Глубокое обучение

Тема 2. Вариационные автокодировщики

Импортируем необходимые библиотеки:

In [1]:
```
import os

os.environ["KERAS_BACKEND"] = "tensorflow"            # "jax" "tens
#os.environ["ENABLE_PJRT_COMPATIBILITY"] = "1"   # added for jax
#os.environ["PYTORCH_ENABLE_MPS_FALLBACK"] = "1" # added for torch
```

In [2]:
```
from silence_tensorflow import silence_tensorflow
silence_tensorflow()
```

In [3]:
```
import keras
from keras import models,layers,datasets,callbacks
keras.__version__, keras.backend.backend()
```

Out[3]:
```
('3.9.2', 'tensorflow')
```

Автокодировщики

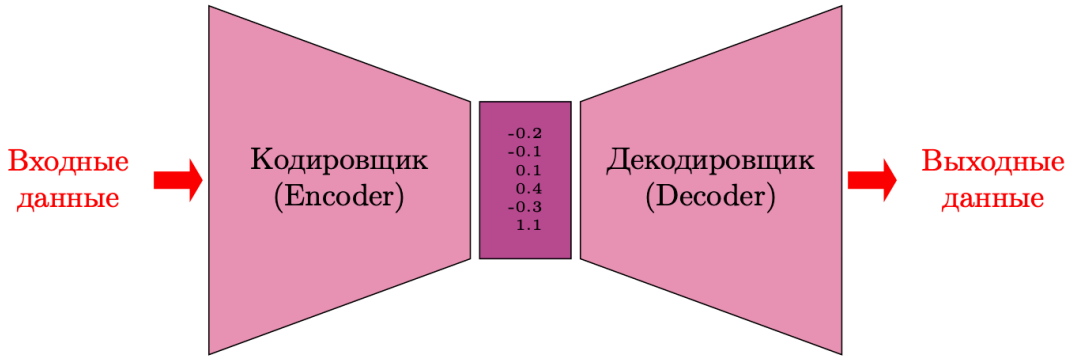
Нейронные сети обычно используются при обучении с учителем (supervised learning). Это означает, что для каждой точки обучающего набора данных $\mathbf{x}_i$ имеется отклик y_i. Во время обучения модель нейронной сети изучает взаимосвязь между входными данными и откликами.

Теперь предположим, что имеются только немаркированные данные, то есть у нас есть только обучающий набор данных $\mathbf{D}$, состоящий из n точек $\mathbf{x}_i \in \mathbb{R}^d$, где $i = 1, \ldots, n$.

Автокодировщики (автоэнкодеры, autoencoders) были введены Румелхартом, Хинтоном и Уильямсом в 1986 году с целью *научиться восстанавливать входные наблюдения $|mathbf{x}*{i}$* с минимально возможной ошибкой_.

Автокодировщик — это тип алгоритма, основной целью которого является обучение *информативному* представлению данных, которое можно использовать для различных приложений, путем обучения достаточно хорошей реконструкции набора входных данных.

Чтобы лучше понять автокодировщики, обратимся к их типичной архитектуре, представленной ниже.



Основными компонентами **автокодировщика** являются:

- **кодировщик**
- **латентное пространство**
- **декодер**.

Кодировщик (энкодер) и декодер — это функции, обычно представленные нейронными сетями, а латентное пространство – это представление скрытых (латентных) признаков, которое является выходом кодировщика и входом декодера. Вообще говоря, требуется, чтобы автокодировщик достаточно хорошо реконструировал входные данные, при этом он должен создавать скрытое представление, которое является полезным и содержательным.

Обозначим энкодер и декодер в составе автокодировщика через f (энкодер) и g (декодер). Энкодер f переводит входной сигнал x в его представление (код) z:

$$z = f(x),$$

а декодер g восстанавливает сигнал x по его коду z:

$$x = g(z).$$

Автокодировщик, изменяя функции f и g, стремится обучиться тождественной функции $x = g(f(x))$, минимизируя какой-либо функционал ошибки $L(x, g(f(x)))$.

При этом семейства функций для энкодера f и декодера g каким-либо образом ограничивают, чтобы автоэнкодер был вынужден отбирать наиболее важные свойства входных данных.

Метод главных компонент

Метод главных компонент (Principal Component Analysis, PCA) – это метод, позволяющий найти пространственный базис, который наилучшим образом отражает дисперсию в наборе данных.

Допустим, что данные уже центрированы путем вычитания средних значений: $\mathbf{x} \leftarrow \mathbf{x} - \mathbb{E}[\mathbf{x}]$.

Первая главная компонента (направление с максимальной дисперсией) $\mathbf{w}_1^*$ находится путем максимизации дисперсии данных, умноженных на вектор $\mathbf{w}_1$ единичной длины:

$$y_1 = \mathbf{x}^T \mathbf{w}_1,\ \|\mathbf{w}_1\| = 1 \Rightarrow \mathbb{E}\left[y_1^2\right] = \mathbb{E}\left[\mathbf{w}_1^T \mathbf{x}\mathbf{x}^T \mathbf{w}_1\right] = \mathbf{w}_1^T \mathbb{E}\left[\mathbf{x}\mathbf{x}^T\right] \mathbf{w}_1 = \mathbf{w}_1^T \mathbf{\Sigma}$$

Отсюда получаем, что решение $\mathbf{w}_1^*$ задается собственным вектором ковариационной матрицы $\mathbf{\Sigma}_\mathbf{x}$, соответствующим наибольшему собственному значению матрицы $\mathbf{\Sigma}_\mathbf{x}$:

$$\mathbf{w}_1^* = \underset{\mathbf{w}_1:\, \|\mathbf{w}_1\|=1}{\arg\max}\, \mathbf{w}_1^T \mathbf{\Sigma}_\mathbf{x} \mathbf{w}_1$$

Вторая главная компонента находится как направление с максимальной дисперсией при условии ортогональности направлению первой главной компоненты (и т.д.).

Пусть построено m-мерное подпространство, порожденное ортонормальным базисом из m главных компонент $\mathbf{w}_1, \ldots, \mathbf{w}_m$:

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1 & \ldots & \mathbf{w}_m \end{bmatrix},\ \mathbf{W}^T \mathbf{W} = \mathbf{I}_m$$

Можно спроектировать d-мерные векторы данных $\mathbf{x}$ на это подпространство:

$$\mathbf{z} = \mathbf{W}^T \mathbf{x}$$

Реконструкция $\mathbf{x}$, которая остается внутри m-мерного подпространства, порожденного $\mathbf{W}$, равна

$$\hat{\mathbf{x}} = \mathbf{W}\mathbf{z} = \mathbf{W}\mathbf{W}^T \mathbf{x}$$

Поэтому мы находим такую матрицу $\mathbf{W}$, что среднеквадратическая ошибка между исходными данными $\mathbf{x}$ и реконструированным вектором $\hat{\mathbf{x}}$ сведена к минимуму:

$$\mathbf{W}_{PCA} = \underset{\mathbf{W}:\, \mathbf{W}^T\mathbf{W} = \mathbf{I}_m}{\arg\min}\ \mathbb{E}\left[\left\| x - \underbrace{\mathbf{W}\mathbf{W}^T \mathbf{x}}_{\hat{\mathbf{x}}} \right\|^2\right],$$

поэтому метод PCA может быть интерпретирован как сжатие с минимальной среднеквадратичной ошибкой (MSE).

Метод PCA как автокодировщик со сжатием

Метод PCA может рассматриваться как автокодировщик. Будем обучать

отображение из $\mathbf{x}$ в $\hat{\mathbf{x}}$:

$$\hat{\mathbf{x}} = g\left(f\left(\mathbf{x}\right)\right),$$

где:

- $f\left(\mathbf{x}\right) = \mathbf{W}_f\,\mathbf{x} + \mathbf{b}_f$ – линейный кодировщик
- $g\left(\mathbf{z}\right) = \mathbf{W}_g\,\mathbf{z} + \mathbf{b}_g$ – линейный декодер
- $\mathcal{L}\left(\mathbf{x}\right) = \mathbb{E}\left[\left\|\mathbf{x} - g\left(f\left(\mathbf{x}\right)\right)\right\|^2\right]$ – функция потерь

Если мы никак не ограничиваем линейные функции f и g, то мы рискуем изучить тривиальное тождественное отображение:

$$\hat{\mathbf{x}} = g\left(f\left(\mathbf{x}\right)\right) = \mathbf{W}_g\,\left(\mathbf{W}_f\,\mathbf{x} + \mathbf{b}_f\right) + \mathbf{b}_g = \left(\mathbf{W}_g\,\mathbf{W}_f\right)\,\mathbf{x} + \left(\mathbf{W}_g\,\mathbf{b}_f + \mathbf{b}_g\right) =$$

$$\Longrightarrow \mathbf{W}_g = \mathbf{W}_f^{-1},\, \mathbf{b}_g = -\mathbf{W}_g\,\mathbf{b}_f$$

Если размерность вектора $\mathbf{z}$ меньше размерности вектора $\mathbf{x}$, то автокодирование полезно, так как мы сжимаем данные.

Таким образом, метод PCA может быть реализован с помощью автокодировщика со сжатием.

Как можно улучшить сжатие, чтобы получить меньшую ошибку реконструкции

$$\mathcal{L}\left(\mathbf{x}\right) = \mathbb{E}\left[\left\|\mathbf{x} - g\left(f\left(\mathbf{x}\right)\right)\right\|^2\right]$$

со слоем сужения того же размера?

Мы можем использовать нелинейные кодировщик f и декодер g такие, что:

- и кодировщик, и декодер являются **глубокими нейронными сетями**
- критерием оптимизации является **среднеквадратическая ошибка реконструкции**:

$$\boldsymbol{\theta}_f,\, \boldsymbol{\theta}_g = \underset{\boldsymbol{\theta}_f,\, \boldsymbol{\theta}_g}{\arg\min} \mathbb{E}\left[\left\|\mathbf{x} - g\left(f\left(\mathbf{x}; \boldsymbol{\theta}_f\right); \boldsymbol{\theta}_g\right)\right\|^2\right]$$

- чтобы предотвратить изучение тривиальной (тождественной) функции f, мы используем латентное пространство $\mathbf{z}$ с меньшим количеством измерений (**слой сужения**).

Автокодировщик обучен восстанавливать изображение после того, как оно прошло через кодировщик и обратно через декодер. На первый

взгляд это может показаться странным: зачем восстанавливать набор изображений, которые у вас уже есть? Однако, как мы увидим, именно **пространство эмбеддинга** (также называемое скрытым или **латентным пространством**) является весьма интересной частью автокодировщика, поскольку выборка из этого пространства позволит нам генерировать новые изображения.

Давайте сначала определим, что подразумевается под вложением (эмбеддингом). **Эмбеддинг** $(\mathbf{z})$ представляет собой сжатие исходного изображения в скрытое пространство меньшей размерности. Идея состоит в том, что, выбрав любую точку в скрытом пространстве, мы можем генерировать новые изображения, пропуская эту точку через декодер, поскольку декодер научился преобразовывать точки в скрытом пространстве в жизнеспособные изображения.

Пример автокодировщика для набора Fashion-MNIST

Рассмотрим в качестве примера создание и обучение автокодировщика на наборе данных **Fashion-MNIST**.

Набор данных Fashion-MNIST — коллекция изображений предметов одежды в оттенках серого, каждое размером 28 × 28 пикселей.

In [14]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [15]:
```
def display(
    images, n=10, size=(20, 3), cmap="gray_r", as_type="float32", sa
):
    """
    Displays n random images from each one of the supplied arrays.
    """
    if images.max() > 1.0:
        images = images / 255.0
    elif images.min() < 0.0:
        images = (images + 1.0) / 2.0

    plt.figure(figsize=size)
    for i in range(n):
        _ = plt.subplot(1, n, i + 1)
        plt.imshow(images[i].astype(as_type), cmap=cmap)
        plt.axis("off")

    if save_to:
        plt.savefig(save_to)
        print(f"\nSaved to {save_to}")

    plt.show()
```

Параметры

В этом примере мы встроим изображения в двумерное скрытое пространство $\mathbb{R}^2$. Это поможет нам визуализировать скрытое пространство, поскольку можно легко построить точки в 2D (на плоскости). На практике скрытое пространство автокодировщика обычно имеет более двух измерений, чтобы иметь больше свободы для улавливания большего количества нюансов изображений.

In [16]:
```
IMAGE_SIZE = 32
CHANNELS = 1
BATCH_SIZE = 100
BUFFER_SIZE = 1000
VALIDATION_SPLIT = 0.2
EMBEDDING_DIM = 2
EPOCHS = 3
```

Подготовка данных

In [17]:
```
# Load the data
(x_train, y_train), (x_test, y_test) = datasets.fashion_mnist.load_
```

Эти готовые изображения размером 28 × 28 в оттенках серого (значения пикселей от 0 до 255) нам необходимо предварительно обработать, чтобы гарантировать, что значения пикселей масштабируются от 0 до 1. Также дополним каждое изображение до размера 32 × 32 для удобства манипулирования формой тензора при его прохождении через нейронную сеть.

In [18]:
```
# Preprocess the data

def preprocess(imgs):
    """
    Normalize and reshape the images
    """
    imgs = imgs.astype("float32") / 255.0
    imgs = np.pad(imgs, ((0, 0), (2, 2), (2, 2)), constant_values=0
    imgs = np.expand_dims(imgs, -1)
    return imgs


x_train = preprocess(x_train)
x_test = preprocess(x_test)
```

In [19]:
```
# Show some items of clothing from the training set
display(x_train)
```

In [20]:
```
x_train.shape
```

Out[20]:
```
(60000, 32, 32, 1)
```

Построение автокодировщика

В автокодировщике работа кодировщика состоит в том, чтобы взять входное изображение и сопоставить ему вектор эмбеддинга в скрытом пространстве. Чтобы добиться этого, мы сначала создаем входной слой для изображения и последовательно пропускаем его через три слоя `Conv2D`, каждый из которых фиксирует все более высокоуровневые признаки. Используется шаг 2, чтобы уменьшить вдвое размер выхода каждого слоя, одновременно увеличивая количество каналов. Последний сверточный слой сглаживается и соединяется с плотным слоем размера 2, который представляет наше двумерное скрытое пространство.

In [21]:
```
# Encoder
encoder_input = layers.Input(
    shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding=
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding=
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", paddin
shape_before_flattening = x.shape[1:]  # the decoder will need this

x = layers.Flatten()(x)
encoder_output = layers.Dense(EMBEDDING_DIM, name="encoder_output")

encoder = models.Model(encoder_input, encoder_output)
encoder.summary()
```

Model: "functional_5"

Layer (type)	Output Shape	
encoder_input (InputLayer)	(None, 32, 32, 1)	
conv2d (Conv2D)	(None, 16, 16, 32)	
conv2d_1 (Conv2D)	(None, 8, 8, 64)	
conv2d_2 (Conv2D)	(None, 4, 4, 128)	
flatten (Flatten)	(None, 2048)	
encoder_output (Dense)	(None, 2)	

Total params: 96,770 (378.01 KB)
Trainable params: 96,770 (378.01 KB)
Non-trainable params: 0 (0.00 B)

Декодер является зеркальным отражением кодировщика — вместо сверточных слоев используются сверточные транспонированные слои.

Стандартные сверточные слои позволяют нам уменьшить вдвое размер входного тензора в обоих измерениях (по высоте и ширине), установив параметр `strides = 2`. Слой сверточного транспонирования использует тот же принцип, что и стандартный сверточный слой (пропускание фильтра по изображению), но отличается тем, что установка `strides = 2` удваивает размер входного тензора в обоих измерениях.

В Keras слой `Conv2DTranspose` позволяет нам выполнять операции сверточного транспонирования тензоров. Накладывая эти слои друг на друга, мы можем постепенно увеличивать размер каждого слоя, используя шаг 2, пока не вернемся к исходному размеру изображения 32 × 32.

In [22]:
```
# Decoder
decoder_input = layers.Input(shape=(EMBEDDING_DIM,), name="decoder_
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose(
    128, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    64, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    32, (3, 3), strides=2, activation="relu", padding="same"
)(x)
decoder_output = layers.Conv2D(
    CHANNELS,
    (3, 3),
    strides=1,
    activation="sigmoid",
    padding="same",
    name="decoder_output",
)(x)

decoder = models.Model(decoder_input, decoder_output)
decoder.summary()
```

Model: "functional_6"

Layer (type)	Output Shape	
decoder_input (InputLayer)	(None, 2)	
dense_8 (Dense)	(None, 2048)	
reshape (Reshape)	(None, 4, 4, 128)	
conv2d_transpose (Conv2DTranspose)	(None, 8, 8, 128)	
conv2d_transpose_1 (Conv2DTranspose)	(None, 16, 16, 64)	
conv2d_transpose_2 (Conv2DTranspose)	(None, 32, 32, 32)	
decoder_output (Conv2D)	(None, 32, 32, 1)	

Total params: 246,273 (962.00 KB)
Trainable params: 246,273 (962.00 KB)
Non-trainable params: 0 (0.00 B)

Чтобы одновременно обучать кодировщик и декодер, необходимо определить модель, которая будет представлять поток изображения через кодировщик и обратно через декодер. Здесь выходные данные автокодировщика — это просто выходные данные кодировщика после того, как они прошли через декодер.

In [23]:
```
# Autoencoder
autoencoder = models.Model(
    encoder_input, decoder(encoder_output)
)  # decoder(encoder_output)
autoencoder.summary()
```

Model: "functional_7"

Layer (type)	Output Shape	
encoder_input (InputLayer)	(None, 32, 32, 1)	
conv2d (Conv2D)	(None, 16, 16, 32)	
conv2d_1 (Conv2D)	(None, 8, 8, 64)	
conv2d_2 (Conv2D)	(None, 4, 4, 128)	
flatten (Flatten)	(None, 2048)	
encoder_output (Dense)	(None, 2)	
functional_6 (Functional)	(None, 32, 32, 1)	

Total params: 343,043 (1.31 MB)

```
Trainable params: 343,043 (1.31 MB)
Non-trainable params: 0 (0.00 B)
```

Обучение автоэнкодера

Теперь, когда наша модель определена, нам просто нужно скомпилировать ее с функцией потерь и оптимизатором. В качестве функции потерь обычно выбирается либо среднеквадратическая ошибка (MSE), либо бинарная перекрестная энтропия (binary cross-entropy) между отдельными пикселями исходного изображения и его реконструкции.

In [24]:
```
# Compile the autoencoder
autoencoder.compile(optimizer="adam", loss="binary_crossentropy")
```

Оптимизация MSE означает, что сгенерированные выходные данные будут симметрично распределены вокруг средних значений пикселей (поскольку завышение оценки штрафуется так же, как и занижение). С другой стороны, функция потерь бинарная кросс-энтропия асимметрична — она штрафует за ошибки в направлении к крайним значениям более строго, чем за ошибки в направлении к центру. Например, если истинное значение пикселя равно, скажем, 0.7, то создание пикселя со значением 0.8 штрафуется более строго, чем создание пикселя со значением 0.6. Если истинное значение пикселя низкое (скажем, 0,3), то создание пикселя со значением 0.2 штрафуется более строго, чем создание пикселя со значением 0.4. Это приводит к тому, что функция потерь бинарная кросс-энтропия создает немного более размытые изображения, чем функция потерь MSE (поскольку она имеет тенденцию приближать прогнозы к 0.5), но иногда это желательно, поскольку MSE может привести к явно пикселизированным краям. Не существует правильного или неправильного выбора — после экспериментов следует выбрать тот, который лучше всего подходит для вашего варианта использования.

In [25]:
```
# Create subdirectories
for path in ["./checkpoint","./logs","./models","./output"]:
    try:
        os.mkdir(path)
        print("Directory '%s' created successfully" % path)
    except OSError as error:
        print("Directory '%s' exists" % path)
```

```
Directory './checkpoint' created successfully
Directory './logs' created successfully
Directory './models' created successfully
Directory './output' created successfully
```

In [26]:
```
# Create a model save checkpoint
```

```
model_checkpoint_callback = callbacks.ModelCheckpoint(
    filepath="./checkpoint.keras",
    save_weights_only=False,
    save_freq="epoch",
    monitor="loss",
    mode="min",
    save_best_only=True,
    verbose=0,
)
tensorboard_callback = callbacks.TensorBoard(log_dir="./logs")
```

In [27]:
```
autoencoder.fit(
    x_train,
    x_train,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    shuffle=True,
    validation_data=(x_test, x_test),
    callbacks=[model_checkpoint_callback, tensorboard_callback],
)
```

```
Epoch 1/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 13s 15ms/step - loss: 0.3521 - val_los
s: 0.2612
Epoch 2/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 8s 14ms/step - loss: 0.2588 - val_loss:
0.2563
Epoch 3/3
600/600 ━━━━━━━━━━━━━━━━━━━━ 8s 14ms/step - loss: 0.2545 - val_loss:
0.2538
```

Out[27]:
```
<keras.src.callbacks.history.History at 0x16d2e1fa0>
```

In [28]:
```
# Save the final models
autoencoder.save("./models/autoencoder.keras")
encoder.save("./models/encoder.keras")
decoder.save("./models/decoder.keras")
```

Реконструкция при помощи автокодировщика

Можно проверить способность автокодировщика реконструировать изображения, пропуская изображения из тестового набора через автокодировщик и сравнивая выходные данные с исходными изображениями.

In [29]:
```
n_to_predict = 5000
example_images = x_test[:n_to_predict]
example_labels = y_test[:n_to_predict]
```

In [30]:
```
predictions = autoencoder.predict(example_images)

print("Example real clothing items")
display(example_images)
print("Reconstructions")
```

```
display(predictions)
```

```
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
```

```
Example real clothing items
```

```
Reconstructions
```

Обратите внимание, что реконструкция не идеальна — в исходных изображениях все еще есть некоторые детали, которые не фиксируются в процессе декодирования, например логотипы. Это связано с тем, что, сокращая каждое изображение всего до двух чисел, мы, естественно, теряем некоторую информацию.

Визуализация латентного пространства

Можно визуализировать изображения в скрытом (латентном) пространстве, пропуская тестовый набор через кодировщик и отображая полученные точки.

In [31]:
```
# Encode the example images
embeddings = encoder.predict(example_images)
```

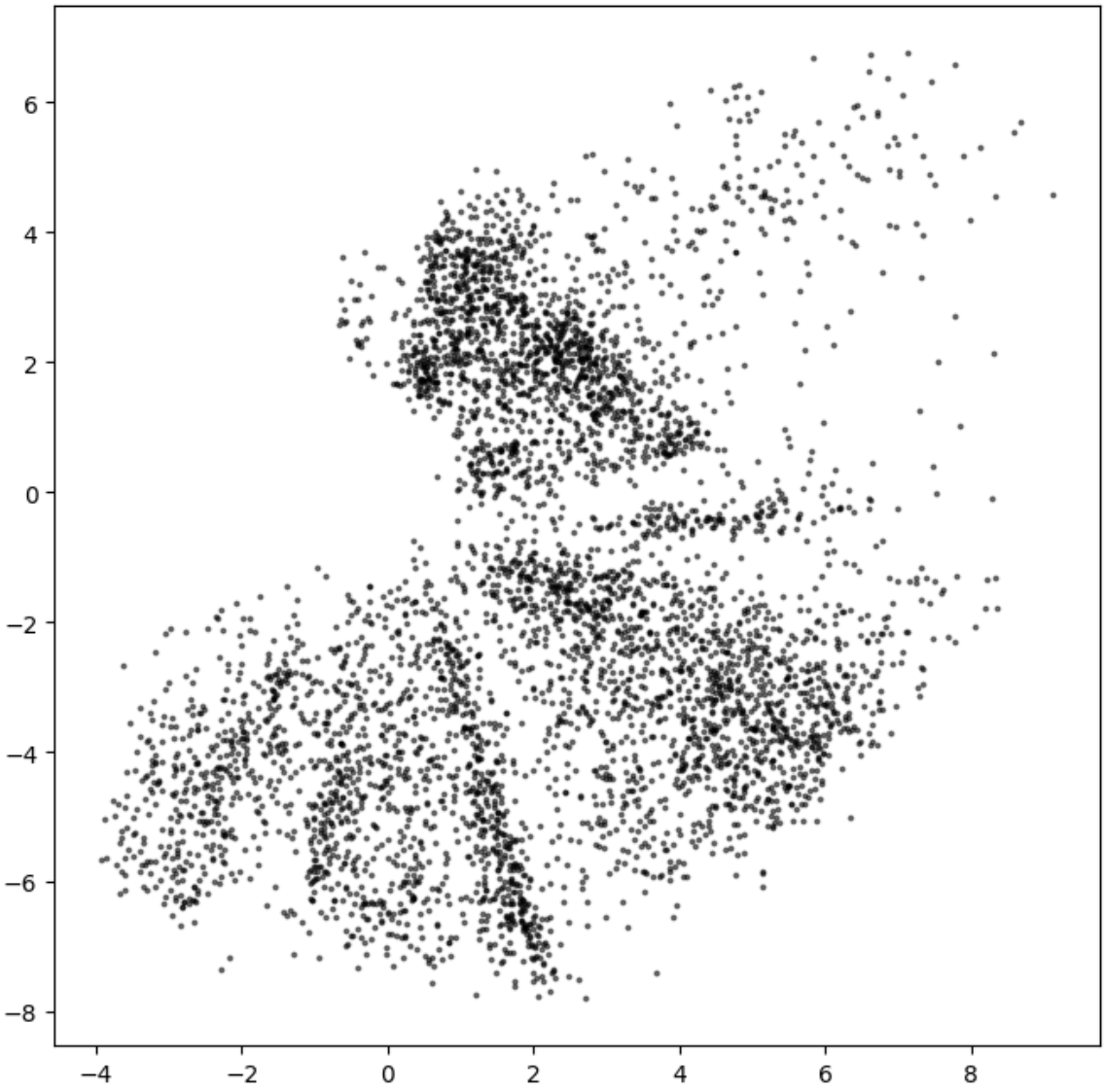
```
157/157 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```

Эмбеддинг (вложение) — это уменьшение размерности признакового пространства ради повышения производительности модели или других целей.

In [32]:
```
# Some examples of the embeddings
print(embeddings[:10])
```

```
[[ 1.7178348   1.8295133 ]
 [ 5.880697   -3.8568764 ]
 [-3.0648448  -6.3299756 ]
 [-1.9945716  -3.6896198 ]
 [ 3.499811   -1.6244596 ]
 [-1.3813131  -5.2777705 ]
 [ 1.8320242  -1.7197794 ]
 [ 2.7477362  -1.7764586 ]
 [-0.41122153  2.2871892 ]
 [ 0.7755359   3.4894876 ]]
```

In [33]:
```
# Show the encoded points in 2D space
figsize = 8

plt.figure(figsize=(figsize, figsize))
plt.scatter(embeddings[:, 0], embeddings[:, 1], c="black", alpha=0.
plt.show()
```

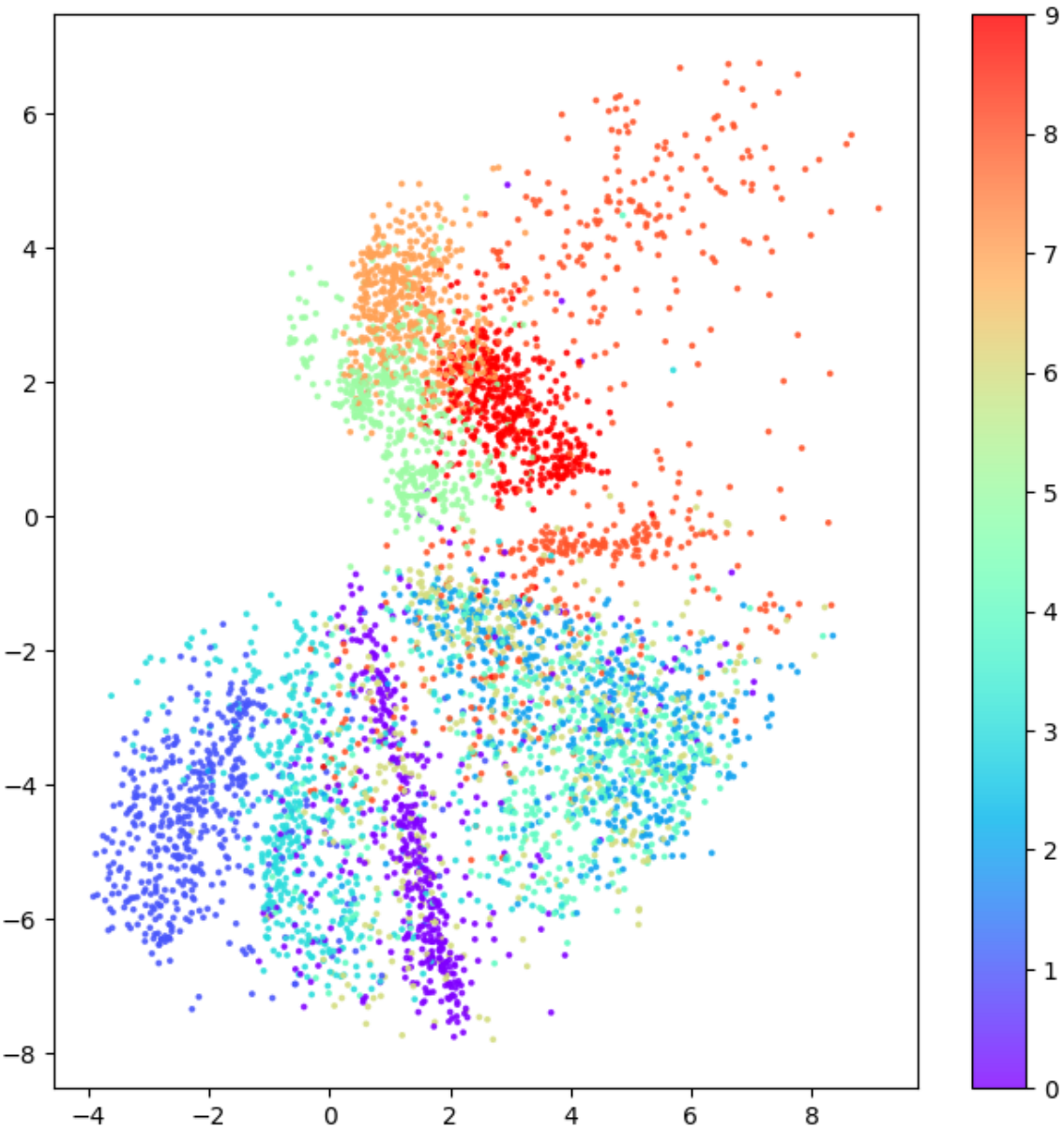


Чтобы лучше понять, как структурировано это скрытое пространство, можно использовать метки, поставляемые с набором данных Fashion-MNIST, описывающие тип предмета на каждом изображении. Всего существует 10 групп предметов:

- 0 – T-shirt/top
- 1 – Trouser
- 2 – Pullover
- 3 – Dress
- 4 – Coat
- 5 – Sandal
- 6 – Shirt
- 7 – Sneaker
- 8 – Bag
- 9 – Ankle boot

In [34]:
```
# Colour the embeddings by their label
example_labels = y_test[:n_to_predict]
```

```
figsize = 8
plt.figure(figsize=(figsize, figsize))
plt.scatter(
    embeddings[:, 0],
    embeddings[:, 1],
    cmap="rainbow",
    c=example_labels,
    alpha=0.8,
    s=3,
)
plt.colorbar()
plt.show()
```



Несмотря на то, что метки одежды никогда не показывались модели во время обучения, автоэнкодер естественным образом сгруппировал предметы, которые выглядят одинаково, в одних и тех же частях скрытого пространства. Например, темно-синее облако точек в правом нижнем углу скрытого пространства — это разные изображения брюк, а красное облако точек ближе к центру — это обувь.

Генерация при помощи декодера

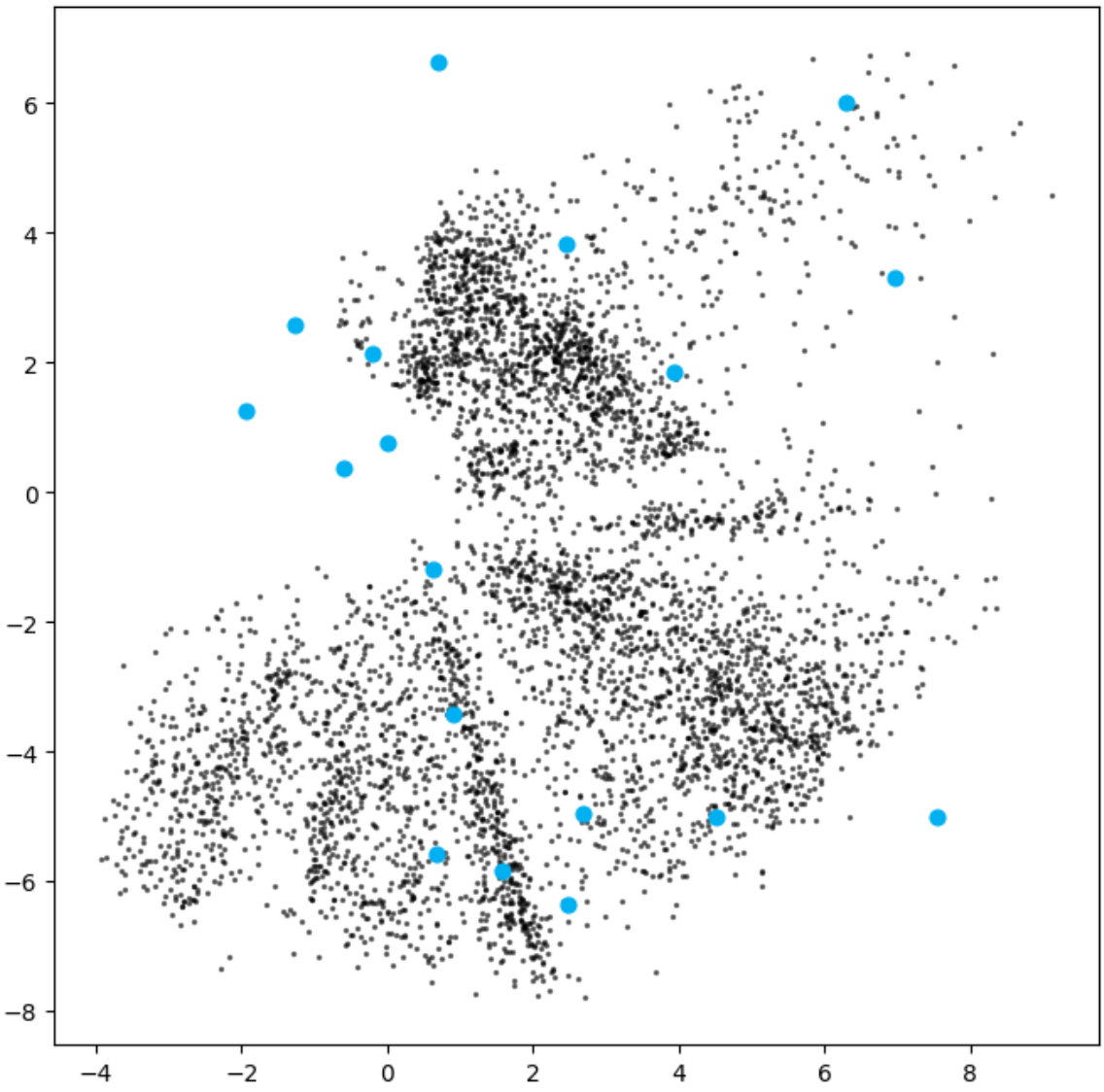
Можно сгенерировать новые изображения, выбирая некоторые точки в скрытом пространстве и используя декодер для преобразования их обратно в пространство пикселей.

In [35]:
```
# Get the range of the existing embeddings
mins, maxs = np.min(embeddings, axis=0), np.max(embeddings, axis=0)

# Sample some points in the latent space
grid_width, grid_height = (6, 3)
sample = np.random.uniform(
    mins, maxs, size=(grid_width * grid_height, EMBEDDING_DIM)
)
```

In [36]:
```
# Decode the sampled points
reconstructions = decoder.predict(sample)
```

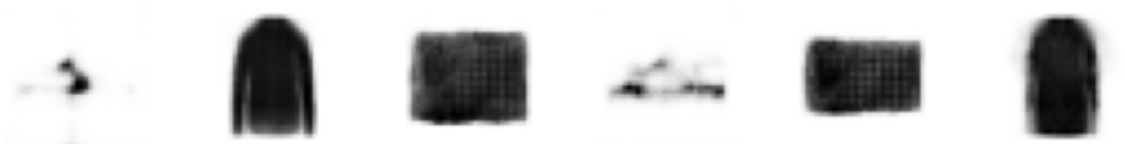
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step
```

In [37]:
```
# Draw a plot of...
figsize = 8
plt.figure(figsize=(figsize, figsize))

# ... the original embeddings ...
plt.scatter(embeddings[:, 0], embeddings[:, 1], c="black", alpha=0.

# ... and the newly generated points in the latent space
plt.scatter(sample[:, 0], sample[:, 1], c="#00B0F0", alpha=1, s=40)
plt.show()

# Add underneath a grid of the decoded images
fig = plt.figure(figsize=(figsize, grid_height * 2))
fig.subplots_adjust(hspace=0.4, wspace=0.4)

for i in range(grid_width * grid_height):
    ax = fig.add_subplot(grid_height, grid_width, i + 1)
    ax.axis("off")
    ax.text(
        0.5,
        -0.35,
        str(np.round(sample[i, :], 1)),
        fontsize=10,
        ha="center",
        transform=ax.transAxes,
    )
    ax.imshow(reconstructions[i, :, :], cmap="Greys")
```



[-1.9 1.3] [2.4 3.8] [2.5 -6.3] [-0.2 2.1] [-1.3 2.6] [3.9 1.9]

[0.7 -5.6] [0.9 -3.4] [0.6 -1.2] [0.7 6.6] [1.6 -5.8] [7.5 -5.]



[-0.6 0.4] [4.5 -5.] [7. 3.3] [0. 0.8] [6.3 6.] [2.7 -4.9]

Каждая синяя точка соответствует одному из изображений, показанных ниже диаграммы, с вектором эмбеддинга, показанным под ней. Некоторые из созданных элементов более реалистичны, чем другие. Почему так?

- Некоторые предметы одежды представлены на очень небольшой площади, а другие — на гораздо большей.
- Распределение не симметрично относительно точки (0, 0) или ограничено. Например, существует гораздо больше точек с положительными значениями оси Y, чем с отрицательными, а некоторые точки даже простираются до значения оси Y > 8.
- Между цветами имеются большие промежутки, содержащие мало точек.

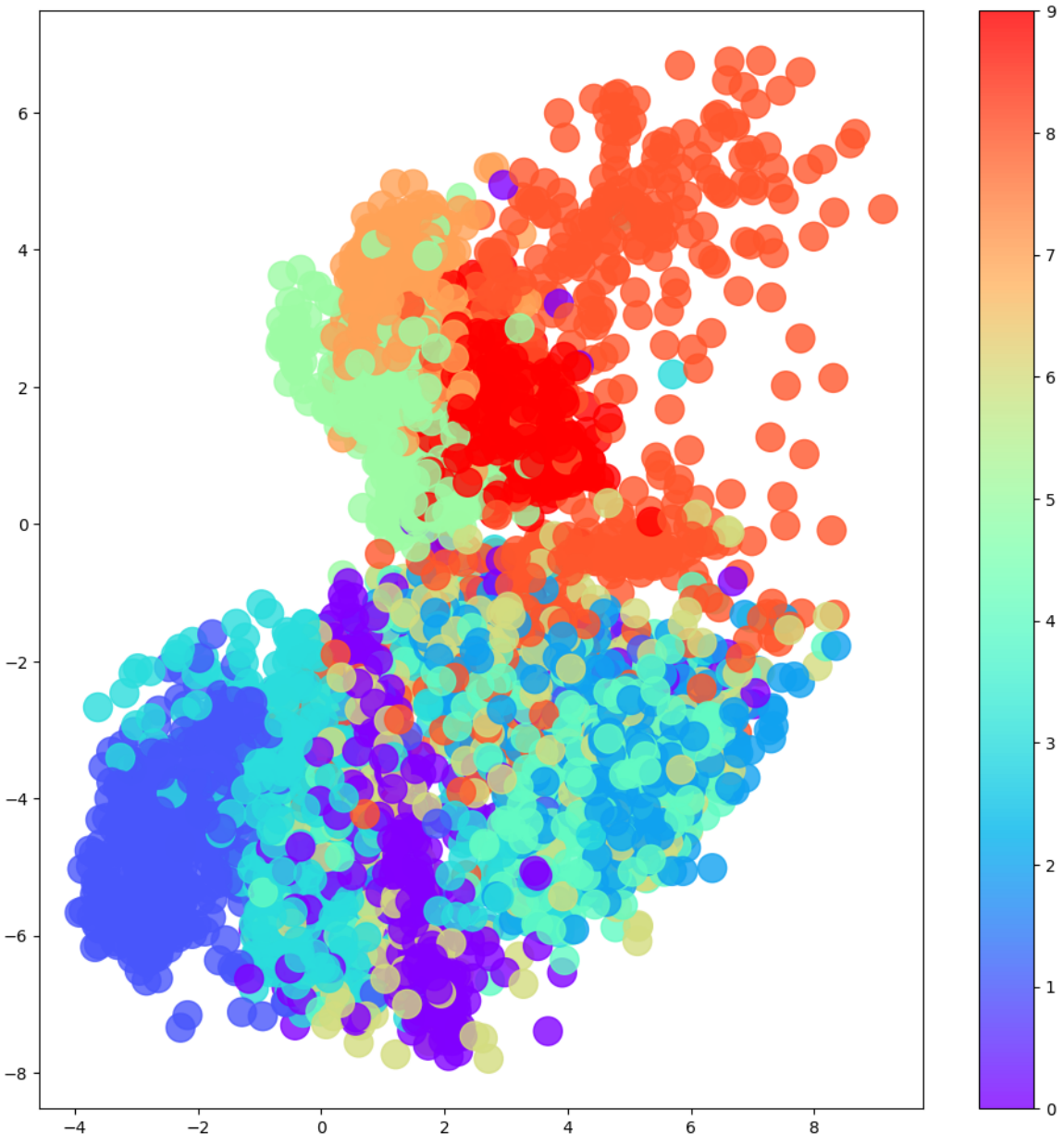
Поэтому выборка из скрытого пространства является весьма сложной задачей. Если наложить на скрытое пространство изображения декодированных точек сетки, то можно понять, почему декодер не всегда может генерировать удовлетворительные изображения.

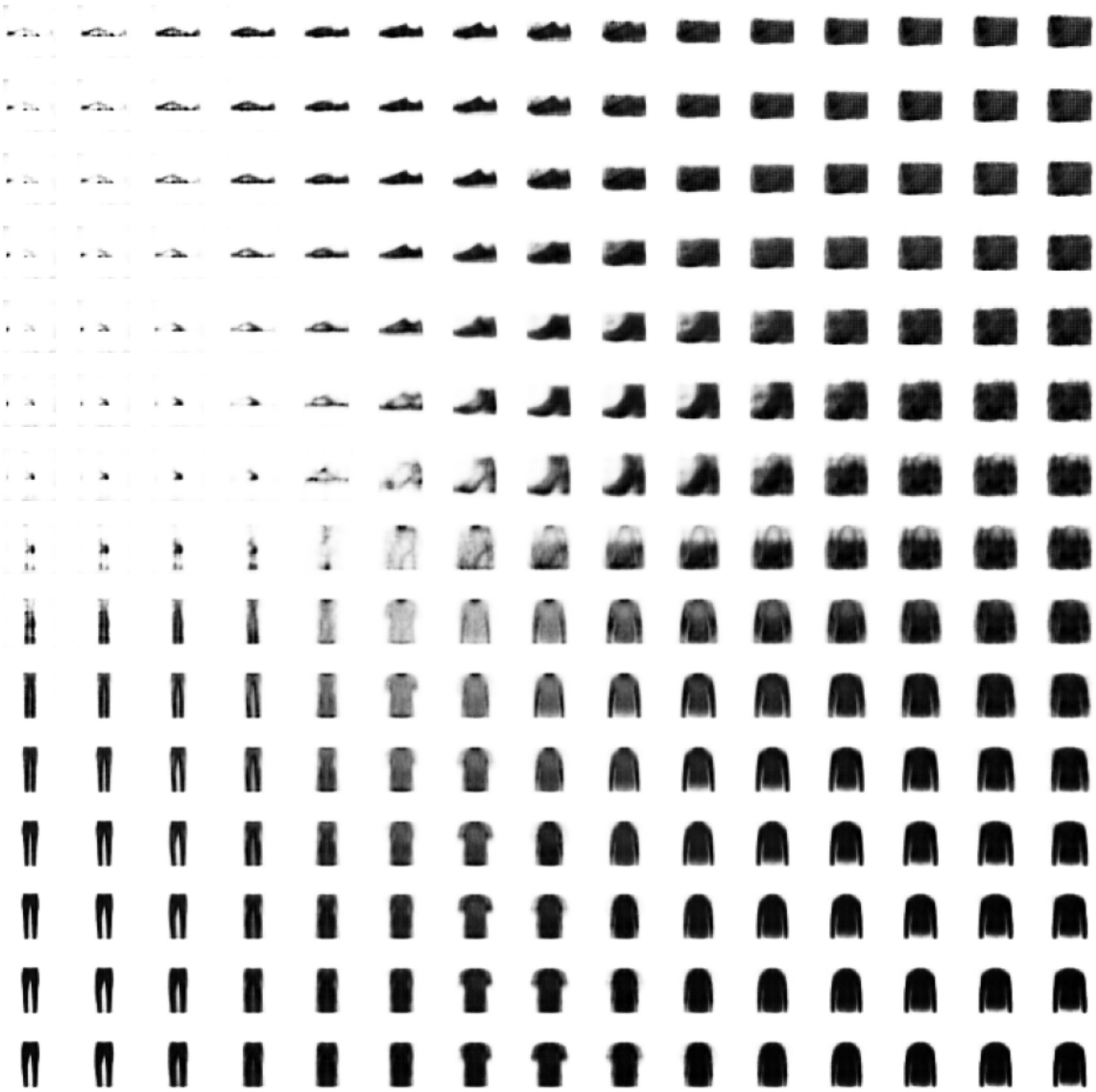
Теперь выполним цветную визуализацию в латентном пространстве.

In [38]:
```
# Colour the embeddings by their label
figsize = 12
grid_size = 15
plt.figure(figsize=(figsize, figsize))
plt.scatter(
    embeddings[:, 0],
    embeddings[:, 1],
    cmap="rainbow",
    c=example_labels,
    alpha=0.8,
    s=300,
)
plt.colorbar()

x = np.linspace(min(embeddings[:, 0]), max(embeddings[:, 0]), grid_
y = np.linspace(max(embeddings[:, 1]), min(embeddings[:, 1]), grid_
xv, yv = np.meshgrid(x, y)
xv = xv.flatten()
yv = yv.flatten()
grid = np.array(list(zip(xv, yv)))

reconstructions = decoder.predict(grid)
# plt.scatter(grid[:, 0], grid[:, 1], c="black", alpha=1, s=10)
plt.show()

fig = plt.figure(figsize=(figsize, figsize))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(grid_size**2):
    ax = fig.add_subplot(grid_size, grid_size, i + 1)
    ax.axis("off")
```

```
    ax.imshow(reconstructions[i, :, :], cmap="Greys")
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step
```





При применении автокодировщика для генерации изображений возникает три проблемы:

- если выбираем точки равномерно в ограниченном пространстве, то мы с большей вероятностью выберем предмет, который занимает большую область пространства
- так как распределение точек в латентном пространстве не определено, неочевидно, как следует выбирать случайную точку в этом пространстве, технически можно бы выбрать любую точку двумерной плоскости
- наконец, в латентном пространстве имеются пустые области, где ни одно из исходных изображений не закодировано. Например, по краям области имеются большие белые пространства — у автокодировщика нет причин гарантировать, что точки здесь декодируются в узнаваемые предметы одежды, поскольку здесь кодируется очень мало изображений в обучающем наборе. В двух измерениях эта проблема не является серьезной – автоэнкодер

имеет лишь небольшое количество измерений, с которыми он может работать, поэтому, естественно, ему приходится сжимать группы одежды вместе, в результате чего пространство между группами одежды становится относительно небольшим. Однако по мере того, как мы начинаем использовать больше измерений скрытого пространства для создания более сложных изображений, таких как лица, эта проблема становится еще более очевидной. Если мы дадим автокодировщику полную свободу действий в том, как он использует скрытое пространство для кодирования изображений, между группами схожих точек возникнут огромные промежутки, и наличие этих промежутков будет мешать генерировать правильно сформированные изображения.

Постановка задачи о генеративной модели

Задан набор данных $\mathbf{X}$, причем предполагается, что данные в $\mathbf{X}$ имеют некоторое неизвестное вероятностное распределением p_d.

Требуется построить генеративную модель, порождающую вероятностное распределение p_m, которое имитирует распределение p_d.

Если эта цель будет достигнута, то можно будет делать выборку из распределения p_m, чтобы генерировать новые данные, которые выглядят как данные из распределения p_d.

В качестве генеративной модели мог бы выступать декодер автокодировщика, но для этого требуется решить проблемы, указанные выше. Чтобы решить эти проблемы, нужно от автокодировщика перейти к **вариационному автокодировщику**.

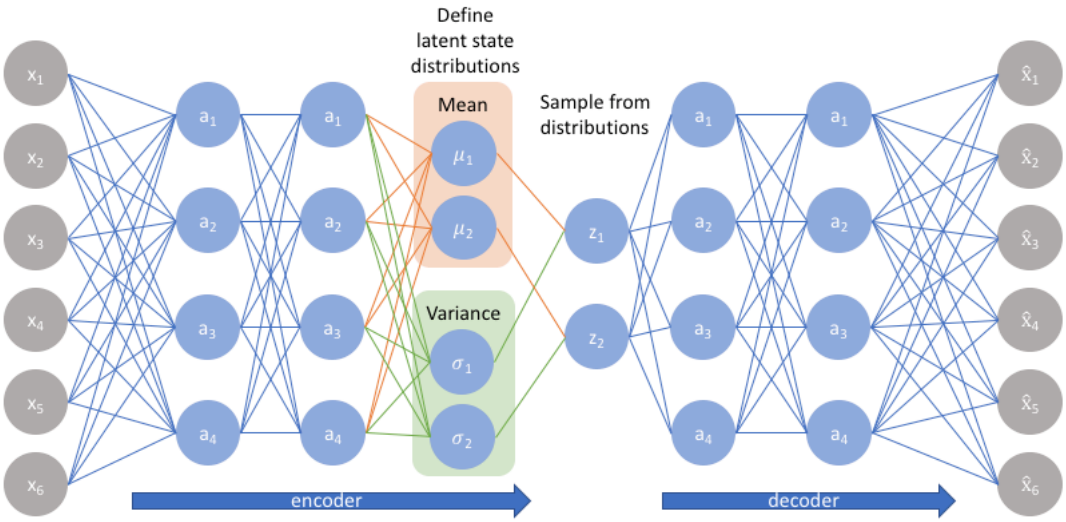
Вариационные автокодировщики

Вместо того, чтобы напрямую выводить значения скрытого состояния, как в стандартном автокодировщике, кодировщик вариационного автокодировщика (VAE) выводит параметры, **описывающие распределение для каждого измерения в скрытом пространстве**. Поскольку мы предполагаем, что наше априорное распределение соответствует нормальному, мы выводим два вектора, описывающих среднее значение и дисперсию распределения скрытого состояния. Если бы мы хотели построить полноценную многомерную гауссову модель, то нам нужно было бы определить ковариационную матрицу, описывающую, как коррелирует каждое из измерений. Однако мы сделаем упрощающее предположение, что наша ковариационная матрица имеет ненулевые значения только на диагонали, что позволяет

нам описать эту информацию в виде простого вектора.

Затем кодировщик генерирует скрытый вектор путем выборки из этого нормального распределения и декодер реконструирует исходный входной сигнал.

Эта организация процесса работы вариационного автокодировщика показана на изображении ниже:



Статистическое обоснование VAE

Следуя общей логике автокодировщиков (AE), предположим, что существует некоторая скрытая переменная $\mathbf{z}$, которая создает наблюдение $\mathbf{x}$.

Получив выходное значение $\mathbf{x}$, мы хотели бы вывести характеристики переменной $\mathbf{z}$. Другими словами, мы хотели бы вычислить условное распределение вероятностей $p(\mathbf{z} \mid \mathbf{x})$, используя байесовский вывод согласно формуле Байеса:

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

(в формуле Байеса могут фигурировать как вероятности, так и плотности вероятностей).

Известно, что вычислить $p(\mathbf{x})$ довольно сложно. Для непрерывных распределений нужно интегрировать по всем $\mathbf{z}$, как показано ниже:

$$p(\mathbf{x}) = \int p(\mathbf{x} \mid \mathbf{z})\, p(\mathbf{z})\, d\mathbf{z},$$

то есть мы исключаем $\mathbf{z}$ из распределения вероятностей $p(\mathbf{x} \mid \mathbf{z})$.

Используя VAE, можно применить т.н. **вариационный вывод** для оценки этого значения. Можно аппроксимировать $p(\mathbf{z} \mid \mathbf{x})$, рассматривая другое, легко поддающееся обработке (tractable) распределение $q(\mathbf{z} \mid \mathbf{x})$. Мы определяем параметры распределения $q(\mathbf{z} \mid \mathbf{x})$ так, чтобы оно было очень похоже на распределение $p(\mathbf{z} \mid \mathbf{x})$, тогда мы сможем использовать его для выполнения приблизительного вывода (approximate inference) для трудноразрешимого распределения $p(\mathbf{z} \mid \mathbf{x})$.

> **Распределение называется легко поддающимся обработке (tractable), если любую порожденную им предельную (маржинальную) вероятность можно вычислить за линейное время.**

Идея VAE состоит в том, чтобы вывести $p\left(\mathbf{z}\right)$, используя $p\left(\mathbf{z} \mid \mathbf{x}\right)$, т.е. мы хотим сделать нашу скрытую переменную **правдоподобной** для имеющихся данных. В случае MNIST мы хотим ограничиться только допустимыми цифрами.

Вариационный вывод (Variational Inference)

Сперва нам нужно вывести апостериорное распределение $p\left(\mathbf{z} \mid \mathbf{x}\right)$, поскольку мы его еще не знаем. В VAE мы выводим $p\left(\mathbf{z} \mid \mathbf{x}\right)$, используя метод под названием **вариационный вывод (VI)**. VI — это один из популярных методов байесовского вывода, другой популярный метод — это метод Марковских цепей Монте-Карло (MCMC).

> Вариационный вывод (VI) сводит задачу байесовского вывода к задаче оптимизации путем моделирования истинного распределения $p\left(\mathbf{z} \mid \mathbf{x}\right)$ при помощи более простого распределения (например, нормального распределения), которое легко оценить, и минимизации разницы между этими двумя распределениями с помощью **KL-дивергенции**, которая показывает, насколько различны распределения p и q.

Дивергенция (расхождение) Кульбака-Лейблера (KL-дивергенция)

KL-дивергенция показывает, насколько хорошо распределение вероятностей Q аппроксимирует распределение вероятностей P путем вычисления перекрестной энтропии P и Q за вычетом энтропии P.

$$D_{KL}(P||Q) = H(P, Q) - H(P)$$

Для дискретных вероятностных распределений

$$D_{KL}(P \parallel Q) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i}$$

Для абсолютно непрерывных вероятностных распределений P и Q дивергенция Кульбака — Лейблера задаётся выражением

$$D_{\mathrm{KL}}(P \parallel Q) = \int_X p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x,$$

где $p(x)$ и $q(x)$ — функции плотности распределений P и Q соответственно, определённые в области $X \subseteq R^k$.

KL-дивергенция – это неотрицательная и асимметричная мера. KL-дивергенция используется для того, чтобы заставить распределение скрытых переменных быть нормальным, чтобы мы могли выбирать скрытые переменные из нормального распределения. Таким образом, расхождение KL включено в функцию потерь, чтобы улучшить сходство между распределением скрытых переменных и нормальным распределением. Применим это на практике для расчета функции потерь.

Приближенное апостериорное распределение

Вариационный автокодировщик (VAE) – это генеративная модель и она оценивает функцию плотности вероятности (PDF) обучающих данных.

Пусть $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$ – генеративная модель с наблюдаемыми переменными $\mathbf{x}$ и латентными (скрытыми) переменными $\mathbf{z}$.

Задачи апостериорного вывода (построения апостериорного распределения $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$) и обучения для глубоких моделей с латентными (скрытыми) переменными (deep latent variable model, DLVM) являются труднорешаемыми.

Чтобы превратить эти задачи в решаемые, вводится параметрическая модель вывода $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$, которая также называется **энкодером** (кодировщиком). Через $\boldsymbol{\phi}$ обозначаются параметры этой модели вывода, также называемые **вариационными параметрами**. Мы оптимизируем вариационные параметры $\boldsymbol{\phi}$ так, чтобы:

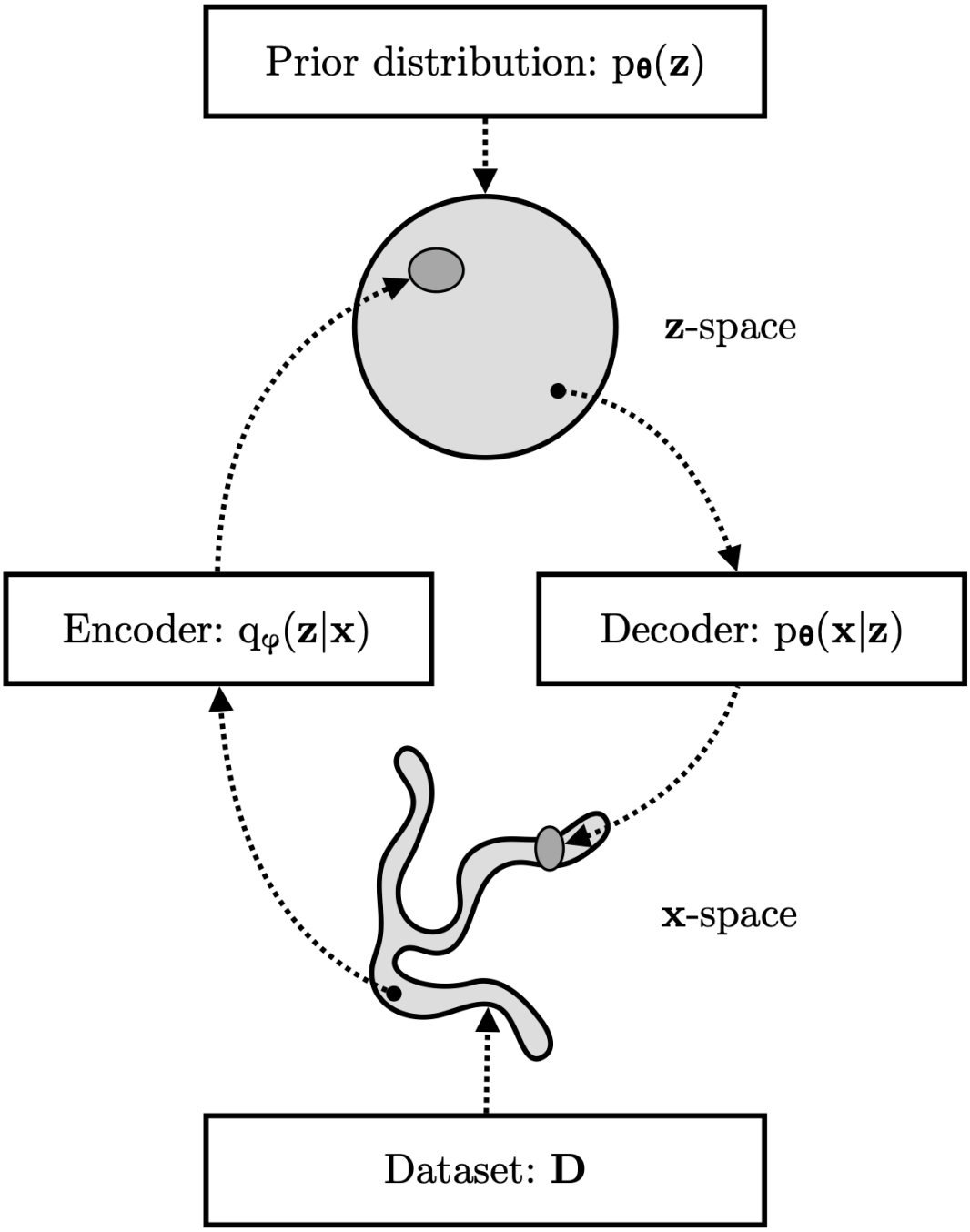
$$q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \approx p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$$

Такая аппроксимация апостериорного распределения поможет нам оптимизировать предельное правдоподобие $p_{\boldsymbol{\theta}}(\mathbf{x})$.

Приближенное апостериорное распределение $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ можно параметризовать с помощью глубоких нейронных сетей. В этом случае вариационные параметры $\boldsymbol{\phi}$ включают в себя веса и смещения нейронной сети. Например, возможно такое построение распределения $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$:

$$q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\left(\mathbf{z}; \boldsymbol{\mu}, \operatorname{diag}(\boldsymbol{\sigma})\right), \ (\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = e_{\boldsymbol{\phi}}(\mathbf{x}),$$

где $e_{\boldsymbol{\phi}}(\mathbf{x})$ – нейронная сеть какой-либо архитектуры.



Как правило, для VAE используется одна нейронная сеть – кодировщик для выполнения апостериорного вывода по всем точкам данных в нашем наборе данных.

Нижняя граница обоснованности (ELBO)

Целью оптимизации вариационного автоэнкодера, как и в других вариационных методах, является **нижняя граница обоснованности**, сокращенно **ELBO** (evidence lower bound). Альтернативным термином для этой целевой функции является термин **вариационная нижняя граница**. Как правило, величину ELBO выводят с помощью неравенства Дженсена (Йенсена). Здесь мы воспользуемся альтернативным выводом, который позволяет избежать неравенства Дженсена и лучше понять точность

ELBO.

Вариационный энкодер (VAE) изучает стохастические отображения между наблюдаемым $\mathbf{x}$-пространством, эмпирическое распределение которого $q_{\mathcal{D}}(\mathbf{x})$ обычно сложное, и латентным (скрытым) $\mathbf{z}$ -пространством, распределение которого может быть относительно простым (например, сферическим, как на рисунке выше). Генеративная модель изучает совместное распределение $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$, разложенное на априорное распределение по латентному (скрытому) пространству $p_{\boldsymbol{\theta}}(\mathbf{z})$ и стохастический декодер $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$. Стохастический кодировщик $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$, также называемый моделью вывода, аппроксимирует истинное, но неразрешимое апостериорное распределение $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ генеративной модели.

При любом выборе модели вывода $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$, включая выбор вариационных параметров $\boldsymbol{\phi}$, имеем:

$$
\begin{aligned}
\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x})\right] = \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})}\right)\right] = \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}\frac{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})}\right)\right] = \\
&= \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}\right)\right]}_{\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})} + \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})}\right)\right]}_{D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right)}
\end{aligned}
$$

Второе слагаемое в уравнении представляет собой **расхождение (дивергенцию) Кульбака-Лейблера (KL-дивергенцию)** между $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ и $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ и является неотрицательным (это вытекает из неравенства Гиббса):

$$
D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})\right) \geqslant 0
$$

и равно нулю тогда и только тогда, когда аппроксимация

$$
q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})
$$

(почти всюду) равна истинному апостериорному распределению

$$
p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})
$$

.

Первое слагаемое в уравнении – это **вариационная нижняя граница**, называемая также **нижней границей обоснованности (ELBO)**:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})}\right)\right] = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid$$

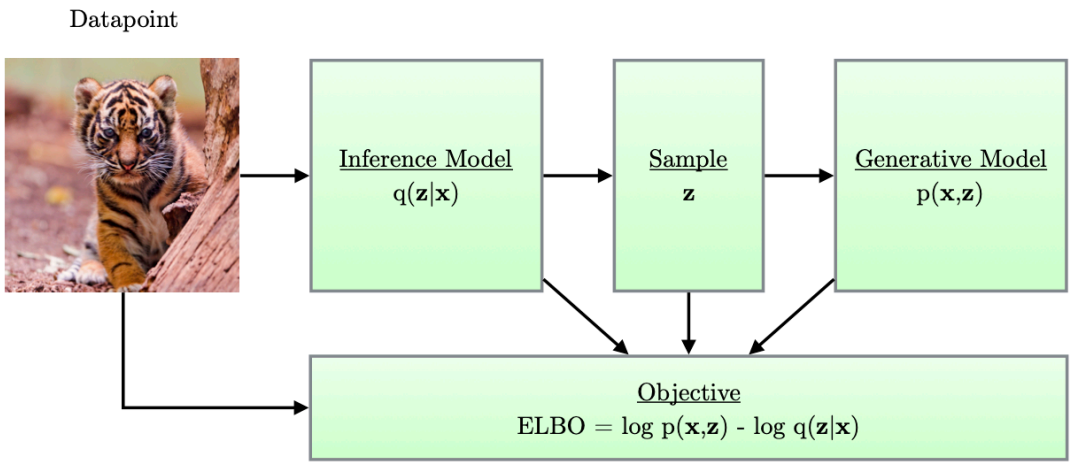
Из-за неотрицательности KL-расхождения ELBO является нижней границей логарифмического правдоподобия данных $\log p_{\boldsymbol{\theta}}(\mathbf{x})$:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x}) - D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}\mid\mathbf{x})\right) \leqslant \log p_{\boldsymbol{\theta}}(\mathbf{x})$$

Итак, KL-дивергенция $D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}\mid\mathbf{x})\right)$ определяет два расстояния:

1. Расхождение (KL-дивергенция) приближенного апостериорного распределения $q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})$ от истинного апостериорного распределения $p_{\boldsymbol{\theta}}(\mathbf{z}\mid\mathbf{x})$.
2. Разрыв между ELBO $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$ и предельным логарифмическим правдоподобием $\log p_{\boldsymbol{\theta}}(\mathbf{x})$. Чем лучше $q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})$ аппроксимирует истинное (апостериорное) распределение $p_{\boldsymbol{\theta}}(\mathbf{z}\mid\mathbf{x})$ с точки зрения KL-дивергенции, тем меньше этот разрыв.

Простая схема вычислительного потока в вариационном автоэнкодере показана на рисунке.



Глядя на неравенство для ELBO, можно понять, что максимизация ELBO $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$ относительно параметров $\boldsymbol{\theta}$ и $\boldsymbol{\phi}$ одновременно оптимизирует две важные для нас величины:

1. Это приблизительно максимизирует предельное правдоподобие $p_{\boldsymbol{\theta}}(\mathbf{x})$. Это означает, что наша генеративная модель работает лучше.
2. Это минимизирует KL-расхождение аппроксимации $q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})$ от истинного апостериорного распределения $p_{\boldsymbol{\theta}}(\mathbf{z}\mid\mathbf{x})$, так что приближенное распределение $q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})$ становится лучше.

Стохастическая градиентная оптимизация ELBO

Важным свойством ELBO является то, что он позволяет проводить совместную оптимизацию по всем параметрам ($\boldsymbol{\theta}$ и $\boldsymbol{\phi}$) с использованием стохастического градиентного спуска (SGD). Мы можем начать со случайных начальных значений $\boldsymbol{\theta}$ и $\boldsymbol{\phi}$ и стохастически оптимизировать их значения до сходимости.

Пусть дан набор i.i.d. (independent identically distributed) независимых и одинаково распределенных данных $\mathcal{D}$, тогда целевая функция для ELBO представляет собой сумму (или среднее значение) ELBO для отдельных точек данных:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{D}) = \sum_{\mathbf{x}\in\mathcal{D}} \mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$$

Величина ELBO в индивидуальной точке данных $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$ и ее градиент $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$ в общем случае не разрешимы в замкнутом виде. Однако хорошие несмещенные оценки $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$ существуют, так что можно применять минипакетный SGD.

Несмещенные градиенты ELBO относительно параметров генеративной модели $\boldsymbol{\theta}$ получить просто:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) &= \nabla_{\boldsymbol{\theta}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})\right] = \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\nabla_{\boldsymbol{\theta}}\left\{\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})\right\}\right] = \\
&= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\nabla_{\boldsymbol{\theta}}\left\{\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})\right\}\right] \approx \\
&\approx \nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}^s)
\end{aligned}$$

Последняя строка представляет собой простую оценку методом Монте-Карло предыдущей строки, где $\mathbf{z}^s$ в последней строке представляет собой случайную выборку из $q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})$: $\mathbf{z}^s \sim q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})$.

Несмещенные градиенты ELBO относительно вариационных параметров $\boldsymbol{\phi}$ получить труднее, так как математическое ожидание ELBO берется относительно распределения $q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})$, которое является функцией $\boldsymbol{\phi}$. Вообще говоря, нельзя поменять местами операцию математического ожидания и градиента:

$$\nabla_{\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})\right] \neq$$

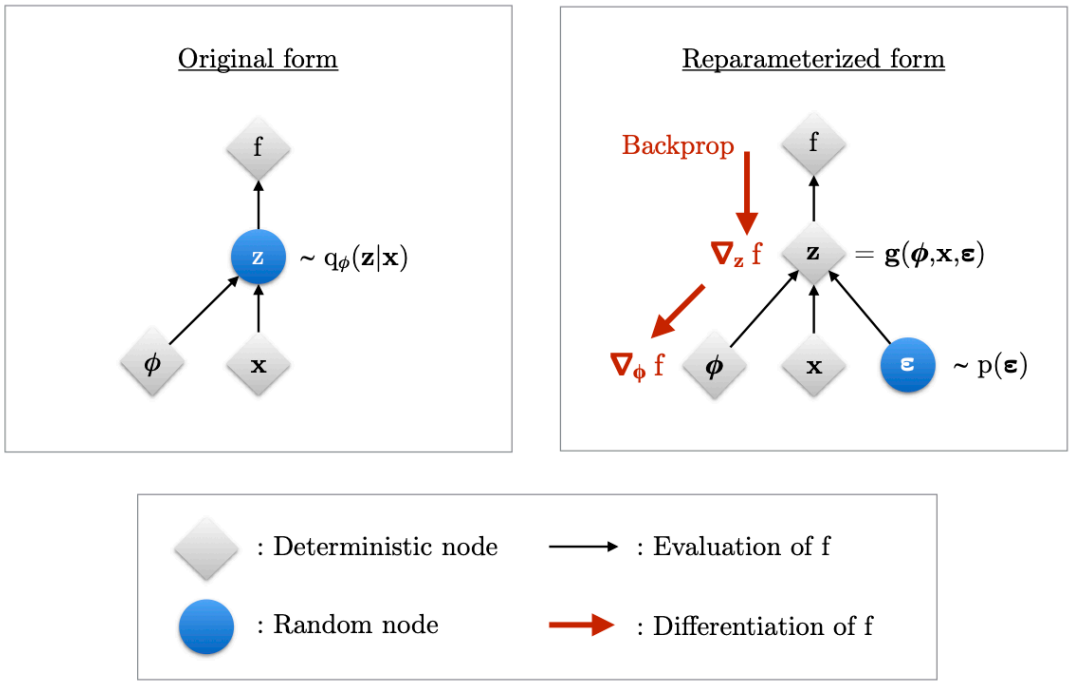
$$\neq \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\nabla_{\boldsymbol{\phi}}\left\{\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}\mid\mathbf{x})\right\}\right]$$

В случае непрерывных скрытых переменных мы можем использовать прием перепараметризации для вычисления несмещенной оценки $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$, как показано ниже. Эта стохастическая оценка позволяет

нам оптимизировать ELBO с помощью SGD.

Прием перепараметризации

Для непрерывных скрытых переменных и дифференцируемого кодировщика и генеративной модели величину ELBO можно прямо дифференцировать по обоим параметрам $\boldsymbol{\theta}$ и $\boldsymbol{\phi}$ при помощи замены переменных, также называемой **приемом перепараметризации** (**reparameterization trick**).



Идея приема перепараметризации состоит в следующем. Вариационные параметры $\boldsymbol{\phi}$ влияют на целевую функцию f через случайную величину $\mathbf{z} \sim q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$. Необходимо вычислить градиенты $\nabla_{\boldsymbol{\phi}} f$ для оптимизации целевой функции с помощью SGD. В исходной форме (слева) мы не можем дифференцировать f по $\boldsymbol{\phi}$, потому что мы не можем напрямую распространять обратно градиенты через случайную переменную $\mathbf{z}$. Мы можем вынести случайность за пределы $\mathbf{z}$, перепараметризовав переменную $\mathbf{z}$ как детерминированную и дифференцируемую функцию от $\boldsymbol{\phi}$, $\mathbf{x}$ и вновь введенной случайной величины $\boldsymbol{\epsilon}$. Это позволяет нам обеспечить обратное распространение ошибки через $\boldsymbol{z}$ и вычислять градиенты $\nabla_{\boldsymbol{\phi}} f$.

Замена переменных

Во-первых, представим случайную величину $\mathbf{z} \sim q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ как некоторое дифференцируемое (и обратимое) преобразование другой случайной величины $\boldsymbol{\epsilon}$ при заданных $\mathbf{z}$ и $\boldsymbol{\phi}$:

$$\mathbf{z} = g(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x}),$$

где распределение случайной величины $\boldsymbol{\epsilon}$ не зависит от $\mathbf{x}$ или $\boldsymbol{\phi}$.

Градиент математического ожидания при замене переменной

При такой замене переменной математическое ожидание можно переписать через $\boldsymbol{\epsilon}$:

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[f\left(\mathbf{z}\right)\right] = \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[f\left(\mathbf{z}\right)\right],$$

где $\mathbf{z} = g\left(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x}\right)$, и операторы математического ожидания и градиента становятся коммутативными, и мы можем использовать простую оценку Монте-Карло:

$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[f\left(\mathbf{z}\right)\right] = \nabla_{\boldsymbol{\phi}} \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[f\left(\mathbf{z}\right)\right] =$$

$$= \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[\nabla_{\boldsymbol{\phi}} f\left(\mathbf{z}\right)\right] \approx \nabla_{\boldsymbol{\phi}} f\left(\mathbf{z}^s\right),$$

где в последней строке $\mathbf{z}^s = g\left(\boldsymbol{\epsilon}^s, \boldsymbol{\phi}, \mathbf{x}\right)$ со случайной выборкой шума $\boldsymbol{\epsilon}^s \sim p\left(\boldsymbol{\epsilon}\right)$.

Градиент ELBO

При репараметризации мы можем заменить математическое ожидание относительно $q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)$ математическим ожиданием относительно $p\left(\boldsymbol{\epsilon}\right)$. Величина ELBO можно представлена как:

$$\mathcal{L}_{\boldsymbol{\theta}, \boldsymbol{\phi}}\left(\mathbf{x}\right) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)\right] =$$

$$= \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[\log p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)\right],$$

где $\mathbf{z} = g\left(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x}\right)$.

В результате можно получить простую оценку методом Монте-Карло $\tilde{\mathcal{L}}_{\boldsymbol{\theta}, \boldsymbol{\phi}}\left(\mathbf{x}\right)$ для величины ELBO отдельной точки данных $\mathbf{x}$, где используется выборка шума из одной точки $\boldsymbol{\epsilon}^s$ из распределения $p\left(\boldsymbol{\epsilon}\right)$:

$$\boldsymbol{\epsilon}^s \sim p\left(\boldsymbol{\epsilon}\right)$$

$$\mathbf{z}^s = g\left(\boldsymbol{\epsilon}^s, \boldsymbol{\phi}, \mathbf{x}\right)$$

$$\tilde{\mathcal{L}}_{\boldsymbol{\theta}, \boldsymbol{\phi}}\left(\mathbf{x}\right) = \log p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}^s\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z}^s \mid \mathbf{x}\right)$$

Эта последовательность операций может быть выражена в виде символьного графа в таких программах, как Tensorflow, и легко продифференцирована по параметрам $\boldsymbol{\theta}$ и $\boldsymbol{\phi}$. Полученный градиент $\nabla_{\boldsymbol{\phi}} \mathcal{L}_{\boldsymbol{\theta}, \boldsymbol{\phi}}\left(\mathbf{x}\right)$ используется для оптимизации ELBO с использованием минипакетного SGD (см. алгоритм 1). Первоначально этот алгоритм назывался алгоритмом Auto-Encoding Variational Bayes (AEVB) (Kingma

and Welling, 2013).

Алгоритм стохастической оптимизации ELBO приводится ниже:

Algorithm 1: Stochastic optimization of the ELBO. Since noise originates from both the minibatch sampling and sampling of $p(\boldsymbol{\epsilon})$, this is a doubly stochastic optimization procedure. We also refer to this procedure as the *Auto-Encoding Variational Bayes* (AEVB) algorithm.

Data:
$\mathcal{D}$: Dataset
$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$: Inference model
$p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$: Generative model
Result:
$\boldsymbol{\theta}, \boldsymbol{\phi}$: Learned parameters

$(\boldsymbol{\theta}, \boldsymbol{\phi}) \leftarrow$ Initialize parameters
while *SGD not converged* **do**
  $\mathcal{M} \sim \mathcal{D}$ (Random minibatch of data)
  $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$ (Random noise for every datapoint in $\mathcal{M}$)
  Compute $\tilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{M}, \boldsymbol{\epsilon})$ and its gradients $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \tilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{M}, \boldsymbol{\epsilon})$
  Update $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ using SGD optimizer
end

Несмещенность

Этот градиент является несмещенной оценкой точного градиента ELBO для одной точки данных; при усреднении по шуму $\boldsymbol{\epsilon} \sim p\left(\boldsymbol{\epsilon}\right)$ этот градиент равен градиенту ELBO для одной точки данных:

$$\mathbb{E}_{p(\boldsymbol{\epsilon})}\left[\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \tilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\mathbf{x}; \boldsymbol{\epsilon}\right)\right] = \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\log p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)\right)\right] =$$

$$= \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\mathbb{E}_{p(\boldsymbol{\epsilon})}\left[\log p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)\right]\right) = \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\mathbf{x}\right)$$

Вычисление $\log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)$

Вычисление (оценка) ELBO требует вычисления логарифма плотности $\log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)$ при заданном значении $\mathbf{x}$ и заданном значении $\mathbf{z}$ или, что эквивалентно, $\boldsymbol{\epsilon}$. Эта логарифмическая плотность представляет собой простое вычисление, если мы выбираем правильное преобразование $g()$.

Обратите внимание, что обычно мы знаем плотность $p\left(\boldsymbol{\epsilon}\right)$, так как это плотность выбранного распределения шума. Если $g()$ является обратимой функцией, то плотности $\boldsymbol{\epsilon}$ и $\mathbf{z}$ связаны следующим образом:

$$\log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right) = \log p\left(\boldsymbol{\epsilon}\right) - \log d_{\boldsymbol{\phi}}\left(\mathbf{x}, \boldsymbol{\epsilon}\right),$$

где второй член представляет собой логарифм абсолютного значения определителя матрицы Якоби $(\partial \mathbf{z} / \partial \boldsymbol{\epsilon})$:

$$\log d_{\boldsymbol{\phi}}\left(\mathbf{x}, \boldsymbol{\epsilon}\right)=\log\left|\det\left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}\right)\right|$$

Мы называем это логарифмом определителя преобразования от $\boldsymbol{\epsilon}$ к $\mathbf{z}$. Мы используем обозначение $\log d_{\boldsymbol{\phi}}\left(\mathbf{x}, \boldsymbol{\epsilon}\right)$, чтобы показать, что этот логарифм определителя, подобно $g()$, является функцией $\mathbf{x}$, $\boldsymbol{\epsilon}$ и $\boldsymbol{\phi}$. Матрица Якоби содержит все первые производные преобразования от $\boldsymbol{\epsilon}$ к $\mathbf{z}$:

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}=\frac{\partial\left(z_1, z_2, \ldots, z_k\right)}{\partial\left(\epsilon_1, \epsilon_2, \ldots, \epsilon_k\right)}=\begin{pmatrix}\frac{\partial z_1}{\partial \epsilon_1} & \cdots & \frac{\partial z_1}{\partial \epsilon_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_k}{\partial \epsilon_1} & \cdots & \frac{\partial z_k}{\partial \epsilon_k}\end{pmatrix}$$

Как мы покажем, мы можем создавать очень гибкие преобразования $g()$, для которых $\log d_{\boldsymbol{\phi}}\left(\mathbf{x}, \boldsymbol{\epsilon}\right)$ легко вычислить, что приводит к очень гибким моделям вывода $q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)$.

Факторизованные гауссовские апостериорные распределения

Обычный выбор преобразования – это простой факторизованный гауссовый кодировщик $q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)=\mathcal{N}\left(\mathbf{z} ; \boldsymbol{\mu}, \operatorname{diag}\left(\boldsymbol{\sigma}^2\right)\right)$:

$$\left(\boldsymbol{\mu}, \log \boldsymbol{\sigma}\right)=e_{\boldsymbol{\phi}}\left(\mathbf{x}\right),$$

$$q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)=\prod_i q_{\boldsymbol{\phi}}\left(z_i \mid \mathbf{x}\right)=\prod_i \mathcal{N}\left(z_i ; \mu_i, \sigma_i^2\right),$$

где $\mathcal{N}\left(z_i ; \mu_i, \sigma_i^2\right)$ – PDF одномерного гауссового распределения. После репараметризации можно написать:

$$\boldsymbol{\epsilon} \sim \mathcal{N}\left(\mathbf{0}, \mathbf{I}\right),$$

$$\left(\boldsymbol{\mu}, \log \boldsymbol{\sigma}\right)=e_{\boldsymbol{\phi}}\left(\mathbf{x}\right),$$

$$\mathbf{z}=\boldsymbol{\mu}+\boldsymbol{\sigma} \odot \boldsymbol{\epsilon},$$

где $\odot$ – поэлементное произведение. Якобиан преобразования от $\boldsymbol{\epsilon}$ к $\mathbf{z}$ равен

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}=\operatorname{diag}\left(\boldsymbol{\sigma}\right),$$

т.е. диагональной матрице с элементами σ_i на диагонали. Определитель диагональной (или, в более общем случае, треугольной) матрицы является произведением ее диагональных элементов. Таким образом, логарифм определителя якобиана равен:

$$\log d_{\boldsymbol{\phi}}\left(\mathbf{x}, \boldsymbol{\epsilon}\right) = \log\left|\det\left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}\right)\right| = \sum_i \log \sigma_i$$

и апостериорная плотность равна

$$\log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right) = \log p\left(\boldsymbol{\epsilon}\right) - \log d_{\boldsymbol{\phi}}\left(\mathbf{x}, \boldsymbol{\epsilon}\right) = \sum_i \left(\log \mathcal{N}\left(\epsilon_i; 0, 1\right) - \log \sigma_i\right),$$

где $\mathbf{z} = g\left(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x}\right) \left(= \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}\right)$.

Гауссовское апостериорное распределение с полной ковариацией

Факторизованное гауссовское апостериорное распределение может быть расширено до гауссовского распределения с полной ковариацией:

$$q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right) = \mathcal{N}\left(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\Sigma}\right)$$

Репараметризация этого распределения определяется следующим образом:

$$\boldsymbol{\epsilon} \sim \mathcal{N}\left(\mathbf{0}, \mathbf{I}\right), \mathbf{z} = \boldsymbol{\mu} + \mathbf{L}\,\boldsymbol{\epsilon},$$

где $\mathbf{L}$ – нижняя (или верхняя) треугольная матрица с ненулевыми элементами на диагонали. Внедиагональные элементы определяют корреляции (ковариации) элементов в $\mathbf{z}$.

Причина такой параметризации гауссовского распределения с полной ковариацией заключается в том, что определитель якобиевой матрицы удивительно прост. Якобиева матрица в этом случай тривиальна: $\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \mathbf{L}$. Заметим, что определитель треугольной матрицы равен произведению ее диагональных элементов. Следовательно, в этой параметризации:

$$\log\left|\det\left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}\right)\right| = \sum_i \log|L_{ii}|$$

И логарифмическая плотность апостериорного распределения равна:

$$\log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right) = \log p\left(\boldsymbol{\epsilon}\right) - \sum_i \log|L_{ii}|$$

Эта параметризация соответствует разложению Холецкого $\boldsymbol{\Sigma} = \mathbf{L}\,\mathbf{L}^T$ ковариации $\mathbf{z}$:

$$\boldsymbol{\Sigma} = \mathbb{E}\left[\left(\mathbf{z} - \mathbb{E}\left[\mathbf{z}\right]\right)\left(\mathbf{z} - \mathbb{E}\left[\mathbf{z}\right]\right)^T\right] =$$

$$= \mathbb{E}\left[\mathbf{L}\boldsymbol{\epsilon}(\mathbf{L}\boldsymbol{\epsilon})^T\right] = \mathbf{L}\mathbb{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^T\right]\mathbf{L}^T = \mathbf{L}\,\mathbf{L}^T$$

Обратите внимание, что $\mathbb{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^T\right] = \mathbf{I}$, так как $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.

Один из способов построить матрицу $\mathbf{L}$ с желаемыми свойствами, а именно, треугольностью и ненулевыми диагональными элементами, состоит в следующем:

$$\left(\boldsymbol{\mu}, \log \boldsymbol{\sigma}, \mathbf{L}'\right) = e_{\boldsymbol{\phi}}\left(\mathbf{x}\right),$$

$$\mathbf{L} = \mathbf{L}_{mask} \odot \mathbf{L}' + \operatorname{diag}\left(\boldsymbol{\sigma}\right)$$

Здесь $\mathbf{L}_{mask}$ – это маскирующая матрица с нулями на диагонали и выше нее и единицами под диагональю. Логарифм детерминанта матрицы идентичен факторизованному гауссовскому случаю:

$$\log\left|\det\left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}\right)\right| = \sum_i \log \sigma_i$$

В более общем случае мы можем заменить $\mathbf{z} = \boldsymbol{\mu} + \mathbf{L}\,\boldsymbol{\epsilon}$ цепочкой (дифференцируемых и нелинейных) преобразований; пока якобиан каждого шага в цепочке треугольный с ненулевыми диагональными элементами, логарифм определителя остается простым. Этот принцип используется обратным авторегрессионным потоком (inverse autoregressive flow, IAF), исследованным Kingma et al. [2016].

Algorithm 2: Computation of unbiased estimate of single-datapoint ELBO for example VAE with a full-covariance Gaussian inference model and a factorized Bernoulli generative model. $\mathbf{L}_{mask}$ is a masking matrix with zeros on and above the diagonal, and ones below the diagonal. Note that $\mathbf{L}$ must be a triangular matrix with positive entries on the diagonal.

Data:
$\mathbf{x}$: a datapoint, and optionally other conditioning information
$\boldsymbol{\epsilon}$: a random sample from $p(\boldsymbol{\epsilon}) = \mathcal{N}(0, \mathbf{I})$
$\boldsymbol{\theta}$: Generative model parameters
$\boldsymbol{\phi}$: Inference model parameters
$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$: Inference model
$p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$: Generative model
Result:
$\tilde{\mathcal{L}}$: unbiased estimate of the single-datapoint ELBO $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$

$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}, \mathbf{L}') \leftarrow \text{EncoderNeuralNet}_{\boldsymbol{\phi}}(\mathbf{x})$
$\mathbf{L} \leftarrow \mathbf{L}_{mask} \odot \mathbf{L}' + \operatorname{diag}(\boldsymbol{\sigma})$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$
$\mathbf{z} \leftarrow \mathbf{L}\boldsymbol{\epsilon} + \boldsymbol{\mu}$
$\text{logqz} \leftarrow -\texttt{sum}(\frac{1}{2}(\boldsymbol{\epsilon}^2 + \log(2\pi) + \log \boldsymbol{\sigma}))$ $\quad \triangleright = q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$
$\text{logpz} \leftarrow -\texttt{sum}(\frac{1}{2}(\mathbf{z}^2 + \log(2\pi)))$ $\quad \triangleright = p_{\boldsymbol{\theta}}(\mathbf{z})$
$\mathbf{p} \leftarrow \text{DecoderNeuralNet}_{\boldsymbol{\theta}}(\mathbf{z})$
$\text{logpx} \leftarrow \texttt{sum}(\mathbf{x} \odot \log \mathbf{p} + (1 - \mathbf{x}) \odot \log(1 - \mathbf{p}))$ $\quad \triangleright = p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$
$\tilde{\mathcal{L}} = \text{logpx} + \text{logpz} - \text{logqz}$

Оценка предельного правдоподобия

После обучения VAE мы можем оценить вероятность данных в рамках модели, используя метод выборки по важности (importance sampling). Предельное правдоподобие точки данных может быть записано как:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}\right]$$

Случайная выборка из $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ дает следующую оценку Монте-Карло для предельного правдоподобия:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) \approx \log\left(\frac{1}{L}\sum_{l=1}^{L}\frac{p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}^{(l)}\right)}{q_{\boldsymbol{\phi}}\left(\mathbf{z}^{(l)} \mid \mathbf{x}\right)}\right),$$

где каждая точка $\mathbf{z}^{(l)} \sim q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ является случайной согласно модели вывода. При больших L аппроксимация становится лучшей оценкой предельного правдоподобия, и фактически, поскольку это оценка методом Монте-Карло, при $L \to \infty$ она сходится к фактическому предельному правдоподобию.

Обратите внимание, что при $L = 1$ эта оценка равна оценке ELBO для VAE.

Предельное правдоподобие и ELBO как KL-дивергенции

Один из способов улучшить потенциальную жесткость границы — повысить гибкость генеративной модели. Это можно понять через связь между ELBO и KL-дивергенцией.

Для набора данных $\mathcal{D}$ размера $N_{\mathcal{D}}$ (с данными i.i.d.) критерий максимального правдоподобия равен:

$$\log p_{\boldsymbol{\theta}}(\mathcal{D}) = \frac{1}{N_{\mathcal{D}}}\sum_{\mathbf{x}\in\mathcal{D}}\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbb{E}_{q_{\mathcal{D}}(\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x})\right],$$

где $q_{\mathcal{D}}(\mathbf{x})$ — эмпирическое распределение (данных), представляющее собой смешанное распределение:

$$q_{\mathcal{D}}(\mathbf{x}) = \frac{1}{N_{\mathcal{D}}}\sum_{i=1}^{N_{\mathcal{D}}} q_{\mathcal{D}}^{(i)}(\mathbf{x}),$$

где каждая компонента $q_{\mathcal{D}}^{(i)}(\mathbf{x})$ обычно соответствует дельта-распределению Дирака с центром в значении $\mathbf{x}^{(i)}$ в случае непрерывных данных или дискретному распределению со всей вероятностной массой,

сосредоточенной в значении $\mathbf{x}^{(i)}$ в случае дискретных данных. Расхождение Кульбака-Лейблера (KL) между данными и распределением модели можно переписать как отрицательное логарифмическое правдоподобие плюс константа:

$$D_{\mathbb{KL}}\left(q_{\mathcal{D}}\left(\mathbf{x}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x}\right)\right) = -\mathbb{E}_{q_{\mathcal{D}}(\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}\left(\mathbf{x}\right)\right] + \mathbb{E}_{q_{\mathcal{D}}(\mathbf{x})}\left[\log q_{\mathcal{D}}\left(\mathbf{x}\right)\right] =$$

$$= -\log p_{\boldsymbol{\theta}}\left(\mathcal{D}\right) + \text{constant},$$

где $\text{constant} = -\mathcal{H}\left(q_{\mathcal{D}}\left(\mathbf{x}\right)\right)$. Таким образом, приведенная выше минимизация KL-дивергенции эквивалентна максимизации логарифмического правдоподобия данных $\log p_{\boldsymbol{\theta}}\left(\mathcal{D}\right)$.

Взяв комбинацию распределения эмпирических данных $q_{\mathcal{D}}\left(\mathbf{x}\right)$ и модели вывода, мы получим совместное распределение по данным $\mathbf{x}$ и скрытым переменным $\mathbf{z}$:

$$q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{x}, \mathbf{z}\right) = q_{\mathcal{D}}\left(\mathbf{x}\right) q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right).$$

KL-дивергенция этого совместного распределения может быть переписана как отрицательный ELBO плюс константа:

$$D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{x}, \mathbf{z}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right)\right) = -\mathbb{E}_{q_{\mathcal{D}}(\mathbf{x})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)\right.\right.$$

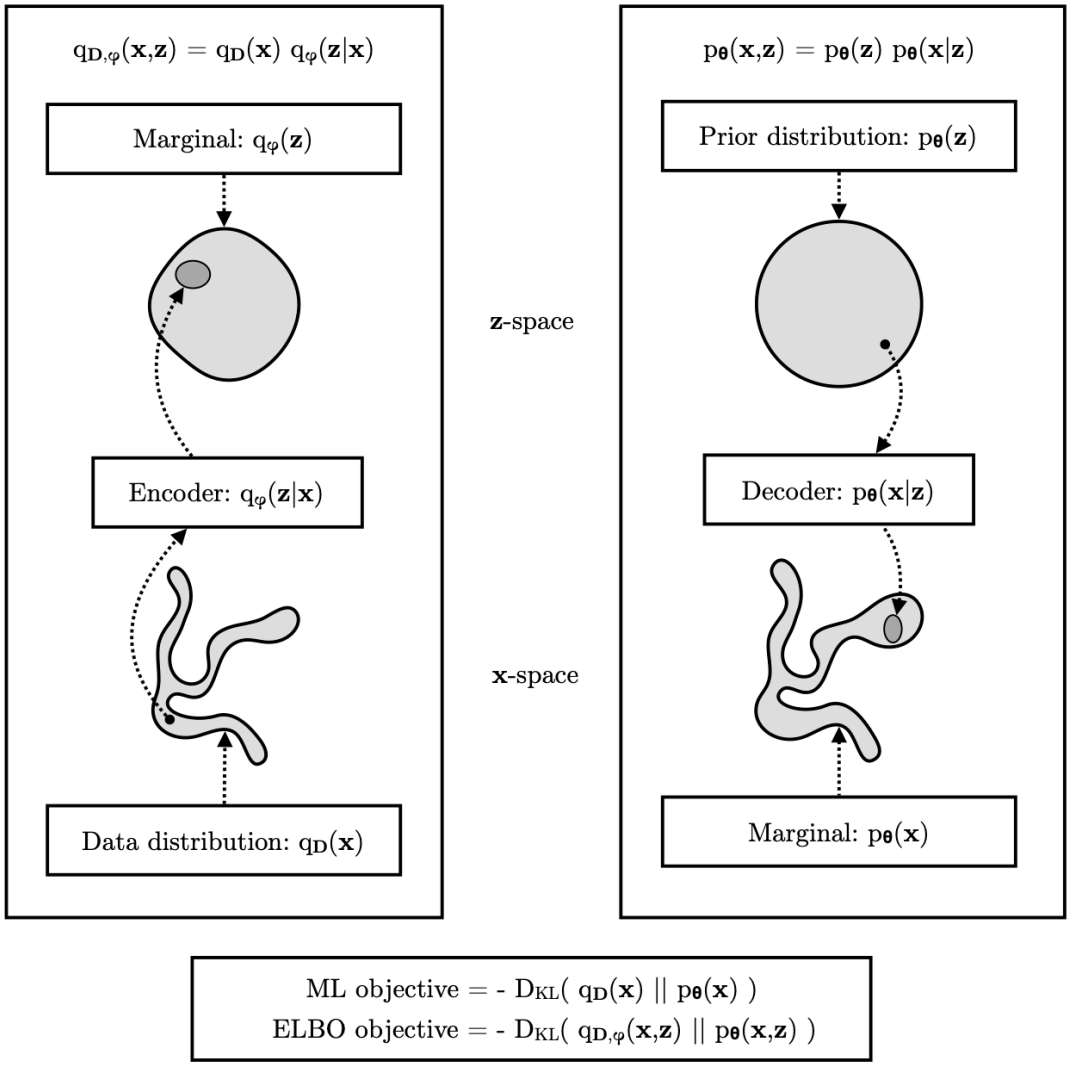
$$= -\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\mathcal{D}\right) + \text{constant},$$

где $\text{constant} = -\mathcal{H}\left(q_{\mathcal{D}}\left(\mathbf{x}\right)\right)$. Таким образом, максимизация ELBO эквивалентна минимизации этой KL-дивергенции $D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{x}, \mathbf{z}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right)\right)$. Взаимосвязь между целевыми функциями ML и ELBO можно резюмировать в следующем простом уравнении:

$$D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{x}, \mathbf{z}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right)\right) = D_{\mathbb{KL}}\left(q_{\mathcal{D}}\left(\mathbf{x}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x}\right)\right) +$$

$$+\mathbb{E}_{q_{\mathcal{D}}(\mathbf{x})}\left[D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{z} \mid \mathbf{x}\right)\right)\right] \geqslant D_{\mathbb{KL}}\left(q_{\mathcal{D}}\left(\mathbf{x}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x}\right)\right)$$

Еще одна точка зрения заключается в том, что ELBO можно рассматривать как целевую функцию максимального правдоподобия в расширенном пространстве. Для некоторого фиксированного выбора кодировщика $q_{\boldsymbol{\phi}}\left(\mathbf{z} \mid \mathbf{x}\right)$ мы можем рассматривать совместное распределение $p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right)$ как расширенное эмпирическое распределение по исходным данным $\mathbf{x}$ и (стохастическим) вспомогательным признакам $\mathbf{z}$, связанным с каждой точкой данных. Затем модель $p_{\boldsymbol{\theta}}\left(\mathbf{x}, \mathbf{z}\right)$ определяет совместную модель распределения исходных данных и вспомогательных признаков.



Целевую функцию максимального правдоподобия (ML) можно рассматривать как минимизацию $D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{x}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x}\right)\right)$, в то время как целевую функцию ELBO можно рассматривать как минимизацию $D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{x},\mathbf{z}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x},\mathbf{z}\right)\right)$, который сверху ограничивает $D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{x}\right) \parallel p_{\boldsymbol{\theta}}\left(\mathbf{x}\right)\right)$. Если идеальное обучение не возможно, то $p_{\boldsymbol{\theta}}\left(\mathbf{x},\mathbf{z}\right)$ обычно будет иметь более высокую дисперсию, чем $q_{\mathcal{D},\boldsymbol{\phi}}\left(\mathbf{x},\mathbf{z}\right)$, из-за направления KL-дивергенции.

In [ ]: