ГЛАВА 13

Генерирование изображений с помощью автокодировщиков

В главе 5 мы исследовали возможности генерирования текстов в стиле существующего корпуса, будь то произведения Шекспира или код из стандартной библиотеки Python, а в главе 12 рассмотрели процесс генерирования изображений путем оптимизации активаций в предварительно обученной сети. В этой главе мы объединим обе методики для генерирования изображений на основе предоставленных примеров.

Генерирование изображений на основе примеров — это область активных исследований, в которой ежемесячно совершаются открытия или прорывы. Однако рассмотрение передовых алгоритмов, учитывая сложность моделей, длительность тренировки и объемы необходимых данных, выходит за рамки данной книги. Вместо этого мы будем работать в более узкой области, связанной с обработкой рукописных текстов.

Начнем с рассмотрения набора данных Quick Draw компании Google. Этот набор является результатом онлайн-игры и содержит множество сделанных от руки рисунков. Все они хранятся в векторном формате, поэтому мы преобразуем их в растровые изображения и выберем рисунки с одной и той же меткой: *cat* (кошка).

На основе эскизов кошек мы создадим модель автокодировщика, способную обучиться распознаванию признаков кошек: она сможет преобразовывать рисунок кошки во внутреннее представление и генерировать из него некий похожий образ. С этого набора мы и начнем визуализировать работу сети.

После этого мы переключимся на набор рукописных цифр, а затем перейдем к *вариационным кодировщикам*. Эти сети создают плотные пространства, являющиеся абстрактным представлением их входов; из них мы можем делать выборки данных. Результатом обработки каждой выборки будет реалистичное изображение. Мы даже можем выполнять интерполяцию между точками и наблюдать за тем, как изображения постепенно меняются.

Наконец, мы рассмотрим *условные вариационные автокодировщики*, которые учитывают метку в процессе тренировки и поэтому могут воспроизводить изображения определенного класса в случайной манере.

Используемый в этой главе код содержится в следующих блокнотах Python:

- *13.1 Quick Draw Cat Autoencoder*;
- *13.2 Variational Autoencoder*.

13.1. Получение рисунков из набора Google Quick Draw

Задача

Где можно получить набор простых рисунков, сделанных от руки?

Решение

Используйте набор данных Google Quick Draw.

Google Quick Draw (`https://quickdraw.withgoogle.com/`) — это онлайн-игра, в которой участникам предлагается нарисовать что-нибудь и посмотреть, сможет ли ИИ угадать, какой объект изображен на рисунке. Побочным результатом игры стала большая база данных маркированных рисунков. Компания Google сделала этот набор данных доступным для каждого, кто проводит эксперименты в области машинного обучения.

Данные доступны в нескольких форматах (`https://github.com/googlecreativelab/quickdraw-dataset`). Мы будем работать с упрощенными векторными рисунками в двоичной кодировке. Начнем с получения всех изображений кошек.

```
BASE_PATH = 'https://storage.googleapis.com/quickdraw_dataset/full/binary/'
path = get_file('cat', BASE_PATH + 'cat.bin')
```

Мы соберем все изображения, распаковывая их одно за другим. Изображения хранятся в двоичном векторном формате, который мы будем рисовать на пустом растровом холсте. Изображения начинаются с 15-байтового заголовка, поэтому мы продолжаем их обрабатывать до тех пор, пока не останется менее 15 байт.

```
x = []
with open(path, 'rb') as f:
    while True:
        img = PIL.Image.new('L', (32, 32), 'white')
        draw = ImageDraw.Draw(img)
        header = f.read(15)
        if len(header) != 15:
            break
```

Рисунок — это список штрихов, каждый из которых состоит из ряда координат x и y. Координаты x и y хранятся по отдельности, поэтому их следует упаковать в список для передачи только что созданному объекту `ImageDraw`.

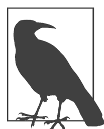
```
strokes, = unpack('H', f.read(2))
for i in range(strokes):
    n_points, = unpack('H', f.read(2))
    fmt = str(n_points) + 'B'
    read_scaled = lambda: (p // 8 for p in unpack(fmt, f.read(n_points)))
    points = [*zip(read_scaled(), read_scaled())]
    draw.line(points, fill=0, width=2)
img = img_to_array(img)
x.append(img)
```

Теперь в нашем распоряжении более сотни тысяч рисунков кошек.

Обсуждение

Сбор генерируемых пользователями данных с помощью игры — интересный способ создания набора данных для машинного обучения. Компания Google использовала этот подход не в первые. Несколько лет назад она уже запускала игру Google Image Labeler (`https://en.wikipedia.org/wiki/Google_Image_Labeler`), в которой два игрока, не знающие друг друга, должны были назначать изображениям метки, получая очки в том случае, если метки совпадали. Однако результаты игры никогда не были сделаны публичными.

Набор данных Quick Draw включает 345 категорий. Мы выбрали лишь изображения котов, но вы можете использовать и другие категории, построив классификатор изображений. У этого набора данных есть недостатки, основным из которых является то, что не все рисунки дорисованы. Игра заканчивалась, когда ИИ распознавал рисунок, и если пользователь пытался изобразить верблюда, то двух горбов для этого вполне могло хватить.



В приведенном решении мы сами растеризовали рисунки. Google предоставляет версию данных в формате массива `numpy`, в котором изображения предварительно растеризованы до размера 28×28 пикселей.

13.2. Создание автокодировщика для изображений

Задача

Существует ли автоматический способ представления изображения, даже если оно не маркировано, в виде вектора фиксированного размера?

Решение

Используйте автокодировщик.

В главе 9 был приведен пример использования сверточной сети для классификации изображений за счет организации ее в виде слоев, последовательно

обрабатывающих изображение на уровне пикселей, затем локальных признаков, на уровне более структурированных признаков и, наконец, абстрактного представления, пригодного для предсказания класса обрабатываемого изображения. В главе 10 мы интерпретировали указанное абстрактное представление как вектор в многомерном семантическом пространстве и использовали тот факт, что взаимно близкие векторы представляют похожие изображения, для создания движка обратного поиска изображений. Наконец, в главе 12 было показано, как визуализировать то, что означают активации нейронов на различных уровнях сверточной сети.

Для того чтобы все это можно было реализовать, нужно иметь маркированные изображения. Лишь благодаря тому что сеть увидела большое количество изображений собак, кошек и множества других объектов, она смогла обучиться их абстрактным представлениям в многомерном пространстве. А как быть в том случае, если изображения не помечены маркерами? Или маркеров недостаточно для того, чтобы сеть могла развить интуицию в отношении различения объектов? В подобных ситуациях нас может выручить автокодировщик.

Лежащая в основе автокодировщиков идея заключается в том, чтобы заставить сеть представлять изображение в виде вектора определенного размера и использовать функцию потерь, вычисляемую на основе того, насколько точно сети удается воспроизвести входное изображение, отталкиваясь от этого представления. Таким образом, входное и выходное изображения должны совпадать, а это означает, что нам не нужны маркированные изображения. В данной ситуации любое изображение будет подходящим.

Структура такой сети очень напоминает то, с чем мы сталкивались раньше. Мы берем оригинальное изображение и используем последовательность сверточных и субдискретизирующих слоев для уменьшения размера и увеличения глубины изображения до тех пор, пока не получим одномерный вектор, являющийся абстрактным представлением этого изображения. Но вместо того чтобы считать дело сделанным и предсказать, чем является изображение, мы продолжаем описанный процесс в обратном порядке и проходим от абстрактного представления через набор слоев *повышающей дискретизации* (upsampling), выполняющих обратное преобразование, до тех пор пока вновь не вернемся к исходному изображению. В качестве функции потерь мы выбираем разницу между входным и выходным изображениями.

```
def create_autoencoder():
    input_img = Input(shape=(32, 32, 1))

    channels = 2
    x = input_img
    for i in range(4):
        channels *= 2
        left = Conv2D(channels, (3, 3),
                      activation='relu', padding='same')(x)
```

```
        right = Conv2D(channels, (2, 2),
                       activation='relu', padding='same')(x)
        conc = Concatenate()([left, right])
        x = MaxPooling2D((2, 2), padding='same')(conc)

    x = Dense(channels)(x)

    for i in range(4):
        x = Conv2D(channels, (3, 3), activation='relu',
                   padding='same')(x)
        x = UpSampling2D((2, 2))(x)
        channels //= 2
    decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

    autoencoder = Model(input_img, decoded)
    autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
    return autoencoder

autoencoder = create_autoencoder()
autoencoder.summary()
```

Архитектуру этой сети можно уподобить песочным часам. Верхний и нижний слои представляют изображения. Самое узкое место находится посередине, и его часто называют *латентным представлением*. В данном случае мы имеем латентное пространство из 128 элементов, т.е. вынуждаем сеть представлять каждое изображение размером 32×32 пикселя, используя 128 вещественных чисел. Единственным способом, с помощью которого сеть может минимизировать разницу между входным и выходным изображениями, является сжатие как можно большего количества информации в это латентное представление.

Как и прежде, для тренировки сети можно использовать следующий код.

```
autoencoder.fit(x_train, x_train,
                epochs=100,
                batch_size=128,
                validation_data=(x_test, x_test))
```

Процесс должен сходиться довольно быстро.

Обсуждение

Автокодировщики — интересный тип нейронной сети, поскольку они способны обучаться компактному представлению своих входов, сжатому с потерями, без какого-либо вмешательства с нашей стороны. В данном разделе мы применили их к изображениям, но они также успешно применяются для обработки текста и других данных в виде временн*ы*х рядов.

Существует несколько интересных расширений идеи автокодировщиков. Одно из них — это *шумоподавляющий автокодировщик* (denoising autoencoder). Суть идеи заключается в том, чтобы сеть предсказывала изображение, отталкиваясь не от самого изображения, а от его поврежденной версии. Например, мы можем добавлять случайный шум во входные изображения. Функция потерь по-прежнему будет сравнивать выход сети с оригинальным (не зашумленным) входом, поэтому сеть будет эффективно обучаться удалению шума из картинок. Эта методика также доказала свою эффективность при восстановлении цветов черно-белых картинок.

В главе 10 мы использовали абстрактное представление изображения для создания движка обратного поиска изображений, но для этого нам нужны были метки. В случае автокодировщиков в этом нет необходимости: мы можем измерять расстояние между изображениями после обучения модели ни на чем ином, как только на наборе изображений. Оказывается, что использование шумоподавляющего автокодировщика позволяет повысить эффективность алгоритма, обеспечивающего сходство изображений. На интуитивном уровне это можно попытаться объяснить тем, что шум указывает сети, на что не следует обращать внимания, аналогично тому, как работает аугментация данных (см. раздел "Предварительная обработка изображений" главы 1).

13.3. Визуализация результатов автокодировщика

Задача

Мы хотим получить представление о том, насколько хорошо работает наш автокодировщик.

Решение

Отберите несколько случайных изображений кошек из входного набора и предоставьте модели возможность предсказать их, после чего визуализируйте вход и выход в виде двух строк.

Предскажем некоторые изображения кошек.

```
cols = 25
idx = np.random.randint(x_test.shape[0], size=cols)
sample = x_test[idx]
decoded_imgs = autoencoder.predict(sample)
```

Затем отобразим их в блокноте (рис. 13.1).

```
def decode_img(tile):
    tile = tile.reshape(tile.shape[:-1])
    tile = np.clip(tile * 400, 0, 255)
    return PIL.Image.fromarray(tile)
```

```
overview = PIL.Image.new('RGB', (cols * 32, 64 + 20), (128, 128, 128))
for idx in range(cols):
    overview.paste(decode_img(sample[idx]), (idx * 32, 5))
    overview.paste(decode_img(decoded_imgs[idx]), (idx * 32, 42))
f = BytesIO()
overview.save(f, 'png')
display(Image(data=f.getvalue()))
```

Рис. 13.1

Как видите, сети удалось уловить базовые формы, но, по-видимому, она чувствует себя не очень уверенно, что проявляется в нечеткости очертаний пиктограмм, которые выглядят почти как тени.

В следующем разделе мы узнаем, можно ли это улучшить.

Обсуждение

Поскольку входное и выходное изображения автокодировщика *должны* быть похожими, наилучший способ проверить работу сети — выбрать случайные пиктограммы из нашего валидационного набора и попросить сеть реконструировать их. Раньше мы уже видели пример использования библиотеки PIL для создания изображения в виде двух строк и его отображения в блокноте.

Одной из проблем такого подхода является то, что используемая нами функция потерь вынуждает сеть смазывать свой выход. Входные рисунки состоят из тонких линий, чего нельзя сказать о выходных рисунках. У нашей модели нет стимулов для предсказания тонких линий, поскольку она не уверена в точном расположении их позиций, поэтому она перестраховывается и проводит размытые линии. Это повышает шансы того, что по крайней мере некоторые пиксели будут корректно захвачены линией. Для устранения указанного недостатка мы можем попытаться спроектировать функцию потерь таким образом, чтобы ограничивать сеть в количестве прорисовываемых ею пикселей или назначать премию за четкие линии.

13.4. Выборка изображений из нормального распределения

Задача

Как убедиться в том, что каждая точка вектора представляет разумное изображение?

Решение

Используйте *вариационный* автокодировщик.

Автокодировщики обеспечивают интересный способ представления изображения в виде вектора, размер которого значительно меньше размера самого изображения. Однако пространство этих векторов не является плотным, т.е. для каждого изображения имеется вектор в этом пространстве, но не каждый вектор представляет разумное изображение. Разумеется, декодер автокодировщика создаст изображение из любого вектора, но большинство таких изображений не будет представлять никаких узнаваемых образов. Вариационные автокодировщики позволяют устранить данный недостаток.

В этом и следующем разделах мы будем работать с набором рукописных цифр, включающим 60000 тренировочных и 10000 тестовых образцов из базы данных MNIST. Описанный подход работает и для пиктограмм, но он усложняет модель, и для достижения хороших результатов нам потребовалось бы гораздо больше пиктограмм, чем те, которыми мы располагаем. Если вас интересует это, то рабочая модель содержится в каталоге блокнота. Начнем с загрузки данных.

```
def prepare(images, labels):
    images = images.astype('float32') / 255
    n, w, h = images.shape
    return images.reshape((n, w * h)), to_categorical(labels)

train, test = mnist.load_data()
x_train, y_train = prepare(*train)
x_test, y_test = prepare(*test)
img_width, img_height = train[0].shape[1:]
```

Ключевая идея, лежащая в основе вариационного автокодировщика, заключается в том, чтобы добавить в функцию потерь член, представляющий разницу в статистическом распределении изображений и абстрактных представлений. Для этого применим *расстояние Кульбака — Лейблера*. Мы можем использовать эту метрику для измерения расстояний в пространстве вероятностных распределений, хотя с технической точки зрения она не является мерой расстояния. Те, кто интересуется математическими выкладками, могут прочитать об этом в Википедии:

`https://ru.wikipedia.org/wiki/Расстояние_Кульбака_—_Лейблера`

Базовая структура нашей модели аналогична той, которая использовалась в предыдущем разделе. Мы начинаем с входного представления пикселей, пропускаем его через скрытые слои и понижаем дискретизацию до представления с очень небольшими размерами. Затем мы повторяем процесс в обратном направлении, пока вновь не вернемся к нашим пикселям.

```
pixels = Input(shape=(num_pixels,))
encoder_hidden = Dense(512, activation='relu')(pixels)
z_mean = Dense(latent_space_depth,
               activation='linear')(encoder_hidden)
z_log_var = Dense(latent_space_depth,
                  activation='linear')(encoder_hidden)
z = Lambda(sample_z, output_shape=(latent_space_depth,))(
        [z_mean, z_log_var])
decoder_hidden = Dense(512, activation='relu')
reconstruct_pixels = Dense(num_pixels, activation='sigmoid')
hidden = decoder_hidden(z)
outputs = reconstruct_pixels(hidden)
auto_encoder = Model(pixels, outputs)
```

Наибольший интерес здесь представляют тензор `z` и лямбда-функция, которая ему присваивается. В этом тензоре будет храниться латентное представление нашего изображения, а для выборки лямбда-функция использует метод `sample_z()`.

```
def sample_z(args):
    z_mean, z_log_var = args
    eps = K.random_normal(shape=(batch_size, latent_space_depth),
                          mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var / 2) * eps
```

Этот метод случайным образом семплирует точки из нормального распределения, используя переменные `z_mean` и `z_log_var`.

Обратимся к нашей функции потерь. Ее первая компонента — это потери реконструкции, которые измеряют разницу между входными и выходными пикселями.

```
def reconstruction_loss(y_true, y_pred):
    return K.sum(K.binary_crossentropy(y_true, y_pred), axis=-1)
```

Мы также включаем в функцию потерь вторую компоненту, которая использует расстояние Кульбака — Лейблера для того, чтобы сместить распределение в подходящем направлении.

```
def KL_loss(y_true, y_pred):
    return 0.5 * K.sum(K.exp(z_log_var) +
                       K.square(z_mean) - 1 - z_log_var,
                       axis=1)
```

Далее просто суммируем обе компоненты.

```
def total_loss(y_true, y_pred):
    return (KL_loss(y_true, y_pred) +
            reconstruction_loss(y_true, y_pred))
```

И компилируем модель с помощью следующего кода.

```
auto_encoder.compile(optimizer=Adam(lr=0.001),
                     loss=total_loss,
                     metrics=[KL_loss, reconstruction_loss])
```

Такая структура кода удобна тем, что позволяет отслеживать обе компоненты по отдельности в процессе тренировки.

Эта модель выглядит усложненной из-за наличия дополнительной компоненты функции потерь и вызова метода `sample_z()`. Чтобы разобраться в деталях, лучше всего заглянуть в код, приведенный в соответствующем блокноте. Далее можно приступить к тренировке модели.

```
cvae.fit(x_train, x_train, verbose = 1, batch_size=batch_size,
         epochs=50, validation_data = (x_test, x_test))
```

По завершении тренировки мы хотим использовать результаты, передавая случайную точку в латентное пространство и наблюдая за тем, какое при этом получается изображение. Это можно осуществить, создав вторую модель, которая будет использовать средний слой модели `auto_encoder` в качестве входа и наше целевое изображение в качестве выхода.

```
decoder_in = Input(shape=(latent_space_depth,))
decoder_hidden = decoder_hidden(decoder_in)
decoder_out = reconstruct_pixels(decoder_hidden)
decoder = Model(decoder_in, decoder_out)
```

Теперь мы можем генерировать случайный вход и преобразовывать его в картинку (рис. 13.2).

```
random_number = np.asarray([[np.random.normal()
                            for _ in range(latent_space_depth)]])
def decode_img(a):
    a = np.clip(a * 256, 0, 255).astype('uint8')
    return PIL.Image.fromarray(a)

decode_img(decoder.predict(random_number).reshape(img_width,
           img_height)).resize((56, 56))
```

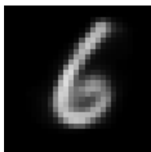


Рис. 13.2

Обсуждение

Вариационные автокодировщики — важное расширение концепции автокодировщиков, если речь идет о генерации изображений, а не просто их воспроизведении.

Убеждаясь в том, что абстрактные представления изображений происходят из плотного пространства, в котором точки, близкие к оригиналу, транслируются в похожие изображения, мы можем генерировать изображения, имеющие то же распределение правдоподобия, что и входные изображения.

Рассмотрение соответствующих математических выкладок выходит за рамки данной книги. Идею можно описать примерно так. Некоторые изображения получаются более или менее "нормальными", а некоторые — довольно неожиданными. Латентное пространство имеет те же характеристики, поэтому точки, извлеченные из близкой окрестности оригинала, соответствуют "нормальным" изображениям, тогда как более удаленные точки транслируются в нечто, мало напоминающее изображения. Выборка из нормального распределения приводит к изображениям, представляющим ту же смесь ожидаемых и неожиданных изображений, которую модель видела в процессе тренировки.

Плотные пространства предоставляют нам очевидные преимущества. Они позволяют выполнять интерполяцию между точками и все равно получать разумные результаты. Например, если нам известно, что одна точка в латентном пространстве транслируется в цифру *6*, а другая — в цифру *8*, то можно ожидать, что точки между ними будут представлять изображения, являющиеся промежуточными результатами морфинга из *6* в *8*. Если же мы обнаружим одно и то же изображение, но в разных стилях, то сможем найти между ними изображения в смешанном стиле или же, пойдя в другом направлении, найти более экстремальный стиль.

В главе 3 мы рассмотрели векторные представления слов, в которых каждому слову сопоставляется вектор, проецирующий данное слово в семантическое пространство, а также соответствующие операции, которые можно выполнять над этими векторами. Несмотря на всю привлекательность идеи интерполяции между двумя словами, в результате которой мы получили бы некое промежуточное слово, рассчитывать на это, как правило, не приходится, поскольку в случае слов пространство не является плотным — нам не удастся найти "мула" между "ослом" и "лошадью". Точно так же мы можем использовать предварительно обученную сеть распознавания изображений для поиска вектора, соответствующего изображению кошки, но не все векторы в окрестности этой точки будут представлять вариации изображения кошки.

13.5. Визуализация пространства вариационного автокодировщика

Задача

Как визуализировать расхождения в изображениях, которые можно сгенерировать из латентного пространства?

Решение

Используйте два измерения из латентного пространства для создания сетки сгенерированных изображений.

Визуализация двух измерений из нашего латентного пространства выполняется достаточно просто. В случае многомерных пространств мы можем сначала попытаться использовать метод t-SNE для перехода к двум измерениям. В этом отношении нам повезло, поскольку в предыдущем разделе мы использовали только два измерения, так что мы можем обойтись плоскостью и транслировать каждую позицию (x, y) в точку в латентном пространстве. А поскольку мы используем нормальное распределение, мы можем рассчитывать на то, что в диапазоне [–1,5, 1,5] появятся содержательные изображения, имеющие смысл.

```
num_cells = 10
overview = PIL.Image.new('RGB',
                         (num_cells * (img_width + 4) + 8,
                          num_cells * (img_height + 4) + 8),
                         (128, 128, 128))
vec = np.zeros((1, latent_space_depth))
for x in range(num_cells):
    vec[:, 0] = (x * 3) / (num_cells - 1) - 1.5
    for y in range(num_cells):
        vec[:, 1] = (y * 3) / (num_cells - 1) - 1.5
        decoded = decoder.predict(vec)
        img = decode_img(decoded.reshape(img_width, img_height))
        overview.paste(img, (x * (img_width + 4) + 6,
                             y * (img_height + 4) + 6))
overview
```

В результате мы получим довольно приличные изображения цифр, которым была обучена сеть (рис. 13.3).

Обсуждение

Транслируя позиции (x, y) в латентное пространство и декодируя результаты в изображения, мы получаем неплохое представление о содержимом нашего пространства. Как видите, это пространство действительно довольно плотное. Не все точки приводят к собственно цифрам: некоторые из них, как и ожидалось, представляют промежуточные формы. Тем не менее нашей модели удается естественным образом распределить цифры по сетке.

Также следует отметить, что вариационный автокодировщик замечательно справился со сжатием изображений. Каждое входное изображение представлено в латентном пространстве всего двумя вещественными числами, тогда как их пиксельные представления используют 28×28 = 784 таких числа. Коэффициент сжатия,

почти достигающий значения 400, намного превышает коэффициент сжатия JPEG. Разумеется, это сжатие с довольно ощутимыми потерями: рукописная цифра *5* после цикла кодирования/декодирования все еще выглядит как рукописная цифра *5* и сохраняет тот же стиль, но на пиксельном уровне соответствие фактически отсутствует. Кроме того, эта форма сжатия сильно зависит от сжимаемых объектов. Она работает только для рукописных цифр, тогда как алгоритм JPEG можно использовать в отношении изображений и фотографий любого типа.

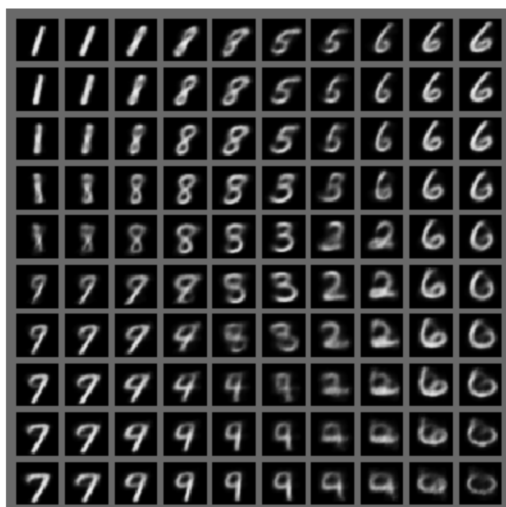


Рис. 13.3

13.6. Условные вариационные автокодировщики

Задача

Как сгенерировать изображения определенного типа, а не просто совершенно случайные изображения?

Решение

Используйте условный вариационный автокодировщик.

Автокодировщик из двух предыдущих разделов отлично справляется с генерированием случайных цифр и способен кодировать цифры, преобразуя их в точки плотного латентного пространства. Но он не может отличить *5* от *3*, и поэтому единственный способ заставить его сгенерировать случайную цифру *3* заключается в том, чтобы сначала найти все “тройки” в латентном пространстве, а затем сделать выборку из этого подпространства. Условные вариационные автокодировщики помогают преодолеть указанное ограничение, получая метку в качестве входа и конкатенируя ее с вектором z латентного пространства модели.

Тем самым преследуются две цели. Во-первых, это позволяет модели учитывать метку при обучении кодированию. Во-вторых, поскольку метка добавляется в латентное пространство, наш декодировщик будет получать как точку латентного пространства, так и метку, что позволяет нам явно запрашивать генерирование определенной цифры. Теперь модель приобретает следующий вид.

```
pixels = Input(shape=(num_pixels,))
label = Input(shape=(num_labels,), name='label')
inputs = concat([pixels, label], name='inputs')

encoder_hidden = Dense(512, activation='relu',
                       name='encoder_hidden')(inputs)
z_mean = Dense(latent_space_depth,
               activation='linear')(encoder_hidden)
z_log_var = Dense(latent_space_depth,
                  activation='linear')(encoder_hidden)
z = Lambda(sample_z,
           output_shape=(latent_space_depth, ))([z_mean, z_log_var])
zc = concat([z, label])

decoder_hidden = Dense(512, activation='relu')
reconstruct_pixels = Dense(num_pixels, activation='sigmoid')
decoder_in = Input(shape=(latent_space_depth + num_labels,))
hidden = decoder_hidden(decoder_in)
decoder_out = reconstruct_pixels(hidden)
decoder = Model(decoder_in, decoder_out)

hidden = decoder_hidden(zc)
outputs = reconstruct_pixels(hidden)
cond_auto_encoder = Model([pixels, label], outputs)
```

Мы тренируем модель, предоставляя ей как изображения, так и метки.

```
cond_auto_encoder.fit([x_train, y_train], x_train, verbose=1,
                      batch_size=batch_size, epochs=50,
                      validation_data = ([x_test, y_test], x_test))
```

Теперь мы можем явно сгенерировать цифру *4* (рис. 13.4).

```
number_4 = np.zeros((1, latent_space_depth + y_train.shape[1]))
number_4[:, 4 + latent_space_depth] = 1
decode_img(cond_decoder.predict(number_4).reshape(img_width, img_height))
```

Рис. 13.4

Поскольку для указания того, какие цифры следует генерировать, используется прямое кодирование, мы также можем запросить генерирование чего-то промежуточного между двумя заданными цифрами.

```
number_8_3 = np.zeros((1, latent_space_depth + y_train.shape[1]))
number_8_3[:, 8 + latent_space_depth] = 0.5
number_8_3[:, 3 + latent_space_depth] = 0.5
decode_img(cond_decoder.predict(number_8_3).reshape(img_width, img_height))
```

Мы действительно получили нечто промежуточное между цифрами *3* и *8* (рис. 13.5).

Рис. 13.5

Другой интересный эксперимент состоит в том, чтобы поместить цифры на ось *y*, а по оси *x* откладывать значения для одного из наших латентных измерений (рис. 13.6).

```
num_cells = 10
overview = PIL.Image.new('RGB',
                         (num_cells * (img_width + 4) + 8,
                          num_cells * (img_height + 4) + 8),
                         (128, 128, 128))
img_it = 0
vec = np.zeros((1, latent_space_depth + y_train.shape[1]))
for x in range(num_cells):
    vec = np.zeros((1, latent_space_depth + y_train.shape[1]))
    vec[:, x + latent_space_depth] = 1
    for y in range(num_cells):
        vec[:, 1] = 3 * y / (num_cells - 1) - 1.5
        decoded = cond_decoder.predict(vec)
        img = decode_img(decoded.reshape(img_width, img_height))
        overview.paste(img, (x * (img_width + 4) + 6,
                             y * (img_height + 4) + 6))
overview
```

Как видите, латентное пространство выражает стиль цифр, который последовательно выдерживается по всем цифрам. Похоже, в данном случае этот стиль контролирует степень наклона.

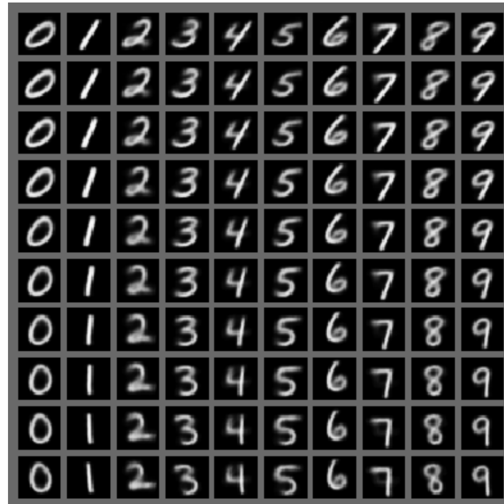


Рис. 13.6

Обсуждение

Условный вариационный автокодировщик является конечным пунктом нашего путешествия по различным автокодировщикам. Этот тип сетей позволяет транслировать цифры на плотное латентное пространство, которое также помечено, что позволяет делать выборки случайных изображений с одновременным указанием того, какого типа они должны быть.

Побочным эффектом предоставления меток сети является то, что теперь ей не нужно обучаться цифрам и она может сосредоточиться только на стиле их написания.