✨ Member-only story

Variational Autoencoder (VAE) with Discrete Distribution using Gumbel Softmax

Theory and PyTorch Implementation

Alexey Kravets · Follow

Published in Towards Data Science
17 min read · Aug 10, 2023

Listen Share ••• More

https://unsplash.com/photos/sbVu5zitZt0

Since this article is going to be extensive, I will provide the reader with an index for better navigation:

Introduction

Generative models have become very popular nowadays thanks to their ability to generate novel samples with inherent variability by learning and capturing the underlying probability distribution of the training data.
We can identify two prominent families of generative models that are Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs) and Diffusion models. In this article, we are going to dive deep into VAEs with a particular focus on VAEs with categorical latent space.

Brief Introduction to Variational Autoencoders (VAEs)

Variational Autoencoders (VAEs) are a type of deep neural network used in unsupervised machine learning. They belong to the family of autoencoders, which are neural networks designed to learn efficient representations of data by compressing and then reconstructing it.

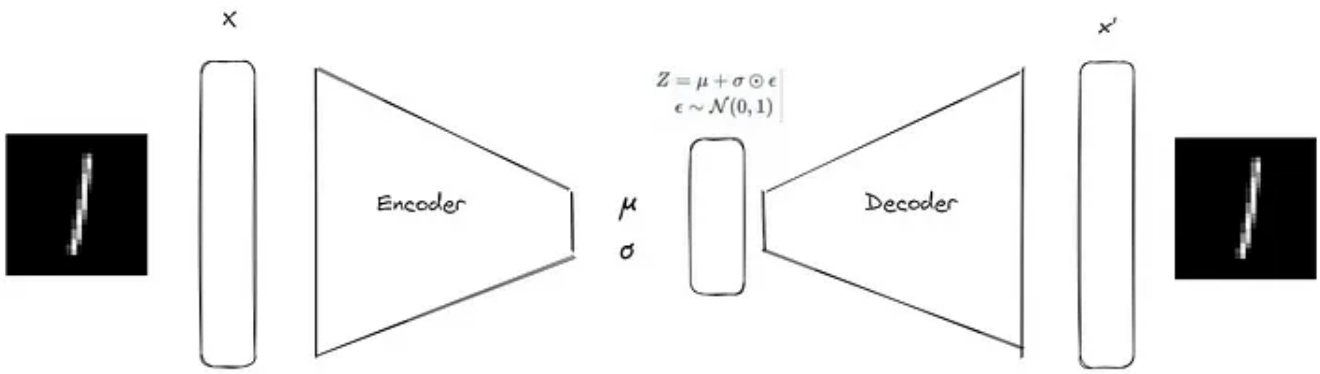
The main idea behind VAEs is to learn a probability distribution of the data in a **latent space**. This latent space is a lower-dimensional representation of the input data, where each point corresponds to a particular data sample. For example, given a vector in the latent space of dimension 3, we can think that the first dimension to represent the eyes shape, 2nd the amount of beard and 3rd the tan on a face of a generated picture of a person.

VAEs have two key components:

1. **Encoder**: The encoder network takes in the input data and maps it to the *parameters* of a probability distribution (usually Gaussian) in the latent space. Instead of directly producing a single point in the latent space, the encoder outputs the mean and variance of the distribution.
   Outputting a distribution instead of a single point in the latent space acts as regularization, so that when we pick a random point in the latent space, we always have a meaningful image once this data point is decoded.

2. **Decoder**: The decoder network takes samples from the latent space and reconstructs them back into the original data space. It converts the latent representation back to the data space using a process similar to that of the encoder but in reverse.

Let's illustrate this process:



VAE encoder-decoder diagram, Image by Author (1)

Where x is the input image, z is a sampled vector in the latent space, μ and σ are latent space parameters where μ is the means vector and σ is the standard deviations vector. Finally, x' is the reconstructed image from the latent variable.

We want this **latent space to have 2 properties**:

1. Close points in the latent space should output similarly looking pictures.

2. Any sampled point from the latent space should produce something similar to the training data, i.e., if we train on peoples faces it should not produce any face with 3 eyes or 4 ears.

To enforce the first, we need the encoder to map similar pictures to close latent space parameters and then the decoder to map them back to similarly looking

pictures — this is achieved via image reconstruction loss. To enforce the second, we need to add a regularization term. This regularization term is the Kullback–Leibler (KL) divergence between the parameters returned by the encoder and the standard Gaussian with mean of 0 and variance of 1 — N(0,1). By keeping the latent space close to N(0,1) we make sure that the encoder does not produce distributions too far apart from each other for each sample (by making means very different and variances very small) that would lead to overfitting. If this happened, sampling a value far away from any training point in the latent space would not produce a meaningful image.

Kullback–Leibler (KL) divergence

KL divergence, short for Kullback-Leibler divergence, is a measure of how one probability distribution differs from another. Given two probability distributions P(X) and Q(X), where X is a random variable, the KL divergence from Q to P, denoted as KL(Q || P), is a non-negative value that indicates how much information is lost when using Q to approximate P. It is not a symmetric measure, meaning KL(Q || P) is generally different from KL(P || Q). The formula for continuous and discrete variables are given by:

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

KL divergence, discrete case (2)

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

KL divergence, continuous case (3)

But what is the intuition behind this formula and how is it derived?
Suppose we have a dataset with observations **sampled from a distribution P(x)** — {x1, x2, …, xn}, and we want to compare how likely these observations are generated under the true distribution P(x) versus the approximation distribution Q(x). The likelihood of observing the entire dataset under a probability distribution can be calculated as the product of the individual probabilities of each observation:

- Likelihood of the data under P(x): L_P = P(x1) * P(x2) * … * P(xn)

- Likelihood of the data under Q(x): L_Q = Q(x1) * Q(x2) * … * Q(xn)

Taking the ratio L_P / L_Q, we can compare how similar they are. If the ratio is close to 1, the approximation distribution is similar to the true one, while if this ratio is high, which means that the likelihood of a sequence sampled from the true distribution according to the approximate distribution is significantly lower, the two distributions are different. Obviously, it cannot be less than 1 because the data are sampled from the true distribution P(x).

Taking the logarithm of this ratio on both sides, we get:

$$\log\left(\frac{L_P}{L_Q}\right) = \log P(x_1) + \log P(x_2) + \ldots + \log P(x_n) - \log Q(x_1) - \log Q(x_2) - \ldots - \log Q(x_n)$$

Log of the ratio L_P / L_Q (4)

Now, if we take the expectation of this logarithm with respect to the true distribution P(x) over the dataset, we get the expected log-likelihood ratio:

$$\mathbb{E}[\log(L_P/L_Q)] = \sum P(x)\log(P(x)) - \sum P(x)\log(Q(x)) = \sum P(x)\log\left(\frac{P(x)}{Q(x)}\right)$$

The expectation of the log of the ratio L_P / L_Q (5)

This is nothing else but the KL divergence! As a bonus, let's now dive a bit deeper to also understand how KL divergence is linked to cross-entropy. An attentive reader has probably recognized that
Σ P(x) * log(P(x)) in the formula is the **negative of the entropy** of P(x), while — Σ P(x) * log(Q(x)) is the **cross-entropy** between P(x) and Q(x). So, we have:

$$\mathbb{E}[\log(L_P/L_Q)] = \mathrm{KL}(P\|Q) = H(P,Q) - H(P)$$

KL divergence as the difference between cross-entropy and entropy (6)

Now, the entropy of the true data distribution P(x) is a constant that does not depend on the approximation distribution Q(x). Therefore, **minimizing the expected log-likelihood ratio E[log(L_P / L_Q)] is equivalent to minimizing the cross-entropy H(P, Q)** between the true distribution P(x) and the approximation distribution Q(x).

VAE loss

In the "Brief Introduction to Variational Autoencoders (VAEs)" section, we provided some intuition about how VAEs are optimized and that the latent space should satisfy 2 properties to generate meaningful images when we sample **any** random data point from the latent space that is enforced by the reconstruction loss and KL divergence regularization. In this section, we are going to dive into the mathematics of these two.

Given some training data x = {x1, x2, …, xn} generated from a latent variable z, our goal is to maximize the likelihood of this data to train our Variational Autoencoder model. The likelihood of the data is given by:

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

Data likelihood (7)

We integrated out the latent variable because its not observable.

Now, $p(x|z)$ can be easily computed with the decoder network, and $p(z)$ was assumed to be a Gaussian. However, we have one big problem here — computing this integral is actually impossible in the finite amount of time because we need to integrate over all the latent space. Thus, we use the Bayesian rule to compute our $p(x)$ differently:

$$p_\theta(x) = \frac{p_\theta(x \mid z)p(z)}{p_\theta(z \mid x)}$$

Bayesian rule for p(x) (8)

Now, $p(z|x)$ is intractable. The intractability of $p(z|x)$ arises because we need to compute the integral of $p(z|x)$ over all possible values of z for each data point x. Formally, this integral can be expressed as:

$$p_\theta(z \mid x) = \frac{p_\theta(x \mid z) \cdot p(z)}{p_\theta(x)} = \frac{p_\theta(x \mid z) \cdot p(z)}{\int p_\theta(x \mid z) \cdot p(z) dz}$$

Bayesian rule for p(z|x) (9)

Because of this intractability, in VAEs, we resort to using an approximate distribution (Gaussian in our case) $q(z|x)$ that is easier to work with and is computationally tractable. This approximate distribution is learned through the encoder network:

$$q_\phi(z \mid x) \approx p_\theta(z \mid x)$$

Approximated distribution of p(z|x) (10)

Now we have all the elements in place and we can approximate $p(x)$ with $p(x|z)$ computed with the decoder network and $p(z|x)$ approximated by the encoder q. Applying the log to both sides of equation 9 and doing some algebraic manipulations, we get:

$$\begin{aligned}\log p_\theta(x) &= \log \frac{p_\theta(x \mid z)p(z)}{p_\theta(z \mid x)} = \log \frac{p_\theta(x \mid z)p(z)q_\phi(z \mid x)}{p_\theta(z \mid x)q_\phi(z \mid x)} \\ &= \log p_\theta(x \mid z) - \log \frac{q_\phi(z \mid x)}{p(z)} + \log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\end{aligned}$$

log probability of p(x) (11)

Now, applying the Expectation operator on both sides :

$$E_z\left[\log p_\theta(x \mid z)\right] - E_z\left[\log \frac{q_\phi(z \mid x)}{p(z)}\right] + E_z\left[\log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\right]$$

$$\log p_\theta(x) = E_{z \sim q_\phi(z|x)}\left[\log p_\theta(x)\right]$$

Expectation of log probability of p(x) (12)

Which is equal to:

$$E_{z \sim q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x), p(z)\right) + D_{KL}\left(q_\phi(z \mid x), p_\theta(z \mid x)\right)$$

Expectation of log probability of p(x) — different form (13)

In the above figure, the first term is the reconstruction term, i.e., how well our model can reconstruct the training data *x* from the latent variable. The second term is the KL divergence between the prior of *z — N(0,1)* and the samples from the encoder. The third term is the KL divergence between the encoder and the posterior of the decoder, which is intractable. If we drop the last term, we get the lower bound on the data likelihood as KL is always ≥ 0 which is called Evidence Lower Bound (ELBO). Thus we finally have:

$$\log p_\theta(x) \geq E_{z\sim q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x), p(z)\right)$$

Evidence Lower Bound (ELBO) (14)

So when training VAE, we are trying to maximize ELBO, which is equivalent to maximizing the probability of our data.

Reparameterization Trick

Let's start with understanding what the reparameterization trick is, as it will be crucial to understand that Gumbel-Softmax uses something similar.
As we have seen in the first section, the encoder outputs the mean and the variance parameters of the Normal distribution, then we sample a random vector from the Normal variable with those parameters and pass this latent vector through the decoder to reconstruct the initial image. To minimize the reconstruction loss and make the network learn, we need to backpropagate from this reconstruction loss, but there is a problem — the **latent variable Z, which is a sample from a Gaussian is not differentiable**. Think about it — how can you differentiate a sample? Thus, we cannot use back-propagation. The solution to this is to use the reparameterization trick.

To make the random variable *Z* differentiable, we need to split it into a deterministic part which is differentiable, and a stochastic part which is not differentiable. Any sample from a random Normal $Z \sim N(\mu, \sigma)$ can be written as:
$Z = \mu + N(0,1) = \sigma = \mu + \boldsymbol{\varepsilon}\,\sigma$
where $\boldsymbol{\varepsilon} \sim N(0,1)$

So **μ and σ are deterministic,** and we can use back-propagation on it, while **ε is the stochastic part** which we cannot backpropagate. Thus, we can differentiate with

respect to μ and σ:

$$\frac{dZ}{d\mu} = \epsilon\,\sigma, \frac{dZ}{d\sigma} = \mu$$

Derivatives of random variable Z wrt mean and std (15)

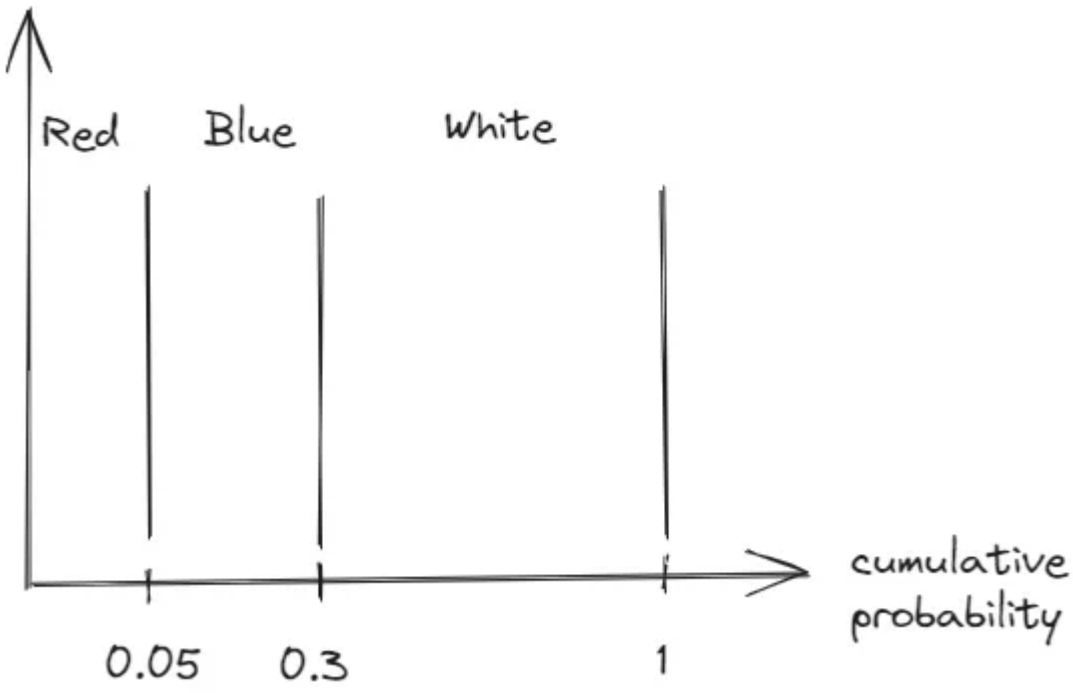
…to **learn the mean and the standard deviation of the Normal distribution in the latent space we sample from.**

Sampling from a categorical distribution & the Gumbel-Max Trick

What if, instead of having a continuous latent distribution, we want to model the latent space as a Categorical distribution? What is even the reason someone wants to do this, you will ask? Well, discrete representations can be useful in many cases, for example sampling discrete action in reinforcement learning problems, generation of discrete tokens in text, and so on.

So how can we sample from a categorical distribution and learn its parameters, making it differentiable? We can reuse the idea of the reparameterization trick, adapting it to this problem!

Firstly though, let's try to understand how to sample from a categorical distribution. Say we have the following vector of probabilities:
theta = [0.05, 0.25, 0.7] that represent the following categories — [Red, Blue, White].
To sample, we need a source of randomness where Uniform distribution between 0 and 1 is normally used. Recall that with a Uniform distribution, sampling between 0 and 1 is equally likely. Thus, we sample from a Uniform, and to transform it to Categorical, we can slice it according to our probabilities *theta.* Let's define a cumulative sum vector *theta_cum* = [0.05, 0.3, 1] which represents the graph below.
Given this sample from a Uniform distribution, e.g., 0.31, we choose the category whose cumulative probability exceeds the generated random number.
argmax(theta_cum ≥ U(0,1)) = argmax([False, True, True])
Which corresponds to "Blue" in the example as argmax takes the first index corresponding to *True*.



Cumulative probability categorical distribution, Image by Author (16)

Now, there is another way we can sample from a categorical distribution — instead of using Uniform distribution, we use Gumbel distribution defined as:

$$G_i \sim -log(-log(U(0,1))$$

Gumbel distribution (17)

Assuming we have a vector of (log) probabilities like before
theta = [log(alpha1), log(alpha2), log(alpha3)], which are parameters that we want to estimate using backpropagation. To use backpropagation, we replicate what was done in the reparameterization trick section — have a deterministic part, i.e., class log probabilities that are our parameters and a stochastic part given by a random standard Gumbel noise.
To sample from a categorical distribution using Gumbel, we do the following:
argmax([log(alpha1) + G1, log(alpha2) + G2, log(alpha3) + G3])

Where *theta* is the deterministic part, and Gumbel noise is the stochastic part. We can propagate through this sum of deterministic and stochastic parts. However, **argmax is not a differentiable** function. Thus we replace it with **Softmax** with a temperature **τ** to make everything differentiable. So the probability of a category *yi* becomes:

$$y_i = \frac{\sum_{j=1}^{k} \exp((\log p_j + g_j)/\tau)}{\exp((\log p_i + g_i)/\tau)}$$

Sample with Gumbel-Softmax distribution (18)

Low τ will make the Softmax more similar to argmax, while higher τ will make it closer to the Uniform distribution. Indeed, as we decrease the temperature to very low values like 1e-05, the probabilities become almost like selecting an argmax, i.e., we basically sample from a discrete distribution.

Implementation

We take as an example the MNIST dataset (License: Public Domain / Source: http://yann.lecun.com/exdb/mnist/, also available in *torchvision.datasets*) with the objective of learning a generative model assuming binary images. The latent variable size is assumed to be 20 with 10 categorical variables (10 numbers). The prior is a categorical distribution over 10 categories with a Uniform prior probability of 1/10.

1. Let's start from implementing the Gumbel softmax function `gumbel_softmax`. As we said previously, this is given by the sum of log probabilities (logits) of each category + some randomness given by the Gumbel distribution. In case of 3 categories we have:
softmax([log(alpha1) + G1, log(alpha2) + G2, log(alpha3) + G3])
Softmax is used instead instead of argmax for differentiability.

```
def sample_gumbel(shape, eps=1e-20):
    # sample from a uniform distribution
    U = torch.rand(shape)
    if is_cuda:
        U = U.cuda()
    return -torch.log(-torch.log(U + eps) + eps)

def gumbel_softmax_sample(logits, temperature):
    y = logits + sample_gumbel(logits.size())
    return F.softmax(y / temperature, dim=-1)


def gumbel_softmax(logits, temperature, hard=False):
    y = gumbel_softmax_sample(logits, temperature)

    if not hard:
```

```
            return y.view(-1, latent_dim * categorical_dim)

        shape = y.size()
        _, ind = y.max(dim=-1)
        y_hard = torch.zeros_like(y).view(-1, shape[-1])
        y_hard.scatter_(1, ind.view(-1, 1), 1)
        y_hard = y_hard.view(*shape)
        # skip the gradient of y_hard
        y_hard = (y_hard - y).detach() + y
        return y_hard.view(-1, latent_dim * categorical_dim)
```

One additional note:

We can notice one small trick in `gambel_softmax` function — if the parameter `hard` is True, we take *argmax* instead of softmax. When evaluating, we normally take the *argmax* (this is what we do in. `model.samle_img`), while during training, we use softmax because of the non-differentiability of the *argmax* operation. However, this is not necessary, and we can take *argmax* during training too, by **skipping** the gradient of `y_hard` in `gumbel_softmax` function and differentiating w.r.t. softmax `y`. A short example will clarify:

```
skip_d = False

a = torch.Tensor([1])
a.requires_grad = True

b = torch.Tensor([2])
b.requires_grad = True

c = 2 * (a + b)

if skip_d:
    d = c ** 2
    d = (d - c).detach() + c
else:
    d = c ** 2

f = d * 4
f.retain_grad()
d.retain_grad()
c.retain_grad()

loss = f * 3
loss.backward()
```

```
print(loss)
print(a.grad, b.grad, c.grad, d.grad, f.grad)
# Loss value: tensor([432.])
# (tensor([288.]), tensor([288.]), tensor([144.]), tensor([12.]), tensor([3.]))

# Running the same with skip_d = True we get:
# tensor([432.])
# (tensor([24.]), tensor([24.]), tensor([12.]), tensor([12.]), tensor([3.]))
```

When *skip_d = False* we have:
dl/df = 3
dl/dd = dl/df * df/dd = (3) * (4) = 12
dl/dc = dl/df * df/dd * dd/dc = (3) * (4) * (2 * c) = 144
dl/da = dl/df * df/dd * dd/dc * dc/da = (3) * (4) * (2 * c) * (2) = 288
dl/db = dl/df * df/dd * dd/dc * dc/db = (3) * (4) * (2 * c) * (2) = 288

While when *skip_d = True:*
dl/df = 3
dl/dd = dl/df * df/dd = (3) * (4) = 12
dl/dc = dl/df * df/dd = (3) * (4) = 12
From now on we skip dd/dc, i.e. we set the gradient dl/dc = dl/dd.
dl/da = dl/df * df/dd * dc/da = (3) * (4) * (2) = 24
dl/db = dl/df * df/dd * dc/db = (3) * (4) * (2) = 24

In the example above, the value of the *loss* is the same but the gradients are different. In our model the value will not be the same though as we are setting `latent_z` equal to `y_hard` when `hard=True` and equal to softmax `y` when `hard=False`, but the backpropagated gradients of `y` will be the same in both cases.

2. Now let's define our VAE model. The encoder, which takes an image and maps it to the log probabilities of the categorical variables, is given by 3 linear layers with ReLU non-linearities. The decoder, that maps back the latent space vector to the image space, is given by 3 linear layers with 2 ReLU non-linearities and last sigmoid non-linearity. Sigmoid outputs directly the probability, which is convenient as we model our MNIST images (each pixel) as a Bernoulli variable.

```
class VAE_model(nn.Module):
    def __init__(self):
```

```
        super(VAE_model, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, latent_dim * categorical_dim)
        self.fc4 = nn.Linear(latent_dim * categorical_dim, 256)
        self.fc5 = nn.Linear(256, 512)
        self.fc6 = nn.Linear(512, 784)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def encode(self, x):
        h1 = self.relu(self.fc1(x))
        h2 = self.relu(self.fc2(h1))
        return self.relu(self.fc3(h2))

    def decode(self, z):
        h4 = self.relu(self.fc4(z))
        h5 = self.relu(self.fc5(h4))
        return self.sigmoid(self.fc6(h5))
```

In the forward function, we first compute the logits from the encoder with the Gumbel Softmax:

```
logits_z = self.encode(data.view(-1,
logits_z = logits_z.view(-1, latent_dim, categorical_dim)
latent_z = gumbel_softmax(logits_z, temp)
latent_z = latent_z.view(-1, latent_dim * categorical_dim)
```

Then, we decode them that gives us the probability of a Bernoulli for each pixel. We can then sample from it to generate an image with the probabilities parameters:

```
probs_x = self.decode(latent_z)
# we assumed distribution of the data is Bernoulli
dist_x = torch.distributions.Bernoulli(probs=probs_x, validate_args=False)
```

Next, let's compute the ELBO loss

$$E_{z\sim q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x), p(z)\right)$$

real data under our estimated model, which this tells us how likely is the real image under our model. We have computed before `dist_x` from the decoder, which is what we are going to use to estimate this probability:

```
# reconstruction loss - log probabilities of the data
rec_loss = dist_x.log_prob(data.view(-1, 784)).sum(dim=-1)
```

Then we compute the regularization given by the KL divergence between the prior given by categorical distribution over 10 categories with a Uniform prior probability 1/10 and the latent space categorical parameters:

```
# KL divergence loss
KL = (posterior_distrib.probs * (logits_z_log - prior_distrib.probs.log())).vie
```

The full code, including the training function and plotting utilities are given below:

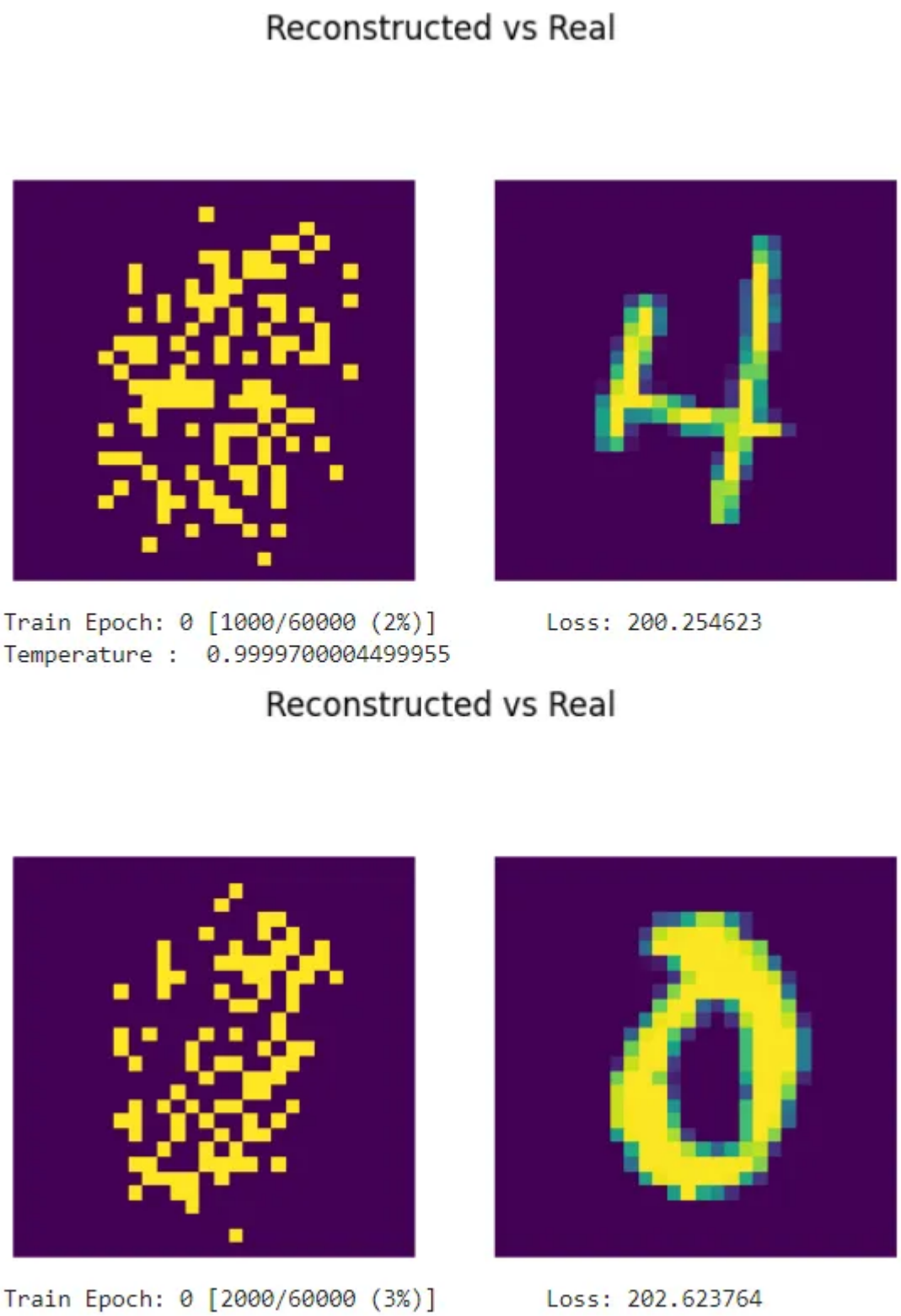
```
torch.manual_seed(0)

batch_size = 100
temperature = 1.0
seed = 0
log_interval = 10
hard = False
is_cuda = torch.cuda.is_available()
torch.manual_seed(seed)
if is_cuda:
    torch.cuda.manual_seed(seed)
kwargs = {'num_workers': 1, 'pin_memory': True} if is_cuda else {}
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data/MNIST', train=True, download=True,
                   transform=transforms.ToTensor()),
    batch_size=batch_size, shuffle=True, **kwargs)

def sample_gumbel(shape, eps=1e-20):
    # sample from a uniform distribution
```

```
    U = torch.rand(shape)
    if is_cuda:
        U = U.cuda()
    return -torch.log(-torch.log(U + eps) + eps)

def gumbel_softmax_sample(logits, temperature):
    y = logits + sample_gumbel(logits.size())
    return F.softmax(y / temperature, dim=-1)


def gumbel_softmax(logits, temperature, hard=False):
    y = gumbel_softmax_sample(logits, temperature)

    if not hard:
        return y.view(-1, latent_dim * categorical_dim)

    shape = y.size()
    _, ind = y.max(dim=-1)
    y_hard = torch.zeros_like(y).view(-1, shape[-1])
    y_hard.scatter_(1, ind.view(-1, 1), 1)
    y_hard = y_hard.view(*shape)
    # skip the gradient of y_hard
    y_hard = (y_hard - y).detach() + y
    return y_hard.view(-1, latent_dim * categorical_dim)


class VAE_model(nn.Module):
    def __init__(self):
        super(VAE_model, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, latent_dim * categorical_dim)
        self.fc4 = nn.Linear(latent_dim * categorical_dim, 256)
        self.fc5 = nn.Linear(256, 512)
        self.fc6 = nn.Linear(512, 784)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def sample_img(self, img, temp, random=True):
        # evaluation
        with torch.no_grad():
            logits_z = self.encode(img.view(-1, 784))
            logits_z = logits_z.view(-1, latent_dim, categorical_dim)
            if random:
                latent_z = gumbel_softmax(logits_z, temp, True)
            else:
                latent_z = logits_z.view(-1, latent_dim * categorical_dim)
            logits_x = self.decode(latent_z)
            # probs instead of logits because we have sigmoid activation
            # in the decoder
            dist_x = torch.distributions.Bernoulli(probs=logits_x)
            sampled_img = dist_x.sample()
        return sampled_img
```

```
    def encode(self, x):
        h1 = self.relu(self.fc1(x))
        h2 = self.relu(self.fc2(h1))
        return self.relu(self.fc3(h2))

    def decode(self, z):
        h4 = self.relu(self.fc4(z))
        h5 = self.relu(self.fc5(h4))
        return self.sigmoid(self.fc6(h5))

    def forward(self, data, temp, hard):
        logits_z = self.encode(data.view(-1, 784))
        logits_z = logits_z.view(-1, latent_dim, categorical_dim)

        # estimated posterior probabiity coefficients
        probs_z = F.softmax(logits_z, dim=-1)
        posterior_distrib = torch.distributions.Categorical(probs=probs_z)
        # categorical prior
        probs_prior = torch.ones_like(logits_z)/categorical_dim
        prior_distrib = torch.distributions.Categorical(probs=probs_prior)

        latent_z = gumbel_softmax(logits_z, temp)
        latent_z = latent_z.view(-1, latent_dim * categorical_dim)

        probs_x = self.decode(latent_z)
        # we assumed distribution of the data is Bernoulli
        dist_x = torch.distributions.Bernoulli(probs=probs_x, validate_args=Fal
        # Losses
        # reconstruction loss - log probabilities of the data
        rec_loss = dist_x.log_prob(data.view(-1, 784)).sum(dim=-1)
        logits_z_log = F.log_softmax(logits_z, dim=-1)
        # KL divergence loss
        KL = (posterior_distrib.probs * (logits_z_log - prior_distrib.probs.log
        elbo = rec_loss - KL
        loss = -elbo.mean()
        return loss

def train(epoch, model, optimizer):
    model.train()
    train_loss = 0
    temp = temperature
    for batch_idx, (data, _) in enumerate(train_loader):
        if is_cuda:
            data = data.cuda()
        optimizer.zero_grad()
        loss = model(data, temp, hard)
        loss.backward()
        train_loss += loss.item() * len(data)
        optimizer.step()
        if batch_idx % 100 == 1:
            temp = np.maximum(temp * np.exp(-ANNEAL_RATE * batch_idx), temp_mir
        if batch_idx % log_interval == 0:
```

```
                print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                    epoch, batch_idx * len(data), len(train_loader.dataset),
                           100. * batch_idx / len(train_loader),
                           loss.item()))
                print("Temperature : ", temp)

                sampled = model.sample_img(data[0].view(-1, 28*28), temp).view(28,
                fig, axs = plt.subplots(1, 2, figsize=(6,4))
                fig.suptitle('Reconstructed vs Real')
                axs[0].imshow(sampled.reshape(28,28))
                axs[0].axis('off')
                axs[1].imshow(data[0].reshape(28,28).detach().cpu())
                axs[1].axis('off')
                plt.show()
        print('====> Epoch: {} Average loss: {:.4f}'.format(
            epoch, train_loss / len(train_loader.dataset)))

    ### Train
    temp_min = 0.5
    ANNEAL_RATE = 0.00003
    latent_dim = 20
    categorical_dim = 10
    my_model = VAE_model()
    my_model.to('cuda:0')
    optimizer = optim.Adam(my_model.parameters(), lr=1e-3)
    for epoch in range(3):
        train(epoch, my_model, optimizer)
```

At the beginning of the training we have high loss and bad reconstruction:



Reconstruction vs Real, **start** of training, Image by Author (20)

Towards the end of the training, we get quite a good reconstruction and much lower loss. Obviously, we could train for longer to get even better reconstruction.

Reconstruction vs Real, **end** of training, Image by Author (21)

Conclusions

In this article, we discovered that VAE can also be modeled with categorical latent space. This becomes very useful when we want to sample discrete actions in reinforcement learning problems or generate discrete tokens for text. We encountered a problem when trying to differentiate the *argmax* operation to select the categorical variable, as *argmax* is not differentiable, but this was solved thanks to the Gumbel Softmax inspired by the reparameterization trick.

References

[1] https://jhui.github.io/2017/03/06/Variational-autoencoders/

[2] https://blog.evjang.com/2016/11/tutorial-categorical-variational.html

[3] https://www.youtube.com/watch?v=Q3HU2vEhD5Y&list=PL5-TkQAfAZFbzxjBHtzdVCWE0Zbhomg7r&index=19

[4] https://arxiv.org/pdf/1611.01144.pdf

[5] https://github.com/shaabhishek/gumbel-softmax-pytorch

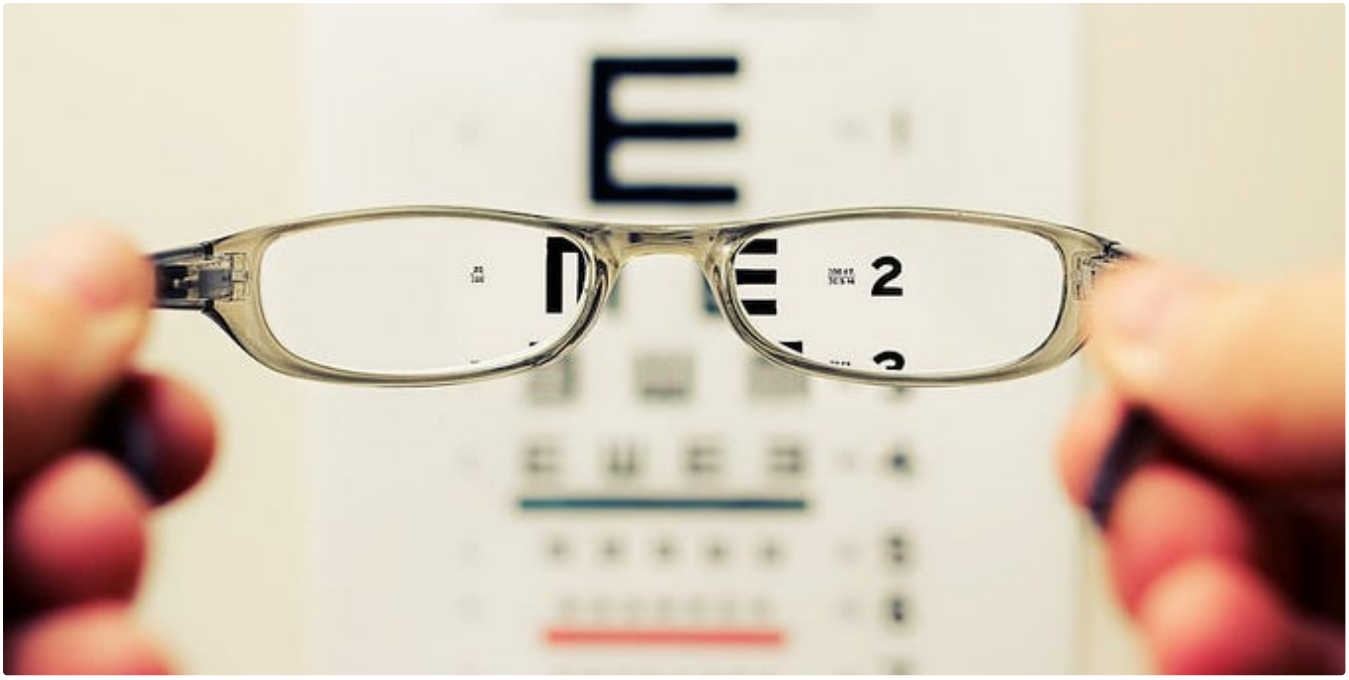
Variational Autoencoder | Data Science | Machine Learning | Deep Learning | Generative Ai Tools

Written by Alexey Kravets

154 Followers · Writer for Towards Data Science

PhD student in AI at the University of Bath researching Vision & Language models. My LinkedIn profile: https://www.linkedin.com/in/alexey-kravets-647663137/

More from Alexey Kravets and Towards Data Science



Alexey Kravets in Towards Data Science

A Deep Dive into the Code of the Visual Transformer (ViT) Model

Breaking down the HuggingFace ViT Implementation

✨ · 14 min read · Aug 16, 2023

132

Thu Vu in Towards Data Science

How to Learn AI on Your Own (a self-study guide)

If your hands touch a keyboard for work, Artificial Intelligence is going to change your job in the next few years.

✨ · 12 min read · Jan 5

2.2K 23

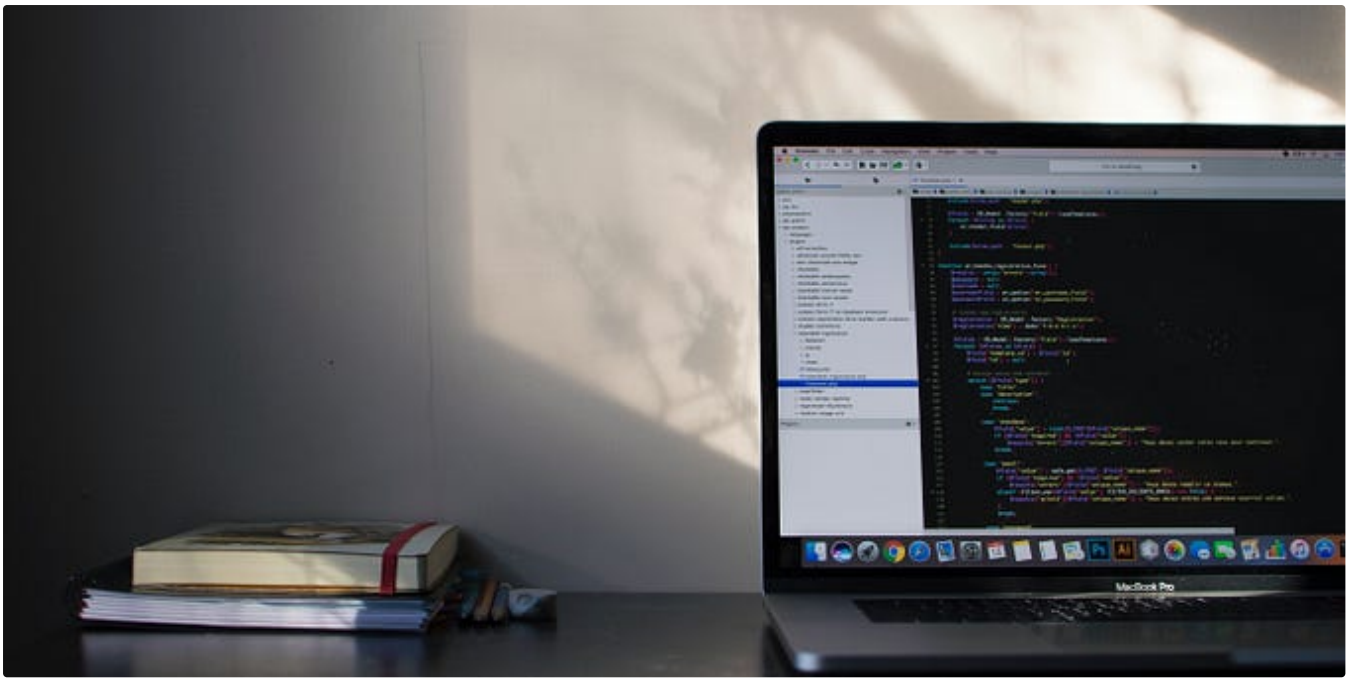
Michael Berk in Towards Data Science

1.5 Years of Spark Knowledge in 8 Tips

My learnings from Databricks customer engagements

8 min read · Dec 24, 2023

1.4K 8



 Alexey Kravets in Towards Data Science

A Deep Dive into the Code of the BERT Model

Breaking down the HuggingFace BERT Implementation

✨ · 10 min read · Dec 14, 2021

162 2

See all from Alexey Kravets

See all from Towards Data Science

Recommended from Medium

Michael Galkin in Towards Data Science

Graph & Geometric ML in 2024: Where We Are and What's Next (Part II — Applications)

Trends and recent advancements in Graph and Geometric Deep Learning

42 min read · 6 days ago

341 4

Christopher Thomas BSc Hons. MIAP in Thought Vector

Latent Diffusion and Perceptual Latent Loss

This article shows a novel approach to training a generative model for image generation at reduced training times using latents and using a...

9 min read · Nov 26, 2023

Tiya Vaj

Variational Autoencoders

Variational Autoencoders (VAEs) are valuable in tasks that involve combining data, removing noise, and representing the inherent...

2 min read · Dec 26, 2023

1

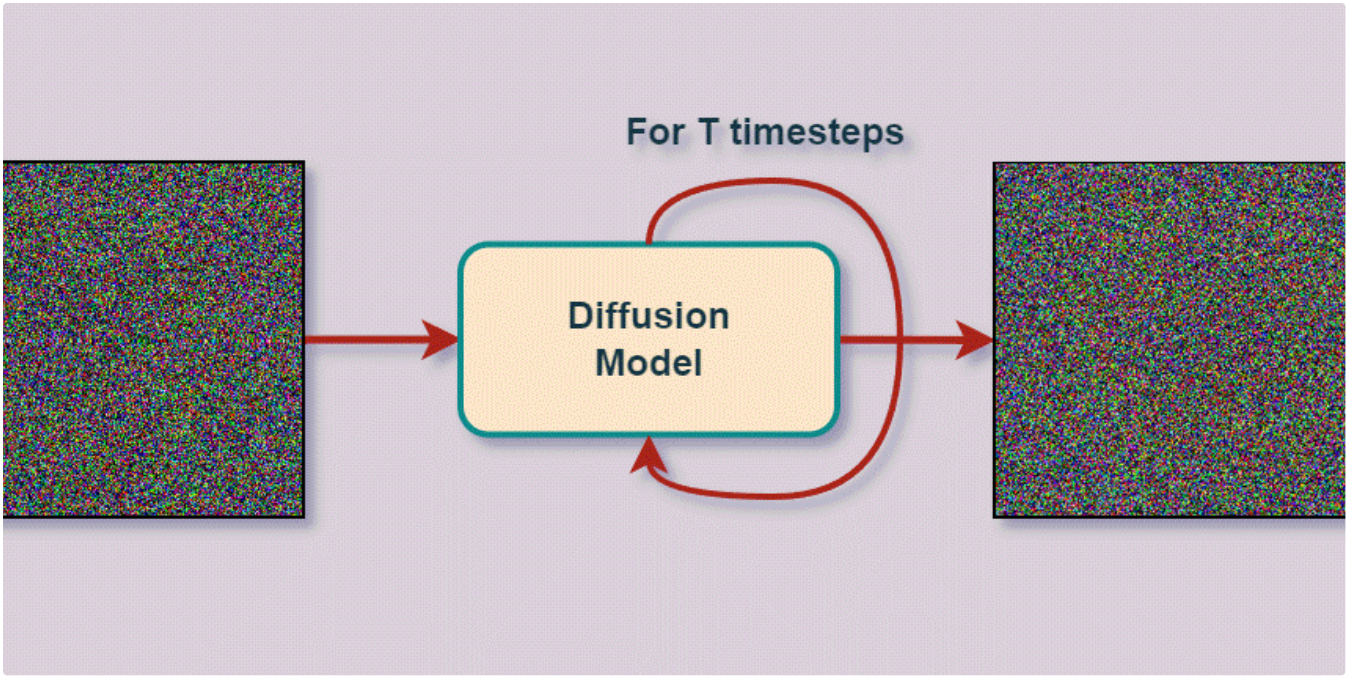
Harshita Sharma in Accredian

Building Intuition: Variational Autoencoders (VAEs)

How Variational Autoencoders are able to generate new data

7 min read · Sep 25, 2023

14



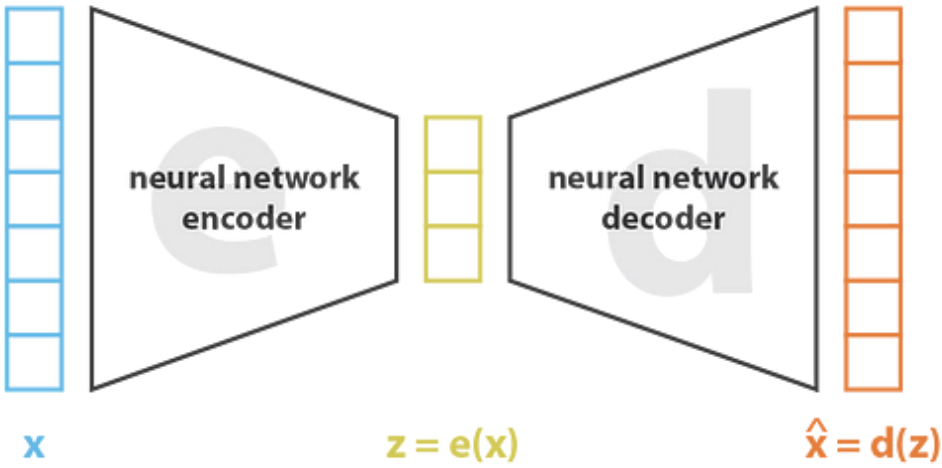
Gabriel Mongaras in Better Programming

Diffusion Models—DDPMs, DDIMs, and Classifier Free Guidance

A guide to the evolution of diffusion models from DDPMs to Classifier Free guidance

28 min read · Mar 13, 2023

616 8



$$\text{loss} = \| x - \hat{x} \|^2 = \| x - d(z) \|^2 = \| x - d(e(x)) \|^2$$

Etorezone

Understanding the Differences Between AutoEncoder (AE) and Variational AutoEncoder (VAE)

Are you familiar with the nuances between AutoEncoder (AE) and Variational AutoEncoder (VAE)?

2 min read · Oct 2, 2023