Introduction

We now present an Autoencoder with a twist

- the latent representation produced for an input
- is limited to be one member of a *finite list* of vectors
- enabling us to describe the latent by the *integer index* in the list

Why is an integer encoding of an input interesting ?

- It is analogous to the way we treat words (tokens) in Natural Language Processing
    - an index into a finite Vocabulary of words
- This opens the possibility of dealing with sequences that are a *mixture* of text and other data types (e.g., images)

Rather than pre-specifying the finite list, we will *learn* the list by training a Neural Network.

In a subsequent module, we will use a similar technique for the task of Text to Image

- given the description of an image in words
- create an image matching the description

But there is a significant problem with a Neural Network that learns discrete values

- the network may need to make a "hard" (as compared to "soft") choice
    - a true `if` statement ("hard") versus a "soft" conditional (sigmoid)
    - a Python `dict` ("hard" lookup) versus a "soft" lookup (Context Sensitive Memory)
- "hard" means derivatives are not continuous
- Gradient Descent won't work

We will introduce a new Deep Learning operator (*Stop Gradient*) to deal with "hard" operators.

References

- paper: vanilla VQ-VAE (https://arxiv.org/pdf/1711.00937.pdf)
- paper: VQ-VAE-2 (https://arxiv.org/pdf/1906.00446.pdf)

From PCA to VQ-VAE

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 $i \ne i'$

One of our goals in using AE's is in generating synthetic data

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

Our goal in introducing the Vector Quantized Autoencoder is not synthesizing data

- it is to create a representation of complex data types that are similar to sequences (e.g., image, audio) t
- so that they can be mixed with other sequence data types (e.g., text)

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

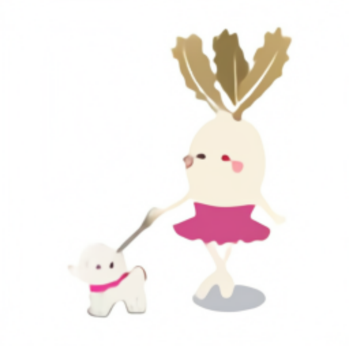
into a representation similar to text

- 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 $(w \times h \times 3)$
- as 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	$(n_1 \times n_2 \dots$ $\times n_{\#S})$	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	$\{1, \dots, K\}^{S \times D}$	latent representation over all spatial dimensions	
$\mathbf{z}$	$\{1, \dots, K\}$	Latent representation at a **single** representative spatial location	
		one integer per spatial location	
$\qr{ \z	\x }$	integer $\in [1 \dots K]$	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 $\mathbf{x}$	
		$\pr{ \x	\z_q(\x) }$
$\mathcal{D}$	$\mathbb{R}^{n'} \to \mathbb{R}^{n}$	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 $\mathrm{z}_e(\mathbf{x})$
- 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 $\mathbf{E}$
$$\mathbf{e}_k = \mathbf{E}_k \in \mathbb{R}^D$$

The codebook is also called an *Embedding* table.

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

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

We denote the codebook vector

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

$$\mathbf{z}_q(\mathbf{x}) = \mathbf{e}_k \text{ where } k = q(\mathbf{z}|\mathbf{x})$$

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 tokens

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

Reconstruction Loss

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 Decoder is responsible for the Reconstruction Loss (through the term $\tilde{\mathbf{x}}$)

- Straight Through Estimation (which we shall subsequently discuss)
- means the gradients from Decoder input $\mathbf{z}_q(\mathbf{x})$ flows directly to Encoder outpput $\mathbf{z}_e(\mathbf{x})$
- *without* affecting the Embeddings
    - i.e., the embedding $e_k \in \mathbb{E}$ (where $k = q(\mathbf{z}|\mathbf{x})$) is **not** affected by the Reconstruction Loss

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

- effectively, for the purpose of gradient/weight update:
$$\mathbf{z}_q(\mathbf{x}) = \mathbf{z}_e(\mathbf{x})$$

The role of the other Loss terms (e.g., Vector Quantizer Loss) are to ensure that the Embeddings $\mathbb{E}$ get updated.

Vector Quantization Loss

The Vector Quantization Loss and Commitment Loss are similar.

Vector Quantization Loss:

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

where `sg` is the *Stop Gradient* Operator (details to follow).

The purpose of the Vector Quantization Loss is to "learn" the Embedding (codebook) $\mathbf{E}$

- by moving e_k closer to Encoder output $\mathbf{z}_e(\mathbf{x})$

Commitment Loss

Commitment Loss:

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

It is similar to the Vector Quantization loss except for the placement of the Stop Gradient operator.

It's purpose is ensure that the embeddings $\mathbb{E}$ converge.

Total Loss

Loss function

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

Need the stop gradient operator sg to control the mutual dependence

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

Stop Gradient operator

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

On the Forward Pass, it acts as an Identity operator

$$\mathrm{sg}(\mathbf{x}) = \mathbf{x}$$

But on the Backward Pass of Backpropagation: *it stops the gradient* from flowing backwards

$$\frac{\partial\, \mathrm{sg}(\mathbf{x})}{\partial \mathbf{y}} = 0 \text{ for all } \mathbf{y}$$

Why is this operator necessary for the VQ-VAE ?

The reason it appears in the Vector Quantization Loss and Commitment Loss is to prevent 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

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.

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

This prevents a feedback loop.

Quantization is not differentiable

There is a subtle but important problem.

The Quantization operation

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

is **not differentiable** because of `argmin`

`argmin` is a problematic operation because

- it contains a "hard choice" so is not differentiable
    - output may change dis-continuously from index k to index $k' \neq k$
    - for small changes in the input
    - not continuous as the point of change
- it may also be non-deterministic
    - when minimum value occurs at more than one index
    - when $\mathbf{e}_k = \mathbf{e}_{k'}$ for $k \neq k'$

There is a work-around

- implement a `VectorQuantizer` layer
- using a [Straight Through Estimator (Straight_Through_Estimator.ipynb)](Straight_Through_Estimator.ipynb)
    - see that module for details of the technique

We see this in the Colab (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)

Learning the distribution of latents

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

- a *convenient* choice is Normal. See our VAE module (VAE_Generative.ipynb#Choosing-$\qr{\z}$)

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.

The authors *flatten*

- the spatial/temporal dimensions $(n_1 \times n_2 \ldots \times n_{\#S})$ of $\mathbf{Z}$
- into a sequence

$$\mathbf{z}_{(1)}, \mathbf{z}_{(2)}, \ldots, \mathbf{z}_{(n_1 * n_2 \ldots * n_{\#S})}$$

For example: for two spatial dimensions $(h \times w)$

- $\mathbf{z}_{(k)}$
- is the quantization of $\mathbf{Z}_c^{(r)}$
- the element at row r column c
- 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 $\mathbf{z}_{(k+1)}$ from its predecessors in the sequence.

We are familiar with this Autoregressive model (in the case of NLP) as the Language Model objective.

Unlike the "convenient" common choice of Normality for the VAE

- the Autoregressive model does **not** assume the type of the distribution

The conditional distribution

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

of each element

- is learned
- is easy to sample
    - seed the model with $\mathbf{z}_{(1)}$, generate the rest of the sequence one element at a time

```
In [2]: print("Done")
```

```
Done
```