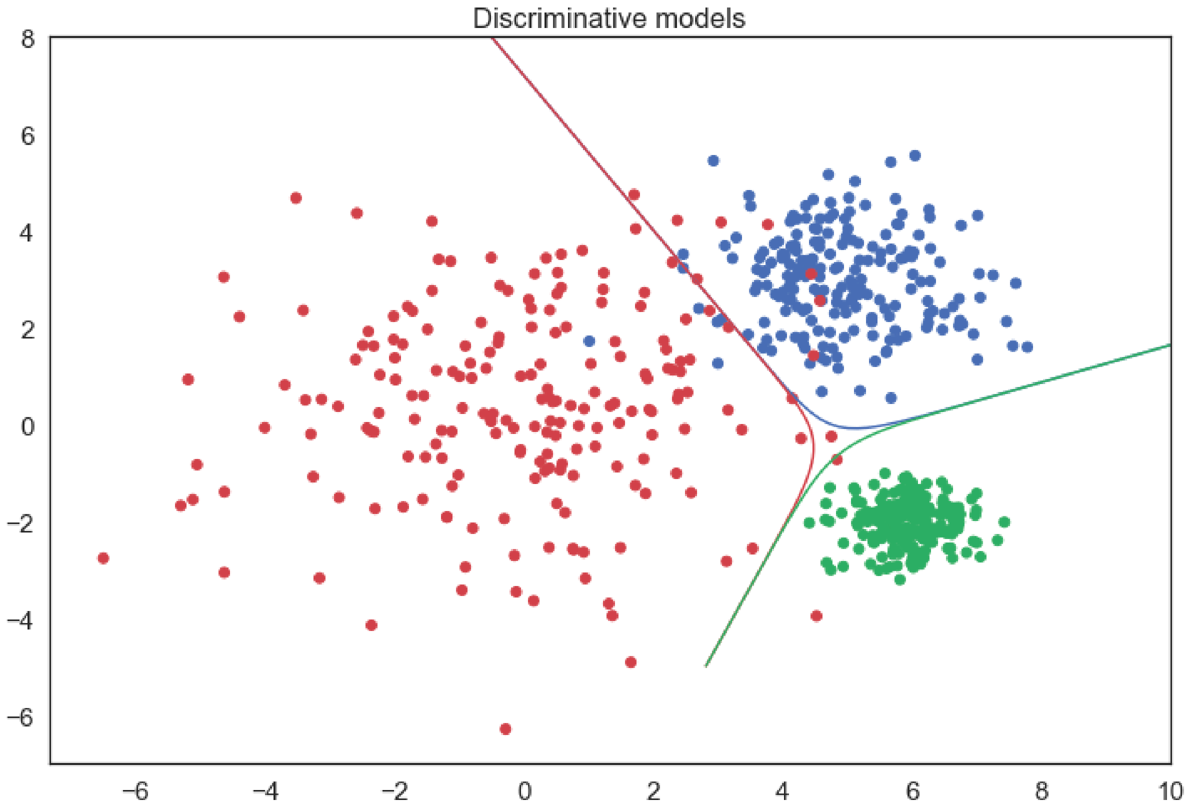
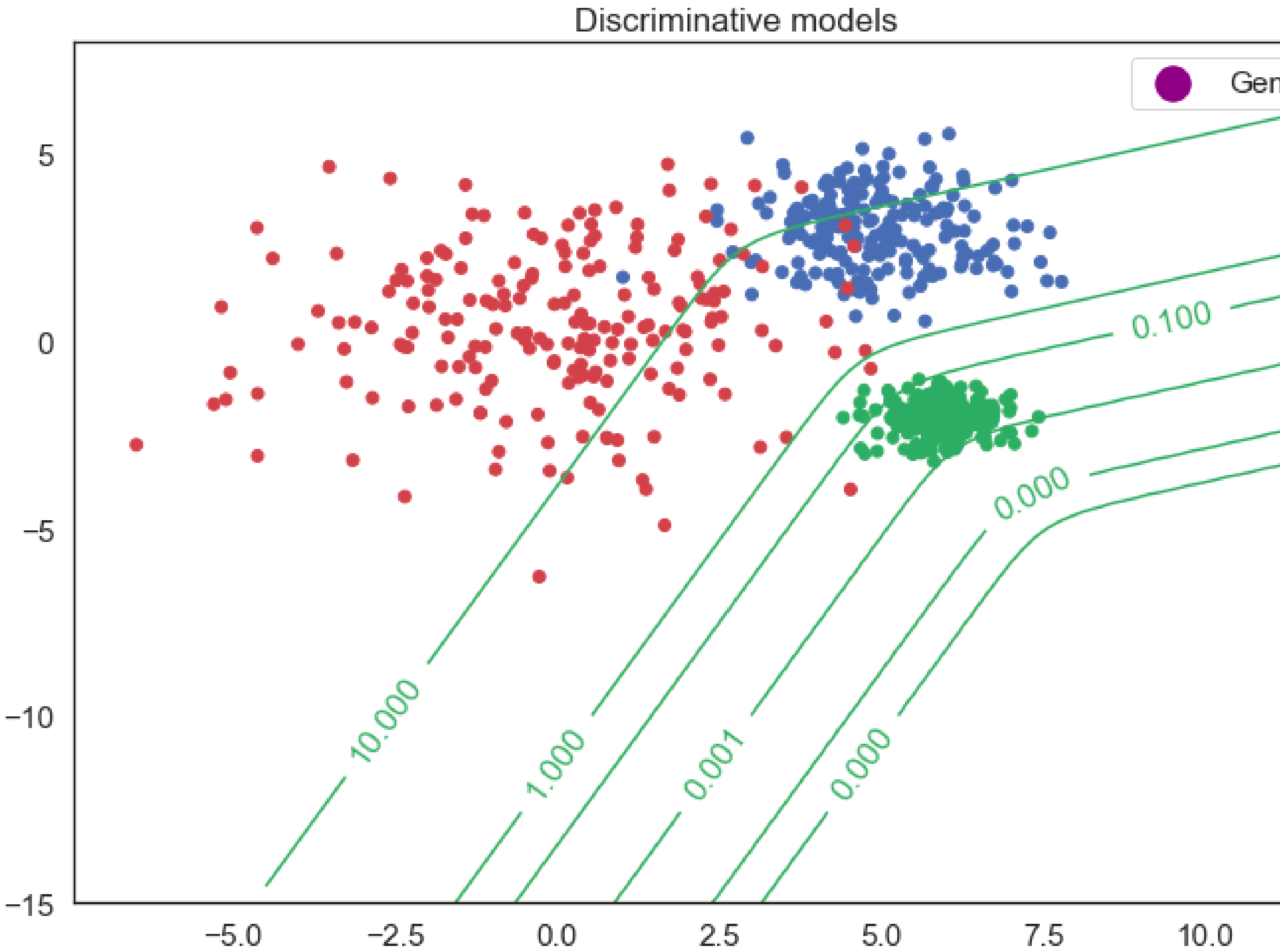
Генеративные модели. Часть 1.

Генеративные / дискриминативные модели

Ранее в курсе мы чаще всего сталкивались с классом моделей сталкивались с классом моделей, которые называются дискриминативные. Это означает что по некоторому x мы хотели предсказать вероятность какой-то метки y (Метка класса, сегментационная маска, bounding box). То есть учили распределение $p(y|x)$. Посмотрим на самый простой случай классификации на три класса.

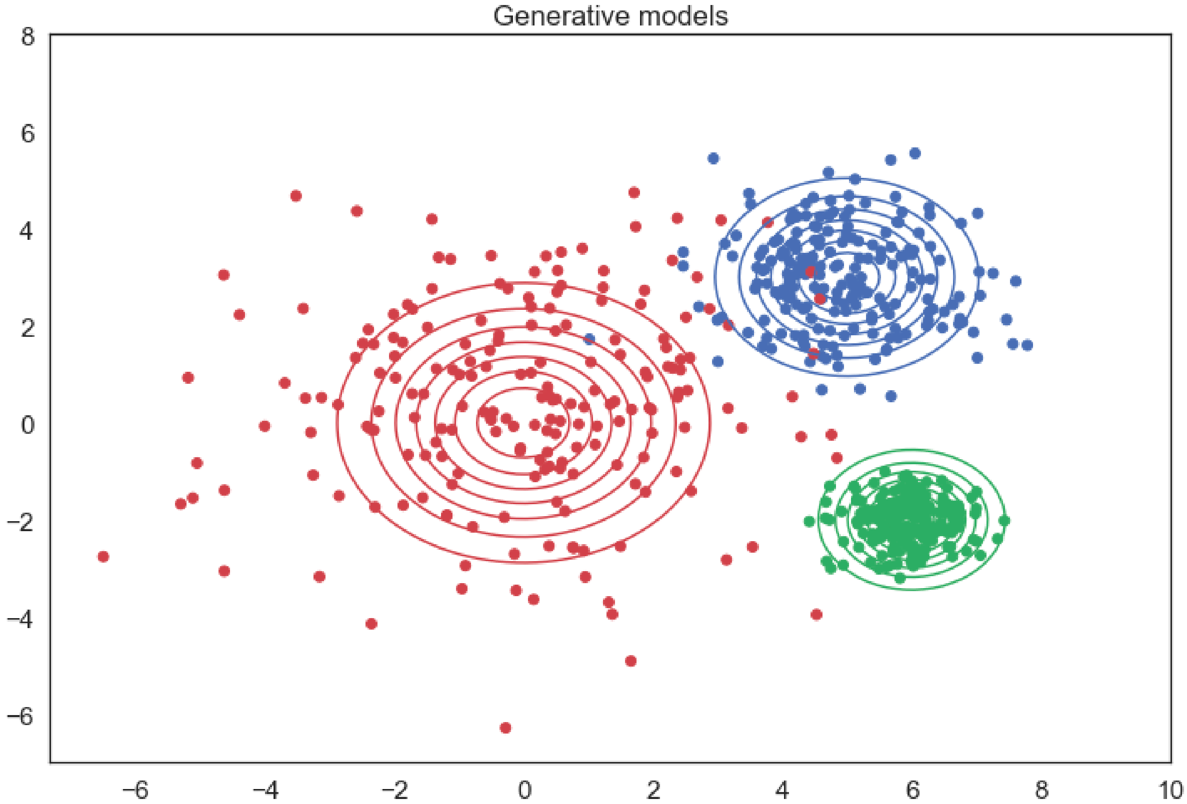


Теперь давайте рассмотрим следующую задачу: пусть мы хотим сгенерировать точку зеленого класса. Для этого давайте возьмем не $p(y|x)$, а $\log(p(y|x))$ и добавим минус (для градиентного спуска). На картинке ниже изображены уровни получившейся функции.



Фиолетовая точка — результат градиентного спуска. Как мы видим эта точка лежит далеко от кластера зеленых точек. И вряд ли может считаться хорошей сгенерированной точкой.

Для генерации нам необходим другой класс моделей, которые стараются выучить распределение, а не границу между классами:



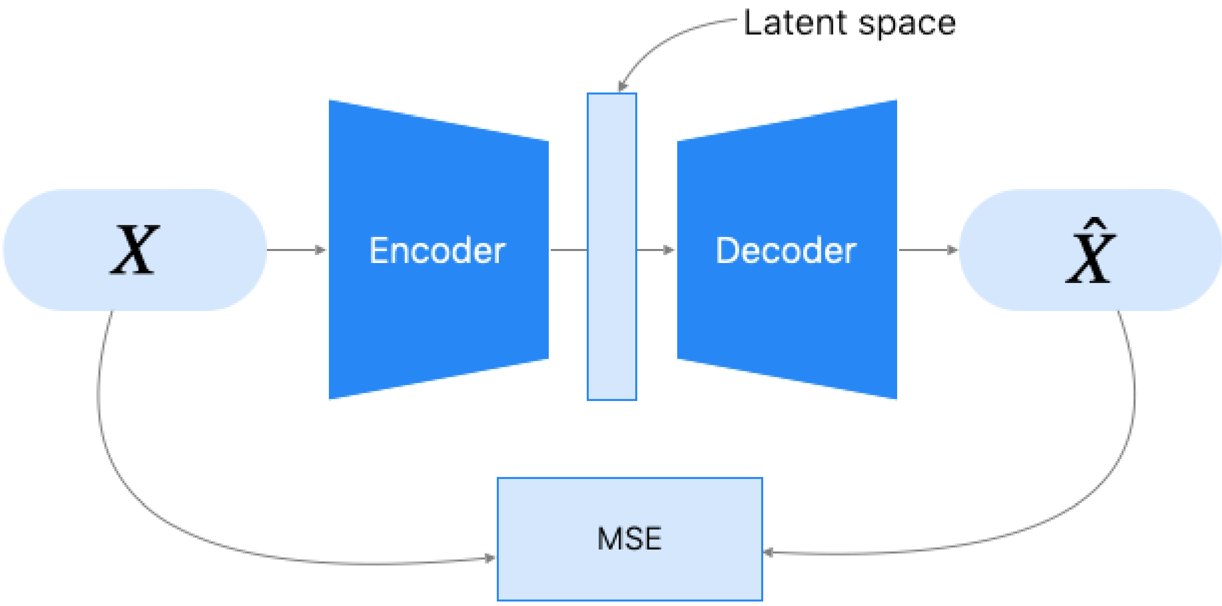
В примере выше форму и параметры распределения подобрать просто — видно что точки распределены нормально (многомерное нормальное распределние), но в реальной жизни такое случается редко. И иногда данные X распределены гораздо сложнее.

Однако, если предположить что существует функция f такая что величина $f(X)$ будет распределена нормально, причем как правило можно взять $f: R^n \to R^m,\ m < n$ (в случае одномерных данных обычно наоборот). Такое пространство размерности m называется латентным.

Однако для генерации нам необходимо найти и другую функцию g, такую, чтобы $g(f(x))$ снова была распределена как и исходные данные. Рассмотрим некоторые частные случаи такой модели:

Auto Encoder

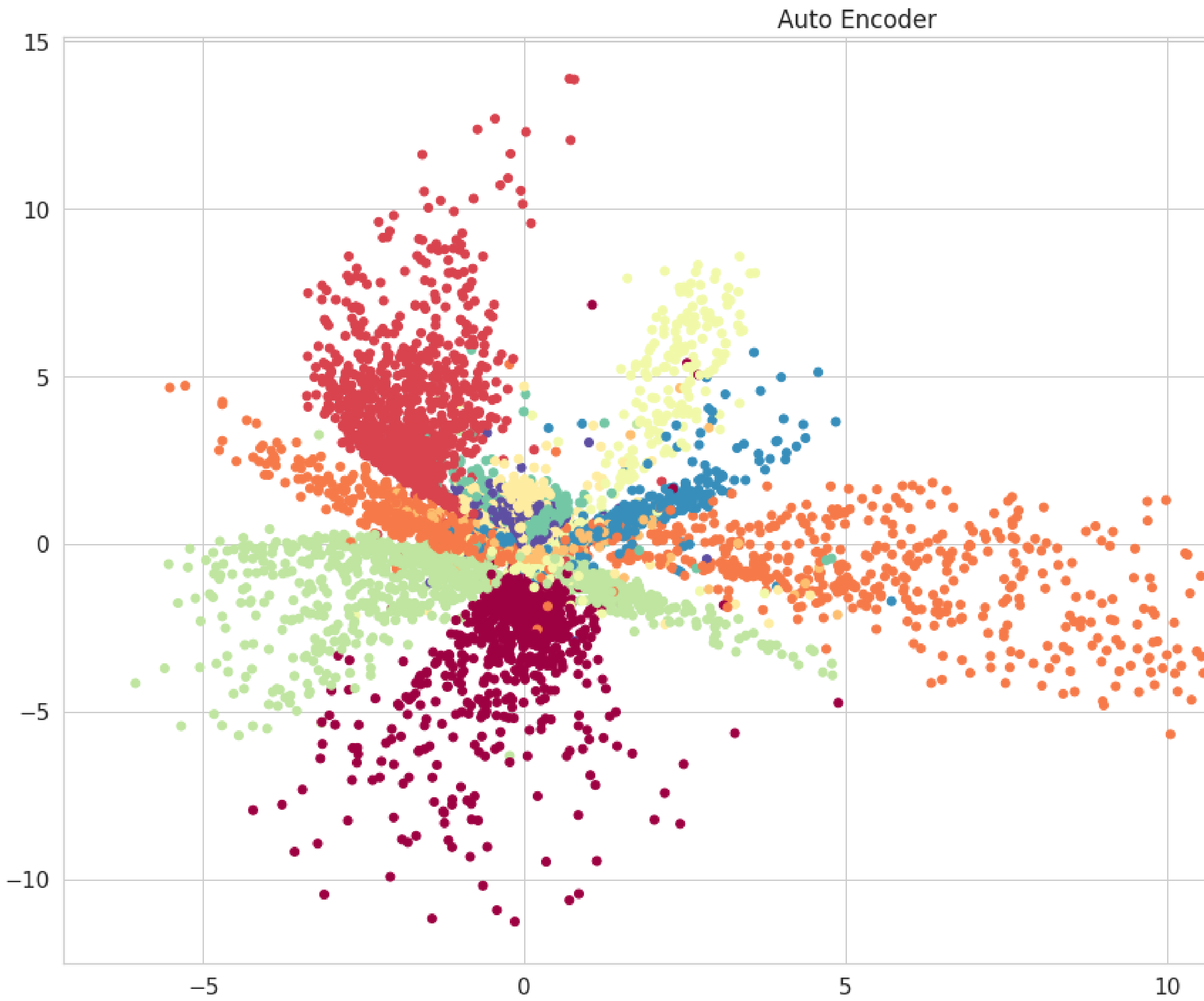
Давайте уберем требование о нормальном распределении величины в латентном пространстве. Оставим лишь то, что $\hat{X} = g(f(X))$ должно быть в том же пространстве, что и X. Тогда нам останется лишь сравнивать меру схожести X и $\hat{X}$. Здесь нам может подойти MSE. То есть при обучении мы минимизируем $MSE(g(f(X)), X) = MSE(\hat{X}, X)$.


Latent space
X
Encoder
Decoder
$\hat{X}$
MSE

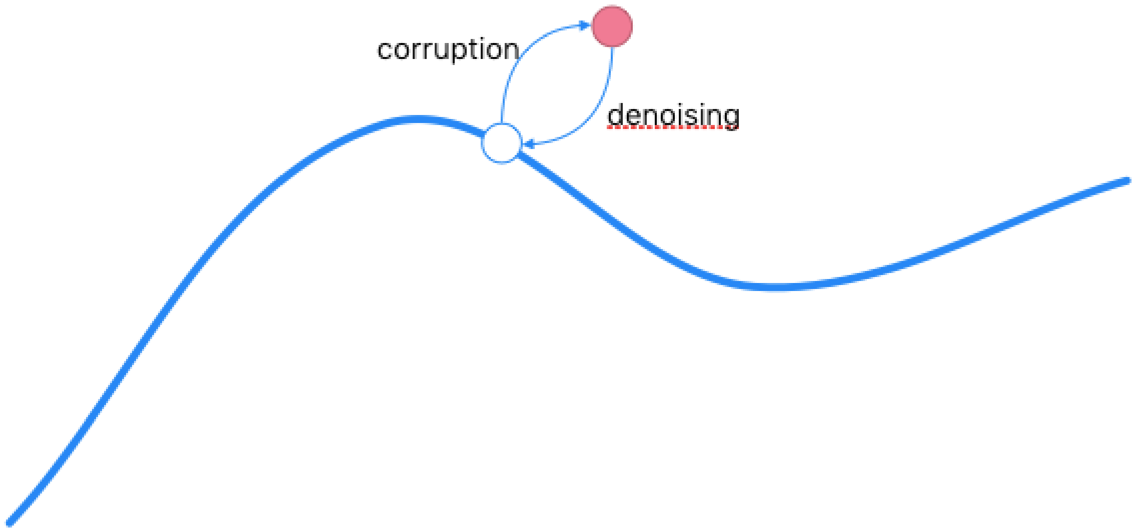
```python
class AE(nn.Module):
    def __init__(self, inp_dim, hidden_dim):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(inp_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, hidden_dim)
        )
        self.decoder = nn.Sequential(
            nn.Linear(hidden_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, inp_dim)
        )

    def forward(self, x):
        shapes = x.shape
        x = x.view(x.size(0), -1)
        return self.decoder(self.encoder(x)).view(*shapes)

    def encode(self, x):
        x = x.view(x.size(0), -1)
        return self.encoder(x)
```

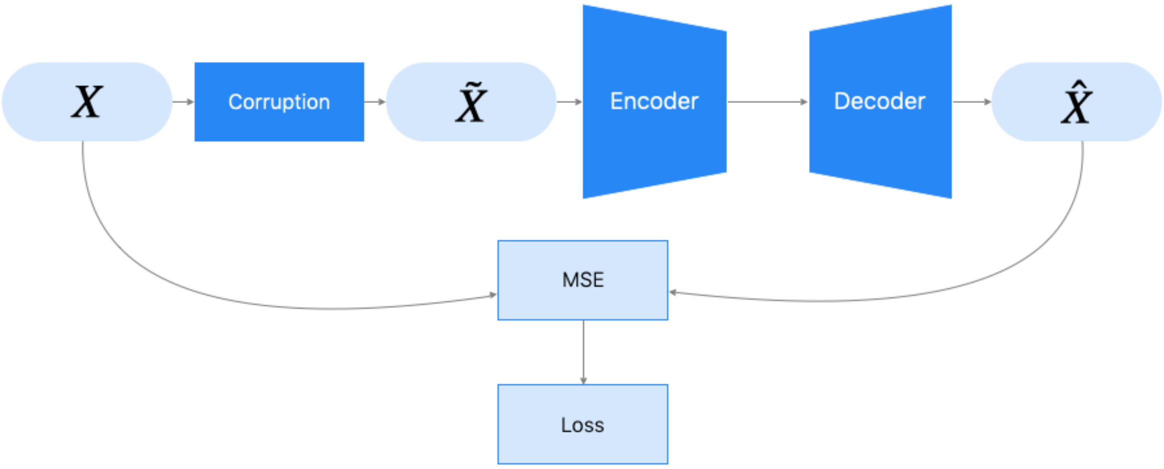
Зачем это нужно?

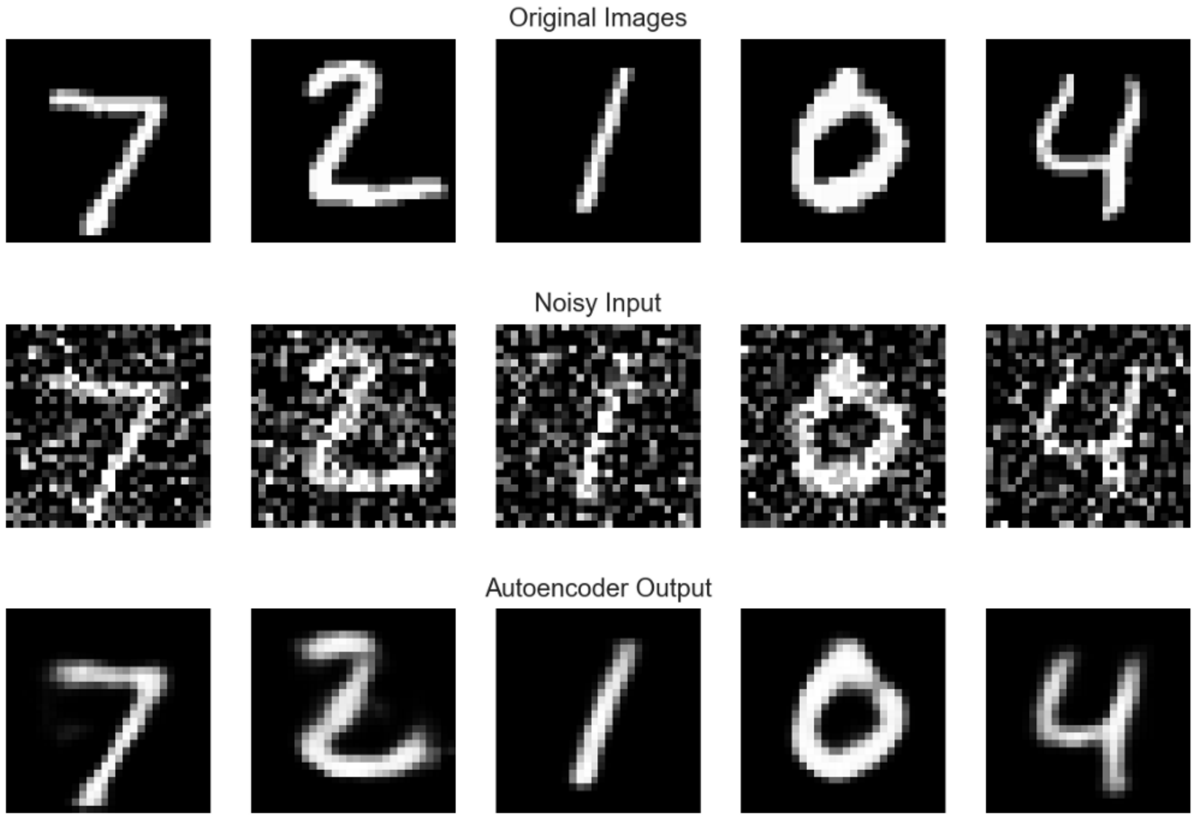
Понижение размерности



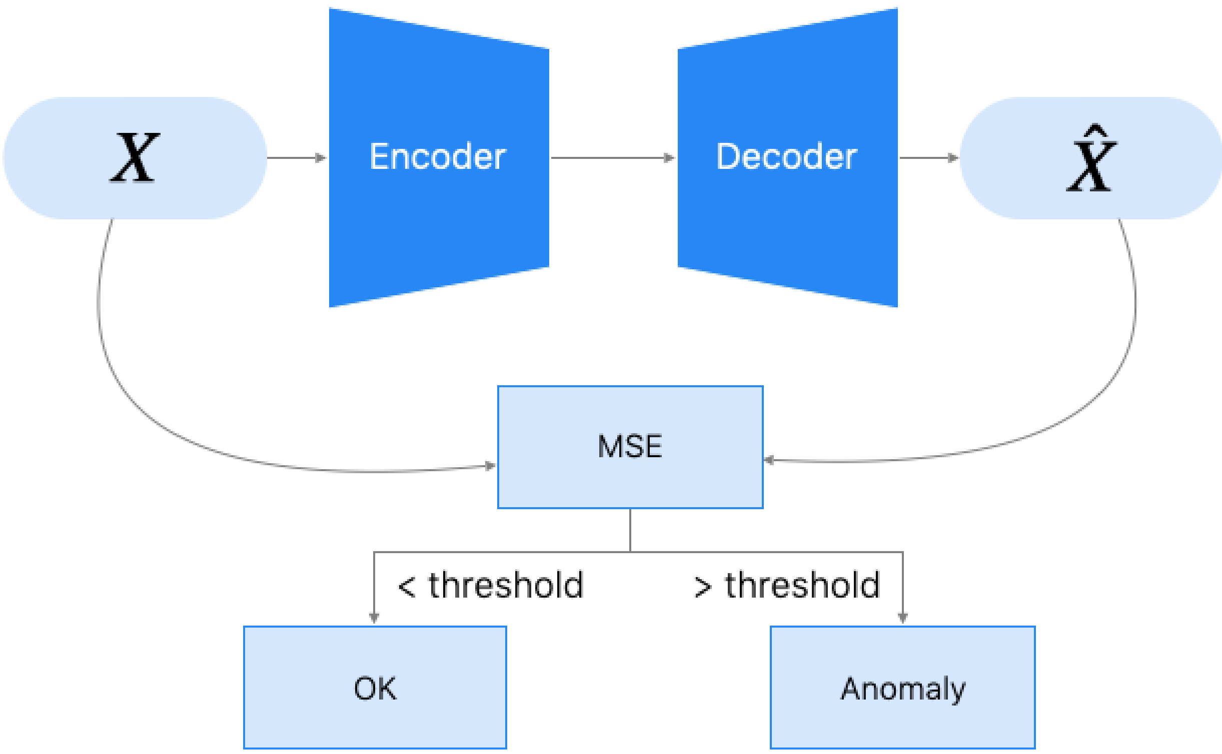
Избавление от шума. Denoising AE.


corruption
denoising


X
Corruption
$\tilde{X}$
Encoder
Decoder
$\hat{X}$
MSE
Loss

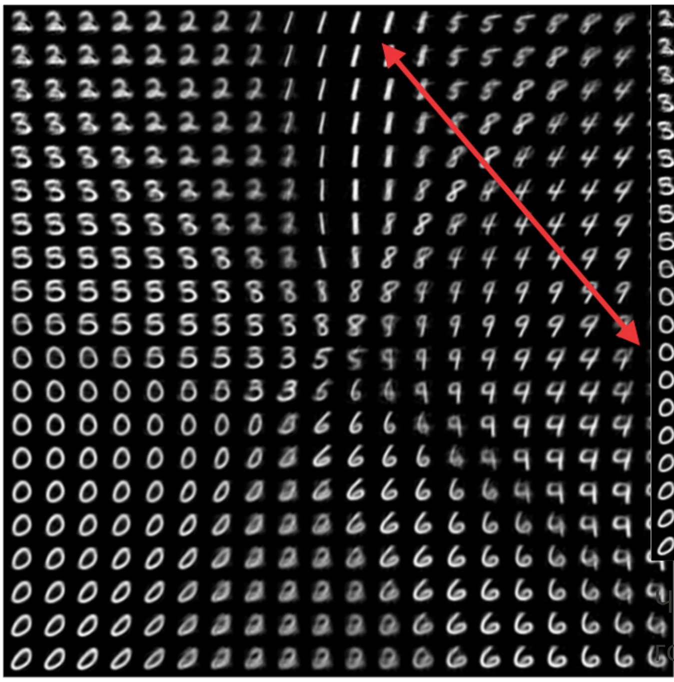


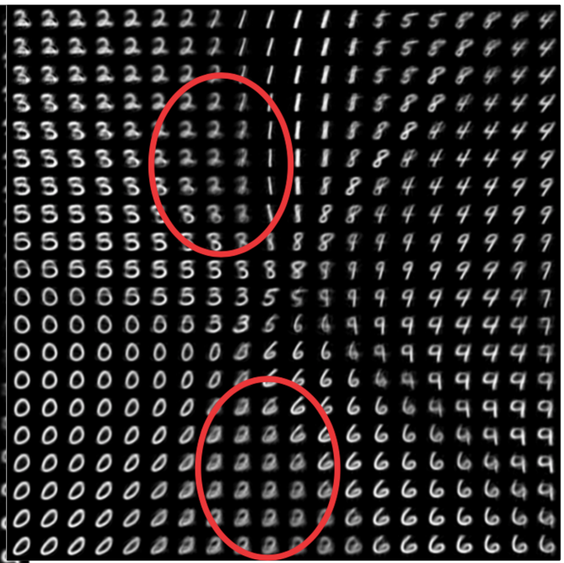
Детектирование аномалий.



Как генерировать?

Давайте рассмотрим латентное пространство AE подробнее. Возьмем точки из равномерной сетки $[-1.5, 1.5]\mathrm{x}[-1.5, 1.5]$ (квадрат) и пропустим их через декодер.





асть цифр неразличимы и вообще оворя не похожи на числа.

Несмотря на то, что цифры 7 и 1 похожи внешне, в латентном пространстве они располагаются далеко друг от друга. Это плохо.

Variational Auto Encoder

Вернемся к задаче генерации. Вспомним, что мы хотим генерировать точки из датасета. В случае с обычным AE генерировать не получится — мы ничего не знаем о распределении в латентном пространстве. Теперь давайте явно скажем, что $f(X)$ должна подчинятся нормальному распределению. Пусть тогда f(X) выдает не точку, а параметры этого распределения. Взглянем на алгоритм

1. $\mu, \Sigma = f(X)$
2. sample x_{hidden} from $N(\mu, \Sigma)$ (этот шаг подробнее описан здесь)
3. $\hat{X} = g(x_{\text{hidden}})$

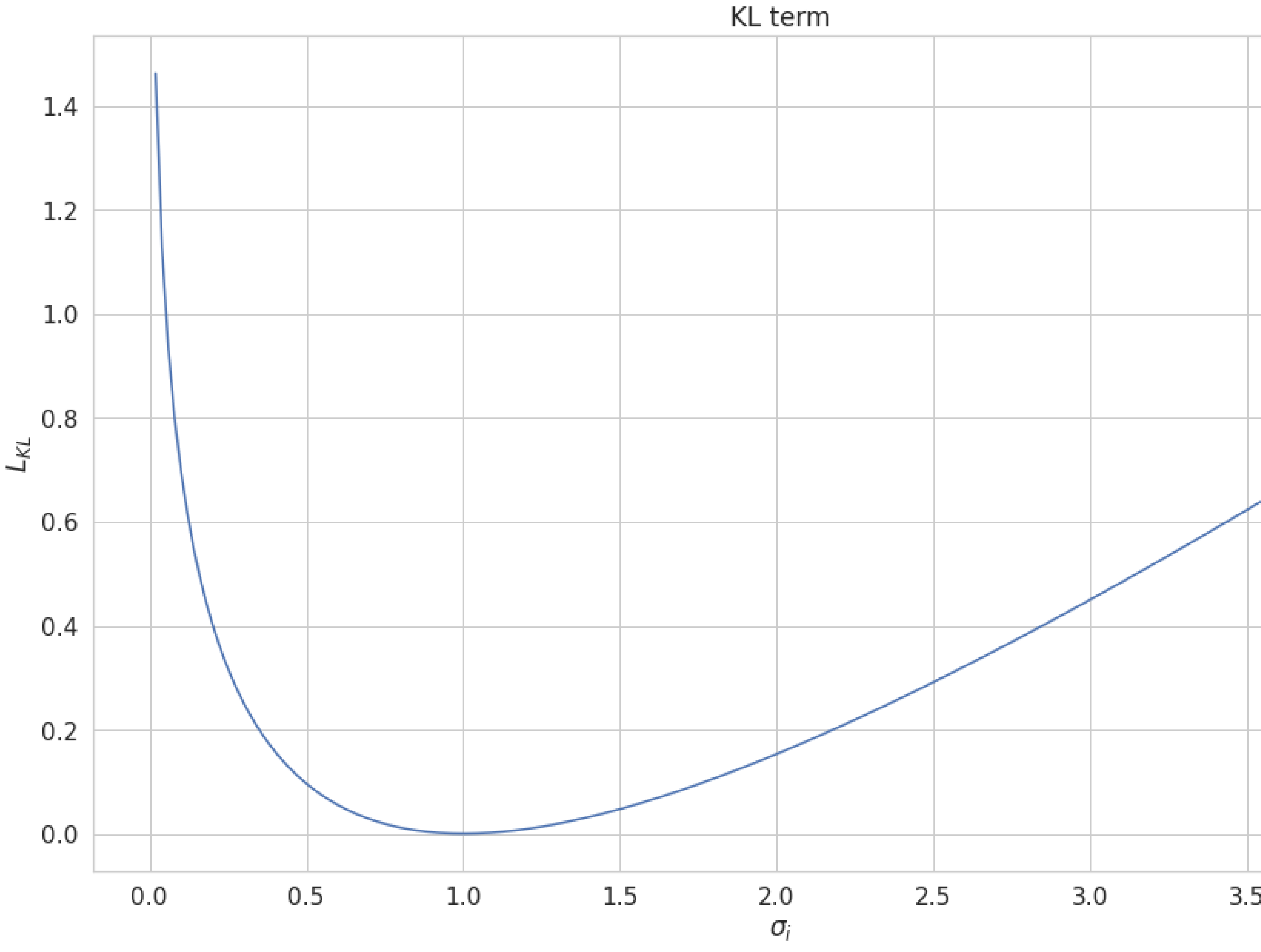
Если мы просто начнем обучать такую модель, то она сколапсирует в обычный AE приблизив дисперсию к нулю. Мы этого не хотим. Давайте стараться делать так, чтобы распределение $N(\mu, \Sigma)$ было максимально похоже на $N(\vec{0}, I)$.

Давайте будем считать матрицу Σ диагональной (компоненты не зависят друг от друга), то есть $\Sigma = \operatorname{diag}(\sigma_i)$. Рассмотрим i-ую компоненту отдельно. Напомню, мы хотим

чтобы $\mu_i = 0, \sigma_i = 1$ (стандартное распределение по каждой компоненте). Рассмотрим вот такую функцию:

$$L_{KL} = \frac{1}{2}\left(\sigma_i - \log\sigma_i - 1 + \mu_i^2\right)$$

При $\mu_i = 0$ получим:



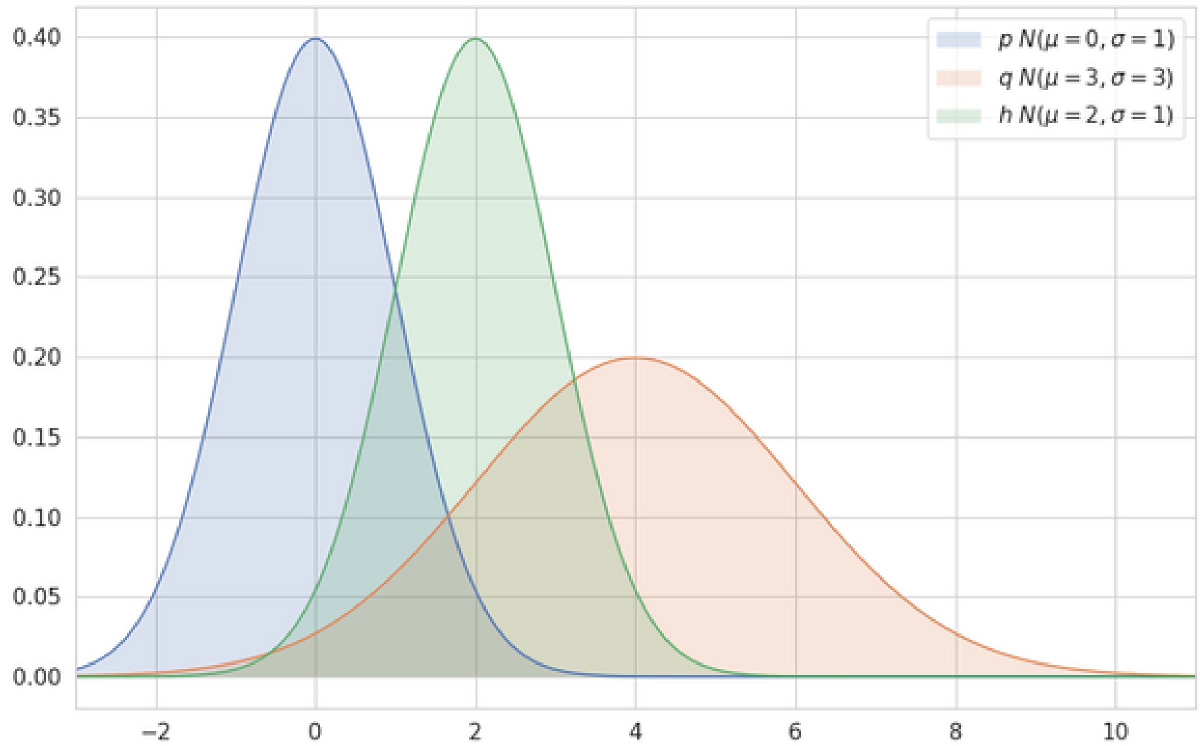
Как видно минимум такой функции как раз в точке $\sigma_i = 1$.

Такая функция взялась не из эмпирических соображений. Существует меры схожести двух распределений, которые называются дивергенциями. Отличия от обычного расстояния в том, что, вообще говоря никто не обязывает чтобы дивергенция не обязана быть симметричной. Те $D(p||q) \neq D(q||p)$, где $p(x)$ и $q(x)$— распределения.

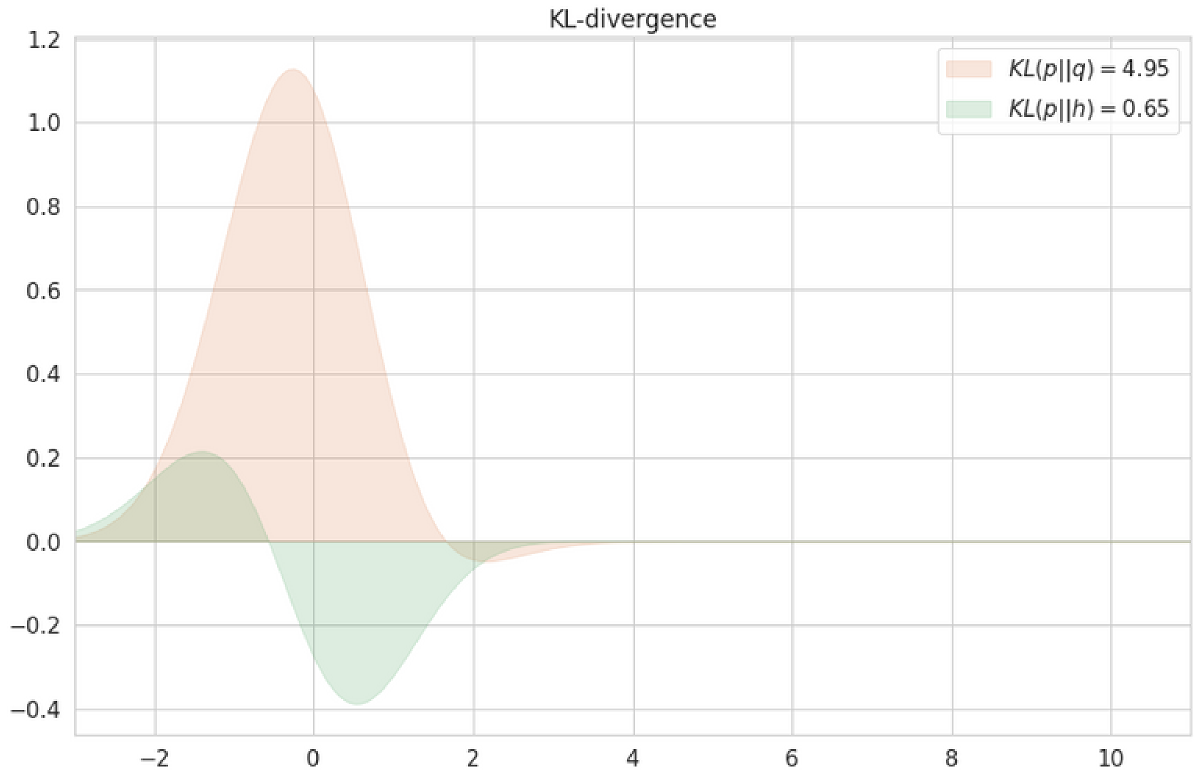
Самая распространенная из таких дивергенций это Kullback–Leibler divergence.

Она имеет вид:

$$KL(p||q) = \sum_{x \in \mathcal{X}} p(x) \log \left(\frac{q(x)}{p(x)} \right)$$



Рассмотрим три распределения — стандартное нормальное ($\mu_p = 0, \sigma_p = 1$), и еще два нормальных ($\mu_q = 3, \sigma_q = 3; \mu_h = 2, \sigma_h = 1$).



Посчитаем KL-дивергенцию из стандартного распределения p в распределение q и h. Как видно для q площадь для интегрирования получилось больше, а значит и дивергенция выше.

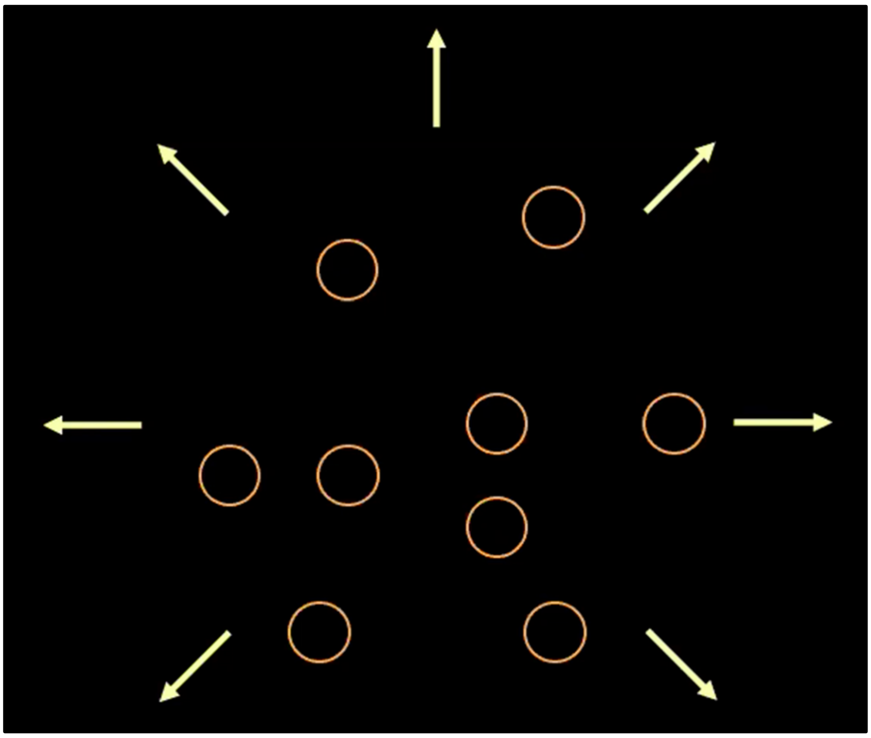
▶ Интуиция за KL дивергенцией

Можно показать что если p и q — распределены нормально, причем p — стандартное распределение ($\mu = 0$, $\sigma = 1$), то мы как раз получим формулу выше.

Соберем все вместе:

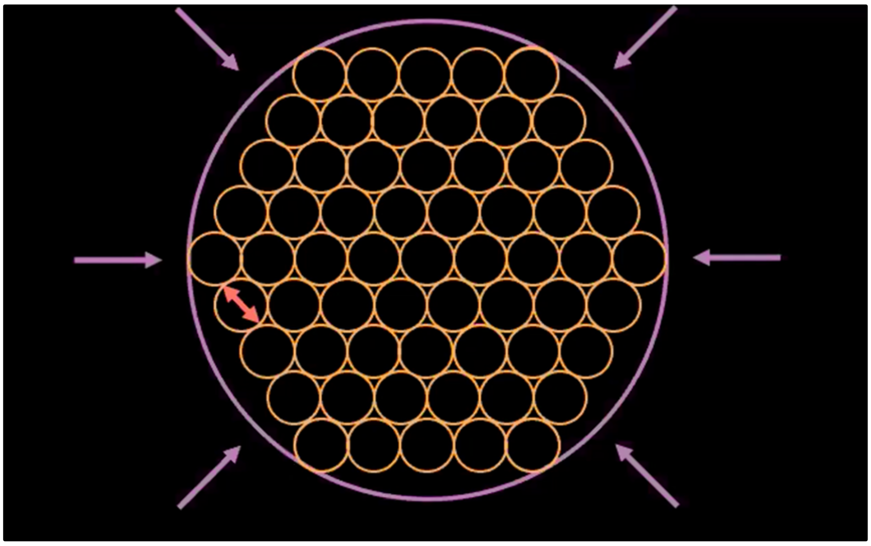
$$L = L_{rec} + \beta \cdot L_{KL}$$

L_{rec} — это обычный лосс от AE. Он стремится чтобы все точки были различимы и с маленькой дисперсией. Поэтому он их отталкивает друг от друга.



from https://atcold.github.io/pytorch-Deep-Learning/en/week08/08-3/

А L_{KL} это наша добавка. Она стремится, чтобы стандартное отклонение было равно 1, а математическое ожидание 0. Поэтому он как бы собирает шарики вместе и не дает шарикам схлопнуться в одну точку.



from https://atcold.github.io/pytorch-Deep-Learning/en/week08/08-3/

Reparametrization trick

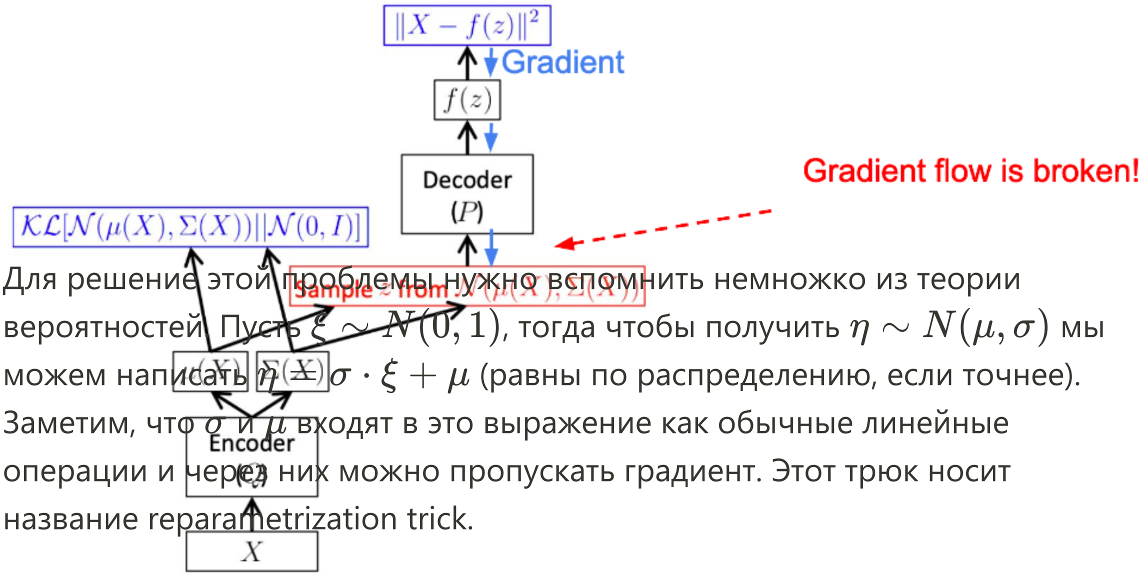
На словах все выглядит хорошо, остаются вопросы: как это все реализовать? Что значит сэмплировать и пропускать дальше?
Для того, чтобы получить параметры распределения энкодер должен иметь выходную размрность $2 \cdot \text{latent dim}$ так как по каждой компоненте у распределения два параметра — $\{\mu; \sigma\}$, для удобства второй параметр — σ сразу интерпретируют как $\log(\sigma)$; декодер принимать на вход просто latent dim:

```
class VAE(nn.Module): def __init__(self, inp_dim, latent_dim):
super().__init__() self.encoder = nn.Sequential( nn.Linear(inp_dim,
inp_dim//2), nn.ReLU(), nn.Linear(inp_dim//2, latent_dim * 2) # mu and
log(sigma) ) self.decoder = nn.Sequential( nn.Linear(latent_dim,
inp_dim//2), nn.ReLU(), nn.Linear(inp_dim//2, inp_dim) )
self.latent_dim = latent_dim
```

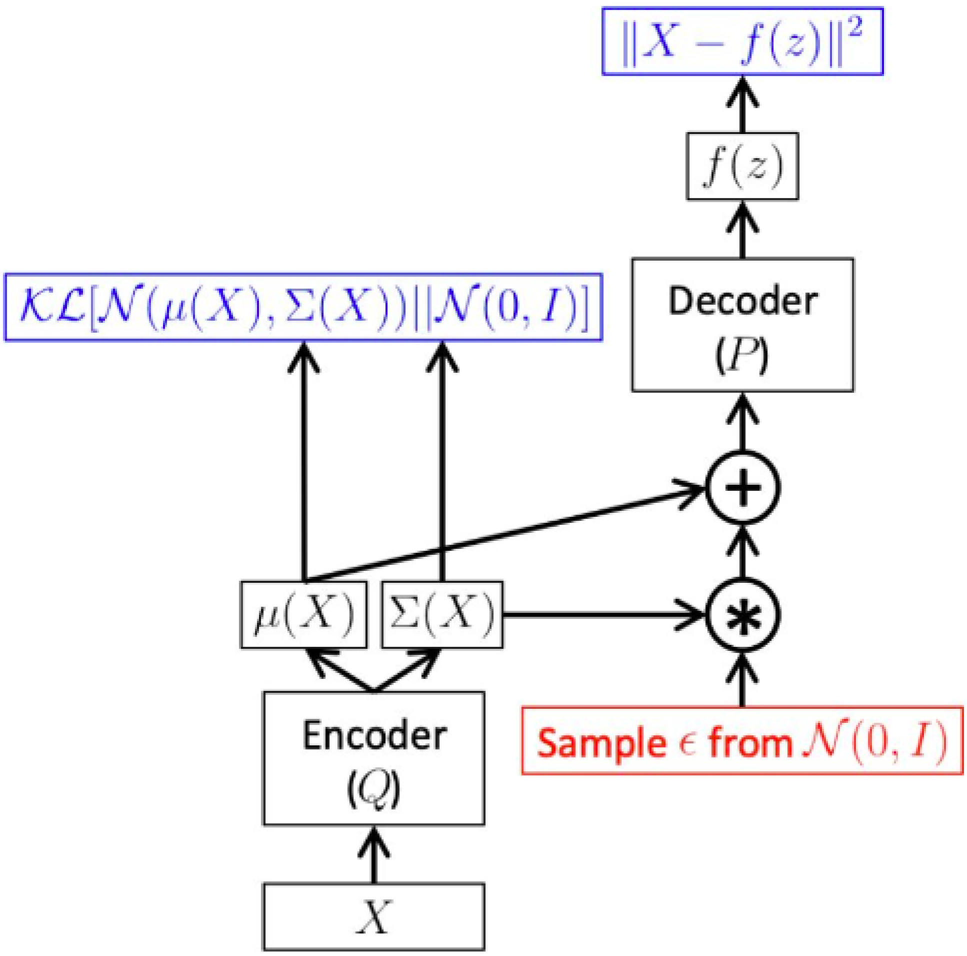
теперь нам нужно научится сэмплировать. Если мы просто возьмем параметры и напишем:

```
mu, log_sigma = torch.split(self.encoder(x), self.latent_dim) sample =
torch.normal(mu=mu, sigma=torch.exp(log_sigma))
```

То при вызове `.backward()` градиенты не пройдут сквозь `torch.normal`



Для решение этой проблемы нужно вспомнить немножко из теории вероятностей. Пусть $\xi \sim N(0, 1)$, тогда чтобы получить $\eta \sim N(\mu, \sigma)$ мы можем написать $\eta = \sigma \cdot \xi + \mu$ (равны по распределению, если точнее). Заметим, что σ и μ входят в это выражение как обычные линейные операции и через них можно пропускать градиент. Этот трюк носит название reparametrization trick.



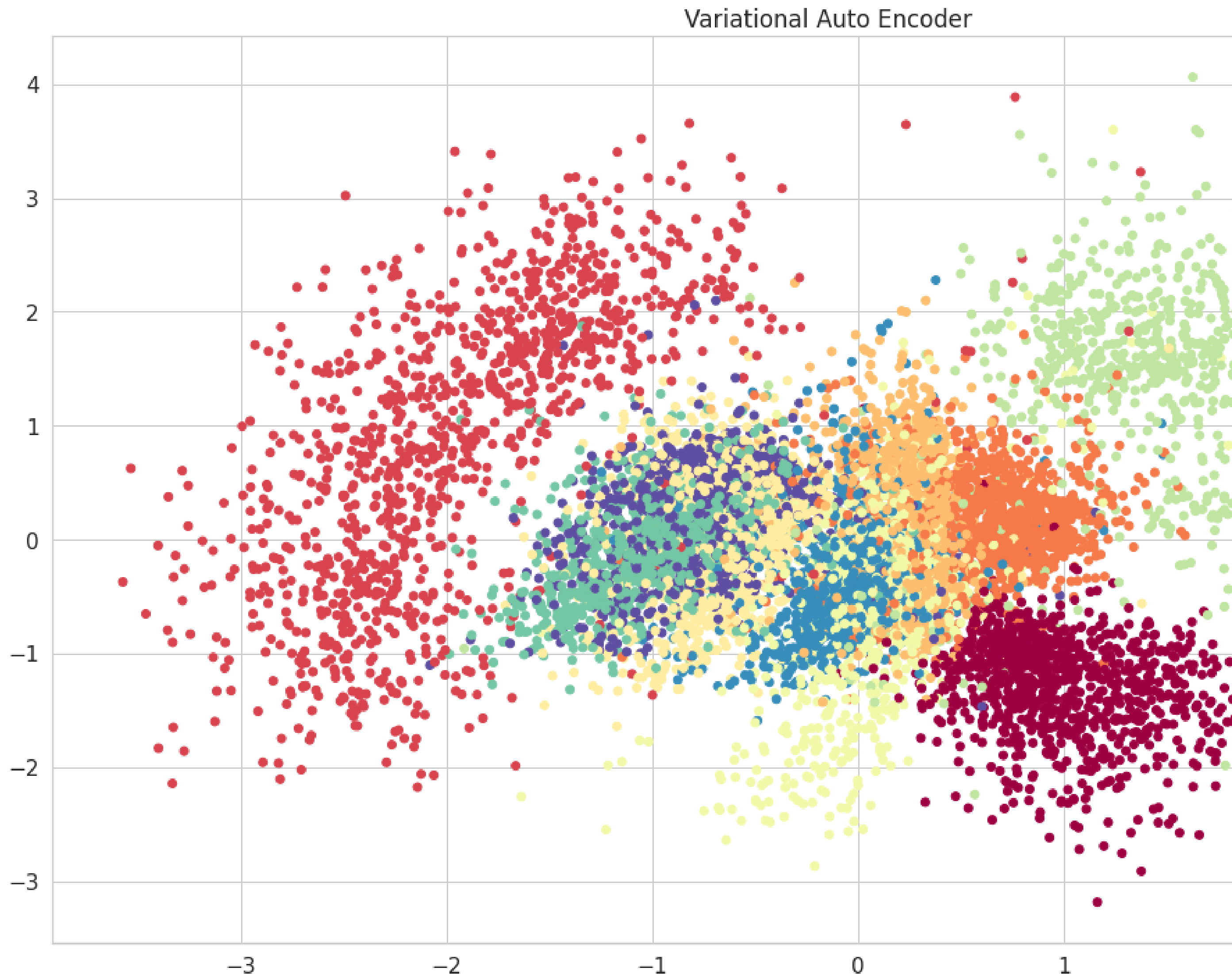
В коде это выглядит так:

```
mu, log_sigma = torch.split(self.encoder(x), self.latent_dim) sample = torch.exp(log_sigma) * torch.randn(batch_size, latent_dim) + mu
```

В итоге, собрав все вместе:

```
class VAE(nn.Module): def __init__(self, inp_dim, latent_dim): super().__init__() self.encoder = nn.Sequential( nn.Linear(inp_dim, inp_dim//2), nn.ReLU(), nn.Linear(inp_dim//2, latent_dim * 2) # mu and log(sigma) ) self.decoder = nn.Sequential( nn.Linear(latent_dim, inp_dim//2), nn.ReLU(), nn.Linear(inp_dim//2, inp_dim) ) self.latent_dim = latent_dim def forward(self, x): shapes = x.shape mu, log_sigma = torch.split(self.encoder(x), self.latent_dim) kl_loss = self.kl_loss(mu, log_sigma) # implement it yourself sample = torch.exp(log_sigma) * torch.randn(shapes[0], self.latent_dim) + mu out = self.decoder(sample) return out.view(*shapes), kl_loss # pseudo code vae = VAE(768, 2) optimizer = ... criterion = nn.MSELoss() for data in dataloader: optimizer.zero_grad() out, loss_kl = vae(data) rec_loss = criterion(out, data) loss = rec_loss + beta * loss_kl loss.backward() optimizer.step()
```

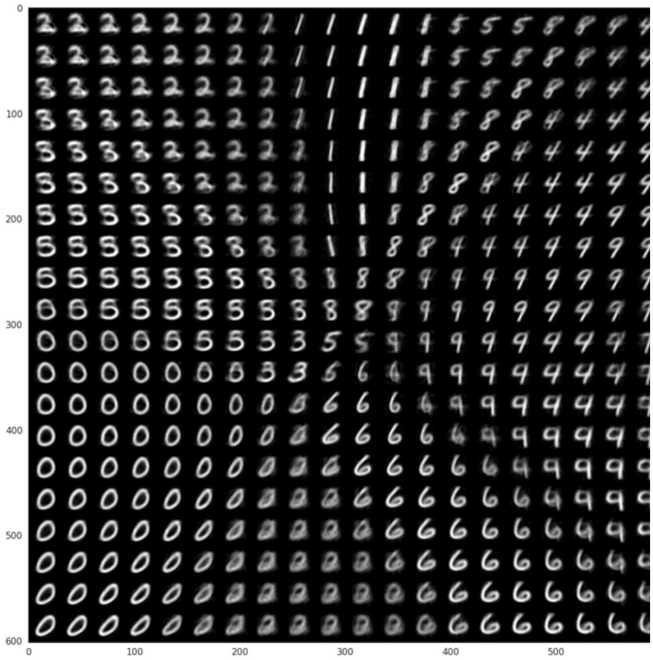
В итоге латентное пространство выглядит так.



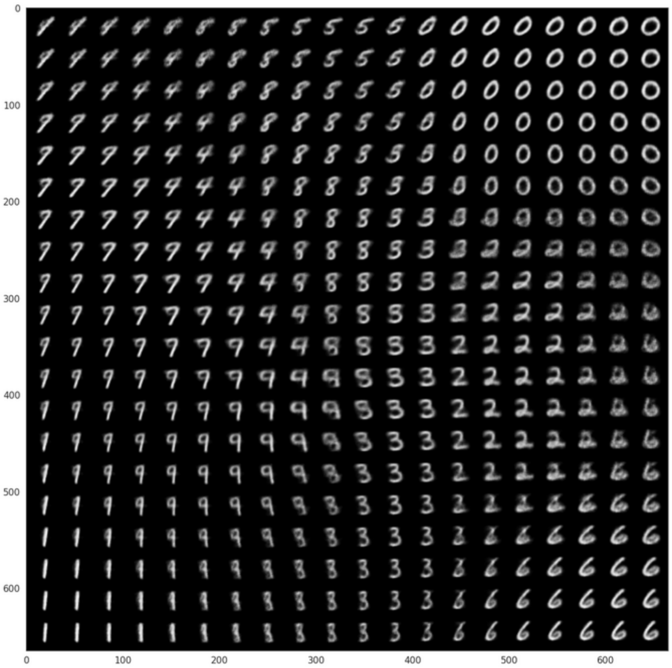
И теперь, мы можем сэмплировать из латентного пространства.

Левая картинка — сэмпл из AE. В нем переходы между картинками резкие, половина картинок не похожи на цифры.

Правая картинка — сэмпл из VAE. Переходы плавные и все картинки это цифры.



AE latents



VAE latents