Deep Learning

Lecture Topic:
Generative Deep Learning

Anxiao (Andrew) Jiang

Learning Objectives:

1. Understand text generation

2. Understand auto-encoder and VAE (variational auto-encoder)

3. Understand GAN (generative adversarial network)

4. Understand neural style transfer

Roadmap of this lecture:

1. Text generation
    - 1.1 General concepts in text generation
    - 1.2 Text generation using IMDB dataset
2. Auto-encoder and VAE (variational auto-encoder)
    - 2.1 Auto-encoder
    - 2.2 VAE
3. GAN (generative adversarial network)
4. Neural style transfer

Text Generation

The universal way to generate sequence data in deep learning is to train a model (usually a Transformer or an RNN) to predict the next token or next few tokens in a sequence, using the previous tokens as input. For instance, given the input "the cat is on the," the model is trained to predict the target "mat," the next word. As usual when working with text data, tokens are typically words or characters, and any network that can model the probability of the next token given the previous ones is called a *language model*. A language model captures the *latent space* of language: its statistical structure.

Text Generation

Once you have such a trained language model, you can *sample* from it (generate new sequences): you feed it an initial string of text (called *conditioning data*), ask it to generate the next character or the next word (you can even generate several tokens at once), add the generated output back to the input data, and repeat the process many times (see figure 12.1). This loop allows you to generate sequences of arbitrary length that reflect the structure of the data on which the model was trained: sequences that look *almost* like human-written sentences.

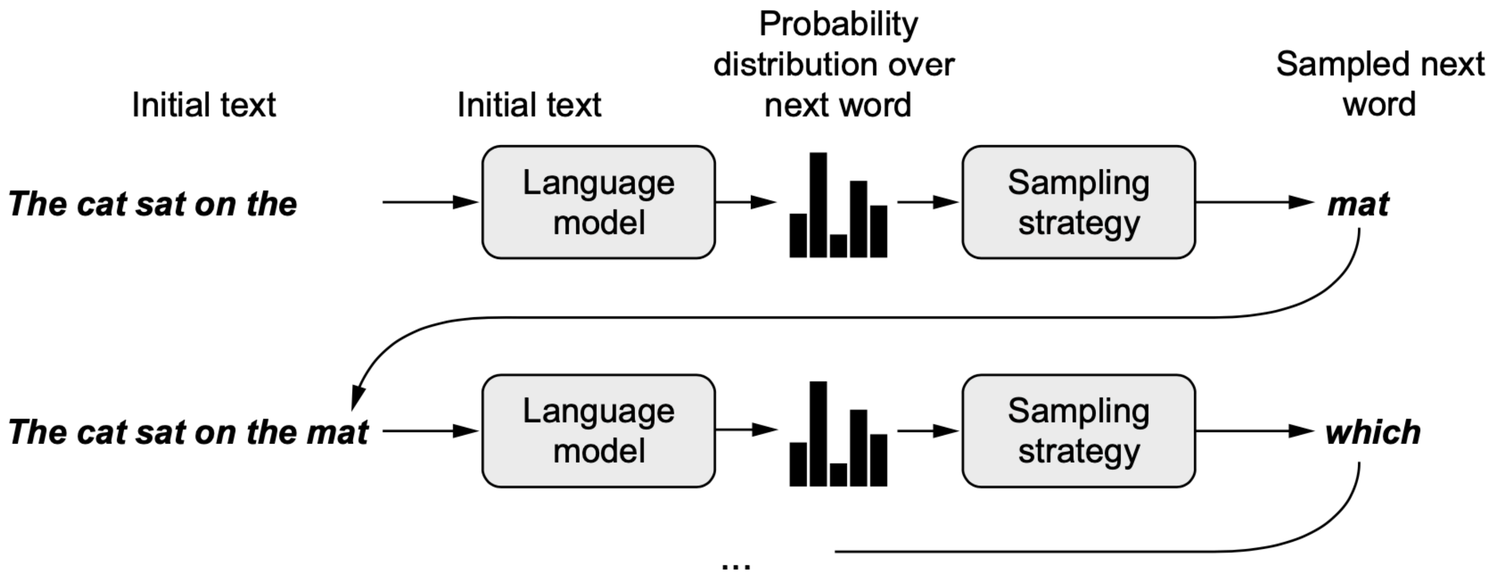


Figure 12.1 The process of word-by-word text generation using a language model

Sampling Strategy for Text Generation

When generating text, the way you choose the next token is crucially important. A naive approach is *greedy sampling*, consisting of always choosing the most likely next character. But such an approach results in repetitive, predictable strings that don't look like coherent language. A more interesting approach makes slightly more surprising choices: it introduces randomness in the sampling process by sampling from the probability distribution for the next character. This is called *stochastic sampling* (recall that *stochasticity* is what we call *randomness* in this field). In such a setup, if a word has probability 0.3 of being next in the sentence according to the model, you'll choose it 30% of the time. Note that greedy sampling can also be cast as sampling from a probability distribution: one where a certain word has probability 1 and all others have probability 0.

Sampling Strategy for Text Generation

In order to control the amount of stochasticity in the sampling process, we'll introduce a parameter called the *softmax temperature*, which characterizes the entropy of the probability distribution used for sampling: it characterizes how surprising or predictable the choice of the next word will be. Given a `temperature` value, a new probability distribution is computed from the original one (the softmax output of the model) by reweighting it in the following way.

Listing 12.1 Reweighting a probability distribution to a different temperature

original_distribution is a 1D NumPy array of probability values that must sum to 1. temperature is a factor quantifying the entropy of the output distribution.

```
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)
```

Returns a reweighted version of the original distribution. The sum of the distribution may no longer be 1, so you divide it by its sum to obtain the new distribution.

Sampling Strategy for Text Generation

original distribuiton: $p_1, p_2, \cdots, p_n$ temperature: T

new distribuiton: $\frac{p_1^{1/T}}{\sum_{i=1}^{n} p_i^{1/T}}, \frac{p_2^{1/T}}{\sum_{i=1}^{n} p_i^{1/T}}, \cdots, \frac{p_n^{1/T}}{\sum_{i=1}^{n} p_i^{1/T}}$

Quiz questions:

1. What is the relation between language modeling and text generation?

2. Why do we use “temperature” to control the distribution of the output text?

Roadmap of this lecture:

1. Text generation
    - 1.1 General concepts in text generation
    - 1.2 Text generation using IMDB dataset
2. Neural style transfer
3. Auto-encoder and VAE (variational auto-encoder)
    - 3.1 Auto-encoder
    - 3.2 VAE
4. GAN (generative adversarial network)

Implement Text Generation with Keras

Dataset: IMDB movie review dataset

Listing 12.2 Downloading and uncompressing the IMDB movie reviews dataset

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

Process data

You're already familiar with the structure of the data: we get a folder named aclImdb containing two subfolders, one for negative-sentiment movie reviews, and one for positive-sentiment reviews. There's one text file per review. We'll call `text_dataset_from_directory` with `label_mode=None` to create a dataset that reads from these files and yields the text content of each file.

Listing 12.3 Creating a dataset from text files (one file = one sample)

```
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 />", " "))
```

Strip the
 HTML tag that occurs in many of the reviews. This did not matter much for text classification, but we wouldn't want to generate
 tags in this example!

Process data

Now let's use a `TextVectorization` layer to compute the vocabulary we'll be working with. We'll only use the first `sequence_length` words of each review: our `TextVectorization` layer will cut off anything beyond that when vectorizing a text.

Listing 12.4 Preparing a `TextVectorization` layer

```
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)
```

We'll only consider the top 15,000 most common words—anything else will be treated as the out-of-vocabulary token, "[UNK]".

We want to return integer word index sequences.

We'll work with inputs and targets of length 100 (but since we'll offset the targets by 1, the model will actually see sequences of length 99).

Process data

Let's use the layer to create a language modeling dataset where input samples are vectorized texts, and corresponding targets are the same texts offset by one word.

Listing 12.5 Setting up a language modeling dataset

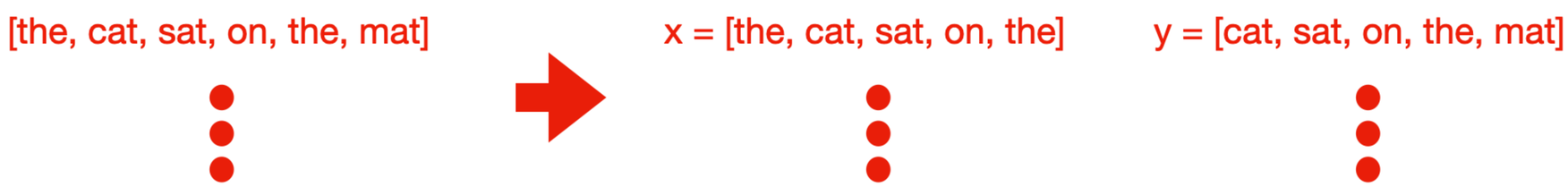
```
def prepare_lm_dataset(text_batch):
    vectorized_sequences = text_vectorization(text_batch)
    x = vectorized_sequences[:, :-1]
    y = vectorized_sequences[:, 1:]
    return x, y

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

Convert a batch of texts (strings) to a batch of integer sequences.

Create inputs by cutting off the last word of the sequences.

Create targets by offsetting the sequences by 1.



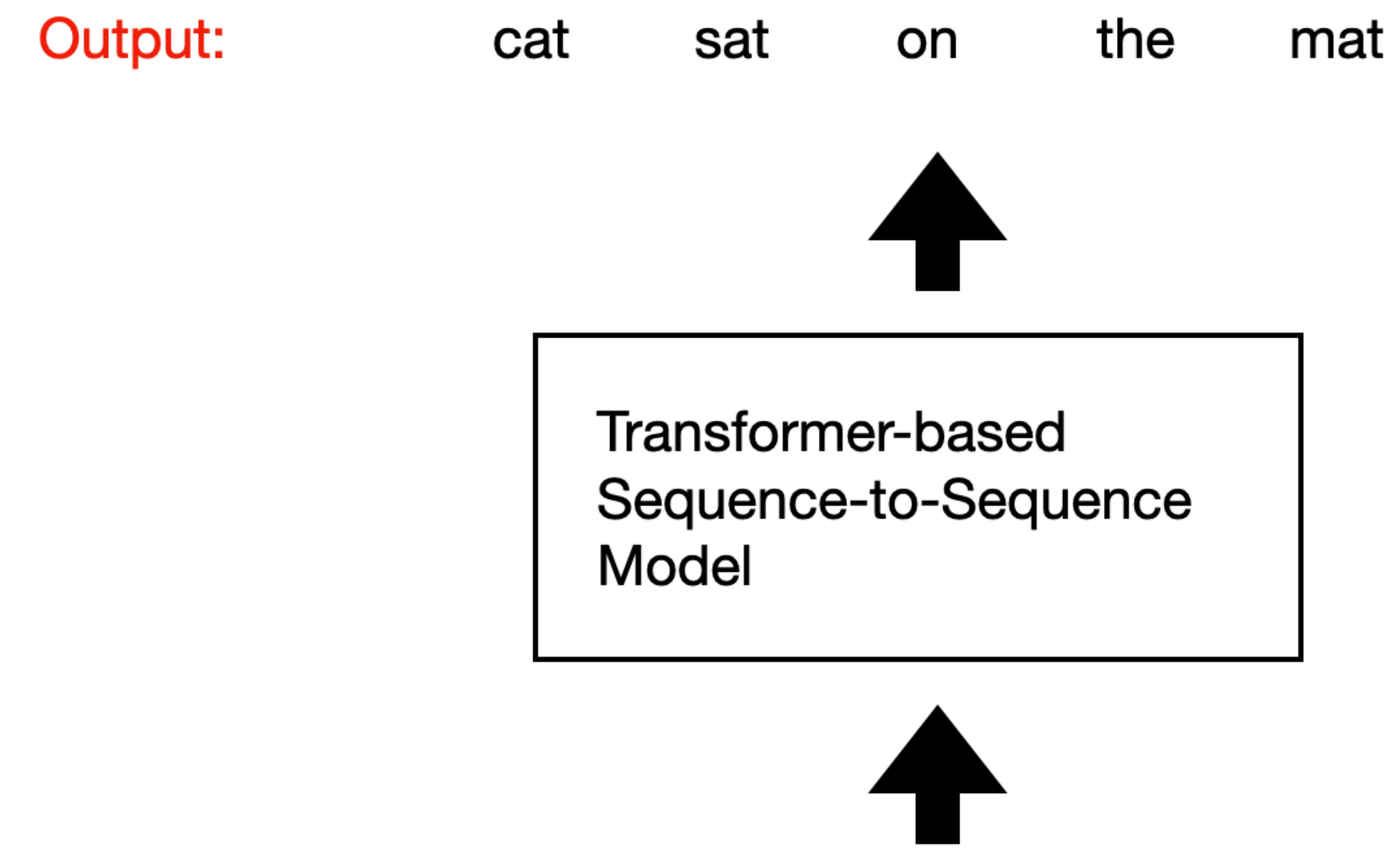
A transformer-based sequence-to-sequence model for text generation

We'll train a model to predict a probability distribution over the next word in a sentence, given a number of initial words. When the model is trained, we'll feed it with a prompt, sample the next word, add that word back to the prompt, and repeat, until we've generated a short paragraph.

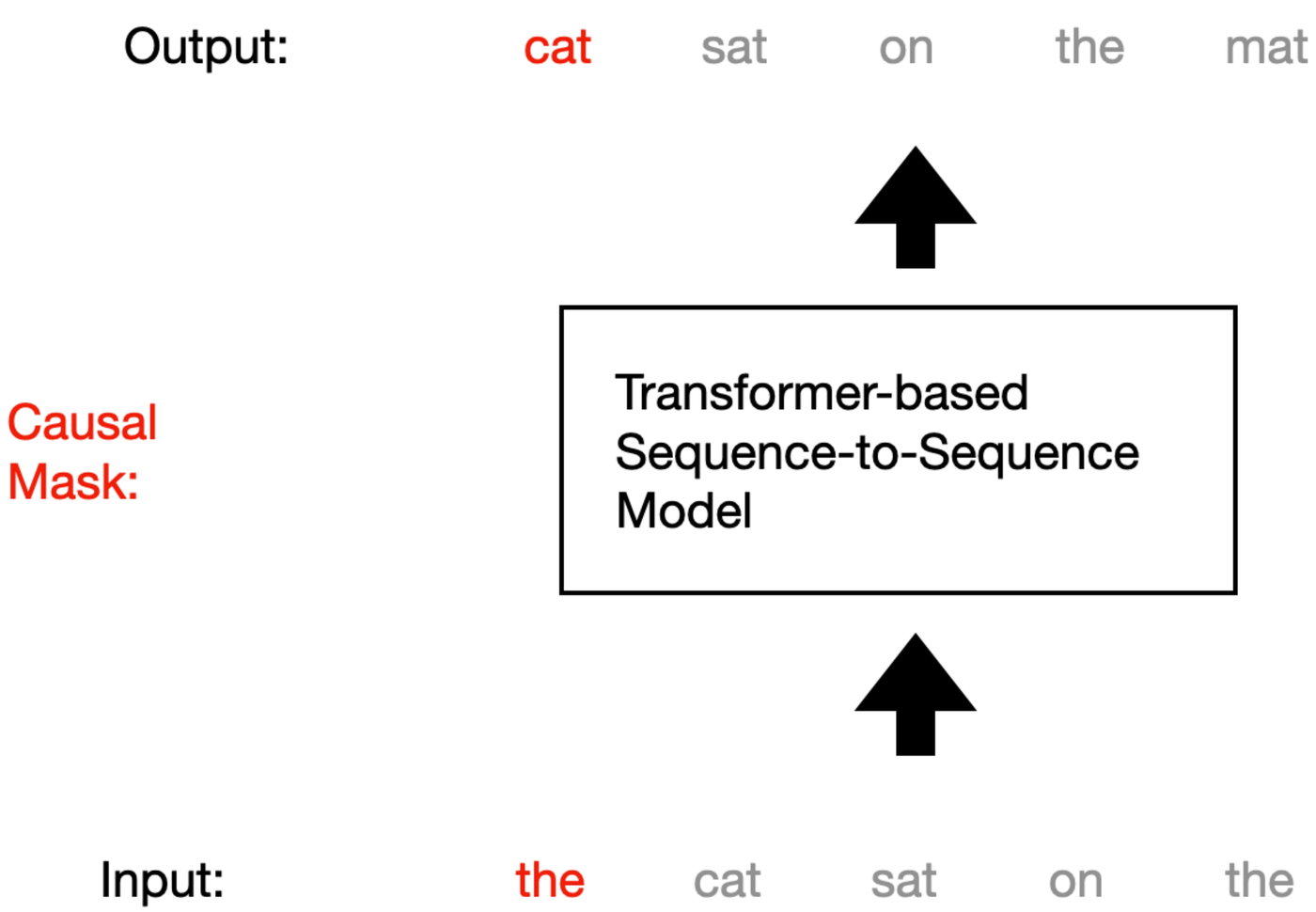
A transformer-based sequence-to-sequence model for text generation

we'll use a *sequence-to-sequence model*: we'll feed sequences of N words (indexed from 0 to N) into our model, and we'll predict the sequence offset by one (from 1 to `N+1`). We'll use causal masking to make sure that, for any `i`, the model will only be using words from 0 to `i` in order to predict the word `i + 1`. This means that we're simultaneously training the model to solve N mostly overlapping but different problems: predicting the next words given a sequence of `1 <= i <= N` prior words (see figure 12.3). At generation time, even if you only prompt the model with a single word, it will be able to give you a probability distribution for the next possible words.

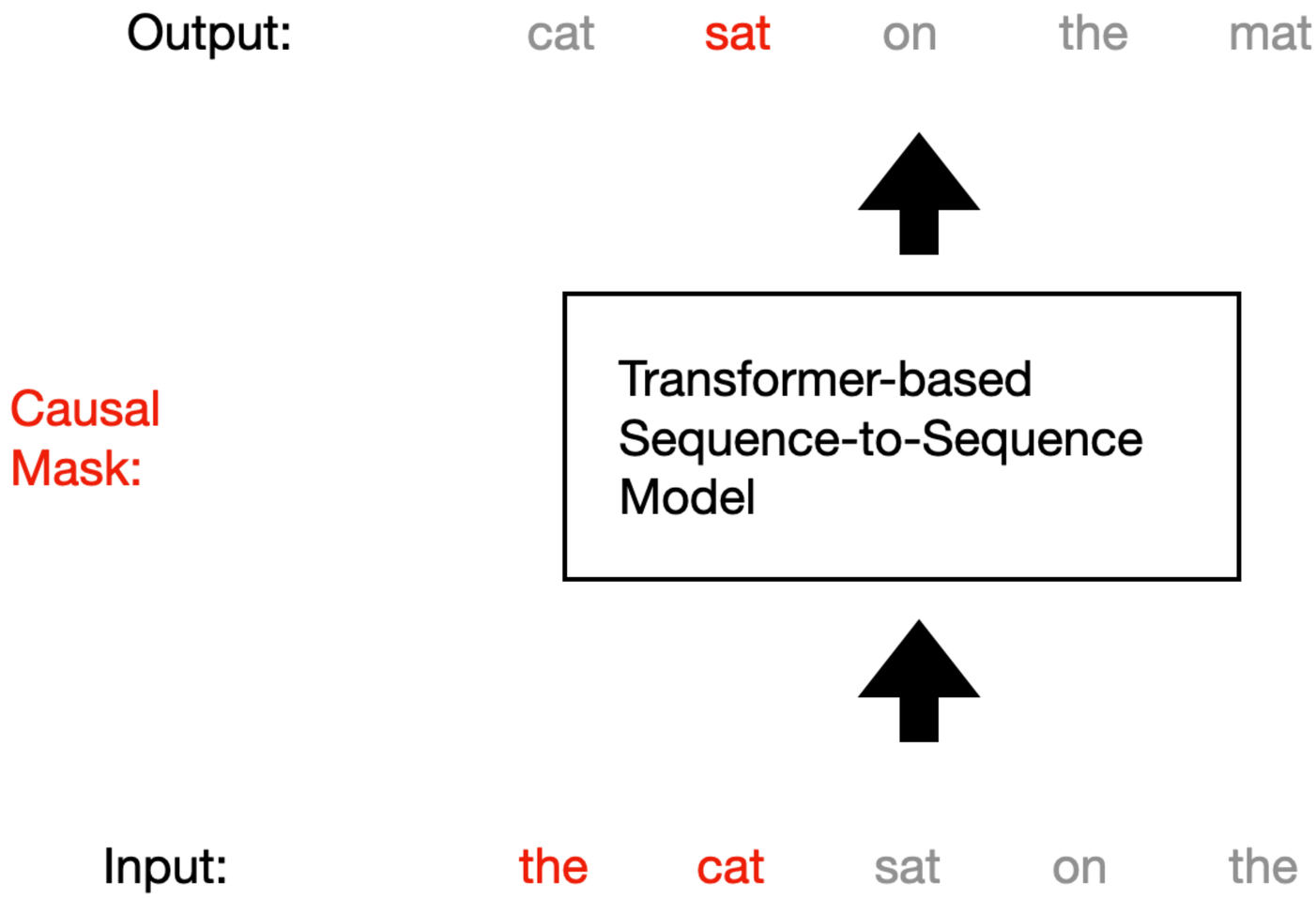
A transformer-based seq2seq model for text generation



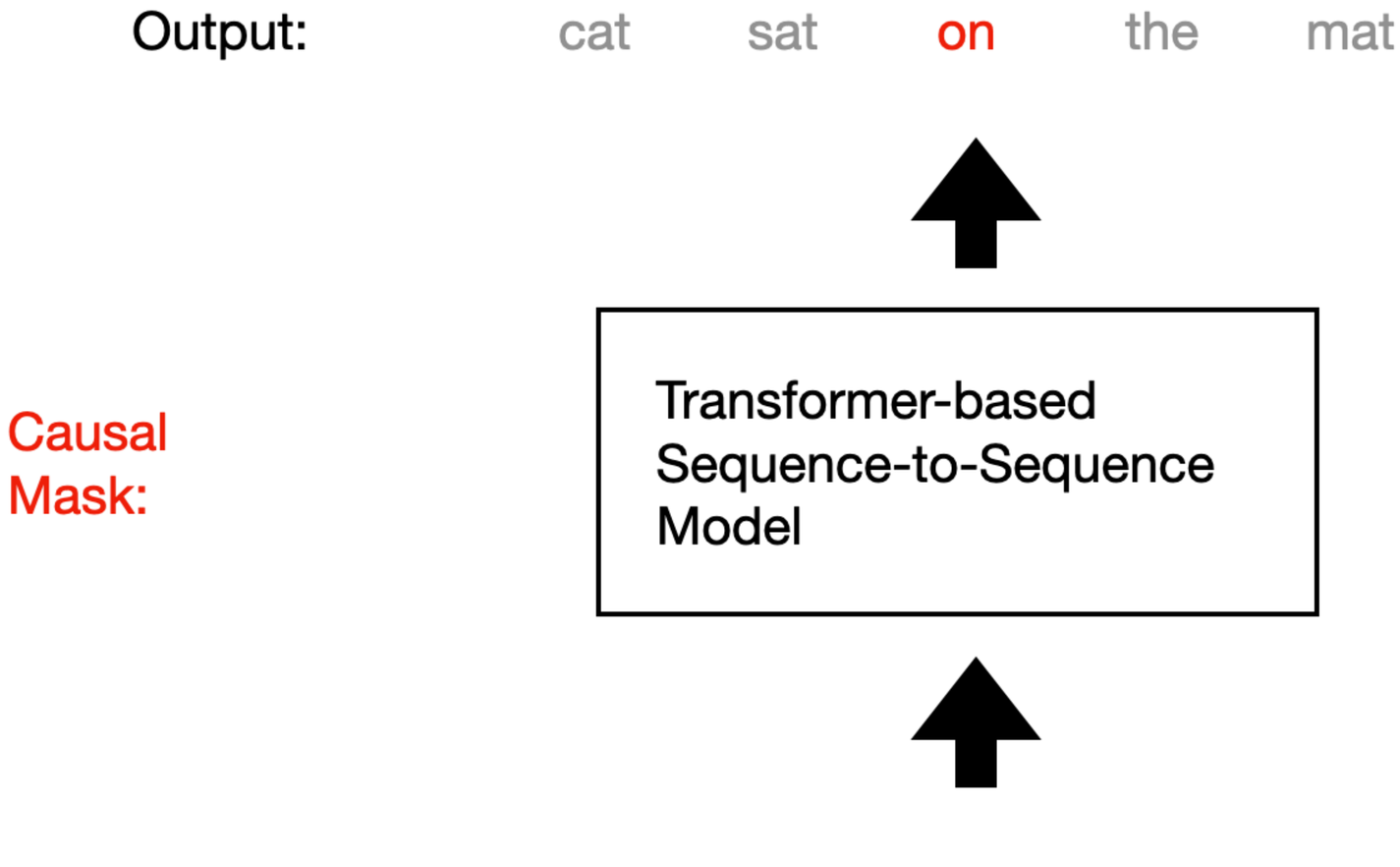
A transformer-based seq2seq model for text generation



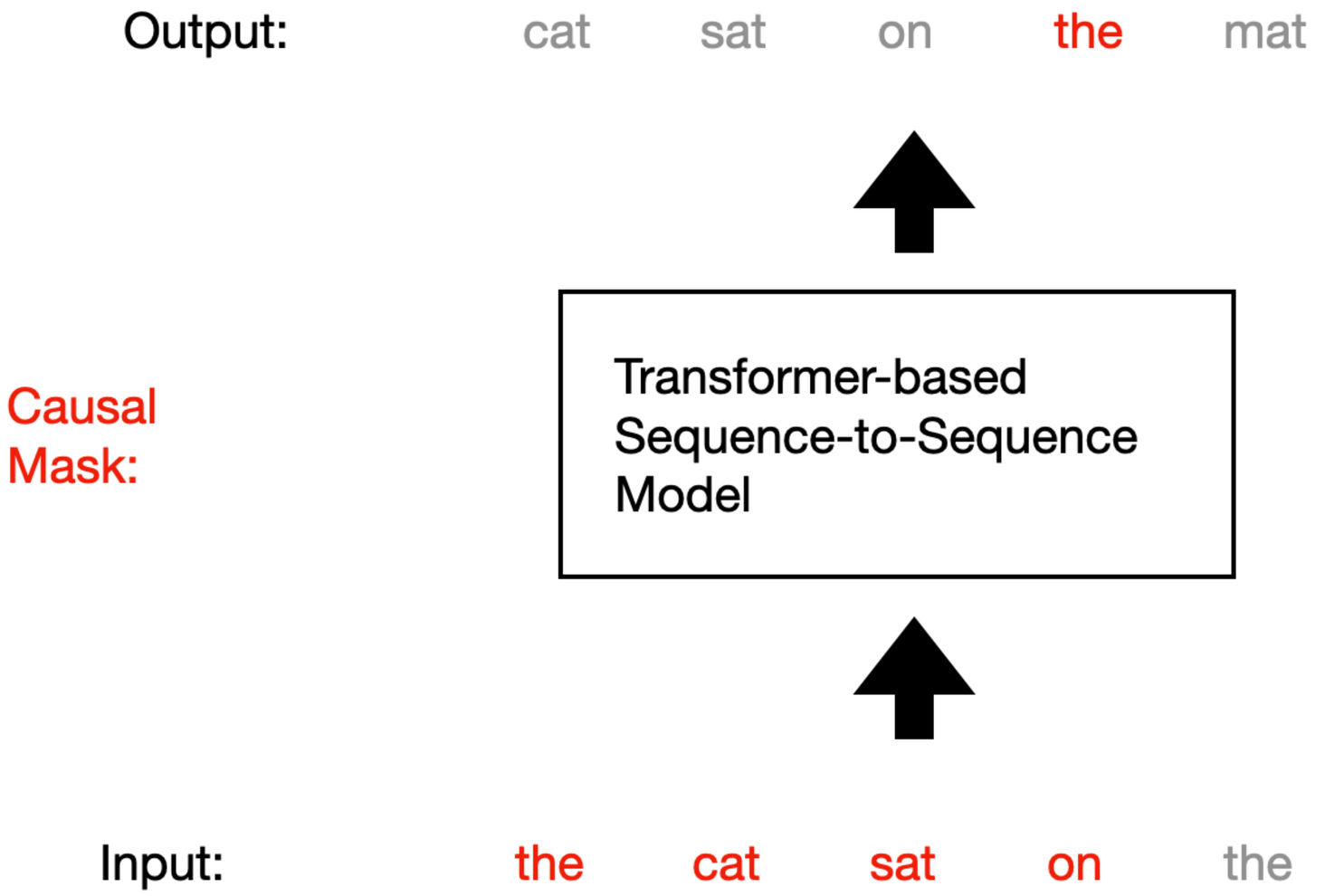
A transformer-based seq2seq model for text generation



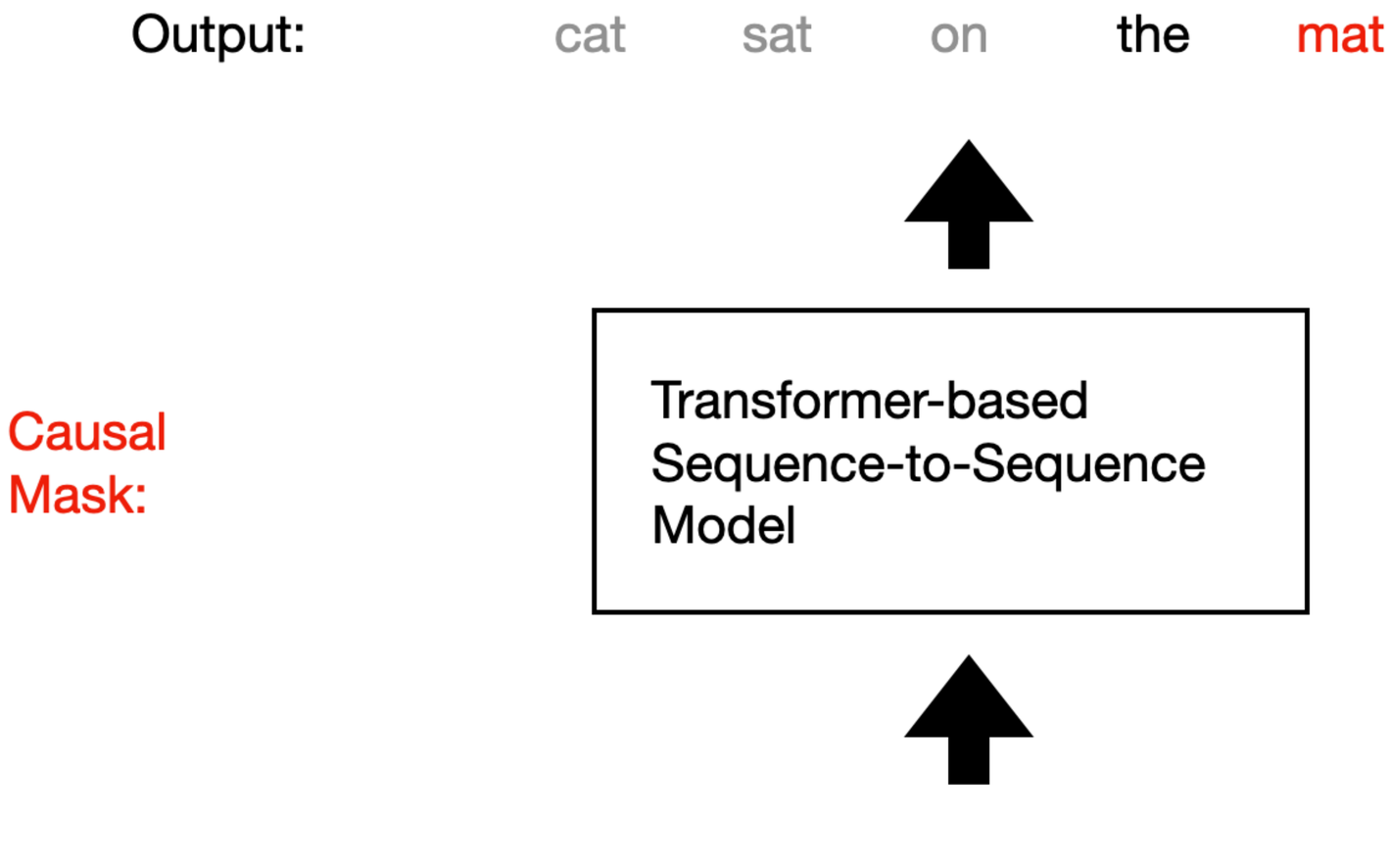
A transformer-based seq2seq model for text generation



A transformer-based seq2seq model for text generation



A transformer-based seq2seq model for text generation



A transformer-based seq2seq model for text generation

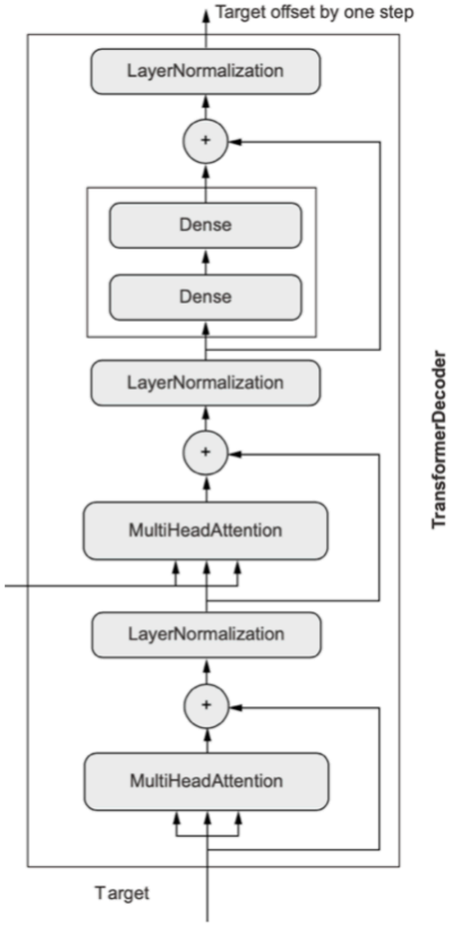
Let's implement our model—we're going to reuse the building blocks we created in chapter 11: `PositionalEmbedding` and `TransformerDecoder`.

Listing 12.6 A simple Transformer-based language model

```
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")
```

Softmax over possible vocabulary words, computed for each output sequence timestep.



A transformer-based seq2seq model for text generation

Let's implement our model—we're going to reuse the building blocks we created in chapter 11: `PositionalEmbedding` and `TransformerDecoder`.

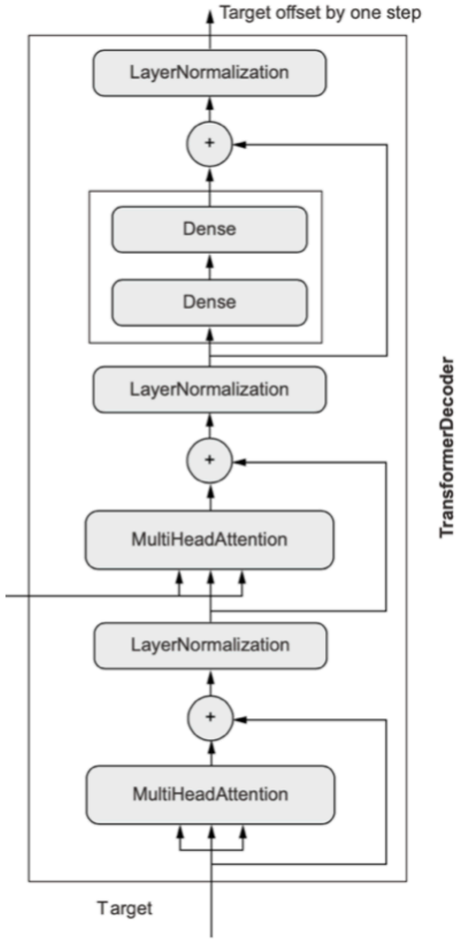
Listing 12.6 A simple Transformer-based language model

```
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")
```

Softmax over possible vocabulary words, computed for each output sequence timestep.

inputs



A transformer-based seq2seq model for text generation

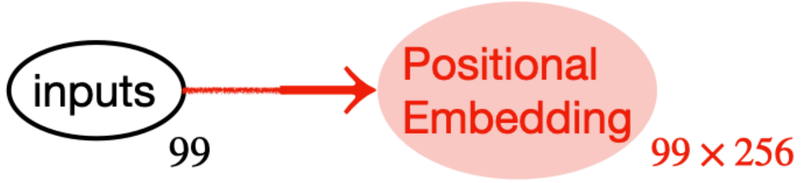
Let's implement our model—we're going to reuse the building blocks we created in chapter 11: `PositionalEmbedding` and `TransformerDecoder`.

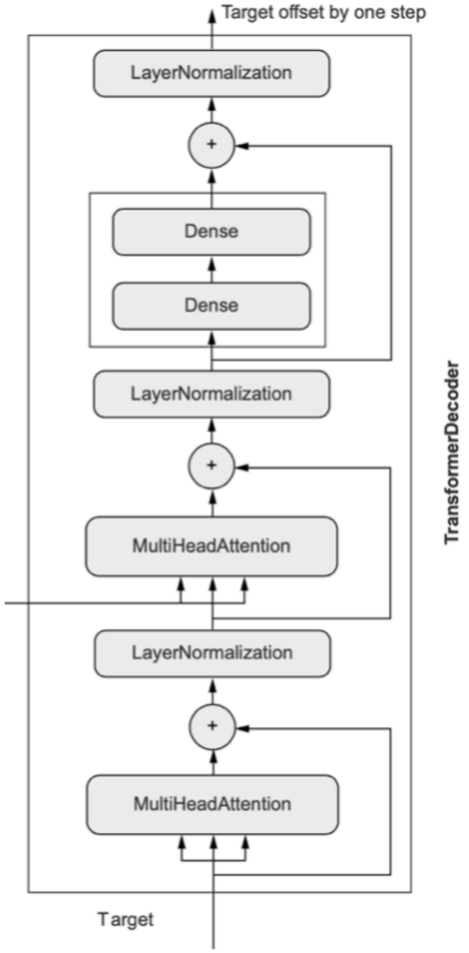
Listing 12.6 A simple Transformer-based language model

```
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")
```

Softmax over possible vocabulary words, computed for each output sequence timestep.





A transformer-based seq2seq model for text generation

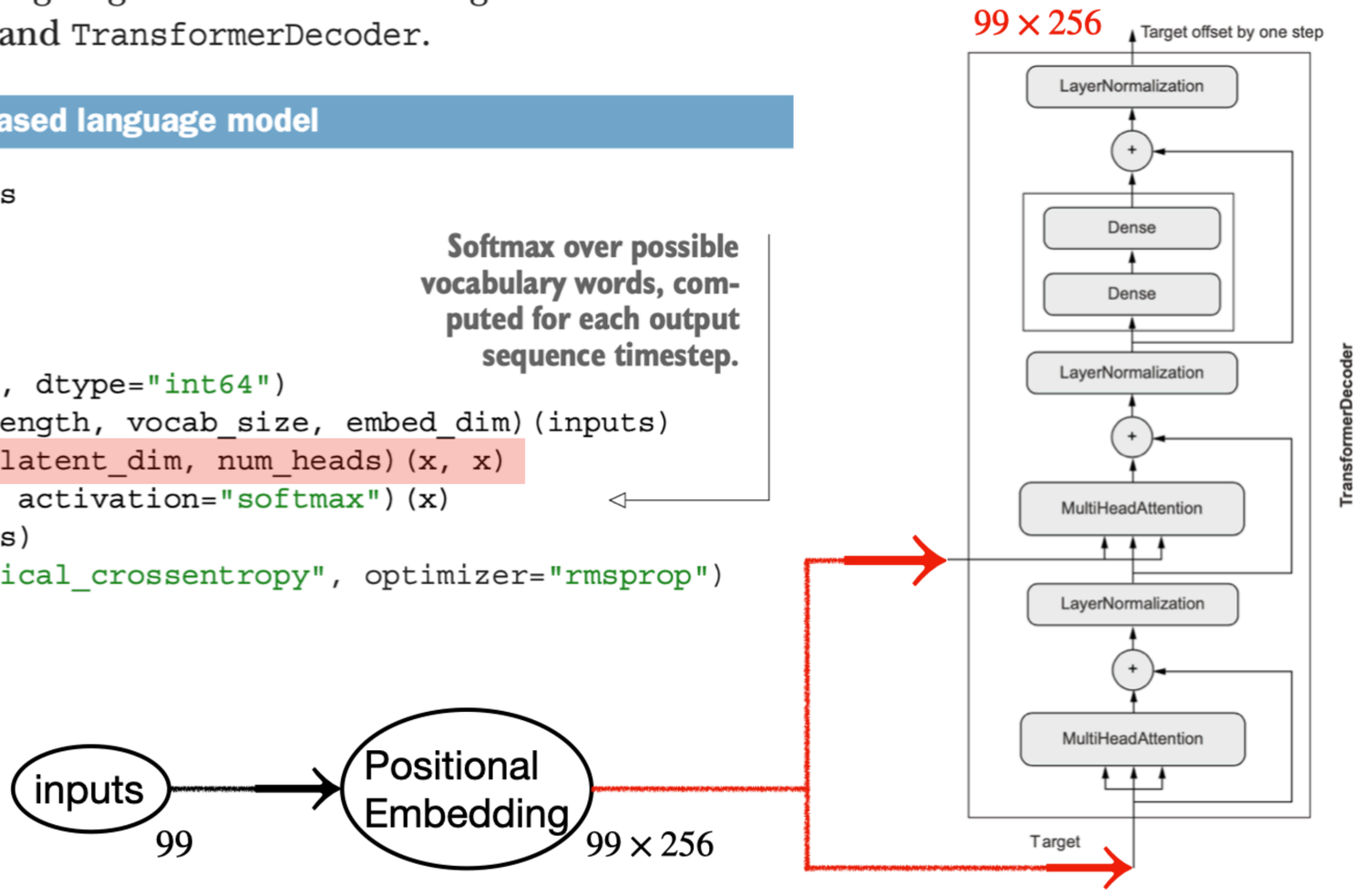
Let's implement our model—we're going to reuse the building blocks we created in chapter 11: `PositionalEmbedding` and `TransformerDecoder`.

Listing 12.6 A simple Transformer-based language model

```
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")
```

Softmax over possible vocabulary words, computed for each output sequence timestep.



A transformer-based seq2seq model for text generation

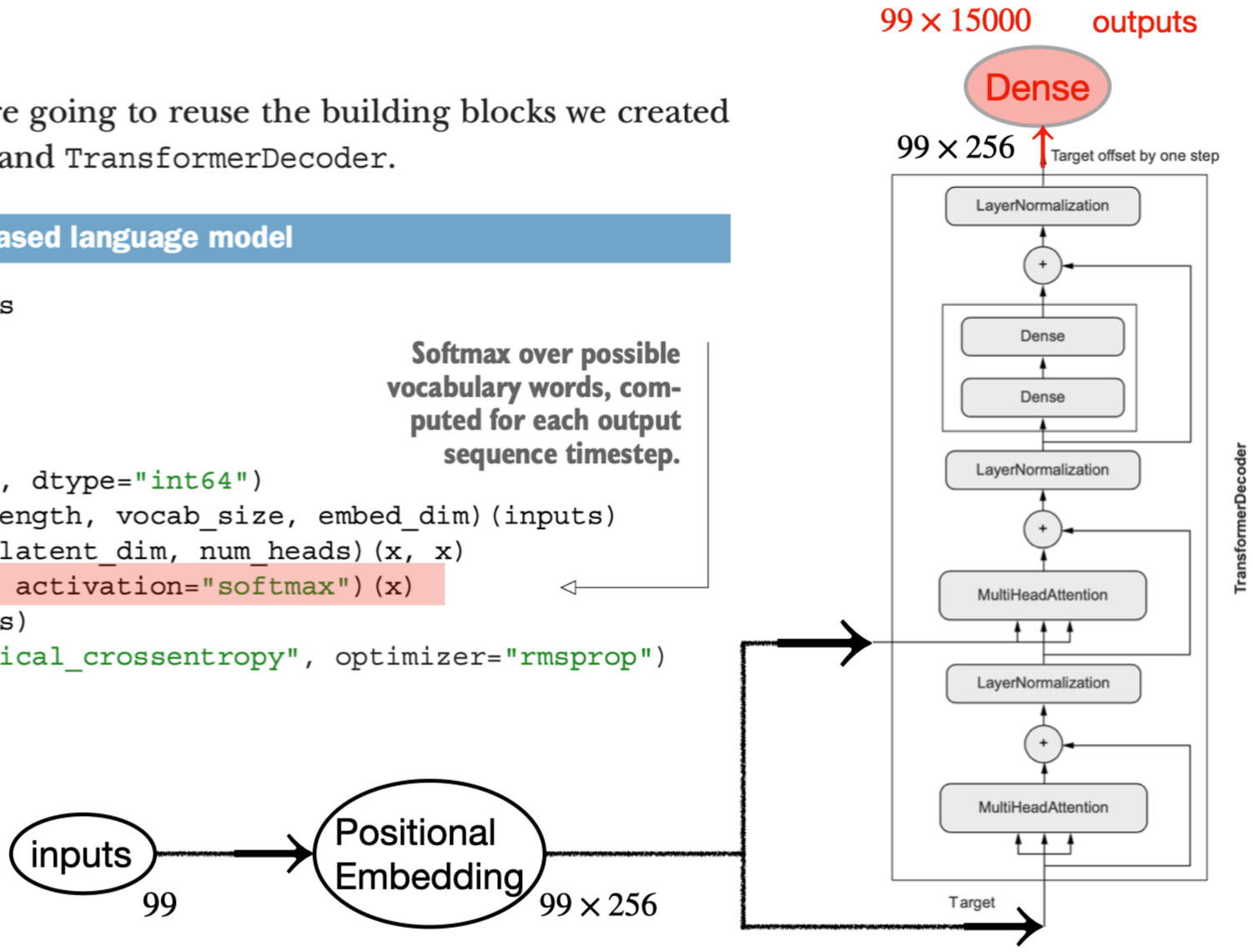
Let's implement our model—we're going to reuse the building blocks we created in chapter 11: `PositionalEmbedding` and `TransformerDecoder`.

Listing 12.6 A simple Transformer-based language model

```
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")
```

Softmax over possible vocabulary words, computed for each output sequence timestep.



A text-generation callback with variable-temperature sampling

We'll use a callback to generate text using a range of different temperatures after every epoch. This allows you to see how the generated text evolves as the model begins to converge, as well as the impact of temperature in the sampling strategy. To seed text generation, we'll use the prompt "this movie": all of our generated texts will start with this.

Listing 12.7 The text-generation callback

```
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)
```

Dict that maps word indices back to strings, to be used for text decoding

Implements variable-temperature sampling from a probability distribution

```
class TextGenerator(keras.callbacks.Callback):
    def __init__(self,
                 prompt,
                 generate_length,
                 model_input_length,
                 temperatures=(1.,),
                 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

    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
            for i in range(self.generate_length):
                tokenized_sentence = text_vectorization([sentence])
                predictions = self.model(tokenized_sentence)
                next_token = sample_next(predictions[0, i, :])
                sampled_token = tokens_index[next_token]
                sentence += " " + sampled_token
            print(sentence)

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))
```

Prompt that we use to seed text generation

How many words to generate

Range of temperatures to use for sampling

When generating text, we start from our prompt.

Feed the current sequence into our model.

Append the new word to the current sequence and repeat.

Retrieve the predictions for the last timestep, and use them to sample a new word.

We'll use a diverse range of temperatures to sample text, to demonstrate the effect of temperature on text generation.

A transformer-based seq2seq model for text generation

Listing 12.8 Fitting the language model

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

Here are some cherrypicked examples of what we're able to generate after 200 epochs of training. Note that punctuation isn't part of our vocabulary, so none of our generated text has any punctuation:

- With `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"

A transformer-based seq2seq model for text generation

- With `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”
- With `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”

A transformer-based seq2seq model for text generation

- With `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"
- With `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"

Quiz questions:

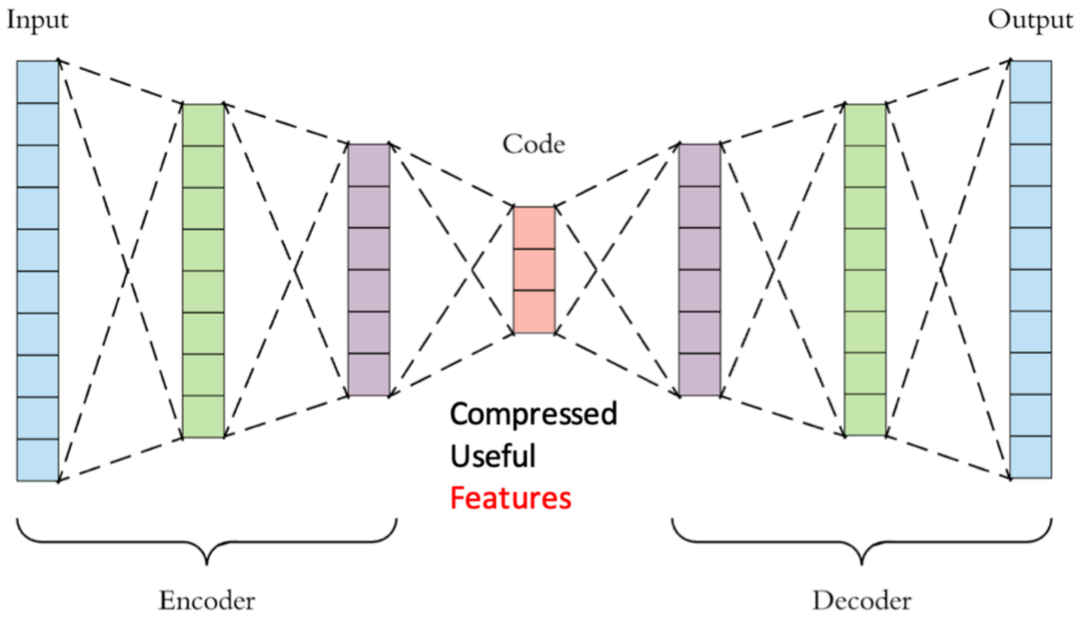
1. How to use Transformer to generate text?

2. What are possible ways to improve the performance of text generation?

Roadmap of this lecture:

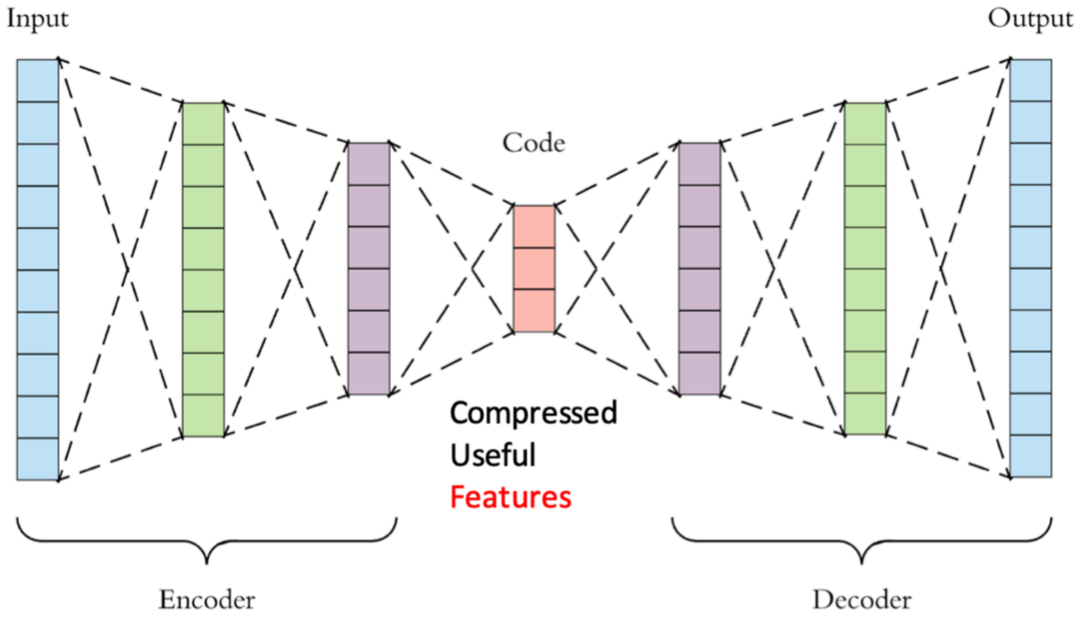
1. Text generation
    - 1.1 General concepts in text generation
    - 1.2 Text generation using IMDB dataset
2. Auto-encoder and VAE (variational auto-encoder)
    - 2.1 Auto-encoder
    - 2.2 VAE
3. GAN (generative adversarial network)
4. Neural style transfer

Auto-Encoder



Partly adapted from lectures by Prof. Hung-yi Lee
“Unsupervised Learning —Auto-encoder”
and “Unsupervised Learning — Generation”.

Auto-Encoder

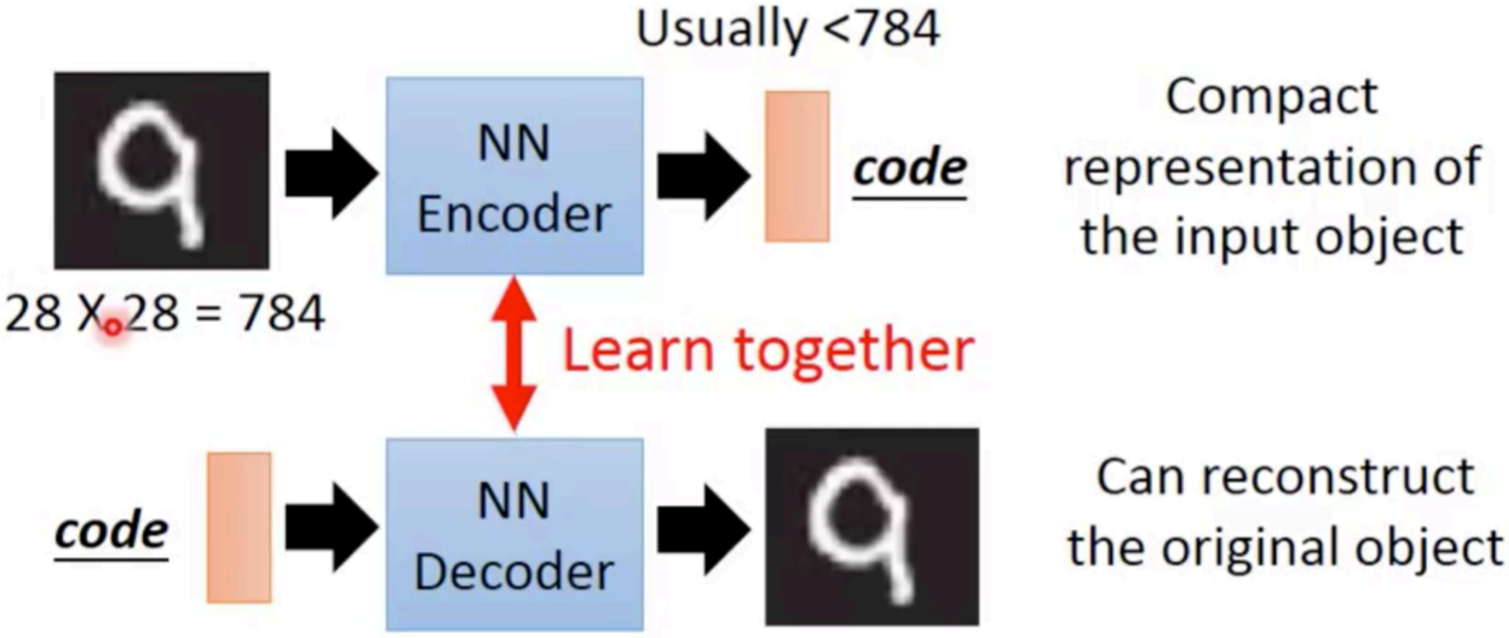


Output: as close to input as possible.

Code: compact, and useful (for feature extraction, generation).

Training: no label needed.

Auto-Encoder



Auto-Encoder

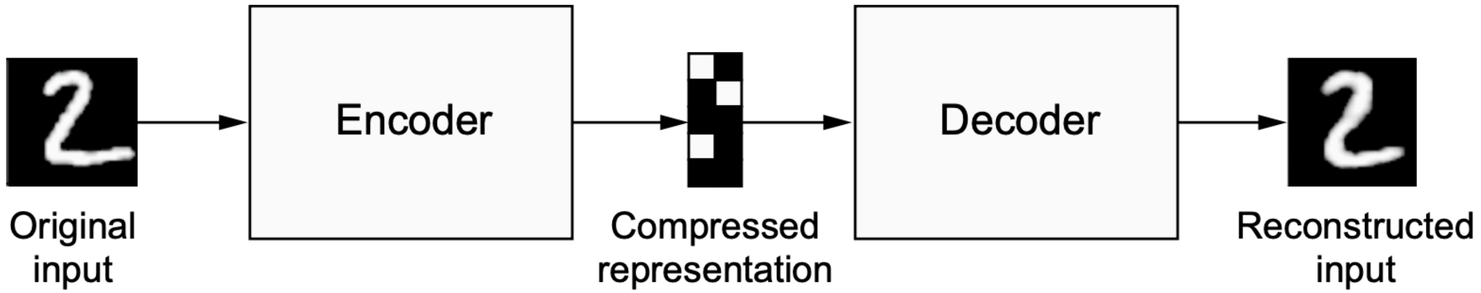
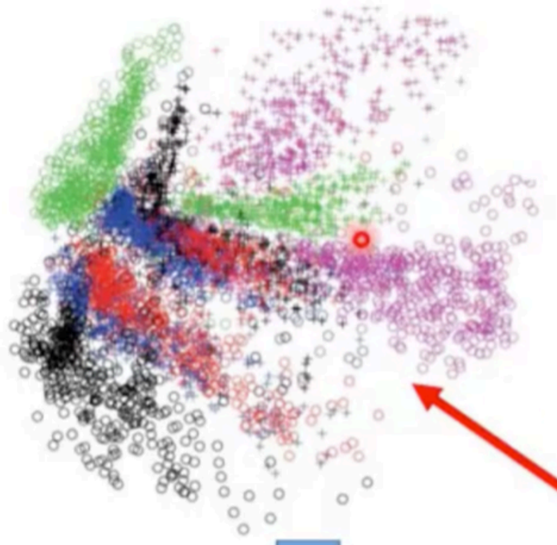
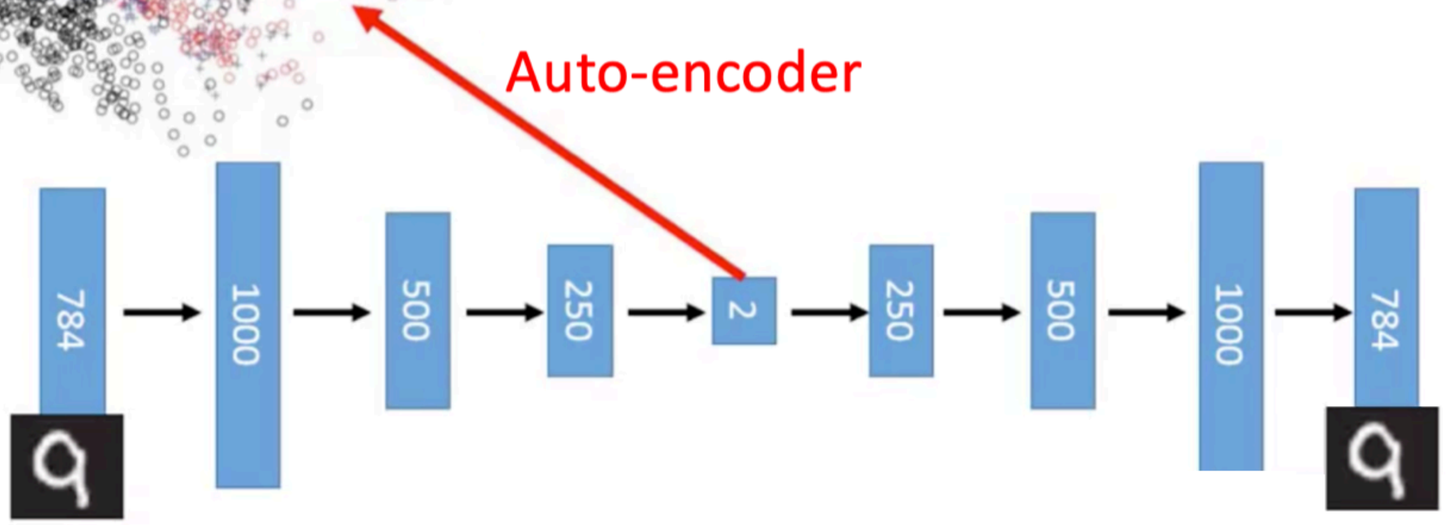


Figure 12.16 An autoencoder mapping an input *x* to a compressed representation and then decoding it back as *x'*

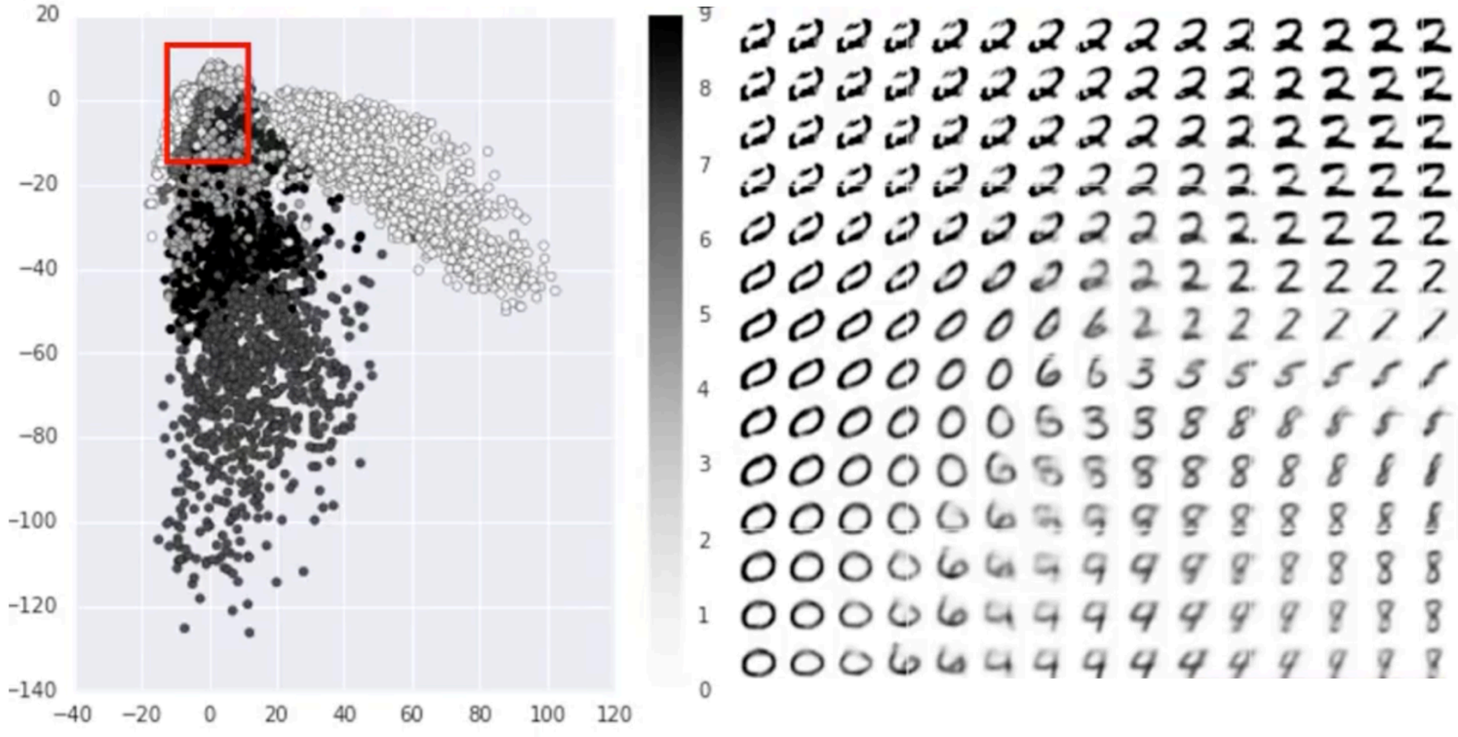
Auto-Encoder



The 10 clusters (for 10 digits) are relatively well separated.

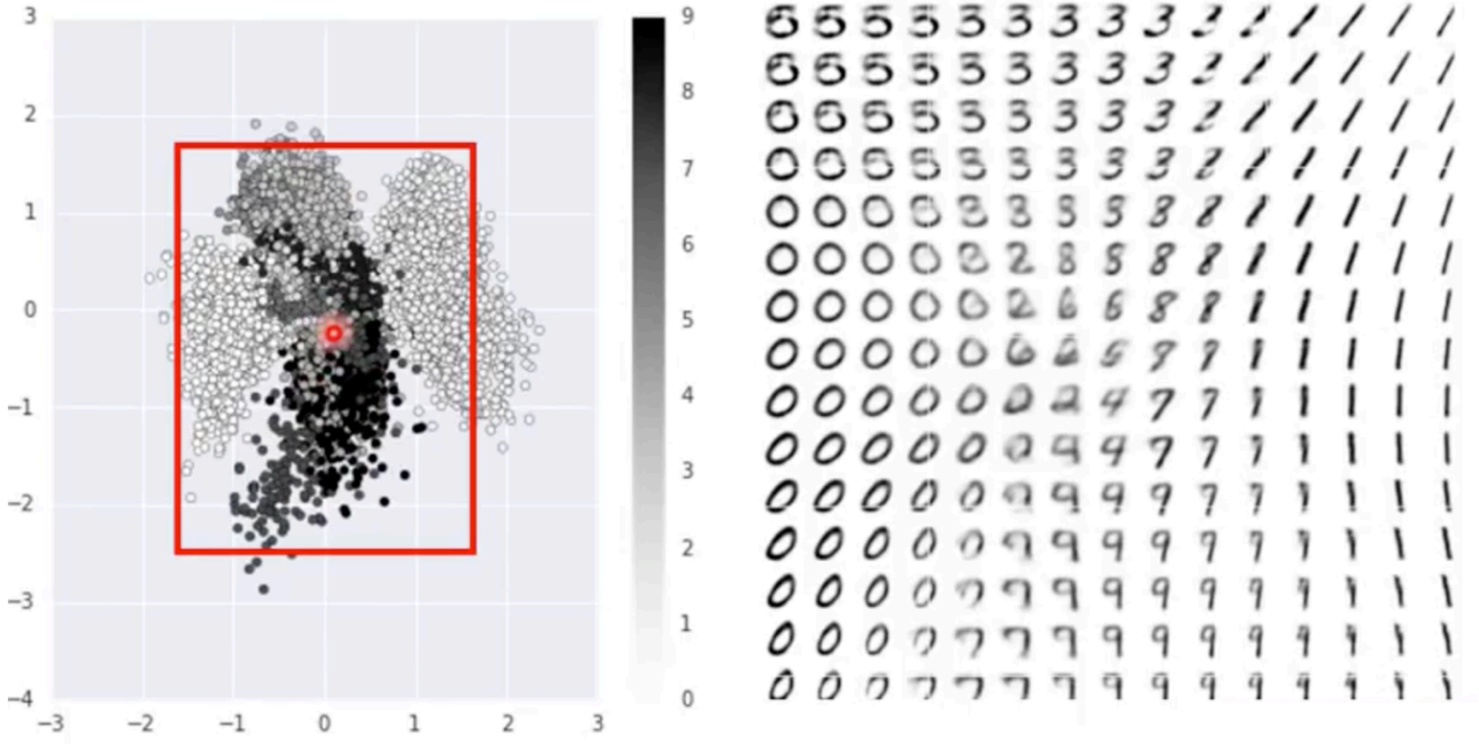


Auto-Encoder

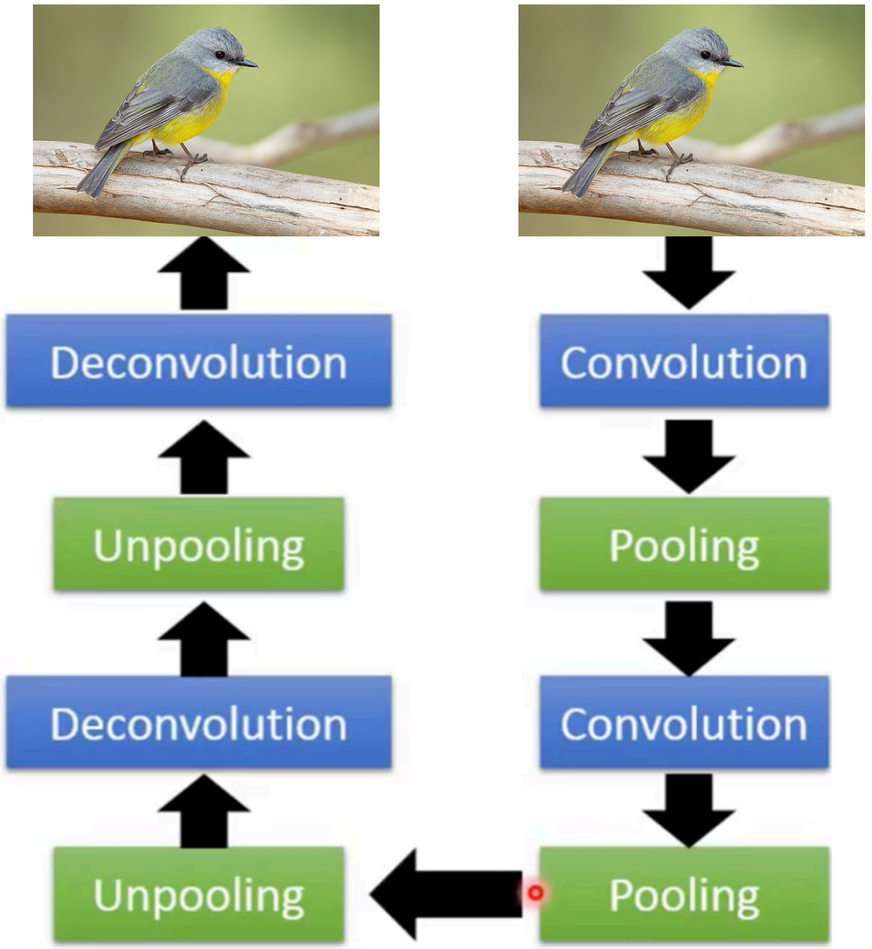


Auto-Encoder

For higher-dimensional embedding (which is harder to visualize and see where to sample), use L2 regularization in auto-encoder so that the samples are centered around the origin point.



Auto-Encoder for CNN

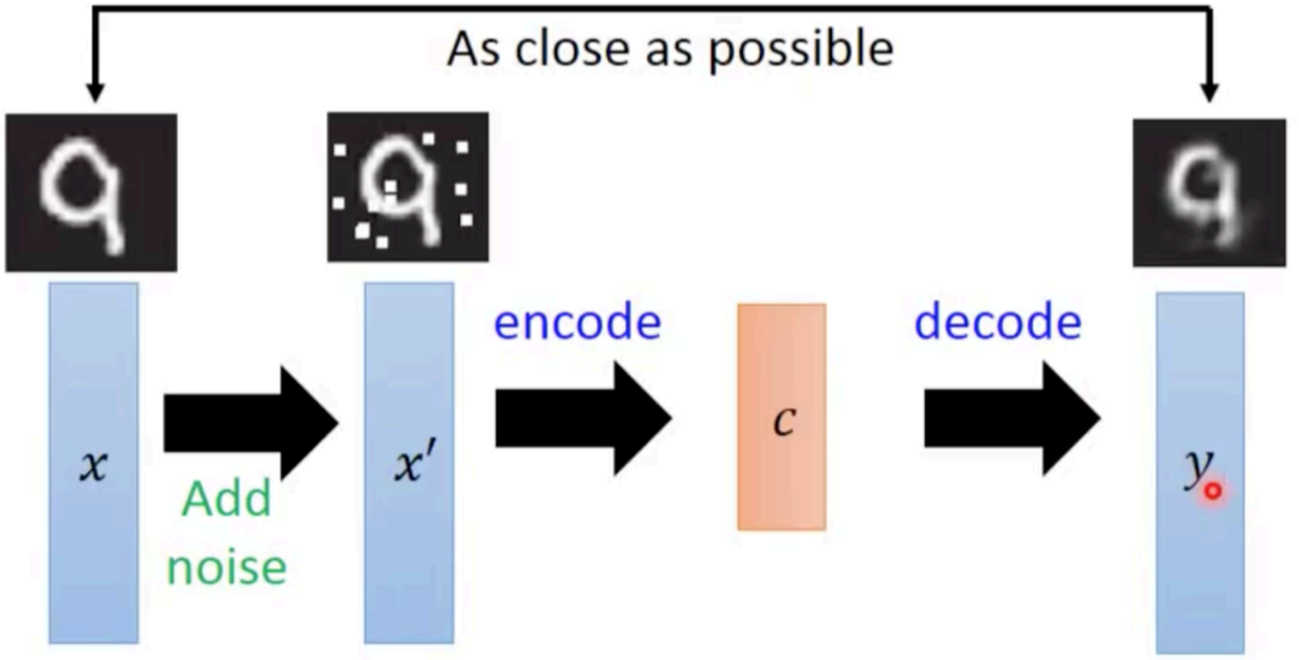


Unpooling: (approximate) reverse operation of pooling, for upsampling.

We can also use strided-convolution and strided-deconvolution.

Auto-Encoder

Another application: denoising (remove noise in data), which is also useful for data generation (e.g., stable-diffusion type of generation).



Vincent, Pascal, et al. "Extracting and composing robust features with denoising autoencoders." *ICML*, 2008.

Quiz questions:

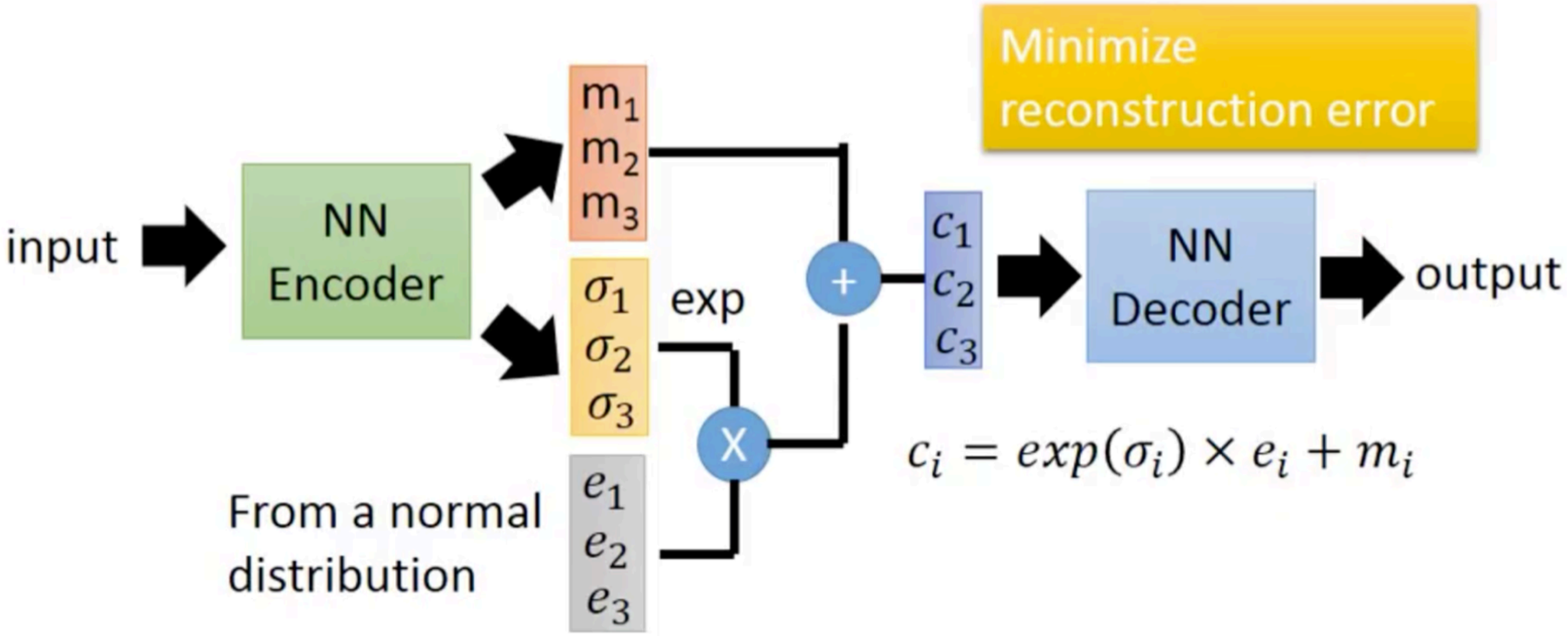
1. What is the basic idea of auto-encoder?

2. How to design an auto-encoder to help its encoder extract features and help its decoder generate data?

Roadmap of this lecture:

1. Text generation
    - 1.1 General concepts in text generation
    - 1.2 Text generation using IMDB dataset
2. Auto-encoder and VAE (variational auto-encoder)
    - 2.1 Auto-encoder
    - 2.2 VAE
3. GAN (generative adversarial network)
4. Neural style transfer

VAE (Variational Auto-Encoder)



Encoder outputs a distribution instead of a point in the embedding space: make nearby embeddings represent similar outputs (to get smoothness in embedding space).

VAE (Variational Auto-Encoder)

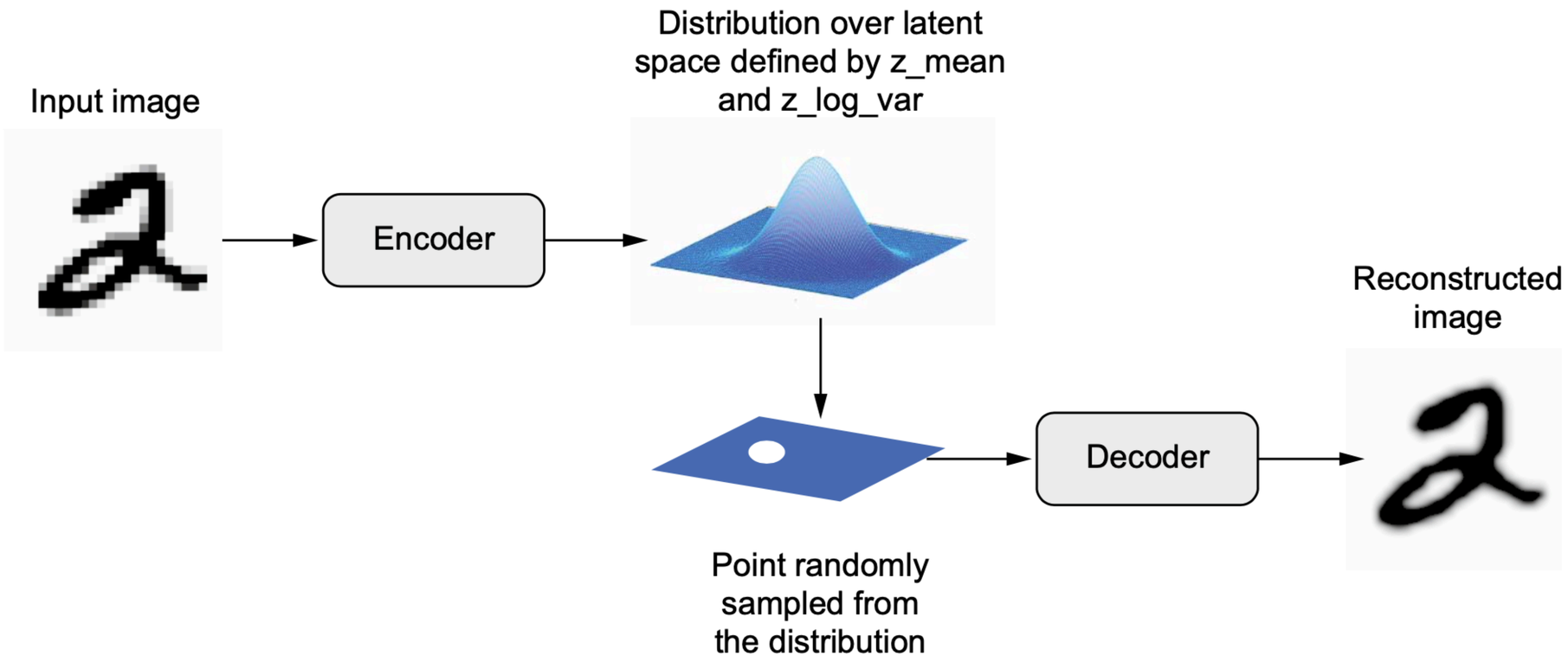
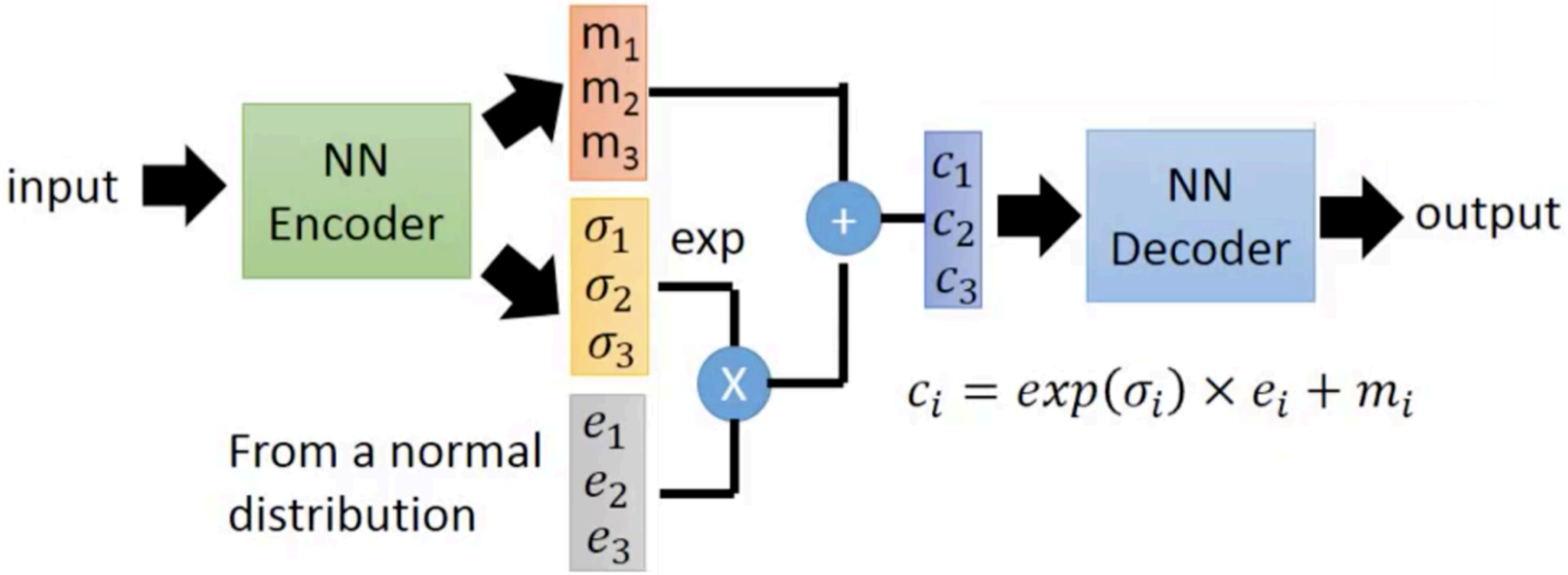


Figure 12.17 A VAE maps an image to two vectors, `z_mean` and `z_log_sigma`, which define a probability distribution over the latent space, used to sample a latent point to decode.

VAE (Variational Auto-Encoder)



Cost Function: Difference between output and input +

$$-\sum_{i=1}^{3}\left(1+\sigma_i-(m_i)^2-exp(\sigma_i)\right)$$

2nd term: to make the mean m_i close to 0 and the standard deviation exp(σ_i) close to 1.

Generate Images with Variational Auto-Encoder (VAE)

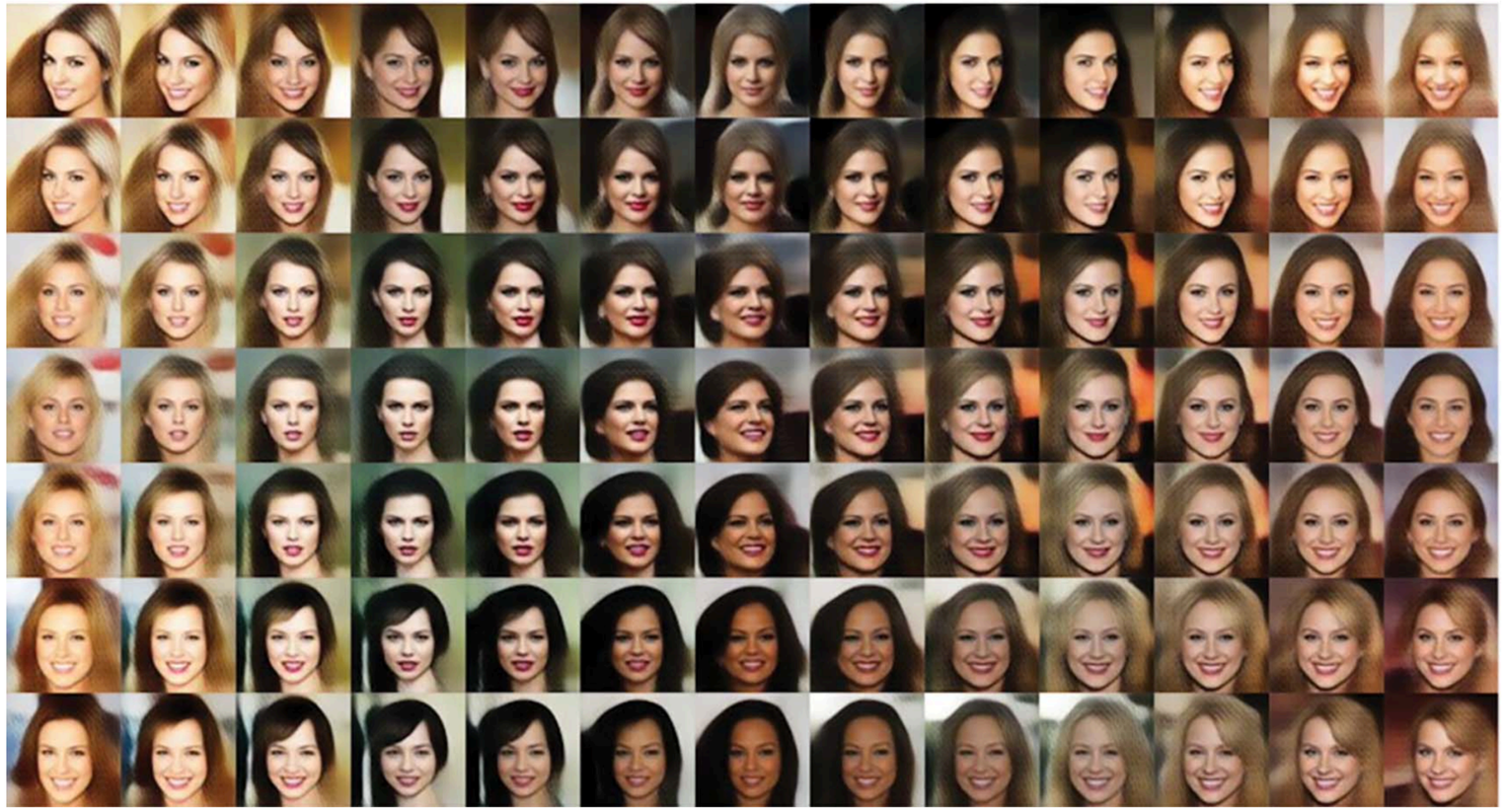


Figure 12.14 A continuous space of faces generated by Tom White using VAEs

Implement VAE

We're going to be implementing a VAE that can generate MNIST digits. It's going to have three parts:

- An encoder network that turns a real image into a mean and a variance in the latent space
- A sampling layer that takes such a mean and variance, and uses them to sample a random point from the latent space
- A decoder network that turns points from the latent space back into images

Implement VAE: use strides for downsampling

The following listing shows the encoder network we'll use, mapping images to the parameters of a probability distribution over the latent space. It's a simple convnet that maps the input image `x` to two vectors, `z_mean` and `z_log_var`. One important detail is that we use strides for downsampling feature maps instead of max pooling. The last time we did this was in the image segmentation example in chapter 9. Recall that, in general, strides are preferable to max pooling for any model that cares about *information location*—that is to say, *where* stuff is in the image—and this one does, since it will have to produce an image encoding that can be used to reconstruct a valid image.

VAE Encoder

Listing 12.24 VAE encoder network

```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.

VAE Encoder

Listing 12.24 VAE encoder network

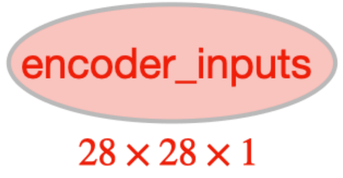
```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.



VAE Encoder

Listing 12.24 VAE encoder network

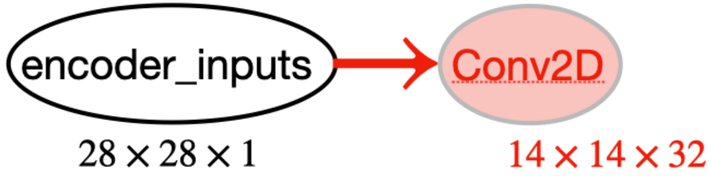
```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.



VAE Encoder

Listing 12.24 VAE encoder network

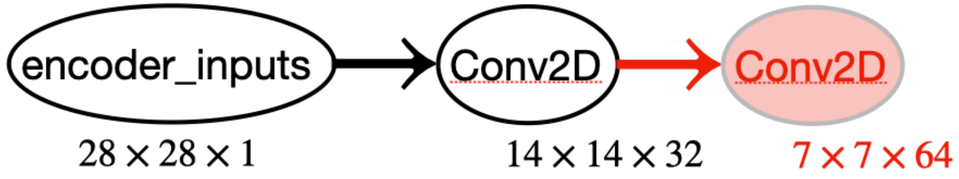
```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.



VAE Encoder

Listing 12.24 VAE encoder network

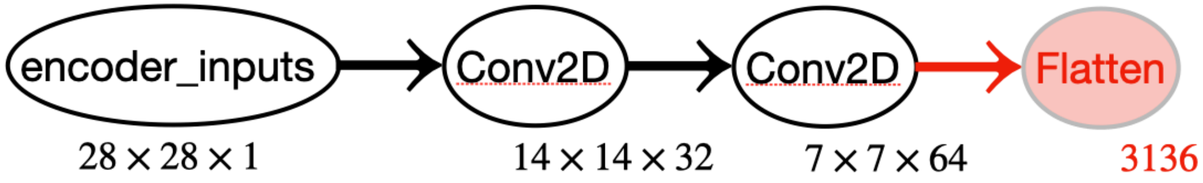
```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.



VAE Encoder

Listing 12.24 VAE encoder network

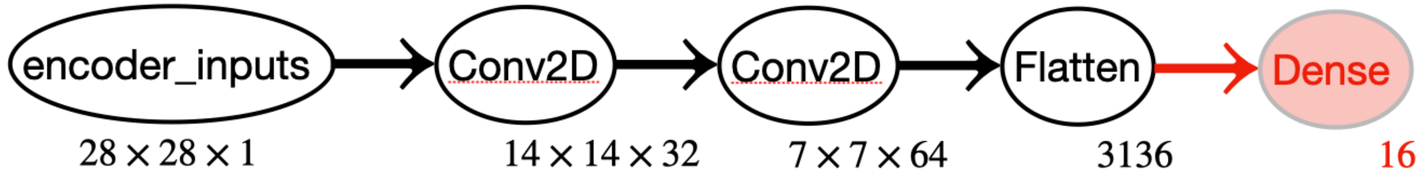
```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.



VAE Encoder

Listing 12.24 VAE encoder network

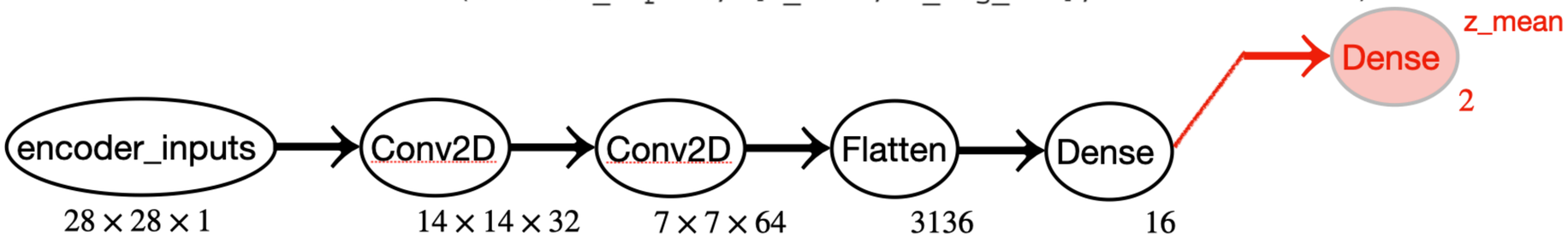
```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.



VAE Encoder

Listing 12.24 VAE encoder network

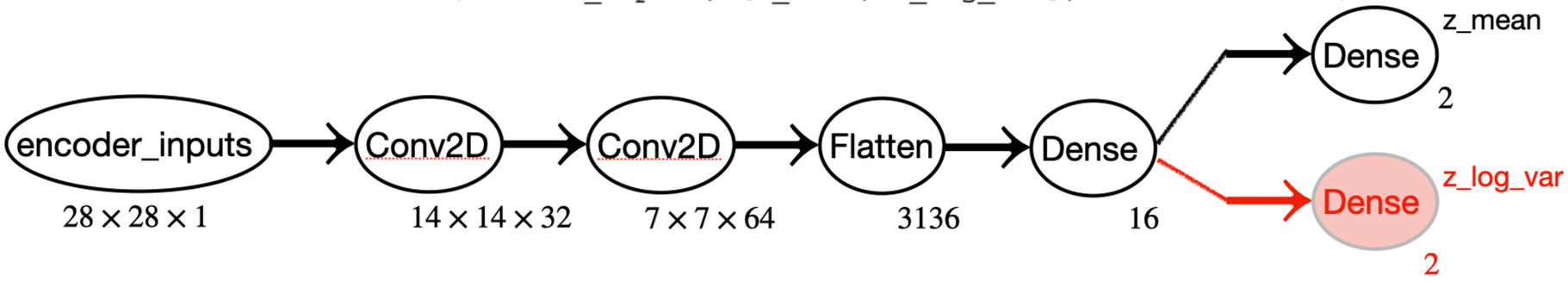
```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.



Sample Latent Space

Next is the code for using `z_mean` and `z_log_var`, the parameters of the statistical distribution assumed to have produced `input_img`, to generate a latent space point `z`.

Listing 12.25 Latent-space-sampling layer

```
import tensorflow as tf

class Sampler(layers.Layer):
    def call(self, z_mean, z_log_var):
        batch_size = tf.shape(z_mean)[0]
        z_size = tf.shape(z_mean)[1]
        epsilon = tf.random.normal(shape=(batch_size, z_size))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

Apply the VAE sampling formula.

Draw a batch of random normal vectors.

$$\mu + \sigma\epsilon$$

VAE Decoder

The following listing shows the decoder implementation. We reshape the vector z to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 12.26 VAE decoder network, mapping latent space points to images

Input where we'll feed z

Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

The output ends up with shape (28, 28, 1).

VAE Decoder

The following listing shows the decoder implementation. We reshape the vector z to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 12.26 VAE decoder network, mapping latent space points to images

Input where we'll feed z

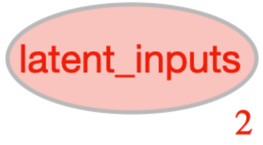
Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

The output ends up with shape (28, 28, 1).



VAE Decoder

The following listing shows the decoder implementation. We reshape the vector `z` to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 12.26 VAE decoder network, mapping latent space points to images

Input where we'll feed z

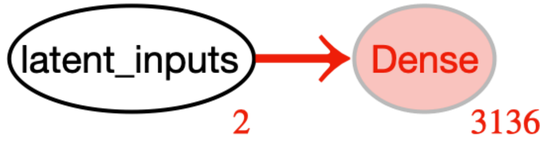
Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

The output ends up with shape (28, 28, 1).



VAE Decoder

The following listing shows the decoder implementation. We reshape the vector z to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 12.26 VAE decoder network, mapping latent space points to images

Input where we'll feed z

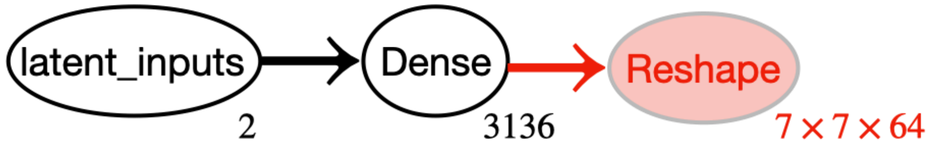
Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

The output ends up with shape (28, 28, 1).



VAE Decoder

The following listing shows the decoder implementation. We reshape the vector z to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 12.26 VAE decoder network, mapping latent space points to images

Input where we'll feed z

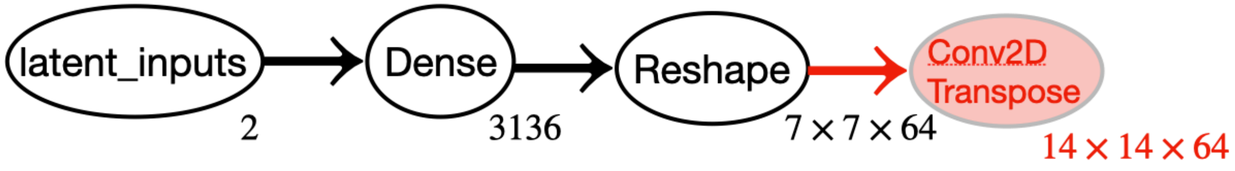
Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

The output ends up with shape (28, 28, 1).



VAE Decoder

The following listing shows the decoder implementation. We reshape the vector `z` to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 12.26 VAE decoder network, mapping latent space points to images

Input where we'll feed z

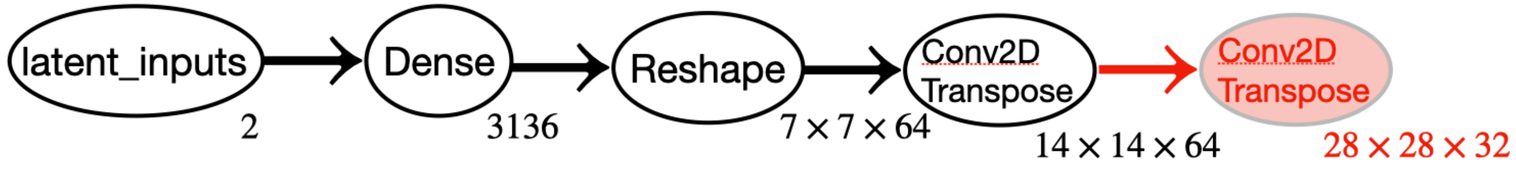
Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

The output ends up with shape (28, 28, 1).



VAE Decoder

The following listing shows the decoder implementation. We reshape the vector `z` to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 12.26 VAE decoder network, mapping latent space points to images

Input where we'll feed z

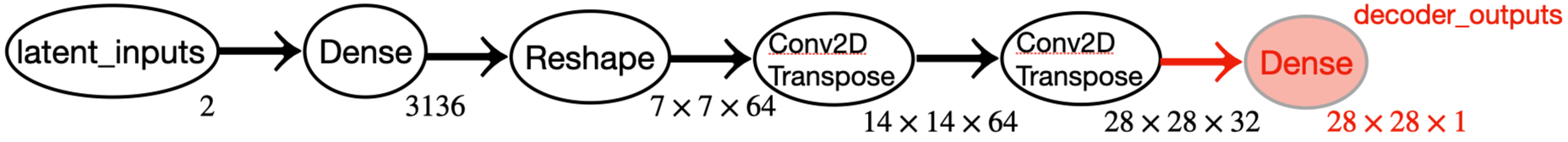
Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

The output ends up with shape (28, 28, 1).



Build the VAE Model

Now let's create the VAE model itself. This is your first example of a model that isn't doing supervised learning (an autoencoder is an example of *self-supervised* learning, because it uses its inputs as targets). Whenever you depart from classic supervised learning, it's common to subclass the `Model` class and implement a custom `train_step()` to specify the new training logic, a workflow you learned about in chapter 7. That's what we'll do here.

Build the VAE Model

Listing 12.27 VAE model with custom `train_step()`

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.sampler = Sampler()
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss")
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [self.total_loss_tracker,
                self.reconstruction_loss_tracker,
                self.kl_loss_tracker]
```

We use these metrics to keep track of the loss averages over each epoch.

We list the metrics in the metrics property to enable the model to reset them after each epoch (or between multiple calls to fit()/evaluate()).

Build the VAE Model

```
def train_step(self, data):
    with tf.GradientTape() as tape:
        z_mean, z_log_var = self.encoder(data)
        z = self.sampler(z_mean, z_log_var)
        reconstruction = decoder(z)
        reconstruction_loss = tf.reduce_mean(
            tf.reduce_sum(
                keras.losses.binary_crossentropy(data, reconstruction),
                axis=(1, 2)
            )
        )
        kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
           tf.exp(z_log_var))
        total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)

        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {
            "total_loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.reconstruction_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }
```

We sum the reconstruction loss over the spatial dimensions (axes 1 and 2) and take its mean over the batch dimension.

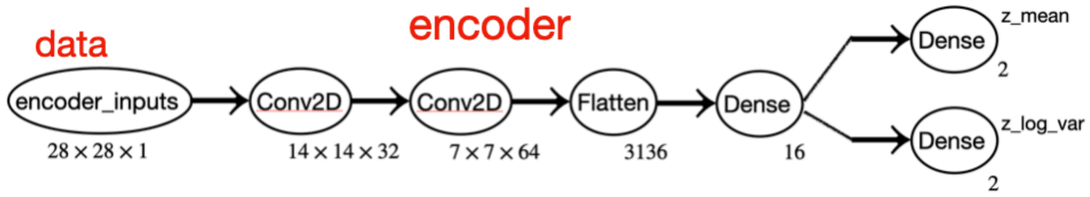
Add the regularization term (Kullback–Leibler divergence).

Build the VAE Model

```
def train_step(self, data):
    with tf.GradientTape() as tape:
        z_mean, z_log_var = self.encoder(data)
        z = self.sampler(z_mean, z_log_var)
        reconstruction = decoder(z)
        reconstruction_loss = tf.reduce_mean(
            tf.reduce_sum(
                keras.losses.binary_crossentropy(data, reconstruction),
                axis=(1, 2)
            )
        )
        kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
           tf.exp(z_log_var))
        total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)
```

We sum the reconstruction loss over the spatial dimensions (axes 1 and 2) and take its mean over the batch dimension.

Add the regularization term (Kullback–Leibler divergence).

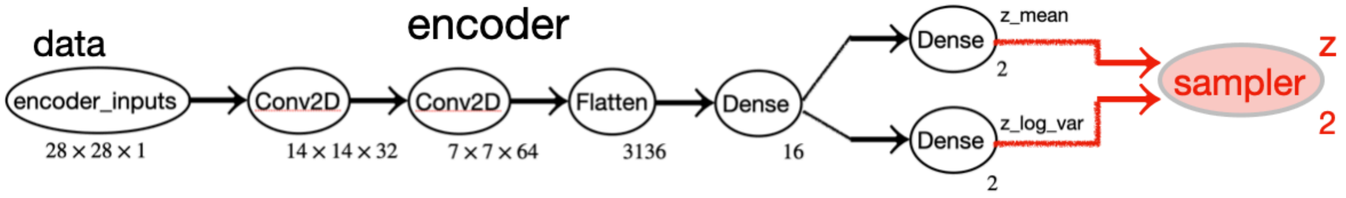


Build the VAE Model

```
def train_step(self, data):
    with tf.GradientTape() as tape:
        z_mean, z_log_var = self.encoder(data)
        z = self.sampler(z_mean, z_log_var)
        reconstruction = decoder(z)
        reconstruction_loss = tf.reduce_mean(
            tf.reduce_sum(
                keras.losses.binary_crossentropy(data, reconstruction),
                axis=(1, 2)
            )
        )
        kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
           tf.exp(z_log_var))
        total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)
```

We sum the reconstruction loss over the spatial dimensions (axes 1 and 2) and take its mean over the batch dimension.

Add the regularization term (Kullback–Leibler divergence).

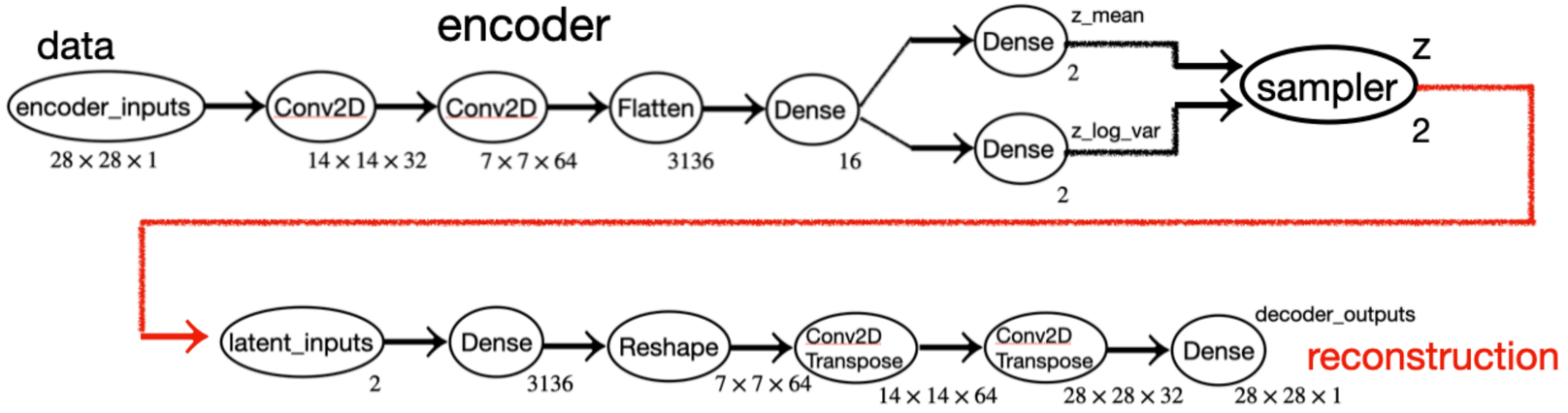


Build the VAE Model

```
def train_step(self, data):
    with tf.GradientTape() as tape:
        z_mean, z_log_var = self.encoder(data)
        z = self.sampler(z_mean, z_log_var)
        reconstruction = decoder(z)
        reconstruction_loss = tf.reduce_mean(
            tf.reduce_sum(
                keras.losses.binary_crossentropy(data, reconstruction),
                axis=(1, 2)
            )
        )
        kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
           tf.exp(z_log_var))
        total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)
```

We sum the reconstruction loss over the spatial dimensions (axes 1 and 2) and take its mean over the batch dimension.

Add the regularization term (Kullback–Leibler divergence).



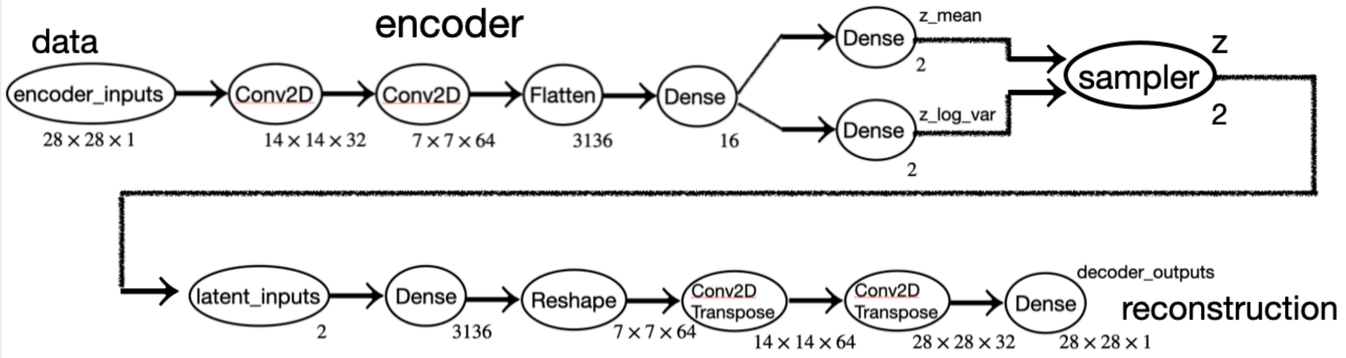
Build the VAE Model

```
def train_step(self, data):
    with tf.GradientTape() as tape:
        z_mean, z_log_var = self.encoder(data)
        z = self.sampler(z_mean, z_log_var)
        reconstruction = decoder(z)
        reconstruction_loss = tf.reduce_mean(
            tf.reduce_sum(
                keras.losses.binary_crossentropy(data, reconstruction),
                axis=(1, 2)
            )
        )
        kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
           tf.exp(z_log_var))
        total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)
```

We sum the reconstruction loss over the spatial dimensions (axes 1 and 2) and take its mean over the batch dimension.

Add the regularization term (Kullback–Leibler divergence).

kl_loss: KL-divergence between two Gaussian distributions N(z_mean,z_var) and N(0,1).



Train the VAE Model

Finally, we're ready to instantiate and train the model on MNIST digits. Because the loss is taken care of in the custom layer, we don't specify an external loss at compile time (`loss=None`), which in turn means we won't pass target data during training (as you can see, we only pass `x_train` to the model in `fit()`).

Listing 12.28 Training the VAE

```
import numpy as np

(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
mnist_digits = np.concatenate([x_train, x_test], axis=0)
mnist_digits = np.expand_dims(mnist_digits, -1).astype("float32") / 255

vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam(), run_eagerly=True)
vae.fit(mnist_digits, epochs=30, batch_size=128)
```

We train on all MNIST digits, so we concatenate the training and test samples.

Note that we don't pass targets in fit(), since train_step() doesn't expect any.

Note that we don't pass a loss argument in compile(), since the loss is already part of the train_step().

Use trained VAE to generate images

Once the model is trained, we can use the `decoder` network to turn arbitrary latent space vectors into images.

Listing 12.29 Sampling a grid of images from the 2D latent space

```
import matplotlib.pyplot as plt

n = 30                                  # We'll display a grid of 30 × 30 digits (900 digits total).
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

grid_x = np.linspace(-1, 1, n)          # Sample points linearly on a 2D grid.
grid_y = np.linspace(-1, 1, n)[::-1]

for i, yi in enumerate(grid_y):         # Iterate over grid locations.
    for j, xi in enumerate(grid_x):
        z_sample = np.array([[xi, yi]])
        x_decoded = vae.decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[                         # For each location, sample a digit and add it to our figure.
            i * digit_size : (i + 1) * digit_size,
```

Use trained VAE to generate images

```
                j * digit_size : (j + 1) * digit_size,
            ] = digit

plt.figure(figsize=(15, 15))
start_range = digit_size // 2
end_range = n * digit_size + start_range
pixel_range = np.arange(start_range, end_range, digit_size)
sample_range_x = np.round(grid_x, 1)
sample_range_y = np.round(grid_y, 1)
plt.xticks(pixel_range, sample_range_x)
plt.yticks(pixel_range, sample_range_y)
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.axis("off")
plt.imshow(figure, cmap="Greys_r")
```

Use trained VAE to generate images

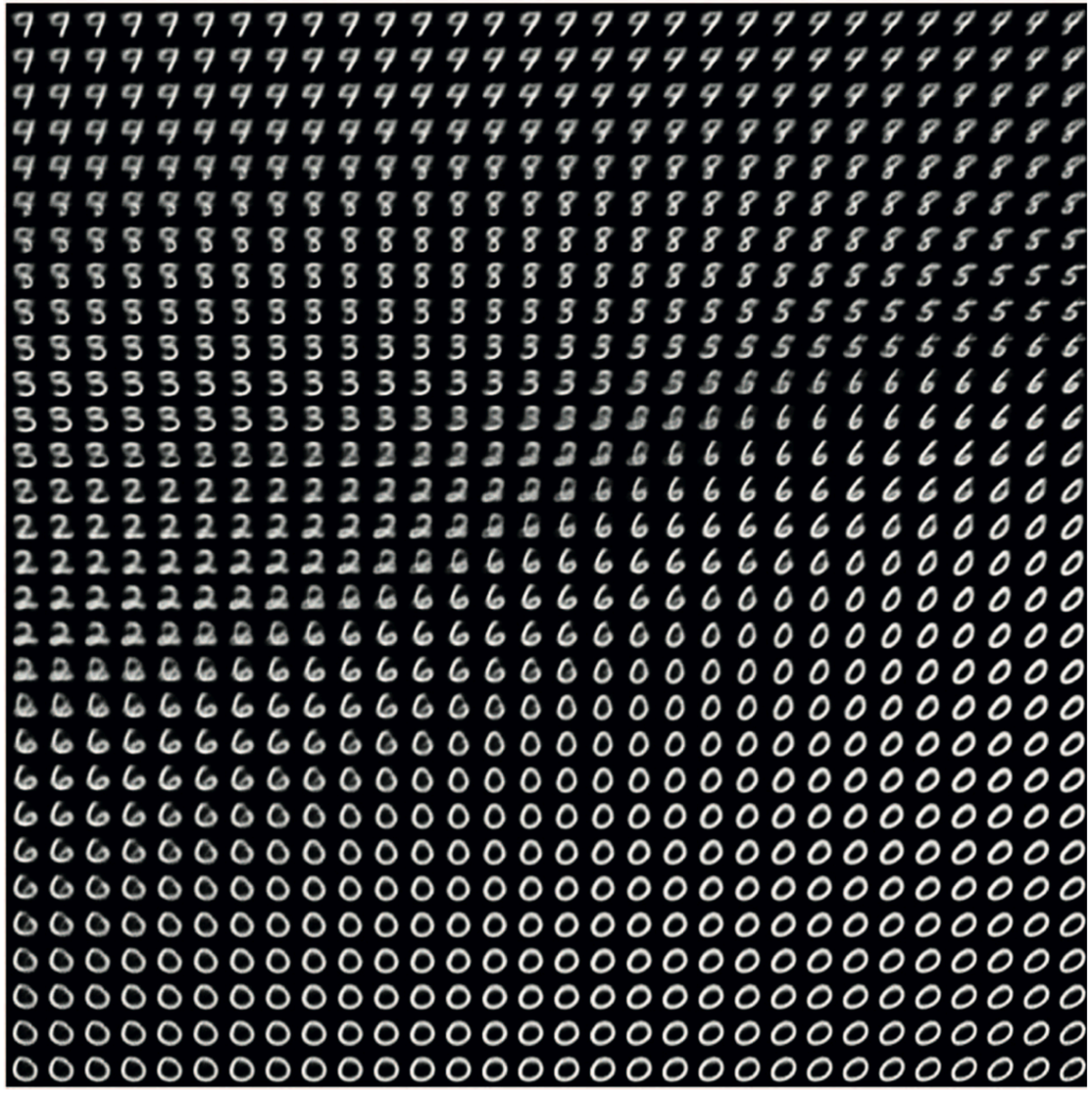


Figure 12.18 Grid of digits decoded from the latent space

Quiz questions:

1. How is VAE different from an ordinary auto-encoder?

2. Why does a VAE encoder generate a distribution instead of a point in the embedding space?

Roadmap of this lecture:

1. Text generation
    - 1.1 General concepts in text generation
    - 1.2 Text generation using IMDB dataset
2. Auto-encoder and VAE (variational auto-encoder)
    - 2.1 Auto-encoder
    - 2.2 VAE
3. GAN (generative adversarial network)
4. Neural style transfer

Generative Adversarial Network (GAN)

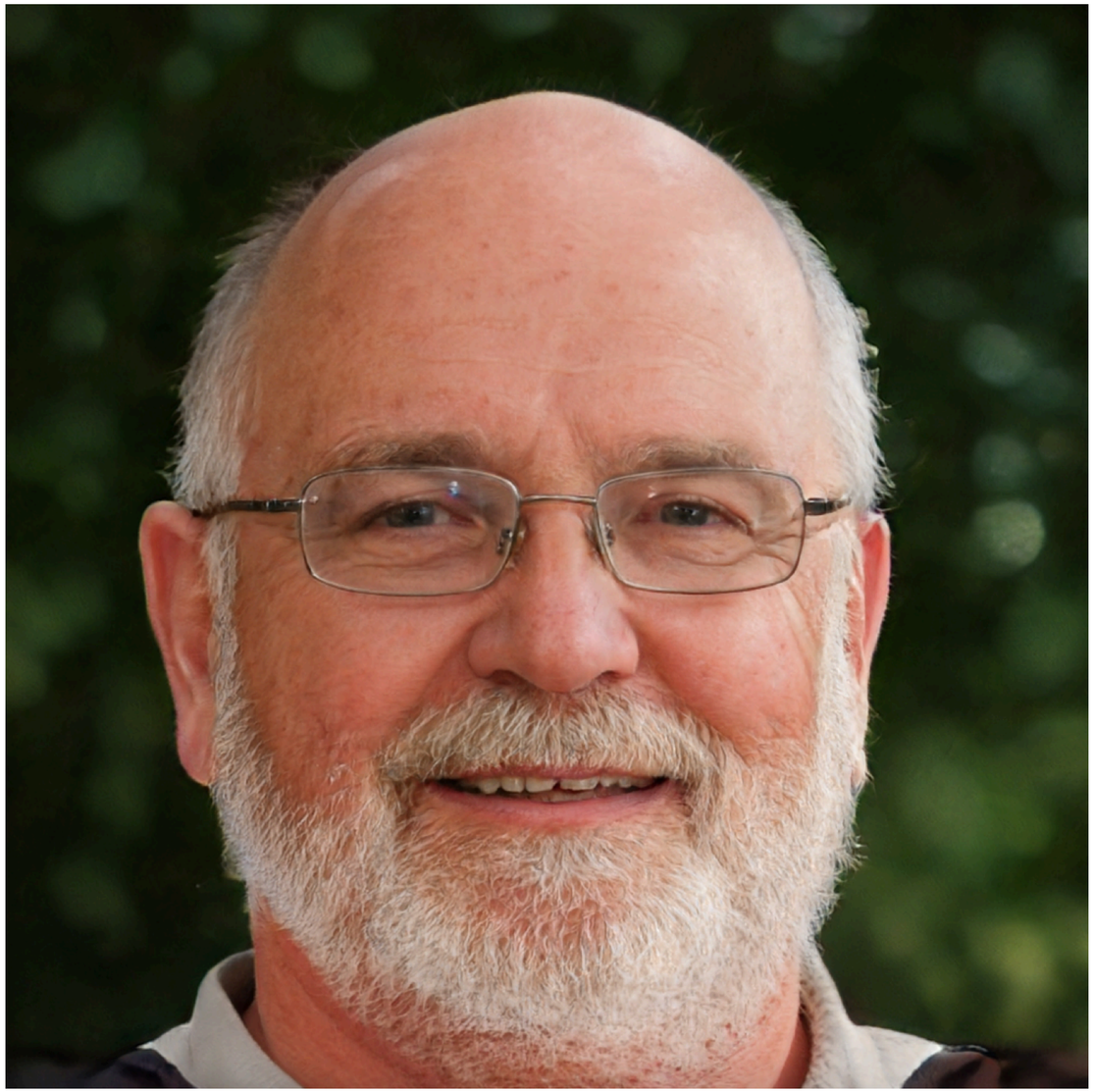
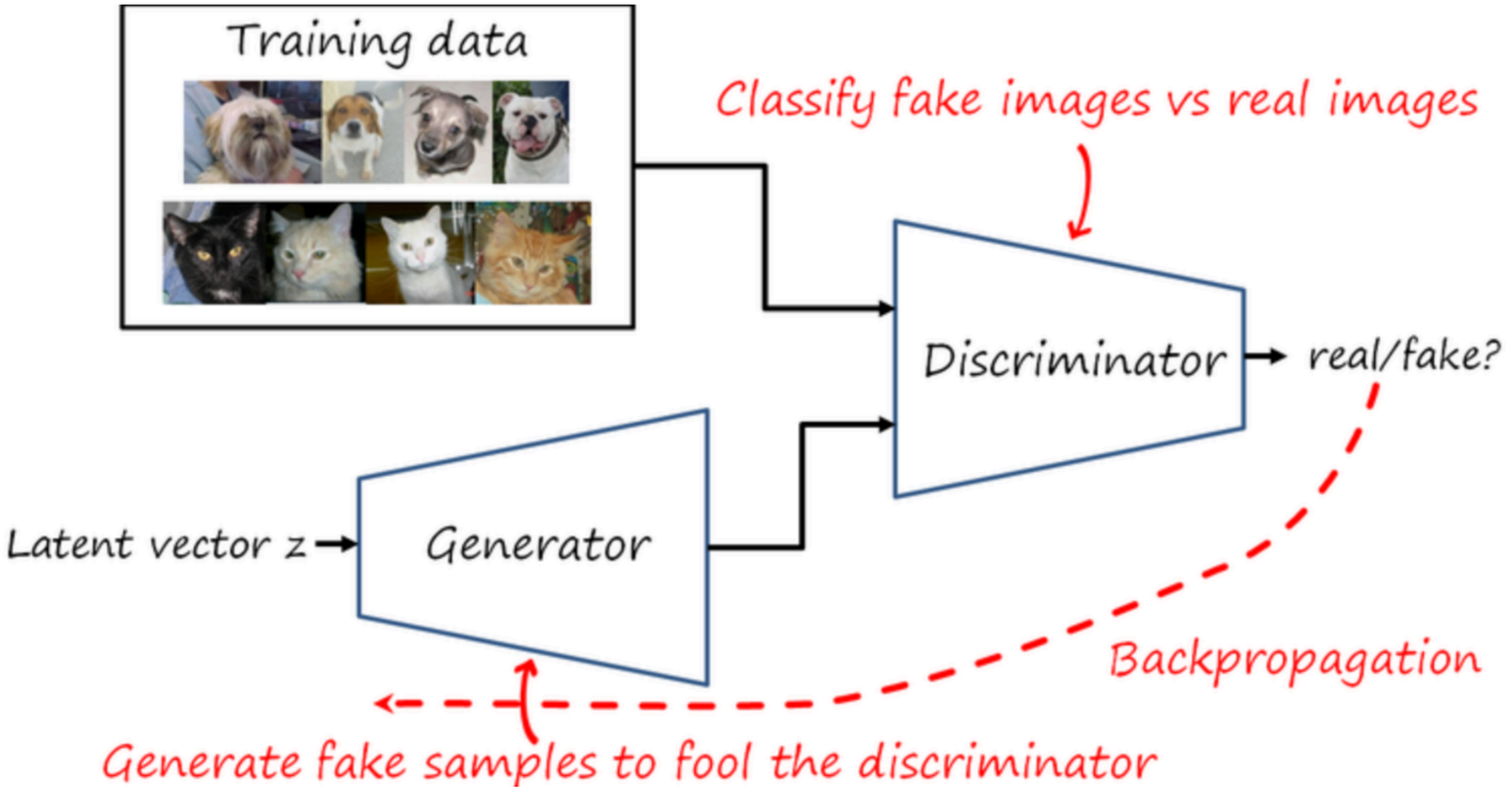


Image generated by GAN, from thispersondoesnotexist.com

Generative Adversarial Network (GAN)



GAN is made of two parts:

- *Generator network*—Takes as input a random vector (a random point in the latent space), and decodes it into a synthetic image
- *Discriminator network (or adversary)*—Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network

Generative Adversarial Network (GAN)

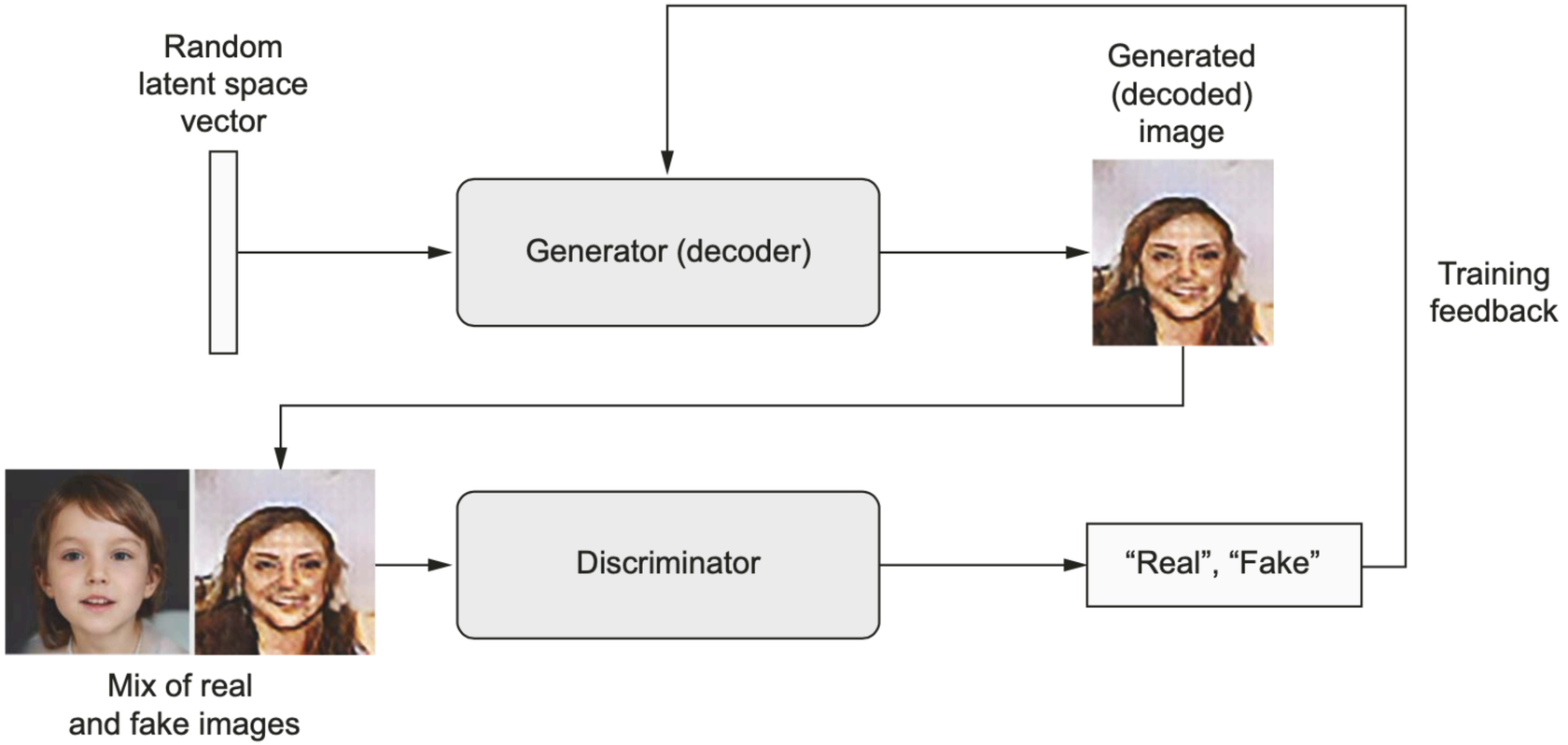
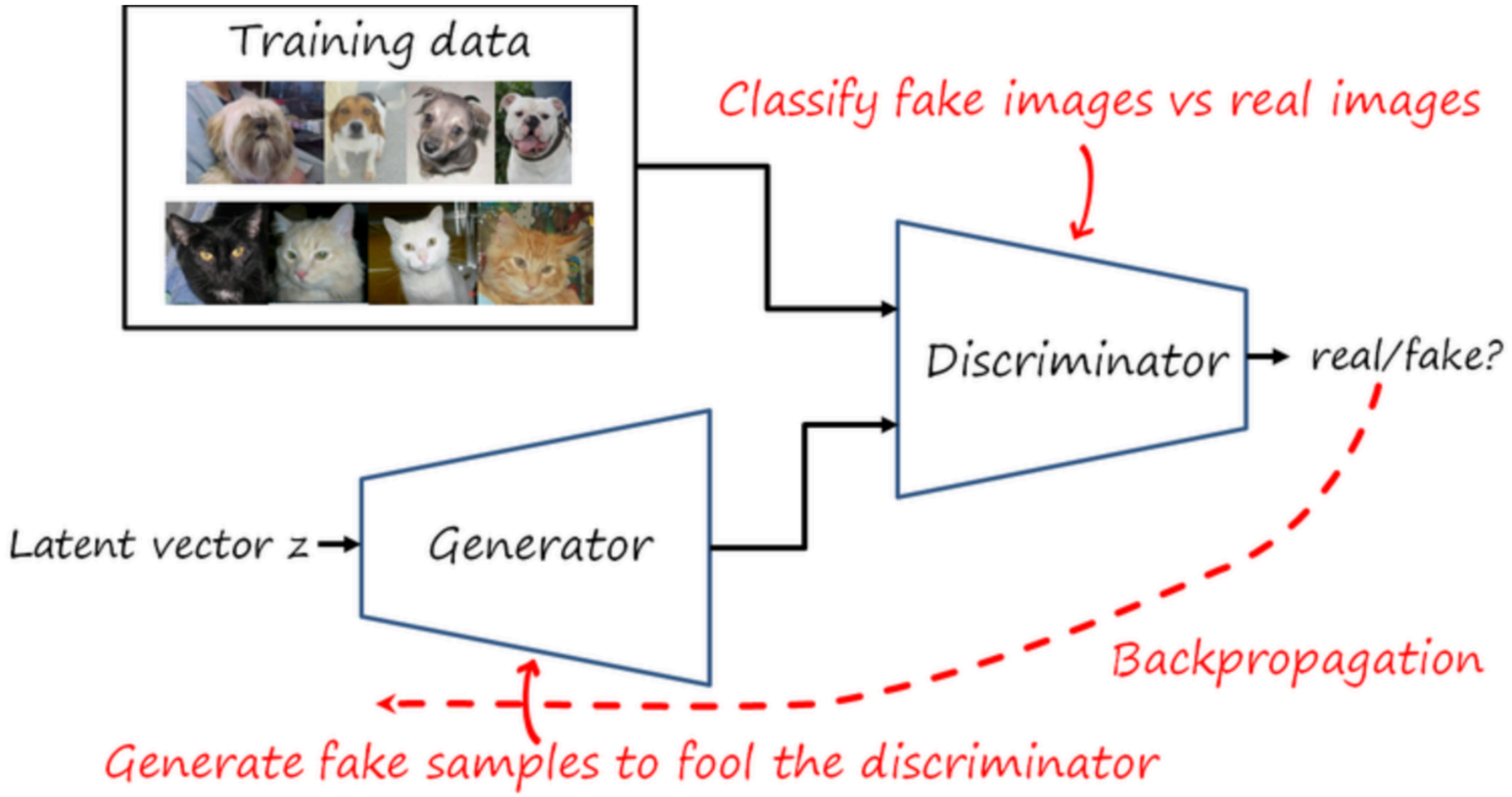


Figure 12.19 A generator transforms random latent vectors into images, and a discriminator seeks to tell real images from generated ones. The generator is trained to fool the discriminator.

Generative Adversarial Network (GAN)



Iterate: (1) Fix generator, train discriminator. (Discriminator gets better.)
(2) Fix discriminator, train generator. (Generator gets better.)

Generative Adversarial Network (GAN)

http://www.lherranz.org/2018/08/07/imagetranslation/

Generative Adversarial Network (GAN)

Figure 12.20 Latent space dwellers. Images generated by https://thispersondoesnotexist.com using a StyleGAN2 model. (Image credit: Phillip Wang is the website author. The model used is the StyleGAN2 model from Karras et al., https://arxiv.org/abs/1912.04958.)

We will use DCGAN (deep convolutional GAN).

Generative Adversarial Network (GAN)

We'll train our GAN on images from the Large-scale CelebFaces Attributes dataset (known as CelebA), a dataset of 200,000 faces of celebrities (http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html) To speed up training, we'll resize the images to 64 × 64, so we'll be learning to generate 64 × 64 images of human faces.

Generative Adversarial Network (GAN)

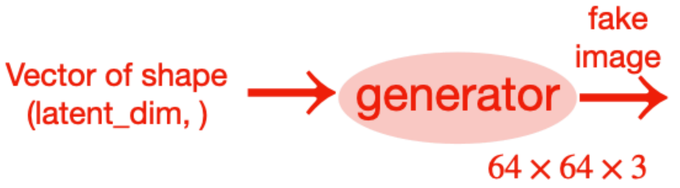
Schematically, the GAN looks like this:

- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(64, 64, 3)`.
- A `discriminator` network maps images of shape `(64, 64, 3)` to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus, this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as we train any regular image-classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that at every step, we move the weights of the generator in a direction that makes the discriminator more likely to classify as "real" the images decoded by the generator. In other words, we train the generator to fool the discriminator.

Generative Adversarial Network (GAN)

Schematically, the GAN looks like this:

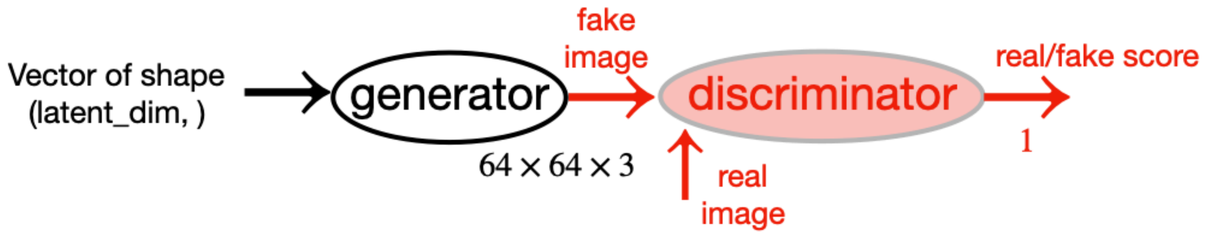
- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(64, 64, 3)`.
- A `discriminator` network maps images of shape `(64, 64, 3)` to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus, this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as we train any regular image-classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that at every step, we move the weights of the generator in a direction that makes the discriminator more likely to classify as "real" the images decoded by the generator. In other words, we train the generator to fool the discriminator.



Generative Adversarial Network (GAN)

Schematically, the GAN looks like this:

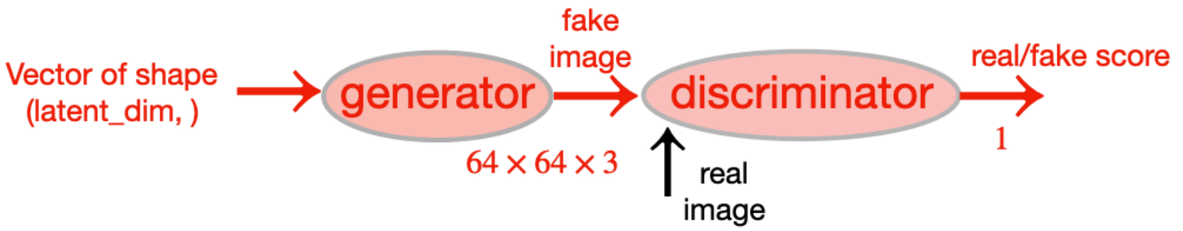
- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(64, 64, 3)`.
- A `discriminator` network maps images of shape `(64, 64, 3)` to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus, this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as we train any regular image-classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that at every step, we move the weights of the generator in a direction that makes the discriminator more likely to classify as "real" the images decoded by the generator. In other words, we train the generator to fool the discriminator.



Generative Adversarial Network (GAN)

Schematically, the GAN looks like this:

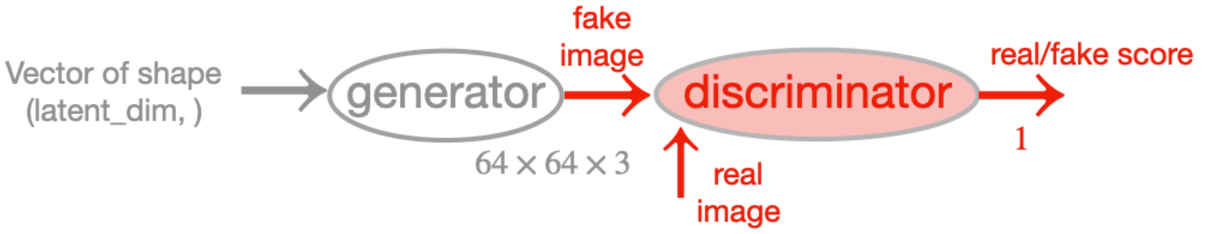
- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(64, 64, 3)`.
- A `discriminator` network maps images of shape `(64, 64, 3)` to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus, this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as we train any regular image-classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that at every step, we move the weights of the generator in a direction that makes the discriminator more likely to classify as "real" the images decoded by the generator. In other words, we train the generator to fool the discriminator.



Generative Adversarial Network (GAN)

Schematically, the GAN looks like this:

- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(64, 64, 3)`.
- A `discriminator` network maps images of shape `(64, 64, 3)` to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus, this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as we train any regular image-classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that at every step, we move the weights of the generator in a direction that makes the discriminator more likely to classify as "real" the images decoded by the generator. In other words, we train the generator to fool the discriminator.



Generative Adversarial Network (GAN)

Schematically, the GAN looks like this:

- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(64, 64, 3)`.
- A `discriminator` network maps images of shape `(64, 64, 3)` to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus, this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as we train any regular image-classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that at every step, we move the weights of the generator in a direction that makes the discriminator more likely to classify as "real" the images decoded by the generator. In other words, we train the generator to fool the discriminator.

A bag of tricks for GAN

The process of training GANs and tuning GAN implementations is notoriously difficult. There are a number of known tricks you should keep in mind.

Here are a few of the tricks used in the implementation of the GAN generator and discriminator in this section. It isn't an exhaustive list of GAN-related tips; you'll find many more across the GAN literature:

- We use strides instead of pooling for downsampling feature maps in the discriminator, just like we did in our VAE encoder.
- We sample points from the latent space using a *normal distribution* (Gaussian distribution), not a uniform distribution.
- Stochasticity is good for inducing robustness. Because GAN training results in a dynamic equilibrium, GANs are likely to get stuck in all sorts of ways. Introducing randomness during training helps prevent this. We introduce randomness by adding random noise to the labels for the discriminator.

A bag of tricks for GAN

- Sparse gradients can hinder GAN training. In deep learning, sparsity is often a desirable property, but not in GANs. Two things can induce gradient sparsity: max pooling operations and `relu` activations. Instead of max pooling, we recommend using strided convolutions for downsampling, and we recommend using a `LeakyReLU` layer instead of a `relu` activation. It's similar to `relu`, but it relaxes sparsity constraints by allowing small negative activation values.
- In generated images, it's common to see checkerboard artifacts caused by unequal coverage of the pixel space in the generator (see figure 12.21). To fix this, we use a kernel size that's divisible by the stride size whenever we use a strided `Conv2DTranspose` or `Conv2D` in both the generator and the discriminator.

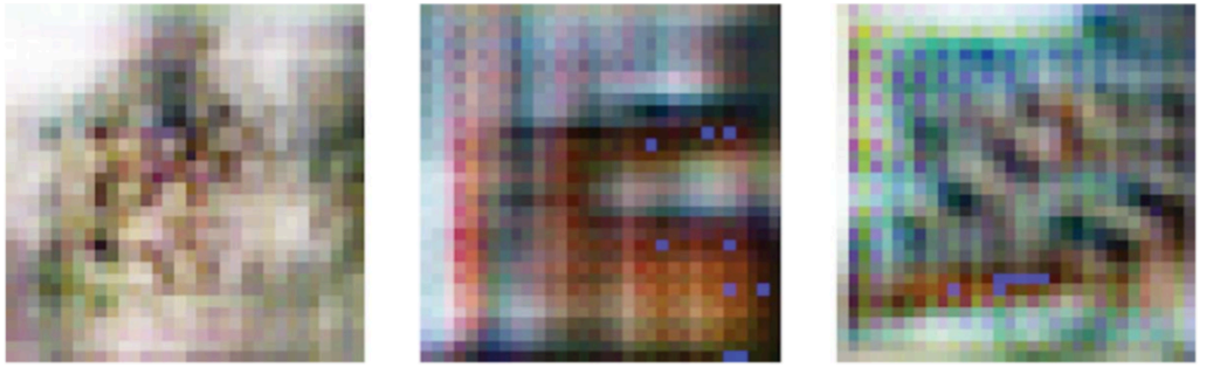


Figure 12.21 Checkerboard artifacts caused by mismatching strides and kernel sizes, resulting in unequal pixel-space coverage: one of the many gotchas of GANs

Get CelebA dataset

You can download the dataset manually from the website: http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html. If you're using Colab, you can run the following to download the data from Google Drive and uncompress it.

Listing 12.30 Getting the CelebA data

```
!mkdir celeba_gan
!gdown --id 1O7m1010EJjLE5QxLZiM9Fpjs7Oj6e684 -O celeba_gan/data.zip
!unzip -qq celeba_gan/data.zip -d celeba_gan
```

Create a working directory.

Download the compressed data using gdown (available by default in Colab; install it otherwise).

Uncompress the data.

Get CelebA dataset

Once you've got the uncompressed images in a directory, you can use `image_dataset_from_directory` to turn it into a dataset. Since we just need the images—there are no labels—we'll specify `label_mode=None`.

Listing 12.31 Creating a dataset from a directory of images

```
from tensorflow import keras
dataset = keras.utils_dataset_from_directory(
    "celeba_gan",
    label_mode=None,
    image_size=(64, 64),
    batch_size=32,
    smart_resize=True)
```

Only the images will be returned—no labels.

We will resize the images to 64 × 64 by using a smart combination of cropping and resizing to preserve aspect ratio. We don't want face proportions to get distorted!

Get CelebA dataset

Finally, let's rescale the images to the `[0-1]` range.

Listing 12.32 Rescaling the images

```
dataset = dataset.map(lambda x: x / 255.)
```

You can use the following code to display a sample image.

Listing 12.33 Displaying the first image

```
import matplotlib.pyplot as plt
for x in dataset:
    plt.axis("off")
    plt.imshow((x.numpy() * 255).astype("int32")[0])
    break
```

Discriminator of GAN

Listing 12.34 The GAN discriminator network

```
from tensorflow.keras import layers

discriminator = keras.Sequential(
    [
        keras.Input(shape=(64, 64, 3)),
        layers.Conv2D(64, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Flatten(),
        layers.Dropout(0.2),
        layers.Dense(1, activation="sigmoid"),
    ],
    name="discriminator",
)
```

One dropout layer: an important trick!

Generator of GAN

Listing 12.35 GAN generator network

```
latent_dim = 128

generator = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        layers.Dense(8 * 8 * 128),
        layers.Reshape((8, 8, 128)),
        layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(3, kernel_size=5, padding="same", activation="sigmoid"),
    ],
    name="generator",
)
```

The latent space will be made of 128-dimensional vectors.

Produce the same number of coefficients we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

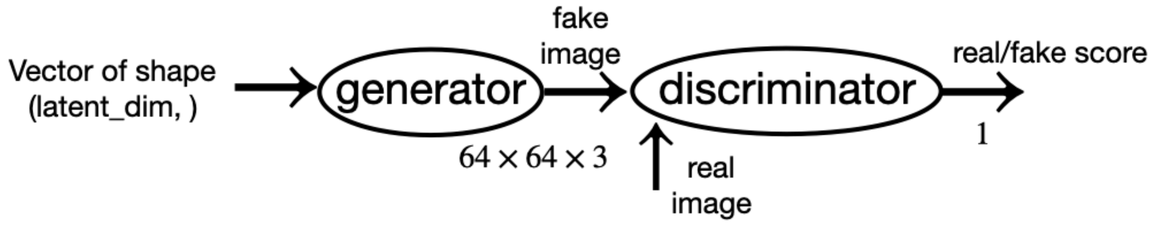
The output ends up with shape (28, 28, 1).

We use LeakyReLU as our activation.

GAN

To recapitulate, this is what the training loop looks like schematically. For each epoch, you do the following:

1. Draw random points in the latent space (random noise).
2. Generate images with `generator` using this random noise.
3. Mix the generated images with real ones.
4. Train `discriminator` using these mixed images, with corresponding targets: either "real" (for the real images) or "fake" (for the generated images).
5. Draw new random points in the latent space.
6. Train `generator` using these random vectors, with targets that all say "these are real images." This updates the weights of the generator to move them toward getting the discriminator to predict "these are real images" for generated images: this trains the generator to fool the discriminator.



GAN

Let's implement it. Like in our VAE example, we'll use a `Model` subclass with a custom `train_step()`. Note that we'll use two optimizers (one for the generator and one for the discriminator), so we will also override `compile()` to allow for passing two optimizers.

Listing 12.36 The GAN `Model`

```
import tensorflow as tf
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.d_loss_metric = keras.metrics.Mean(name="d_loss")
        self.g_loss_metric = keras.metrics.Mean(name="g_loss")

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]
```

Sets up metrics to track the two losses over each training epoch

```
    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim))
        generated_images = self.generator(random_latent_vectors)
        combined_images = tf.concat([generated_images, real_images], axis=0)
        labels = tf.concat(
            [tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))],
            axis=0
        )
        labels += 0.05 * tf.random.uniform(tf.shape(labels))

        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim))

        misleading_labels = tf.zeros((batch_size, 1))

        with tf.GradientTape() as tape:
            predictions = self.discriminator(
                self.generator(random_latent_vectors))
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(
            zip(grads, self.generator.trainable_weights))

        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)
        return {"d_loss": self.d_loss_metric.result(),
                "g_loss": self.g_loss_metric.result()}
```

Decodes them to fake images

Combines them with real images

Samples random points in the latent space

Assembles labels, discriminating real from fake images

Adds random noise to the labels—an important trick!

Trains the discriminator

Samples random points in the latent space

Assembles labels that say "these are all real images" (it's a lie!)

Trains the generator

GAN

Before we start training, let's also set up a callback to monitor our results: it will use the generator to create and save a number of fake images at the end of each epoch.

Listing 12.37 A callback that samples generated images during training

```
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=3, latent_dim=128):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.normal(
            shape=(self.num_img, self.latent_dim))
        generated_images = self.model.generator(random_latent_vectors)
        generated_images *= 255
        generated_images.numpy()

        for i in range(self.num_img):
            img = keras.utils.array_to_img(generated_images[i])
            img.save(f"generated_img_{epoch:03d}_{i}.png")
```

GAN

Finally, we can start training.

Listing 12.38 Compiling and training the GAN

```
epochs = 100

gan = GAN(discriminator=discriminator, generator=generator,
          latent_dim=latent_dim)
gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    loss_fn=keras.losses.BinaryCrossentropy(),
)

gan.fit(
    dataset, epochs=epochs,
    callbacks=[GANMonitor(num_img=10, latent_dim=latent_dim)]
)
```

You'll start getting interesting results after epoch 20.

GAN

When training, you may see the adversarial loss begin to increase considerably, while the discriminative loss tends to zero—the discriminator may end up dominating the generator. If that's the case, try reducing the discriminator learning rate, and increase the dropout rate of the discriminator.

Figure 12.22 shows what our GAN is capable of generating after 30 epochs of training.

Figure 12.22 Some generated images around epoch 30

Quiz questions:

1. What is the different between the generator and the discriminator in GAN?

2. How to train the generator and the discriminator in a GAN?

Roadmap of this lecture:

1. Text generation
    - 1.1 General concepts in text generation
    - 1.2 Text generation using IMDB dataset
2. Auto-encoder and VAE (variational auto-encoder)
    - 2.1 Auto-encoder
    - 2.2 VAE
3. GAN (generative adversarial network)
4. Neural style transfer

Neural Style Transfer

Figure 12.9 A style transfer example

Neural Style Transfer

In this context, *style* essentially means textures, colors, and visual patterns in the image, at various spatial scales, and the content is the higher-level macrostructure of the image. For instance, blue-and-yellow circular brushstrokes are considered to be the style in figure 12.9 (using *Starry Night* by Vincent Van Gogh), and the buildings in the Tübingen photograph are considered to be the content.

Neural Style Transfer

The key notion behind implementing style transfer is the same idea that's central to all deep learning algorithms: you define a loss function to specify what you want to achieve, and you minimize this loss. We know what we want to achieve: conserving the content of the original image while adopting the style of the reference image. If we were able to mathematically define *content* and *style*, then an appropriate loss function to minimize would be the following:

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

Content target

Style reference

Combination image

Neural Style Transfer: the content loss

As you already know, activations from earlier layers in a network contain *local* information about the image, whereas activations from higher layers contain increasingly global, abstract information. Formulated in a different way, the activations of the different layers of a convnet provide a decomposition of the contents of an image over different spatial scales. Therefore, you'd expect the content of an image, which is more global and abstract, to be captured by the representations of the upper layers in a convnet.

A good candidate for content loss is thus the L2 norm between the activations of an upper layer in a pretrained convnet, computed over the target image, and the activations of the same layer computed over the generated image. This guarantees that, as seen from the upper layer, the generated image will look similar to the original target image. Assuming that what the upper layers of a convnet see is really the content of their input images, this works as a way to preserve image content.

Neural Style Transfer: the style loss

The content loss only uses a single upper layer, but the style loss as defined by Gatys et al. uses multiple layers of a convnet: you try to capture the appearance of the style-reference image at all spatial scales extracted by the convnet, not just a single scale. For the style loss, Gatys et al. use the *Gram matrix* of a layer's activations: the inner product of the feature maps of a given layer. This inner product can be understood as representing a map of the correlations between the layer's features. These feature correlations capture the statistics of the patterns of a particular spatial scale, which empirically correspond to the appearance of the textures found at this scale.

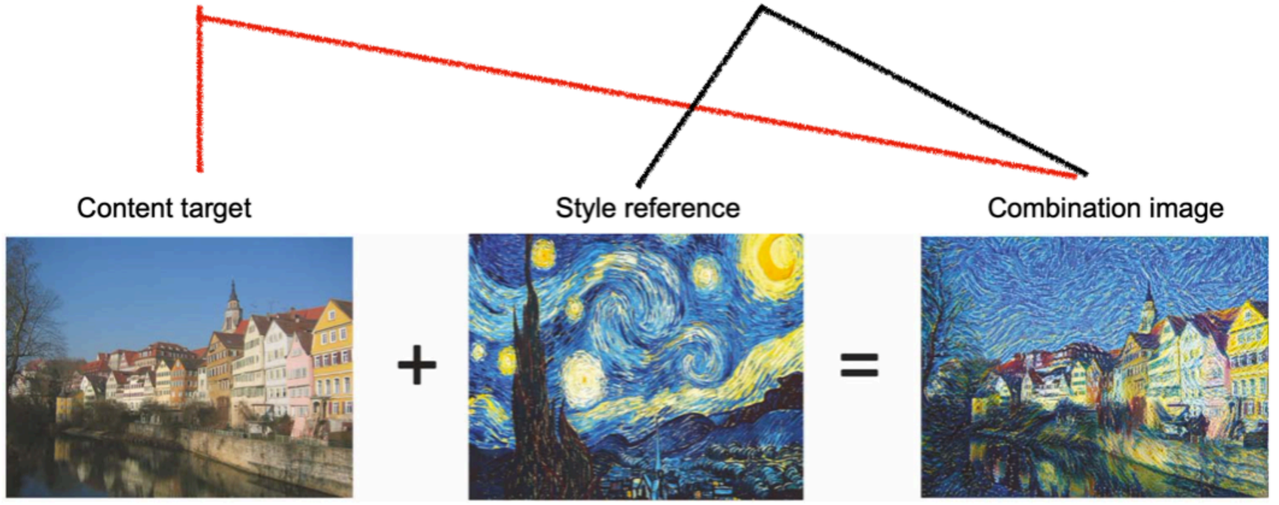
Neural Style Transfer

In short, you can use a pretrained convnet to define a loss that will do the following:

- Preserve content by maintaining similar high-level layer activations between the original image and the generated image. The convnet should "see" both the original image and the generated image as containing the same things.
- Preserve style by maintaining similar *correlations* within activations for both low-level layers and high-level layers. Feature correlations capture *textures*: the generated image and the style-reference image should share the same textures at different spatial scales.

Content loss
A Higher layer
L2 norm

Style loss
Low and high layers
Gram matrix (statistical correlation)



Neural Style Transfer

Use a pre-trained CNN.
We use: VGG19

Here's the general process:

- Set up a network that computes VGG19 layer activations for the style-reference image, the base image, and the generated image at the same time.
- Use the layer activations computed over these three images to define the loss function described earlier, which we'll minimize in order to achieve style transfer.
- Set up a gradient-descent process to minimize this loss function.

Neural Style Transfer: get two images (content and style)

Let's start by defining the paths to the style-reference image and the base image. To make sure that the processed images are a similar size (widely different sizes make style transfer more difficult), we'll later resize them all to a shared height of 400 px.

Listing 12.16 Getting the style and content images

```
from tensorflow import keras

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

Path to the image we want to transform

Path to the style image

Dimensions of the generated picture

Neural Style Transfer: get two images (content and style)

content

style

Neural Style Transfer: get two images (content and style)

We also need some auxiliary functions for loading, preprocessing, and postprocessing the images that go in and out of the VGG19 convnet.

Listing 12.17 Auxiliary functions

```
import numpy as np

def preprocess_image(image_path):
    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

def deprocess_image(img):
    img = img.reshape((img_height, img_width, 3))
    img[:, :, 0] += 103.939
    img[:, :, 1] += 116.779
    img[:, :, 2] += 123.68
    img = img[:, :, ::-1]
    img = np.clip(img, 0, 255).astype("uint8")
    return img
```

Util function to open, resize, and format pictures into appropriate arrays

Util function to convert a NumPy array into a valid image

Zero-centering by removing the mean pixel value from ImageNet. This reverses a transformation done by vgg19.preprocess_input.

Converts images from 'BGR' to 'RGB'. This is also part of the reversal of vgg19.preprocess_input.

Neural Style Transfer: build model

Let's set up the VGG19 network. Like in the DeepDream example, we'll use the pretrained convnet to create a feature exactor model that returns the activations of intermediate layers—all layers in the model this time.

Listing 12.18 Using a pretrained VGG19 model to create a feature extractor

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

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

Build a VGG19 model loaded with pretrained ImageNet weights.

Model that returns the activation values for every target layer (as a dict)

Neural Style Transfer: content loss

Let's define the content loss, which will make sure the top layer of the VGG19 convnet has a similar view of the style image and the combination image.

Listing 12.19 Content loss

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

L2 norm

Neural Style Transfer: style loss

Next is the style loss. It uses an auxiliary function to compute the Gram matrix of an input matrix: a map of the correlations found in the original feature matrix.

Listing 12.20 Style loss

```
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))
```

Neural Style Transfer: total variational loss

To these two loss components, you add a third: the *total variation loss*, which operates on the pixels of the generated combination image. It encourages spatial continuity in the generated image, thus avoiding overly pixelated results. You can interpret it as a regularization loss.

Listing 12.21 Total variation loss

```
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))
```

Neural Style Transfer: total variational loss

The loss that you minimize is a weighted average of these three losses. To compute the content loss, you use only one upper layer—the `block5_conv2` layer—whereas for the style loss, you use a list of layers that spans both low-level and high-level layers. You add the total variation loss at the end.

Depending on the style-reference image and content image you're using, you'll likely want to tune the `content_weight` coefficient (the contribution of the content loss to the total loss). A higher `content_weight` means the target content will be more recognizable in the generated image.

Listing 12.22 Defining the final loss that you'll minimize

```
style_layer_names = [
    "block1_conv1",
    "block2_conv1",
    "block3_conv1",
    "block4_conv1",
    "block5_conv1",
]
content_layer_name = "block5_conv2"
total_variation_weight = 1e-6
```

List of layers to use for the style loss

The layer to use for the content loss

Contribution weight of the total variation loss

Neural Style Transfer: total variational loss

```
style_weight = 1e-6
content_weight = 2.5e-8

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=())
    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
    )
    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
```

Contribution weight of the content loss

Contribution weight of the style loss

Initialize the loss to 0.

Add the content loss.

Add the style loss.

Add the total variation loss.

Neural Style Transfer: compile the model using SGD

Listing 12.23 Setting up the gradient-descent process

```
import tensorflow as tf

@tf.function
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

optimizer = keras.optimizers.SGD(
    keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=100.0, decay_steps=100, decay_rate=0.96
    )
)
```

We make the training step fast by compiling it as a tf.function.

We'll start with a learning rate of 100 and decrease it by 4% every 100 steps.

Neural Style Transfer: generate the combined image

```
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"Iteration {i}: loss={loss:.2f}")
        img = deprocess_image(combination_image.numpy())
        fname = f"combination_image_at_iteration_{i}.png"
        keras.utils.save_img(fname, img)
```

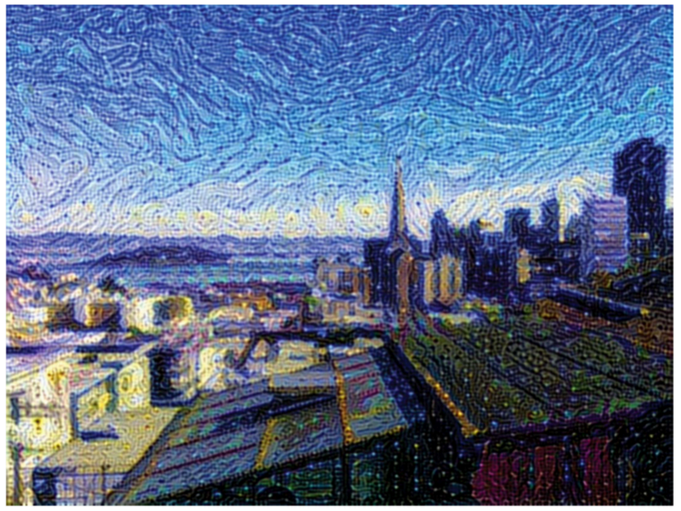
Use a Variable to store the combination image since we'll be updating it during training.

Update the combination image in a direction that reduces the style transfer loss.

Save the combination image at regular intervals.

Neural Style Transfer: generate the combined image





Quiz questions:

1. What is the difference between “content” and “style”?

2. How to define loss functions for neural style transfer?