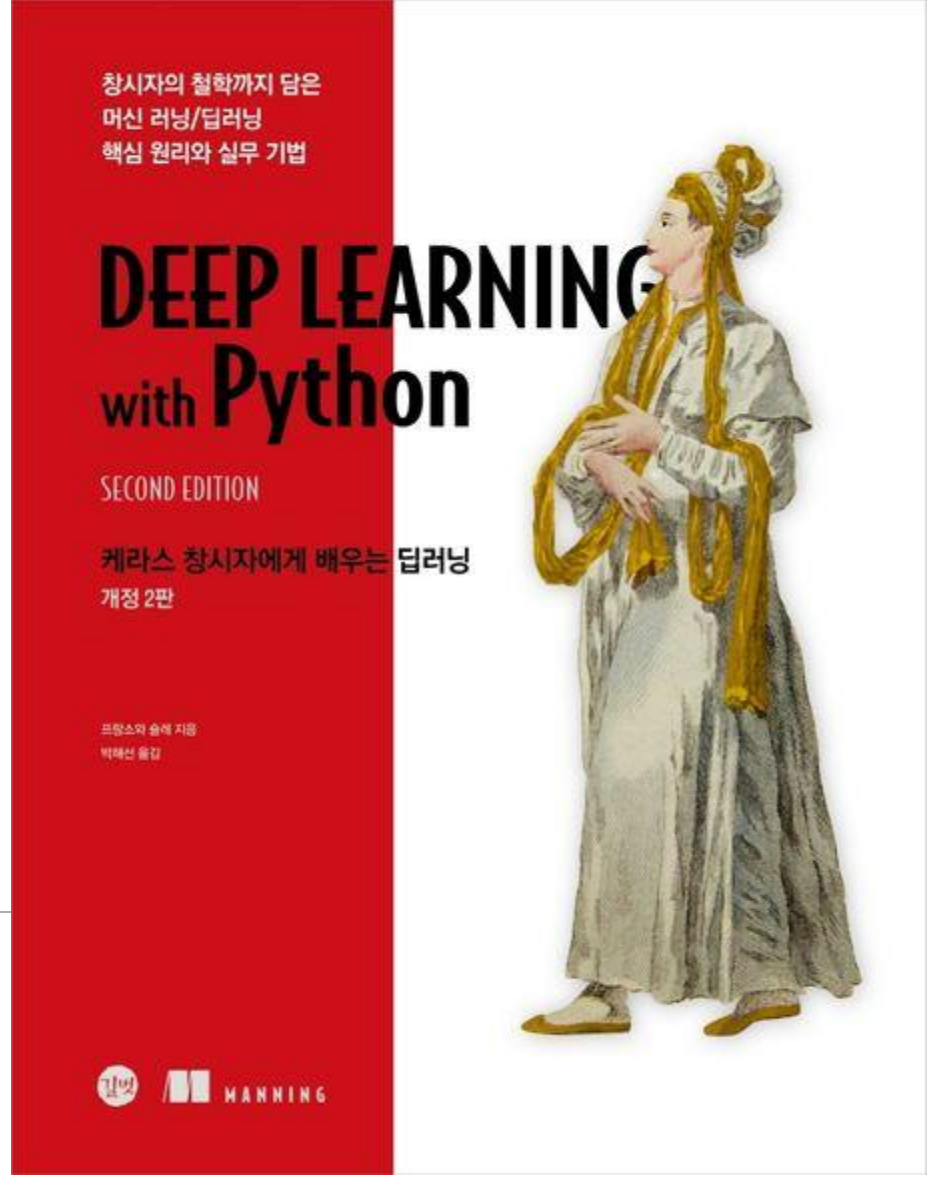
딥러닝

대구가톨릭대학교 AI빅데이터공학과

이 승 민



12章 用于生成模型的深度学习

12.1 文本生成

12.2 DeepDream（深度梦境）

12.3 神经风格迁移（Neural Style Transfer）

12.4 利用变分自编码器（VAE）进行图像生成

12.5 生成对抗网络（GAN）简介

12.6 小结（总结）

12.1 文本生成

12.1 文本生成

文本生成

- 以文本生成为例，但同样的技术也可以用于生成任何类型的序列数据。
- 例如，可以将其应用于音符序列来创作新的音乐，或者应用于连续笔触序列（比如记录画家在 iPad 上绘画的过程），从而实现一笔一划地绘制图像。
- 序列数据生成并不限于艺术内容。
- 还被成功地应用于语音合成和聊天机器人对话功能。
- 例如，谷歌在2016年推出的 Smart Reply（智能回复）也使用了类似的技术，
- 它能够自动生成简短句子，用于电子邮件或短信的自动回复。

12.1 文本生成

序列生成的深度学习模型的简要历史

- 在2014 年后期，即使在机器学习界，也只有少数人知道“LSTM（长短期记忆网络）”这个术语。
- 利用循环神经网络（RNN）成功生成序列数据的应用，直到2016 年才开始进入主流。
- 这项技术（正如我们在第10章中看到的）自1997 年LSTM 算法被提出以来，已有相当长的历史。
- 最初，该算法被用于逐字生成文本。
- 2002 年，Douglas Eck在瑞士Schmidhuber的研究室中首次将LSTM 应用于音乐生成，并取得了有潜力的结果。
- 后来，Eck 成为Google Brain的研究员，并于2016 年创建了“Magenta”研究团队，
- 该团队专注于利用最新的深度学习技术创作出精彩的音乐作品。
- 这也说明——有时，一个好点子的实现可能需要15 年的时间。

12.1 文本生成

用于序列生成的深度学习模型的简要历史

- 在2000年代后期和2010年代初期，Alex Graves（亚历克斯·格雷夫斯）在使用循环神经网络（RNN）生成序列数据方面，做出了极其重要的开创性工作。
- 2013年，他利用记录笔迹位置的时间序列数据，构建了一个结合循环网络与全连接网络的模型，成功生成了与人类书写极为相似的手写文字。这项研究成为了一个转折点。
- 这一特殊的神经网络应用的出现，让人们开始想象"会做梦的计算机"，并且也成为作者（即Keras的开发者）在开发Keras时的重要灵感来源。
- Graves在他2013年上传至arXiv（论文预印本平台）的LaTeX文件中，留下了类似的注释：
- "生成序列数据就像让计算机做梦一样。"如今，几年过去了，我们已经理所当然地接受了这样的应用。
- 但在当时，仅凭Graves的实验，很难仅因为其潜在可能性就断言它会如此具有突破性。
- 在2015年至2017年之间，循环神经网络（RNN）被成功应用于文本与对话生成、音乐生成以及语音合成等领域。

12.1 文本生成

用于序列生成的深度学习模型的简要历史

- 在2017年至2018年左右，Transformer 架构开始在自然语言处理的监督学习任务中，以及在序列生成模型中（尤其是语言建模，即词级文本生成）全面超越循环神经网络（RNN）。
- 最著名的生成型 Transformer 模型之一是拥有1750亿个参数的文本生成模型——GPT-3。
- 该模型由 OpenAI 训练，使用了一个极其庞大的文本语料库，其中包含几乎整个互联网的内容，包括大多数已公开的书籍和维基百科。
- 由于 GPT-3 能够在几乎所有主题上生成看似合理、自然流畅的文本，因此它登上了世界各地的新闻头条。
- 这项技术在短时间内引发了极大的关注，甚至可以说是引来了“史上最炙热的人工智能之夏”。

12.1 文本生成

如何生成序列数据？

- 在深度学习中，生成序列数据的常见方法是：将前面的 token（标记）作为输入，通过 Transformer 或 RNN 来预测序列中接下来的一个或多个 token。
- 例如，若输入为 “the cat is on the”，模型将被训练去预测下一个单词——目标 “mat”。
- 在处理文本数据时，token 通常指的是单词或字母。
- 当给定前面的 token 时，能够建模下一个 token 出现概率的网络被称为
- 语言模型（language model）。
- 语言模型的本质是探索语言的统计结构的潜在空间（latent space）。

12.1 文本生成

如何生成序列数据？

- 当语言模型训练完成后，就可以从该模型中进行采样（sampling），从而生成新的序列。
- 向模型输入一个初始文本字符串（称为条件数据，conditioning data），模型会生成新的字母或单词（有时可以一次生成多个 token）。
- 生成的输出会被重新添加回输入数据中。
- 这个过程会重复多次（见图 12-1）。
- 通过这种重复，模型可以生成反映其所学习数据结构的、长度任意的序列。
- 这样的序列往往与人类撰写的句子极为相似。

12.1 文本生成

▼ 图 12-1 语言模型如何逐步生成序列

图示说明：语言模型如何逐步生成序列

1.初始文本 (Initial Text)

输入为：`The cat sat on the`

2.语言模型 (Language Model)

模型根据输入的上下文计算**下一个单词的概率分布**。

例如，它可能预测：

1. mat → 70%
2. chair → 15%
3. floor → 10%
4. etc.

3.采样策略 (Sampling Strategy)

从概率分布中**选取一个单词**作为输出（例如"mat"）。

4.将生成结果加入输入

新输入变为：`The cat sat on the mat`

5.重复以上过程

模型再一次预测下一个词的概率分布（例如可能生成"which"）。

每次生成的词都会被**添加到输入序列末尾**，从而**逐步构建整句话**。

12.1 文本生成

采样策略的重要性

- 在生成文本时，如何选择下一个字符或单词非常关键。
- 最简单的方法是总是选择概率最高的单词，这种方法称为贪心采样（greedy sampling）。
- 但这种方法会生成重复且可预测的句子，看起来不够自然、不像人类语言。
- 更有趣的做法是引入一定的随机性，让模型做出略微出乎意料的选择。
- 在从下一个单词的概率分布中进行采样时引入随机性，这种方法称为随机采样（stochastic sampling）。（在机器学习中，“stochastic”意味着随机（random）。）
- 使用这种方法时，例如某个单词成为下一个词的概率是 0.3，那么模型大约有 30% 的几率会选择该单词。
- 可以把“贪心采样”看作“随机采样”的一种特殊情况：当一个单词的概率是 1，而其他所有单词的概率都是 0 时。

12.1 文本生成

采样策略的重要性

- 模型的 Softmax 输出 非常适合用于概率采样（stochastic sampling）。
- 有时，模型会选中那些看起来不太可能被采样的单词。
- 这样做虽然这些词未出现在训练数据中，但可以生成更自然、更有趣的新句子，甚至展现出一定的创造性。
- 不过，这种策略也有一个问题：无法控制采样过程中随机性的强弱

12.1 文本生成

采样策略的重要性

- 为什么随机性要有大有小？
- 我们来想一下极端的情况。
- 如果从均匀概率分布中完全随机地抽取下一个单词，那么所有单词的概率都是相同的。
- 这种结构的随机性最大。
- 换句话说，这种概率分布具有最大的熵（entropy）。
- 但显然，这样的结果无法生成有趣或有意义的内容。

12.1 文本生成

采样策略的重要性

- 相反地，贪心采样（greedy sampling） 由于没有随机性，因此也无法生成有趣的内容。
- 贪心采样的概率分布具有最小熵（entropy）。
- 从模型 Softmax 输出的“实际”概率分布 进行采样，位于完全随机采样与完全确定采样之间的中间地带。
- 在这个中间地带，可以尝试不同程度的高或低熵。
- 低熵会生成结构可预测、逻辑性强的句子（看起来更真实）。
- 高熵则会生成更出乎意料、富有创造性的句子。
- 因此，在生成模型中进行采样时，应当尝试调整随机性的强度。
- “有趣”这一特性非常主观，所以无法事先知道哪种熵水平最合适。
- 最终，生成的内容有多有趣，仍需由人来判断。

12.1 文本生成

采样策略的重要性

- 在采样过程中，为了调节概率分布的随机性强弱，引入了一个名为 Softmax 温度（softmax temperature） 的参数。
- 这个参数反映了用于采样的概率分布的熵（entropy）水平。
- 它决定了模型在生成下一个词时，选择“出乎意料的词”还是“更符合预期的词”的程度。
- 给定一个 temperature 值 后，会根据如下公式对原始概率分布（即模型 Softmax 输出）进行重新加权，从而计算出一个新的概率分布。

코드 12-1 다른 온도 값을 사용하여 확률 분포의 가중치 바꾸기[6]

```
import numpy as np

def reweight_distribution(original_distribution, temperature=0.5):
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    return distribution / np.sum(distribution)
```

original_distribution은 전체 합이 1인 1D 넘파이 배열입니다.
temperature는 출력 분포의 엔트로피의 양을 결정합니다.

원본 분포의 가중치를 변경하며 반환합니다. 이 분포의 합은 1이 아닐 수 있으므로 새로운 분포의 합으로 나눕니다.

- `original_distribution`：一个总和为 1 的一维 NumPy 数组，表示模型输出的原始概率分布（例如 Softmax 输出）。
- `temperature`：控制随机性的参数，**决定输出分布的熵（entropy）大小**。

12.1 文本生成

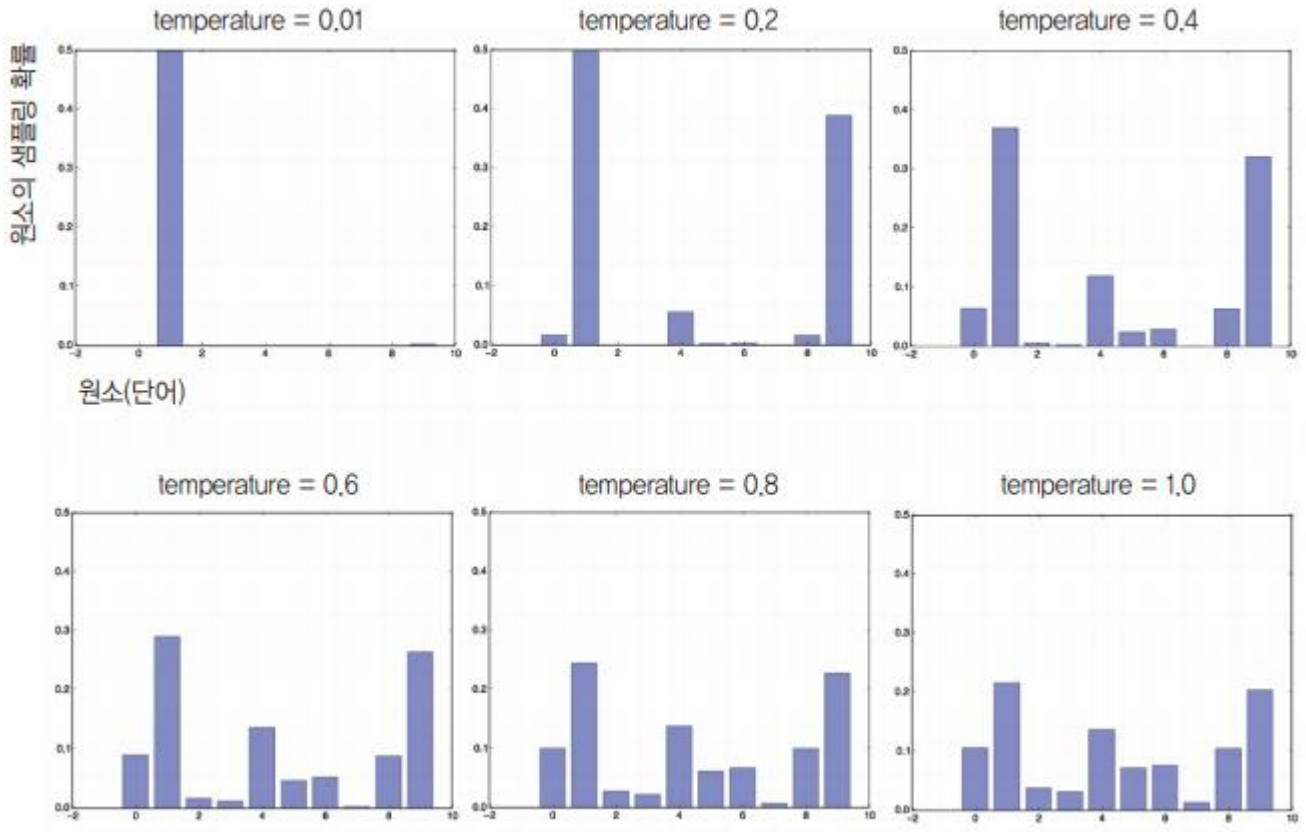
采样策略的重要性

- 高温（High Temperature） 会产生熵更高的采样分布，从而生成更出乎意料、更多样化的数据。
- 相反，低温（Low Temperature） 的随机性较低，因此会生成更可预测、更稳定的数据（如图 12-2 所示）。

12.1 文本生成

▼ 图 12-2 | 对同一个概率分布应用不同温度加权的示例

- **低温（Low Temperature）** → 结果更加**确定（deterministic）**，模型几乎总是选择最有可能的输出。
- **高温（High Temperature）** → 结果具有更强的**随机性（randomness）**，生成内容更加多样化、富有创造性。



12.1 文本生成

使用 Keras 实现文本生成模型

- 我们来用 Keras 实现这种想法吧。
- 首先，要训练一个语言模型，需要大量的文本数据。
- 可以使用像 Wikipedia（维基百科） 或 《指环王 (The Lord of the Rings) 这样非常大的文本文件，或者由多个文本文件组成的语料集合。
- 在这个示例中，我们将继续使用前一章的 IMDB 电影评论数据集，让模型学习如何生成以前从未见过的电影评论。
- 这个语言模型不是对一般英语进行建模，而是学习电影评论的风格与主题。

12.1 文本生成

使用 Keras 实现文本生成模型

数据准备

- 与上一章相同，先下载并解压 **IMDB 电影评论数据集**。

코드 12-2 IMDB 영화 리뷰 데이터셋 내려받아 압축 풀기

```
!wget https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
```

12.1 文本生成

使用 Keras 实现文本生成模型

- 现在我们对这个数据结构应该已经很熟悉了。
- 在 aclImdb 文件夹中包含两个子文件夹。
- 一个文件夹存放负面电影评论，另一个文件夹存放正面电影评论。
- 每条评论都是一个独立的 文本文件（.txt）。
- 我们将使用 label_mode=None 选项调用 text_dataset_from_directory 函数，来读取每个文件并返回其文本内容，从而构建一个纯文本数据集（dataset）。

코드 12-3 텍스트 파일(한 파일=한 샘플)에서 데이터셋 만들기

```
import tensorflow as tf
from tensorflow import keras

dataset = keras.utils.text_dataset_from_directory(
    directory="aclImdb", label_mode=None, batch_size=256)
dataset = dataset.map(lambda x: tf.strings.regex_replace(x, "<br />", " "))
```

이 리뷰에 많이 등장하는 〈br /〉 HTML 태그를 제거합니다. 텍스트 분류 작업에서는 중요하지 않지만 이 예제에서는 〈br /〉 태그를 생성하고 싶지 않기 때문입니다!

12.1 文本生成

使用 Keras 实现文本生成模型

- 现在使用 TextVectorization 层 来创建本示例要使用的词汇表（vocabulary）。
- 每条评论中只使用前 sequence_length 个单词。
- 换句话说，当 TextVectorization 层 将文本向量化时，会自动截断超过指定长度的评论内容。

코드 12-4 TextVectorization 층 준비하기

```
from tensorflow.keras.layers import TextVectorization

sequence_length = 100
vocab_size = 15000
text_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
text_vectorization.adapt(dataset)
```

가장 자주 등장하는 1만 5,000개 단어만 사용하겠습니다.
그 외 단어는 모두 OOV 토큰인 "[UNK]"로 처리합니다.

정수 단어 인덱스의 시퀀스를 반환하도록 설정합니다.

길이가 100인 입력과 타깃을 사용합니다(타깃은 한 스텝 차이가 나기 때문에 실제로 모델은 99개의 단어 시퀀스를 보게 됩니다).

```
from tensorflow.keras.layers import TextVectorization

sequence_length = 100     # 每个文本序列保留前100个单词
vocab_size = 15000        # 仅使用最常出现的15,000个单词
                          # 其余单词会被标记为 [UNK]（未知词）

text_vectorization = TextVectorization(
    max_tokens=vocab_size,            # 词汇表最大容量
    output_mode="int",                # 输出为整数索引序列
    output_sequence_length=sequence_length # 每个序列固定为100个单词长度
)

# 让该层学习数据集中的词汇分布
text_vectorization.adapt(dataset)
```

12.1 文本生成

使用 Keras 实现文本生成模型

- 使用刚才创建的 TextVectorization 层 来构建语言建模数据集。
- 输入样本是向量化后的文本（vectorized text），而目标（target）则是比输入提前一个时间步的同一文本。

코드 12-5 언어 모델링 데이터셋 만들기

```
def prepare_lm_dataset(text_batch):                    텍스트(문자열)의 배치를 정수 시퀀스의 배치로 변환합니다.
    vectorized_sequences = text_vectorization(text_batch) --------
    x = vectorized_sequences[:, :-1] -------- 시퀀스의 마지막 단어를 제외한 입력을 만듭니다.
    y = vectorized_sequences[:, 1:] -------- 시퀀스의 첫 단어를 제외한 타깃을 만듭니다.
    return x, y

lm_dataset = dataset.map(prepare_lm_dataset, num_parallel_calls=4)
```

```
def prepare_lm_dataset(text_batch):
    vectorized_sequences = text_vectorization(text_batch)
    # 将文本批次（字符串序列）转换为整数序列

    x = vectorized_sequences[:, :-1]
    # 输入：去掉每个序列的最后一个单词
    y = vectorized_sequences[:, 1:]
    # 目标：去掉每个序列的第一个单词（即向右偏移一位）

    return x, y

# 将该函数应用到整个数据集上，并行处理以加快速度
lm_dataset = dataset.map(prepare_lm_dataset, num_parallel_calls=4)
```

12.1 文本生成

使用 Keras 实现文本生成模型

—— 基于 Transformer 的序列到序列（Sequence-to-Sequence）模型

- 训练一个模型，用于在给定几个初始单词的情况下，预测句子中下一个单词的概率分布。
- 在训练过程中，向模型输入初始句子，然后对下一个单词进行采样，并将该单词添加到句子末尾，如此循环，直到生成一段完整的文本。
- 就像第 10 章“温度预测问题”中那样，模型接收一个包含 N 个单词 的序列作为输入，并预测第 N+1 个单词。
- 从“序列生成”的角度来看，这种方法仍然存在一些需要注意的问题。

12.1 文本生成

使用 Keras 实现文本生成模型

- 首先， 该模型虽然是通过 N 个单词 来学习预测下一个单词的方法，但它也必须能够在少于 N 个单词的情况下开始进行预测。
- 否则，就会出现一个限制——生成时必须提供一个较长的起始句子（在本例中 N=100 个单词）。
- 第 10 章中没有这样的要求。

12.1 文本生成

使用 Keras 实现文本生成模型

- 第二，训练中使用的许多序列是重复重叠的。
- 以 N = 4 为例：句子"A complete sentence must have, at minimum, three things: a subject, verb and an object"
- 将被分割成以下训练序列：

"A complete sentence must"

"complete sentence must have"

"sentence must have at"

（依次继续到）"verb and an object"

12.1 文本生成

使用 Keras 实现文本生成模型

- 将这些序列作为独立样本处理的模型，大多需要重复编码之前已经处理过的序列，因此会产生大量冗余计算。
- 这是因为早期的训练样本数量并不多，而且所使用的模型（例如密集网络或卷积网络）在设计上必须每次都重新处理完整输入序列。
- 为了缓解这种重复问题，可以在连续的两个样本之间跳过若干个单词，即采用“步进式采样（stride sampling）”的方式来生成训练序列。
- 不过，这种方法并不完美，因为它会导致训练样本数量减少。

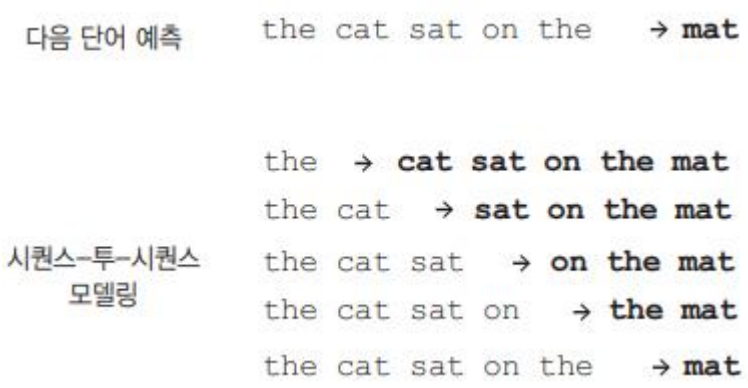
12.1 文本生成

使用 Keras 实现文本生成模型

- 为了解决前面提到的两个问题，采用序列到序列（sequence-to-sequence）模型。
- 即：将由 N 个单词组成的序列（从 0 到 N） 输入模型，并让模型预测下一步的序列（从 1 到 N+1）。
- 使用因果遮罩（causal masking），使得模型在预测第 i+1 个单词时，只能使用从第 0 个到第 i 个单词的信息。
- 这意味着虽然训练样本之间仍存在重叠，但模型可以同时学习 N 个不同位置的预测任务。
- 换句话说，模型会在每个位置 i（$1 \le i \le N$）上，学习如何根据前 i 个单词来预测下一个单词（参见图 12-3）。
- 在文本生成阶段，即使只输入一个单词，模型也能生成关于下一个单词概率分布的预测结果。

12.1 文本生成

图 12-3：与普通的“下一个词预测”相比，序列到序列模型同时优化多个预测任务。



12.1 文本生成

使用 Keras 实现文本生成模型

- 可以在第10章的温度预测问题中使用类似的序列到序列（Seq2Seq）设定
- 可以采用与第10章温度预测问题类似的 序列到序列设置（sequence-to-sequence setup）。
- 当输入由 120小时的数据点 组成的序列时，模型将学习生成从第24小时之后到第120小时为止的温度序列。
- 这不仅能解决原始预测问题，还能同时解决 119个子问题：
- 当给定第 i（$1 \le i < 120$）个时间点的数据时，模型预测 24小时后的温度。
- 如果将第10章中的 RNN 以序列到序列的方式重新训练，结果会相似，但略微差一些。
- 这是因为在同一个模型中要求它同时解决 119个额外的预测任务，会稍微干扰到模型对主要目标（目标时间点温度预测）的专注。

12.1 文本生成

使用 Keras 实现文本生成模型

- 序列到序列模型中的解码机制（Decoder 机制）将 源序列（source sequence） 输入编码器（encoder），编码后的序列与目标序列（target sequence）一起传入解码器（decoder），让解码器预测目标序列中下一步的内容。
- 但在 文本生成（text generation） 中，实际上没有源序列（source sequence）。
- 只有“过去的 token”（前面的词），模型要根据这些已生成的 token 来预测目标序列中的下一个 token。
- 这项任务只需要使用解码器（decoder）部分即可完成。
- 由于使用了 因果遮罩（causal masking），解码器在预测第 N+1 个词时，只能看到第 0~N 个词，不会“偷看”后面的词。

12.1 文本生成

使用 Keras 实现文本生成模型

- 让我们来构建模型吧
- 将会复用第11章中已经实现的两个核心组件：PositionalEmbedding（位置嵌入层）与TransformerDecoder（Transformer 解码器）。

코드 12-6 간단한 트랜스포머 기반 언어 모델

```
from tensorflow.keras import layers
embed_dim = 256
latent_dim = 2048
num_heads = 2

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(loss="sparse_categorical_crossentropy", optimizer="rmsprop")
```

출력 시퀀스 타임스텝마다 가능한 어휘 사전의 단어에 대해 소프트맥스 확률을 계산합니다.

12.1 文本生成

使用可变温度采样的文本生成回调（Callback）

- 使用 回调函数（callback），在每个 epoch（训练轮次）结束时，以不同的温度（temperature）生成文本。
- 可以观察模型在训练过程中生成的文本如何逐渐改善，以及温度变化对采样策略（sampling strategy）的影响。
- 初始输入词设为 “this movie”。

코드 12-7 텍스트 생성 콜백

```
import numpy as np

tokens_index = dict(enumerate(text_vectorization.get_vocabulary()))

def sample_next(predictions, temperature=1.0):
    predictions = np.asarray(predictions).astype("float64")
    predictions = np.log(predictions) / temperature
    exp_preds = np.exp(predictions)
    predictions = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, predictions, 1)
    return np.argmax(probas)
```

단어 인덱스를 문자열로 매핑하는 딕셔너리입니다. 텍스트 디코딩에 사용합니다.

어떤 확률 분포에 대한 가변 온도 샘플링을 구현합니다.

•将词汇表中的索引（index）与单词（token）对应，形成一个字典。
•在文本生成（decoding）过程中，用来将模型输出的数字索引重新映射为实际单词。

12.1 文本生成

使用可变温度采样的文本生成回调

```
class TextGenerator(keras.callbacks.Callback):
    def __init__(self,
                 prompt, -------- 텍스트 생성을 위한 시작 문장입니다.
                 generate_length, -------- 생성할 단어 개수
                 model_input_length,
                 temperatures=(1.,), -------- 샘플링에 사용할 온도 범위
                 print_freq=1):
        self.prompt = prompt
        self.generate_length = generate_length
        self.model_input_length = model_input_length
        self.temperatures = temperatures
        self.print_freq = print_freq
```

```
class TextGenerator(keras.callbacks.Callback):
  def __init__(self,
          prompt,            # 文本生成的起始句子
          generate_length,      # 要生成的单词数
          model_input_length,    # 模型输入序列的最大长度
          temperatures=(1.,),    # 采样时使用的温度范围（可多组）
          print_freq=1):       # 生成频率（每几轮打印一次）
    self.prompt = prompt
    self.generate_length = generate_length
    self.model_input_length = model_input_length
    self.temperatures = temperatures
    self.print_freq = print_freq
```

12.1 文本生成

使用可变温度采样的文本生成回调

```
def on_epoch_end(self, epoch, logs=None):
    if (epoch + 1) % self.print_freq != 0:
        return
    for temperature in self.temperatures:
        print("== Generating with temperature", temperature)
        sentence = self.prompt ------- 시작 단어에서부터 텍스트를 생성합니다.
        for i in range(self.generate_length):
            tokenized_sentence = text_vectorization([sentence]) ┐ 현재 시퀀스를 모델에
            predictions = self.model(tokenized_sentence)        ┘ 주입합니다.
            next_token = sample_next(predictions[0, i, :]) ┐ 마지막 타임스텝의 예측을 추출하여
            sampled_token = tokens_index[next_token]       ┘ 다음 단어를 샘플링합니다.
            sentence += " " + sampled_token ------- 새로운 단어를 현재 시퀀스에 추가하고 반복합니다.
        print(sentence)
```

```
def on_epoch_end(self, epoch, logs=None):
  if (epoch + 1) % self.print_freq != 0:
    return

  for temperature in self.temperatures:
    print("== Generating with temperature", temperature)
    sentence = self.prompt  # 从起始文本开始生成

    for i in range(self.generate_length):
      tokenized_sentence = text_vectorization([sentence])  # 将句子向量化
      predictions = self.model(tokenized_sentence)       # 让模型预测下一个词的概率分布
      next_token = sample_next(predictions[0, i, :], temperature)  # 按温度采样下一个单词
      sampled_token = tokens_index[next_token]          # 将索引映射回单词
      sentence += " " + sampled_token                   # 将生成的单词拼接到句子后面

    print(sentence)
```

12.1 文本生成

使用可变温度采样的文本生成回调

```
prompt = "This movie"
text_gen_callback = TextGenerator(

prompt,
generate_length=50,
model_input_length=sequence_length,
temperatures=(0.2, 0.5, 0.7, 1., 1.5))
```

텍스트 샘플링에 다양한 온도를 사용하여 텍스트 생성에 미치는 온도의 영향을 확인하겠습니다.

我们将使用不同的温度值进行文本采样，以观察温度对文本生成的影响。

12.1 文本生成

使用可变温度采样的文本生成回调

- 现在我们来调用 fit() 方法

코드 12-8 언어 모델 훈련하기

```
model.fit(lm_dataset, epochs=200, callbacks=[text_gen_callback])
```

12.1 文本生成

使用可变温度采样的文本生成回调

- 以下示例是模型经过 200 次 epoch 训练后 所生成的文本中选取的部分。
- 由于标点符号 未被包含在词汇表（vocabulary）中，因此生成的所有文本都 不含任何标点符号。

12.1 文本生成

使用可变温度采样的文本生成回调

- temperature=0.2
  - “this movie is a [UNK] of the original movie and the first half hour of the movie is pretty good but it is a very good movie it is a good movie for the time period”
  - “this movie is a [UNK] of the movie it is a movie that is so bad that it is a [UNK] movie it is a movie that is so bad that it makes you laugh and cry at the same time it is not a movie i dont think ive ever seen”

12.1 文本生成

使用可变温度采样的文本生成回调

- temperature=0.5
  - “this movie is a [UNK] of the best genre movies of all time and it is not a good movie it is the only good thing about this movie i have seen it for the first time and i still remember it being a [UNK] movie i saw a lot of years”
  - “this movie is a waste of time and money i have to say that this movie was a complete waste of time i was surprised to see that the movie was made up of a good movie and the movie was not very good but it was a waste of time and”

12.1 文本生成

使用可变温度采样的文本生成回调

- temperature=0.7
  - “this movie is fun to watch and it is really funny to watch all the characters are extremely hilarious also the cat is a bit like a [UNK] [UNK] and a hat [UNK] the rules of the movie can be told in another scene saves it from being in the back of ”
  - “this movie is about [UNK] and a couple of young people up on a small boat in the middle of nowhere one might find themselves being exposed to a [UNK] dentist they are killed by [UNK] i was a huge fan of the book and i havent seen the original so it”

12.1 文本生成

使用可变温度采样的文本生成回调

- temperature=1.0
  - “this movie was entertaining i felt the plot line was loud and touching but on a whole watch a stark contrast to the artistic of the original we watched the original version of england however whereas arc was a bit of a little too ordinary the [UNK] were the present parent [UNK]”
  - “this movie was a masterpiece away from the storyline but this movie was simply exciting and frustrating it really entertains friends like this the actors in this movie try to go straight from the sub thats image and they make it a really good tv show”

12.1 文本生成

使用可变温度采样的文本生成回调

- temperature=1.5
  - “this movie was possibly the worst film about that 80 women its as weird insightful actors like barker movies but in great buddies yes no decorated shield even [UNK] land dinosaur ralph ian was must make a play happened falls after miscast [UNK] bach not really not wrestlemania seriously sam didnt exist”
  - “this movie could be so unbelievably lucas himself bringing our country wildly funny things has is for the garish serious and strong performances colin writing more detailed dominated but before and that images gears burning the plate patriotism we you expected dyan bosses devotion to must do your own duty and another”

12.1 文本生成

使用可变温度采样的文本生成回调

- 如这里所示，较低温度会生成非常单调且重复的文本。
- 因此，生成过程有时可能会陷入循环（loop）中。
- 较高温度下生成的文本通常更有趣、令人惊讶且富有创造性。
- 但在温度过高的情况下，局部结构会崩溃，输出结果看起来几乎是随机的。
- 在此实验中，较好的生成温度约为 0.7。
- 应当始终尝试多种采样策略来比较效果！
- 如果能在学习到的结构与随机性之间取得良好平衡，就能生成既合理又富有创造力的文本。“

12.1 文本生成

使用可变温度采样的文本生成回调

- 如果用更多的数据训练更大更深的模型，就能生成比本例中更加逻辑清晰、贴近真实的文本样本。
- GPT-3 是这方面的一个很好的例子（GPT-3 实际上与本例所训练的模型原理相同，只是堆叠了更多的 Transformer 解码器层，并使用了规模更大的训练数据集）。
- 除了“偶然”或“神奇”般的解读之外，不要期待模型能真正理解并生成有意义的文本。
- 它只是一个统计模型，在连续排列单词时根据概率分布进行采样而已。
- 语言本身是形式化的，并不具有真实的“实体”存在。

12.1 文本生成

使用可变温度采样的文本生成回调

- 自然语言的作用有很多种
- 它既是交流的媒介、也是影响世界的方式，是社会的润滑剂，也是整理、储存与检索思想的工具。
- 语言在这些用途中的使用，正是语言意义开始形成的地方。
- 但深度学习中的“语言模型（Language Model）”——尽管名字里有“语言”，却无法感知语言的这些根本特征。
- 它不能交流（没有交流的对象或人），不能影响世界（没有行动者或意图），不具备社会性，也没有任何思考过程，只是通过词语进行概率上的符号处理。
- 语言是心灵的操作系统，因此，若要让语言具有意义，就必须存在一个使用语言的“心”或“意识”。

12.1 文本生成

使用可变温度采样的文本生成回调

- 语言模型的工作，是去捕捉人类在生活中使用语言时所创造的可观测人工制品（如：书籍、在线电影评论、推文等）中的统计结构。
- 这些人工制品之所以具有统计结构，完全是人类使用语言方式的“副产品”。
- 让我们做一个思维实验：

如果人类的语言像计算机压缩数字通信那样高效地压缩了信息，会怎样呢？

- 那么，语言的意义不会减少，依然可以实现各种交流目的。
- 但同时，语言中那些独特的统计结构将会消失，因此，就无法再训练出像现在这样的语言模型

12.1 文本生成

整理

- 当给定前一个 token（标记） 时，可以通过训练一个模型来预测下一个（或多个）token，从而生成序列型数据（sequence data）。
- 在文本领域中，这类模型被称为 “语言模型（Language Model）”。
- 训练时既可以以单词为单位，也可以以字符为单位。
- 在对下一个token进行采样时，需要在模型输出的确定性（集中性）与随机性（多样性）之间找到平衡。
- 为此，使用了Softmax温度（Temperature）的概念。
- 应当尝试多种不同温度值，以找到最合适的参数，使生成结果自然且富有变化。

12.2 DeepDream（深度梦境）

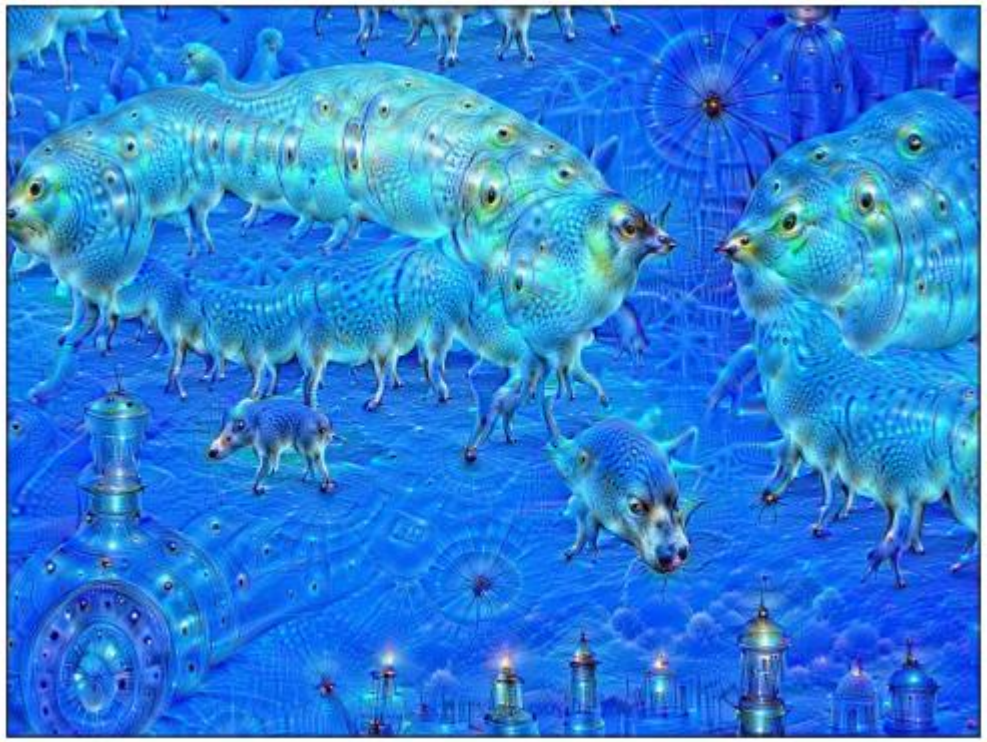
12.2 DeepDream（深度梦境）

深度梦境

- DeepDream 是一种利用卷积神经网络（CNN）所学到的特征表示来进行艺术化图像操作的技术。
- 由 Google 于 2015 年夏季首次发布，使用的是当时的 Caffe 深度学习框架（比 TensorFlow 公开发布早几个月）。
- DeepDream 生成的那些梦幻般的图像在网络上引起了轰动（如图 12-4 所示）。
- 这些图像往往包含被算法扭曲的奇异形态——例如羽毛、狗的眼睛、喙状结构等。
- 这种 DeepDream 模型使用的是在 ImageNet 数据集上训练的卷积神经网络（ConvNet），该数据集中包含了各种狗和鸟的类别，因此生成的梦境中充满了这些元素。

12.2 DeepDream（深度梦境）

▼ 图 12-4 深度梦境导出的图像



12.2 DeepDream（深度梦境）

深度梦境

- 深度梦算法几乎与第9章中介绍的卷积网络滤波器可视化技术相同，通过反向执行卷积网络滤波器来进行。
- 为了最大化卷积网络上层中某些滤波器的激活，使用梯度上升法应用于网络的输入。
- 除了一些小的差异，深度梦也使用了相同的思想。
  - 在深度梦中，目标不是最大化特定滤波器的激活，而是最大化整个层的激活。
    - 许多滤波器的特征被组合在一起进行可视化。
  - 许多滤波器的特征被组合在一起进行可视化。结果是，图像的元素根据现有的视觉模式被扭曲成一种有点艺术性的风格
  - 输入图像通过不同的尺度（称为“八度（octave）”）进行处理，以提高视觉质量。

12.2 DeepDream（深度梦境）

Keras 深度梦境实现

- 首先准备要在深度梦境中使用的测试图像。
- 使用的是北加利福尼亚海岸的冬季照片，岩石较多的区域（图12-5）。

코드 12-9 테스트 이미지 내려받기

```
from tensorflow import keras
import matplotlib.pyplot as plt

base_image_path = keras.utils.get_file(
    "coast.jpg", origin="https://img-datasets.s3.amazonaws.com/coast.jpg")

plt.axis("off")
plt.imshow(keras.utils.load_img(base_image_path))
```

12.2 DeepDream（深度梦境）

▼ 图 12-5 测试图片

12.2 DeepDream（深度梦境）

Keras 深度梦境实现

- 接下来需要预先训练好的卷积神经网络（ConvNet）。
- Keras中有很多可以使用的卷积神经网络，如VGG16，VGG19，Xception，ResNet50等。
- 这些网络都提供了在ImageNet上训练的权重。
- 无论使用哪一个网络，都可以实现DeepDream。
- 当然，选择哪个卷积神经网络会影响生成的可视化效果。
- 因为不同卷积神经网络结构学习到的特征各不相同。
- 原本DeepDream使用的卷积神经网络是Inception模型，实际上Inception能够生成很棒的DeepDream图像。

12.2 DeepDream（深度梦境）

Keras 深度梦境实现

- 这里同样使用了Keras的Inception V3模型

코드 12-10 사전 훈련된 InceptionV3 모델 로드하기

```
from tensorflow.keras.applications import inception_v3

model = inception_v3.InceptionV3(weights="imagenet", include_top=False)
```

12.2 DeepDream（深度梦境）

Keras深度梦实现

- 使用预训练的卷积神经网络（Convolutional Neural Network，CNN），如代码12-11所示，创建一个特性提取模型，返回不同中间层的激活值。
- 在梯度上升法的阶段中，为了最大化损失函数对每一层贡献的加权，选择一个标量值。
- 如果想选择其他层，可以参考在model.summary()中提供的所有层的名称。

12.2 DeepDream（深度梦境）

Keras深度梦实现

코드 12-11 딥드림 손실에 대한 각 층의 기여도 설정하기

```
layer_settings = {
    "mixed4": 1.0,
    "mixed5": 1.5,
    "mixed6": 2.0,
    "mixed7": 2.5,
}
outputs_dict = dict(
    [
        (layer.name, layer.output)
        for layer in [model.get_layer(name)
                      for name in layer_settings.keys()]
    ]
)
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

활성화를 최대화할 층과 전체 손실에 대한 가중치. 이 설정을 바꾸면 새로운 시각 효과를 얻을 수 있습니다.

각 층의 심볼릭 출력

각 타깃 층의 활성화 값을 (하나의 딕셔너리로) 반환하는 모델

```
# 设置各层的激活权重
layer_settings = {
  "mixed4": 1.0, # 激活权重
  "mixed5": 1.5, # 激活权重
  "mixed6": 2.0, # 激活权重
  "mixed7": 2.5, # 激活权重
}

# 通过设置的各层权重，生成每一层的激活输出
outputs_dict = dict(
  [
    (layer.name, layer.output)
    for layer in [model.get_layer(name) for name in layer_settings.keys()]
  ]
)

# 创建提取器模型，将每一层的输出（经过设置权重）作为输出
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

12.2 DeepDream（深度梦境）

Keras DeepDream 实现

- 接下来计算损失：
- 使用梯度上升法最大化每个尺度的值。
- 最大化第9章中的过滤器可视化中的特定层的过滤器值。
- 在这里，同时最大化多个层中所有过滤器的激活。
- 特别是，最大化高层激活的 L2 范数的加权平均。
- 选择哪些层：精确选择哪些层（以及它们对最终损失的贡献程度）会对生成的视觉元素产生重大影响。
- 必须能够轻松地改变选择层的参数。
- 低层生成几何图案，高层则生成与 ImageNet 中类相关的视觉元素（例如鸟类、狗等）。

12.2 DeepDream（深度梦境）

Keras DeepDream 实现

- 首先随便选择四个层
- 随后可以尝试其他不同的设置，看看效果如何。

코드 12-12 딥드림 손실

```
def compute_loss(input_image):
    features = feature_extractor(input_image) -------- 활성화를 추출합니다.
    loss = tf.zeros(shape=()) -------- 손실을 0으로 초기화합니다.
    for name in features.keys():
        coeff = layer_settings[name]
        activation = features[name]
        loss += coeff * tf.reduce_mean(tf.square(activation[:, 2:-2, 2:-2, :])) ------
    return loss
```

경계 부근의 인공적인 패턴을 피하기 위해 테두리가 아닌 픽셀만 손실에 추가합니다.

```
def compute_loss(input_image):
  features = feature_extractor(input_image) # 提取特征
  loss = tf.zeros(shape=()) # 初始化损失为0
  for name in features.keys():
    coeff = layer_settings[name] # 获取层的设置
    activation = features[name] # 获取该层的激活
    loss += coeff * tf.reduce_mean(tf.square(activation[:, 2:-2, 2:-2, :])) # 添加损失，避免边界的人工模式
  return loss
```

12.2 DeepDream（深度梦境）

Keras 深度梦境实现

- 现在，我们来准备在每个八度上执行的梯度上升步骤。
- 您将会发现它与第9章的滤波器可视化方法非常相似！
- 深度梦境算法实际上只是滤波器可视化的多尺度版本。

코드 12-13 딥드림 경사 상승법 단계

```
import tensorflow as tf

@tf.function ···· tf.function으로 컴파일하여 훈련 스텝의 속도를 높입니다.
def gradient_ascent_step(image, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)
        loss = compute_loss(image)
    grads = tape.gradient(loss, image)
    grads = tf.math.l2_normalize(grads) ···· 그레이디언트를 정규화합니다(9장에서 사용한 것과 동일한 트릭).
    image += learning_rate * grads
```

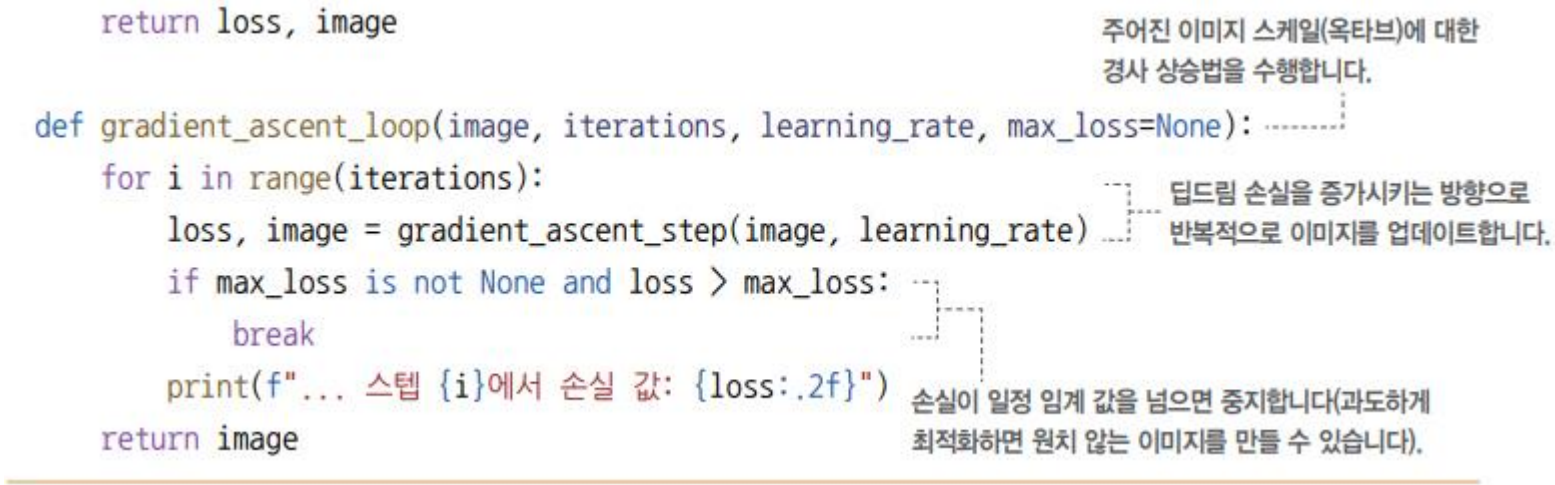
현재 이미지에 대한 딥드림 손실의 그레이디언트를 계산합니다.

```
import tensorflow as tf

@tf.function  # 使用 tf.function 提升训练步骤的速度
def gradient_ascent_step(image, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)  # 监视图像
        loss = compute_loss(image)  # 计算图像的损失
    grads = tape.gradient(loss, image)  # 计算梯度
    grads = tf.math.l2_normalize(grads)  # 归一化梯度
    image += learning_rate * grads  # 更新图像
```

12.2 DeepDream（深度梦境）

Keras 深度梦境实现




```
def gradient_ascent_loop(image, iterations, learning_rate, max_loss=None):
    for i in range(iterations):
        loss, image = gradient_ascent_step(image, learning_rate) # 通过梯度上升更新图像
        if max_loss is not None and loss > max_loss: # 如果损失超过最大值，则停止
            break
        print(f"步骤 {i} 中的损失值: {loss:.2f}") # 打印每步的损失
    return image
```

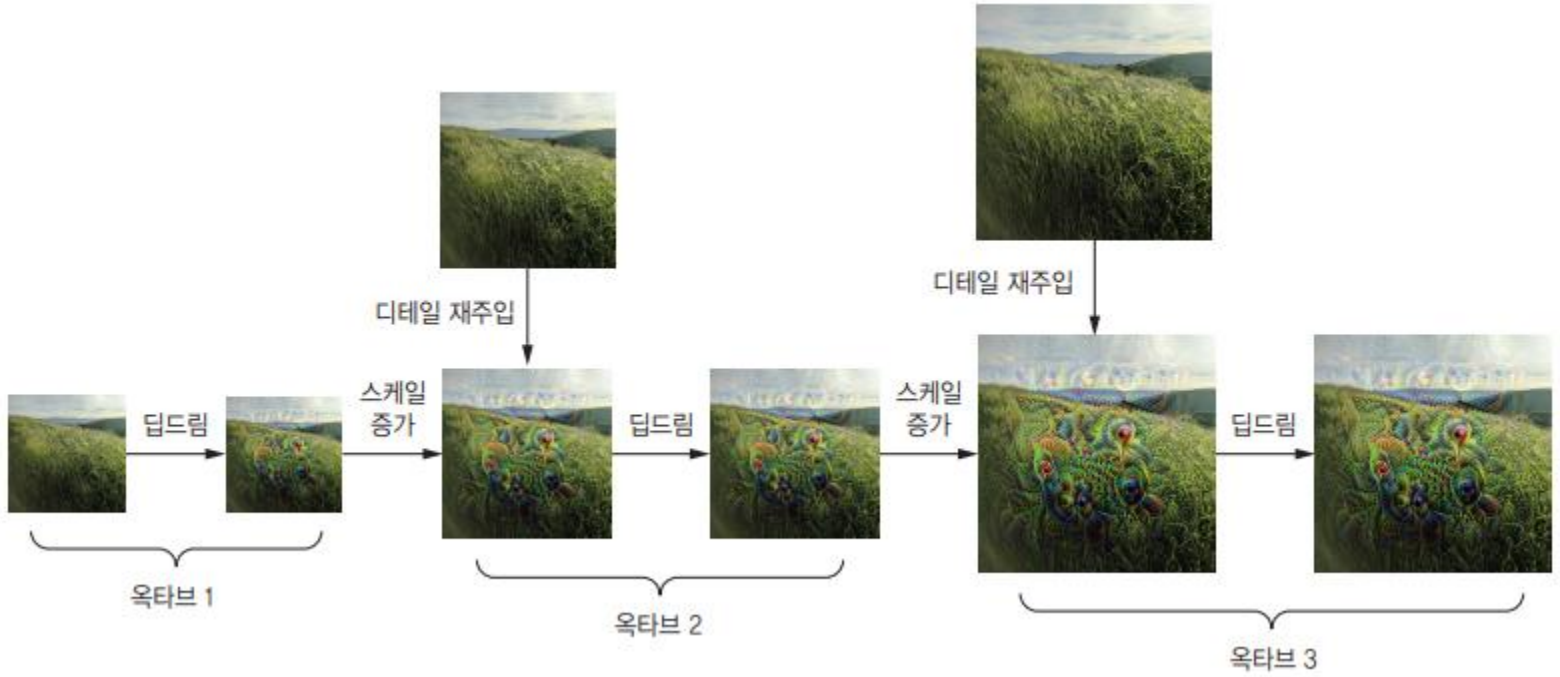
12.2 DeepDream（深度梦境）

Keras 深度梦境实现

- 最后是深度梦境算法的外部循环部分。
- 首先定义一个列表来处理图像的尺度（也称为“八度”）。
- 处理三个不同的“八度”尺度的图像。
- 从最小值到最大值，在每个八度中通过 gradient_ascent_loop() 执行 30 次梯度上升步骤，最大化先前定义的损失值。
- 每个八度之间，图像将增长 40%（1.4倍）。
- 从小图像开始，逐步增加图像大小。（见图 12-6）

12.2 DeepDream（深度梦境）

▼ 图 12-6 通过逐步增加尺度（八度）并将细节重新注入到更大尺度的图像中。这是在执行深度梦境算法时，通过逐步增强图像的细节和增强其特征的视觉效果。



12.2 DeepDream（深度梦境）

Keras 深度梦境实现

- 在接下来的代码中定义这个过程所需的参数
- 通过修改这些参数，您可以获得新的效果！

```
step = 20. -------- 경사 상승법 단계 크기
num_octave = 3 -------- 경사 상승법을 실행할 스케일 횟수
octave_scale = 1.4 -------- 연속적인 스케일 사이의 크기 비율
iterations = 30 -------- 스케일 단계마다 수행할 경사 상승법 단계 횟수
max_loss = 15. -------- 이보다 손실이 커지면 현재 스케일에서 경사 상승법 과정을 중지합니다.
```

step = 20. — 每次梯度上升的步长
num_octave = 3 — 执行梯度上升的尺度数量
octave_scale = 1.4 — 连续尺度之间的尺度大小比例
iterations = 30 — 每个尺度执行的梯度上升步骤数
max_loss = 15. — 如果损失超过该值，当前尺度上的梯度上升过程将停止

12.2 DeepDream（深度梦境）

Keras 深度梦境实现

- 我也会创建一个用来加载和保存图像的实用函数

코드 12-14 이미지 처리 유틸리티

```
import numpy as np

def preprocess_image(image_path): ····· 이미지를 로드하고, 크기를 바꾸어 적절한 배열로 변환하는 유틸리티 함수
    img = keras.utils.load_img(image_path)
    img = keras.utils.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = keras.applications.inception_v3.preprocess_input(img)
    return img

def deprocess_image(img): ····· 넘파이 배열을 이미지로 변환하는 유틸리티 함수
    img = img.reshape((img.shape[1], img.shape[2], 3))
    img += 1.0    ···
                  ├··· InceptionV3 전처리 복원하기[11]
    img *= 127.5  ···
    img = np.clip(img, 0, 255).astype("uint8") ····· uint8로 바꾸고 [0, 255] 범위로 클리핑합니다.
    return img
```

```
import numpy as np

def preprocess_image(image_path):  # 图像预处理函数：加载并调整图像大小
    img = keras.utils.load_img(image_path)
    img = keras.utils.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = keras.applications.inception_v3.preprocess_input(img)
    return img

def deprocess_image(img):  # 图像反处理函数：将numpy数组转换为图像
    img = img.reshape((img.shape[1], img.shape[2], 3))
    img += 1.0  # 逆转InceptionV3预处理
    img *= 127.5
    img = np.clip(img, 0, 255).astype("uint8")  # 转换为uint8并将像素值限制在[0, 255]范围内
    return img
```

12.2 DeepDream（深度梦境）

Keras DeepDream实现

- 这是外部循环
- 通过连续增加比例（由于逐渐模糊或出现像素边界），我们可以使用简单的技巧，避免图像丢失过多细节。
- 增加比例后，将丢失的细节重新注入图像。
- 由于我们知道原始图像放大时的情况，因此这是可能的。
- 给定小图像大小S和大图像大小L，计算更改为L大小的原始图像与更改为S大小的原始图像之间的差异。
- 这是S变更为L时丢失的细节。

코드 12-15 연속적인 여러 개의 '옥타브'에 걸쳐 경사 상승법 실행하기

```
original_img = preprocess_image(base_image_path) -------- 테스트 이미지를 로드합니다.
original_shape = original_img.shape[1:3]

successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape])
    successive_shapes.append(shape)
successive_shapes = successive_shapes[::-1]
```

여러 옥타브에서 이미지 크기를 계산합니다.

Keras DeepDream实现
对多个'Octave'进行连续的梯度上升

```
original_img = preprocess_image(base_image_path) -------- 加载测试图像。
original_shape = original_img.shape[1:3] -------- 获取原始图像的形状

successive_shapes = [original_shape] -------- 初始化成功的图像形状

for i in range(1, num_octave): -------- 按照多个尺度执行梯度上升
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape]) -------- 计算每个尺度的图像大小
    successive_shapes.append(shape) -------- 将该尺度添加到列表
successive_shapes = successive_shapes[::-1] -------- 反转尺寸顺序
```

12.2 DeepDream（深度梦境）

Keras DeepDream实现

```
shrunk_original_img = tf.image.resize(original_img, successive_shapes[0])

img = tf.identity(original_img) ····· 이미지를 복사합니다(원본 이미지는 그대로 보관합니다).
for i, shape in enumerate(successive_shapes): ····· 여러 옥타브에 대해 반복합니다.
    print(f"{shape} 크기의 {i}번째 옥타브 처리")
    img = tf.image.resize(img, shape) ····· 딥드림 이미지의 스케일을 높입니다.
    img = gradient_ascent_loop( ····· 경사 상승법을 실행하고 딥드림 이미지를 수정합니다.
        img, iterations=iterations, learning_rate=step, max_loss=max_loss
    )
    upscaled_shrunk_original_img = tf.image.resize(shrunk_original_img, shape) ·····
                                        작은 버전의 원본 이미지의 스케일을 높입니다. 픽셀 경계가 보일 것입니다.
    same_size_original = tf.image.resize(original_img, shape) ·····
                                        이 크기에 해당하는 고해상도 버전의 원본 이미지를 계산합니다.
    lost_detail = same_size_original - upscaled_shrunk_original_img ·····
    img += lost_detail ····· 손실된 디테일을 딥드림 이미지에 다시 주입합니다.
                                        두 이미지의 차이가 스케일을 높였을 때 손실된 디테일입니다.
    shrunk_original_img = tf.image.resize(original_img, shape)

                                        최종 결과를 저장합니다.
keras.utils.save_img("dream.png", deprocess_image(img.numpy())) ·····
```

```
# 先将图像调整为原始尺寸
shrunken_original_img = tf.image.resize(original_img, successive_shapes[0])

# 图像复制
img = tf.identity(original_img)

# 遍历每个尺度
for i, shape in enumerate(successive_shapes):
  print(f"{shape} 大小的 {i+1} 次 Octave 处理")
  img = tf.image.resize(img, shape)  # 增加图像的尺度
  img = gradient_ascent_loop(
    img, iterations=iterations, learning_rate=step, max_loss=max_loss
  )  # 执行梯度上升并更新图像

# 增大原始图像的尺寸
upscaled_shrunken_original_img = tf.image.resize(shrunken_original_img, shape)

# 与原始尺寸一致的图像
same_size_original = tf.image.resize(original_img, shape)

# 计算丢失的细节
lost_detail = same_size_original - upscaled_shrunken_original_img

# 将丢失的细节添加到当前图像中
img += lost_detail

# 还原原始图像尺寸
shrunken_original_img = tf.image.resize(original_img, shape)

# 保存最终结果
keras.utils.save_img("dream.png", deprocess_image(img.numpy()))
```

12.2 DeepDream（深度梦境）

Keras DeepDream实现

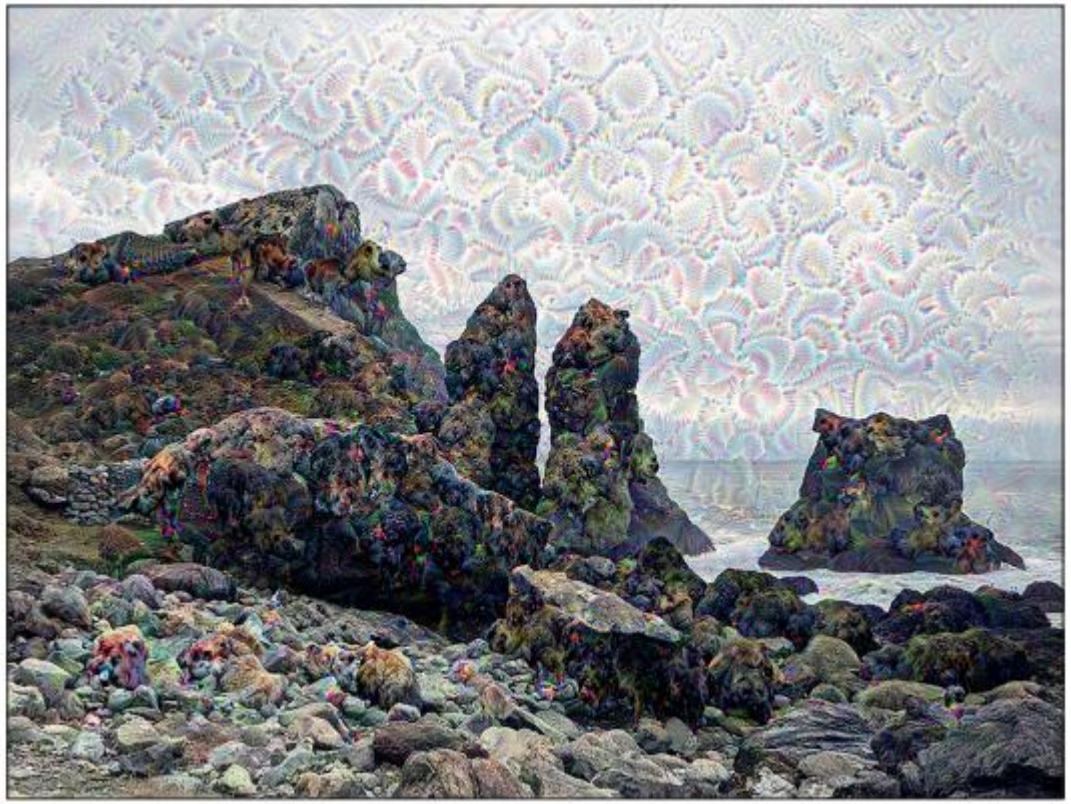
- 原始的 Inception V3 网络是在 299×299 大小的图像上训练的。
- 因此，这个 DeepDream 实现最适合用于将图像大小调整到适当范围的图像，通常在 300×300 到 400×400 之间。
- 这个代码可以在任何大小或比例的图像上运行。

12.2 DeepDream（深度梦境）

Keras DeepDream实现

- 如果使用GPU，运行整个过程只需几秒钟。

▼ 图 12-7 示例图片中 DeepDream 代码执行结果



12.2 DeepDream（深度梦境）

Keras DeepDream 实现

- 尝试更改用于损失的层，并观察会发生什么。
- 网络的较低层具有局部的、相对不那么抽象的表示，这就是为什么在 DeepDream 图像中会出现许多几何图案。
- 较高层则基于 ImageNet 中常见的物体（如狗的眼睛、鸟的羽毛）创建出明显的视觉模式。
- 通过随机生成 layer_settings 字典中的参数，可以快速探索不同的层组合。
- 图 12-8 展示了通过改变层组合得到的自制糕点图像的结果。

12.2 DeepDream（深度梦境）

▼ 图 12-8：在示例图像上尝试的各种 DeepDream 设置。

12.2 DeepDream（深度梦境）

总结

- DeepDream 是基于网络学到的表示，通过反向执行卷积神经网络来生成输入图像。
- 它会产生有趣的结果，有时甚至像是由于幻觉而视线模糊的人所创作的图像。
- 这个过程不仅限于图像模型或卷积神经网络，也可以应用于语音、音乐等领域。

12.3 神经风格迁移

12.3神经风格迁移

神经风格迁移

- 除了 DeepDream，使用深度学习改变图像的另一个主要领域是神经风格迁移（neural style transfer）。
- 该技术由 Leon Gatys 等人在 2015 年夏天首次提出。
- 自神经风格迁移算法首次介绍以来，已经得到了许多改进，并且衍生出了多个变种。
- 它也被应用于智能手机的照片应用程序中。
- 神经风格迁移在保留目标图像内容的同时，将参考图像的风格应用于目标图像。

▼ 图 12-9 风格迁移示例"

12.3神经风格迁移

神经风格迁移

- 在这里，“风格”指的是质感、颜色以及图像中不同大小的视觉元素；
- “内容”则指的是图像中高层次的主要结构。
- 例如，在图 12-9（文森特·梵高（Vincent Van Gogh）的《星空》）中，可以把画出蓝色和黄色圆形的笔触视为一种风格；
- 图像中图宾根（Tübingen）照片的建筑则可以看作内容。
- 与纹理生成密切相关的风格迁移的思想，在2015年神经风格迁移开发之前，已经在图像处理领域有着悠久的历史。
- 基于深度学习的风格迁移实现提供了与经典计算机视觉技术所能达到的结果无法比拟的效果，
- 并为创造性的计算机视觉应用领域开辟了新的“文艺复兴”。

12.3神经风格迁移

神经风格迁移

- 风格迁移实现背后的核心概念与所有深度学习算法的核心相同：
- 定义表示目标的损失函数，并最小化这个损失。
- 在这里，目标是：
- 在应用参考图像的风格的同时，保留原始图像的内容。
- 如果我们能数学地定义内容和风格，那么最小化的损失函数将如下所示：

```
loss = distance(style(reference_image) - style(combination_image)) +
       distance(content(original_image) - content(combination_image))
```

12.3神经风格迁移

神经风格迁移

- 这里的"distance"是指像 L2 范数这样的范数函数。
- "content"函数计算图像的内容表示，"style"函数计算图像的风格表示。
- 最小化这个损失后，style(combination_image) 将会接近 style(reference_image)，而 content(combination_image) 将会接近 content(original_image)。
- 通过这种方式，可以实现前面定义的风格迁移目标。
- Gatys 等人展示了使用深度卷积神经网络，可以数学地定义 style 和 content 函数。

12.3神经风格迁移

内容损失

- 网络中较低层的激活包含图像的局部信息，而较高层的激活则包含越来越全局化和抽象化的信息。
- 换个角度看，卷积神经网络（Convolutional Neural Network，简称 CNN）层的激活可以看作是将图像分解为不同尺度的内容。
- 使用卷积神经网络的较高层表示，我们可以找到全局性和抽象性的图像内容。
- 通过将目标图像和生成图像输入到预先训练好的卷积神经网络中，计算其高层激活。
- 这两个值之间的 L2 范数非常适合作为内容损失。
- 从较高层来看，我们会让生成的图像与原始目标图像尽可能相似。
- 如果假设在卷积神经网络的较高层看到的内容是输入图像的内容，那么它可以作为一种方法来保留图像的内容。

12.3神经风格迁移

内容损失

- 内容损失只使用一个较高层，而 Gatys 等人定义的风格损失则使用卷积神经网络的多个层。
- 不仅仅是一个风格，而是需要捕捉参考图像中卷积神经网络提取的所有不同尺度的风格。
- Gatys 等人使用层激活输出的 Gram 矩阵（Gram matrix）作为风格损失。
- Gram 矩阵是层特征图之间的内积（inner product），内积可以理解为表示层特征之间的相关性。
- 这些特征的相关性捕捉到的是特定尺度的空间模式统计。
- 根据经验来看，这些特征对应于该层中找到的纹理。
- 通过计算风格参考图像和生成图像的层激活，
- 风格损失的目标是保留其中内在的相关性。
- 它使得风格参考图像和生成图像中的多个尺度的纹理看起来相似。

12.3神经风格迁移

内容损失

- 总结来说，通过使用预训练的卷积神经网络（Convolutional Neural Network，简称 CNN），可以定义以下损失：
  - 为了保留内容，在原始图像和生成图像之间保持较高层激活的相似性。
    这个卷积神经网络在原始图像和生成图像中应该看到相同的内容。

  - 为了保留内容，在原始图像和生成图像之间保持较高层激活的相似性。
    - 特征之间的相关性表示纹理。
    - 生成的图像和风格参考图像将共享多个尺度的纹理。

12.3神经风格迁移

使用 Keras 实现神经风格迁移

- 神经风格迁移可以使用任何预训练的卷积神经网络（CNN）来实现，
- 这里我们使用 Gatys 等人使用的 VGG19 网络。
- VGG19 是第 5 章中介绍的 VGG16 网络的变种，具有额外的 3 个卷积层。

- 一般的过程如下：
  - 1. 设置一个网络，同时计算 VGG19 中的层激活，用于风格参考图像、基础图像（base image）和生成图像。
  - 2. 使用这三张图像中计算得到的层激活来定义前面提到的损失函数。

  通过最小化这个损失函数来实现风格迁移。
  - 3. 设置一个梯度下降过程来最小化损失函数。

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 从定义风格参考图像和基础图像的路径开始。
- 处理的图像大小应相似（如果图像大小差异较大，风格迁移的实现会更困难）。
- 将所有图像的高度调整为 400 像素。

코드 12-16 스타일 이미지와 콘텐츠 이미지 준비하기

```
from tensorflow import keras

base_image_path = keras.utils.get_file( -------- 변환할 이미지 경로
    "sf.jpg", origin="https://img-datasets.s3.amazonaws.com/sf.jpg")
style_reference_image_path = keras.utils.get_file( -------- 스타일 이미지 경로
    "starry_night.jpg",
    origin="https://img-datasets.s3.amazonaws.com/starry_night.jpg")
original_width, original_height = keras.utils.load_img(base_image_path).size
img_height = 400
img_width = round(original_width * img_height / original_height)  ---- 생성 이미지의 차원
```

12.3 神经风格迁移

▼ 图 12-10 内容图像：从 Nob Hill 看到的旧金山

12.3 神经风格迁移

▼ 图 12-11 风格图像：梵高的《星空》

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 还需要为 VGG19 卷积神经网络加载、预处理和后处理输入输出图像的工具函数。

코드 12-17 유틸리티 함수

```
import numpy as np

def preprocess_image(image_path): -------- 이미지를 로드하고, 크기를 바꾸어 적절한 배열로 변환하는 유틸리티 함수
    img = keras.utils.load_img(
        image_path, target_size=(img_height, img_width))
    img = keras.utils.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = keras.applications.vgg19.preprocess_input(img)
    return img
```

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

```
def deprocess_image(img): -------- 넘파이 배열을 이미지로 변환하는 유틸리티 함수
    img = img.reshape((img_height, img_width, 3))
    img[:, :, 0] += 103.939 ---┐
    img[:, :, 1] += 116.779    ├---- ImageNet의 평균 픽셀 값을 더합니다. 이는 vgg19.preprocess_
    img[:, :, 2] += 123.68 ----┘     input 함수에서 수행한 변환을 복원합니다.[17]
    img = img[:, :, ::-1] -------- 이미지를 'BGR'에서 'RGB'로 변환합니다. 이것도 vgg19.preprocess_
                                   input 함수에서 수행한 변환을 복원하기 위해서입니다.
    img = np.clip(img, 0, 255).astype("uint8")
    return img
```

1.图像被重新调整为所需的高度、宽度和3个颜色通道（RGB）。

2.从ImageNet中减去平均像素值。这是为了逆转`vgg19.preprocess_input`应用的预处理步骤。

3.将图像从BGR（VGG19使用的格式）转换为RGB。

4.将像素值限制在0到255之间，并转换为无符号8位整数格式（`uint8`），这是显示图像的标准格式。
这个过程确保在经过神经网络处理后，图像能转换回可显示的格式。

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 准备 VGG19 网络
- 与 DeepDream 示例相同，我们将使用预训练的卷积神经网络（CNN），创建一个返回中间层激活的特征提取模型。
- 这次，我们将使用模型中的所有层。

코드 12-18 사전 훈련된 VGG19 모델을 사용해서 특성 추출기 만들기

```
model = keras.applications.vgg19.VGG19(weights="imagenet", include_top=False)
```

ImageNet에서 사전 훈련된 가중치로 VGG19 모델을 만듭니다.

在 ImageNet 上预训练的 VGG19 模型

```
outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

이 모델은 모든 타깃 층의 활성화 값을 (하나의 딕셔너리로) 반환합니다.

为模型中的每一层创建一个字典，包含层的名称和输出。

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 定义内容损失
- VGG19 卷积神经网络的高层应该在基础图像和生成图像中看到相同的内容。

코드 12-19 콘텐츠 손실

```
def content_loss(base_img, combination_img):
    return tf.reduce_sum(tf.square(combination_img - base_img))
```

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 接下来是风格损失
- 使用实用函数计算输入矩阵的 Gram 矩阵。
- 这个矩阵记录了原始特征矩阵的相关性。

코드 12-20 스타일 손실

```
def gram_matrix(x):
    x = tf.transpose(x, (2, 0, 1))
    features = tf.reshape(x, (tf.shape(x)[0], -1))
    gram = tf.matmul(features, tf.transpose(features))
    return gram

def style_loss(style_img, combination_img):
    S = gram_matrix(style_img)
    C = gram_matrix(combination_img)
    channels = 3
    size = img_height * img_width
    return tf.reduce_sum(tf.square(S - C)) / (4.0 * (channels ** 2) * (size ** 2))
```

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 在两种损失函数中再添加一个损失：
- 使用生成图像的像素计算的总变异损失（total variation loss）。
- 这有助于生成的图像保持空间连续性，并防止像素的网格模式过度显现。
- 可以将其视为一种正则化项。

코드 12-21 총 변위 손실

```
def total_variation_loss(x):
    a = tf.square(
        x[:, : img_height - 1, : img_width - 1, :] - x[:, 1:, : img_width - 1, :]
    )
    b = tf.square(
        x[:, : img_height - 1, : img_width - 1, :] - x[:, : img_height - 1, 1:, :]
    )
    return tf.reduce_sum(tf.pow(a + b, 1.25))
```

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 最小化的损失是这三种损失的加权平均值
- 内容损失是仅使用 block5_conv2 层来计算的。
- 为了计算风格损失，需要跨越多个层，使用从低层到高层的多个层。
- 最后，添加总变异损失。

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 根据使用的风格参考图像和内容图像，调整 content_weight 系数（即贡献给整体损失的内容损失的程度）是很重要的。
- 如果 content_weight 较高，则生成的图像中目标内容将更为显著。

코드 12-22 최소화할 최종 손실 정의하기

```
style_layer_names = [ -------- 스타일 손실에 사용할 층
    "block1_conv1",
    "block2_conv1",
    "block3_conv1",
    "block4_conv1",
    "block5_conv1",
]
content_layer_name = "block5_conv2" -------- 콘텐츠 손실에 사용할 층
total_variation_weight = 1e-6 -------- 총 변이 손실의 기여 가중치
style_weight = 1e-6 -------- 스타일 손실의 기여 가중치
content_weight = 2.5e-8 -------- 콘텐츠 손실의 기여 가중치
```

```
style_layer_names = [  # -------- 用于风格损失的层
   "block1_conv1",
   "block2_conv1",
   "block3_conv1",
   "block4_conv1",
   "block5_conv1",
]
content_layer_name = "block5_conv2"  # -------- 用于内容损失的层
total_variation_weight = 1e-6  # -------- 总变异损失的权重系数
style_weight = 1e-6  # -------- 风格损失的权重系数
content_weight = 2.5e-8  # -------- 内容损失的权重系数
```

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

```
def compute_loss(combination_image, base_image, style_reference_image):
    input_tensor = tf.concat(
        [base_image, style_reference_image, combination_image], axis=0)
    features = feature_extractor(input_tensor)
    loss = tf.zeros(shape=()) -------- 손실을 0으로 초기화합니다.
    layer_features = features[content_layer_name]
    base_image_features = layer_features[0, :, :, :]
    combination_features = layer_features[2, :, :, :]
    loss = loss + content_weight * content_loss(
        base_image_features, combination_features
    )
```

콘텐츠 손실을 더합니다.

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

```
for layer_name in style_layer_names:
    layer_features = features[layer_name]
    style_reference_features = layer_features[1, :, :, :]
    combination_features = layer_features[2, :, :, :]
    style_loss_value = style_loss(
        style_reference_features, combination_features)
    loss += (style_weight / len(style_layer_names)) * style_loss_value

loss += total_variation_weight * total_variation_loss(combination_image)
return loss
```

스타일 손실을 더합니다.

총 변위 손실을 더합니다.

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 最后，设置梯度下降步骤。
- 在 Gatys 的原始论文中，使用 L-BFGS 算法来执行优化。
- 由于 TensorFlow 中没有这个算法，因此使用 SGD 优化器执行小批量梯度下降。
- 在这里，我们首次使用优化器功能中的学习率调度（learning rate schedule）。
- 通过这个功能，学习率从非常高的值（100）逐渐减小到非常低的值（约 20）。
- 这样，在训练初期，模型以较快的速度进行，而随着接近最小损失，训练过程会变得越来越谨慎。

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

코드 12-23 경사 하강법 단계 설정하기

```
import tensorflow as tf

@tf.function -------- tf.function으로 컴파일하여 훈련 스텝의 속도를 높입니다.
def compute_loss_and_grads(
    combination_image, base_image, style_reference_image):
    with tf.GradientTape() as tape:
        loss = compute_loss(
            combination_image, base_image, style_reference_image)
    grads = tape.gradient(loss, combination_image)
    return loss, grads

                                          학습률 100에서 시작하여 100
optimizer = keras.optimizers.SGD(         스텝마다 4%씩 감소시킵니다.
    keras.optimizers.schedules.ExponentialDecay( ------
        initial_learning_rate=100.0, decay_steps=100, decay_rate=0.96
    )
)
```

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

```
base_image = preprocess_image(base_image_path)
style_reference_image = preprocess_image(style_reference_image_path)
combination_image = tf.Variable(preprocess_image(base_image_path))
iterations = 4000

for i in range(1, iterations + 1):
    loss, grads = compute_loss_and_grads(
        combination_image, base_image, style_reference_image
    )
    optimizer.apply_gradients([(grads, combination_image)])
    if i % 100 == 0:
        print(f"{i}번째 반복: loss={loss:.2f}")
        img = deprocess_image(combination_image.numpy())
        fname = f"combination_image_at_iteration_{i}.png"
        keras.utils.save_img(fname, img)
```

훈련하는 동안 합성된 이미지를 업데이트하기 때문에 Variable에 저장합니다.

由于训练过程中会更新合成图像，因此将其存储为 Variable。

스타일 트랜스퍼 손실이 감소되는 방향으로 합성 이미지를 업데이트 합니다.

更新合成图像，最小化风格迁移损失。

일정한 간격으로 합성 이미지를 저장합니다.

将当前合成图像保存为 PNG 文件。

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 图 12-12 展示了生成的图像
- 请记住，这种方法是改变图像的纹理或将纹理进行转移。
- 当风格图像的纹理更加突出且有许多相似的模式时，它能够很好地工作。
- 当不需要深入理解内容目标时，它也能很好地工作。
- 通常，像将人物照片的风格转移到另一张人物照片上这种非常抽象的技巧是无法完成的。
- 这个算法更接近于经典的信号处理，而不是 AI，因此不要期待像魔法一样的结果！

12.3 神经风格迁移

▼ 图 12-12 展示了风格转移的结果。

12.3 神经风格迁移

使用 Keras 实现神经风格迁移

- 风格迁移算法虽然比较慢，但由于它执行的是简单的转换，可以使用小而快速的卷积神经网络（CNN）进行学习。
- 当然，必须有适量的训练数据。
- 首先，对于固定的风格参考图像，通过此方法生成大量的输入输出训练样本。
- 然后，训练一个简单的卷积神经网络来学习这种风格转换，就可以快速执行风格迁移。
- 使用这种模型，给定任何图像时，可以瞬间改变其风格。
- 只需要将该图像通过这个小的卷积神经网络即可。

12.3 神经风格迁移

整理

- 风格迁移 是一种在应用参考图像的风格的同时，保留目标图像的内容，从而生成新图像的方法。
- 内容可以从卷积神经网络（CNN）高层的激活中获得。
- 风格则可以从多个卷积神经网络层的激活中固有的相关性中获得。
- 在深度学习中，可以使用预训练的卷积神经网络定义损失函数，并通过优化该损失来实现风格迁移。
- 从这个基本的思路出发，可以进行多种变种和改进。