

软件测试代码大作业

T14 薛定谔的测试 组长：王昊旻

学号	姓名
211250107	王昊旻
211250108	谷雨阳
211250097	丁晟元
211250074	左皓升

1. 代码大作业选题

面向XXX场景的深度学习模型测试技术

2. 实验简介

在智能模型的测试中，不同场景的领域特性会对测试方法的设计带来额外的要求，例如自动驾驶场景下无意义的噪声添加并不能代表现实生活中会遇到的天气、光照等真实问题，因此本小组希望能探究环境因素对自动驾驶模型准确率的影响，以探究如何更好地测试自动驾驶模型。基于上一阶段的文献阅读，我们选取了Testing DNN-based Autonomous Driving Systems under Critical Environmental Conditions这篇论文中的方法作为我们的参考。我们使用MUNIT对输入图像进行风格变换处理，输入进基于dave_dropout网络训练的自动驾驶模型中，通过测试神经元覆盖率kmnc和nbc，以及输出结果转向角与实际的偏差，得出在特定天气场景下检测模型最优的风格向量。

3. 分析代码

我们基于论文提出的TACTIC，实现了Dave_dropout模型上的测试代码

关键代码目录结构：

```
├── ads          //参考Dave2-Keras项目，用于自动驾驶模型的训练，保存一个模型权重，包含数据集
├── cache
├── └── Dave_dropout
│   ├── test_outputs  //保存临时的KNC、NBC、（预期）转向角度
│   └── train_outputs //保存模型层信息（layer_bounds）
```

- └─logger //test日志
- └─munit //MUNIT项目
- └─engine.py //搜索算法实现（ES搜索和随机搜索）
- └─test.py //初始化，加载模型并测试在源数据集上的KNC、NBC、转向角并存入cache
- └─test_knc.py //利用MUNIT风格转换和搜索算法测量并提高KNC，并记录出错数，输出日志
- └─test_nbc.py //利用MUNIT风格转换和搜索算法测量并提高NBC，并记录出错数，输出日志

大作业代码仓库: https://git.nju.edu.cn/t14/software_testing_a

TACTIC: <https://github.com/SEG-DENSE/TACTIC>

Dave2-Keras: <https://github.com/tech-rules/DAVE2-Keras>

MUNIT: <https://github.com/NVlabs/MUNIT>

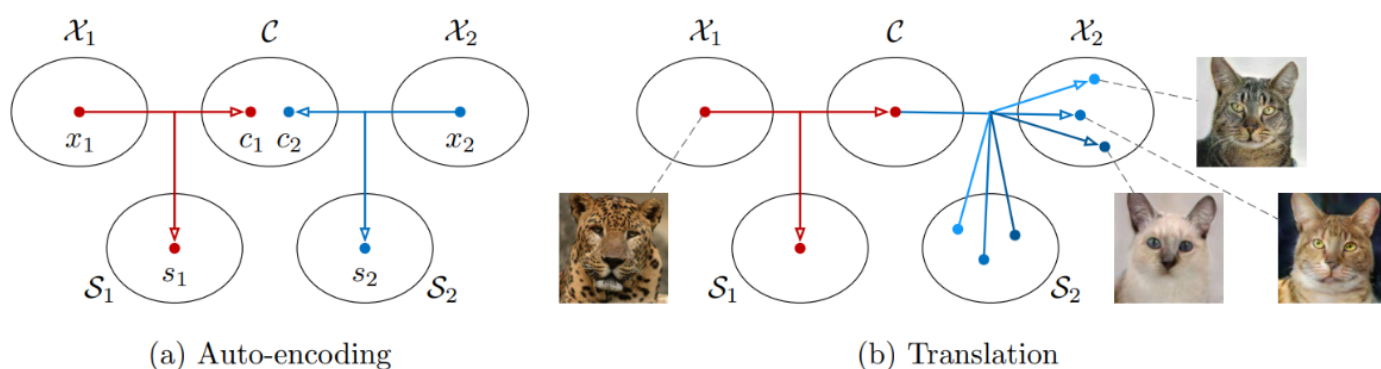
如果复现可以直接使用我们的MUNIT模型权重: <https://box.nju.edu.cn/f/ba86ab0bbd1f402bb4d8/>

自己本机训练的dave_dropout模型效果可能并不是很好，可以使用deepxplore仓库的模型

Model3.h5: <https://github.com/peikexin9/deepxplore/tree/master/Driving>

3.1 MUNIT代码分析

MUNIT（多模态无监督图像到图像转换）认为图像被编码出的latent code可以被进一步细化为内容编码 c 和风格编码 s 。不同类别的图像共享内容编码空间 C 而独享风格编码空间 S ，记domain X_1 的风格编码空间为 S_1 ，domain X_2 的风格编码空间为 S_2 。对于一个输入的图像，将其编码分解为共享内容 C 和目标的独享风格 S_1 ，之后将共享内容 C 和目标的独享风格 S_2 结合就可以进行图片风格转换。同时，从图中可以看到，不同的独享风格与共享内容的结合可以得到多样的结果。



下面是MUNIT_Trainer的部分代码，我们在测试中主要使用了MUNIT的gen_a和gen_b用于对图片进行编码和解码。

```

1 class MUNIT_Trainer(nn.Module):
2     def __init__(self, hyperparameters):
3         super(MUNIT_Trainer, self).__init__()
4         lr = hyperparameters['lr']
5         # Initiate the networks

```

```

6         # 生成模型a, 即由数据集A到数据集B的映射
7         self.gen_a = AdaINGen(hyperparameters['input_dim_a'],
hyperparameters['gen']) # auto-encoder for domain a
8         # 生成网络模型b, 即由数据集B到数据集A的映射
9         self.gen_b = AdaINGen(hyperparameters['input_dim_b'],
hyperparameters['gen']) # auto-encoder for domain b
10        # 鉴别模型a, 鉴别生成的图像, 是否和数据集A的分布一致
11        self.dis_a = MsImageDis(hyperparameters['input_dim_a'],
hyperparameters['dis']) # discriminator for domain a
12        # 鉴别模型b, 鉴别生成的图像, 是否和数据集B的分布一致
13        self.dis_b = MsImageDis(hyperparameters['input_dim_b'],
hyperparameters['dis']) # discriminator for domain b
14        # 使用正则化的方式
15        self.instancenorm = nn.InstanceNorm2d(512, affine=False)
16        # style 输出的特征码维度
17        self.style_dim = hyperparameters['gen']['style_dim']

```

3.2 test核心代码分析

对于KNC，每层神经元输出范围被分为K个部分，边界为 `layer_bounds_bin`，我们将中间层输出取平均值并根据边界做分箱操作得到部分的索引（1000之后视作超出边界），在 `knc_cov_dict` 中记作True，NBC则是对于边界被覆盖到的区域在 `nbc_cov_dict` 中记作True。

```

1 def update_knc(intermediate_layer_outputs):
2     for i, intermediate_layer_output in enumerate(intermediate_layer_outputs):
3         layer = layer_to_compute[i]
4         bins = layer_bounds_bin[layer.name]
5         output = intermediate_layer_output[0]
6         for id_neuron in range(output.shape[-1]):
7             _bin = np.digitize(np.mean(output[..., id_neuron]),
bins[id_neuron]) - 1
8             if _bin >= 1000 or knc_cov_dict[layer.name][id_neuron][_bin]:
9                 continue
10            knc_cov_dict[layer.name][id_neuron][_bin] = True
11
12 def update_nbc(intermediate_layer_outputs, layer_bounds_):
13     for i, intermediate_layer_output in enumerate(intermediate_layer_outputs):
14         output = intermediate_layer_output[0]
15         layer = layer_to_compute[i]
16         (low_bound, high_bound) = layer_bounds_[layer.name]
17         for id_neuron in range(output.shape[-1]):
18             val = np.mean(output[..., id_neuron])
19             if val < low_bound[id_neuron] and not nbc_cov_dict[layer.name]
[id_neuron][0]:
20                 nbc_cov_dict[layer.name][id_neuron][0] = True

```

```

21         elif val > high_bound[id_neuron] and not nbc_cov_dict[layer.name]
           [id_neuron][1]:
22             nbc_cov_dict[layer.name][id_neuron][1] = True
23         else:
24             continue

```

计算KNC/NBC

```

1 def current_knc_coverage():
2     """
3     Calculate the current K-Multi Section Neuron Coverage
4     :return:
5     """
6     covered = 0
7     total = 0
8     for layer in layer_to_compute:
9         covered = covered + np.count_nonzero(knc_cov_dict[layer.name])
10        total = total + np.size(knc_cov_dict[layer.name])
11    return covered / float(total)
12
13 def current_nbc_coverage():
14     """
15     Calculate the current Neuron Boundary Coverage
16     :return:
17     """
18     covered = 0
19     total = 0
20     for layer in layer_to_compute:
21         covered = covered + np.count_nonzero(nbc_cov_dict[layer.name])
22         total = total + np.size(nbc_cov_dict[layer.name])
23    return covered / float(total)

```

风格变换图像生成

```
encode = munit.gen_a.encode, decode = munit.gen_b.decode
```

对照上图MUNIT结构，此处 `encode` 为X1的编码器，`decoder` 为X2的解码器。测试过程中，将 `img` 通过 `encoder` 提取得到内容特征 `content` 和风格特征，然后选择另一个风格特征 `style` 与 `content` 组合，通过 `decoder` 合成 `outputs`，这样输出就与初始 `img` 有着相同的内容特征和不同的风格特征，其中 `style` 来自适应度函数指导的搜索算法。

```

1 def generator(img, style):
2     with torch.no_grad():
3         img = Variable(transform(img).unsqueeze(0).cuda())

```

```

4         s = Variable(style.unsqueeze(0).cuda())
5         content, _ = encode(img)
6
7         outputs = decode(content, s)
8         outputs = (outputs + 1) / 2.
9         del img
10        del s
11        del content
12        return outputs.data

```

适应度函数

(1)检测更多样化错误行为的能力，是由覆盖率的提高衡量的，设 R_t 是在基于搜索的测试过程中迄今为止尚未发现的KMNC/NBC区域集，而 R_s 是将被使用风格向量 s 的合成驾驶场景新覆盖的区域集。 s 检测各种错误行为的能力计算为： $F_c(s) = |R_s| / |R_t|$ 。

(2)检测更多错误行为的能力，体现在原始驾驶场景数据集与新合成的具有该风格向量的驾驶场景之间的平均转向角差异，形式上，假设 $g(x, s)$ 是使用风格向量 s 将驾驶场景 x 转换为目标环境类型的函数，而 $f(x)$ 是返回驾驶场景 x 的转向角度的函数。假设 I_o 为驾驶场景的原始数据集，则将样式向量 s 检测更多错误行为的能力计算为：

$$F_d(s) = \frac{1}{|I_o|} \sum_{x \in I_o} |f(x) - f(g(x, s))|.$$

整体适应度能力函数如下，其中 $\text{norm}()$ 为归一化函数：

$$F(s) = w_c * F_c(s) + w_d * \text{norm}(F_d(s)),$$

```

1 def fitness_function(original_images, original_preds,
2   theoretical_uncovered_sections, style):
3     preds = []
4     cov_dict = deepcopy(knc_cov_dict)
5     new_covered_sections = 0
6     logger.info("do prediction")
7     for img_num, img in enumerate(original_images):
8         if img_num >= 1000:
9             break
10        if img_num % 100 == 0:
11            print("the {i}th image".format(i=img_num))
12            logger.info("generate driving scenes {i}".format(i=img_num))
13            transformed_image = generator(img, style)[0]
14            transformed_image = preprocess_transformed_images(transformed_image)

```

```

14
15     logger.info("obtain internal outputs")
16     internal_outputs = intermediate_model.predict(transformed_image)
17     intermediate_outputs = internal_outputs[0:-1]
18     preds.append(internal_outputs[-1][0][0])
19
20     logger.info("calculate coverage")
21     new_covered_sections +=
get_new_covered_knc_sections(intermediate_outputs, cov_dict)
22
23     logger.info(new_covered_sections)
24     transformed_preds = np.asarray(preds)
25     o1 = float(new_covered_sections) / float(theoretical_uncovered_sections)
26     o2 = np.average(np.abs(transformed_preds - original_preds))
27     o2 = o2 / (o2 + 1) # normalize
28     logger.info("the o1 is {}".format(o1))
29     logger.info("the o2 is {}".format(o2))
30     del new_covered_sections
31     del transformed_preds
32     del cov_dict
33     return o1 + o2

```

4. 实验过程

4.1 实验平台

镜像 PyTorch 1.11.0, Python 3.8(ubuntu20.04), Cuda 11.3

GPU RTX A5000(24GB) * 2

CPU 32 vCPU Intel(R) Xeon(R) Platinum 8350C CPU @ 2.60GHz

内存 84GB

4.2 实验详情

实验一：自选了另一个数据集进行测试

数据集为Udacity的自动驾驶数据集，我们主要使用其中sunny风格（数据集原始图片）的图像作为测试输入

名称	修改日期	类型	大小
data	2023/12/16 20:21	文件夹	
IMG	2023/12/16 20:22	文件夹	
night	2023/12/16 20:22	文件夹	
rainy	2023/12/16 20:22	文件夹	
snow_night	2023/12/16 20:23	文件夹	
snowy	2023/12/16 20:22	文件夹	
sunny	2023/12/16 20:23	文件夹	
vec2	2023/12/16 22:45	文件夹	
vec2_night	2023/12/16 20:23	文件夹	
vec2_rainy	2023/12/16 20:23	文件夹	
vec2_snow_night	2023/12/16 20:24	文件夹	
vec2_snowy	2023/12/16 20:23	文件夹	
.DS_Store	2023/11/25 19:55	DS_STORE 文件	9 KB
data.txt	2023/4/11 15:18	文本文档	2,706 KB
driving_log.csv	2023/4/10 21:30	XLS 工作表	1,148 KB
Night.csv	2023/12/16 22:45	XLS 工作表	7,157 KB
Night1.csv	2023/11/24 16:45	XLS 工作表	113,059 KB
Origin.csv	2023/11/24 16:05	XLS 工作表	7,645 KB
Output.csv	2023/4/12 18:11	XLS 工作表	664 KB
Rainy.csv	2023/4/13 11:16	XLS 工作表	4,809 KB
Rainy1.csv	2023/11/24 17:57	XLS 工作表	112,348 KB
snow_night.csv	2023/4/13 11:00	XLS 工作表	4,704 KB
Snow_night1.csv	2023/11/24 17:37	XLS 工作表	112,436 KB
Snowy.csv	2023/4/13 10:40	XLS 工作表	4,984 KB
Snowy1.csv	2023/11/24 17:11	XLS 工作表	114,026 KB
Sunny.csv	2023/4/13 10:09	XLS 工作表	4,895 KB

第一步在Dave自带的训练集上训练Dave_dropout模型的权重，数据集下载链接

<https://box.nju.edu.cn/f/ed7bd122899147d5bea7/>。

第二步应当为训练风格向量encoder，由于平台性能限制，我们直接采用了TACTIC论文作者提供的encoder模型(.pt文件)。在实验简介中我们已经给出了nju box链接。

第三步为加载权重并测试神经元覆盖率

分别运行test.py初始化bounds，之后运行test_knc.py和test_nbc.py来计算knc和nbc。

由于平台性能及内存容量限制，对于同一个模型我们实际进行了如下实验：

1. 使用sunny的encoder模型（与对应的配置文件），每隔10张图片选取一张，进行4轮迭代，每轮执行75次搜索（30次不变即退出），测试KNC
2. 使用rainy的encoder模型（与对应的配置文件），每隔5张图片选取一张，进行4轮迭代，每轮执行64次搜索（30次不变即退出），测试KNC
3. 使用rainy的encoder模型（与对应的配置文件），每隔5张图片选取一张，进行4轮迭代，每轮执行64次搜索（30次不变即退出），测试NBC

实验二：使用DeepXplore的数据集和权重进行测试

原数据集获取链接

<https://github.com/udacity/self-driving-car/tree/master/datasets/CH2>

但由于种子原因只能从别处下载，因此从deepxplore仓库里进行了下载

<https://github.com/peikexin9/deepxplore/tree/master/Driving/testing>

由于实验一尝试的结果没有达到预期，我们选择尝试一些其他的测试输入，并希望对KNC作进一步的测试提升，实验二主要还是研究KNC相关的测试提升

4.2.1 数据集描述

HMB_1: 221 秒，阳光直射，光线变化多。开始时转弯很好，路肩线不连续，结束时车道合并，分道高速公路

HMB_2: 791 秒，双车道道路，阴影普遍存在，交通信号（绿灯），转弯处非常狭窄，中央摄像头无法看到太多路面，阳光直射，海拔高度变化快，导致山顶上的陡坡增高/降低。350 秒左右转入分道高速公路，很快恢复为双车道

HMB_4: 99 秒，返回山顶的分隔高速公路路段

HMB_5: 212 秒，护栏和双车道公路，开始时的阴影可能会给训练带来困难，结束时基本恢复正常

HMB_6: 371 秒，车流量较大的多车道分隔高速公路

4.2.2 具体步骤

4.2.2.1 初始化bounds

首先将DeepXplore的Model3.h5权重载入

然后用训练集的数据初始化bounds

运行testing/dave_dropout下的dave_init.py（这部分代码可能与主分支略有不同，但逻辑是相通的，这里参照dsy分支的代码目录来进行叙述）

```
1 # init bound的时候这段先注释掉
2 # with open('/root/autodl-
  tmp/software_testing_a/testing/dave_dropout/train_outputs/layer_bounds.pkl',
  'rb') as f:
3     layer_bounds = pickle.load(f)
4
5 # with open('/root/autodl-
  tmp/software_testing_a/testing/dave_dropout/train_outputs/layer_bounds_bin.pkl'
  , 'rb') as f:
6     layer_bounds_bin = pickle.load(f)
7
8 # ...
9
10 train_image_paths = '/root/autodl-tmp/training/'
```













```

11 # test_image_paths = '/root/autodl-tmp/test/center/'
12 filelist = []
13 for image_file in sorted(os.listdir(train_image_paths)):
14     if image_file.endswith(".jpg"):
15         filelist.append(os.path.join(train_image_paths, image_file))
16 print(len(filelist))
17 init_bounds(filelist)

```

生成的结果会存储在train_outputs下

 layer_bounds_bin.pkl	9 天前
 layer_bounds.pkl	9 天前
 / ... / testing / dave_dropout /	
名称	修改时间
 logger	8 天前
 test_outputs	9 天前
 train_outputs	9 天前
 __init__.py	10 天前
 dave_init.py	4 分钟前
 dave_knc.py	7 天前
 dave_nbc.py	10 天前

4.2.2.2 获取在测试集上的KNC初始值

```

1 # 加载
2 with open('/root/autodl-
  tmp/software_testing_a/testing/dave_dropout/train_outputs/layer_bounds.pkl',
  'rb') as f:
3     layer_bounds = pickle.load(f)
4
5 with open('/root/autodl-
  tmp/software_testing_a/testing/dave_dropout/train_outputs/layer_bounds_bin.pkl'
  , 'rb') as f:
6     layer_bounds_bin = pickle.load(f)
7
8 # ...
9
10 test_image_paths = '/root/autodl-tmp/test/center/'

```

```

11 filelist = []
12 for image_file in sorted(os.listdir(test_image_paths)):
13     if image_file.endswith(".jpg"):
14         filelist.append(os.path.join(test_image_paths, image_file))
15 print(len(filelist))
16 init_cov(filelist)

```

```

1/1 [=====] - 0s 35ms/step
5601
1/1 [=====] - 0s 36ms/step
5602
1/1 [=====] - 0s 36ms/step
5603
1/1 [=====] - 0s 35ms/step
5604
1/1 [=====] - 0s 35ms/step
5605
1/1 [=====] - 0s 36ms/step
5606
1/1 [=====] - 0s 35ms/step
5607
1/1 [=====] - 0s 36ms/step
5608
1/1 [=====] - 0s 36ms/step
5609
1/1 [=====] - 0s 35ms/step
5610
1/1 [=====] - 0s 35ms/step
5611
1/1 [=====] - 0s 34ms/step
5612
1/1 [=====] - 0s 34ms/step
5613
1/1 [=====] - 0s 33ms/step
0.28154234972677594
0.07991803278688525
(base) root@autodl-container-936011833c-f247d5f8:~/autodl-tmp/software_testing_a/testing/dave_dropout#

```

可以看到，CH2的5614张测试集初始的knc_coverage和nbc_coverage分别是0.2815和0.0799

4.2.2.3 应用风格转化进行搜索

由于资源受限，作者原话“因为 tactic 涉及到搜索，太慢了”

我们选择将最大搜索轮次设定为25

运行dave_knc.py结果如下

```
168512 2023-12-23 23:18:32,338 - __main__ - INFO - new_covered_sections is 22867
168513 2023-12-23 23:18:32,339 - __main__ - INFO - the o1 is 0.04348074103793227
168514 2023-12-23 23:18:32,340 - __main__ - INFO - the o2 is 0.438873145223192
168515 2023-12-23 23:18:32,340 - __main__ - INFO - my current coverage is 0.3127814207650273
168516 2023-12-23 23:18:32,340 - __main__ - INFO - finish selection
168517 2023-12-23 23:18:32,340 - __main__ - INFO - the 9's fitness is 0.48235388626112424
```

可以看到覆盖了2万多个new_section，knc覆盖率也有了明显的提升

5. 实验结果

5.1 实验一

实验序号	encoder	源测试输入图片数	搜索次数	初始KNC/NBC	优化后KNC/NBC	出错数
1	sunny	562	75	KNC=35.4781%	KNC=35.5296%	228
2	rainy	1204	64	KNC=49.9287%	KNC=50.0799%	444
3	rainy	1204	64	NBC=0.0%	NBC=21.0572%	444

虽然我们的测试规模远不及原论文的实验，但从实验结果与日志信息来看，风格变换和风格ES搜索在提升源测试集的神经元覆盖率KNC/NBC以及检测错误上是有效的，尤其是对于NBC指标提升明显，使得边界区域的神经元也能被覆盖到。

KNC的提升不大我们推测主要原因是源测试输入数量太低。

5.2 实验二

实验序号	encoder	源测试输入图片数	搜索次数	初始KNC	优化后KNC
1	snow_night	5614	25	KNC=28.1542%	KNC=31.2781%