Chapter 12 Autoencoder

> Suppose machine learning is a cake, reinforcement learning is the cherry on the cake, supervised learning is the icing on the outside, and unsupervised learning is the cake itself. —Yann LeCun

Earlier we introduced the neural network learning algorithm given the sample and its corresponding label. This type of algorithm actually learns the conditional probability $P(y|\boldsymbol{x})$ given the sample $\boldsymbol{x}$. With the booming social network today, it is relatively easy to obtain massive sample data $\boldsymbol{x}$, such as photos, voices, texts, etc., but the difficulty is to obtain the label information corresponding to these data. For example, in addition to collecting source language text, the target language text data to be translated is also required for machine translation. Data labeling is mainly based on human prior knowledge. For example, Amazon's Mechanical Turk system is responsible for data labeling, recruiting part-time staff from all over the world to complete customer data labeling tasks. The scale of data required for deep learning is generally very large. This method of relying heavily on manual data annotation is expensive and inevitably introduces the subjective prior bias of the annotator.

For massive unlabeled data, is there a way to learn the data distribution $P(\boldsymbol{x})$ from it? This is the Unsupervised Learning algorithm that we will introduce in this chapter. In particular, if the algorithm learns $\boldsymbol{x}$ as a supervised signal, this type of algorithm is called Self-supervised Learning, and the autoencoder algorithm introduced in this chapter is one type of self-supervised learning algorithms.

12.1 Principle of Autoencoder

Let us consider the function of neural networks in supervised learning:

$$\boldsymbol{o} = f_\theta(\boldsymbol{x}), \boldsymbol{x} \in R^{d_{\text{in}}}, \boldsymbol{o} \in R^{d_{\text{out}}}$$

d_{in} is the length of the input feature vector, and d_{out} is the length of the network output vector. For classification problems, the network model transforms the input feature vector $\boldsymbol{x}$ of length d_{in} to the output vector $\boldsymbol{o}$ of length d_{out}. This process can be considered as a feature reduction process, transforming the original high-dimensional input vector $\boldsymbol{x}$ to a low-dimensional variable $\boldsymbol{o}$. Dimensionality Reduction has a wide range of applications in machine learning, such as file compression and data preprocessing. The most common dimension reduction algorithm is Principal Components Analysis (PCA), which obtains the main components of the data by eigen-decomposing the covariance matrix, but PCA is essentially a linear transformation, and the ability to extract features is limited.

So can we use the powerful nonlinear expression capabilities of neural networks to learn low-dimensional data representation? The key to the problem is that training neural networks generally requires an explicit label data (or supervised signal), but unsupervised data has no additional labeling information, only the data $\boldsymbol{x}$ itself.

Therefore, we try to use the data $\boldsymbol{x}$ itself as a supervision signal to guide the training of the network, that is, we hope that the neural network can learn the mapping $f_\theta: \boldsymbol{x} \to \boldsymbol{x}$. We divide the network f_θ into two parts. The first sub-network tries to learn the mapping relationship: $g_{\theta_1}: \boldsymbol{x} \to \boldsymbol{z}$, and the latter sub-network tries to learn the mapping relationship $h_{\theta_2}: \boldsymbol{z} \to \boldsymbol{x}$, as shown in Figure 12.1. We consider g_{θ_1} as a process of data encoding which encodes the high-dimensional input $\boldsymbol{x}$ into a low-dimensional hidden variable $\boldsymbol{z}$ (Latent Variable, or Hidden Variable), which is called a Encoder network. h_{θ_2} is considered as the process of data decoding, which decodes the encoded input $\boldsymbol{z}$ into high-dimensional $\boldsymbol{x}$, which is called a Decoder network.

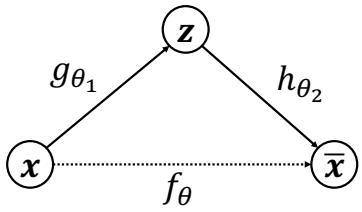

Figure 0.1 Autoencoder model

The encoder and decoder jointly complete the encoding and decoding process of the input data $\boldsymbol{x}$. We call the entire network model f_θ an Auto-Encoder for short. If a deep neural network is used to parameterize the g_{θ_1} and h_{θ_2} functions, it is called Deep Auto-encoder, as shown in Figure 12.2.

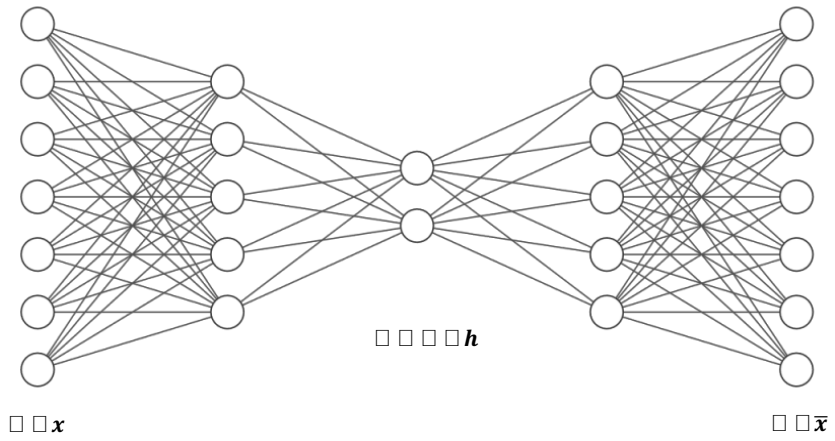

Figure 0.2 Autoencoder using neural network parameterization

The self-encoder can transform the input to the hidden vector $\boldsymbol{z}$, and reconstruct $\bar{\boldsymbol{x}}$ through the decoder. We hope that the output of the decoder can perfectly or approximately recover the original input, that is $\bar{\boldsymbol{x}} \approx \boldsymbol{x}$, then the optimization goal of the autoencoder can be written as:

$$\min \mathcal{L} = \text{dist}(\boldsymbol{x}, \bar{\boldsymbol{x}})$$

$$\bar{\boldsymbol{x}} = h_{\theta_2}(g_{\theta_1}(\boldsymbol{x}))$$

where $\text{dist}(\boldsymbol{x}, \bar{\boldsymbol{x}})$ represents the distance measurement between $\boldsymbol{x}$ and $\bar{\boldsymbol{x}}$, which is called the reconstruction error function. The most common measurement method is the square of the Euclidean distance. The calculation method is as follows:

$$\mathcal{L} = \sum_i (x_i - \bar{x}_i)^2$$

It is equivalent in principle to the mean square error. There is no essential difference between the autoencoder network and the ordinary neural network, except that the trained supervision signal has changed from the label $\boldsymbol{y}$ to its own $\boldsymbol{x}$. With the help of the non-linear feature extraction capability of deep neural networks, the autoencoder can obtain good data representation. Compared with linear methods such as PCA, the autoencoder has better performance and can even recover the input $\boldsymbol{x}$ more perfectly.

In Figure 12.3(a), the first row is a real MNIST handwritten digit picture randomly sampled from the test set, and the second, third, and fourth rows are reconstructed using a hidden vector of length 30, using Autoencoder, Logistic PCA and standard PCA, respectively. In Figure 12.3(b), the first row is a real portrait image, and the second and third rows are based on a hidden vector of length 30, which is recovered using the Autoencoder and the standard PCA algorithm. It can be seen that the image reconstructed by the Autoencoder is relatively clear and has a high degree of restoration, while the image reconstructed by the PCA algorithm is blurry.

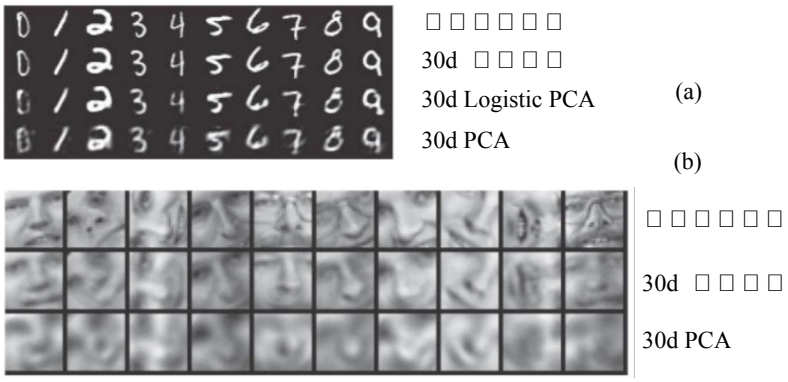

Figure 0.3 Autoencoder vs. PCA [1]

12.2 Hands-on Fashion MNIST image reconstruction

The principle of the autoencoder algorithm is very simple, easy to implement, and stable in training. Compared with the PCA algorithm, the powerful expression ability of the neural network can learn the high-level abstract hidden feature vector $\boldsymbol{z}$ of the input, and it can also reconstruct the input based on $\boldsymbol{z}$. Here we perform actual picture reconstruction based on the Fashsion MNIST dataset.

12.2.1 Fashion MNIST Data Set

Fashion MNIST is a data set that is a slightly more complicated problem than MNIST image recognition. Its settings are almost the same as MNIST. It contains 10 types of grayscale images of different types of clothes, shoes, bags, and the size of the image is 28×28, a total of 70,000 pictures, of which 60,000 are used for the training set and 10,000 are used for the test set, as shown in Figure 12.4. Each row is a category of pictures. As you can see, Fashion MNIST has the same settings except that the picture content is different from MNIST. In most cases, the original algorithm code based on MNIST can be directly replaced without additional modification. Since Fashion MNIST image recognition is more difficult than MNIST, it can be used to test the performance of a slightly more complex algorithm.

Figure 0.4 Fashion MNIST Data Set

In TensorFlow, it is also very convenient to load the Fashion MNIST dataset, which can be downloaded, managed and loaded online using the keras.datasets.fashion_mnist.load_data() function as below:

```
# Load Fashion MNIST data set
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_dat
a()
# Normalize
x_train, x_test = x_train.astype(np.float32) / 255., x_test.astype(np.float3
2) / 255.
# Only need to use image data to build data set objects, no tags required
train_db = tf.data.Dataset.from_tensor_slices(x_train)
train_db = train_db.shuffle(batchsz * 5).batch(batchsz)
#  Build test set objects
test_db = tf.data.Dataset.from_tensor_slices(x_test)
test_db = test_db.batch(batchsz)
```

12.2.2 Encoder

We use the encoder to reduce the dimensionality of the input picture $\boldsymbol{x} \in R^{784}$ to a lower-dimensional hidden vector: $\boldsymbol{h} \in R^{20}$, and use the decoder to reconstruct the picture based on the hidden vector $\boldsymbol{h}$. The autoencoder model is shown in Figure 12.5. The decoder is composed of a 3-layer fully connected network with output nodes of 256, 128, and 20 respectively. The decoder is also composed of a 3-layer fully connected network with output nodes of 128, 256, and 784 respectively.

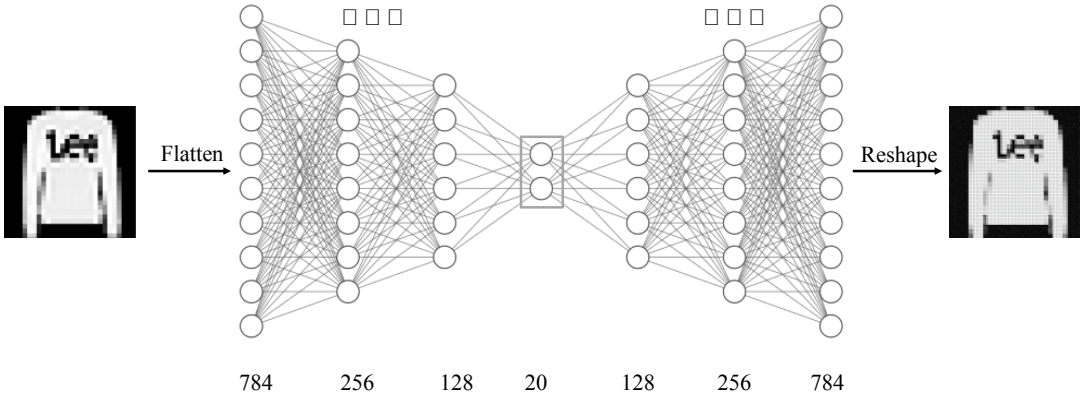

Figure 0.5 Fashion MNIST autoencoder network architecture

The first is the realization of the encoder sub-network. A 3-layer neural network is used to reduce the dimensionality of the image vector from 784 to 256, 128, and finally to h_dim. Each layer uses the ReLU activation function, and the last layer does not use any activation function.

```
        # Create Encoders network, implemented in the initialization
function of the autoencoder class
        self.encoder = Sequential([
            layers.Dense(256, activation=tf.nn.relu),
            layers.Dense(128, activation=tf.nn.relu),
            layers.Dense(h_dim)
        ])
```

12.2.3 Decoder

Let's create the decoder sub-network. Here, the hidden vector h_dim is upgraded to the length of 128, 256, and 784 in turn. Except for the last layer, the ReLU activation function are used. The output of the decoder is a vector of length 784, which represents a 28×28 size picture after being flattened, and can be restored to a picture matrix through the reshape operation as below:

```
        # Create Decoders network
        self.decoder = Sequential([
            layers.Dense(128, activation=tf.nn.relu),
            layers.Dense(256, activation=tf.nn.relu),
            layers.Dense(784)
        ])
```

12.2.4 Autoencoder

The above two sub-networks of encoder and decoder are implemented in the auto-encoder class AE, and we create these two sub-networks in the initialization function at the same time.

```
class AE(keras.Model):
    # Self-encoder model class, including Encoder and Decoder 2 subnets
    def __init__(self):
        super(AE, self).__init__()
```

```
        #  Create Encoders network
        self.encoder = Sequential([
            layers.Dense(256, activation=tf.nn.relu),
            layers.Dense(128, activation=tf.nn.relu),
            layers.Dense(h_dim)
        ])
        #  Create Decoders network
        self.decoder = Sequential([
            layers.Dense(128, activation=tf.nn.relu),
            layers.Dense(256, activation=tf.nn.relu),
            layers.Dense(784)
        ])
```

Next, the forward propagation process is implemented in the call function. The input image first obtains the hidden vector h through the encoder sub-network, and then obtains the reconstructed image through the decoder. Just call the forward propagation function of the encoder and decoder in turn as follows:

```
    def call(self, inputs, training=None):
        # Forward propagation function
        # Encoding to obtain hidden vector h,[b, 784] => [b, 20]
        h = self.encoder(inputs)
        # Decode to get reconstructed picture, [b, 20] => [b, 784]
        x_hat = self.decoder(h)

        return x_hat
```

12.2.5 Network training

The training process of the autoencoder is basically the same as that of a classifier. The distance between the reconstructed vector $\bar{\boldsymbol{x}}$ and the original input vector $\boldsymbol{x}$ is calculated through the error function, and then the gradients of the encoder and decoder are simultaneously calculated using the automatic derivation mechanism of TensorFlow.

First create an instance of the autoencoder and optimizer, and set an appropriate learning rate. E.g:

```
#  Create network objects
model = AE()
#  Specify input size
model.build(input_shape=(4, 784))
#  Print network information
model.summary()
#  Create an optimizer and set the learning rate
optimizer = optimizers.Adam(lr=lr)
```

Here 100 Epochs are trained, and the reconstructed image vector is obtained through forward

calculation each time, and the tf.nn.sigmoid_cross_entropy_with_logits loss function is used to calculate the direct error between the reconstructed image and the original image. In fact, it is also feasible to use the MSE error function as below:

```
for epoch in range(100): # Train 100 Epoch
    for step, x in enumerate(train_db): #  Traverse the training set
        # Flatten, [b, 28, 28] => [b, 784]
        x = tf.reshape(x,    [-1, 784])
        # Build a gradient recorder
        with tf.GradientTape() as tape:
            # Forward calculation to obtain the reconstructed picture
            x_rec_logits = model(x)
            # Calculate the loss function between the reconstructed picture
and the input
            rec_loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=x, log
its=x_rec_logits)
            # Calculate the mean
            rec_loss = tf.reduce_mean(rec_loss)
        # Automatic derivation, including the gradient of 2 sub-networks
        grads = tape.gradient(rec_loss, model.trainable_variables)
        # Automatic update, update 2 subnets at the same time
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
        if step % 100 ==0:
            # Interval print training error
            print(epoch, step, float(rec_loss))
```

12.2.6 Image reconstruction

Different from the classification problem, the model performance of the autoencoder is generally not easy to quantify. Although the $\mathcal{L}$ value can represent the learning effect of the network to a certain extent, we ultimately hope to obtain reconstruction samples with a higher degree of reduction and richer styles. Therefore, it is generally necessary to discuss the learning effect of the autoencoder according to specific issues. For image reconstruction, it generally depends on the quality of artificial subjective evaluation of the image generation, or the use of certain image fidelity calculation methods such as Inception Score and Frechet Inception Distance.

In order to test the effect of image reconstruction, we divide the data set into a training set and a test set, where the test set does not participate in training. We randomly sample the test picture $\boldsymbol{x} \in \mathbb{D}^{\text{test}}$ from the test set, calculate the reconstructed picture through the autoencoder, and then save the real picture and the reconstructed picture as a picture array and visualize it for easy comparison as below:

```
    # Reconstruct pictures, sample a batch of pictures from the test set
    x = next(iter(test_db))
    logits = model(tf.reshape(x, [-1, 784])) # Flatten and send to
autoencoder
```

```
    x_hat = tf.sigmoid(logits) # Convert the output to pixel values, using
the sigmoid function
    # Recover to 28x28,[b, 784] => [b, 28, 28]
    x_hat = tf.reshape(x_hat, [-1, 28, 28])

    # The first 50 input + the first 50 reconstructed pictures merged,
[b, 28, 28] => [2b, 28, 28]
    x_concat = tf.concat([x[:50], x_hat[:50]], axis=0)
    x_concat = x_concat.numpy() * 255. #  Revert to 0~255 range
    x_concat = x_concat.astype(np.uint8)  #  Convert to integer
    save_images(x_concat, 'ae_images/rec_epoch_%d.png'%epoch)  # Save picture
```

The effect of image reconstruction is shown in Figure 12.6, Figure 12.7, and Figure 12.8. The 5 columns on the left of each picture are real pictures, and the 5 columns on the right are the corresponding reconstructed pictures. It can be seen that in the first Epoch, the picture reconstruction effect is poor, the picture is very blurry, and the fidelity is poor. As the training progresses, the edges of the reconstructed picture become clearer and clearer. At the 100th Epoch, the reconstructed picture effect is already closer to the real picture.

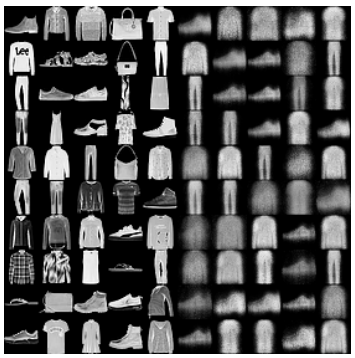

Figure 0.6 1st Epoch

Figure 0.7 10th Epoch

Figure 0.8 100th Epoch

The save_images function here is responsible for merging multiple pictures and saving them as a big picture. This is done using the PIL picture library. The code is as follows:

```
def save_images(imgs, name):
    #  Create 280x280 size image array
    new_im = Image.new('L', (280, 280))
    index = 0
    for i in range(0, 280, 28): # 10-row image array
        for j in range(0, 280, 28): # 10-column picture array
            im = imgs[index]
            im = Image.fromarray(im, mode='L')
            new_im.paste(im, (i, j)) # Write the corresponding location
            index += 1
    # Save picture array
    new_im.save(name)
```

12.3 Autoencoder variants

Generally speaking, the training of the autoencoder network is relatively stable, but because the loss function directly measures the distance between the reconstructed sample and the underlying features of the real sample, rather than evaluating abstract indicators such as the fidelity and diversity of the reconstructed sample, the effect on some tasks is mediocre, such as image reconstruction where the edges of the reconstructed image are prone to be blurred, and the fidelity is not good compared to the real image. In order to learn the true distribution of the data, a series of autoencoder variant networks were produced. Denoising Auto-Encoder

In order to prevent the neural network from memorizing the underlying features of the input data, Denoising Auto-Encoders adds random noise disturbances to the input data, such as adding noise ε sampled from the Gaussian distribution to the input $\boldsymbol{x}$:

$$\tilde{x} = x + \varepsilon, \varepsilon \sim \mathcal{N}(0, \text{var})$$

After adding noise, the network needs to learn the real hidden variable $\mathbf{z}$ of the data from $\boldsymbol{x}$, and restore the original input $\boldsymbol{x}$, as shown in Figure 12.9. The optimization goals of the model are:

$$\theta^* = \underbrace{\text{argmin}}_{\theta} \text{dist}(h_{\theta_2}(g_{\theta_1}(\tilde{\boldsymbol{x}})), \boldsymbol{x})$$

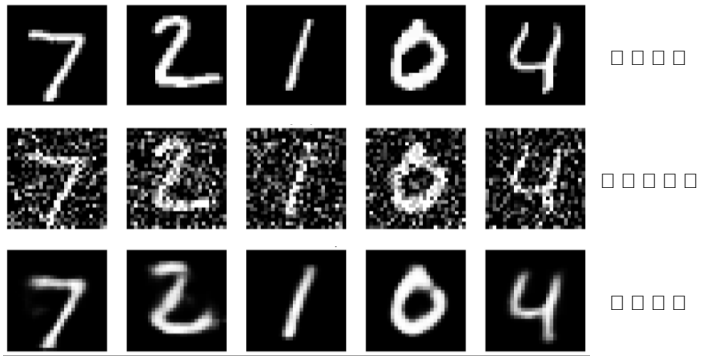

Figure 0.9 Denoising autoencoder diagram

12.3.1 Dropout Auto-Encoder

The auto-encoder network also faces the risk of over-fitting. Dropout Auto-Encoder reduces the expressive power of the network by randomly disconnecting the network and prevents over-fitting. The implementation of Dropout Auto-Encoder is very simple. Random disconnection of the network connection can be achieved by inserting the Dropout layer in the network layer.

12.3.2 Adversarial Auto-Encoder

In order to be able to conveniently sample the hidden variable $\boldsymbol{z}$ from a known prior distribution $p(\boldsymbol{z})$, it is convenient to use $p(\boldsymbol{z})$ to reconstruct the input, and the Adversarial Auto-Encoder uses an additional discriminator network (Discriminator, referred to as D network) to determine whether the hidden variable $\boldsymbol{z}$ for dimensionality reduction is sampled from the prior distribution $p(\boldsymbol{z})$, as shown in Figure 12.10. The output of the discriminator network is a variable belonging to the interval [0,1], which represents whether the hidden vector is sampled from the

prior distribution $p(\boldsymbol{z})$: all samples from the prior distribution $p(\boldsymbol{z})$ are marked as true, and those generated from the conditional probability $q(\boldsymbol{z}|\boldsymbol{x})$ are marked as false. In this way, in addition to reconstructing samples, the conditional probability distribution $q(\boldsymbol{z}|\boldsymbol{x})$ can also be constrained to approximate the prior distribution $p(\boldsymbol{z})$.

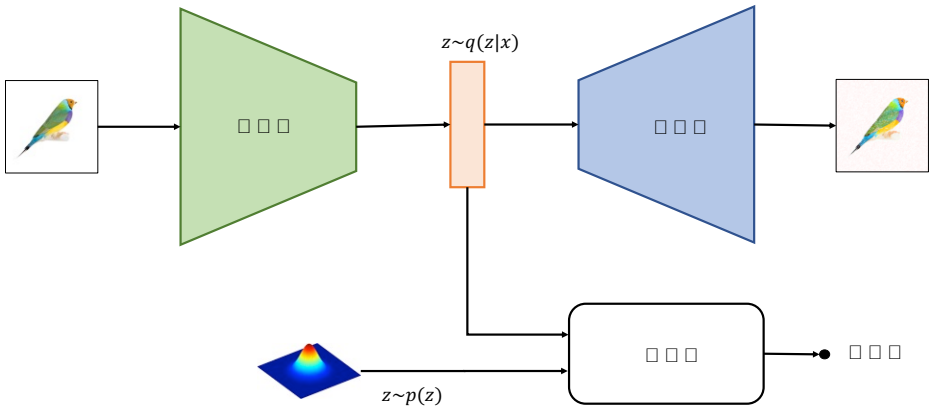

Figure 0.10 Adversarial autoencoder

The adversarial autoencoder is derived from the generative adversarial network algorithm introduced in the next chapter. After learning the adversarial generative network, you can deepen your understanding of the adversarial autoencoder.

12.4 Variational autoencoder

The basic autoencoder essentially learns the mapping relationship between the input $\boldsymbol{x}$ and the hidden variable $\boldsymbol{z}$. It is a Discriminative model, not a Generative model. So can the autoencoder be adjusted to a generative model to easily generate samples?

Given the distribution of hidden variables $\mathrm{P}(\boldsymbol{z})$, if the conditional probability distribution $\mathrm{P}(\boldsymbol{x}|\boldsymbol{z})$ can be learned, then we can sample the joint probability distribution $\mathrm{P}(\boldsymbol{x},\boldsymbol{z}) = \mathrm{P}(\boldsymbol{x}|\boldsymbol{z})\mathrm{P}(\boldsymbol{z})$ to generate different samples. Variational Auto-Encoders (VAE) can achieve this goal, as shown in Figure 12.11. If you understand it from the perspective of neural networks, VAE is the same as the previous autoencoders, which is very intuitive and easy to understand; but the theoretical derivation of VAE is a little more complicated. Next, we will first explain VAE from the perspective of neural networks, and then derive VAE from the perspective of probability.

从神经网络的角度来看，VAE 相对于自编码器模型，同样具有编码器和解码器两个子网络。解码器接受输入$\boldsymbol{x}$，输出为隐变量$\boldsymbol{z}$；解码器负责将隐变量$\boldsymbol{z}$解码为重建的$\bar{\boldsymbol{x}}$。不同的是，VAE 模型对隐变量$\boldsymbol{z}$的分布有显式地约束，希望隐变量$\boldsymbol{z}$符合预设的先验分布$\mathrm{P}(\boldsymbol{z})$。因此，在损失函数的设计上，除了原有的重建误差项外，还添加了隐变量$\boldsymbol{z}$分布的约束项。

From the point of view of neural network, VAE also has two sub-networks of encoder and decoder compared to the self-encoder model. The decoder accepts the input x, and the output is the latent variable z; the decoder is responsible for decoding the latent variable z into the reconstructed $\bar{x}$. The difference is that the VAE model has explicit constraints on the distribution

of the hidden variable z, and hopes that the hidden variable z conforms to the preset prior distribution P(z). Therefore, in the design of the loss function, in addition to the original reconstruction error term, a constraint term for the z distribution of the hidden variable is added.

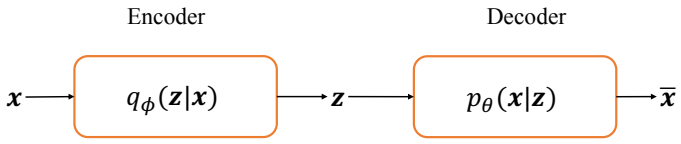

Figure 0.11 VAE model structure

12.4.1 Principle of VAE

From a probability point of view, we assume that any data set is sampled from a certain distribution $p(\boldsymbol{x}|\boldsymbol{z})$, $\boldsymbol{z}$ is a hidden variable and represents a certain internal feature, such as a picture of handwritten digits $\boldsymbol{x}$, $\boldsymbol{z}$ can represent font size, writing style, bold, Italic and other settings, which conform to a certain prior distribution $p(\boldsymbol{z})$. Given a specific hidden variable $\boldsymbol{z}$, we can sample a series of samples from the learned distribution $p(\boldsymbol{x}|\boldsymbol{z})$. These samples all have the commonality represented by $\boldsymbol{z}$.

It is usually assumed that $p(\boldsymbol{z})$ follows a known distribution, such as $\mathcal{N}(0,1)$. Under the condition that $p(\boldsymbol{z})$ is known, our goal is to learn to a generative probability model $p(\boldsymbol{x}|\boldsymbol{z})$. The Maximum Likelihood Estimation method can be used here: a good model should have a high probability of generating a real sample $\boldsymbol{x} \in \mathbb{D}$. If our generative model $p(\boldsymbol{x}|\boldsymbol{z})$ is parameterized with θ, then the optimization goal of our neural network is:

$$\max_\theta p(\boldsymbol{x}) = \int_{\boldsymbol{z}} p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$$

Unfortunately, since $\boldsymbol{z}$ is a continuous variable, the above integral cannot be converted into a discrete form, which makes it difficult to optimize directly.

Another way of thinking is using the idea of Variational Inference, we approximate $p(\boldsymbol{z}|\boldsymbol{x})$ through the distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x})$, that is, we need to minimize the distance between $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ and $p(\boldsymbol{z}|\boldsymbol{x})$:

$$\min_\phi \mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}|\boldsymbol{x})\right)$$

The KL divergence $\mathbb{D}_{KL}$ is a measure of the gap between the distribution q and p, defined as:

$$\mathbb{D}_{KL}(q||p) = \int_{\boldsymbol{x}} q(\boldsymbol{x}) \log \frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} d\boldsymbol{x}$$

Strictly speaking, the distance is generally symmetric, while the KL divergence is asymmetric. Expand the KL divergence to

$$\mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}|\boldsymbol{x})\right) = \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{q_\phi(\boldsymbol{z}|\boldsymbol{x})}{p(\boldsymbol{z}|\boldsymbol{x})} d\boldsymbol{z}$$

Use

$$p(\boldsymbol{z}|\boldsymbol{x}) \cdot p(\boldsymbol{x}) = p(\boldsymbol{x}, \boldsymbol{z})$$

Get

$$\mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}|\boldsymbol{x})\right) = \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{q_\phi(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x})}{p(\boldsymbol{x}, \boldsymbol{z})} d\boldsymbol{z}$$

$$= \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{q_\phi(\boldsymbol{z}|\boldsymbol{x})}{p(\boldsymbol{x}, \boldsymbol{z})} dz + \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log p\,(\boldsymbol{x}) d\boldsymbol{z}$$

$$= \underbrace{-\left(-\int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{q_\phi(\boldsymbol{z}|\boldsymbol{x})}{p(\boldsymbol{x}, \boldsymbol{z})} dz\right)}_{\mathcal{L}(\phi,\theta)} + \log p\,(\boldsymbol{x})$$

We define $-\int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{q_\phi(\boldsymbol{z}|\boldsymbol{x})}{p(\boldsymbol{x},\boldsymbol{z})} dz$ as $\mathcal{L}(\phi, \theta)$, so the above equation becomes

$$\mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}|\boldsymbol{x})\right) = -\mathcal{L}(\phi, \theta) + \log p\,(\boldsymbol{x})$$

where

$$\mathcal{L}(\phi, \theta) = -\int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{q_\phi(\boldsymbol{z}|\boldsymbol{x})}{p(\boldsymbol{x}, \boldsymbol{z})} dz$$

Consider

$$\mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}|\boldsymbol{x})\right) \geq 0$$

We have

$$\mathcal{L}(\phi, \theta) \leq \log p\,(\boldsymbol{x})$$

In other words, $\mathcal{L}(\phi, \theta)$ is the lower bound of $\log p\,(\boldsymbol{x})$, and the optimization objective $\mathcal{L}(\phi, \theta)$ is called Evidence Lower Bound Objective (ELBO). Our goal is to maximize the likelihood probability $p(\boldsymbol{x})$, or to maximize $\log p\,(x)$, which can be achieved by maximizing its lower bound $\mathcal{L}(\phi, \theta)$.

Now let's analyze how to maximize the $\mathcal{L}(\phi, \theta)$ function, and expand it to get:

$$\mathcal{L}(\theta, \phi) = \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{p_\theta(\boldsymbol{x}, \boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})}$$

$$= \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{p(\boldsymbol{z})p_\theta(\boldsymbol{x}|\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})}$$

$$= \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log \frac{p(\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x})} + \int_{\boldsymbol{z}} q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log p_\theta(\boldsymbol{x}|\boldsymbol{z})$$

$$= -\int_{z} q_\phi(\boldsymbol{z}|\boldsymbol{x})\, log\, \frac{q_\phi(\boldsymbol{z}|\boldsymbol{x})}{p(\boldsymbol{z})} + \mathbb{E}_{\boldsymbol{z}\sim q}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})]$$

$$= -\mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})\right) + \mathbb{E}_{\boldsymbol{z}\sim q}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})]$$

So,

$$\mathcal{L}(\theta,\phi) = -\mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})\right) + \mathbb{E}_{\boldsymbol{z}\sim q}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] \tag{0-1}$$

You can use the encoder network to parameterize the $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ function, and the decoder network to parameterize the $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ function. The target function $\mathcal{L}(\theta,\phi)$ can be optimized by calculating KL divergence between the output distribution of the decoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ and the prior distribution $p(\boldsymbol{z})$, and the likelihood probability $\log p_\theta(\boldsymbol{x}|\boldsymbol{z})$ of the decoder.

In particular, when both $q_\phi(z|x)$ and $p(z)$ are assumed to be normally distributed, the calculation of $\mathbb{D}_{KL}\left(q_\phi(z|x)||p(z)\right)$ can be simplified to:

$$\mathbb{D}_{KL}\left(q_\phi(z|x)||p(z)\right) = \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}$$

More specifically, when $q_\phi(z|x)$ is the normal distribution $\mathcal{N}(\mu_1,\sigma_1)$ and $p(z)$ is the normal distribution $\mathcal{N}(0,1)$, that is, $\mu_2 = 0, \sigma_2 = 1$, at this time

$$\mathbb{D}_{KL}\left(q_\phi(z|x)||p(z)\right) = -\log\sigma_1 + 0.5\sigma_1^2 + 0.5\mu_1^2 - 0.5 \tag{0-2}$$

The above process makes the $\mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})\right)$ term in $\mathcal{L}(\theta,\phi)$ easier to calculate, while $\mathbb{E}_{\boldsymbol{z}\sim q}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})]$ can also be implemented based on the reconstruction error function in the autoencoder.

Therefore, the optimization objective of the VAE model is transformed from maximizing the $\mathcal{L}(\phi,\theta)$ function to:

$$\min \mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})\right)$$

and

$$\max \mathbb{E}_{\boldsymbol{z}\sim q}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})]$$

The first optimization goal can be understood as constraining the distribution of latent variable $\boldsymbol{z}$, and the second optimization goal can be understood as improving the reconstruction effect of the network. It can be seen that after our derivation, the VAE model is also very intuitive and easy to understand.

12.4.2 Reparameterization Trick

Now consider a serious problem encountered in the implementation of the above-mentioned VAE model. The hidden variable z is sampled from the output $q_\phi(z|x)$ of the encoder, as shown on the left in Figure 12.12. When both $q_\phi(z|x)$and $p(z)$ are assumed to be normally distributed, the encoder outputs the mean μ and variance σ^2 of the normal distribution, and the decoder's input is sampled from $\mathcal{N}(\mu,\sigma^2)$. Due to the existence of the sampling operation, the gradient propagation is discontinuous, and the VAE network cannot be trained end-to-end through the gradient descent algorithm.

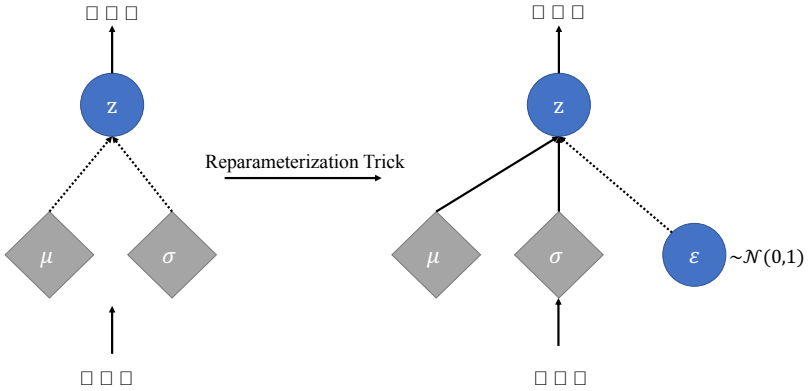

Figure 0.12 Reparameterization trick diagram

The paper [2] proposed a continuous and derivable solution called Reparameterization Trick. It samples the hidden variable z through $z = \mu + \sigma \odot \varepsilon$, where $\frac{\partial z}{\partial \mu}$ and $\frac{\partial z}{\partial \sigma}$ are both continuous and differentiable, thus connecting the gradient propagation. As shown on the right of Figure 12.12, the ε variable is sampled from the standard normal distribution $\mathcal{N}(0, I)$, and μ and σ are generated by the encoder network. The hidden variable after sampling can be obtained through $z = \mu + \sigma \odot \varepsilon$, which ensures that the gradient propagation is continuous.

The VAE network model is shown in Figure 12.13, the input $\boldsymbol{x}$ is calculated through the encoder network $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ to obtain the mean and variance of the hidden variable $\boldsymbol{z}$, and the hidden variable $\boldsymbol{z}$ is obtained by sampling through the Reparameterization Trick method, and sent to the decoder network to obtain the distribution $p_\theta(\boldsymbol{x}|\boldsymbol{z})$, and calculate the error and optimize the parameters by formula (12 1).

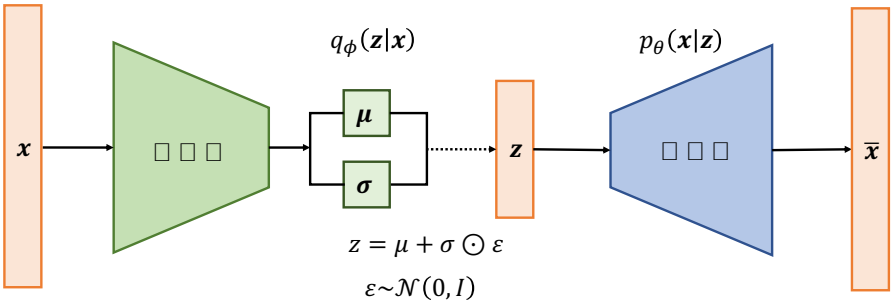

Figure 0.13 VAE model architecture

12.5 Hands-on VAE image reconstruction

In this section, we'll work on the reconstruction and generation of Fashion MNIST pictures based on the VAE model. As shown in Figure 12.13, the input is the Fashion MNIST picture vector. After 3 fully connected layers, the mean and variance of the hidden vector $\mathbf{z}$ are obtained, which are represented by 2 fully connected layers with 20 output nodes. The 20 output nodes of FC2 represent the mean vector $\boldsymbol{\mu}$ of the 20 feature distributions, and the 20 output nodes of FC3

represent the log variance vectors of the 20 feature distributions. The hidden vector $\mathbf{z}$ with a length of 20 is obtained through Reparameterization Trick sampling, and the sample picture is reconstructed through FC4 and FC5.

As a generative model, VAE can not only reconstruct the input samples, but also use the decoder alone to generate samples. The hidden vector $\mathbf{z}$ is obtained by directly sampling from the prior distribution $p(\mathbf{z})$, and the generated samples can be generated after decoding.

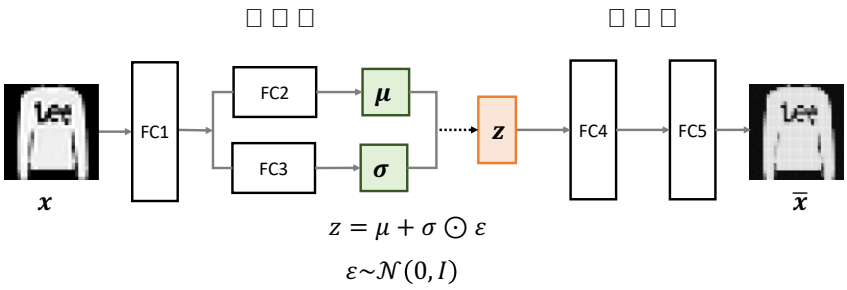

Figure 0.14 VAE model architecture

12.5.1 VAE model

We implement Encoder and Decoder sub-networks in the VAE category. In the initialization function, we create the network layers required by Encoder and Decoder respectively as below:

```
class VAE(keras.Model):
    #  Variational Encoder
    def __init__(self):
        super(VAE, self).__init__()

        # Encoder
        self.fc1 = layers.Dense(128)
        self.fc2 = layers.Dense(z_dim) # output mean
        self.fc3 = layers.Dense(z_dim) # output variance

        # Decoder
        self.fc4 = layers.Dense(128)
        self.fc5 = layers.Dense(784)
```

The input of the Encoder first passes through the shared layer FC1, and then through the FC2 and FC3 networks respectively, to obtain the log vector value of the mean vector and variance of the hidden vector distribution.

```
    def encoder(self, x):
        # Get mean and varaince
        h = tf.nn.relu(self.fc1(x))
        # Mean vector
        mu = self.fc2(h)
        # Log of variance
```

```
    log_var = self.fc3(h)

    return mu, log_var
```

Decoder accepts the hidden vector $\mathbf{z}$ after sampling, and decodes it into picture output.

```
def decoder(self, z):
    #  Generate image data based on hidden variable z
    out = tf.nn.relu(self.fc4(z))
    out = self.fc5(out)
    #  Return image data, 784 vector
    return out
```

In the forward calculation process of VAE, the distribution of the input latent vector $\mathbf{z}$ is first obtained by the encoder, and then the latent vector $\mathbf{z}$ is obtained by sampling the reparameterize function implemented by Reparameterization Trick, and finally the reconstructed picture vector can be restored by the decoder. The implementation is as follows:

```
def call(self, inputs, training=None):
    # Forward calculation
    # Encoder [b, 784] => [b, z_dim], [b, z_dim]
    mu, log_var = self.encoder(inputs)
    # Sampling - reparameterization trick
    z = self.reparameterize(mu, log_var)
    # Decoder
    x_hat = self.decoder(z)
    # Return sample, mean and log variance
    return x_hat, mu, log_var
```

12.5.2 Reparameterization Trick

The Reparameterize function accepts the mean and variance parameters, and obtains ε by sampling from the normal distribution $\mathcal{N}(0, I)$, and returns the sampled hidden vector by $z = \mu + \sigma \odot \varepsilon$.

```
def reparameterize(self, mu, log_var):
    # reparameterize trick
    eps = tf.random.normal(log_var.shape)
    # calculate standard variance
    std = tf.exp(log_var)**0.5
    # reparameterize trick
    z = mu + std * eps
    return z
```

12.5.3 Network Training

The network is trained for 100 Epochs, and the reconstruction samples are obtained from the

forward calculation of the VAE model each time. The reconstruction error term $\mathbb{E}_{\boldsymbol{z}\sim q}[log\ p_\theta(\boldsymbol{x}|\boldsymbol{z})]$ is calculated based on cross-entropy loss function. The error term $\mathbb{D}_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})\right)$ is calculated based on equation (12-2).

```
# Create network objects
model = VAE()
model.build(input_shape=(4, 784))
# Optimizer
optimizer = optimizers.Adam(lr)

for epoch in range(100): # Train 100 Epoches
    for step, x in enumerate(train_db): #  Traverse the training set
        # Flatten, [b, 28, 28] => [b, 784]
        x = tf.reshape(x, [-1, 784])
        #  Build a gradient recorder
        with tf.GradientTape() as tape:
            # Forward calculation
            x_rec_logits, mu, log_var = model(x)
            #  Reconstruction loss calculation
            rec_loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=x, log
its=x_rec_logits)
            rec_loss = tf.reduce_sum(rec_loss) / x.shape[0]
            # Calculate KL convergence N(mu, var) VS N(0, 1)
            # Refernece: https://stats.stackexchange.com/questions/7440/kl-
divergence-between-two-univariate-gaussians
            kl_div = -0.5 * (log_var + 1 - mu**2 - tf.exp(log_var))
            kl_div = tf.reduce_sum(kl_div) / x.shape[0]
            # Combine error
            loss = rec_loss + 1. * kl_div
        # Calculate gradients
        grads = tape.gradient(loss, model.trainable_variables)
        # Update paramaters
        optimizer.apply_gradients(zip(grads, model.trainable_variables))


        if step % 100 == 0:
            # Print error
            print(epoch, step, 'kl div:', float(kl_div), 'rec loss:', float(
rec_loss))
```

12.5.4 Image generation

Picture generation only uses the decoder network. First, the hidden vector is sampled from

the prior distribution $\mathcal{N}(0, I)$, and then the picture vector is obtained through the decoder, and finally is reshaped to picture matrix. E.g:

```
    #  Test generation effect, randomly sample z from normal distribution
    z = tf.random.normal((batchsz, z_dim))
    logits = model.decoder(z) #  Generate pictures only by decoder
    x_hat = tf.sigmoid(logits) #  Convert to pixel range
    x_hat = tf.reshape(x_hat, [-1, 28, 28]).numpy() *255.
    x_hat = x_hat.astype(np.uint8)
    save_images(x_hat, 'vae_images/epoch_%d_sampled.png'%epoch) # Save
pictures


    # Reconstruct the picture, sample pictures from the test set
    x = next(iter(test_db))
    logits, _, _ = model(tf.reshape(x, [-1, 784])) # Flatten and send to
autoencoder
    x_hat = tf.sigmoid(logits) #  Convert output to pixel value
    # Restore to 28x28,[b, 784] => [b, 28, 28]
    x_hat = tf.reshape(x_hat, [-1, 28, 28])
    # The first 50 input + the first 50 reconstructed pictures merged,
[b, 28, 28] => [2b, 28, 28]
    x_concat = tf.concat([x[:50], x_hat[:50]], axis=0)
    x_concat = x_concat.numpy() * 255.
    x_concat = x_concat.astype(np.uint8)
    save_images(x_concat, 'vae_images/epoch_%d_rec.png'%epoch)
```

The effect of picture reconstruction is shown in Figure 12.15, Figure 12.16, and Figure 12.17, which show the reconstruction effect obtained by inputting the pictures of the test set at the first, tenth, and 100th Epoch respectively. The left 5 columns of each picture are real pictures, and the 5 columns on the right are the corresponding reconstruction effects. The effect of picture generation is shown in Figure 12.18, Figure 12.19, and Figure 12.20, respectively showing the effect of the image generation at the first, tenth, and 100th Epoch.

Figure 0.15 Picture reconstruction：epoch=1

Figure 0.16 Picture reconstruction：epoch=10

Figure 0.17 Picture reconstruction：epoch=100

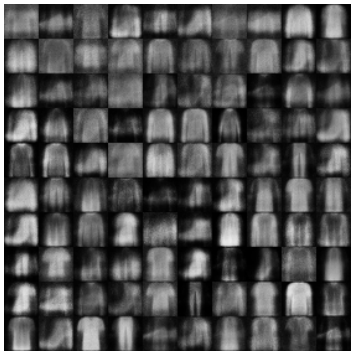

Figure 0.18 Picture generation：epoch=1

Figure 0.19 Picture generation：epoch=10

Figure 0.20 Picture generation：epoch=100

It can be seen that the effect of image reconstruction is slightly better than that of image generation, which also shows that image generation is a more complex task. Although the VAE model has the ability to generate images, the generated effect is still not good enough, and the human eye can still distinguish the difference between machine-generated and real picture samples. The generative confrontation network that will be introduced in the next chapter performs better in image generation.

12.6 References

[1] G. E. Hinton, “Reducing the Dimensionality of Data with Neural,” 2008.

[2] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.