

```

104     xb_gan_loss = tf.losses.mean_squared_error(tf.zeros_like(xb_logit_gan), xb_logit_gan)
105     d_loss_gan = xa_gan_loss + xb_gan_loss
106     gp = models.gradient_penalty(D, xa)
107
108 #计算分类器模型的重建损失
109 xa_loss_att = tf.losses.sigmoid_cross_entropy(a, xa_logit_att)
110 d_loss = d_loss_gan + gp * 10.0 + xa_loss_att #最终的判别器模型损失
111
112 #计算生成器模型损失
113 if mode == 'wgan':
114     xb_loss_gan = -tf.reduce_mean(xb_logit_gan) #用 wgan-gp 方式
115 elif mode == 'lsgan':
116     xb_loss_gan = tf.losses.mean_squared_error(tf.ones_like(xb_logit_gan), xb_logit_gan)
117
118 #计算分类器模型的重建损失
119 xb_loss_att = tf.losses.sigmoid_cross_entropy(b, xb_logit_att)
120 #用于校准生成器模型的生成结果
121 xa_loss_rec = tf.losses.absolute_difference(xa, xa_)
122 #最终的生成器模型损失
123 g_loss = xb_loss_gan + xb_loss_att * 10.0 + xa_loss_rec * 100.0
124
125 t_vars = tf.trainable_variables() #获得训练参数
126 d_vars = [var for var in t_vars if 'D' in var.name]
127 g_vars = [var for var in t_vars if 'G' in var.name]
128 #定义优化器 OP
129 d_step = tf.train.AdamOptimizer(lr, beta1=0.5).minimize(d_loss,
    var_list=d_vars)
130 g_step = tf.train.AdamOptimizer(lr, beta1=0.5).minimize(g_loss,
    var_list=g_vars)
131 #按照指定属性生成数据，用于测试模型的输出效果
132 x_sample = Gdec(Genc(xa_sample, is_training=False), _b_sample,
    is_training=False)
133
134 def summary(tensor_collection, #定义 summary 处理函数
135             summary_type=['mean', 'stddev', 'max', 'min', 'sparsity',
136             'histogram'],
137             scope=None):
138     def _summary(tensor, name, summary_type):
139         if name is None:
140             name = re.sub('%s_[0-9]*/' % 'tower', '', tensor.name)
141             name = re.sub(':', '-', name)
142
143         summaries = []
144         if len(tensor.shape) == 0:

```

```

145     summaries.append(tf.summary.scalar(name, tensor))
146 else:
147     if 'mean' in summary_type:
148         mean = tf.reduce_mean(tensor)
149         summaries.append(tf.summary.scalar(name + '/mean', mean))
150     if 'stddev' in summary_type:
151         mean = tf.reduce_mean(tensor)
152         stddev = tf.sqrt(tf.reduce_mean(tf.square(tensor - mean)))
153         summaries.append(tf.summary.scalar(name + '/stddev', stddev))
154     if 'max' in summary_type:
155         summaries.append(tf.summary.scalar(name + '/max',
156             tf.reduce_max(tensor)))
157     if 'min' in summary_type:
158         summaries.append(tf.summary.scalar(name + '/min',
159             tf.reduce_min(tensor)))
160     if 'sparsity' in summary_type:
161         summaries.append(tf.summary.scalar(name + '/sparsity',
162             tf.nn.zero_fraction(tensor)))
163     if 'histogram' in summary_type:
164         summaries.append(tf.summary.histogram(name, tensor))
165 return tf.summary.merge(summaries)
166
167 if not isinstance(tensor_collection, (list, tuple, dict)):
168     tensor_collection = [tensor_collection]
169
170 with tf.name_scope(scope, 'summary'):
171     summaries = []
172     if isinstance(tensor_collection, (list, tuple)):
173         for tensor in tensor_collection:
174             summaries.append(_summary(tensor, None, summary_type))
175     else:
176         for tensor, name in tensor_collection.items():
177             summaries.append(_summary(tensor, name, summary_type))
178
179 return tf.summary.merge(summaries)
180
181 # 定义生成 summary 的相关节点
182 d_summary = summary({d_loss_gan: 'd_loss_gan', gp: 'gp',
183     xa_loss_att: 'xa_loss_att', }, scope='D') # 定义判别器模型日志
184
185 lr_summary = summary({lr: 'lr'}, scope='Learning_Rate') # 定义学习率日志
186
187 g_summary = summary({ xb_loss_gan: 'xb_loss_gan',      # 定义生成器模型日志
188     xb_loss_att: 'xb_loss_att',xa_loss_rec: 'xa_loss_rec',
189 }, scope='G')
190
191 d_summary = tf.summary.merge([d_summary, lr_summary])
192
193 def counter(start=0, scope=None):
194     # 对张量进行计数

```

```

189     with tf.variable_scope(scope, 'counter'):
190         counter = tf.get_variable(name='counter',
191                               initializer=tf.constant_initializer(start),
192                               shape=(),
193                               dtype=tf.int64)
194         update_cnt = tf.assign(counter, tf.add(counter, 1))
195     return counter, update_cnt
196 # 定义计数器
197 it_cnt, update_cnt = counter()
198
199 # 定义 saver, 用于读取模型
200 saver = tf.train.Saver(max_to_keep=1)
201
202 # 定义摘要日志写入器
203 summary_writer = tf.summary.FileWriter('./output/%s/summaries' %
204                                         experiment_name, sess.graph)

```

在计算损失值方面，代码第 97~123 行中提供了对抗神经网络模型中计算损失值的两种方式——wgan-gp 与 lsgan-gp。这两种方式都是对抗神经网络模型中主流的计算 loss 值的方式。它可以在训练过程中，使生成器模型与判别器模型很好地收敛。

代码第 123 行，在合成最终的生成器模型的损失时，分别为模拟标签的分类损失和真实图片的重建损失添加了 10 和 100 的缩放参数。这样做是为了使损失处于同一数量级。类似的还有代码第 110 行，合成判别器模型的损失部分。



提示：

在 AttGAN 的论文 (<https://arxiv.org/pdf/1711.10678.pdf>) 中，作者对重建损失、分类损失与对抗损失分别做了单独的实验，从而总结各个损失值对模型的约束意义，以便更好地理解模型内部的机制。具体如下：

- 重建损失是为了表示属性以外的信息，可以保证与属性无关的人脸部分不被改变。
- 分类损失是为了表示属性信息，使生成器模型能够按照指定的属性来生成图片。
- 对抗损失是为了强化生成器模型的属性生成功能，让属性信息可以显现出来。

如果没有对抗损失，生成器模型生成的图片会很不稳定，用肉眼看去，有的具有属性，有的却没有属性。但这并不代表生成器模型生成的图片没有对应的属性，只不过是人眼无法看出这些属性而已。这时生成器模型相当于一个用于攻击模型的对抗样本生成器模型，即生成具有人眼识别不出来的图片属性（具体参考第 11 章）。而对抗损失用真实的图片与标签进行校准，正好加固了生成器模型的分类生成功能，让生成器模型可以生成人眼可见的属性图片。

代码第 121 行用 `tf.losses.absolute_difference` 函数计算重建损失。该函数计算的是生成图片与原始图片的平均绝对误差 (MAD)。相对于 MSE 算法，平均绝对误差受偏离正常范围的离群样本影响较小，让模型具有更好的泛化性，可以更好地帮助模型在重建方面进行收敛。但缺点是收敛速度比 MSE 算法慢。

代码第 119 行，在计算分类损失时，使用了激活函数 sigmoid 的交叉熵函数 sigmoid_cross_entropy。sigmoid 的交叉熵是将预测值与标签值中的每个分类各做一次 sigmoid 变化，再计算交叉熵。这种方法常常用来解决非互斥类的分类问题。它不同于 softmax 的交叉熵：softmax 的交叉熵在 softmax 环节限定预测值中所有分类的概率值的“和”为 1，标签值中所有分类的概率值的“和”也为 1，这会导致概率值之间是互斥关系，所以 softmax 的交叉熵适用于互斥类的分类问题。

代码第 134~186 行实现了输出 summary 日志的功能。待模型训练结束之后，可以在 TensorBoard 中查看。

10.3.10 代码实现：训练模型

首先定义 3 个函数 immerge、to_range、imwrite，用在测试模型的输出图片环节。

接着通过循环迭代训练模型。在训练的过程中，每训练 5 次判别器模型，就训练一次生成器模型。具体代码如下：

代码 10-6 trainattgan（续）

```

204 def immerge(images, row, col):#合成图片
205     h, w = images.shape[1], images.shape[2]
206     if images.ndim == 4:
207         img = np.zeros((h * row, w * col, images.shape[3]))
208     elif images.ndim == 3:
209         img = np.zeros((h * row, w * col))
210     for idx, image in enumerate(images):
211         i = idx % col
212         j = idx // col
213         img[j * h:j * h + h, i * w:i * w + w, ...] = image
214
215     return img
216
217 #转化图片值域，从[-1.0, 1.0] 到 [min_value, max_value]
218 def to_range(images, min_value=0.0, max_value=1.0, dtype=None):
219
220     assert np.min(images) >= -1.0 - 1e-5 and np.max(images) <= 1.0 + 1e-5 \
221         and (images.dtype == np.float32 or images.dtype == np.float64), \
222         ('The input images should be float64(32) ' \
223          'and in the range of [-1.0, 1.0]!')
224     if dtype is None:
225         dtype = images.dtype
226     return ((images + 1.) / 2. * (max_value - min_value) +
227             min_value).astype(dtype)
228
229 def imwrite(image, path): #保存图片，数值为 [-1.0, 1.0]
230     if image.ndim == 3 and image.shape[2] == 1: #保存灰度图
231         image = np.array(image, copy=True)
232         image.shape = image.shape[0:2]
```

```

233     return scipy.misc.imsave(path, to_range(image, 0, 255, np.uint8))
234 #创建或加载模型
235 ckpt_dir = './output/%s/checkpoints' % experiment_name
236 try:
237     thisckpt_dir = tf.train.latest_checkpoint(ckpt_dir)
238     restorer = tf.train.Saver()
239     restorer.restore(sess, thisckpt_dir)
240     print(' [*] Loading checkpoint succeeds! Copy variables from %s!' % thisckpt_dir)
241 except:
242     print(' [*] No checkpoint')
243     os.makedirs(ckpt_dir, exist_ok=True)
244     sess.run(tf.global_variables_initializer())
245
246 #训练模型
247 try:
248     #计算训练一次数据集所需的迭代次数
249     it_per_epoch = len(tr_data) // (batch_size * (n_d + 1))
250     max_it = epoch * it_per_epoch
251     for it in range(sess.run(it_cnt), max_it):
252         start_time = time.time()
253         sess.run(update_cnt)           #更新计数器
254         epoch = it // it_per_epoch    #计算训练一次数据集所需要的迭代次数
255         it_in_epoch = it % it_per_epoch + 1
256         lr_ipt = lr_base / (10 ** (epoch // 100))    #计算学习率
257         for i in range(n_d):          #训练n_d次判别器模型
258             d_summary_opt, _ = sess.run([d_summary, d_step], feed_dict={lr: lr_ipt})
259             summary_writer.add_summary(d_summary_opt, it)
260             g_summary_opt, _ = sess.run([g_summary, g_step], feed_dict={lr: lr_ipt})      #训练一次生成器模型
261             summary_writer.add_summary(g_summary_opt, it)
262             if (it + 1) % 1 == 0:        #显示计算时间
263                 print("Epoch: {} {} time: {}".format(epoch, it_in_epoch,
264                     it_per_epoch, time.time() - start_time))
265             if (it + 1) % 1000 == 0:      #保存模型
266                 save_path = saver.save(sess, '%s/Epoch_(%d)_(%dof%d).ckpt' %
267 (ckpt_dir, epoch, it_in_epoch, it_per_epoch))
268                 print('Model is saved at %s!' % save_path)
269 #用模型生成一部分样本，以便观察效果
270 if (it + 1) % 100 == 0:
271     x_sample_opt_list = [xa_sample_ipt, np.full((n_sample, img_size,
272     img_size // 10, 3), -1.0)]
273     for i, b_sample_ipt in enumerate(b_sample_ipt_list):
274         b_sample_ipt = (b_sample_ipt * 2 - 1) * thres_int#标签预处理
275         if i > 0:    #将当前属性的值域变成 [-1, 1]。如果 i 为 0，则是原始标签

```

```

275         _b_sample_ipt[..., i - 1] = _b_sample_ipt[..., i - 1] *
276             test_int / thres_int
277         x_sample_opt_list.append(sess.run(x_sample,
278             feed_dict={xa_sample: xa_sample_ipt, _b_sample: _b_sample_ipt}))
279         sample = np.concatenate(x_sample_opt_list, 2)
280         save_dir = './output/%s/sample_training' % experiment_name
281         os.makedirs(save_dir, exist_ok=True)
282         imwrite(immerge(sample, n_sample, 1),
283             '%s/Epoch_(%d)_(%dof%d).jpg' % (save_dir, epoch, it_in_epoch,
284             it_per_epoch))
285     except:
286     traceback.print_exc()
287 finally: #在程序最后保存模型
288     save_path = saver.save(sess, '%s/Epoch_(%d)_(%dof%d).ckpt' % (ckpt_dir,
289     epoch, it_in_epoch, it_per_epoch))
290     print('Model is saved at %s!' % save_path)
291 sess.close()

```

代码运行后，输出如下结果：

```

.....
Epoch: 116 233/947 time: 10.196768760681152
Epoch: 116 234/947 time: 10.141278266906738
Epoch: 116 235/947 time: 10.229653596878052
Epoch: 116 236/947 time: 10.178789377212524
.....

```

结果中只显示了训练的进度和时间。内部的损失值可以通过TensorBoard来参看，如图 10-7 所示。

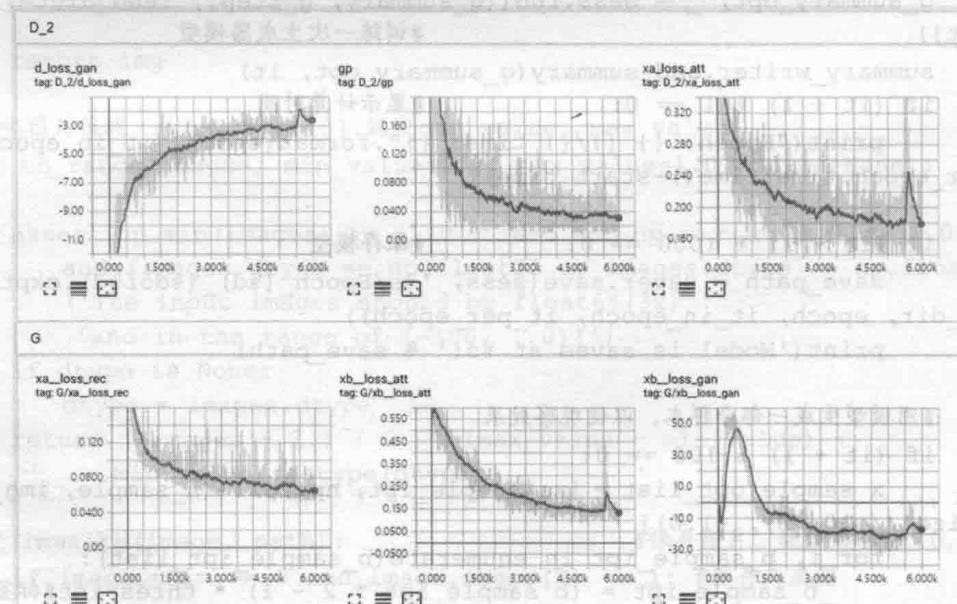


图 10-7 AttGAN 的损失值

在当前目录的 output\128_shortcut1_inject1_None\sample_training 文件下，可以看到生成的人脸图片情况，如图 10-8 所示。



图 10-8 AttGAN 所合成的人脸图片

图 10-8 中，每一行是具体图片按照指定属性生成的结果。其中，第 1 列为原始图片。第 2 列到最后 1 列是按照代码 66 行 b_sample_ipt_list 变量中的属性标签生成的，其中包括带有眼袋、头帘、黑头发、金色头发、棕色头发等属性的人脸图片。

代码第 274 行，在生成图片时，将每个用于显示图片主属性的值设为 1，高于训练时的特征最大值 0.5。这么做是为了让生成器模型生成特征更加明显的人脸图片。另外还可以通过该值的大小来调节属性的强弱，见 10.3.11 小节。

10.3.11 实例 56：为人脸添加不同的眼镜

在 AttGAN 模型中，每个属性都是通过数值大小来控制的。按照这个规则，可以通过调节某个单一的属性值，来实现在编辑人脸时某个属性显示的强弱。

下面通过编码来实现具体的实验效果。在定义参数、构建模型之后，按如下步骤实现：

- (1) 添加代码载入模型（见代码第 1~14 行）。
- (2) 设置图片的人脸属性及标签强弱（见代码第 16~19 行）。
- (3) 生成图片，并保存。

具体代码如下：

代码 10-7 testattgan（片段）

```

01 .....
02 ckpt_dir = './output/%s/checkpoints' % experiment_name
03 print(ckpt_dir)
04 thisckpt_dir = tf.train.latest_checkpoint(ckpt_dir)
05 print(thisckpt_dir)
06 restorer = tf.train.Saver()
07 restorer.restore(sess, thisckpt_dir)
08
09 try:                                #载入模型

```

```

10     thisckpt_dir = tf.train.latest_checkpoint(ckpt_dir)
11     restorer = tf.train.Saver()
12     restorer.restore(sess, thisckpt_dir)
13 except:
14     raise Exception(' [*] No checkpoint!')
15
16 n_slide = 10          #生成 10 个图片
17 test_int_min = 0.7    #特征值从 0.7 开始
18 test_int_max = 1.2    #特征值到 1.2 结束
19 test_att = 'Eyeglasses' #只使用一个眼镜属性
20 try:
21     for idx, batch in enumerate(te_data):#遍历样本数据
22         xa_sample_ipt = batch[0]
23         b_sample_ipt = batch[1]
24         #处理标签
25         x_sample_opt_list = [xa_sample_ipt, np.full((1, img_size, img_size // 10, 3), -1.0)]
26         for i in range(n_slide):#生成 10 个图片
27             test_int = (test_int_max - test_int_min) / (n_slide - 1) * i + test_int_min
28             _b_sample_ipt = (_b_sample_ipt * 2 - 1) * thres_int
29             _b_sample_ipt[..., att_default.index(test_att)] = test_int
30             #用模型生成图片
31             x_sample_opt_list.append(sess.run(x_sample,
32             feed_dict={xa_sample: xa_sample_ipt, _b_sample: _b_sample_ipt}))
33             sample = np.concatenate(x_sample_opt_list, 2)
34             #保存结果
35             save_dir = './output/%s/sample_testing_slide_%s' % (experiment_name,
36             test_att)
37             os.makedirs(save_dir, exist_ok=True)
38             imwrite(sample.squeeze(0), '%s/%d.png' % (save_dir, idx + 182638))
39             print('%d.png done!' % (idx + 182638))
40     except:
41         traceback.print_exc()
42 finally:
43     sess.close()

```

代码运行后会看到，在本地 output\128_shortcut1_inject1_None\sample_testing_slide_Eyeglasses 文件夹下生成了若干图片。以其中的一个为例，如图 10-9 所示。



图 10-9 带有不同眼镜的人脸图片

可以看到，从左到右眼镜的颜色在变深、变大，这表示 AttGAN 模型已经能够学到眼镜属

性在人脸中的特征分布情况。根据眼镜属性值的大小不同，生成的眼镜的风格也不同。

10.3.12 扩展：AttGAN 模型的局限性

看似强大的 AttGAN 模型也有它的短板。AttGAN 模型的作者在用 AttGAN 模型处理跨域较大的风格转化任务时（例如，将现实图片转换成油画风格），发现效果并不理想。这表明 AttGAN 模型适用于图片纹理变化相对较小的图片风格转换任务（例如，根据风景图片生成四季的效果），但不适用于纹理或颜色变化较大的图片转换任务。

这是因为，AttGAN 模型更侧重于单个样本的生成，即对单个样本进行微小改变。所以，该模型在批量数据上的风格改变效果并不优秀。在实际应用中，读者应根据具体的问题选择合适的模型。

10.4 实例 57：用 RNN.WGAN 模型模拟生成恶意请求

实例描述

从网络中获取到一部分恶意请求数据。用该数据来训练 RNN.WGAN 模型，让模型可以拟合现有样本的特征，并模拟生成相似的恶意请求数据。

该实例源于网络安全领域中的一个真实场景。在用有监督的训练方式训练模型时，通常需要很大的样本量。然而准备大量带有标注的样本并不是一件很容易的事。在有限的样本下，如想实现用海量数据训练出来的模型效果，则可以用生成式网络模型模拟出更多的样本，以扩充训练数据集。

10.4.1 获取样本：通过 Panabit 设备获取恶意请求样本

本案例使用的数据集来自 Panabit 设备。该设备的主要功能是：对网络流量进行控制、管控 DNS、优化网络效率。它在识别网络应用的基础上，还实现了智选路由、负载均衡、二级路由、移动设备等功能。

在深度学习中，可以从该设备源源不断地取出数据，用于训练。下面简单介绍一下从 Panabit 设备获取数据的方法。

1. 创建自己的 Panabit 设备

Panabit 官网提供了一个可以独立安装的免费软件。可以将其安装在 PC 上，以实现 Panabit 设备同等的能力。对于手里没有 Panabit 设备但也想得到网络实时数据的读者，可以按本节的方法自己动手创建一台 Panabit 设备。

（1）Panabit 的安装包。

Panabit 有两部分构成：Panabit 系统、Panalog 软件。

- Panabit 系统：该安装包是一个 IOS 格式的镜像文件。其安装方法与操作系统的安装方法类似。需要制作 U 盘的启动盘，并从 U 盘启动进行安装。

- Panalog 软件：是在 Panabit 系统中安装的一个软件，可以实现日志收集、保存安全防护、态势感知，以及大数据分析等功能。

(2) Panabit 的安装方法。

Panabit 软件的具体下载路径及安装方式可以从如下链接中获得：

<http://forum.panabit.com/>

该网站是一个技术论坛。论坛中的“Panabit 路由流控”和“Panalog 日志分析”板块中有详细的安装教程，如图 10-10 所示。



图 10-10 Panabit 论坛

在“Panabit 教程和 Tips”板块中，介绍了 Panabit 软件的安装方法。在“PanaLog 教程和 Tips”板块中，介绍了如何安装 PanaLog，以及导出网络日志的方法。



提示：

Panabit 作为一款网络管理软件，要求本机至少配有 3 张或以上 Intel 1000M 网卡：一个用于管理，另外两个用作网桥。

按照教程中的方法将 Panabit 软件安装好之后，直接串联到自己的内网出口，便可以创建一台自己的 Panabit 设备。

2. 从 Panabit 设备中导出样本

在 Panabit 软件安装完成之后，可以用以下步骤导出样本。

(1) 打开 Panabit 日志分析软件，选择“用户行为”界面下的“URL 查询”，然后根据访问方式、目的 IP 地址、域名关键词、URL 关键词、协议等信息进行具体查询。

例如，查询 IP 地址为 100.64.160.209 的主机以 GET 方式访问的 URL 信息，则可以直接在界面中输入 IP 地址，如图 10-11 所示。

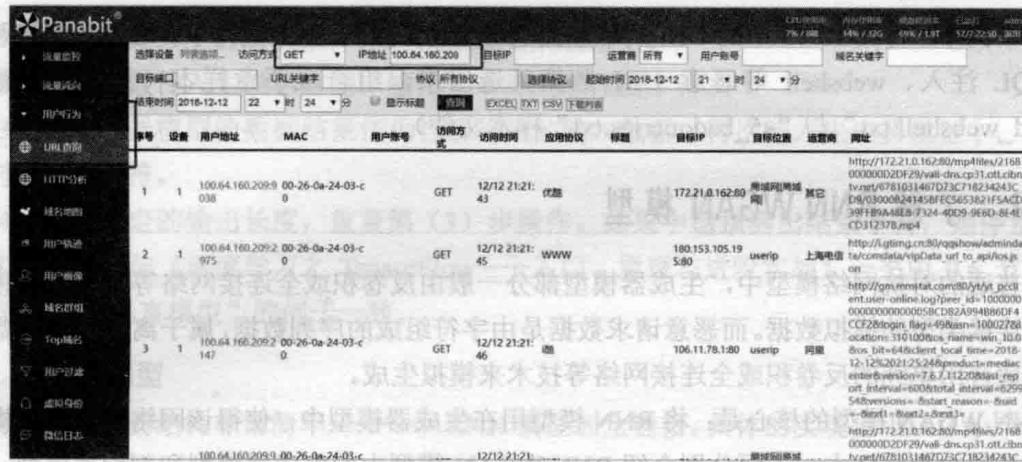


图 10-11 Panalog 界面

(2) 根据以上检索条件查询到的 URL 信息，可以按照 EXCEL、TXT、CSV 等格式生成报表，并提供下载，如图 10-12 所示。

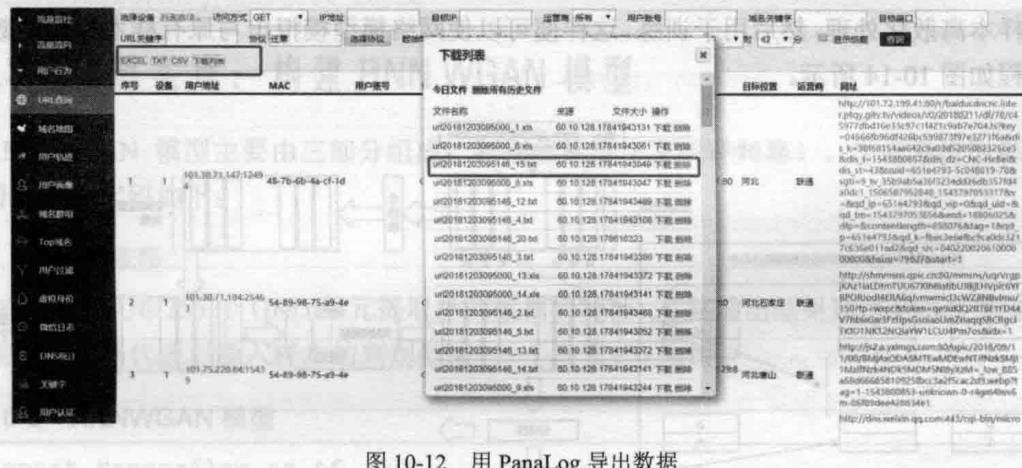


图 10-12 用 PanaLog 导出数据

(3) 下载后的 Excel 文档包含 URL 的所有关键信息，包括目标 IP、域名、URL、协议类型、所属运营商、所属地区等，如图 10-13 所示。

图 10-13 查看 PanaLog 中导出的数据

在本例中，将图 10-13 中的 URI 列单独提取出来，保存到 TXT 文件中。通过人工将跨站攻击、SQL 注入、webshell 等恶意 URI 数据挑选出来，组合成恶意样本（见随书资源中的“s2_bad_webshell.txt”与“s5_badqueries.txt”样本文件）。

10.4.2 了解 RNN.WGAN 模型

在普通的对抗网络模型中，生成器模型部分一般由反卷积或全连接网络等组成。该模型擅长生成连续类型的模拟数据。而恶意请求数据是由字符组成的序列数据，属于离散类型的数据，所以不适合用单纯的反卷积或全连接网络等技术来模拟生成。

RNN.WGAN 模型的核心是：将 RNN 模型用在生成器模型中，使得该网络模型可以模拟生成离散类型的数据样本。下面分别介绍 RNN.WGAN 模型中的生成器模型和判别器模型，以及训练方式。

1. 生成器模型

在 RNN.WGAN 模型中，生成器模型部分把随机数与真实样本放在一起，并对其做了基于序列的样本离散化处理，然后用于训练。这样便可以使网络模型模拟出与原有样本相似的数据。具体过程如图 10-14 所示。

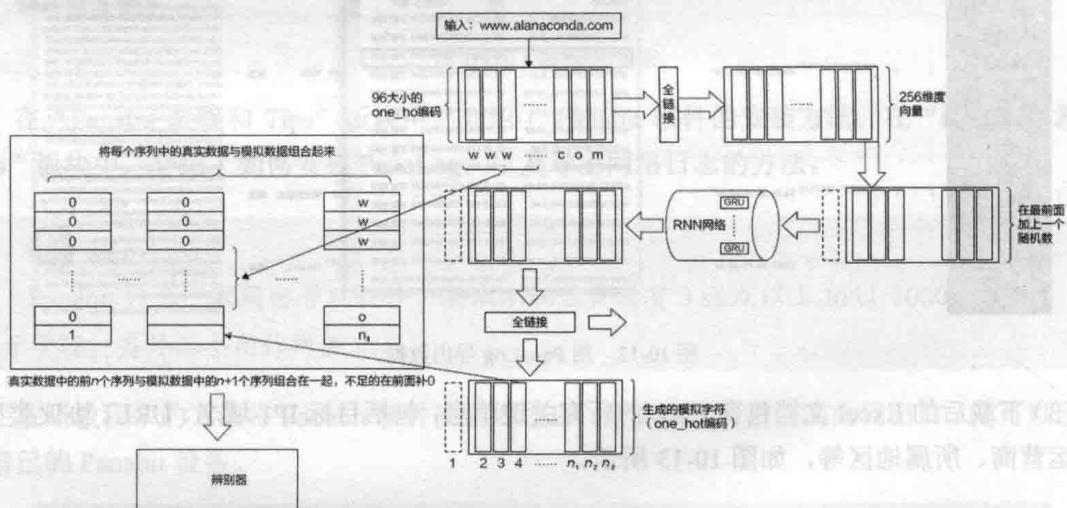


图 10-14 RNN.WGAN 的流程

如图 10-14 所示，生 RNN.WGAN 的成器模型的处理过程大体可以分为以下步骤：

- (1) 将 one_hot 编码的输入字符通过全链接转成与 RNN 模型对应维度的向量序列数据。
- (2) 将生成的随机数作为生成的向量序列数据的第一个序列，并一起放到 RNN 模型中。
- (3) 将 RNN 模型输出的结果再通过一次全链接，转成 one_hot 编码的模拟字符向量。
- (4) 将模拟字符与输入字符混合起来，作为训练用的生成器模型的输出样本。模拟字符中的每一个序列数据，都作为生成样本中的最后一个数据。该序列前面的数据使用输入字符的数据，不足的地方用 0 填充。

在生成器模型训练好之后，用该模型生成模拟数据的步骤如下：

- (1) 生成指定维度的随机数，作为模拟数据的起始值。
 - (2) 将随机数传入生成器模型，输出下一时刻的预测字符。
 - (3) 将生成器模型的数据结果作为当前时刻的输入数据，再次输入生成器模型中，得到下一时刻的预测字符。
 - (4) 按照指定的输出长度，重复第(3)步操作。如果中途预测出结束字符，则停止循环。
- 这部分内容与《深度学习之 TensorFlow——入门、原理与进阶实战》一书中 9.6 节“利用 RNN 模型训练语言模型”的例子一致。

2. 判别器模型

判别器模型的结构非常简单——一个 RNN 模型加全链接。具体的实现见 10.4.3 小节的详细代码。

3. 训练方式

RNN.WGAN 模型使用了 WGAN 模型的方法进行训练。详细做法可以参考如下论文：

<https://arxiv.org/abs/1704.00028>

10.4.3 代码实现：构建 RNN.WGAN 模型

RNN.WGAN 模型主要由三部分组成：判别器模型（又叫评判器）、生成器模型和反向传播部分。具体代码如下：

1. 判别器模型

判别器模型用 256 个 GRU 单元提取序列数据的特征，并将输出结果通过全链接网络生成 1 维数据。该数据代表对输入样本的判别结果——1 为真、0 为假。

代码 10-8 RNNWGAN 模型

```

01 import tensorflow as tf
02 from tensorflow.contrib.rnn import GRUCell
03
04 # 定义网络参数
05 DISC_STATE_SIZE = 256          # 定义判别器模型中 RNN 模型 cell 节点的个数
06 GEN_STATE_SIZE = 256          # 定义生成器模型中 RNN 模型 cell 节点的个数
07 GEN_RNN_LAYERS = 1            # 生成器模型中 RNN 模型 cell 的层数
08 LAMBDA = 10.0                # 惩罚参数
09
10 # 定义判别器模型函数
11 def Discriminator_RNN (inputs, charmap_len, seq_len, reuse=False, rnn_cell
12 =None):
13     with tf.variable_scope("Discriminator", reuse=reuse):
14         flat_inputs = tf.reshape(inputs, [-1, charmap_len])
15         weight = tf.get_variable("embedding", shape=[charmap_len,
DISC_STATE_SIZE],

```

```

16     initializer=tf.random_uniform_initializer(minval=-0.1,
17                                                 maxval=0.1))
18     #通过全链接转成与 RNN 模型同样维度的向量
19     inputs = tf.reshape(flat_inputs@weight, [-1, seq_len,
20                                               DISC_STATE_SIZE])
21     inputs = tf.unstack(tf.transpose(inputs, [1, 0, 2]))
22     #输入 RNN
23     cell = rnn_cell(DISC_STATE_SIZE)
24     output, state =
25     tf.contrib.rnn.static_rnn(cell, inputs, dtype=tf.float32)
26     weight = tf.get_variable("W", shape=[DISC_STATE_SIZE, 1],
27                             initializer=tf.random_uniform_initializer(minval=-0.1, maxval=0.1))
28     bias = tf.get_variable("b", shape=[1],
29                           initializer=tf.random_uniform_initializer(minval=-0.1, maxval=0.1))
28     #通过全链接网络生成判别结果
29     prediction = output[-1]@weight + bias
30
31     return prediction

```

2. 生成器模型

在生成器模型函数 Generator_RNN 中实现的步骤如下：

- (1) 定义内置函数 get_noise 和 create_initial_states，用于初始化 RNN 的状态值。
 - (2) 搭建生成器的主体结构。
 - (3) 根据传入的参数 gt 来选择内部所运行的代码分支是运行训练代码，还是运行评估代码。参数 gt 代表所传入的真实样本。如果参数 gt 有值，则运行训练代码，否则运行评估代码。
- 具体代码如下：

代码 10-8 RNNWGAN 模型（续）

```

32 #定义生成器模型函数
33 def Generator_RNN (n_samples, charmap_len,
34                      BATCH_SIZE,LIMIT_BATCH,seq_len=None, gt=None, rnn_cell=None):
35     def get_noise(BATCH_SIZE):#生成随机数
36         noise_shape = [BATCH_SIZE, GEN_STATE_SIZE]
37         return tf.random_normal(noise_shape,mean = 0.0, stddev=10.0),
38     noise_shape
39     def create_initial_states(noise):
40         states = []
41         for l in range(GEN_RNN_LAYERS):
42             states.append(noise)
43         return states
44     with tf.variable_scope("Generator"):

```

```

45     sm_weight = tf.Variable(tf.random_uniform([GEN_STATE_SIZE,
46                                         charmap_len], minval=-0.1, maxval=0.1))
47     sm_bias = tf.Variable(tf.random_uniform([charmap_len], minval=-0.1,
48                                         maxval=0.1))
49
50     #获得生成器模型的原始随机数
51     char_input = tf.Variable(tf.random_uniform([GEN_STATE_SIZE],
52                                         minval=-0.1, maxval=0.1))
53     #转成一批次的原始随机数据
54     char_input = tf.reshape(tf.tile(char_input, [n_samples]), [n_samples,
55                                         1, GEN_STATE_SIZE])
56
57     cells = []
58     for l in range(GEN_RNN_LAYERS):
59         cells.append(rnn_cell(GEN_STATE_SIZE))
60
61     if seq_len is None:
62         seq_len = tf.placeholder(tf.int32, None,
63                                name="ground_truth_sequence_length")
64
65     #初始化RNN模型的states
66     noise, noise_shape = get_noise(BATCH_SIZE)
67     train_initial_states = create_initial_states(noise)
68     inference_initial_states = create_initial_states(noise)
69     if gt is not None:          #如果GT不为None，则表示当前为训练状态
70         train_pred = get_train_op(cells, char_input, charmap_len,
71                                   embedding, gt, n_samples, GEN_STATE_SIZE, seq_len, sm_bias, sm_weight,
72                                   train_initial_states, BATCH_SIZE, LIMIT_BATCH)
73         inference_op = get_inference_op(cells, char_input, embedding,
74                                         seq_len, sm_bias, sm_weight, inference_initial_states,
75                                         GEN_STATE_SIZE, charmap_len, BATCH_SIZE, reuse=True)
76     else:                      #如果GT为None，则表示当前为eval状态
77         inference_op = get_inference_op(cells, char_input, embedding,
78                                         seq_len, sm_bias, sm_weight, inference_initial_states,
79                                         GEN_STATE_SIZE, charmap_len, BATCH_SIZE, reuse=False)
80
81     train_pred = None
82
83     return train_pred, inference_op
84
85
86 #生成用于训练的模拟样本
87 def get_train_op(cells, char_input, charmap_len, embedding, gt, n_samples,
88                  num_neurons, seq_len, sm_bias, sm_weight, states, BATCH_SIZE, LIMIT_BATCH):
89     gt_embedding = tf.reshape(gt, [n_samples * seq_len, charmap_len])
90     gt_RNN_input = gt_embedding@embedding

```

```

80     gt_RNN_input = tf.reshape(gt_RNN_input, [n_samples, seq_len,
81         num_neurons])[:, :-1]          ((1.0=levnmo ,1.0=levnmo ,not carried
82     gt_sentence_input = tf.concat([char_input, gt_RNN_input], axis=1) #gt_sentence_input 的 shape[n_samples, seq_len+1, num_neurons]
83     RNN_output, _ = rnn_step_prediction(cells, charmap_len,
84         gt_sentence_input, num_neurons, seq_len, sm_bias, sm_weight,
85         states, BATCH_SIZE)
86     train_pred = []
87     #从 seq_len+1 中取出前 seq_len 个特征，每一个生成的特征都与原来的输入重新组成一个
88     #序列
89     for i in range(seq_len):
90         train_pred.append( #每个序列特征前面加 0 数据，前 i-1 行数据
91             tf.concat([tf.zeros([BATCH_SIZE, seq_len - i - 1, charmap_len]),
92             gt[:, :i], RNN_output[:, i:i + 1, :]], axis=1))
93     train_pred = tf.reshape(train_pred, [BATCH_SIZE * seq_len, seq_len,
94         charmap_len])
95     prediction = output[-1].weight * bias + sm_bias
96     if LIMIT_BATCH:#从 BATCH_SIZE*seq_len 个序列中随机取出 BATCH_SIZE 个样本进行
97     #判断
98     indices = tf.random_uniform([BATCH_SIZE], 0, BATCH_SIZE * seq_len,
99         dtype=tf.int32) #获得随机索引
100    train_pred = tf.gather(train_pred, indices) #按照随机索引取数据
101
102    return train_pred
103
104
105
106
107
108
109
110
111
112
113
114
# 定义模型函数，通过 RNN 对数据进行特征分析
def rnn_step_prediction (cells, charmap_len, gt_sentence_input, num_neurons,
    seq_len, sm_bias, sm_weight, states,BATCH_SIZE,
    reuse=False):
    with tf.variable_scope("rnn", reuse=reuse):
        RNN_output = gt_sentence_input
        for l in range(GEN_RNN_LAYERS):
            RNN_output, states[l] = tf.nn.dynamic_rnn(cells[l], RNN_output,
                dtype=tf.float32,
                initial_state=states[l], scope="layer_%d" % (l + 1))
        RNN_output = tf.reshape(RNN_output, [-1, num_neurons])
        RNN_output = tf.nn.softmax(RNN_output@sm_weight + sm_bias)
        RNN_output = tf.reshape(RNN_output, [BATCH_SIZE, -1, charmap_len])
    return RNN_output, states
# 模拟生成真实样本
def get_inference_op(cells, char_input, embedding, seq_len,asm_bias,
    sm_weight, states, num_neurons, charmap_len,BATCH_SIZE,
    reuse=False):
    inference_pred = []

```

```

115 if embedded_pred = [char_input] #第一个序列字符是随机生成的，后面的序列字符由 RNN
116 模型生成的，每个字符通过全链接转成与 RNN 模型匹配的向量，再输入 RNN 模型
117     for i in range(seq_len):
118         step_pred, states = rnn_step_prediction (cells, charmap_len,
119         tf.concat(embedded_pred, 1), num_neurons, seq_len, epsilon_dss_lls
120         sm_bias, sm_weight,
121         states, BATCH_SIZE, reuse=reuse)
122         best_chars_tensor = tf.argmax(step_pred, axis=2)
123         best_chars_one_hot_tensor = tf.one_hot(best_chars_tensor,
124         charmap_len)
125         best_char = best_chars_one_hot_tensor[:, -1, :]
126         inference_pred.append(tf.expand_dims(best_char, 1))
127         embedded_pred.append(tf.expand_dims(best_char@embedding, 1))
128         reuse = True #设置变量生成方式为 reuse
129
130     return tf.concat(inference_pred, axis=1)

```

3. 反向传播部分

该部分代码主要实现了 WGAN 模型的损失值计算。代码如下：

代码 10-8 RNNWGAN 模型（续）

```

127 #获得指定训练参数
128 def params_with_name(name):
129     return [p for p in tf.trainable_variables() if name in p.name]
130
131 def get_optimization_ops(disc_cost, gen_cost, global_step, gen_lr,
132     disc_lr):
133     gen_params = params_with_name('Generator')
134     disc_params = params_with_name('Discriminator')
135     print("Generator Params: %s" % gen_params)
136     print("Disc Params: %s" % disc_params)
137     gen_train_op = tf.train.AdamOptimizer(learning_rate=gen_lr, beta1=0.5,
138     beta2=0.9).minimize(gen_cost,
139     var_list=gen_params,
140     global_step=global_step)
141     disc_train_op = tf.train.AdamOptimizer(learning_rate=disc_lr, beta1=0.5,
142     beta2=0.9).minimize(disc_cost,
143     var_list=disc_params)
144 #将输入的序列数据打散。每一个序列作为一个样本
145 def get_substrings_from_gt(real_inputs, seq_length,
146     charmap_len, BATCH_SIZE, LIMIT_BATCH):
147     train_pred = []
148     for i in range(seq_length):
149         train_pred.append(

```

```

149     tf.concat([tf.zeros([BATCH_SIZE, seq_length - i - 1, charmap_len]),
150                 real_inputs[:, :, i + 1]], axis=1))
151
152     all_sub_strings = tf.reshape(train_pred, [BATCH_SIZE * seq_length,
153                                               seq_length, charmap_len])
154
155     if LIMIT_BATCH: #按照指定批次随机取值
156         indices = tf.random_uniform([BATCH_SIZE], 1,
157                                       all_sub_strings.get_shape()[0], dtype=tf.int32)
158         all_sub_strings = tf.gather(all_sub_strings, indices)
159     return all_sub_strings[:BATCH_SIZE]
160
161
162 def define_objective(charmap, real_inputs_discrete, seq_length, len,
163                      BATCH_SIZE, LIMIT_BATCH):
164
165     real_inputs = tf.one_hot(real_inputs_discrete, len(charmap))
166
167     train_pred, _ = Generator_RNN(BATCH_SIZE, len(charmap),
168                                   BATCH_SIZE, LIMIT_BATCH, seq_len=seq_length, gt=real_inputs,
169                                   rnn_cell=GRUCell)
170
171     #将输入 real_inputs 按照序列展开, 再随机取值
172     real_inputs_substrings = get_substrings_from_gt(real_inputs, seq_length,
173                                                    len(charmap), BATCH_SIZE, LIMIT_BATCH)
174
175     disc_real = Discriminator_RNN( real_inputs_substrings, len(charmap),
176                                    seq_length, reuse=False, rnn_cell=GRUCell)
177     disc_fake = Discriminator_RNN( train_pred, len(charmap), seq_length,
178                                    reuse=True, rnn_cell=GRUCell)
179
180     disc_cost, gen_cost = loss_d_g(disc_fake, disc_real, train_pred,
181                                     real_inputs_substrings, charmap, seq_length, Discriminator_RNN, GRUCell)
182
183     #计算 WGAN 模型的惩罚项
184     alpha = tf.random_uniform(

```

```

185     shape=[tf.shape(real_inputs)[0], 1, 1],
186     minval=0.,
187     maxval=1.
188 )
189 differences = fake_inputs - real_inputs
190 interpolates = real_inputs + (alpha * differences)
191 gradients = tf.gradients(Discriminator(interpolates, len(charmap),
192 seq_length, reuse=True, rnn_cell=GRUCell), [interpolates])[0]
193 slopes = tf.sqrt(tf.reduce_sum(tf.square(gradients),
194 reduction_indices=[1, 2]))
195 gradient_penalty = tf.reduce_mean((slopes - 1.) ** 2)
196 disc_cost += LAMBDA * gradient_penalty
197
198 feed_dict = {
199     ...
200 }
201
202 return disc_cost, gen_cost

```

10.4.4 代码实现：训练指定长度的 RNN.WGAN 模型

训练模型部分主要分为两步：样本处理与训练模型。

1. 样本处理

样本处理有如下步骤：

- (1) 将样本“no”文件夹放到本地代码的同级目录下。
- (2) 实现字典的生成（见本书配套资源中的代码文件“10-9 prepro.py”）。
- (3) 实现数据集的建立（见本书配套资源中的代码文件“10-10 mydataset.py”）。
- (4) 编写样本的处理函数（用于训练和测试）。

数据预处理函数 `getbatchdata` 用于训练。它主要是把数据集返回的批次字符数据转化成向量，并进行统一长度的填充。

样本数据处理函数 `generate_argmax_samples_and_gt_samples` 用于测试。它主要是生成模拟样本，并取出真实样本，方便在训练过程中对模型效果进行评估。

代码 10-11 train_a_sequence

```

01 import os
02 import time
03 import sys
04
05 sys.path.append(os.getcwd())
06
07 import numpy as np
08 import tensorflow as tf
09
10 model = __import__("10-8 RNNWGAN 模型")
11 prepro = __import__("10-9 prepro")
12 preprosample = prepro.preprosample
13 dataset = __import__("10-10 mydataset")
14 mydataset = dataset.mydataset

```

```

15
16 # 定义单个长度训练的相关参数
17 CRITIC_ITERS = 2      # 每次训练 GAN 中，迭代训练两次判别器模型
18 GEN_ITERS = 10        # 每次训练 GAN 中，迭代训练 10 次生成器模型
19 DISC_LR = 2e-4        # 定义判别器模型的学习率
20 GEN_LR = 1e-4         # 定义生成器模型的学习率
21
22 PRINT_ITERATION = 100 # 定义输出打印信息的迭代频率
23 SAVE_CHECKPOINTS_EVERY = 1000 # 定义保存检查点的迭代频率
24
25 LIMIT_BATCH = True    # 让生成器模型生成同批次的数据
26
27 # 样本数据预处理（用于训练）
28 def
29     getbatchdata(sess,dosample,next_element,words_redic,BATCH_SIZE,END_SEQ):
30         def getone():
31             batchx,batchlabel = sess.run(next_element)
32             batchx = dosample.ch_to_v([strname.decode() for strname in
33                                         batchx],words_redic,0)
34             batchlabel = np.asarray(batchlabel,np.int32)#no==0 yes==1
35             samplepad ,samplengths = dosample.pad_sequences(batchx,
36             maxlen=END_SEQ)#都填充为最大长度END_SEQ
37             return samplepad,batchlabel,samplengths
38         iii = 0
39         while np.shape(samplepad)[0] !=BATCH_SIZE: #取出不够批次的尾数据
40             iii=iii+1
41             tf.logging.warn("          iii %d"%iii)
42             samplepad,batchlabel,samplengths = getone()
43
44         samplepad = np.asarray(samplepad,np.int32)
45         return samplepad,batchlabel,samplengths
46
47 # 获得模拟样本和真实样本（用于测试）
48 def generate_argmax_samples_and_gt_samples(session, inv_charmap,
49     fake_inputs, disc_fake, _data, real_inputs_discrete, feed_gt=True):
50     scores = []
51     samples = []
52     samples_probabilites = []
53     for i in range(10):
54         argmax_samples, real_samples, samples_scores =
55         generate_samples(session, inv_charmap, fake_inputs, disc_fake,
56                         _data,
57                         real_inputs_discrete, feed_gt=feed_gt)
58         samples.extend(argmax_samples)
59         scores.extend(samples_scores)

```

```

56     samples_probabilites.extend(real_samples)
57     return samples, samples_probabilites, scores
58
59 #获得生成的模拟样本
60 def generate_samples(session, inv_charmap, fake_inputs, disc_fake, _data,
61     real_inputs_discrete, feed_gt=True):
62     if feed_gt:
63         f_dict = {real_inputs_discrete: _data}
64     else:
65         f_dict = {}
66     fake_samples, fake_scores = session.run([fake_inputs, disc_fake],
67     feed_dict=f_dict)
68     fake_scores = np.squeeze(fake_scores)
69     decoded_samples = decode_indices_to_string(np.argmax(fake_samples,
70     axis=2), inv_charmap)
71     return decoded_samples, fake_samples, fake_scores
72 #将向量转成字符
73 def decode_indices_to_string(samples, inv_charmap):
74     decoded_samples = []
75     for i in range(len(samples)):
76         decoded = []
77         for j in range(len(samples[i])):
78             decoded.append(inv_charmap[samples[i][j]])
79     strde = "".join(decoded)
80     decoded_samples.append(strde)
81
82 return decoded_samples

```

2. 完成训练流程

因为序列数据具有长度的属性，所以在训练模型时需要指定所训练数据的序列长度。这里用 run 函数中的参数 seq_length 来设置序列长度。具体代码如下：

代码 10-11 train_a_sequence（续）

```

83 def run(iterations, seq_length, is_first, BATCH_SIZE,
84     prev_seq_length, DATA_DIR, END_SEQ):
85     if len(DATA_DIR) == 0:
86         raise Exception('Please specify path to data directory in
87         single_length_train.py!')
88     dosample = preprosample()
89     inv_charmap, charmap = dosample.make_dictionary()
90     #获取数据
91     next_element = mydataset(DATA_DIR, BATCH_SIZE)

```



```

134      tf.logging.info("开始训练生成器模型 : 随机升序 2.4.0f")
135      #训练生成器模型
136      for i in range(GEN_ITERS):
137          _data,batchlabel,samplelengths
138          =getbatchdata(session,dosample,next_element,charmap,BATCH_SIZE,END_SEQ)
139          _data= _data[:,seq_length]
140          _gen_cost, _ = session.run([gen_cost, gen_train_op],
141          feed_dict={real_inputs_discrete: _data})
142          _gen_cost_list.append(_gen_cost)
143
144          _step_time_list.append(time.time() - start_time)
145          if iteration % PRINT_ITERATION == PRINT_ITERATION-1:
146              _data,batchlabel,samplelengths
147              =getbatchdata(session,dosample,next_element,charmap,BATCH_SIZE,END_SEQ)
148              _data= _data[:,seq_length]
149              tf.logging.info("iteration %s/%s" %(iteration, iterations))
150              tf.logging.info("disc cost {} gen cost {} average step time
151              {}".format( np.mean(_disc_cost_list), np.mean(_gen_cost_list),
152              np.mean(_step_time_list)))
153              _gen_cost_list, _disc_cost_list, _step_time_list = [],[],[]
154
155              fake_samples, samples_real_probabilites, fake_scores =
156              generate_argmax_samples_and_gt_samples(session, inv_charmap, fake_inputs,
157              disc_fake, _data, real_inputs_discrete,feed_gt=True)
158              print(fake_samples[:2], fake_scores[:2], iteration,
159              seq_length, "train")
160              print(decode_indices_to_string(_data[:2], inv_charmap),
161              real_scores[:2], iteration, seq_length, "gt")
162
163      #保存检查点
164      if iteration % SAVE_CHECKPOINTS_EVERY ==
165      SAVE_CHECKPOINTS_EVERY-1:
166          saver.save(session, checkpoint_dir+"/gan.cpkt",
167          global_step=iteration)
168
169      saver.save(session, checkpoint_dir+"/gan.cpkt",
170      global_step=iteration)
171      session.close()
172

```

在训练过程中，考虑到生成器模型相对于判别器模型收敛速度较慢，在训练过程中，以 2 次判别器模型的训练与 10 次生成器模型的训练为一组，进行多次迭代。

10.4.5 代码实现：用长度依次递增的方式训练模型

为了让生成式模型性能更稳定，可用长度依次递增的方式训练模型：

- (1) 让生成器模型输出的最大长度为 1，并按照指定迭代次数训练模型。
- (2) 在训练完成后，将生成器模型输出的最大长度加 1。
- (3) 加载上一次训练后的模型权重，然后按照指定迭代次数再次训练模型。
- (4) 通过循环来重复执行第(2)、(3)步，直到模型输出的最大长度达到最终长度的指定值（见代码第 21 行，最终长度设置为 256）。

具体代码如下：

代码 10-12 train_model

```

01 import tensorflow as tf
02 train_a_sequence = __import__("10-11_train_a_sequence")
03
04 tf.logging.set_verbosity(tf.logging.INFO)
05
06 # 定义相关参数
07 DATA_DIR = './no' # 定义载入的样本路径
08
09 TRAIN_FROM_CKPT = False # 是否从检查点开始训练
10
11 DYNAMIC_BATCH = False # 是否使用动态批次
12 BATCH_SIZE = 256 # 定义批次大小
13
14 SCHEDULE_ITERATIONS = True # 是否根据长度调整训练次数
15 SCHEDULE_MULT = 200 # 每个长度增加的训练次数
16 ITERATIONS_PER_SEQ_LENGTH = 2000 # 定义每个长度训练时的迭代次数
17
18 REAL_BATCH_SIZE = BATCH_SIZE
19
20 START_SEQ = 1 # 待训练的起始长度
21 END_SEQ = 256 # 最终长度
22
23 # 开始训练
24 stages = range(START_SEQ, END_SEQ)
25 printstr = '----Stages : ' + ' '.join(map(str, stages)) + "----"
26 tf.logging.info(printstr)
27
28 for i in range(len(stages)): # 从 START_SEQ 开始依次对每个长度的模型进行训练
29     prev_seq_length = stages[i-1] if i>0 else 0 # 定义变量，用于获得上次模型的
    路径名称
30     seq_length = stages[i]
31
32     printstr = "--Training on Seq Len = %d, BATCH SIZE: %d--" % (seq_length,
    BATCH_SIZE)

```

```

33     tf.logging.info(printstr)
34
35     tf.reset_default_graph()
36
37     if SCHEDULE_ITERATIONS:      #计算训练的迭代次数
38         iterations = min((seq_length + 1) * SCHEDULE_MULT,
39                           ITERATIONS_PER_SEQ_LENGTH)
40     else:
41         iterations = ITERATIONS_PER_SEQ_LENGTH
42
43     is_first = seq_length == stages[0] and not (TRAIN_FROM_CKPT)
44     #开始训练
45     train_a_sequence.run( iterations, seq_length, is_first, BATCH_SIZE ,
46                           prev_seq_length, DATA_DIR, END_SEQ )
47     if DYNAMIC_BATCH:
48         BATCH_SIZE = REAL_BATCH_SIZE / seq_length

```

可以看到，训练任务从标签 START_SEQ 一步一步地来到标签 END_SEQ。每增加一步，都会调用一次 run 函数进行训练。在 run 函数中，会将上一次的序列长度模型载入，并接着开始本次序列长度模型的训练。

由于刚开始的长度很短，不需要训练太多次数，所以这里采用了动态次数的设计。即在刚开始时，每增加一个长度，训练的次数增加 200 次（见代码第 38~41 行）。

10.4.6 运行代码

代码运行后，可以看到以下输出：

```

INFO:tensorflow:iteration 99/400
INFO:tensorflow:disc cost 0.9284327626228333 gen cost -0.14586441218852997 average
step time 0.27192285776138303
[ "'", '<' ] [0.100293025, 0.2281375] 99 1 train
[ '/', '/' ] [[0.807168 ]
[0.4004431]] 99 1 gt
INFO:tensorflow:iteration 199/400
INFO:tensorflow:disc cost 0.018268248066306114 gen cost -0.49142125248908997 average
step time 0.2696471452713013
[ '%', '%' ] [0.43393168, 0.43629712] 199 1 train
[ '?', '/' ] [[0.44106907]
[0.3808497 ]] 199 1 gt
.....

```

其中，倒数第 3 行是模型输出的测试结果，第 1 个['%', '%']是模型生成的模拟数据，后面的[0.43393168, 0.43629712]是判别器模型的输出结果，大于 0 的都是正确数据。倒数第 2 行是判别器模型对真实数据的判别结果。

10.4.7 扩展：模型的使用及优化

实例 57 是一个很通用的框架，可以仿照训练模型的代码使用模型。还可以在判别器模型和生成器模型的实现函数中，增加更深层复杂的网络结构。

1. 模型的应用

在收集样本困难的情况下，模型可以很好地补充现有数据集。例如：在训练结束之后，编写简单的代码调用生成器模型，让其生成如下模拟样本数据：

CALULNULULULUL0003C 0.897428
/eve/tyx.php?stuff="print;bsh3s 0.537700
/javascript/dmad.exe 0.844558
/eve/tyx.php?stuff="print;bsh3s 0.537700
??%u2216??%u2215??%u2215??%u2215??%u2215 0.552321
/top.exe 0.991244
<yvgrang="Ca:./winkodkes/search.mscripti 0.200083
%2e.0x2f..0x2f..0x2f..0x2f..0x2f..0x2f.. 0.791354
/main.php?stuff="print;bsh3s 0.598888
%2e.0x2f..0x2f..0x2f..0x2f..0x2f..0x2f.. 0.791354
< 1.046342
/ionstalsart/oounttaysart.chp 0.646257
'">><a href="x:x # 0.961426
/ionstalsart/oounttaysart.chp 0.646257
%2e.0x2f..0x2f..0x2f..0x2f..0x2f..0x2f.. 0.791354
%f0%80%80%80%80%80%80%80%80%80%80%80% -0.147070
<img srint(ber0=0003C 0.855602
/javascript/dmad.exe 0.844558
/eve/tyx.php?stuff="print;bsh3s 0.537700
%2e.0x2f..0x2f..0x2f..0x2f..0x2f..0x2f.. 0.791354
%5C..%35%63boot.ini 0.801754
%c0%fe%c0%fe%c0%fe%c0%fe%c0%fe%c0%fe%80% 0.384254
...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\...\\ 0.182383
/eve/tyx.php?stuff="print;bsh3s 0.537700
/od-wcyascriptionsbgis/iastovest.htm?<sc 0.240135
%uff0e%uff0x6e\\xa0bbsveep2endar aa aaaa 0.224024
N.../-ss.exe 0.963894
ssse\x09>q5968855#assue 0.828035

以上结果是用训练好的模型生成的模拟恶意域名请求数据。可以看到，这些都是很明显的 webshell 注入命令。使用这样的模型，可以非常方便地丰富数据集。

2. 模型的优化

该实例使用的 RNN 模型相对简单，意在提供一个对抗神经网络模型解决序列数据的思路，以及配合 RNN 模型的方法。可以在包含判别器模型和生成器模型的 RNN 模型中使用更为优秀的 cell 单元，例如 IndRNN 单元、JANENT 单元等。还可以使用更高级的网络结构，例如：多层 RNN、双向 RNN 等。

另外，还可以在判别器模型中加上分类层，将其改造成 GAN-cls 模型或 ACGAN 模型，让整个模型具备分类的功能（关于 GAN-cls 模型和 ACGAN 模型可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书中 12.7 节与 12.3.2 小节）。带有分类功能的判别器模型，可以直接用在恶意请求的识别分类任务中。

第 11 章 生成式模型——能够输出内容的模型

11.1 生成对抗网络

生成对抗网络（Generative Adversarial Network，GAN）是一种深度学习模型，由两个部分组成：生成器（Generator）和判别器（Discriminator）。生成器的任务是生成假数据，而判别器的任务是区分真假数据。通过不断地迭代训练，生成器逐渐学会生成更逼真的数据，而判别器则逐渐提高辨别能力，从而实现生成器生成的数据越来越逼真，判别器辨别越来越困难的目标。

生成对抗网络的应用非常广泛，例如生成艺术作品、图像修复、文本生成等。生成对抗网络在中国也有许多应用，例如语音合成、图像增强、自然语言处理等。生成对抗网络的原理非常复杂，但其核心思想是通过生成器和判别器的不断对抗，使得生成器能够生成逼真的数据，而判别器能够准确地识别出生成的数据。

11.1.3 什么是黑盒攻击

黑盒攻击

攻击模型的方法，本质上是“试探”着对网络来欺骗自己。所以要得逞，需要有相对应的模型文件。这样，如果想要黑盒攻击的话，就必须先将模型文件进行读取，然后根据对其进行攻击，从而达到攻击的目的。

在实际生活中，攻击者很难拿到被攻击模型的源代码或模型文件，如果想要对模型进行攻击，则需要使用黑盒攻击。

黑盒攻击是指攻击者无法直接观察到模型内部的结构和参数，只能通过输入输出的数据来推断模型的行为。本攻击更偏向于以猜、垫底山文本识别为主本特征，本特后果然输出敏感文本，且输出结果为乱码。本攻击的基本流程如下所示。
首先，攻击者向自己的网络，随机选择一个样本进行攻击。在攻击前，攻击者将长句输入到生成模型中，生成模型的对称文本。通过对比生成的对称文本和真实的长句，判断是否一致。如果一致，则认为攻击成功；如果生成的对称文本和真实的长句不一致，则认为攻击失败。

- 攻击者将一个长句输入到生成模型中，生成模型将长句输出为乱码。
- 攻击者将长句输入到识别模型中，识别模型将长句输出为乱码。
- 将生成的乱码转成正常文件，生成的文件名为“`乱码.txt`”。

在生成文件后，攻击者将生成的文件输入到识别模型中，识别模型将生成的文件输出为乱码。如果生成的乱码和输入的长句一致，则认为攻击成功；如果生成的乱码和输入的长句不一致，则认为攻击失败。

第 11 章

模型的攻与防——看似智能的AI也有脆弱的一面

在实际项目中，很多时候并不需要我们从头来开发或是训练模型，而是使用已有的模型进行改造。这样的模型实现方便，且性能稳定、可靠。

但是，原封不动地使用现成的模型，也会带来一定的安全隐患。了解深度学习的人只要稍加处理，便可以让模型失效。

另外，即使是自己原生开发的模型，在应用中也会因受到攻击而失效。模型的攻防技术，伴随着模型的发展也在不断地革新和进步。如果要将 AI 工程化，则必须了解这部分的知识。

本章讲解模型的攻防技巧。

11.1 快速导读

在学习实例之前，有必要了解一下模型攻防方面的基础知识。

11.1.1 什么是 FGSM 方法

攻击模型主要通过对抗样本来实现。对抗样本是一种看上去与真实样本一样，但又会使模型输出错误结果的样本。该样本主要用于攻击模型，所以又被叫作攻击样本。

FGSM (Fast Gradient Sign Method) 是一种生成对抗样本的方法。该方法的描述如下：

(1) 将输入图片当作训练的参数，使其在训练过程中可以被调整。

(2) 在训练时，通过损失函数诱导模型对图片生成错误的分类。

(3) 当多次迭代导致模型收敛后，训练出来的图片就是所要得到的对抗样本。

具体可以参考论文：

<https://arxiv.org/pdf/1607.02533.pdf>

11.1.2 什么是 cleverhans 模块

在 TensorFlow 中有一个子项目叫作 cleverhans。该项目可以被当作模块单独引入代码中。

cleverhans 模块中封装了多种生成对抗样本的方法和多种加固模型的方法。可以通过以下命

令安装 cleverhans 模块：

```
pip install cleverhans
```

在 <https://github.com/tensorflow/cleverhans> 中，有与 cleverhans 模块相关的教程、文档和代码实例。

该链接的示例代码中，提供了 TensorFlow（包括静态图和动态图）、Keras、Pytorch 相关的攻防代码。



提示：

cleverhans 模块的代码并不像 TensorFlow 的代码那样具有较好的向下兼容性。使用不同版本的 cleverhans 模块开发的代码，有可能互不兼容。

这会导致从 GitHub 网站上下载的 cleverhans 代码实例有可能无法在本机的 cleverhans 模块中成功运行。因为，使用 pip 命令安装的 cleverhans 发布版本往往要落后于 GitHub 网站上正在开发的 cleverhans 版本。

为了解决这种问题，可以单独把 GitHub 网站中的 cleverhans 源码下载下来，并将当前代码的工作区设为 cleverhans 源码所在的路径（具体操作参考 11.4 节），然后让示例代码优先加载源码中的库模块。

11.1.3 什么是黑箱攻击

攻击模型的方法，本质是“训练”神经网络来欺骗自己。所以前提是，需要有被攻击网络的模型文件。

在实际生活中，攻击者很难拿到被攻击模型的源代码或模型文件。如果想要对其进行攻击，则需要使用黑箱攻击技术。

黑箱攻击是指，在没有被攻击模型的源代码或模型文件的情况下制作出对抗样本，对目标模型进行攻击。

该方法的主要原理是，从表象上复制被攻击模型。即：利用探测目标网络模型的结果动向来镜像同步自己的网络，从而训练一个可以替代目标网络模型的被攻击模型；然后制作关于替代模型的对抗样本，通过攻击替代模型的方式来间接的攻击目标网络。

这种攻击技术适用于任何场景，没有范围限制。所产生的后果取决于被攻击网络所应用的场景，例如：

- 将“停车”路标伪造成一个绿灯，来欺骗自动驾驶汽车。容易引起车祸！
- 将指纹伪造成万能钥匙，使指纹锁失效。容易丢失财务！
- 将病毒伪造成正常文件，让网络防护失效。会导致病毒的恶意传播！



提示：

如果重要场景中的模型被攻击成功，则导致的后果将是灾难性的。

建议读者滥用该技术进行，以免承担不必要的道德或法律责任。

如果要想训练出一个健壮的模型，则必须要了解和掌握黑箱攻击技术。这样才能做到“知己知彼”。即在了解对手的攻击方式之上研究自己的防护手段，才会使模型更安全，使用起来更放心。

11.1.4 什么是基于雅可比矩阵的数据增强方法

基于雅可比 (Jacobian) 矩阵的数据增强方法，是一种常用的黑箱攻击方法。该方法可以快速构建出近似于被攻击模型的决策边界，从而使用最少量的输入样本。即：构建出代替模型，并进行后续的攻击操作。

详细请见如下链接：

<https://arxiv.org/abs/1602.02697>

1. 黑箱方式的攻击思路

黑箱方式的攻击思路如下：

- (1) 将收集好的样本送入被攻击模型里，得到标签。
- (2) 将样本与标签合成待训练的样本数据集。
- (3) 搭建一个具有同等功能的模型。
- (4) 使用第 (2) 步的样本数据集训练第 (3) 步的模型，得到替代模型。
- (5) 对替代模型进行攻击，获得对抗样本。

因为神经网络的能力与架构之间具有可转移性，所以从理论上说，在替代模型下生成的对抗样本同样也适应于被攻击的模型。

2. 黑箱方式的挑战

训练神经网络模型需要依赖大量的样本数据。数据收集问题是训练替代模型所面临的挑战。为了获取被攻击模型对应的样本，只能通过向模型输入数据并获得其返回结果的方式制作数据集。然而大量的访问行为很容易被屏蔽（一般的网络模型都会有攻击防护机制）。

3. 雅可比矩阵的数据增强方法的原理

雅可比矩阵的数据增强方法，主要用来解决训练替代模型中的数据收集问题。它是一种寻找输入样本的方法。通过少量的输入样本，即可试出目标模型的决策边界。通过决策边界的来制作样本，并训练出与目标模型相同决策边界的替代模型。

因为在攻击场景中，构建替代模型关注的是被攻击模型的决策边界，所以，只要替代模型与被攻击模型的决策边界拟合，即可在其基础之上进行攻击。

雅可比矩阵的启发式方法的主要作用是：在输入域进行探索，最小化地找到有用输入样本。这些样本沿着某个方向上的连续取值，并输入目标模型，可以快速找到能够使目标模型预测出不同标签的样本。

雅可比矩阵本质上是，函数的所有分量 (m 个) 对向量 x 的所有分量 (n 个) 的一阶偏导数组成的矩阵。该矩阵是对梯度的一种泛化。矩阵中每个导数的符号代表该输入点相对于

目标模型决策边界的方向（正向或负向）。

4. 实现雅可比矩阵的数据增强方法——构建 jacobian 矩阵图

实现雅可比矩阵的数据增强方法分为两步。

(1) 构建雅可比矩阵图：使用 cleverhans.attacks_tf 模块中的 jacobian_graph 函数。

(2) 数据增强算法的实现：使用 cleverhans.attacks_tf 模块中的 jacobian_augmentation 函数。

5. 构建雅可比矩阵图

jacobian_graph 函数可以构建一个关于输入 x 的导数列表。该列表用于后续的增强算法实现。

在该列表中，元素的个数是标签类别的个数。具体源码如下：

```
def jacobian_graph(predictions, x, nb_classes): #构建雅可比矩阵图
    list_derivatives = [] #存放导数的列表
    for class_ind in xrange(nb_classes): #计算每一类结果关于 x 的偏导数
        derivatives, = tf.gradients(predictions[:, class_ind], x)
        list_derivatives.append(derivatives) #将 TF 图形式的导数保存到列表中
    return list_derivatives #返回该列表，创建雅可比矩阵图完成
```

在 jacobian_graph 函数中，参数的含义如下：

- predictions 代表所构建替代模型的输出张量。
- x 代表输入。
- nb_classes 代表生成的标签个数（用于定义被攻击模型的策略边界）。

6. 数据增强算法的实现

jacobian_augmentation 函数的作用是，从传入的 jacobian 矩阵图中选出具体的输入样本 x。其内部的实现步骤如下：

(1) 选出一部分待输入的样本数据。

(2) 在原有梯度模型上，计算出该输入样本标签所对应的导数方向（通过取该导数符号 sign 的方式获得方向）。

(3) 在导数方向上对样本进行变化，公式是： $\lambda(x + \lambda \text{sign})$ 。

(4) 将变化后的结果作为下一次的输入样本。

该做法可以保证每次的样本选取都是有针对性的，即根据自身模型梯度来选取输入样本。

具体代码如下：

```
def jacobian_augmentation(sess, x, X_sub_prev, Y_sub, grads, lmbda,
                           feed=None):
    input_shape = list(x.get_shape()) #获取输入样本形状
    input_shape[0] = 1 #得到单个样本形状
    X_sub = np.vstack([X_sub_prev, X_sub_prev]) #复制一份输入样本
    for ind, prev_input in enumerate(X_sub_prev): #循环取出每个输入样本
        grad = grads[Y_sub[ind]] #从雅可比矩阵图中获取该样本的梯度
        feed_dict = {x: np.reshape(prev_input, input_shape)} #构造注入字典
        if feed is not None: #将额外的数据更新到字典里，用于注入
            feed_dict.update(feed)
```

```

grad_val = sess.run([tf.sign(grad)], feed_dict=feed_dict)[0] #获得样本的梯度
方向
X_sub[X_sub_prev.shape[0] + ind] = X_sub[ind] + lmbda * grad_val#构造输入样本
return X_sub #返回结果

```

在 jacobian_graph 函数中，各个参数的具体意义如下。

- sess：传入当前的会话。
- x：输入的占位符。
- X_sub_prev：用于输入模型的原始样本。
- Y_sub：原始样本所对应的标签。
- grads：存放每个决策边界的导数列表（即雅可比矩阵图）。
- lmbda：在更新输入样本时，让每个输入点沿着梯度方向所前进的步长 λ 。
- feed：在注入模型时，除样本数据外的其他输入。

函数 jacobian_graph 最终返回的结果 X_sub 包含两部分内容：原始的输入样本、构造好的输入样本。

在构造好输入样本之后，便可以将其输入被攻击模型中，从而得到对应的标签。接着就可以用这组数据（输入样本和标签）训练替代模型。将在该替代模型上做出的对抗样本放到被攻击模型中，也会起到一样的攻击效果。

11.1.5 什么是数据中毒攻击

在联网模式的模型应用中，常常会用再训练模式来应用模型，即模型在应用的过程中同步收集样本。收集到的样本又会自动用于模型再训练，训练好的模型在线继续提供服务。这一套流程全部自动化实现。

由于模型都是基于现有数据集进行训练的，没有人可以完全掌控结果数据的分布情况。所以，一旦结果数据的分布情况出现较大的变化，则直接影响模型的使用效果。这种部署方式，可以让模型一直随最新的数据分布来调整其拟合规则，从而实现与时俱进。

看似完美的流程，却忽略了一个致命的环节——来自未来的假数据。数据中毒的攻击方式正式利用了这一缺陷。它的原理是：

- (1) 伪造带有引导性的样本（构建大量带有负向样本特征的正向样本）。
- (2) 使用第(1)步的样本，利用僵尸网络之类的大规模数据源对模型发起攻击。
- (3) 大量的伪造样本会使模型的准确度降低，模型为了能够适应最新的数据会自动触发再训练行为。
- (4) 在模型启动再训练后，会把这些假样本当作真实环境下的数据来修正自己，最终使得模型的预测结果与真实数据差距越来越大。

典型的例子就是针对垃圾邮件识别模型的攻击。具体过程如下：

- (1) 攻击者伪造大量具有恶意邮件特征的真实邮件。
- (2) 用这些邮件在多个账户中频繁往来。
- (3) 反垃圾邮件模型会报出识别率下降的事件，从而触发再训练机制。

(4) 一旦训练好了之后，模型将丧失对垃圾邮件的识别功能。

基于再训练模式的部署，一定要对数据中毒这种情况加以防范。包括：

- 指定模型的回退机制。
- 利用其他版本的模型或算法来辅助告警机制。

一旦发生告警事件，还需要由人工来对样本进行抽样检查，以便核对所收集样本的真实性。

11.2 实例 58：用 FGSM 方法生成样本，并攻击 PNASNet 模型，让其将“狗”识别成“盘子”

实例描述

将一张哈士奇狗的照片输入 PNASNet 模型，观察其返回结果。

通过梯度下降算法训练一个模拟样本，让 PNASNet 模型对模拟样本识别错误：将“哈士奇”识别成“盘子”。

11.2.1 代码实现：创建 PNASNet 模型

代码文件“3-1 使用 AI 模型来识别图像.py”是一个用预训练模型 PNASNet 识别图片的例子。本实例基于它进行修改：

(1) 复制代码文件“3-1 使用 AI 模型来识别图像.py”所在的整个工作目录，并将代码文件“3-1 使用 AI 模型来识别图像.py”改名为“11-1 用梯度下降方法攻击 PNASNet 模型.py”。

(2) 在代码文件“11-1 用梯度下降方法攻击 PNASNet 模型.py”中编写代码，添加 pnasnetfun 函数，实现模型的创建。

完整的代码如下：

代码 11-1 用梯度下降方法攻击 PNASNet 模型

```

01 import sys # 初始化环境变量
02 nets_path = r'slim'
03 if nets_path not in sys.path:
04     sys.path.insert(0,nets_path)
05 else:
06     print('already add slim')
07
08 import tensorflow as tf # 引入模块
09 from nets.nasnet import pnasnet
10 import numpy as np
11 from tensorflow.python.keras.preprocessing import image
12
13 import matplotlib as mpl
14 import matplotlib.pyplot as plt
15 mpl.rcParams['font.sans-serif']=['SimHei'] # 用来正常显示中文标签
16 mpl.rcParams['font.family'] = 'STSong'
```

```

17 mpl.rcParams['font.size'] = 15
18
19 slim = tf.contrib.slim
20 arg_scope = tf.contrib.framework.arg_scope
21
22 tf.reset_default_graph()
23 image_size = pnasnet.build_pnasnet_large.default_image_size#获得图片的输入尺寸
24 LANG = 'ch' #使用中文标签
25
26 if LANG=='ch':
27     def getone(onestr):
28         return onestr.replace(',',' ').replace('\n','')
29
30     with open('中文标签.csv','r+') as f: #打开文件
31         labelnames =list( map(getone,list(f)) )
32         print(len(labelnames),type(labelnames),labelnames[:5])#输出中文标签
33     else:
34         from datasets import imagenet
35         labelnames = imagenet.create_readable_names_for_imagenet_labels() #获得数据集标签
36         print(len(labelnames),labelnames[:5]) #显示输出标签
37
38 def pnasnetfun(input_imgs,reuse ):
39     preprocessed = tf.subtract(tf.multiply(tf.expand_dims(input_imgs, 0),
40     2.0), 1.0)
41     arg_scope = pnasnet.pnasnet_large_arg_scope() #获得模型命名空间
42     with slim.arg_scope(arg_scope): #创建 PNASNet 模型
43         with slim.arg_scope([slim.conv2d,
44                               slim.batch_norm, slim.fully_connected,
45                               slim.separable_conv2d],reuse=reuse):
46             logits, end_points =
47             pnasnet.build_pnasnet_large(preprocessed,num_classes = 1001,
48             is_training=False)
49             prob = end_points['Predictions']
50     return logits, prob

```

代码第 13~17 行是对显示的图像进行设置，使其可以支持中文。

代码第 43 行用共享变量的方式复用 PNASNet 模型的权重参数。



提示：

在构建模型时，需要将其设为不可训练（见代码第 47 行）。这样才能保证，在后面 11.2.3 小节中通过训练生成对抗样本时 PNASNet 模型不会有变化。

11.2.2 代码实现：搭建输入层并载入图片，复现PNASNet模型的预测效果

在构建输入层时，用张量来代替占位符。该张量由函数 `tf.Variable` 定义，其用法与占位符的使用方式一样，同样支持静态图的注入机制。这么做是为了在制作对抗样本时，可以对其进行修改（因为张量支持修改操作，而占位符只能用作输入）。

在构建好输入层之后，便是载入图片、载入预训练模型、将图片注入预训练模型，并最终以可视化的形式输出预测结果。完整的代码如下：

代码 11-1 用梯度下降方法攻击 PNASNet 模型（续）

```

50 input_imgs = tf.Variable(tf.zeros((image_size, image_size, 3)))
51 logits, probs = pnasnetfun(input_imgs,reuse=False)
52 checkpoint_file = r'pnasnet-5_large_2017_12_13\model.ckpt' #定义模型路径
53 variables_to_restore = slim.get_variables_to_restore()
54 init_fn = slim.assign_from_checkpoint_fn(checkpoint_file,
   variables_to_restore,ignore_missing_vars=True)
55
56 sess = tf.InteractiveSession()          #建立会话
57 init_fn(sess)                         #载入模型
58
59 img_path = './dog.jpg'                #载入图片
60 img = image.load_img(img_path, target_size=(image_size, image_size))
61 img = (np.asarray(img) / 255.0).astype(np.float32)
62
63 def showresult(img,p):                 #定义函数，将模型输出结果可视化
64     fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 8))
65     fig.sca(ax1)
66
67     ax1.axis('off')
68     ax1.imshow(img)
69     fig.sca(ax1)
70
71     top10 = list((-p).argsort()[:10])
72     lab= [labelnames[i][:15] for i in top10]
73     topprobs = p[top10]
74     print(list(zip(top10,lab,topprobs)))
75
76     barlist = ax2.bar(range(10), topprobs)
77
78     barlist[0].set_color('g')
79     plt.sca(ax2)
80     plt.ylim([0, 1.1])
81     plt.xticks(range(10), lab, rotation='vertical')
82     fig.subplots_adjust(bottom=0.2)
83     plt.show()

```

```

84
85 p = sess.run(probs, feed_dict={input_imgs: img})[0]
86 showresult(img,p)

```

代码运行后，显示如下结果：

```

[(249, '爱斯基摩犬 哈士奇', 0.35189062), (251, '哈士奇', 0.34352344), (250, '雪橇犬 阿拉斯加爱斯基摩狗', 0.007250515), (271, '白狼 北极狼', 0.0034629034), (175, '挪威猪犬', 0.0028237076), (538, '狗拉雪橇', 0.0025286602), (270, '灰狼', 0.0022800271), (274, '澳洲野狗 澳大利亚野犬', 0.0018357899), (254, '巴辛吉狗', 0.0015468642), (280, '北极狐狸 白狐狸', 0.0009330675)]

```

并得到可视化图片，如图 11-1 所示。



图 11-1 PNASNet 模型输出

在图 11-1 中，左侧是输入的图片，右侧是预测的结果。可以看到，模型成功地预测出该图片的内容是一只哈士奇狗。

11.2.3 代码实现：调整参数，定义图片的变化范围

在制作样本时，不能让图片的变化太大，要让图片通过人眼看上去能够接收才行。这里需要手动设置阈值，限制图片的变化范围。然后将生成的图片显示出来，由人眼判断图片是否正常可用，以确保没有失真。

完整的代码如下：

代码 11-1 用梯度下降方法攻击 PNASNet 模型（续）

```

87 def floatarr_to_img(floatarr): #将浮点型数值转化为图片像素
88     floatarr=np.asarray(floatarr*255)
89     floatarr[floatarr>255]=255
90     floatarr[floatarr<0]=0

```

```

91     return floatarr.astype(np.uint8)
92
93 x = tf.placeholder(tf.float32, (image_size, image_size, 3)) # 定义占位符
94 assign_op = tf.assign(input_imgs, x) # 为 input_imgs 赋值
95 sess.run(assign_op, feed_dict={x: img})
96
97 below = input_imgs - 2.0/255.0 # 定义图片的变化范围
98 above = input_imgs + 2.0/255.0
99
100 belowv, abovev = sess.run([below, above]) # 生成阈值图片
101
102 plt.imshow(floatarr_to_img(belowv)) # 显示图片, 用于人眼验证
103 plt.show()
104 plt.imshow(floatarr_to_img(abovev))
105 plt.show()

```

代码第 94 行, 用 `tf.assign` 函数将输入图片 `x` 赋值给张量 `input_imgs`。因为输入层的张量 `input_imgs` 被定义之后一直没有被初始化, 所以该值必须在被赋值之后才可以使用。如果在使用时没有对其赋值, 则需要先用 `input_imgs.initializer` 函数将其初始化。

代码第 97、98 行, 将图片的变化范围设置为: 每个像素上下变化最大不超过 2。

代码运行后, 输出的图片如图 11-2、图 11-3 所示。

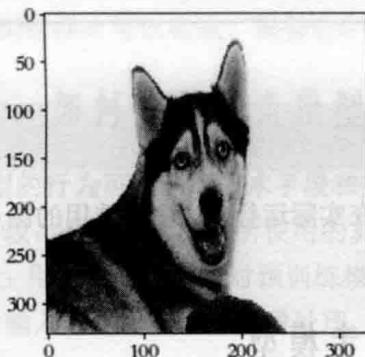


图 11-2 变化图片的阈值下限

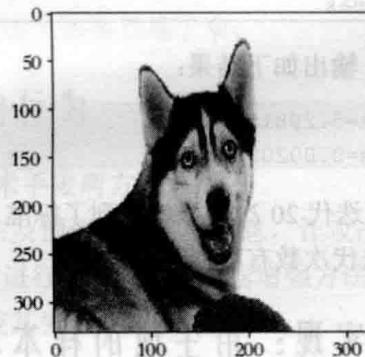


图 11-3 变化图片的阈值上限

如图 11-2、图 11-3 所示, 该图片完全在人眼接受范围。一眼看去, 就是一只哈士奇狗。

11.2.4 代码实现: 用梯度下降方式生成对抗样本

与正常的模型训练目标不同, 这里的训练目标不是模型中的权重, 而是输入模型的张量。在 11.2.1 小节中, 在生成模型的同时, 已经将模型固定好 (设为不可训练)。这样在训练模型的过程中, 反向梯度传播所修改的值就是输入的张量 `input_imgs`, 即最终的所要得到的对抗样本。

具体训练步骤: (1) 设定一个其他类别的标签; (2) 创建损失值 `loss` 节点, 使得每次优化时模型的输出都接近于设置的标签类别。

完整的代码如下:

代码 11-1 用梯度下降方法攻击 PNASNet 模型（续）

```

106 label_target = 880          #设定一个其他类别的目标标签
107 label = tf.constant(label_target)
108 #定义计算 loss 值的节点
109 loss      = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits,
110                 labels=[label])           #设置一个其他类别的目标标签
111 learning_rate = 1e-1          #定义学习率
112 optim_step = tf.train.GradientDescentOptimizer()    #定义优化器
113     .minimize(loss, var_list=[input_imgs])
114
115 projected = tf.clip_by_value(tf.clip_by_value(input_imgs, below, above), 0, 1)  #将调整后的图片按照指定阈值截断
116 with tf.control_dependencies([projected]):  #开始训练
117     project_step = tf.assign(input_imgs, projected)
118
119 demo_steps = 400             #定义迭代次数
120 for i in range(demo_steps):  #开始训练
121     _, loss_value = sess.run([optim_step, loss])  #输出训练状态
122     sess.run(project_step)
123
124     if (i+1) % 10 == 0:  #输出训练状态
125         print('step %d, loss=%g' % (i+1, loss_value))
126     if loss_value < 0.02:  #达到标准后提前结束
127         break

```

代码运行后，输出如下结果：

```

step 10, loss=5.29815
step 20, loss=0.00202808

```

代码显示，仅迭代 20 次模型就达到了标准。在实际运行中，由于选用的图片和指定的其他类别不同，所以迭代次数有可能不同。

11.2.5 代码实现：用生成的样本攻击模型

接下来就是最有意思的环节了——用训练好的对抗样本来攻击模型。

实现起来非常简单，直接将张量 input_imgs 作为输入，运行模型。具体代码如下：

代码 11-1 用梯度下降方法攻击 PNASNet 模型（续）

```

128 adv = input_imgs.eval()          #获取图片
129 p = sess.run(probs)[0]          #得到模型结果
130 showresult(floatarr_to_img(adv), p)  #可视化模型结果
131 plt.imsave('dog880.jpg', floatarr_to_img(adv))  #保存模型

```

代码运行后，得到如下结果：

```

[(880, '伞      ', 0.9981244), (930, '雪糕 冰棍 冰棒    ', 0.00011489283), (630, '唇膏 口
红      ', 8.979097e-05), (553, '女用长围巾      ', 4.4915465e-05), (615, '和服      ',
3.441378e-05), (729, '塑料袋      ', 3.353129e-05), (569, '裘皮大衣      ', 3.0552055e-05),

```

```
(904, '假发      ', 2.2152075e-05), (898, '洗衣机 自动洗衣机      ', 2.1341652e-05), (950, '草莓      ', 2.0412743e-05)]
```

并得到可视化图片，如图 11-4 所示。

11.3 实例 59：击破数据增强

攻与防是一对矛盾，而数据增强就是一种重要的缓解方法。攻与防两方面刚性对抗，是通过模型的训练来实现的，而柔性的对抗则是在模型的使用过程中实现的。本节将通过一个具体的例子，来说明如何通过对抗样本来突破数据增强。

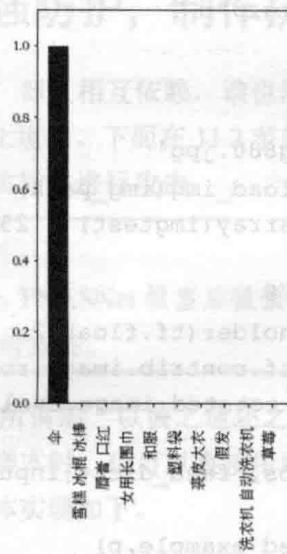


图 11-4 模型识别对抗样本的结果

从输出结果和图 11-4 可以看出，模型已经把哈士奇狗当成了伞。

11.2.6 扩展：如何防范攻击模型的行为

防范攻击模型的行为可以从非技术手段和技术手段两方面入手。

- 非技术手段：严格保密模型所使用的算法及预训练文件的信息，让攻击者无从下手。
- 技术手段：用数据增强方式对预训练模型进行微调，并将数据增强方法同步施加于应用场景。对输入的对抗样本做混淆处理，使其失效。

11.2.7 代码实现：将数据增强方式用在使用场景，以加固 PNASNet 模型，防范攻击

本实例所使用的预训练模型文件“pnasnet-5_large_2017_12_13”是 PNASNet 模型在 ImageNet 数据集上训练出来的。从 slim 模块的代码中可以看到，该预训练模型在训练过程中使用了数据增强方法，即 ImageNet 数据集中的图片在进行翻转、旋转、明暗变化后都能够被正确识别。

这里以图片的旋转操作举例。输入 11.2.5 小节生成的对抗样本 dog880.jpg，并对输入图片进行旋转，然后输入模型中，让对抗样本失效。具体代码如下：

代码 11-2 用数据增强抗攻击

01

```

02 def pnasnetfun(input_imgs,reuse):
03     .....
04     return logits, prob
05 .....
06 def showresult(img,p):
07     .....
08     plt.show()
09
10 img_path = './dog880.jpg' #载入图片
11 imgtest = image.load_img(img_path, target_size=(image_size, image_size))
12 imgtest = (np.asarray(imgtest) / 255.0).astype(np.float32)
13
14 ex_angle = np.pi/8 #对图片进行旋转
15 angle = tf.placeholder(tf.float32, ())
16 rotated_image = tf.contrib.image.rotate(input_imgs, angle)
17 rotated_example = rotated_image.eval(feed_dict={input_imgs: imgtest, angle: ex_angle})
18 p = sess.run(probs, feed_dict={input_imgs: rotated_example})[0]#对旋转后的图片进行预测
19 showresult(rotated_example,p) #输出结果

```

前半部分代码来自于代码文件“11-1 用梯度下降方法攻击 PNASNet 模型.py”中的代码，这里直接略过。

代码第 10 行，载入对抗样本 dog880.jpg，接着将其旋转（见代码第 14~17 行）。最后输入模型中并显示结果（见代码第 18、19 行）。代码运行后输出以下文件并输出可视化图片（如图 11-5 所示）。

```

[(251, '哈士奇', 0.43900266), (249, '爱斯基摩犬 哈士奇', 0.26704812), (250, '雪橇犬 阿拉斯加爱斯基摩狗', 0.0077105653), (538, '狗拉雪橇', 0.0028206212), (271, '白狼 北极狼', 0.00260258), (175, '挪威猎犬', 0.0025108932), (254, '巴辛吉狗', 0.0018674432), (274, '澳洲野狗 澳大利亚野犬', 0.0018472295), (270, '灰狼', 0.0017202504), (224, '舒柏奇犬', 0.0011743238)]

```

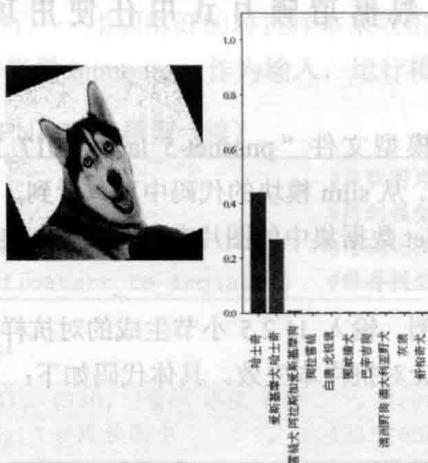


图 11-5 模型识别对抗样本的结果

从结果可以看到，将对抗样本进行旋转再输入模型得到了正确的结果。旋转输入图片的方式使得对抗样本失效，起到了加固模型的作用。

11.3 实例 59：击破数据增强防护，制作抗旋转对抗样本

攻与防是一对矛盾，互相衍化，相互制约，却又相互依赖，谁也消灭不了谁。在模型的攻防领域中，攻与防两方面的技术发展也是永无止境的。下面在 11.2 节的基础上，再介绍一种效果更好的攻击模型方法：通过制作抗旋转的对抗样本进行攻击。

实例描述

对一张哈士奇狗的照片进行处理，让其输入 PNASNet 模型后被预测成为盘子。并且，无论如何旋转图片，PNASNet 模型都无法输出正确的结果。

该实例的原理是典型的攻防博弈实例，即所谓的“以彼之道还之彼身”。既然 PNASNet 模型使用数据增强进行防守，那么在生成对抗样本时，也可以直接用数据增强方法进行生成。同样，也是以数据增强中的旋转操作为例，具体实现如下。

11.3.1 代码实现：对输入的数据进行多次旋转

为了覆盖所有角度的应用场景，所以在训练时，需要对图片进行随机角度的旋转。将一张图片变成多张旋转后的图片，进行批次输入。

在实现时，还需要将 11.2 节的单张处理改成批次处理。具体的代码如下：

代码 11-3 制造鲁棒性更好的对抗样本

```

01 .....  

02     print(len(labelnames),labelnames[:5])          #显示输出标签  

03  

04 batchsize=4                                     #定义批次数据为4  

05 .....  

06 def pnasnetfunrotate(input_imgs,reuse ):       #创建带数据增强的模型  

07     rotatedarr = []                                #存放旋转样本  

08     for i in range(batchsize):                     #按照指定批次进行旋转  

09         rotated = tf.contrib.image.rotate(input_imgs,  

10             tf.random_uniform(() , minval=-np.pi/4,  

11             maxval=np.pi/4))  

12             rotatedarr.append(tf.reshape(rotated,[1,image_size,image_size,3]))  

13     inputarr = tf.concat(rotatedarr, axis = 0)      #组合样本  

14     preprocessed = tf.subtract(tf.multiply(inputarr, 2.0), 1.0) #2  

15     *( input_imgs / 255.0)-1.0                      #样本预处理  

16     arg_scope = pnasnet.pnasnet_large_arg_scope()   #获得模型的命名空间  

17

```

```

18     with slim.arg_scope(arg_scope): #构建模型
19         with slim.arg_scope([slim.conv2d,
20             slim.batch_norm, slim.fully_connected,
21             slim.separable_conv2d], reuse=reuse):
22             rotated_logits, end_points =
23                 pnasnet.build_pnasnet_large(preprocessed, num_classes = 1001,
24                 is_training=False)
25             prob = end_points['Predictions']
26             return rotated_logits, prob #返回批次输出结果
27
28 input_imgs = tf.Variable(tf.zeros((image_size, image_size, 3))) #定义输入
29 rotated_logits, probs = pnasnetfunrotate(input_imgs,reuse=False) #构建模型
30 checkpoint_file = r'pnasnet-5_large_2017_12_13\model.ckpt' #定义模型路径
31 variables_to_restore = slim.get_variables_to_restore()#(exclude=exclude)
32 init_fn = slim.assign_from_checkpoint_fn(checkpoint_file,
33     variables_to_restore, ignore_missing_vars=True)
34 sess = tf.InteractiveSession() #建立会话
35 init_fn(sess) #载入模型
36 img_path = './dog.jpg' #读入原始图片
37
38 def showresult(img,p): #可视化结果
39     fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 8))
40     .....
41     plt.show()
42
43 p = sess.run(probs, feed_dict={input_imgs: img})[0] #进行预测
44 showresult(img,p)

```

在以上代码中，省略号部分均与文件“11-1 用梯度下降方法攻击 PNASNet 模型.py”一致，这里不再详细描述。将输入图片随机旋转 4 次，并输入模型里进行预测。输出结果均是哈士奇。输出的结果与 11.2.7 小节类似，这里省略。

11.3.2 代码实现：生成并保存鲁棒性更好的对抗样本

修改 11.2 节训练部分的代码，将单张处理改成批次处理，让模型向预测错误的方向训练。具体的代码如下：

代码 11-3 制造鲁棒性更好的对抗样本（续）

```

45 def floatarr_to_img(floatarr): #定义输入
46     .....
47
48 x = tf.placeholder(tf.float32, (image_size, image_size, 3)) #定义输入
49 assign_op = tf.assign(input_imgs, x) #为 input_imgs 赋值

```

```

50
51 sess.run(assign_op, feed_dict={x: img})
52
53 below = input_imgs - 8.0/255.0          # 定义图片的变化范围
54 above = input_imgs + 8.0/255.0
55
56 belowv, abovev = sess.run([below, above]) # 输出结果并人工校验
57 .....
58
59 label_target = 880                      # 指定其他类别标签
60 label = tf.constant(label_target)
61 labels = tf.tile([label], [batchsize])      # 按照 batchsize 进行复制
62 loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
63     logits=rotated_logits, labels=labels) )
64
65 learning_rate=2e-1                         # 定义学习率
66 optim_step_rotated = tf.train.GradientDescentOptimizer(
67     learning_rate).minimize(loss, var_list=[input_imgs])
68
69 projected = tf.clip_by_value(tf.clip_by_value(input_imgs, below, above), 0,
    1)
70 with tf.control_dependencies([projected]):
71     project_step = tf.assign(input_imgs, projected) # 按照控制的阈值生成图片
72
73 demo_steps = 400                           # 定义训练次数
74 for i in range(demo_steps):
75     _, loss_value = sess.run([optim_step_rotated, loss])
76     sess.run(project_step)
77     if (i+1) % 10 == 0:
78         print('step %d, loss=%g' % (i+1, loss_value))
79     if loss_value < 0.02:                      # 提前结束
80         break
81 adv = input_imgs.eval()                     # 获取图片
82
83 p = sess.run(probs)[0]
84 showresult(floatarr_to_img(adv), p)
85 plt.imsave('dog880rotated.jpg', floatarr_to_img(adv)) # 保存图片
86 sess.close()

```

该代码的流程与 11.2 节非常类似，这里不再详细介绍。代码运行后，输出如下结果：

```

step 10, loss=5.33923
step 20, loss=0.0115749

```

同样迭代了 20 次完成了训练。生成了图片 dog880rotated.jpg。

11.3.3 代码实现：在 PNASNet 模型中比较对抗样本的效果

为了更直观地显示 11.3 节的对抗样本与 11.2 节的对抗样本在 PNASNet 模型中的效果，下

面通过一系列连续的旋转角度对两种对抗样本进行变化，并将结果可视化。在代码文件“11-2 用数据增强抗攻击.py”中添加以下代码：

代码 11-2 用数据增强抗攻击（续）

```

20 img_path = './dog880rotated.jpg'          # 载入支持旋转的对抗样本
21 imgtestrotated = image.load_img(img_path, target_size=(image_size,
22     image_size))
23 imgtestrotated = (np.asarray(imgtestrotated) / 255.0).astype(np.float32)
24 thetas = np.linspace(-np.pi/4, np.pi/4, 301)    # 生成一系列连续旋转角度
25 label_target = 880
26 p_naive = []
27 p_robust = []
28 for theta in thetas:                         # 对两个样本进行旋转，并输入模型进行结果预测
29     rotated = rotated_image.eval(feed_dict={input_imgs: imgtestrotated,
30         angle: theta})
31     p_robust.append(probs.eval(feed_dict={input_imgs:
32         rotated})[0][label_target])
33     rotated = rotated_image.eval(feed_dict={input_imgs: imgtest, angle:
34         theta})
35     p_naive.append(probs.eval(feed_dict={input_imgs:
36         rotated})[0][label_target])
37 # 可视化结果
38 robust_line, = plt.plot(thetas, p_robust, color='b', linewidth=2, label='支持旋转的对抗样本')
39 naive_line, = plt.plot(thetas, p_naive, color='r', linewidth=2, label='不支持旋转对抗样本')
40 plt.ylim([0, 1.05])
41 plt.xlabel('旋转角度')
42 plt.ylabel('880 类别的概率')
43 plt.legend(handles=[robust_line, naive_line], loc='lower right')
44 plt.show()
45 sess.close()

```

代码运行后，输出结果如图 11-6 所示。

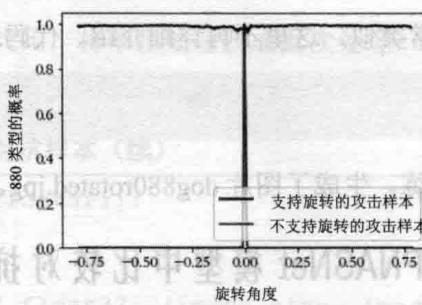


图 11-6 对抗样本的比较结果

在图 11-6 中有两条线。具体解读如下：

- 上面的那条线是支持旋转的对抗样本。可以看出，在整个横坐标区域内，模型预测 880 类的概率都为 1。
- 下面的那条线是不支持旋转的对抗样本。可以看出，只有横坐标值为 0 时，模型预测 880 类的概率为 1，其他情况下都是 0。

11.4 实例 60：以黑箱方式攻击未知模型

实例描述

通过黑箱方式攻击一个能够分类 MNIST 数据集的神经网络模型，构造出对抗样本，让未知结构的 MNIST 数据集分类器失效。

这里重点介绍黑箱攻击的实现原理。读者对黑箱攻击技术有了直观的印象和感受之后，便可以更有针对性地对自有模型进行加固。

11.4.1 准备工程代码

按以下步骤准备工程代码。

1. 获取代码

按照 11.1.2 小节中的 Gitlab 地址，将源码下载下来。将其解压缩之后，在 cleverhans-master\cleverhans_tutorials 路径下，找到 mnist_blackbox.py。该代码即为本实例所要讲解的示例代码。

2. 配置工作区

配置工作区的方法在 11.1.2 小节的“提示”部分已经介绍过。cleverhans 项目的不同版本代码，兼容性不是很友好。为了避免多版本不兼容的问题，这里直接在工作区里设置优先加载与示例代码版本相同的 cleverhans 库。具体做法如下：

(1) 在 spyder 中单击当前代码文件，确保该代码文件在主工作区内。接着选择菜单栏“Run”→“configuration per file...”命令，如图 11-7 所示。

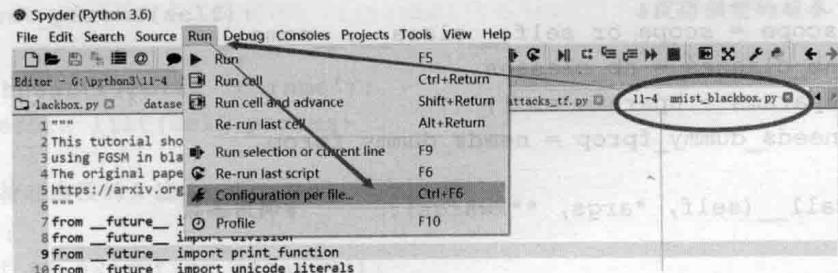


图 11-7 找到工作区设置菜单

(2) 弹出配置窗口。在“Working Directory Settings”中单击“The following directory”选项，

以及后面的文件夹按钮，在弹出的对话框中选择 cleverhans-master 工程源码所在的路径，如图 11-8 所示。

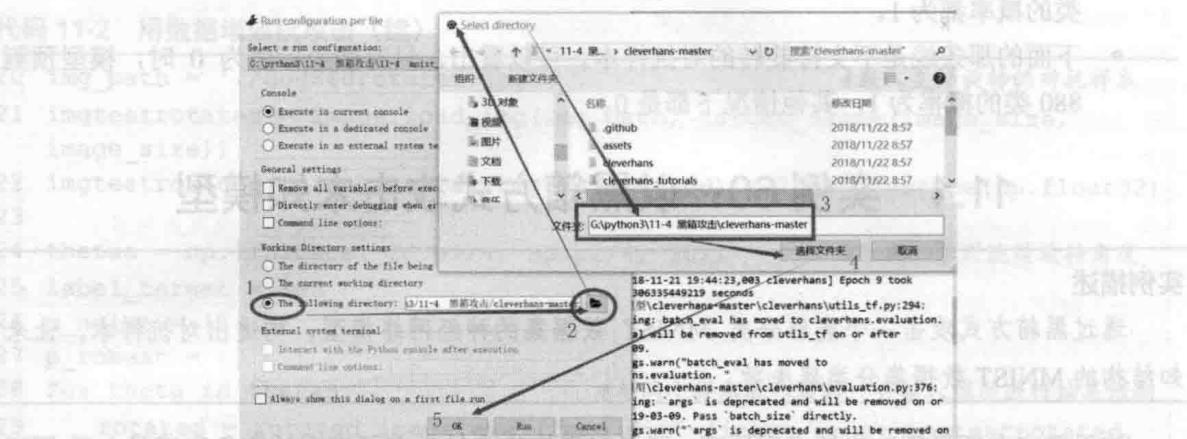


图 11-8 工作区设置窗口

11.4.2 代码实现：搭建通用模型框架

在 cleverhans 模块中单独实现了一套模型框架，用于构建攻防实验模型。

该框架只是在原有的 TensorFlow 接口上做了一些微小的封装。如果读者已经掌握了 TensorFlow 的机制，会很容易了解该框架的原理，并学会使用该框架。下面详细解读一下。

在源码的 cleverhans-master\cleverhans\model.py 文件里有一个 Model 类，它用于实现攻防模型的基本框架。其核心代码如下：

代码 model（片段）

```

01 class Model(object):
02     __metaclass__ = ABCMeta # 定义元类
03     # 定义常量字符串（用于构建字典类型中的 key。在描述网络层时使用）
04     O_LOGITS, O_PROBS, O_FEATURES = 'logits probs features'.split()
05
06     def __init__(self, scope=None, nb_classes=None, hparams=None,
07                  needs_dummy_fprop=False):      # 初始化函数
08
09         self.scope = scope or self.__class__.__name__
10        self.nb_classes = nb_classes
11        self.hparams = hparams or {}
12        self.needs_dummy_fprop = needs_dummy_fprop
13
14    def __call__(self, *args, **kwargs):      # 调用函数
15        .....
16
17        return self.get_probs(*args, **kwargs)
18
19    def get_logits(self, x, **kwargs):       # 获得输出层

```

```

20
21     outputs = self.fprop(x, **kwargs)
22     if self.O_LOGITS in outputs:
23         return outputs[self.O_LOGITS]
24     raise NotImplementedError(str(type(self)) + "must implement"
`get_logits`")
25             " or must define a " + self.O_LOGITS + "
26             " output in `fprop`")
27
28 def get_predicted_class(self, x, **kwargs):    #获得模型预测的分类结果
29
30     return tf.argmax(self.get_logits(x, **kwargs), axis=1)
31
32 def get_probs(self, x, **kwargs):                #获得模型的输出结果
33
34     d = self.fprop(x, **kwargs)
35     if self.O_PROBS in d:
36         output = d[self.O_PROBS]
37         min_prob = tf.reduce_min(output)
38         max_prob = tf.reduce_max(output)
39         asserts = [utils_tf.assert_greater_equal(min_prob,
40                                         tf.cast(0., min_prob.dtype)),
41                     utils_tf.assert_less_equal(max_prob,
42                                         tf.cast(1., min_prob.dtype))]
43         with tf.control_dependencies(asserts):
44             output = tf.identity(output)
45         return output
46     elif self.O_LOGITS in d:
47         return tf.nn.softmax(logits=d[self.O_LOGITS])
48     else:
49         raise ValueError('Cannot find probs or logits.')
50
51 def fprop(self, x, **kwargs):                    #构建模型的前向结构
52
53     raise NotImplementedError(`fprop` not implemented.)
54
55 def get_params(self):                          #获得模型的超参数
56
57     if hasattr(self, 'params'):
58         return list(self.params)
59
60     #支持动态图的方法
61     try:
62         if tf.executing_eagerly():
63             raise NotImplementedError("For Eager execution - get_params "
64                                         "must be overridden.")
65     except AttributeError:

```

```

66     pass # 处理，在算出的对流体中选择 cleverhans-mnist 1 的值和对应的梯度，如果
67     # 有多个梯度，则将它们加权平均
68     # 对静态图的处理
69     scope_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
70         scope=scope) # (self.scope + "/") 为自定义的子图
71
72     if len(scope_vars) == 0: # 如果没有参数，则使用参数
73         self.make_params()
74     scope_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
75         scope=scope) # (self.scope + "/") 为自定义的子图
76     assert len(scope_vars) > 0
77
78     # 断言语句。如果参数发生变化，则程序将会报错
79     if hasattr(self, "num_params"):
80         if self.num_params != len(scope_vars):
81             print("Scope: ", self.scope)
82             print("Expected " + str(self.num_params) + " variables")
83             print("Got " + str(len(scope_vars)))
84             for var in scope_vars:
85                 print("\t" + str(var))
86             assert False
87         else:
88             self.num_params = len(scope_vars)
89
90     return scope_vars
91
92     def make_params(self): # 设置模型的超参数
93         if self.needs_dummy_fprop:
94             if hasattr(self, "_dummy_input"):
95                 return
96             self._dummy_input = self.make_input_placeholder()
97             self.fprop(self._dummy_input)
98
99     def get_layer_names(self): # 获取网络层名称（以列表形式呈现）
100        raise NotImplementedError
101
102    def get_layer(self, x, layer, **kwargs): # 根据指定名称获取网络层
103        return self.fprop(x, **kwargs)[layer]
104
105    def make_input_placeholder(self): # 定义输入样本层
106
107        raise NotImplementedError(str(type(self)) + " does not implement "
108                                "make_input_placeholder")
109
110    def make_label_placeholder(self): # 定义输入标签层
111
112        raise NotImplementedError(str(type(self)) + " does not implement "

```

```

113 5. 代码实现：模型
114                                     "make_label_placeholder") 0.3[ea] 0.3[ea] 0.3[ea]
115 def __hash__(self):                      #重载 hash 算法的函数
116     return hash(id(self))
117
118 def __eq__(self, other):                  #重载相等的函数
119     return self is other

```

代码第2行引入了元类属性，让 model 类成为一个模板类。



提示：

关于元类的介绍，请参考《Python 带我起飞——入门、进阶、商业实战》一书的 9.11 节。

model 类作为攻防模型的基类，负责统一管理整个模型的接口。在定义模型时，只需要继承该类，并实现相关的必要接口。具体使用方法可以参考 11.4.3 小节。

11.4.3 代码实现：搭建被攻击模型

在本例中需要准备一个被攻击的模型，作为黑盒攻击的对象。

这里直接使用 cleverhans-master\cleverhans_tutorial\tutorial_models.py 中的 ModelBasicCNN 模型。该模型的类定义代码如下：

代码 tutorial_models（片段）

```

01 class ModelBasicCNN(Model):          #定义模型类
02     #初始化模型
03     def __init__(self, scope, nb_classes, nb_filters, **kwargs):
04         del kwargs
05         Model.__init__(self, scope, nb_classes, locals())
06         self.nb_filters = nb_filters
07
08     #构建模型
09     self.fprop(tf.placeholder(tf.float32, [128, 28, 28, 1]))
10     self.params = self.get_params()      #获得模型的超参
11
12     def fprop(self, x, **kwargs):        #定义模型的前向结构
13         del kwargs
14         my_conv = functools.partial(
15             tf.layers.conv2d, activation=tf.nn.relu,
16             kernel_initializer=initializers.HeReLUNormalInitializer)
17         with tf.variable_scope(self.scope, reuse=tf.AUTO_REUSE):
18             y = my_conv(x, self.nb_filters, 8, strides=2, padding='same')
19             y = my_conv(y, 2 * self.nb_filters, 6, strides=2, padding='valid')
20             y = my_conv(y, 2 * self.nb_filters, 5, strides=1, padding='valid')
21             logits = tf.layers.dense(
22                 tf.layers.flatten(y), self.nb_classes,
23                 kernel_initializer=initializers.HeReLUNormalInitializer)

```

```

24     return {self.O_LOGITS: logits,
25             self.O_PROBS: tf.nn.softmax(logits=logits)}

```

代码第 12 行，在 ModelBasicCNN 类中重载了 fprop 方法。在 fprop 方法中定义了模型的前向网络结构：

- (1) 使用 3 层卷积操作加一个全连接的网络结构，对 MNIST 数据集分类。
- (2) 在结果中返回了模型的最终预测值 O_LOGITS 和分类结果 O_PROBS。

11.4.4 代码实现：训练被攻击模型

cleverhans 模块中的模型框架，只支持到正向连接部分。如果对其进行训练，则还需要实现计算 loss 值部分，并调用另外一个封装好的函数——cleverhans.train 函数。

在本实例的代码文件“11-4_mnist_blackbox.py”中，用 prep_bbox 函数实现了模型的训练过程。

具体代码如下：

代码 11-4 mnist_blackbox（片段）

```

01 def prep_bbox(sess, x, y, x_train, y_train, x_test, y_test,
02               nb_epochs, batch_size, learning_rate,
03               rng, nb_classes=10, img_rows=28, img_cols=28, nchannels=1):
04
05     # 定义被攻击模型
06     nb_filters = 64
07     model = ModelBasicCNN('model1', nb_classes, nb_filters)
08     loss = CrossEntropy(model, smoothing=0.1)      # 定义损失函数
09     predictions = model.get_logits(x)              # 获取输出结果
10    print("Defined TensorFlow model graph.")
11
12    # 定义训练参数
13    train_params = {'nb_epochs': nb_epochs, 'batch_size': batch_size,
14                    'learning_rate': learning_rate}
15    # 训练模型
16    train(sess, loss, x_train, y_train, args=train_params, rng=rng)
17
18    # 评估模型
19    eval_params = {'batch_size': batch_size}
20    accuracy = model_eval(sess, x, y, predictions, x_test, y_test,
21                          args=eval_params)
22    print('Test accuracy of black-box on legitimate test '
23          'examples: ' + str(accuracy))                # 输出准确率
24
25    return model, predictions, accuracy

```

本小节完成了一个很普通的分类器模型，充当实例中的目标模型。在后面的操作中，将要对该分类器模型进行攻击。

11.4.5 代码实现：搭建替代模型

在搭建替代模型时，要求输入和输出必须一致。其他的内部结构和参数对整个攻击的结果影响不大。一个有经验的深度学习开发者，会根据模型的具体任务搭建出与其近似的模型。



提示：

有时可能需要搭建多个不同架构的模型进行尝试，从中找出效果更好的模型。

在本实例中，用多层全连接网络来搭建替代模型。具体代码如下：

代码 11-4 mnist_blackbox（片段）

```

26 class ModelSubstitute(Model):#构建替代模型
27     def __init__(self, scope, nb_classes, nb_filters=200, **kwargs):
28         del kwargs
29         Model.__init__(self, scope, nb_classes, locals())
30         self.nb_filters = nb_filters
31
32     def fprop(self, x, **kwargs):#搭建网络的前向结构
33         del kwargs
34         my_dense = functools.partial(
35             tf.layers.dense, kernel_initializer=HeReLUNormalInitializer)
36         with tf.variable_scope(self.scope, reuse=tf.AUTO_REUSE):
37             y = tf.layers.flatten(x)
38             y = my_dense(y, self.nb_filters, activation=tf.nn.relu)
39             y = my_dense(y, self.nb_filters, activation=tf.nn.relu)
40             logits = my_dense(y, self.nb_classes)
41         return {self.O_LOGITS: logits,
42                 self.O_PROBS: tf.nn.softmax(logits)}
```

从代码第 32~40 行可以看到，模型 ModelSubstitute 的输入为 x，输出为 logits。输入 x 需要在调用时被指定。输出 logits 返回的维度与初始化参数 nb_classes 有关（见代码第 40 行）。



提示：

在构建替代模型的过程中，核心内容是对决策边界的发现。对替代模型的层数要求并不是特别严格。例如在本实例中，如果在原有的替代模型基础上多加几层全连接，也不会有很明显的效果提升。

11.4.6 代码实现：训练替代模型

训练替代模型是本实例的关键环节。该环节主要是用雅可比矩阵的数据增强技术来制作有效样本（见 11.1.4 小节）。具体代码如下：

代码 11-4 mnist_blackbox（片段）

```
43 def train_sub(sess, x, y, #定义参数：会话、输入的样本、标签占位符
```

```

44         bbox_preds,      #黑箱模型的输出张量
45         x_sub, y_sub,    #初始一部分训练样本、标签
46         nb_classes,     #分类个数
47         nb_epochs_s, batch_size, learning_rate, data_aug, lmbda,
48         aug_batch_size, rng, img_rows=28, img_cols=28,
49         nchannels=1):
50
51     model_sub = ModelSubstitute('model_s', nb_classes)      #定义替代模型的结构
52     preds_sub = model_sub.get_logits(x)
53     loss_sub = CrossEntropy(model_sub, smoothing=0)        #定义损失函数
54
55     print("Defined TensorFlow model graph for the substitute.")
56     grads = jacobian_graph(preds_sub, x, nb_classes)       #定义雅可比矩阵
57
58     for rho in xrange(data_aug):                          #按照指定数据增强次数训练替代模型
59         print("Substitute training epoch #" + str(rho))
60         train_params = {                                  #指定替代模型的训练参数
61             'nb_epochs': nb_epochs_s,                      #训练的迭代次数
62             'batch_size': batch_size,                     #批次大小
63             'learning_rate': learning_rate}            #学习率
64
65         with TemporaryLogLevel(logging.WARNING, "cleverhans.utils.tf"):
66             train(sess, loss_sub, x_sub, to_categorical(y_sub, nb_classes),
67             init_all=False, args=train_params, rng=rng,
68             var_list=model_sub.get_params()) #用生成的数据训练模型
69
70     if rho < data_aug - 1:    #最后一次不需要再做雅可比数据增强了，直接退出
71         print("Augmenting substitute training data.") #执行基于雅可比矩阵的数据增强方法
72
73     lmbda_coef = 2 * int(int(rho / 3) != 0) - 1 #动态调整步长参数 lmbda_coef
74     x_sub = jacobian_augmentation(sess, x, x_sub, y_sub, grads,
75                                     lmbda_coef * lmbda, aug_batch_size)
76     print("Labeling substitute training data.")
77
78     y_sub = np.hstack([y_sub, y_sub])
79     x_sub_prev = x_sub[int(len(x_sub)/2):]          #获得构造好的输入点
80     eval_params = {'batch_size': batch_size}        #定义评估参数
81     #将构造好的 x 放入被攻击模型，生成标签
82     bbox_val = batch_eval(sess, [x], [bbox_preds], [x_sub_prev],
83                           args=eval_params)[0]
84     #获得输入点对应的标签
85     y_sub[int(len(x_sub)/2):] = np.argmax(bbox_val, axis=1)
86     showimg(rho,y_sub[int(len(x_sub)/2):],x_sub_prev,batch_size) #显示图片
87     return model_sub, preds_sub

```

代码第 74 行，用 `jacobian_augmentation` 函数获得了一部分输入样本点。在 11.1.4 小节介绍过，该返回值分为原始的输入样本与计算出来的输入样本。

在代码第 79 行，从 jacobian_augmentation 函数的返回值 `x_sub_prev` 中，取出输入样本点的后半部分（构造好的输入点）作为下次训练模型的样本数据。

代码第 82 行，用 `batch_eval` 函数将构造好的输入点送入被攻击模型 `bbox_preds` 中，以便获取对应的标签。如果被攻击模型在网络侧（云端），则这行代码要改成通过网络请求向模型发送输入样本，并取得标签结果。

训练替代模型的次数与构造样本的次数相对应。每次生成的样本都会与原始的输入样本结合起来，再构造出对应的标签（见代码第 85 行）。这些样本与标签组成数据集会被送入模型中，然后按照指定的迭代次数 `nb_epochs_s` 进行训练。

代码第 86 行，用函数 `showimg` 将生成的样本显示出来。函数 `showimg` 的定义见 4.3.4 小节。

11.4.7 代码实现：黑箱攻击目标模型

训练替代模型是本实例的关键环节。该环节主要是用雅可比矩阵的数据增强技术来制作有效样本。具体代码如下：

代码 11-4 `mnist_blackbox`（片段）

```

88 def mnist_blackbox(train_start=0, train_end=60000, test_start=0,
89                     test_end=10000, nb_classes=NB_CLASSES,
90                     batch_size=BATCH_SIZE, learning_rate=LEARNING_RATE,
91                     nb_epochs=NB_EPOCHS, holdout=HOLDOUT, data_aug=DATA_AUG,
92                     nb_epochs_s=NB_EPOCHS_S, lmbda=LMBDA,
93                     aug_batch_size=AUG_BATCH_SIZE):
94
95     set_log_level(logging.DEBUG)          # 设置日志级别
96
97     accuracies = {}
98
99     assert setup_tutorial()           # 定义 session
100
101    # 构建 MNIST 数据集
102
103    mnist = MNIST(train_start=train_start, train_end=train_end,
104                  test_start=test_start, test_end=test_end)
105
106    x_train, y_train = mnist.get_set('train')
107    x_test, y_test = mnist.get_set('test')
108
109    x_sub = x_test[:holdout]      # 取出一部分原始样本，用于训练替代模型所需的数据集
110    y_sub = np.argmax(y_test[:holdout], axis=1)
111
112    # 定义图片参数
113    img_rows, img_cols, nchannels = x_train.shape[1:4]
114    nb_classes = y_train.shape[1]      # 定义分类个数
115

```

```

116 # 定义占位符
117 x = tf.placeholder(tf.float32, shape=(None, img_rows, img_cols,
118                      nchannels))
119 y = tf.placeholder(tf.float32, shape=(None, nb_classes))
120 rng = np.random.RandomState([2017, 8, 30]) # 定义随机值种子
121
122 # 训练一个模型，作为黑箱攻击的目标
123 print("Preparing the black-box model.")
124 prep_bbox_out = prep_bbox(sess, x, y, x_train, y_train, x_test, y_test,
125                           nb_epochs, batch_size, learning_rate,
126                           rng, nb_classes, img_rows, img_cols, nchannels)
127 model, bbox_preds, accuracies['bbox'] = prep_bbox_out
128
129 # 训练替代模型
130 print("Training the substitute model.")
131 train_sub_out = train_sub(sess, x, y, bbox_preds, x_sub, y_sub,
132                           nb_classes, nb_epochs_s, batch_size,
133                           learning_rate, data_aug, lmbda, aug_batch_size,
134                           rng, img_rows, img_cols, nchannels)
135 model_sub, preds_sub = train_sub_out
136
137 # 评估替代模型
138 eval_params = {'batch_size': batch_size}
139 acc = model_eval(sess, x, y, preds_sub, x_test, y_test, args=eval_params)
140 accuracies['sub'] = acc
141 print("The accuracy of substitute model", acc) # 输出替代模型的准确率
142 # 用 FGSM 方法攻击模型
143 fgsm_par = {'eps': 0.3, 'ord': np.inf, 'clip_min': 0., 'clip_max': 1.}
144 fgsm = FastGradientMethod(model_sub, sess=sess) # 构建 fgsm 操作
145
146 eval_params = {'batch_size': batch_size}
147 x_adv_sub = fgsm.generate(x, **fgsm_par) # 对输入 x 用 fgsm 进行变换
148
149 # 将变换后的 x_adv_sub 张量放到被攻击模型里，并注入测试数据，测试准确率
150 accuracy = model_eval(sess, x, y, model.get_logits(x_adv_sub),
151                        x_test, y_test, args=eval_params)
152 print('Test accuracy of oracle on adversarial examples generated '
153       'using the substitute: ' + str(accuracy)) # 输入模型的准确率
154 accuracies['bbox_on_sub_adv_ex'] = accuracy
155
156 return accuracies

```

与 11.2 节的例子相比，cleverhans 模块中的 FGSM 方法功能更为强大。在 cleverhans 模块中，FGSM 方法的实现被封装在 FastGradientMethod 类中，该类用 FGSM 方法可以生成两种类型的对抗样本：

- 使模型输出错误标签的对抗样本。

- 使模型输出指定标签的对抗样本。

代码第 143 行, 通过构建参数字典 `fgsm_par` 指定 FGSM 的实现细节。代码中使用了默认值, 即让模型输出错误标签的对抗样本。有关字典 `fgsm_par` 中每个 key 的含义, 可以参考代码文件 “cleverhans-master\cleverhans\attacks.py” (见 11.4.1 小节) 中 `FastGradientMethod` 类的定义。



提示:

如果想让模型输出指定标签的对抗样本, 则需要在字典 `fgsm_par` 中添加 key 为 “`y_target`” 的键值对, 并将目的标签设置到 `value` 中。

代码第 144 行, 将替代模型 `model_sub` 对象传入 `FastGradientMethod` 类的初始化方法里, 得到实例化对象 `fgsm`。该对象用于实现 FGSM 方法。

代码第 147 行, 用 `fgsm` 的 `generate` 方法对原始的输入 `x` 进行变化。在 `generate` 方法中, 调用 `fgm` 函数, 用梯度下降的方法对输入的样本进行扰动处理。

在函数 `fgm` 中, 具体的处理过程如下:

- (1) 将输入样本 `x` 传入替代模型, 算出对应的标签。
- (2) 求出该标签与模型的原始输出值之间的 loss 值。
- (3) 根据 loss 值求出 `x` 的梯度。
- (4) 在梯度中添加干扰项 (本实例中干扰项为 0.3, 见代码第 143 行)。
- (5) 将干扰项更新到原来的输入样本 `x` 上。

经过多次迭代之后, 再将变化后的输入样本 `x` 传入替代模型时, 替代模型将会输出错误的结果。

`fgm` 的代码片段如下:

```
def fgm(x, logits, y=None, eps=0.3, ord=np.inf, clip_min=None, clip_max=None,
targeted=False,
    sanity_checks=True):#ord 为扰动样本的计算方式
    asserts = []
    ...
    if y is None:                                     #用模型中的 y 值做标签
        preds_max = reduce_max(logits, 1, keepdims=True) #取出替代模型的分类结果
        y = tf.to_float(tf.equal(logits, preds_max))      #将 y 变为 one-hot 标签
        y = tf.stop_gradient(y)                          #固定 y 值, 停止梯度
        y = y / reduce_sum(y, 1, keepdims=True)          #使其总概率为 1 (对 one-hot 无影响)
        loss = softmax_cross_entropy_with_logits(labels=y, logits=logits) #计算损失
    ...
    grad, = tf.gradients(loss, x)                      #求关于 x 的梯度
    ...
    optimal_perturbation = optimize_linear(grad, eps, ord) #对梯度做干扰
    adv_x = x + optimal_perturbation                  #更新 x 的值
    if (clip_min is not None) or (clip_max is not None): #对 x 值进行剪辑, 变化在指定值之间
        assert clip_min is not None and clip_max is not None
        adv_x = utils_tf.clip_by_value(adv_x, clip_min, clip_max)
    ...
    return adv_x                                       #返回对抗样本
```

在代码第 150 行，将扰动后的张量作为输入接入被攻击模型。将测试数据注入进去，并利用 `model_eval` 函数评测其准确率。观察黑箱攻击的效果。



提示：

由于本实例只是模拟攻击，所以在代码第 150 行，将经过 FGSM 方法处理后的张量传入被攻击模型的输入接口进行评测。这里采用这样的做法，只是为了简化代码，并不符合真实场景。

在实际场景中，攻击者接触不到被攻击模型的真正输入接口。所以，只能将测试数据注入 `fgsm` 模型以生成扰动后的图片，然后再将该图片作为对抗样本输入被攻击模型。

代码运行后，输入如下结果：

(1) 训练被攻击模型的结果。

```
Defined TensorFlow model graph.  
num_devices: 1  
.....  
[INFO 2018-11-23 14:37:26,917 cleverhans] Epoch 9 took 7.253604888916016 seconds  
Test accuracy of black-box on legitimate test examples: 0.99248730964467
```

结果显示，该模型的准确率为 0.99。

(2) 训练替代模型的结果。

```
Training the substitute model.  
Defined TensorFlow model graph for the substitute.  
Substitute training epoch #0  
num_devices: 1  
0  
.....  
4
```



图 11-9 生成的待输入样本

```
Substitute training epoch #5  
num_devices: 1  
The accuracy of substitute model 0.7469035532994924
```

图 11-9 所显示的是使用雅可比矩阵生成的待输入样本。最后一行是该模型的准确率。

可以看到，准确率只有 0.74。虽然比被攻击模型的准确率低很多，但它们拥有同样的决策边界。

(3) 输出模型被攻击后的准确率。

```
Test accuracy of oracle on adversarial examples generated using the substitute:  
0.686497461928934
```

从结果可以看到，将用黑盒攻击方式得到的对抗样本输入目标模型中，让模型的准确率降低到了 0.68。

11.4.8 扩展：利用黑箱攻击中的对抗样本加固模型

加固模型的方法有很多种，最直接的就是通过扩充样本集。在实现时，可以将用黑箱攻击得到的对抗样本放入训练集中对模型做二次训练。这样训练出的模型会有更强的抗攻击能力。

学习实际应用

本篇侧重于深度学习的工程化应用，即用 TensorFlow 框架训练出模型之后的事情。本篇主要介绍在不同场景中模型的制作方法和部署方法（包括网络端和移动端的部署）、人工智能在工程化项目中的应用方法和技巧、以及人工智能的价值和要面对的挑战。

本篇讲述者从专注技术的视角提升阅读行业的视角。

“以需求为中心，以价值为导向”的技术，才是最有用途的技术。本篇的内容可以帮助读者更好地驾驭这些技术。

- 第 12 章 TensorFlow 模型制作——一种功能、多种身份
- 第 13 章 TensorFlow 部署部署——模型与项目的深度结合
- 第 14 章 商业案例——科技源于生活、用于生活

第 5 篇 实战——深度学习实际应用

本篇侧重于深度学习的工程化应用，即用 TensorFlow 框架训练出模型之后的事情。本篇主要介绍在不同场景中模型的制作方法和部署方法（包括网络端和移动端的部署）、人工智能在工程化项目中的应用方法和技巧，以及人工智能的价值和要面对的挑战。

本篇将读者从专注技术的视角提升到关注行业的视角。

“以商业为中心，以价值为导向”的技术，才是最有用途的技术。本篇的内容可以帮助读者更好地驾驭这种技术。

- 第 12 章 TensorFlow 模型制作——一种功能，多种身份
- 第 13 章 TensorFlow 模型部署——模型与项目的深度结合
- 第 14 章 商业实例——科技源于生活，用于生活

1. 使用方法

前面已经对比过 Python 和 C++ 的模型运行效率，但是，这不能够说 Python 工具或 print 语句就是最好的核心语言。

另外，因为习惯了 Python，所以对 Python 的文档资料非常熟悉，尽管现阶段中英文阅读系更适合 Python 语义，熟悉操作又需要对 Python 有些经历，将文档读进那个 `__init__.py`

`if contrib_cg.read` 函数的使用非常熟练这个环节还是颇费力。`10000-to-00000-stab.*`

`10000-to-00000-stab.*`，首先应该阅读前文中的章节中讲解矩阵 `stab.*`

`locally()` 的具体操作文字，很容易有两点在理解上将会出现，以前的矩阵是零 `stab.*`

`10000-to-00000-stab.*`，通过该语句 `stab` 变量正相关系数好理解了，但相加后文本 `stab.*`，不理解函数的真正

含义，所以先看 `tf.matmul` 的实现，再看 `tf.matmul` 的实现，才能明白 `tf.matmul` 的真正

含义。所以先看 `tf.matmul` 的实现，再看 `tf.matmul` 的实现，才能明白 `tf.matmul` 的真正

第 12 章

TensorFlow 模型制作——一种功能， 多种身份

本章主要介绍与模型文件相关的操作。通过本章的例子，读者可以掌握模型的导入和导出方法，以及制作冻结图的方法。

12.1 快速导读

在学习实例之前，有必要了解一下模型制作方面的基础知识。

12.1.1 详细分析检查点文件

在训练过程中，TensorFlow 生成的检查点文件是由多个文件组成的。下面以 6.2 节的线性回归程序为例。运行 6.2 节的代码后，在 log 文件夹中生成的检查点文件如图 12-1 所示。

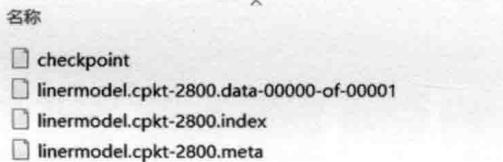


图 12-1 检查点文件

从图 12-1 中可以看到，一共有 4 种类型的文件。

- **checkpoint**: 一个模型索引文件，记录当前最新的检查点文件的名称。
- ***.data-00000-of-00001**: 存放模型中每个参数的具体值。
- ***.index**: 存放模型中参数名称与值之间的对应关系。
- ***.meta**: 存放模型的结构，即神经网络模型结构的节点名称。文件格式是 pb (protocol buffer)。

在有源码的情况下，*.meta 文件是没用的。可以通过设置来关闭生成*.meta 文件的功能。设置方法请见 6.2.1 小节“提示”部分。

在没有源码的情况下，需要用*.meta 文件来恢复模型结构。具体做法见 12.2 节实例。

**提示：**

在《深度学习之 TensorFlow——入门、原理与进阶实战》的 4.1.11 小节中，介绍了用 `print_tensors_in_checkpoint_file` 函数查看模型文件中张量内容的方法。可以通过该方法查看模型中的具体张量的名称及对应的数值。

12.1.2 什么是模型中的冻结图

冻结图是一个模型的最终导出文件。训练结束后，可以用冻结图来实现具体的应用。

冻结图不可以被用来做二次训练，只能用来计算结果。因为在运行冻结图时，可以不需要模型的源代码。所以，冻结图一般用在项目的最终交付环节。

具体生成和使用冻结图的方法请见 12.3 节。

12.1.3 什么是 TF Serving 模块与 `saved_model` 模块

TF Serving 模块的主要功能是将训练好的模型部署到生产环境中。可以让模型以远程调用的方式对外提供服务，并能够保持很高的性能。

TensorFlow 中还为 TF Serving 提供一个 `saved_model` 模块。用 `saved_model` 模块可以很方便地生成带有标签的冻结图文件。这种带有 TF Serving 标签的冻结图文件可以直接用到 TF Serving 的部署中。具体做法见 12.5 节、13.2 节。

12.1.4 用编译子图（`defun`）提升动态图的执行效率

编译子图的 API 为 `tf.contrib.eager.defun`，其中的 `defun` 是 `define function` 的缩写。`tf.contrib.eager.defun` 函数的作用是，将 Python 函数编译成一个可调用的子图，完成计算功能。这种方式可以提升代码的运行速度。

1. 使用方法

被编译的子图比 Python 函数的运行效率更高。但是，它不能够被 `pdb` 调试工具或 `print` 函数跟踪内部的执行情况。

另外，因为在程序运行时，被编译的子图在首次加载时也需要一定的开销。所以这种方案更适合 Python 函数中运算操作较多、较复杂的情况。

`tf.contrib.eager.defun` 函数的使用方法举例，代码片段如下：

```
import tensorflow as tf
tf.enable_eager_execution()

def f(x, y):                                # 定义函数 f
    return tf.reduce_mean(tf.multiply(x ** 2, 3) + y)
g = tf.contrib.eager.defun(f)                  # 将函数 f 编译成可调用子图
```

```

x = tf.constant([[2.0, 3.0]])
y = tf.constant([[3.0, -2.0]])

print(f(x, y).numpy())          #输出调用函数 f 的结果: 20.0
print(g(x, y).numpy())          #输出调用函数 g 的结果: 20.0

```

`tf.contrib.eager.defun` 函数除可以将 Python 函数编译成子图外，还可以将类中的 call 函数编译成子图。例如以下代码片段：

```

class MyModel(tf.keras.Model):          #定义模型类
    def __init__(self, keep_probability=0.2):      #定义节点
        super(MyModel, self).__init__()
        .....

    @tf.contrib.eager.defun           #装饰 call 方法
    def call(self, inputs, training=True):
        .....

```

2. 指定输入

向子图中传入形状不同的张量后，系统会生成不同的子图。可以通过输入来指定使用某个固定形状的子图。例如：

```

@tf.contrib.eager.defun(input_signature=[
    tf.contrib.eager.TensorSpec(shape=[None, 50, 300], dtype=tf.float32),
    tf.contrib.eager.TensorSpec(shape=[300, 100], dtype=tf.float32)
])
def my_sequence_model(words, another_tensor):
    .....

```

在上面代码片段中，用 `tf.contrib.eager.TensorSpec` 函数指定参数 `words` 和 `another_tensor` 的形状。其中，参数 `words` 支持不同批次的输入。

3. 注意事项

用 `tf.contrib.eager.defun` 函数修饰后的方法，与原有方法的处理逻辑是一样的。但也有几个特殊情况。具体如下：

(1) 在函数中存在取随机数的情况。

在被编译后的子图里，随机数会失效。例：

```

import tensorflow as tf
tf.enable_eager_execution()
import numpy as np
def fn():                      #定义函数 fn
    a = np.random.randn(1)       #取随机值
    x = tf.constant(2.) + a      #计算张量
    print("a", a, end=',')       #输出随机值
    return x                     #返回张量

g = tf.contrib.eager.defun(fn)   #编译子图 g
print(fn().numpy())             #输出调用函数 fn 的结果: a [0.1783442], [2.1783442]
print(fn().numpy())             #输出调用函数 fn 的结果: a [0.1783442], [2.1783442]

```

```
print(g().numpy())          #输出调用子图 g 的结果: a [-0.39338869], [1.6066113]
print(g().numpy())          #输出调用子图 g 的结果: [1.6066113]
print(g().numpy())          #输出调用子图 g 的结果: [1.6066113]
```

从上面的输出中可以看到：

- 每次调用子图 fn 都会得到不同值。
- 每次调用函数 g 都会得到相同值。

而且子图 g 只有第一次被调用时会输出 print 信息，再次被调用时不会有信息输出。这表示在第一次被调用时，系统将 fn 编译生成了子图，固化了随机值并且去掉了 print 信息。在后面运行时，直接用子图中的运算流来处理，所以每次都是一样的值。



提示：

为避免在子图中随机数失效这种情况发生，尽量不要把随机值放到函数内部。可以将其当作一个参数来输入，进行计算。

(2) 在函数中存在 BOOL（逻辑）运算的情况。

在被编译后的子图里，只允许对输入参数进行 Python 语法的 BOOL（逻辑）运算。对内部的张量做 BOOL（逻辑）计算时，需要用函数 tf.cond 来代替。例如：

```
import tensorflow as tf
tf.enable_eager_execution()
def fn(train = True):           # 定义函数 fn
    x = tf.constant(2.)
    y = tf.constant(2.)
    if train:                   # 对输入进行判断
        x = x*2
    else :
        x= x-1
    def f1():                   # 分支函数
        return x*10
    def f2():                   # 分支函数
        return x*100
    x =tf.cond(x < y, f1,f2)   # 在内部进行判断

    return x

g = tf.contrib.eager.defun(fn)
print(fn(True).numpy())         # 编译子图 g
print(fn(False).numpy())        # 输出调用函数 fn 的结果: 400.0
print(g(True).numpy())          # 输出调用函数 fn 的结果: 10.0
print(g(False).numpy())         # 输出调用子图 g 的结果: 400.0
print(g(False).numpy())         # 输出调用子图 g 的结果: 10.0
```

可以看到，调用函数 fn 与子图 g 的结果完全相同。



提示：

在编译子图的内部，除需要替换 BOOL 语句外，还需要将循环语句（while）替换成 tf.while_loop 语句。

(3) 在函数中存在定义变量语句的情况。

在被编译后的子图里，只有第一次被调用时会运行全部代码。在后续调用时，只运行其编译后的子图分支。分支内容是在编译时决定的。

在编译过程中，定义参数的函数 `tf.Variable` 也会被优化掉。这是需要注意的地方。例如：

```
import tensorflow as tf
tf.enable_eager_execution()
def fn():
    x = tf.Variable(0.0)
    x.assign_add(1.0)
    return x.read_value()

g = tf.contrib.eager.defun(fn)
print(fn().numpy())
print(fn().numpy())
print(g().numpy())
print(g().numpy())
```

定义函数 fn
定义变量
执行加 1 操作
返回结果

编译子图 g
输出调用函数 fn 的结果: 1.0
输出调用函数 fn 的结果: 1.0
输出调用子图 g 的结果: 1.0
输出调用子图 g 的结果: 2.0

从程序运行的结果中可以看到：

- 在两次运行函数 `fn` 时，内部都会重新定义一个变量给 `x`。每次运行时，变量 `x` 的值都会先变成 0，然后再加 1，最终返回结果 1.0。
- 在两次调用子图 `g` 时，第一次调用时，与运行函数 `fn` 一样——返回了 1.0。第二次调用时，`x` 已经在编译好的子图中存在，且 `x = tf.Variable(0.0)` 语句已经被优化掉了。于是变量 `x` 再加 1，变成了 2.0。



提示：

编译子图只适用与 TensorFlow 1.x 版本。在 TensorFlow 2.x 版本中推荐使用更高级的自动图功能，该功能不仅有编译子图同样的性能，而且还有比编译子图更简单的开发方式。详情请见 6.1.16 小节。

12.1.5 什么是 TF_Lite 模块

`TF_Lite` (TensorFlow Lite) 模块可以将现有的 TensorFlow 模型文件，转化成体积比较小的模型文件。它是 TensorFlow 针对移动和嵌入式设备的轻量级解决方案。它可以让神经网络模型很方便地运行在计算资源受限的设备上。

1. `TF_Lite` 模块的使用方式及帮助文档

`TF_Lite` 模块提供了命令行和调用 API 两种转化方式，用于生成 `lite` 格式的文件，并配有非常丰富的使用文档。其中：

(1) 命令行方式的使用文档如下：

https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/g3doc/convert/cmdline_reference.md

(2) API 接口调用方式的使用文档如下：

```
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/g3doc/convert/python\_api.md
```

需要说明的是，在 TensorFlow 1.12.0 及之前的版本中 TF_Lite 还并不完善：

- Windows 系统中的 TF_Lite 模块无法使用。如果使用 TensorFlow 1.12.0 及之前的版本，则只能在 Linux 系统中使用 TF_Lite 模块。
- 官方文档中的 API 与代码中的 API 对应不上。如果在使用过程中发生错误，则可以通过参考 TF_Lite 模块的源码进行解决。

在 TensorFlow 1.13.0 及之后的版本中，TF_Lite 模块相对比较成熟，并且支持在 Windows 系统下运行。

2. TF_Lite 的使用举例

以命令行的方式为例。用 TF_Lite 模块转化模型文件时，具体命令如下：

```
toco --graph_def_file=./retrained_graph.pb      --input_format=TENSORFLOW_GRAPHDEF
--output_format=TFLITE          --output_file=graph.tflite        --inference_type=FLOAT
--input_type=FLOAT              --input_arrays=input           --output_arrays=final_result
--input_shapes=1,224,224,3
```

该例子是在 TensorFlow 1.13.1 版本上实现的。其中，参数 `input_arrays` 与参数 `output_arrays` 是模型文件 `retrained_graph.pb` 中输入和输出节点的名称。这两个节点名称需要单独提取。提取的方式有两种：

- 通过 TensorBoard 工具在浏览器中查看。具体方法可以参考 12.4.2 小节。
- 在代码中，用 `print` 函数将张量的名字打印出来。

执行 `toco` 命令后，TF_Lite 模块将根据输入的冻结图生成 `graph.tflite` 文件。该文件可以应用在 Android 或 IOS 等系统上。具体实例可参考 13.3 节。

12.1.6 什么是 TFjs-converter 模块

TFjs-converter 模块是 TFjs 模块的配套接口，可以很方便地将训练好的 SavedModel、Keras h5、Frozen Model、TF-Hub 模型文件转化为 Web 接口的模型文件，以便通过 JavaScript 语言调用它。

具体细节可以参考如下地址：

```
https://github.com/tensorflow/tfjs-converter
```

12.2 实例 61：在源码与检查点文件分离的情况下，对模型进行二次训练

在公司与公司，或部门与部门之间合作开发时，出于对知识产权的保护，往往都需要将模

型的源代码进行隐藏。

例如：乙方为甲方提供模型算法服务，乙方开发的模型需要用甲方的数据进行训练。甲方的数据比较机密，希望将乙方的模型拿到甲方公司里来训练；而在合同没有履行完之前，乙方不希望将模型源码全部交给甲方。

在这种情况下，可以用模型的源代码与检查点文件相分离的方式进行合作。具体的实现方法为：

- (1) 乙方将模型的源代码与检查点文件分离。
- (2) 将检查点文件及流程代码（非模型的源代码部分）交给甲方。
- (3) 甲方拿到后，在自己公司内部进行训练模型，并反馈给乙方。
- (4) 乙方根据反馈进行模型的调优及改良。
- (5) 双方经过多次交互之后，完成模型的开发及训练过程。
- (6) 待合同流程全部履行完之后，乙方再将源码交付给甲方。

下面实现一个在源码与检查点文件分离情况的进行二次训练的实例。

实例描述

开发一个模型，让模型在一组混乱的数据集中找到 $y \approx 2x$ 的规律。并通过脱离模型源代码的情况下，对模型进行二次训练。

本实例的实现原理很简单：在模型训练的过程中，会用到网络模型代码中的几个节点（例如：输入、输出、优化器等）。核心思想就是，将需要用到的节点单独添加到模型的集合中，在二次训练时，将模型中用到的节点取出来。实现方式是：通过 `tf.add_to_collection` 函数将指定网络节点保存到模型的集合中，并用 `tf.get_collection` 函数读出模型中要使用的节点。

12.2.1 代码实现：在线性回归模型中，向检查点文件中添加指定节点

定义一个张量对象 `saver`（见代码第 37 行）。在会话运行中，用张量对象 `saver` 的 `save` 方法将检查点文件保存下来（见代码第 71、74 行）。完整的代码如下：

代码 12-1 在线性回归模型中添加指定节点到检查点文件

```

01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04
05 # (1) 生成模拟数据
06 train_X = np.linspace(-1, 1, 100)
07 train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3 #y=2x, 但是
加入了噪声
08 #图形显示
09 plt.plot(train_X, train_Y, 'ro', label='Original data')
10 plt.legend()
11 plt.show()
12

```

```

13 tf.reset_default_graph()
14
15 # (2) 构建网络模型
16 X = tf.placeholder("float")
17 Y = tf.placeholder("float")
18
19 # 模型参数
20 W = tf.Variable(tf.random_normal([1]), name="weight")
21 b = tf.Variable(tf.zeros([1]), name="bias")
22 # 前向结构
23 z = tf.multiply(X, W) + b
24 global_step = tf.Variable(0, name='global_step', trainable=False)
25 # 反向优化
26 cost = tf.reduce_mean(tf.square(Y - z))
27 learning_rate = 0.01
28 optimizer =
29     tf.train.GradientDescentOptimizer(learning_rate).minimize(cost, global_step) # 梯度下降
30 # 初始化所有变量
31 init = tf.global_variables_initializer()
32 training_epochs = 20
33 display_step = 2
34 savedir = "log2/"
35 saver = tf.train.Saver(tf.global_variables(), max_to_keep=1) # 生成saver,
    max_to_keep=1 表示只保留一个检查点文件
36 import tensorflow as tf
37 # 将指定节点通过添加到集合的方式放到模型里
38 tf.add_to_collection('optimizer', optimizer)
39 tf.add_to_collection('X', X)
40 tf.add_to_collection('Y', Y)
41 tf.add_to_collection('cost', cost)
42 tf.add_to_collection('result', z)
43 tf.add_to_collection('global_step', global_step)
44 # 定义生成 loss 值可视化的函数
45 plotdata = { "batchsize": [], "loss": [] }
46 def moving_average(a, w=10):
47     if len(a) < w:
48         return a[:]
49     return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in
50     enumerate(a)]
51 # (3) 建立 session 进行训练
52 with tf.Session() as sess:
53     sess.run(init)
54     kpt = tf.train.latest_checkpoint(savedir)
55     if kpt != None:

```

```

56     saver.restore(sess, kpt)
57
58     # 向模型输入数据
59     while global_step.eval() / len(train_X) < training_epochs:
60         step = int(global_step.eval() / len(train_X))
61         for (x, y) in zip(train_X, train_Y):
62             sess.run(optimizer, feed_dict={X: x, Y: y})
63
64         # 显示训练中的详细信息
65         if step % display_step == 0:
66             loss = sess.run(cost, feed_dict={X: train_X, Y: train_Y})
67             print("Epoch:", step+1, "cost=", loss, "W=", sess.run(W), "b=",
68             sess.run(b))
69             if not (loss == "NA"):
70                 plotdata["batchsize"].append(global_step.eval())
71                 plotdata["loss"].append(loss)
72             saver.save(sess, savedir+"linermodel.cpkt", global_step)
73
74     print(" Finished!")
75     saver.save(sess, savedir+"linermodel.cpkt", global_step)
76     print("cost=", sess.run(cost, feed_dict={X: train_X, Y: train_Y}), "W=",
77           sess.run(W), "b=", sess.run(b))
78
79     # 显示模型
80     plt.plot(train_X, train_Y, 'ro', label='Original data')
81     plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted
82     line')
83     plt.legend()
84     plt.show()
85
86     plotdata["avgloss"] = moving_average(plotdata["loss"])
87     plt.figure(1)
88     plt.subplot(211)
89     plt.plot(plotdata["batchsize"], plotdata["avgloss"], 'b--')
90     plt.xlabel('Minibatch number')
91     plt.ylabel('Loss')
92     plt.title('Minibatch run vs. Training loss')
93     plt.show()

```

上面代码运行完后，输出如下结果：

```

Epoch: 1 cost= 1.1687633 W= [0.5381455] b= [0.4353026]
.....
Epoch: 19 cost= 0.10066107 W= [2.1154778] b= [-0.03748273]
Finished!
cost= 0.100661084 W= [2.1154814] b= [-0.03748437]

```

生成的 loss 值图如图 12-2 所示。

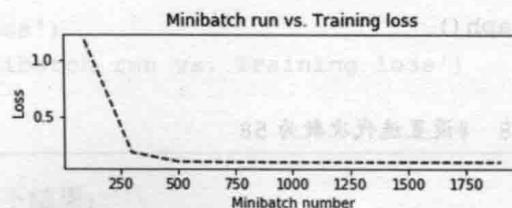


图 12-2 回归模型训练 2000 次的损失值

在程序运行之后，系统会在 log2 文件夹下生成了几个以“linemodel.cpkt-2000”开头的文件。它们就是检查点文件。

12.2.2 代码实现：在脱离源码的情况下，用检查点文件进行二次训练

将检查点文件中的模型的结构载入到当前运行图中，并从运行图中取得可操作的张量节点。具体步骤如下：

(1) 调用函数 `tf.train.import_meta_graph`，将检查点文件（该检查点文件是由 12.2.1 小节代码所生成的）中的节点名称导入到当前运行图中（见代码第 30 行）。

(2) 调用 `tf.get_collection` 函数，在当前运行图的集合中根据结合的名称找到对应的张量（见代码第 33~39 行）。

(3) 建立会话，训练模型。

完整的代码如下：

代码 12-2 用源码分离的方式进行二次训练

```

01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04
05 #生成模拟数据
06 train_X = np.linspace(-1, 1, 100)
07 train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3 #y=2x, 但是加入了噪声
08 #图形显示
09 plt.plot(train_X, train_Y, 'ro', label='Original data')
10 plt.legend()
11 plt.show()
12
13 #定义生成 loss 值可视化的函数
14 plotdata = { "batchsize":[], "loss":[] }
15 def moving_average(a, w=10):
16     if len(a) < w:
17         return a[:]
18     return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in enumerate(a)]

```

```

20 tf.reset_default_graph()
21
22 # 定义学习参数
23 training_epochs = 58 # 设置迭代次数为 58
24 display_step = 2
25
26 with tf.Session() as sess:
27     savedir = "log2/"
28     kpt = tf.train.latest_checkpoint(savedir) # 找到检查点文件
29     print("kpt:", kpt)
30     new_saver = tf.train.import_meta_graph(kpt+'.meta') # 从检查点的 meta 文件
31     中导入变量
32     new_saver.restore(sess, kpt) # 恢复检查点数据
33     print(tf.get_collection('optimizer')) # 通过集合取张量
34     optimizer = tf.get_collection('optimizer')[0] # 返回的是一个 list,
35     只是取第 1 个
36     X=tf.get_collection('X')[0]
37     Y=tf.get_collection('Y')[0]
38     cost=tf.get_collection('cost')[0]
39     result=tf.get_collection('result')[0]
40     global_step = tf.get_collection('global_step')[0]
41
42     # 节点恢复完成，可以继续训练
43     while global_step.eval()/len(train_X) < training_epochs:
44         step = int( global_step.eval()/len(train_X) )
45         for (x, y) in zip(train_X, train_Y):
46             sess.run(optimizer, feed_dict={X: x, Y: y})
47
48         # 显示训练中的详细信息
49         if step % display_step == 0:
50             loss = sess.run(cost, feed_dict={X: train_X, Y:train_Y})
51             print ("Epoch:", step+1, "cost=", loss)
52             if not (loss == "NA" ):
53                 plotdata["batchsize"].append(global_step.eval())
54                 plotdata["loss"].append(loss)
55             new_saver.save(sess, savedir+"linermodel.cpkt", global_step)
56
57             print (" Finished!")
58             new_saver.save(sess, savedir+"linermodel.cpkt", global_step)
59             print ("cost=", sess.run(cost, feed_dict={X: train_X, Y: train_Y}))
60
61             plotdata["avgloss"] = moving_average(plotdata["loss"])
62             plt.figure(1)
63             plt.subplot(211)
64             plt.plot(plotdata["batchsize"], plotdata["avgloss"], 'b--')
65             plt.xlabel('Minibatch number')

```

```

65     plt.ylabel('Loss')
66     plt.title('Minibatch run vs. Training loss')
67
68     plt.show()

```

代码运行后，显示如下结果：

```

kpt: log2/linermodel.cpkt-2000
INFO:tensorflow:Restoring parameters from log2/linermodel.cpkt-2000
[<tf.Operation 'GradientDescent' type=AssignAdd>]
Epoch: 21 cost= 0.099750884
....
Epoch: 57 cost= 0.099868104
Finished!
cost= 0.099868104

```

输出结果的第 1 行表示从检查点文件“log2/linermodel.cpkt-2000”中载入模型数据。

从输出结果的第 4 行可以看到，输出的损失值是 0.09。这说明模型是接着 2000 次的训练结果继续进行的（12.2.1 小节中，模型迭代训练 2000 次后损失值是 0.1）。

在训练结束后，程序生成的损失值（loss）图如图 12-3 所示。

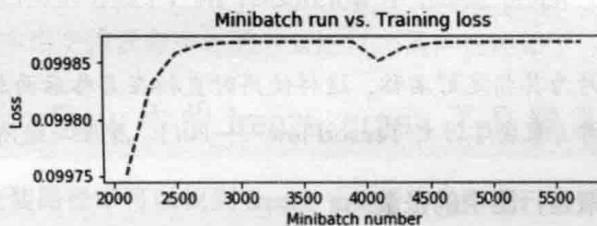


图 12-3 回归模型 5800 训练次的损失值

在图 12-3 中，表面看去好像是损失值越来越高。其实该曲线是在小数点后 4 位发生的抖动（见 y 坐标轴的单位），这属于正常现象。



提示：

如果是动态图或是估算器框架生成的模型，则可以先将其先转成静态图模式，再用本实例的方式将源码与文件分离。

将估算器模型代码转成静态图模型代码的例子可以参考 6.5 节。

将动态图模型代码转为静态图模型代码相对难度不大，读者可以自行尝试。

12.2.3 扩展：更通用的二次训练方法

在 12.2.2 小节的代码 30 行，调用函数 `tf.train.import_meta_graph`，将检查点文件中的节点导入程序的运行图中，并根据集合的名称恢复节点实现模型的二次训练。这种方法用起来相对比较简单，也好理解。

在这里再介绍一种更为通用的方法：直接用张量的名字代替集合进行操作。具体如下。

1. 获得运行图中需要的节点名称

在 12.2.1 小节代码的最后添加如下代码，将运行图中需要的节点打印出来：

```
print(optimizer.name)
print(X.name)
print(Y.name)
print(cost.name)
print(z.name)
print(global_step.name)
```

代码运行后，输出如下结果：

```
GradientDescent
Placeholder:0
Placeholder_1:0
Mean:0
add:0
global_step:0
```

显示的名称都是该节点在运行图中对应的名字。在载入模型时，可以通过这些名称获得具体的张量节点。



提示：

还可以在定义张量时为其指定好名称，这样使用时直接在名称后面加上索引值即可。名称和索引值的关系可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书的 4.3 节。

2. 根据节点名称获取运行图中的张量

在 12.2.2 小节代码第 33~39 行，将从集合中恢复节点的部分换作从图中恢复节点。具体代码如下：

```
my_graph = tf.get_default_graph() #获得当前运行图
#根据名字从运行图中获得对应的操作符 (OP) 及张量
optimizer = my_graph.get_operation_by_name('GradientDescent')
X = my_graph.get_tensor_by_name('Placeholder:0')
Y = my_graph.get_tensor_by_name('Placeholder_1:0')
cost = my_graph.get_tensor_by_name('Mean:0')
result = my_graph.get_tensor_by_name('add:0')
global_step = my_graph.get_tensor_by_name('global_step:0')
```

得到张量后，便可以对模型进行二次训练了。完整的代码见配套资源的代码文件。“12-3 使用源码分离方式二次训练-扩展.py”



提示：

在根据名字恢复节点的操作中，恢复 OP 与恢复张量的函数是不同的：

- 根据名字获取 OP，要使用 `get_operation_by_name` 函数。
- 根据名字获取张量，要使用 `get_tensor_by_name` 函数。

有关更多的图操作可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书

的 4.4 节。

另外，还可以用 `as_graph_def` 方法获取图中全部变量的定义。见源码文件“12-3 使用源码分离方式二次训练-扩展.py”的最后 3 行代码：

```
graph_def = my_graph.as_graph_def()          #获得全部定义
print(graph_def)                           #输出全部定义
tf.train.write_graph(graph_def, savedir, 'expert-graph.pb')  #将定义保存到文件里
```

在大型的模型文件中，节点的信息会有很多，可以先将其输出到文件中，再进行查看。

12.3 实例 62：导出/导入冻结图文件

冻结图文件是开发模型过程中的最终产物，专用于在生产环境中进行。

实例描述

开发一个模型，让模型在一组混乱的数据集中找到 $y \approx 2x$ 的规律。在模型训练好之后，将其导出成冻结图文件，并通过编写代码将该冻结图文件导入，用于预测。

在 6.2 节的线性回归模型基础上，用 TensorFlow 中 `freeze_graph` 工具的脚本实现冻结图的导出、导入功能。该脚本也支持以命令行的方式运行。具体描述如下：

12.3.1 熟悉 TensorFlow 中的 `freeze_graph` 工具脚本

在 TensorFlow 的安装路径中可以找到 `freeze_graph` 工具脚本，具体如下：

```
Anaconda3\lib\site-packages\tensorflow\python\tools\freeze_graph.py
```

该脚本可以命令行的方式使用，也可以通过模块载入的方式在代码中使用。脚本中的核心函数是 `freeze_graph` 函数，具体如下：

```
def freeze_graph(input_graph,      #图定义文件 GraphDef, 见 12.2.3 小节“提示”部分的介绍
                 input_saver,    #要载入的 saver 文件, 一般为空
                 input_binary,   #输入文件是否为二进制格式
                 input_checkpoint, #输入的检查点文件
                 output_node_names, #要导出的节点名称, 不能带后面的序号
                 restore_op_name, #该参数已舍弃
                 filename_tensor_name, #该参数已舍弃
                 output_graph,    #输出的冻结图路径及名称
                 clear_devices,  #是否删除节点的设备信息, 一般选 True, 否则会导致设备不兼容
                 initializer_nodes, #运行冻结图之前需要被初始化的节点
                 variable_names_whitelist="", #需要将变量转化为常量的白名单
                 variable_names_blacklist="", #指定某些变量不需要转化为常量
                 input_meta_graph=None,      #检查点的 meta 文件。
                 input_saved_model_dir=None, #输入检查点文件的路径
                 saved_model_tags=tag_constants.SERVING, #模型的标签。默认支持 TF Serving
                布署
                 checkpoint_version=saver_pb2.SaverDef.V2): #冻结图版本
```

可以看到，虽然该函数有很多参数，但是大部分参数都有默认值。使用时，只需要关注少量的必填参数即可。具体如下：

- 如果以命令行方式运行 `freeze_graph` 脚本，则需要在参数 `input_graph` 或参数 `input_meta_graph` 中指定一个，并填入参数 `output_node_names`、参数 `output_graph_path` 的值。
- 如果在代码里调用 `freeze_graph` 函数，则参数没有默认值，这时可以按照命令行中的默认值为 `freeze_graph` 函数的参数赋值。

12.3.2 代码实现：从线性回归模型中导出冻结图文件

将 6.2 节的全部代码复制过来，在其后面添加代码，导出冻结图文件。具体如下：

1. 添加函数，导出冻结图文件，并输出节点名称

编写代码实现如下步骤：

- (1) 定义函数 `exportmodel`，将现有运行图中的节点导出成冻结图。
- (2) 将模型代码中的输入节点 `X`、输出节点 `z` 的名称打印出来。



提示：

因为在应用场景中是不需要输入标签的，所以没有导出 `Y`。

具体代码如下：

代码 12-4 将线性回归模型导出成为冻结图文件

```

01 ..... #6.2 节中的全部代码，这里略过
02 import os
03 from tensorflow.python.tools import freeze_graph
04 def exportmodel(thisgraph,saverex,thissavedir,outnode='',freeze_file_name
= 'expert-graph-yes.pb'):
05
06     with tf.Session(graph=thisgraph) as sessex:
07         sessex.run(tf.global_variables_initializer())
08         kpt = tf.train.latest_checkpoint(thissavedir)
09
10         print("kpt:",kpt)
11
12         if kpt!=None:
13             saverex.restore(sessex, kpt)
14
15         #获取图中全部变量的定义
16         graph_def = thisgraph.as_graph_def()
17         #将变量的定义信息保存到 expert-graph.pb 文件中
18         tf.train.write_graph(graph_def, thissavedir, 'expert-graph.pb')
19
20         input_graph_path = os.path.join( thissavedir, 'expert-graph.pb')

```

```

21     input_saver_def_path = ""
22     input_binary = False
23     #指定导出节点的名字
24     output_node_names = outnode
25     restore_op_name = "save/restore_all"
26     filename_tensor_name = "save/Const:0"
27     output_graph_path = os.path.join(thissavedir, freeze_file_name)
28     clear_devices = True
29     input_meta_graph = ""
30
31     freeze_graph.freeze_graph(
32         input_graph_path, input_saver_def_path, input_binary, kpt,
33         output_node_names, restore_op_name, filename_tensor_name,
34         output_graph_path, clear_devices, "", "")
35
36 print(z.name,X.name)#将节点打印出来

```

代码第 4 行定义了函数 `exportmodel`，该函数用来实现具体的动态图导出操作。在该函数中，具体步骤如下：

- (1) 载入指定图中的检查点文件。
- (2) 将图中全部变量的定义信息导出到模型文件“expert-graph.pb”中。
- (3) 调用 `freeze_graph` 函数，按照参数 `freeze_file_name` 所指定的文件名称生成冻结图。

代码运行后，输出如下结果：

```
add:0 Placeholder:0
```

在输出结果可以看到：

- 输出节点 z（通过计算得来，具有可变属性）的名称是“add:0”（add 为的名称，0 为序号）。
- 输入节点 X（来自于样本，具有不变的属性）的名称是“Placeholder:0”。

2. 调用函数，实现导出冻结图文件功能

编写代码实现如下步骤：

- (1) 获得当前的运行图。
- (2) 生成 `saver` 对象用于保存模型文件
- (3) 用函数 `exportmodel` 生成冻结图文件。

具体代码如下：

代码 12-4 将线性回归模型导出成为冻结图文件（续）

```

37 thisgraph = tf.get_default_graph()
38 saverex = tf.train.Saver() #生成 saver 对象
39 exportmodel(thisgraph,saverex,savedir,"add,Placeholder")

```

代码第 39 行，在调用函数 `exportmodel` 时传入“`add,Placeholder`”，用来指定导出节点的名称。其中，“`add`”是输出节点 z 的名称，“`Placeholder`”是输入节点 X 的名称。这里只需要传

入节点名称中“：“之前的部分。

代码运行后，会在本地 log 文件夹中生成两个模型文件：

- expert-graph.pb（运行图的定义文件）。
- expert-graph-yes.pb（冻结图文件）。

其中的模型文件 expert-graph-yes.pb 就是最终输出的冻结图文件。

12.3.3 代码实现：导入冻结图文件，并用模型进行预测

编写代码实现如下步骤：

- (1) 用 `tf.GraphDef` 函数获得一个运行图对象 `my_graph_def`。
- (2) 将冻结图导入运行图对象 `my_graph_def` 中。
- (3) 用运行图对象 `my_graph_def` 的 `ParseFromString` 方法对冻结图文件进行解析。
- (4) 用 `tf.import_graph_def` 函数将冻结图对象 `my_graph_def` 的内容导入运行图中（见代码第 13 行）。
- (5) 用 `tf.get_default_graph` 函数获得运行图对象 `my_graph`。
- (6) 用运行图对象 `my_graph` 的 `get_tensor_by_name` 方法获得运行图中的输入、输出张量。
- (7) 建立会话，向输入张量中注入数据，获得输出张量的预测结果。

具体代码如下：

代码 12-5 导入冻结图并用模型进行预测

```

01 import tensorflow as tf
02
03 tf.reset_default_graph()
04
05 savedir = "log/"
06 PATH_TO_CKPT = savedir +'/expert-graph-yes.pb'
07
08 my_graph_def = tf.GraphDef()          # 定义 GraphDef 对象
09 with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
10     serialized_graph = fid.read()
11     my_graph_def.ParseFromString(serialized_graph) # 读取 pb 文件
12     print(my_graph_def)
13     tf.import_graph_def(my_graph_def, name='') # 恢复到运行图中
14
15 my_graph = tf.get_default_graph()      # 获得运行图
16 result = my_graph.get_tensor_by_name('add:0') # 将运行图中的 z 赋值给 result
17 x = my_graph.get_tensor_by_name('Placeholder:0') # 运行图中的 x 赋值给 x
18
19 with tf.Session() as sess:
20     y = sess.run(result, feed_dict={x: 5}) # 传入 5, 进行预测
21     print(y)

```

代码运行后，输出如下内容：

```

node {
  name: "Placeholder"
  .....
}

node {
  name: "weight"
  op: "Const"
  attr {
    key: "dtype"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_FLOAT
        tensor_shape {
          dim {
            size: 1
          }
        }
        float_val: 2.004253387451172
      }
    }
  }
}
.....
}
library {
}

```

[10.033242]

输出结果的最后一行是模型的预测结果（向模型中输入 5，预测出 10）。

输出结果中最后一行之前的所有信息都是运行图中定义的节点。这些定义就是从冻结图中读取出来的模型内容。

12.4 实例 63：逆向分析冻结图文件

冻结图虽然隐藏了很多信息，但是通过 TensorFlow 中的第三方工具，还是可以看到原始模型的样子。本实例用 TensorBoard 工具查看冻结图的网络结构。

实例描述

有一个冻结图文件，要求将其翻译到 TensorBoard 中以观察其模型结构。