

Generative AI for Images

COMP 646: Deep Learning for Vision and Language

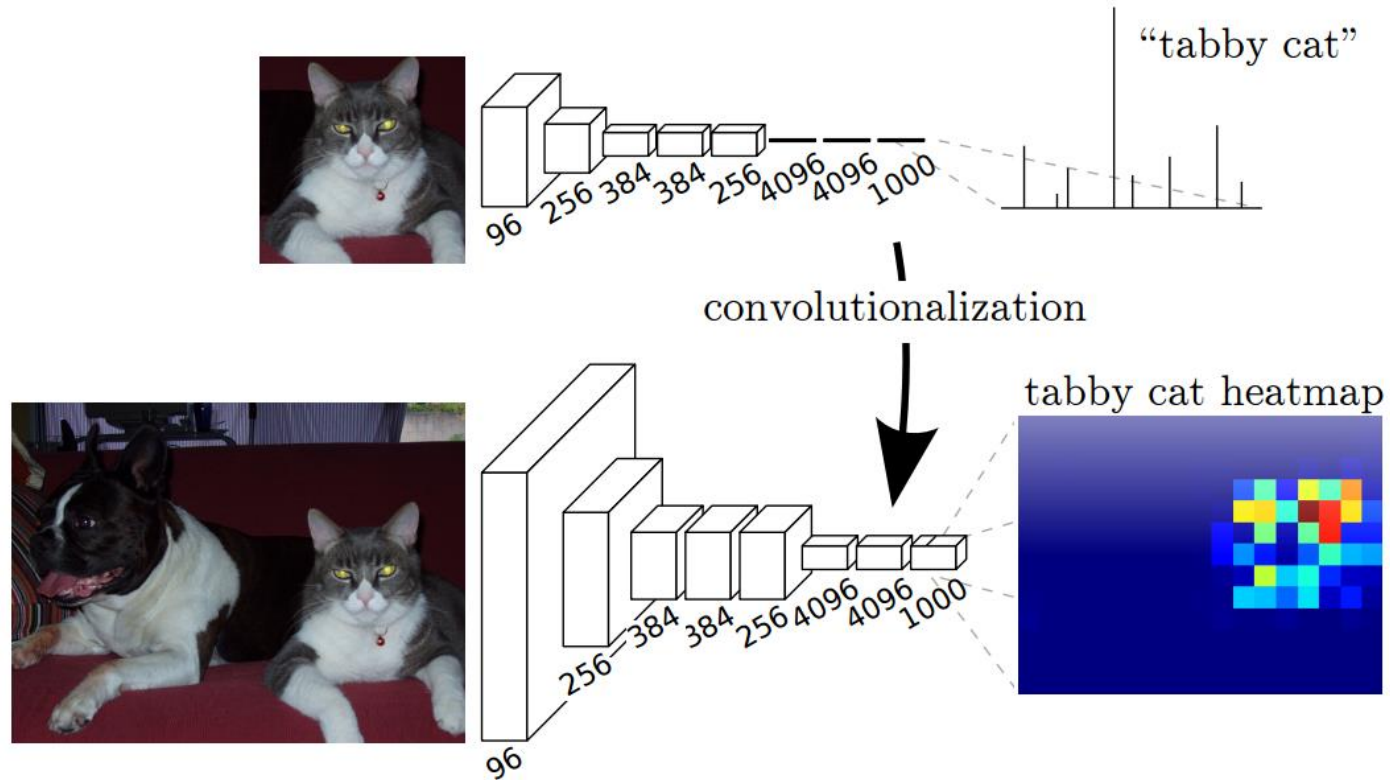


RICE UNIVERSITY

Last Class

- Image Segmentation
 - Idea 1: Converting Linear layers to Conv layers
 - Idea 2: ConvTranspose2d
 - Idea 3: Dilated Convolutions

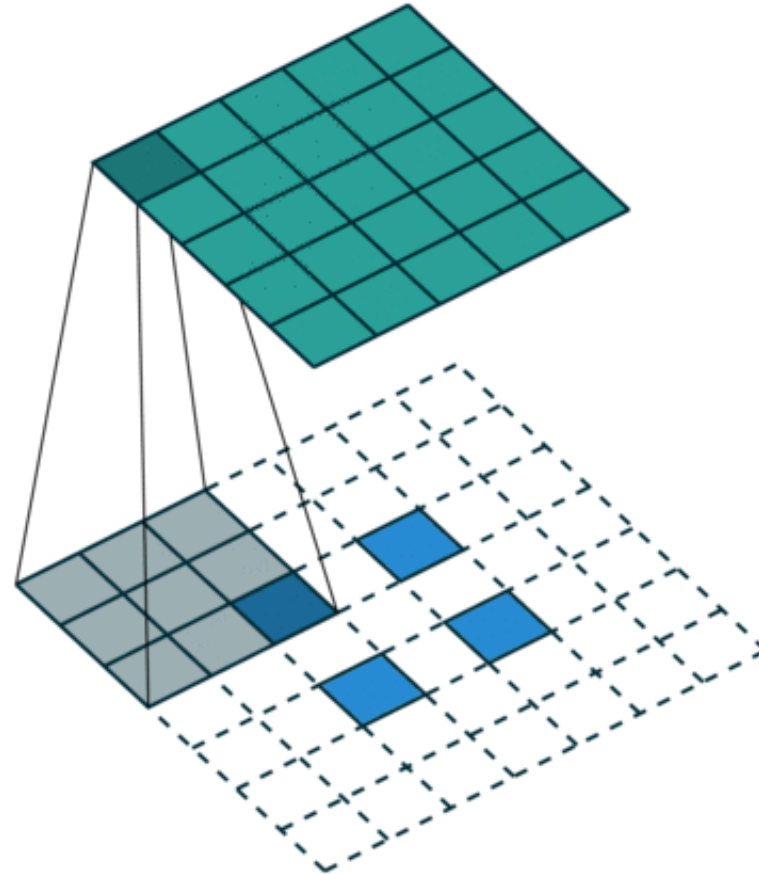
Idea 1: Convolutionalization



However resolution of the segmentation map is low.

https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf

Idea 2: Up-sampling Convolutions or "Deconvolutions" or Transposed Convolutions



https://github.com/vdumoulin/conv_arithmetic

Pytorch

Docs > [torch.nn](#) > ConvTranspose2d



CONVTRANSPOSE2D

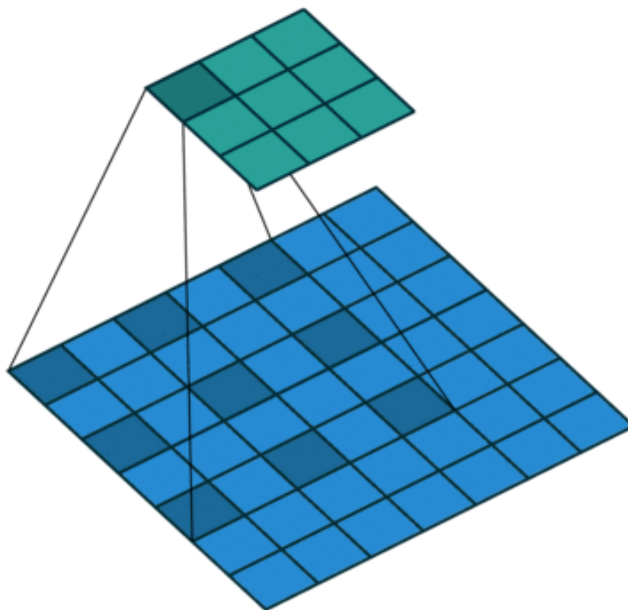
```
CLASS torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0,  
    output_padding=0, groups=1, bias=True, dilation=1, padding_mode='zeros', device=None,  
    dtype=None) \[SOURCE\]
```

Applies a 2D transposed convolution operator over an input image composed of several input planes.

This module can be seen as the gradient of Conv2d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation as it does not compute a true inverse of convolution). For more information, see the visualizations [here](#) and the [Deconvolutional Networks](#) paper.

This module supports [TensorFloat32](#).

Idea 3: Dilated Convolutions



MULTI-SCALE CONTEXT AGGREGATION BY
DILATED CONVOLUTIONS

Fisher Yu
Princeton University

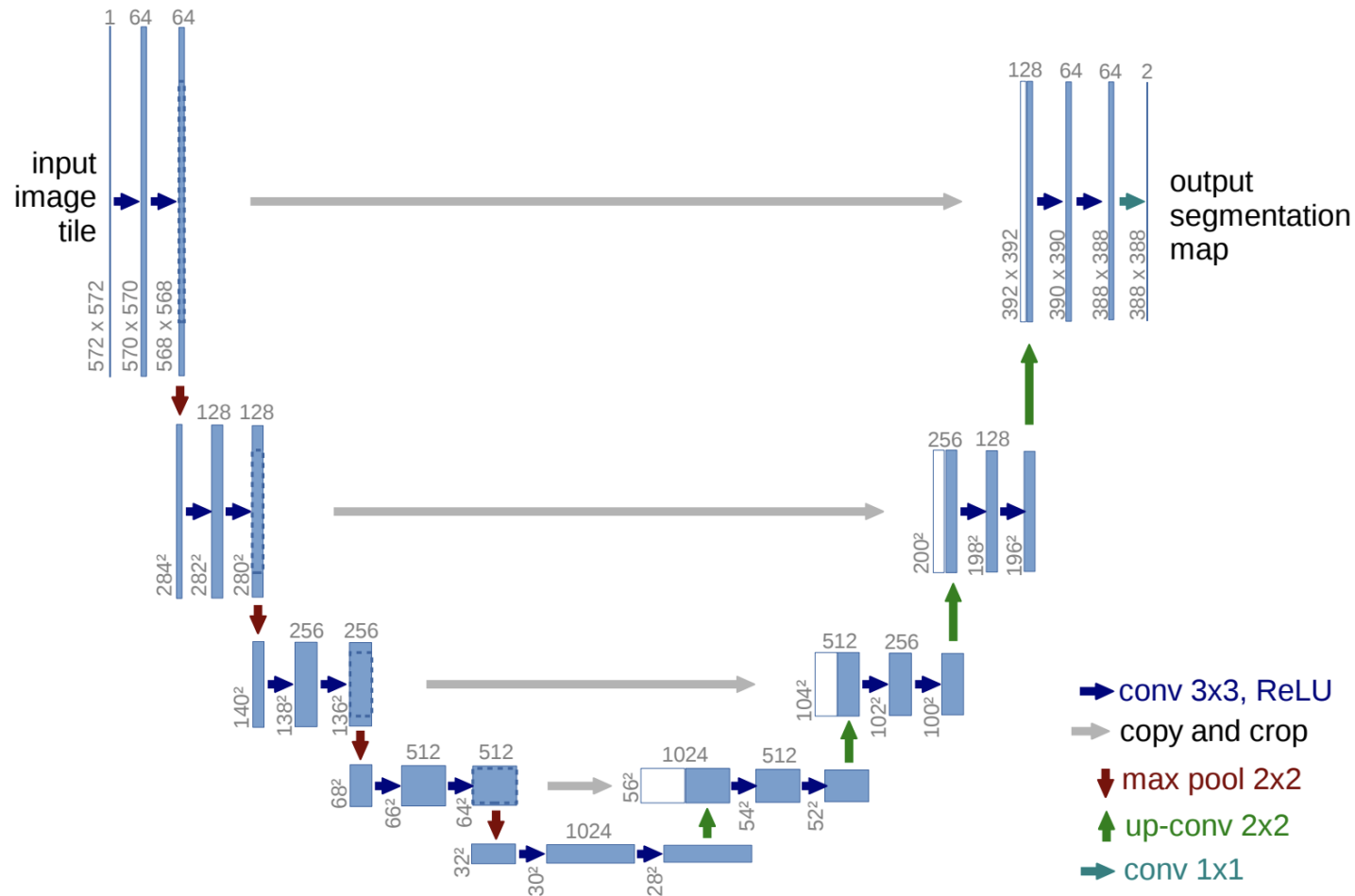
Vladlen Koltun
Intel Labs

ICLR 2016

U-Net: Convolutional Networks for Biomedical Image Segmentation

Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and BIOS Centre for Biological Signalling Studies,
University of Freiburg, Germany



<https://arxiv.org/abs/1505.04597>

<https://github.com/milesial/Pytorch-UNet>

<https://github.com/usuyama/pytorch-unet>

UNet in Pytorch

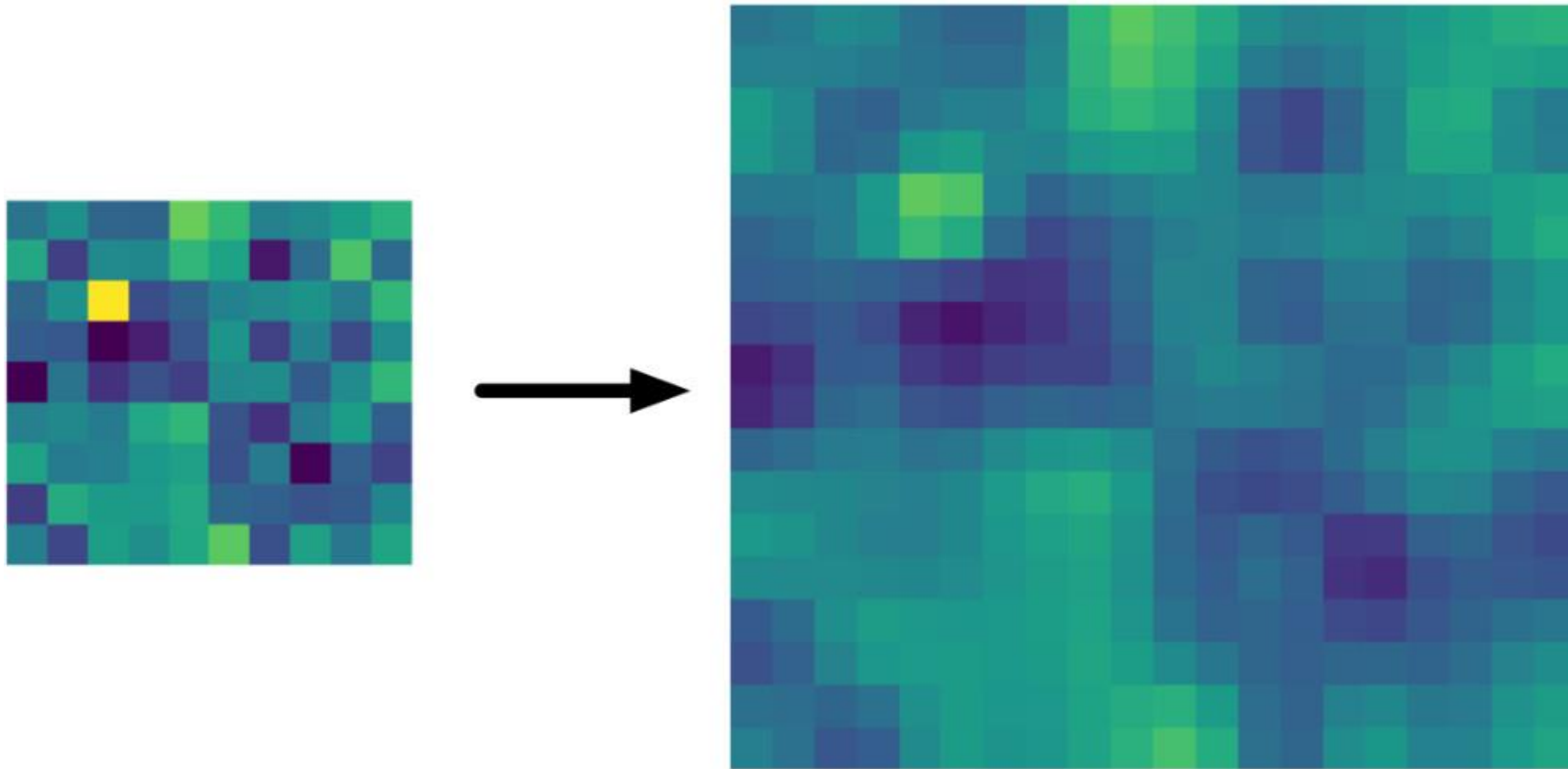
```
from .unet_parts import *

class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = (DoubleConv(n_channels, 64))
        self.down1 = (Down(64, 128))
        self.down2 = (Down(128, 256))
        self.down3 = (Down(256, 512))
        factor = 2 if bilinear else 1
        self.down4 = (Down(512, 1024 // factor))
        self.up1 = (Up(1024, 512 // factor, bilinear))
        self.up2 = (Up(512, 256 // factor, bilinear))
        self.up3 = (Up(256, 128 // factor, bilinear))
        self.up4 = (Up(128, 64, bilinear))
        self.outc = (OutConv(64, n_classes))

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits
```

Bilinear Upsampling Layer

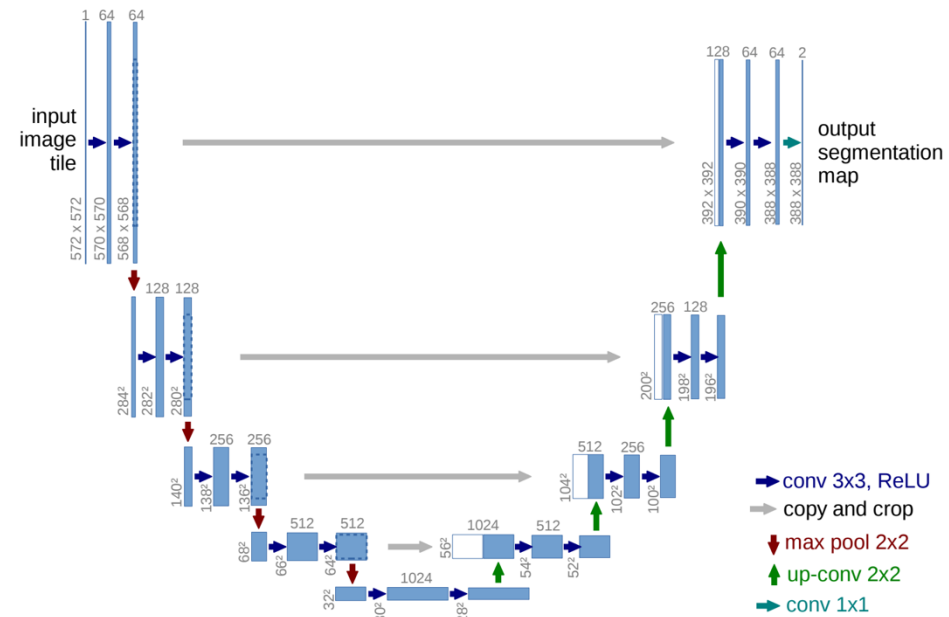


<https://machinethink.net/blog/coreml-upsampling/>

UNet in Pytorch

```
42  ✓ class Up(nn.Module):
43      """Upscaling then double conv"""
44
45  ✓  def __init__(self, in_channels, out_channels, bilinear=True):
46      super().__init__()
47
48      # if bilinear, use the normal convolutions to reduce the number of channels
49      if bilinear:
50          self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
51          self.conv = DoubleConv(in_channels, out_channels, in_channels // 2)
52      else:
53          self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
54          self.conv = DoubleConv(in_channels, out_channels)
55
56  ✓  def forward(self, x1, x2):
57      x1 = self.up(x1)
58      # input is CHW
59      diffY = x2.size()[2] - x1.size()[2]
60      diffX = x2.size()[3] - x1.size()[3]
61
62      x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
63                    diffY // 2, diffY - diffY // 2])
64      # if you have padding issues, see
65      # https://github.com/HaiyongJiang/U-Net-Pytorch-Unstructured-Buggy/commit/0e854509c2cea854e247a9c615f175f76fbb2e3a
66      # https://github.com/xiaopeng-liao/Pytorch-UNet/commit/8ebac70e633bac59fc22bb5195e513d5832fb3bd
67      x = torch.cat([x2, x1], dim=1)
68      return self.conv(x)
```

Chair segmentation - Training



Chair Segments: A Compact Benchmark for the Study of Object Segmentation

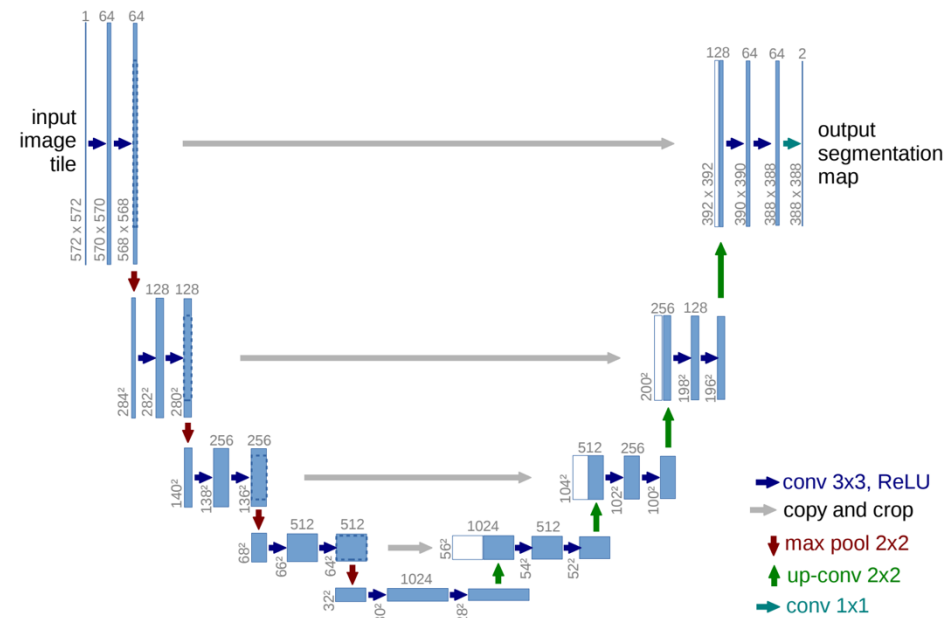
Leticia Pinto-Alva^{††}, Ian K. Torres^{‡*}, Rosangel Garcia^{§*}, Ziyang Yang[†], Vicente Ordonez[†]

[‡]Universidad Católica San Pablo, [‡]University of Massachusetts, Amherst, [§]Le Moyne College,

[†]University of Virginia

lp2rv@virginia.edu, zy3cx@virginia.edu, vicente@virginia.edu

Chair segmentation - Prediction



Chair Segments: A Compact Benchmark for the Study of Object Segmentation

Leticia Pinto-Alva^{††}, Ian K. Torres^{b*}, Rosangel Garcia^{s*}, Ziyang Yang[†], Vicente Ordonez[†]

[†]Universidad Católica San Pablo, ^bUniversity of Massachusetts, Amherst, ^sLe Moyne College,

^{††}University of Virginia

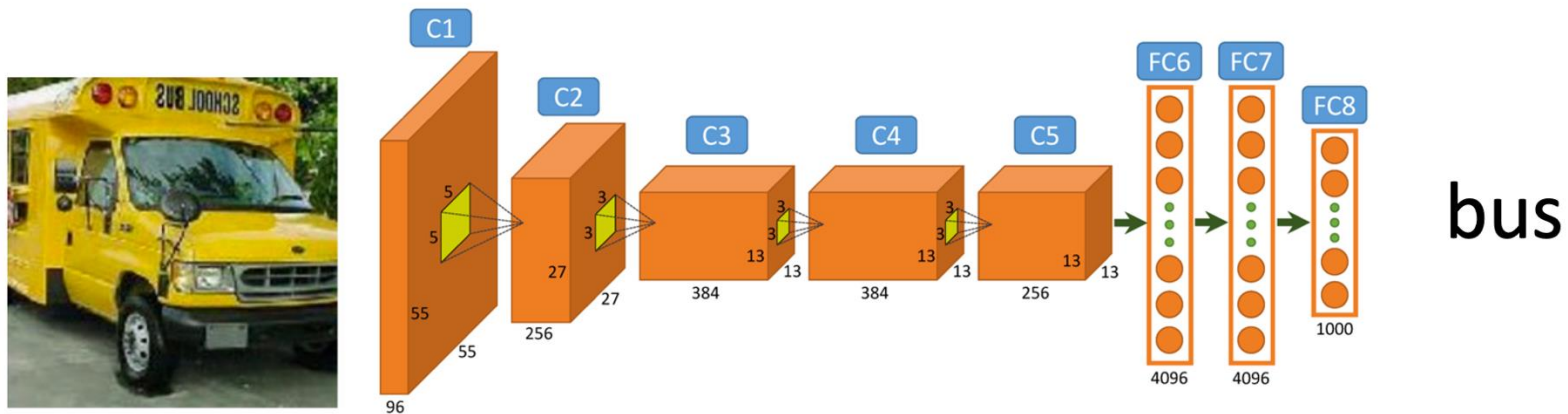
lp2rv@virginia.edu, zy3cx@virginia.edu, vicente@virginia.edu

Training Neural Networks

I

$$y = f(I; w)$$

Something gets predicted



$$w = w - \lambda \frac{\partial L}{\partial w}$$

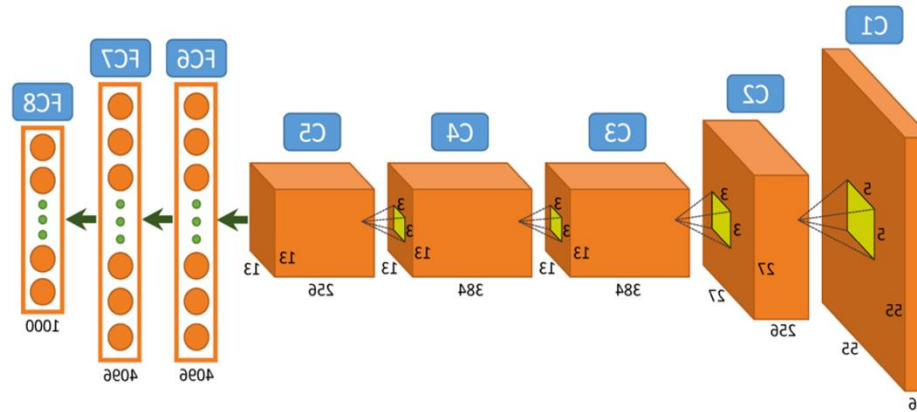
What we want

*Something
provided as
input*

$$\hat{I} = f(y; w)$$

$$L(\hat{I}, I)$$

bus



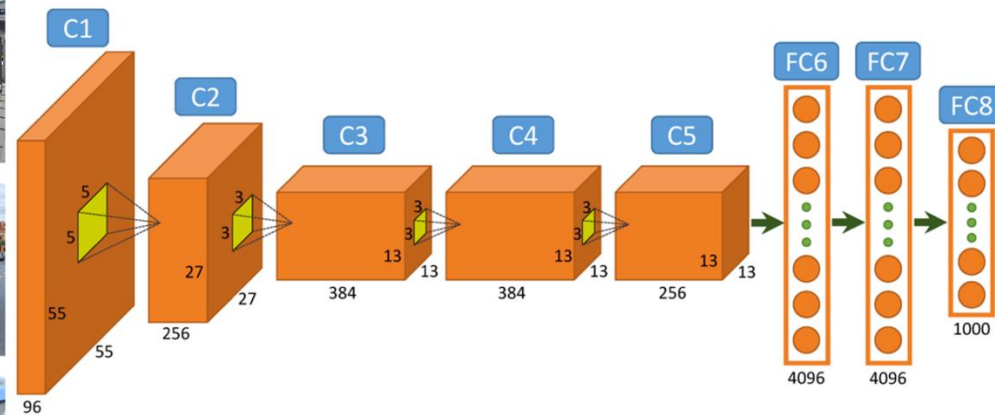
$$w = w - \lambda \frac{\partial L}{\partial w}$$

Training Neural Networks

I

$$y = f(I; w)$$

Something gets predicted



$$w = w - \lambda \frac{\partial L}{\partial w}$$

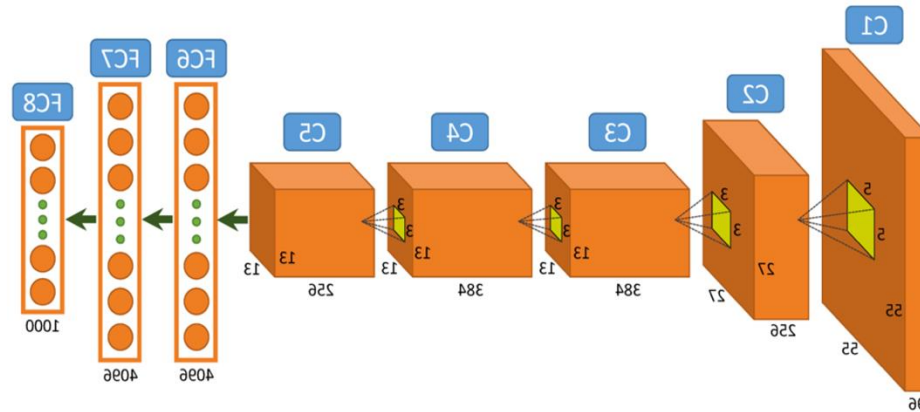
What we want

Something
provided as
input

$$\hat{I} = f(y; w)$$

$$L(\hat{I}, I)$$

bus



$$w = w - \lambda \frac{\partial L}{\partial w}$$

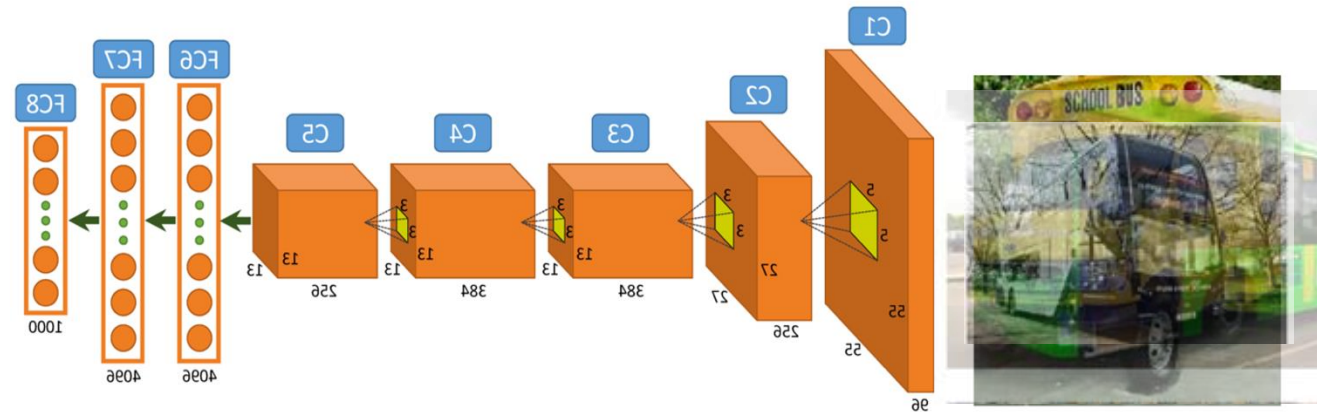
What we might get naively

Something provided as input

$$\hat{I} = f(y; w)$$

$$L(\hat{I}, I)$$

bus



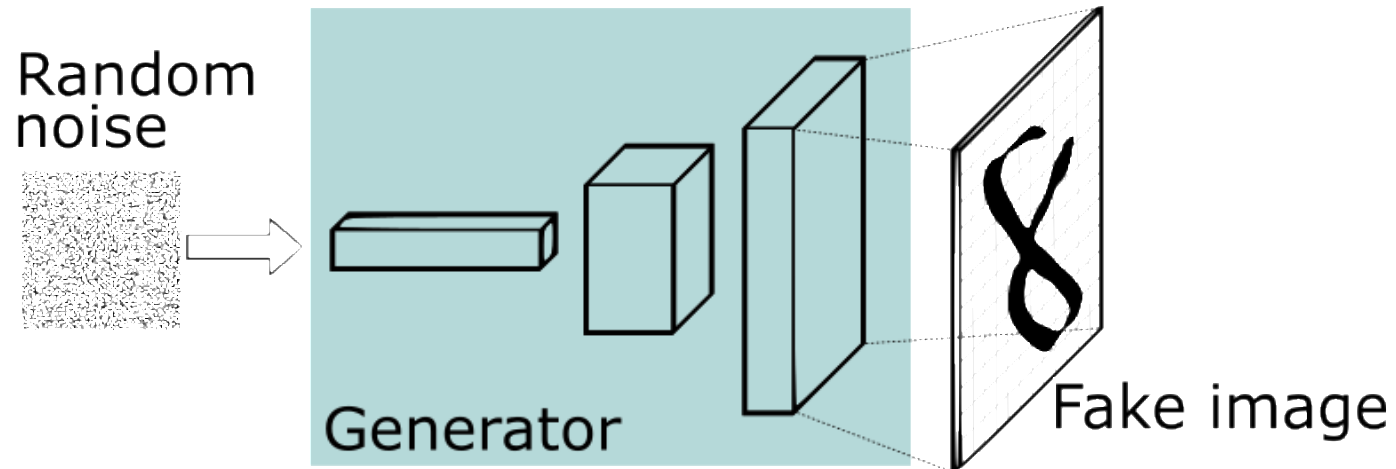
$$w = w - \lambda \frac{\partial L}{\partial w}$$

Generative Adversarial Networks (GANs)

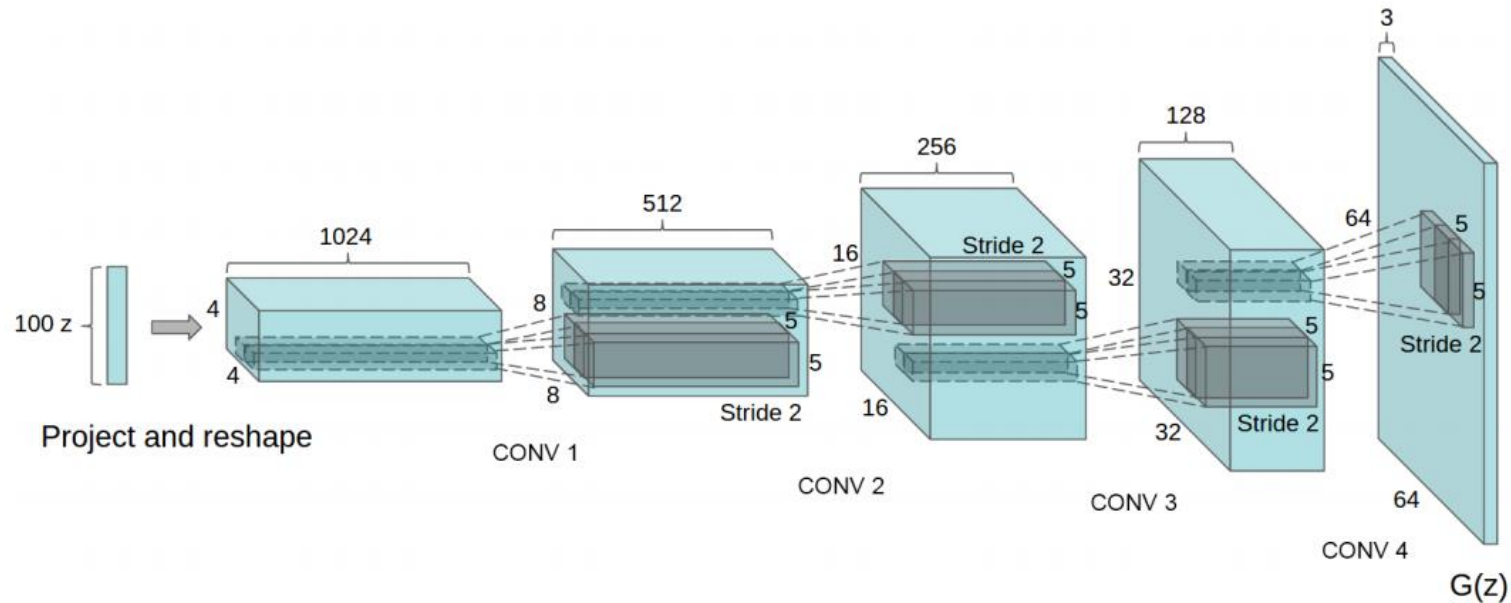
Generator and Discriminator Networks trained adversarially.

Generative Adversarial Networks (GAN)

[Goodfellow et al 2014]

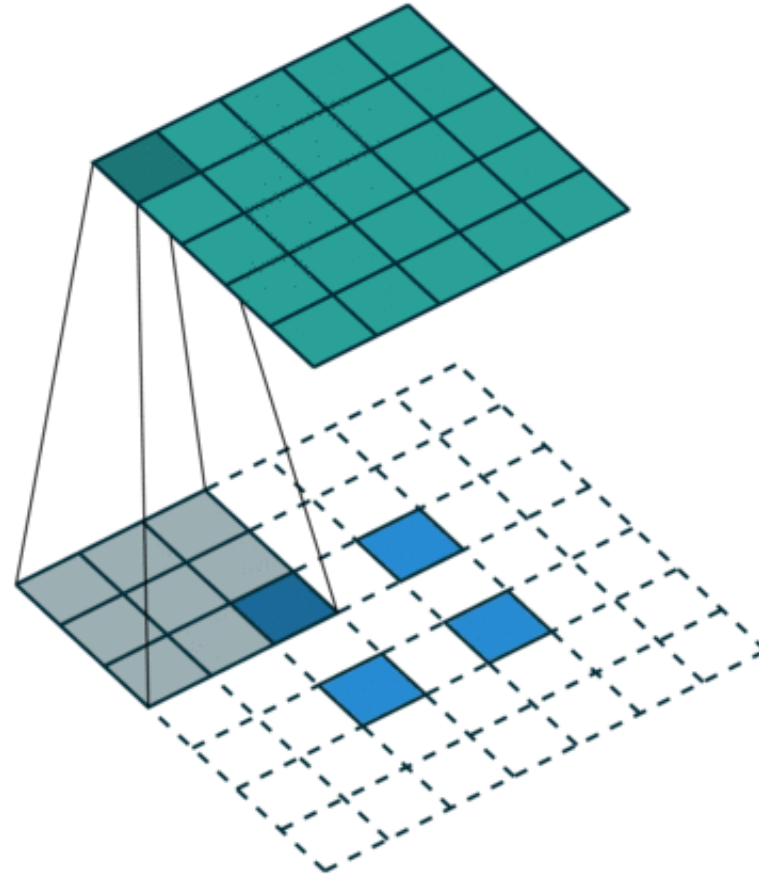


Generative Network (closer look)



Radford et. al. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. ICLR 2016

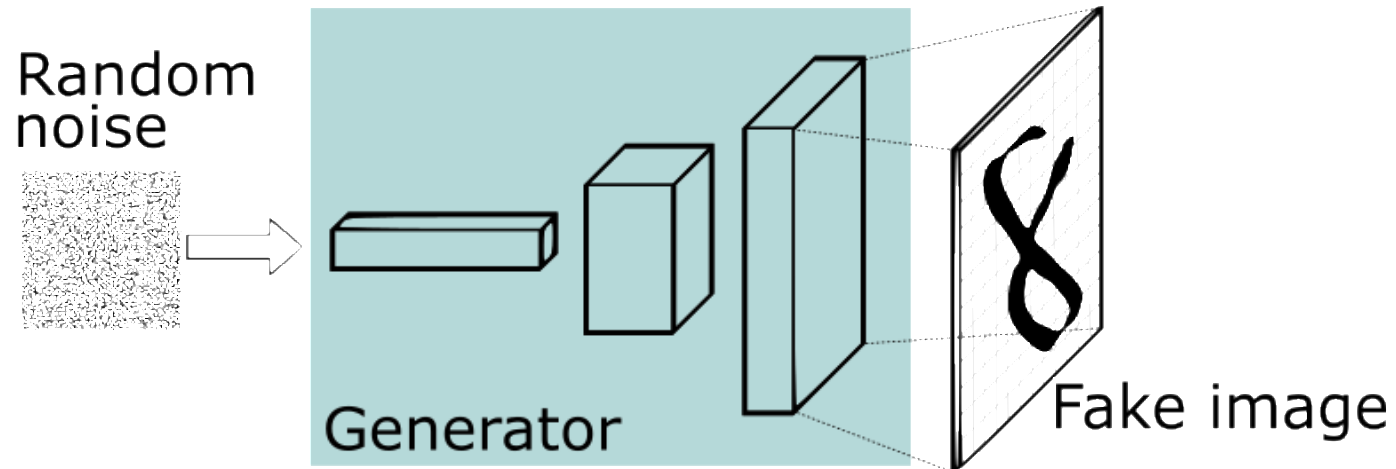
Idea 2: Up-sampling Convolutions or "Deconvolutions" or Transposed Convolutions



https://github.com/vdumoulin/conv_arithmetic

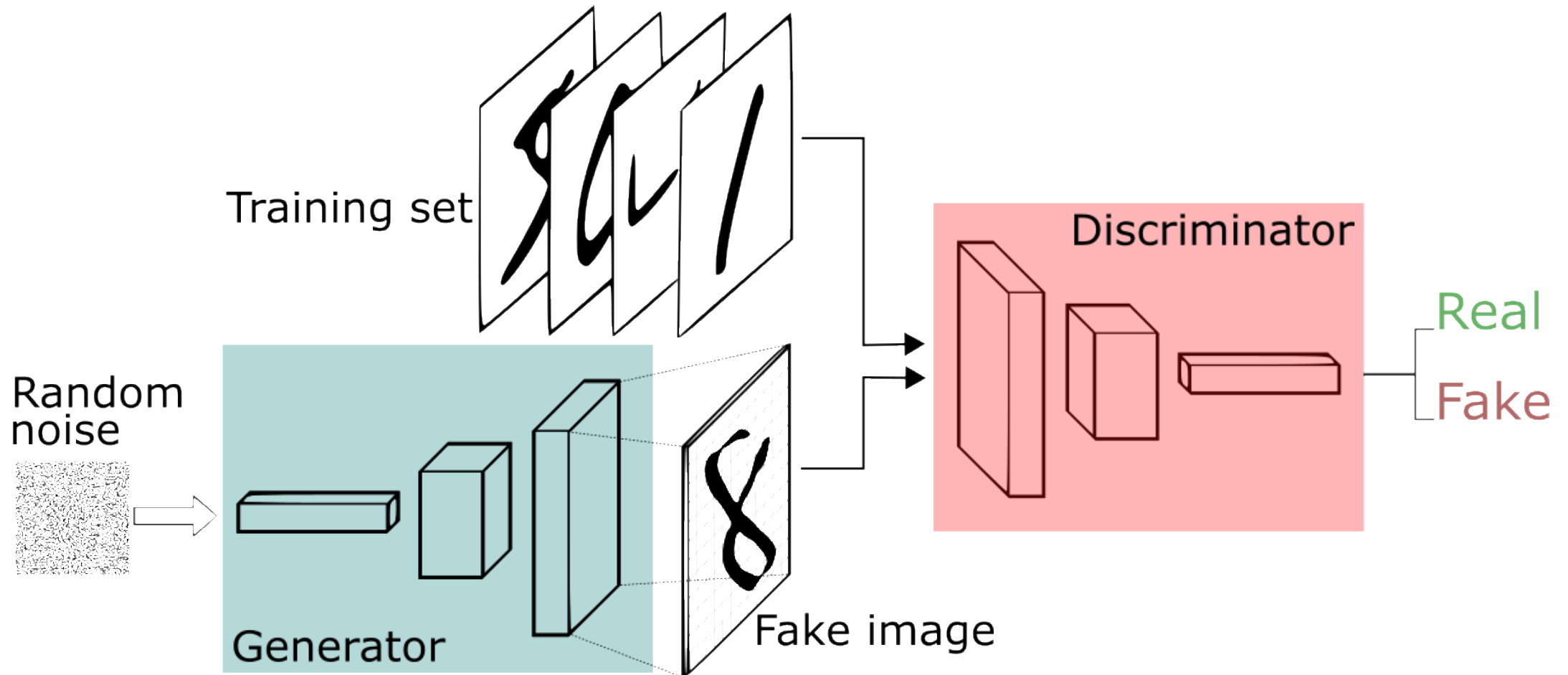
Generative Adversarial Networks (GAN)

[Goodfellow et al.]



Generative Adversarial Networks (GAN)

[Goodfellow et al.]



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log \left(1 - D(G(\mathbf{z}^{(i)})) \right) \right].$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D(G(\mathbf{z}^{(i)})) \right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log \left(1 - D(G(\mathbf{z}^{(i)})) \right) \right].$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D(G(\mathbf{z}^{(i)})) \right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Update
Discriminator
D

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log \left(1 - D(G(\mathbf{z}^{(i)})) \right) \right].$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D(G(\mathbf{z}^{(i)})) \right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Update
Generator
G

Until
Desirable
Results are
Achieved?

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(\mathbf{x}^{(i)}) + \log \left(1 - D(G(\mathbf{z}^{(i)})) \right) \right].$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

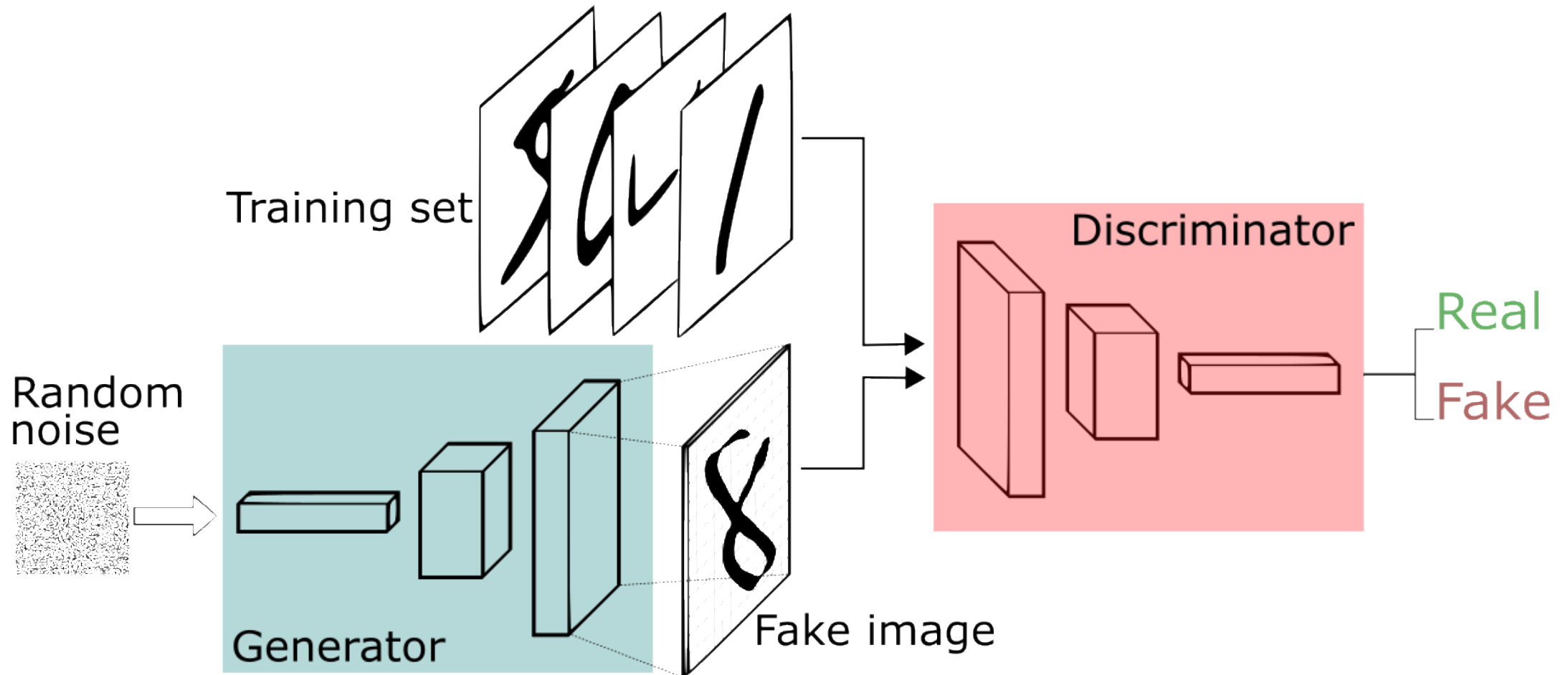
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log \left(1 - D(G(\mathbf{z}^{(i)})) \right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

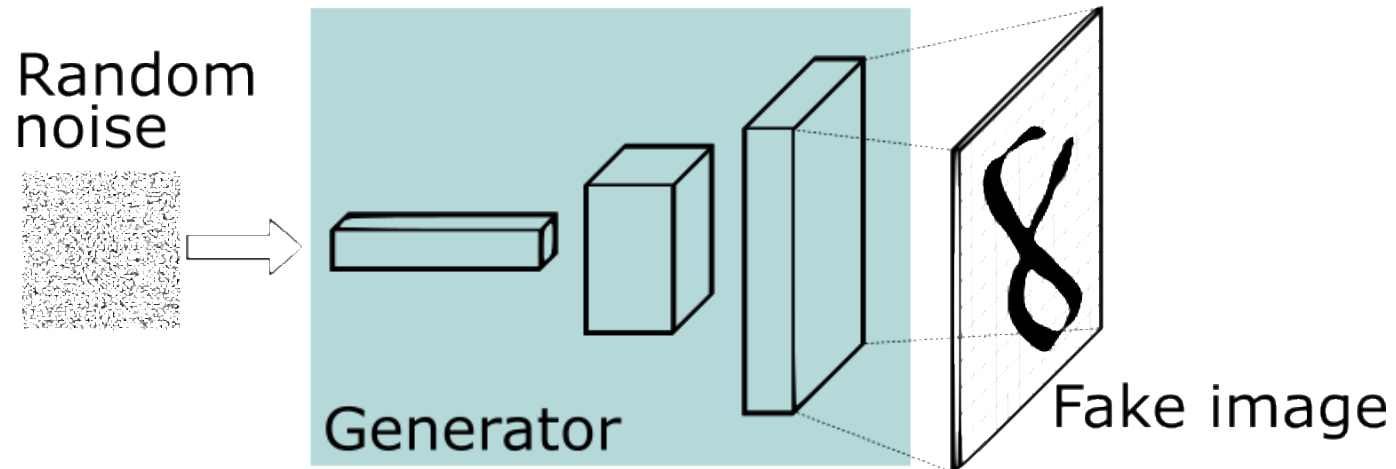
Generative Adversarial Networks (GAN)

[Goodfellow et al.]



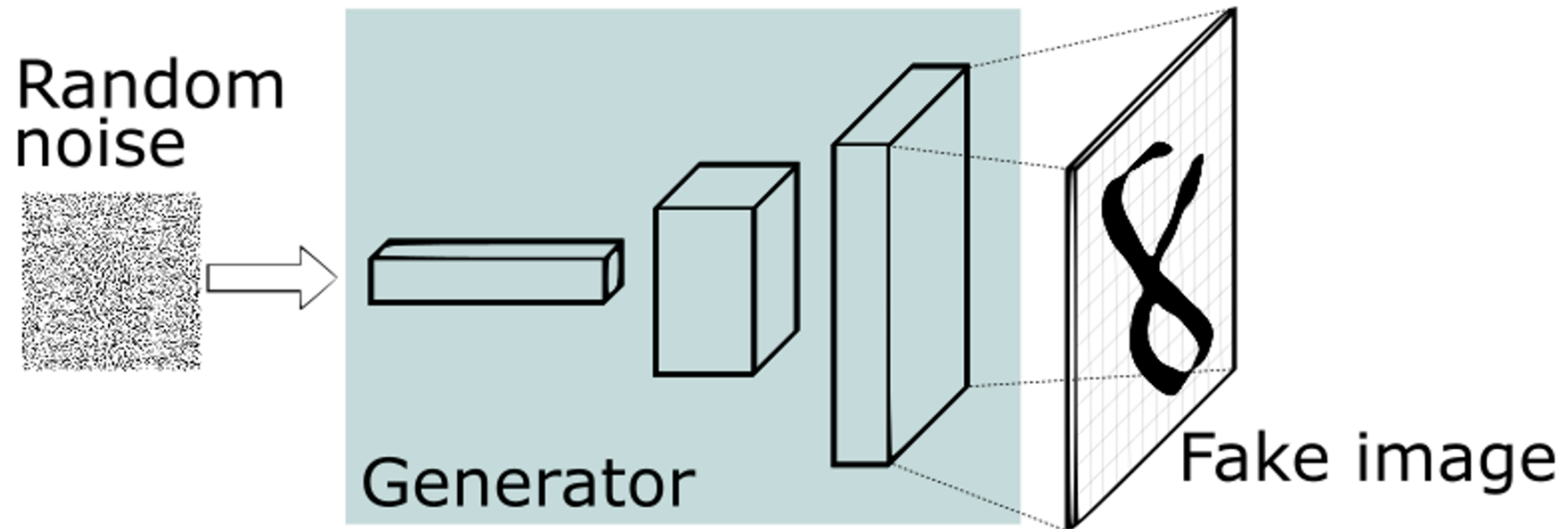
Generative Adversarial Networks (GAN)

[Goodfellow et al.]



Generative Adversarial Networks (GAN)

[Goodfellow et al.]



<https://shorturl.at/pz6tk>

NVIDIA's progressive GANs ICLR 2018



Google's BigGAN



Google's BigGAN

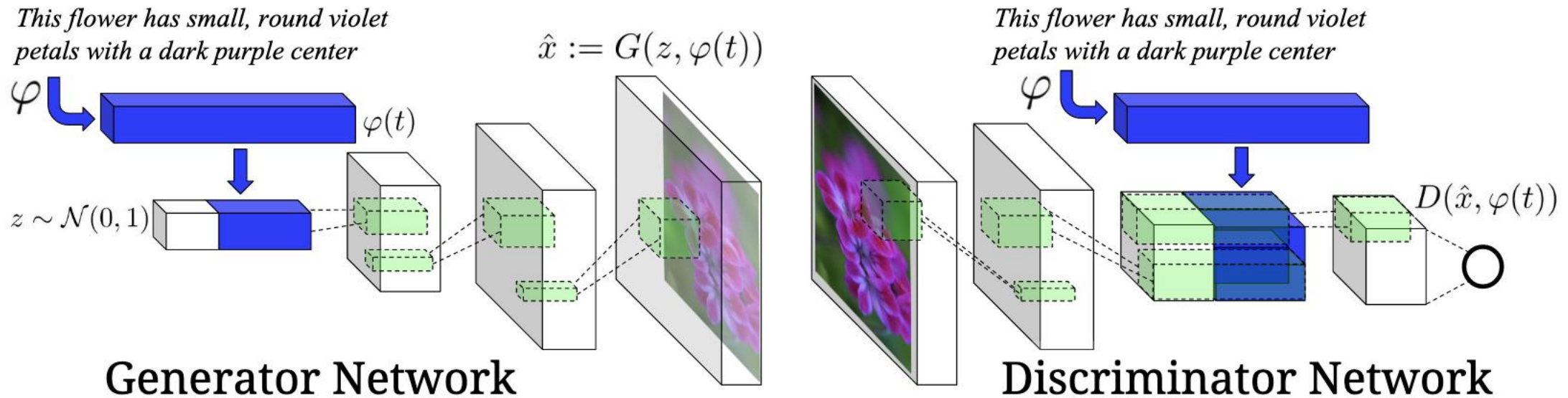
Teddy Bear



Microphone



Conditional GANs / Text-conditioned



Conditional GANs / Text-conditioned

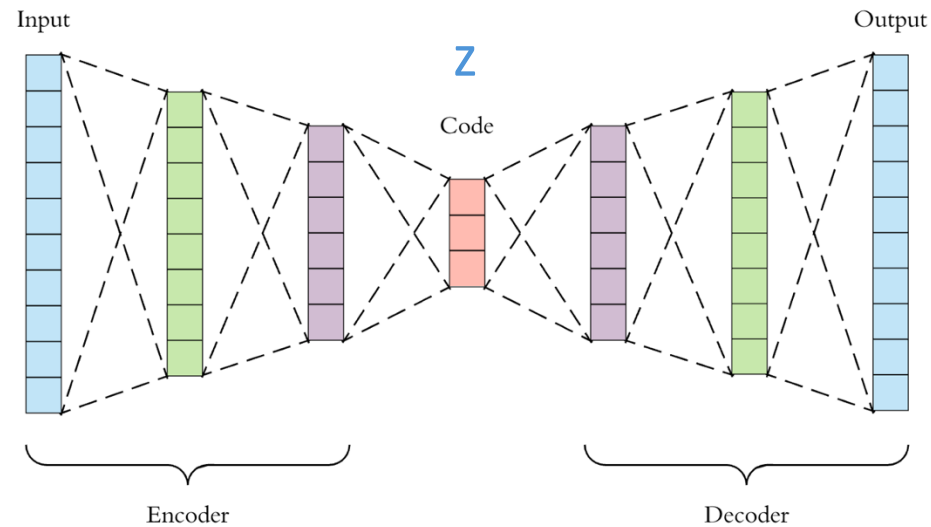
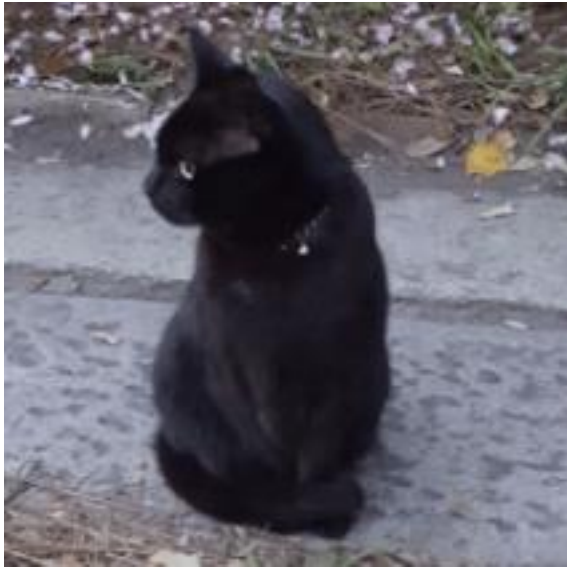
this small bird has a pink
breast and crown, and black
primaries and secondaries.



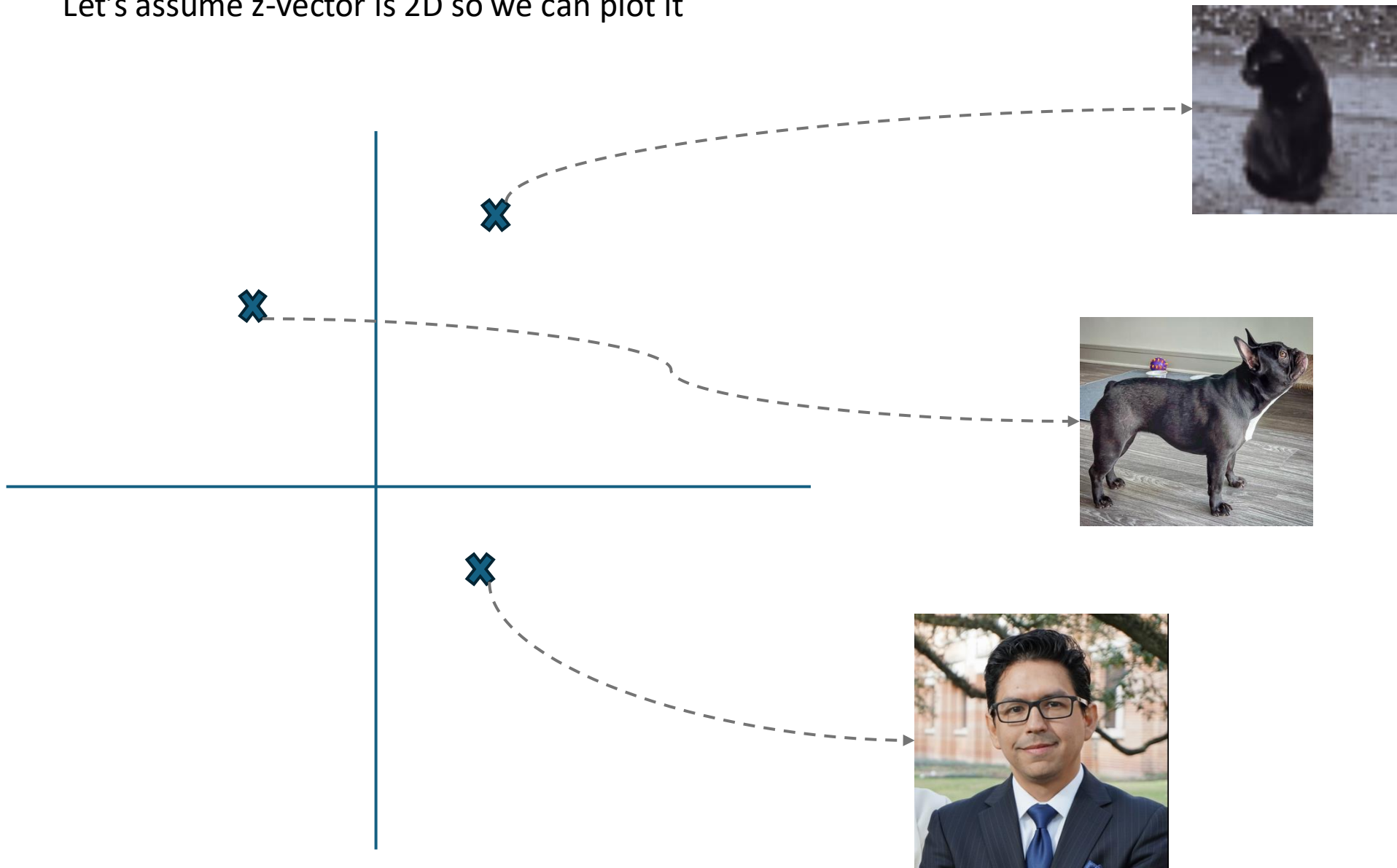
Variational AutoEncoders (VAEs)

Training a network with the identity function

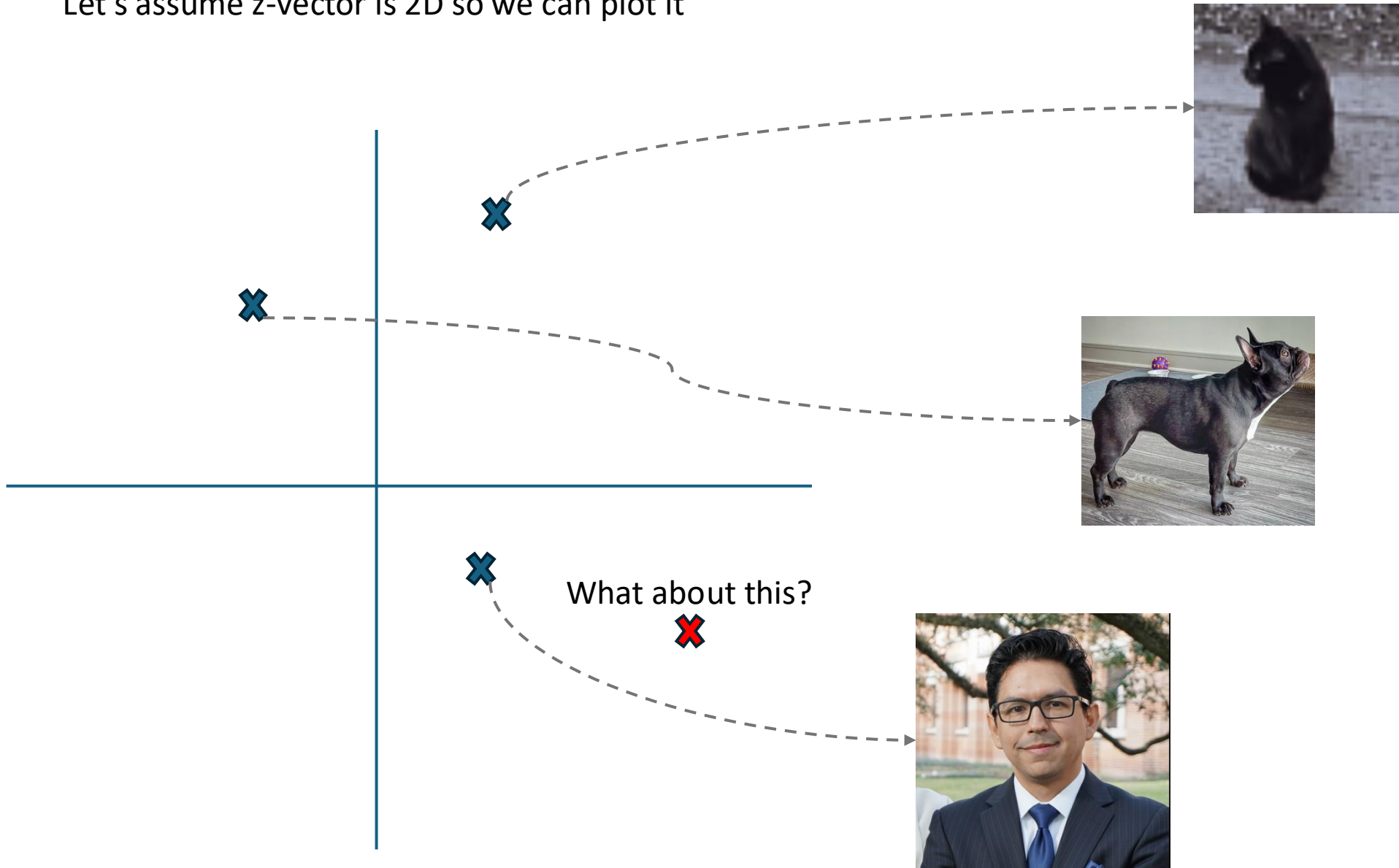
AutoEncoder Models (Downsample, Upsample)



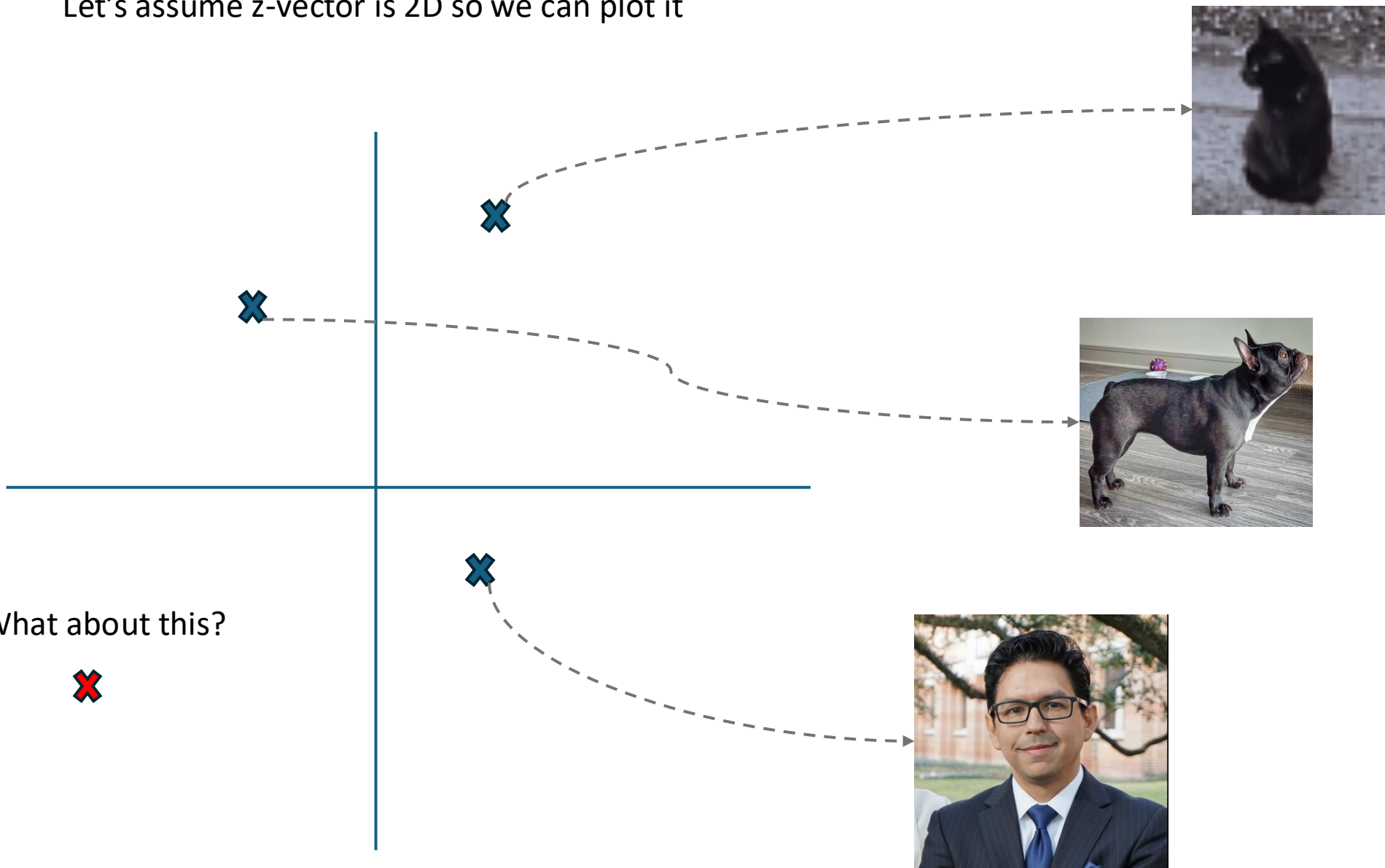
Let's assume z-vector is 2D so we can plot it



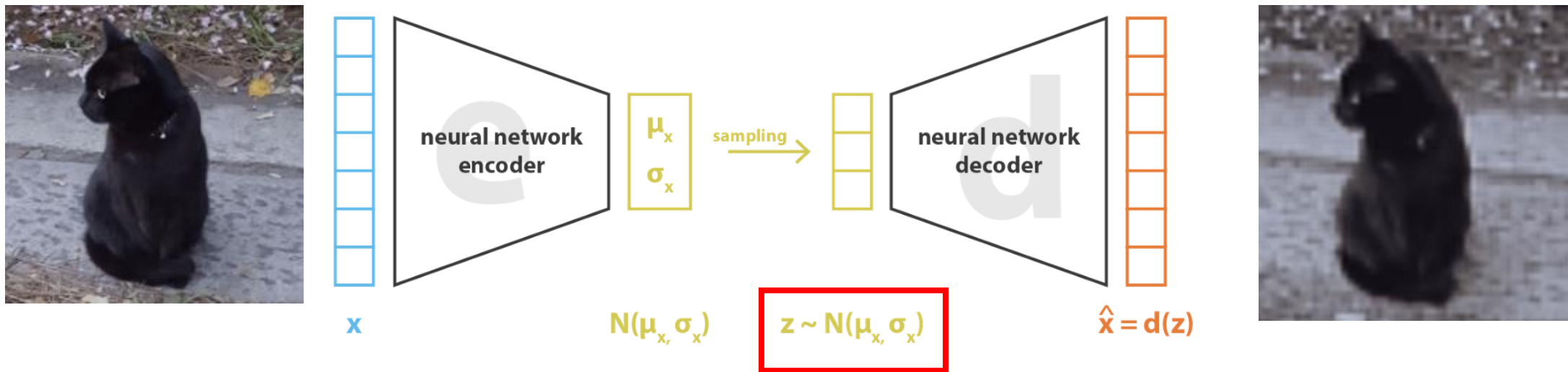
Let's assume z-vector is 2D so we can plot it



Let's assume z-vector is 2D so we can plot it



Variational AutoEncoders (VAE)



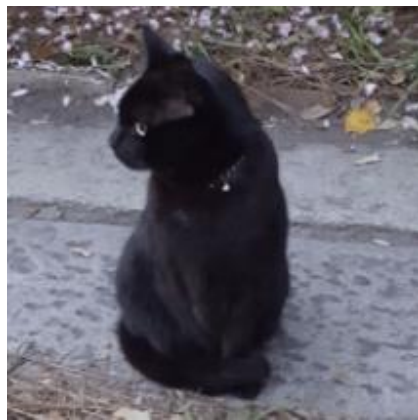
$$\text{loss} = \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

Reparameterization “trick”

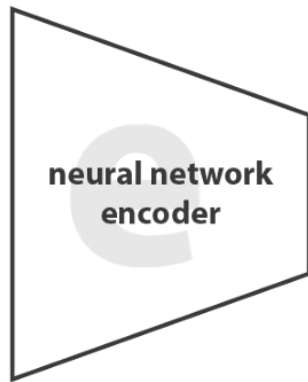
$$z = z_mean + sigma * epsilon$$

$$sigma = exp(z_log_var/2)$$

$$\epsilon \sim \text{Normal}(0,1)$$



x



neural network
encoder

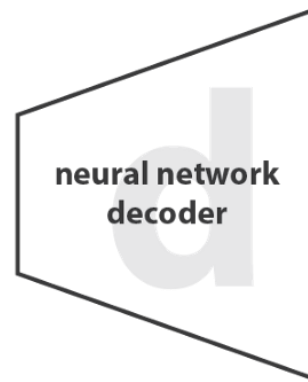


$N(\mu_x, \sigma_x)$

sampling →



$z \sim N(\mu_x, \sigma_x)$



neural network
decoder

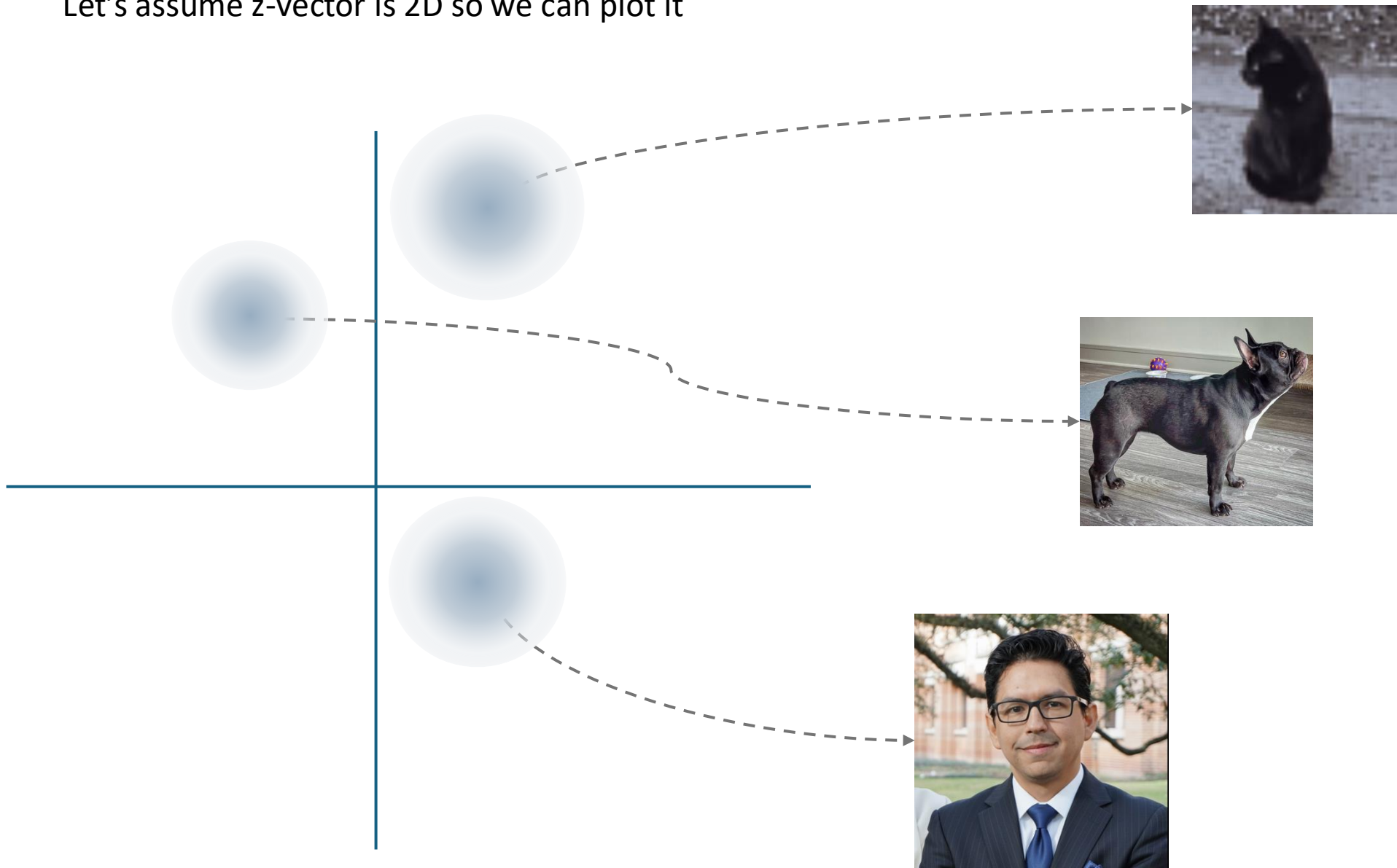


$\hat{x} = d(z)$

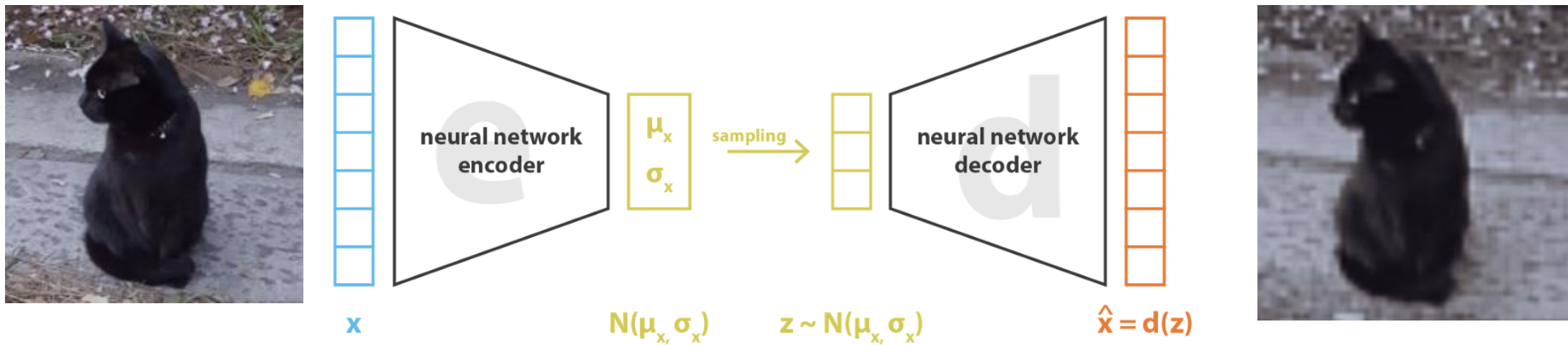


$$\text{loss} = \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

Let's assume z-vector is 2D so we can plot it



KL-Divergence Regularization Loss



$$\text{loss} = \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

Kullback-Leibler Divergence

$$D_{\text{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right)$$

For Gaussian Distributions KL Divergence even simpler

$$D_{KL}(\mathcal{N}(\mu, \sigma^2) || \mathcal{N}(0, 1)) = \frac{1}{2} \sum_{i=1}^k (\mu_i^2 + \sigma_i^2 - \ln \sigma_i^2 - 1)$$

```
def loss_function(self,
                  *args,
                  **kwargs) -> dict:
    recons = args[0]
    input = args[1]
    mu = args[2]
    log_var = args[3]

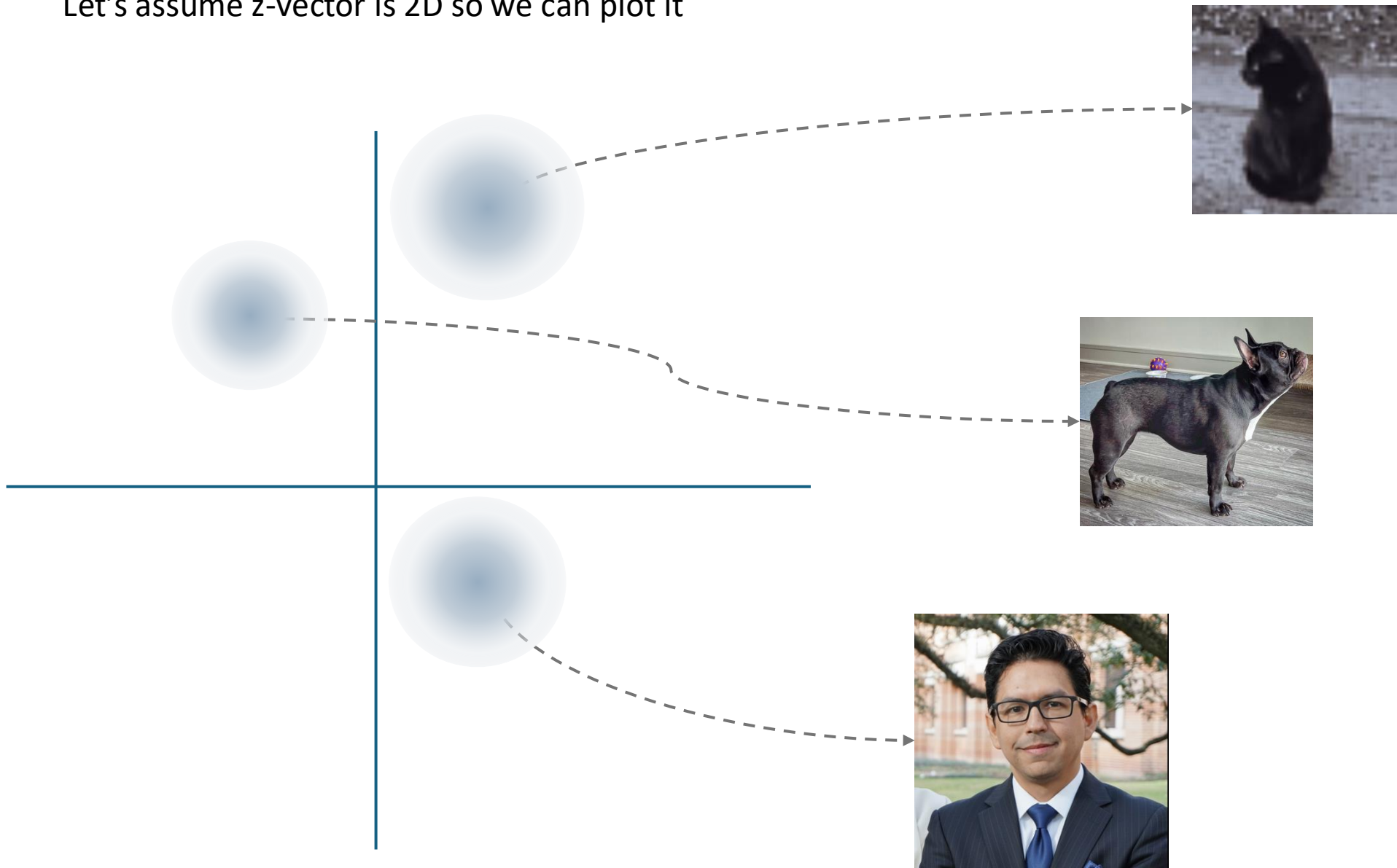
    kld_weight = kwargs['M_N'] # Account for the minibatch samples from the dataset
    recons_loss =F.mse_loss(recons, input)

    kld_loss = torch.mean(-0.5 * torch.sum(1 + log_var - mu ** 2 - log_var.exp(), dim = 1), dim = 0)

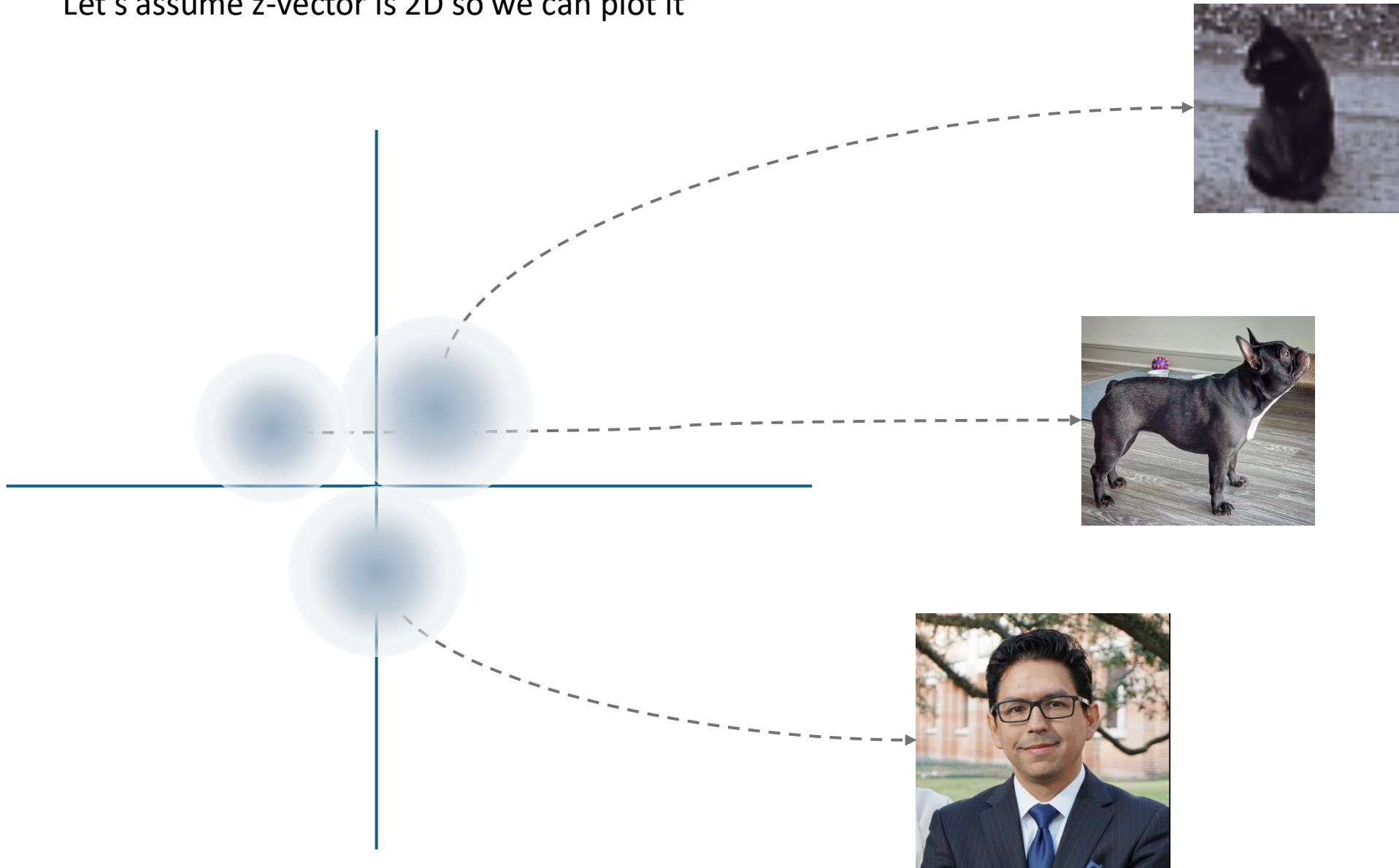
    loss = recons_loss + kld_weight * kld_loss
    return {'loss': loss, 'Reconstruction_Loss':recons_loss, 'KLD':-kld_loss}
```

<https://github.com/AntixK/PyTorch-VAE/blob/master/models/cvae.py>

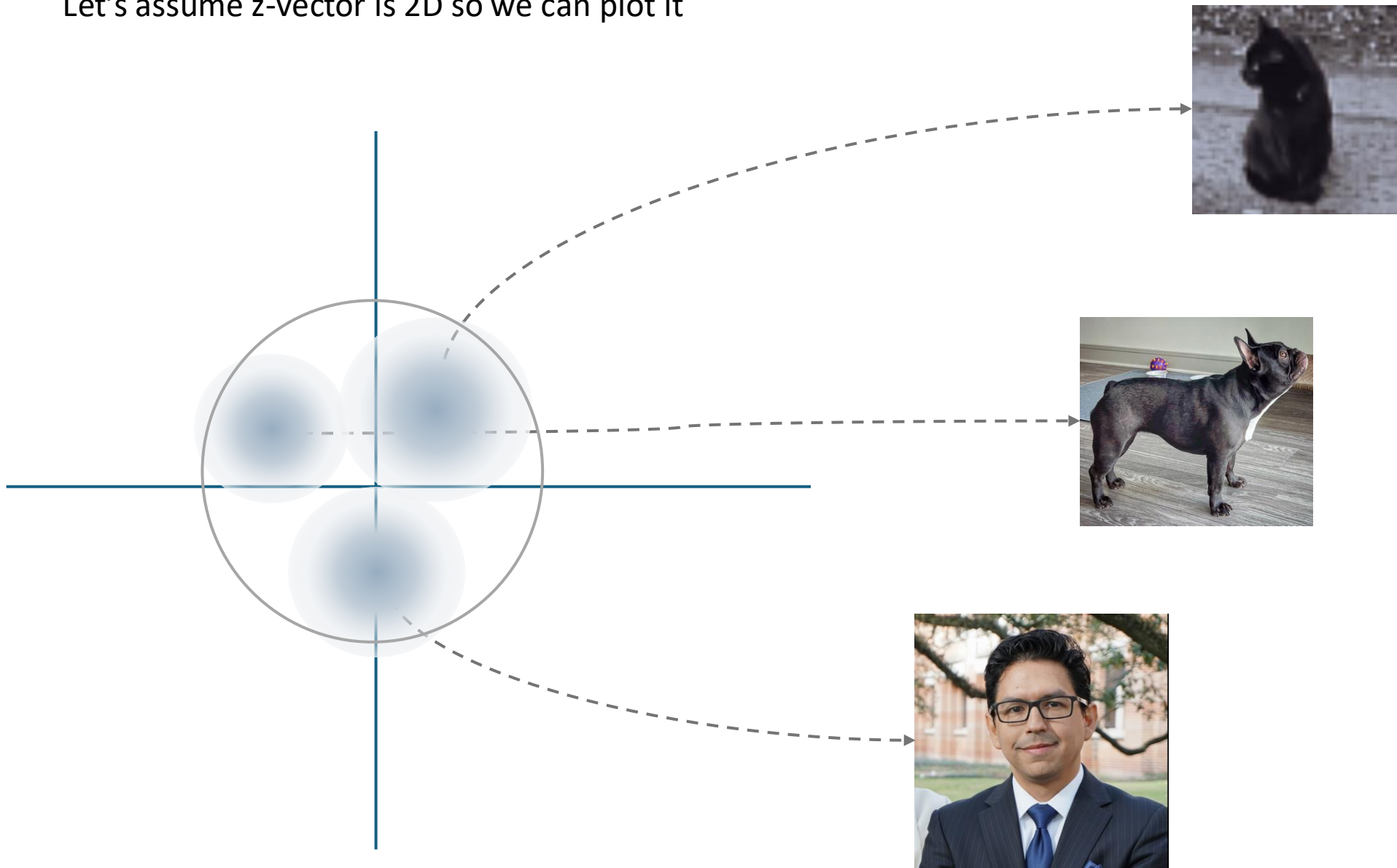
Let's assume z-vector is 2D so we can plot it



Let's assume z-vector is 2D so we can plot it



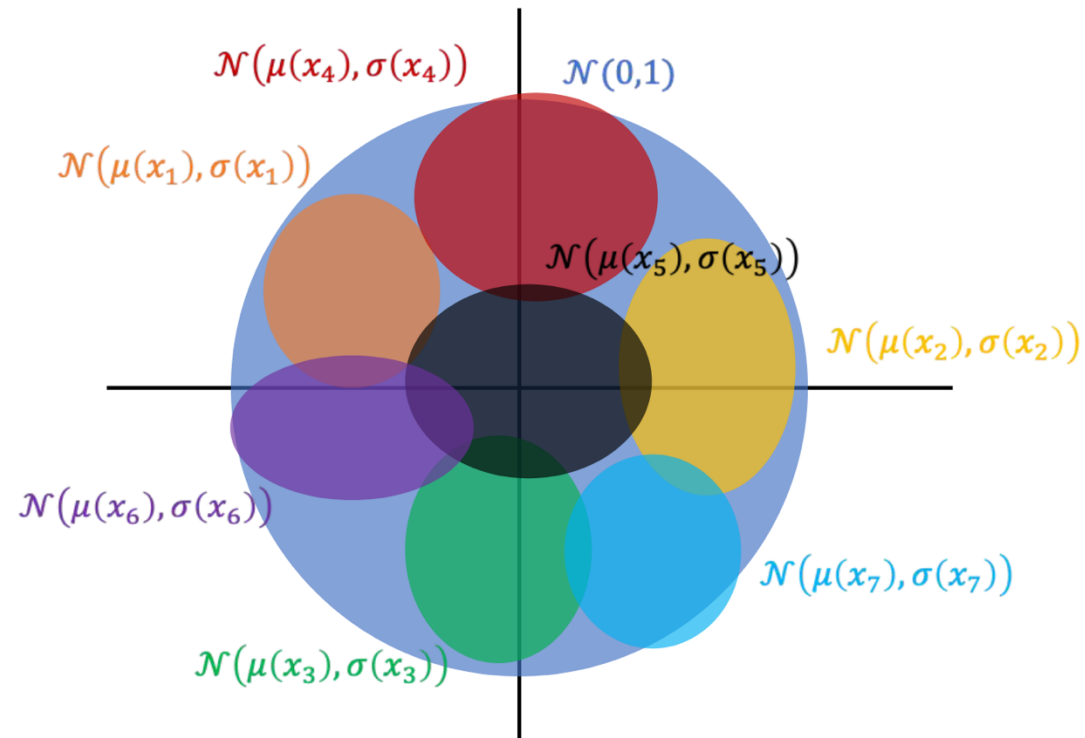
Let's assume z-vector is 2D so we can plot it



Encoding different points into latent space

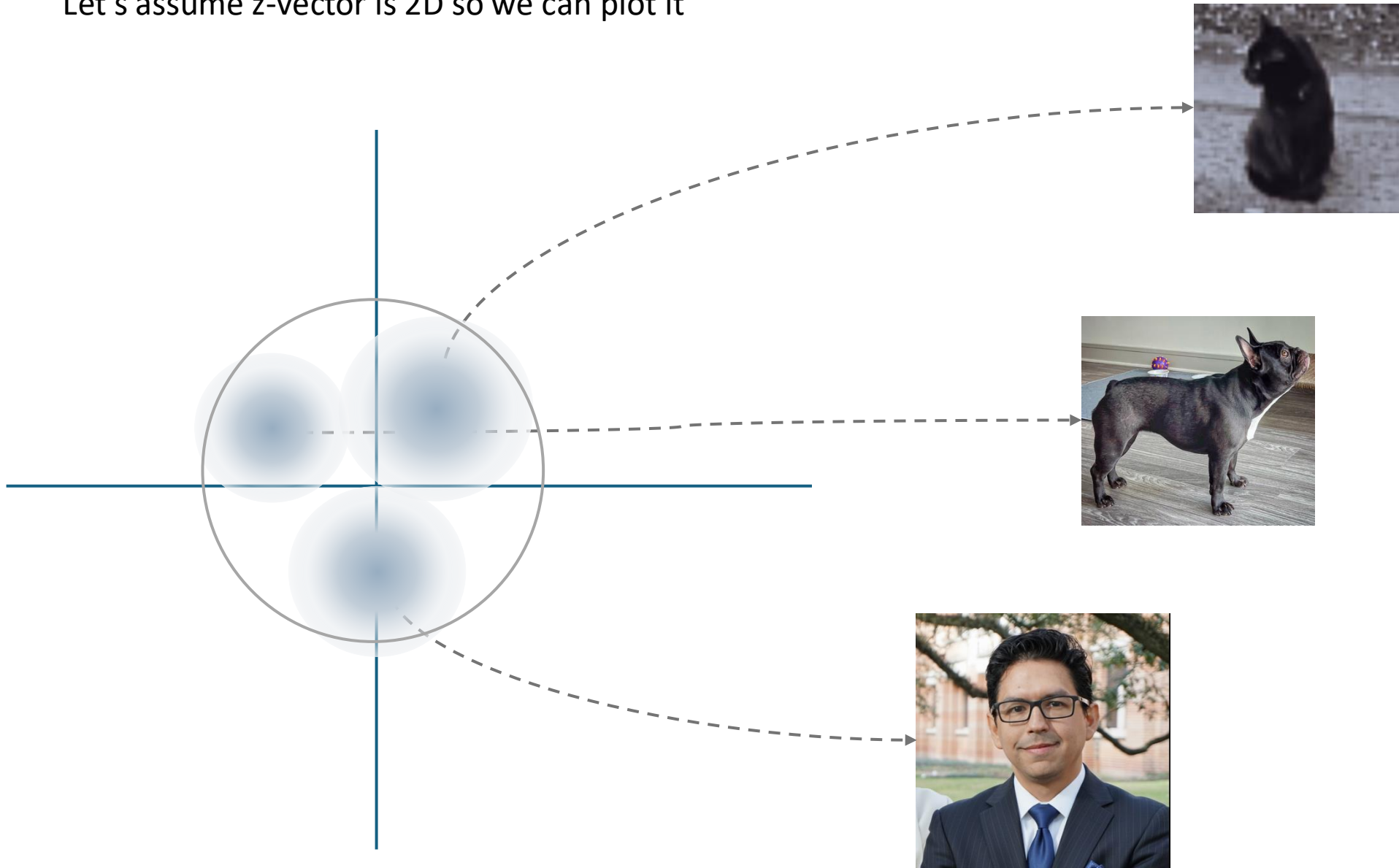
$$L = ||x - \hat{x}||_2^2 + \lambda D_{KL}(\mathcal{N}(\mu, \sigma), \mathcal{N}(0, 1))$$

Because of our KL divergence loss, the $\mathcal{N}(\mu, \sigma)$ for any input data point has to be somewhat similar to $\mathcal{N}(0, 1)$

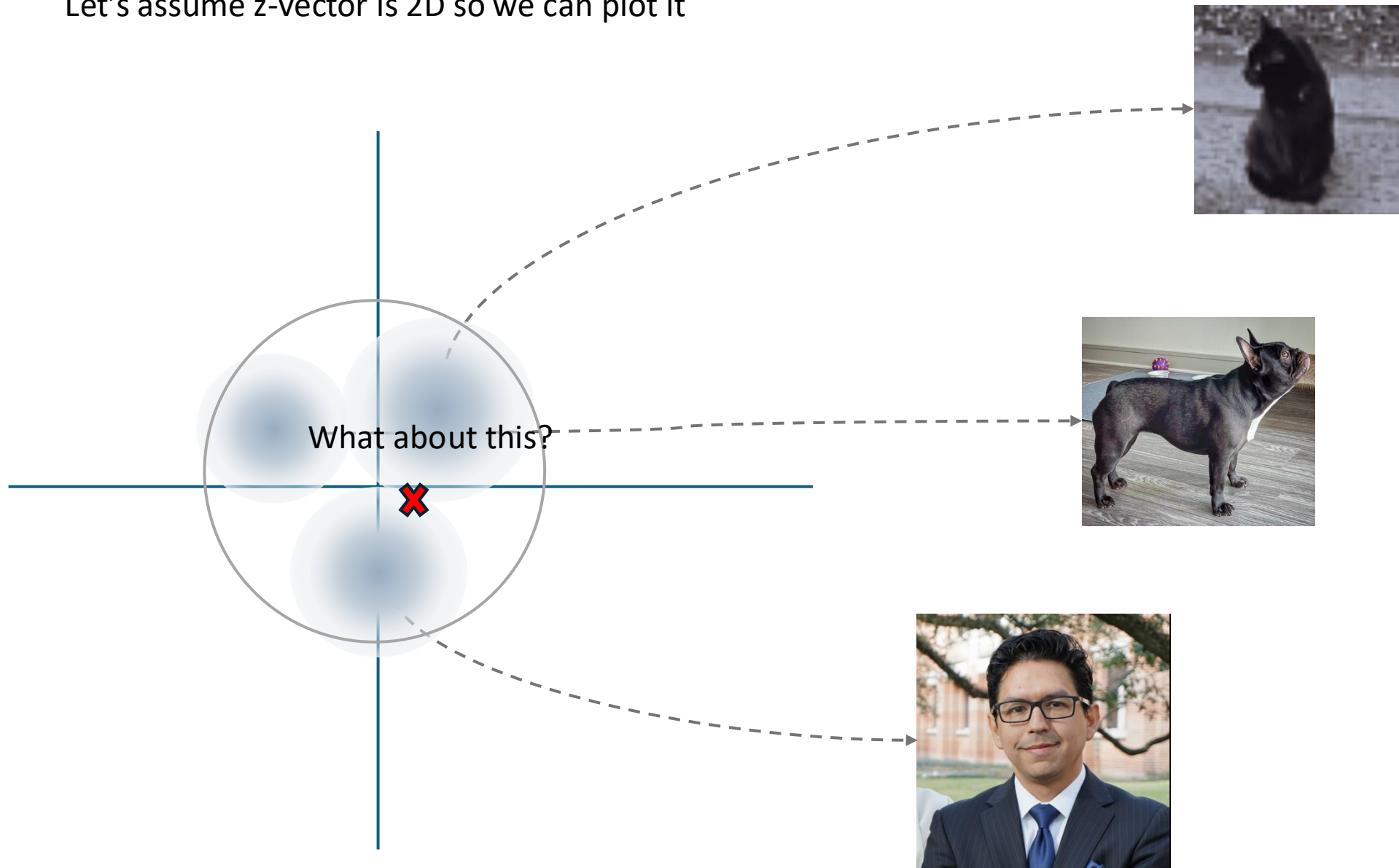


So, if we sample a point from $\mathcal{N}(0, 1)$, it is very likely to fall within one of these encoded

Let's assume z-vector is 2D so we can plot it



Let's assume z-vector is 2D so we can plot it



Let's assume z-vector is 2D so we can plot it

