

Context: Undergraduate Project "Machine Learning" at TU Dortmund

Based on:

- *"Auto-Encoding Variational Bayes"* by Diederik P. Kingma and MaxWelling
- *"Neural Discrete Representation Learning"* by Aaron van den Oord, Oriol Vinyals and Koray Kavukcuoglu

Project Summary: VAE, VQ-VAE

Presented on **17.07.2024** by Anastasiia Korzhylova,
Ivan Shishkin, Ramneek Agnihotri and Rodi Mehi

Table of contents

- 01** Theory Revision:
VAE vs VQ-VAE
- 02** Difficulties during
Implementation

- 03** VAE:
Optimizations
- 04** VQ-VAE:
Optimizations
- 05** Final Results &
Retrospective

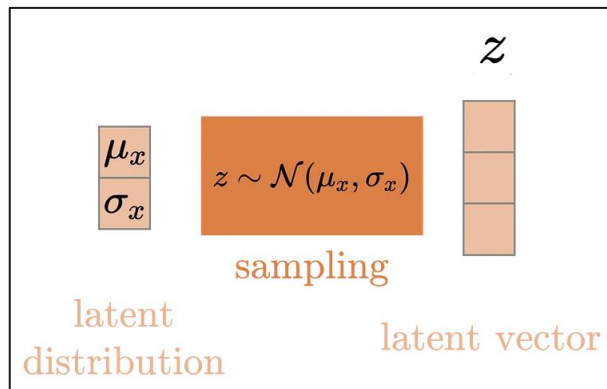
01

Theory Revision: VAE vs VQ-VAE

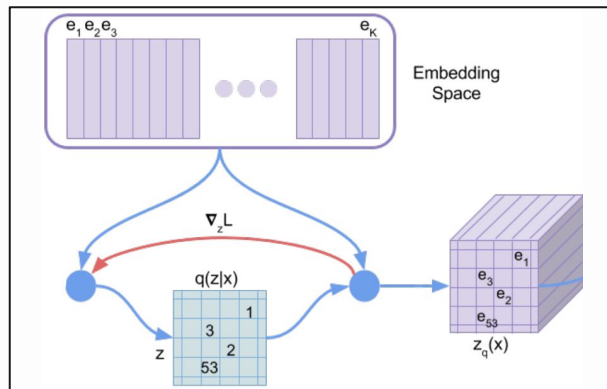
Comparison Table

Feature	VAE	VQ-VAE
Latent Space	Gaussian distribution (Continuous)	Discrete codebook
Loss Function	Reconstruction loss & KL divergence	Reconstruction loss & commitment loss & codebook loss
Complexity	Simpler, fewer hyperparameters	More complex, requires tuning of codebook size and commitment weight
Training Stability	Can suffer from posterior collapse	Typically more stable due to discrete latent space
Reconstruction	Generally good, but blurry	High fidelity due to discrete latent space

VAE



VQ-VAE



Comparison Table

Feature	VAE	VQ-VAE
Latent Space	Gaussian distribution (Continuous)	Discrete codebook
Loss Function	Reconstruction loss & KL divergence	Reconstruction loss & commitment loss & codebook loss
Complexity	Simpler, fewer hyperparameters	More complex, requires tuning of codebook size and commitment weight
Training Stability	Can suffer from posterior collapse	Typically more stable due to discrete latent space
Reconstruction	Generally good, but blurry	High fidelity due to discrete latent space

VAE

$$\text{ELBO}(\theta, \phi) = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] - \text{KL}(q_{\phi}(z|x) || p(z))$$

VQ-VAE

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

Comparison Table

Feature	VAE	VQ-VAE
Latent Space	Gaussian distribution (Continuous)	Discrete codebook
Loss Function	Reconstruction loss & KL divergence	Reconstruction loss & commitment loss & codebook loss
Complexity	Simpler, fewer hyperparameters	More complex, requires tuning of codebook size and commitment weight
Training Stability	Can suffer from posterior collapse	Typically more stable due to discrete latent space
Reconstruction	Generally good, but blurry	High fidelity due to discrete latent space

VAE

```
batch_size = 128
num_epochs = 100
latent_dim = 256
input_channels = 3
learning_rate = 5e-5
output_frequency = 150
```

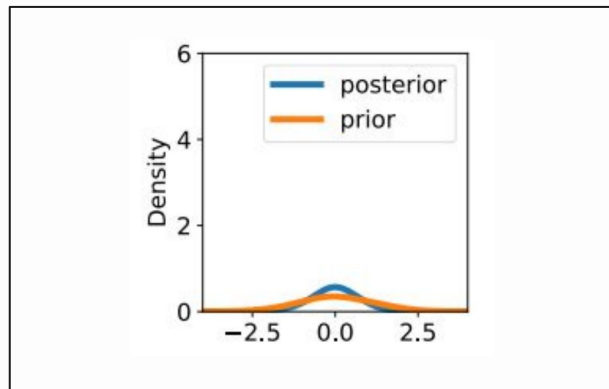
VQ-VAE

```
batch_size = 128
num_epochs = 30
input_channels = 3
learning_rate = 2e-5
output_frequency = 150
number_embedding_vectors = 100
embedding_dimension = 128
beta = 0.1
```

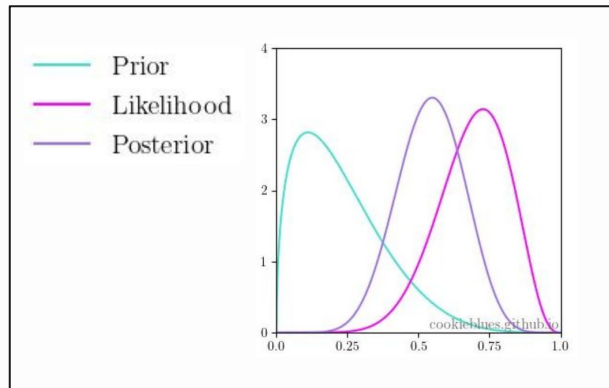
Comparison Table

Feature	VAE	VQ-VAE
Latent Space	Gaussian distribution (Continuous)	Discrete codebook
Loss Function	Reconstruction loss & KL divergence	Reconstruction loss & commitment loss & codebook loss
Complexity	Simpler, fewer hyperparameters	More complex, requires tuning of codebook size and commitment weight
Training Stability	Can suffer from posterior collapse	Typically more stable due to discrete latent space
Reconstruction	Generally good, but blurry	High fidelity due to discrete latent space

VAE



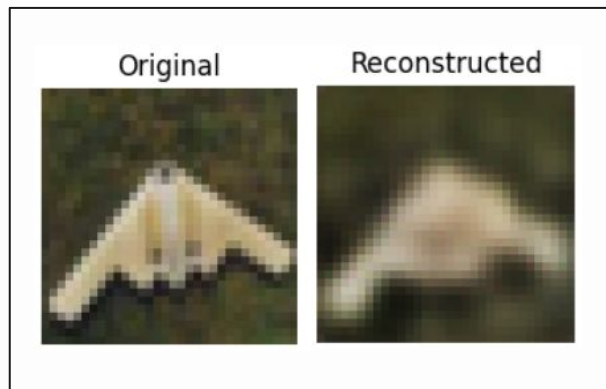
VQ-VAE



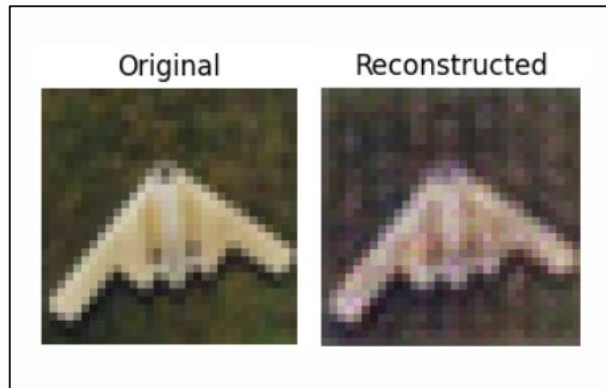
Comparison Table

Feature	VAE	VQ-VAE
Latent Space	Gaussian distribution (Continuous)	Discrete codebook
Loss Function	Reconstruction loss & KL divergence	Reconstruction loss & commitment loss & codebook loss
Complexity	Simpler, fewer hyperparameters	More complex, requires tuning of codebook size and commitment weight
Training Stability	Can suffer from posterior collapse	Typically more stable due to discrete latent space
Reconstruction	Generally good, but blurry	High fidelity due to discrete latent space

VAE



VQ-VAE



02

Difficulties during Implementation

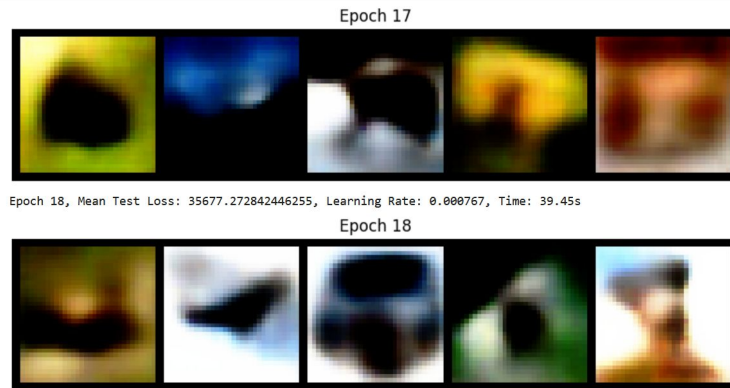
Incorrect Normalization

Initially, we downloaded and normalized the dataset with incorrect parameters, which caused the appearance of **shapeless black spots** in the sampled images:

```
transforms.Normalize((0.4,0.4,0.4), (0.4,0.4,0.4))
```

As we couldn't explain the origin of the black spots, we were looking for an error in the training / model, while it actually hid in the **dataset transformation** step.

The normalization parameters are supposed to be (0.5,0.5,0.5), (0.5,0.5,0.5), as the images should be encoded in the range [-1,1]. Parameter mismatch was generally one of our most frequent errors.



We also used a **Color Jitter** in the hope to improve robustness and generalization capability of the model, but it turned out to be counterproductive to generating good reconstructions.

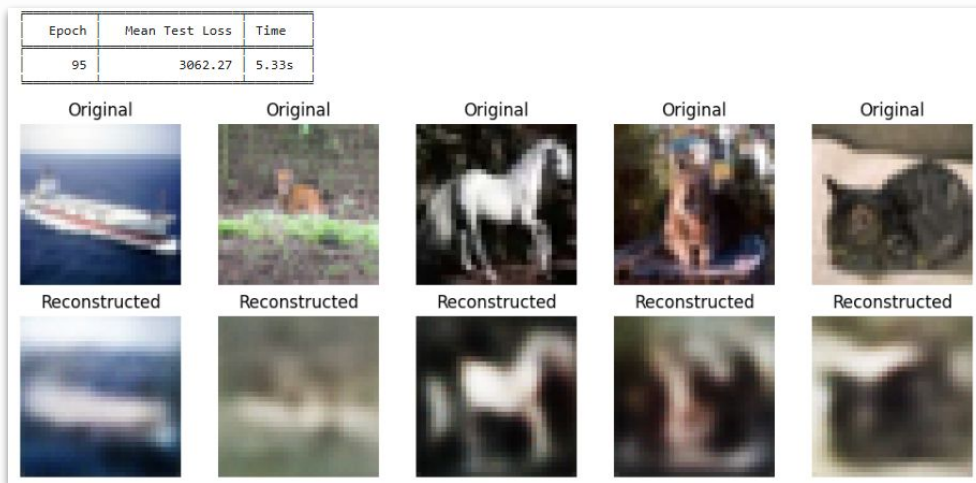
Training Stagnation & Misleading Focus

- In the early stages of our VAE implementation, we noticed that the reconstructed and sampled **image quality stopped improving** after around 10 epochs, while still being too blurry. When looking for a way to improve our model, we spent a lot of time **optimizing the hyperparameters** rather than changing the model structure or the loss function.
- One reason for that was focusing too much on improving the **numerical indicators** of training success. For example, we saw that decreasing the batch size and increasing the number of epochs significantly **reduced the test loss**, but were wondering why the reconstructions were not getting any better.



Training Stagnation & Misleading Focus

Changing the batch size (128), the learning rate ($5e-5$) and the latent dimension size (256) was just one part of the necessary improvements. Adding **more layers** to the model, **reducing the weight of the KL divergence** in the loss function, adding **bias to the linear layer** and introducing some state-of-the-art optimizations actually made the reconstructions look recognizable.



03

VAE: Optimizations

Network Optimization

```
class Encoder(nn.Module):
    def __init__(self, input_channels, latent_dim):
        super(Encoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(input_channels, 64, 4, 2, 1,
                bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU() )
        self.fc1 = nn.Linear(128 * 8 * 8, 1024)
        self.fc2_mean = nn.Linear(1024, latent_dim)
        self.fc2_logvar = nn.Linear(1024, latent_dim)

    def forward(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1)
        h = F.relu(self.fc1(x))
        mean = self.fc2_mean(h)
        logvar = self.fc2_logvar(h)
        return mean, logvar
```

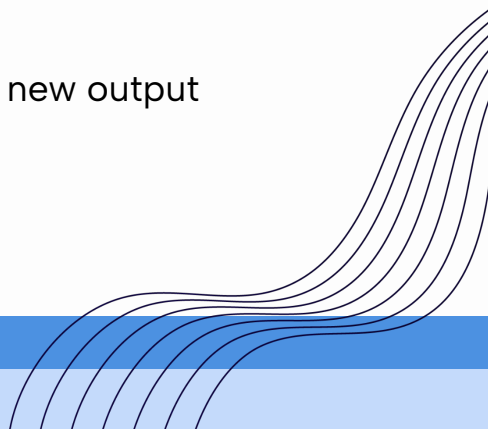
```
class Decoder(nn.Module):
    def __init__(self, latent_dim):
        super(Decoder, self).__init__()
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 1024, bias=False),
            nn.ReLU(),
            nn.Linear(1024, 128 * 8 * 8, bias=False),
            nn.ReLU() )
        self.deconv = nn.Sequential(
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 3, 4, 2, 1) )

    def forward(self, z):
        z = self.decoder(z)
        z = z.view(z.size(0), 128, 8, 8)
        z = self.deconv(z)
        z = torch.sigmoid(z)
        return z
```

Network Optimization

`torch.nn.Dropout(p=0.5, inplace=False)`

- **Regularization:** `torch.nn.Dropout(p=0.5)` helps to prevent overfitting by randomly deactivating 50% of neurons during training.
- **Parameter p :** The probability `p=0.5` sets the dropout rate, meaning that 50% of neuron activations are set to zero.
- **Parameter *inplace*:** `inplace=False` ensures that the operation creates a new output tensor, leaving the input unchanged.



Training Optimization

`torch.optim.Adam(model.parameters(), lr=learning_rate)`

- combines adaptive learning rates and momentum for efficient optimization.

`torch.optim.AdamW(model.parameters(), lr=learning_rate)`

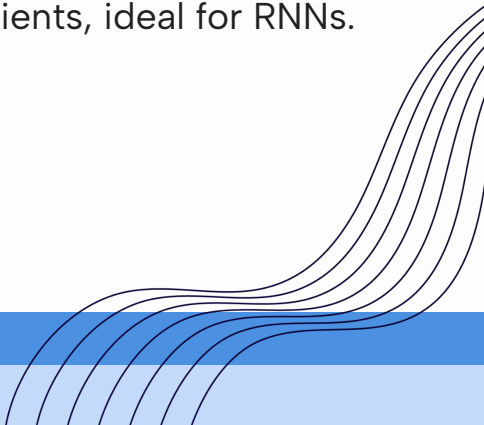
- uses Adam with decoupled weight decay for better regularization.

`torch.optim.RMSprop(model.parameters(), lr=learning_rate)`

- adapts learning rates based on a moving average of recent gradients, ideal for RNNs.

`torch.optim.SGD(model.parameters(), lr=learning_rate)`

- performs standard stochastic gradient descent, can include momentum for acceleration.



Training Optimization

scheduler = **StepLR**(optimizer, step_size=30, gamma=0.5)

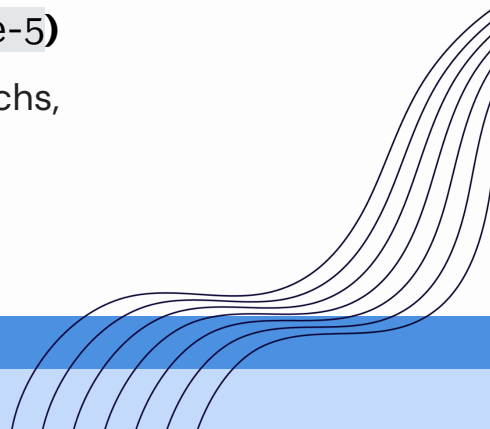
- reduces the learning rate by half every 30 epochs to improve convergence.

scheduler = **ExponentialLR**(optimizer, gamma=0.9)

- decreases the learning rate exponentially by 10% at each epoch for gradual reduction.

scheduler = **CosineAnnealingLR**(optimizer, T_max=50, eta_min=1e-5)

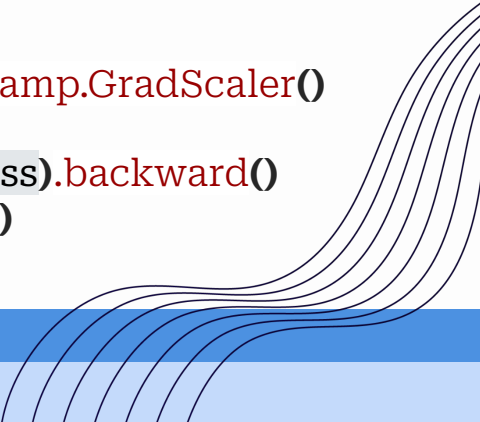
- adjusts the learning rate following a cosine curve over 50 epochs, reaching a minimum of 1e-5.



Training Optimization

- [Automatic Mixed Precision \(AMP\)](#) speeds up the training and reduces memory consumption by using both 32-bit and 16-bit floating-point types in training.
- This is achieved by performing the **forward pass** and **matrix multiplications** in **FP16**, and then **scaling the loss** and **accumulating the gradients** in **FP32** to avoid precision loss during backpropagation.
- Using FP16 halves the memory consumption for storing tensors, allowing larger batch sizes, deeper models and higher throughput.

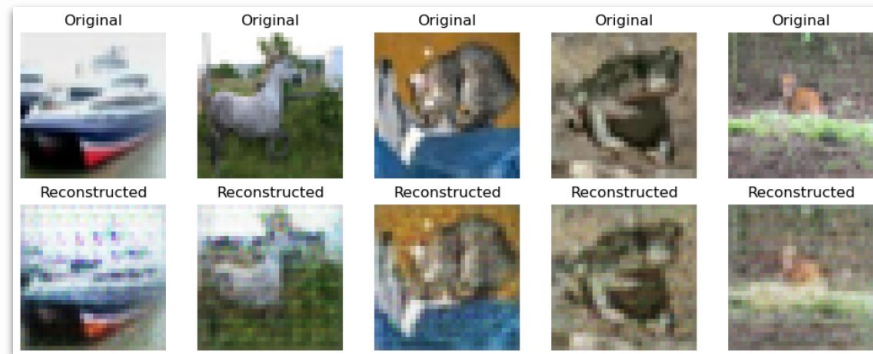
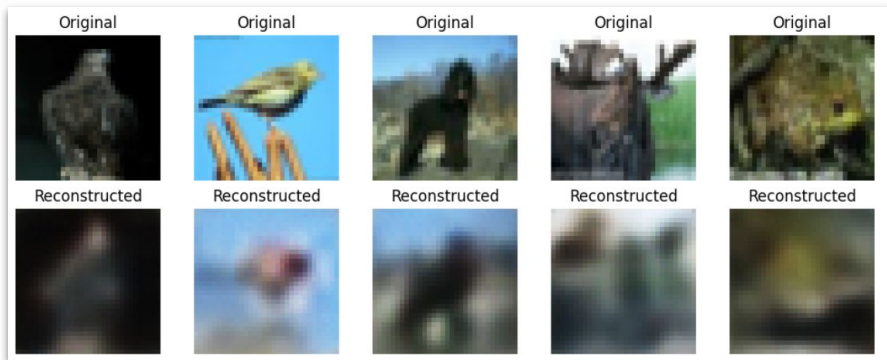
```
scaler = torch.cuda.amp.GradScaler()  
...  
scaler.scale(loss).backward()  
scaler.update()
```

A series of thin, dark blue wavy lines that originate from the bottom right corner and curve upwards and to the left, ending near the center of the slide.

04

VQ-VAE: Optimizations

Reconstructions: From suboptimal to good



The VAE model performed suboptimally. Through Vector Quantization, optimizations and model changes, significant improvements were made.

Improvement: Encoder/Decoder adjustments

```
def __init__(self, input_channels, embedding_dim):
    super(Encoder, self).__init__()
    self.conv1 = nn.Conv2d(input_channels, 64, kernel_size=4, stride=2, padding=1)
    self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
    self.conv3 = nn.Conv2d(128, embedding_dim, kernel_size=4, stride=2, padding=1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.conv3(x)
        return x
```

The model used to have too little complexity, causing it not to learn enough patterns. Lack of Batch Normalization, as well as the issue of exploding and vanishing gradients were not addressed. Idea: **add more layers!**

Improvement: Encoder/Decoder adjustments

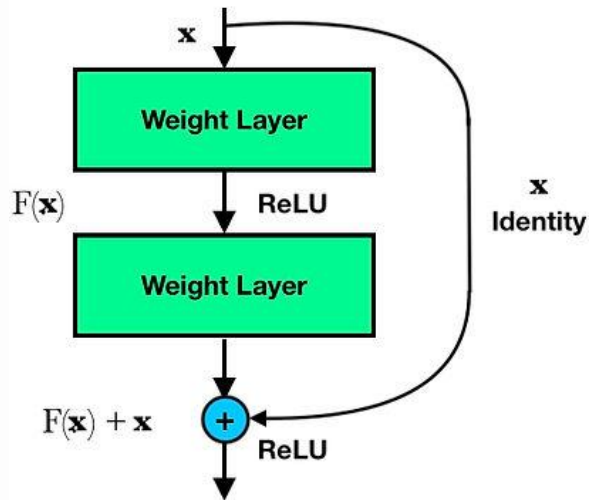


Batch Norm added,
but too many layers!
=> **model overfitting**

```
super(Encoder, self).__init__()
self.conv = nn.Sequential(
    nn.Conv2d(input_channels, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Conv2d(64, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(),
    nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(512),
    nn.ReLU(),
)
```

Improvement: Residual Block

```
class ResidualBlock(nn.Module):  
    def __init__(self, in_channels, out_channels):  
        super(ResidualBlock, self).__init__()  
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)  
        self.conv2 = nn.Conv2d(in_channels, out_channels, kernel_size=1)  
        self.bn1 = nn.BatchNorm2d(out_channels)  
        self.relu = nn.ReLU()  
  
    def forward(self, x):  
        identity = x  
        out = self.relu(self.bn1(self.conv1(x)))  
        out = self.bn1(self.conv2(out))  
        out += identity  
        return out
```



- Output is $f(x) + x$
- Addresses the problem of **vanishing gradients** by propagating x **directly** to the output (gradients don't become too small)
- Model can learn the **identity function** if the transformations do not improve the performance. This ensures that adding more layers **does not degrade** the model.

Final Encoder Architecture

Adjusted model by:

- adding Batch Normalization
- adding two Residual Blocks

Philosophy:

- Inspired by the architecture described in the **second paper** (especially the usage of resnets)
- Refined through **experimentation**

```
def __init__(self, input_channels, emb_dim):
    super(Encoder, self).__init__()
    self.conv1 = nn.Conv2d(input_channels, 128, kernel_size=4, stride=2, padding=1)
    self.bn1 = nn.BatchNorm2d(128)
    self.conv2 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
    self.bn2 = nn.BatchNorm2d(256)
    self.resblock1 = ResidualBlock(256,256)
    self.resblock2 = ResidualBlock(256,256)
    self.conv3 = nn.Conv2d(256, emb_dim, kernel_size=3, padding=1)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.resblock1(x)
        x = self.resblock2(x)
        x = self.conv3(x)
        return x
```


Improvement: Hyperparameter optimization

Previous Hyperparameters

Number of Embedding Vectors = 512
Embedding Dimension = 512
Commitment cost = 0.25
Learning Rate = $5e-5$
Optimizer = AdamW
Learning Rate = Cosine

Optimized Hyperparameters:

Number of Embedding Vectors = 100
Embedding Dimension = 128
Commitment cost = 0.1
Learning Rate = $2e-5$
Optimizer = Adam
Learning Rate = Exponential

Note: other hyperparameters stayed the same

- Less and lower dimensional embedding vectors, more appropriate for the model
- Commitment cost was forgotten initially, added later
- Learning rate was adjusted
- Weight decay not required

Improvement: Channel Last Memory Format

- Encoder output tensor was transformed from **BCHW** to **BHWC** before doing the transformation step.
- BCHW = (Batch Size, Channels, Height, Width)
- BHWC = (Batch Size, Height, Width, Channels)

Trained using BCHW

Original



Reconstructed



Trained using BHWC

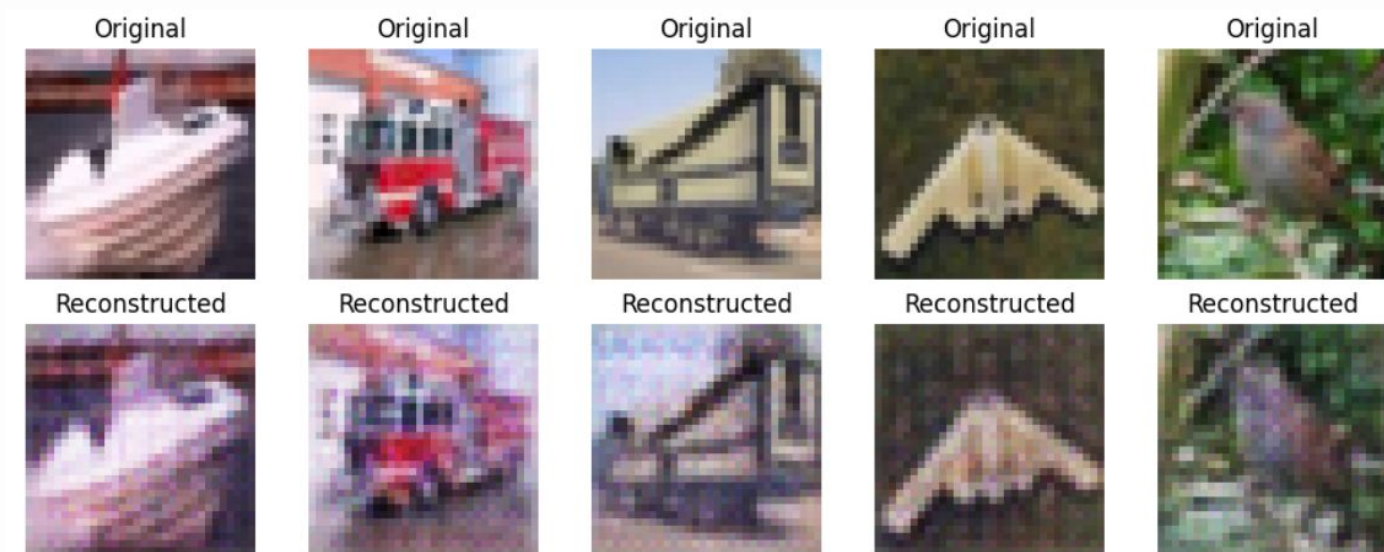
Original



Reconstructed



How can the results be further improved?

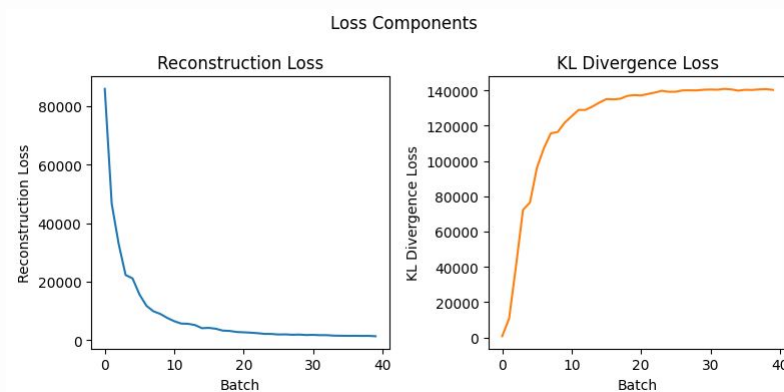
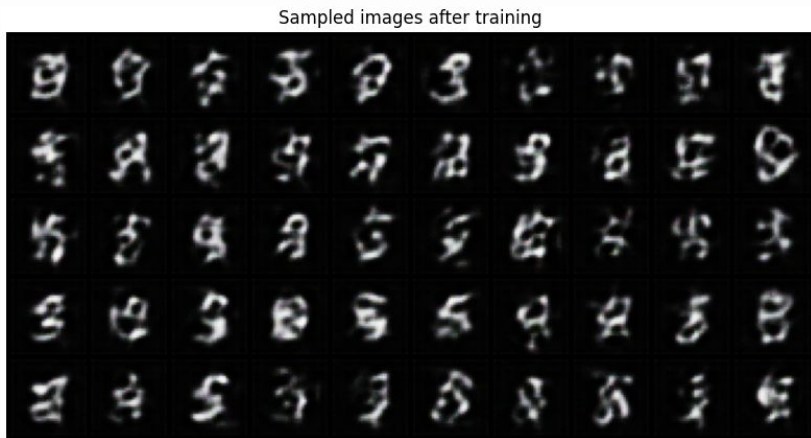
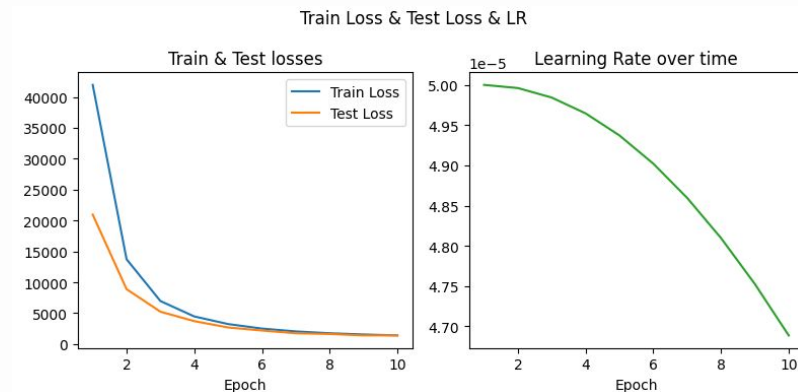
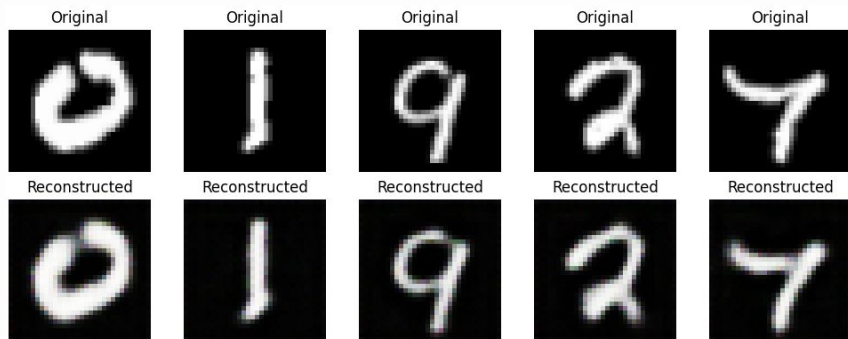


- Usage of **clustering algorithms** for embedding vectors
- Using a **different distribution** for initial embedding weights (uniform is better)
- Training with **more epochs**, adjusting **hyperparameters**, trying a different **learning approach**

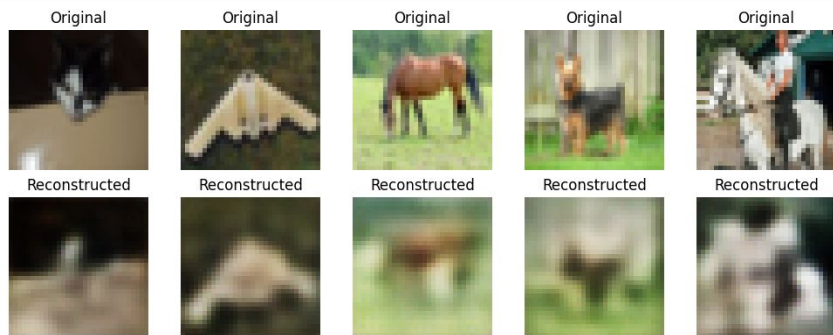
05

Final Results & Retrospective

VAE: MNIST Dataset after 10 epochs



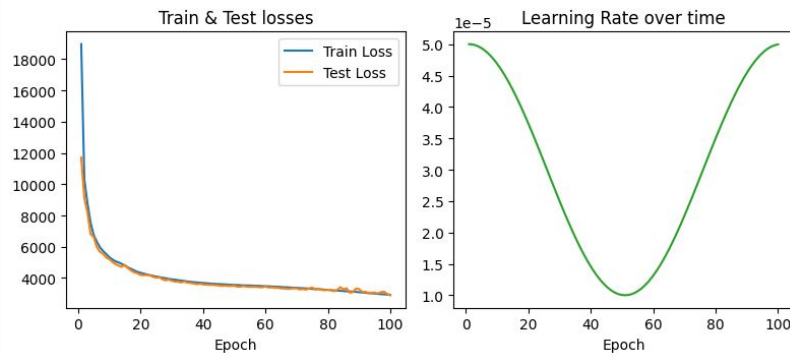
VAE: CIFAR-10 Dataset after 100 epochs



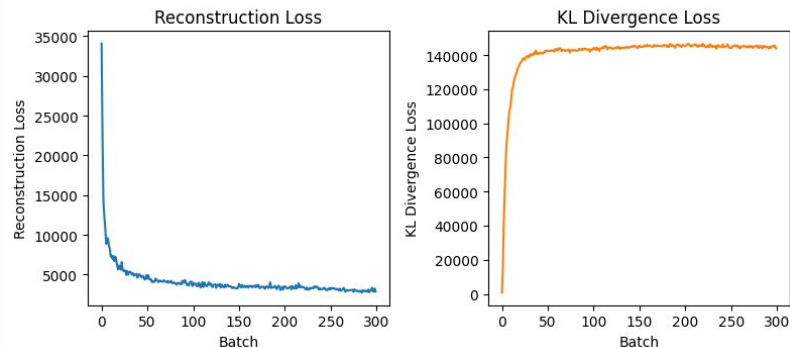
Sampled images after training



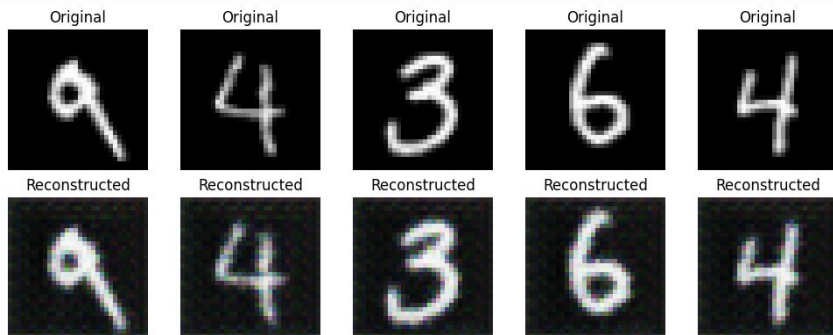
Train Loss & Test Loss & LR



Loss Components



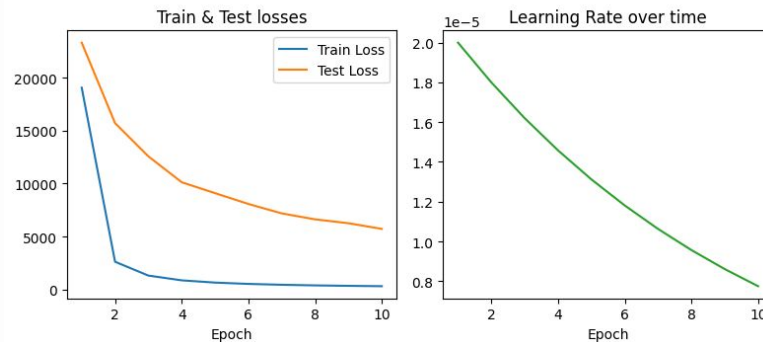
VQ-VAE: MNIST Dataset after 10 epochs



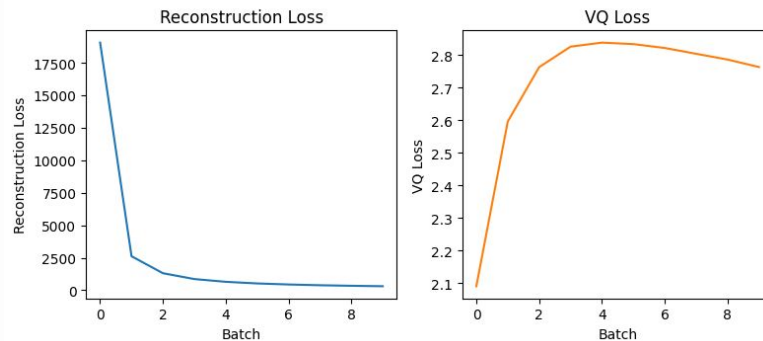
Vector Interpolation



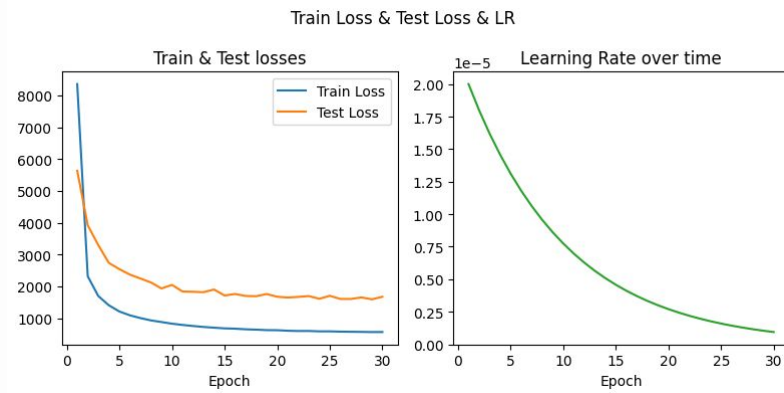
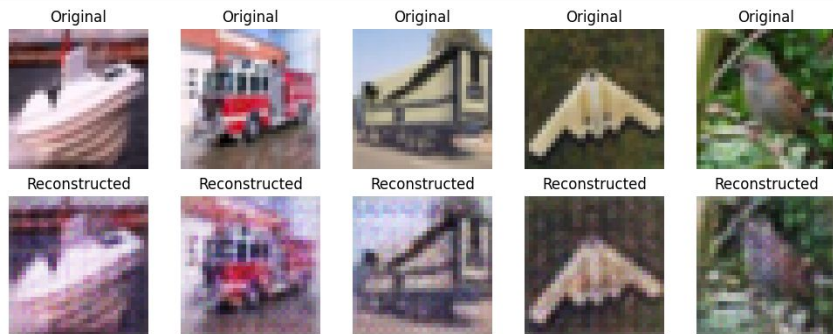
Train Loss & Test Loss & LR



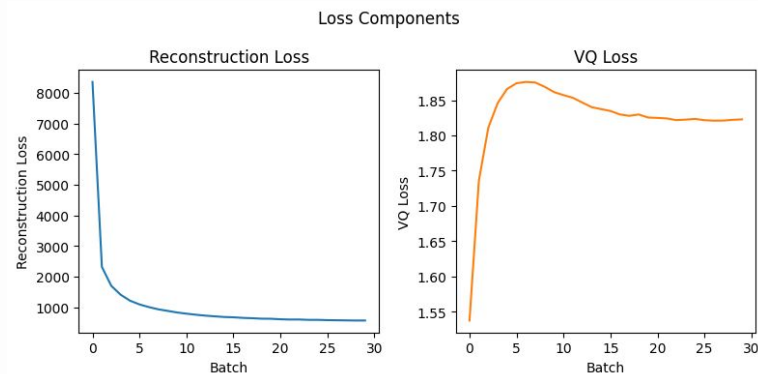
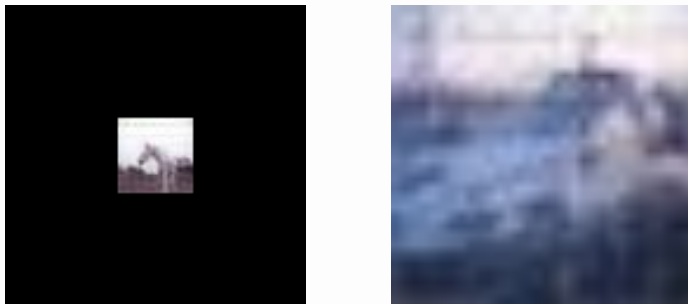
Loss Components



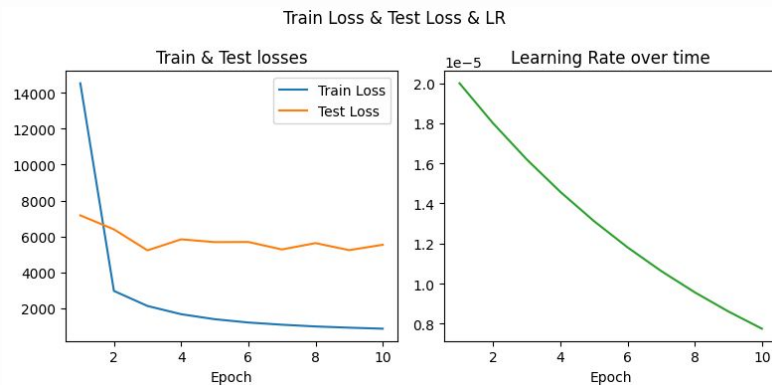
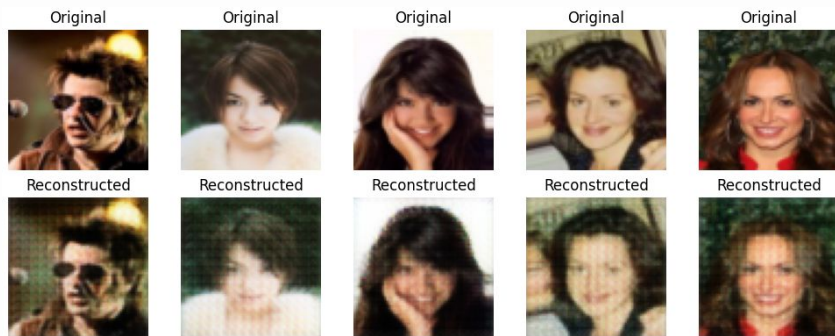
VQ-VAE: CIFAR-10 Dataset after 30 epochs



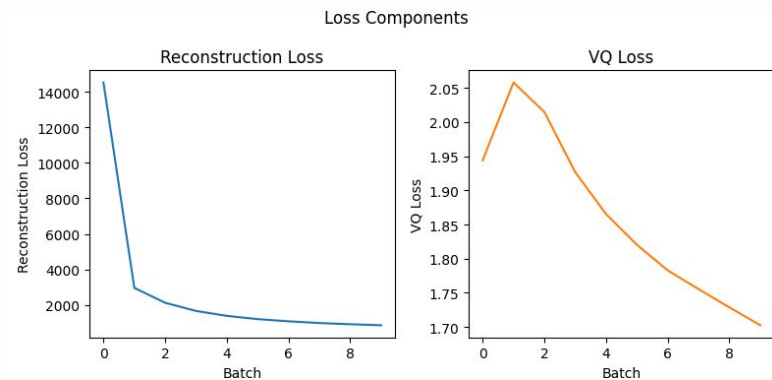
Vector Interpolation



VQ-VAE: CelebA Dataset after 10 epochs



Vector Interpolation



Teamwork Retrospective

After completing the Undergraduate Project, we believe that not just the results of our work were a success, but also the **organization of our teamwork**.

- Fair task distribution within the group,
- Setting intermediate goals and deadlines,
- Members' active participation in discussions,
- Pair programming and debugging,
- Asynchronous communication and Git usage

are all examples of things that we handled **smoothly**. Everybody contributed their best effort to the team with quite of an enthusiasm and curiosity. 🤝



Thanks!

Do you have any questions?



CREDITS: This presentation template was created by [Slidesgo](#), and includes icons by [Flaticon](#), and infographics & images by [Freepik](#)