```
>>> presentation.info()
```

Date: January 19, 2022
Title: Variational Autoencoder

Author: Annada Behera
Organization[0]: National Institute of Science Education and Research, Bhubaneswar
Organization[1]: Homi Bhabha National Institute, Mumbai

>>> References

* Kingma, Diederik P., and Max Welling. *Auto-encoding variational bayes.* arXiv preprint arXiv:1312.6114, 2013.
* Foster, David. *Generative Deep Learning: Teaching Machines to Paint, Write, Compose and Play.* O'Reilly Media Inc, 2019

>>> The Encoder

The encoder, $\mathsf{Enc} : \mathbb{R}^{1\times 28\times 28} \to \mathbb{R}^2$.

Layer	Output size
Input	[1, 28, 28]
Conv2d(in:01, out:32, stride:1)	[32, 28, 28]
BAD()	[32, 28, 28]
Conv2d(in:32, out:64, stride:2)	[64, 14, 14]
BAD()	[64, 14, 14]
Conv2d(in:64, out:64, stride:2)	[64, 7, 7]
BAD()	[64, 7, 7]
Conv2d(in:64, out:64, stride:1)	[64, 7, 7]
BAD()	[64, 7, 7]
Flatten()	[3136]
Linear(in:3136, out:2)	[2]
Output	[2]

The kernel size is 3×3.

>>> BAD Layer

```python
from torch import nn;
class BAD(nn.Module):
    def __init__(_, in_channels):
        super().__init__();
        _.B = nn.BatchNorm2d(in_channels);
        _.A = nn.LeakyReLU();
        _.D = nn.Dropout();

    def forward(_, x):
        x = _.B(x);
        x = _.A(x);
        x = _.D(x);
        return x;
```

Table: BAD Layer

Layer	Output size
BatchNorm2d	[same]
LeakyReLU	[same]
Dropout	[same]

Obviously, Torch doesn't have a BAD layer. So, I make my own.

>>> Building the encoder

```
class Encoder(nn.Module):
    def __init__(_, ls_dim):
        super().__init__();
        _.ls_dim = ls_dim;
        _.lyr = nn.ModuleList();
        chan = [1, 32, 64, 64, 64];

        for i in range(4):
            _.lyr.append(nn.Conv2d( chan[i], chan[1+i],
                stride = 1 if i in [0,3] else 2,
                kernel_size = 3, padding = 1 ));
            _.lyr.append(BAD( in_channels = chan[1+i] ));
        _.lyr.append(nn.Flatten());
        _.lyr.append(nn.Linear(3136, _.ls_dim));

    def forward(_, x):
        for L in _.lyr:
            x = L(x);
        return x;
```

>>> Debugging the encoder

```python
class Encoder(nn.Module):
    # continued from the last slide
    def debug(_):
        fmt = "%-15s%s";
        x = torch.randn(32, 1, 28, 28);
        print(fmt%(L.__class__.__name__, x.size()));
        for L in _.lyr:
            x = L(x);
            print(fmt%(L.__class__.__name__, x.size()));
        return x;
```

Output:

```
Input          torch.Size([32, 1, 28, 28])
Conv2d         torch.Size([32, 32, 28, 28])
BAD            torch.Size([32, 32, 28, 28])
Conv2d         torch.Size([32, 64, 14, 14])
BAD            torch.Size([32, 64, 14, 14])
Conv2d         torch.Size([32, 64, 7, 7])
BAD            torch.Size([32, 64, 7, 7])
Conv2d         torch.Size([32, 64, 7, 7])
BAD            torch.Size([32, 64, 7, 7])
Flatten        torch.Size([32, 3136])
Linear         torch.Size([32, 2])
```

>>> Decoder layer

The decoder is a function, $\mathrm{Dec}: \mathbb{R}^2 \rightarrow \mathbb{R}^{1\times 28\times 28}$. The opposite of what $\mathrm{Enc}(x)$ does. However, the architecture may not be exactly inverse, though.

Layer	Output size
Input	[2]
Linear(in:2, out:3136)	[3136]
Reshape()	[64, 7, 7]
ConvTranspose2d(in:64, out:64, stride:1)	[64, 7, 7]
BAD()	[64, 7, 7]
ConvTranspose2d(in:64, out:64, stride:2)	[64, 14, 14]
BAD()	[64, 14, 14]
ConvTranspose2d(in:64, out:32, stride:2)	[32, 28, 28]
BAD()	[32, 28, 28]
ConvTranspose2d(in:32, out:01, stride:1)	[1, 28, 28]
BAD()	[1, 28, 28]
Output	[1, 28, 28]

Torch, unlike Keras/Tensorflow, doesn't have Reshape. So, I make new one.

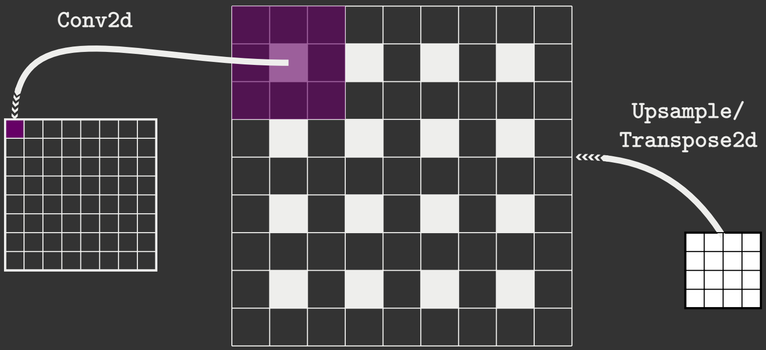
```
class Reshape(nn.Module):
    def __init__(_, size):
        super().__init__();
        _.size = size;

    def forward(_, x):
        x = torch.reshape(x, (-1, *_.size));
        return x;
```

* This is just a wrapper around torch.reshape().
* The parameters (-1, *_.size) tells Torch to use the first dimension as is, i.e, for the batch, N and the construct, * unpacks the arguments of size and passes it to the function.

```
[*(1, 2, 3)] == [1, 2, 3]     # True
```

>>> Upsampling v. ConvTranspose2d

For increasing the size of the image, there are two techniques that are popular in literature: Upsampling and ConvTranspose2d both of which are present in Torch.



with kernel size 3×3 and stride $(1, 1)$. You can use:

* Copy values: Upsampling()+Conv2d()
* Fill with zeros: ConvTranspose2d()

>>> The Decoder

```
class Decoder(nn.Module):
    def __init__(_, ls_dim):
        super().__init__();
        _.ls_dim = ls_dim;
        _.lyr = nn.ModuleList();
        chan = [64, 64, 64, 32, 1]

        _.lyr.append(nn.Linear(ls_dim, 3136));
        _.lyr.append(Reshape((64, 7, 7)));
        for i in range(4):
            _.lyr.append(nn.ConvTranspose2d(
                chan[i], chan[1+i], padding = 1, kernel_size = 3,
                stride = 1 if i in [0, 3] else 2,
                output_padding=0 if i in [0,3] else 1
            ));
            _.lyr.append(BAD(in_channels = chan[1+i]));

   def forward(_, x):
       for L in _.lyr:
           x = L(x);
       return x;
```

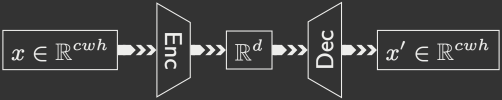
>>> Debugging the encoder

```python
def debug(_):
    fmt = "%-20s%s";
    x = torch.randn(32, _.ls_dim);
    print(fmt%("Input", x.size()));
    for L in _.lyr:
        x = L(x);
        print(fmt%(L.__class__.__name__, x.size()));
    return x;
```

Output:

```
Input               torch.Size([32, 2])
Linear              torch.Size([32, 3136])
Reshape             torch.Size([32, 64, 7, 7])
ConvTranspose2d     torch.Size([32, 64, 7, 7])
BAD                 torch.Size([32, 64, 7, 7])
ConvTranspose2d     torch.Size([32, 64, 14, 14])
BAD                 torch.Size([32, 64, 14, 14])
ConvTranspose2d     torch.Size([32, 32, 28, 28])
BAD                 torch.Size([32, 32, 28, 28])
ConvTranspose2d     torch.Size([32, 1, 28, 28])
BAD                 torch.Size([32, 1, 28, 28])
```

>>> The Autoencoder



We can now join the Encoder and the Decoder end to end to train the network and compare the input to the output. The Autoencoder takes a imaage and reconstructs the image back, $\mathsf{AE} : \mathbb{R}^{cwh} \to \mathbb{R}^{cwh}$

$$x' = \mathsf{AE}(x) = \mathsf{Dec}(\mathsf{Enc}(x)) \tag{1}$$

The Autoencoder network is now easy that we have already done all the heavy lifting.

```
class VAE(nn.Module):
    def __init__(_, ls_dim):
        super().__init__();
        _.enc = Encoder(ls_dim);
        _.dec = Decoder(ls_dim);

    def forward(_, x):
        x = _.enc(x);
        x = _.dec(x);
        return x;
```

```
def encode(_, x):
    with torch.no_grad():
        ls = _.enc(x.unsqueeze(0));
    return ls;

def decode(_, x):
    with torch.no_grad():
        img = _.dec(x);
    return img;
```

>>> Before we train: loss function and optimizer

Torch doesn't have the loss that I want, but it has one that is close enough, nn.MSELoss. I use that to make new one!

$$\text{loss\_fn}(x', x) = \sqrt{\frac{1}{n}\sum_i (x'_i - x_i)^2} \tag{2}$$

```
class RMSELoss(nn.Module):
        def __init__(_, eps=1e-8):
                super().__init__();
                _.eps = eps;

        def forward(_, y_, y):
                loss = nn.MSELoss();
                return torch.sqrt(loss(y_, y)+_.eps);
```

With that I can define my loss function and the optimizer, Adam.

```
    from torch.optim import Adam;
    optimizer = torch.optim.Adam(params=vae.parameters(), lr=5e-4);
    loss_fn = RMSELoss();
```

>>> Loading the data

Before we can train, we need the data. The MNIST data is already available in the torchvision library. So, let's import it.

```
import torchvision as tv;
data_mn = tv.datasets.MNIST(
    root = '.', train = True, download = True,
    transform = tv.transforms.Compose([
        tv.transforms.PILToTensor(),
        tv.transforms.Lambda(lambda x:x/255.)
    ])
```

Making own dataset.

```
import torch.utils.data as U;
class my_own_dataset(U.Dataset):
    def __init__(_):
        # initialize your data here using PIL, Pandas or whatever.
    def __len__(_):
        # return the length of your dataset.
        return length;
    def __getitem__(_, i):
        # return the i-th data point.
        return _.data[i];
```

>>> The Dataloader

```
class my_data(U.Dataset):
    def __init__(_):
        super().__init__();
    def __len__():
        return 64000;
    def __getitem__(_, i):
        return torch.sin(i*.0001*2);
```

All the children of U.Dataset can be passed to U.DataLoader like so,

```
mnist = U.DataLoader(
    dataset = data_mn,
    batch_size = 32,
    shuffle = True,
    num_workers = 8,
    pin_memory = False );
```

* Set pin_memory to True, if you want GPU.
* Let Torch worry about shuffling, batching and multithreading.
* U.DataLoader is an iterator that will give your data points[1].

---

[1] You will see what I mean when I train the network.

>>> The training loop

Train the model and save it,

```
device = 'cpu';     # or 'cuda' if you want GPU.
ls_dim = 2;
vae = VarAE(ls_dim=ls_dim).to(device);
DEBUG = False;
for epoch in range(epochs):
    if DEBUG:
        break;
    for imgc,_ in mnist:
        img = imgc.to(device);
        restore = vae(img);
        loss = loss_fn(restore, img);

        vae.zero_grad();
        loss.backward();
        optimizer.step();
torch.save(vae.state_dict(), "savemodel.torch");
```

And if you have already saved the model, then load it,

```
vae.load_state_dict(torch.load("savemodel.torch"));
```

Question: What makes a autoencoder ‘‘variational’’?

>>> The gaussian

The gaussian (or normal) distribution function takes a value from sample space and maps it onto a probability distribution with a distinct expectation μ and standard deviation, σ. It is characterized by the *bell curve* shape.

$$\mathcal{N}(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{3}$$

in one-dimension. For an arbitray dimension, k, the curve is given as,

$$\mathcal{N}(x;M,\sigma) = \frac{\exp(-\frac{1}{2}(x-M)^T\Sigma^{-1}(x-M))}{\sqrt{(2\pi)^k|\Sigma|}} \tag{4}$$

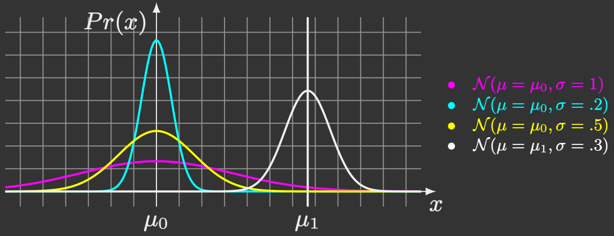
where, M[2] is the mean vector and Σ is the symmetric co-variance matrix. In two-dimension, it is defined as,

$$M = \begin{bmatrix} \mu_0 \\ \mu_1 \end{bmatrix} \quad \text{and} \quad \Sigma = \begin{bmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{bmatrix} \tag{5}$$

where ρ is the covariance coefficient.

[2] this is Greek uppercase μ, not Roman alphabet M.

>>> The gaussian (continued)



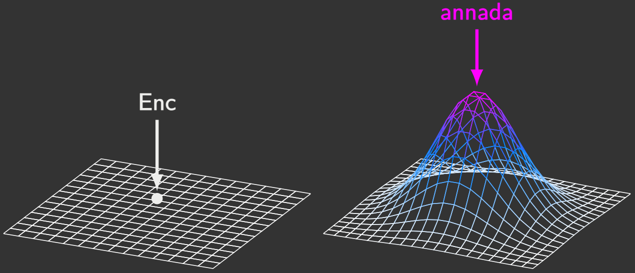
The gaussian with $\mathcal{N}(\mu = 0, \sigma = 1)$ is called the standard gaussian and to sample from any other gaussian, we can,

$$z = \mu + \sigma\epsilon, \qquad \epsilon \sim N(\mu = 0, \sigma = \mathbf{I}_z) \tag{6}$$

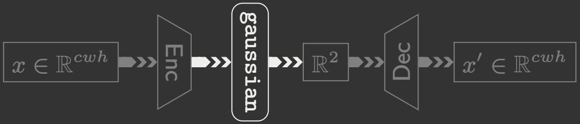
Torch only provides function to sample from $\mathcal{N}(x; 0, 1)$ called `torch.randn()`. This expression helps us sample from an arbitrary gaussian. For stability reasons in our autoencoder, we take the logarithm of the variance instead of just σ^2.

$$\sigma = \exp(\log(\sigma)) = \exp(2\log(\sigma)/2) = \exp(\log(\sigma^2)/2) \tag{7}$$

>>> Gaussian in encoder



Instead of mapping the points directly to the latent space, map the point to a multivariate gaussian, where the mean, μ and the standard deviation, σ is a learnable parameter.



Of course, Torch doesn't have a Gaussian layer. So, I make new one!

>>> The gaussian layer

I just overload my gaussian on two Torch's Linear layer, one for the mean and the other for the standard deviation. The learnable weights for this layer is now learning the mean and standard deviation, respectively.

```
class Gaussian(nn.Module):
    def __init__(_, inp, out):
        super().__init__();
        _.out = out;
        _.mean = nn.Linear(inp, out);
        _.stdv = nn.Linear(inp, out);

    def forward(_, x):
        eps = torch.randn(_.out, requires_grad=False);
        return _.mean(x)+eps*torch.exp(_.stdv(x)/2.);
```

So that I can finally replace the last nn.Linear with my custom built Gaussian in the Encoder.

```
        # _.lyr.append(nn.Linear(3136, _.ls_dim));
        _.lyr.append(Gaussian(3136, _.ls_dim));
```

>>> Relative entropy

Relative entropy is the asymmetric ``distance'' between two probability distribution function. For two given distribution function, $p(x)$ and $q(x)$. The relative entropy is given as,

$$\mathsf{KL}(p||q) = \int_{x\in\mathbb{R}} p(x) \log \frac{p(x)}{q(x)} \tag{8}$$

* Since we have gaussian layer, we need to track how far away is the learnable parameters away from standard gaussian and we penalize if it goes too ``far''.
* To penalize, we can use relative entropy as a loss function.
* Torch has a built-in layer, called nn.KLDivLoss. But, it doesn't work for me[3]. So, I make new one!

[3] if anybody knows why, please let me know! My KLDivLoss is going negative.

>>> Relative entropy between two univariate gaussians

$$\mathsf{KL}(p||q) = \int_{x\in\mathbb{R}} \mathcal{N}(x;\mu,\sigma)\log\frac{\mathcal{N}(x;\mu,\sigma)}{\mathcal{N}(x;0,\mathbf{I}_z)} \tag{9}$$

$$= \int_{x\in\mathbb{R}} \mathcal{N}(x;\mu,\sigma)\log\mathcal{N}(x;\mu,\sigma) - \int_{x\in\mathbb{R}} \mathcal{N}(x;\mu,\sigma)\log\mathcal{N}(x;0,\mathbf{I}_z) \tag{10}$$

The relative entropy in this form is not easy to code or calculate, so we can get a closed form by symbol juggling,

$$= \frac{1}{2}\left[\log\frac{\Sigma_1}{\Sigma_0} - n + Tr(\Sigma_1^{-1}\Sigma_0) + (\mu_1-\mu_0)\Sigma_1^{-1}(\mu_1-\mu_0)\right] \tag{11}$$

Some more symbol juggling,

$$= \frac{1}{2}\left[\sum_{i=0}^{z}\mu_i^2 + \sum_{i=0}^{z}\sigma_i^2 - \sum_{i=0}^{z}\left(\log(\sigma_i^2)+1\right)\right] \tag{12}$$

>>> Loss

For reference,

$$\mathsf{KL}(\mathcal{N}(M,\Sigma)||\mathcal{N}(0,\mathbf{I}_z)) = \frac{1}{2}\left[\sum_{i=0}^{z}\mu_i^2 + \sum_{i=0}^{z}\sigma_i^2 - \sum_{i=0}^{z}\left(\log(\sigma_i^2)+1\right)\right] \tag{13}$$

```
class KLLoss(nn.Module):
    def __init__(_, model, eps=1e-8):
        super().__init__();
        _.mul = model.enc.lyr[9].mean;
        _.sdl = model.enc.lyr[9].stdv;
    def forward(_, y_, y):
        mus = torch.sum(torch.pow(_.mul(y_), 2));
        sds = torch.sum(torch.pow(torch.exp(_.sdl(y_)), 2));
        lvs = torch.sum(_.sdl(y_)+1);
        return .5*(mus+sds+lvs);
class VAELoss
    def __init__(_):
        super().__init__();
        _.rl = RMSELoss();
        _.kl = KLLoss();
    def forward(_, y_, y):
        return _.rl(y_, y)+_.kl(y_, y);sds-lvs);
loss_fn = VAELoss();
```

>>> Latent space arithmatic: Object morphing

Given an image, I_u of an object, if we want to morph into say into the image I_v of another object, we use the following algorithm,

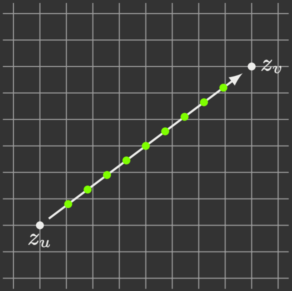
INPUT: Initial image, I_u and final image, I_v
BEGIN SUBROUTINE

$z_u = \mathsf{VAE}_{\mathsf{enc}}(I_u);$
$z_v = \mathsf{VAE}_{\mathsf{enc}}(I_v);$
$S \leftarrow ()$
for $i \leftarrow 0$ to n BEGIN LOOP

$S \leftarrow S \cup \mathsf{VAE}_{\mathsf{dec}}(z_u + \frac{(z_v - z_u)i}{n})$

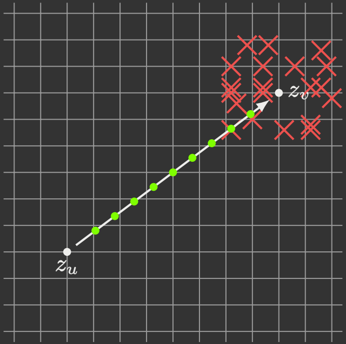
END LOOP
return S

END SUBROUTINE



S now contains a sequence of intermediate images of morphing between I_u and I_v.

>>> Latent space arithematic: style transfer



The algorithm is same as before, except that the final image, I_v is decoded as the average of the latent space vectors of the given label,

$$I_v = \mathsf{VAE}_{\mathsf{dec}}\left(\sum_{i \in S} \frac{i}{|S|}\right) \tag{14}$$

where $S = \{x : x \in \text{dataset and } \text{feat}(x) = \text{target}\}$

```
>>>  sudo init 0
System is going down for halt NOW!
```