Introduction

TL;DR

- Text is represented as
    - a sequence of integers: an integer index into the list of tokens in the vocabulary
- We want to represent other types of data (e.g., images) as
    - a sequence of integers: an integer index into a "code book": finite list of vectors
- By making the "shape" (sequence) and "type" (integer) compatible between text and other data types
    - We facilitate mixing text and other data types
    - Will enable an implementation of Text to Image as a simple extension of the Language Modeling objective

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)

Vector Quantized Autoencoder

A *Vector Quantized VAE* is a VAE with similarities to PCA.

Given an input $\mathbf{x}$

- vector of continuous values

it creates another continuous vector

- that is an *approximation* of $\mathbf{x}$
- but is chosen from a finite collection of K continuous vectors

It outputs $\mathbf{z}$: the index within the codebook of the approximating vector

The integer index $\mathbf{z}$ can be thought of

- as a token (Categorical variable)
- from a finite Vocabulary of K tokens (the codebook)

That is $\mathbf{z}$

- is a *discrete* representation
- of a continuous vector

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 words
- 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

Details

A VQ-VAE can produce a sequence of integers that encodes many different types of data

- including data with "shape": *non-feature* dimensions
    - image
    - audio

Each "location" in the space of non-feature dimensions

- is a vector consisting only of features

So an image, for example,

- is a collection of feature vectors
- whose locations are arranged into a 2D grid
    - a location is a pair of row number/column number
- the feature vector at each location
    - has 3 features: Red, Green, Blue

We illustrate the VQ-VAE using Image examples.

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
    - each location
    - has an integer index
        - which references a vector of length n in the codebook (with K codes in the codebook)

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

VQ-VAE

The picture: in words

The **Encoder**

- the *input* image (dog) $\mathbf{x}$ of shape $(h \times w \times n)$
    - feature vectors of length $n = 3$ at locations defined in a 2D grid of shape $(h \times w)$
- is transformed by a CNN
- into a 3D alternate representation $\mathbf{z}_e(\mathbf{x})$
    - feature vectors of length D over a 2D grid of shape $S = (h' \times w')$
    - $(h' \times w')$ may differ from $(h \times w)$
        - strided convolution, downsampling

The result is called the *Encoder Output* $\mathbf{z}_e(\mathbf{x})$

- shape $(S \times D)$
- where each location is a vector of continuous values

The feature vector of length D at **each location** of the Encoder Output $\mathrm{z}_e(\mathbf{x})$

- is mapped to one of the possible Embedding Vectors
- defined by the *Codebook* $\mathbf{E}$
  - shape $(K \times D)$
  - a collection of K continuous embedding vectors of length D

This results in an *approximation* $\mathrm{z}_q(\mathbf{x})$ of the Encoder Output $\mathrm{z}_e(\mathbf{x})$

$$\mathrm{z}_q(\mathbf{x}) \approx \mathrm{z}_e(\mathbf{x})$$

- the actual vector at each location in Encoder Output $\mathrm{z}_e(\mathbf{x})$
- is replaced by some vector e_k in $\mathbf{E}$

This approximation is called the *quantized* equivalent of $\mathbf{x}$, denoted $\mathrm{z}_q(\mathbf{x})$

- shape $(S \times D)$
- where each location is a vector of continuous values
  - one of the K embedding vectors

Since each location in the quantized equivalent $\mathrm{z}_q(\mathbf{x})$ is from a finite set $\mathbf{e}$

- we can equivalently represent the information via a spatial arrangement of integers in $[1:K]$
- the index within $\mathbf{e}$ of the vector at the location

denoted by $\mathrm{q}(\mathbf{z}|\mathbf{x})$

- shape S
- where each location is an integer

So mapping the integers in $\mathrm{q}(\mathbf{z}|\mathbf{x})$ through the codebook $\mathbf{E}$ results in $\mathrm{z}_q(\mathbf{x})$.

The **Decoder**

- attempts to map the quantized equivalent $\mathrm{z}_q(\mathbf{x})$
- into the reconstructed
$$\mathrm{p}(\mathbf{x} \mid \ \mathrm{z}_q)$$
- that is close to original input $\mathbf{x}$
$$\mathrm{p}(\mathbf{x} \mid \ \mathrm{z}_q) \approx \mathbf{x}$$

The picture thus describes an Autoencoder

- where the "bottleneck"
- is caused by quantization rather than dimensionality reduction.

Notation

Notation warning

Our inputs $\mathbf{x}$ and their alternate representations

- $\mathrm{z}_e(\mathbf{x})$
- $\mathrm{z}_q(\mathbf{x})$
- $\mathrm{q}(\mathbf{z}|\mathbf{x})$
- $\mathrm{p}(\mathbf{x} \mid \mathrm{z}_q)$

all have non-feature dimensions of shape $S = (n_1 \times n_2 \ldots n_{\#S})$.

Rather than explicitly iterating over each location

- we use the unscripted version of the letter
- to refer to the value at a single location

That is, the single location equivalents are denoted

- $\mathbf{z}_e(\mathbf{x})$
- $\mathbf{z}_q(\mathbf{x})$
- $q(\mathbf{z}|\mathbf{x})$
- $p(\mathbf{x} \mid \mathbf{z}_q)$

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 $\mathbf{x}$	
		$\pr{ \x	\z_q(\x) }$

term	shape	meaning
$\mathcal{D}$	$\mathbb{R}^{n'} \to \mathbb{R}^n$	Decoder
		input: element of codebook $\mathbf{E}$

Quanitization

Given

- continuous vector $\mathbf{z}_e(\mathbf{x})$ at a single location
- we need to map it to one vector $\mathbf{e}_k$ in codebook $\mathbf{E}$
- creating the approximation vector $\mathbf{z}_q(\mathbf{x})$

The approximation vector $\mathbf{z}_q(\mathbf{x})$ can then be represented as an integer k

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

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** $\mathbf{z}_e(\mathbf{x})$

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

In our notation:

- $\mathbf{z}_q(\mathbf{x})$
- is the quantized version of $\mathbf{z}_e(\mathbf{x})$
- where
$$\mathbf{z}_q(\mathbf{x}) = \mathbf{e}_k$$
- such that
$$k = q(\mathbf{z}|\mathbf{x})$$

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 CNN Encoder takes

- input shape $(h \times w \times 3)$ for RGB images
- output shape $(h' \times w; \times D)$

When $h' = h, w' = w$

- the CNN is *creating* many synthetic features at each spatial location

Eventually, the continuous vector $\mathbf{z}_e(\mathbf{x})$ of length D for one location

- is mapped to an approximate vector of length D from the codebook

Even if we

- limit the range of values of the elements of $\mathbf{z}_e(\mathbf{x})$
- and scale/round them to integers

there are *many* possible vectors of length D

Perhaps we need to expand the feature dimension from 3 to $D \gg 3$

- because a location needs to
    - represent *semantic* information on the image
    - rather than color intensities as for the input
    - in order to concisely encode the many possibilities for vectors of length D

Recall the concept of the *receptive field* for one location in the output of a CNN

- the pixels in the input layer that affect the value at the location
- receptive field size grows with number of CNN layers
    - three layers of a CNN with filter size 3 at each layer is a (7×7) region of the input

So the continuous vector at each location represents *several* input pixels

- and thus is more likely to represent "semantic" concepts of a region
- rather than intensity of a single input location

Although our illustration preserved the non-feature dimensions

- it may be useful for the CNN to *down-sample* spatial dimension S to a smaller S'
- resulting in shorter sequences when we eliminate the non-feature dimensions by "flattening"

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

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 sequence of integers

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.

Loss function

We will show the Loss Function for the VQ-VAE.

In particular, we want to highlight

- some *non-standard* aspects: the need for a Stop Gradient operator

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 codebook entries

Stop Gradient operator

Some of the Loss terms will involve an operator that we have not yet seen:

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

The need for this operator stems from the Gradient Descent update process

- the partial derivative assumes "everything else" other than the denominator (variable being updated) remains constant
- if "everything else" *also* changes: the gradient update step may not reduce Loss

But Quantization can easily result in violation of the "everything else held constant" assumption

- a small change to the Encoder parameters
- which causes a small change in Encoder output $\mathbf{z}_e(\mathbf{x})$
- may cause a **large** change in the quantized output $\mathbf{z}_q(\mathbf{x})$
    - the index of the nearest vector in $\mathbb{E}$ to $\mathbf{z}_e(\mathbf{x})$ switches from k to $k' \neq k$

For similar reasons

- a small change in the Embeddings in the codebook
- can cause a large change in the quantized output $\mathbf{z}_q(\mathbf{x})$
- if the index switches from k to k'

So the discrete nature of quantization means

- **even if** we could differentiate the operation
- the fact that its outputs are *not continuous*
    - sudden jumps caused by change of index from k to $k' \neq k$
- invalidates the "everything else held constant" assumption

If we could diffentiate the Quantization

But **Quantization is not differentiable** !

- selecting a single choice from the codebook
- "hard" rather than "soft" choice
- which is the cause of the jumps

So we can't use Gradient Descent to update weights.

The Stop Gradient operator will address both these issues with Quantization.

On the Forward Pass, Stop Gradient `sg` 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

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

Essentially, Stop Gradient "defines" the gradient of a non-differentiable operator to be zero.

So no update is passed backwards through the operation

- thereby ensuring that weights of nodes that are closer to the Input
- don't change
- ensuring "everything else held constant" for those nodes

Reconstruction Loss

The Reconstruction Loss term is our familiar: Maximize Likelihood

- written to minimize the negative of the log likelihood, as usual
$$p(\mathbf{x}|\mathbf{z}_q(\mathbf{x}))$$

It's purpose

- to ensure that the reconstruction is as close to the input as possible

This loss will serve to update parameters for the Encoder and Decoder.

As we will see (section: "Quantization is not differentiable")

- during the backward pass
- the Loss gradient from the quantized $\mathbf{z}_q(\mathbf{x})$
    - which is an input to the Decoder
- flows directly to the (continuous) Encoder output $\mathbf{z}_e(\mathbf{x})$

Effectively, for the purpose of gradient/weight update due to Reconstruction Loss:

$$\mathbf{z}_q(\mathbf{x}) = \mathbf{z}_e(\mathbf{x})$$

- the "quantization" node acts like an Identity function

So the Reconstruction Loss will update all parameters

- **except** for the codebook embeddings

The update of the codebook embeddings is the job of two other loss terms.

Vector Quantization Loss

The Vector Quantization Loss and Commitment Loss are similar.

- differ only in the placement of the Stop Gradient

We need two versions of the same Loss

- in order to satisfy "everything else held constant"
- by holding constant one of
    - Codebook update
    - Encoder weight update
- in each of the cases

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).

Its purpose

- is to ensure that the quantized and un-quantized vectors for an input are **as similar as possible**.

This loss will serve to update the parameters of the codebook $\mathbf{E}$.

- while holding the Encoder output constant
    - by preventing update of its weights: $\text{sg}(\mathbf{z}_e(\mathbf{x}))$

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.

Its purpose

- is to ensure that the quantized and un-quantized vectors for an input are **as similar as possible**.

This loss serves to update the weights of the Encoder

- while holding the Codebook constant: $\mathrm{sg}(\mathbf{z}_q(\mathbf{x}))$
- by preventing updates to $\mathbf{E}$

Total Loss

Loss function

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

Need the stop gradient operator sg to control the mutual dependence

- of the Encoder output $\mathcal{E}(\mathbf{x})$ and the chosen code $\mathbf{e}$

Quantization is not differentiable

There is a subtle but important problem.

The Quantization operation

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

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

The Straight Through Estimator. when applied to a node

- uses the Stop Gradient operator
- in conjunction with a specific construction
- to cause the Loss Gradient to *pass through the node unchanged* during back propagation
    - essentially treating the operation of the node as the Identity operation
    - instead of its true computation

We will learn the details in the module Straight Through Estimation (VQ_VAE_Generative.ipynb#Quantization-is-not-differentiable).

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)

Code

Here (https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/generative/ipynb/vq_vae.ipynb#scrollTo=LWYJf1MYvzap) is a Colab notebook.

Creating synthetic examples

Consider an example that is an Image.

It is a structured arrangement of feature vectors

- a 2D grid
- each element of the grid is a vector of features

In general: our examples may have an arbitrary number of dimensions.

By convention, we will refer

- to the last dimensions as the "feature" dimension
- all the preceding dimensions as *non-feature* dimensions
    - for an Image: *spatial* dimensions

Denoting #S as the number of non-feature dimensions and $n_{(0)}$$ as the number of features, examples have shape

$$(n_1 \times \ldots \times n_{\#S} \times n_{(0)})$$

where #S denotes the number of non-feature dimensions (#S = 2 for an Image)

The Encoder creates an output $\mathbf{z}_e(\mathbf{x})$ of shape

$$(n_1 \times \ldots \times n_{\#S} \times n_e)$$

which is quantized into $\mathbf{z}_q(\mathbf{x})$ of shape

$$(n_1 \times \ldots \times n_{\#S} \times 1)$$

That is: the feature dimension of $\mathbf{z}_q(\mathbf{x})$ is a single integer index into a learned codebook.

In order for us to generate synthetic examples

- we must create a Tensor $\mathbf{z}_q(\mathbf{x})$ of $(n_1 \times \ldots \times n_{\#S} \times 1)$ integers (*latents*)
- feed this to the Decoder
- get a synthetic example as output

Learning the distribution of latents

But we can't create $\mathbf{z}_q(\mathbf{x})$ *completely at random.*

- The elements at adjacent locations in the non-feature dimensions may not be **independent**

For example: consider an Image

- the pixels in an image are related to one another

We must learn a distribution of $\mathbf{z}_q(\mathbf{x})$ that respects the dependencies.

One solution

- flatten $\mathbf{z}_q(\mathbf{x})$
- from shape $(n_1 \times \ldots \times n_{\#S} \times 1)$
- into a sequence

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

  of integer indices.

We can then learn the distribution of the latents

- through auto-regressive modeling of the sequence sequence $\mathbf{z}$
$$p(\mathbf{z}_{(k+1)} | \mathbf{z}_{(1)}, \ldots, \mathbf{z}_{(k)})$$

This is just like the Language Model objection for NLP.

Aside

Learning a distribution is less restrictive than *assuming* a distribution

For the case of "structured" examples: we have no choice.

- adjacent elements are not independent

But recall that, for the VAE, we *assumed* that the latents came from a Normal distribution.

Use Auto-regressive modeling is a nice "trick" to learn distributions rather than having to assume a "convenient" functional form.

PCA versus 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

That is:

- $\mathbf{V}^T$ is a basis space that can define all vectors in n-dimensional space
- with loadings $\mathbf{z}'$ on the basis vectors.

To illustrate

- think of the familiar basis space $I \in \mathbb{R}^{n \times n} = \mathbf{diagonal}(n)$
    - $\mathbf{x}$ are the "coordinates" of a point within the n-dimensional space spanned by I
- z' are the "coordinates" of *the same point* but described in basis space $\mathbf{V}^T$

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

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

- 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 $I \in \mathbb{R}^{n \times n} = \text{diagonal}(n)$

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

So can we take the PCA of images ?

Yes !

We did this for the MNIST digits and Fashion MNIST

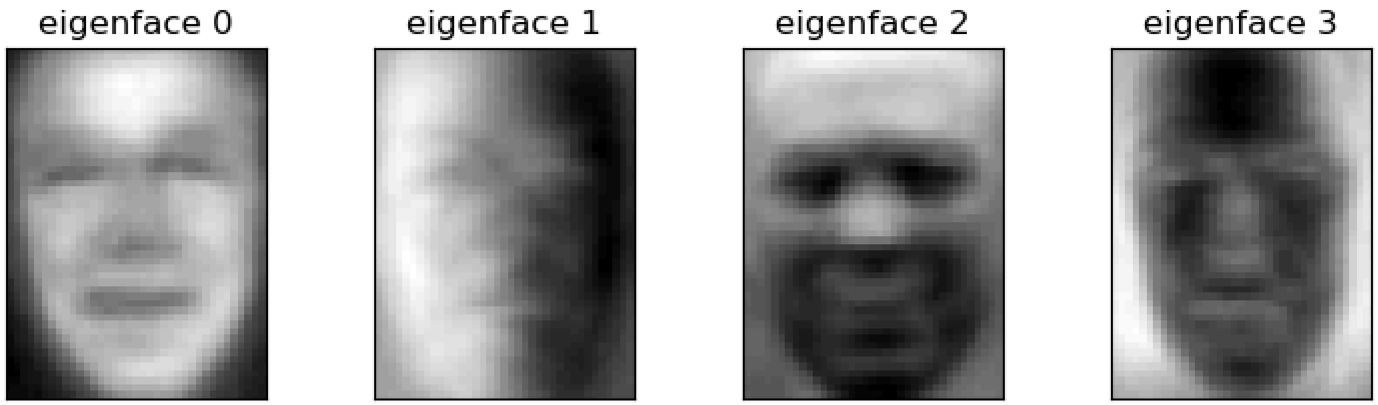
- each basis vector is a pseudo digit/clothing item
- which can be combined
- into an approximation of any digit/clothing item in the *training* dataset

But these were small and simple examples (28×28 pixels)

We can do this for more complex images

- Faces

See Eigenfaces (https://scikit-learn.org/stable/auto_examples/applications/plot_face_recognition.html)


eigenface 0
eigenface 1
eigenface 2
eigenface 3

Thus, the "eigen faces" are a "codebook" of basic faces.

So a face in the training dataset can be encoded

- as a linear combination of $r \le n$ codebook entries

Comparing PCA to VQ-VAE

- PCA chooses $r \leq n$ items from the codebook
    - versus $r = 1$ for VQ-VAE
- The weights $\mathbf{z}'$ (vector of length r) are *continuous*
    - versus a single integer index for the $\mathbf{z}$ of VQ-VAE

Also

- PCA is often used for dimensionality reduction
    - so the size of the codebook $r < n$ is relatively small
- VQ-VAE codebooks are larger
    - a couple hundred to a couple thousand entries
    - codebook entries are typically of length 128 to 512

So PCA is an alternative to VQ-VAE for representing images

- for **some** use cases
- but **not** for the use case of creating a discrete representation of the image

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

```
Done
```