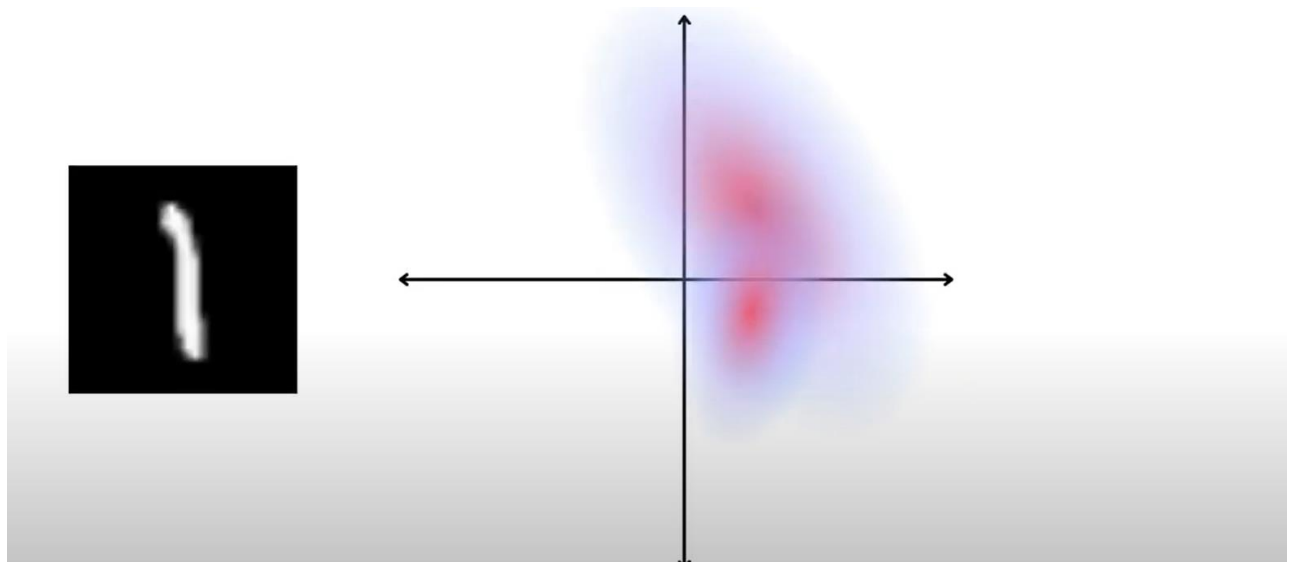
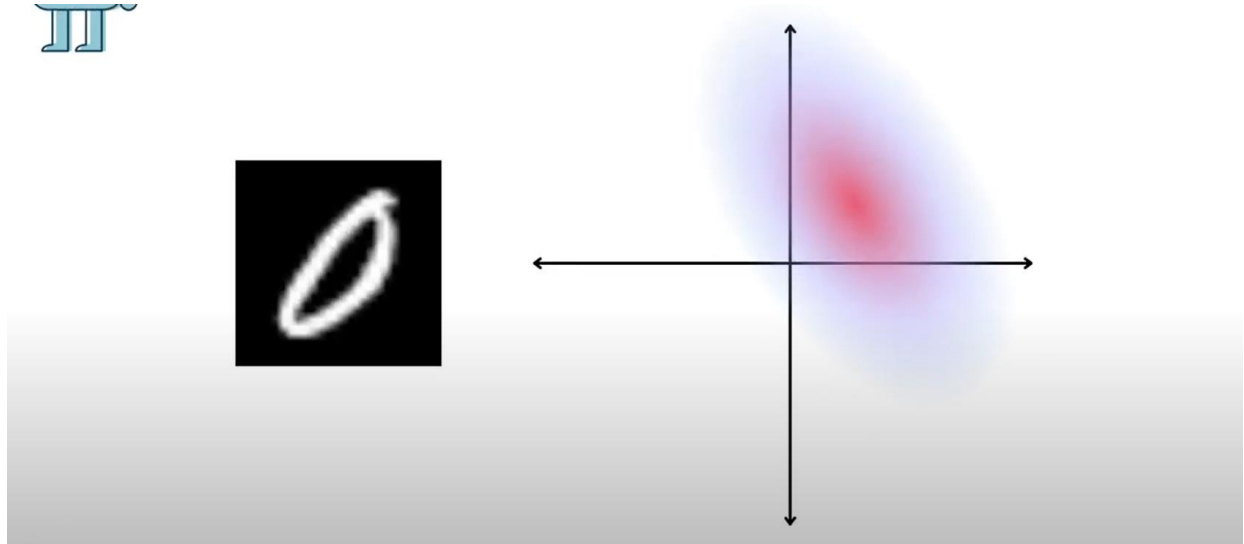
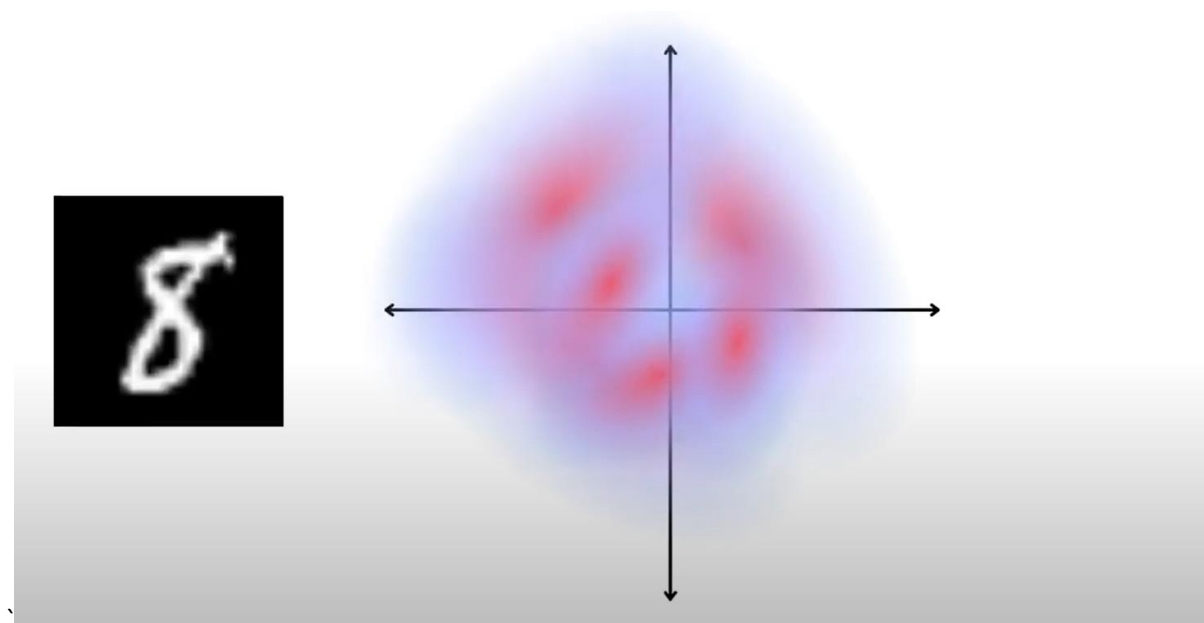
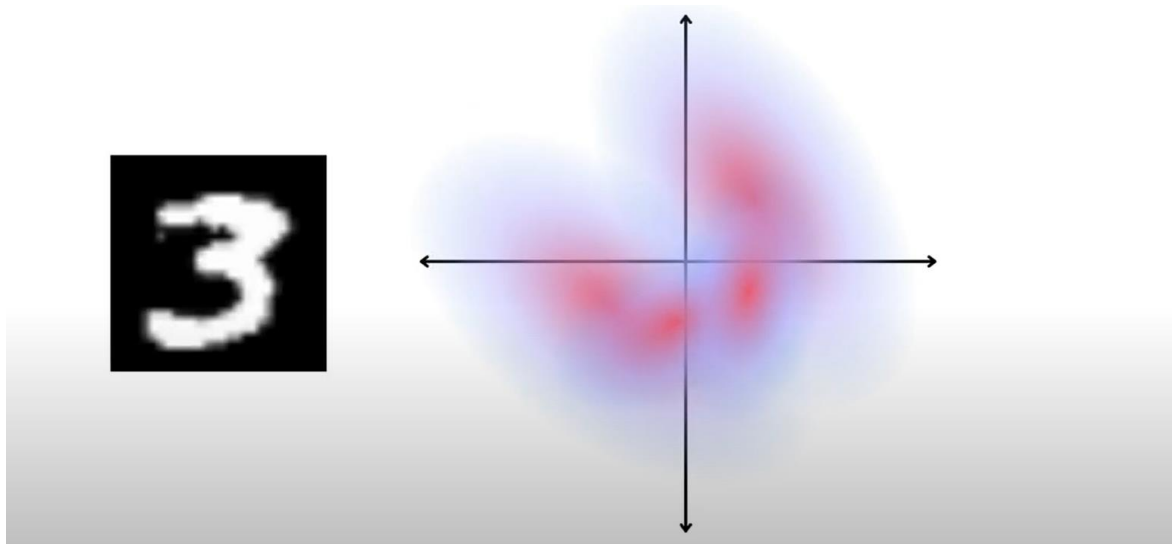
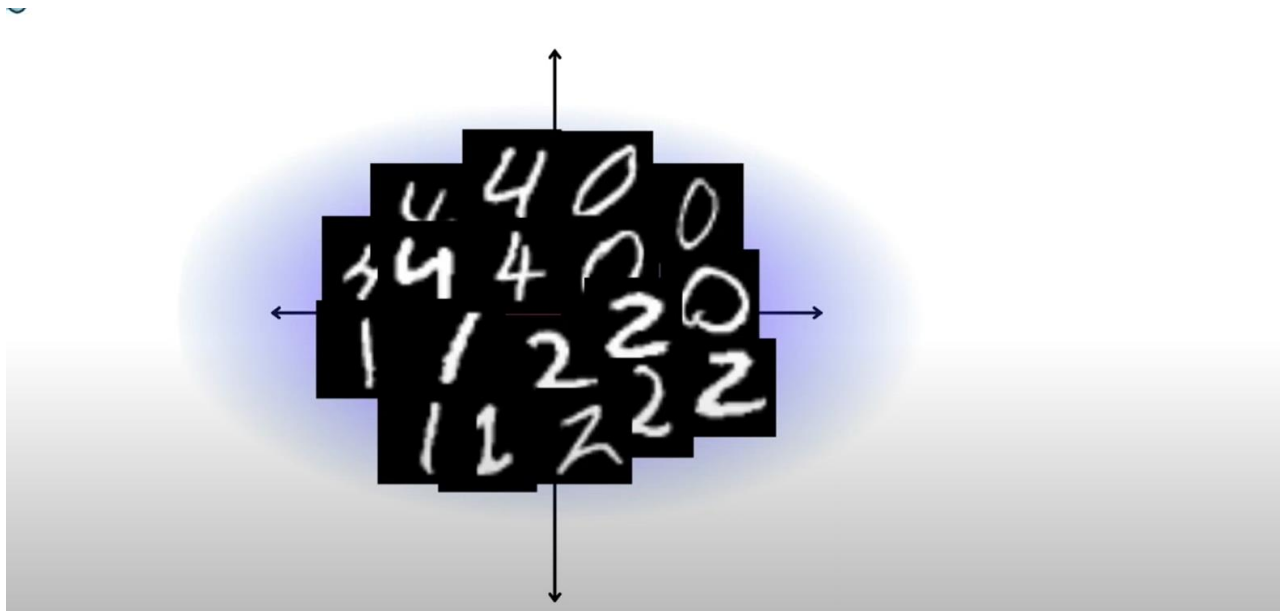


<https://www.youtube.com/watch?v=1ZHzAOutcnw&t=913s>





In VAE , We encode each point as the distribution in the latent space Interpolation of data point is very smooth manner



*But I already have VAE.....I dont need VQVAE*

Lower dimensional latent space becomes weak/noisy for a strong decoder and VAE could collapse into generating images that look like mean representation of data.

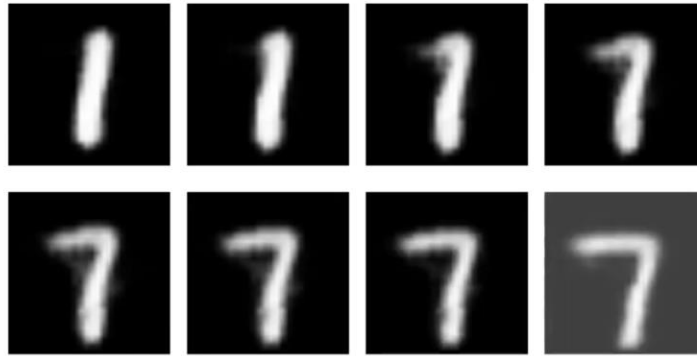
On increasing the latent dimension, we are requiring VAE to ensure all points sampled from this higher dimensional latent space generate valid image which VAE might find difficult to learn to do.



Interpolation may generate the morphed image



*But I already have VAE.....I dont need VQVAE*



*But I already have VAE.....I dont need VQVAE*



But the real word data is not a continuous, but the data is discrete

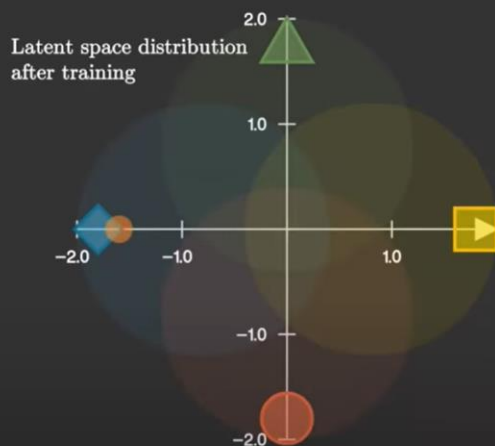


*But I already have VAE.....I dont need VQVAE*

But real world data across classes is not continuous and interpolating across these classes might not make sense at all. Either the image is of a cat or a person, interpolating across classes cat and person does not generate a real world image.



## Related Work - Variational Autoencoder problem



Latent space is regularized. Vectors sampled from latent space can generate valid data.

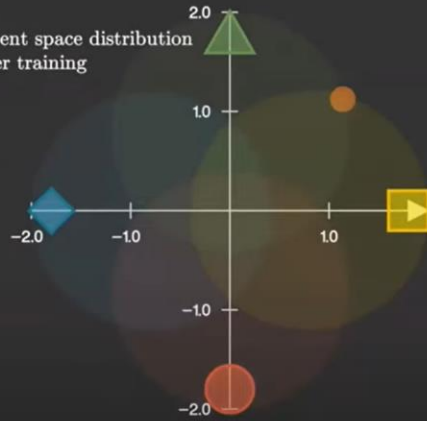


Vectors sampled from overlapping distribution generates morphed data.



## Related Work - VAE problem

Latent space distribution after training

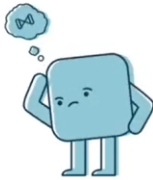


Latent space is regularized. Vectors sampled from latent space can generate valid data.



Vectors sampled from overlapping distribution generates morphed data.

UCF



*But I already have VAE.....I dont need VQVAE*

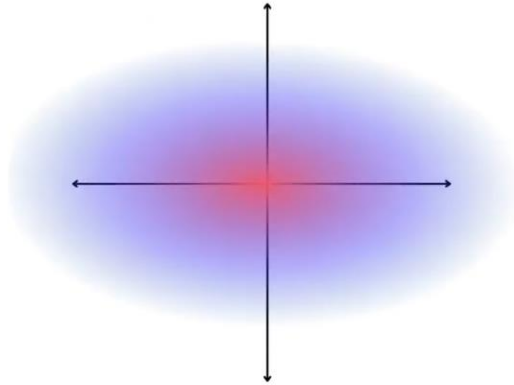
**VAE** continuous latent representation





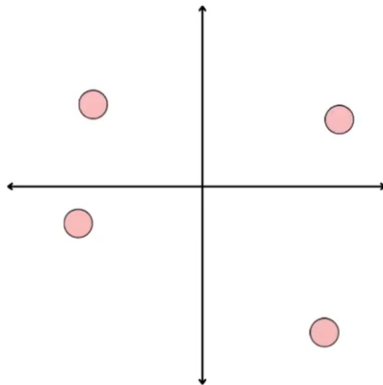
*But I already have VAE.....I dont need VQVAE*

**VQVAE** discrete latent representation



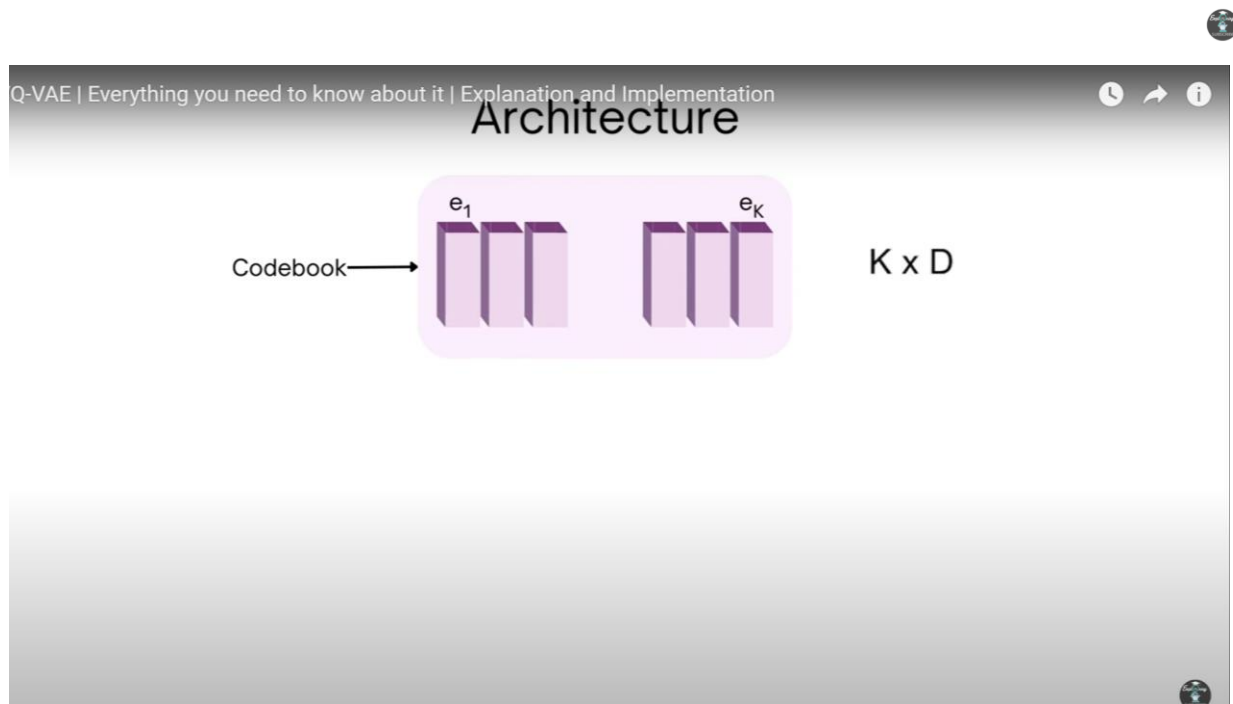
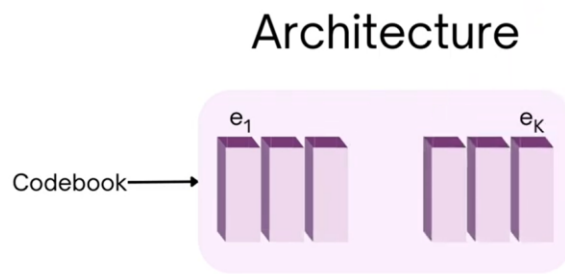
*But I already have VAE.....I dont need VQVAE*

**VQVAE** discrete latent representation + sparse



We learn discrete which and also finite.

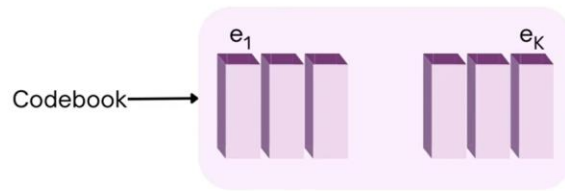
Let discuss the different components of VQVAE, only difference is code book



Second components is encoder



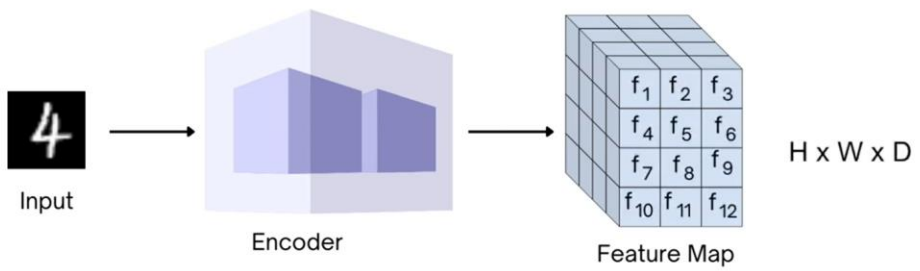
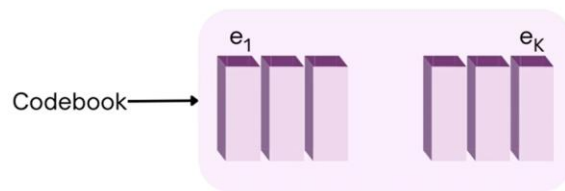
## Architecture



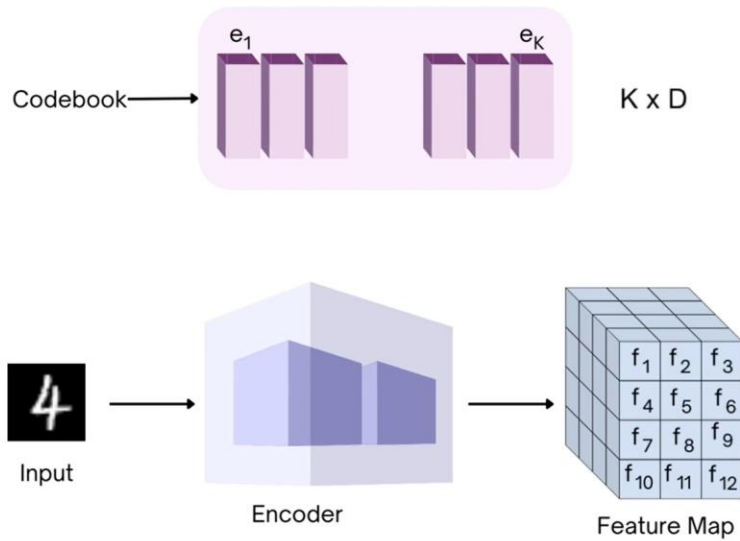
Encoder



## Architecture

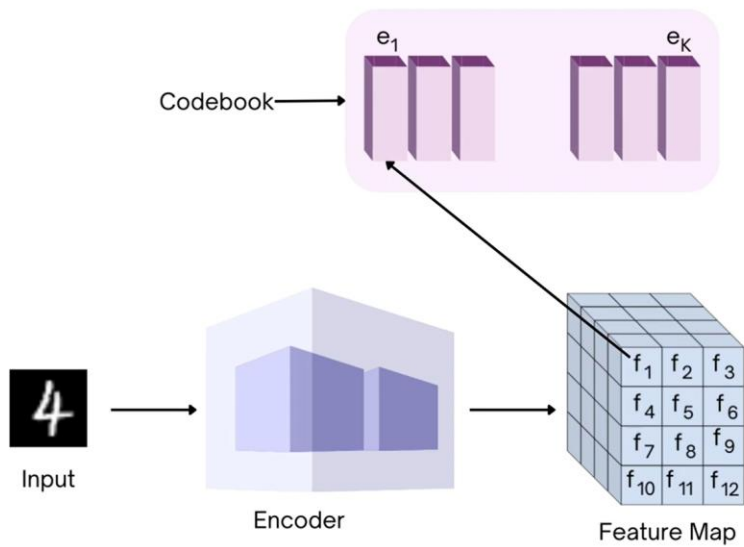


## Architecture

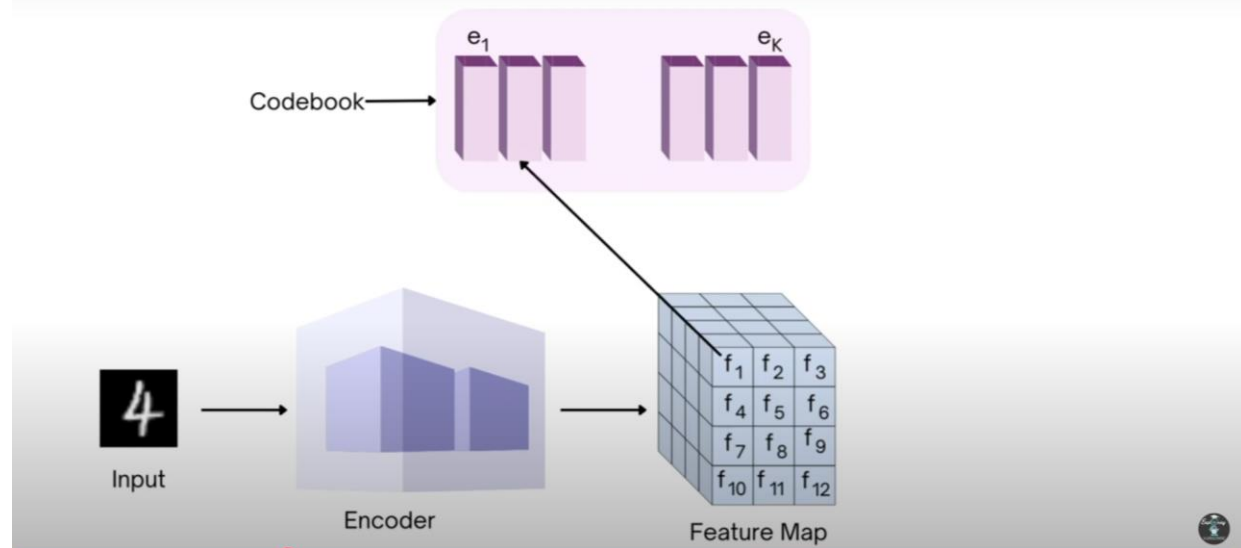


If the feature maps differ from the Codebook output then, we can use the **1x1 conv operation** to match the depth of feature maps with the code book vector size. That D

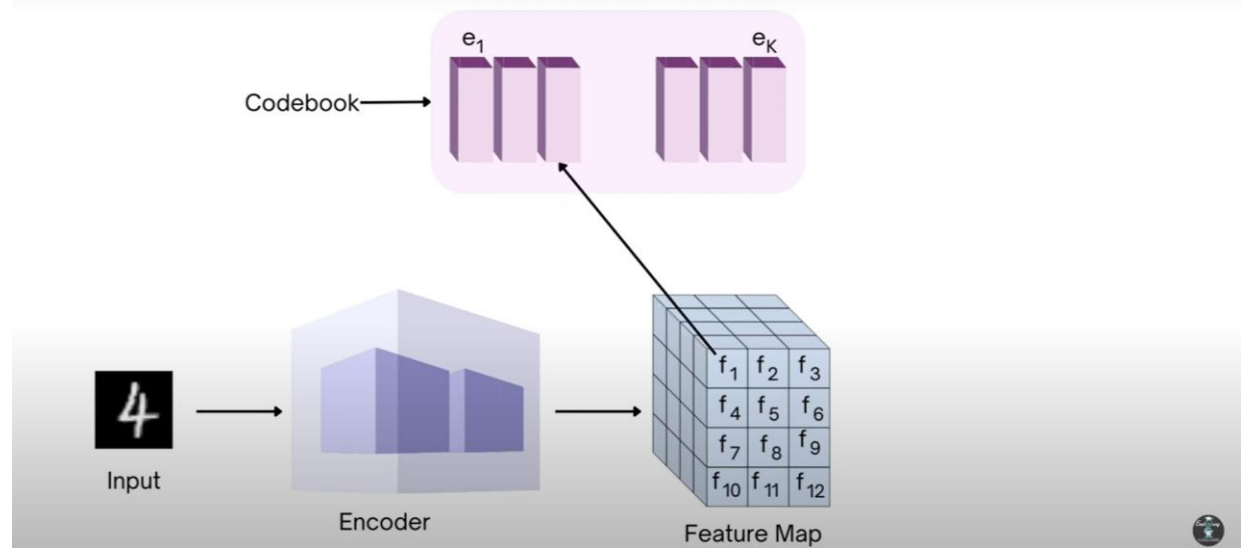
## Architecture



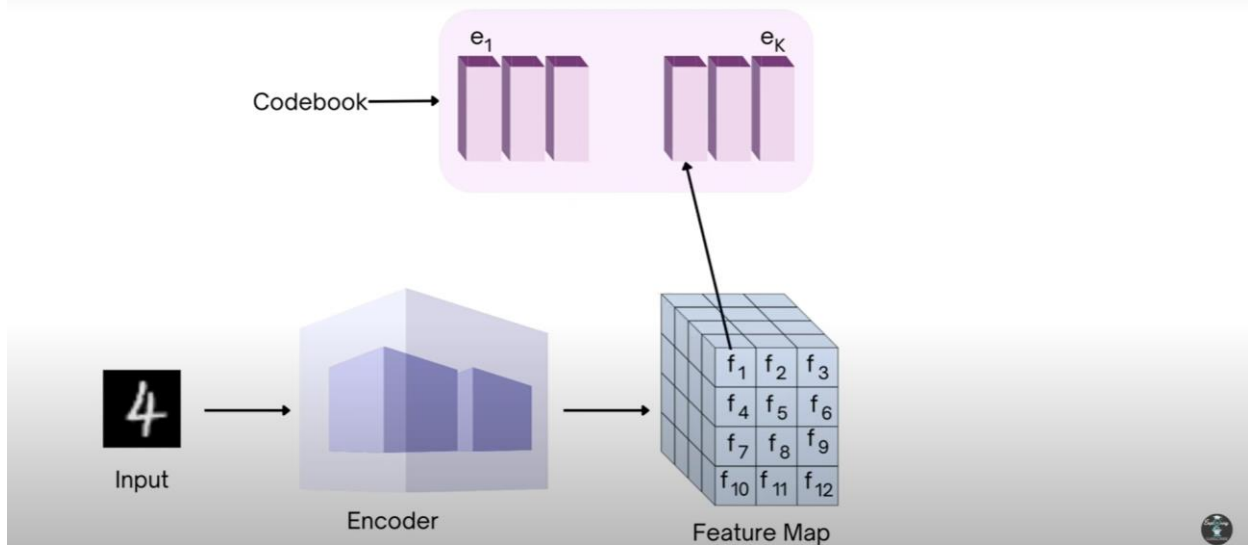
## Architecture



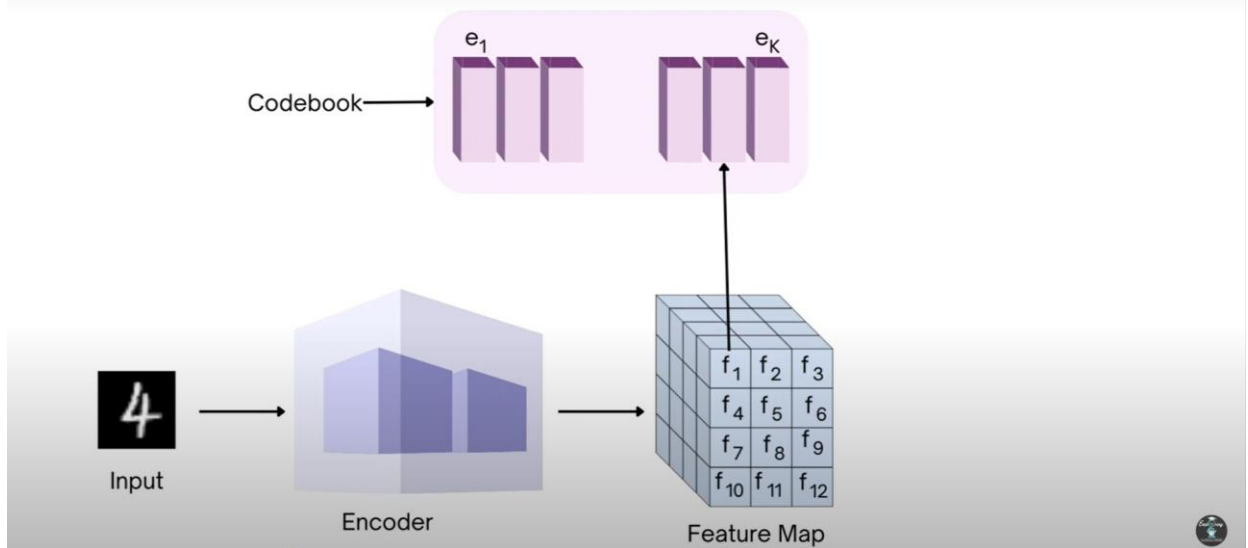
## Architecture



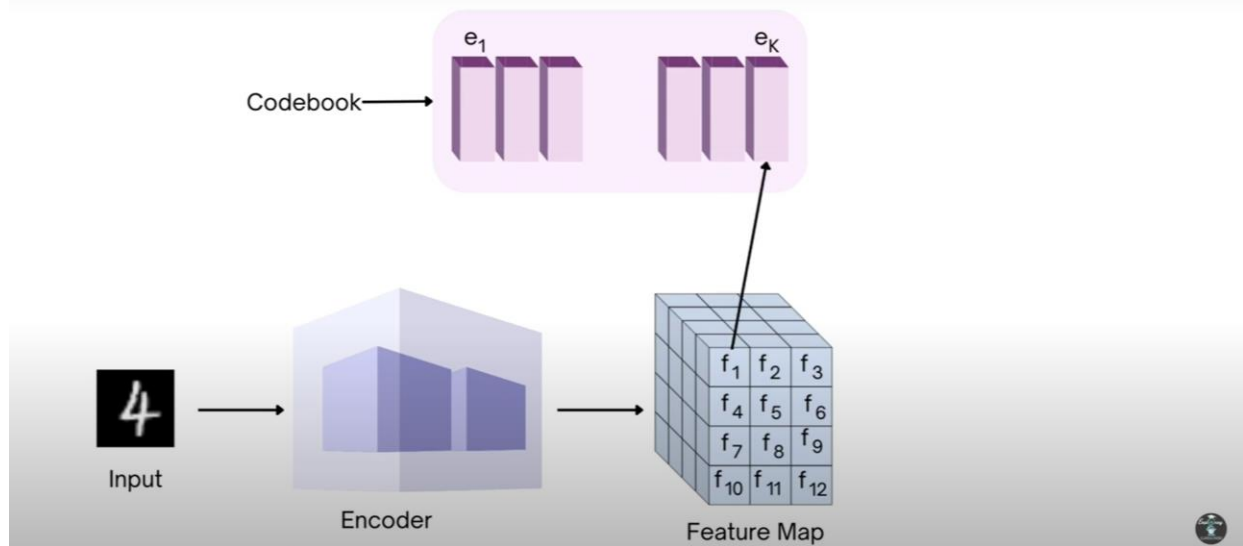
## Architecture



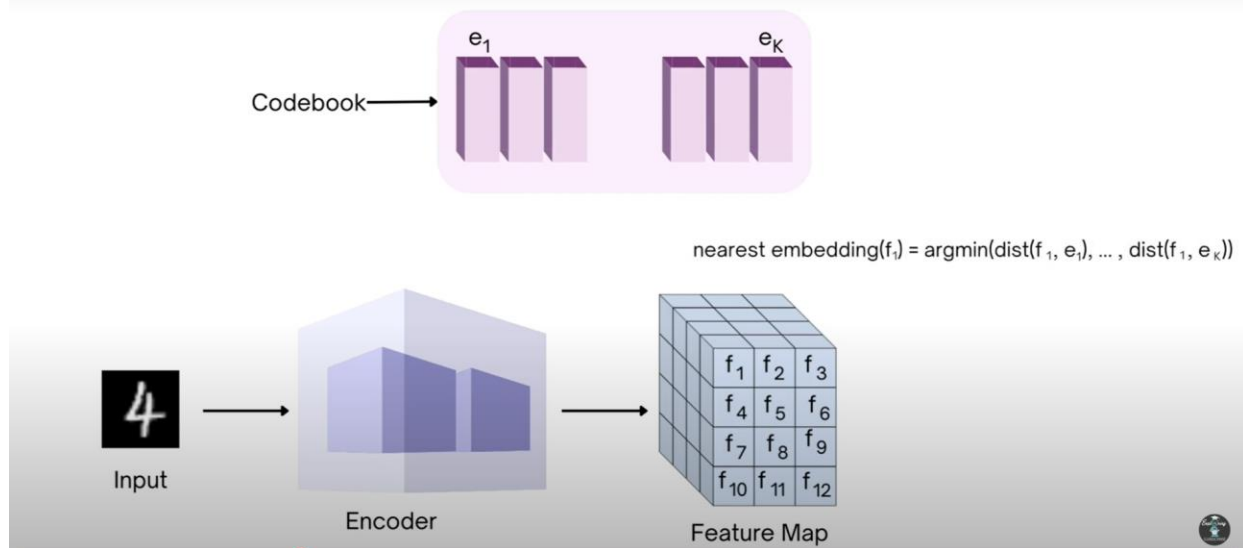
## Architecture

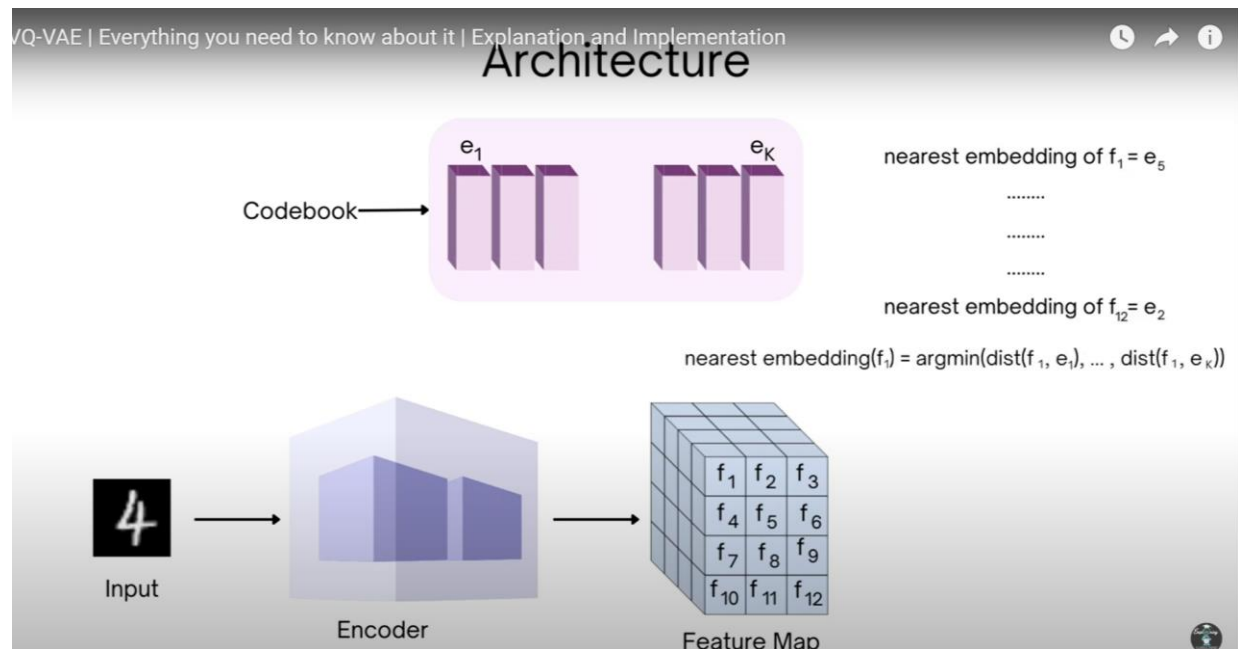


## Architecture

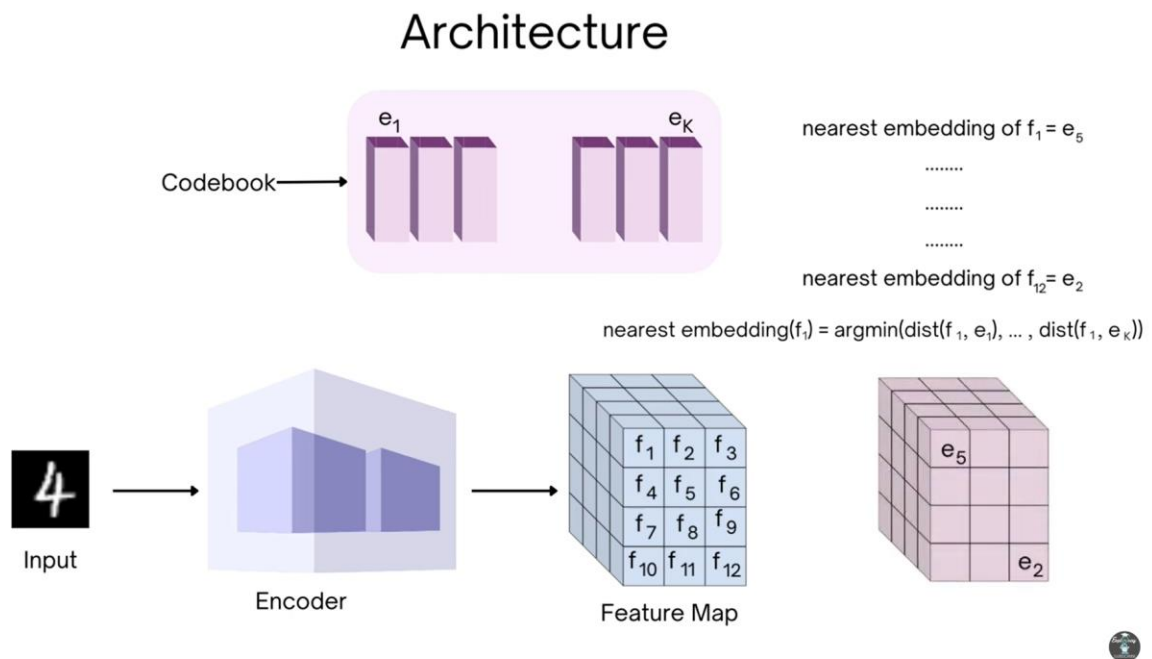


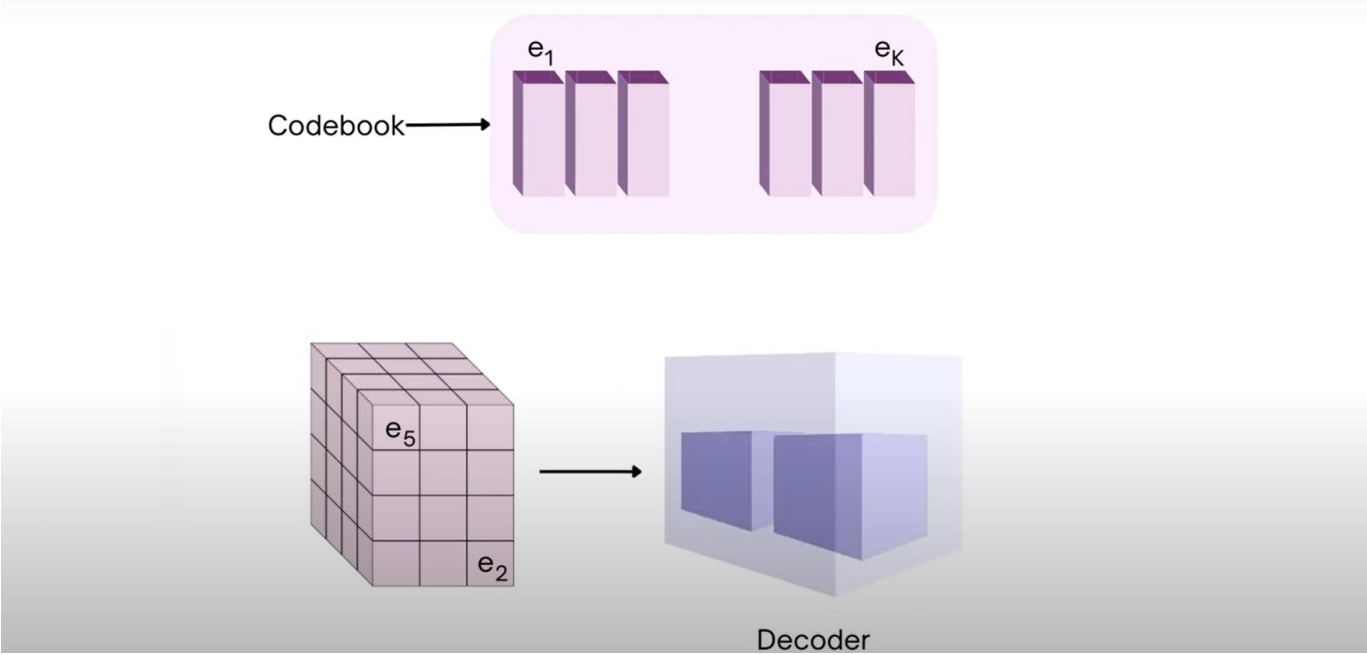
## Architecture



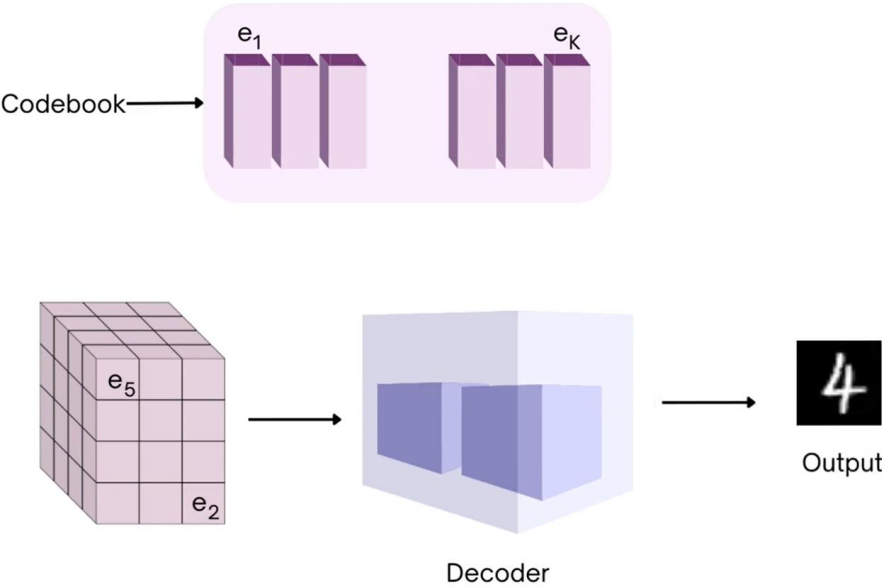


Here is the encoding representation in the form of feature maps



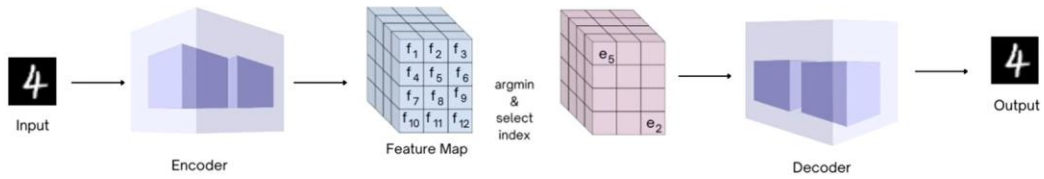


# Architecture

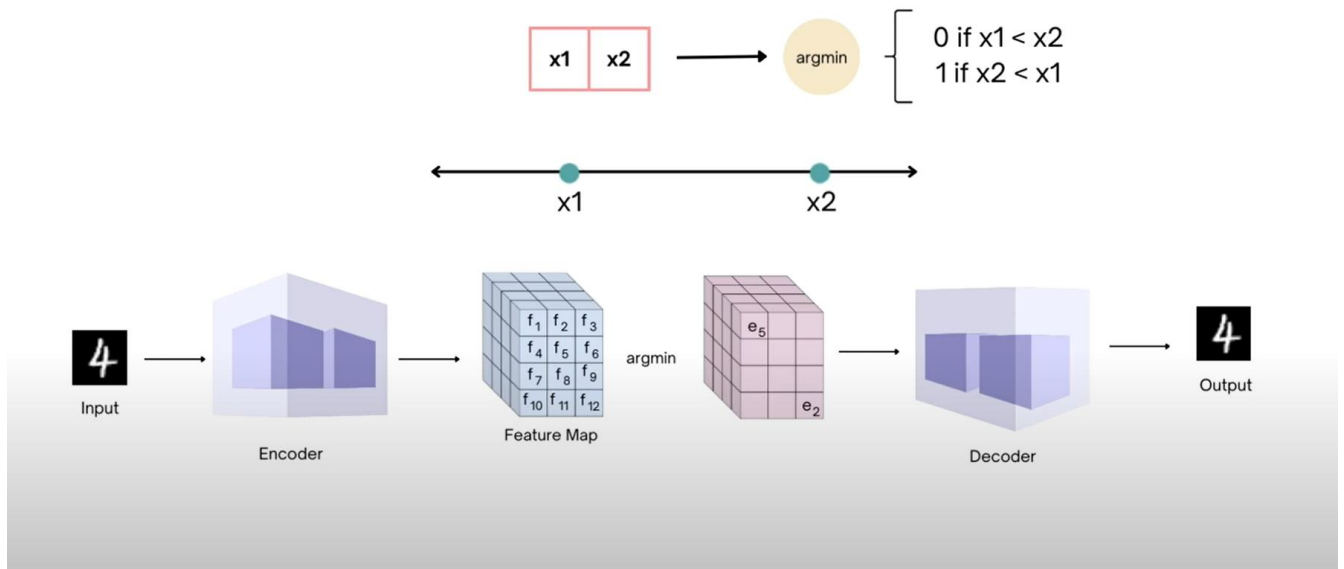




Where did my gradients go???



Gradient could not be computed at the argument place

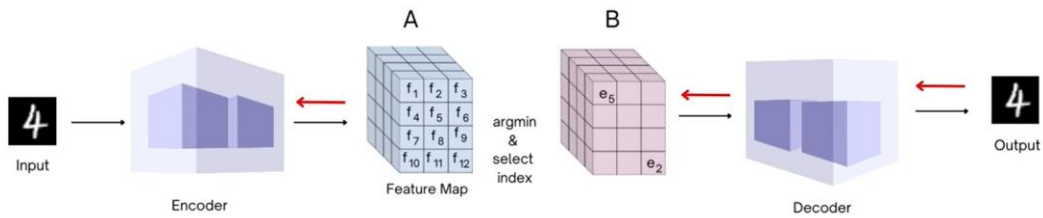


Solution



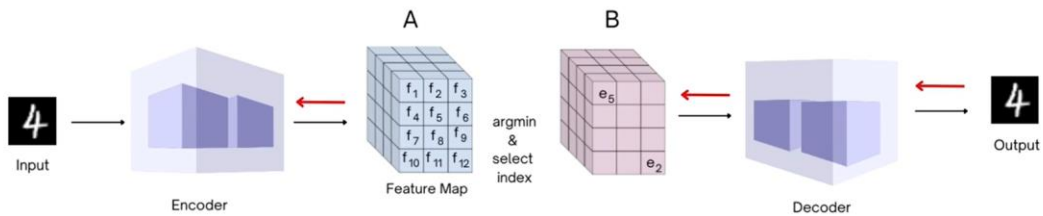
$$A \longrightarrow f \longrightarrow B \quad B = f(A)$$

$$B = A + (f(A) - A).detach()$$



$$A \longrightarrow f \longrightarrow B \quad B = f(A)$$

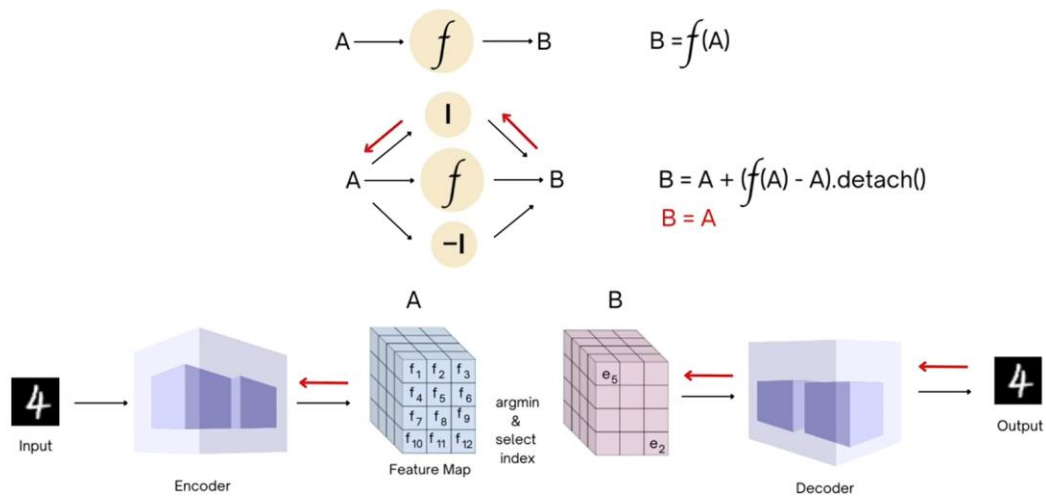
$$A \xrightarrow{\begin{matrix} I \\ f \\ -I \end{matrix}} B \quad B = A + (f(A) - A).detach()$$



quantized = inputs + (quantized - inputs).detach()

### Key Mechanisms in Forward Pass

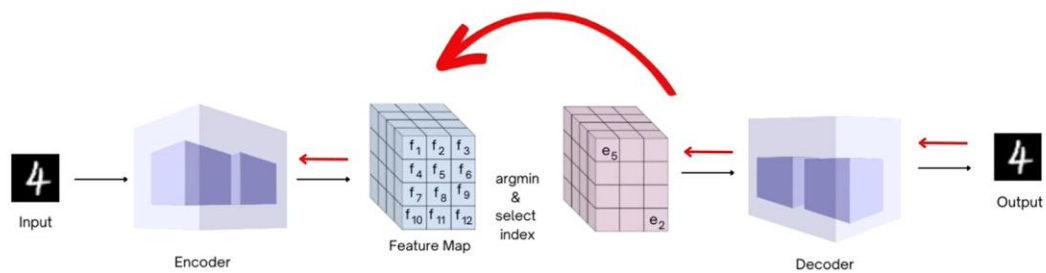
1. **Straight-Through Estimator (STE):** Ensures gradients bypass non-differentiable steps in the quantization process.
2. **Detaching Tensors:** Stops gradients from propagating to irrelevant parts like the **codebook** during the commitment loss or to the **encoder** during the codebook update.



`quantized = inputs + (quantized - inputs).detach()`

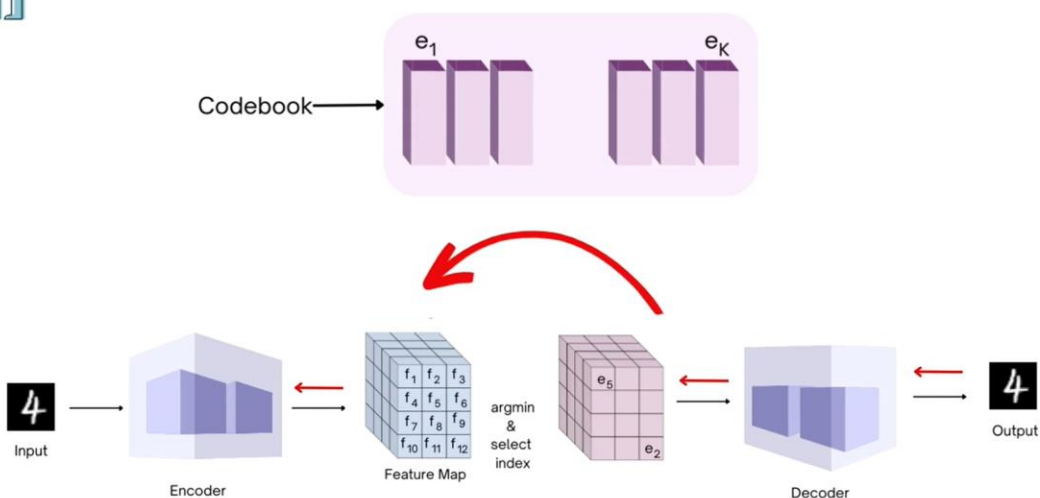
Actually happen, is thought straight thought gradient

## Architecture

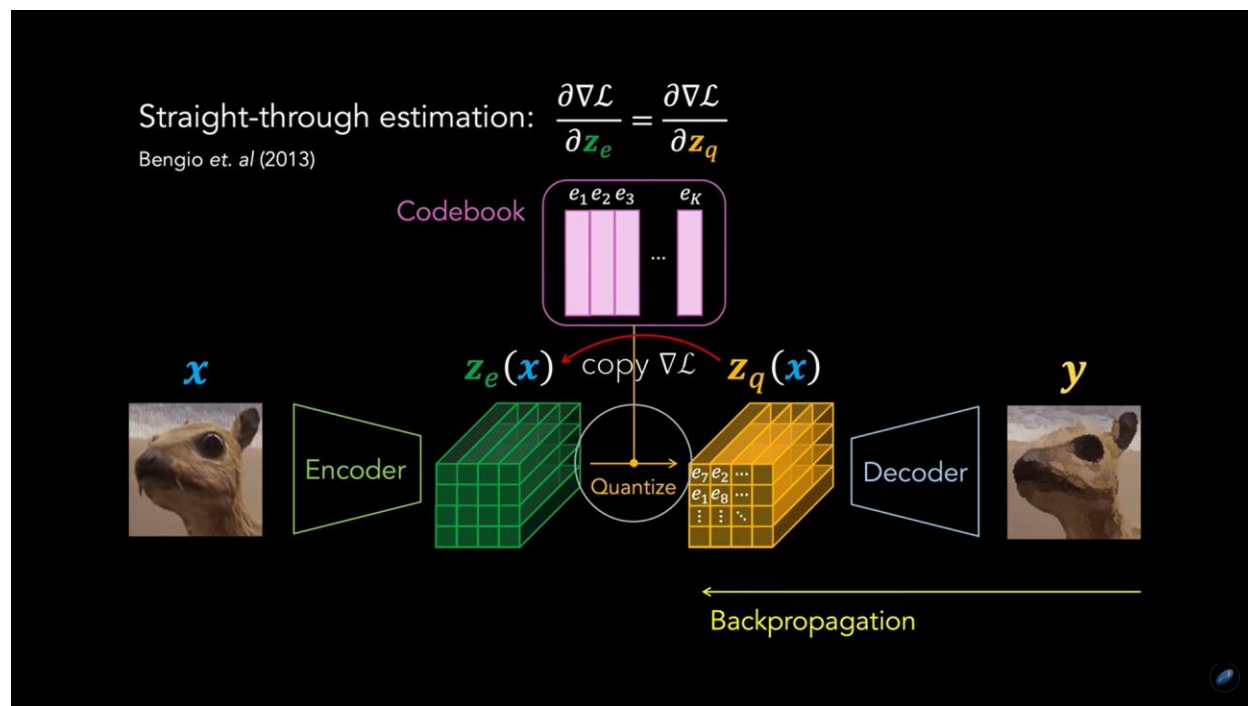




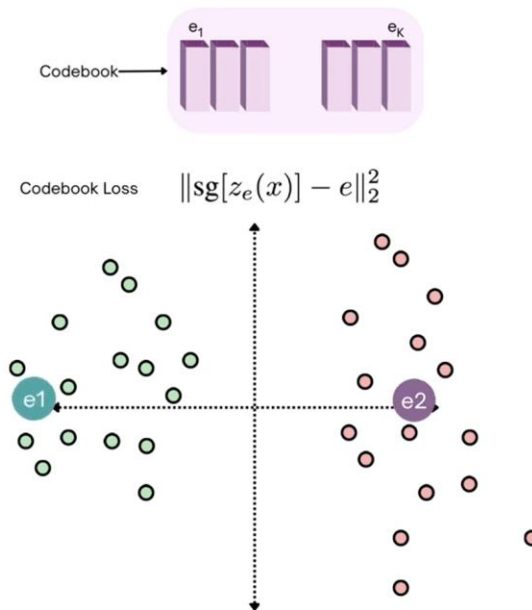
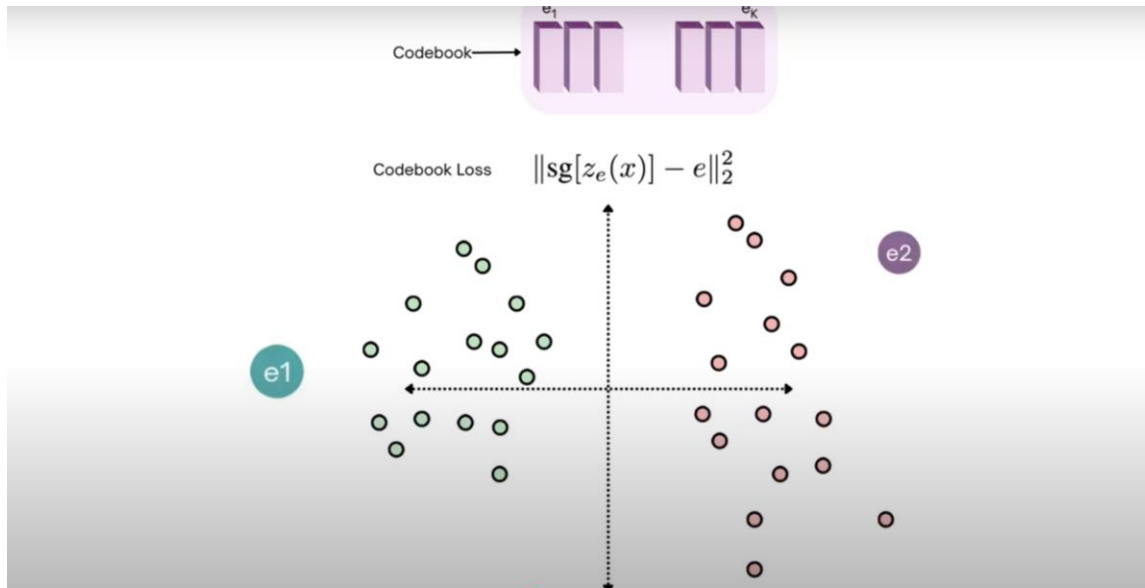
Okay...reconstruction loss done...are we ready to implement?



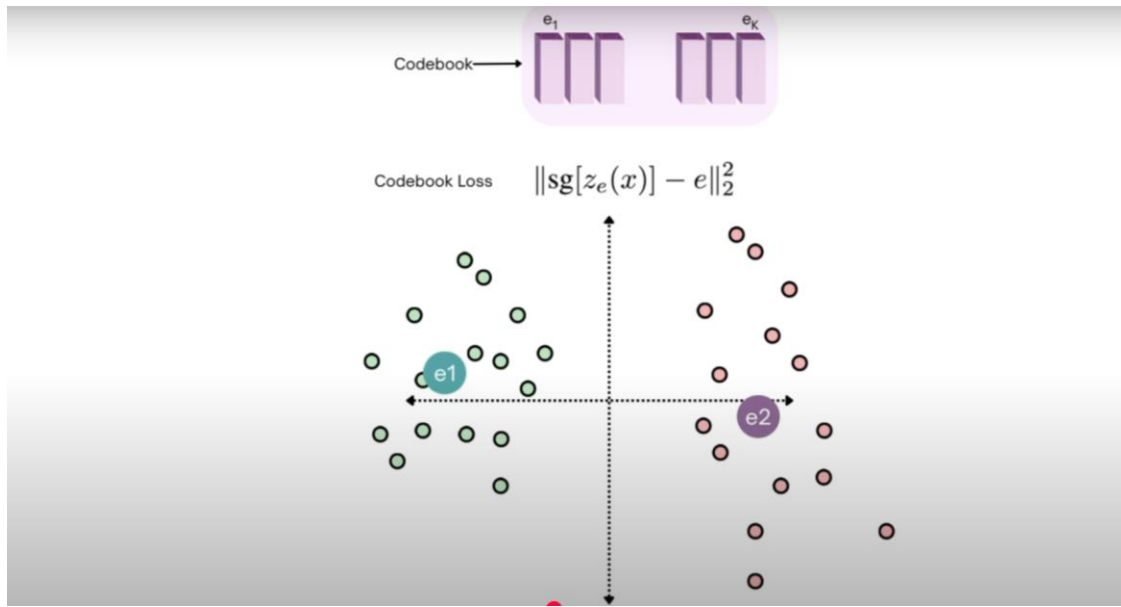
no gradients are passed through codebook vectors



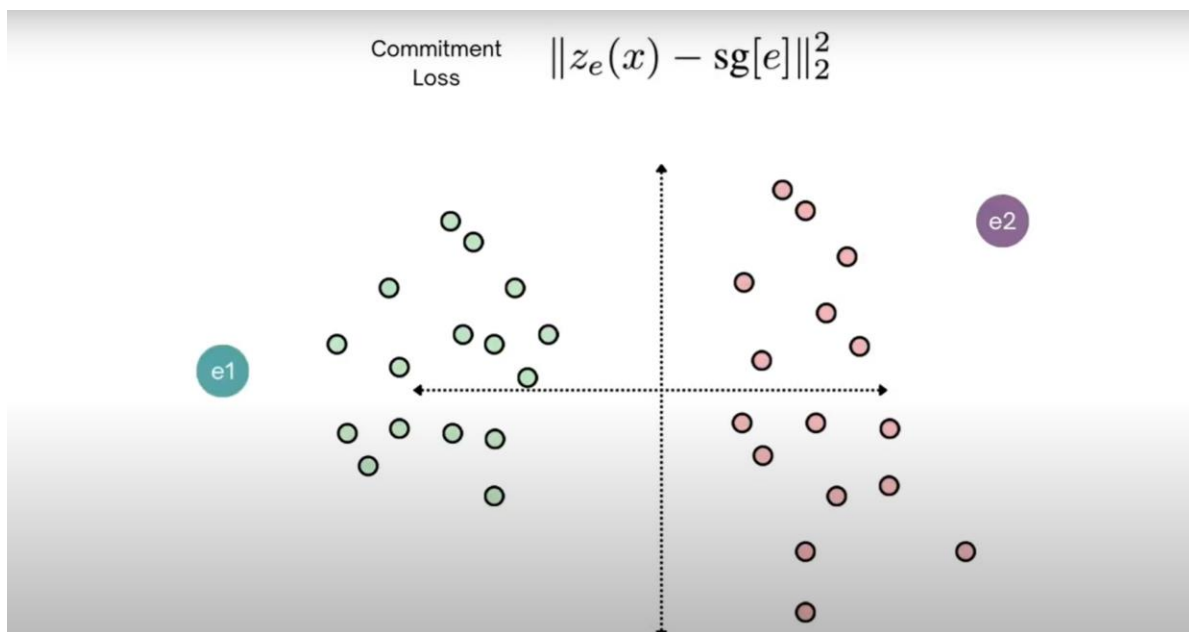
## Loss function:



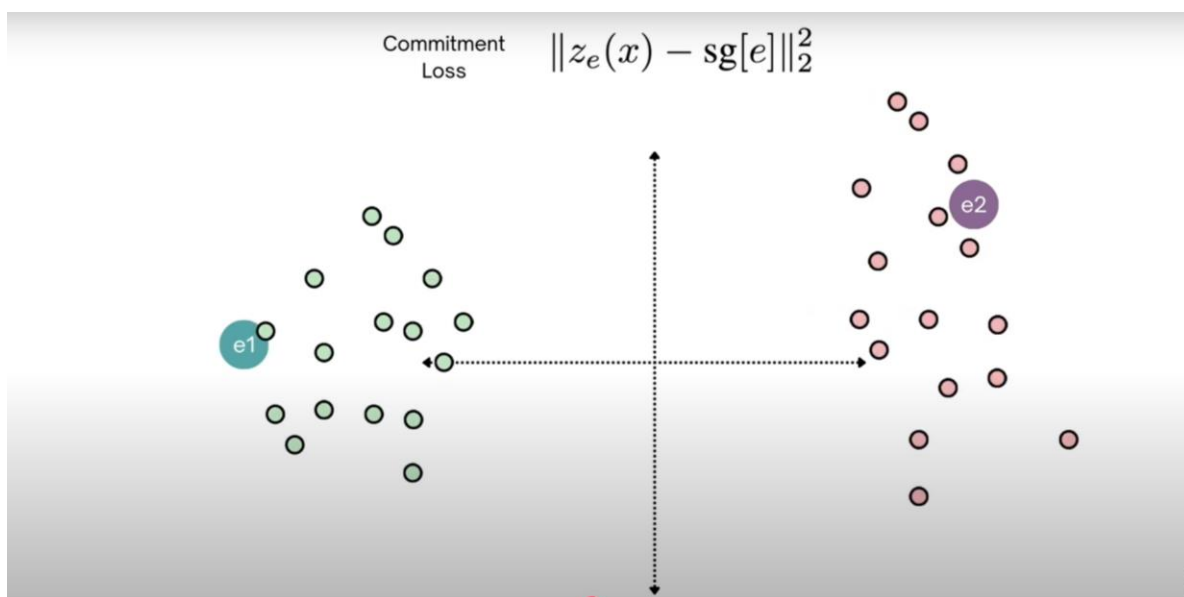
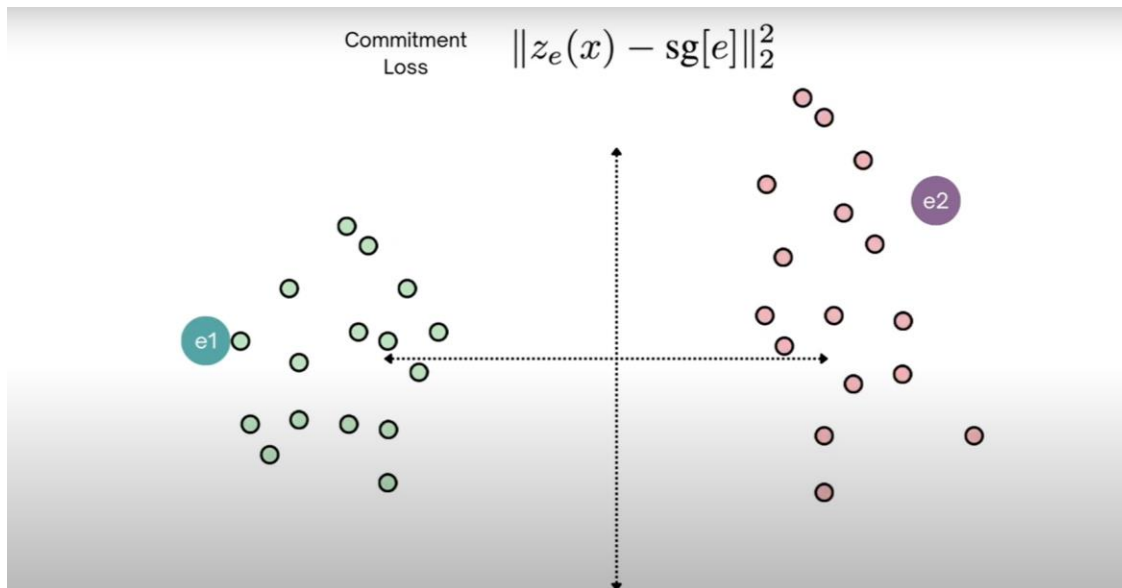
sg(Stop gradient) make the  $z_e(x)$  is considered.



Commitment loss is also used



In commitment loss  $e$  is treated as the constant,  $sg[e]$ , make the  $e$  constant, it brings the encoder points closer to the embedding of code book



The encoder adapts to the embeddings using the commitment loss, where the codebook embeddings are treated as constant. Conversely, the codebook loss updates the embeddings to align with the encoder's output, treating the encoder's output as constant.

**After training the encoder output become closer to notebook embedding.**

Reconstruction Loss

MSE

Codebook Loss

$$\|\text{sg}[z_e(x)] - e\|_2^2$$

Commitment Loss

$$\|z_e(x) - \text{sg}[e]\|_2^2$$

$$L = \log p(x|z_q(x)) + \|\text{sg}[z_e(x)] - e\|_2^2 + \beta \|z_e(x) - \text{sg}[e]\|_2^2$$

(typically weighted by  $\gamma$ )

$$\mathcal{L}(\theta, \phi, C; \mathbf{x}, \mathbf{z}_q) = -\log p_{\theta}(\mathbf{x}|\mathbf{z}_q) + \|\text{sg}[\mathbf{z}_e(\mathbf{x})] - \mathbf{e}\|_2^2 + \|\mathbf{z}_e(\mathbf{x}) - \text{sg}[\mathbf{e}]\|_2^2$$

Reconstruction loss  
(optimizes  $\phi, \theta$ )

Codebook loss  
(optimizes  $C$ )

Commitment loss  
(optimizes  $\phi$ )

## Toy example

Here's a step-by-step toy example of the forward pass, loss computation, and backward pass in a Vector Quantized Variational Autoencoder (VQ-VAE):

### Toy Setup

- **Input data:** A single input  $x$  of size 2 (e.g.,  $x = [1.2, 2.4]$ ).
- **Encoder output:** The encoder generates a continuous latent vector  $z_e(x)$ .
- **Codebook:** A codebook with 3 vectors ( $k = 3$ ), each of size 2:  
 $e_1 = [1.0, 2.0]$ ,  $e_2 = [3.0, 3.0]$ ,  $e_3 = [0.5, 0.5]$ .
- **Decoder:** The decoder reconstructs  $\hat{x}$  from the quantized vector  $z_q(x)$ .



## 1. Forward Pass

1. **Input:**

$$x = [1.2, 2.4]$$

2. **Encoder Output:**

The encoder generates a continuous latent vector:

$$z_e(x) = [1.4, 2.1]$$

3. **Find the Nearest Codebook Vector:** Compute the distance between  $z_e(x)$  and each codebook vector  $e_k$ :

$$\text{distance to } e_1 = \|[1.4, 2.1] - [1.0, 2.0]\|^2 = (1.4 - 1.0)^2 + (2.1 - 2.0)^2 = 0.16 + 0.01 = 0.17$$

$$\text{distance to } e_2 = \|[1.4, 2.1] - [3.0, 3.0]\|^2 = (1.4 - 3.0)^2 + (2.1 - 3.0)^2 = 2.56 + 0.81 = 3.37$$

$$\text{distance to } e_3 = \|[1.4, 2.1] - [0.5, 0.5]\|^2 = (1.4 - 0.5)^2 + (2.1 - 0.5)^2 = 0.81 + 2.56 = 3.37$$

The closest codebook vector is  $e_1 = [1.0, 2.0]$ .

4. **Quantized Latent Vector:**

The quantized latent vector is:

$$z_q(x) = e_1 = [1.0, 2.0]$$

5. **Decoder Output:**

The decoder reconstructs the input from  $z_q(x)$ :

$$\hat{x} = \text{Decoder}(z_q(x)) = [1.1, 2.2]$$

## 2. Loss Function Computation

The total loss includes reconstruction loss, commitment loss, and optionally a codebook update loss.

(a) **Reconstruction Loss**

$$\mathcal{L}_{\text{reconstruction}} = \|x - \hat{x}\|^2 = \|[1.2, 2.4] - [1.1, 2.2]\|^2 = (1.2 - 1.1)^2 + (2.4 - 2.2)^2 = 0.01 + 0.04 = 0.05$$

(b) **Commitment Loss**

$$\mathcal{L}_{\text{commitment}} = \|z_e(x) - \text{sg}(z_q(x))\|^2 = \|[1.4, 2.1] - [1.0, 2.0]\|^2 = 0.16 + 0.01 = 0.17$$

(c) **Codebook Update Loss (Optional)**

$$\mathcal{L}_{\text{codebook}} = \|\text{sg}(z_e(x)) - z_q(x)\|^2 = \|[1.4, 2.1] - [1.0, 2.0]\|^2 = 0.17$$

### 3. Backward Pass

- **Gradients for Encoder:** The encoder is updated using the **reconstruction loss** and the **commitment loss**. The stop-gradient operator ensures no gradients flow into  $z_q(x)$ .
  - **Gradients for Decoder:** The decoder is updated using the **reconstruction loss** to improve the output  $\hat{x}$ .
  - **Codebook Update:** The codebook is updated using the **codebook update loss** (via gradient descent or EMA) to make  $z_q(x)$  closer to  $z_e(x)$ .
- 

### Summary of Updates

1. **Encoder:** Trained using  $\mathcal{L}_{reconstruction}$  and  $\mathcal{L}_{commitment}$ .
2. **Decoder:** Trained using  $\mathcal{L}_{reconstruction}$ .
3. **Codebook:** Updated using  $\mathcal{L}_{codebook}$  to align embeddings.

This toy example demonstrates the flow and ensures clarity in the purpose of each component and loss term in the VQ-VAE training process.

why we say that the **decoder** is primarily updated by the **reconstruction loss**, even though all components are part of the total loss.

## 1. The Total Loss in VQ-VAE

The total loss is typically written as:

$$L_{\text{total}} = L_{\text{reconstruction}} + L_{\text{commitment}} + L_{\text{codebook update}}$$

Where:

- $L_{\text{reconstruction}} = \|x - \hat{x}\|^2$  — measures the difference between the input  $x$  and the reconstructed output  $\hat{x}$ , and it involves both the encoder and the decoder.
- $L_{\text{commitment}} = \beta \|z_e(x) - \text{sg}[e]\|^2$  — penalizes the encoder for straying too far from the selected embedding  $e$ , and only involves the **encoder**.
- $L_{\text{codebook update}} = \|\text{sg}[z_e(x)] - e\|^2$  — updates the codebook embeddings  $e$ , and does not backpropagate to the encoder or decoder (due to stop-gradient  $\text{sg}$ ).

---

## 2. How Gradients Flow Through the Total Loss

When backpropagating from  $L_{\text{total}}$ , the gradients for the encoder and decoder are computed separately because of how each term affects them.

### a) Decoder Gradients

The decoder receives input  $z_q(x)$ , the quantized output from the encoder (via the codebook). The decoder's task is solely to reconstruct the input  $x$ , so its gradients come **only from the reconstruction loss**:

$$\frac{\partial L_{\text{reconstruction}}}{\partial \text{decoder parameters}}$$

The decoder has no direct dependence on  $z_e(x)$  or the codebook embeddings  $e$ , so the **commitment loss** and **codebook update loss** do not affect the decoder. This is why we say the decoder is trained **only via the reconstruction loss**.

---

### b) Encoder Gradients

The encoder has two tasks:

1. Minimize the reconstruction loss (indirectly, via the decoder).
2. Ensure its outputs  $z_e(x)$  stay close to the selected codebook embeddings  $e$  (commitment loss).

Thus, the encoder gradients are influenced by both:

$$\frac{\partial L_{\text{reconstruction}}}{\partial \text{encoder parameters}} + \frac{\partial L_{\text{commitment}}}{\partial \text{encoder parameters}}$$

↓

### c) Codebook Gradients

The codebook embeddings  $e$  are updated via the **codebook update loss**, using either gradient descent or exponential moving averages (EMA). This process ensures that  $e$  moves closer to  $z_e(x)$ , but does not backpropagate into the encoder or decoder due to the stop-gradient operator  $\text{sg}$ .

## 6. Final Summary

- The **decoder** uses **only reconstruction loss** for gradient updates, even though the total loss includes all terms, because:
  - The decoder's gradients depend only on  $\hat{x}$  (output) and  $z_q(x)$  (input).
- The **encoder** is updated based on both reconstruction and commitment loss.
- The **codebook** is updated separately using the codebook update loss (no gradients flow to the encoder or decoder).

# Sample implementation of VQ-VAE code

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# Set the device to GPU if available, otherwise fallback to CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# Define the VQ-VAE model

#Summary of Functionality of
#Input Quantization: Maps encoder outputs to the nearest embeddings in the codebook.
#Loss Computation: Balances updating the embeddings (codebook loss) and constraining
the encoder's outputs (commitment loss).
#Straight-Through Estimation: Ensures gradients flow correctly during
backpropagation.

class VectorQuantizer(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, commitment_cost):
        super(VectorQuantizer, self).__init__()
        self.num_embeddings = num_embeddings #num_embeddings: Number of discrete
embeddings in the codebook.
        self.embedding_dim = embedding_dim #embedding_dim: Dimensionality of each
embedding vector.
        self.commitment_cost = commitment_cost #commitment_cost: A hyperparameter
that balances the encoder's commitment to the quantized embeddings during training.

        self.embedding = nn.Embedding(num_embeddings, embedding_dim) #self.embedding:
An embedding layer with num_embeddings entries, each of size embedding_dim.
        self.embedding.weight.data.uniform_(-1/self.num_embeddings,
1/self.num_embeddings) #Initialization of weights: The embeddings are initialized
uniformly within a small range.??

    def forward(self, inputs): ##The forward method 1.maps the inputs to the nearest
embedding in the codebook,2. computes the losses, and outputs the quantized tensor
and the loss.
        # Flatten the input

        #1.maps the inputs to the nearest embedding in the codebook

        inputs = inputs.permute(0, 2, 3, 1).contiguous() # Permute: Changes the
input tensor's shape to move the channel dimension to the last (e.g., [B, C, H, W] →
[B, H, W, C]), making it easier to process spatial features.
        input_shape = inputs.shape
```

```
flat_inputs = inputs.view(-1, self.embedding_dim) # Flatten: Reshapes the
input to a 2D tensor of shape [N, embedding_dim] where N= B × H × W.
```

```
# Calculate distances and find closest embeddings : This computes the squared
Euclidean distance between each input vector and each embedding vector in the
codebook:
```

```
distances = (torch.sum(flat_inputs**2, dim=1, keepdim=True)
              + torch.sum(self.embedding.weight**2, dim=1)
              - 2 * torch.matmul(flat_inputs, self.embedding.weight.t()))
```

```
encoding_indices = torch.argmax(distances, dim=1).unsqueeze(1) #
torch.argmax: Finds the index of the closest embedding for each input vector based on
the computed distances.
```

```
encodings = torch.zeros(encoding_indices.size(0), self.num_embeddings,
device=inputs.device)
```

```
encodings.scatter_(1, encoding_indices, 1) # Creates a one-hot encoded
matrix (encodings) where the closest embedding index is set to 1 for each input
vector.
```

```
quantized = torch.matmul(encodings, self.embedding.weight).view(input_shape)
#The one-hot encoded encodings is used to select the corresponding embedding vectors
from the codebook.
```

```
#torch.matmul: Performs a matrix multiplication between encodings and the embedding
weights to retrieve the quantized vectors.
```

```
#Reshape: Converts the quantized vectors back to the original input shape
```

```
#2. computes the losses
```

```
# Compute losses
```

```
e_latent_loss = F.mse_loss(quantized.detach(), inputs) #e_latent_loss
(Encoder Commitment Loss):
```

```
#Ensures the encoder
outputs vectors close to the chosen embeddings.
```

```
#Uses a detached
quantized tensor to prevent gradient updates to the embeddings(codebook).
```

```
q_latent_loss = F.mse_loss(quantized, inputs.detach()) #q_latent_loss
(Codebook Update Loss):
```

```
#Updates the
embeddings to match the encoder's output.
```

```
#Uses a detached
input tensor to prevent encoder updates.
```

```
loss = q_latent_loss + self.commitment_cost * e_latent_loss # loss =
Codebook Update Loss +  $\beta$  × Commitment Loss, where  $\beta$  is the commitment_cost.
```

```
#Straight-Through Estimator (STE): Ensures gradients bypass non-
differentiable steps in the quantization process.
```

```
quantized = inputs + (quantized - inputs).detach()
```

```
#Combines the
quantized tensor with the input using a straight-through gradient estimator:
```

#During the forward pass, it outputs the quantized tensor.

#During the backward pass, it passes gradients to the encoder as if the quantized tensor were equal to the input.

```
    return quantized.permute(0, 3, 1, 2).contiguous(), loss    # Quantized  
Output: Reshaped back to [B, C, H, W].
```

```
class VQVAE(nn.Module):  
    def __init__(self, input_channels, hidden_channels, latent_dim, num_embeddings,  
commitment_cost):  
  
        #input_channels: Number of input channels in the data (e.g., 3 for RGB images  
or 1 for grayscale images).  
        #hidden_channels: Number of feature maps (channels) in the hidden layers.  
        #latent_dim: Dimensionality of the latent space representation.  
        #num_embeddings: Number of discrete embeddings in the vector quantizer  
(codebook).  
        #commitment_cost ( $\beta$ ): Regularization parameter to encourage the encoder to  
commit to the quantized embeddings.  
  
        super(VQVAE, self).__init__()  
        self.encoder = nn.Sequential(  
            nn.Conv2d(input_channels, hidden_channels, kernel_size=4, stride=2,  
padding=1),  
            nn.ReLU(),  
            nn.Conv2d(hidden_channels, hidden_channels, kernel_size=4, stride=2,  
padding=1),  
            nn.ReLU(),  
            nn.Conv2d(hidden_channels, latent_dim, kernel_size=3, stride=1,  
padding=1)  
        )  
  
        #The decoder reconstructs the input from the quantized latent representation  
        #It uses transposed convolutional layers to upsample the feature maps back to the  
original spatial dimensions.  
        self.decoder = nn.Sequential(  
            nn.ConvTranspose2d(latent_dim, hidden_channels, kernel_size=4, stride=2,  
padding=1),  
            nn.ReLU(),  
            nn.ConvTranspose2d(hidden_channels, hidden_channels, kernel_size=4,  
stride=2, padding=1),  
            nn.ReLU(),  
            nn.ConvTranspose2d(hidden_channels, input_channels, kernel_size=3,  
stride=1, padding=1),  
            nn.Sigmoid() # Sigmoid: Normalizes the output values to [0,1], making it  
suitable for image reconstruction tasks.  
        )
```

```

        self.vq = VectorQuantizer(num_embeddings, latent_dim, commitment_cost) # The
VectorQuantizer module (defined earlier) is responsible
#
for quantizing the latent representation  $z_e$  into discrete embeddings  $z_q$ .

    def forward(self, x):
        z_e = self.encoder(x)
        z_q, vq_loss = self.vq(z_e)
        x_recon = self.decoder(z_q)
        return x_recon, vq_loss

# Set hyperparameters
input_channels = 1 # For MNIST
hidden_channels = 128
latent_dim = 64
num_embeddings = 512
commitment_cost = 0.25 # ( $\beta$ )
batch_size = 64
learning_rate = 1e-3
num_epochs = 10

# Load the MNIST dataset
transform = transforms.Compose([transforms.ToTensor()])
train_dataset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

# Initialize the model, optimizer, and loss function
model = VQVAE(input_channels, hidden_channels, latent_dim, num_embeddings,
commitment_cost).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
for epoch in range(num_epochs):
    model.train()
    total_loss, recon_loss, vq_loss = 0, 0, 0

    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)

        # Forward pass
        recon, vq_loss_batch = model(data) # vq_loss_batch: The vector quantization
loss for this batch. include both encoder comitment + codebook updation loss

        # Reconstruction loss
        recon_loss_batch = F.mse_loss(recon, data)

        # Total loss
        loss = recon_loss_batch + vq_loss_batch

        # Backward pass

```



```

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

        total_loss += loss.item()
        recon_loss += recon_loss_batch.item()
        vq_loss += vq_loss_batch.item()

    print(f"Epoch [{epoch+1}/{num_epochs}], Total Loss: {total_loss:.4f}, "
          f"Reconstruction Loss: {recon_loss:.4f}, VQ Loss: {vq_loss:.4f}")

import matplotlib.pyplot as plt

# Set the model to evaluation mode
model.eval()

# Generate input and reconstructed images
with torch.no_grad():
    # Get a batch of data
    sample_data, _ = next(iter(train_loader))
    sample_data = sample_data.to(device)

    # Generate reconstructed images
    reconstructed, _ = model(sample_data)

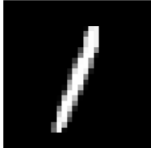




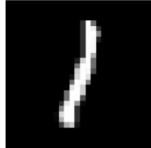


# Prepare the plot
num_images = 8 # Number of images to display
fig, axes = plt.subplots(2, num_images, figsize=(num_images * 2, 5))

    for i in range(num_images):
        # Original image
        axes[0, i].imshow(sample_data[i].cpu().squeeze(), cmap='gray')
        axes[0, i].set_title("Original")
        axes[0, i].axis('off')

        # Reconstructed image
        axes[1, i].imshow(reconstructed[i].cpu().squeeze(), cmap='gray')
        axes[1, i].set_title("Reconstructed")
        axes[1, i].axis('off')

# Adjust layout for better visualization
plt.tight_layout()
plt.show()

```

Original	Original	Original	Original	Original	Original	Original	Original
							
Reconstructed	Reconstructed	Reconstructed	Reconstructed	Reconstructed	Reconstructed	Reconstructed	Reconstructed
