Assignment 4 - Variational Autoencoders

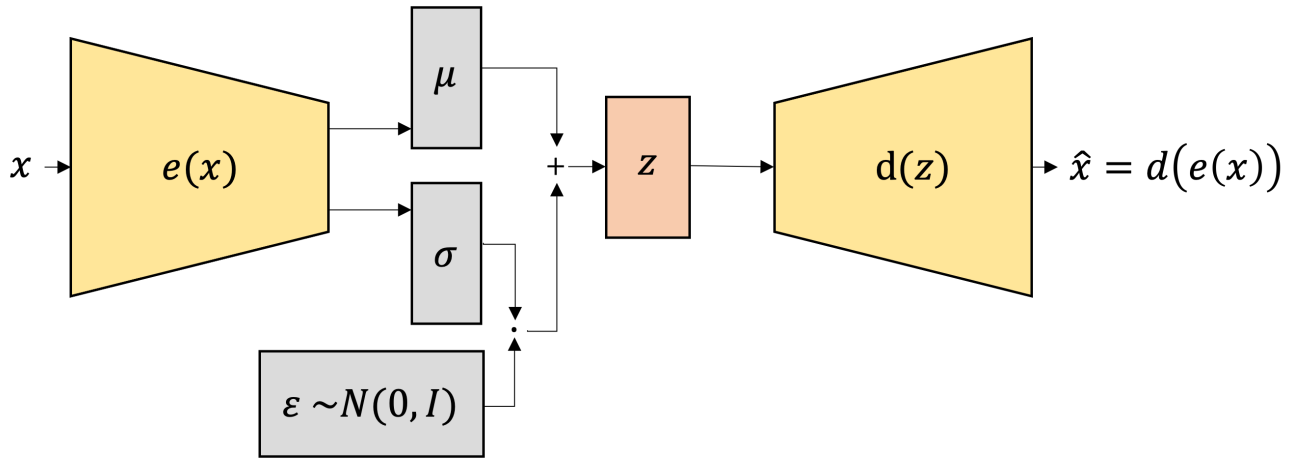
In this assignment, we will train a model to produce new human faces with variational autoencoders (VAEs). Variational autoencoders let us design complex generative models of data, and fit them to large datasets. They can generate images of fictional celebrity faces (as we'll do in this assignment), high-resolution digital artwork and many more tasks. These models also yield state-of-the-art machine learning results in image generation and reinforcement learning. Variational autoencoders (VAEs) were defined in 2013 by Kingma and Wellings [1].

In this assignment, you will build, train and analyze a VAE with the CelebA dataset. You will analyze how well images can be reconstructed from the lower dimensional representations and try to generate images that look similar to the images in the CelebA dataset.

[1] Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." arXiv preprint arXiv:1312.6114 (2013).

Section 1: Variational Autoencoders

Let us recall the structure of the variational autoencoder:



Imports

Before we begin, we import the needed libraries.

You may modify the starter code as you see fit, including changing the signatures of functions and adding/removing helper functions. However, please make sure that we can understand what you are doing and why.

In [1]:

```
from torchvision import datasets
from torchvision import transforms
from torch.autograd import Variable
import torch
import torch.nn as nn
import torch.nn.functional as nnF
from torchvision.utils import make_grid
from IPython.display import Image
import matplotlib.pyplot as plt
import numpy as np
import random
import torchvision.transforms.functional as F
from torchvision.utils import make_grid
import os
import zipfile

%pip install wget
import wget

# use GPU for computation if possible: Go to RUNTIME -> CHANGE RUNTIME TYPE -> GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/col
ab-wheels/public/simple/
Collecting wget
  Downloading wget-3.2.zip (10 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9674 sha
256=7e7e8218ffa0825e8c9219e3be99b6a7c4f61bedc53ba2ffa0ab36fe3845620c
  Stored in directory: /root/.cache/pip/wheels/bd/a8/c3/3cf2c14a1837a4e04b
d98631724e81f33f462d86a1d895fae0
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

Connect to your Google Drive, select the path in your drive for saving the checkpoints of your model, which we will train later.

In [2]:

```
from google.colab import drive
drive.mount('/content/gdrive')

# Path to save the dataset.
PATH_TO_SAVE_MODEL = '/content/gdrive/MyDrive/' # TODO - UPDATE ME!
```

```
Mounted at /content/gdrive
```

Define random seeds in order to reproduce your results.

In [3]:

```
# TO DO: Set random seeds - to your choice
torch.manual_seed(1)         # Insert any integer
torch.cuda.manual_seed(1)    # Insert any integer
```

Question 1. Basic Principles (10 %)

Part (a) -- 3%

What is the difference between deterministic autoencoder we saw in class and the variational autoencoder?

In [4]:

```
# Write your explanation here
# AoutoEncoder deternministic is build a low dimentional representation for the data.
# Variational AutoEncoder is a stocastic AE that determines a priori distribution
# for the latent space z.
# for example: a normal distribution with a mean of 0 and covariance matrix.
```

Part (b) -- 3%

In which manner Variational Autoencoder is trained? Explain.

In [5]:

```
# Write your explanation here
# The encoder trains a network that receives samples from the training set,
# and tries to derive from them distribution parameters that are as close as possible
# to a priori distribution, which was determined in advance.
# From this learned distribution, new vectors are sampled and transferred to the decode
r.
# The decoder performs the opposite operation - takes a vector sampled from the latent
space,
# and uses it to produce a new example similar to the original data.
# The training process will be such that it minimizes the error of both parts of
# the VAE - the output will also be as close as possible to the original,
# and the distribution will also be as close as possible to the prior z distribution.
```

Part (c) -- 4%

In class we saw another generative model, known as generative adversarial network (GAN). What are the differences in terms of task objective between GANs and VAEs? Give an example for a task which a VAE is more suitable than GAN, and vice versa.

In [6]:

```
# Write your explanation here
# The main objective of a GAN is to generate data points that are indistinguishable
# from real data points. A GAN consists of two neural networks: a generator and a discr
iminator.
# The objective of the GAN is to find the equilibrium point where the generator
# is producing synthetic data points that are indistinguishable from real data points.

# On the other hand, the main objective of a VAE is to learn a compact, latent
# representation of the training data, and then use this representation to generate
# new data points. A VAE consists of an encoder and a decoder. The VAE is trained
# by minimizing the reconstruction loss, which measures how well the decoder is
# able to reconstruct the original data points from the latent representation.

# Example - VAE is more suitable than a GAN:
# Training a VAE on images of faces, might be able to use the latent
# representation to generate images of faces with specific characteristics
# for example, male vs. female, smiling vs. not smiling, by manipulating the latent
# representation . With a GAN, it is more difficult to control the characteristics
# of the generated data in this way.

# Example - GAN is more suitable than a VAE:
# Training a GAN to generate high-resolution images, might prioritize the realism of th
e
# generated images over the ability to control specific characteristics of the images
```

Question 2. Data (15 %)

In this assignement we are using the CelebFaces Attributes Dataset (CelebA).

The CelebA dataset, as its name suggests, is comprised of celebrity faces. The images cover large pose variations, background clutter, diverse people, supported by a large quantity of images and rich annotations. This data was originally collected by researchers at MMLAB, The Chinese University of Hong Kong.

Overall

- 202,599 number of face images of various celebrities
- 10,177 unique identities, but names of identities are not given
- 40 binary attribute annotations per image
- 5 landmark locations

In this torchvision version of the dataset, each image is in the shape of $[218, 178, 3]$ and the values are in $[0, 1]$.

Here, you will download the dataset to the Google Colab disk. It is highly recommended not to download the dataset to your own Google Drive account since it is time consuming.

In [7]:

```
data_path = "datasets" ## TO DO -- UPDATE ME!

base_url = "https://graal.ift.ulaval.ca/public/celeba/"

file_list = [
    "img_align_celeba.zip",
    "list_attr_celeba.txt",
    "identity_CelebA.txt",
    "list_bbox_celeba.txt",
    "list_landmarks_align_celeba.txt",
    "list_eval_partition.txt",
]

# Path to folder with the dataset
dataset_folder = f"{data_path}/celeba"
os.makedirs(dataset_folder, exist_ok=True)

for file in file_list:
    url = f"{base_url}/{file}"
    if not os.path.exists(f"{dataset_folder}/{file}"):
        wget.download(url, f"{dataset_folder}/{file}")

with zipfile.ZipFile(f"{dataset_folder}/img_align_celeba.zip", "r") as ziphandler:
    ziphandler.extractall(dataset_folder)
```

Part (a) -- 5%

Apply transformations:

The data is given as PIL (Python Imageing Library) images. Since we are working with PyTorch, we wish to apply transformations to the data in order to process it properly.

Here you should apply transformations to the data. There are many kinds of transformations which can be found here: https://pytorch.org/vision/stable/transforms.html (https://pytorch.org/vision/stable/transforms.html). Note that transformations can be chained together using Compose method.

Think which transformations can be suitable for this task and apply it in the form of:

trfm = transforms.Compose([transforms.transform1(),transforms.transform2(),...])

We recommend to consider:

- transforms.ToTensor()
- transforms.Resize()

In [8]:

```
height, width = 96, 80
trfm = transforms.Compose([transforms.ToTensor(), transforms.Resize((height, width))])
# You can add additional transformations which you think could be fit to the data.

training_data = datasets.CelebA(root=data_path, split='train', download=False, transfor
m=trfm) #load the dataset (without download it directly) from our root directory on goo
gle drive disk.
test_data = datasets.CelebA(root=data_path, split='test', download=False, transform=trf
m)
```

Part (b) -- 5%

In order to get in touch with the dataset, and to see what we are dealing with (which is always recommended), we wish to visualize some data samples from the CelebA dataset.

Write a function: show():

INPUT: Python list of length 32 where each element is an image, randomly selected from the training data.

OUTOUT: Showing a 8X4 grid of images.

In [17]:

```
def show(imgs):
    # your code goes here:
    plt.figure(figsize=(15,15))
    for i in range(8):
        for j in range(4):
            if not isinstance(imgs[i*4 + j], torch.Tensor):
                imgs[i*4 + j] = torch.Tensor(imgs[i*4 + j])
            plt.subplot(8, 4, i*4 + j + 1)
            plt.imshow(imgs[i*4 + j].transpose(0,2).transpose(0,1))
            plt.axis('off')
    plt.show()
rand_img = np.random.randint(0,162770,32)
img_lst = [training_data.__getitem__(i)[0] for _,i in enumerate(rand_img)]
show(img_lst)
```

Part (c) -- 5%

Extrapolate in the image domain:

Here, randomly take 2 images from the training dataset, combine them together and plot the result. For example, consider X_1 and X_2 to be 2 images randomly taken from the training data. Plot $\alpha \cdot X_1 + (1 - \alpha) \cdot X_2$.

Explain the results, is extrapolation in the image domain reasonable?

Note: Recall that the images should be in the $[0, 1]$ interval.

In [18]:

```
# Your code goes here

x1_index = random.randint(0,len(training_data))
x2_index = random.randint(0,len(training_data))
x1 = training_data.__getitem__(x1_index)[0]
x2 = training_data.__getitem__(x2_index)[0]
alpha = 0.3

plt.imshow(x1.transpose(0,2).transpose(0,1))
plt.axis('off')
plt.show()
plt.imshow(x2.transpose(0,2).transpose(0,1))
plt.axis('off')
plt.show()
x = alpha*x1 + (1-alpha)*x2
plt.imshow(x.transpose(0,2).transpose(0,1))
plt.axis('off')
plt.show()
```

In [19]:

```
# Your explanation goes here:

# I our result, combining two images in this way looks like we make one of the
# images a bit transperent and put it on the other.
# In addition, alpha is 0.3 so we can see that the second image is much more
# dominante than the first image.
```

Question 3. VAE Foundations (15 %)

Let us start by recalling the analytical derivation of the VAE.

The simplest version of VAE is comprised of an encoder-decoder architecture. The *encoder* is a neural network which its input is a datapoint x, its output is a hidden representation z, and it has weights and biases θ. We denote the encoder's mapping by $P_\theta(z|x)$. The *decoder* is another neural network which its input is the data sample z, its output is the reconstructed input x, and its parameters ϕ. Hence, we denote the decoder's mapping by $P_\phi(x|z)$.

The goal is to determine a posterior distribution $P_\theta(z|x)$ of a latent variable z given some data evidence x. However, determining this posterior distribution is typically computationally intractable, because according to Bayes:

$$(1)\ P(z|x) = \frac{P(x|z)P(z)}{P(x)}$$

The term $P(x)$ is called the evidence, and we can calculate it by marginalization on the latent variable:

$$P(x) = \int_z P(x|z)P(z)dz$$

Unfortunately, this term is intractable because it requires computation of the integral over the entire latent space z. To bypass this intractability problem we approximate the posterior distribution with some other distribution $q(z|x_i)$. This approximation is made by the KL-divergence:

$$(2)\ D_{KL}\left(q(z|x_i)||P(z|x_i)\right) = \int_z q(z|x_i)\cdot\log\left(\frac{q(z|x_i)}{P(z|x_i)}\right)dz = -\int_z q(z|x_i)\cdot\log\left(\frac{P(z|x_i)}{q(z|x_i)}\right)dz \geq 0$$

Applying Bayes' theorem to the above equation yields,

$$(3)\ D_{KL}\left(q(z|x_i)||P(z|x_i)\right) = -\int_z q(z|x_i)\cdot\log\left(\frac{P(x_i|z)P(z)}{q(z|x_i)P(x_i)}\right)dz \geq 0$$

This can be broken down using laws of logarithms, yielding,

$$(4)\ -\int_z q(z|x_i)\cdot\left[\log\left(\frac{P(x_i|z)P(z)}{q(z|x_i)}\right) - \log(P(x_i))\right]dz \geq 0$$

Distributing the integrand then yields,

$$(5)\ -\int_z q(z|x_i)\cdot\log\left(\frac{P(x_i|z)P(z)}{q(z|x_i)}\right)dz + \int_z q(z|x_i)\log(P(x_i))dz \geq 0$$

In the above, we note that $log(P(x))$ is a constant and can therefore be pulled out of the second integral above, yielding,

$$(6)\ -\int_z q(z|x_i)\cdot\log\left(\frac{P(x_i|z)P(z)}{q(z|x_i)}\right)dz + \log(P(x_i))\int_z q(z|x_i)dz \geq 0$$

And since $q(z|x_i)$ is a probability distribution it integrates to 1 in the above equation, yielding,

$$(7)\ -\int_z q(z|x_i)\cdot\log\left(\frac{P(x_i|z)P(z)}{q(z|x_i)}\right)dz + \log(P(x_i)) \geq 0$$

Then carrying the integral over to the other side of the inequality, we get,

$$(8)\ \log(P(x_i)) \geq \int_z q(z|x_i)\cdot\log\left(\frac{P(x_i|z)P(z)}{q(z|x_i)}\right)dz$$

From Equation (8) it follows that:

$$(9)\ \log(P(x_i)) \geq \int_z q(z|x_i) \cdot \log\left(\frac{P(z)}{q(z|x_i)}\right)dz + \int_z q(z|x_i) \cdot \log\left(P(x_i|z)\right)dz$$

Which is equivalent to:

$$(10)\ \log(P(x_i)) \geq -D_{KL}\left(q(z|x_i)||P(z)\right) + E_{q(z|x_i)}\left[\log\left(P(x_i|z)\right)\right]$$

The right hand side of the above equation is the Evidence Lower BOund (ELBO). Its bounds $\log(P(x))$ which is the term we seek to maximize. Therefore, maximizing the ELBO maximizes the log probability of our data.

Part (a) -- 5%

A we see above, the $ELBO = -D_{KL}\left(q(z|x_i)||P(z)\right) + E_{q(z|x_i)}\left[\log\left(P(x_i|z)\right)\right]$ is comprised of 2 terms. Explain the meaning of each one of them in terms of a loss function.

In [20]:

```
# Write your explanation here

# The first term, DKL(q(z|xi)||P(z)), is the Kullback-Leibler divergence between
# the approximate posterior q(z|xi) and the prior distribution P(z).
# This term measures the difference between the two distributions, with smaller
# values indicating that the approximate posterior is closer to the prior.
# The KL divergence is always non-negative, so minimizing this term encourages
# the approximate posterior to be similar to the prior.

# The second term, E q(z|xi)[log(P(xi|z))], is the expected log-likelihood of the
# observations under the approximate posterior distribution. This term measures
# how well the observations fit the model, with larger values indicating a better fit.
# Minimizing the ELBO is therefore equivalent to finding an approximate posterior
# that is both close to the prior and has a high likelihood of the observations.
```

Part (b) -- 10%

As we saw in class, in traditional variational autoencoder we assume:

$$P(z) \sim N(\mu_p, \sigma_p^2) = \frac{1}{\sqrt{2\pi\sigma_p^2}}\exp\left(-\frac{(z-\mu_p)^2}{2\sigma_p^2}\right)$$

and

$$q(z|x) \sim N(\mu_q, \sigma_q^2) = \frac{1}{\sqrt{2\pi\sigma_q^2}}\exp\left(-\frac{(z-\mu_q)^2}{2\sigma_q^2}\right)$$

Assume $\mu_p = 0$ and $\sigma_p^2 = 1$. Show that:

$$-D_{KL}\left(q(z|x_i)||P(z)\right) = \frac{1}{2}\left[1 + \log(\sigma_q^2) - \sigma_q^2 - \mu_q^2\right]$$

WRITE YOUR SOLUTIUON HERE. (You can also upload your solution as an image.)

$$-D_{KL}\left(q(z|x_i)||P(z)\right) = \frac{1}{2}\left[1+\log(\sigma_q^2)\right] - \frac{1}{2\cdot\sigma_q^2}\int_z \exp\left(-\frac{(z-\mu_q)^2}{2\sigma_q^2}\right)\cdot((z-\mu_q)^2+\sigma_q^2)dz =$$

$$= \frac{1}{2}\left[1+\log(\sigma_q^2) - \sigma_q^2 - \mu_q^2\right]$$

Minimizing the loss function, over a batch in the dataset now can be written as:

$$\mathcal{L}(\theta,\phi) = -\sum_j^J\left(\frac{1}{2}\left[1+\log(\sigma_{q_j}^2) - \sigma_{q_j}^2 - \mu_{q_j}^2\right]\right) - \frac{1}{M}\sum_i^M\left(E_{q_\theta(z|x_i)}\left[\log\left(P_\phi(x_i|z)\right)\right]\right)$$

where J is the dimension of the latent vector z and M is the number of samples stochastically drawn from the dataset.

Question 4. VAE Implementation (25 %)

As seen in class, a suitable way to extract features from dataset of images is by convolutional neural network (CNN). Hence, here you will build a convolutional VAE. \ The basic idea is to start from full resolution images, and by convolutional kernels extract the important features of the dataset. Remember that the output of the VAE should be in the same dimensions (H_1, W_1, C_1) as the input images.

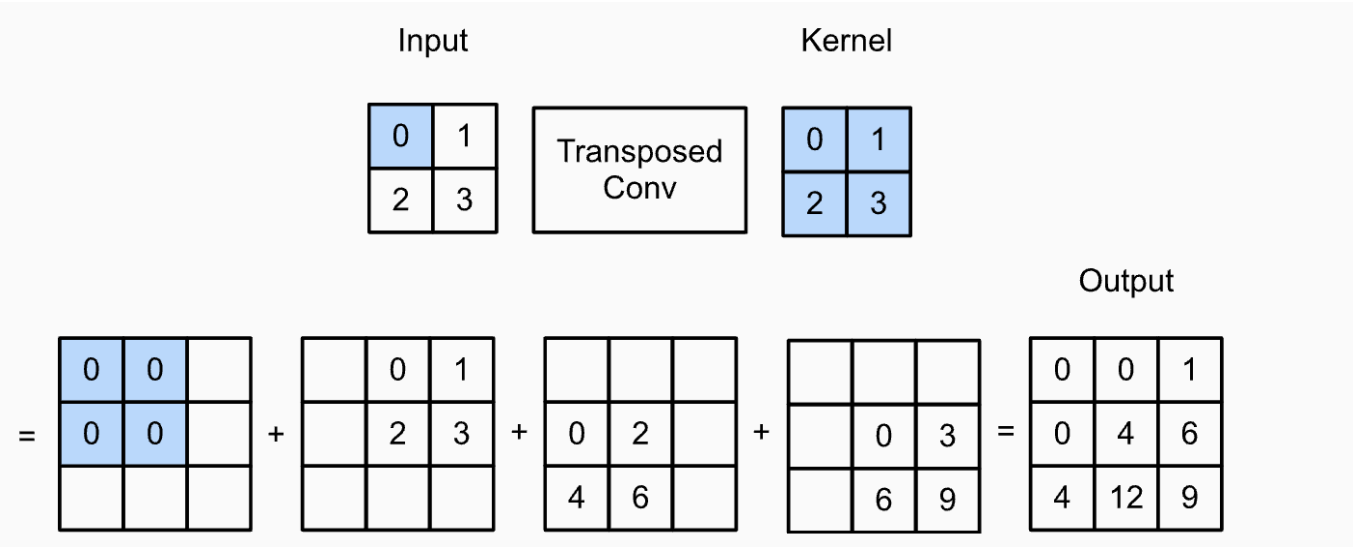
The encoder should be comprised of convolutional layers (nn.Conv2d). Recall that the dimension of the input images is changing according to:

$$Z = \left(H_2(=\frac{H_1-F+2P}{S}+1), W_2(=\frac{W_1-F+2P}{S}+1), C_2\right)$$

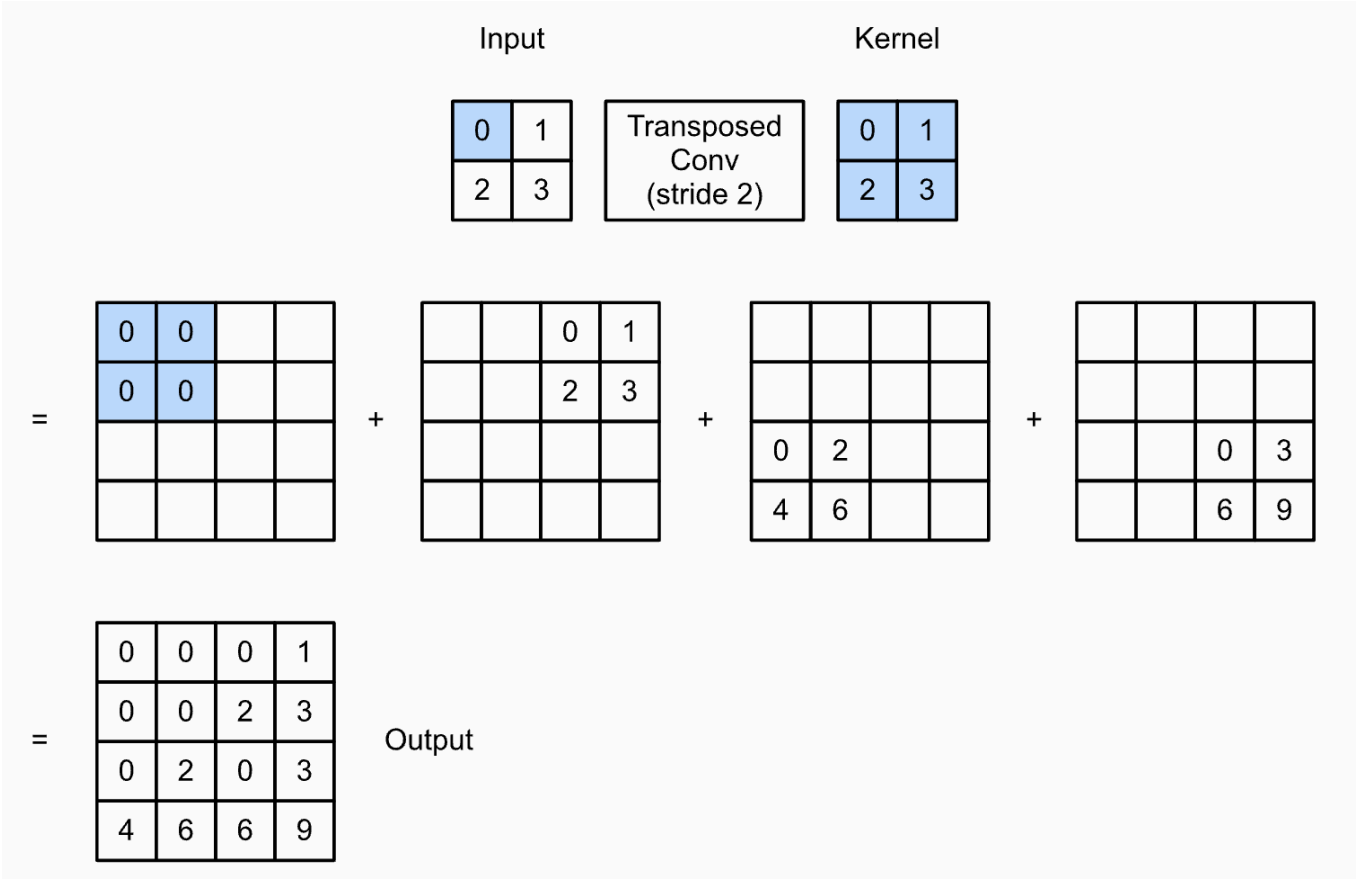
where S is the stride, F is the kernel size, P is the zero padding and C_2 is the selected output channels. Z is the output image.

The decoder should reconstruct the images from the latent space. In order to enlarge dimensions of images, your network should be comprised of transposed convolutional layers (nn.ConvTranspose2d). See the following images of the operation of transpose convolution to better understand the way it works.

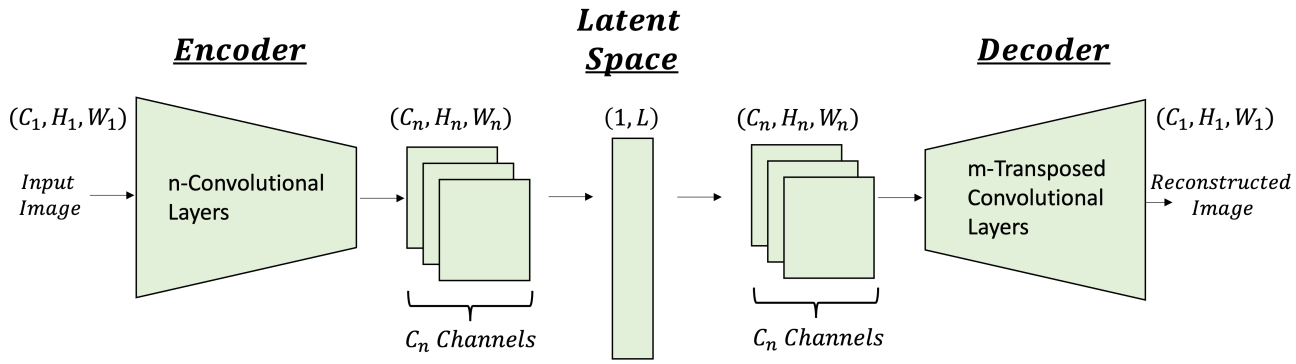
Transposed Convolution with Stride = 1



Transposed Convolution with Stride = 2



The architecture of your VAE network should be in the following form:



Part (a) -- 7%

Encoder

Here, you will implement the architecture of the encoder.

The encoder should consist of 4 Blocks as follows:

BLOCK 1:

- Convolutional layer (nn.Conv2D(in_channels, num_hidden, kernel_size=(3,3), stride=(2,2)))
- Batch Normalization(num_hidden)
- Activation Function: nn.ReLU()

BLOCK 2:

- Convolutional layer (nn.Conv2D(num_hidden, num_hidden * 2, kernel_size=(3,3), stride=(2,2))
- Batch Normalization(num_hidden * 2)
- Activation Function: nn.ReLU()

BLOCK 3:

- Convolutional layer (nn.Conv2D(num_hidden *2, num_hidden* 4, kernel_size=(3,3), stride=(2,2))
- Batch Normalization(num_hidden * 4)
- Activation Function: nn.ReLU()

BLOCK 4:

- Convolutional layer (nn.Conv2D(num_hidden *4, num_hidden* 8, kernel_size=(3,3), stride=(2,2))
- Batch Normalization(num_hidden * 8)
- Activation Function: nn.ReLU()

In addidion to the 4 Blocks, you should add the following linear layers:

Linear μ:

- nn.Linear(___ ,latent).

Linear $\log(\sigma)$:

- nn.Linear(___ ,latent).

NOTES:

- The input of the linear layer should be according to the size of the images you picked in the transformation part. (If you did resize the images)
- Consider using Padding in the convolutional layers to correct mismatches in sizes.
- In the forward function, you will have to reshape the output from the 4'th block to $(H_4 \cdot W_4 \cdot C_4, latent)$, where H_4 is the height of the output image from the 4'th block, W_4 is the width of the output image from the 4'th block and C_4 is num_hidden*8 (number of channels of the output image from the 4'th block).

You can change any parameter of the network to suit your code - this is only a recommendation.

In [21]:

```
class Encoder(nn.Module):
    def __init__(self, in_channels, num_hiddens, latent, height=96, width=80):
        super(Encoder, self).__init__()
        # YOUR CODE GOES HERE:
        self.kernel_size = 3
        self.padding_size = (self.kernel_size - 1) // 2
        self.num_hiddens = num_hiddens
        self.latent = latent
        self.block1 = nn.Sequential(nn.Conv2d(in_channels, num_hiddens, kernel_size=(3,
3), stride=(2,2), padding=self.padding_size),
                                    nn.BatchNorm2d(num_hiddens),
                                    nn.ReLU())

        self.block2 = nn.Sequential(nn.Conv2d(num_hiddens, num_hiddens * 2, kernel_size
=(3,3), stride=(2,2), padding=self.padding_size),
                                    nn.BatchNorm2d(num_hiddens * 2),
                                    nn.ReLU())

        self.block3 = nn.Sequential(nn.Conv2d(num_hiddens * 2, num_hiddens * 4, kernel_
size=(3,3), stride=(2,2), padding=self.padding_size),
                                    nn.BatchNorm2d(num_hiddens * 4),
                                    nn.ReLU())

        self.block4 = nn.Sequential(nn.Conv2d(num_hiddens * 4, num_hiddens * 8, kernel_
size=(3,3), stride=(2,2), padding=self.padding_size),
                                    nn.BatchNorm2d(num_hiddens * 8),
                                    nn.ReLU())

        self.size =  int((height//16)*(width//16)*num_hiddens*8)
        self.fc_mu = nn.Linear(self.size,latent)     # Insert the input size
        self.fc_logvar = nn.Linear(self.size,latent) # Insert the input size

    def forward(self, inputs):
        # YOUR CODE GOES HERE:
        x = self.block1(inputs)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x = x.view(-1, self.size)
        mu = self.fc_mu(x)
        logvar = self.fc_logvar(x)
        return mu, logvar
```

Notice: We output $\log \sigma$ and not σ^2, this is a convention when training VAEs but it is completely equivalent.

Part (b) -- 7%

Decoder

Here, you will implement the architecture of the decoder.

First, Apply a linear layer to the input of the decoder as follows:

- nn.Linear(latent, ___).

The output of the linear layer should match to $H_4 \cdot W_4 \cdot C_4$, which were the same parameters from the encoder 4'th block's output.

Then, the decoder should consist of 4 Blocks as follows:

BLOCK 1:

- Transposed Convolutional layer (nn.ConvTranspose2d(in_channels, num_hidden // 2, kernel_size=(4,4), stride=(2,2)))
- Batch Normalization(num_hidden // 2)
- Activation Function: nn.ReLU() or nn.LeakyReLU()

BLOCK 2:

- Transposed Convolutional layer (nn.ConvTranspose2d(num_hidden // 2, num_hidden // 4, kernel_size=(4,4), stride=(2,2)))
- Batch Normalization(num_hidden // 4)
- Activation Function: nn.ReLU() or nn.LeakyReLU()

BLOCK 3:

- Transposed Convolutional layer (nn.ConvTranspose2d(num_hidden // 4, num_hidden // 8, kernel_size=(4,4), stride=(2,2)))
- Batch Normalization(num_hidden // 8)
- Activation Function: nn.ReLU() or nn.LeakyReLU()

BLOCK 4:

- Transposed Convolutional layer (nn.ConvTranspose2d(num_hidden // 8, num_hidden // 8, kernel_size=(4,4), stride=(2,2)))
- Batch Normalization(num_hidden // 8)
- Activation Function: nn.ReLU() or nn.LeakyReLU()

Afterwards, we should generate an image in the same size as our input images. Thus add 1 more block consisting of:

BLOCK 5:

- nn.Conv2d(num_hiddens//8, out_channels=3,kernel_size=(3,3), stride=(1,1), padding=(1,1)),
- Activation function.

NOTES:

- The output of the linear layer should be according to the size of the images you picked in the transformation part. (If you did resize the images)
- Consider using Padding in the transposed convolutional layers to correct mismatches in sizes.

- In the forward function, you will have to reshape the output of the linear layer to $(Batch, H_4, W_4, C_4)$
- The output of the decoder should be of values in $[0, 1]$.

You can change any parameter of the network to suit your code, this is only a recommendation.

In [22]:

```
class Decoder(nn.Module):
    def __init__(self, in_channels, num_hiddens,latent, height=96, width=80):
        super(Decoder, self).__init__()
        # YOUR CODE GOES HERE:
        self.kernel_size = 3
        self.padding_size = (self.kernel_size - 1) // 2
        self.size = int((height//16)*(width//16)*num_hiddens)
        self.num_hiddens = num_hiddens
        self.fc_dec = nn.Linear(latent, self.size) # Insert the output size

        self.block1 = nn.Sequential(nn.ConvTranspose2d(in_channels, num_hiddens // 2, k
ernel_size=(4,4), stride=(2,2), padding=self.padding_size),
                                    nn.BatchNorm2d(num_hiddens // 2),
                                    nn.ReLU())

        self.block2 = nn.Sequential(nn.ConvTranspose2d(num_hiddens // 2, num_hiddens //
4, kernel_size=(4,4), stride=(2,2), padding=self.padding_size),
                                    nn.BatchNorm2d(num_hiddens // 4),
                                    nn.ReLU())

        self.block3 = nn.Sequential(nn.ConvTranspose2d(num_hiddens // 4, num_hiddens //
8, kernel_size=(4,4), stride=(2,2), padding=self.padding_size),
                                    nn.BatchNorm2d(num_hiddens // 8),
                                    nn.ReLU())

        self.block4 = nn.Sequential(nn.ConvTranspose2d(num_hiddens // 8, num_hiddens //
8, kernel_size=(4,4), stride=(2,2), padding=self.padding_size),
                                    nn.BatchNorm2d(num_hiddens // 8),
                                    nn.ReLU())

        self.block5 = nn.Sequential(nn.ConvTranspose2d(num_hiddens // 8, out_channels=
3, kernel_size=(3,3), stride=(1,1), padding=(1,1)),
                                    nn.ReLU(),
                                    nn.Sigmoid()) # Add convolution layer and activatio
n layer


    def forward(self, inputs):
        # YOUR CODE GOES HERE:
        x = self.fc_dec(inputs)
        x = x.view(-1, self.num_hiddens, height//16, width//16)
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)
        x_rec = self.block5(x)

        return x_rec
```

Part (c) -- 4%

VAE Model

Once you have the architecture of the encoder and the decoder, we want to put them together and train the network end-to-end.

Remember that in VAEs, you need to sample from a gaussian distribution at the input of the decoder. In order to backpropagate through the network, we use the reparametrization trick. The reparametrization trick is saying that sampling from $z \sim N(\mu, \sigma)$ is equivalent to sampling $\varepsilon \sim N(0, 1)$ and setting $z = \mu + \sigma \odot \varepsilon$. Where, epsilon is an input to the network while keeping your sampling operation differentiable. The reparametrization function is given to you in the VAE class.

Here, you should write the *forward()* function and to combine all the model's settings to a final network.

In [23]:

```
class VAE(nn.Module):
    def __init__(self, enc_in_chnl,enc_num_hidden,dec_in_chnl,dec_num_hidden,latent):
        super(VAE, self).__init__()
        self.encode = Encoder(in_channels=enc_in_chnl, num_hiddens=enc_num_hidden,laten
t=latent)
        self.decode = Decoder(in_channels=dec_in_chnl, num_hiddens=dec_num_hidden,laten
t=latent)

    # Reparametrization Trick
    def reparametrize(self, mu, logvar):
      std = torch.exp(0.5 * logvar)
      eps = torch.randn_like(std)
      return eps.mul(std).add_(mu)

    # Initialize Weights
    def weight_init(self, mean, std):
        for m in self._modules:
            if isinstance(m, nn.ConvTranspose2d) or isinstance(m, nn.Conv2d):
                m.weight.data.normal_(mean, std)
                m.bias.data.zero_()

    def forward(self, x):
        # YOUR CODE GOES HERE:
        mu, logvar = self.encode(x)
        z = self.reparametrize(mu, logvar)
        x_rec = self.decode(z)
        #x_rec = torch.clamp(x_rec, min=0.0000001, max=0.9999999)

        return x_rec , mu, logvar
```

Part (d) -- 7%

Loss Function

As we saw earlier, the loss function is based on the ELBO; Over a batch in the dataset, it can be written as:

$$\mathcal{L}(\theta,\phi) = -\sum_j^J \left(\frac{1}{2}\left[1+\log(\sigma_{q_j}^2) - \sigma_{q_j}^2 - \mu_{q_j}^2\right]\right) - \frac{1}{M}\sum_i^M \left(E_{q_\theta(z|x_i)}\left[\log\left(P_\phi(x_i|z)\right)\right]\right)$$

where J is the dimension of the latent vector z and M is the number of samples stochastically drawn from the dataset.

β-Variational Autoencoder (β-VAE)

As seen in class, the fact that the ELBO is comprised of the sum of two loss terms implies that these can be balanced using an additional hyperparameter β, i.e.,

$$\beta \cdot D_{KL}\left(q(z|x_i)||P(z)\right) - E_{q(z|x_i)}\left[\log\left(P(x_i|z)\right)\right]$$

It is highly recommended to use the β-loss for increasing performance.

Explain what could be the purpose of the hyperparameter β in the loss function? If $\beta = 1$ is same as VAE, What is the effect of $\beta \neq 1$?

In [24]:

```
# Write your explanation here
# The hyperparameter β controls the balance between the reconstruction loss and
# the KL divergence term in the ELBO loss function.

# If β > 1, then the KL divergence term is more efficient in the loss function,
# which means that the approximations of the latent distribution will be more accuracy
# and will try to more closely match the prior.
# Adventages: prevent overfitting and improve the generalization capabilities of the VA
E.

# If β < 1, then the reconstruction loss term is more efficient in the loss
# function, which means that reconstructing the input data is getting priority over
# matching the prior.
#Adventages: higher reconstruction quality at the cost of potentially lower latent spac
e interpretability.
```

Here you should write specifically the code for the loss function.

In [25]:

```
beta = 0.3
def vae_loss(x_recon, x, mu, logvar):
    # YOUR CODE GOES HERE....
    MSE = nnF.mse_loss(x_recon, x, reduction='sum')
    # BCE = nnF.binary_cross_entropy(x_recon, x, reduction='mean')
    KLD = 0.5*torch.mean(logvar.exp()+mu.pow(2)-logvar-1, dim=-1).mean(dim=0)
    return MSE + KLD*beta
```

Here, define all the hyperparameters values for the training process.

We gave you recommended values for the VAE model. You can modify and change it to suit your code better if needed.

In [26]:

```
learning_rate = 0.0001
batch_size = 32
num_epochs = 50
dataset_size = 30000 # How many data samples to use for training, 30,000 should be enou
gh.

#VAE Class inputs:
enc_in_chnl = 3
enc_num_hidden = 32
dec_in_chnl = 256
dec_num_hidden = 256
```

In [27]:

```
train_loader = torch.utils.data.DataLoader(training_data, batch_size=batch_size, shuffl
e=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=Tru
e)
```

Question 5. VAE Training (15 %)

Part (a) -- 4%

Complete the training function below

In [29]:

```
def train(num_epochs,batch_size,dataset_size,model, train_loader, test_loader, learning
_rate):
    '''
    This is a starter code for the training process. You can modify it for your
    own conveinient.
    num_epochs - number of training epochs
    batch_size - size of the batches
    dataset_size - How many training samples to use.
    model - The model you are training.

    Note: decide what are the outputs of the function.
    '''
    # Your code goes Here:
    train_losses = []
    test_losses = []
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    model.to(device)
    # training
    for epoch in range(num_epochs):
        model.train()
        train_loss = 0
        for batch_idx, batch in enumerate(train_loader):
            imgs, labels = batch
            imgs = imgs.to(device)
            if torch.cuda.is_available():
                imgs = imgs.cuda()
            # forward
            x_rec , mu, logvar = model(imgs)
            loss = vae_loss(x_rec, imgs, mu, logvar)
            train_loss += loss.item()

            # backward
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            # Break the training loop when you have reached the desired number of sampl
es
            if dataset_size//batch_size == batch_idx:
                break
        train_losses.append(train_loss/batch_idx)


        # Validation set
        with torch.no_grad():
            model.eval()
            test_loss = 0
            for batch_idx, batch in enumerate(test_loader):
                img, label = batch
                img = img.to(device)
                if torch.cuda.is_available():
                    img = img.cuda()
                # forward
                x_rec , mu, logvar = model(img)
                loss = vae_loss(x_rec, img, mu, logvar)
                test_loss += loss.item()

            test_losses.append(test_loss/batch_idx)
        print(f"epoch: {epoch+1}/{num_epochs} - train loss: {train_losses[-1]:0.3f}, te
st loss: {test_losses[-1]:0.3f}")
```

```
    tensor_to_move = [x_rec, mu, logvar, img, imgs]
    for tensor in tensor_to_move:
      tensor = tensor.to('cpu')

    model = model.to('cpu')

    return x_rec, mu, logvar, train_losses, test_losses
```

Part (b) -- 4%

We first train with dimension of latent space $L = 3$

We recommend to use *weight_init()* function, which helps stabilize the training process.

In [30]:

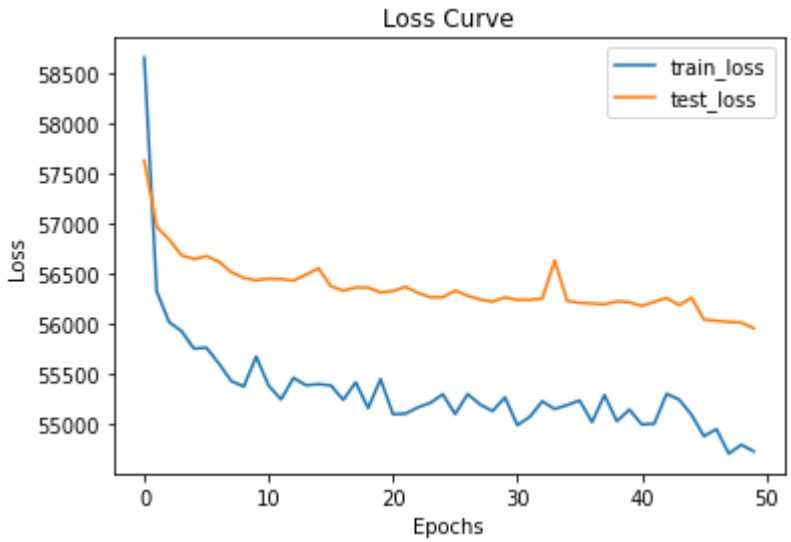
```
latent1 = 3

if torch.cuda.is_available():
    model_1 = VAE(enc_in_chnl,enc_num_hidden,dec_in_chnl,dec_num_hidden,latent1).cuda()
    model_1.weight_init(mean=0, std=0.02)
else:
    model_1 = VAE(enc_in_chnl,enc_num_hidden,dec_in_chnl,dec_num_hidden,latent1)
    model_1.weight_init(mean=0, std=0.02)

optimizer = torch.optim.Adam(model_1.parameters(), lr=learning_rate)
```

Train your model, plot the train and the validation loss graphs. Explain what is seen.

In [31]:

```
# Your Code Goes Here
def plot_losses(train_losses, test_losses):
    train_iters = list(range(len(train_losses)))
    test_iters = train_iters[::len(train_iters)//len(test_losses)]
    plt.title("Loss Curve")
    plt.plot(train_iters, train_losses, label="train_loss")
    plt.plot(test_iters, test_losses, label="test_loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()

x_rec, mu, logvar, train_losses, test_losses = train(num_epochs, batch_size, dataset_si
ze, model_1, train_loader, test_loader, learning_rate)
plot_losses(train_losses, test_losses)
```

```
epoch: 1/50 - train loss: 58652.674, test loss: 57627.675
epoch: 2/50 - train loss: 56323.641, test loss: 56967.895
epoch: 3/50 - train loss: 56018.531, test loss: 56840.174
epoch: 4/50 - train loss: 55927.077, test loss: 56683.479
epoch: 5/50 - train loss: 55753.817, test loss: 56645.834
epoch: 6/50 - train loss: 55763.231, test loss: 56675.026
epoch: 7/50 - train loss: 55605.580, test loss: 56620.573
epoch: 8/50 - train loss: 55430.480, test loss: 56515.756
epoch: 9/50 - train loss: 55376.213, test loss: 56458.799
epoch: 10/50 - train loss: 55675.120, test loss: 56434.216
epoch: 11/50 - train loss: 55384.998, test loss: 56449.951
epoch: 12/50 - train loss: 55248.028, test loss: 56446.688
epoch: 13/50 - train loss: 55462.151, test loss: 56431.585
epoch: 14/50 - train loss: 55388.631, test loss: 56491.018
epoch: 15/50 - train loss: 55399.879, test loss: 56554.792
epoch: 16/50 - train loss: 55387.394, test loss: 56376.177
epoch: 17/50 - train loss: 55243.650, test loss: 56331.436
epoch: 18/50 - train loss: 55417.653, test loss: 56362.439
epoch: 19/50 - train loss: 55160.293, test loss: 56361.176
epoch: 20/50 - train loss: 55449.737, test loss: 56313.872
epoch: 21/50 - train loss: 55100.440, test loss: 56326.408
epoch: 22/50 - train loss: 55106.132, test loss: 56370.316
epoch: 23/50 - train loss: 55168.423, test loss: 56309.122
epoch: 24/50 - train loss: 55213.305, test loss: 56266.482
epoch: 25/50 - train loss: 55299.044, test loss: 56264.892
epoch: 26/50 - train loss: 55101.430, test loss: 56331.435
epoch: 27/50 - train loss: 55301.471, test loss: 56280.554
epoch: 28/50 - train loss: 55197.500, test loss: 56242.840
epoch: 29/50 - train loss: 55131.305, test loss: 56221.120
epoch: 30/50 - train loss: 55268.839, test loss: 56263.700
epoch: 31/50 - train loss: 54990.434, test loss: 56239.801
epoch: 32/50 - train loss: 55074.494, test loss: 56240.326
epoch: 33/50 - train loss: 55229.027, test loss: 56252.753
epoch: 34/50 - train loss: 55152.010, test loss: 56631.220
epoch: 35/50 - train loss: 55191.322, test loss: 56228.957
epoch: 36/50 - train loss: 55238.316, test loss: 56210.451
epoch: 37/50 - train loss: 55021.509, test loss: 56202.575
epoch: 38/50 - train loss: 55291.275, test loss: 56195.841
epoch: 39/50 - train loss: 55029.866, test loss: 56224.417
epoch: 40/50 - train loss: 55146.862, test loss: 56216.045
epoch: 41/50 - train loss: 54997.927, test loss: 56178.837
epoch: 42/50 - train loss: 55005.922, test loss: 56220.688
epoch: 43/50 - train loss: 55304.097, test loss: 56258.517
epoch: 44/50 - train loss: 55246.472, test loss: 56187.202
epoch: 45/50 - train loss: 55094.852, test loss: 56260.159
epoch: 46/50 - train loss: 54881.484, test loss: 56043.415
epoch: 47/50 - train loss: 54951.164, test loss: 56030.645
epoch: 48/50 - train loss: 54707.696, test loss: 56021.103
epoch: 49/50 - train loss: 54797.204, test loss: 56012.547
epoch: 50/50 - train loss: 54730.707, test loss: 55956.728
```



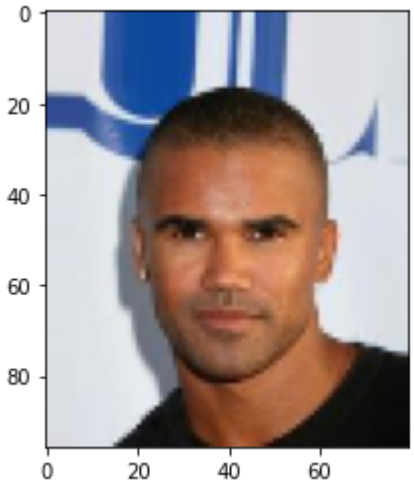
Answer:

As we can see, the training data and the test data both have a decreasing by the loss function in the same rate.

Visualize, from the test dataset, an original image against a reconstructed image. Has the model reconstructed the image successfully? Explain.

In [113]:

```
# Your Code Goes Here
with torch.no_grad():
    model_1.eval()
    model_1.to(device)
    imgs, label = next(iter(test_loader))
    if torch.cuda.is_available():
        imgs = imgs.cuda()

    x_rec, mu, logvar = model_1(imgs)

    imgs = imgs.transpose(1, 3)
    imgs = imgs.transpose(1, 2)
    plt.imshow(imgs[0].cpu())
    plt.show()
    x_rec = x_rec.transpose(1, 3)
    x_rec = x_rec.transpose(1, 2)
    plt.imshow(x_rec[0].cpu())
    plt.axis('off')
    plt.show()
```



Answer:

As we can see, the reconstruction image doesn't match to the original image due to the small latent size.

We also saw that the reconstruction images was blurred and the same for different original images due to the fact that the latent save only outlines like eyes, nose and mouth.

Part (c) -- 7%

Next, we train with larger $L > 3$

Based on the results for $L = 3$, choose a larger L to improve your results. Train new model with your choice for L.

In [41]:

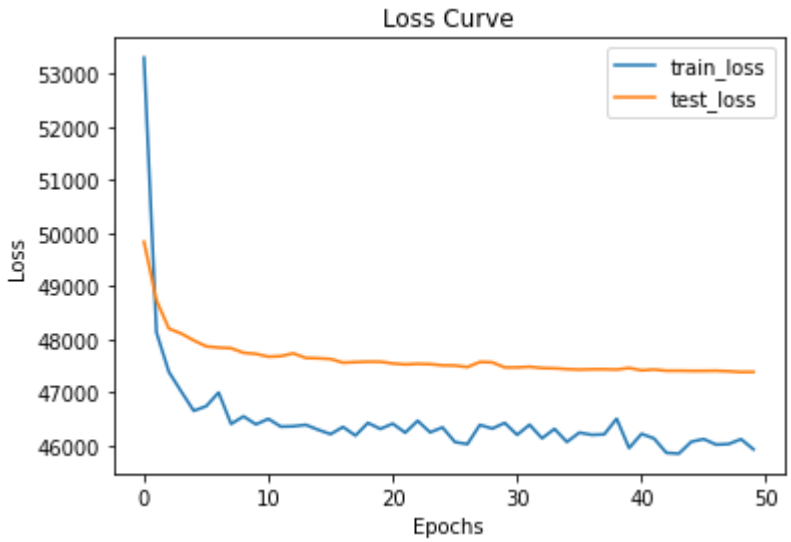
```
latent2 = 80 # TO DO: Choose latent space dimension.

if torch.cuda.is_available():
    model_2 = VAE(enc_in_chnl,enc_num_hidden,dec_in_chnl,dec_num_hidden,latent2).cuda()
    model_2.weight_init(mean=0, std=0.02)
else:
    model_2 = VAE(enc_in_chnl,enc_num_hidden,dec_in_chnl,dec_num_hidden,latent2)
    model_2.weight_init(mean=0, std=0.02)

optimizer = torch.optim.Adam(model_2.parameters(), lr=learning_rate)
```

Plot the train and the validation loss graphs. Explain what is seen.

In [42]:

```
# Your Code Goes Here
beta = 0.5
x_rec, mu, logvar, train_losses, test_losses = train(num_epochs, batch_size, dataset_si
ze, model_2, train_loader, test_loader, learning_rate)
plot_losses(train_losses, test_losses)
```

```
epoch: 1/50 - train loss: 53300.328, test loss: 49834.825
epoch: 2/50 - train loss: 48129.079, test loss: 48730.841
epoch: 3/50 - train loss: 47383.844, test loss: 48199.732
epoch: 4/50 - train loss: 47014.012, test loss: 48106.225
epoch: 5/50 - train loss: 46652.603, test loss: 47977.900
epoch: 6/50 - train loss: 46739.881, test loss: 47866.032
epoch: 7/50 - train loss: 46996.755, test loss: 47842.444
epoch: 8/50 - train loss: 46405.382, test loss: 47830.746
epoch: 9/50 - train loss: 46548.155, test loss: 47744.983
epoch: 10/50 - train loss: 46397.819, test loss: 47723.688
epoch: 11/50 - train loss: 46501.338, test loss: 47671.282
epoch: 12/50 - train loss: 46354.926, test loss: 47681.575
epoch: 13/50 - train loss: 46361.913, test loss: 47733.835
epoch: 14/50 - train loss: 46389.050, test loss: 47647.805
epoch: 15/50 - train loss: 46296.097, test loss: 47640.693
epoch: 16/50 - train loss: 46213.692, test loss: 47627.939
epoch: 17/50 - train loss: 46348.619, test loss: 47557.974
epoch: 18/50 - train loss: 46187.201, test loss: 47569.465
epoch: 19/50 - train loss: 46425.269, test loss: 47577.011
epoch: 20/50 - train loss: 46312.298, test loss: 47575.473
epoch: 21/50 - train loss: 46410.667, test loss: 47544.074
epoch: 22/50 - train loss: 46241.224, test loss: 47525.534
epoch: 23/50 - train loss: 46464.479, test loss: 47536.876
epoch: 24/50 - train loss: 46247.716, test loss: 47530.577
epoch: 25/50 - train loss: 46339.924, test loss: 47508.430
epoch: 26/50 - train loss: 46064.596, test loss: 47504.792
epoch: 27/50 - train loss: 46026.234, test loss: 47475.685
epoch: 28/50 - train loss: 46387.841, test loss: 47572.152
epoch: 29/50 - train loss: 46317.941, test loss: 47564.144
epoch: 30/50 - train loss: 46424.607, test loss: 47469.271
epoch: 31/50 - train loss: 46200.833, test loss: 47468.992
epoch: 32/50 - train loss: 46390.585, test loss: 47481.629
epoch: 33/50 - train loss: 46136.235, test loss: 47456.708
epoch: 34/50 - train loss: 46312.046, test loss: 47452.489
epoch: 35/50 - train loss: 46066.907, test loss: 47437.267
epoch: 36/50 - train loss: 46242.926, test loss: 47428.482
epoch: 37/50 - train loss: 46200.663, test loss: 47434.109
epoch: 38/50 - train loss: 46209.233, test loss: 47435.125
epoch: 39/50 - train loss: 46504.336, test loss: 47428.767
epoch: 40/50 - train loss: 45951.132, test loss: 47459.230
epoch: 41/50 - train loss: 46218.732, test loss: 47413.892
epoch: 42/50 - train loss: 46133.317, test loss: 47429.387
epoch: 43/50 - train loss: 45862.338, test loss: 47406.792
epoch: 44/50 - train loss: 45846.494, test loss: 47407.056
epoch: 45/50 - train loss: 46071.299, test loss: 47403.212
epoch: 46/50 - train loss: 46120.893, test loss: 47403.539
epoch: 47/50 - train loss: 46016.319, test loss: 47405.770
epoch: 48/50 - train loss: 46027.594, test loss: 47397.054
epoch: 49/50 - train loss: 46119.436, test loss: 47382.896
epoch: 50/50 - train loss: 45926.729, test loss: 47384.923
```



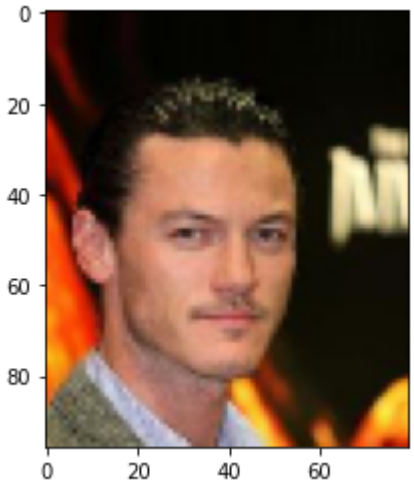
Answer:

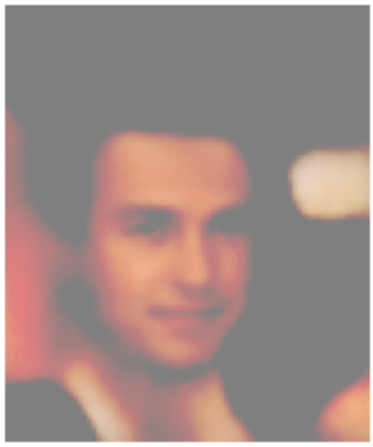
As we can see, the results are the same to model_1, the training data and the test data both have a decreasing by the loss function in the same rate.

Visualize, from the test dataset, an original image against a reconstructed image. Has the model reconstructed the image successfully? Are the images identical? Explain.

In [119]:

```
# Your Code Goes Here
with torch.no_grad():
    model_2.eval()
    model_2.to(device)
    imgs, label = next(iter(test_loader))
    if torch.cuda.is_available():
        imgs = imgs.cuda()

    x_rec, mu, logvar = model_2(imgs)

    imgs = imgs.transpose(1, 3)
    imgs = imgs.transpose(1, 2)
    plt.imshow(imgs[0].cpu())
    plt.show()
    x_rec = x_rec.transpose(1, 3)
    x_rec = x_rec.transpose(1, 2)
    plt.imshow(x_rec[0].cpu())
    plt.axis('off')
    plt.show()
```





Answer:

As we can see, the reconstruction image match better to the original image than model_1 due to a bigger latent size.

We also saw that now the reconstruction images was less blurred and not the same for different original images due to the fact that now the latent save more than outlines like eyes, nose and mouth, more details in the image like color of the skine, hair and background.

What will happened if we choose extremely high dimension for the latent space?

In [48]:

```
# Write your explanation here

# If we choose an extremely high dimension for the latent space, it may lead to
# overfitting. This can lead to a poor performance on the test set.
# Additionally, it may also cause computational issues as the model becomes more
# difficult to train and requires a larger amount of data and computational resources.
```

Did you output blurry reconstructed images? If the answer is yes, explain what could be the reason. If you got sharp edges and fine details, explain what you did in order to achieve that.

Note: If you got blurry reconstructed images, just explain why. You dont need to change your code or retrain your model for better results (as long as your results can be interpreted as a human face).

In [49]:

```
# Write your explanation here
# Blurry reconstructed images can be caused by a number of factors, one of which
# could be a low-dimensional latent space. Another reason could be that the model
# is not trained for enough steps or with enough data, which can also lead to poor
# performance. in our model we choose a small number of epochs due to a long
# running time with our resourses.
```

Question 6: Generate New Faces (10 %)

Now, for the fun part!

We are going to generate new celebrity faces with our VAE models. A function for new faces generation is given to you. Modify it (if needed) to fit your code.

In [122]:

```
# creates random noise sample in the correct shape.
def generate_faces(model, grid_size, latent):
    model.eval()
    dummy = torch.empty([batch_size,latent])
    z = torch.randn_like(dummy).to(device)
    if latent == 3:
      for i in range(grid_size):
        n = random.randint(2,5)
        z[i] = n*z[i]
    else:
      for i in range(grid_size):
        n = random.randint(7,10)
        z[i] = n*z[i]

    #insert the random noise to the decoder to create new samples.
    sample = model.decode(z).cpu()
    new_face_list = [sample[j].detach().numpy() for j in range(grid_size)]

    show(new_face_list)
```

Model 1 ($L = 3$) results:

In [62]:

```
generate_faces(model_1,grid_size=32,latent=latent1)
```

Model 2 results:

In [124]:

```
generate_faces(model_2,grid_size=32,latent=latent2)
```

Q1: Generate new faces with VAE model with latent space dimension = 3. Did you get diverse results? What are the most prominent features that the latent space capture?

Q2: Generate new faces with VAE model with your decision for latent space dimension. What are the most prominent features that the latent space capture?

Q3: What are the differences? Your results are similar to the dataset images? Do you get realistic images for your chosen latent space dimension? If not, change your decision or your network to acheive more realistic results.

In [99]:

```
#YOUR ANSWERS GOES HERE

# Q1 Answer:
# We didn't get diverse results, the images look the same. The most prominent features
# that the latent space capture are outline of the face, eyes, mouth and nose.
# Q2 Answer:
# The most prominent features that the latent space capture are outline of the
# face shape, hair shape and color.
# Q3 Answer:
# The differences are the details in the images, in the first model, we get the
# inside element of the face more shaper, and on the other hand, the second model
# is showen the outside element more shaper
```

Question 7: Extrapolation (10 %)

Recall that we extrapolate in the images domain in Question 2, part (c). Here, extrapolate in the latent space domain to generate new images.

Define $\beta = [0, 0.1, 0.2, \ldots, 0.9, 1]$ and randomly sample from $Z \sim \mathcal{N}(0, 1)$ 2 different samples and generate 2 new face images: X_1, X_2.

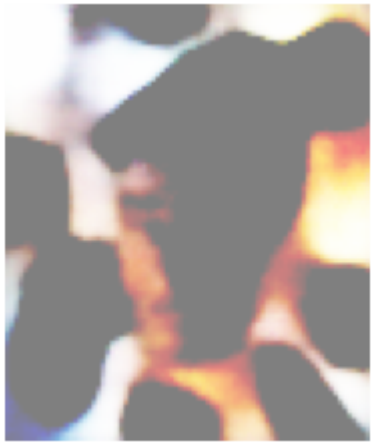
Extrapolate in the latent domain as follows: $\beta_i \cdot Z_1 + (1 - \beta_i) \cdot Z_2$ for each $\beta_i \in \beta$.

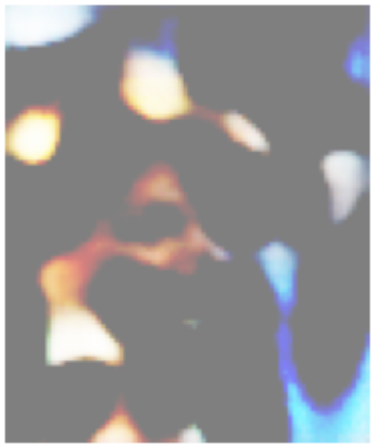
Plot the extrapolation of the images for each β and discuss your results. Repeat the process for 3 different samples.

In [105]:

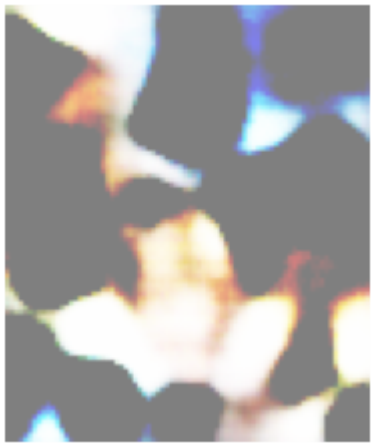
```
# YOUR CODE GOES HERE

def show2(imgs):
    plt.figure(figsize=(15,15))
    for i in range(2):
        for j in range(5):
            if not isinstance(imgs[i*5 + j], torch.Tensor):
                imgs[i*5 + j] = torch.Tensor(imgs[i*5 + j])
            plt.subplot(2, 5, i*5 + j + 1)
            plt.imshow(imgs[i*5 + j].transpose(0,2).transpose(0,1))
            plt.axis('off')
    plt.show()


beta_lst = [0, 0.1,0.2,0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

model_2.eval()

dummy = torch.empty([batch_size,latent2])
z = torch.randn_like(dummy).to(device)
for i in range(32):
  n = random.randint(7,10)
  z[i] = n*z[i]
sample = model_2.decode(z).cpu()
for i in range(3):
  print('sample={}'.format(i+1))
  x1_index = random.randint(0,batch_size-1)
  x2_index = random.randint(0,batch_size-1)
  x1 = sample[x1_index].detach().numpy()
  x2 = sample[x2_index].detach().numpy()

  x1 = torch.Tensor(x1)
  x2 = torch.Tensor(x2)

  x_lst = []
  plt.imshow(x1.transpose(0,2).transpose(0,1))
  plt.axis('off')
  plt.show()
  plt.imshow(x2.transpose(0,2).transpose(0,1))
  plt.axis('off')
  plt.show()
  for b in beta_lst:
    x = b*x1 + (1-b)*x2
    x_lst.append(x)
  show2(x_lst)
```

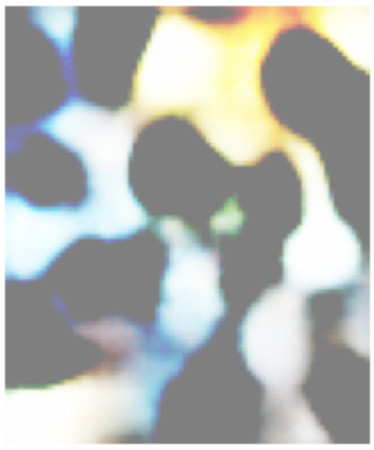
sample=1

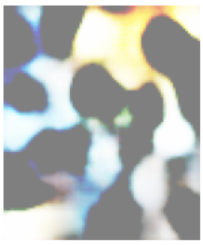
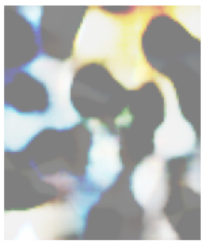
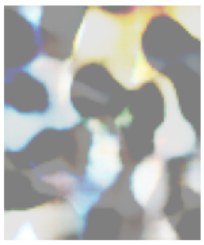
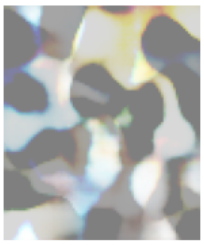
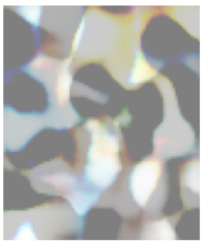


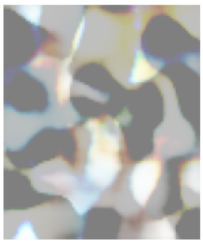
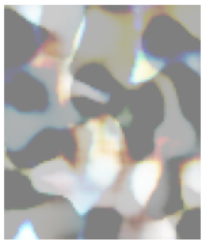
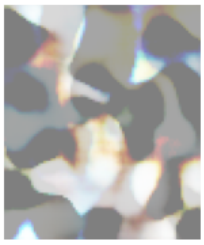
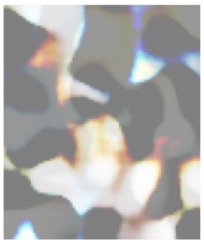
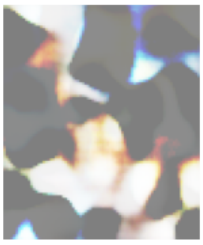


sample=2





```
sample=3
```