SHANGHAI JIAO TONG UNIVERSITY

研究生课程项目报告

GRADUATE COURSE PROJECT REPORT

课程：神经网络与机器学习
题目：基于 MNIST 数据集使用 DCGAN 生成手写数字

学生姓名：于泽汉
学生学号：118039910141
任课教师：陈海宝 副教授
学院(系)：电子信息与电气工程学院 （微纳电子学系）
开课学期：2019 年 （春季）

1 摘要

在近年的计算机视觉应用中，卷积神经网络（CNN）的监督学习颇受青睐，而无监督学习相关的研究较少。

就在这时，DCGAN（深度卷积生成对抗网络）应运而生，诸多研究和实验证明，它们是无监督学习的有力候选者，弥补了计算机视觉领域中监督学习和无监督学习之间的鸿沟。

在研究者提出 DCGAN 之后，人们用它做了进一步的研究和更多的实验。这一网络结构在训练各种图像数据集时展现出了优异的能力，从物体的局部到整个场景，生成器和判别器都学习到了丰富的层次表达。

2 模型建立与代码实现

Listing 1: 导入必要的库

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

import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
import time

from IPython import display
```

2.1 导入数据集

本次实验采用的是 MNIST 数据集。

Listing 2: 导入数据集

```python
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
```

查看数据集的信息：

Listing 3: 查看数据集信息

```python
train_images.shape,train_labels.shape
train_labels[0:20]
```

输出：

```
((60000, 28, 28), (60000,))
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9], dtype=
    uint8)
```

输出信息表明，该数据集中共有 60000 张图像，每张图像的大小为 28×28，单位为像素。这些图像都对应一个标签，标签的数据类型是 8 位无符号整数，也就是是手写数字图像对应实际数字。

接着查看手写图像的内容：

Listing 4: 查看手写图像

```python
row,col = 10,20
fig_dataset = plt.figure(frameon=False)

for i in range(row):
    j,k = 0,0
    while (k<col):
        if train_labels[j]==i:
            fig_dataset.add_subplot(row, col, col*i+k+1)
            plt.imshow(train_images[j],cmap="gray")
            plt.axis('off')
            k += 1
        j += 1

plt.savefig("./report/images/dateset_preview.png", bbox_inches = 'tight',
    pad_inches=0)
```

得到的图像如下：

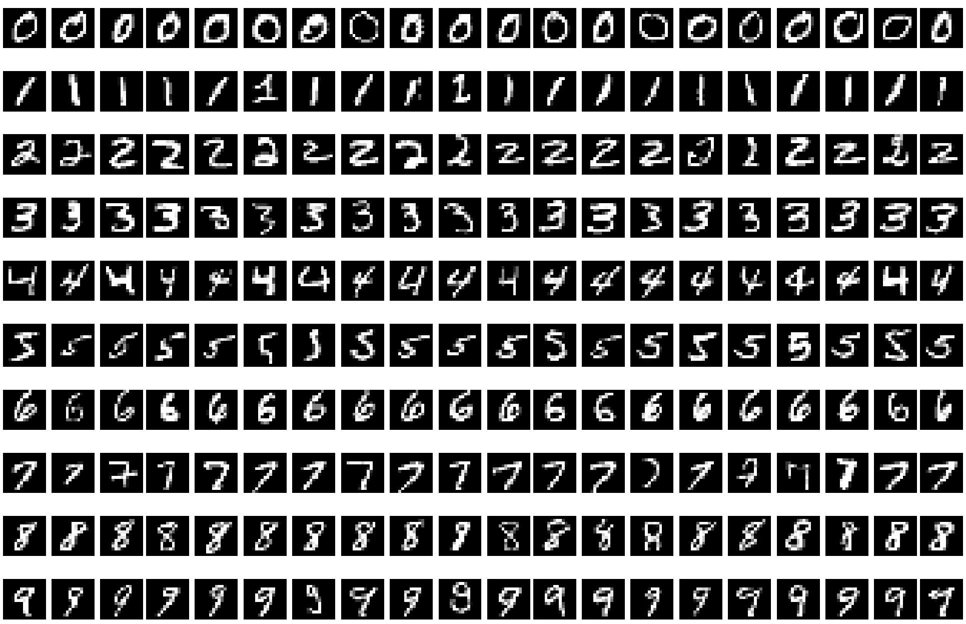

图 1: 手写数字 0 - 9 的图像

Listing 5: 预处理图像数据

```python
# 将图像像素矩阵尺寸转成 28 x 28
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype
    ('float32')
# 重映射像素值范围到 [-1, 1]
train_images = (train_images - 127.5) / 127.5
# 设置缓存大小
BUFFER_SIZE = len(train_labels)
# 设置批处理大小
BATCH_SIZE = 256
# 打乱数据集并设置批处理数目
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(
    BUFFER_SIZE).batch(BATCH_SIZE)
```

2.2 创建模型

2.2.1 判别器模型

Listing 6: 创建判别器模型

```python
def make_discriminator_model():
    # 采用序贯模型，也就是将多个网络层直接线性堆叠
    model = tf.keras.Sequential()

    # 添加一个卷积层
    model.add(tf.keras.layers.Conv2D(64, (5, 5), strides=(2, 2), padding='
        same'))
    # 使用带泄露的线性整流函数
    model.add(tf.keras.layers.LeakyReLU())

    # 添加一个 Dropout 层，随机扔掉部分神经元，避免过拟合
    model.add(tf.keras.layers.Dropout(0.3))

    # 添加一个卷积层
    model.add(tf.keras.layers.Conv2D(128, (5, 5), strides=(2, 2), padding='
        same'))
    # 使用带泄露的线性整流函数
    model.add(tf.keras.layers.LeakyReLU())

    # 添加一个 Dropout 层
    model.add(tf.keras.layers.Dropout(0.3))

    # 添加一个 Flatten 层，连接卷积层和全连接层
    model.add(tf.keras.layers.Flatten())

```

```
    # 添加一个全连接层
    model.add(tf.keras.layers.Dense(1))

    return model

discriminator = make_discriminator_model()
```

2.2.2 生成器模型

Listing 7: 创建生成器模型

```
def make_generator_model():
    # 采用序贯模型
    model = tf.keras.Sequential()

    # 添加一个全连接层
    model.add(tf.keras.layers.Dense(7*7*256, use_bias=False, input_shape
        =(100,)))
    # 对该层进行批标准化，使学习收敛更快更稳
    model.add(tf.keras.layers.BatchNormalization())
    # 使用带泄露的线性整流函数
    model.add(tf.keras.layers.LeakyReLU())

    # 调整该层输出的尺寸
    model.add(tf.keras.layers.Reshape((7, 7, 256)))
    # 检查该层输出的尺寸，这里的 None 表示使用批处理大小
    assert model.output_shape == (None, 7, 7, 256)

    # 添加一个反卷积层
    model.add(tf.keras.layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
        padding='same', use_bias=False))

    # 检查该层输出的尺寸
    assert model.output_shape == (None, 7, 7, 128)
    # 对该层进行批标准化
    model.add(tf.keras.layers.BatchNormalization())
    # 使用带泄露的线性整流函数
    model.add(tf.keras.layers.LeakyReLU())

    # 添加一个反卷积层
    model.add(tf.keras.layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
        padding='same', use_bias=False))
    # 检查该层输出的尺寸
    assert model.output_shape == (None, 14, 14, 64)
    # 对该层进行批标准化
    model.add(tf.keras.layers.BatchNormalization())
```

```
    # 使用带泄露的线性整流函数
    model.add(tf.keras.layers.LeakyReLU())

    # 添加一个反卷积层
    model.add(tf.keras.layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
        padding='same', use_bias=False, activation='tanh'))
    # 检查该层输出的尺寸
    assert model.output_shape == (None, 28, 28, 1)

    return model

generator = make_generator_model()
```

2.2.3 损失函数和优化器

Listing 8: 判别器损失函数

```
def discriminator_loss(real_output, generated_output):
    # [1,1,...,1] with real output since it is true and we want our
        generated examples to look like it
    real_loss = tf.losses.sigmoid_cross_entropy(multi_class_labels=tf.
        ones_like(real_output), logits=real_output)

    # [0,0,...,0] with generated images since they are fake
    generated_loss = tf.losses.sigmoid_cross_entropy(multi_class_labels=tf.
        zeros_like(generated_output), logits=generated_output)

    total_loss = real_loss + generated_loss

    return total_loss
```

Listing 9: 生成器损失函数

```
def generator_loss(generated_output):
    return tf.losses.sigmoid_cross_entropy(tf.ones_like(generated_output),
        generated_output)
```

Listing 10: 生成器损失函数

```
learn_rate = 1e-4
generator_optimizer = tf.train.AdamOptimizer(learn_rate)
discriminator_optimizer = tf.train.AdamOptimizer(learn_rate)
```

2.3 训练网络

2.3.1 配置网络

Listing 11: 配置网络参数

```
EPOCHS = 200
noise_dim = 100
num_examples_to_generate = 16

# We'll re-use this random vector used to seed the generator so
# it will be easier to see the improvement over time.
random_vector_for_generation = tf.random_normal([num_examples_to_generate,
    noise_dim])
```

Listing 12: 设置训练步骤

```
def train_step(images):
    noise = tf.random_normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        generated_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(generated_output)
        disc_loss = discriminator_loss(real_output, generated_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.variables
        )
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator
        .variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator,
        generator.variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
        discriminator.variables))
train_step = tf.contrib.eager.defun(train_step)
```

Listing 13: 完整训练过程

```
def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for images in dataset:
            train_step(images)
```

```
        display.clear_output(wait=True)
        generate_and_save_images(generator, epoch + 1, random_vector_for_generation)

        print ('Time taken for epoch {} is {} sec'.format(epoch + 1, time.time()-start))
        # generating after the final epoch
    display.clear_output(wait=True)
    generate_and_save_images(generator, epochs, random_vector_for_generation)

%%time
train(train_dataset, EPOCHS)
```

2.4 生成图像

Listing 14: 生成图像

```
epoch_images_path = 'epoch_images/'
def generate_and_save_images(model, epoch, test_input):
    # make sure the training parameter is set to False because we
    # don't want to train the batchnorm layer when doing inference.
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4,4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')


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

    plt.savefig(epoch_images_path+'epoch_{:04d}.png'.format(epoch))
    plt.show()

# Display a single image using the epoch number
def display_image(epoch_no):
  return PIL.Image.open(epoch_images_path+'epoch_{:04d}.png'.format(epoch_no))

display_image(EPOCHS)
```

3 结果与分析

4 参考文献