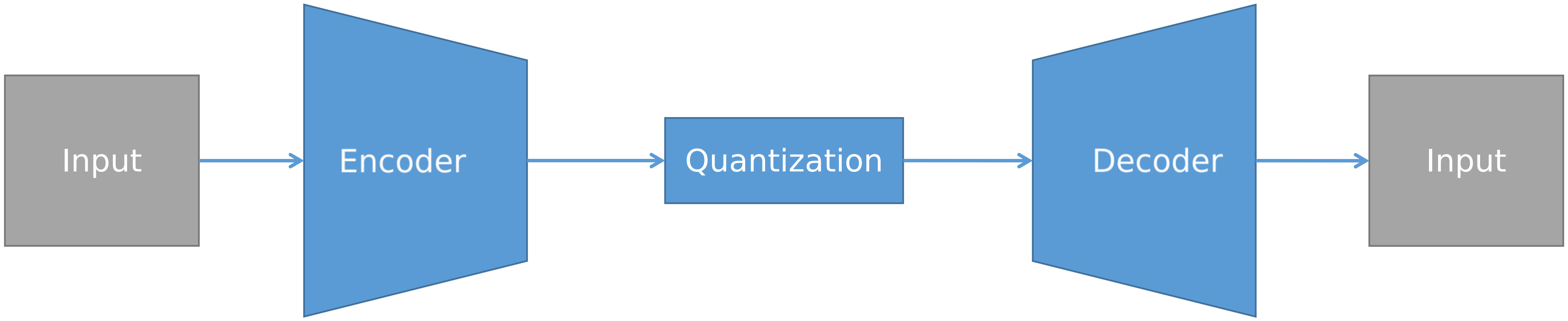
VQ-VAE

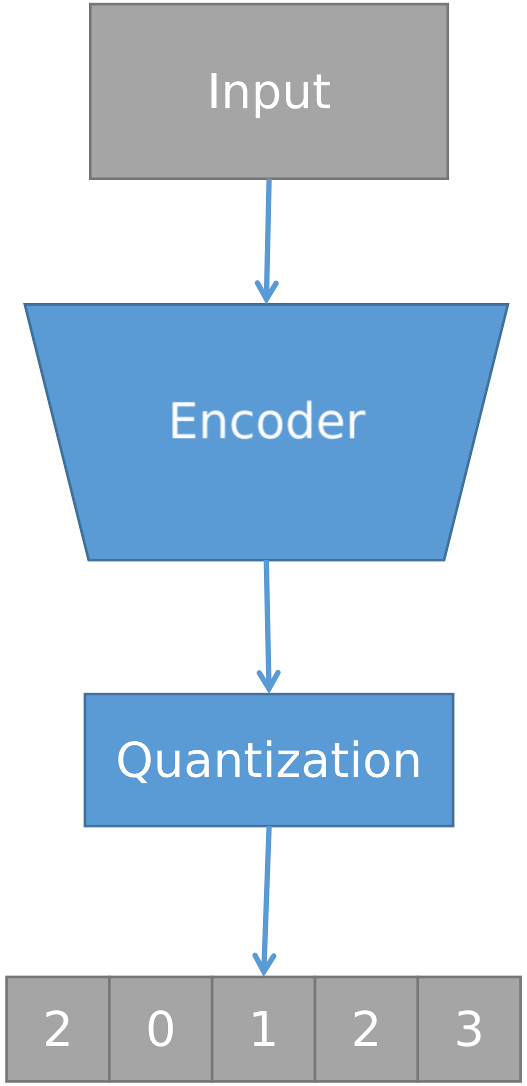
Vector Quantized Variational Autoencoder

A **Vector Quantized Variational Autoencoder** is a type of neural network **architecture** that combines elements of **variational autoencoders** (VAEs) and **vector quantization** to learn **discrete** representations of data.

It is particularly **useful** for applications where **discrete** or **categorical representations** are **beneficial**



Vector Quantized Variational Autoencoder



Similar to a traditional VAE, **VQ-VAE** has an **encoder** network that takes **input** data and maps it to a **continuous latent space**.

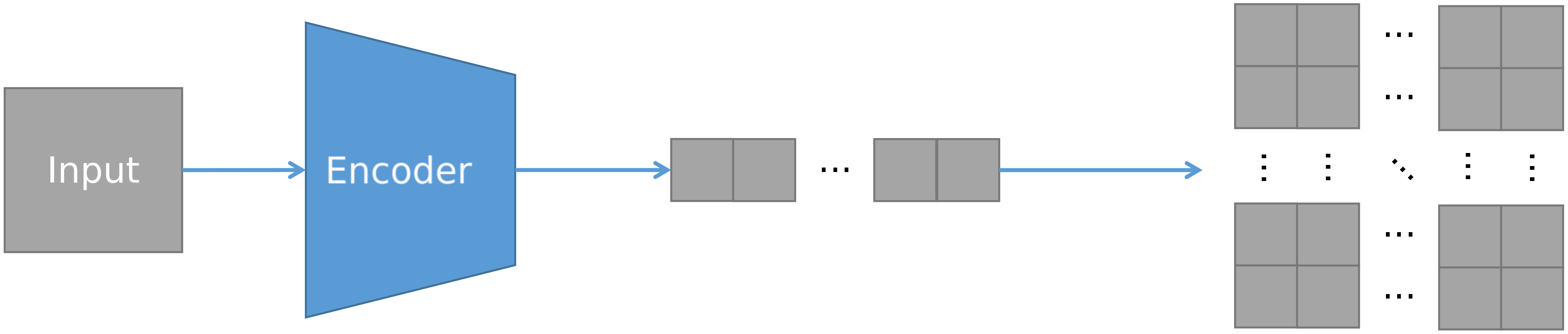
However, unlike a standard VAE, the **encoder** in **VQ-VAE** does **not** output **continuous values**.

Instead, it **outputs discrete** codes that **correspond** to specific entries in a **codebook**.

Entry 0	0.15	0.71	0.34	0.31	0.19	0.75
Entry 1	0.17	0.76	0.82	0.17	0.82	0.84
Entry 2	0.89	0.92	0.56	0.72	0.43	0.94
Entry 3	0.38	0.75	0.90	0.17	0.06	0.69

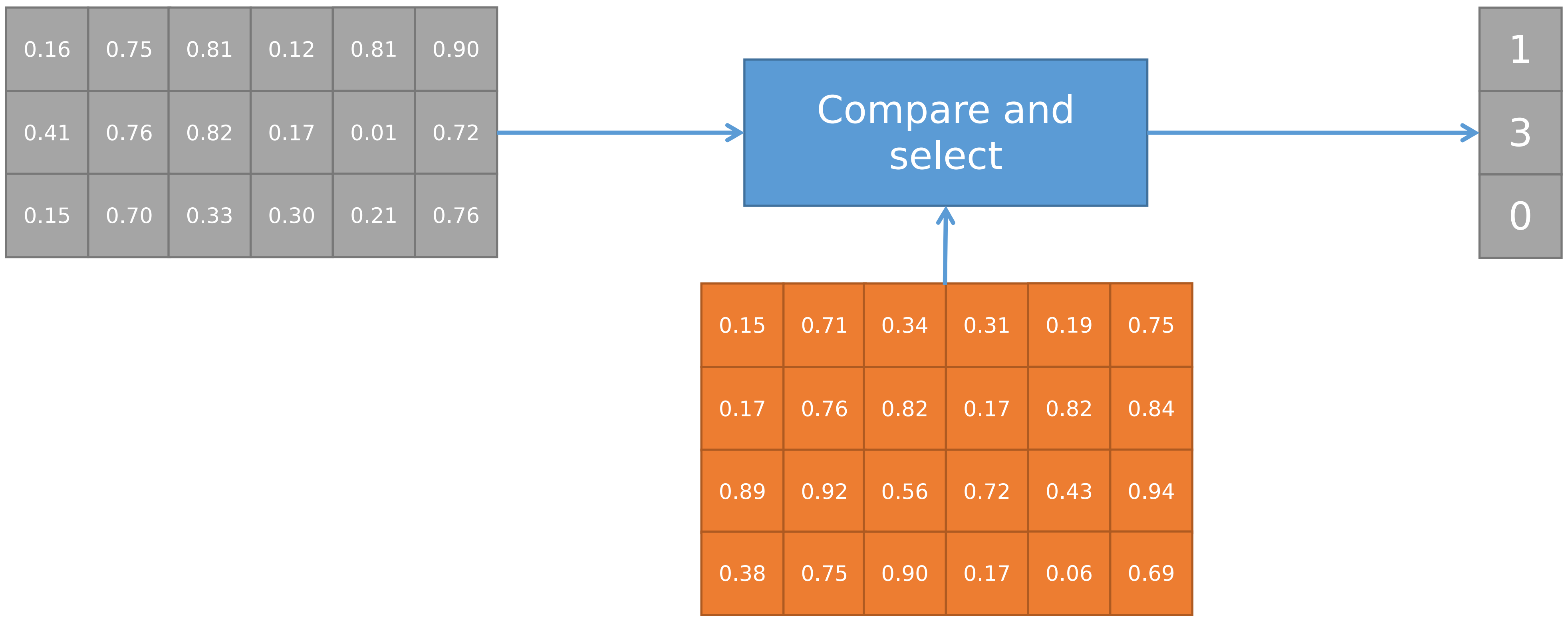
VQ-VAE Encoder

The **continuous representation** from the **encoder** is **reshaped** into a **matrix** with the same number of **columns** as the **dimension** of the **codebook** and the same number of **rows** as the **desired number** of **codes** for encoding.



VQ-VAE Encoder: Quantization (1)

Each **row** of this matrix is **compared** with the **rows** of the **codebook**, and the **closest matching** codebook entry is **selected** based on the **distance** between the **rows**.



VQ-VAE Encoder: Quantization (2)

0.16	0.75	0.81	0.12	0.81	0.90
0.41	0.76	0.82	0.17	0.01	0.72
0.15	0.70	0.33	0.30	0.21	0.76

$$\begin{array}{|c|c|c|c|}\hline 0.80 & 0.08 & 1.06 & 0.81 \\ \hline 0.59 & 0.85 & 0.92 & 0.10 \\ \hline 0.02 & 0.80 & 0.96 & 0.65 \\ \hline \end{array} \xrightarrow{\text{argmin}} \begin{array}{|c|}\hline 1 \\ \hline 3 \\ \hline 0 \\ \hline \end{array}$$

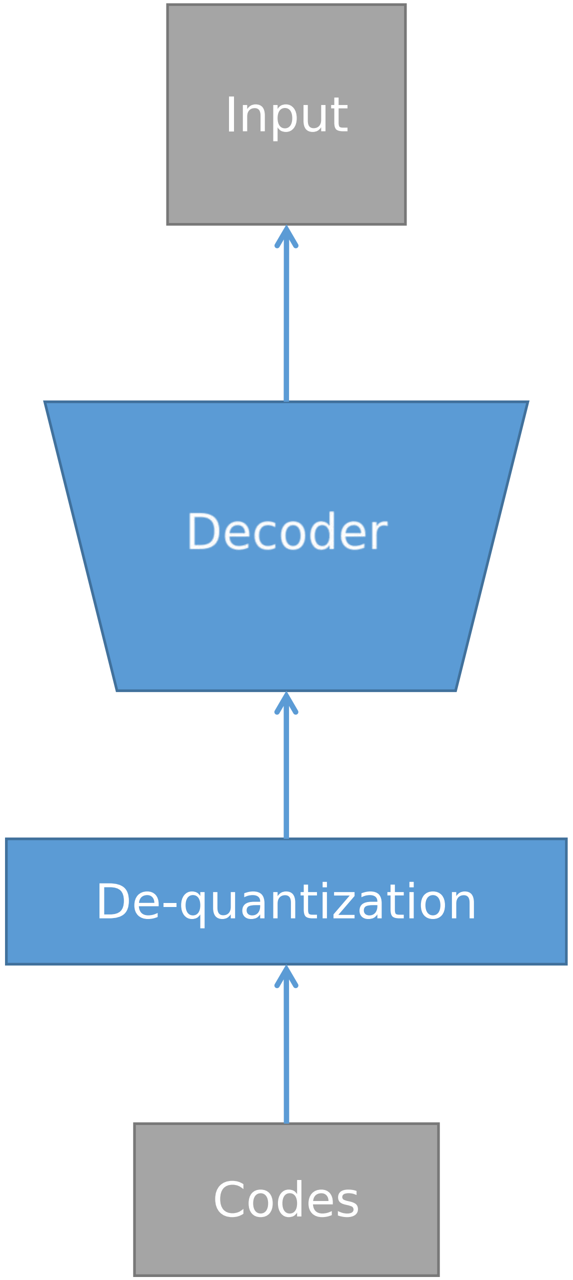
0.15	0.17	0.89	0.38
0.71	0.76	0.92	0.75
0.34	0.82	0.56	0.90
0.31	0.17	0.72	0.17
0.19	0.82	0.43	0.06
0.75	0.84	0.94	0.69

In **PyTorch**, the function used to calculate the **distance matrix between** two **matrices** is called **cdist**

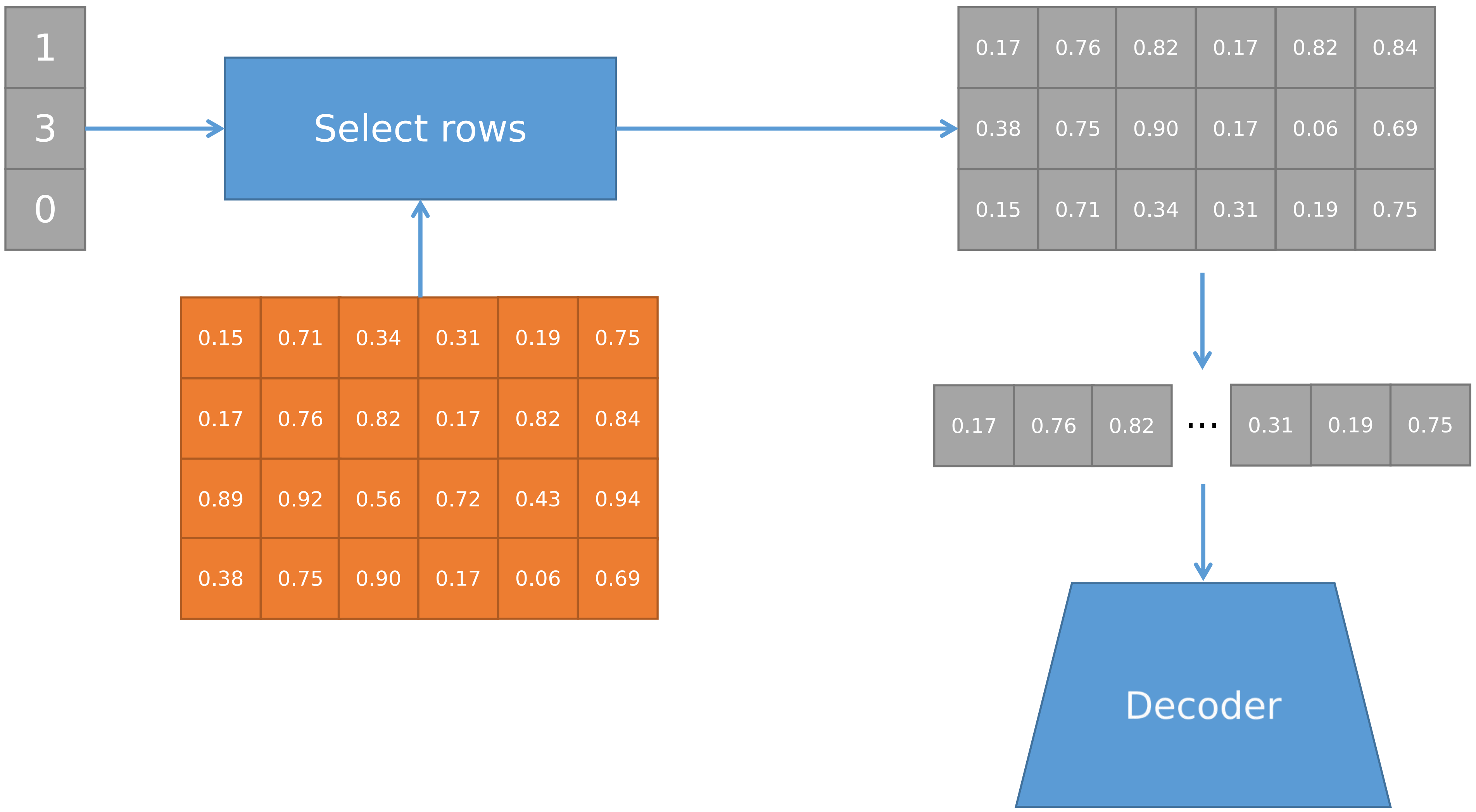
VQ-VAE Decoder

The **decoder** is responsible for **generating** data samples **from discrete** codes **produced** by the **encoder** and codebook.

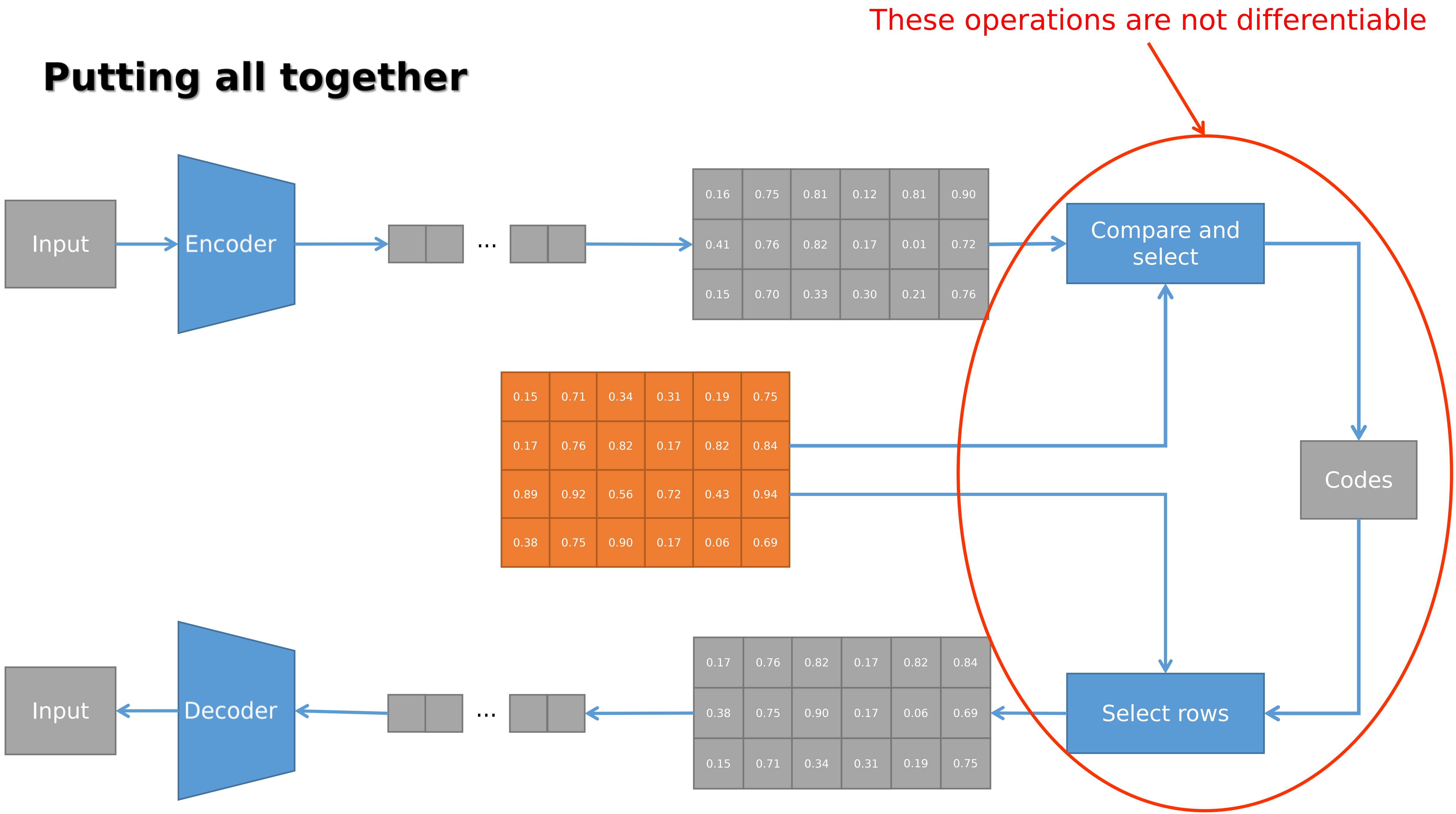
It **takes** these **discrete codes**, **looks up** corresponding entries in the **codebook**, and **maps** them back **to** the **original** data **space** to reconstruct or generate data points.



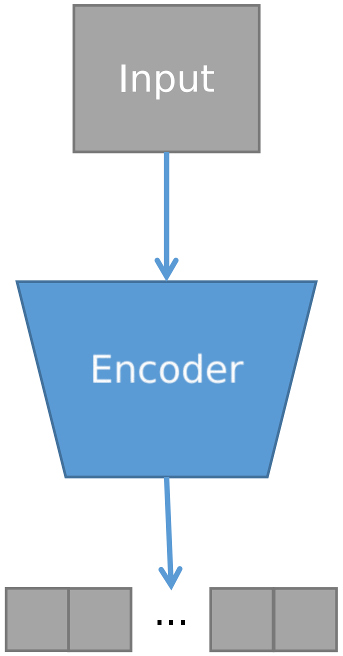
VQ-VAE Decoder



Putting all together



VQVAE: quantization trick



x

0.16	0.75	0.81	0.12	0.81	0.90
0.41	0.76	0.82	0.17	0.01	0.72
0.15	0.70	0.33	0.30	0.21	0.76

x is produced by the encoder

q

0.17	0.76	0.82	0.17	0.82	0.84
0.38	0.75	0.90	0.17	0.06	0.69
0.15	0.71	0.34	0.31	0.19	0.75

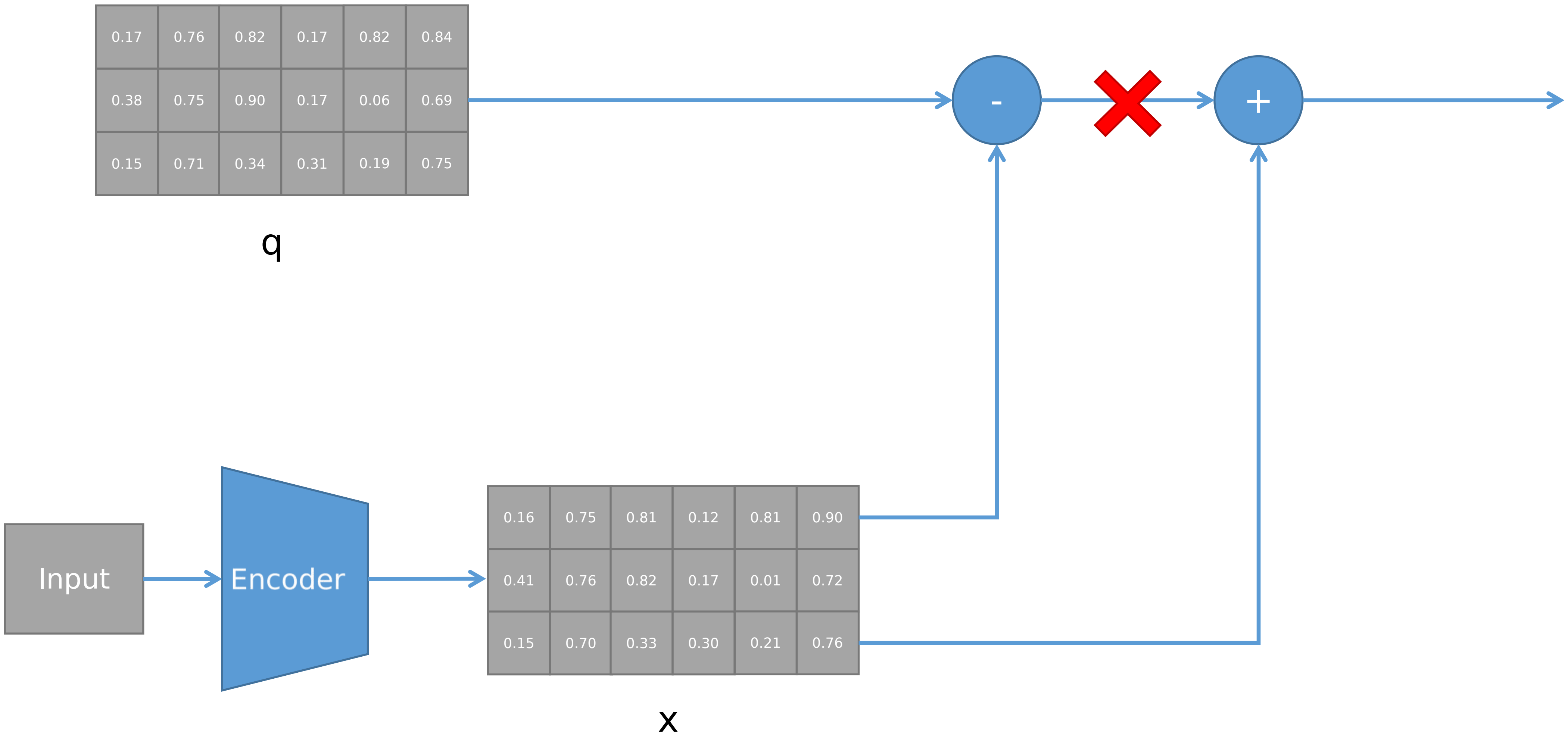
q is composed by elements of the codebook

0.15	0.71	0.34	0.31	0.19	0.75
0.17	0.76	0.82	0.17	0.82	0.84
0.89	0.92	0.56	0.72	0.43	0.94
0.38	0.75	0.90	0.17	0.06	0.69

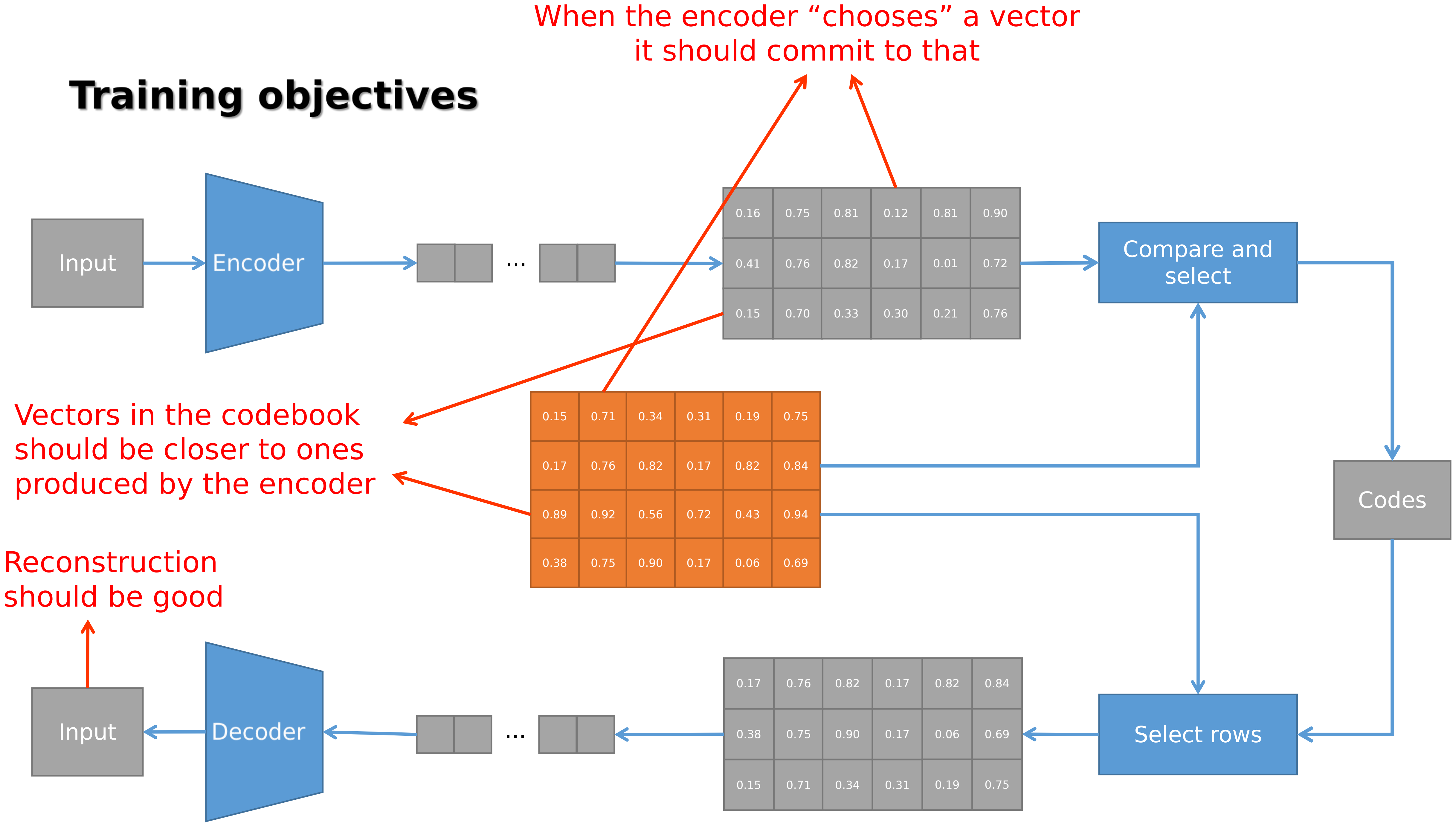
We want to transform x into q while retaining a path to the encoder in the CG

VQVAE: quantization trick

(q - x).detach() + x



Training objectives



Training objectives

- **Reconstruction loss**
  - **Trains** the **model** (**encoder and decoder**) to **minimize** the **discrepancy** between the **original input** data and the **reconstructed output** data, encouraging the model to **learn** a meaningful and **faithful** representation of the input in the latent space.
- **Codebook loss**
  - **Trains** the **codebook** to have **vectors closer** to the **ones produced** by the **encoder**, promoting the creation of a **codebook** that captures the essential information in the data and encourages the **model** to utilize the **codebook** entries **effectively** during the quantization proces
- **Commitment loss**
  - **Trains** the **encoder** to produce **vectors closer** to the **ones** in the **codebook**, **ensuring** that each data point's **latent** representation is strongly **associated** with a single **codebook** entry, thus improving the efficiency of the learned representations.

VQVAE: PyTorch

```
codebook = torch.tensor([
    [0.15801739, 0.71236613, 0.34401087, 0.31100241, 0.19731029, 0.75124264],
    [0.17008199, 0.76689422, 0.82562077, 0.17558232, 0.82136028, 0.84026041],
    [0.89155776, 0.92834241, 0.56760302, 0.72009988, 0.43676168, 0.94838489],
    [0.38546037, 0.75875292, 0.90014871, 0.17816013, 0.06095272, 0.69844905]
])

vectors = torch.tensor([[
      [0.16, 0.75, 0.81, 0.12, 0.81, 0.90],
      [0.41, 0.76, 0.82, 0.17, 0.01, 0.72],
      [0.15, 0.70, 0.33, 0.30, 0.21, 0.76]
  ],
  [
      [0.50, 0.72, 0.35, 0.13, 0.98, 0.52],
      [0.61, 0.54, 0.73, 0.28, 0.76, 0.57],
      [0.67, 0.12, 0.36, 0.55, 0.73, 0.71]
]])
```

```
print(codes)
print(quantized)
print(vq)
```

```
tensor([[1, 3, 0],
        [1, 1, 2]])
tensor([[[0.1701, 0.7669, 0.8256, 0.1756, 0.8214, 0.8403],
         [0.3855, 0.7588, 0.9001, 0.1782, 0.0610, 0.6984],
         [0.1580, 0.7124, 0.3440, 0.3110, 0.1973, 0.7512]],

        [[0.1701, 0.7669, 0.8256, 0.1756, 0.8214, 0.8403],
         [0.1701, 0.7669, 0.8256, 0.1756, 0.8214, 0.8403],
         [0.8916, 0.9283, 0.5676, 0.7201, 0.4368, 0.9484]]])
tensor([[[0.1701, 0.7669, 0.8256, 0.1756, 0.8214, 0.8403],
         [0.3855, 0.7588, 0.9001, 0.1782, 0.0610, 0.6984],
         [0.1580, 0.7124, 0.3440, 0.3110, 0.1973, 0.7512]],

        [[0.1701, 0.7669, 0.8256, 0.1756, 0.8214, 0.8403],
         [0.1701, 0.7669, 0.8256, 0.1756, 0.8214, 0.8403],
         [0.8916, 0.9283, 0.5676, 0.7201, 0.4368, 0.9484]]])
```

```
distances = torch.cdist(vectors, codebook)
codes = distances.argmin(dim=-1)

quantized = codebook[codes]
vq = (quantized - vectors).detach() + vectors
```

VQVAE: PyTorch

```python
import torch

class VectorQuantize(torch.nn.Module):
    def __init__(self, *, codebook_size, dim):
        super(VectorQuantize, self).__init__()
        self.codebook_size = codebook_size
        self.dim = dim

        self.codebook = torch.nn.Parameter(torch.rand(codebook_size, dim, requires_grad=True))

    def forward(self, x):
        # compute distances between x and codebook
        # compute codes as argmin of distances
        # select quantized vectors from codebook
        # compute vq using the quantization trick

        commit_loss = torch.nn.functional.mse_loss(x, quantized.detach())
        codebook_loss = torch.nn.functional.mse_loss(x.detach(), quantized)

        loss = commit_loss + codebook_loss

        return vq, codes, loss
```

vector-quantize-pytorch library

vector-quantize-pytorch is a **Python library** and implementation that facilitates the use of **Vector Quantized Variational Autoencoders** (VQ-VAEs) in **PyTorch.**
It contains the **implementation** of **various vector quantization methods**:

- **Residual VQ**: Implements a method from a paper where multiple vector quantizers recursively quantize the residuals of the waveform
- **RQ-VAE**: Another paper's approach uses Residual-VQ to generate high-resolution images with more compressed codes. It introduces sharing the codebook across all quantizers and stochastically sampling codes rather than always taking the closest match
- **Grouped Residual VQ**: A recent paper proposes doing residual VQ on groups of the feature dimension, achieving results comparable to Encodec with fewer codebooks
- **KMeans Initialization**: The SoundStream paper suggests initializing the codebook with the kmeans centroids of the first batch, a feature available in both VectorQuantize and ResidualVQ classes

vector-quantize-pytorch library

To use the VectorQuantize module:

```
import torch
from vector_quantize_pytorch import VectorQuantize

vq = VectorQuantize(
    dim = 256,
    codebook_size = 512,
    decay = 0.8,
    commitment_weight = 1.0
)

x = torch.randn(8, 1024, 256)
quantized, indices, vq_loss = vq(x)
print("quantized:", quantized.shape)
print("indices:", indices.shape)
```

To install vector-quantize-pytorch:

```
$ pip install vector-quantize-pytorch
```

You dont need to worry about commitment loss and codebook loss

Exercise 1

- Train a **VQ-VAE** on the **MNIST** dataset using the **vector-quantize-pytorch** implementaion of the quantization layer
- Use **MLP** as encoder and decoder
- **Plot** some **inputs** and their **reconstruction**
- **Plot** some **generated** images from **random** codes

```
self.encoder = torch.nn.Sequential(
    torch.nn.Linear(input_dim, 300),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(300, 300),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(300, vector_dim*num_embeddings),
)
```

```
self.decoder = torch.nn.Sequential(
    torch.nn.Linear(vector_dim*num_embeddings, 300),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(300, 300),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(300, input_dim),
    torch.nn.Sigmoid()
)
```

```
self.vq = VectorQuantize(
    dim = vector_dim,
    codebook_size = codebook_dim,
    decay = 0.8,
    commitment_weight = 1.0
)
```

Exercise 2

- Train a **VQ-VAE** on the **MNIST** dataset <u>**WITHOUT**</u> using the **vector-quantize-pytorch** implementaion of the quantization layer
- Implement your own **VectorQuantize** class
- Use **MLP** as encoder and decoder
- **Plot** some **inputs** and their **reconstruction**
- **Plot** some **generated** images from **random** codes