```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
torch.manual_seed(0)
```

```
<torch._C.Generator at 0x7f64512924b0>
```

```
print(torch.cuda.get_device_name(0))
```

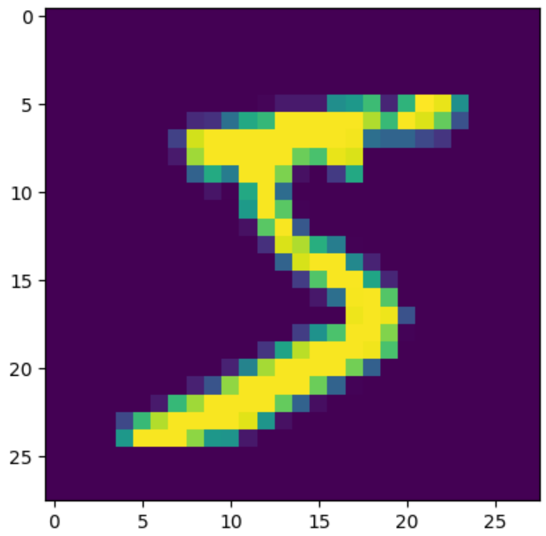
```
Tesla T4
cuda:0
```

```
from torchvision.datasets import MNIST
from torchvision import transforms as T
import os

cwd = os.getcwd()
trans1 = T.ToTensor()
```

```
# familiarise with mnist
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'True' # thank you stack overflow
print(mnist_train.__len__(), mnist_test.__len__())
an_image, label = mnist_train.__getitem__(0)
print(label, an_image.shape, an_image.min(), an_image.max())
plt.imshow(an_image.squeeze())
```

```
60000 10000
5 torch.Size([1, 28, 28]) tensor(0.) tensor(1.)
<matplotlib.image.AxesImage at 0x7efd0f20edd0>
```




```
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
DEBUG = False
print(DEVICE)
```

```
cuda:0
```

```
mnist_train = MNIST(root=cwd, train = True, transform = trans1, download=True)
mnist_test = MNIST(root=cwd, train = False, transform = trans1, download=True)
dataset_flatten = MNIST(root=cwd, download=True,
    transform=T.Compose([T.ToTensor(), T.Lambda(torch.flatten)]))
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to /content/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:00<00:00, 383922277.22it/s]
Extracting /content/MNIST/raw/train-images-idx3-ubyte.gz to /content/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to /content/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 20300937.46it/s]
Extracting /content/MNIST/raw/train-labels-idx1-ubyte.gz to /content/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to /content/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:00<00:00, 204485124.53it/s]
```

```
Extracting /content/MNIST/raw/t10k-images-idx3-ubyte.gz to /content/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to /content/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 22491769.50it/s]Extracting /content/MNIST/raw/t10k-labels-idx1-ubyte.gz to /content/MNIST/
```

```python
from tqdm import tqdm
from torch.utils.data import Dataset, DataLoader


class MNISTEncoder(nn.Module):

    def __init__(self, hidden_dim):

        # simple 1 hidden layer architecture as described in original VAE paper
        super(MNISTEncoder,self).__init__()
        self.hidden_dim = hidden_dim
        self.linear_stack = nn.Sequential(nn.Conv2d(1, 32, kernel_size=3, stride=2),
            nn.LeakyReLU(0.2),
            nn.Conv2d(32, 64, 3, stride=2),
            nn.LeakyReLU(0.2),
            nn.Flatten(),
            nn.Linear(2304, 1 + self.hidden_dim))

    def forward(self, img):
        # return mu, log_sigma
        res = self.linear_stack(img)
        mu, log_sigma = torch.split(res, self.hidden_dim, dim = -1)
        return mu, log_sigma

    def sample(self, mu, log_sigma):
        # Input: mu is a N by hidden_dim tensor, log_sigma is a N tensor
        # Output: N samples as a (N, hidden_dim) tensor, the ith sampled from N(mu_i, sigma_i)
        N = mu.shape[0]

        # broadcasting by column
        return torch.t(torch.exp(log_sigma)) @ torch.randn((N,self.hidden_dim), device=DEVICE)  + mu


class MNISTDecoder(nn.Module):

    def __init__(self, hidden_dim):
        super(MNISTDecoder, self).__init__()
        self.hidden_dim = hidden_dim
        self.linear_stack = nn.Sequential(nn.Linear(self.hidden_dim, 2048),
            nn.LeakyReLU(0.2),
            nn.Unflatten(1, (32, 8, 8)),
            nn.ConvTranspose2d(32, 64, kernel_size=3, stride=2, padding=(1, 1)),
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=(1, 1)),
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(32, 1, kernel_size=2, padding=(1, 1)),
            nn.Sigmoid()) # MNIST pixels in [0,1]

    def forward(self, z):
        return self.linear_stack(z)

class MNISTVAE(nn.Module):
    def __init__(self, hidden_dim, train_set):
        super(MNISTVAE,self).__init__()

        self.hidden_dim = hidden_dim
        self.encoder = MNISTEncoder(hidden_dim).to(device=DEVICE)
        self.decoder = MNISTDecoder(hidden_dim).to(device=DEVICE)
        self.train_set = train_set

        # nn hyper-parameters

        # coeff of the KL & E[log p(x|z)] parts, used to make sure the parts
        # have comparable magnitude in final loss
        self.beta = 1
        self.alpha = 0.01

        self.step_size = 1e-3
        self.batch_size = 100
        self.optimizer_fn = torch.optim.Adam

        # initialise optimizers & dataloader
        # currently use same lr for encoder & decoder
```

```
        self.optimizer = self.optimizer_fn(self.parameters(), lr = self.step_size, eps=1e-3)

        self.dataloader = DataLoader(self.train_set, batch_size = self.batch_size, shuffle=True)

    def estimate_elbo(self, batch):
        """
        Input: batch, a B by 784 vector (B is size of minibatch)
        Output: tensor with grad, the elbo estimate from batch

        In Gaussian case, if J is dimensionality of z then
        L(params, x) = 1/2 * sum_{j=1}^J [1 - 2log(sigma_j) + mu_j ^ 2 + sigma_j ^2]
                    + E_{q(z|x)}[log p(x|z)] is to be minimized

        The constant '+1' can be ignored.

        We estimate E_{q(z|x)}[log p(x|z)] by taking a single sample z
        (corresponding to L = 1 in section 2.3) in the paper.

        Take p(x|z) ~ N(mu(z), I), so the likelihood is proportional to ||mu(z)-x||^2
        """
        mus, log_sigmas = self.encoder(batch)

        # compute KL part
        KLD = 1/2 * torch.sum (torch.square(mus) - 2 * log_sigmas + torch.exp(2*log_sigmas))

        # samples Zs
        Zs = self.encoder.sample(mus, log_sigmas)

        if DEBUG: # compare scale of loss components
            print(KLD, torch.sum (torch.square(batch - self.decoder(Zs))).item())

        Loss = self.alpha * KLD +  self.beta * torch.sum (torch.square(batch - self.decoder(Zs)))

        return Loss/self.batch_size # normalise loss as is best practice (nn.MSELoss() by default normalises too)

    # TODO: training procedure (basically just backprop on L)
    def train_loop(self, epochs):
        for i in range(epochs):
            print(f"Epoch #{i}:")
            total_loss = 0

            # not quite random sampling but dataloader is re-randomised every
            # time enumerate is called so close enough
            for batch_num, (X,_) in enumerate(tqdm(self.dataloader)):
                self.optimizer.zero_grad()

                X = X.to(device=DEVICE)
                elbo_estimate = self.estimate_elbo(X)
                elbo_estimate.backward()

                self.optimizer.step()
                total_loss += elbo_estimate.item()

                if DEBUG and batch_num == 1:
                    break
            print(f"Loss on epoch = {total_loss}")

    def sample(self):

        latent_code = torch.randn(self.hidden_dim, device=DEVICE)
        decode = self.decoder(latent_code.unsqueeze(0))
        gen = torch.reshape(decode, (28,28))
        return gen
```

```
VAE = MNISTVAE(2, mnist_train)
DEBUG=False
VAE.train_loop(5)
```

```
Epoch #0:
100%|██████████| 600/600 [00:12<00:00, 47.22it/s]
Loss on epoch = 31431.6510887146
Epoch #1:
100%|██████████| 600/600 [00:12<00:00, 47.72it/s]
Loss on epoch = 24539.44556427002
Epoch #2:
100%|██████████| 600/600 [00:12<00:00, 48.00it/s]
Loss on epoch = 23155.565521240234
Epoch #3:
100%|██████████| 600/600 [00:12<00:00, 48.59it/s]
Loss on epoch = 22462.387329101562
Epoch #4:
```

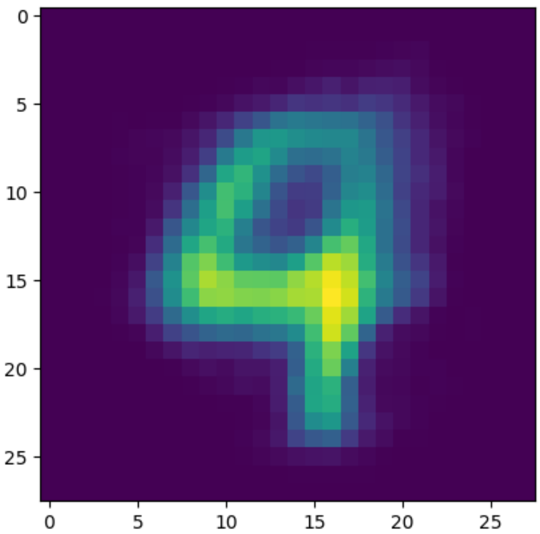
```
100%|██████████| 600/600 [00:12<00:00, 48.80it/s]Loss on epoch = 21975.97852706909
```

```
VAE.train_loop(10) # train some more
```

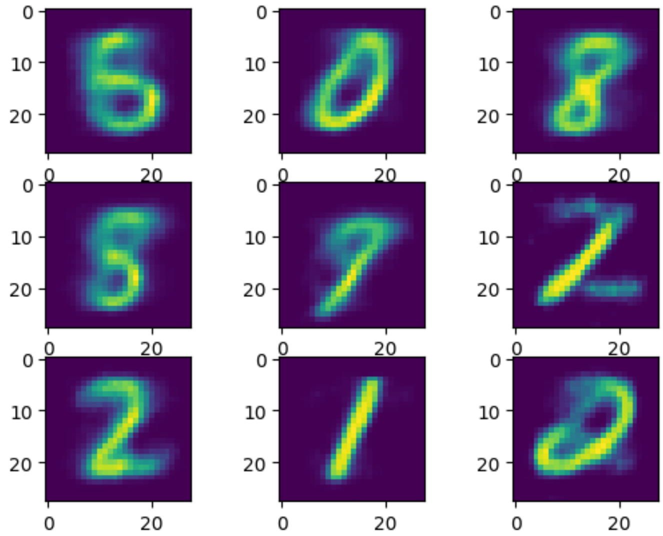
```
Epoch #0:
100%|██████████| 600/600 [00:12<00:00, 48.16it/s]
Loss on epoch = 21580.826454162598
Epoch #1:
100%|██████████| 600/600 [00:12<00:00, 48.00it/s]
Loss on epoch = 21305.884408950806
Epoch #2:
100%|██████████| 600/600 [00:12<00:00, 46.63it/s]
Loss on epoch = 21076.60096359253
Epoch #3:
100%|██████████| 600/600 [00:12<00:00, 49.38it/s]
Loss on epoch = 20896.202068328857
Epoch #4:
100%|██████████| 600/600 [00:12<00:00, 49.58it/s]
Loss on epoch = 20756.550184249878
Epoch #5:
100%|██████████| 600/600 [00:12<00:00, 48.55it/s]
Loss on epoch = 20657.363130569458
Epoch #6:
100%|██████████| 600/600 [00:13<00:00, 44.84it/s]
Loss on epoch = 20536.32221031189
Epoch #7:
100%|██████████| 600/600 [00:12<00:00, 49.38it/s]
Loss on epoch = 20454.89065170288
Epoch #8:
100%|██████████| 600/600 [00:12<00:00, 48.98it/s]
Loss on epoch = 20378.638193130493
Epoch #9:
100%|██████████| 600/600 [00:12<00:00, 49.35it/s]Loss on epoch = 20303.969120025635
```

```
sample = VAE.sample().cpu().detach().numpy()
plt.imshow(sample)
```

```
<matplotlib.image.AxesImage at 0x7f6440035810>
```




```
fig, ax = plt.subplots(3,3)
for i in range(3):
  for j in range(3):
    ax[i][j].imshow(VAE.sample().cpu().detach().numpy())
```



✓ 0s completed at 22:28