HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL & ELECTRONICS ENGINEERING
DEPARTMENT OF TELECOMMUNICATIONS

MASTER PROGRAM

Optimization Methods and Applications:

VARIATIONAL AUTOENCODER

Student: Thuy Nguyen-Chinh – 1513372

Supervisor: Assoc. Prof., Dr., Kha Ha-Hoang

Ho Chi Minh city, June 15th 2019

Contents

Chapter 1

Theory

1.1 Machine Learning

Machine learning is the scientific study of algorithms and statistical models that computer systems use in order to perform a specific task effectively without using explicit instructions, relying on patterns and inference instead. It is seen as a subset of artificial intelligence. Machine learning algorithms build a mathematical model based on sample data, known as “training data”, in order to make predictions or decisions without being explicitly programmed to perform the task. Machine learning algorithms are used in a wide variety of applications, such as email filtering, and computer vision, where it is infeasible to develop an algorithm of specific instructions for performing the task.

Machine learning tasks are classified into several broad categories. In **supervised learning**, the algorithm builds a mathematical model from a set of data that contains both the inputs and the desired outputs. For example, if the task were determining whether an image contained a certain object, the training data for a supervised learning algorithm would include images with and without that object (the input), and each image would have a label (the output) designating whether it contained the object. In special cases, the input may be only partially available, or restricted to special feedback. Classification algorithms and regression algorithms are types of supervised learning. Classification algorithms are used when the outputs are restricted to a limited set of values. For a classification algorithm that filters emails, the input would be an incoming email, and the output would be the name of the folder in which to file the email. For an algorithm that identifies spam emails, the output would be the prediction of either "spam" or "not spam", represented by the Boolean values true and false. Regression algorithms are named for their continuous outputs, meaning they may have any value within a range. Examples of a continuous value are the temperature, length, or price of an object.

In **unsupervised learning**, the algorithm builds a mathematical model from a set of data which contains only inputs and no desired output labels. Unsupervised

learning algorithms are used to find structure in the data, like grouping or clustering of data points. Unsupervised learning can discover patterns in the data, and can group the inputs into categories, as in feature learning. Dimensionality reduction is the process of reducing the number of "features", or inputs, in a set of data.

Active learning algorithms access the desired outputs (training labels) for a limited set of inputs based on a budget, and optimize the choice of inputs for which it will acquire training labels. When used interactively, these can be presented to a human user for labeling. Reinforcement learning algorithms are given feedback in the form of positive or negative reinforcement in a dynamic environment, and are used in autonomous vehicles or in learning to play a game against a human opponent. Other specialized algorithms in machine learning include topic modeling, where the computer program is given a set of natural language documents and finds other documents that cover similar topics. Machine learning algorithms can be used to find the unobservable probability density function in density estimation problems. Meta learning algorithms learn their own inductive bias based on previous experience. In developmental robotics, robot learning algorithms generate their own sequences of learning experiences, also known as a curriculum, to cumulatively acquire new skills through self-guided exploration and social interaction with humans. These robots use guidance mechanisms such as active learning, maturation, motor synergies, and imitation.

1.2 Unsupervised Learning

Unsupervised learning is a type of self-organized learning that helps find previously unknown patterns in dataset without pre-existing labels. It is also known as self-organization and allows modeling probability densities of given inputs. It is one of the main three categories of machine learning, along with supervised and active learning. Semi-supervised learning has also been described, and is a hybridization of supervised and unsupervised techniques.

A central application of unsupervised learning is in the field of density estimation in statistics, though unsupervised learning encompasses many other domains involving summarizing and explaining data features. It could be contrasted with supervised learning by saying that whereas supervised learning intends to infer a conditional probability distribution $p_X(x \mid y)$ conditioned on the label y of input data, while, unsupervised learning intends to infer an a prior probability distribution $p_X(x)$.

Autoencoder is a type of unsupervised learning used to learn efficient data codings. 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. Recently, the autoencoder concept has become more widely used for learning generative models of data. Some of the

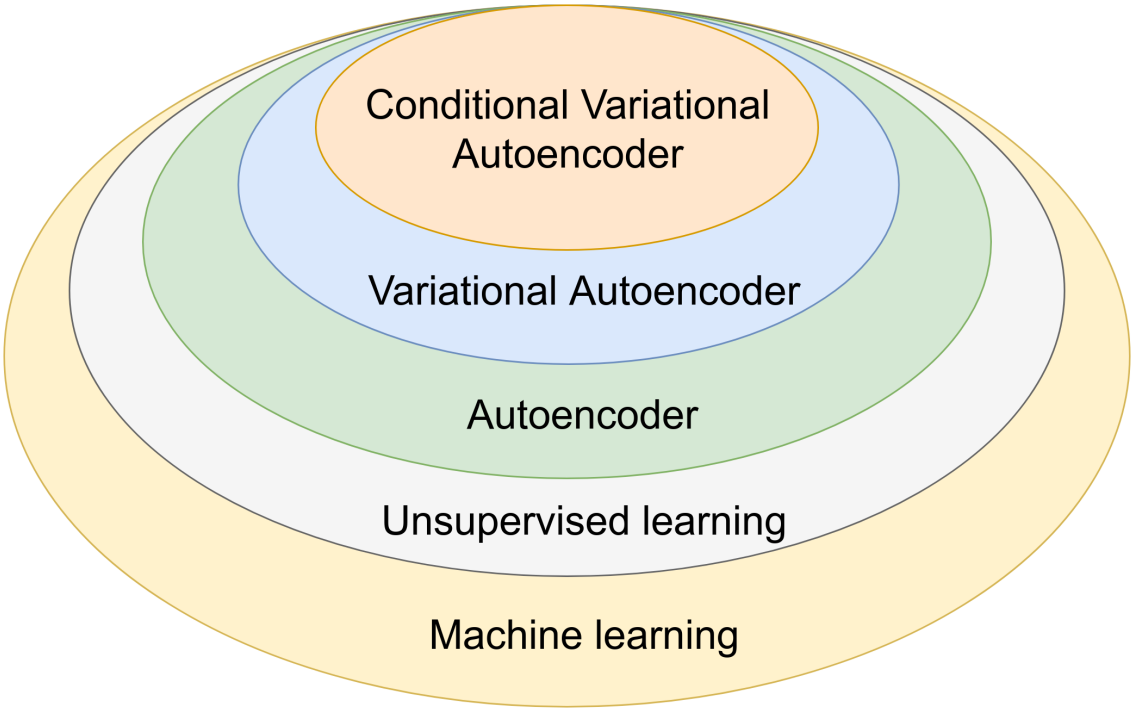


Figure 1.2.1: Hirerachy of Autoencoders within Machine Learning

most powerful AIs in the 2010s involved sparse autoencoders stacked inside of deep neural networks. The hirerachy of autoencoders is shown in Fig. 1.2.1.

1.3 Autoencoder

An autoencoder learns to compress data from the input layer into a short code, and then uncompress that code into something that closely matches the original data. This forces the autoencoder to engage in dimensionality reduction, for example by learning how to ignore noise. Some architectures use stacked sparse autoencoder layers for image recognition. The first encoding layer might learn to encode basic features such as corners, the second to analyze the first layer's output and then encode less local features like the tip of a nose, the third might encode a whole nose, etc., until the final encoding layer encodes the whole image into a code that matches (for example) the concept of "cat". The decoding layers learn to decode the representation back into its original form as closely as possible. An alternative use is as a generative model: for example, if a system is manually fed the codes it has learned for "cat" and "flying", it may attempt to generate an image of a flying cat, even if it has never seen a flying cat before.

The general atchitecture of autoencoders is visualized in Fig. 2.3.1. Given an input vector, encoder is to compress the input vector in high dimension into a lower dimension. The compressed vector is also called the hidden representation.. Then, a decoder performs the decoding to map the hidden representation in the

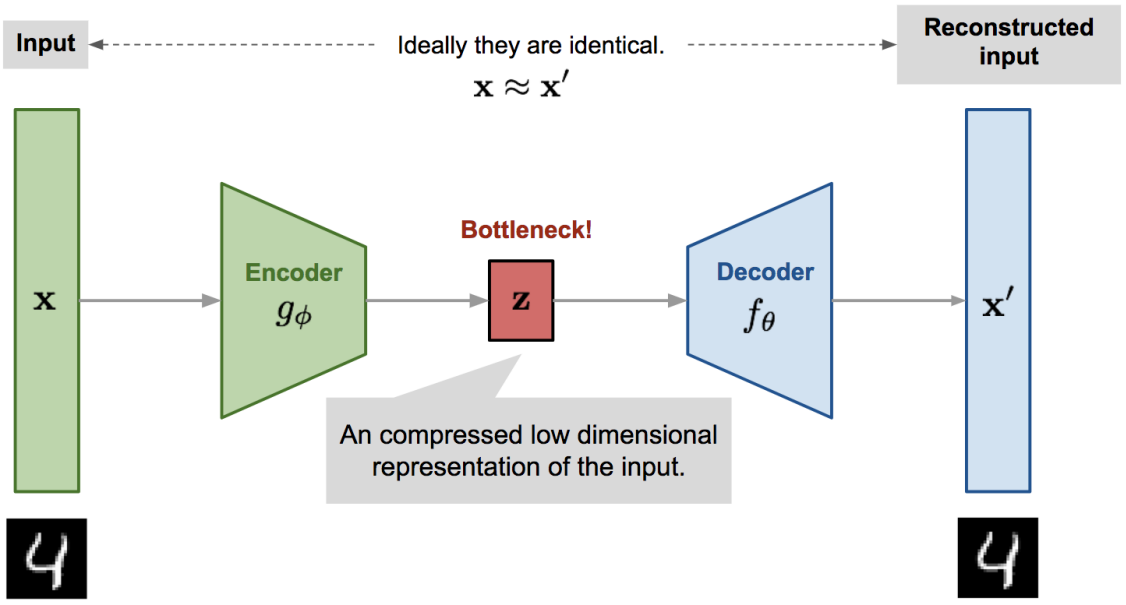


Figure 1.3.1: General architecture of autoencoders. The picture is taken from *https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html*

low-dimensional space into the original high-dimensional space as the input vector.

Formally, an input sample is drawn from a distribution $x \sim X$. The encoder in the autoencoder is represented by a function $f : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$, consequently, the hidden representation is $z = f(x)$. Then, the decoder is a function $g : \mathbb{R}^{d_2} \to \mathbb{R}^{d_1}$ to map the hidden representation to the original domain $\tilde{x} = g(z)$. To train the autoencoder, the loss function is the well-known mean square error:

$$L = \sum_{i=1}^{N} MSE^{(i)} = \sum_{i=1}^{N} \left(x^{(i)} - \tilde{x}^{(i)}\right)^2 \tag{1.1}$$

where the index (i) indicates the ith sample in the training set, which has totally N samples. This loss function is also called the reconstruction loss because the loss function computes the difference between the constructed output with the corresponding input.

One of considerations when designing an autoencoder is choosing a dimension (d_2) for the hidden space. Actually, we expect the hidden dimension as small as possible for the compression purpose. However, if the hidden space is too low-dimensional, the decoder seems hard to reconstruct the compressed information to the original input. Therefore, there is a trade-off between the compression rate and the reconstruction capability of the autoencoder.

Besides, there is a more issue with the hidden space of generic autoencoders. Let's denote $p(\tilde{x})$ as the probability of the reconstructed output $\tilde{x}$ to be realistic. In other word, this is the probability that the decoder reconstructs $\tilde{x}$ so that $\tilde{x} \in X$, where

X is the distribution of input samples. As expectation, the higer value of $p(\tilde{x})$, the better performance of the autoencoder. This expectation can be represented as an optimization problem:

$$\begin{aligned} &maximize \quad p(\tilde{x}) \\ &subject\ to \quad z \in \mathbb{R}^{d_2} \end{aligned} \tag{1.2}$$

Furthermore, we can express $p(\tilde{x})$ as the following:

$$p(\tilde{x}) = \int_{\mathbb{R}^{d_2}} p(\tilde{x}|z)p(z)dz \tag{1.3}$$

The integral can be approximated by a sum if we can sample a large number M of samples of z in the hidden space:

$$p(\tilde{x}) \simeq \sum_{i=1}^{M} p(\tilde{x}|z^{(i)})p(z^{(i)}) \tag{1.4}$$

However, during the training phase of the autoencoder, there is no constraint on the hidden space, hence, z could be randomly any where in the hidden space $\mathbb{R}^{d_2}$. This implies that there are a lot of values of z in the hidden space, which yield $p(\tilde{x}|z^{(i)}) = 0$. As a result, the number of samples is required to be infinity ($M \to \inf$) to cover the whole hidden space in order to ensure the approxiation of the sum over the integral. This is not effective because the computation would be significantly giant. Therefore, we need a wiser approach to address this issue.

1.4 Variational Autoencoder

Variational Autoencoder is a type of autoencoder family, which is theoratically designed based on an inspiration of the mentioned issue in the previous section.

Firstly, let's review an important metric in the information theory: Kullback–Leibler divergence. In mathematical statistics, the Kullback–Leibler divergence (also called relative entropy) is a measure of how one probability distribution is different from a second, reference probability distribution. For discrete probability distributions P and Q defined on the same probability space X, the Kullback–Leibler divergence between P and Q is defined to be:

$$D_{KL}\big[P(x) \mid\mid Q(x)\big] = \sum_{x \in X} P(x) log\left(\frac{P(x)}{Q(x)}\right) \tag{1.5}$$

It is worth mentioning that the notation log represents for ln. This notation is widely used by researchers in the field, so I also adopt this notation in this document.

Regarding the issue which is mentioned in the previous section, variational autoencoder is to address it. Instead of sampling a huge number of z in the hidden space to

calculate $p(\tilde{x})$, we can sample z in another space Q where almost $z \in Q$ yields $p(\tilde{x}|z) \neq 0$. By using the above definition of Kullback–Leibler divergence, we can derive:

$$D_{KL}\big[Q(z) \mid\mid P(z|\tilde{x})\big] = E_{z \in Q}\big[logQ(z) - logP(z|\tilde{x})\big] \tag{1.6}$$

By substituting $P(z|\tilde{x}) = P(\tilde{x}|z)P(z)/P(\tilde{x})$ into equation 1.6, we obtain:

$$D_{KL}\big[Q(z) \mid\mid P(z|\tilde{x})\big] = E_{z \in Q}\big[logQ(z) - logP(\tilde{x}|z) - logP(z) + logP(\tilde{x})\big] \tag{1.7}$$

Then, by rearranging equation 1.7, we have:

$$\begin{aligned} logP(\tilde{x}) - D_{KL}\big[Q(z) \mid\mid P(z|\tilde{x})\big] &= E_{z \in Q}\big[- logQ(z) + logP(\tilde{x}|z) + logP(z)\big] \\ &= E_{z \in Q}\big[logP(\tilde{x}|z)\big] - D_{KL}\big[Q(z) \mid\mid P(z)\big] \end{aligned} \tag{1.8}$$

Because the Kullback–Leibler divergence is always non-negative, therefore, instead of optimize $p(\tilde{x})$, we can tolerate the strictness by optimizing its lower bound $logP(\tilde{x}) - D_{KL}\big[Q(z) \mid\mid P(z|\tilde{x})\big]$. Subsequently, the optimization problem is now reduced to:

$$\begin{aligned} &maximize \quad E_z\big[logP(\tilde{x}|z)\big] - D_{KL}\big[Q(z) \mid\mid P(z)\big] \\ &subject\ to \quad z \in Q \end{aligned} \tag{1.9}$$

From this formula, we can model terms by designing their corresponding neural networks. Firstly, $Q(z)$ can be assigned to be the distribution of the hidden vector, which is compressed by the encoder given an input, in which, most of samples yield $p(\tilde{x}|z) \neq 0$. Secondly, $P(z)$ is a distribution that $Q(z)$ need to be "close" to (to maximize the objective function of the optimization problem in equation 1.9). For convinience, $P(z)$ is chosen to be the normal distribution ($P(z) = \mathcal{N}(z)$) for inheritting nice properties of this distribution. Lastly, $logP(\tilde{x}|z)$ can be modelled by the decoder, given a hidden vector, the decoder reconstructs an output to be identical with the input of the encoder. Overall, variational autoencoder is the same as generic autoencoder, excepting the hidden space is forced to have the normal distribution.

Regarding the loss funtion for training the variational autoencoder, it can be inferred from the objective function of the optimization statement. The first term in the objective function corresponds to the reconstruction loss (the same as autoencoder). Meanwhile, the second term is the Kullback–Leibler divergence distance between two distributions, consequently, Kullback–Leibler divergence loss is an appropriate choice. In sum, the total loss is the combination of the two losses:

$$L^{(i)} = L^{(i)}_{recons} + \lambda L^{(i)}_{KL} = \left(x^{(i)} - \tilde{x}^{(i)}\right)^2 + \lambda \sum_{j=1}^{d_2} 0.5\left(\mu_j^2 + \sigma_j^2 - 2log\sigma_j - 1\right) \tag{1.10}$$

where λ is the weight of the KL loss, μ_j and σ_j are the j-th dimension of mean and standard deviation of the hidden representation $z^{(i)}$, respectively. To obtain the

KL-term of the loss function in equation 1.10, let's assume we have to compute the KL loss of the two Gaussian distributions, namely $P(\mu_1, \sigma_1^2)$ and $Q(\mu_2, \sigma_2^2)$:

$$\begin{aligned} D_{KL}[P \mid\mid Q] &= -\int P(x) log Q(x) dx + \int P(x) log P(x) dx \\ &= \frac{1}{2} log(2\pi\sigma_2^2) + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \end{aligned} \tag{1.11}$$

If $Q(\mu_2, \sigma_2^2)$ is the normal distribution $\mathcal{N}$, we obtain the result:

$$D_{KL}[P \mid\mid \mathcal{N}] = \frac{1}{2}\big(\mu_1^2 + \sigma_1^2 - 2log\sigma_1 - 1\big) \tag{1.12}$$

1.5 Conditional Variational Autoencoder

The purpose of conditional variational autoencoder is to utilize the label of input so that we can control the label of generated outputs. Let's denote y as the label of a sample x. Firstly, the below term is computed:

$$\begin{aligned} &D_{KL}\big[Q(z|x,y) \mid\mid P(z|x,y)\big] + E_{Q(z|x,y)}\big[-logQ(z|x,y) + logP(x,z|y)\big] \\ &= E_{Q(z|x,y)}\big[logQ(z|x,y) - logP(z|x,y)\big] + E_{Q(z|x,y)}\big[-logQ(z|x,y) + logP(x,z|y)\big] \\ &= E_{Q(z|x,y)}\big[-logP(z|x,y) + logP(x,z|y)\big] \\ &= logP(x|y) \end{aligned} \tag{1.13}$$

From this, the lower bound of the $logP(x|y)$ can be obtained:

$$\begin{aligned} logP(x|y) &\geq E_{Q(z|x,y)}\big[-logQ(z|x,y) + logP(x,z|y)\big] \\ &= E_{Q(z|x,y)}\big[-logQ(z|x,y) + logP(z|y) + logP(x|z,y)\big] \\ &= -D_{KL}\big[Q(z|x,y) \mid\mid P(z|y)\big] + E_{Q(z|x,y)}\big[logP(x|z,y)\big] \end{aligned} \tag{1.14}$$

Terms in equation 1.14 can be modelled as the following. Firstly, $Q(z|x,y)$ can be represented by the encoder, which encodes the information from both the input and its label. Secondly, $P(z|y)$ can be chosen to be a normal distribution in the hidden space. Besides, the hidden representation corresponds to a specific label. Lastly, $P(x|z,y)$ is modelled by the decoder, which reconstruct the output from the information concatenated from both the hidden representation and the input label.

Chapter 2

Applications

In this chapter, applications of autoencoders are performed in order to verify the above theoretical analysis. Firstly, section 2.1 is problemt statement to choose an application of autoencoder. Secondly, section 2.2 describes the dataset for training the neural network. Then, the whole implementation is shown in section 2.3. Finally, experimental results are reported in section 2.4.

2.1 Problem statement

One of the preliminary purposes of autoencoder is compressing information into a lower-dimensional space for storage or anomaly detection. However, in this document, that application is not preferred. Instead, **image generation** is a new interesting application that has recently attracted attetion of research community significantly. Therefore, image generation is chosen for practising with autoencoders in this report.

More specifically, image generation is to generate image given a random input vector. In the field of machine learning and deep learning, a neural network is utilized to perform this task. It is worth mentioning that image generation is constrainted by the training dataset. Concretely, if the training set contains car images, the trained neural network is just capable to generate car images, meanwhile, bike images, which are not included in the training set, would not be generated by the trained network. So, the image generation process, implemented by autoencoders, consists of three stages. Firstly, input images are fed into the encoder, outputing hidden vectors. Then, hidden vectors are decoded by the decoder in order to reconstruct input images. Lastly, the reconstruction loss is applied to measure the distance between the input and the reconstructed output.

Once being trained, the neural network is able to generate images drawn from the learnt distrubution. To generate an image, firstly, a vector is sampled in the hidden space, then the sampled vector is fed into the decoder to generate a new image, which is similar to images in the training set. It is worth noting that in the inference process (after training), the encoder is unnecessary and ignored.

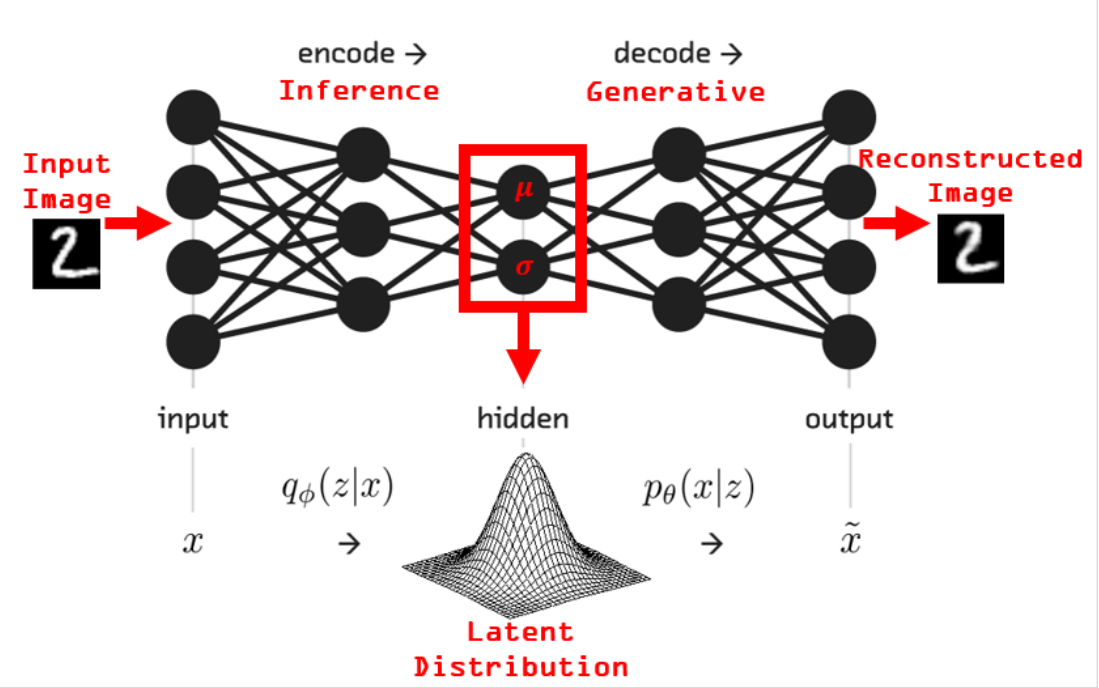


Figure 2.1.1: Image generation implemented by the autoencoder. The image is taken from `https://sergioskar.github.io/Autoencoder`

2.2 Dataset

The MNIST dataset [6] is a large database of handwritten digits that is commonly used for training various image processing systems. The dataset is also widely used for training and testing in the field of machine learning. It contains 60,000 training images and 10,000 testing images. In which, each sample is a grayscale image of size 28x28. There are ten categories corresponding to ten number characters (from 0 to 9). Fig. 2.2.1 shows some examples in the MNIST dataset.

2.3 Implementation

In this section, implementations of generic autoencoder (AE), variational autoencoder (VAE), and conditional variational autoencoder (CVAE) are described in subsection 2.3.1, 2.3.2, and 2.3.3, respectively.

2.3.1 Generic autoencoder

To train a neural network to learn the image generation, I setup an autoencoder with 9 layers, which can be seen in Fig. 2.3.1. Images for training the neural network has size 28x28=784, therefore, the input size of the neural network is also 784. Then, subsequent layers in the encoder of the autoencoder is progressively decreased in the number of neurons until reach the size 100, which is also the dimension of the

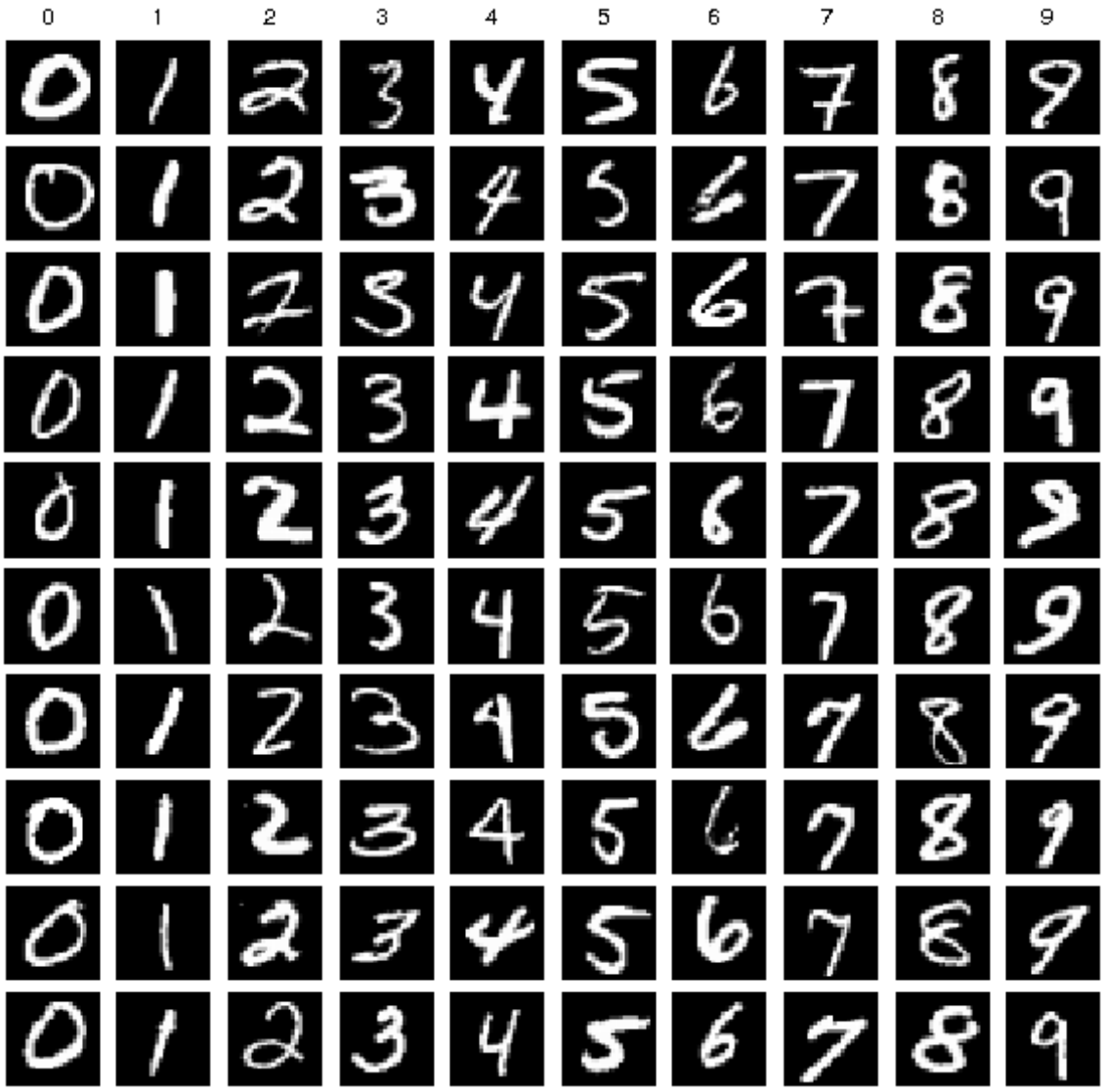


Figure 2.2.1: Some examples of the MNIST dataset with ten categories corresponding to ten number characters.

hidden representation. Afterwards, the decoder is symetric to the encoder in order to reconstruct the encoded feature to the output of size 784. Activation is Leaky Relu with the negative slope of 0.1. The implementation code of the designed autoencoder is shown below:

```
class AE(nn.Module):
   def __init__(self, in_dims=784, hid_dims=100, negative_slope=0.1):
      super(AE, self).__init__()
      # Encoder
      self.encoder = nn.Sequential(OrderedDict([
         ('layer1', nn.Linear(in_dims, 512)),
         ('relu1', nn.LeakyReLU(negative_slope=negative_slope,
```

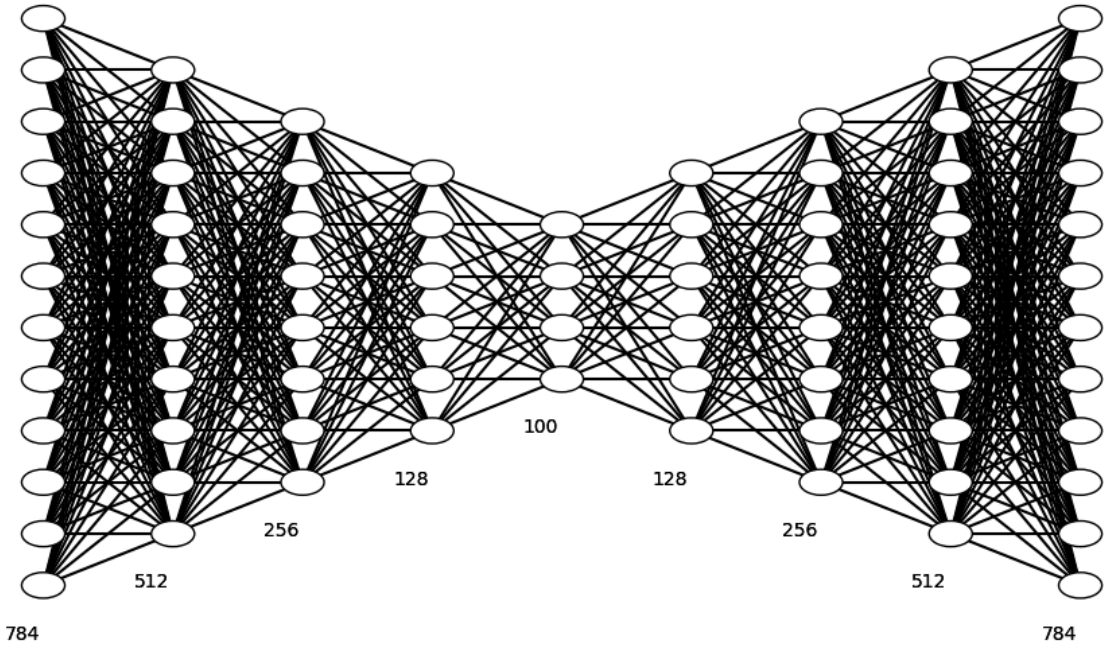


Figure 2.3.1: Architecture of the autoencoder for generating hand-written-digit images

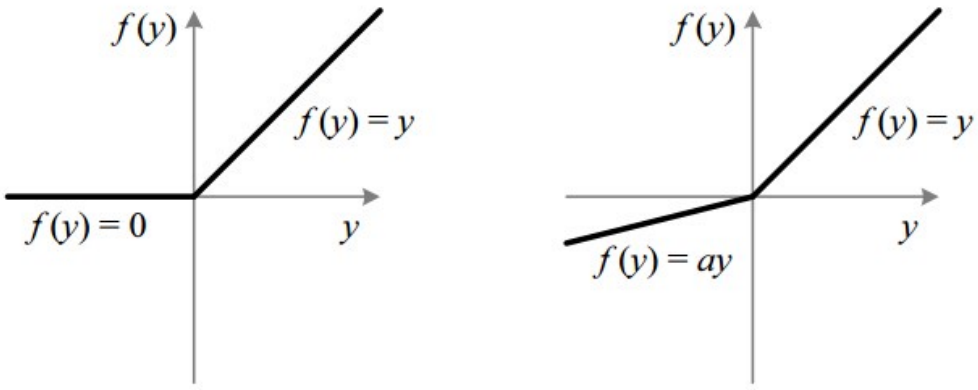


Figure 2.3.2: The left image is the ReLU, while the right image is the Leaky ReLU, which has a negative side with a predefined slope.

```
    inplace=True)),
('layer2', nn.Linear(512, 256)),
('relu2', nn.LeakyReLU(negative_slope=negative_slope,
    inplace=True)),
('layer3', nn.Linear(256, 128)),
('relu3', nn.LeakyReLU(negative_slope=negative_slope,
    inplace=True)),
('layer4', nn.Linear(128, hid_dims)),
```

```
            ('relu4', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
        ]))
        # Decoder
        self.decoder = nn.Sequential(OrderedDict([
            ('layer1', nn.Linear(hid_dims, 128)),
            ('relu1', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer2', nn.Linear(128, 256)),
            ('relu2', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer3', nn.Linear(256, 512)),
            ('relu3', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer4', nn.Linear(512, in_dims)),
            ('sigmoid', nn.Sigmoid()),
        ]))
        self._init_weights()

    def forward(self, x):
        z = self.encoder(x)
        y = self.decoder(z)
        return y

    def _init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                m.weight.data.normal_(0, 0.01)
                m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
```

2.3.2 Variational autoencoder

The variational autoencoder architecture is same as the autoencoder in Fig. 2.3.1, excepting in the middle (hidden representation), there are two additional elements, e.g., two fully connected layers for computing mean and variance for the encoded vector. The implementation code for the variational autoencoder is shown as the following:

```
class VAE(nn.Module):
    def __init__(self, in_dims=784, hid_dims=100, negative_slope=0.1):
        super(VAE, self).__init__()
        # Encoder
        self.encoder = nn.Sequential(OrderedDict([
```

```
            ('layer1', nn.Linear(in_dims, 512)),
            ('relu1', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer2', nn.Linear(512, 256)),
            ('relu2', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer3', nn.Linear(256, 128)),
            ('relu3', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
        ]))
        self.fc_mu = nn.Linear(128, hid_dims)
        self.fc_var = nn.Linear(128, hid_dims)
        # Decoder
        self.decoder = nn.Sequential(OrderedDict([
            ('layer1', nn.Linear(hid_dims, 128)),
            ('relu1', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer2', nn.Linear(128, 256)),
            ('relu2', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer3', nn.Linear(256, 512)),
            ('relu3', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer4', nn.Linear(512, in_dims)),
            ('sigmoid', nn.Sigmoid()),
        ]))
        self._init_weights()

    def forward(self, x):
        if self.training:
            h = self.encoder(x)
            mu, logvar = self.fc_mu(h), self.fc_var(h)
            z = self._reparameterize(mu, logvar)
            y = self.decoder(z)
            return y, mu, logvar
        else:
            z = self.represent(x)
            y = self.decoder(z)
            return y

    def represent(self, x):
        h = self.encoder(x)
        mu, logvar = self.fc_mu(h), self.fc_var(h)
        z = self._reparameterize(mu, logvar)
        return z

    def _reparameterize(self, mu, logvar):
```

```
      std = logvar.mul(0.5).exp_()
      esp = torch.randn(*mu.size()).type_as(mu)
      z = mu + std * esp
      return z

   def _init_weights(self):
      for m in self.modules():
         if isinstance(m, nn.Linear):
            m.weight.data.normal_(0, 0.01)
            m.bias.data.zero_()
         elif isinstance(m, nn.BatchNorm2d):
            m.weight.data.fill_(1)
            m.bias.data.zero_()
```

2.3.3 Conditional variational autoencoder

In term of conditional variational autoencoder, the architecture is identical to the two above autoencoder for fair comparison. However, the conditional variational autoencoder has one more element, named Conditioner alongside with Encoder and Decoder. The conditioner is to encode the one-hot label input into an unconstrained hidden space. Then, the encoded label is concatenated with the encoded input to form a new input fed into the decoder. By this method, the reconstructed/generated output is formed based on the information from both input and label so that the label of the output can be controlled. The implementation code of the conditional variational autoencoder is listed below:

```
class CVAE(nn.Module):
   def __init__(self, in_dims=784, hid1_dims=100, hid2_dims=64,
       num_classes=10, negative_slope=0.1):
      super(CVAE, self).__init__()
      self.in_dims = in_dims
      self.hid1_dims = hid1_dims
      self.hid2_dims = hid2_dims
      self.num_classes = num_classes
      self.negative_slope = negative_slope

      # Encoder
      self.encoder = nn.Sequential(OrderedDict([
         ('layer1', nn.Linear(in_dims, 512)),
         ('relu1', nn.LeakyReLU(negative_slope=negative_slope,
            inplace=True)),
         ('layer2', nn.Linear(512, 256)),
         ('relu2', nn.LeakyReLU(negative_slope=negative_slope,
            inplace=True)),
         ('layer3', nn.Linear(256, 128)),
         ('relu3', nn.LeakyReLU(negative_slope=negative_slope,
```

```
            inplace=True)),
        ]))
        self.fc_mu = nn.Linear(128, hid1_dims)
        self.fc_var = nn.Linear(128, hid1_dims)

        # Conditioner
        self.conditioner = nn.Sequential(OrderedDict([
            ('layer1', nn.Linear(num_classes, 16)),
            ('relu1', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer2', nn.Linear(16, 32)),
            ('relu2', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer3', nn.Linear(32, hid2_dims)),
            ('relu3', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
        ]))

        # Decoder
        self.decoder = nn.Sequential(OrderedDict([
            ('layer1', nn.Linear(hid1_dims+hid2_dims, 128)),
            ('relu1', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer2', nn.Linear(128, 256)),
            ('relu2', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer3', nn.Linear(256, 512)),
            ('relu3', nn.LeakyReLU(negative_slope=negative_slope,
                inplace=True)),
            ('layer4', nn.Linear(512, in_dims)),
            ('sigmoid', nn.Sigmoid()),
        ]))

        self._init_weights()

    def forward(self, x, y):
        if self.training:
            # Encode input
            h = self.encoder(x)
            mu, logvar = self.fc_mu(h), self.fc_var(h)
            hx = self._reparameterize(mu, logvar)
            # Encode label
            y_onehot = self._onehot(y)
            hy = self.conditioner(y_onehot)
            # Hidden representation
            h = torch.cat([hx, hy], dim=1)
            # Decode
```

```
            y = self.decoder(h)
            return y, mu, logvar
        else:
            hx = self._represent(x)
            hy = self.conditioner(self._onehot(y))
            h = torch.cat([hx, hy], dim=1)
            y = self.decoder(h)
            return y

    def generate(self, y):
        hy = self.conditioner(self._onehot(y))
        hx = self._sample(y.shape[0]).type_as(hy)
        h = torch.cat([hx, hy], dim=1)
        y = self.decoder(h)
        return y

    def _represent(self, x):
        h = self.encoder(x)
        mu, logvar = self.fc_mu(h), self.fc_var(h)
        hx = self._reparameterize(mu, logvar)
        return hx

    def _reparameterize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()
        esp = torch.randn(*mu.size()).type_as(mu)
        z = mu + std * esp
        return z

    def _onehot(self, y):
        y_onehot = torch.FloatTensor(y.shape[0], self.num_classes)
        y_onehot.zero_()
        y_onehot.scatter_(1, y, 1)
        return y_onehot

    def _sample(self, num_samples):
        return torch.FloatTensor(num_samples, self.hid1_dims).normal_()

    def _init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                m.weight.data.normal_(0, 0.01)
                m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
```

2.4 Experimental results

In this section, experiments are conducted to demonstrate ability of types of autoencoders on the task of image generation.

2.4.1 Generic autoencoder

Regarding the autoencoder, there are two experiments. Firstly, given an image, the autoencoder encode the image and then decode the encoded vector to form a reconstructed output. Secondly, to prove that by randomly sampling a point in the hidden space, the generated output is likely to be different from the input distribution.

The first experimental result is revealed in Fig. 2.4.1. The results indicate that the generic autoencoder can reconstruct images, which are quite similar to the input. However, the outputs have a little difference from the input as well as they are blurred.

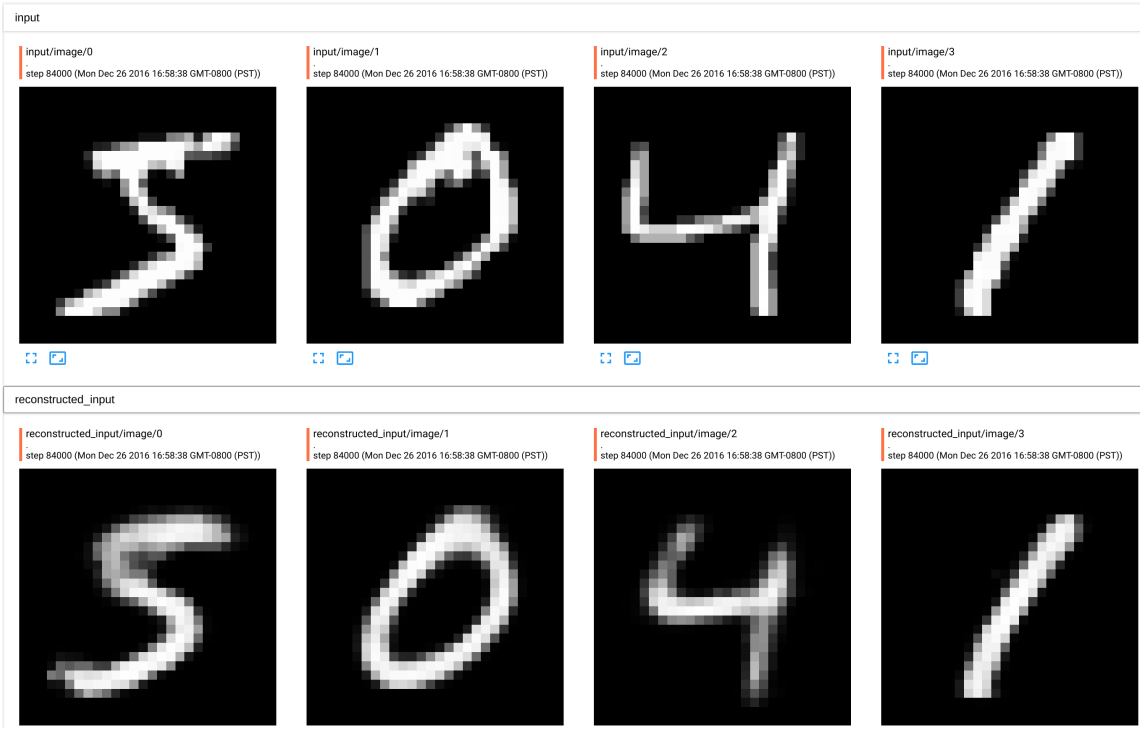


Figure 2.4.1: Some examples of reconstructed images by the autoencoder. The first line is input images, and the second line is reconstructed images.

Then, the second experimental result is illustrated in Fig. 2.4.2. As we can see that, the generated output of the autoencoder is totally different from samples that we use to train the neural network. This proves the theoretical analysis about the shortcoming the the unconstrained hidden space.

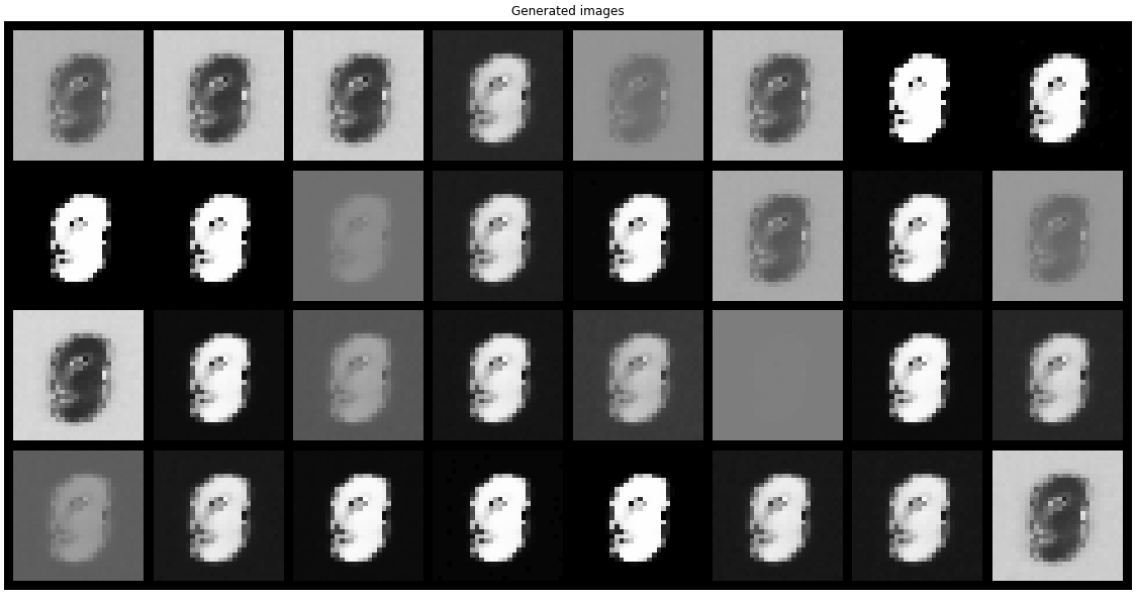


Figure 2.4.2: Some examples of generated images by the autoencoder.

2.4.2 Variational autoencoder

In this subsection, experiments for the variational autoencoder is the same as the generic autoencoder. Firstly, reconstructed images are shown in Fig. 2.4.4. Comparing to results from the generic autoencoder, we can see that the reconstruction of variational autoencoder is better than its counterpart. More specifically, the outputs of variational autoencoder are not blurred as in cases of the autoencoder.

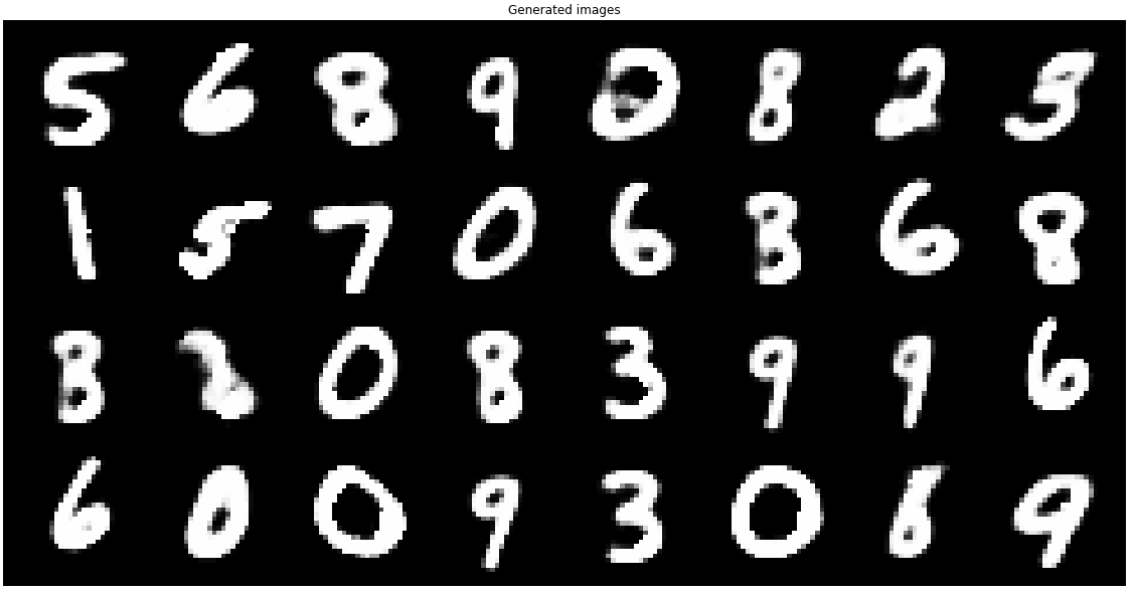


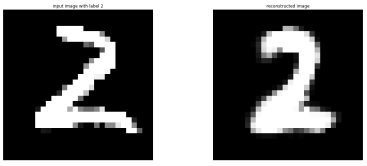
Figure 2.4.3: Some examples of generated images by the variational autoencoder.

Secondly, generated images of the variational autoencoder are illustrated in Fig.

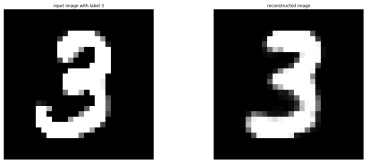
2.4.3. Those images are more realistic than ones of the generic autoencoder. This indicates that the constrained hidden space works that results in realisic images in the output.

(a) Pair of 0s

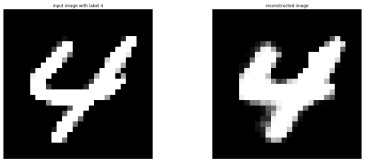
(b) Pair of 1s



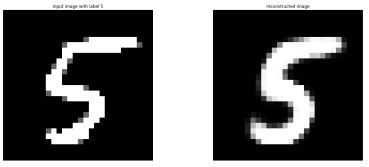
(c) Pair of 2s



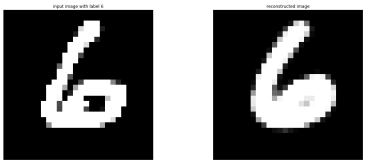
(d) Pair of 3s



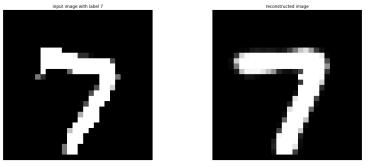
(e) Pair of 4s



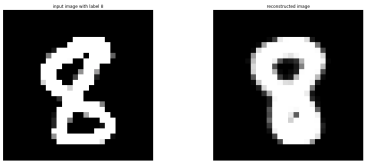
(f) Pair of 5s



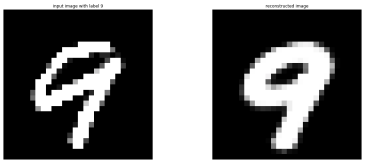
(g) Pair of 6s



(h) Pair of 7s



(i) Pair of 8s

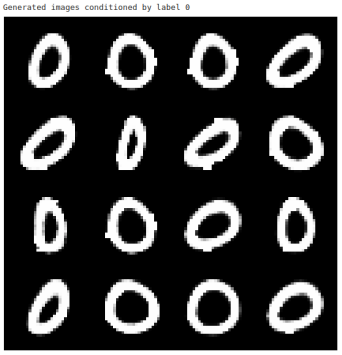


(j) Pair of 9s

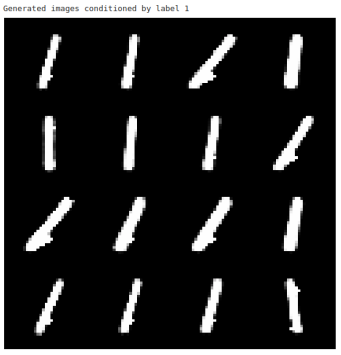
Figure 2.4.4: Some examples of pairs of input and reconstructed images by the variational autoencoder.

2.4.3 Conditional Variational autoencoder

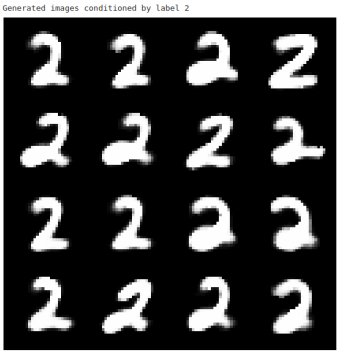
In this subsection, the experiment is focused only on the task of image generation conditioned by label. Fig. 2.4.5 shows examples to demonstrate that the Conditional Variational Autoencoder is capable to generate images with labelling control.



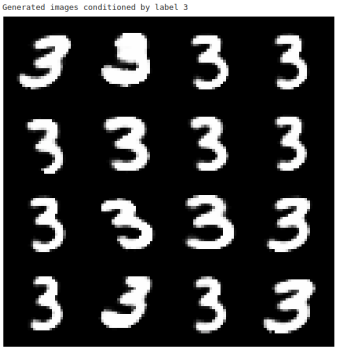
(a) Generation of 0s



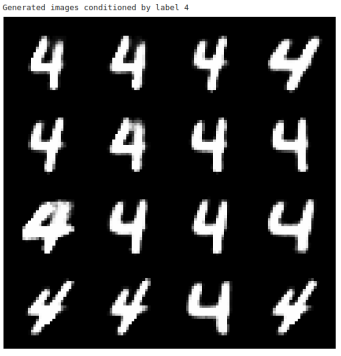
(b) Generation of 1s



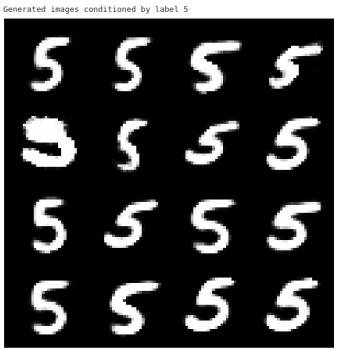
(c) Generation of 2s



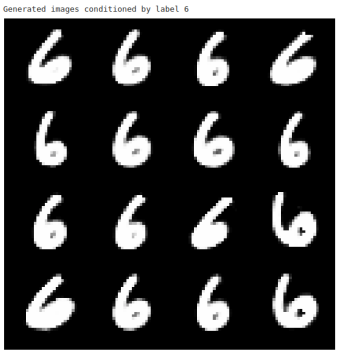
(d) Generation of 3s



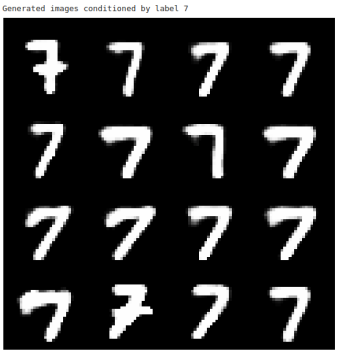
(e) Generation of 4s



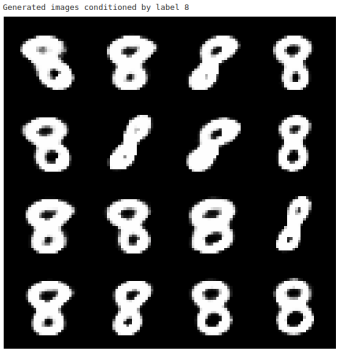
(f) Generation of 5s



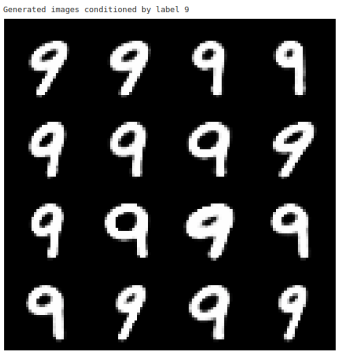
(g) Generation of 6s



(h) Generation of 7s



(i) Generation of 8s



(j) Generation of 9s

Figure 2.4.5: Some examples of generated images conditioned by labels, using the conditional variational autoencoder.

Chapter 3

Conlusion and Future work

In this final assignment, I have explored a specific field of Unsupervised learning (a learning method in Machine Learning): Variational Autoencoder. Actually, Variational Autoencoder is a branch of Autoencoder, which is typically used for compressing information and estimating distribution density. However, the hidden space of the generic autoencoder is unconstrained, consequently, sampling a random vector in this hidden space would result in a non-realistic reconstructed output.

Variational autoencoder is theoratically designed in order to overcome the mentioned shortcoming of the generic autoencoder. More specifically, instead of optimizing the realistic probability of reconstructed samples, its lower bound is figured out and selected to be optimized. Besides, the hidden space is forced to be the normal distribution by using an additional Kullback-Leibler divergence loss term. As the result, when sampling a random point in the hidden space, the reconstructed output is similar to samples from the training dataset.

Regarding the application, image generation is chosen to demonstrate he ability of the variational autoencoder. Experimental results indicate that the variational autoencoder is capable to generate images (hand-written digits in this case), which are used to trained the neural network. Nevertheless, the generation is arbitrary, in other word, people can not control which digit label is generated. This is a drawback of the variational autoencoder.

This issue of the variational autoencoder can be solved by its off-spring, named Conditional Variational Autoencoder. By re-formulating transformations inferred from the theoretical analysis of the variational autoencoder, input label information is utilized to embed the labelling control signal to the neural network so that it can generate images conditioned by the label of input.

In this work, experiments are conducted on a small dataset (MNIST). Therefore, in the future, larger datasets, for example, SHVN, WIDEN, CIFAR10, and CIFAR100, will be examined to verify the ability of types of the autoencoder.

Bibliography

[1] Wikipedia, *"Machine Learning"*, https://en.wikipedia.org/wiki/Machine_learning

[2] Wikipedia, *"Unsupervised Learning"*, https://en.wikipedia.org/wiki/Unsupervised_learning

[3] Wikipedia, *"Autoencoder"*, https://en.wikipedia.org/wiki/Autoencoder

[4] Diederik P Kingma, Max Welling *"Auto-Encoding Variational Bayes"*, https://arxiv.org/abs/1312.6114

[5] Raymond Yeh, Junting Lou, Teck-Yian Lim *"CS598LAZ - Variational Autoencoders"*, http://slazebni.cs.illinois.edu/spring17/lec12_vae.pdf

[6] LeCun, Yann; Corinna Cortes; Christopher J.C. Burges, *"MNIST handwritten digit database"*, Retrieved 17 August 2013.

[7] Sohn, Kihyuk, Honglak Lee, and Xinchen Yan, *"Learning Structured Output Representation using Deep Conditional Generative Models"*, Advances in Neural Information Processing Systems, 2015.

[8] Agustinus Kristiadi, *"Conditional Variational Autoencoder: Intuition and Implementation"*, https://wiseodd.github.io/techblog/2016/12/17/conditional-vae