瓦力人工智能
粉丝: 阅读:
关注
更多
成为创作者
申请成为专栏UP主

深度学习笔记53_给我一张脸还你一个微笑_图像生成算法实践

学习

VAE 编码网络的构建

我们知道VAE第一步就是对输入的图像进行编码，编码最好生成的是平均值和方差两个参数。这里我们需注意的是。我们使用的是一个网络进行编码，利用一个简单的卷积神经网络来进行，最后网络利用densen层输出为两个向量。这样就可以完成编码的工作。具体coding如下。

```
import keras
from keras import layers
from keras import backend as K
from keras.models import Model
import numpy as np

# 输入是大小为28x28，灰度图像
img_shape = (28,28,1)
# batchsize 为16
batch_size = 16
# 输出的潜在空间的维度
latent_dim = 2

input_img = keras.Input(shape = img_shape)

x = layers.Conv2D(32,3,
                  padding='same',
```

```
                  activation='relu')(input_img)
x = layers.Conv2D(64,3,
                  padding='same',
                  activation='relu',
                  strides=(2,2))(x)
x = layers.Conv2D(64,3,
                  padding='same',
                  activation='relu')(x)
x = layers.Conv2D(64,3,
                  padding='same',
                  activation='relu')(x)
# 返回张量x的尺寸:(14, 14, 64)
shape_before_flattening = K.int_shape(x)
# 拉伸成一列向量
x = layers.Flatten()(x)
x = layers.Dense(32,activation='relu')(x)

# 图像被编码成两个参数
z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)

model = Model(input_img,z_mean)
model.summary()
```

采样获得潜在空间点

上面的网络中获得的潜在空间z_mean,z_log_var，接下来我们需要随机获得潜在的空间点采样。这里有几个注意点：

- epsilon是一个很小的随机值
- lambda来封装层

lambda来封装层

lambda函数的功能：将任意表达式封装为 Layer 对象。

函数的方式：

```
keras.layers.Lambda(function, output_shape=None, mask=None,
arguments=None)
```

里面参数的定义

> - function: 需要封装的函数。将输入张量作为第一个参数。
> - output_shape: 预期的函数输出尺寸。只在使用 Theano 时有意义。可以是元组或者函数。如果是元组，它只指定第一个维度；样本维度假设与输入相同：
> - output_shape = (input_shape[0],) + output_shape 或者，输入是 None 且样本维度也是 None：output_shape = (None,) + output_shape 如果是函数，它指定整个尺寸为输入尺

寸的一个函数：output_shape = f(input_shape)

- arguments: 可选的需要传递给函数的关键字参数。

```
def sampling(args):
    z_mean,z_log_var =args
    # 计算epsilon 随机数
    epsilon = K.random_normal(
        shape=(K.shape(z_mean)[0],latent_dim),
        mean=0.,stddev=1.)
    # 获得返回值采样的值
    return z_mean + K.exp(z_log_var) * epsilon

# 利用lambda来创建一个层
z = layers.Lambda(sampling)([z_mean,z_log_var])
```

VAE 解码器的实践

解码器的主要目的是将采样向量z通过解码将其恢复到与原始图的相同尺寸的图像。
利用用到的技术手段：

- 对z进行上采样
- 转置卷积：完成对数据的尺寸的放大
- 卷积：完成对数据的维度的缩放，让其达到只有一张图的状态

里面的转置卷积有点像反转卷和空洞卷积的效果，但是实现原理不一样，其过程如下：
在理解转置卷积（Transposed Convolution）计算过程之前，先来看一下如何用矩阵相乘的方法代替传统的卷积。

```
decoder_input = layers.Input(K.int_shape(z)[1:])

# 对输入进行上采样
#shape_before_flattening: (None, 14, 14, 64)
# np.prod()函数用来计算所有元素的乘积:14*14*64
# 获得上采样神经元的个数
x = layers.Dense(np.prod(shape_before_flattening[1:]),
                 activation='relu')(decoder_input)
# 通过reshape 将dense转换成（14, 14, 64)
x = layers.Reshape(shape_before_flattening[1:])(x)

# 采用转置卷积，图像转换成原始图像大小
x = layers.Conv2DTranspose(32,3,
                           padding='same',
                           activation='relu',
                           strides=(2,2))(x)
# 将多层的堆叠变换成为一层，也就是一张图像
x = layers.Conv2D(1,3,
```

```
                padding='same',
                activation='sigmoid')(x)

# 将解码器模型实例化，它将 decoder_input转换为解码后的图像
decoder = Model(decoder_input,x)

z_decoded = decoder(z)
```

VAE 损失函数层定义

在keras中标准的损失函数无法应用到本实例，因为VAE 的双重损失不符合这种形
式。标准的函数具有如下特征：
你可以传递一个现有的损失函数名。该符号函数为每个数据点返回一个标量，有以下两个参数:

- y_true: 真实标签。TensorFlow/Theano 张量。
- y_pred: 预测值。TensorFlow/Theano 张量，其 shape 与 y_true 相同

在keras中很多损失函数都是定义好的，如下：

- mean_squared_error
- squared_hinge
- categorical_crossentropy
- binary_crossentropy

但是这里我们可以通过定义一个自定义的层，并在其内部使用内置的 add_loss 层方法
来创建一个你想要的损失.
具体coding如下：

```
class CustomVariationalLayer(keras.layers.Layer):
    # 定义loss函数
    def vae_loss(self,x,z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)
        # 一个是重构损失
        xent_loss = keras.metrics.binary_crossentropy(x,z_decod
ed)
        # 另一个是正则化损失
        kl_loss = -5e-4 * K.mean(
            1+z_log_var - K.square(z_mean) - K.exp(z_log_var),a
xis=-1)
        return K.mean(xent_loss+kl_loss)

    # 通过编写一个 call 方法来实现自定义层
    def call(self,inputs):
        x = inputs[0]
        z_decoded = inputs[1]
```

```
        loss = self.vae_loss(x,z_decoded)
        self.add_loss(loss,inputs=inputs)
        # 我们不使用这个输出，但层必须要有返回值
        return x
y = CustomVariationalLayer()([input_img,z_decoded])
```

VAE 训练

将模型实例化并开始训练。因为损失包含在自定义层中，所以在编译时无须指定外部损失（即 loss=None），这意味着在训练过程中不需要传入目标数据.

```
from keras.datasets import mnist

vae = Model(input_img,y)
vae.compile(optimizer='rmsprop',loss = None)
vae.summary()

(x_train,_),(x_test,y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape + (1,))

vae.fit(x=x_train, y=None,
        shuffle=True,
        epochs=10,
        batch_size=batch_size,
        validation_data=(x_test, None))
_________________________________________________________________
_____________________________________
Layer (type)                 Output Shape              Param #
  Connected to
=================================================================
=====================================
input_1 (InputLayer)         (None, 28, 28, 1)         0


_________________________________________________________________
_____________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320
  input_1[0][0]

_________________________________________________________________
_____________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
  conv2d_1[0][0]
```

```
__________________________________________________________________________
__________________________________
conv2d_3 (Conv2D)               (None, 14, 14, 64)   36928
  conv2d_2[0][0]
__________________________________________________________________________
__________________________________
conv2d_4 (Conv2D)               (None, 14, 14, 64)   36928
  conv2d_3[0][0]
__________________________________________________________________________
__________________________________
flatten_1 (Flatten)             (None, 12544)        0
  conv2d_4[0][0]
__________________________________________________________________________
__________________________________
dense_1 (Dense)                 (None, 32)           401440
  flatten_1[0][0]
__________________________________________________________________________
__________________________________
dense_2 (Dense)                 (None, 2)            66
  dense_1[0][0]
__________________________________________________________________________
__________________________________
dense_3 (Dense)                 (None, 2)            66
  dense_1[0][0]
__________________________________________________________________________
__________________________________
lambda_1 (Lambda)               (None, 2)            0
  dense_2[0][0]

  dense_3[0][0]
__________________________________________________________________________
__________________________________
model_3 (Model)                 (None, 28, 28, 1)    56385
  lambda_1[0][0]
__________________________________________________________________________
__________________________________
custom_variational_layer_2 (Cus [(None, 28, 28, 1),  0
  input_1[0][0]

  model_3[1][0]
==========================================================================
==================================
Total params: 550,629
Trainable params: 550,629
Non-trainable params: 0
__________________________________________________________________________
```

```
_________________________________

Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 32s 529us/step -
 loss: 0.2154 - val_loss: 0.2006
Epoch 2/10
60000/60000 [==============================] - 28s 463us/step -
 loss: 0.1973 - val_loss: 0.1934
Epoch 3/10
60000/60000 [==============================] - 27s 451us/step -
 loss: 0.1927 - val_loss: 0.1914
Epoch 4/10
60000/60000 [==============================] - 28s 460us/step -
 loss: 0.1900 - val_loss: 0.1884
Epoch 5/10
60000/60000 [==============================] - 27s 455us/step -
 loss: 0.1879 - val_loss: 0.1862
Epoch 6/10
60000/60000 [==============================] - 27s 456us/step -
 loss: 0.1863 - val_loss: 0.1855
Epoch 7/10
60000/60000 [==============================] - 27s 450us/step -
 loss: 0.1851 - val_loss: 0.1853
Epoch 8/10
60000/60000 [==============================] - 28s 464us/step -
 loss: 0.1840 - val_loss: 0.1830
Epoch 9/10
60000/60000 [==============================] - 27s 455us/step -
 loss: 0.1832 - val_loss: 0.1840
Epoch 10/10
60000/60000 [==============================] - 27s 452us/step -
 loss: 0.1825 - val_loss: 0.1826
```

```
<keras.callbacks.History at 0x172b2f1ab00>
```

处理效果

一旦训练好了这样的模型（本例中是在 MNIST 上训练），我们就可以使用 decoder 网络将任意潜在空间向量转换为图像。

11.jpg

采样数字的网格展示了不同数字类别的完全连续分布：当你沿着潜在空间的
一条路径观察时，你会观察到一个数字逐渐变形为另一个数字。这个空间的特定方向具有一定的意义，比如，有一个方向表示“逐渐变为 4”、有一个方向表示“逐渐变为 1”等

```
import matplotlib.pyplot as plt
from scipy.stats import norm

# 我们将显示 15×15 的数字网格（共 255 个数字）
n = 15
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# 使用 SciPy 的 ppf 函数对线性分隔的坐
# 标进行变换，以生成潜在变量 z 的值（因为潜在空间的先验分布是高斯分布）
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        # 将 z 多次重复，以构建一个完整的批量
        z_sample = np.tile(z_sample, batch_size).reshape(batch_
size, 2)
        # 将批量解码为数字图像
        x_decoded = decoder.predict(z_sample, batch_size=batch_
size)
        # 将批量第一个数字的形状从 28×28×1 转变为 28×28
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.show()
```

分享关于人工智能，机器学习，深度学习以及计算机视觉的好文章，同时自己对于这个领域学习心得笔记。想要一起深入学习人工智能的小伙伴一起结伴学习吧！扫码上车！

本文为我原创

深度学习 keras 专栏日更挑战

-- -- -- 分享到： 投诉或建议

评论