

  :param aug_type:
  """
  super().__init__()
  if aug_type == 'n':
    self.x1 = tf.keras.layers.Dense(64, activation=tf.keras.layers.ReLU(
      negative_slope=1e-2), kernel_initializer='glorot_normal')
    self.x_out = tf.keras.layers.Dense(2, kernel_initializer='glorot_normal')

  elif aug_type == 'r' or aug_type == 'd':
    self.x1 = tf.keras.layers.Dense(10, activation=tf.keras.layers.ReLU(
      negative_slope=1e-2), kernel_initializer='glorot_normal')
    self.x2 = tf.keras.layers.Dense(10, activation=tf.keras.layers.ReLU(
      negative_slope=1e-2), kernel_initializer='glorot_normal')
    self.x_out = tf.keras.layers.Dense(2, kernel_initializer='glorot_normal')
  else:
    raise ValueError(f"aug_type={aug_type} is not valid.")

  self.x_activation = tf.keras.layers.Softmax()
```

根据预先的设定（可在命令中修改，否则就是默认值）确定攻击模型的网络结构。定义两个全连接层：self.out 用于最后分类，self.x1 根据 relu 函数对输出进行相应的处理。

B

```
def call(self, inputs, training=False):
  x = inputs
  for layer in self.layers:
    x = layer(x)
  return x
```

Inputs 就是一个批次的样本，放入模型后， 以此进入每一层对输入样本进行相应的处理。

（7）train_best_attack_model（）函数

```
def train_best_attack_model(train_set, test_set, type_, n_classes=10):
  val_best = {'acc': -1, 'f1': -1, 'prec': -1}
  test_best = {'acc': -1, 'f1': -1, 'prec': -1}

  l = np.array(
    [1 * 10 ** i for i in range(-3, -1)] + [5 * 10 ** i for i in range(-3, -1)])
  best_models = None
  for i in range(len(l)):
    K.clear_session()
    learning_rate = l[i]
    models, epochs = train_model(train_set, learning_rate=learning_rate, type_=type_, n_classes=n_classes)
    acc, _, f1, _, _, _, _, p = test_model(models, train_set, 'source')
    testacc, _, testf1, _, _, _, _, testp = test_model(models, test_set, 'target')

    best_models = assign_best(val_best, test_best, 'acc', acc, [('f1', f1), ('prec', p)], best_models, models, testacc,
                              [('f1', testf1), ('prec', testp)])
  return test_best
```

这个函数就是用来寻找最好的模型，在指定训练集、测试集下对一组模型进行训练、评估，最后返回测试结果最好的模型的 acc、f1、precision。其中涉及到的函数都是前面的，该函数本身没有什么处理重点。

复现过程：

运行如下命令

```
# root @ d7d149c7484f in ~/jxd/membership-inference on git:main x [2:14:42]
$ python training.py
```

发现：

```
25/25 - 0s - loss: 2.3195e-04 - sparse_categorical_accuracy: 1.0000 - 196ms/epoch - 8ms/step
Epoch 43/1000
25/25 - 0s - loss: 2.1915e-04 - sparse_categorical_accuracy: 1.0000 - 193ms/epoch - 8ms/step
Epoch 44/1000
25/25 - 0s - loss: 2.0721e-04 - sparse_categorical_accuracy: 1.0000 - 188ms/epoch - 8ms/step
Epoch 45/1000
25/25 - 0s - loss: 1.9607e-04 - sparse_categorical_accuracy: 1.0000 - 199ms/epoch - 8ms/step
Epoch 46/1000
25/25 - 0s - loss: 1.8629e-04 - sparse_categorical_accuracy: 1.0000 - 190ms/epoch - 8ms/step
Epoch 47/1000
25/25 - 0s - loss: 1.7670e-04 - sparse_categorical_accuracy: 1.0000 - 194ms/epoch - 8ms/step
Epoch 48/1000
25/25 - 0s - loss: 1.6841e-04 - sparse_categorical_accuracy: 1.0000 - 189ms/epoch - 8ms/step
Epoch 1/1000
25/25 - 4s - loss: 2.1958 - sparse_categorical_accuracy: 0.1916 - 4s/epoch - 146ms/step
```

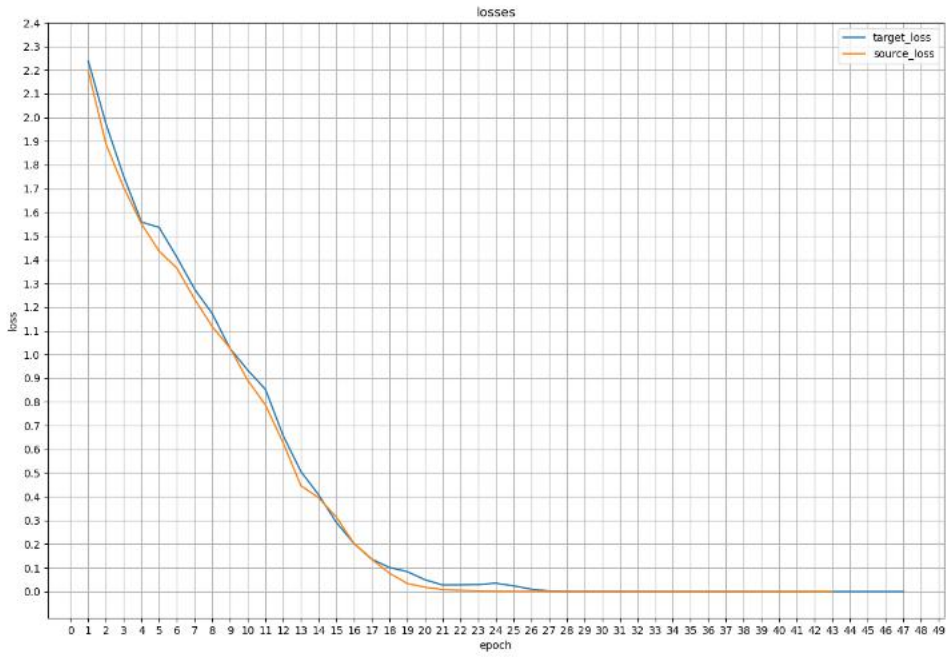
这里从 epoch=48 之间到 epoch=1，表明前面的 target_model 训练结束，并 EarlyStoppoing。

后面的就是 source_model 了。

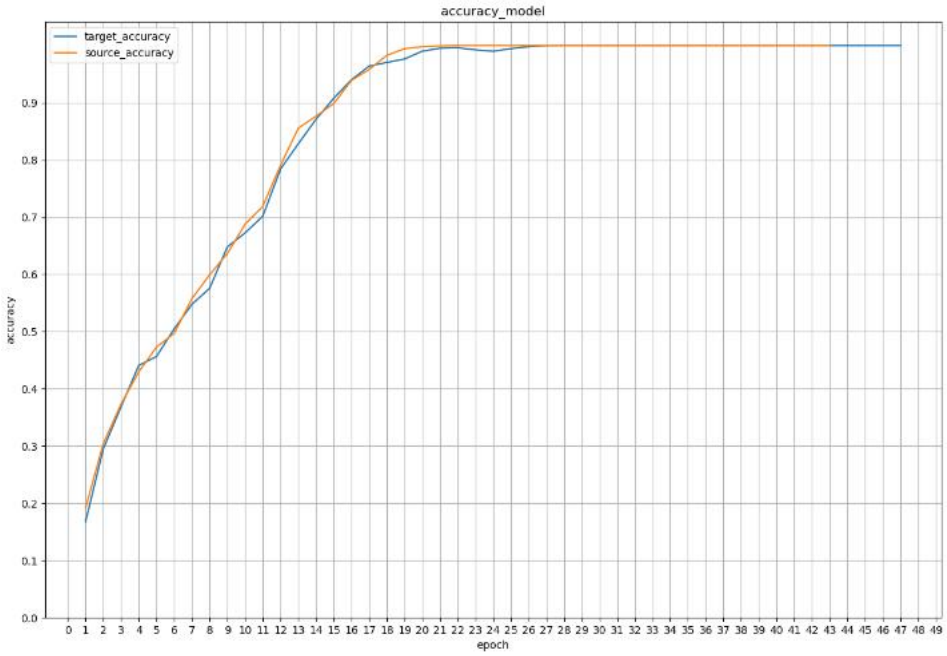
```
25/25 - 0s - loss: 4.8720e-04 - sparse_categorical_accuracy: 1.0000 - 192ms/epoch - 8ms/step
Epoch 33/1000
25/25 - 0s - loss: 4.4384e-04 - sparse_categorical_accuracy: 1.0000 - 240ms/epoch - 10ms/step
Epoch 34/1000
25/25 - 0s - loss: 4.1618e-04 - sparse_categorical_accuracy: 1.0000 - 255ms/epoch - 10ms/step
Epoch 35/1000
25/25 - 0s - loss: 3.7751e-04 - sparse_categorical_accuracy: 1.0000 - 199ms/epoch - 8ms/step
Epoch 36/1000
25/25 - 0s - loss: 3.4684e-04 - sparse_categorical_accuracy: 1.0000 - 207ms/epoch - 8ms/step
Epoch 37/1000
25/25 - 0s - loss: 3.2243e-04 - sparse_categorical_accuracy: 1.0000 - 193ms/epoch - 8ms/step
Epoch 38/1000
25/25 - 0s - loss: 2.9868e-04 - sparse_categorical_accuracy: 1.0000 - 196ms/epoch - 8ms/step
Epoch 39/1000
25/25 - 0s - loss: 2.7799e-04 - sparse_categorical_accuracy: 1.0000 - 194ms/epoch - 8ms/step
Epoch 40/1000
25/25 - 0s - loss: 2.6197e-04 - sparse_categorical_accuracy: 1.0000 - 192ms/epoch - 8ms/step
Epoch 41/1000
25/25 - 0s - loss: 2.4530e-04 - sparse_categorical_accuracy: 1.0000 - 192ms/epoch - 8ms/step
Epoch 42/1000
25/25 - 0s - loss: 2.2945e-04 - sparse_categorical_accuracy: 1.0000 - 191ms/epoch - 8ms/step
Epoch 43/1000
25/25 - 0s - loss: 2.1533e-04 - sparse_categorical_accuracy: 1.0000 - 185ms/epoch - 7ms/step
end
```

这个“end”是我自己加上的，用于表示训练阶段的完成。

两个模型的训练 loss 变化曲线如下：



准确率变化如下：


accuracy_model
target_accuracy
source_accuracy
accuracy
epoch