Assignment No. 8

Aim: Develop an autoencoder to encode and decode the image. Analyze the results.

a) Develop AE for MNIST dataset

b) Use output of AE as input to CNN

Objectives:

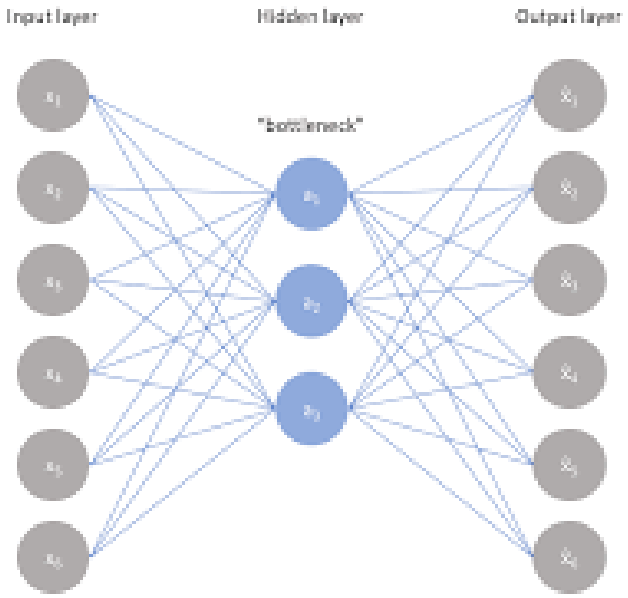
1. To learn AE

2. To implement AE

Theory:

Autoencoders

An **autoencoder** is a type of artificial neural network used to learn efficient data coding in an unsupervised manner. The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal “noise”. Along with the reduction side, a reconstructing side is learnt, where the autoencoder tries to generate from the reduced encoding a representation as close as possible to its original input, hence its name. Several variants exist to the basic model, with the aim of forcing the learned representations of the input to assume useful properties. Examples are the regularized autoencoders (*Sparse*, *Denoising* and *Contractive* autoencoders), proven effective in learning representations for subsequent classification tasks, and *Variational* autoencoders, with their recent applications as generative models. Autoencoders are effectively used for solving many applied problems, from face recognition to acquiring the semantic meaning of words.



There are, basically, 7 types of autoencoders:

- Denoising autoencoder
- Sparse Autoencoder
- Deep Autoencoder
- Contractive Autoencoder
- Undercomplete Autoencoder
- Convolutional Autoencoder
- Variational Autoencoder

1) Denoising Autoencoder

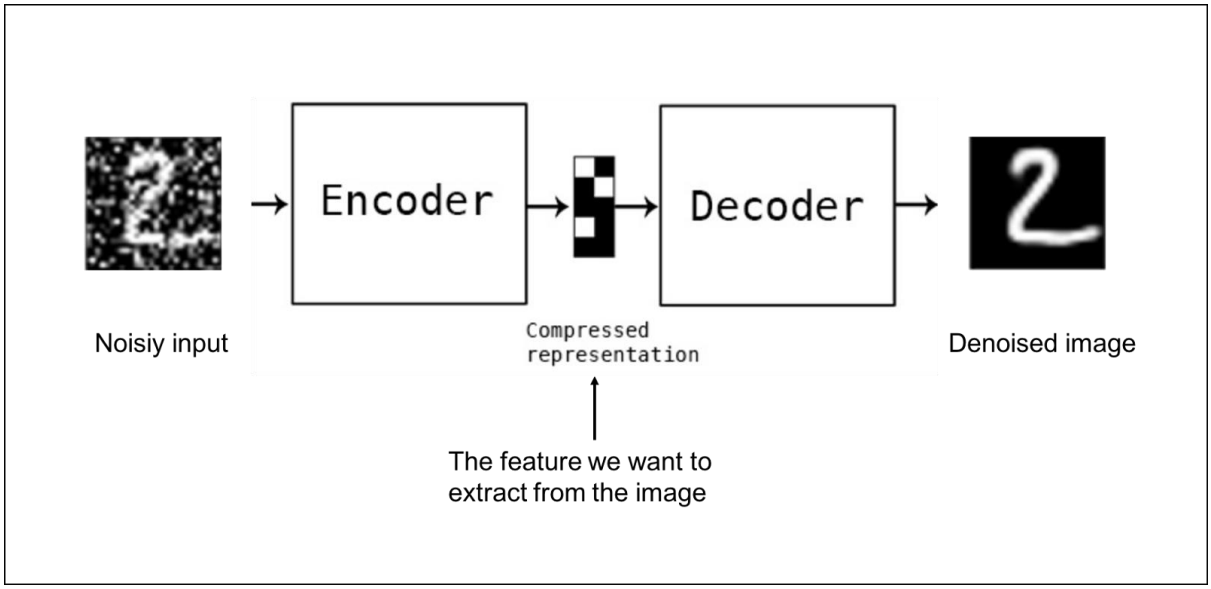
Denoising autoencoders create a corrupted copy of the input by introducing some noise. This helps to avoid the autoencoders to copy the input to the output without learning features about the data. These autoencoders take a partially corrupted input while training to recover the original undistorted input. The model learns a vector field for mapping the input data towards a lower dimensional manifold which describes the natural data to cancel out the added noise.

Advantages-

- It was introduced to achieve good representation. Such a representation is one that can be obtained robustly from a corrupted input and that will be useful for recovering the corresponding clean input.
- Corruption of the input can be done randomly by making some of the input as zero. Remaining nodes copy the input to the noised input.
- Minimizes the loss function between the output node and the corrupted input.
- Setting up a single-thread denoising autoencoder is easy.

Drawbacks-

- To train an autoencoder to denoise data, it is necessary to perform preliminary stochastic mapping in order to corrupt the data and use as input.
- This model isn't able to develop a mapping which memorizes the training data because our input and target output are no longer the same.



2) Sparse Autoencoder

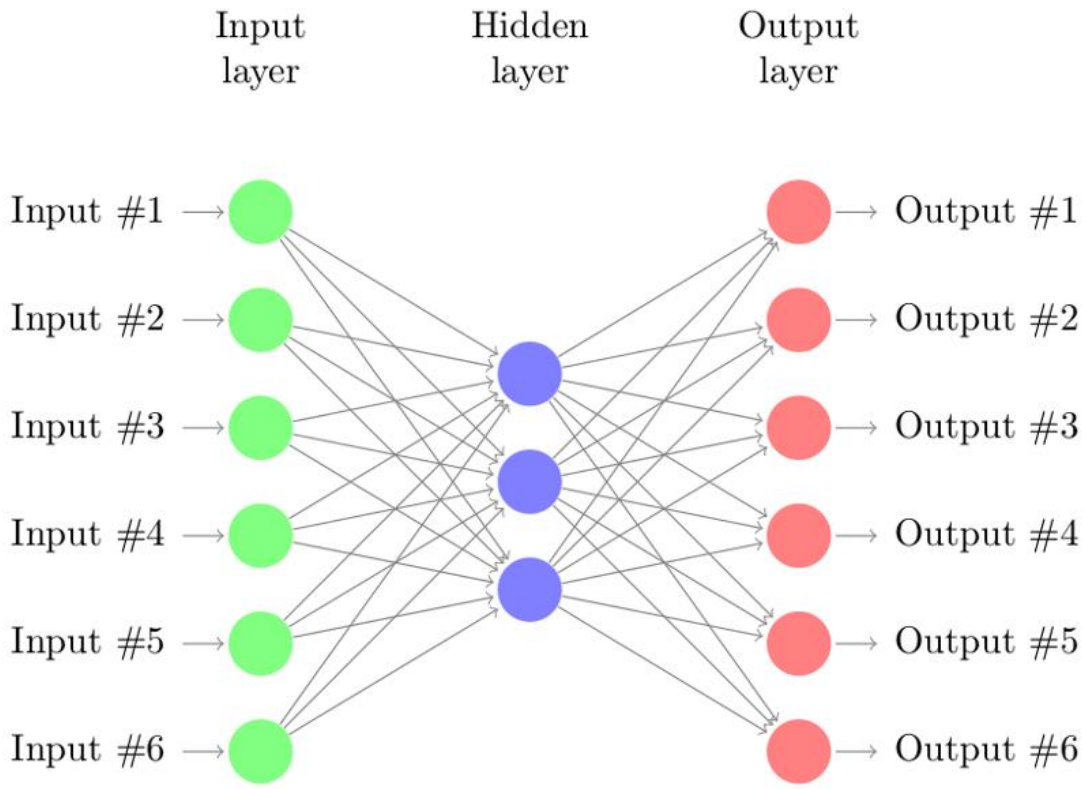
Sparse autoencoders have hidden nodes greater than input nodes. They can still discover important features from the data. A generic sparse autoencoder is visualized where the obscurity of a node corresponds with the level of activation. Sparsity constraint is introduced on the hidden layer. This is to prevent output layer copy input data. Sparsity may be obtained by additional terms in the loss function during the training process, either by comparing the probability distribution of the hidden unit activations with some low desired value,or by manually zeroing all but the strongest hidden unit activations. Some of the most powerful AIs in the 2010s involved sparse autoencoders stacked inside of deep neural networks.

Advantages-

- Sparse autoencoders have a sparsity penalty, a value close to zero but not exactly zero. Sparsity penalty is applied on the hidden layer in addition to the reconstruction error. This prevents overfitting.
- They take the highest activation values in the hidden layer and zero out the rest of the hidden nodes. This prevents autoencoders to use all of the hidden nodes at a time and forcing only a reduced number of hidden nodes to be used.

Drawbacks-

- For it to be working, it's essential that the individual nodes of a trained model which activate are data dependent, and that different inputs will result in activations of different nodes through the network.



3) Deep Autoencoder

Deep Autoencoders consist of two identical deep belief networks, oOne network for encoding and another for decoding. Typically deep autoencoders have 4 to 5 layers for encoding and the next 4 to 5 layers for decoding. We use unsupervised layer by layer pre-training for this model. The layers are Restricted Boltzmann Machines which are the building blocks of deep-belief networks. Processing the benchmark dataset MNIST, a deep autoencoder would use binary transformations after each RBM. Deep autoencoders are useful in topic modeling, or statistically modeling abstract topics that are distributed across a collection of documents. They are also capable of compressing images into 30 number vectors.

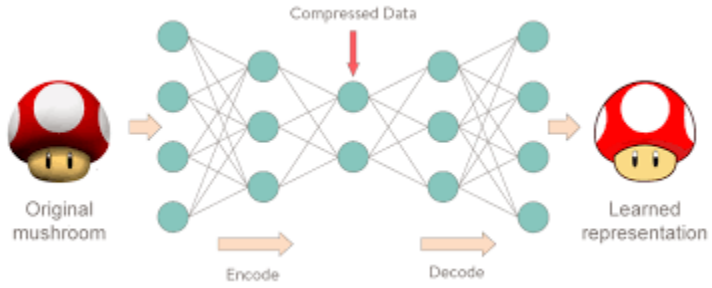
Advantages-

- Deep autoencoders can be used for other types of datasets with real-valued data, on which you would use Gaussian rectified transformations for the RBMs instead.
- Final encoding layer is compact and fast.

Drawbacks-

- Chances of overfitting to occur since there's more parameters than input data.

- Training the data maybe a nuance since at the stage of the decoder's backpropagation, the learning rate should be lowered or made slower depending on whether binary or continuous data is being handled.

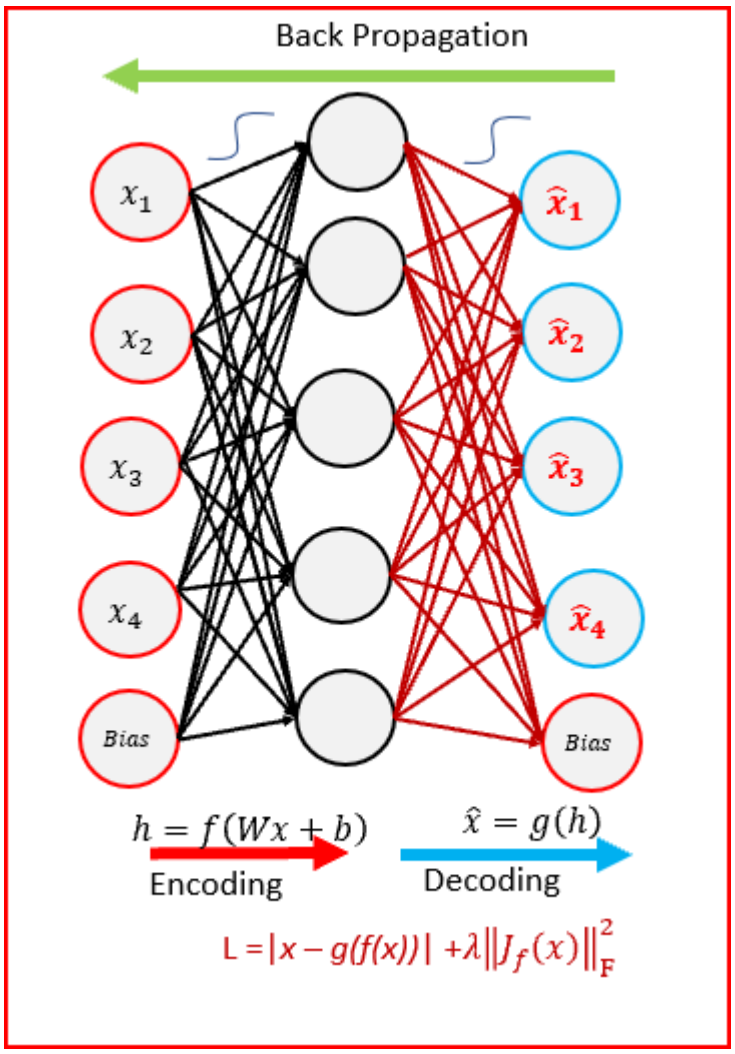


4) Contractive Autoencoder

The objective of a contractive autoencoder is to have a robust learned representation which is less sensitive to small variation in the data. Robustness of the representation for the data is done by applying a penalty term to the loss function. Contractive autoencoder is another regularization technique just like sparse and denoising autoencoders. However, this regularizer corresponds to the Frobenius norm of the Jacobian matrix of the encoder activations with respect to the input. Frobenius norm of the Jacobian matrix for the hidden layer is calculated with respect to input and it is basically the sum of square of all elements.

Advantages-

- Contractive autoencoder is a better choice than denoising autoencoder to learn useful feature extraction.
- This model learns an encoding in which similar inputs have similar encodings. Hence, we're forcing the model to learn how to contract a neighborhood of inputs into a smaller neighborhood of outputs.



5) Undercomplete Autoencoder

The objective of undercomplete autoencoder is to capture the most important features present in the data. Undercomplete autoencoders have a smaller dimension for hidden layer compared to the input layer. This helps to obtain important features from the data. It minimizes the loss function by penalizing the g(f(x)) for being different from the input x.

Advantages-

- Undercomplete autoencoders do not need any regularization as they maximize the probability of data rather than copying the input to the output.

Drawbacks-

- Using an overparameterized model due to lack of sufficient training data can create overfitting.

6) Convolutional Autoencoder

Autoencoders in their traditional formulation does not take into account the fact that a signal can be seen as a sum of other signals. Convolutional Autoencoders use the convolution operator to exploit

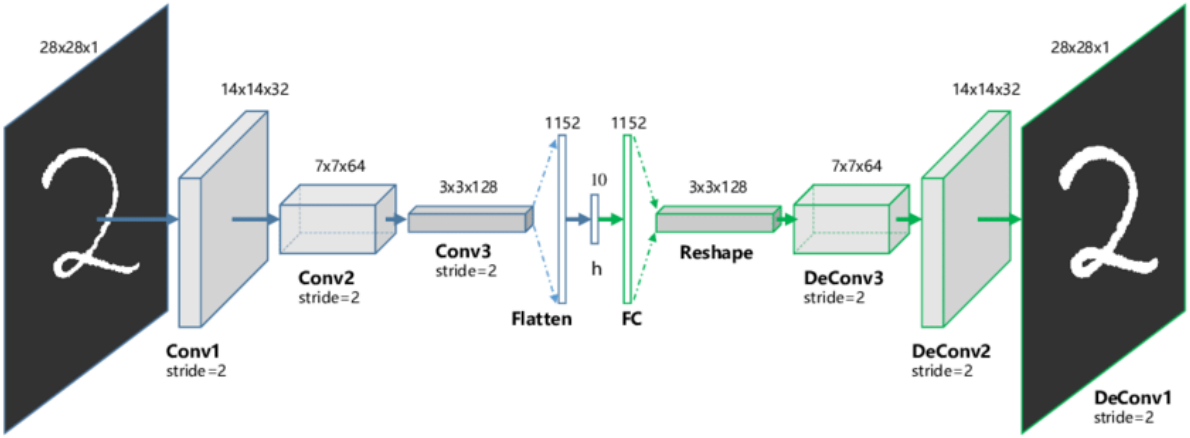
this observation. They learn to encode the input in a set of simple signals and then try to reconstruct the input from them, modify the geometry or the reflectance of the image. They are the state-of-art tools for unsupervised learning of convolutional filters. Once these filters have been learned, they can be applied to any input in order to extract features. These features, then, can be used to do any task that requires a compact representation of the input, like classification.

Advantages-

- Due to their convolutional nature, they scale well to realistic-sized high dimensional images.
- Can remove noise from picture or reconstruct missing parts.

Drawbacks-

- The reconstruction of the input image is often blurry and of lower quality due to compression during which information is lost.



7) Variational Autoencoder

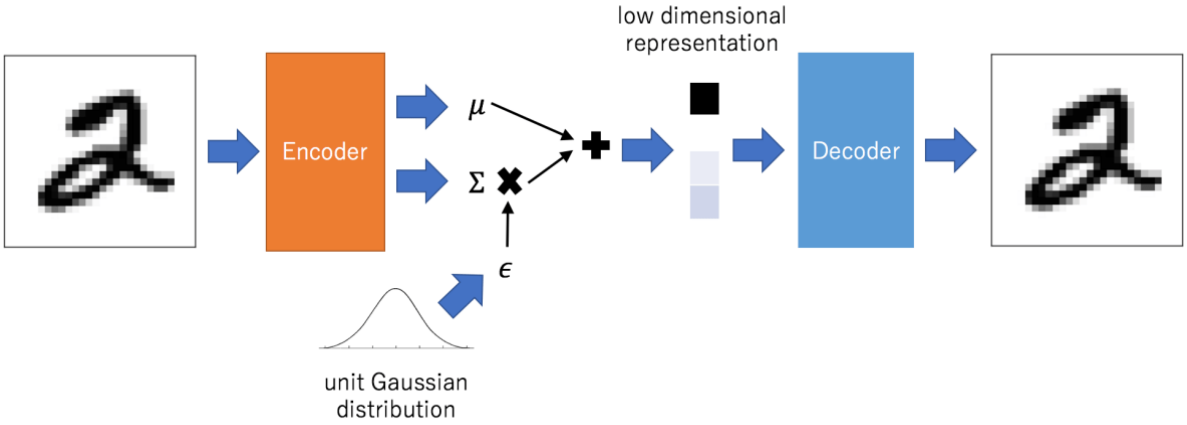
Variational autoencoder models make strong assumptions concerning the distribution of latent variables. They use a variational approach for latent representation learning, which results in an additional loss component and a specific estimator for the training algorithm called the Stochastic Gradient Variational Bayes estimator. It assumes that the data is generated by a directed graphical model and that the encoder is learning an approximation to the posterior distribution where Φ and θ denote the parameters of the encoder (recognition model) and decoder (generative model) respectively. The probability distribution of the latent vector of a variational autoencoder typically matches that of the training data much closer than a standard autoencoder.

Advantages-

- It gives significant control over how we want to model our latent distribution unlike the other models.
- After training you can just sample from the distribution followed by decoding and generating new data.

Drawbacks-

- When training the model, there is a need to calculate the relationship of each parameter in the network with respect to the final output loss using a technique known as backpropagation. Hence, the sampling process requires some extra attention.



Applications

Autoencoders work by compressing the input into a latent space representation and then reconstructing the output from this representation. This kind of network is composed of two parts:

1. Encoder: This is the part of the network that compresses the input into a latent-space representation. It can be represented by an encoding function h=f(x).
2. Decoder: This part aims to reconstruct the input from the latent space representation. It can be represented by a decoding function r=g(h).

If the only purpose of autoencoders was to copy the input to the output, they would be useless. We hope that by training the autoencoder to copy the input to the output, the latent representation will take on useful properties. This can be achieved by creating constraints on the copying task. If the autoencoder is given too much capacity, it can learn to perform the copying task without extracting any useful information about the distribution of the data. This can also occur if the dimension of the latent representation is the same as the input, and in the overcomplete case, where the dimension of the latent representation is greater than the input. In these cases, even a linear encoder and linear decoder can learn to copy the input to the output without learning anything useful about the data distribution. Ideally, one could train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modeled.

Autoencoders are learned automatically from data examples. It means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input and that it does not require any new engineering, only the appropriate training data. However, autoencoders will do a poor job for image compression. As the autoencoder is trained on a given set of data, it will achieve reasonable compression results on data similar to the training set used but will be poor

general-purpose image compressors. Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder, but are also trained to make the new representation have various nice properties.

Code:

```
from keras.datasets import mnist
from keras.layers import Input, Dense
from keras.models import Model
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

%matplotlib inline
(X_train, _), (X_test, _) = mnist.load_data()

X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255

X_train = X_train.reshape(len(X_train), np.prod(X_train.shape[1:]))
X_test = X_test.reshape(len(X_test), np.prod(X_test.shape[1:]))
print(X_train.shape)
print(X_test.shape)
input_img= Input(shape=(784,))
encoded = Dense(units=32, activation='relu')(input_img)
decoded = Dense(units=784, activation='sigmoid')(encoded)
autoencoder=Model(input_img, decoded)
encoder = Model(input_img, encoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy', metrics=
['accuracy'])
autoencoder.fit(X_train, X_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(X_test, X_test))
encoded_imgs = encoder.predict(X_test)
predicted = autoencoder.predict(X_test)

plt.figure(figsize=(40, 4))
for i in range(10):
    # display original
    ax = plt.subplot(3, 20, i + 1)
    plt.imshow(X_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

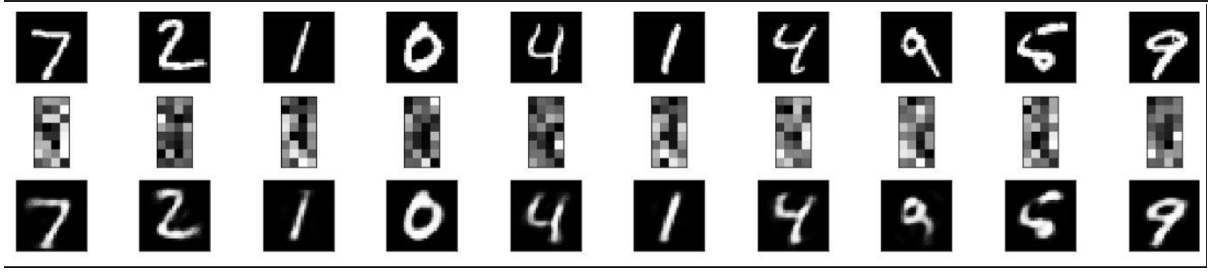
```
    # display encoded image
    ax = plt.subplot(3, 20, i + 1 + 20)
    plt.imshow(encoded_imgs[i].reshape(8,4))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # display reconstruction
    ax = plt.subplot(3, 20, 2*20 +i+ 1)
    plt.imshow(predicted[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

```
plt.show()
```

Results:

```
autoencoder.fit(X_train, X_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(X_test, X_test))
Epoch 40/50
60000/60000 [==============================] - 4s 62us/step - loss: 0.1092 - accuracy: 0.8129 - val_loss: 0.1072 - val_accuracy: 0.8122
Epoch 41/50
60000/60000 [==============================] - 4s 63us/step - loss: 0.1085 - accuracy: 0.8130 - val_loss: 0.1066 - val_accuracy: 0.8122
Epoch 42/50
60000/60000 [==============================] - 4s 60us/step - loss: 0.1078 - accuracy: 0.8131 - val_loss: 0.1059 - val_accuracy: 0.8123
Epoch 43/50
60000/60000 [==============================] - 3s 54us/step - loss: 0.1072 - accuracy: 0.8131 - val_loss: 0.1053 - val_accuracy: 0.8123
Epoch 44/50
60000/60000 [==============================] - 4s 60us/step - loss: 0.1066 - accuracy: 0.8132 - val_loss: 0.1047 - val_accuracy: 0.8124
Epoch 45/50
60000/60000 [==============================] - 3s 57us/step - loss: 0.1061 - accuracy: 0.8132 - val_loss: 0.1042 - val_accuracy: 0.8124
Epoch 46/50
60000/60000 [==============================] - 4s 62us/step - loss: 0.1056 - accuracy: 0.8133 - val_loss: 0.1037 - val_accuracy: 0.8125
Epoch 47/50
60000/60000 [==============================] - 4s 60us/step - loss: 0.1051 - accuracy: 0.8133 - val_loss: 0.1032 - val_accuracy: 0.8125: 0.1
Epoch 48/50
60000/60000 [==============================] - 4s 62us/step - loss: 0.1046 - accuracy: 0.8134 - val_loss: 0.1028 - val_accuracy: 0.8126
Epoch 49/50
60000/60000 [==============================] - 3s 57us/step - loss: 0.1041 - accuracy: 0.8134 - val_loss: 0.1023 - val_accuracy: 0.8125
Epoch 50/50
60000/60000 [==============================] - 3s 57us/step - loss: 0.1037 - accuracy: 0.8135 - val_loss: 0.1019 - val_accuracy: 0.8126

<keras.callbacks.callbacks.History at 0x18a34b15848>
```



Conclusion:

Thus, we have understood how autoencoders work and how to program them