From PCA to VQ-VAE

[paper: vanilla VQ-VAE (https://arxiv.org/pdf/1711.00937.pdf)](https://arxiv.org/pdf/1711.00937.pdf)

[paper: VQ-VAE-2 (https://arxiv.org/pdf/1906.00446.pdf)](https://arxiv.org/pdf/1906.00446.pdf)

The common element in the design of any Autoencoder method is

- to create a latent representation $\mathbf{z}$ of input $\mathbf{x}$
- such that $\mathbf{z}$ can be (approximately) inverted to reconstruct $\mathbf{x}$.

Principal Components Analysis is a type of Autoencoder that produces a latent representation $\mathbf{z}$ of $\mathbf{x}$

- $\mathbf{x}$ is a vector of length n: $\mathbf{x} \in \mathbb{R}^n$
- $\mathbf{z}$ is a vector of length $n' \leq n$: $\mathbf{z} \in \mathbb{R}^{n'}$

Usually $n' << n$: achieving *dimensionality reduction*

This is accomplished by decomposing $\mathbf{x}$ into a weighted product of n *Principal Components*

- $\mathbf{V} \in \mathbb{R}^{n \times n}$
$$\mathbf{x} = \mathbf{z}' \mathbf{V}^T$$
  - where $\mathbf{z}' \in \mathbb{R}^n$
  - rows of $\mathbf{V}^T$ are the components

So $\mathbf{x}$ can be decomposed into the weighted sum (with $\mathbf{z}'$ specifying the weights)

- of n component vectors
- each of length n

Since $\mathbf{z}' \in \mathbb{R}^n$: there is **no** dimensionality reduction just yet.

One can view $\mathbf{V}^T$ as a kind of *code book*

- any $\mathbf{x}$ can be represented (as a linear combination) of the *codes* (components) in V^T

$$\mathbf{x} = \mathbf{z}'\mathbf{V}^T$$

$\mathbf{z}'$ is like a translation of $\mathbf{x}$, using $\mathbf{V}$ as the vocabulary.

- weights in the codebook
- rather than weights in the standard basis space $I \in \mathbb{R}^{n \times n} = \text{diagonal}(n)$

$$\mathbf{x} = \mathbf{x}I$$

Dimensionality reduction is achieved by defining $\mathbf{z}$ as a length n' prefix of $\mathbf{z}$

- $\mathbf{z} = \mathbf{z}'_{1:n'}$
- $\mathbf{z} \in \mathbb{R}^{n'}$

Similarly, we needed only n' components from $\mathbf{V}$

- $\mathbb{V}^T = \mathbf{V}^T_{1:n'}$
- $\mathbb{V}^T \in \mathbb{R}^{n' \times n}$

We can construct an *approximation* $\hat{\mathbf{x}}$ of $\mathbf{x}$ using *reduced dimension* $\mathbf{z}'$ and $\mathbb{V}$

$$\hat{\mathbf{x}} = \mathbf{z}\mathbb{V}^T$$

The Autoencoder (and variants such as VAE) produces $\mathbf{z^{(i)}}$, the latent representation of $\mathbf{x^{(i)}}$

- directly
- independent of any other training example $\mathbf{x}^{(i')}$ for $i \neq i'$

Our goal in using AE's is in generating synthetic data

- the dimensionality reduction achieved thus far was a necessity, not a goal

Vector Quantized Autoencoder

A *Vector Quantized VAE* is a VAE with similarities to PCA. It creates $\mathbf{z}$

- which is an **integer**
- that is the index of a row
- in a codebook with K rows

That is: the input is represented by one of K possible vectors.

The goal is **not necessarily** dimensionality reduction.

Rather, there are some advantages to a **discrete** representation of a continuously-valued vector.

- Each vector
- Drawn from the infinite space of continuously-valued vectors of length n
- Can be approximated by one of K possible vectors of length n

Thus, a sequence of T continuously valued vectors

- can be represented as a sequence of T integers
- over a "vocabulary" defined by the code book

This is analogous to text

- sequence of works
- represented as a sequence of integer indices in a vocabulary of tokens

Once we put complex objects

- like images
- timeseries
- speech

into a representation similar to text

- we can have *mixed type* sequences
    - e.g., words, images

In a subsequent module we will take advantage of mixed type sequences

- to produce an image
- from a text *description* of the image
- using the "predict the next" element of a sequence technique of Large Language Models

DALL-E: Text to Image
Text input: "An illustration of a baby daikon radish in a tutu walking a dog"
Image output:

Details

Here is diagram of a VQ-VAE

- that creates a latent representation of a 3-dimensional image $(w \times h \times 3)$
- a a 2-dimensional matrix of integers

There is a bit of notation: referring to the diagram should facilitate understanding the notation.

VQ-VAE

In general, we assume the input has $\#S$ *spatial* dimensions

- where each location in the spatial dimension is a vector of length n
- input shape $(n_1 \times n_2 \ldots \times n_{\#S} \times n)$

We will explain this diagram in steps.

First, we summarize the notation in a single spot for easy subsequent reference.

Notation summary

term	shape	meaning	
S	$(n_1 \times n_2 \ldots \times n_{\#S})$	Spatial dimensions of $\#S$-dimensional input	
$\mathbf{x}$	$\mathbb{R}^{S \times n}$	Input	
D		length of latent vectors (Encoder output, Quantized Encoder output, Codebook entry)	
$\mathcal{E}$		Encoder function	
$\mathrm{z}_e(\mathbf{x})$	$\mathbb{R}^{S \times D}$	Encoder output over each location of spatial dimension	
		$\mathbf{z}_e(\mathbf{x}) = \mathcal{E}(\mathbf{x})$	
$\mathbf{z}_e(\mathbf{x})$	$\mathbb{R}^{D}$	Encoder output at a **single** representative spatial location	
		$\mathbf{z}_e(\mathbf{x}) = \mathcal{E}(\mathbf{x})$	
K		number of codes	
$\mathbf{E}$	$\mathbb{R}^{K \times D}$	Codebook/Embedding	
		K codes, each of length D	
$e \in \mathbf{E}$	$\mathbb{R}^{D}$	code/embedding	
z	$\{1, \ldots, K\}^{S \times D}$	latent representation over all spatial dimensions	
$\mathbf{z}$	$\{1, \ldots, K\}$	Latent representation at a **single** representative spatial location	
		one integer per spatial location	
$\qr{\z	\x }$	integer $\in [1 \ldots K]$	Index k of $e_k \in \mathbf{E}$ that is closest to $\mathbf{z}_e(\mathbf{x})$
		$k = \underset{j \in [1,K]}{\operatorname{argmin}} \lvert \mathbf{z}_e(\mathbf{x}) - \mathbf{e}_j \rvert_2$	
		actually: encoded as a OHE vector of length K	
$\mathbf{z}_q(\mathbf{x})$	$\mathbb{R}^{D}$	Quantized $\mathbf{z}_e(\mathbf{x})$	
		$\mathbf{z}_q(\mathbf{x}) = e_k$ where $k = \qr{\z	\x }$
		i.e, the element of codebook that is closest to $\mathbf{z}_e(\mathbf{x})$	
		$\mathbf{z}_q(\mathbf{x}) \approx \mathbf{z}_e(\mathbf{x})$	
$\tilde{\mathbf{x}}$	n	Output: reconstructed $\mathbf{x}$	

term	shape	meaning	
		$\pr{ \x	\z_q(\x) }$
$\mathcal{D}$	$\mathbb{R}^{n'} \to \mathbb{R}^n$	Decoder	
		input: element of codebook $\mathbf{E}$	
		$\tilde{\mathbf{x}} = \mathcal{D}(\mathbf{z}_q(\mathbf{x}))$	

Quanitization

Let S denote the spatial dimensions, e.g. $S = (n_1 \times n_2)$ for 2D

So input $\mathbf{x} \in \mathbb{R}^{S \times n}$

- n features over S spatial locations

The input $\mathbf{x}$ is transformed in a sequence of steps

- Encoder output (continuous value)
- Latent representation (discrete value)
    - Quantized (continuous value)

In the first step, the *Encoder* maps input $\mathbf{x}$

- to Encoder output $\mathbf{z}_e(\mathbf{x})$
- an alternate representation of D features over S' spatial locations

(For simplicity, we will assume $S' = S$)

Notational simplification

In the sequel, we will apply the same transformation **to each element** of the spatial dimension

Rather than explicitly iterating over each location we write

$$\mathbf{z}_e(\mathbf{x}) \in \mathbb{R}^D$$

to denote a representative element of $\mathrm{z}_e(\mathbf{x})$ at a single location $s = (i_1, \ldots, i_{\#S})$

$$\mathbf{z}_e(\mathbf{x}) = \mathrm{z}_e(\mathbf{x})_s$$

We will continue the transformation at the single representative location

- and implicitly iterate over all locations $s \in S$

The continuous (length D) Encoder output vector $\mathbf{z}_e(\mathbf{x})$

- is mapped to a *latent representation* $q(\mathbf{z}|\mathbf{x})$
- which is a **discrete** value (integer)

$$k = q(\mathbf{z}|\mathbf{x}) \in \{1, \ldots, K\}$$

where k is the *index* of a row $\mathbf{e}_k$ in codebook $\mathbf{E}$

$$\mathbf{e}_k = \mathbf{E}_k \in \mathbb{R}^D$$

k is chosen such that $\mathbf{e}_k$ is the row in $\mathbf{E}$ **closest to** $\mathbf{z}_e(\mathbf{x})$

$$
\begin{aligned}
k &= q(\mathbf{z}|\mathbf{x}) \\
&= \operatorname*{argmin}_{j \in \{1,...,K\}} \|\mathbf{z}_e(\mathbf{x}) - \mathbf{e}_j\|_2
\end{aligned}
$$

We denote the codebook vector

- closest to representative encoder output $\mathbf{z}_e(\mathbf{x})$
- as $\mathbf{z}_q(\mathbf{x})$

$$\mathbf{z}_q(\mathbf{x}) \in \{1, \ldots, K\} = e_k$$

The Decoder tries to invert the codebook entry $\mathbf{e}_k = \mathbf{z}_q(\mathbf{x})$ so that

$$\begin{aligned} \tilde{\mathbf{x}} &= \mathcal{D}(\mathbf{z}_q(\mathbf{x})) \\ &\approx \mathbf{x} \end{aligned}$$

Discussion

Why do we need the CNN Encoder ?

The input $\mathbf{x}$ is first transformed into an *alternate representation*

- the **number** and shape of the spatial dimensions are preserved (not necessary)
- but the number of features is transformed from n raw features to $D \geq n$ synthetic features
    - typical behavior for, e.g., an image classifier

The part of the VQ-VAE after the initial CNN

- reduces the size of the **feature dimension** from D to 1
- this is the primary source of dimensionality reduction
    - the raw n of image input is usually only $n = 3$ channels

It may be useful for the CNN to *down-sample* spatial dimension S to a smaller S'

For example

- 3 layers of stride 2 CNN layers
- will reduce a 2D image of spatial dimension $(n_1 \times n_2)$
- to spatial dimension $(\frac{n_1}{8} \times \frac{n_2}{8})$

This replaces each $(8 \times 8 \times n)$ *patch* of raw input

- into a single vector of length D
- that summarizes the (8×8) the patch

One possible role (not strictly necessary) for the CNN Encoder

- is to replace a large spatial dimensions
- by smaller "summaries" of local neighborhoods (patches)

Why quantize ?

Quantization

- converts the continuous $\mathbf{z}_e(\mathbf{x})$
- into discrete $q(\mathbf{z}|\mathbf{x})$
- representing the approximation $\mathbf{z}_q(\mathbf{x}) \approx \mathbf{z}_e(\mathbf{x})$

The Decoder inverts the approximation.

Why bother when the Quantization/De-Quantization is Lossy ?

One motivation comes from observing what happens if we *quantize and flatten* the $\#S'$-dimensional spatial locations to a one-dimensional vector.

Quantizing replaces each patch with a single integer index.

- the integer is the index of an *image token* within a list of K possible toke

By flattening the quantized higher dimensional matrix of patches, we convert the input

- into a sequence of image tokens
- over a "vocabulary" defined by the codebook $\mathbf{E}$.

This yields an image representation

- similar to the representation of text

Thus, we open the possibility of processing sequences of mixed text and image tokens.

Quantized image embeddings mixed with Text: preview of DALL-E

The Large Language Model operates on a sequence of text tokens

- where the text tokens are fragments of words
- when run autoregressively
    - concatenating each output to the initial input sequence
    - the LLM shows an ability to produce a "sensible" continuation of an initial "thought"

Suppose we train a LLM on input sequences

- that start with a sequence of *text* tokens describing an image
- followed by a separator `[SEP]` token
- followed by a sequence of of quantized image tokens

```
<text token> <text token> ... <text token> [SEP] <image token> <image token> ...
```

What continuation will our trained LLM produce given prompt

```
<text token> <text token> ... <text token> [SEP]
```

Hopefully:

- a sequence of *image tokens*
- that can be reconstructed
- into an image matching the description given by the text tokens !

That is the key idea behind a Text to Image model called DALL-E that we will discuss in a later module.

There remains an important technical detail

- the embedding space of text and image are distinct
- they need to be merged into a common embedding space

We will visit these issues in the module on CLIP.

Loss function

The Loss function for the VQ-VAE entails several parts

- Reconstruction loss
  - enforcing constraint that reconstructed image is similar to input
$$\tilde{\mathbf{x}} \approx \mathbf{x}$$
- Vector Quantization (VQ) Loss:
  - enforcing similarity of quantized encoder output and actual encoder output
$$\mathbf{z}_q(\mathbf{x}) \approx \mathbf{z}_e(\mathbf{x})$$
- Commitment Loss
  - a constraint that prevents the Quantization of $\mathbf{z}_e(\mathbf{x})$ from alternating rapidly between code book entries

The Reconstruction Loss term is our familiar: Maximize Likelihood

- written to minimize the negative of the log likelihood, as usual

$$p(\mathbf{x}|\mathbf{z}_q(\mathbf{x}))$$

The Vector Quantization Loss is more complex

$$\|\text{sg}(\mathbf{z}_e(\mathbf{x})) - \mathbf{z}_q(\mathbf{x})\|$$

The `sg` operator is the *Stop Gradient* operator.

We will explain this in more detail below and give reference to a `VectorQuantizer` layer type.

Commitment Loss:

$$\|\mathbf{z}_e(\mathbf{x}) - \text{sg}(\mathbf{z}_q(\mathbf{x}))\|$$

The Commitment and Vector Quantization losses are similar except for the placement of the Stop Gradient.

The Stop Gradient in the Commitment Loss prevents a change in the Embeddings from affecting the Encoder weights (and thus, $z_e(\mathbf{x})$).

The Stop Gradient of the Vector Quantization Loss prevents a change in the Encoder weights (and thus, $z_e(\mathbf{x})$) from affecting the embeddings.

This prevents a feedback loop

- Encoder updating $\mathbf{z}_e(\mathbf{x})$ reduces Reconstruction Loss *assuming* embeddings remain constant
- But changing Encoder output results in embeddings being updated
- So embeddings *do not* remain constant
- The net effect may not be a reduction in Reconstruction Loss

Which parts of the architecture are responsible for each Loss component

- The Decoder is responsible for the Reconstruction Loss (through the term $\tilde{\mathbf{x}}$
- The Encoder (through the term $\mathbf{z}_e(\mathbf{x})$ is responsible for
    - The Reconstruction Loss
    - The Commitment Loss
- The embeddings $\mathbb{E}$ are updated via the Vector Quantizer Loss
    - Does not affect the Encoder or Decoder weights

Straight Through Estimation (discussed below) causes the gradient from Reconstruction Loss to "by-pass" $\mathbb{E}$

- effectively, for the purpose of gradient/weight update:
  $$\mathbf{z}_q(\mathbf{x}) = \mathbf{z}_e(\mathbf{x})$$
  If there were no Vector Quantizer Loss, the Reconstruction Loss would not lead to Embeddings $\mathbb{E}$ being updated

Loss function

$$
\begin{aligned}
\mathcal{L}(\mathbf{x}, \mathcal{D}(\mathbf{e})) \quad = \quad & ||\mathbf{x} - \mathcal{D}(\mathbf{e})||_2^2 && \text{Reconstruction Loss} \\
& + ||\text{sg}[\mathcal{E}(x)] - \mathbf{e}||_2^2 && \text{VQ loss, codebook loss: train codebook} \\
& + \beta ||\text{sg}[\mathbf{e}] - \mathcal{E}(\mathbf{x})||_2^2 && \text{Commitment Loss: force } E(\mathbf{x}) \text{ to be clo} \\
& \text{where } \mathbf{e} = \mathbf{z}_q(\mathbf{x})
\end{aligned}
$$

Need the stop gradient operator sg to control the mutual dependence

- of $\mathcal{E}(\mathbf{x})$ and $\mathbf{e}$

Straight-through Estimation and the Stop Gradient operator sg

Gradient Descent is the algorithm that we use to find values for a model's weights that minimize the model's Loss Function.

Recall: it works by recursively (backwards from head to input) layer by layer

- updating the partial of the Loss with respect to the layer's inputs
    - respectively: the partial of the Loss with respect to each operation

But there is a problem in the Quantization operation

- argmin is not differentiable !
- it is not continuous at the point that its value switches between k and $k' \neq k$
    - For example,
        - Non-unique arguments: when $\mathbf{e}_k = \mathbf{e}_{k'}$ for $k \neq k'$
        - small changes in the arguments cause a change from k to k'

The non-differentiability of certain operators led to the creation of the Stop Gradient operator `sg`

$$
\begin{aligned}
\text{sg}(\mathbf{x}) &= \mathbf{x} \\
\frac{\partial \, \text{sg}(\mathbf{x})}{\partial \mathbf{y}} &= 0 \quad \text{for all } \mathbf{y}
\end{aligned}
$$

It is the identity operation on the Forward pass.

But on the Backward pass (Gradient Descent) it treats its argument as if it were a constant.

Straight through estimation

The Stop Gradient operator can be used in conjunction with *Straight Through Estimation*.

Let's recall the definition of the Loss Gradient

Let

$$\mathcal{L}'_{(l)} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}}$$

denote the derivative of $\mathcal{L}$ with respect to the output of layer l, i.e., $\mathbf{y}_{(l)}$.

This is called the **loss gradient**.

- although we state this with respect to a "layer-ed" architecture this is for notational convenience only
- the same if true if we replace "layer" with "operator" whose input is denoted $\mathbf{y}_{(l-1)}$ and output denoted $\mathbf{y}_{(l)}$

Back propagation inductively updates the Loss Gradient from the output of layer l to its inputs (e.g., prior layer's output $\mathbf{y}_{(l-1)}$)

- Given $\mathcal{L}'_{(l)}$
- Compute $\mathcal{L}'_{(l-1)}$
- Using the chain rule

$$
\begin{aligned}
\mathcal{L}'_{(l-1)} &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l-1)}} \\
&= \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} \\
&= \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}
\end{aligned}
$$

The loss gradient "flows backward", from $\mathbf{y}_{(L+1)}$ to $\mathbf{y}_{(1)}$.

This is referred to as the *backward pass.*

That is:

- the upstream Loss Gradient $\mathcal{L}'_{(l)}$
- is modulated by the local gradient $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$
- where the "layer" is the operation transforming input $\mathbf{y}_{(l-1)}$ to output $\mathbf{y}_{(l)}$

What happens when the operation implemented by the function that takes $\mathbf{y}_{(l-1)}$ to $\mathbf{y}_{(l)}$ is either

- non-differentiable
- or has zero derivative almost everywhere
- non-deterministic (e.g., `tf.argmin` when two inputs are identical)

This is the case with any type of quantization operation (uses `tf.argmin`) resulting in

- $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} = 0$
- and $\mathcal{L}'_{(l-1)} = \mathcal{L}_{(l)} * 0 = 0$

So the quantization operation disconnects the gradient flow from the Decoder backwards to the Encoder.

Hence, the notion of a Straight Through Estimator is developed

- identity operation on forward pass
- with local derivation $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$ **defined** to be equal to 1

We see this in the [Colab (https://keras.io/examples/generative/vq_vae/)](https://keras.io/examples/generative/vq_vae/) implementation of Vector Quantization (the `VectorQuantizer` layer)

```
class VectorQuantizer(layers.Layer):
...
    def call(self, x):
...
        # Straight-through estimator.
        quantized = x + tf.stop_gradient(quantized - x)
```

Code similar to the [`VectorQuantizer` of the paper's authors (https://github.com/deepmind/sonnet/blob/v1/sonnet/python/modules/nets/vqvae.py)](https://github.com/deepmind/sonnet/blob/v1/sonnet/python/modules/nets/vqvae.py)

The last line is a ["straight through estimator" (https://www.hassanaskary.com/python/pytorch/deep%20learning/2020/09/19/intuitive-explanation-of-straight-through-estimators.html)](https://www.hassanaskary.com/python/pytorch/deep%20learning/2020/09/19/intuitive-explanation-of-straight-through-estimators.html)

- On the forward pass: identity assignment `quantized = quantized`
- On the backward pass, the Loss gradient is passed through unchanged from upstream
    - i.e, from output (the tensor `quantizer`) to the *layer input* (denoted by formal parameters x, don't confuse it with the VQ-VAE's input)
    - this is because the `tf.stop_gradient` causes the enclosed expression to be treated as a constant
        - hence will contribute 0 to the loss gradient back propagation

So

- `tf.stop_gradient` **kills** the gradient along one path
- the Straight Through Estimator passes it through unchanged

In the VQ-VAE, straight through estimation

- passes the gradient from the Decoder input back to the Encoder outputs
- ignoring the quantization
- allowing the Encoder to adapt to reduce Reconstruction Loss

Learning the distribution of latents

For a VAE, we assume a functional form for the prior distribution of latents $q(\mathbf{z})$

- usually Normal

The authors wish to do away with an assumption of the prior distribution $q(\mathbf{z})$.

Retaining spatial/temporal dimensions in $\mathbf{z}_q(\mathbf{x})$ is key to achieving this goal.b

The authors *flatten* the spatial/temporal dimensions

- Assume (for example) a two dimensional $\mathbf{Z}$ with h rows and w columns
- $\mathbf{Z}_j^{(\mathbf{i})}$ denotes the vector of length D at row i, column j of $\mathbf{Z}$
- Flatten $\mathbf{Z}$ into a *sequence* $[\mathbf{z}_1, \mathbf{z}_2, \ldots]$
    - where $\mathbf{z}_k$ is the quantization of $\mathbf{Z}_c^{(r)}$
        - for $r = \text{int}(\frac{k}{w}), c = (k \mod w)$

The authors then learn an autoregressive model for sequences

$$p(\mathbf{z}_{k+1}|\mathbf{z}_1, \ldots, \mathbf{z}_k)$$

by using some Autoregressive model (e.g, PixelCNN) to predict $\mathbf{z}_k$ from its predecessors.

The Autoressive model

- learns $\mathbf{z}_k$. Doesn't assume what type of distribution it comes from
- can be sampled
    - seed the model with $\mathbf{z}_1$, generate the rest of the sequence
    - append predicted $\mathbf{z}_k$ to sequence upon which $\mathbf{z}_{k+1}$ is conditioned
- Is trained *subsequent* to learning the Embeddings
    - future research: learn them jointly

Thus, adding the Autoregressive step facilitates generating new sample sequences from which to generate synthetic examples.

In [2]:

```
print("Done")
```

```
Done
```