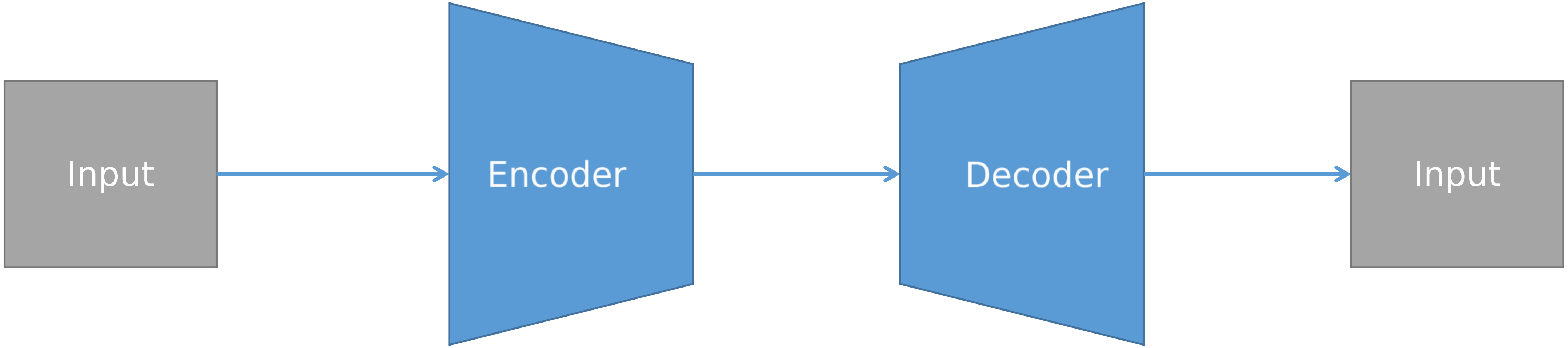
# Autoencoders

# Simple Autoencoder (AE)

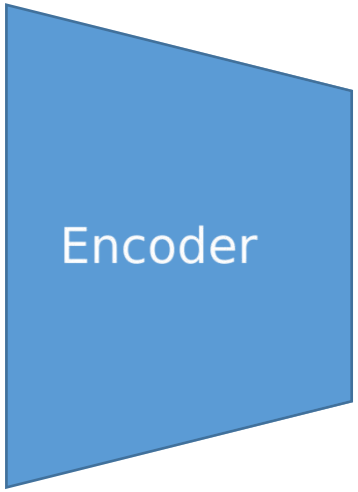
An **autoencoder is** a type of artificial **neural network** designed for **unsupervised learning**.

It consists of **two** main **components**: an **encoder** and a **decoder**. The primary **goal** of an **autoencoder** is to **learn** a **compact representation** of input data, typically for the purpose of **data compression**, **feature learning**, or **denoising**.



# Autoencoder Training

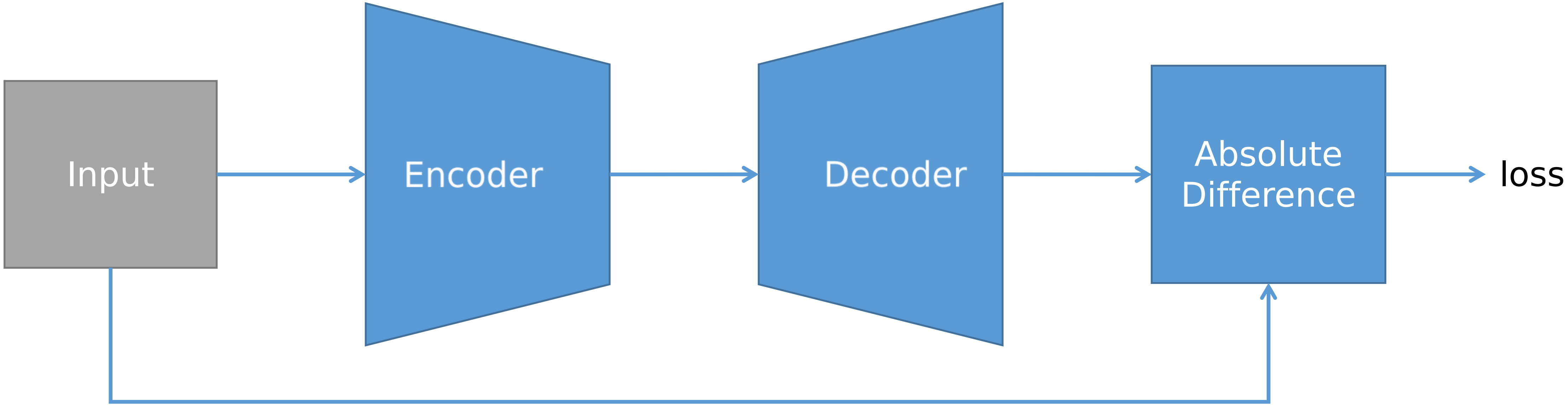
The **encoder** part of the autoencoder **compresses** the **input data** into a **lower-dimensional** representation, often referred to as the "**latent space**" or "**encoding**."




The **decoder reconstructs** the original **input** data **from** the **compressed** representation **produced** by the **encoder**. The **reconstructed** output should ideally be a close **approximation** of the original **input**.

# Autoencoder Training

By **training** the **autoencoder** on a dataset and **minimizing** the **reconstruction error**,  the model **learns** to **capture** the **most salient features** of the data in the latent space



# Simple Autoencoder (AE) in PyTorch

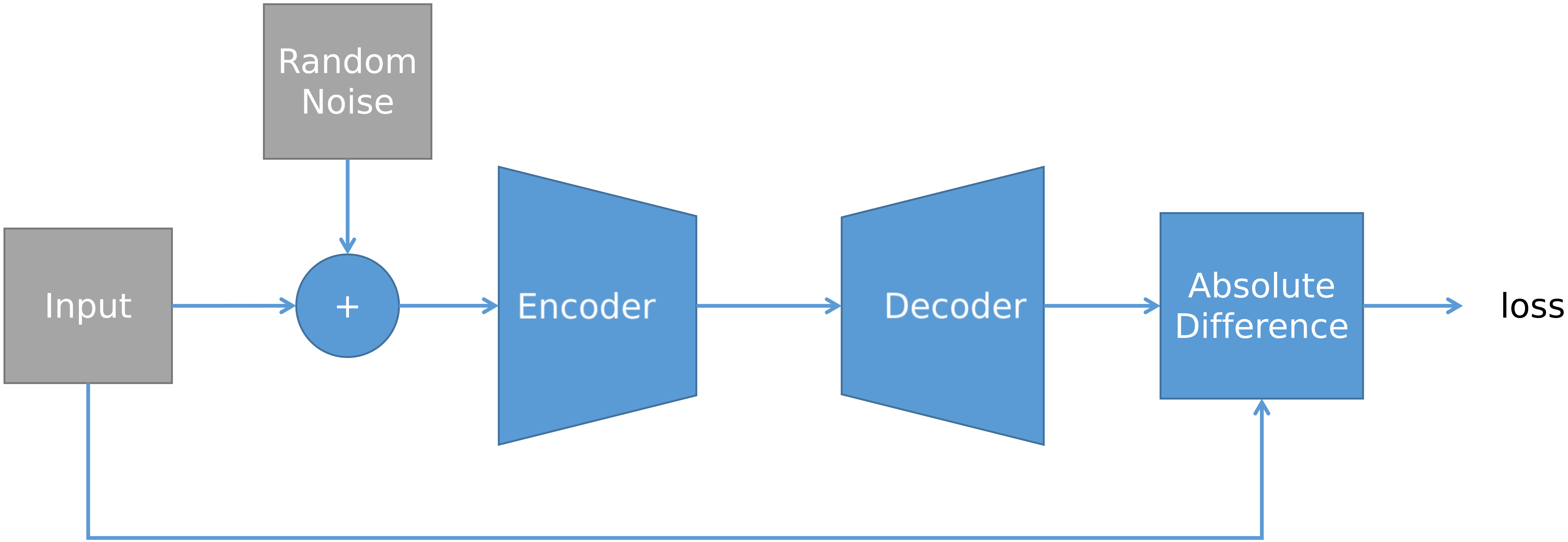
```
class Autoencoder(torch.nn.Module):
  def __init__(self, input_dim, bottleneck_dim):
    super(Autoencoder, self).__init__()
    self.encoder = torch.nn.Sequential(
        torch.nn.Linear(input_dim, 300),
        torch.nn.LeakyReLU(),
        torch.nn.Linear(300, 300),
        torch.nn.LeakyReLU(),
        torch.nn.Linear(300, bottleneck_dim),
    )

    self.decoder = torch.nn.Sequential(
        torch.nn.Linear(bottleneck_dim, 300),
        torch.nn.LeakyReLU(),
        torch.nn.Linear(300, 300),
        torch.nn.LeakyReLU(),
        torch.nn.Linear(300, input_dim),
    )

  def forward(self, x):
    latent = self.encoder(x)
    reconstructed = self.decoder(latent)
    return reconstructed
```

```
ae = Autoencoder(784, 10)
loss_fn = torch.nn.L1Loss()
optimizer = torch.optim.Adam(ae.parameters(), lr=0.001)

for batch in dataset:
  reconstructed = ae(batch)
  error = loss_fn(reconstructed, batch)

  optimizer.zero_grad()
  error.backward()
  optimizer.step()
```

# Denoising Autoencoder (DAE)

A **denoising autoencoder** is a **variation** of the traditional autoencoder **designed** to **handle noisy input** data.

The primary **objective** of a denoising autoencoder is to **learn** a **robust representation** of the underlying structure in **data by removing noise**



# Denoising Autoencoder (DAE) in PyTorch

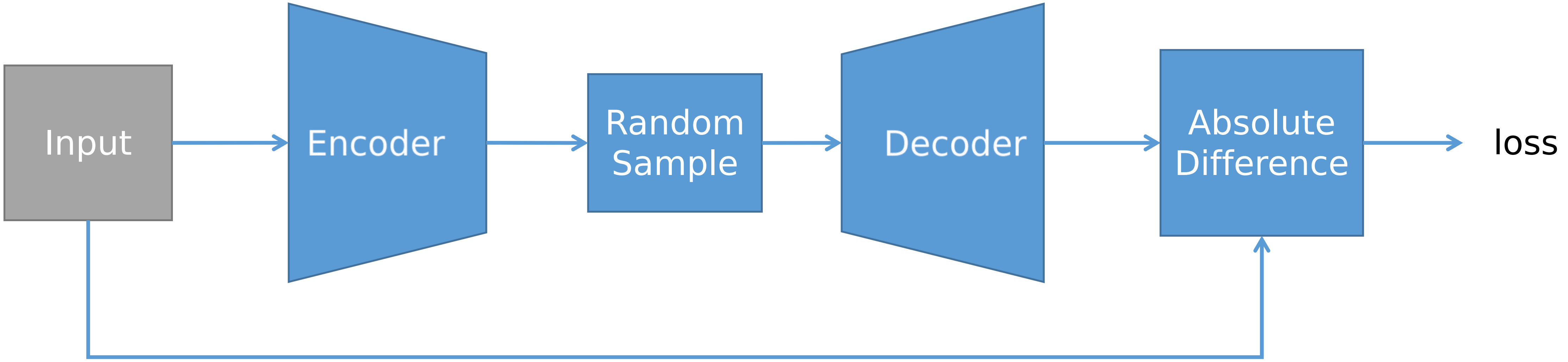
```
def forward(self, x):
  noisy_input = x + torch.randn(*x.shape)
  latent = self.encoder(noisy_input)
  reconstructed = self.decoder(latent)
  return reconstructed
```

```
dae = DenoisingAutoencoder(784, 10)
loss_fn = torch.nn.L1Loss()
optimizer = torch.optim.Adam(dae.parameters(), lr=0.001)

for batch in dataset:
  reconstructed = dae(batch)
  error = loss_fn(reconstructed, batch)

  optimizer.zero_grad()
  error.backward()
  optimizer.step()
```

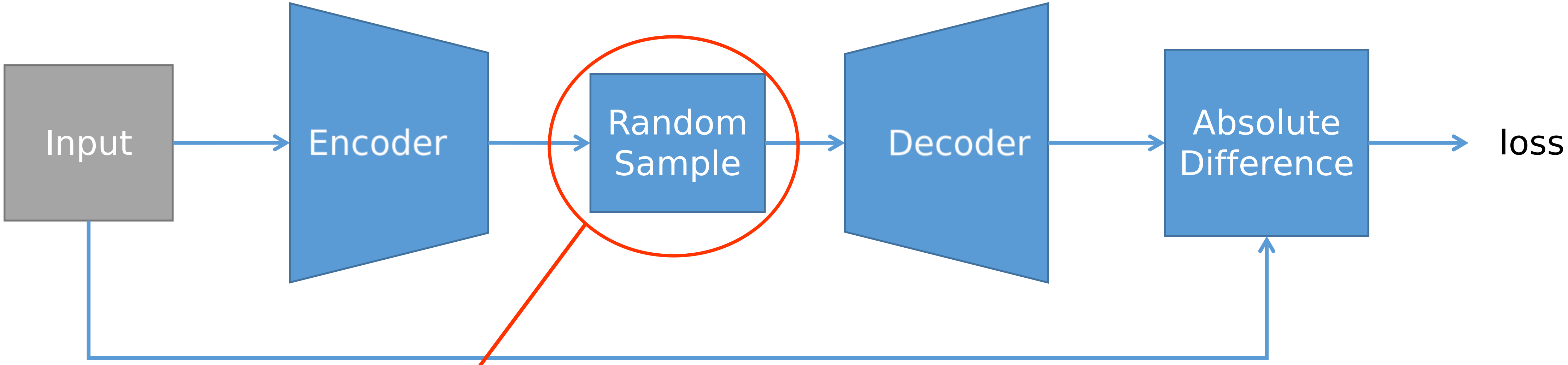
# Variational Autoencoder (VAE)

VAEs are designed to not only learn a compressed representation of input data but also to generate new data points from that learned representation.

Unlike a regular autoencoder that produces a deterministic encoding, VAEs adopt a probabilistic approach. The encoder generates a stochastic compressed representation of the input data.



# Variational Autoencoder (VAE)




This is not a differentiable operation!
How can i run the backpropagation?

# Variational Autoencoder (VAE): Random Sample

```
import torch

a_tensor = torch.tensor([1, 10 ,100], dtype=float, requires_grad=True)

b_tensor = a_tensor*torch.tensor([1, 2, 3], dtype=float)
b_sum = b_tensor.sum()
print("b_tensor:", b_tensor)
print("b_sum:", b_sum)
b_sum.backward()
print("a_tensor.grad:", a_tensor.grad)
```

```
b_tensor: tensor([  1.,  20., 300.], dtype=torch.float64, grad_fn=<MulBackward0>)
b_sum: tensor(321., dtype=torch.float64, grad_fn=<SumBackward0>)
a_tensor.grad: tensor([1., 2., 3.], dtype=torch.float64)
```

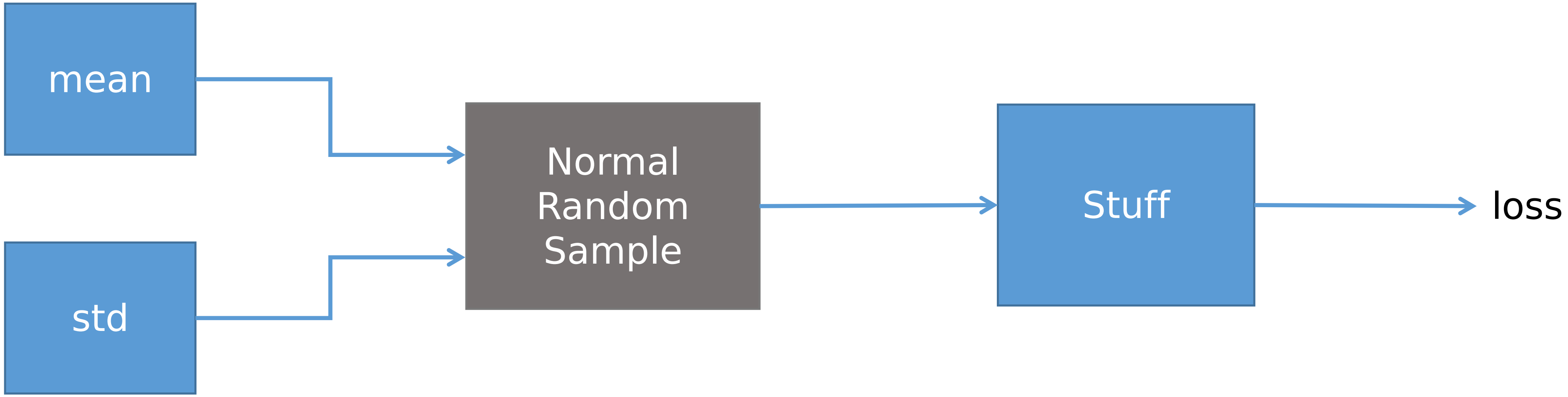
This works fine. * and sum() are differentiable operations

# Variational Autoencoder (VAE): Random Sample

```python
import torch
import numpy as np

a_tensor = torch.tensor([1, 10 ,100], dtype=float, requires_grad=True)

b_tensor = random_sample(mu=a_tensor)
b_sum = b_tensor.sum()
print("b_tensor:", b_tensor)
print("b_sum:", b_sum)
b_sum.backward()
print("a_tensor.grad:", a_tensor.grad)
```

```
    250     # calls in the traceback and some print out the last line
--> 251     Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pas
    252         tensors,
    253         grad_tensors_,

RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn
```

# Variational Autoencoder (VAE): Random Sample CG




We want to **compute** the **gradients** of our **loss w.r.t.** the **mean** and the **std**.
But the **Normal Random Sample** is a **non differentiable** operation.

Is there a way out?

# Variational Autoencoder (VAE): Random Sample CG

Yes. There is a way to do that.

A random variable $z \sim N(\mu, \sigma)$ can be transformed into $\overline{z} \sim N(0, 1)$ with:

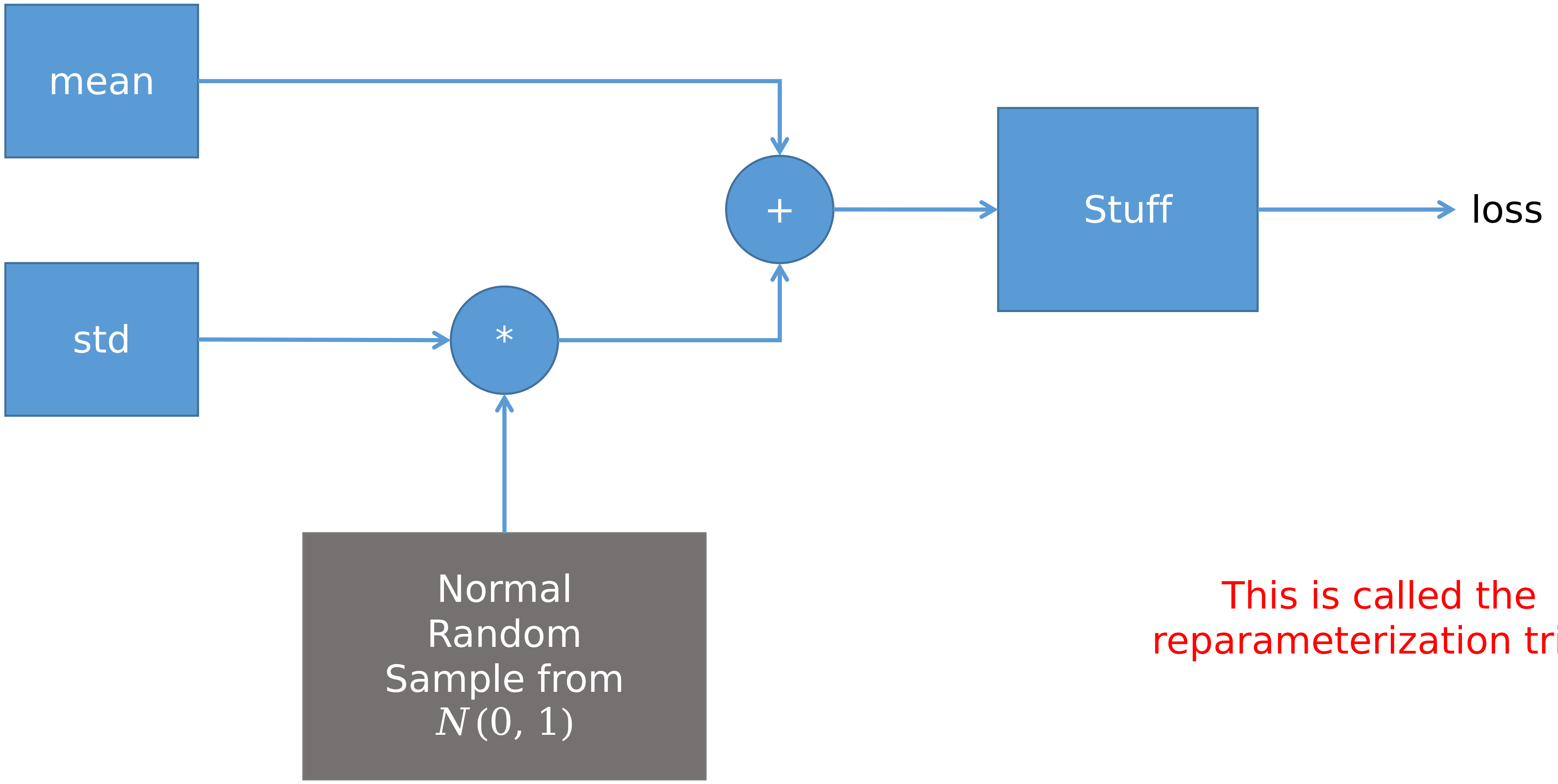
$$\overline{z} = \frac{z - \mu}{\sigma}$$

In the same way a random variable $\overline{z} \sim N(0, 1)$ can be transformed into a $z \sim N(\mu, \sigma)$ with:

$$z = \overline{z}\sigma + \mu$$

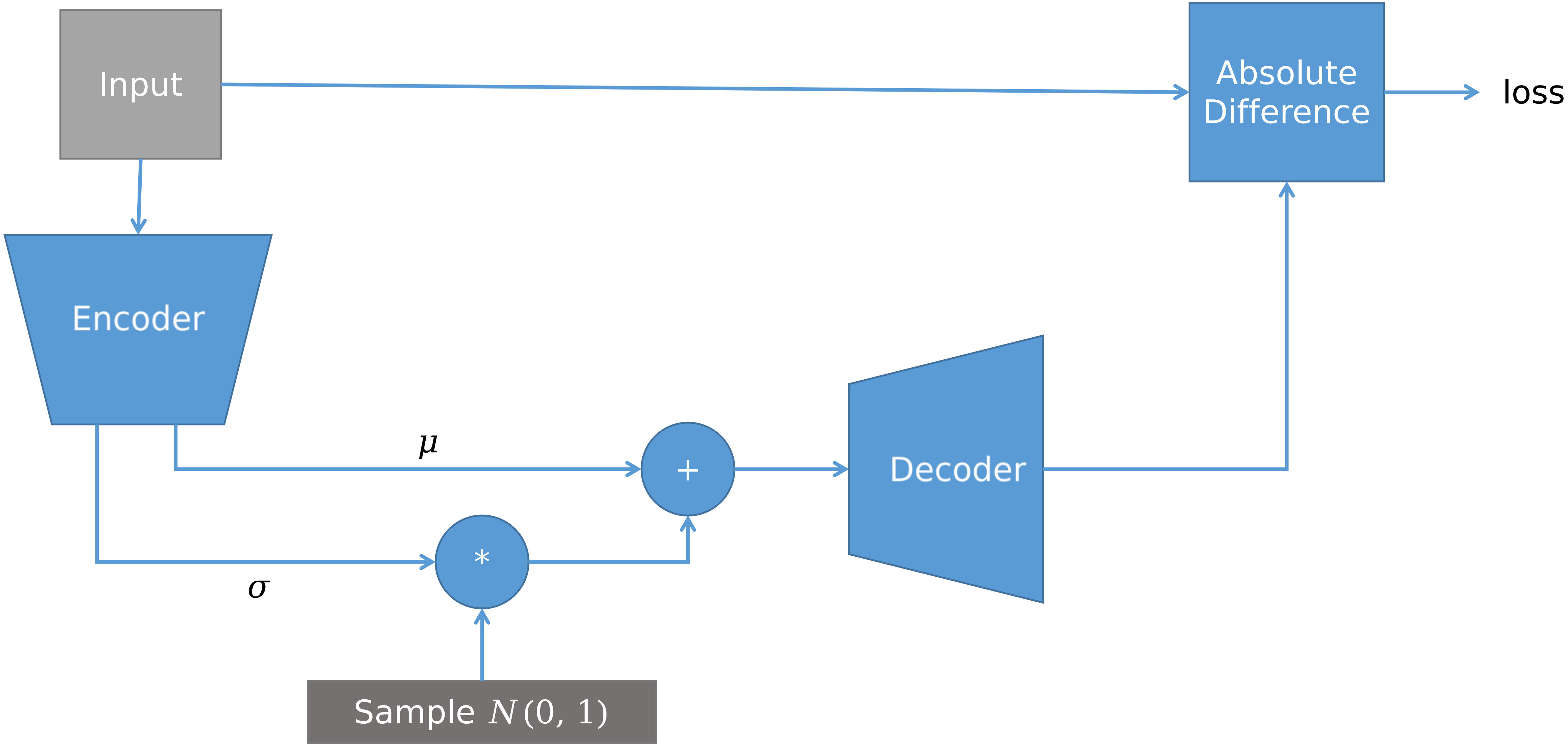
And both multiplication and addition are differentiable operations!

# Variational Autoencoder (VAE): Random Sample CG




This is called the reparameterization trick

# Variational Autoencoder (VAE): Putting all together

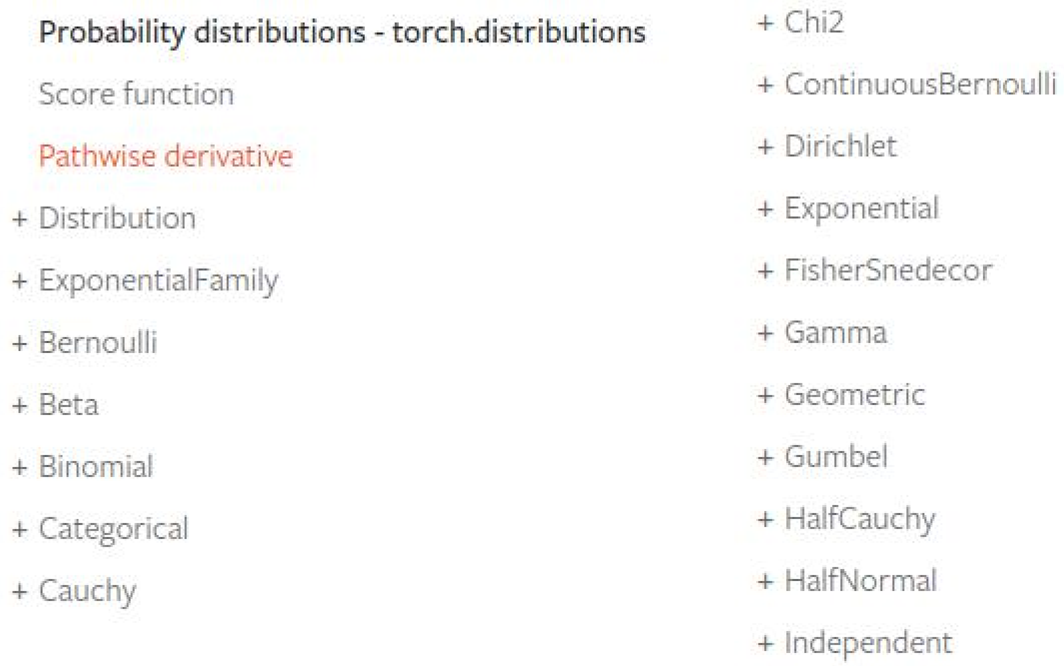


# PyTorch: Distributions

The **distributions** package contains **parameterizable** probability **distributions** and **sampling functions**.

This **allows** the construction of **stochastic computation graphs** and stochastic gradient estimators for optimization

Every **distribution** has the **sample()** function to compute a new sample. **Reparameterizable** distributions, on the other hand, possess a special **function** called **rsample()** that employs the **reparametrization trick** to compute a new sample.


Probability distributions - torch.distributions

Score function

Pathwise derivative

+ Distribution
+ ExponentialFamily
+ Bernoulli
+ Beta
+ Binomial
+ Categorical
+ Cauchy
+ Chi2
+ ContinuousBernoulli
+ Dirichlet
+ Exponential
+ FisherSnedecor
+ Gamma
+ Geometric
+ Gumbel
+ HalfCauchy
+ HalfNormal
+ Independent

# Variational Autoencoder (VAE): In PyTorch

```python
class VariationalAutoencoder(torch.nn.Module):
    def __init__(self, input_dim, bottleneck_dim):
        super(VariationalAutoencoder, self).__init__()

        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(input_dim, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, 300),
            torch.nn.LeakyReLU(),
        )

        self.mu_head = torch.nn.Linear(300, bottleneck_dim)
        self.std_head = torch.nn.Linear(300, bottleneck_dim)

        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(bottleneck_dim, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, 300),
            torch.nn.LeakyReLU(),
            torch.nn.Linear(300, input_dim),
        )
```

```python
def encode(self, x):
    x = self.encoder(x)

    mu = self.mu_head(x)
    std = torch.abs(self.std_head(x))

    return mu, std

def forward(self, x):
    mu, std = self.encode(x)

    p = distributions.Normal(mu, std)
    z = p.rsample()

    return self.decoder(z)
```

# VAE Training Regularization: KL-divergence

The K**ullback-Leibler** divergence (**KL**-divergence), is a **measure** of how **one** probability **distribution diverges** from a **second**, expected probability **distribution**.

It **quantifies** the **difference between two** probability **distributions**.

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

In a **Variational Autoencoder** (VAE), the addition of the **KL-divergence** term **serves** as a **regularization** mechanism during training.

In this context **KL-divergence** is **used** to **encourage** the learned **latent distribution** to **approximate** a **uncorrelated multivariate distribution**

# VAE Training Regularization: KL-divergence

$$\boldsymbol{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \\ \vdots \\ \mu_k \end{bmatrix}, \quad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1^2 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & 0 & \cdots & 0 \\ 0 & 0 & \sigma_3^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \sigma_k^2 \end{bmatrix}$$

We want our learned representation to be like this

$$\mathrm{tr}\left(\boldsymbol{\Sigma}\right) = \sum_{i=1}^{k} \sigma_i^2$$

$$\boldsymbol{\mu}^T \boldsymbol{\mu} = \sum_{i=1}^{k} \mu_i^2$$

$$k = \sum_{i=1}^{k} 1$$

$$\log\left(\det\left(\boldsymbol{\Sigma}\right)\right) = \log\left(\prod_{i=1}^{k} \sigma_i^2\right) = \sum_{i=1}^{k} \log\left(\sigma_i^2\right)$$

$$\frac{1}{2}\left(\mathrm{tr}\left(\boldsymbol{\Sigma}\right) + \boldsymbol{\mu}^T\boldsymbol{\mu} - k - \log\left(\det\left(\boldsymbol{\Sigma}\right)\right)\right) = \frac{1}{2}\left(\sum_{i=1}^{k} \sigma_i^2 + \sum_{i=1}^{k} \mu_i^2 - \sum_{i=1}^{k} 1 - \sum_{i=1}^{k} \log\left(\sigma_i^2\right)\right)$$

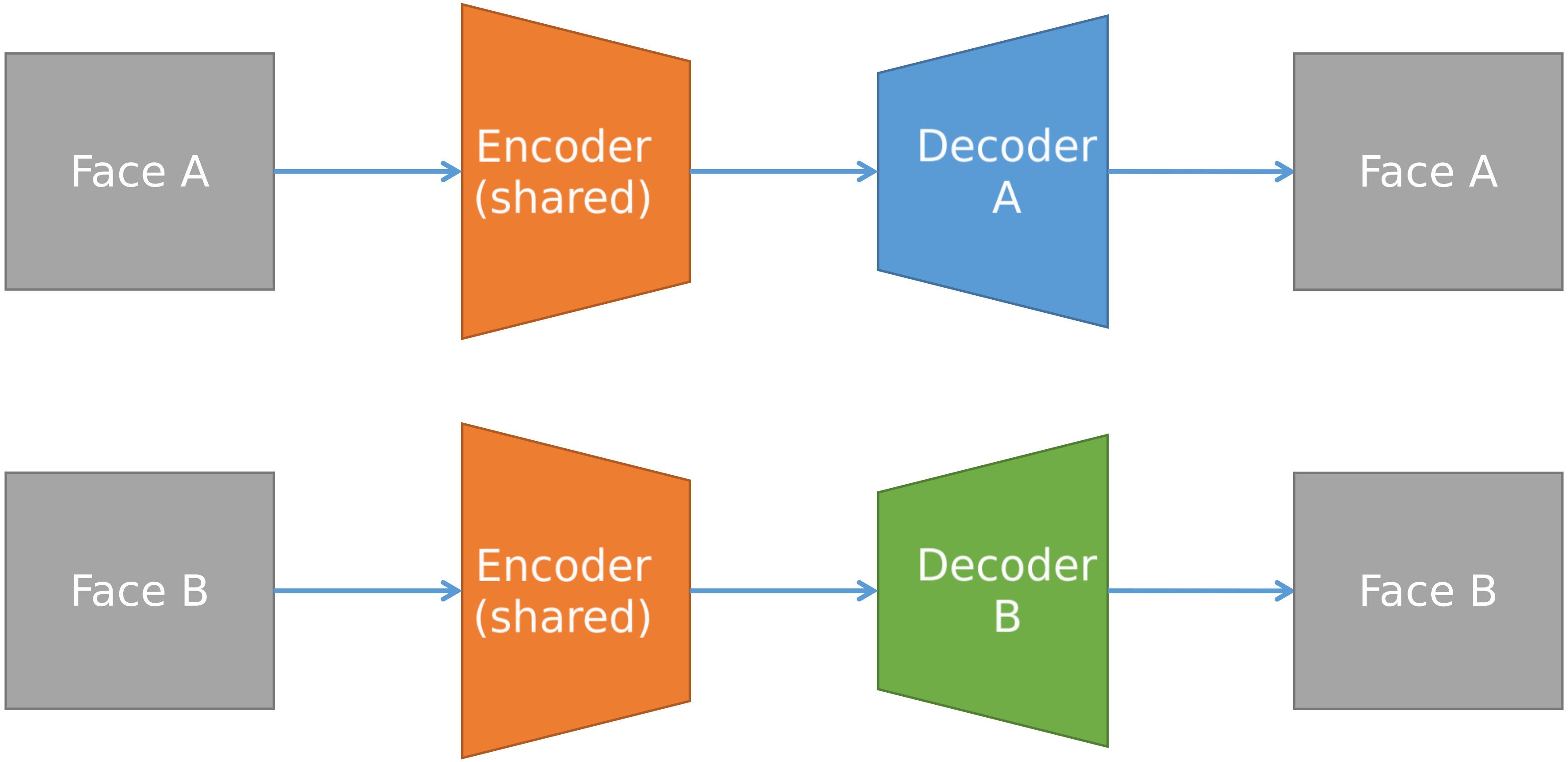
$$= \frac{1}{2} \sum_{i=1}^{k} \left(\sigma_i^2 + \mu_i^2 - 1 - \log\left(\sigma_i^2\right)\right)$$

# VAE Training Regularization: KL-divergence in PyTorch

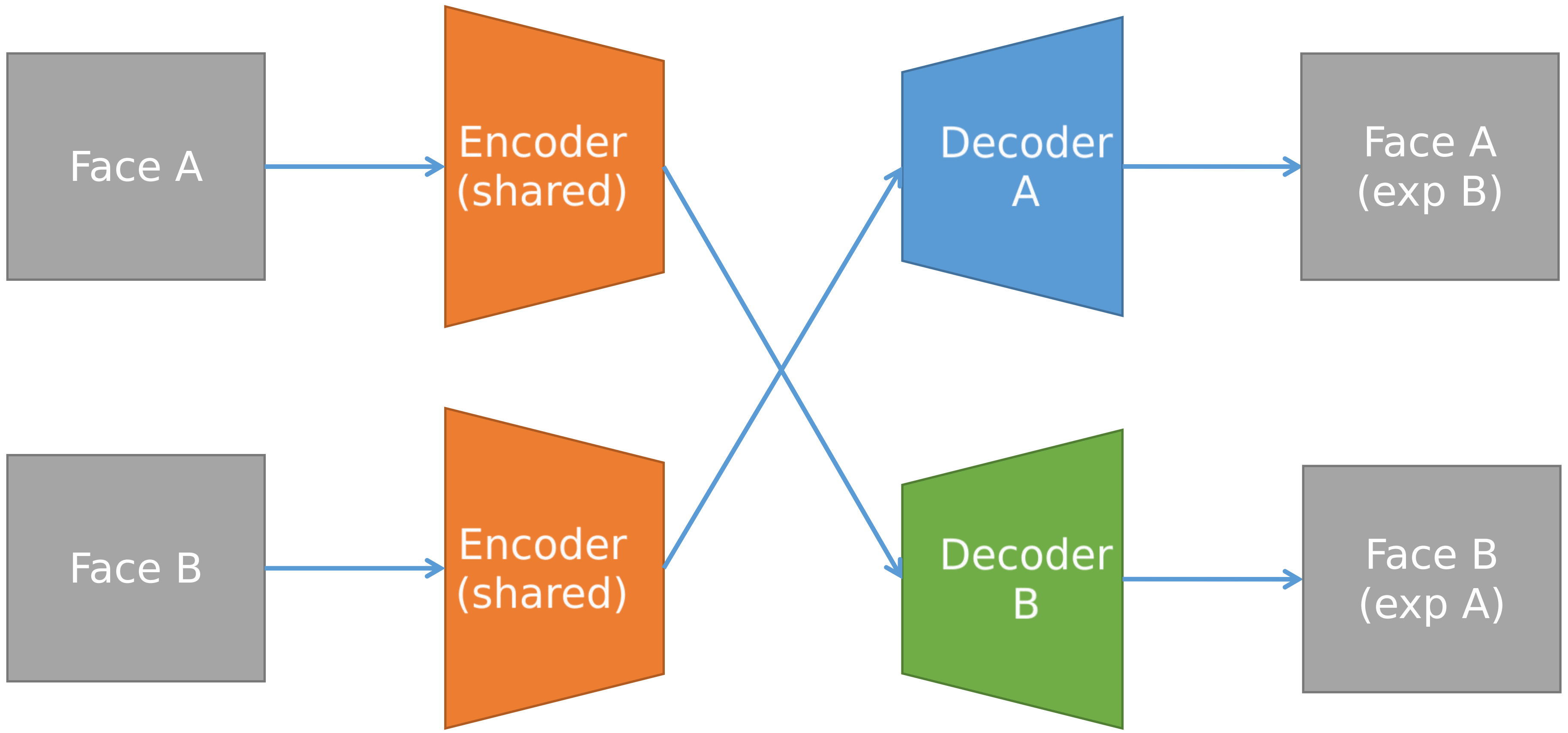
```
vae = VariationalAutoencoder(784, 10)
loss_fn = torch.nn.L1Loss()
optimizer = torch.optim.Adam(vae.parameters(), lr=0.001)

for batch in dataset:
  mu, std, reconstructed = vae(batch)
  error = loss_fn(reconstructed, batch)
  KL = 0.5 * (std + mu.pow(2) - 1 - torch.log(std)).sum(dim=1).mean(dim=0)

  loss = (alpha*error + beta*KL)

  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
```

# FaceSwap Architecture: Training

**FaceSwap** is based on **two Autoencoders** (usually denoising)

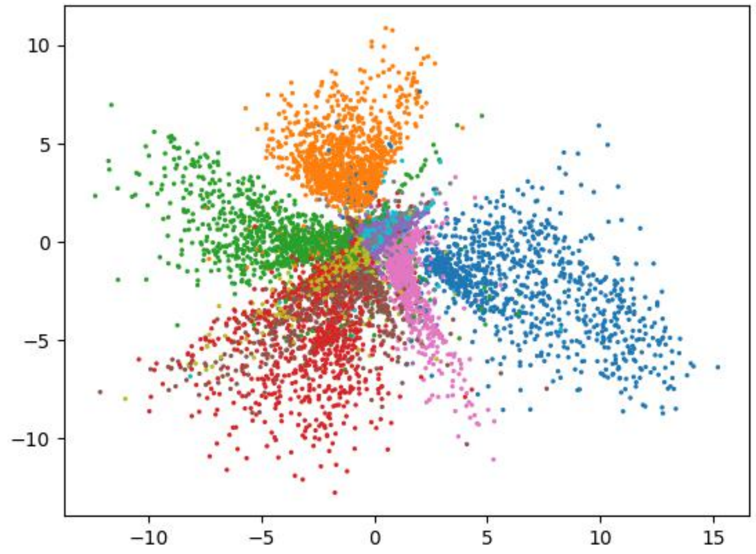


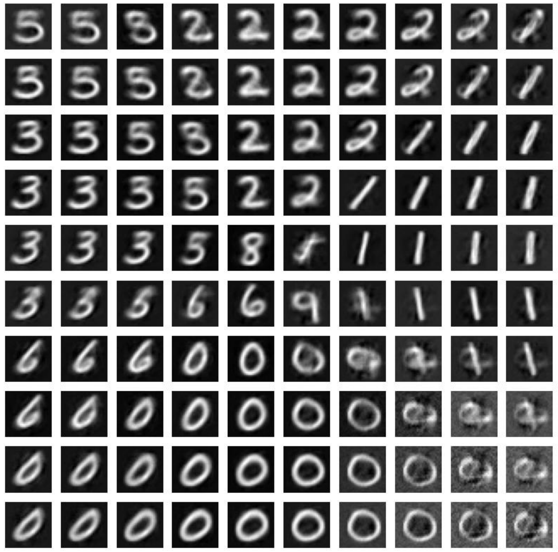
# FaceSwap Architecture: Inference



# Exercise 1

- **Train** a simple **Autoencoder** on the **MNIST** dataset.
  - Use a **latent** space of **<u>dimension 2</u>**
- **Encode** all the digits in the **test set** and plot a **scatterplot** of the encoder's **latent space**
- **Plot** a map of the **recostructed digits** from the **latent space**



# Exercise 2

- **Train** a **Variational Autoencoder** on the **MNIST** dataset.
  - Use a **latent** space of **<u>dimension 2</u>** (2 mu and 2 std)
- **Encode** all the digits in the **test set** and plot a **scatterplot** of the encoder's **mu**
- **Plot** some **reconstructed** digits **random sampling** the **latent space**