VQ-VAE and Transformer

Songlin Zhao

August 27, 2023

Abstract

This report presents my implementation of an approach of generating new samples using VQ-VAE and transformer. This work involves a two-stage process for image generation. I first implemented a ResNet-style encoder and decoder for the VQ-VAE to reconstruct the input data, and then a transformer to model the latent representations of the VQ-VAE. Experiments were conducted on MNIST, Fashion-MNIST, and CelebA. The results demonstrate the capability of VQ-VAE for reconstruction and Transformers in learning the distribution of a sequence of discrete latents. A comparative analysis was performed to reveal how specific parameters can influence the model capability.

1 Background

1.1 AE and VAE

An autoencoder consists of two parts: an encoder that transforms the original data into a lower-dimensional latent representation, and a decoder that reconstructs the original data from this latent representation. However, autoencoders cannot generate new samples because their latent space is unstructured. In contrast, VAEs map the original data to distributions. By imposing constraints on these distributions, random generation can be achieved by sampling from the specified distribution. Nonetheless, samples generated from VAEs tend to be blurry. This blurriness arises because, during training, latent representations are sampled from a distribution. The decoder then has to reconstruct the input data from any potential samples drawn from this distribution. Consequently, the decoder often produces an averaged reconstruction, resulting in blurry outputs. This phenomenon is called the "averaging effect."

1.2 Transformer

The Transformer model is particularly effective in handling sequential data. The self-attention layer in transformer allows it to capture long-range dependencies in the data, making it suitable for tasks like translation and text generation.

2 Details

2.1 VQ-VAE Architecture

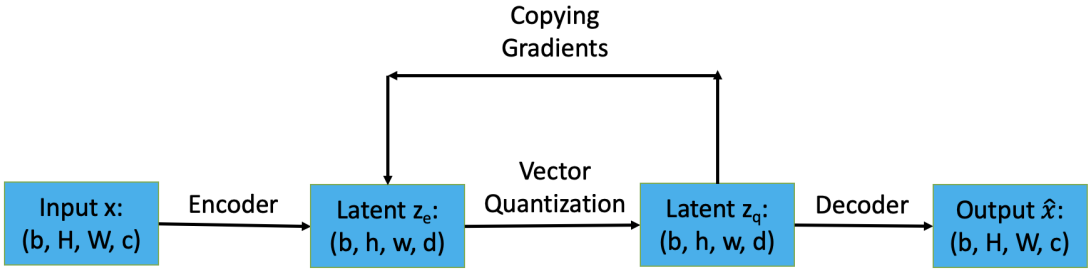


Figure 1: VQ-VAE Architecture

The overall architecture of VQ-VAE is shown above. b is batch size, H and W is the height and width of input data, and c is the number of channels of the input data. The Encoder(typically CNN) maps the input data to z_e which is of shape (b, h, w, d). h and w is smaller than H and W. Instead of mapping original data to a continuous latent space, VQ-VAE maps it to a discrete latent space by vector quantization layer(which will be discussed in the next section). The Decoder(typically CNN) transforms the quantized latent representations into $\hat{x}$, and the model aims to train the encoder and decoder so that $\hat{x}$ is as close to input x as possible. In essence, VQ-VAE is more like AE instead of VAE, aiming to do reconstruction rather than generation.

2.2 Vector Quantization Layer

The vector quantization layer is responsible for mapping each d-dimensional vector in encoder output z_e, to the nearest discrete latent vector in the codebook. The codebook is shown below, it has K vectors, each of dimension d. VQ layer computes the Euclidean distance between each d-dimensional vector in z_e and all the vectors in the codebook, and selecting the one with the smallest distance. This results in a discrete latent representation z_q, in which each d-dimensional vector is taken from the codebook. Now, each image in the original data is compressed into a smaller image of size (h, w), where each pixel is the index of the codebook(or the codes).

2.3 Straight-through Estimator

One problem this network facing is that vector quantization layer involves argmin operation, which is non-differentiable. To address this, the original paper uses straight-through estimator approach. Specifically, during the backward pass, the gradient of the loss with respect to z_q is directly passed to z_e, as if the vector quantization layer was just an identity operation.

During implementation, it uses stop gradient operator:

$$sg(x) = \begin{cases} x(\text{ in forward propagation }) \\ 0(\text{ in backward propagation }) \end{cases} \tag{1}$$

When calculating reconstruction loss:

$$L_{\text{reconstruct}} = \|x - \text{decoder}\left(z_e(x) + sg\left(z_q(x) - z_e(x)\right)\right)\|_2^2$$

In this case, z_q is used to calculate $L_{\text{reconstruct}}$ in forward pass:

$$L_{\text{reconstruct}} = \|x - \text{decoder}\left(z_q(x)\right)\|_2^2$$

While in backward pass, $sg\left(z_q(x) - z_e(x)\right) = 0$, the gradients were passed directly to z_e:

$$L_{\text{reconstruct}} = \|x - \text{decoder}\left(z_e(x)\right)\|_2^2$$

In pytorch, $\left(z_q(x) - z_e(x)\right).detach()$ represents $sg\left(z_q(x) - z_e(x)\right)$, which is used as the input to the decoder.

2.4 Train the Codebook

The vectors in the codebook are trainable. Since during backpropagation, the gradient of z_q is used to update z_e, ideally, $z_q = z_e$. As a result, we want the following to be minimized:

$$L_e = \|z_e(x) - z_q(x)\|_2^2 \tag{2}$$

However, the author of VQ-VAE pointed out that we should make z_q approach z_e, rather than make z_e approach z_q. This is because we want both the encoder and decoder to perform similarly to how they would without the vector quantization layer. As a result, the author proposed to modify L_e:

$$L_e = \underbrace{\|sg\left(z_e(x)\right) - z_q(x)\|_2^2}_{\text{codebook loss}} + \underbrace{\beta\,\|z_e(x) - sg\left(z_q(x)\right)\|_2^2}_{\text{commitment loss}} \tag{3}$$

The stop gradient operation stops gradient flow. In this case, during backward pass, codebook loss is used to update z_q and commitment loss is used to update z_e. β controls the updating speed of the encoder's outputs relative to the codebook vectors. In practice, β is usually set to be less than 1, making codebook vectors move towards the encoder output. By controlling codebook loss and commitment loss, the encoder learns to generate more accurate and consistent latent representations, and the codebook refines its vectors to better represent the data distribution. The overall loss is as follows:

$$\begin{aligned} L &= L_{\text{reconstruct}} + \alpha L_e \\ &= \|x - \text{decoder}\left(z_e(x) + sg\left(z_q(x) - z_e(x)\right)\right)\|_2^2 \\ &+ \alpha\,\|sg\left(z_e(x)\right) - z_q(x)\|_2^2 + \alpha\beta\,\|z_e(x) - sg\left(z_q(x)\right)\|_2^2 \end{aligned} \tag{4}$$

where the $L_{\text{reconstruct}}$ prompts the model to achieve better reconstruction, and L_e prompts the model to make z_q better align with z_e.

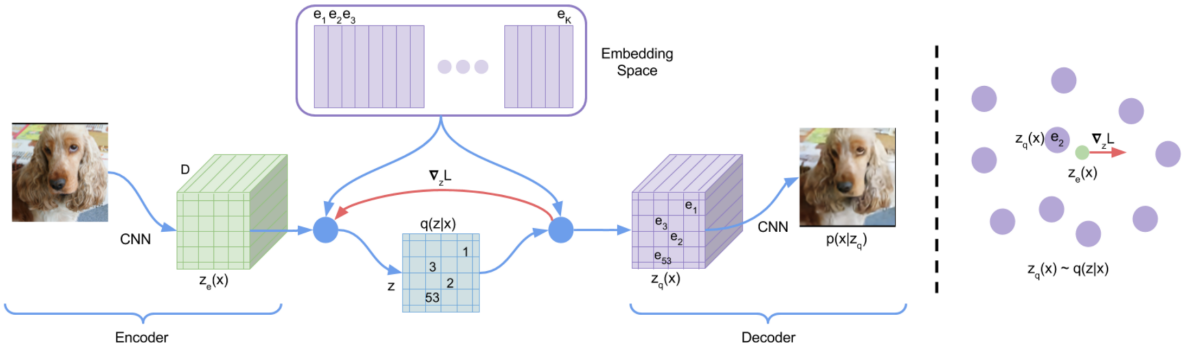


Figure 2: VQ-VAE in the original paper

In brief, the CNN encoder produces a (b, h, w, d) vector z_e, and it becomes z_q after vector quantization layer. In z_q, each d-dimensional vector has a correspondence in the codebook, we can replace each vector by their index in the codebook. As a result, we can generate new samples by generating a smaller image in the latent space, i.e., after learning a VQ-VAE model, we can learn the distribution of $q(z \mid x)$, and sample z from it and then use decoder to get $\hat{x}$. There are many ways to model $q(z \mid x)$. The original paper uses PixelCNN. In my implementation, I uses Transformer to model the prior $p(z)$.

2.5 Autoregressive Models

Autoregressive models can be used for learning a distribution. In this project, the prior distribution of VQ-VAE is a categorical distribution, where each x_i has K(the length of the codebook) choices. A multivariate distribution $p(x)$ can be modeled as follows in an autoregressive manner:

$$p(x) = p\left(x_1\right) p\left(x_2 \mid x_1\right) \ldots p\left(x_n \mid x_1, x_2, \ldots, x_{n-1}\right) \tag{5}$$

After training a VQ-VAE model, we can get all the latent representations of the original data by passing them to the encoder. Each image can be represented by a smaller image of shape (h, w) in the latent space. Now we flatten the latent representations to get a sequence of indices of shape $(hw, 1)$. I uses the encoder part of Transformer to learn the prior distribution.

2.6 Transformer to Model the Prior

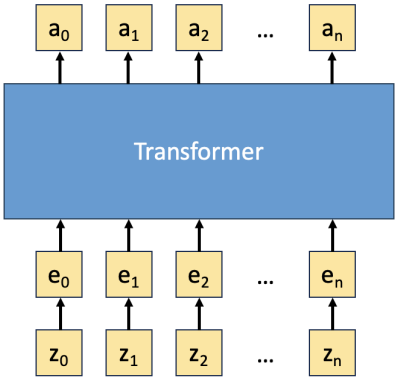


Figure 3: Transformer Model

The training data for the Transformer are latent representations z. z is of dimension $(hw, 1)$, which is a sequence of indices. Before training, each sequence is prepended by a $<sos>$ token. The input is first passed through an embedding layer, and then their embeddings are passed to the Transformer to get the output a. Each element a_i is the logits predicting the next token. These logits have a shape of $(K, 1)$, where K is the length of the codebook.

2.6.1 Teacher Forcing Technique

During training, rather than feeding the model's own previous predictions as input for subsequent time steps, the actual ground-truth tokens from the training data are fed in. In this case, the Transformer model always uses the true data for training, which helps to speed up convergence. The autoregressive Transformer, by design, has a future mask, preventing the data from seeing future data. As a result, during inference, softmax(a_i) is the distribution of the token z_{i+1}.

2.6.2 Loss of the Transformer

Predicting the next token in the sequence is a classification problem over the K possible codebook entries. Therefore, the loss function used to train the Transformer in this context is the cross-entropy loss:

$$L_{\text{CE}} = -\sum_{i=1}^{K} y_i \log(p_i) \tag{6}$$

During inference, the $<sos>$ is passed to the Transformer to get a_0, which is the distribution of z_1. After sampling from a_0 to get z_1, z_0 and z_1 is used to get a_1, and so on, until all the sequences are generated.

3 Implementation and Results

3.1 VQ-VAE Architecture

I uses a modified ResNet architecture for both encoder and decoder. A ResBlock is demonstrated blow:

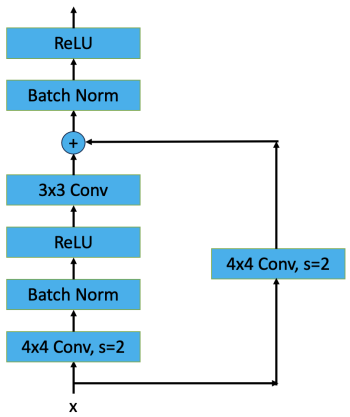


Figure 4: Residual Block

3.2 Transformer Architecture

For modeling the prior distribution of VQ-VAE's latent space, the Transformer architecture was employed. I uses the same architecture as the original transformer paper. An embedding layer is utilized to convert these indices into embedding vectors, and learnable positional encodings are added to the input embeddings. The final output of the tranformer is followed by a linear layer, producing a probability distribution over the K codebook entries for each position in the sequence. Future masking which zeros out all future scores is applied during self-attention.

3.3 Sampling Algorithm

Given the trained Transformer, the sampling algorithm works as follows:

1. The forward pass of the Transformer model is called to produce logits for the next token, considering all previously generated tokens.

2. These logits given by the Transformer are divided by the temperature parameter, and the scaled logits are processed by softmax to produce the probability for the next token. The temperature is a hyperparameter that controls the randomness during sampling. A higher temperature scales down all logits, leading to more randomness , whereas a lower temperature makes the sampling process more deterministic.

3. The top k tokens with the highest probabilities are selected, where K is a hyperparameter.

4. From the top k tokens, one token is randomly sampled using the multinomial distribution. This token is then appended to the current sequence.

In my implementation, I uses $temperature = 2$, and $K = 8$.

3.4 Reconstruction Results

The following results are downscaled by a factor of 4, with codebook length 128 and latent dimension 8. The first row is the original image and the second row is reconstruction. In terms of the loss, I uses $\alpha = 2$ and $\beta = 0.1$. Each model is trained by 12 epochs using learning rate 0.003.

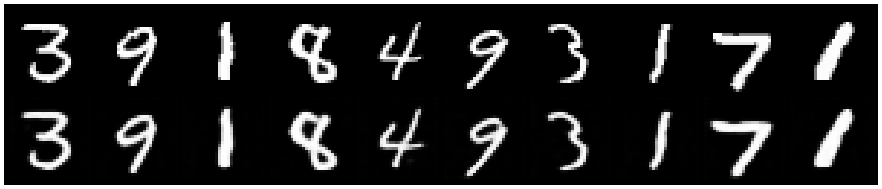


Figure 5: Reconstruction using MNIST

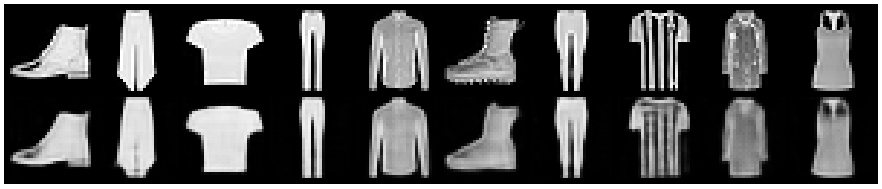


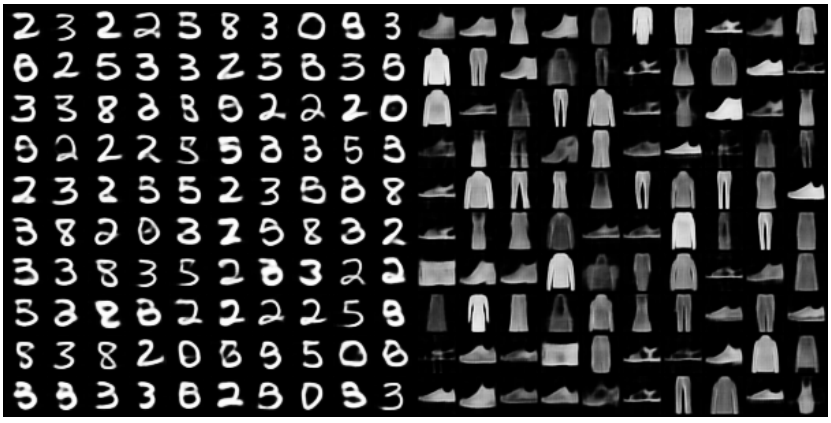
Figure 6: Reconstruction using Fashion-MNIST

Figure 7: Reconstruction using CelebA

3.5 Generation Results

The generation is performed as shown in Sampling Algorithm part. Each image is initialized by a $<sos>$ token, and the sequence of latents is generated using Transformer. The performance is bad due to insufficient training data for the Transformer. However, in the following,

conditional generation solves the problem. The transformer model has latent dimension 64 and 3 transformer layers, each layer has 5 heads. I uses dropout probability 0.1 and learning rate 0.0015. Each model is trained for 10 epochs.



(a) Generation on MNIST

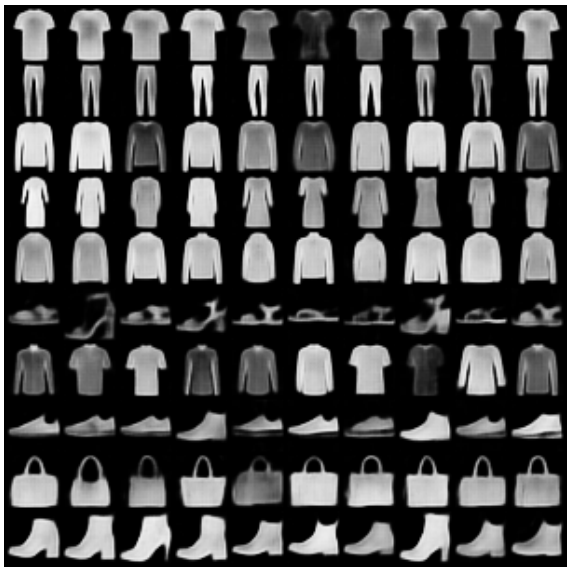
(b) Generation on Fashion-MNIST

(c) Generation on CelebA

3.6 Conditional Generation

Since generating all samples using the same <*sos*> token works bad, I uses different <*sos*> token for different classes. In this case, the generation uses additional information of the image for training, and thus achieves a significantly better result.

(a) Generation on MNIST



(b) Generation on Fashion-MNIST

4 Comparative Analysis

1. The codebook length is 128 optimally. Larger codebooks like 256 will only slightly improve the performance. Latent dimension of VQ-VAE plays an important role in reconstruction, and I uses latent dimension 8 in my implementation. Bigger dimensions like 16 tends to make the model use less diverse codes in the codebook, thus focusing on updating only some of the vectors.

2. The sequence length greatly impact the Transformer model performance. The ideal sequence length for transformer is $\frac{hw}{8}$. Longer sequence length will make the transformer harder to capture the true distribution, while shorter sequences will lose too much information of the original image.

5 Conclusion

In this project, I learned the codebook idea of generative models. VQ-VAE essentially generates new samples by generating a sequence in the latent space. This idea is further explored in Latent Diffusion Models. In addition, I learned the stop gradient operation, which allows a model to update specific parameters. Furthermore, I gained a better understanding of autoregressive Transformer model, especially teacher forcing technique and start of sequence token. Finally, prepending the sequence by additional information of the image, like label or text description, can significantly improve the model performance.