50.039 Theory and Practice of Deep Learning

W10-S2 Generative Models in Deep Learning

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 10)

1. What are typical **generative models** in deep learning?
2. What is an **autoencoder** and what are its uses?
3. What is a **fractionally strided convolution layer**?
4. What are **variational autoencoders** and why do **noise representations of latent features** work better than the ones in standard autoencoders?
5. What are **Generative Adversarial Networks** (**GANs**) and their uses?
6. What are the **advanced techniques** in GANs and deepfakes?

About this week (Week 10)

1. What are typical **generative models** in deep learning?
2. What is an **autoencoder** and what are its uses?
3. What is a **fractionally strided convolution layer**?
4. What are **variational autoencoders** and why do **noise representations of latent features** work better than the ones in standard autoencoders?
5. What are **Generative Adversarial Networks** (**GANs**) and their uses?
6. What are the **advanced techniques** in GANs and deepfakes?

Outline

In this lecture

- Main issues of traditional Autoencoders
- Diversity on feature representation using stochastic latent representations
- Variational Autoencoder and stochastic latent representations
- Basic Generative Adversarial Networks: ideas and procedure.

In the next lectures

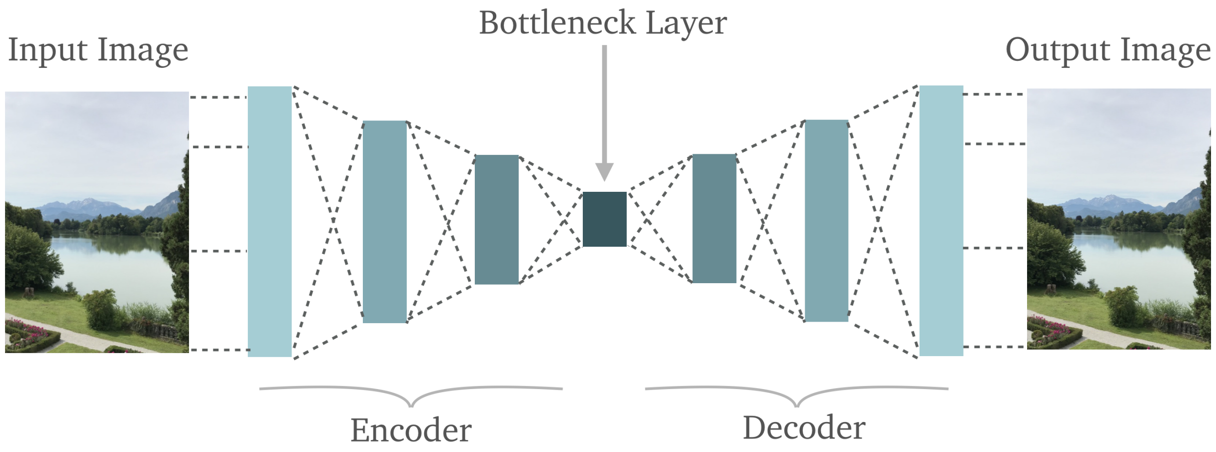
- More advanced concepts on GANs

AutoEncoders: a definition

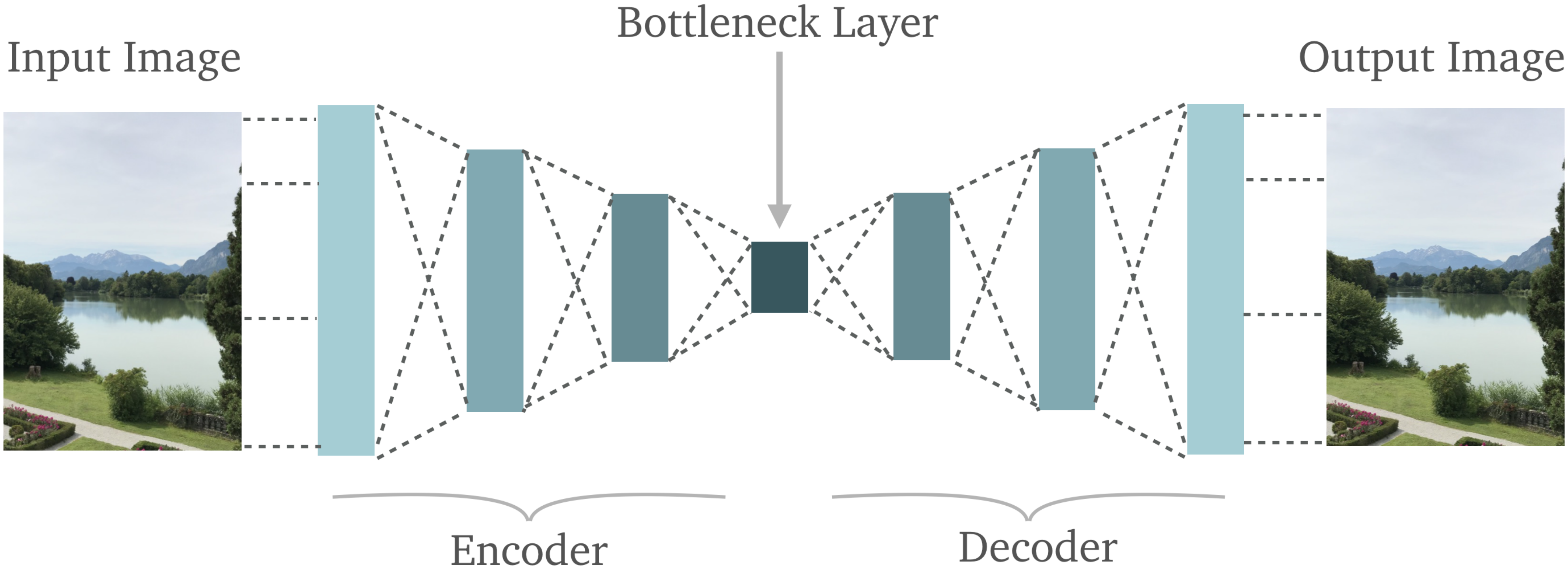
Definition (AutoEncoder):

An **AutoEncoder** is a neural network that learns to **copy its input to its output**. Basically, it attempts to approximate the identity function.

It has an internal (hidden) layer that describes a code used to represent the input, and is called the **feature representation** or **latent representation of the input**.



AutoEncoders: a definition



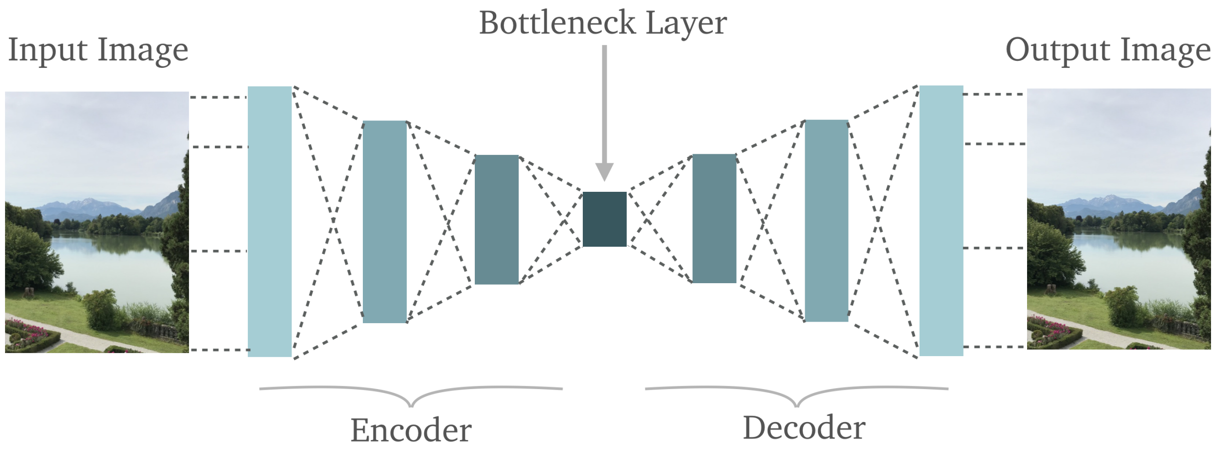
AutoEncoders: a definition

Definition (AutoEncoder):

An **AutoEncoder** is a neural network that learns to **copy its input to its output**.

It is constituted of two main parts:

- an **encoder** that maps the input into the code,
- and a **decoder** that maps the code to a reconstruction of the input.

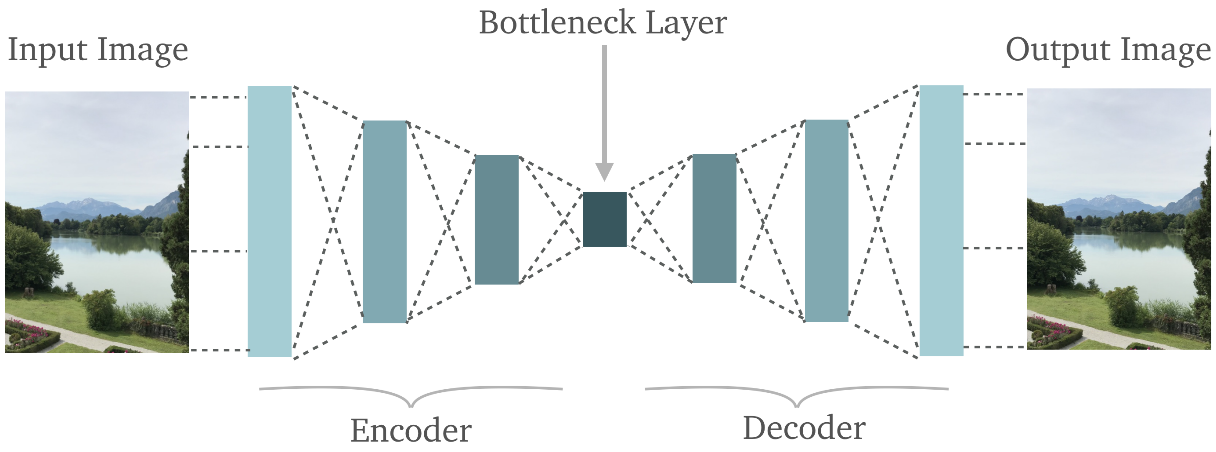


AutoEncoders: a definition

Definition (AutoEncoder):

Autoencoders are usually restricted in their **latent representation dimensionality**.

This forces them to reconstruct the input approximately, **preserving only the most relevant aspects of the data in the copy**.



AutoEncoders structure

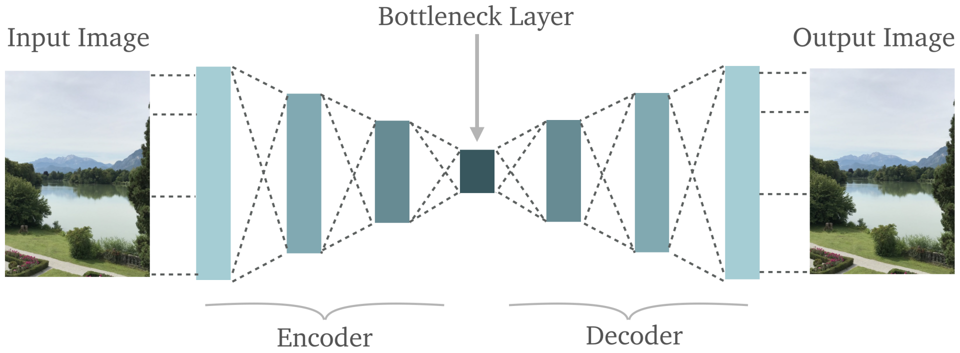
Reconstruction Loss: This is the method that measures measure how well the decoder is performing and how close the output is to the original input.

In the case of images, we can typically use an MSE on all pixels, or variations of it.

Example: input x on the right, output y on the left.

$$MSE(x, y) = \sqrt{\sum_{i,j} \left(x_{i,j} - y_{i,j}\right)^2}$$

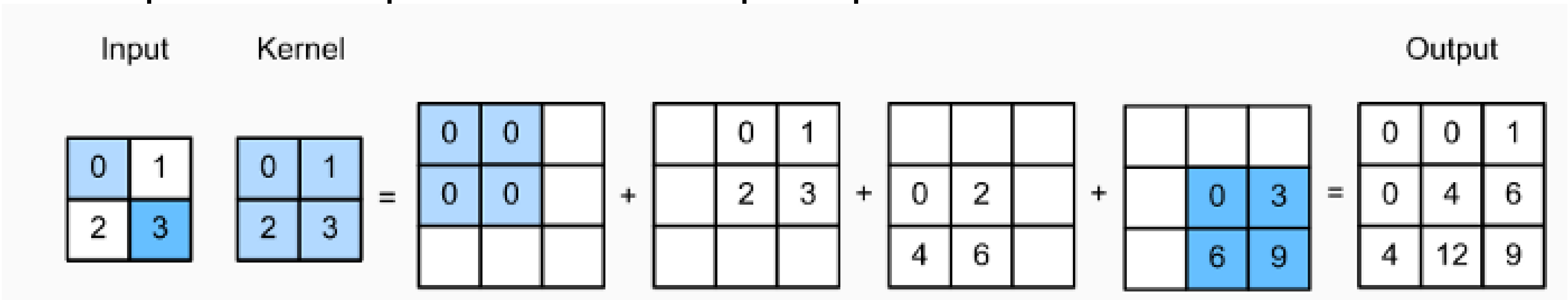
Deconvolution operation



Definition (Deconvolution layer):

The **Deconvolution** (also commonly referred to as **Transposed Convolution**, or **Fractionally Strided Convolution**) **layer** is used to **upsample the input feature map** to a **desired output feature** map using **some learnable parameters**.

Works as a convolution, multiply input with kernel elements but copy output in multiple locations to upsample.



Deconvolution operation

Deconvolution layers are trainable layers, which allow to upsample an input (as opposed to downsampling Convolution layers).

They are easily implemented on PyTorch, using the **ConvTranspose** types of layers (exists in 1D, 2D, and n-D if needed).

```
# A standard 2D Convolution
conv = nn.Conv2d(in_channels = 8, \
                 out_channels = 8, \
                 kernel_size = 5)
```

```
x = torch.randn(2, 8, 64, 64)
print(x.shape)
```

```
torch.Size([2, 8, 64, 64])
```

```
y = conv(x)
print(y.shape)
```

```
torch.Size([2, 8, 60, 60])
```

```
# A deconvolution layer
convt = nn.ConvTranspose2d(in_channels = 8, \
                           out_channels = 8, \
                           kernel_size = 5)
z = convt(y)
print(z.shape)
```

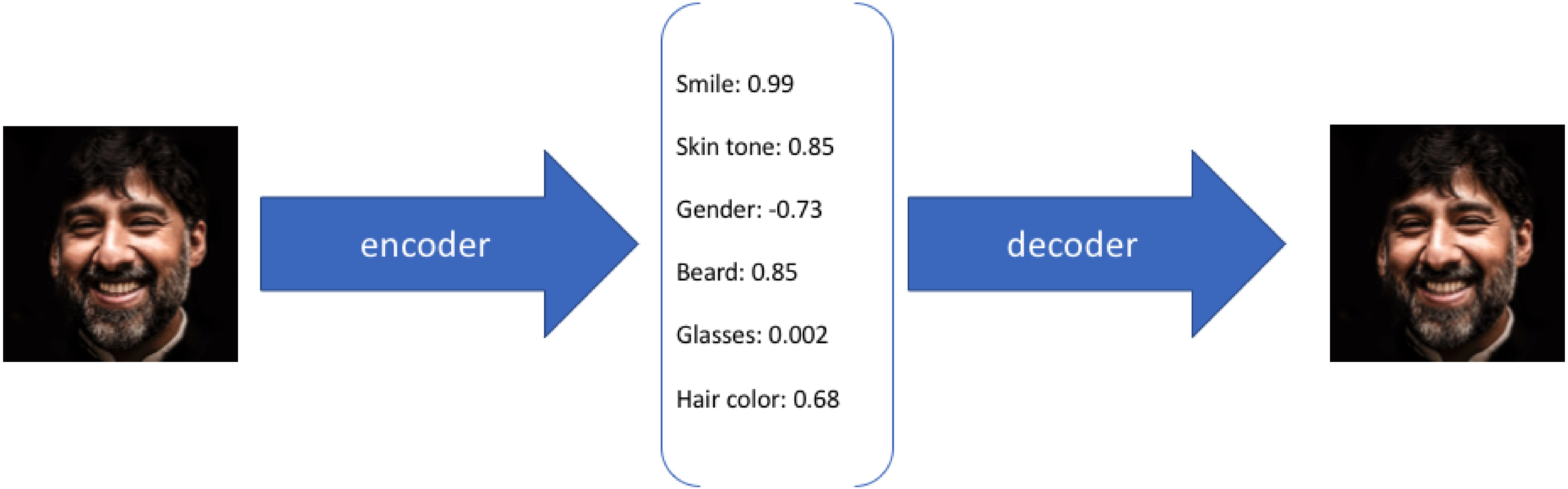
```
torch.Size([2, 8, 64, 64])
```

Limits of “vanilla” autoencoders

- Encoders in “Vanilla” autoencoders take an input and output a **fixed single value** for each encoding dimension of the latent representation.
- The decoder network then subsequently takes these values and attempts to recreate the original input.

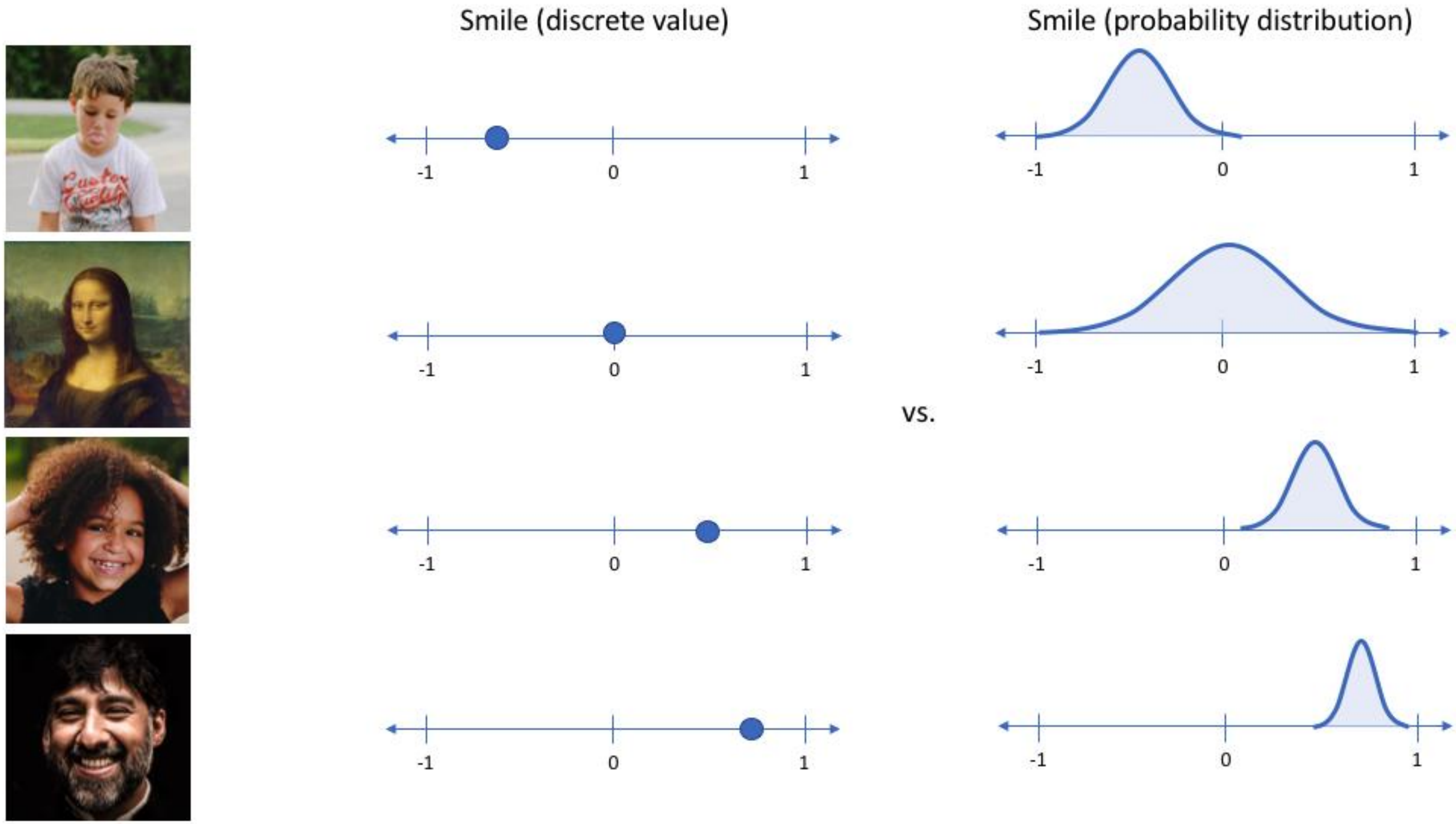
- A variational autoencoder (VAE) provides a **probabilistic** manner for describing an observation in latent space.
- Thus, rather than building an encoder which outputs a single value to describe each latent state attribute, we will formulate our encoder to describe a **probability distribution for each latent attribute**.

Limits of “vanilla” autoencoders

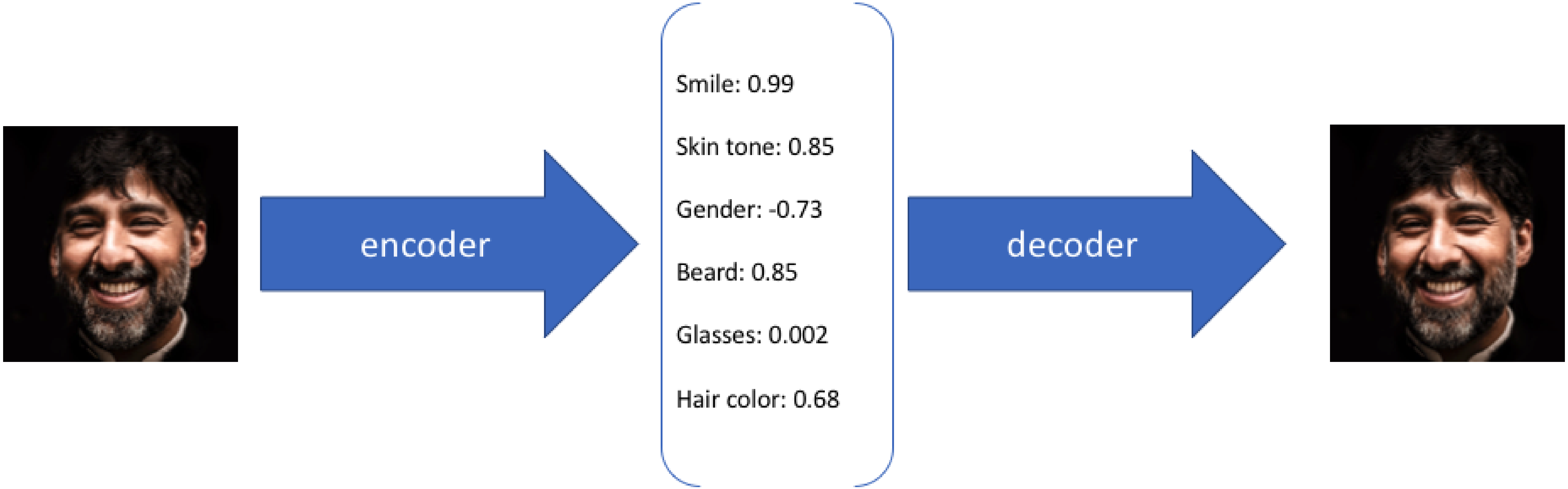


Latent attributes

Limits of “vanilla” autoencoders

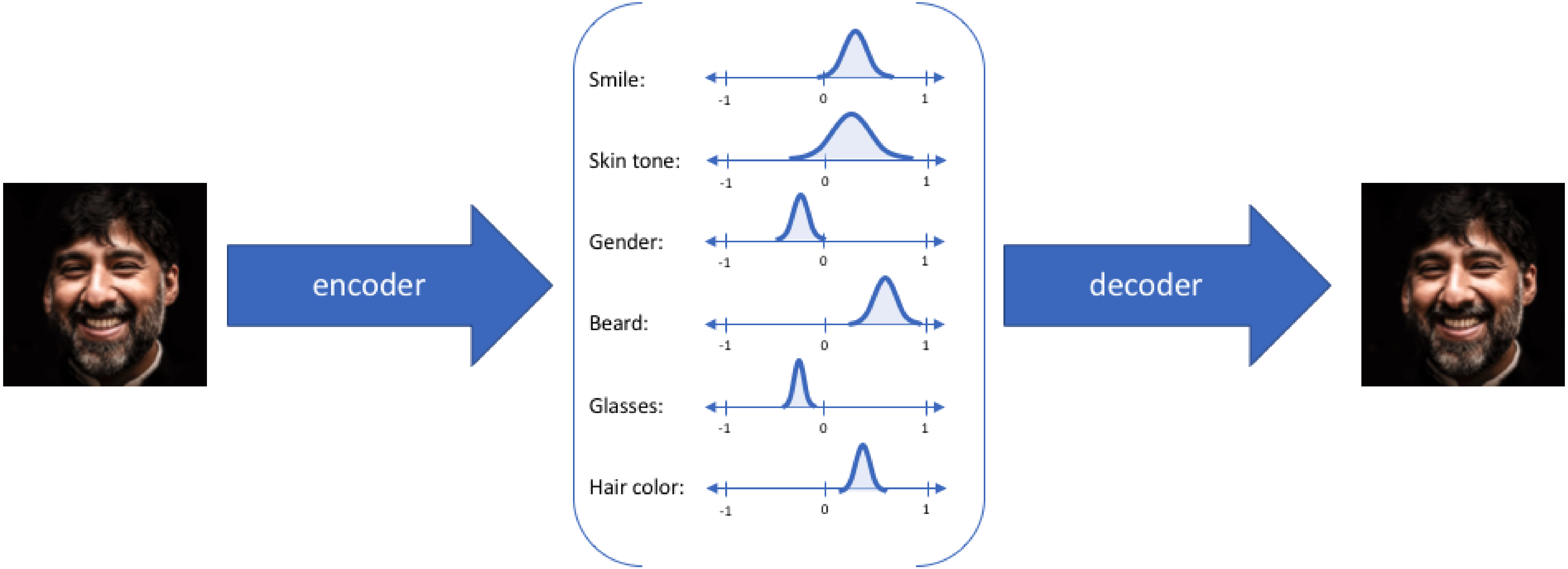


Limits of “vanilla” autoencoders



Latent attributes

Limits of “vanilla” autoencoders



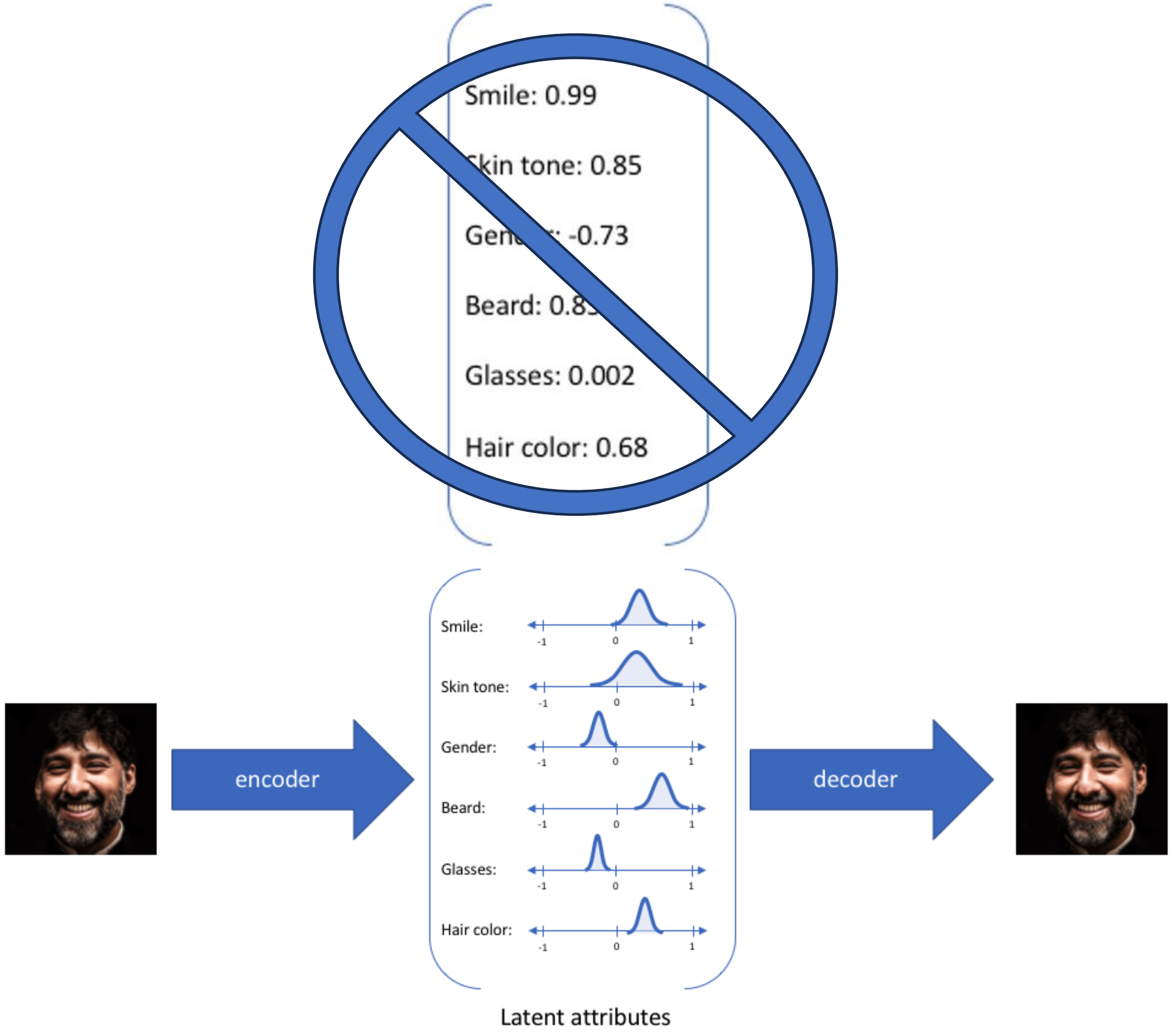
Latent attributes

A twist on the “vanilla” autoencoder

Definition (Variational Autoencoder):

A **variational autoencoder** fulfils the same job as a standard autoencoder but attempts to learn

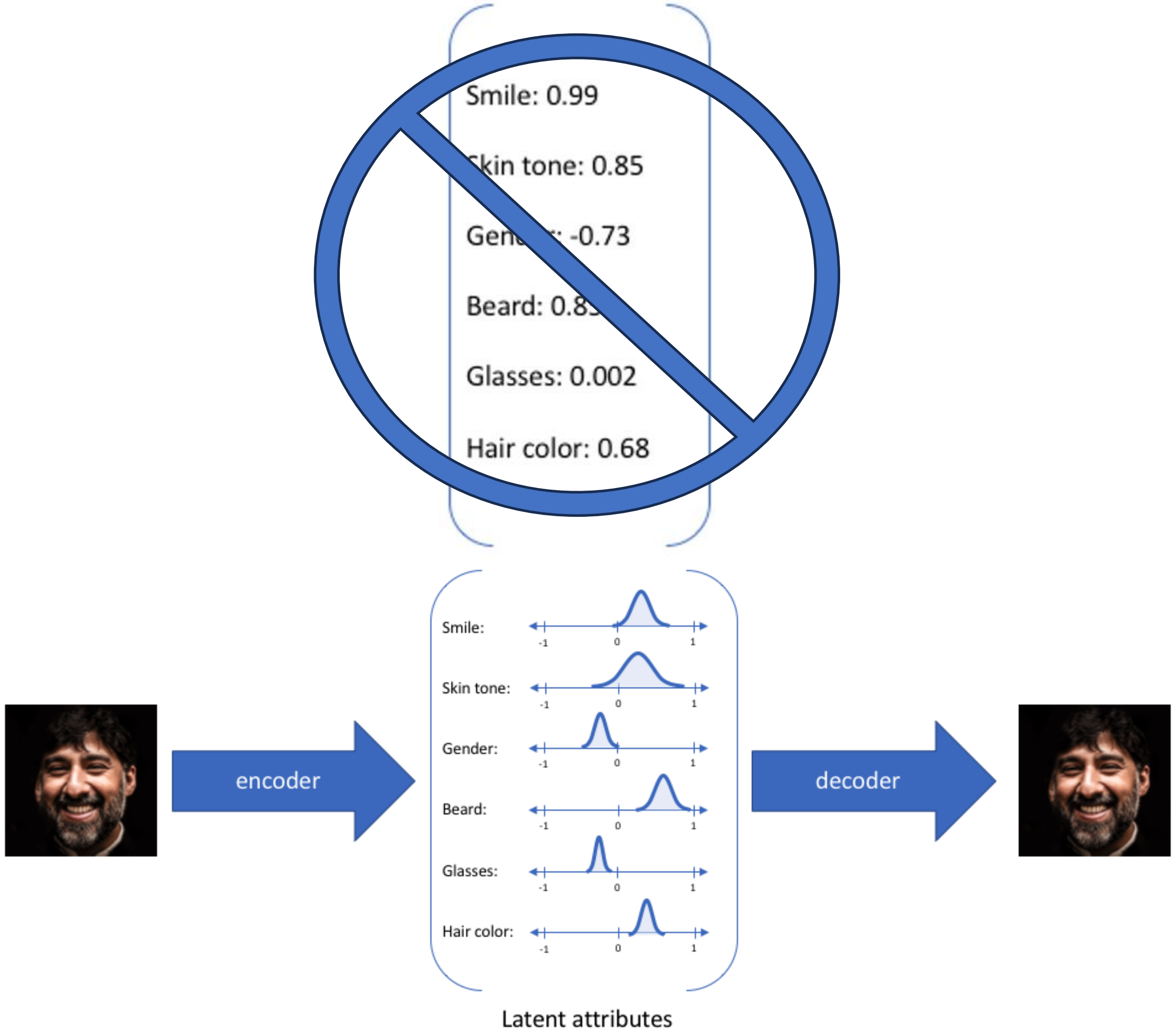
- how to represent inputs into,
- and how to reconstruct outputs from a **probabilistic latent space/representation instead of a deterministic one**.



A twist on the "vanilla" autoencoder

Just like before with vanilla autoencoders, a variational autoencoder consists of four parts.

1. **Encoder**
2. **Probabilistic latent Representation in bottleneck layer**
3. **Decoder**
4. **Reconstruction loss**

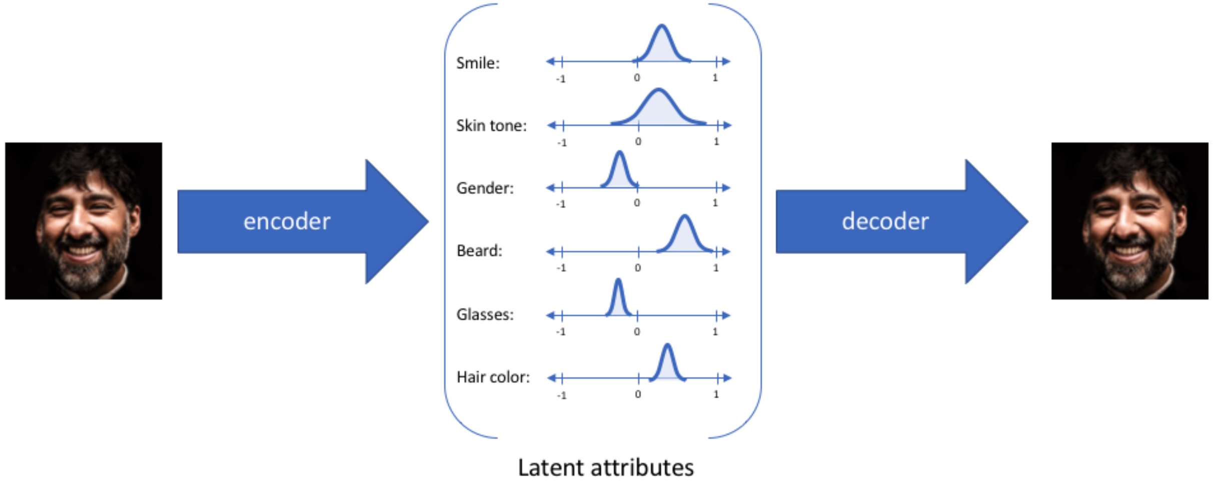


A twist on the “vanilla” autoencoder

1. **Encoder (as before)**
3. **Decoder (as before)**

Note: for VAEs, the encoder model is sometimes referred to as the **recognition model**,

Whereas the decoder model is sometimes referred to as the **generative model**.

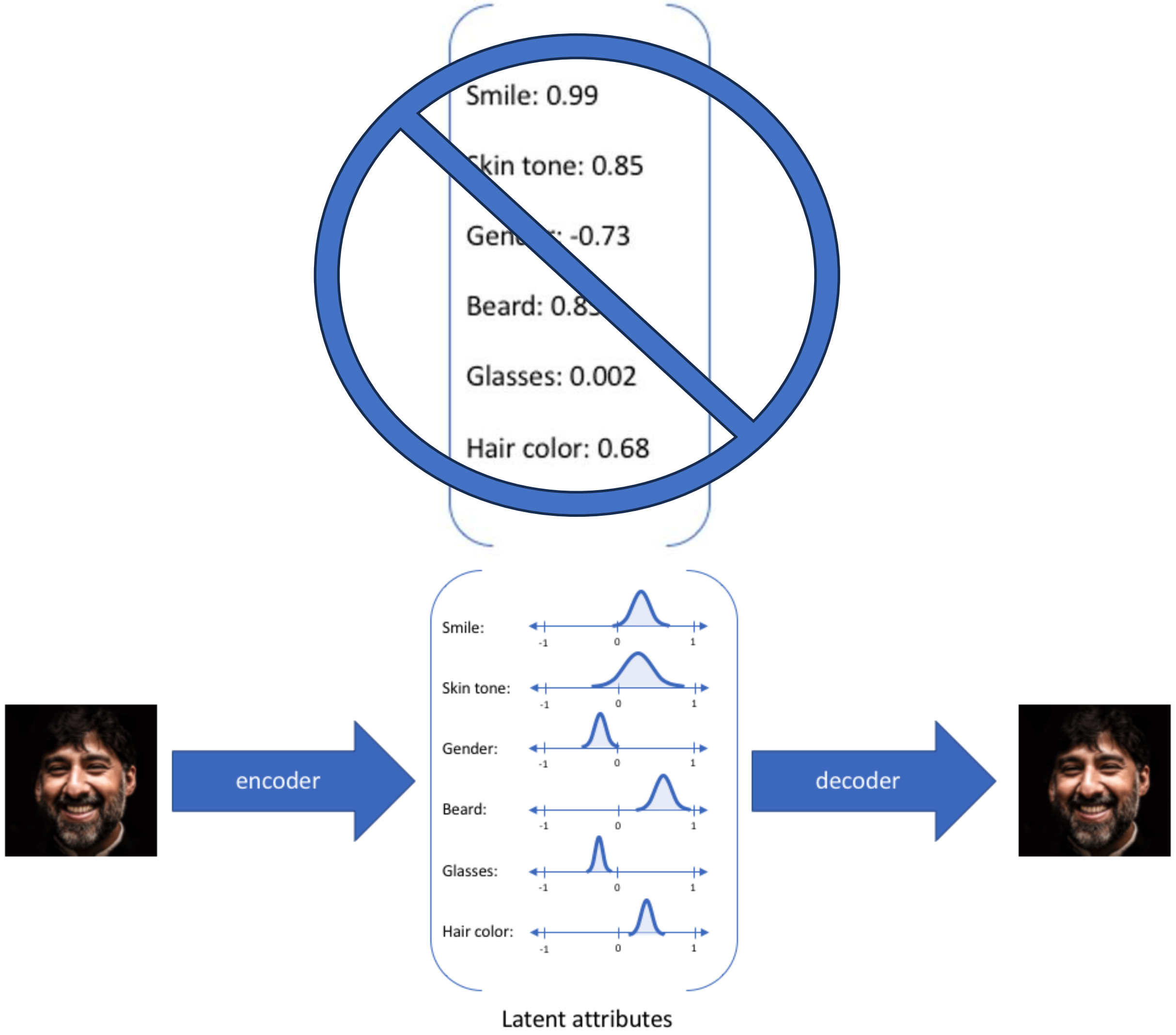


A twist on the “vanilla” autoencoder

2. **Probabilistic latent representation in bottleneck layer**

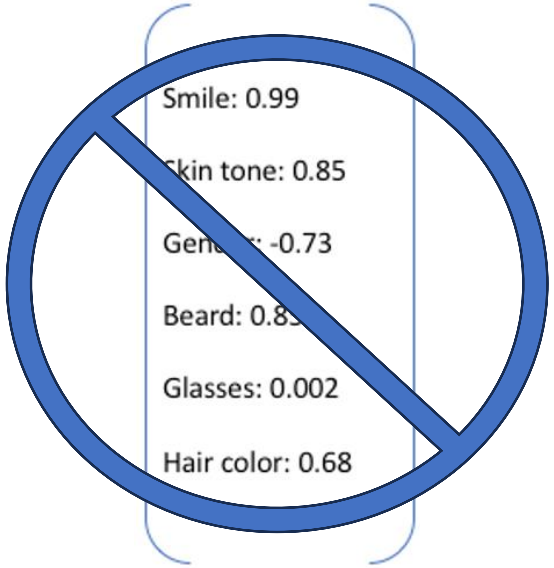
There is a slight change in the way we formulate our latent representation of the features.

Each value of the feature vector is assumed to follow from a normal distribution represented with its own set of (mean μ, std σ).



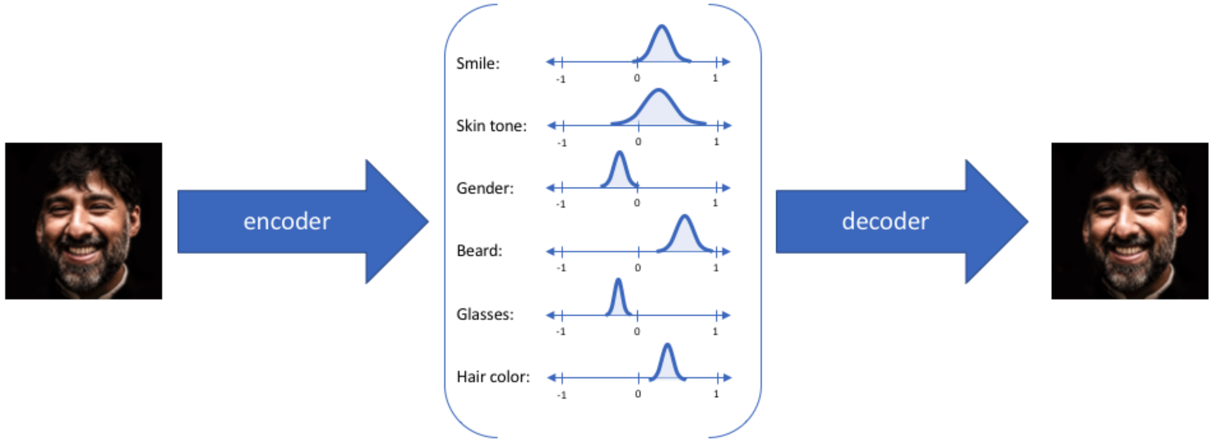
A twist on the “vanilla” autoencoder

2. **Probabilistic latent representation in bottleneck layer**

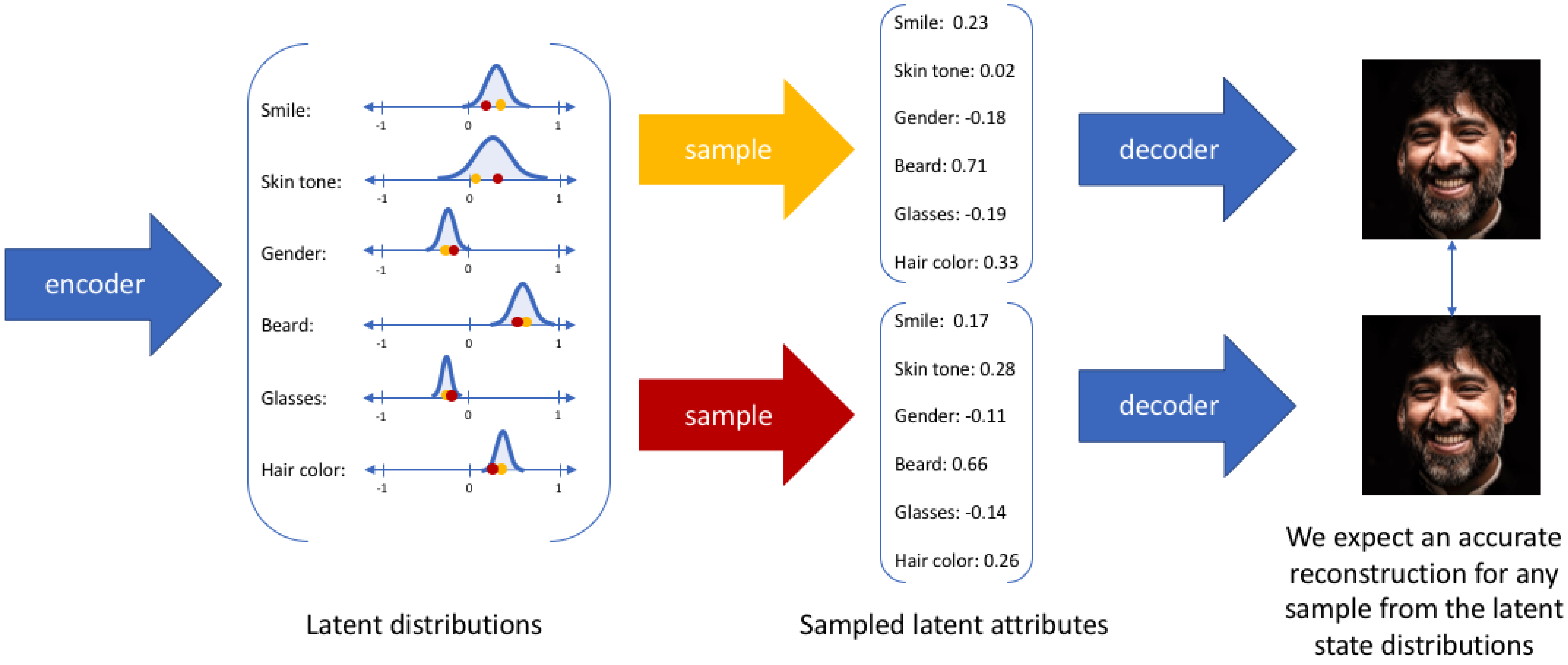


Right before the decoder phase, we then **sample a latent vector from the distributions** we have identified.

This vector is then used for reconstruction in the decoder.

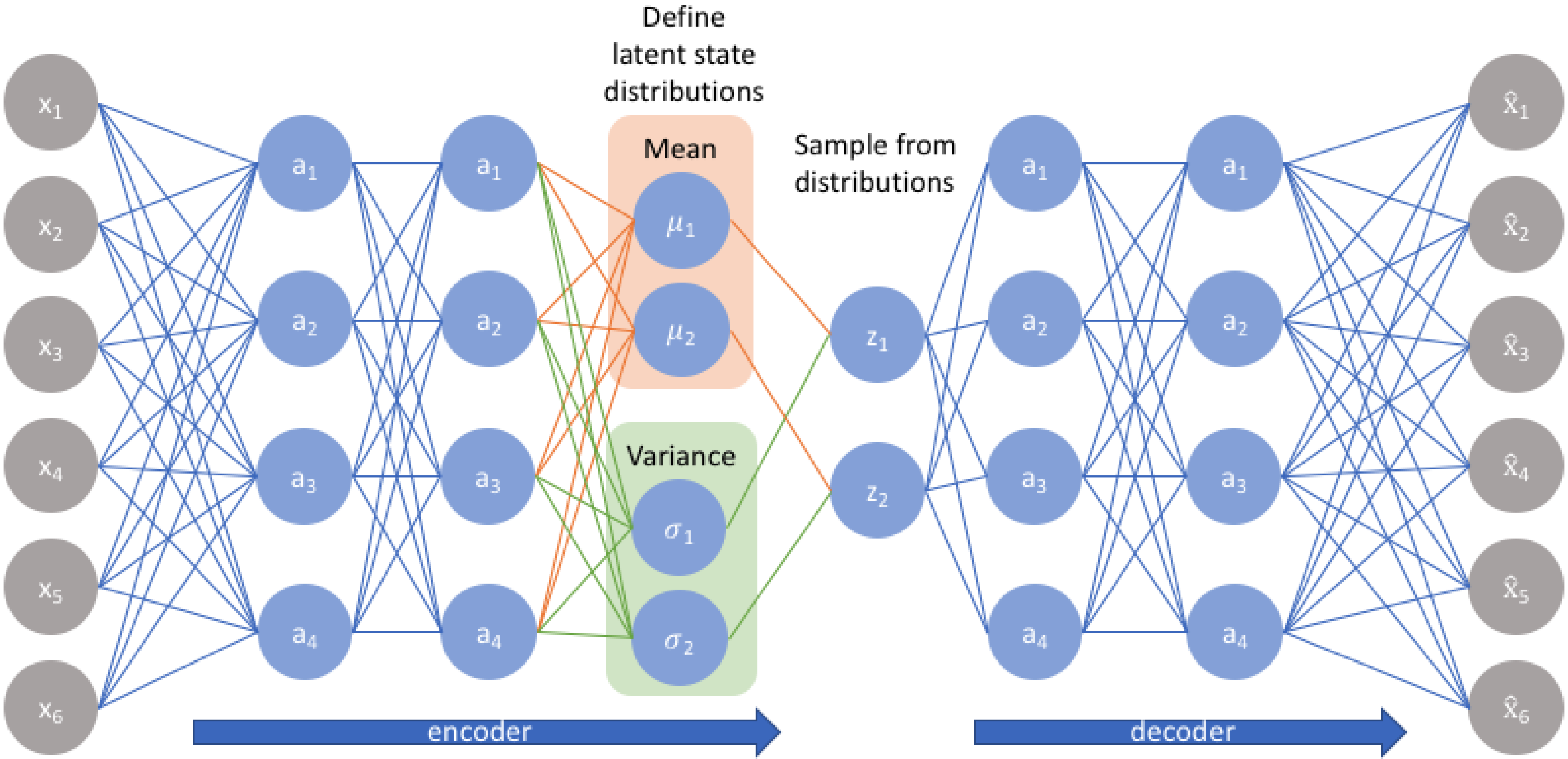


Latent attributes



Important note: this means that two forward passes of the same image could have different latent vectors and reconstructed images.

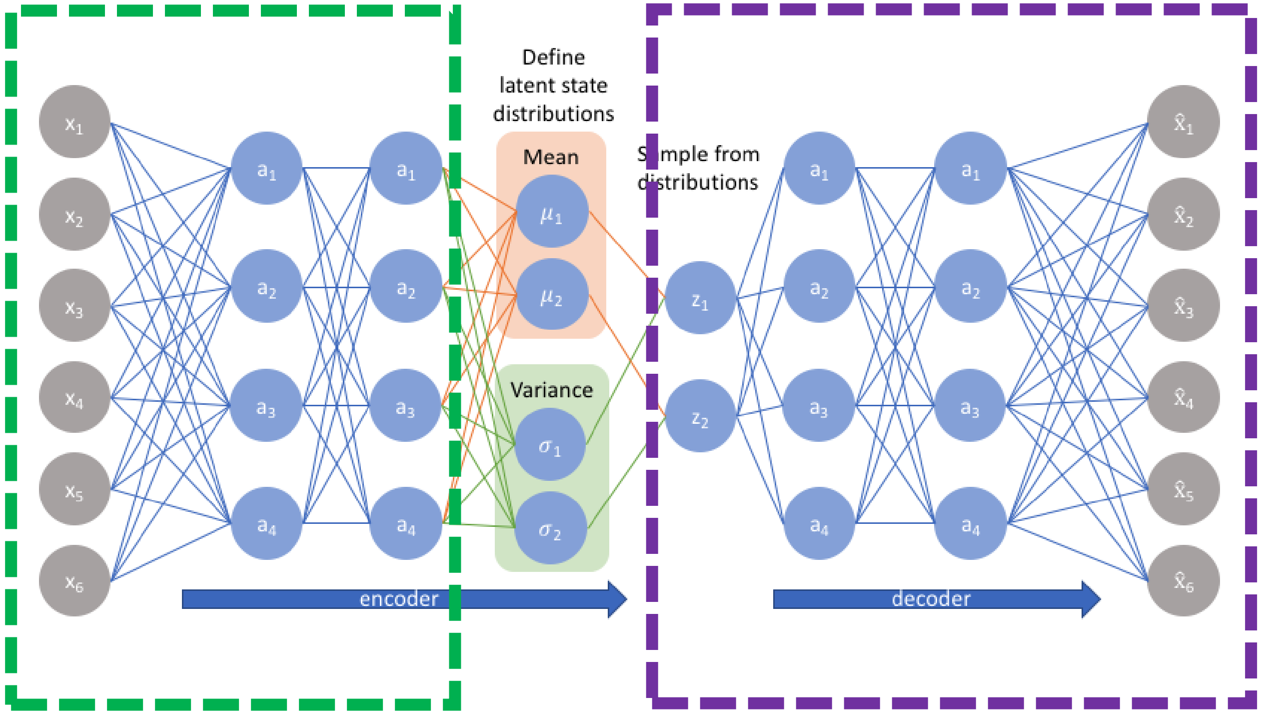
A twist on the “vanilla” autoencoder



A twist on the “vanilla” autoencoder

Just like before with vanilla autoencoders, a variational autoencoder consists of four parts.

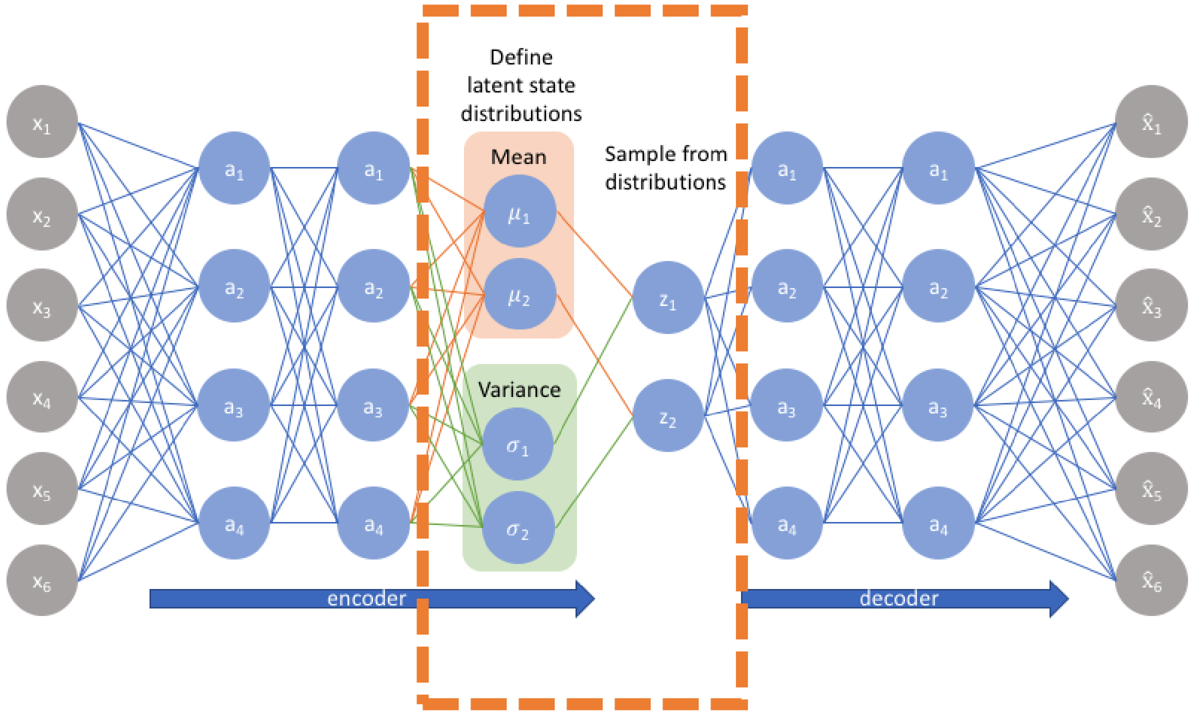
1. **Encoder (as before)**
2. **Probabilistic latent Representation in bottleneck layer**
3. **Decoder (as before)**
4. **Reconstruction loss**



A twist on the "vanilla" autoencoder

Just like before with vanilla autoencoders, a variational autoencoder consists of four parts.

1. **Encoder (as before)**
2. **Probabilistic latent representation in bottleneck layer**
3. **Decoder (as before)**
4. **Reconstruction loss**



A twist on the “vanilla” autoencoder

Why is that better?

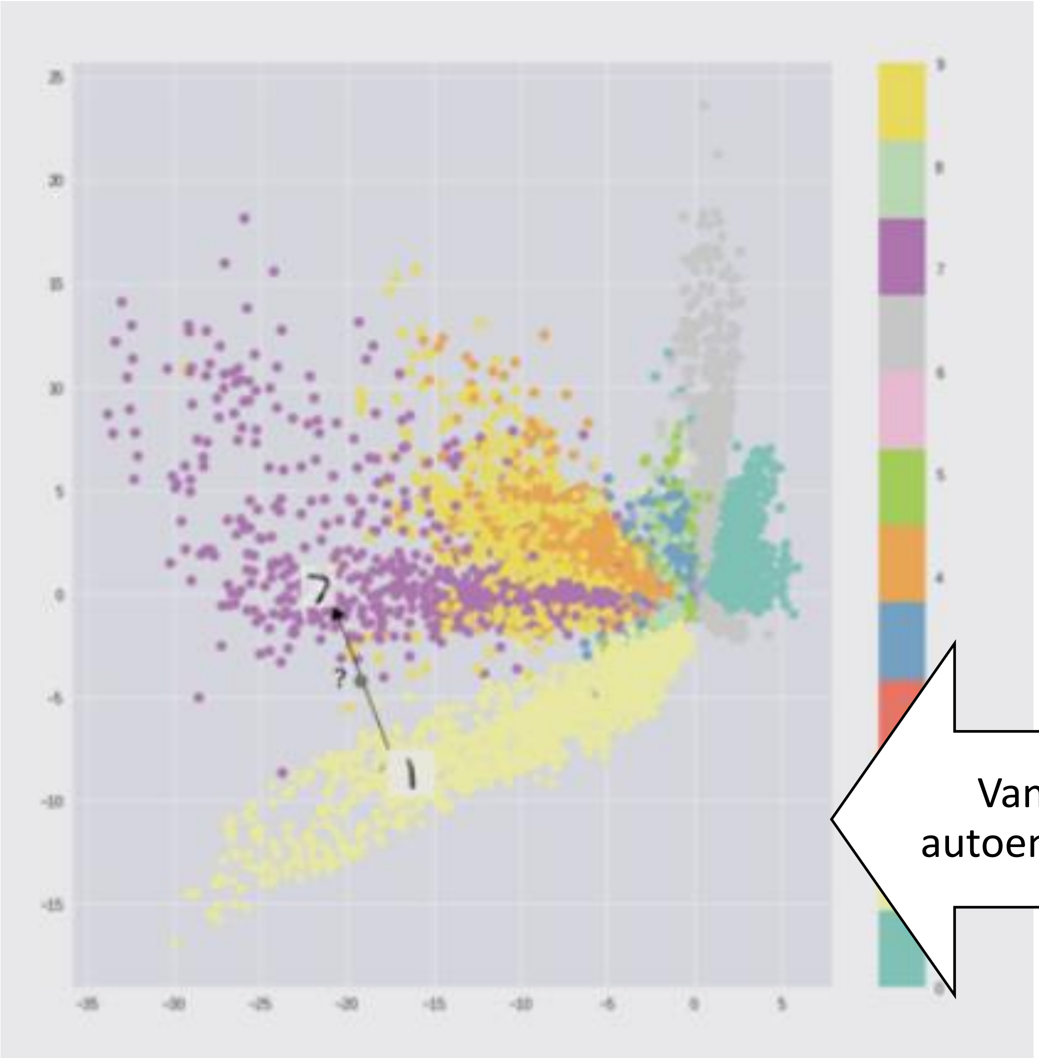
- By constructing our encoder model to output a range of possible values (a statistical distribution) from which we will randomly sample to feed into our decoder model, we are essentially **enforcing a continuous, smooth latent space representation**.

A twist on the “vanilla” autoencoder

This will have two effects

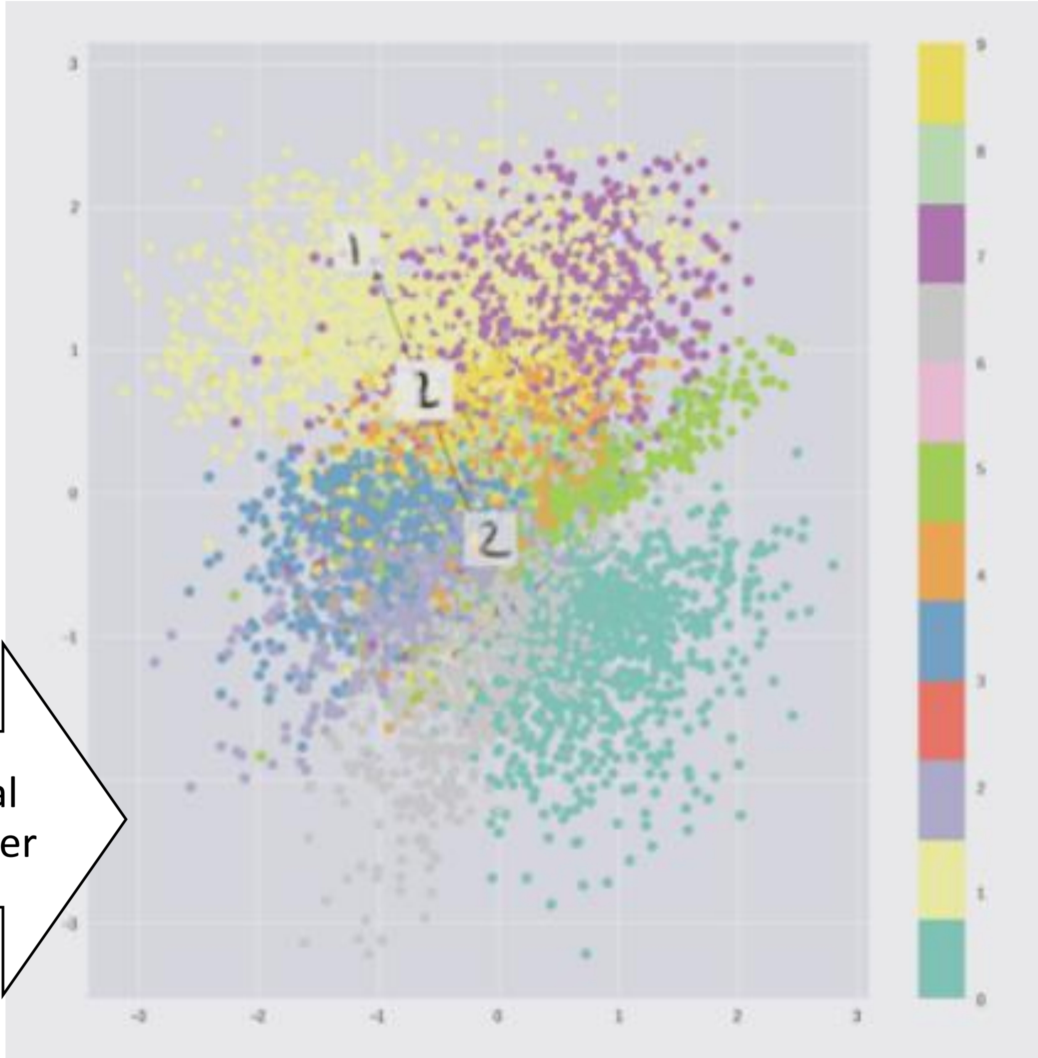
- The decoder will have to **work harder** to reconstruct images as the latent vector could be different for two forward passes (SkipGram vs. CBoW effect?)
- The space for latent vectors will be used **more uniformly** (which could mean less compression on the encoder side?)

A twist on the “vanilla” autoencoder



Vanilla autoencoder

Variational autoencoder



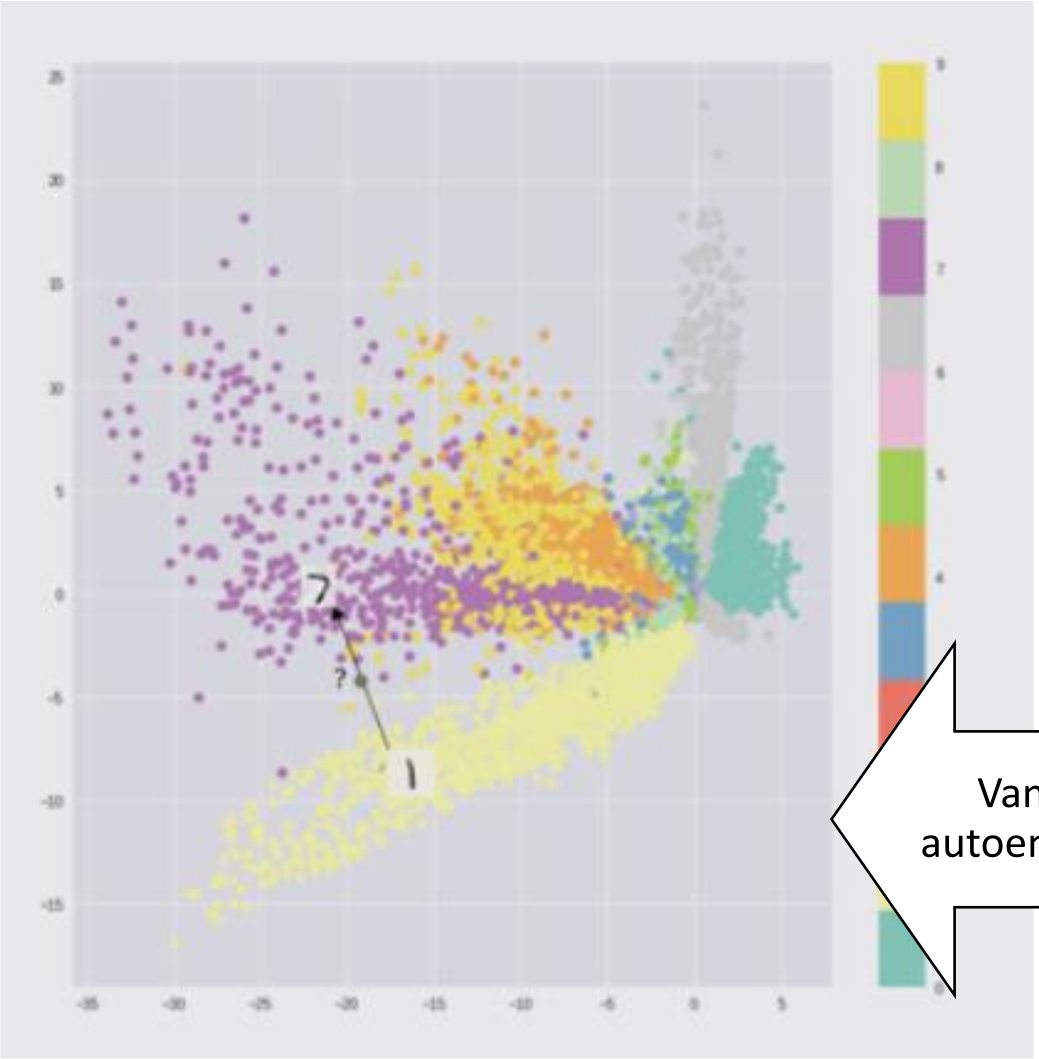
A twist on the “vanilla” autoencoder

Another effect is that

- For any sampling of the latent distributions, we are expecting our decoder model to be able to accurately reconstruct an image.
- Thus, values which are nearby to one another in latent space should correspond with very similar reconstructions.

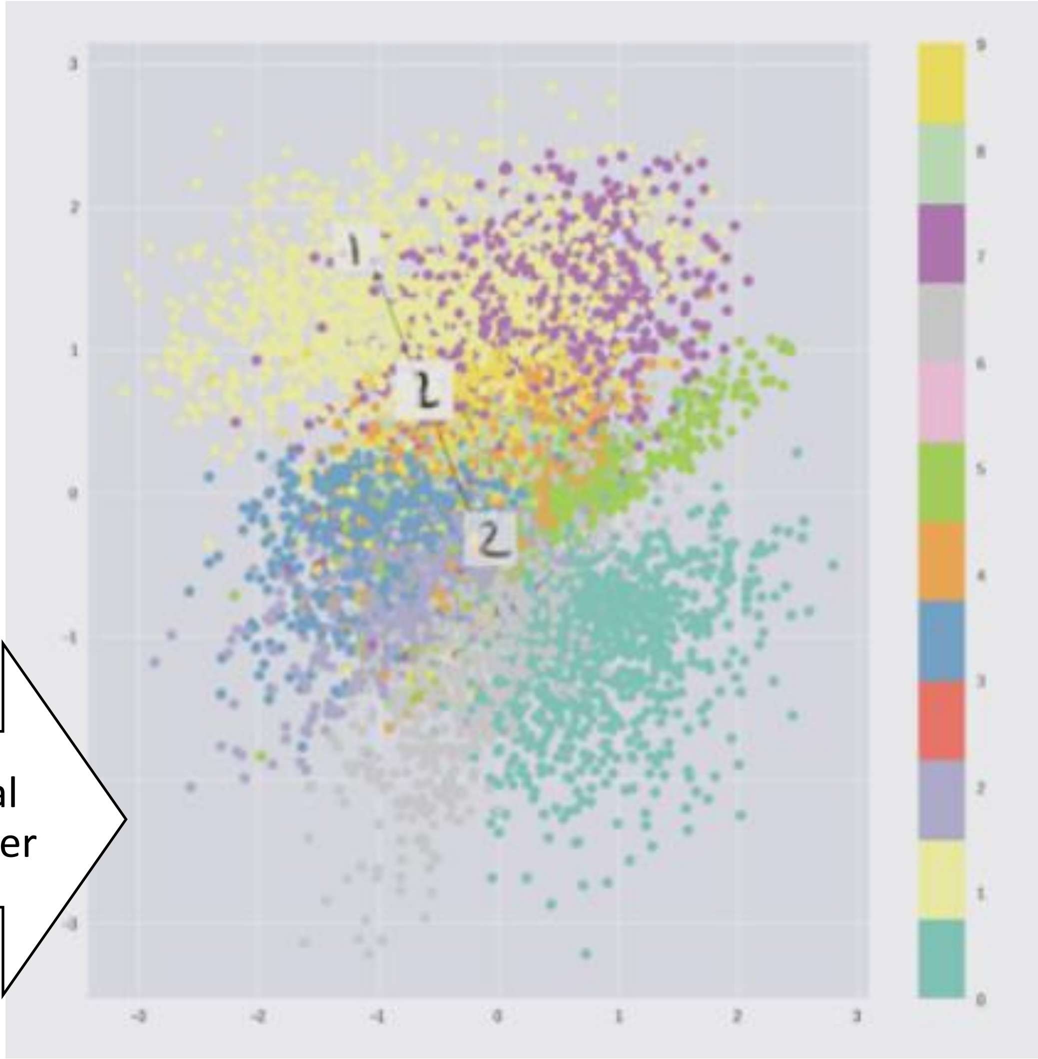
Restricted

A twist on the “vanilla” autoencoder



Vanilla autoencoder

Variational autoencoder



A twist on the “vanilla” autoencoder

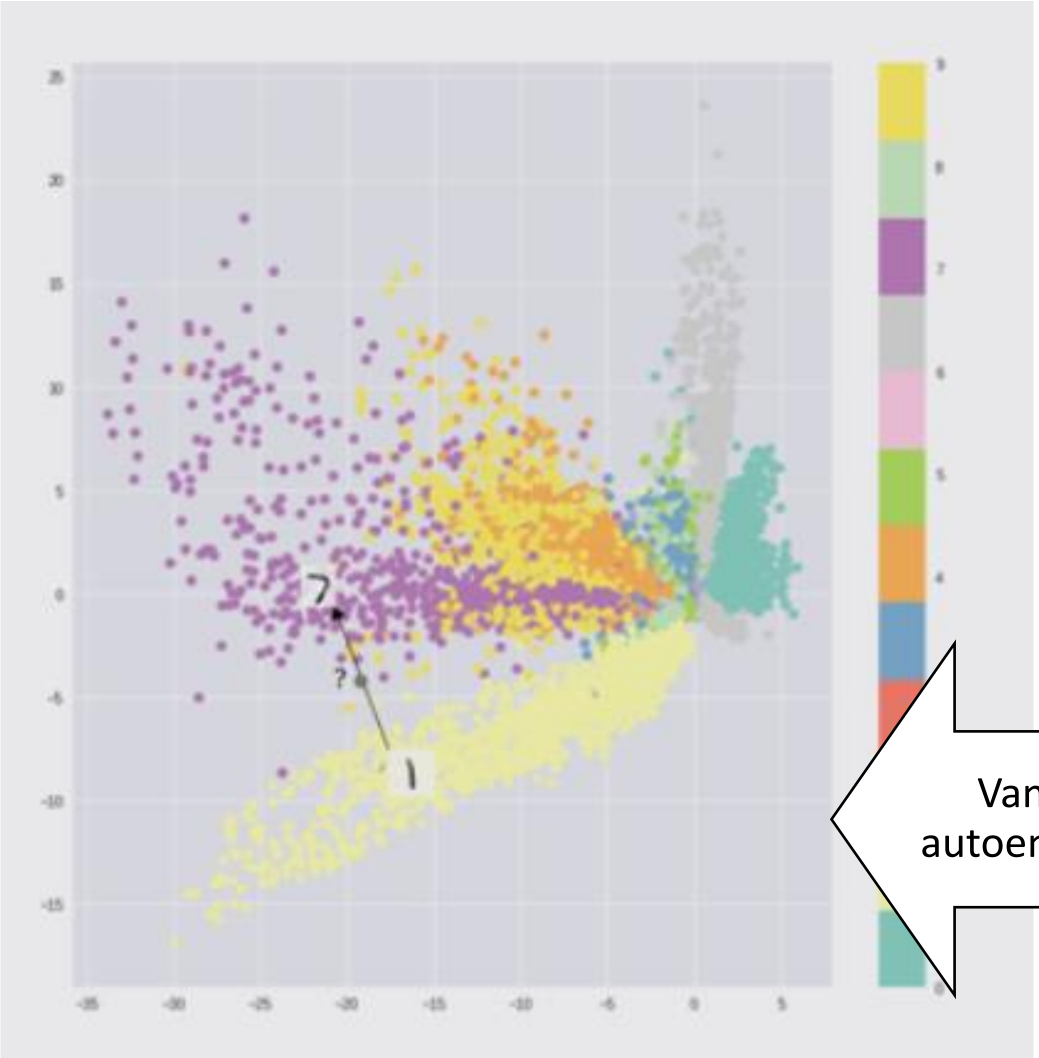
More importantly,

- This also means that any random noise vector can be used used as an input for the decoder,
- And this random encoding vector given to the decoder has high chances of producing an output which looks plausible!

In other words,

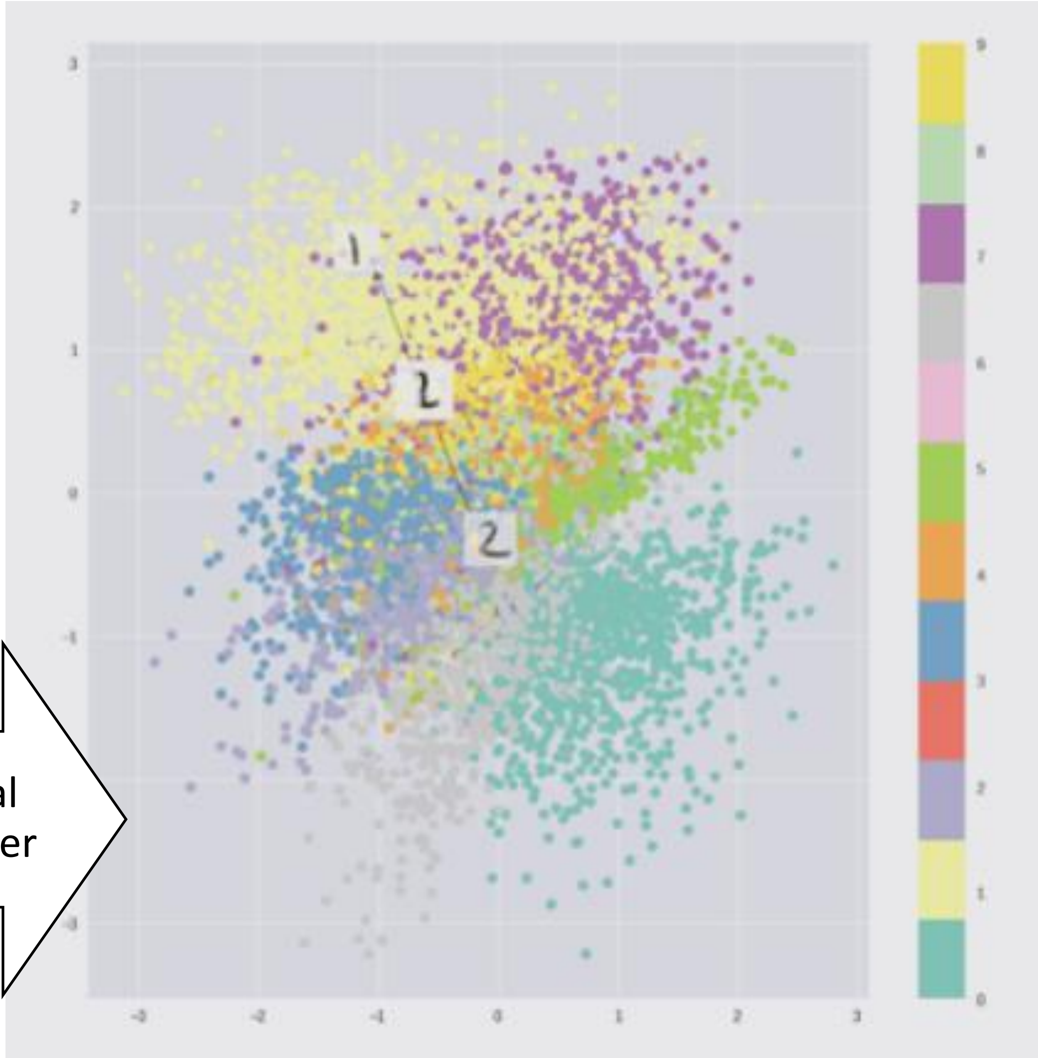
- **We could therefore get a decoder Neural Network, which could generate plausible images out of <u>any noise vector</u>,**
- **Or in other words, a model that can generate plausible images out of thin air!**

A twist on the “vanilla” autoencoder



Vanilla autoencoder

Variational autoencoder



The Kullback–Leibler Divergence

Definition (KL divergence):

The **Kullback–Leibler divergence** (or **KL divergence**, in short) is a measure of how one probability distribution q is different from a second, reference probability distribution p.

$$D_{KL}(p||q) = \sum_x p(x) \log\left(\frac{p(x)}{q(x)}\right)$$

- Simple, but not symmetrical!

$$D_{KL}(p||q) \neq D_{KL}(q||p)$$

- **Finding a good distribution q that matches p, then requires to minimize the KL divergence.**

The Jensen-Shannon Divergence

Definition (extra - JS divergence):

The **Jensen–Shannon divergence** (or **JS divergence**, in short) is another measure of how one probability distribution q is different from a second, reference probability distribution p.

It reuses the KL divergence formula and simply averages it.

$$D_{KL}(p||q) = \sum_x p(x) \log\left(\frac{p(x)}{q(x)}\right)$$

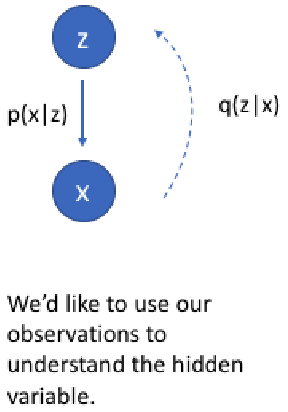
$$D_{KL}(p||q) \neq D_{KL}(q||p)$$

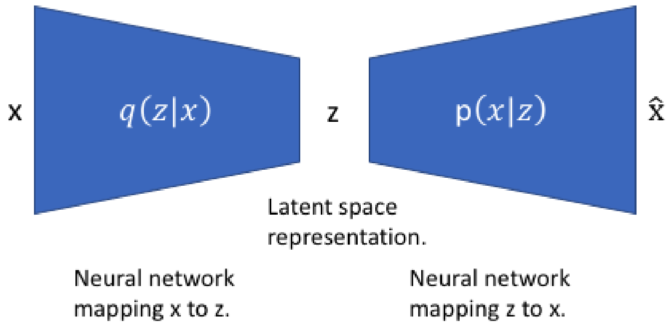
$$\boldsymbol{D_{JS}(p||q) = \frac{D_{KL}(p||q) + D_{KL}(q||p)}{2}}$$

- Better because D_{JS} is symmetrical! (by definition)

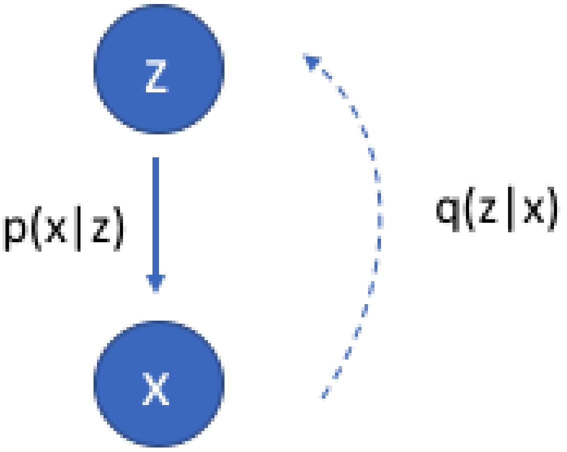
Back to our VAE

- To revisit our graphical model, we can use q to infer the possible hidden variables (i.e. our latent representation or bottleneck vector z of size D)
- Later, this vector z will be used to generate an observation $\hat{x}$.

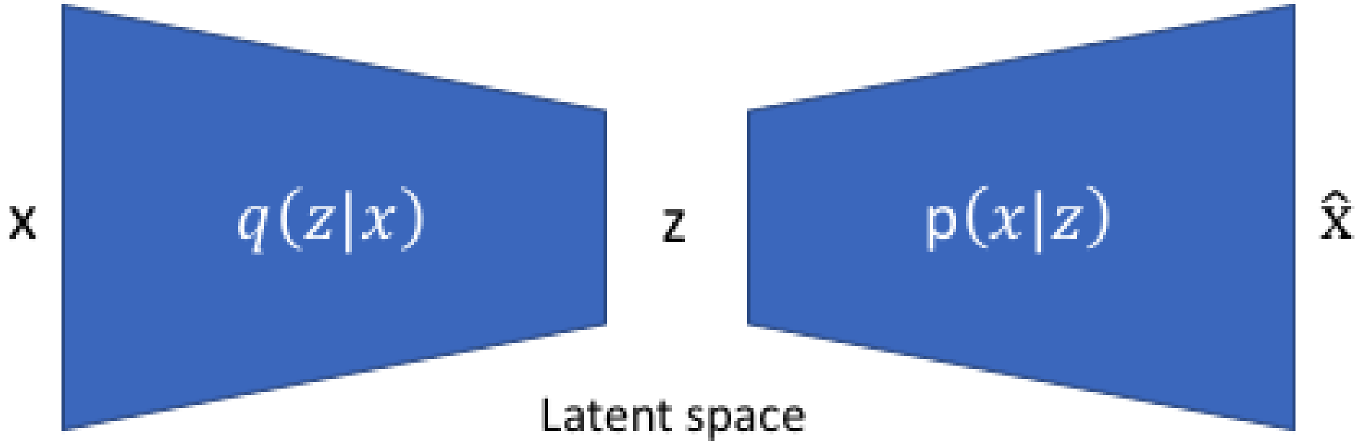




Back to our VAE



We'd like to use our observations to understand the hidden variable.



Latent space representation.

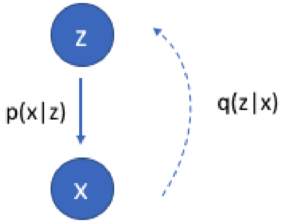
Neural network mapping x to z.

Neural network mapping z to x.

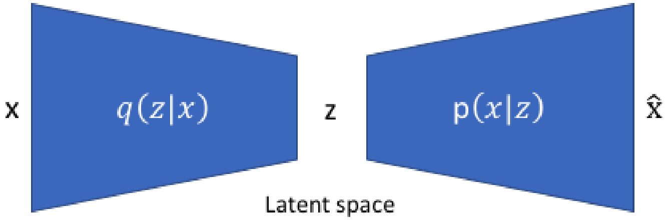
Back to our VAE

We can further construct this model into a neural network architecture, where:

- the encoder model learns a mapping from x to z, that is $\boldsymbol{q(z|x)}$.
- and the decoder model learns a mapping from z back to x, that is $\boldsymbol{p(x|z)}$.



We'd like to use our observations to understand the hidden variable.



Latent space representation.

Neural network mapping x to z.

Neural network mapping z to x.

Back to our VAE

4. Reconstruction loss

Our loss function for this network will now consist of two terms:

- one which penalizes **reconstruction error** (which can be thought of maximizing the reconstruction likelihood as discussed earlier)
- and a second term which **encourages our learned distribution $\boldsymbol{q(z|x)}$** to be similar to a target distribution $\boldsymbol{p(z)}$.

$$L(x, \hat{x}) + \sum_j D_{KL}(q_j(z|x)||p(z))$$

Restricted

Back to our VAE

4. Reconstruction loss

Our loss function for this network will now consist of two terms:

- one which penalizes **reconstruction error** (which can be thought of maximizing the reconstruction likelihood as discussed earlier)
- and a second term which **encourages our learned distribution $\boldsymbol{q(z|x)}$** to be similar to a target distribution $\boldsymbol{p(z)}$.
- For instance, let us assume that $\boldsymbol{p(z)}$ follows a **unit Gaussian distribution** (could be any other nice distribution really), for each dimension j of the latent space.

$$L(x, \hat{x}) + \alpha \sum_j D_{KL}(q_j(z|x)||N(0,1))$$

Back to our VAE

4. Reconstruction loss

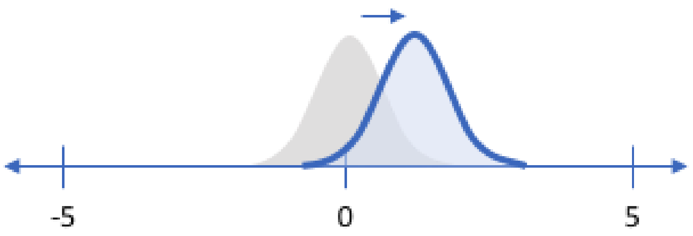
Our loss function for this network will now consist of two terms:

- one which penalizes **reconstruction error** (which can be thought of maximizing the reconstruction likelihood as discussed earlier)
- and a second term which **encourages our learned distribution $\boldsymbol{q(z|x)}$** to be similar to a target distribution $\boldsymbol{p(z)}$.
- For instance, let us assume that $\boldsymbol{p(z)}$ follows a **unit Gaussian distribution** (could be any other nice distribution really), for each dimension j of the latent space.

$$MSE(x,\hat{x}) + \alpha \sum_j D_{KL}(q_j(z|x)||N(0,1))$$

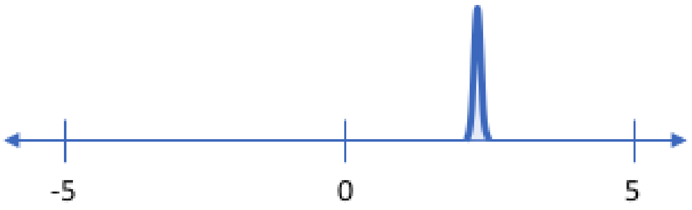
Back to our VAE

Penalizing reconstruction loss encourages the distribution to describe the input



Our distribution deviates from the prior to describe some characteristic of the data

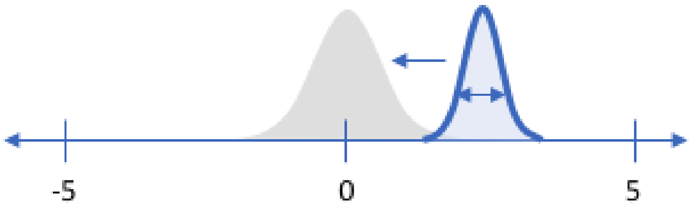
Without regularization, our network can “cheat” by learning narrow distributions



With a small enough variance, this distribution is effectively only representing a single value

Penalizing KL divergence acts as a regularizing force

Attract distribution to have zero mean



Ensure sufficient variance to yield a smooth latent space

$$MSE(x,\hat{x}) + \alpha \sum_{j} D_{KL}(q_j(z|x)||N(0,1))$$

Implementation of a VAE

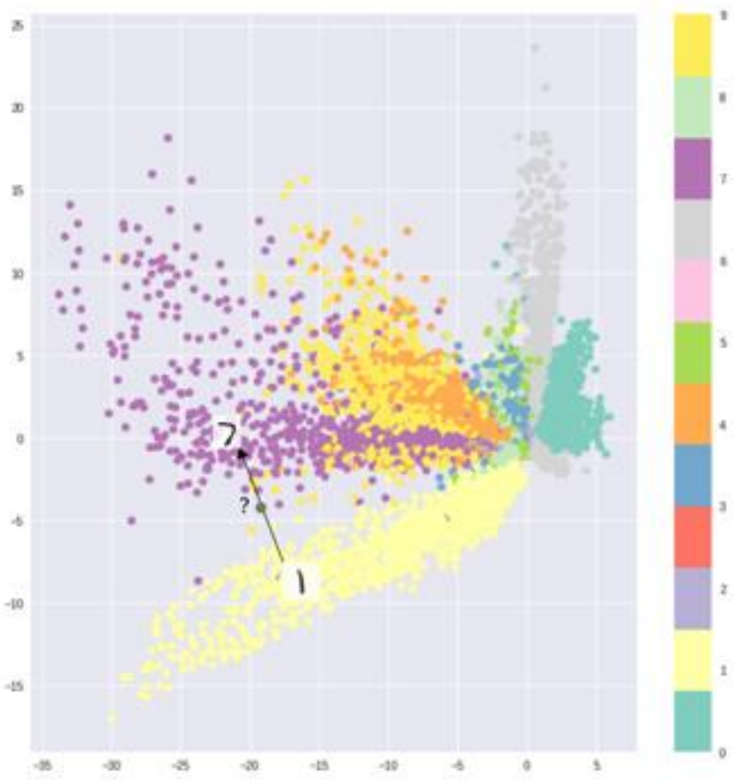
The implementation of a VAE is out of the scope of this class.

The important notions we want to learn are

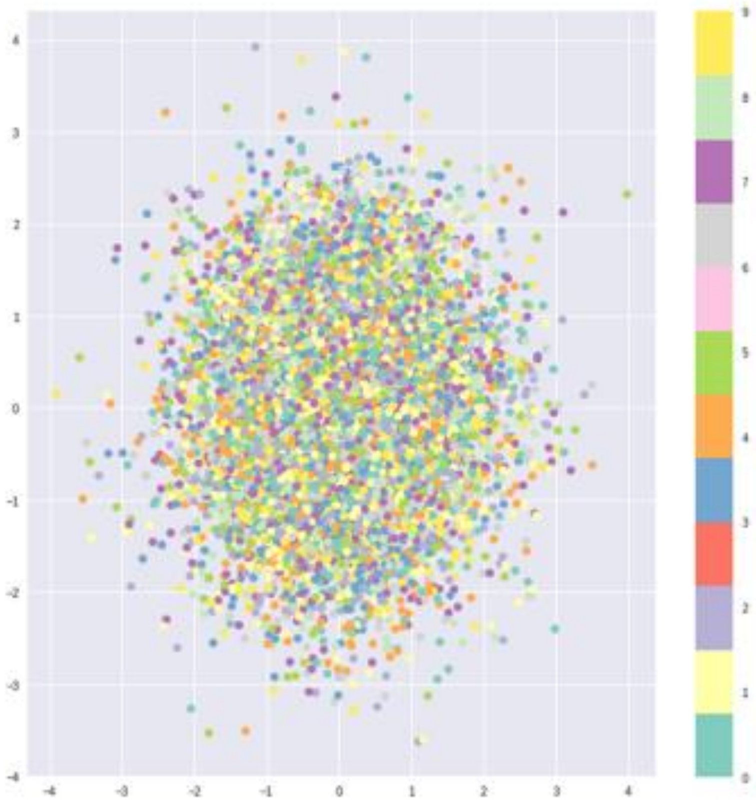
- Using **probabilistic latent representations**, instead of fixed ones, **distributes the embeddings or features vectors over the latent space in a more uniform way, occupying the entire space**.

Effect of KL term on latent space

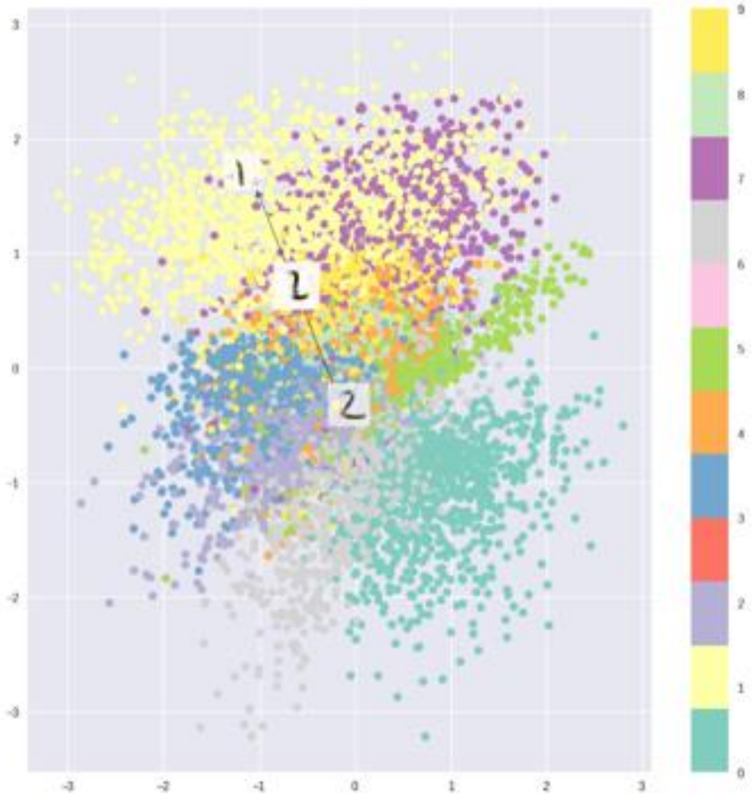
Only reconstruction loss



Only KL divergence



Combination



Implementation of a VAE

The training of a VAE is out of the scope of this class.

The important notions we want to learn are:

- Using **probabilistic latent representations**, instead of fixed ones, **distributes the embeddings or features vectors over the latent space in a more uniform way, occupying the entire space**.
- This is done by adding a simple KL term to our reconstruction loss, which acts as a **regularization** of some sort.
- **This also means that any randomly generated latent vector could technically produce a good-looking output via the decoder**.

```
# Define Variational AutoEncoder Model for MNIST
class MNIST_VAE(nn.Module):

    def __init__(self, image_channels, init_channels, kernel_size, latent_dim):
        super().__init__()

        # Encoder with stacked Conv
        self.enc1 = nn.Conv2d(image_channels, init_channels, kernel_size, \
                              stride = 2, padding = 1)
        self.enc2 = nn.Conv2d(init_channels, init_channels*2, kernel_size, \
                              stride = 2, padding = 1)
        self.enc3 = nn.Conv2d(init_channels*2, init_channels*4, kernel_size, \
                              stride = 2, padding = 1)
        self.enc4 = nn.Conv2d(init_channels*4, 64, kernel_size, \
                              stride = 2, padding = 0)

        # FC layers for learning representations
        self.fc1 = nn.Linear(64, 128)
        self.fc_mu = nn.Linear(128, latent_dim)
        self.fc_log_var = nn.Linear(128, latent_dim)
        self.fc2 = nn.Linear(latent_dim, 64)

        # Decoder, simply mirroring the encoder with ConvTranspose
        self.dec1 = nn.ConvTranspose2d(64, init_channels*8, kernel_size, \
                                       stride = 1, padding = 0)
        self.dec2 = nn.ConvTranspose2d(init_channels*8, init_channels*4, kernel_size, \
                                       stride = 2, padding = 1)
        self.dec3 = nn.ConvTranspose2d(init_channels*4, init_channels*2, kernel_size, \
                                       stride = 2, padding = 1)
        self.dec4 = nn.ConvTranspose2d(init_channels*2, image_channels, kernel_size, \
                                       stride = 2, padding = 1)
```

```python
    def sample(self, mu, log_var):
        """
        mu: mean from the encoder's latent space
        log_var: log variance from the encoder's latent space
        """

        # Standard deviation
        std = torch.exp(0.5*log_var)

        # randn_like is used to produce a vector with same dimensionality as std
        eps = torch.randn_like(std)

        # Sampling
        sample = mu + (eps * std)
        return sample

```

```python
    def forward(self, x):

        # Encoder
        x = F.relu(self.enc1(x))
        x = F.relu(self.enc2(x))
        x = F.relu(self.enc3(x))
        x = F.relu(self.enc4(x))

        # Pooling
        batch, _, _, _ = x.shape
        x = F.adaptive_avg_pool2d(x, 1).reshape(batch, -1)

        # FC layers to get mu and log_var
        hidden = self.fc1(x)
        mu = self.fc_mu(hidden)
        log_var = self.fc_log_var(hidden)

        # Get the latent vector through reparameterization
        z = self.sample(mu, log_var)
        z = self.fc2(z)
        z = z.view(-1, 64, 1, 1)

        # Decoding
        x = F.relu(self.dec1(z))
        x = F.relu(self.dec2(x))
        x = F.relu(self.dec3(x))
        x = torch.sigmoid(self.dec4(x))
        return x, mu, log_var
```

Implementation of a VAE

The training of a VAE is out of the scope of this class.

The important notions we want to learn are

- **This also forces the VAE to work harder to figure out a good embedding (like Skipgram vs. CBoW), which in turn will be better?**
- Another fun thing is that VAEs produce embeddings which could preserve similarity as the word embeddings we have seen on W9!

$$w_{queen} = w_{king} - w_{man} + w_{woman}$$

- With images VAE, the vectors will for instance allow for this type of shenaningans!

$$w_{man\ with\ glasses} = w_{man} + w_{glasses}?$$

From VAE to GANs

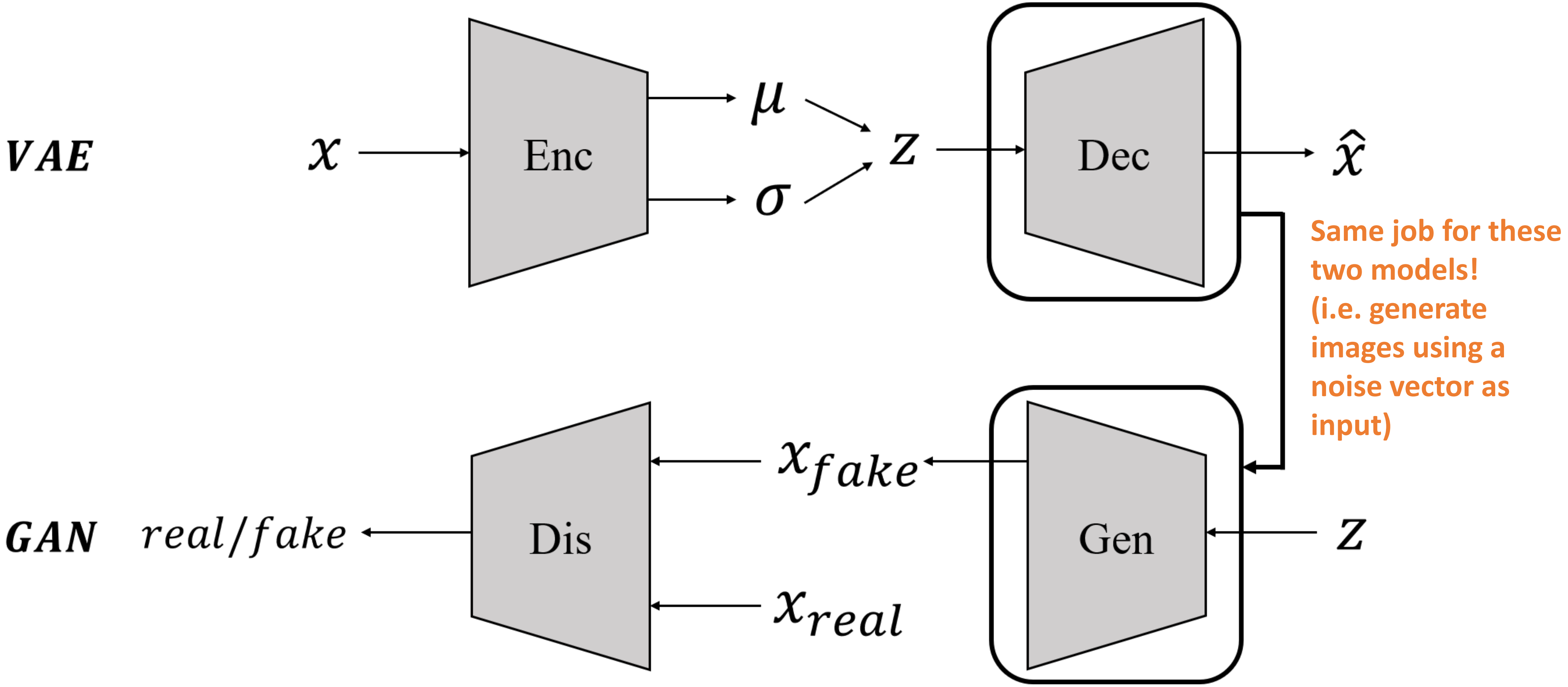
- **Our intuition:** train an encoder + decoder type of architecture on a meaningless task to learn a good embedding for images.
- **After training:** throw away the decoder and keep the encoder for good image embeddings and reuse on some other applications.

→ But what if we kept the decoder for a change?

- We would then have a generator type of object, which only requires a noise vector sample (any vector really) as input…
- And generates good looking outputs as a result!

→ This is the core idea behind Generative Adversarial Networks!

From VAE to GANs



GAN definition

Definition (GAN):

A **Generative Adversarial Network** (or **GAN**, in short) is a particular type of architecture, which attempts to learn to reproduce the samples distribution in a given dataset X.

- Train a model G, with noise sample inputs $z \in \mathbb{R}^K$ drawn from normalized Gaussian distributions.

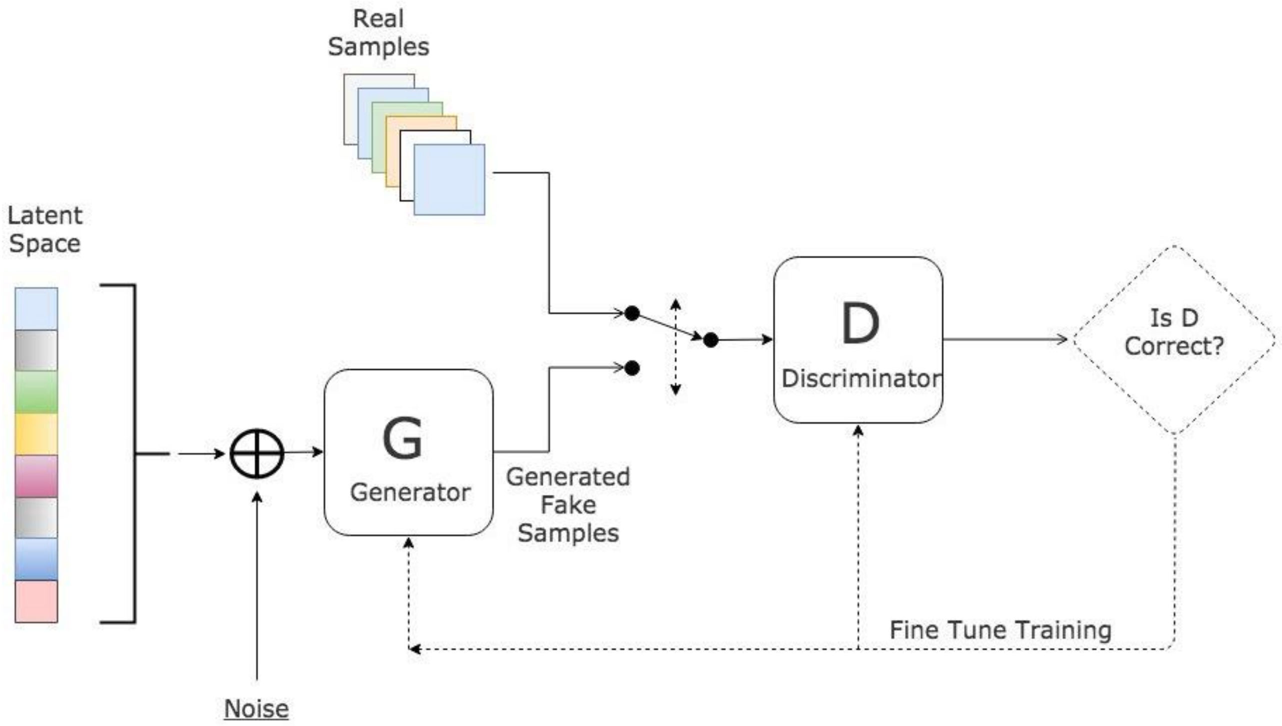
$$\hat{x} = G(z), \qquad z \to N(0_K, I_K)$$

The training objective for G, is then to ensure that the distribution of $\hat{X}$ matches the one of our original dataset X.

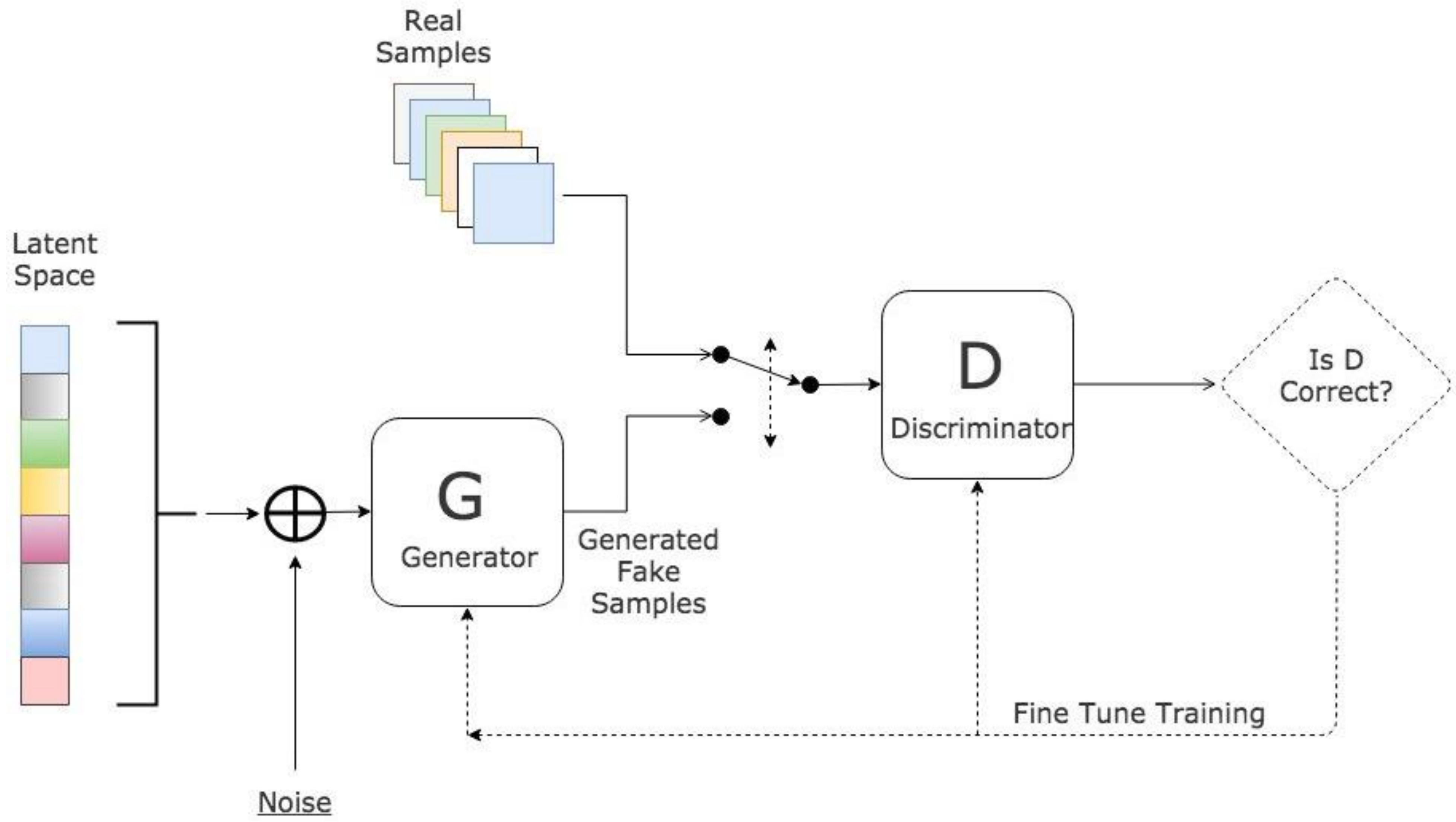
Vanilla GAN architecture

To train a GAN, we need five components.

1. **Noise sample generator Z**
2. **Generator G**
3. **Discriminator D**
4. **Loss L_D on discriminator D**
5. **Loss L_G on generator G**



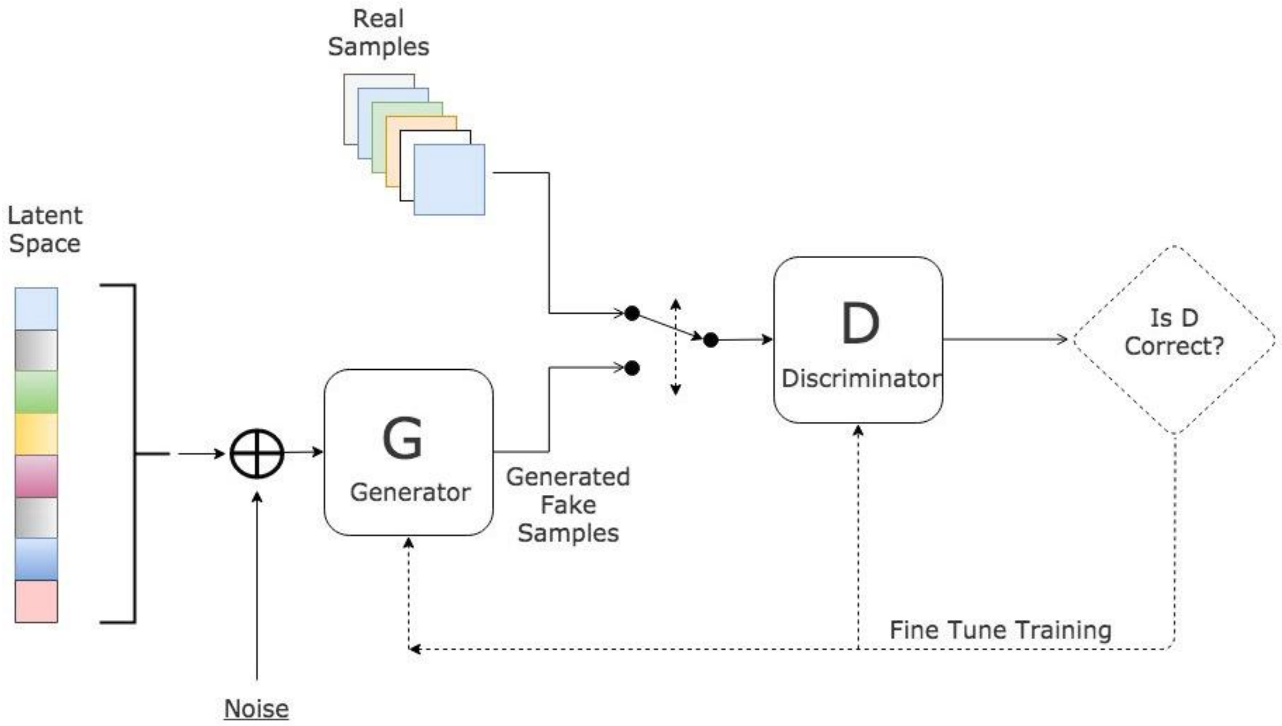
Vanilla GAN architecture



Vanilla GAN architecture

To train a GAN, we need five components.

1. **Noise sample generator $\boldsymbol{Z}$**
2. **Generator $\boldsymbol{G}$**
3. **Discriminator $\boldsymbol{D}$**
4. **Loss $\boldsymbol{L_D}$ on discriminator $\boldsymbol{D}$**
5. **Loss $\boldsymbol{L_G}$ on generator $\boldsymbol{G}$**



Vanilla GAN architecture

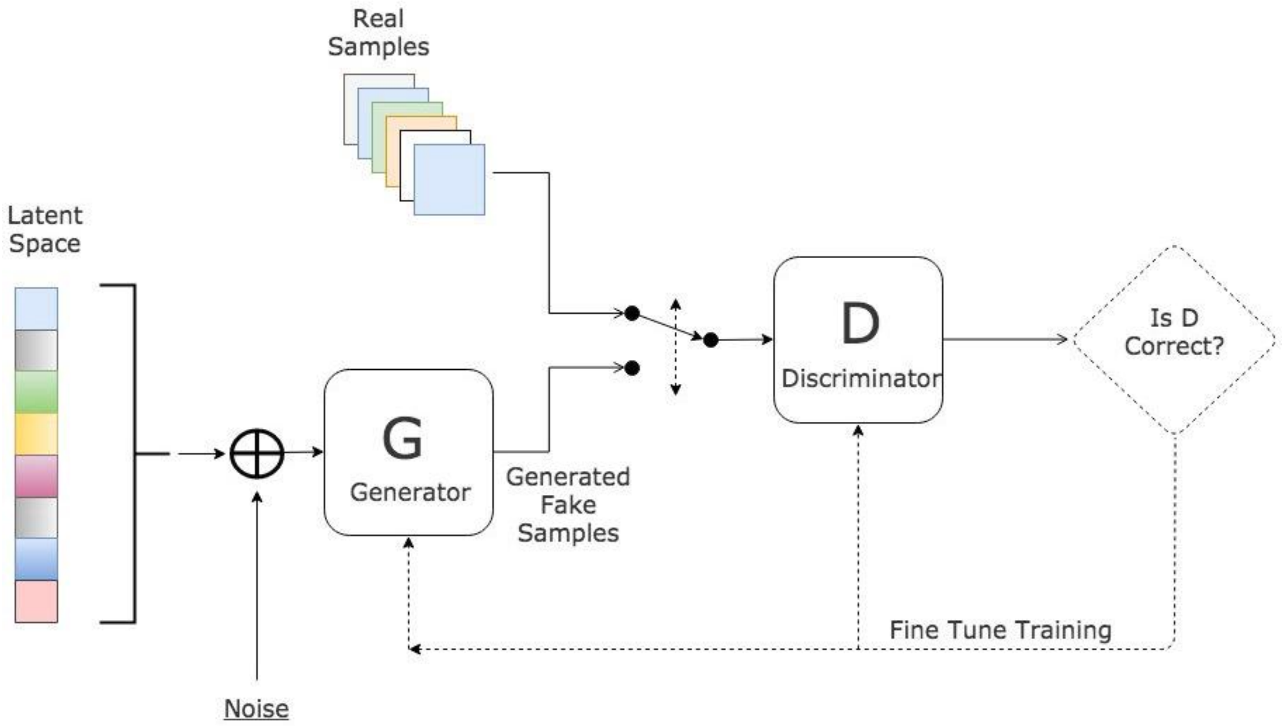
To train a GAN, we need five components.

1. **Noise sample generator Z**

Simply draw some random vector $z \in \mathbb{R}^K$, by using random noise.

$$z \to N(0_K, I_K)$$

It will be fed to our generator G.



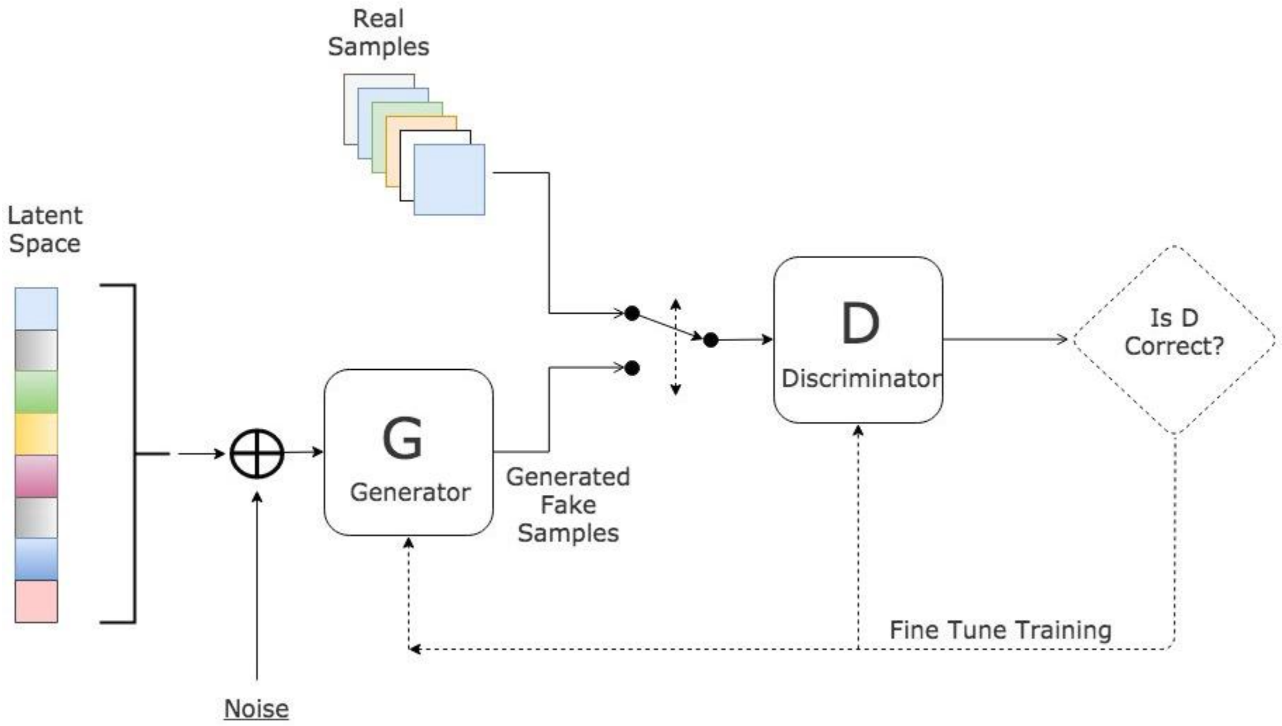
Vanilla GAN architecture

To train a GAN, we need five components.

2. Generator $\boldsymbol{G}$

Receives the noise vector z as input and produces a (fake) image $\hat{x}$ as output.

In terms of architecture, same logic as the decoder part of our AE/VAE, i.e. **upsampling FC** or **TransposeConv** layers!



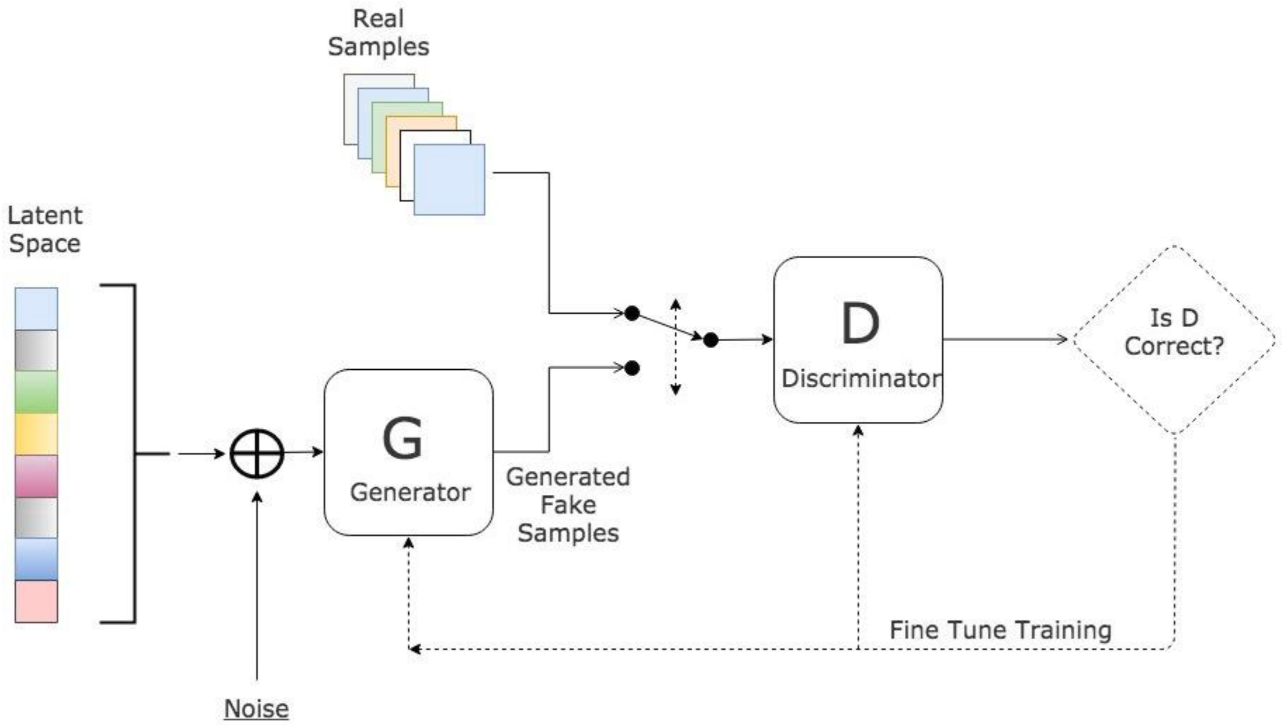
Vanilla GAN architecture

To train a GAN, we need five components.

3. Discriminator D

Will receive

- an image x from the dataset X half of the time,
- an image $\hat{x} = G(z)$ from the generator the other half.



Binary classification: needs to classify the image as fake (0) or real (1).

Vanilla GAN architecture

To train a GAN, we need five components.

4. **Loss L_D on discriminator D**

The purpose of D is to correctly guess whether the image is real or was generated by G (BCE loss!)

- The first component checks if the discriminator is correctly classifying the real samples.
- The second component checks if the discriminator correctly classifies the fake samples.

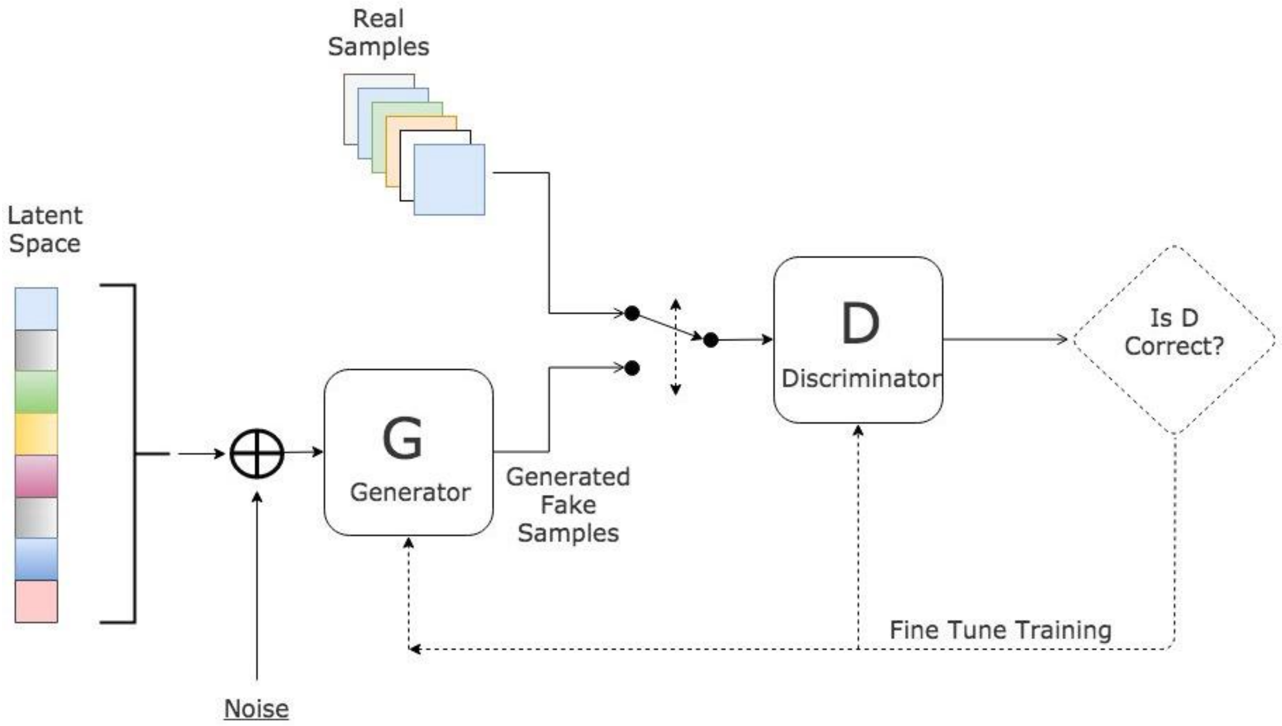
$$L_D = \frac{1}{N} \sum_{x_i \in minibatch(X_N)} -\log\left(D(x_i)\right) + \frac{1}{N} \sum_{z_i \in minibatch(N)} -\log\left(1 - D\left(G(z_i)\right)\right)$$

Vanilla GAN architecture

To train a GAN, we need five components.

5. **Loss $\boldsymbol{L_G}$ on generator $\boldsymbol{G}$**

The purpose of G is to create images that are good enough to fool the discriminator D.



$$L_G = \frac{1}{N} \sum_{z_i \in minibatch(N)} -1.\log\Big(D\big(G(z_i)\big)\Big)$$

Vanilla GAN architecture

To train a GAN, we need five components.

5. **Loss $\boldsymbol{L_G}$ on generator $\boldsymbol{G}$**

The purpose of G is to create images that are good enough to fool the discriminator D.

Intuitively,

- This loss checks that the discriminator classifies the **generated samples $\boldsymbol{G(z_i)}$**
- as real samples (1), instead of fake ones (0).

$$L_G = \frac{1}{N} \sum_{z_i \in minibatch(N)} -1 . \log\left(D\left(G(z_i)\right)\right)$$

GAN implementation: dataset

```
# Image transform to be applied to dataset
# - Tensor conversion
transform = transforms.Compose([transforms.ToTensor()])
```

```
# MNIST train dataset
mnist = torchvision.datasets.MNIST(root = './data/',
                                   train = True,
                                   transform = transform,
                                   download = True)
```

```
# Data loader
batch_size = 32
data_loader = torch.utils.data.DataLoader(dataset = mnist,
                                          batch_size = batch_size,
                                          shuffle = True)
```

GAN implementation: discriminator

- For simplicity, our discriminator, will be designed as stacked downsampling FC layers.
- Freely decide on the hidden layer sizes.
- (Later on, we will replace these layers with Conv2d ones, which are easier to train and much better at working with images!)

```
# Discriminator
class Dicriminator(nn.Module):

    def __init__(self, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.D = nn.Sequential(nn.Linear(image_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, 1),
                               nn.Sigmoid())

    def forward(self, x):
        return self.D(x)
```

GAN implementation: discriminator

```
# Discriminator
class Dicriminator(nn.Module):

    def __init__(self, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.D = nn.Sequential(nn.Linear(image_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, hidden_size),
                               nn.LeakyReLU(0.2),
                               nn.Linear(hidden_size, 1),
                               nn.Sigmoid())

    def forward(self, x):
        return self.D(x)
```

GAN implementation: generator

- For simplicity, our generator will mirror the operations of our discriminator.
- It will therefore consist of upsampling FC layers.
- (Later on, we will replace these layers with TransposeConv2d ones, which are easier to train and much better at working with images!)

```python
# Generator
class Generator(nn.Module):

    def __init__(self, latent_size, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.G = nn.Sequential(nn.Linear(latent_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, image_size),
                               nn.Tanh())

    def forward(self, x):
        return self.G(x)
```

GAN implementation: generator

```python
# Generator
class Generator(nn.Module):

    def __init__(self, latent_size, hidden_size, image_size):
        # Init from nn.Module
        super().__init__()

        # FC layers
        self.G = nn.Sequential(nn.Linear(latent_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, image_size),
                               nn.Tanh())

    def forward(self, x):
        return self.G(x)
```

GAN implementation: trainer

Start by defining hyperparameters and models.

- Latent size for noise samples fed to generator arbitrarily fixed.
- Adam, with almost default parameters, arbitrarily chosen.
- Losses on D and G for training curves later on.
- Also, accuracy scores of D on real and fake samples for visualization after training.

```python
# Hyperparameters for model generation and training
latent_size = 64
hidden_size = 256
image_size = 784
num_epochs = 300
batch_size = 32
```

```python
# Create discriminator model
D = Dicriminator(hidden_size, image_size)
D.to(device)
```

```python
# Create generator model
G = Generator(latent_size, hidden_size, image_size)
G.to(device)
```

```python
# Losses and optimizers
criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(D.parameters(), lr = 0.0002)
g_optimizer = torch.optim.Adam(G.parameters(), lr = 0.0002)
```

```python
# History trackers for training curves
# Keeping track of losses and accuracy scores
d_losses = np.zeros(num_epochs)
g_losses = np.zeros(num_epochs)
real_scores = np.zeros(num_epochs)
fake_scores = np.zeros(num_epochs)
```

GAN implementation: trainer

Start by processing the samples

- Generate N (mini-batch size) random noise samples, which will later be fed to generator. Make their labels 0.
- Draw N (mini-batch size) samples from the dataset X, which will be fed to discriminator later on. Make their labels 1.
- Flatten images (FC layers!)

```python
total_step = len(data_loader)
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(data_loader):
        # 1. Flatten image
        images = images.view(batch_size, -1).cuda()
        images = Variable(images)

        # 2. Create the labels which are later used as input for the BCE loss
        real_labels = torch.ones(batch_size, 1).cuda()
        real_labels = Variable(real_labels)
        fake_labels = torch.zeros(batch_size, 1).cuda()
        fake_labels = Variable(fake_labels)

```

GAN implementation: trainer

```
total_step = len(data_loader)
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(data_loader):
        # 1. Flatten image
        images = images.view(batch_size, -1).cuda()
        images = Variable(images)

        # 2. Create the labels which are later used as input for the BCE loss
        real_labels = torch.ones(batch_size, 1).cuda()
        real_labels = Variable(real_labels)
        fake_labels = torch.zeros(batch_size, 1).cuda()
        fake_labels = Variable(fake_labels)

```

GAN implementation: trainer

Train the discriminator

1. Pass real samples to D, check if it is able to classify these samples correctly as real (1).
2. Compute the first half of the loss and the accuracy of D on these real samples.
3. Pass noise samples to generator G, and its outputs to discriminator D, check if it is able to classify these samples correctly as fake (0).
4. Compute the second half of the loss and the accuracy of D on these fake samples.
5. Backpropagate D on the computed combined loss.

GAN implementation: trainer

```
        # 3. Compute BCE_Loss using real images
        # Here, BCE_Loss(x, y): - y * log(D(x)) - (1-y) * log(1 - D(x))
        # Second term of the loss is always zero since real_labels = 1
        outputs = D(images)
        d_loss_real = criterion(outputs, real_labels)
        real_score = outputs

        # 3.bis. Compute BCELoss using fake images
        # Here, BCE_Loss(x, y): - y * log(D(x)) - (1-y) * log(1 - D(x))
        # First term of the loss is always zero since fake_labels = 0
        z = torch.randn(batch_size, latent_size).cuda()
        z = Variable(z)
        fake_images = G(z)
        outputs = D(fake_images)
        d_loss_fake = criterion(outputs, fake_labels)
        fake_score = outputs

        # 4. Backprop and optimize for D
        # Remember to reset gradients for both optimizers!
        d_loss = d_loss_real + d_loss_fake
        d_optimizer.zero_grad()
        g_optimizer.zero_grad()
        d_loss.backward()
        d_optimizer.step()
```

GAN implementation: trainer

Train the generator

1. Produce a fresh batch of noise samples to be fed to the generator.
2. Produce fake images by feeding these noise samples to generator G.
3. Pass fake images to discriminator D, check if it is misclassifying these samples as real (1) instead of fake.
4. Backpropagate G on the computed loss.

GAN implementation: trainer

```python
        # 5. Generate fresh noise samples and produce fake images
        z = torch.randn(batch_size, latent_size).cuda()
        z = Variable(z)
        fake_images = G(z)
        outputs = D(fake_images)

        # 6. We train G to maximize log(D(G(z))
        # instead of minimizing log(1-D(G(z)))
        # (Strictly equivalent but empirically better)
        g_loss = criterion(outputs, real_labels)

        # 7. Backprop and optimize G
        # Remember to reset gradients for both optimizers!
        d_optimizer.zero_grad()
        g_optimizer.zero_grad()
        g_loss.backward()
        g_optimizer.step()
```

GAN implementation: trainer

Definition (interleaved training):

In general, we like to train a discriminator, while the generator is fixed, and vice-versa. We then alternate a few rounds of training on the discriminator/generator.

This prevents the discriminator from getting used to what the generator is producing and in turn, forces the generator to generate better samples, with higher chance of fooling the discriminator.

This is called an **interleaved training**, and is very common in GANs, or **game theory** with multiple players trying to figure out their best strategies.

However, this does not ensure that the GAN training will converge!

GAN implementation: trainer

Update loss and accuracy history after each mini-batch.

Display on periodic epoch values for convenience.

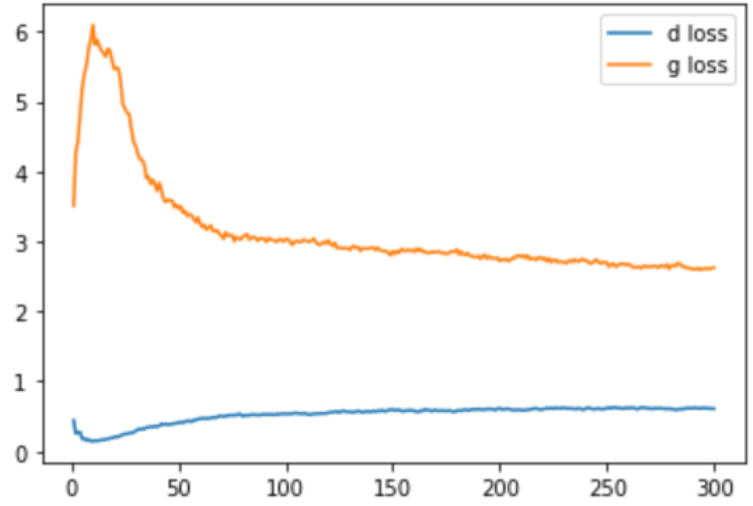
```
        # 8. Update the losses and scores for mini-batches
        d_losses[epoch] = d_losses[epoch]*(i/(i+1.)) \
            + d_loss.item()*(1./(i+1.))
        g_losses[epoch] = g_losses[epoch]*(i/(i+1.)) \
            + g_loss.item()*(1./(i+1.))
        real_scores[epoch] = real_scores[epoch]*(i/(i+1.)) \
            + real_score.mean().item()*(1./(i+1.))
        fake_scores[epoch] = fake_scores[epoch]*(i/(i+1.)) \
            + fake_score.mean().item()*(1./(i+1.))

        # 9. Display
        if (i+1) % 200 == 0:
            print('Epoch [{}/{}], Step [{}/{}], d_loss: {:.4f}, g_loss: {:.4f}, D(x): {:.2f}, D(G(z)): {:.2f}'
                  .format(epoch, num_epochs, i+1, total_step, d_loss.item(), g_loss.item(),
                          real_score.mean().item(), fake_score.mean().item()))
```

GAN training visuals

- After training, we can visualize the losses.
- Convergence seems to be happening on the losses.
- (A few more iterations would have probably been good).

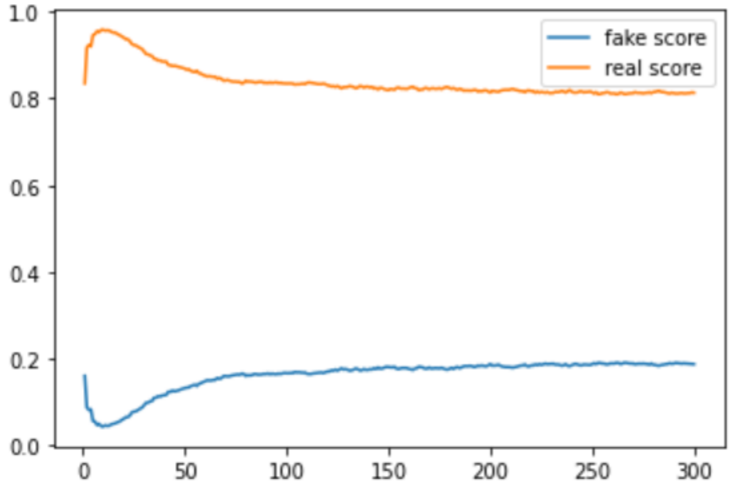
```python
# Display losses for both the generator and discriminator
plt.figure()
plt.plot(range(1, num_epochs + 1), d_losses, label = 'd loss')
plt.plot(range(1, num_epochs + 1), g_losses, label = 'g loss')
plt.legend()
plt.show()
```



GAN training visuals

- After training, we can visualize the accuracies as well.
- Convergence seems to be happening on the accuracies too.
- (A few more iterations would have probably been good).

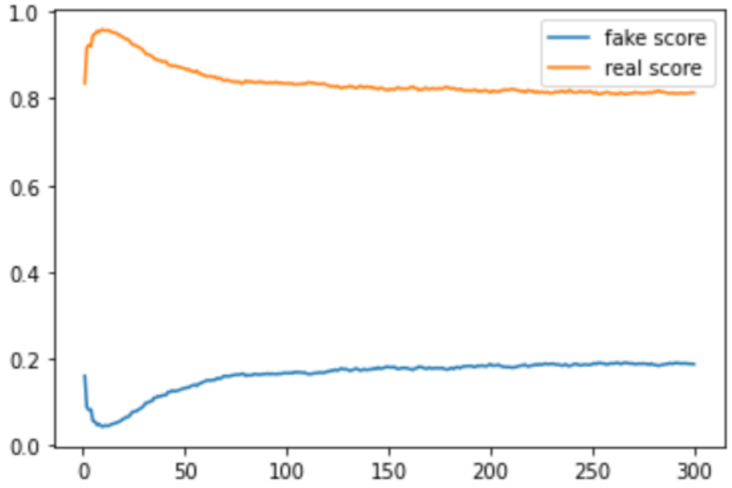
```python
# Display accuracy scores for both the generator and discriminator
plt.figure()
plt.plot(range(1, num_epochs + 1), fake_scores, label='fake score')
plt.plot(range(1, num_epochs + 1), real_scores, label='real score')
plt.legend()
plt.show()
```



GAN training visuals

- After training, we can visualize the accuracy scores of D on both the real and fake samples.
- Convergence seems to be happening on the accuracies as well.
- It does not seem that G was able to completely fool D (50% score accuracy as optimal target?)
- **But do we care?**

```python
# Display accuracy scores for both the generator and discriminator
plt.figure()
plt.plot(range(1, num_epochs + 1), fake_scores, label='fake score')
plt.plot(range(1, num_epochs + 1), real_scores, label='real score')
plt.legend()
plt.show()
```



GAN training visuals

Seriously, do we care?

- No, because, what matters is that we get a generator G that is trained well enough to generate **plausible (!) images**!
- (On a side note, when is the last time we saw this "**plausible**" keyword BTW? How about the "**adversarial**" one?
  Could that mean GANs could be used as attack functions?)

```
# Generate a few fake samples (5 of them) for visualization
n_samples = 5
z = torch.randn(n_samples, latent_size).cuda()
z = Variable(z)
fake_images = G(z)
fake_images = fake_images.cpu().detach().numpy().reshape(n_samples, 28, 28)
print(fake_images.shape)
```

```
(5, 28, 28)
```

```
# Display
plt.figure()
plt.imshow(fake_images[0])
plt.show()
plt.figure()
plt.imshow(fake_images[1])
plt.show()
plt.figure()
plt.imshow(fake_images[2])
plt.show()
plt.figure()
plt.imshow(fake_images[3])
plt.show()
plt.figure()
plt.imshow(fake_images[4])
plt.show()
```

GAN training visuals

```
# Generate a few fake samples (5 of them) for visualization
n_samples = 5
z = torch.randn(n_samples, latent_size).cuda()
z = Variable(z)
fake_images = G(z)
fake_images = fake_images.cpu().detach().numpy().reshape(n_samples, 28, 28)
print(fake_images.shape)
```

```
(5, 28, 28)
```

```
# Display
plt.figure()
plt.imshow(fake_images[0])
plt.show()
plt.figure()
plt.imshow(fake_images[1])
plt.show()
plt.figure()
plt.imshow(fake_images[2])
plt.show()
plt.figure()
plt.imshow(fake_images[3])
plt.show()
plt.figure()
plt.imshow(fake_images[4])
plt.show()
```

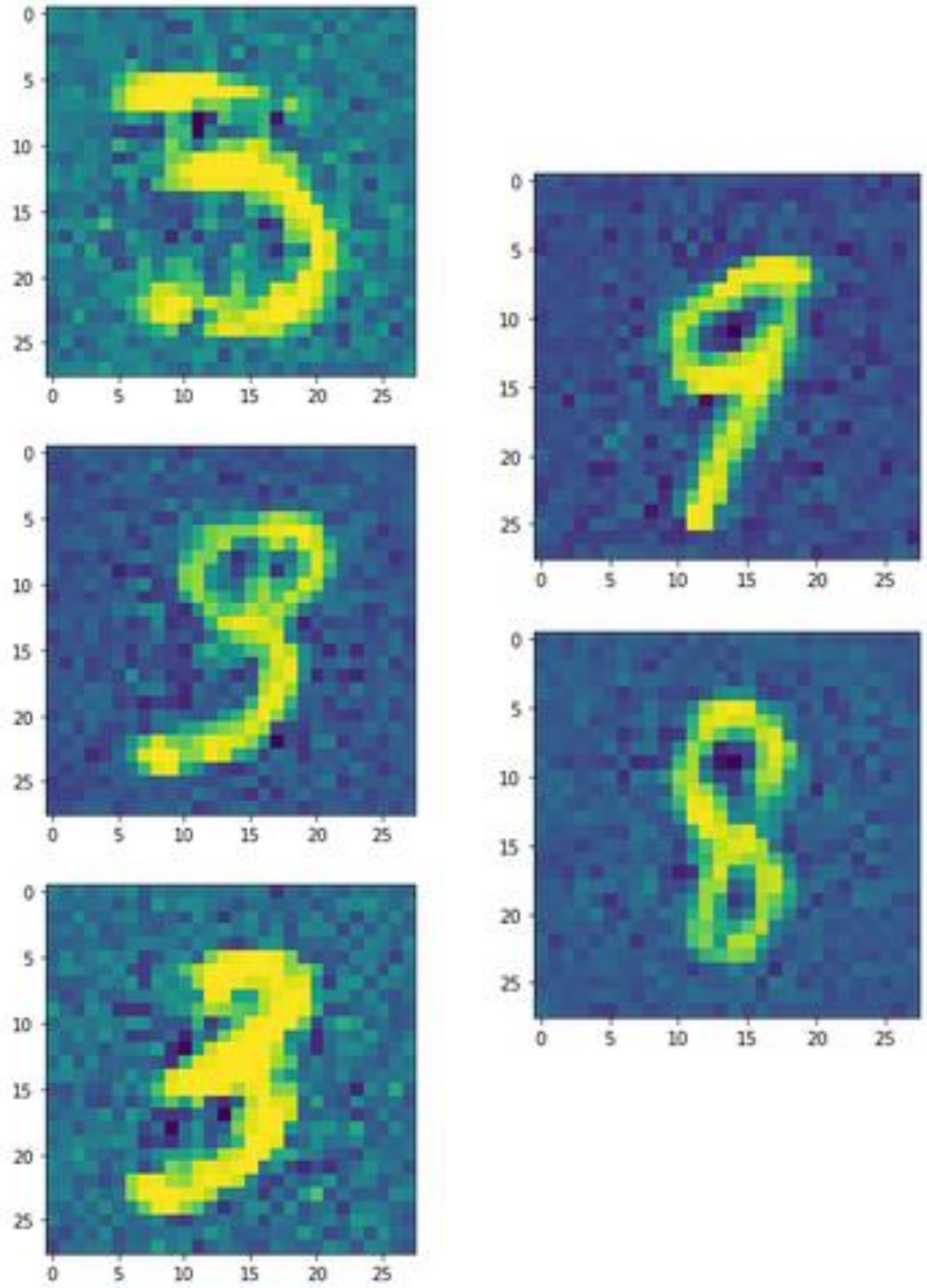
Restricted

GAN training visuals

Seriously, do we care?

- No, because, what matters is that we get a generator G that is trained well enough to generate **plausible (!) images**!

- Overall, plausible images!
- **Extra implementation:** could we remove noise, by using TransposeConv2d instead of FC layers in the generator?



Conclusion

In this lecture

- Main issues of traditional Autoencoders
- Diversity on feature representation using stochastic latent representations
- Variational Autoencoder and stochastic latent representations
- Basic Generative Adversarial Networks: ideas and procedure.

In the next lectures

- More advanced concepts on GANs

Learn more about these topics

Out of class, for those of you who are curious

- Reference paper on VAEs.
  Kingma et al., “An Introduction to Variational Autoencoders”, 2013.
  https://arxiv.org/abs/1906.02691
- Reference paper on vanilla GANs.
  Goodfellow et al., “Generative Adversarial Networks”, 2014.
  https://arxiv.org/abs/1406.2661
- Make music with VAEs!
  https://www.youtube.com/watch?v=G5JT16flZwM&list=PLBUMAYA6kvGU8Cgqh709o5SUvo-zHGTxr

A mathematical motivation (out-of-class)

- Suppose that there exists some hidden variable z which generates an observation x.
- Suppose that we would like to infer the characteristics of z, but we only see x.
- In other words, we'd like to compute $p(z|x)$.
- That is what our encoder does with x = original image and z = bottleneck vector.

- Using Bayes Theorem, we have

$$p(z|x) = \frac{p(x|z)\,p(z)}{p(x)}$$

- Unfortunately, computing $p(x)$ is difficult

$$p(x) = \int p(x|z)\,p(z)\,dz$$

A mathematical motivation (out-of-class)

- Suppose that there exists some hidden variable z which generates an observation x.
- Suppose that we would like to infer the characteristics of z, but we only see x.
- In other words, we'd like to compute $p(z|x)$.
- That is what our encoder does with x = original image and z = bottleneck vector.

- Using Bayes Theorem, we have
$$p(z|x) = \frac{p(x|z)\, p(z)}{p(x)}$$
- Unfortunately, computing $p(x)$ is difficult
$$p(x) = \int p(x|z)\, p(z)\, dz$$

This part is a Gaussian distribution (easy)

A mathematical motivation (out-of-class)

- Suppose that there exists some hidden variable z which generates an observation x.
- Suppose that we would like to infer the characteristics of z, but we only see x.
- In other words, we'd like to compute $p(z|x)$.
- That is what our encoder does with x = original image and z = bottleneck vector.

- Using Bayes Theorem, we have

$$p(z|x) = \frac{p(x|z)\, p(z)}{p(x)}$$

- Unfortunately, computing $p(x)$ is difficult

$$p(x) = \int p(x|z)\, p(z)\, dz$$

This part is a our decoder effect (easy)

A mathematical motivation (out-of-class)

- Suppose that there exists some hidden variable z which generates an observation x.
- Suppose that we would like to infer the characteristics of z, but we only see x.
- In other words, we'd like to compute $p(z|x)$.
- That is what our encoder does with x = original image and z = bottleneck vector.

- Using Bayes Theorem, we have

$$p(z|x) = \frac{p(x|z)\, p(z)}{p(x)}$$

- Unfortunately, computing $p(x)$ is difficult

$$p(x) = \int p(x|z)\, p(z)\, dz$$

Summing it over all the possible z representations? Hard to compute (requires understanding of encoder).

A mathematical motivation (out-of-class)

- Suppose that there exists some hidden variable z which generates an observation x.
- Suppose that we would like to infer the characteristics of z, but we only see x.
- In other words, we'd like to compute $p(z|x)$.
- That is what our encoder does with x = original image and z = bottleneck vector.
- Using Bayes Theorem, we have

$$p(z|x) = \frac{p(x|z)\, p(z)}{p(x)}$$

- Unfortunately, computing $p(x)$ is difficult

$$p(x) = \int p(x|z)\, p(z)\, dz$$

- **Oh, no, this is mathematically untractable!**

A mathematical motivation (out-of-class)

- **Workaround:** approximate $p(z|x)$ by another distribution $q(z|x)$ which will be defined such that it has a tractable distribution.
- If we can define the parameters of $q(z|x)$ such that it is very similar to $p(z|x)$, we can use it to perform approximate inference of the intractable distribution.
- We later simply compute $p(z|x)$, by using $q(z|x)$ instead.
- The only thing we need, then, is **a way to measure** if $q(z|x)$ is a good approximate measure of $p(z|x)$.
- In general, we will even choose a distribution q, independently of x, that is $q(z|x) = q(z)$.

On the minimization of the KL Divergence (out-of-class)

- Finding a good distribution q that matches p, then requires to minimize the KL divergence.

$$\min \sum_x p(x) \log\left(\frac{p(x)}{q(x)}\right)$$

- Applied to our problem

$$\min \sum_z -q(z) \log\left(\frac{q(z)}{p(z|x)}\right)$$

- This is equivalent to

$$\max \sum_z -q(z) \log\left(\frac{q(z)}{p(z|x)}\right)$$

On the minimization of the KL Divergence (out-of-class)

- Finding a good distribution q that matches p, then requires to minimize the KL divergence.

$$\min \sum_{x} p(x) \log\left(\frac{p(x)}{q(x)}\right)$$

- Applied to our problem

$$\min \sum_{x} -q(z) \log\left(\frac{q(z)}{p(z|x)}\right)$$

- This is equivalent to

$$\max \sum_{z} q(z) \log\left(\frac{q(z)}{p(z|x)}\right)$$

- Which is equivalent to

$$\max \sum_{z} -q(z) \log\left(\frac{p(z|x)}{q(z)}\right)$$

On the minimization of the KL Divergence (out-of-class)

- Using Bayes' formula

$$\max \sum_z -q(z) \log\left(\frac{p(x,z)}{q(z)}\right) + \sum_z q(z) \log(p(x))$$

$$\max \sum_z -q(z) \log\left(\frac{p(x,z)}{q(z)}\right) + \log(p(x)) \sum_z q(z)$$

- By definition, we have $\sum_z q(z) = 1$, and

$$\max \sum_z -q(z) \log\left(\frac{p(x,z)}{q(z)}\right) + \log(p(x))$$

On the minimization of the KL Divergence (out-of-class)

- But $\log(p(x))$ is a constant quantity here. It is then equivalent to

$$\max \sum_{z} -q(z) \log\left(\frac{p(x,z)}{q(z)}\right)$$

On the minimization of the KL Divergence (out-of-class)

- But $\log(p(x))$ is a constant quantity here. It is then equivalent to

$$\max \sum_{z} -q(z) \log\left(\frac{p(x,z)}{q(z)}\right)$$

- Splitting it with Bayes' formula again gives

$$\max \sum_{z} -q(z) \log\big(p(z|x)\big) + \sum_{z} q(z) \log\left(\frac{p(x)}{q(z)}\right)$$

$$\max \left[-E_{q(z)}[\log(p(z|x))] + D_{KL}(q||p) \right]$$

$$\min \left[E_{q(z)}[\log(p(z|x))] - D_{KL}(q||p) \right]$$

On the minimization of the KL Divergence (out-of-class)

- **Final twist:** $q(z)$ and $q(z|x)$ are identical (can you see why?)

$$\min \left[E_{q(z|x)}[\log(p(z|x))] - D_{KL}(q||p) \right]$$

- The first term represents the reconstruction log-likelihood, or in other word, a good reconstruction of the images!
- And the second term ensures that our learned distribution q is similar to the true prior distribution p!

(Need to take it slow? Full proof of previous calculation here https://www.youtube.com/watch?v=uaaqyVS9-rM&t=1182s)