Video Interpolation

Python 3.7.4
Pytorch 1.2, numpy, pandas

In [ ]:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import torch.nn.functional as F
import torch.utils as utils
import torch.utils.data as data
import torchvision
import torchvision.transforms as transforms
import torchvision.transforms.functional as TF
import torchvision.datasets as dsets
from torch.utils.data import Dataset, DataLoader
from torchvision import models
MVN = torch.distributions.MultivariateNormal

import gpytorch

import cv2
from skimage import io

import matplotlib.pyplot as plt

import time
import copy
import random as rd
import numpy.random as nprd
import collections
import sys
import glob
import os
os.chdir("/mnt/juhyeong/projects/2019연구학점제/")
os.environ["CUDA_VISIBLE_DEVICES"] = "0"

#codes.py
sys.path.insert(0, '../')
import codes
import UCF101 as UCF101

device = torch.device('cuda')
nprd.seed(0)
rd.seed(0)
```

In [ ]:

```
def populateTrainList(folderPath, train = True):
    folderList_pre = [x[0] for x in os.walk(folderPath)]
    folderList = []
    dataList = []
    nprd.seed(0)
    rd.seed(0)

    for folder in folderList_pre:
        if folder[-3:] == '240':
            folderList.append(folder + "/" + folder.split("/")[-2])

    dirList = nprd.choice(folderList, int(len(folderList) * 0.2), replace=False)

    if not train:
        folderList = [x for x in folderList if x not in dirList]
        folderList = nprd.choice(folderList, int(len(folderList) * 0.2))

    for folder in folderList:
```

```
    for folder in folderList:
        imageList = sorted(glob.glob(folder + '/' + '*.jpg'))
        for i in range(0, len(imageList), 12):
            tmp = imageList[i:i+12]
            if len(tmp) == 12:
                dataList.append(imageList[i:i+12])

    return dataList

def randomCropOnList(image_list, output_size):

    cropped_img_list = []

    h,w = output_size
    height, width, _ = image_list[0].shape

    i = rd.randint(0, height - h)
    j = rd.randint(0, width - w)

    st_y = 0
    ed_y = w
    st_x = 0
    ed_x = h

    or_st_y = i
    or_ed_y = i + w
    or_st_x = j
    or_ed_x = j + h

    #print(st_x, ed_x, st_y, ed_y)
    #print(or_st_x, or_ed_x, or_st_y, or_ed_y)


    for img in image_list:
        new_img = np.empty((h,w,3), dtype=np.float32)
        new_img.fill(128)
        new_img[st_y: ed_y, st_x: ed_x, :] = img[or_st_y: or_ed_y, or_st_x: or_ed_x, :].copy()
        cropped_img_list.append(np.ascontiguousarray(new_img))


    return cropped_img_list

def randomCropOnBatch(image, output_size):

    h,w = output_size
    height, width, channel, frame = image.shape

    i = rd.randint(0, height - h)
    j = rd.randint(0, width - w)

    st_y = 0
    ed_y = w
    st_x = 0
    ed_x = h

    or_st_y = i
    or_ed_y = i + w
    or_st_x = j
    or_ed_x = j + h

    new_img = np.empty((h, w, channel, frame), dtype=np.float32)
    new_img.fill(128)
    new_img[st_y: ed_y, st_x: ed_x, :, :] = image[or_st_y: or_ed_y, or_st_x: or_ed_x, :, :].copy()

    return np.ascontiguousarray(new_img)


class expansionLoader(data.Dataset):

    def __init__(self, folderPath, train = True):

        self.trainList = populateTrainList(folderPath, train)
        print("# of training samples:", len(self.trainList))


    def __getitem__(self, index):

        image_path_list = self.trainList[index]
        img_path_list = self.trainList[index]
        start = rd.randint(0,3)
        h,w,c = cv2.imread(img_path_list[0]).shape

        image = cv2.cv2.imread(img_path_list[0])


        img_list = []

        flip = rd.randint(0,1)
        if flip:
            for img_path in img_path_list[start:start+9]:
                tmp = cv2.resize(cv2.imread(img_path), (64, 64))[:,:,(2,1,0)]
                img_list.append(np.array(cv2.flip(tmp,1), dtype=np.float32))
        else:
            for img_path in img_path_list[start:start+9]:
                tmp = cv2.resize(cv2.imread(img_path), (64, 64))[:,:,(2,1,0)]
                img_list.append(np.array(tmp,dtype=np.float32))
        img = np.stack(img_list, axis = 3)

        img /= 255
        img[:,:,0,:] -= 0.485 #(img_list[i]/127.5) - 1
        img[:,:,1,:] -= 0.456
        img[:,:,2,:] -= 0.406

        img[:,:,0,:] /= 0.229
        img[:,:,1,:] /= 0.224
        img[:,:,2,:] /= 0.225

        return torch.from_numpy(img.transpose((3, 2, 0, 1)))

"""
        if h > w:
            scaleX = int(320*(h/w))
            scaleY = 320
        elif h <= w:
            scaleX = 320
            scaleY = int(320*(w/h))

        img_list = []

        flip = rd.randint(0,1)
        if flip:
            for img_path in img_path_list[start:start+9]:
                tmp = cv2.resize(cv2.imread(img_path), (scaleX,scaleY))[:,:,(2,1,0)]
                img_list.append(np.array(cv2.flip(tmp,1), dtype=np.float32))
        else:
            for img_path in img_path_list[start:start+9]:
                tmp = cv2.resize(cv2.imread(img_path), (scaleX, scaleY))[:,:,(2,1,0)]
                img_list.append(np.array(tmp,dtype=np.float32))

        img = randomCropOnBatch(np.stack(img_list, axis = 3), (256,256))


"""


    def __len__(self):
        return len(self.trainList)
```

In [9]:

```
batch_size = 4
train_loader = DataLoader(expansionLoader('/mnt/ssd0/datasets/adobe240fps/', train = True),
                          batch_size=batch_size,
                          shuffle=True, num_workers=18,
                          pin_memory=True)
test_loader = DataLoader(expansionLoader('/mnt/ssd0/datasets/adobe240fps/', train = False),
                          batch_size=batch_size,
                          shuffle=True, num_workers=18,
                          pin_memory=True)
print(len(train_loader))
print(len(test_loader))
```

```
# of training samples: 31870
# of training samples: 6706
# of training samples: 6766
7968
1677
```

In [7]:

```
%%timeit
for img in train_loader:
    break
```

```
2.2 s ± 131 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [10]:

```
for img in train_loader:
    break
img.shape
```

Out[10]:

```
torch.Size([4, 9, 3, 64, 64])
```

In [ ]:

```
train = adobe240fps(train=True)
print(train[0].shape)
#train_one = adobe240fps_one_video(train=True)
#print(train_one[0][0].shape)
```

In [ ]:

```
plt.imshow(train[0][0].transpose(0,2).numpy())
plt.show()
```

In [ ]:

```
batch_size = 16
train_loader = DataLoader(dataset = adobe240fps(train=True),
                          batch_size=batch_size,
                          shuffle=True,
                          num_workers=0)
test_loader = DataLoader(dataset = adobe240fps(train=False),
                         batch_size=batch_size,
                         shuffle=True,
                         num_workers=0)
print(len(train_loader))
print(len(test_loader))
```

In [10]:

```
train_loader_one_video = DataLoader(dataset = adobe240fps_one_video(train=True),
                          batch_size=32,
                          shuffle=True,
                          num_workers=0)
test_loader_one_video = DataLoader(dataset = adobe240fps_one_video(train=False),
                          batch_size=32,
                          shuffle=True,
                          num_workers=0)
```

In [7]:

```
for img in train_loader:
    break
img.shape
```

Out[7]:

```
torch.Size([16, 9, 3, 224, 224])
```

In [20]:

```
%%timeit
for img in train_loader:
    break
```

```
3.53 s ± 382 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [11]:

```
def consistent_img(img):
    return torch.cat((img[:,0:1,:,:,:], img[:,-1:,:,:,:]), 1)

def consistent_label(x):
    return torch.stack((x[:,0], x[:,-1]), 1).unsqueeze(2)

def inconsistent_img(img):
    return img[:,1:-1,:,:,:]

def inconsistent_label(x):
    return x[:,1:-1].unsqueeze(2)

def new_latent(mean, logvar):
    return torch.randn(mean.shape).cuda().mul(logvar.exp()).add(mean)
```

In [2]:

```
class VAE(nn.Module):
    def __init__(self, latent_dim):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim

        #####Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 4, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(4),
            nn.LeakyReLU(0.1),
            # 4 x 32 x 32

            nn.Conv2d(4, 8, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(0.1),
            # 8 x 16 x 16

            nn.Conv2d(8, 16, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(0.1),
            # 16 x 8 x 8

            nn.Conv2d(16, 32, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.1),
            # 32 x 4 x 4

            nn.Conv2d(32, 64, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.1),
            # 64 x 2 x 2

            nn.Conv2d(64, 64, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.1),
            # 64 x 1 x 1

            codes.Flatten()
        )



        # Linear layers
        num_features = 64
        self.mean_z = nn.Linear(num_features, latent_dim, bias = True)
        self.logvar_z = nn.Linear(num_features, latent_dim, bias = True)
        #####Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, num_features, bias = True),

            codes.UnFlatten(),

            nn.ConvTranspose2d(64, 32, kernel_size = 2, bias = False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(32, 32, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(32, 16, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(16, 8, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(8, 4, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(4),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(4, 3, kernel_size = 2, stride = 2, bias = False),
            nn.BatchNorm2d(3),
            nn.LeakyReLU(0.1),

        )


        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.constant_(m.bias, 0)

    def decode(self, x):
        out = self.decoder(x.view(-1, self.latent_dim))
        return out.view(batch_size, -1, out.shape[1], out.shape[2], out.shape[3])

    def reparameterize(self, x):
        start = self.encoder(x[:,0,:,:,:])
        end = self.encoder(x[:,1,:,:,:])

        return self.mean_z(start), self.mean_z(end), self.logvar_z(start), self.logvar_z(end)


    def forward(self, x):
        mean_z_start, mean_z_end, logvar_z_start, logvar_z_end = self.reparameterize(x)
        var_z_start = logvar_z_start.mul(0.5).exp()
        var_z_end = logvar_z_end.mul(0.5).exp()

        if next(self.parameters()).is_cuda:
            latent_start = torch.randn(self.latent_dim).cuda().mul(var_z_start).add(mean_z_start)
            latent_end = torch.randn(self.latent_dim).cuda().mul(var_z_start).add(mean_z_start)
        else:
            latent_start = torch.randn(self.latent_dim).mul(var_z_start).add(mean_z_start)
            latent_end = torch.randn(self.latent_dim).mul(var_z_start).add(mean_z_start)

        return (torch.stack((self.decoder(latent_start), self.decoder(latent_end)), 1),
                torch.stack((mean_z_start, mean_z_end), 1),
                torch.stack((logvar_z_start, logvar_z_end), 1),
                torch.stack((latent_start, latent_end), 1))
```

In [ ]:

```
model = VAE(latent_dim = 64)
print(img.shape)
print(model.encoder(img[:,0,:,:,:])[0].shape)
print(model.forward(consistent_img(img))[0][:,0,:,:,:].shape)
print(model.forward(consistent_img(img))[0][:,0,:,:,:].shape)
print([x.numel() for x in model.parameters()])
print(sum([x.numel() for x in model.parameters()]))
```

In [14]:

```
def loss(recon, recon_gt, mean, logvar):
    # BCE = F.binary_cross_entropy(output, gt, reduction='sum')
    BCE = F.mse_loss(recon, recon_gt, reduction='mean')
    KLD = -0.25 * torch.mean(1 + logvar - mean.pow(2) - logvar.exp())

    return (BCE + KLD).mean(), BCE, KLD
```

In [15]:

```
model = VAE(latent_dim = 64).cuda()
epoch = 100
lr = 3e-4
optimizer = torch.optim.Adam(model.parameters(), lr = lr)
#optimizer = torch.optim.Adam(list(model.parameters()) + list(GP_model.parameters()), lr = lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.1)
```

In [ ]:

```
#VAE pretraining
running_test_loss = 0.0

for run in range(epoch):
    start = time.time()

    #Training
    model.train()
    batch_train_loss = 0.0

    for ind, data in enumerate(train_loader):
        img = data
        optimizer.zero_grad()
        recon, mean_z, logvar_z, latent = model(consistent_img(img).cuda())
        recon_new = model.decode(latent)

        train_loss, BCE, KLD = loss(recon, consistent_img(img).cuda(), mean_z, logvar_z)
        train_loss.backward()
        batch_train_loss += train_loss.item()

        optimizer.step()
        scheduler.step()

    batch_train_loss /= len(train_loader)

    #Test
    model.eval()
    with torch.no_grad():
        batch_test_loss = 0.0
        for ind, data in enumerate(test_loader):
            img = data
            recon, mean_z, logvar_z, latent = model(consistent_img(img).cuda())
            recon_new = model.decode(latent)

            test_loss, BCE, KLD = loss(recon, consistent_img(img).cuda(), mean_z, logvar_z)
            batch_test_loss += test_loss.item()

        batch_test_loss /= len(test_loader)
        running_test_loss += batch_test_loss

    print("epoch : %d, train loss = %5.5f, test loss = %5.5f, running test loss = %5.5f, time: %.2
sec"
          %(run, batch_train_loss, batch_test_loss, running_test_loss/(run + 1), time.time() - star
t))
```

Combined</h1>

In [45]:

```python
class VAE(nn.Module):
    def __init__(self, latent_dim=64, if_device = False):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim
        self.if_device = if_device

        #####Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 4, kernel_size = 4, stride = 3, bias = False),
            nn.BatchNorm2d(4),
            nn.LeakyReLU(0.1),
            # 4 x 85 x 85

            nn.Conv2d(4, 8, kernel_size = 4, stride = 3, bias = False),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(0.1),
            # 8 x 28 x 28

            nn.Conv2d(8, 16, kernel_size = 4, stride = 3, bias = False),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(0.1),
            # 16 x 9 x 9

            nn.Conv2d(16, 32, kernel_size = 3, stride = 2, bias = False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.1),
            # 32 x 4 x 4

            nn.Conv2d(32, 32, kernel_size = 3, bias = False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.1),
            # 32 x 2 x 2

            nn.Conv2d(32, 64, kernel_size = 2, bias = False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.1),
            # 32 x 1 x 1

            codes.Flatten()
        )



        # Linear layers
        num_features = 64
        self.mean_z = nn.Linear(num_features, latent_dim, bias = True)
        self.logvar_z = nn.Linear(num_features, latent_dim, bias = True)

        #####Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, num_features, bias = True),

            codes.UnFlatten(64),

            nn.ConvTranspose2d(64, 32, kernel_size = 2, bias = False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(32, 32, kernel_size = 3, bias = False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(32, 16, kernel_size = 3, stride = 2, bias = False),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(16, 8, kernel_size = 4, stride = 3, bias = False),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(8, 4, kernel_size = 4, stride = 3, bias = False),
            nn.BatchNorm2d(4),
            nn.LeakyReLU(0.1),

            nn.ConvTranspose2d(4, 3, kernel_size = 4, stride = 3, bias = False),
            nn.BatchNorm2d(3),
            nn.LeakyReLU(0.1),
        )


        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.constant_(m.bias, 0)

    def decode(self, x):
        out = self.decoder(x.view(-1, self.latent_dim))
        return out.view(batch_size, -1, out.shape[1], out.shape[2], out.shape[3])

    def reparameterize(self, x):
        start = self.encoder(x[:,0,:,:,:])
        end = self.encoder(x[:,1,:,:,:])

        return self.mean_z(start), self.mean_z(end), self.logvar_z(start), self.logvar_z(end)


    def forward(self, x):
        mean_z_start, mean_z_end, logvar_z_start, logvar_z_end = self.reparameterize(x)
        var_z_start = logvar_z_start.mul(0.5).exp()
        var_z_end = logvar_z_end.mul(0.5).exp()

        if next(self.parameters()).is_cuda:
            latent_start = torch.randn(self.latent_dim).cuda().mul(var_z_start).add(mean_z_start)
            latent_end = torch.randn(self.latent_dim).cuda().mul(var_z_start).add(mean_z_start)
        else:
            latent_start = torch.randn(self.latent_dim).mul(var_z_start).add(mean_z_start)
            latent_end = torch.randn(self.latent_dim).mul(var_z_start).add(mean_z_start)

        return (torch.stack((self.decoder(latent_start), self.decoder(latent_end)), 1),
                torch.stack((mean_z_start, mean_z_end), 1),
                torch.stack((logvar_z_start, logvar_z_end), 1),
                torch.stack((latent_start, latent_end), 1))
```

Multi-output Gaussian Process

Kernel function: $ k(\mathrm{x}_{n}, \mathrm{x}_{m}) = \theta_{0} \mathrm{exp}\left\{-{1 \over 2}\sum_{i=1}^D\eta_i(x_{n} - x_{m})^2\right\} + \theta_{2} + \theta_{3}\sum_{i=1}^Dx_{ni}x_{mi} $

In [46]:

```
class MOGP(nn.Module):
    def __init__(self, input_dim, output_dim, batch_size):
        super(MOGP, self).__init__()
        self.batch_size = batch_size
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.num_data = 0

        B = batch_size
        N = 0
        self.input_data = torch.empty(B, N, self.input_dim)
        self.output_data = torch.empty(B, N, self.output_dim)
        self.dot_prod = torch.empty(B, self.input_dim, N, N)
        self.dist = torch.empty(B, self.input_dim, N, N)

        self.ARD_params = nn.Parameter(torch.randn(self.input_dim, self.output_dim))
        self.kernel_params = nn.ParameterDict({
            "scale_ARD":nn.Parameter(torch.randn(output_dim)),
            "scale_prod":nn.Parameter(torch.randn(output_dim)),
            "constant":nn.Parameter(torch.randn(output_dim)),
        })
        self.noise_homo = nn.Parameter(torch.zeros(output_dim))
    def cal_stat(self):
        N = self.num_data
        D = self.input_dim
        B = self.batch_size
        data = self.input_data.transpose(1, 2)

        self.dot_prod = torch.einsum("bdm,bdn->bdmn", data, data)
        self.dist = ((data ** 2).unsqueeze(3).expand(B, D, N, N)
                     + (data ** 2).unsqueeze(2).expand(B, D, N, N)
                     - 2 * self.dot_prod)


    def kernel_fitted(self):
        self.cal_stat()
        gram = (-0.5 * torch.einsum("bdmn,do->bomn", self.dist, F.softplus(self.ARD_params))).exp()
        gram = torch.einsum("bdmn,d->bdmn", gram, F.softplus(self.kernel_params["scale_ARD"]))
        gram += torch.einsum("bmn,d->bdmn", self.dot_prod.sum(1), F.softplus(self.kernel_params["sc
ale_prod"]))
        gram += F.softplus(self.kernel_params["constant"]).unsqueeze(1).unsqueeze(2)
        return gram


    def kernel_new(self, x):
        N = self.num_data
        D = self.input_dim
        M = x.shape[1]
        B = self.batch_size

        fitted_data = self.input_data.transpose(1, 2)
        new_data = x.transpose(1, 2)

        new_dot_prod_wing = torch.einsum("bdn,bdt->bdnt", fitted_data, new_data)
        new_dist_wing = ((fitted_data ** 2).unsqueeze(3).expand(B, D, N, M)
                     + (new_data ** 2).unsqueeze(2).expand(B, D, N, M)
                     - 2 * new_dot_prod_wing)

        wing = (-0.5 * torch.einsum("bdmn,do->bomn", new_dist_wing, F.softplus(self.ARD_params))).e
xp()
        wing = torch.einsum("bdmn,d->bdmn", wing, F.softplus(self.kernel_params["scale_ARD"]))
        wing += torch.einsum("bmn,d->bdmn", new_dot_prod_wing.sum(1), F.softplus(self.kernel_params
["scale_prod"]))
        wing += F.softplus(self.kernel_params["constant"]).unsqueeze(1).unsqueeze(2)


        new_dot_prod_cov = torch.einsum("bdn,bdt->bdnt", new_data, new_data)
        new_dist_cov = ((new_data  ** 2).unsqueeze(3).expand(B, D, M, M)
                     + (new_data ** 2).unsqueeze(2).expand(B, D, M, M)
                     - 2 * new_dot_prod_cov)

        cov = (-0.5 * torch.einsum("bdmn,do->bomn", new_dist_cov, F.softplus(self.ARD_params))).exp
()
        cov = torch.einsum("bdmn,d->bdmn", cov, F.softplus(self.kernel_params["scale_ARD"]))
        cov += torch.einsum("bmn,d->bdmn", new_dot_prod_cov.sum(1), F.softplus(self.kernel_params["
scale_prod"]))
        cov += F.softplus(self.kernel_params["constant"]).unsqueeze(1).unsqueeze(2)

        return wing, cov


    def NLL(self):
        N = self.num_data
        B = self.batch_size

        self.train()
        gram = self.kernel_fitted()
        if next(self.parameters()).is_cuda:
            gram += torch.einsum("d,bmn->bdmn", F.softplus(self.noise_homo), torch.eye(N).cuda().ex
pand(B, N, N))
        else:
            gram += torch.einsum("d,bmn->bdmn", F.softplus(self.noise_homo), torch.eye(N).expand(B,
N, N))

        #Removed constant term from nll
        nll = torch.stack([x.squeeze().logdet() for x in gram.view(-1, gram.shape[2], gram.shape[3]
).split(1,0)], 0).view(B, -1).sum(1)
        nll += torch.einsum("bmt,bdmn,bnt->bd", self.output_data, gram.inverse(), self.output_data)
.sum(1)
        nll *= 0.5
        return nll


    def fit(self, x, y):
        if next(self.parameters()).is_cuda:
            self.input_data = x.cuda()
            self.output_data = y.cuda()
        else:
            self.input_data = x
            self.output_data = y
        self.num_data = self.input_data.shape[1]


    def forward(self, x):
        self.eval()
        N = self.num_data
        B = self.batch_size
        gram = self.kernel_fitted()

        if next(self.parameters()).is_cuda:
            gram += torch.einsum("d,bmn->bdmn", F.softplus(self.noise_homo), torch.eye(N).cuda().ex
pand(B, N, N))
            x = x.cuda()
        else:
            gram += torch.einsum("d,bmn->bdmn", F.softplus(self.noise_homo), torch.eye(N).expand(B,
N, N))

        gram_inv = torch.cat([x.inverse() for x in gram.split(1,0)], 0)

        wing, cov = self.kernel_new(x)

        mean = torch.einsum("btnm,btnk,bkt->btm", wing, gram_inv, self.output_data)
        var = cov - torch.einsum("btnm,btnk,btkl->btml", wing, gram_inv, wing)
        return mean, var
```

In [ ]:

```
GP_model = MOGP(input_dim = 32, output_dim = 50, batch_size = batch_size)
x = torch.randn(16, 100, 32)
GP_model.fit(x, torch.randn(16, 100, 50) * 0.01)
GP_model.cal_stat()
print(GP_model.dist.shape)
print(GP_model.kernel_fitted().shape)
for name, param in GP_model.named_parameters():
    if not param.requires_grad:
        print(f"{name} has no grad")

with torch.autograd.detect_anomaly():
    print(GP_model.NLL())
    GP_model.NLL().sum().backward()
    for name, param in GP_model.named_parameters():
        print(f"{name} = {param.grad.min()} ~ {param.grad.max()}")

x_new = torch.randn(16, 10, 32)
mean, var = GP_model(x_new)
print(mean.shape)
print(var.shape)
```

In [13]:

```
GP_model = MOGP(input_dim = 1, output_dim = 64, batch_size = batch_size).cuda()
model = VAE().cuda()
```

In [14]:

```
def loss(recon, recon_gt, recon_new, recon_new_gt, mean, logvar, GP_model):
    # BCE = F.binary_cross_entropy(output, gt, reduction='sum')
    BCE = F.mse_loss(recon, recon_gt, reduction='mean')
    BCE_new = F.mse_loss(recon_new, recon_new_gt, reduction='mean')
    KLD = -0.25 * torch.mean(1 + logvar - mean.pow(2) - logvar.exp())
    fit = GP_model.NLL()

    return (BCE + BCE_new + KLD + fit).sum(), BCE, BCE_new, KLD
```

In [21]:

```
epoch = 100
lr = 3e-4
optimizer = torch.optim.Adam(list(model.parameters()) + list(GP_model.parameters()), lr = lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.1)
```

In [ ]:

```
running_test_loss = 0.0
label_base = (torch.arange(9, device = device).float() - 4)/4
label_base = label_base.unsqueeze(0).expand(16, 9)

for run in range(epoch):
    start = time.time()

    #Training
    model.train()
    batch_train_loss = 0.0
    scheduler.step()
    for ind, data in enumerate(train_loader):
        img = data
        label = label_base + torch.clamp(torch.randn(batch_size, 9, device = device) / 12, min = -0.
25, max = 0.25)

        optimizer.zero_grad()
        recon, mean_z, logvar_z, latent = model(consistent_img(img).cuda())
        GP_model.fit(consistent_label(label).cuda(), latent)
        pred_mean, pred_var_cholesky = GP_model(inconsistent_label(label).cuda())
        recon_new = model.decode(torch.einsum("bdmn,bdn->bdm", pred_var_cholesky, torch.randn(batch
_size, 64, 7, device=device)) + pred_mean)

        train_loss, BCE, BCE_new, KLD = loss(recon, consistent_img(img).cuda(), recon_new,
inconsistent_img(img).cuda(), mean_z, logvar_z, GP_model)
        train_loss.backward()
        batch_train_loss += train_loss.item()
        optimizer.step()

    batch_train_loss /= len(train_loader)

    #Test
    model.eval()
    with torch.no_grad():
        batch_test_loss = 0.0
        for ind, data in enumerate(test_loader):
            img = data
            label = label_base + torch.clamp(torch.randn(batch_size, 9) / 12, min = -0.25, max = 0.2
5)

            optimizer.zero_grad()
            recon, mean_z, logvar_z, latent = model(consistent_img(img).cuda())
            pred_mean, pred_var_cholesky = GP_model(inconsistent_label(label).cuda())
            recon_new = model.decode(torch.einsum("bdmn,bdn->bdm", pred_var_cholesky, torch.randn(b
atch_size, 64, 7, device=device)) + pred_mean)
            train_loss, BCE, BCE_new, KLD = loss(recon, consistent_img(img).cuda(), recon_new,
inconsistent_img(img).cuda(), mean_z, logvar_z, GP_model)

        batch_test_loss /= len(test_loader)
        running_test_loss += test_loss.item()

    print("epoch : %d, train loss = %5.5f, test loss = %5.5f, running test loss = %5.5f, time: %.2:
sec"
          %(run, batch_train_loss, batch_test_loss, running_test_loss/(run + 1), time.time() - star
t))
```

In [ ]:

```
!jupyter nbconvert --to script test2_VAE_reg.ipynb
```