1

1. LSTM applied to SMILES string generation. (10 pt) Using the SMILES string from the ANI dataset with upto 6 heavy atoms, build a LSTM generative model that can generate new smiles string with given initial character.

(a) (3pt) Process the smiles strings from ANI dataset by adding a starting character at the beginning and an ending character at the end. Look over the dataset and define the vocabulary, use one hot encoding to encode your smiles strings.

In [1]:
```
from sklearn.preprocessing import OneHotEncoder
```

In [2]:

```
from ANI1_release.ANI1_release.readers.lib.pyanitools import anidataloader
# data = anidataloader("../../ANI1_dataset/ANI-1_release/ani_gdb_s01.h5")
data = anidataloader("ANI1_release/ANI1_release/ani_gdb_s01.h5")
data_iter = data.__iter__()

mols = next(data_iter)
# Extract the data
P = mols['path']
X = mols['coordinates']
E = mols['energies']
S = mols['species']
sm = mols['smiles']

# Print the data
print("Path:   ", P)
print("  Smiles:      ","".join(sm))
print("  Symbols:     ", S)
print("  Coordinates: ", X.shape)
print("  Energies:    ", E.shape, "\n")




data_iter = data.__iter__()
count=0
count_conf =0
for mol in data_iter:
    count+=1
    count_conf += len(mol['energies'])
print(count)
print(count_conf)
```

```
Path:    /gdb11_s01/gdb11_s01-0
  Smiles:       [H]C([H])([H])[H]
  Symbols:      ['C', 'H', 'H', 'H', 'H']
  Coordinates:  (5400, 5, 3)
  Energies:     (5400,)

3
10800
```

In [3]:
```
import ANI1_release.ANI1_release.readers.lib.pyanitools as pya
import torch; torch.manual_seed(0)
import torch.nn as nn
import torch.nn.functional as F

import numpy as np

# Set the HDF5 file containing the data
hdf5file = 'ANI1_release/ANI1_release/ani_gdb_s01.h5'

# Construct the data loader class
adl = pya.anidataloader(hdf5file)

# Print the species of the data set one by one
seq=[]

for data in adl:
#     print(data.keys())


    # Extract the data
    P = data['path']
    X = data['coordinates']
    E = data['energies']
    S = data['species']
    sm = data['smiles']

    # Print the data
    print("Path:   ", P)
    print("  Smiles:      ","".join(sm))
    print("  Symbols:     ", S)
    print("  Coordinates: ", X.shape)
    print("  Energies:    ", E.shape, "\n")

    seq.append(data)

# Closes the H5 data file
adl.cleanup()
smiles=[]
for i in range(count):
    smiles.append(seq[i]['smiles'])


species=[]
for i in range(count):
    species.append(seq[i]['species'])
```

```
Path:    /gdb11_s01/gdb11_s01-0
  Smiles:       [H]C([H])([H])[H]
  Symbols:      ['C', 'H', 'H', 'H', 'H']
  Coordinates:  (5400, 5, 3)
  Energies:     (5400,)

Path:    /gdb11_s01/gdb11_s01-1
  Smiles:       [H]N([H])[H]
  Symbols:      ['N', 'H', 'H', 'H']
  Coordinates:  (3600, 4, 3)
  Energies:     (3600,)

Path:    /gdb11_s01/gdb11_s01-2
  Smiles:       [H]O[H]
  Symbols:      ['O', 'H', 'H']
  Coordinates:  (1800, 3, 3)
  Energies:     (1800,)
```

(b) (7pt) Build a LSTM model with 1 recurrent layer. Starting with the starting character and grow a string character by character using model prediction until it reaches a ending character. Look at the string you grown, is it a valid SMILES string?

In [4]:
```
class CharRNN(nn.Module):

    def __init__(self, chars, n_hidden=10, n_layers=2,drop_prob=0.1, lr=0.00
        super().__init__()
        self.drop_prob = drop_prob
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.lr = lr

        self.chars = chars
        self.int_char = dict(enumerate(self.chars))
        self.char_int = {ch: ii for ii, ch in self.int_char.items()}
        #define the LSTM
        self.lstm = nn.LSTM(len(self.chars), n_hidden, n_layers, dropout=dro
        #define a dropout layer
        self.dropout = nn.Dropout(drop_prob)
        #define the final, fully-connected output layer
        self.fc = nn.Linear(n_hidden, len(self.chars))

    def forward(self, x, hidden):
        ''' Forward pass through the network. These inputs are x, and the hi
        r_output, hidden = self.lstm(x, hidden)

        #pass through a dropout layer
        out = self.dropout(r_output)
        # Stack up LSTM outputs using view
        out = out.contiguous().view(-1, self.n_hidden)
        #put x through the fully-connected layer
        out = self.fc(out)
        return out, hidden

    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        weight = next(self.parameters()).data
        hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_

        return hidden
```

In [5]:
```
def one_hot_encode(arr, n_labels):
    one_hot_enc = np.zeros((np.multiply(*arr.shape), n_labels), dtype=np.flo
    one_hot_enc[np.arange(one_hot_enc.shape[0]), arr.flatten()] = 1.
    one_hot_enc = one_hot_enc.reshape((*arr.shape, n_labels))
    return one_hot_enc
```

In [6]:
```
def batches_gen(arr, batch_size, seq_length):
    '''Create a generator that returns batches of size
       batch_size x seq_length from arr.

       Arguments
       ---------
       arr: Array you want to make batches from
       batch_size: Batch size, the number of sequences per batch
       seq_length: Number of encoded chars in a sequence
    '''

    batch_size_total = batch_size * seq_length
    # total number of batches we can make
    n_batches = len(arr)//batch_size_total
    arr = arr[:n_batches * batch_size_total]
    arr = arr.reshape((batch_size, -1))
    for n in range(0, arr.shape[1], seq_length):
        # The features
        x = arr[:, n:n+seq_length]
        # The targets, shifted by one
        y = np.zeros_like(x)
        try:
            y[:, :-1], y[:, -1] = x[:, 1:], arr[:, n+seq_length]
        except IndexError:
            y[:, :-1], y[:, -1] = x[:, 1:], arr[:, 0]
        yield x, y
```

In [7]:
```
def predict(net, char, h=None, top_k=None):

    ''' Given a onehot encoded character, predict the next character.
            Returns the predicted onehot encoded character and the hidden st
        Arguments:



            net: the lstm model
            inputs: input to the lstm model. shape (batch, time_step/length_
            h: hidden state (h,c)


            top_k: int. sample from top k possible characters

        '''
    # tensor inputs
    x = np.array([[net.char_int[char]]])
    x = one_hot_encode(x, len(net.chars))
    inputs = torch.from_numpy(x)


    # detach hidden state from history
    h = tuple([each.data for each in h])
    out, h = net(inputs, h)
    p = F.softmax(out, dim=1).data

    if top_k is None:
        top_ch = np.arange(len(net.chars))
    else:
        p, top_ch = p.topk(top_k)
        top_ch = top_ch.numpy().squeeze()

    p = p.numpy().squeeze()

    char = np.random.choice(top_ch, p=p/p.sum())

    return net.int_char[char], h
```

In [8]:
```python
def sample(net, size, prime='e', top_k=None):

    """
    generate a smiles string starting from prime character

    """

    net.eval() # eval mode
    # First off, run through the prime characters
    chars = [ch for ch in prime]
    h = net.init_hidden(1)
    for ch in prime:
        char, h = predict(net, ch, h, top_k=top_k)
        chars.append(char)

    for ii in range(size):
        char, h = predict(net, chars[-1], h, top_k=top_k)
        chars.append(char)
    return ''.join(chars)
```

In [9]:
```python
net=CharRNN(smiles[0])
print(sample(net, 30, prime='C'))
```

```
CH]HH[(C[(HH[][[[HH[)(HH[(HH]][]
```

2

1. Variational Autoencoder(VAE) applied to MNIST dataset.(10 pt)

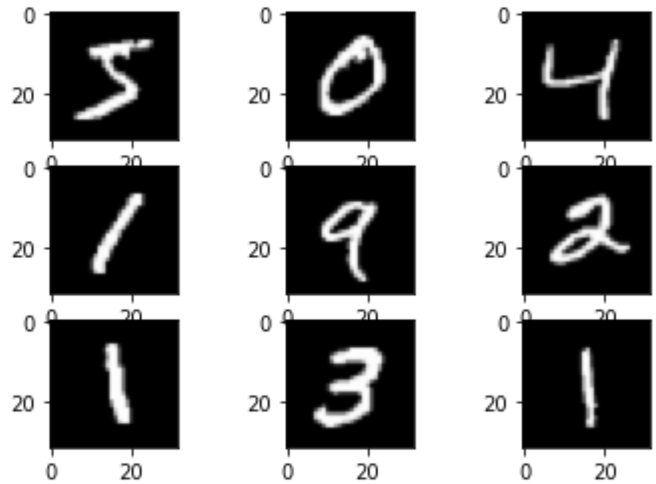
Train an VAE model for the MNIST dataset. The encoder and decoder of the VAE model are convolutional neural networks. Encoder have 4 convolutional layers, each with 4, 8, 16, 32 channels, kernal size of 4x4, padding of 1 and stride of 2. and the decoder is the reverse of that. In the bottleneck region, the encoder output is flattened and mapped to two latent vector μ and σ each represented with 32 hidden neurons by two separate linear layers. Then the latent state z with 32 hidden neurons is formulated by applying reparameterization with addition of noise ϵ, which is then passed to decoder. Use binary cross entropy plus KL divergence as your loss function. Train this model with the MNIST dataset and use the provided reconstruction code to show that your model is able to reproduce the images.

In [10]:
```
import pickle
(train_X, train_y), (test_X, test_y) = pickle.load(open("mnist.pkl", "rb"))

#shape of dataset
print('X_train: ' + str(train_X.shape))
print('Y_train: ' + str(train_y.shape))
print('X_test:  '  + str(test_X.shape))
print('Y_test:  '  + str(test_y.shape))

#plotting
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure()
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(train_X[i], cmap=plt.get_cmap('gray'))
```

```
X_train: (60000, 32, 32)
Y_train: (60000,)
X_test:  (10000, 32, 32)
Y_test:  (10000,)
```



Encoder have 4 convolutional layers, each with 4, 8, 16, 32 channels,

kernal size of 4x4, padding of 1 and stride of 2. and

the decoder is the reverse of that. In the

bottleneck region, the encoder output is flattened and mapped to two latent vector and each represented with 32 hidden neurons by two separate linear layers.

Then the latent state z with 32 hidden neurons is formulated by applying reparameterization with addition of noise , which is then passed to decoder.

Use binary cross entropy plus KL divergence as your loss function.

Train this model with the MNIST dataset and use the provided reconstruction code to show that your model is able to reproduce the images.

In [11]:
```
class VAE_CNN(nn.Module):
    def __init__(self, image_channels, h_dim, z_dim):
        super(VAE_CNN, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(image_channels, 4, kernel_size=4, stride=2, padding=1)
            nn.ReLU(),
            nn.Conv2d(4, 8, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(8, 16, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.Flatten()
        )

        self.fc1 = nn.Linear(h_dim, z_dim)
        self.fc2 = nn.Linear(h_dim, z_dim)
        self.fc3 = nn.Linear(z_dim, h_dim)
        self.unflatten = lambda x: x.view(-1,32,2,2)

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(16, 8, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(8, 4, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(4, image_channels, kernel_size=4, stride=2, p
            nn.Sigmoid(),
        )

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5*logvar)
        # return torch.normal(mu, std)
        epsilon = torch.randn_like(std)
        z = mu + std * epsilon
        return z

    def bottleneck(self, h):
        mu, logvar = self.fc1(h), self.fc2(h)
        z = self.reparameterize(mu, logvar)
        return z, mu, logvar

    def encode(self, x):
        h = self.encoder(x)
        z, mu, logvar = self.bottleneck(h)
        return z, mu, logvar

    def decode(self, z):
        z = self.fc3(z)
        z = self.unflatten(z)
        z = self.decoder(z)
        return z

    def forward(self, x):
        z, mu, logvar = self.encode(x)
        z = self.decode(z)
        return z, mu, logvar
```

In [12]:
```
from torchsummary import summary
```

In [13]:
```
train_X_norm=train_X/255
test_X_norm=test_X/255

from functools import wraps
from time import time

def timing(f):
    @wraps(f)
    def wrap(*args, **kw):
        ts = time()
        result = f(*args, **kw)
        te = time()
        print('func:%r  took: %2.4f sec' % (f.__name__,  te-ts))
        return result
    return wrap
```

In [14]:
```
def data_gen(X,y, batchsize):
    '''
    Generator for data
    '''
    for i in range(len(X)//batchsize):
        yield X[i*batchsize:(i+1)*batchsize],y[i*batchsize:(i+1)*batchsize]
    i+=1
    yield X[i*batchsize:],y[i*batchsize:]
```

In [15]:
```
def reconstruct(vae,data_gen):
    """given a VAE model, plot original data and reconstructed data from VAE
    inp = next(data_gen)[0]
    print('Original Data:')
    plot_digits(inp)
    with torch.no_grad():
        reconst,mu,log_var = vae(torch.tensor(inp,dtype=torch.float))

    print('Reconstructed Data:')
    plot_digits(reconst.detach().numpy())

def plot_digits(data):
    #plot 100 digit. data shape(100,32,32)
    fig, ax = plt.subplots(10, 10, figsize=(12, 12),
                           subplot_kw=dict(xticks=[], yticks=[]))
    fig.subplots_adjust(hspace=0.1, wspace=0.1)
    for i, axi in enumerate(ax.flat):
        im = axi.imshow(data[i].reshape(32, 32), cmap=plt.get_cmap('gray'))
        im.set_clim(0, 1)
```

In [16]:
```
from torch.optim import SGD, Adam
import torch.nn.functional as F
import random
from tqdm import tqdm
import math
from sklearn.model_selection import train_test_split
class Trainer():
    def __init__(self, model, optimizer_type, learning_rate, epoch, batch_si
        """ The class for training the model
        model: nn.Module
            A pytorch model
        optimizer_type: 'adam' or 'sgd'
        learning_rate: float
        epoch: int
        batch_size: int
        input_transform: func
            transforming input. Can do reshape here
        """
        self.model = model
        if optimizer_type == "sgd":
            self.optimizer = SGD(model.parameters(), learning_rate,momentum=
        elif optimizer_type == "adam":
            self.optimizer = Adam(model.parameters(), learning_rate)

        self.epoch = epoch
        self.batch_size = batch_size
        self.input_transform = input_transform


    def loss_fn(recon_x, x, mu, logvar):
        BCE = F.binary_cross_entropy(recon_x, x, size_average=False)
        KLD = -0.5 * torch.mean(1 + logvar - mu.pow(2) - logvar.exp())
        return BCE + KLD


    @timing
    def train(self, inputs, outputs, val_inputs, val_outputs,draw_curve=Fals
        """ train self.model with specified arguments
        inputs: np.array, The shape of input_transform(input) should be (nda
        outputs: np.array shape (ndata,)
        val_nputs: np.array, The shape of input_transform(val_input) should
        val_outputs: np.array shape (ndata,)
        early_stop: bool
        l2: bool
        silent: bool. Controls whether or not to print the train and val err
        """
        inputs = self.input_transform(torch.tensor(inputs, dtype=torch.float
        outputs = torch.tensor(outputs, dtype=torch.int64)
        val_inputs = self.input_transform(torch.tensor(val_inputs, dtype=tor
        val_outputs = torch.tensor(val_outputs, dtype=torch.int64)

        losses = []
        accuracies = []
        val_losses = []
        val_accuracies = []
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
```

```
        for n_epoch in tqdm(range(self.epoch), leave=False):
            self.model.train()
            #shuffle the data in each epoch
            idx =torch.randperm(inputs.size()[0])
            inputs=inputs[idx]
            outputs=outputs[idx]
            train_gen = data_gen(inputs,outputs,self.batch_size)

            epoch_loss = 0
            epoch_acc = 0
            for batch_input,batch_output in train_gen:
                batch_importance = len(batch_output) / len(outputs)
                batch_predictions = self.model(batch_input)


                gen=batch_predictions[0]
                mu=batch_predictions[1]
                sigma=batch_predictions[2]
                self.optimizer.zero_grad()


                loss = self.loss_function(batch_input, gen, mu, sigma)
                loss.backward()
                self.optimizer.step()

                epoch_loss += loss.item()* batch_importance
            val_loss = self.evaluate(val_inputs, val_outputs, print_acc=Fals
            if n_epoch % 10 ==0 and not silent:
                print("Epoch %d/%d - Loss: %.3f " % (n_epoch + 1, self.epoch
                print("              Val_loss: %.3f " % (val_loss))
            losses.append(epoch_loss)
            val_losses.append(val_loss)
            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    weights = self.model.state_dict()
        if draw_curve:
            plt.figure()
            plt.plot(np.arange(self.epoch) + 1,losses,label='Training loss')
            plt.plot(np.arange(self.epoch) + 1,val_losses,label='Validation
            plt.xlabel('Epochs')
            plt.ylabel('Loss')
            plt.legend()

        if early_stop:
            self.model.load_state_dict(weights)

        return {"losses": losses, 'val_losses': val_losses}

    def evaluate(self, inputs, outputs, print_acc=True):

        self.model.eval()
        gen = data_gen(inputs,outputs,self.batch_size)
        losses = 0

        for batch_input,batch_output in gen:
            batch_importance = len(batch_output) / len(outputs)
            with torch.no_grad():
                batch_predictions = self.model(batch_input)
                x=batch_input
```

```
                x=batch_input
                gen=batch_predictions[0]
                mu=batch_predictions[1]
                sigma=batch_predictions[2]
                self.optimizer.zero_grad()

                loss = self.loss_function(x, gen, mu, sigma)
                loss.backward()
                self.optimizer.step()


            losses += loss.item()


        return losses
```

In [17]:

```
from sklearn.model_selection import train_test_split,KFold
def train_model(model_func,Xs,ys,test_Xs,test_ys,epochs,draw_curve=True,earl

    train_Xs, val_Xs, train_ys, val_ys = train_test_split(Xs, ys, test_size=

    model=model_func(batchsize*32*32,32,100)
    print(f"{model_func.__name__} parameters:", sum([len(item.flatten()) for

    trainer = Trainer(model, optimizer, lr, epochs, batchsize, lambda x: x.r
    log=trainer.train(train_Xs, train_ys,val_Xs,val_ys,early_stop=early_stop
```

In [18]:

```
train_model(VAE_CNN,train_X_norm,train_y,test_X_norm,test_y,50)
```

```
VAE_CNN parameters: 13224592
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [18], in <cell line: 1>()
----> 1 train_model(VAE_CNN,train_X_norm,train_y,test_X_norm,test_y,50)

Input In [17], in train_model(model_func, Xs, ys, test_Xs, test_ys, epochs,
draw_curve, early_stop, batchsize, optimizer, lr, input_shape)
      7 print(f"{model_func.__name__} parameters:", sum([len(item.flatten
()) for item in model.parameters()]))
      9 trainer = Trainer(model, optimizer, lr, epochs, batchsize, lambda
x: x.reshape(input_shape))
---> 10 log=trainer.train(train_Xs, train_ys,val_Xs,val_ys,early_stop=early
_stop)

Input In [13], in timing.<locals>.wrap(*args, **kw)
      8 @wraps(f)
      9 def wrap(*args, **kw):
     10     ts = time()
---> 11     result = f(*args, **kw)
     12     te = time()
     13     print('func:%r  took: %2.4f sec' % (f.__name__,  te-ts))

Input In [16], in Trainer.train(self, inputs, outputs, val_inputs, val_outp
uts, draw_curve, early_stop, l2, silent)
     69 for batch_input,batch_output in train_gen:
     70     batch_importance = len(batch_output) / len(outputs)
---> 71     batch_predictions = self.model(batch_input)
     74     gen=batch_predictions[0]
     75     mu=batch_predictions[1]

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:11
10, in Module._call_impl(self, *input, **kwargs)
   1106 # If we don't have any hooks, we want to skip the rest of the logic
in
   1107 # this function, and just call forward.
   1108 if not (self._backward_hooks or self._forward_hooks or self._forwar
d_pre_hooks or _global_backward_hooks
   1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
   1111 # Do not call functions when jit is used
   1112 full_backward_hooks, non_full_backward_hooks = [], []

Input In [11], in VAE_CNN.forward(self, x)
     51 def forward(self, x):
---> 52     z, mu, logvar = self.encode(x)
     53     z = self.decode(z)
     54     return z, mu, logvar

Input In [11], in VAE_CNN.encode(self, x)
     40 def encode(self, x):
---> 41     h = self.encoder(x)
     42     z, mu, logvar = self.bottleneck(h)
     43     return z, mu, logvar

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:11
10, in Module._call_impl(self, *input, **kwargs)
   1106 # If we don't have any hooks, we want to skip the rest of the logic
in
```

```
   1107 # this function, and just call forward.
   1108 if not (self._backward_hooks or self._forward_hooks or self._forwar
d_pre_hooks or _global_backward_hooks
   1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
   1111 # Do not call functions when jit is used
   1112 full_backward_hooks, non_full_backward_hooks = [], []

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/container.p
y:141, in Sequential.forward(self, input)
    139 def forward(self, input):
    140     for module in self:
--> 141         input = module(input)
    142     return input

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:11
10, in Module._call_impl(self, *input, **kwargs)
   1106 # If we don't have any hooks, we want to skip the rest of the logic
in
   1107 # this function, and just call forward.
   1108 if not (self._backward_hooks or self._forward_hooks or self._forwar
d_pre_hooks or _global_backward_hooks
   1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
   1111 # Do not call functions when jit is used
   1112 full_backward_hooks, non_full_backward_hooks = [], []

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/conv.py:447,
in Conv2d.forward(self, input)
    446 def forward(self, input: Tensor) -> Tensor:
--> 447     return self._conv_forward(input, self.weight, self.bias)

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/conv.py:443,
in Conv2d._conv_forward(self, input, weight, bias)
    439 if self.padding_mode != 'zeros':
    440     return F.conv2d(F.pad(input, self._reversed_padding_repeated_tw
ice, mode=self.padding_mode),
    441                     weight, bias, self.stride,
    442                     _pair(0), self.dilation, self.groups)
--> 443 return F.conv2d(input, weight, bias, self.stride,
    444                 self.padding, self.dilation, self.groups)

RuntimeError: Given groups=1, weight of size [4, 102400, 4, 4], expected in
put[100, 1, 32, 32] to have 102400 channels, but got 1 channels instead
```

In [19]:
```
reconstruct(VAE_CNN(32,16*4*4,32),data_gen(train_X_norm,train_y,128))
```

```
Original Data:
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [19], in <cell line: 1>()
----> 1 reconstruct(VAE_CNN(32,16*4*4,32),data_gen(train_X_norm,train_y,12
8))

Input In [15], in reconstruct(vae, data_gen)
      5 plot_digits(inp)
      6 with torch.no_grad():
----> 7     reconst,mu,log_var = vae(torch.tensor(inp,dtype=torch.float))
      9 print('Reconstructed Data:')
     10 plot_digits(reconst.detach().numpy())

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:11
10, in Module._call_impl(self, *input, **kwargs)
   1106 # If we don't have any hooks, we want to skip the rest of the logic
in
   1107 # this function, and just call forward.
   1108 if not (self._backward_hooks or self._forward_hooks or self._forwar
d_pre_hooks or _global_backward_hooks
   1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
   1111 # Do not call functions when jit is used
   1112 full_backward_hooks, non_full_backward_hooks = [], []

Input In [11], in VAE_CNN.forward(self, x)
     51 def forward(self, x):
---> 52     z, mu, logvar = self.encode(x)
     53     z = self.decode(z)
     54     return z, mu, logvar

Input In [11], in VAE_CNN.encode(self, x)
     40 def encode(self, x):
---> 41     h = self.encoder(x)
     42     z, mu, logvar = self.bottleneck(h)
     43     return z, mu, logvar

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:11
10, in Module._call_impl(self, *input, **kwargs)
   1106 # If we don't have any hooks, we want to skip the rest of the logic
in
   1107 # this function, and just call forward.
   1108 if not (self._backward_hooks or self._forward_hooks or self._forwar
d_pre_hooks or _global_backward_hooks
   1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
   1111 # Do not call functions when jit is used
   1112 full_backward_hooks, non_full_backward_hooks = [], []

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/container.p
y:141, in Sequential.forward(self, input)
    139 def forward(self, input):
    140     for module in self:
--> 141         input = module(input)
    142     return input

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/module.py:11
10, in Module._call_impl(self, *input, **kwargs)
```

```
   1106 # If we don't have any hooks, we want to skip the rest of the logic
in
   1107 # this function, and just call forward.
   1108 if not (self._backward_hooks or self._forward_hooks or self._forwar
d_pre_hooks or _global_backward_hooks
   1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
   1111 # Do not call functions when jit is used
   1112 full_backward_hooks, non_full_backward_hooks = [], []

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/conv.py:447,
in Conv2d.forward(self, input)
    446 def forward(self, input: Tensor) -> Tensor:
--> 447     return self._conv_forward(input, self.weight, self.bias)

File ~/miniconda3/lib/python3.9/site-packages/torch/nn/modules/conv.py:443,
in Conv2d._conv_forward(self, input, weight, bias)
    439 if self.padding_mode != 'zeros':
    440     return F.conv2d(F.pad(input, self._reversed_padding_repeated_tw
ice, mode=self.padding_mode),
    441                     weight, bias, self.stride,
    442                      _pair(0), self.dilation, self.groups)
--> 443 return F.conv2d(input, weight, bias, self.stride,
    444                 self.padding, self.dilation, self.groups)

RuntimeError: Given groups=1, weight of size [4, 32, 4, 4], expected input
[1, 128, 32, 32] to have 32 channels, but got 128 channels instead
```