AUTOENCODER

NEED FOR AUTOENCODERS:

- The need for autoencoders arises from their ability to address challenges such as high dimensionality, unsupervised learning, noise in data, and the extraction of meaningful features. Their versatility makes them valuable in a wide range of applications across various domains.
- In many real-world applications, data can be high-dimensional and redundant. High-dimensional data may suffer from the curse of dimensionality, making it computationally expensive and challenging to process. Autoencoders help reduce the dimensionality of data by learning compact representations.
- Unsupervised learning scenarios involve datasets without labeled target variables. Autoencoders, as unsupervised models, can learn patterns and structures in the data without the need for labeled examples.
- Extracting relevant features from raw data is crucial for effective machine learning models. Autoencoders are capable of learning meaningful representations or features from input data, which can be used for downstream tasks such as classification or clustering.
- Real-world data is often noisy, and learning robust representations is essential. Denoising autoencoders are specifically designed to reconstruct clean data from noisy inputs, helping the model focus on essential patterns and filter out noise.
- Identifying anomalies or outliers in a dataset is crucial for tasks like fraud detection or system monitoring. Autoencoders can learn normal patterns during training and highlight deviations during testing, making them useful for anomaly detection.
- Learning meaningful representations of data is essential for understanding and interpreting complex patterns. Autoencoders help in capturing hierarchical representations, allowing for better insights into the underlying structure of the data.

AUTO ENCODER:

An auto encoder is a type of artificial neural network used for unsupervised learning. Its main purpose is to learn efficient representations or encodings of input data, typically for the purpose of dimensionality reduction or feature learning. Autoencoders consist of an encoder and a decoder, both of which are neural networks.

Encoder: The encoder takes input data and maps it to a lower-dimensional representation, often called the encoding or latent space. This step involves compressing the input data into a compact representation.

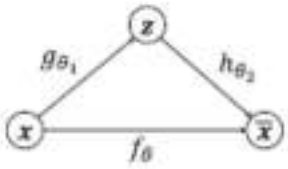
Decoder: The decoder then takes this lower-dimensional representation and attempts to reconstruct the original input data from it. The goal is to generate an output that is as close as possible to the input.

Objective Function: The training of an autoencoder involves minimizing a loss function, which measures the difference between the input data and the reconstructed output. Common loss functions include mean squared error (MSE) or binary cross-entropy, depending on the type of data being used.

PRINCIPLE OF AUTO ENCODER:

- For classification problems, the network model transforms the input feature vector x of length d_{in} to the output vector o of length d_{out}.
- This process can be considered as a **feature reduction process**, transforming the original high-dimensional input vector x to a low-dimensional variable 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 component analysis (PCA),** but PCA is essentially a linear transformation, and the ability to extract features is limited.
- Under **unsupervised learning**, we try to use the data 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 $f_\theta : x \rightarrow x$.
- We divide the network f_θ into two parts. The first sub-network tries to learn the mapping relationship: $g_{\theta_1} : x \rightarrow z$

  And the latter sub-network tries to learn the mapping relationship $h_{\theta_2} : z \rightarrow x$.
- The function "g" **encodes** the high-dimensional input x into a low-dimensional hidden variable z (latent variable or hidden variable), which is called **an encoder network**.
- h_{θ_2} is considered as the process of data decoding, which decodes the encoded input z into high-dimensional x, which is called **a decoder network**.



Autoencoder model

- The encoder and decoder jointly complete the encoding and decoding process of the input data x. We call the entire network model f_θ an **autoencoder** for short.
- If a deep neural network is used to parameterize g_{θ_1} and h_{θ_2} functions, it is called **deep autoencoder.**

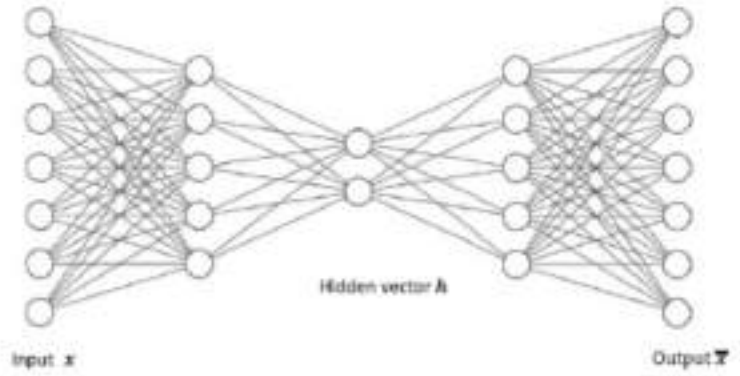


***Figure 12-2.** Autoencoder using neural network parametertzation*

- The self-encoder can transform the input to the hidden vector z, and reconstruct x through the decoder.

- We hope that the output of the decoder can perfectly or approximately recover the original input, that is $x\_ \approx x$, then the optimization goal of the autoencoder can be written as:

$$\min L = dist(x, \underline{x})$$

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

- where $dist(x, x\_)$ represents the distance measurement between x and $x\_$, which is called the **reconstruction error function**. The most common measurement method is the square of the Euclidean distance.

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

HANDS-ON FASHION MNIST IMAGE RECONSTRUCTION:

- Fashion MNIST is a dataset that is a slightly more complicated problem than MNIST image recognition. Its settings are almost the same as MNIST.
- It contains ten types of grayscale images of different types of clothes, shoes, and bags, and the size of the image is 28 × 28, with 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.
-

STEP1: LOADING THE DATA SET

```
# Load Fashion MNIST data set
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_
mnist.load_data()
# Normalize
x_train, x_test = x_train.astype(np.float32) / 255., x_test.astype(np.float32) / 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)
```

STEP2: ENCODER
Here we reduce the dimension from 784 to 20(h_dim , here) . The Encoder is composed of a 3-layer fully connected network with output nodes of 256, 128, and 20, respectively.
A three-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)
])
```

Step3:DECODER
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 in the following:

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

ARCHITECTURE:

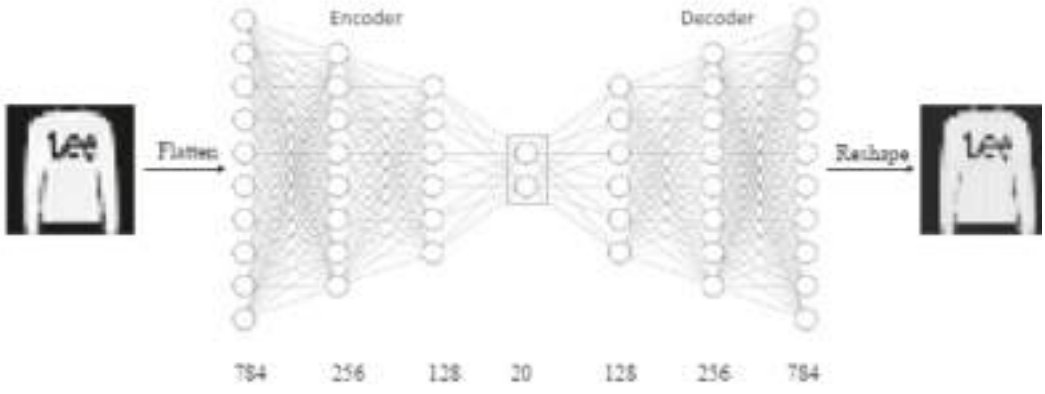


Figure 12-5. Fashion MNIST autoencoder network architecture

STEP4: AUTOENCODER

The preceding two sub-networks of encoder and decoder are implemented in the autoencoder 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)
])
```

STEP5: FORWARD PROPOGATION

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
```

STEP6: NETWORK TRAINING

The training process of the autoencoder is basically the same as that of a classifier. The distance between the reconstructed vector *x* and the original input vector *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.
- Here 100 Epochs are trained, and the reconstructed image vector is obtained through forward calculation each time
- 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.(You can also use MSE)

```
# 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)

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, logits=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))
```

STEP7: IMAGE RECONSTRUCTION

In order to test the effect of image reconstruction, we divide the dataset into a training set and a test set, We randomly sample the test picture $x \in D_{test}$ 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 in the following:

```
# 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

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)
```

EFFECTS OF RECONSTRUCTION:

First Epoch

Tenth Epoch

Hundredth Epoch

VARIATIONAL AUTOENCODER

INTRODUCTION:

- The basic autoencoder essentially learns the mapping relationship between the input *x* and the hidden variable *z*. It is a discriminative model, not a generative model.
- Autoencoders have emerged as an architecture for data representation and generation. Among them, Variational Autoencoders (VAEs) stand out, introducing probabilistic encoding and opening new avenues for diverse applications.
- Autoencoders are neural network architectures that are intended for the compression and reconstruction of data. It consists of an encoder and a decoder; these networks are learning a simple representation of the input data.
- The encoder aims to learn efficient data encoding from the dataset and pass it into a bottleneck architecture. The other part of the autoencoder is a decoder that uses latent space in the bottleneck layer to regenerate images similar to the dataset. These results backpropagate the neural network in the form of the loss function.
- Variational autoencoder was proposed in 2013 by Diederik P. Kingma and Max Welling at Google and Qualcomm.
- A variational autoencoder (VAE) provides a probabilistic manner for describing an observation in latent space.
- Thus, rather than building an encoder that outputs a single value to describe each latent state attribute, we'll formulate our encoder to describe a probability distribution for each latent attribute.
- In a nutshell, a VAE is an autoencoder whose encodings distribution is regularised during the training in order to ensure that its latent space has good properties allowing us to generate some new data.
- It has many applications, such as data compression, synthetic data creation, etc.

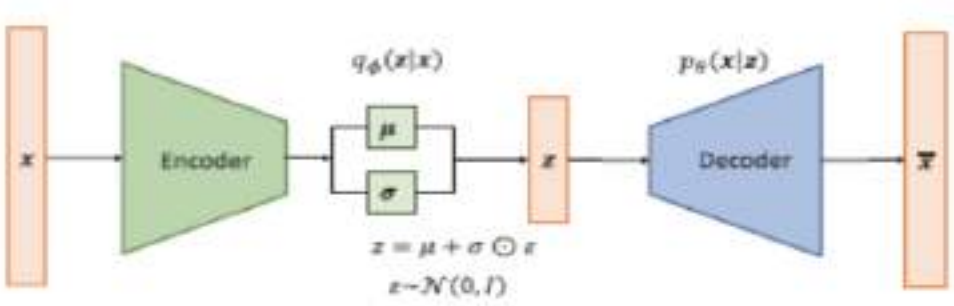
PRINCIPLE OF VARIATONAL AUTOENCODER:

- Variational autoencoder is different from an autoencoder in a way that it provides a statistical manner for describing the samples of the dataset in latent space.
- Therefore, in the variational autoencoder, the encoder outputs a probability distribution in the bottleneck layer instead of a single output value.
- Just as a standard autoencoder, a variational autoencoder is an architecture composed of both an encoder and a decoder and that is trained to minimise the reconstruction error between the encoded-decoded data and the initial data. However, in order to introduce some regularisation of the latent space, we proceed to a slight modification of the encoding-decoding process: **instead of encoding an input as a single point, we encode it as a distribution over the latent space**.
- The principle of VAE is based on combining the concepts of variational inference and neural networks to learn a probabilistic mapping between the input data and a latent space.
- VAE is a **probabilistic latent variable model** , VAE assumes that the observed data (input) is generated from an underlying probabilistic process involving latent variables.The goal is to infer the distribution of these latent variables given the observed data.

- The model is trained as follows:

- first, the input is encoded as distribution over the latent space
- second, a point from the latent space is sampled from that distribution
- third, the sampled point is decoded and the reconstruction error can be computed
- finally, the reconstruction error is backpropagated through the network

VAE MODEL ARCHITECTURE:



Components of a Variational Autoencoder (VAE):

1. Encoder:

The encoder takes input data and maps it to a probabilistic latent space.

Input Layer:

Accepts the input data, typically flattened or represented as a vector.

Hidden Layers:

Comprise fully connected (dense) layers or convolutional layers.
These layers progressively reduce the dimensionality of the input, capturing relevant features.

Mean and Variance Layers:

Two separate layers output the mean (μ) and the logarithm of the variance (log(σ^2)) of the latent distribution.
These parameters define a normal distribution from which the latent vector will be sampled.

Sampling Layer (Reparameterization Trick):
A layer that samples from a normal distribution using the mean (μ) and variance (σ^2) obtained from the previous layers. This layer introduces stochasticity in the model.

2. Latent Space:
The latent space is a lower-dimensional space where each point represents a potential encoding of the input data.
Latent Vector (z):
The output of the sampling layer in the encoder. Represents a point in the continuous latent space.

3. Decoder:
The decoder takes a point from the latent space and reconstructs the input data.
Input Layer:
Accepts the latent vector (z) sampled from the latent space.
Hidden Layers:
Symmetric to the encoder, progressively increasing the dimensionality to produce the reconstructed data.
Output Layer:
Produces the reconstructed data, attempting to match the input data.

4. Loss Function:
The loss function for a VAE has two components:

Reconstruction Loss:
Measures the difference between the input data and the reconstructed data.
Commonly, binary cross-entropy loss for binary data or mean squared error for continuous data.

KL Divergence Loss:
Measures the difference between the learned latent distribution and a prior distribution (often a standard normal distribution).
This term regularizes the model, encouraging the latent space to follow a specific structure.

NOTE:
Reparameterization Trick:

- Instead of sampling directly, a parameterized transformation is applied to the parameters of the distribution.
- The parameters (mean and standard deviation) of the distribution are treated as deterministic variables.
- A noise term sampled from a fixed distribution (e.g., standard normal) is then multiplied by the standard deviation and added to the mean to obtain the sampled latent vector.
- In the context of a VAE, let's say we have a normal distribution with mean (μ) and standard deviation (σ).

- The reparameterization trick transforms the sampling process as follows:

$$z=\mu+\sigma\cdot\epsilon$$

Where:

z is the sampled latent vector.

μ is the mean of the distribution.

σ is the standard deviation of the distribution.

ϵ is a noise term sampled from a fixed distribution (e.g., standard normal).

By using this reparameterization, the entire process becomes differentiable, enabling the gradients to flow through the sampling operation during backpropagation.
This is crucial for training generative models with latent variables, such as VAEs, where the stochasticity of the latent space is a fundamental part of the model.

GAN-GENERATIVE ADVERSARIAL NETWORK

INTRODUCTION:

DISADVANTAGE OF VAE:

- VAE-Variational Auto Encoder is very stable when trained using neural networks, and the resulting images are more approximate.
- But the human eyes can still easily distinguish real pictures and machine-generated pictures.

GAN:

- In 2014, Ian Goodfellow, a student of Yoshua Bengio (the winner of the Turing Award in 2018) at the Université de Montréal, proposed the GAN.

Figure 13-1. GAN generated image effect from 2014 to 2018

- GAN APPLICATIONS: Generate Examples for Image Datasets
- Generate Photographs of Human Faces
- Generate Realistic Photographs
- Generate Cartoon Characters
- Image-to-Image Translation
- Text-to-Image Translation
- Semantic-Image-to-Photo Translation
- Face Frontal View Generation
- Photos to Emojis
- Photograph Editing
- Face Aging

- Photo Inpainting
- Clothing Translation
- Video Prediction

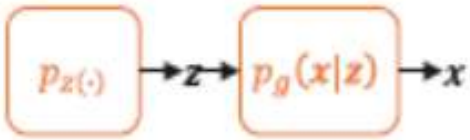
GAN PRINCIPLE:

NETWORK STRUCTURE OF GAN:

- GAN contains two sub-networks: the generator network (referred to as G) and the discriminator network (referred to as D).
- Adversarial training is a machine learning technique that involves training a model in an adversarial manner, typically in the context of Generative Adversarial Networks (GANs).
- The idea is to set up a game between two neural networks, where they are in opposition to each other, hence the term "adversarial."
- The generator network G is responsible for learning the true distribution of samples.
- The generator is tasked with creating data, such as images, that is indistinguishable from real data.
- It takes random noise as input and transforms it into data.
- The discriminator network D is tasked with distinguishing between real data (from the actual dataset) and generated data (produced by the generator).
- It classifies input as either real or fake.

GENERATOR:

- The generator takes random noise as input (usually sampled from a simple distribution, like a Gaussian distribution).
- The input noise is then passed through a neural network, often composed of multiple layers.
- The neural network transforms the input noise into data that ideally resembles samples from the real dataset.
- Common architectural choices for the generator include fully connected layers, transposed convolutional layers (in the case of image data), or other architectures depending on the nature of the data.
- The hidden variables $z \sim Pz(\cdot)$ are sampled from the prior distribution $Pz(\cdot)$.
- The generated sample $x \sim Pg(x|z)$ is obtained by the parameterized distribution $Pg(x|z)$ of the generator network G, as shown in Figure



Generator G

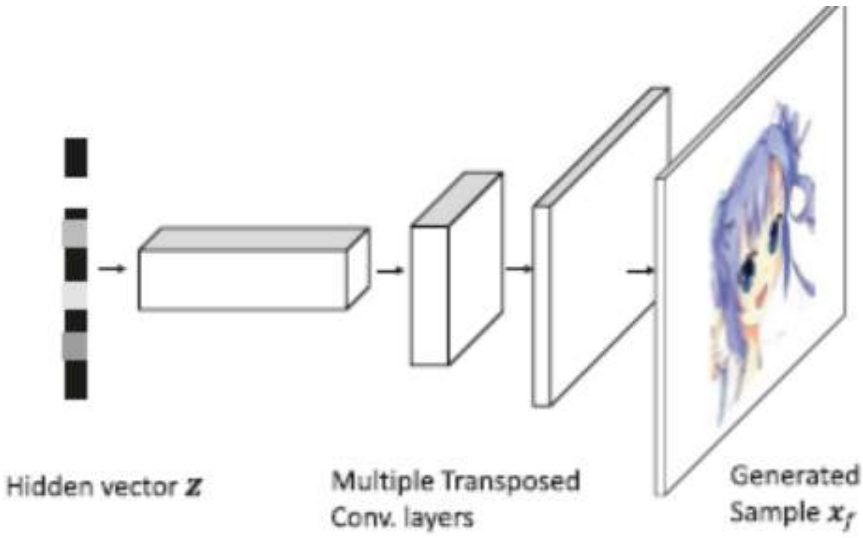


Figure 13-4. *Generator network composed of transposed convolution*

DISCRIMINATOR:

- The discriminator takes input data, which can be either real data from the actual dataset or generated data from the generator.
- Like the generator, the discriminator is also a neural network, typically composed of multiple layers.
- The neural network classifies the input as either real or fake (generated).
- The output of the discriminator is a probability score indicating the likelihood that the input is real.
- It accepts a dataset of input sample *x*, including samples *xr*~*Pr*(·) sampled from the real data distribution *Pr*(·), and also includes fake samples sampled from the generator network *xf*~*Pg*(*x*| *z*).
- *xr* and *xf* together form the training data set of the discriminator network.
- The output of the discriminator network is the probability of *x* belonging to the real sample *P*(*x is real* | *x*).
- We label all the real samples *xr* as true (1), and all the samples *xf* generated by the generator network are labeled as false (0).

- The error between the predicted value of the discriminator network D and the label is used to optimize the discriminator network parameters as shown in Figure

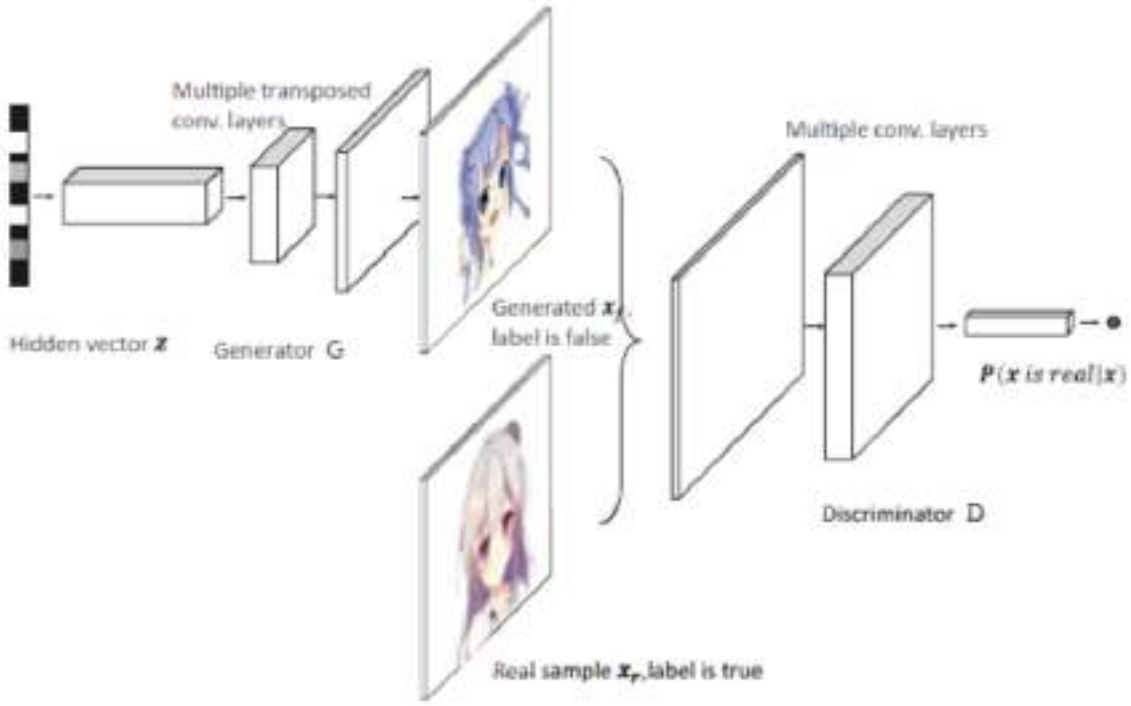
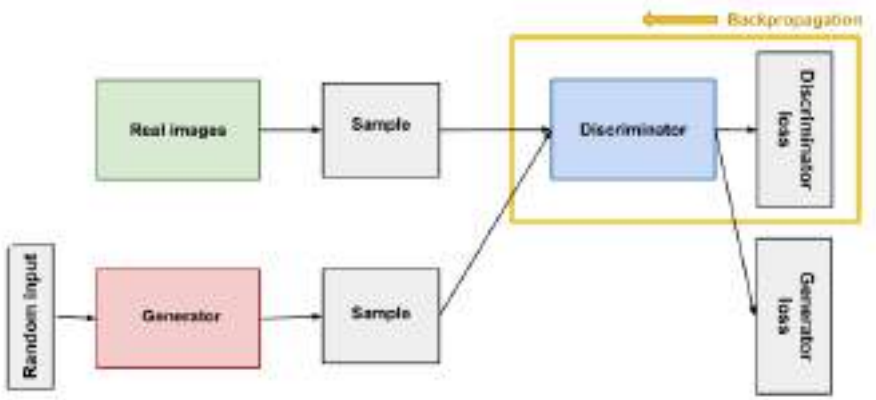


Figure 13-5. Generator network and discriminator network

TRAINING PROCESS:

- During training, the generator and discriminator are trained simultaneously in an adversarial manner.
- The generator aims to generate data that is indistinguishable from real data, making the discriminator unable to differentiate between real and generated samples.
- The discriminator aims to correctly classify real and generated samples. It is trained to maximize the probability of assigning the correct label (real or fake).

LOSS FUNCTION:



- The training process involves a minimax game, where the generator tries to minimize the probability of the discriminator correctly classifying generated samples as fake, and the discriminator tries to maximize this probability.
- The loss functions for the generator (G) and discriminator (D) are typically defined as follows:
    - Generator Loss (L_G): -log(D(G(z)))
    - Discriminator Loss (L_D): -[log(D(x)) + log(1 - D(G(z)))]
- These loss functions guide the optimization process during training.

GAN VARIANTS:

DCGAN:

- The initial GAN network is mainly based on the fully connected layer to realize the generator G and the discriminator D.
- Due to the high dimensionality of the picture and the huge amount of network parameters, the training effect is not excellent.
- DCGAN proposed a generator network implemented using transposed convolutional layers, and a discriminator network implemented by ordinary convolutional layers, which greatly reduces the amount of network parameters and greatly improves the effect of image generation, showing that the GAN model has the potential of outperforming the VAE model in image generation.
- **Architecture:** DCGANs use convolutional neural networks (CNNs) in both the generator and discriminator.
- **Benefits:** CNNs enable the model to capture spatial hierarchies of features, making DCGANs particularly effective for image generation tasks.
- **Stability:** DCGANs are known for providing more stable training compared to the original GAN architecture, helping to overcome issues such as mode collapse.

INFOGAN:

- InfoGAN, short for Information Maximizing Generative Adversarial Network, is a variant of Generative Adversarial Networks (GANs) designed to learn disentangled and interpretable representations in the latent space.
- InfoGAN was introduced to address the challenge of controlling and understanding the factors of variation in generated data.
- It extends the basic GAN framework by incorporating an information-theoretic regularization term into the GAN loss.
- **Generator:**

  Similar to a standard GAN, the generator takes random noise as input and generates samples.

- **Discriminator:**

  The discriminator has two main tasks:

  - Distinguish between real and generated samples.
  - Predict additional structured information about the generated data.

- **Latent Code:**

  InfoGAN introduces a latent code (c) in addition to the random noise input (z).

  The latent code represents specific, interpretable factors of variation in the generated data.

- InfoGAN adds an information-theoretic regularization term to the GAN loss. The objective is to maximize the mutual information between the latent code (c) and the generated data. The regularization term encourages the network to learn disentangled representations:
- The first term is the standard adversarial loss to distinguish between real and generated samples.
- The second term encourages the generator to produce samples that are difficult for the discriminator to classify correctly.
- The third term is the information-theoretic regularization term, where $I(c;G(z,c))$ is the mutual information between the latent code and the generated sample.
- λ is a hyperparameter controlling the strength of the regularization.

CYCLEGAN:

- CycleGAN, short for Cycle-Consistent Generative Adversarial Network, is a type of deep learning model used for image-to-image translation tasks.
- CycleGAN is an unsupervised algorithm for image style conversion proposed by Zhu Junyan.
- The basic assumption of CycleGAN is that if you switch from picture A to picture B, and then from picture B to A', then A' should be the same picture as A. Therefore, in addition to setting up the standard GAN loss item, CycleGAN also adds cycle consistency loss to ensure that A' is as close to A as possible.
- CycleGAN is particularly designed for scenarios where you have a set of images from one domain and a set of images from another domain, but you don't have a direct mapping between the two. Unlike traditional GANs, CycleGAN does not require paired training data, where each image in one domain has a corresponding image in the other domain.

- The model consists of two generators (G and F) and two discriminators (D_X and D_Y). The generators learn to map images from one domain to another (e.g., from horses to zebras or from summer to winter landscapes), while the discriminators help evaluate the realism of the generated images.

WGAN:

- The training problem of GAN has been criticized all the time, and it is prone to the phenomenon of training non-convergence and mode collapse.
- WGAN, or Wasserstein Generative Adversarial Network, is a type of generative adversarial network (GAN) introduced by Martin Arjovsky, Soumith Chintala, and Léon Bottou in their 2017 paper titled "Wasserstein GAN." WGAN aims to address some of the training instability and mode collapse issues associated with traditional GANs.

- The key innovation of WGAN lies in its use of Wasserstein distance (also known as Earth Mover's Distance) as the training objective, as opposed to the Jensen-Shannon divergence or the Kullback-Leibler divergence used in traditional GANs. The Wasserstein distance provides a more meaningful measure of the difference between two probability distributions and has certain properties that make it more suitable for GAN training.

- The Wasserstein distance is defined as the minimum amount of "work" needed to transform one distribution into another. In the context of GANs, it encourages the generator to produce samples that are closer to the real data distribution, leading to more stable training and better convergence.

- The loss function in WGAN is based on the Wasserstein distance and is often referred to as the Wasserstein loss. The discriminator in WGAN is no longer constrained to output probabilities between 0 and 1, as in traditional GANs. Instead, it outputs a real-valued score, and the Wasserstein loss is computed as the difference between the scores assigned to real and generated samples.

SELF ATTENTION GAN:

The attention mechanism has been widely used in natural language processing (NLP). Self-Attention GAN (SAGAN) borrowed from the attention mechanism and proposed a variant of GAN based on the self-attention mechanism.

SAGAN improved the fidelity index of the picture: Inception score from the 36.8 to 52.52, and Frechet inception distancefrom 27.62 to 18.65. From the effect of image generation perspective,
SAGAN's breakthrough is very significant, and it also inspired the industry's attention to the self-attention mechanism.

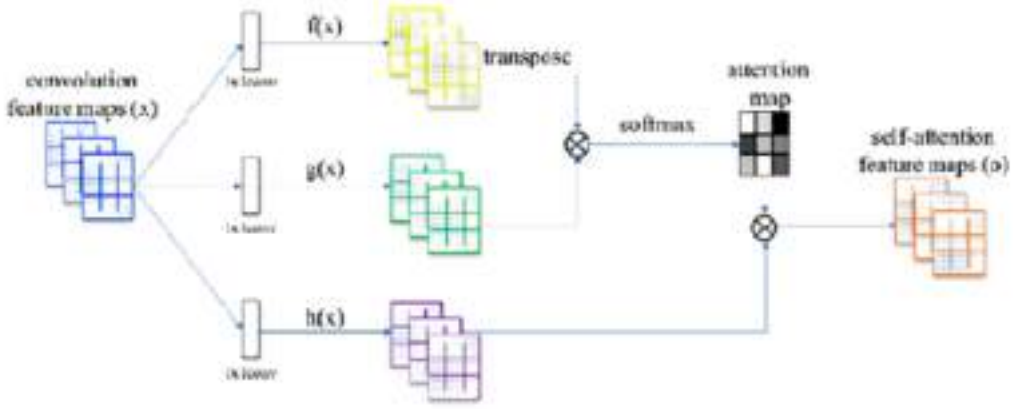
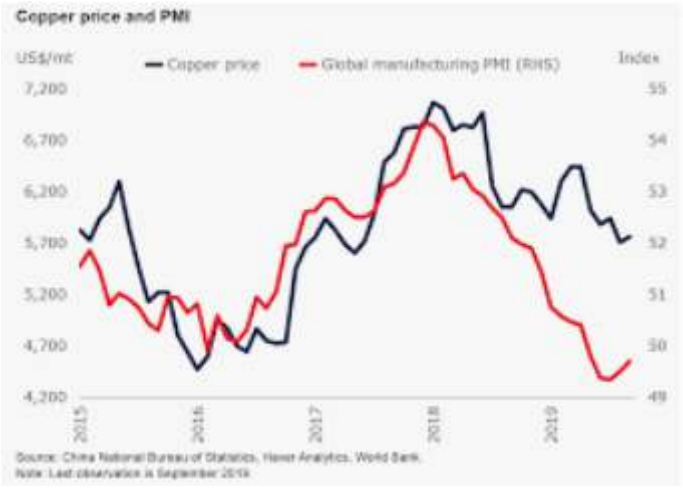


Figure 13-12. *Attention mechanism in SAGAN [8]*

SEQUENCE REPRESENTATION METHOD:

- Data with order is generally called a sequence.
- **For example**, commodity price data that changes over time is a very typical sequence.
- Assume, price trend of a commodity between January to June as a 1D vector [*x*1, *x*2, *x*3, *x*4, *x*5, *x*6], and its shape is [6].



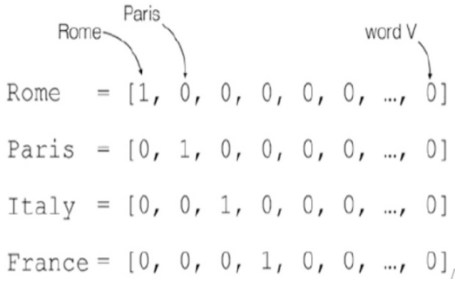
- **Example 2:**
- price change trend of ***b* goods** from January to June, you can record it as a 2-dimensional tensor:

$$\left[\left[x_1^{(1)}, x_2^{(1)}, \cdots, x_6^{(1)}\right], \left[x_1^{(2)}, x_2^{(2)}, \cdots, x_6^{(2)}\right], \cdots, \left[x_1^{(b)}, x_2^{(b)}, \cdots, x_6^{(b)}\right]\right]$$

- where *b* represents the number of commodities, and the tensor shape is [*b*, 6].
- SO WHAT'S THE PROBLEM?? For the previous case, A tensor with shape [b, s] is needed, where **b is the number of sequences and s is the length of the sequence.**
- However, many signals cannot be directly represented by a scalar value.
- For example, to represent feature vectors of length n generated by each timestamp, a tensor of shape **[b, s, n]** is required.
- Consider **more complex text data: sentences.**
- The word generated on each timestamp is a character, not a numerical value.
- Neural networks cannot directly process string data.
- Conversion of words or characters into numerical values becomes particularly critical, in neural network applications like NLP.

REPRESENTATION METHOD OF TEXT SEQUENCES:

- Consider a sentence having ‘n’ words.
- The process of encoding text into numbers as **Word Embedding.**Lets use **one – hot encoding** location names is as shown below:



- **Disadvantage of One Hot Encoding:** The one-hot encoding vector is high-dimensional and extremely sparse, with a large number of positions as 0s.
- Therefore, it is computationally expensive and also not conducive to the neural network training.
- It ignores the **semantic relevance inherent in words. Example:** “like,” “dislike,” “Rome,” “Paris,” “like,” and “dislike”.
- For a group of such words, if one-hot encoding is used, there is no correlation between the obtained vectors, and the semantic relevance of the original text cannot be well reflected.

SEMANTIC LEVEL OF RELEVANCE:

- In NLP, the semantic level of relevance can be well reflected through the **word vector.**
- One way to measure the correlation between word vectors is the cosine similarity:

$$similarity(a,b) \triangleq cos\,cos(\theta) = \frac{a \cdot b}{|a| \cdot |b|}$$

where *a* and *b* represent two word vectors.

EMBEDDING LAYER:

- In a neural network, the representation vector of a word can be obtained directly through training.
- We call the representation layer of the word **Embedding layer.**

- The Embedding layer is responsible for encoding the word into a word vector *v*.
- It accepts the word number *i* using digital encoding, such as 2 for “I” and 3 for “me”.
- The total number of words in the system is recorded as *Nvocab*, and the output is vector *v* with length *n*:

$$v = f_\theta(i|N_{vocab}, n)$$

- The Embedding layer is very simple to implement. Build a lookup table with shape [*Nvocab*, *n*].
- For any word number *i*, you only need to query the vector at the corresponding position and return: *v = table[i]*
- The Embedding layer is trainable. It can be placed in front of the neural network to complete the conversion of words to vectors.
- The resulting representation vector can continue to pass through the neural network to complete subsequent tasks, and calculate the error *L*.
- Sample Code:

```
x = tf.range(10) # Generate a digital code of 10 words
x = tf.random.shuffle(x) # Shuffle
# Create a layer with a total of 10 words, each word is
represented by a vector of length 4
net = layers.Embedding(10, 4)
out = net(x) # Get word vector
```

PRETRAINED WORD VECTORS:

- The lookup table of the Embedding layer is initialized randomly and needs to be trained from scratch.
- In fact, we can use pre-trained Word Embedding models to get the word representation.
- The word vector based on pre-trained models is equivalent to transferring the knowledge of the entire semantic space, which can often get better performance.
- Currently, the widely used pre-trained models include Word2Vec and GloVe.

- They have been trained on a massive corpus to obtain a better word vector representation and can directly export the learned word vector table to facilitate migration to other tasks.
- For example, the GloVe model GloVe.6B.50d has a vocabulary of 400,000, and each word is represented by a vector of length 50.
- Users only need to download the corresponding model file in order to use it. The “glove6b50dtxt.zip” model file is about 69MB.
- For the Embedding layer, random initialization is no longer used. Instead, we use the pre-trained model parameters to initialize the query table of the Embedding layer.

```
# Load the word vector table from the pre-trained model
embed_glove = load_embed('glove.6B.50d.txt')
# Initialize the Embedding layer directly using the pre-trained
word vector table
net.set_weights([embed_glove])
```