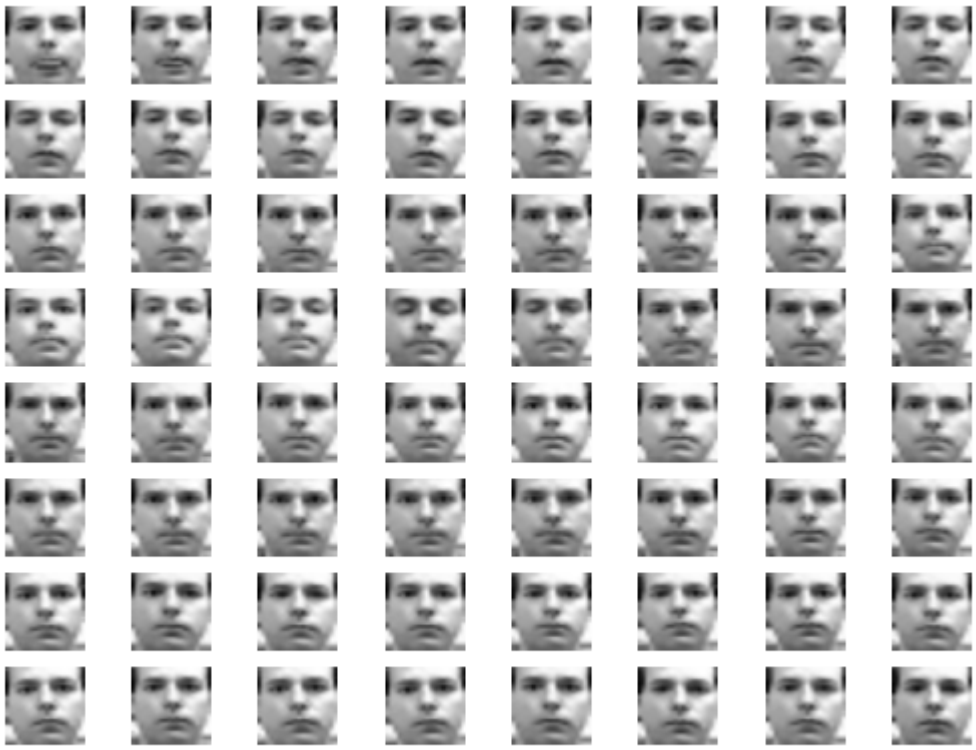
Fray dataset - Convolutional Auto Encoder 적용

데이터 구조:

- Brendan Frey라는 사람의 얼굴 사진 2000장 (20x28)
- 이미지량이 간단하고 적고 흑백 사진이라 컬러 채널도 하나이기 때문에 Convolutional Auto Encoder을 처음 공부할 때도움이 됐음



코드 구조:

- convolutional block에 필요한 모수 (Kernel size, stride, padding, initial number of filters) 정의
- Convolutional AE class는 다음과 같은 구조:
  - Encoder: 5개의 2D Convolution 층 사용
  - Decoder: 5개의 2D Transpose Convolution 층 사용
  - decoder의 input 채널의 수는 encoder의 input 채널의 수의 역순
  - Forward 단계는 input 사진을 encoding, encoding의 출력값을 reparameterize하여 latent vector 추출, latent vector을 input으로 decoding 진행
    - Optimizer은 Adam optimizer
    - loss function (Variational Lower Bound 또는 ELBO를 Maximize):
    - Reconstruction loss는 Binary Cross Entropy(BCE) loss 사용
      - input 이미지와 decoder로 재구성된 이미지 간 loss
    - KL-Divergence loss
      - ELBO를 maximize하기 위해서 KL-Divergence도 최소화 해야함
    - 따라서 Final loss function은 이 둘의 합!

복습!

여기서 하한 $\mathcal{L}(\theta, \phi; \mathbf{x}^{(i)})$ 을 $Variational\ Lower\ Bound$ 혹은 $ELBO$ (*evidence lower bound*) 라고 한다.

$$\theta^*, \phi^* = \arg\max_{\theta,\phi} \sum_{l=1}^{N} \mathcal{L}(\theta, \phi; \mathbf{x}^{(i)})$$

우리의 목적은 variational parameter ϕ 와 generative parameter θ 에 대해 이 하한을 미분하고 최적화하는 것이다.

$$\begin{aligned} ELBO = \mathcal{L}(\theta, \phi; \mathbf{x}^{(i)}) = -D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z})) \cdots (a) \\ + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z})\right] \cdots (b) \end{aligned} \tag{3}$$

Frey Dataset을 train과 validate!

사진의 첫 열은 original data이며, 두 번째 열은 Convolutional AE를 통해 새로 구현된 이미지이다.

Epoch1:

- 여기서 찾아볼 수 있는 feature는 눈과 입의 형태 정도이다.

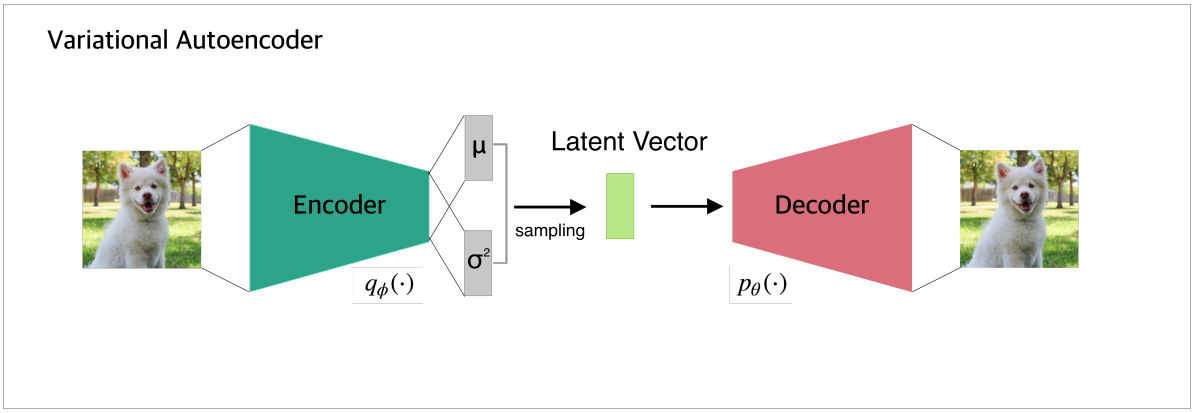
Epoch 20:

- 20번째 epoch에서는 얼굴의 feature를 잘 살려 눈, 코, 입, 그리고 T-자 라인이 뚜렷한 것을 볼 수 있다

2. 시행착오 과정

Conv. vae / VGG vae / ResNet vae 등 다양하게 시도해보았음.

그 중 조원 모두가 시도해보았던 것이 ResNet VAE



- Latent Variable 기반으로 원래 이미지 복원 수행

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

- 방학에 배웠던 Resnet
  - ResNet은 VGG를 뼈대로 convolution층을 추가하여 깊게 만든 후 shortcut을 추가함

```
class Resnet VAE(Resnet):
    def __init__:
        # CNN architechtures Setting #

        # encoding components #
        using resnet model
        Latent vectors mu and sigma: output = CNN embedding latent variables
        Sampling vector with latent variables

        # Decoder:
        upsampling (transpose convolution)

    def encode(self, x):
        resnet(X) & Fclayers(x)
        return mu, logvar

    def reparameterize(self, mu, logvar):
```

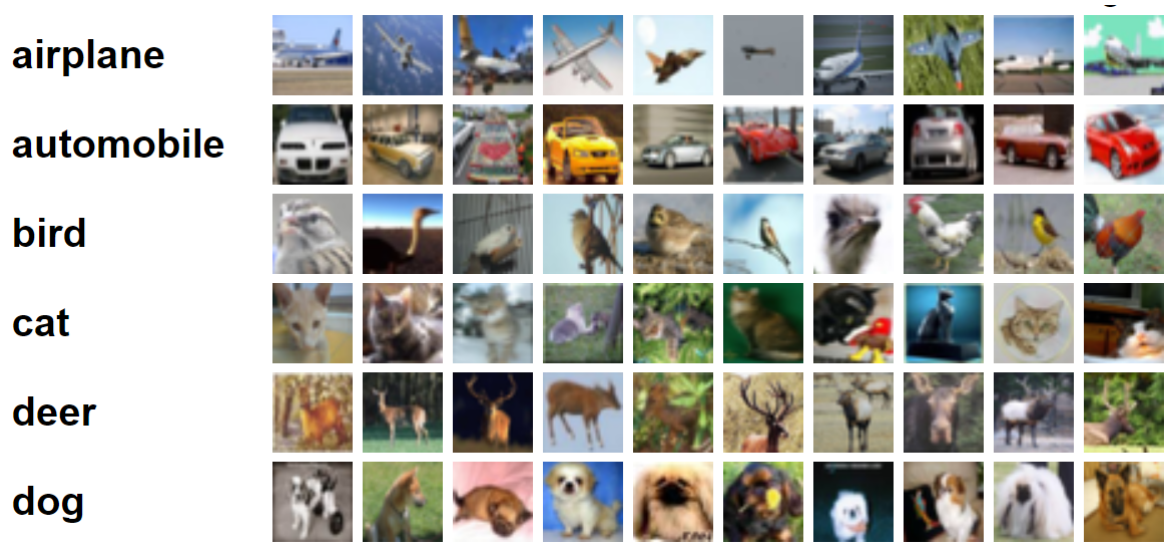
```
        return epsilon(std)+mu

    def decode(self, z):
        transpose conv(z)
        return result

    def forward(self, x):
        mu, logvar = encode(x)
        z = reparameterize(mu, logvar)
        x_reconstrunction = decode(z)

        return x_reconstrunction, z, mu, logvar
```

hsinyilin19 (HYLin) · GitHub (코드 참고)



- cifar 10 data로 Resnet VAE 결과를 구현해보려 하였음. 그러나 decoding 과정에서 에러가 뜨고, epoch 한 번이 20시간을 돌려도 끝나지 않을정도로 느려서 다른 방안을 모색하게 됨.

3. VAE - Linear

- Encoder와 Decoder의 layer를 Convolutional layer가 아닌 **Linear layer**로 시도
  - 이미지를 벡터로 flatten하여 encoding / decoding한 후 reconstruction 과정에서 다시 2차원 이미지로

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
from torchvision.utils import save_image
```

In [2]:
```
batch_size = 100

# MNIST Dataset
train_dataset = datasets.MNIST(root='./mnist_data/', train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./mnist_data/', train=False, transform=transforms.ToTensor(), download=False)

# Data Loader (Input Pipeline)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```

- 먼저 필요한 모듈과 MNIST 데이터셋을 불러온다 (batch size = 100)

In [3]:
```
class VAE(nn.Module):
    def __init__(self, x_dim, h_dim1, h_dim2, z_dim):
        super(VAE, self).__init__()

        # encoder part
        self.fc1 = nn.Linear(x_dim, h_dim1)
        self.fc2 = nn.Linear(h_dim1, h_dim2)
        self.fc31 = nn.Linear(h_dim2, z_dim)
        self.fc32 = nn.Linear(h_dim2, z_dim)
        # decoder part
        self.fc4 = nn.Linear(z_dim, h_dim2)
        self.fc5 = nn.Linear(h_dim2, h_dim1)
        self.fc6 = nn.Linear(h_dim1, x_dim)

    def encoder(self, x):
        h = F.relu(self.fc1(x))
        h = F.relu(self.fc2(h))
        return self.fc31(h), self.fc32(h) # mu, log_var

    def sampling(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return eps.mul(std).add_(mu) # return z sample

    def decoder(self, z):
        h = F.relu(self.fc4(z))
        h = F.relu(self.fc5(h))
        return F.sigmoid(self.fc6(h))

    def forward(self, x):
        mu, log_var = self.encoder(x.view(-1, 784))
        z = self.sampling(mu, log_var)
        return self.decoder(z), mu, log_var

# build model
vae = VAE(x_dim=784, h_dim1= 512, h_dim2=256, z_dim=2)
```

1) input dimension = 28 x 28 = 784

2) Encoder : Fully Connected layer ($approximate\ posterior\ q_\phi(\mathbf{z}|\mathbf{x}^{(i)})$)를 통과시켜 얻은 잠재변수 $\mathbf{z}$ (2차원으로 설정)

3) Sampling (=reparameterization) : $\mathbf{z}$ 를 input $\mathbf{x}$와 random noise ϵ 에 의해 결정되는 함수의 결과값처럼 변경

4) Decoder : input dimension에 맞게 다시 decoding $\Rightarrow p_\theta(\mathbf{x}^{(i)}|\mathbf{z})$ (reconstruction)

```
In [20]: for epoch in range(1, 101):
             train(epoch)
             test()
```

```
Train Epoch: 98 [50000/60000 (83%)]     Loss: 129.440361
====> Epoch: 98 Average loss: 131.9866
====> Test set loss: 136.4088
Train Epoch: 99 [0/60000 (0%)]  Loss: 130.954785
Train Epoch: 99 [10000/60000 (17%)]     Loss: 128.791475
Train Epoch: 99 [20000/60000 (33%)]     Loss: 128.797461
Train Epoch: 99 [30000/60000 (50%)]     Loss: 129.802441
Train Epoch: 99 [40000/60000 (67%)]     Loss: 141.090986
Train Epoch: 99 [50000/60000 (83%)]     Loss: 134.013076
====> Epoch: 99 Average loss: 131.9408
====> Test set loss: 136.3428
Train Epoch: 100 [0/60000 (0%)] Loss: 129.631719
Train Epoch: 100 [10000/60000 (17%)]    Loss: 131.028369
Train Epoch: 100 [20000/60000 (33%)]    Loss: 128.021318
Train Epoch: 100 [30000/60000 (50%)]    Loss: 135.539307
Train Epoch: 100 [40000/60000 (67%)]    Loss: 127.112393
Train Epoch: 100 [50000/60000 (83%)]    Loss: 137.547197
====> Epoch: 100 Average loss: 132.2733
====> Test set loss: 137.0579
```

- 총 100번의 epoch $\Rightarrow$ Loss 136 정도로 수렴

```
In [21]: with torch.no_grad():
             z = torch.randn(64, 2)
             sample = vae.decoder(z)

             save_image(sample.view(64, 1, 28, 28), './samples/sample_100' + '.png')
```

- 2차원의 표준 정규 난수 ($\mathbf{z} \sim N(0, 1)$) 로부터 새로운 MNIST 사진 생성

- 50번의 epoch 결과

- 100번의 epoch 결과

4. CelebFaces Attributes Dataset (CelebA)

- ✓ CelebA dataset은 Facial attribute와 관련된 40개의 label이 포함된 유명인의 image dataset

- ✓ label에는 hair color, gender, age 등을 포함.

- ✓ MNIST에는 train data 55000개, test data 10000개, validation data 5000개, 총 70000장의 data로 구성되는데 비해, CelebA data는 전체 이미지가 23만장에 달할 정도로 large scale dataset.

Celeba dataset에 VAE code를 적용해보려 하였으나 data size가 방대해서 시간 내 train하기에 부적합하다고 판단하여 결과만 제시. 아래는 VAE input으로 들어가는 plain image.

Source: https://github.com/yzwxx/vae-celebA

다음은 VAE를 통해 random하게 generate된 image 예시

https://github.com/yzwxx/vae-celebA

MNIST dataset에 VAE를 적용한 결과 이미지와 유사하게, Sample로 얻은 image들이 blurry한 형태로 도출된다는 단점을 확인할 수 있다.