Residual Vector Quantization

An exploration of the heart of neural audio codecs

COMPRESSION SSL

AUTHOR
Scott H. Hawley

PUBLISHED
June 12, 2023

Introduction

"Vector Quantization." Sounds intimidating. "Residual Vector Quantization" sounds even more intimidating, even near-meaningless to most people. Turns out, these are easy to understand given a few pictures, even to the point a child could understand them – uh… if the child *wanted* to. Certainly, there can be sophisticated ways to implement these algorithms, and we'll cover a bit of that later on, but the basics are very simple.

Residual Vector Quantization (RVQ) is a data compression technique found in state-of-the-art neural audio codecs such as Google's SoundStream, and Facebook/Meta AI's Encodec, which in turn form the backbone of generative audio models such as AudioLM (Google) and MusicGen (Facebook). It's also the subject of Lucidrain's library `vector-quantize-pytorch`, which we'll use toward the end because it's *so fast* and *so good*.

What is RVQ and how does it work?

First we should consider regular vector quantization (VQ). VQ has been around for decades, and it shows up in many areas of signal processing when compression is involved.

> Note: Whenever I say something that applies to both VQ and RVQ, I'll use the abbreviation "(R)VQ."

Tip: Two Analogies

1. **Cities and Towns:** RVQ is like a "hub-and-spoke" *graph* that often appears in logistics: Consider air travel, in which the major cities are "hubs" (Chicago, LA, Atlanta) from which you take smaller flights to get to smaller cities and towns. VQ would be like replacing every address with its nearest town – which could result in a lot of vectors! RVQ means we have a short list of hubs, then from each hub we have a list of smaller cities, from which we could then have lists connecting smaller cities to nearby towns.

2. **Numbers and Digits:** In one dimension, RVQ is like the way we represent numbers using digits. Instead of creating 10,000 separate categories for each of the integers from 0 to 9999, we use 4 "codebooks" (for thousands, hundreds, tens, and ones) consisting of the ten digits 0 through 9. 4 * 10 = 40, which is a lot less than 10,000! We even can represent real numbers to arbitrary precision by using more codebooks to include ever-smaller "residual vectors" to the right of the decimal point.

Vector Quantization = Partitioning Space

"Vector Quantization" is really about dividing up the space of your data points into a discrete set of regions. Put differently, we "partition" the space.

Let's say we have a bunch of points in space:

▼ Show the code

```
import numpy as np
import matplotlib.pyplot as plt

# make some data
n_points = 25
DATA_MIN, DATA_MAX = -0.5, 0.5 # we'll let these be globals
np.random.seed(9)  # for reproducibility
data = DATA_MIN + (DATA_MAX-DATA_MIN)*np.random.rand(n_points, 2)

# plot it
fig, ax = plt.subplots(figsize=(3,3))
ax.set_xlim(DATA_MIN, DATA_MAX)
ax.set_ylim(DATA_MIN, DATA_MAX)
#ax.set_xticks([])  # hide axis ticks
#ax.set_yticks([])
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.scatter(data[:, 0], data[:, 1], s=16)
plt.show()
```

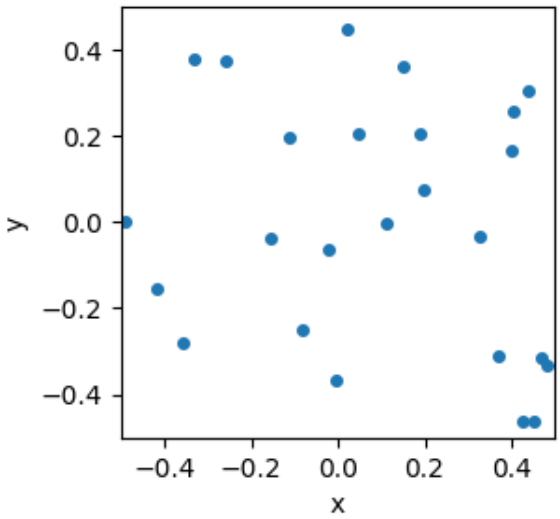


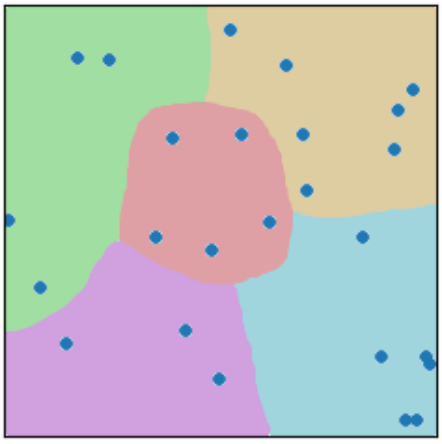
Figure 1. A set of data points, aka 'vectors'.

To computer scientists, the (x, y) coordinates of each point define a "vector". (To mathematicians and physicists, the "vector" points from the origin to each point, but that distinction isn't going to matter to us.)

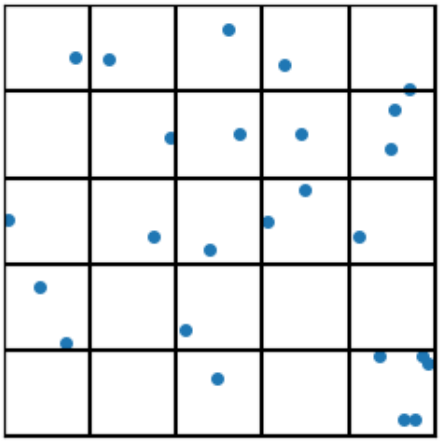
> **Clarification**
>
> For neural audio codecs, the vectors could be vertical slices (columns) from an audio spectrogram, i.e., a list of amplitudes for a set of frequencies that were found over a short time interval called a "frame." More often, however, the vectors are *themselves* the outputs of some other audio encoder (e.g. a Convolutional Neural Network), and the (R)VQ is done to compress *those* encoded vectors even more in order to do things like Latent Diffusion.

Now divide up the space into a bunch of regions. How we do that can take many forms. For now, just consider the two examples of my hand drawing with colors, and a set of tiled squares. There are "fancier" algorithms to partition

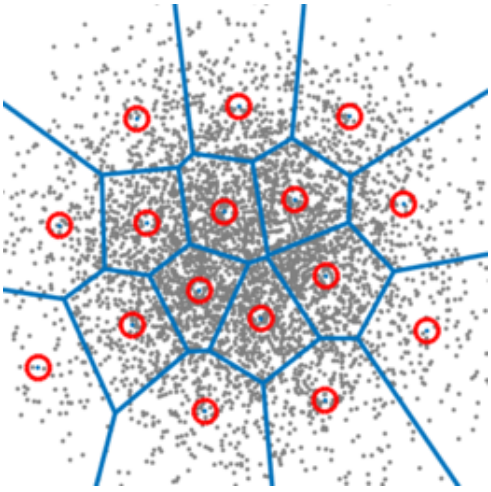
the space in a way that best "fits" the data (e.g., see "k-Means", below). We can cover schemes like the third image later.



(a) By hand



(b) Squares



(c) Fancy (credit: Aalto University, Finland)

Figure 2. Examples of ways to divide up or "quantize" space, aka "Partitioning schemes."

Moving forward, I can do a bit more code-wise with the squares, so let's start there. ;-)

Let me just formalize that a bit: We'll let the number of squares be controlled by the variable "`n_grid`". So for our two-dimensional examples, there will be `n_grid` ^{2} square regions.

With vector quantization, we give an index value to every region (e.g. 0 to 24 for a 5x5 square grid), and then we *replace each vector's value with the index of the region*.

▶ Show the code

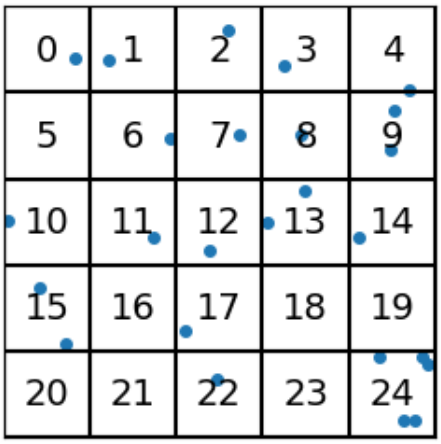


Figure 3. (Integer) Indices for each region.

For each of the "vectors" represented by the data points, we no longer use the (x,y) coordinate pairs, but rather the (integer) index of the region it's in.

> **Note**
>
> We've gone from needing two floating point numbers per point to just one integer value. In two dimensions, the "savings" or data-compression amount that we get from that may not be readily apparent, but stick around: As we go to large numbers of dimensions later, this scheme will save us a *ton* of data.

If we want the coordinates that go with the indices, we'll use the centroid of each region. In this sense, the vectors are "quantized" so that they can only take on values given by the centroids of the regions. In the following example, the centroids are shown in red:

▶ Show the code

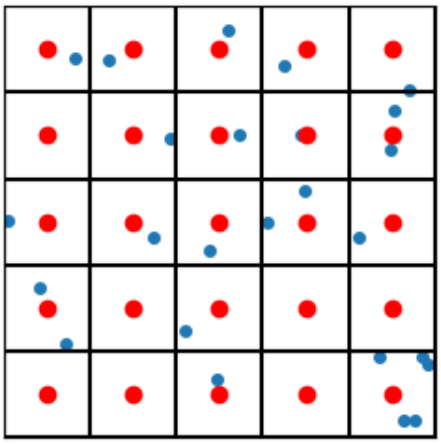


Figure 4. Centroid locations

So each blue point will effectively be replaced by the nearest red point. In this sense we have "quantized" the vectors (because we've quantized the space itself).

> **Terminology**

> The set of locations of centroids is called the "**codebook**". When we want to use an actual vector value (in space), we convert the codebook index into a (centroid) location by looking up the codebook.

So a full (albeit unweidly) picture showing the data points, the region indices, and the centroids, looks like this:

▼ Show the code

```
plot_data_grid(data, n_grid=n_grid, show_indices=True, show_centroids=True, hide_tick_labels=
```

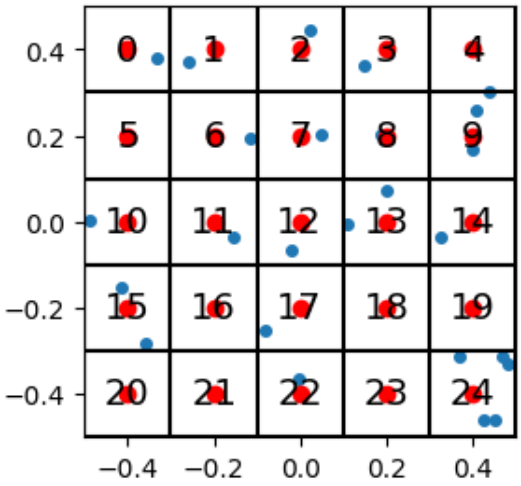


Figure 5. Detailed figure showing data points, region indices, and centroid locations.

For our choice of coordinates, the codebook (i.e. the mapping of indices to centroid locations) looks like this:

▶ Show the code

index	vector
0	(-0.4, -0.4)
1	(-0.2, -0.4)
2	(0.0, -0.4)
3	(0.2, -0.4)
4	(0.4, -0.4)
5	(-0.4, -0.2)
6	(-0.2, -0.2)
7	(0.0, -0.2)
8	(0.2, -0.2)
9	(0.4, -0.2)
10	(-0.4, 0.0)
11	(-0.2, 0.0)
12	(0.0, 0.0)
13	(0.2, 0.0)
14	(0.4, 0.0)
15	(-0.4, 0.2)
16	(-0.2, 0.2)
17	(0.0, 0.2)

index	vector
18	(0.2, 0.2)
19	(0.4, 0.2)
20	(-0.4, 0.4)
21	(-0.2, 0.4)
22	(0.0, 0.4)
23	(0.2, 0.4)
24	(0.4, 0.4)

Reconstruction Error

When we do this quantization (i.e. replacing vectors by their nearest centroid), the centroid locations will naturally be a bit "off" compared to the original data vectors themselves. The finer the grid you make, the smaller the regions, and the less error. For a 2D grid, the error will be on the order of h^2 where h is the grid spacing (h = 1/5 = 0.2 in our example).

> **Note**
>
> Note that the vectors in the codebook are *not* "basis vectors": We do not add linear combinations of the codebook vectors, as that would not be "quantization" (and would subject us to the same number of data points as the original, resulting in nearly no compression). VQ helps us get around some problems of having large numbers of data points by approximating them by their nearest codebook vectors, and the R in RVQ allows us to increase the provide for good "resolution" within the space *without* requiring extremely long codebooks.

Let's check how the error changes as we vary the grid spacing, i.e., as we vary `n_grid`.

▶ Show the code

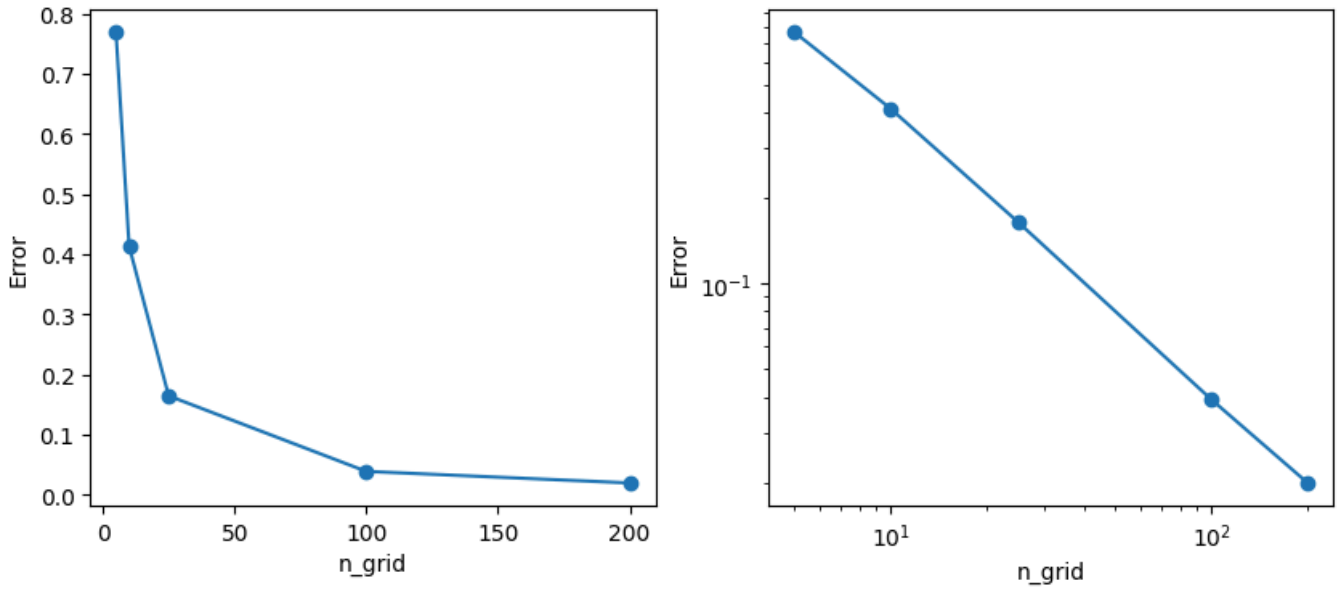


Figure 6. Plots of error vs. resolution with linear (left) and logarithmic (right) axes. Note that the computational cost will scale like the number of regions, which is `n_grid` ^{2}.

```
lowest error (for n_grid=200) =  0.019903707672610238
```

So, the more "gridlines" you use, the lower the error, but at what cost? To get an error of 0.02 we need $200^2 = 400$ regions. And in higher dimensions than 2, the "cost" of higher resolution / lower error goes up *immensely*: to double

the resolution in d dimensions, the computational cost goes up by a factor of 2^d. (Imagine d=32, 64, 128,...)

But we don't need to cover the entire space uniformly! That's where Residual Vector Quantization comes in. Feel free to skip ahead to the section on RVQ. For the next bit we'll take an optional aside to learn about an alternate way to partition space, known as a the "k-Means" algorithm.

> **Note**
>
> Another key point: By replacing all the coordinate values for a vector (i.e. d floating point numbers) with a *single integer*, VQ achieves data compression by a factor of d (times however many bit floats take up compared to integers). For large numbers of dimensions – regardless of the partitioning scheme – this compression can be *significant*.

k-Means (Partitioning Scheme)

> **Optional**
>
> This discussion on k-Means is actually not crucial to understanding (R)VQ. At all. It's quite skippable, to be honest. So...only read if you're really curious. Otherwise skip down to the section on Residual Vector Quantization.

▶ Details on k-Means

Residual Vector Quantization (RVQ)

By the way, RVQ has been around for quite a while.

Basic Idea: "Codebooks in Codebooks"

The trick with RVQ is, rather than having a single high-resolution codebook, to instead have "codebooks inside of codebooks", or, if you like, "stacking codebooks". Let's say we want to quintuple the resolution of our initial 5x5 grid. Instead of using a 25x25 grid (which would be 25x the computational cost of the original), what if we put a little 5x5 grid "inside" the region to which a vector was quantized?

For example, in the "middle" region (region 12), we can do...

▶ Show the code

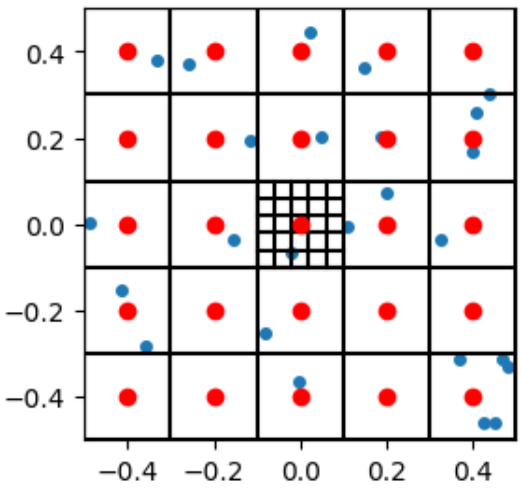


Figure 13. Illustration of 'codebook within a codebook', with a smaller 5x5 codebook which will be relative to the middle region's codebook.

The difference between that blue point in the middle "main" square and its corresponding red centroid will be the "residual". We will also quantize that within the "little" 5x5 grid. This will serve as a codebook to use "after" the

original codebook. And we'll get the same resolution as if we had a 25x25 grid, except our computational cost will instead be 2*(5*5)=50 instead of 25*25=625! So our cost will be 12.5 smaller than the full-grid approach.

And interestingly, if we only consider the *residual*, i.e. the difference between the main centroid and the vector in question, then we can use the same "next level" codebook for *all* points in the space! In the following figure, we show the residuals as purple line segments that run from each point to its corresponding nearest centroid:

▶ Show the code

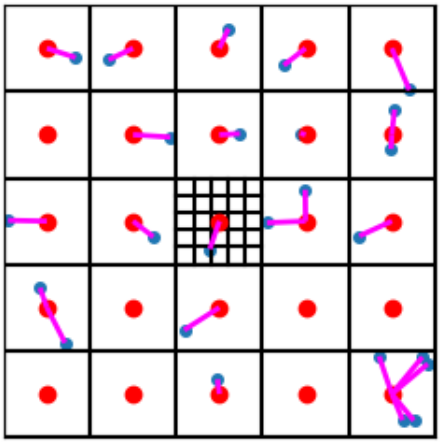


Figure 14. Illustration of redisuals, shown as purple line segments connecting vectors (blue points) with their nearest centroids (red points).

Also, because we cleverly set up our data to be centered around the origin $(0, 0)$, we can treat the original data points as "residuals" relative to the "centroid" of the whole domain, namely the origin!

▶ Show the code

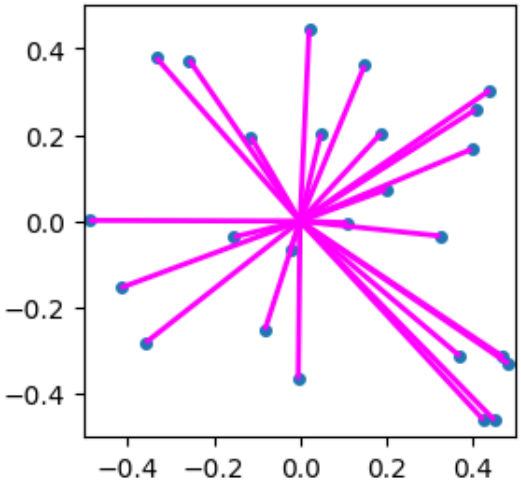


Figure 15. How we can treat the data points themselves as residuals relative to the origin, for a 'level 0' codebook

Also because of our clever choice of coordinates, for the next "level" of quantization, we can take the codebook at the next level to be just the previous codebook divided by `n_grid` ! This won't always be the case; I'm just feeling clever and lazy.

Quantizer algorithm

Up til now, we've hidden the code by default. But to really get the RVQ method, I'm going to show the code.

Let's write a general quantizer multiple "levels" of nested codebooks. It will take our data points and return the various levels of codebook indices.

Will be following "Algorithm 1" from page 4 of Google's 2021 paper "SoundStream: An End-to-End Neural Audio Codec":

Algorithm 1: Residual Vector Quantization

Input: $y = \text{enc}(x)$ the output of the encoder, vector quantizers Q_i for $i = 1..N_q$
Output: the quantized $\hat{y}$
$\hat{y} \leftarrow 0.0$
residual $\leftarrow y$
for $i = 1$ *to* N_q **do**
  $\hat{y}$ += Q_i(residual)
  residual −= Q_i(residual)
return $\hat{y}$

Figure 16. SoundStream's RVQ Algorithm

But I like the way I write it:

```
def quantizer(data, codebooks, n_grid=5):
    "this will spit out indices for residuals in a series of 'nested' codebooks"
    resids = data
    indices = []
    for cb in codebooks:
        indices_l = get_region_membership(resids, codebook=cb)
        resids = resids - cb[indices_l]
        indices.append(indices_l)
    return np.array(indices)

# Make the nested codebooks
n_codebooks = 3
codebook = generate_codebook(n_grid)
codebooks = [codebook/n_grid**level for level in range(n_codebooks)]

indices = quantizer(data, codebooks)   # call the quantizer
display(indices)
```

```
array([[10,  2,  5,  7,  5, 20,  4, 13, 19, 17, 16, 14, 24,  4, 22, 23,
         4, 21, 19, 11, 13, 12,  4, 18,  4],
       [10, 17,  3,  5, 17,  9,  3, 22,  7, 13, 14,  5,  3, 24, 18,  6,
        24,  6, 17,  8, 10,  1, 21, 12,  3],
       [11,  6, 12,  7, 15, 20, 23,  7, 17, 13,  8, 18,  2,  7, 15, 11,
        16, 20, 23, 13, 11, 24, 18, 10, 20]])
```

Let's test this by trying to reconstruct our original data using each level of codebook. In the following, the original data will be in blue, and the other colors will show the results of quantization using an increasing number of codebooks:

▶ Show the code

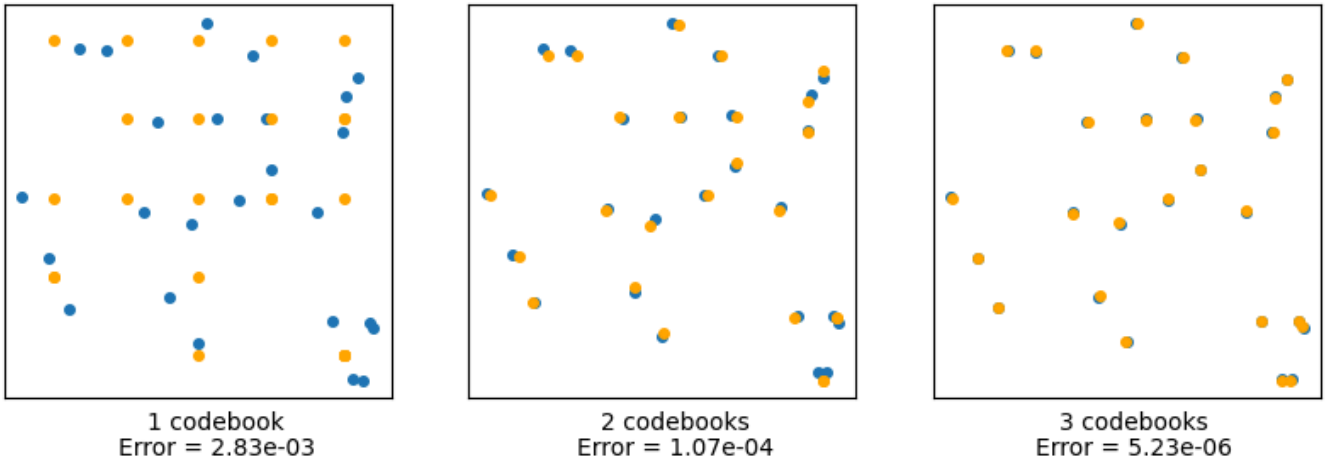


Figure 17. Reconstructing data (blue points) using multiple levels of RVQ codebooks (orange dots)

We see that the more (levels of) codebooks we use, the better we can approximate the original data. The rightmost image has an effective resolution of $5^3 = 125$ little squares, but instead uses only $5x5x3 = 75$. In two dimensions, this is not a huge savings, but let's see how important this is for higher numbers of dimensions.

Let d be the number of dimensions and K be the number of codebooks (no relation to the k from k-Means, I'm afraid). We'll populate a d-dimensional hypercube with a lot of data, and subdivide it into nested groups of little hypercubes using RVQ, and compute the error – as well as the compuational cost "savings" from if we had used regular VQ instead of RVQ.

> **Note**
>
> Using uniform square/(hyper)cube regions is a *really dumb* idea for this. Because the number of regions will scale like `n_grid` ^{d}, which can actually be *much* larger than the number of data vectors we have! We'll try a more sophisticated partitioning scheme further below.

▶ Show the code

```
Here we show the error for high-dimensional datasets using various levels of RVQ.
'cost savings factor' refers to the ratio of using regular VQ (at uniform resolution)
vs RVQ.

d = 2:
  K = 1, error = 3.41e-03, cost savings factor = 1.0
  K = 2, error = 1.29e-04, cost savings factor = 12.5
  K = 3, error = 5.26e-06, cost savings factor = 208.3
  K = 4, error = 2.16e-07, cost savings factor = 3906.2

d = 3:
  K = 1, error = 3.37e-03, cost savings factor = 1.0
  K = 2, error = 1.29e-04, cost savings factor = 62.5
  K = 3, error = 5.31e-06, cost savings factor = 5208.3
  K = 4, error = 2.18e-07, cost savings factor = 488281.2

d = 4:
  K = 1, error = 3.36e-03, cost savings factor = 1.0
  K = 2, error = 1.32e-04, cost savings factor = 312.5
  K = 3, error = 5.34e-06, cost savings factor = 130208.3
  K = 4, error = 2.16e-07, cost savings factor = 61035156.2

d = 6:
  K = 1, error = 3.37e-03, cost savings factor = 1.0
```

```
K = 2, error = 1.33e-04, cost savings factor = 7812.5
K = 3, error = 5.35e-06, cost savings factor = 81380208.3
K = 4, error = 2.16e-07, cost savings factor = 953674316406.2
```

Those "cost savings factors" were artifically high though, because we're still using squares/hypercubes for our regions, and we don't need to shape them that way and we don't need *that many* of them. The great thing about (R)VQ is that you can specify how many centroids you want – i.e. how "long" you want your codebook to be – and you can keep that to some manageable number even as the number of dimensions skyrockets.

So, to tackle higher dimensions, we need to stop using uniform squares so we can have a codebook "length" of less than a few thousand centroids (instead of what just had, which was going into the hundreds of thousands, e.g. $5^8 = 390625$). To get our non-uniform regions that follow the data, we'll use the k-Means method described above.

Let's look at how the reconstruction error behaves in high dimensions.

Error Analysis: Exponential Convergence

We can try a given number of (initially) random centroids, and try to match them to the data via k-Means.

> **Note**
>
> The residuals at different levels of our RVQ calculations will likely have different data distributions. Which means that instead of "sharing (scaled) codebooks" like we did before, we'll need to re-calculate a new codebook at each "level". Otherwise we won't see any advantage from RVQ (trust me I tried).

In the following calculation, we'll vary the number of dimensions, the length of the codebooks, and the number of codebooks, and see how these all affect the reconstruction error.

But rather than using our code, use let's use the wonderful repository by lucidrains: [lucidrains/vector-quantize-pytorch]

The following set of error values is a "wall of numbers" which may not interest you. Feel free to scroll past it and skip down to the graphical representation of (some of) the numbers.

▶ Show the code

d = 2:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	7.5e-02	6.8e-03	8.9e-04	1.2e-04	4.8e-06	1.1e-07	2.6e-09
cb_len = 64	3.1e-02	1.7e-03	1.3e-04	1.3e-05	2.1e-07	3.5e-09	6.9e-11
cb_len = 256	8.5e-03	1.9e-04	7.4e-06	4.5e-07	3.1e-09	1.8e-11	1.5e-13
cb_len = 1024	2.0e-03	2.4e-05	9.6e-07	6.1e-08	9.4e-11	4.8e-14	3.5e-15
cb_len = 2048	6.0e-04	1.8e-06	3.9e-08	7.8e-10	1.4e-12	3.8e-15	1.4e-15

d = 3:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	1.8e-01	3.5e-02	7.7e-03	1.8e-03	1.3e-04	1.0e-05	9.1e-07
cb_len = 64	9.8e-02	1.1e-02	1.4e-03	2.2e-04	5.7e-06	2.7e-07	5.8e-09
cb_len = 256	3.6e-02	1.7e-03	9.8e-05	9.6e-06	1.1e-07	1.8e-09	4.2e-11
cb_len = 1024	1.2e-02	2.7e-04	8.0e-06	4.5e-07	1.5e-09	4.2e-12	2.4e-14

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 2048	4.4e-03	4.7e-05	9.3e-07	2.5e-08	7.5e-12	3.8e-15	1.8e-15

d = 6:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	4.3e-01	1.9e-01	8.2e-02	3.7e-02	8.6e-03	2.1e-03	5.5e-04
cb_len = 64	3.2e-01	1.0e-01	3.3e-02	1.1e-02	1.3e-03	1.7e-04	2.2e-05
cb_len = 256	1.8e-01	3.2e-02	6.0e-03	1.1e-03	4.0e-05	1.5e-06	5.6e-08
cb_len = 1024	8.0e-02	6.7e-03	5.8e-04	5.0e-05	4.0e-07	3.6e-09	3.0e-11
cb_len = 2048	3.8e-02	1.6e-03	7.9e-05	3.6e-06	1.1e-08	4.5e-11	2.5e-13

d = 8:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	5.3e-01	2.8e-01	1.5e-01	8.2e-02	2.4e-02	7.7e-03	2.6e-03
cb_len = 64	4.2e-01	1.7e-01	7.4e-02	3.2e-02	6.2e-03	1.2e-03	2.6e-04
cb_len = 256	2.7e-01	7.1e-02	1.9e-02	5.2e-03	3.9e-04	3.0e-05	2.2e-06
cb_len = 1024	1.3e-01	1.8e-02	2.5e-03	3.5e-04	6.6e-06	1.3e-07	2.7e-09
cb_len = 2048	6.6e-02	4.7e-03	3.4e-04	2.8e-05	1.8e-07	1.6e-09	1.5e-11

d = 16:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	7.3e-01	5.3e-01	3.9e-01	2.8e-01	1.5e-01	7.9e-02	4.2e-02
cb_len = 64	6.4e-01	4.0e-01	2.6e-01	1.6e-01	6.6e-02	2.7e-02	1.1e-02
cb_len = 256	4.9e-01	2.4e-01	1.2e-01	5.6e-02	1.3e-02	3.2e-03	7.8e-04
cb_len = 1024	3.0e-01	9.0e-02	2.7e-02	8.1e-03	7.7e-04	7.6e-05	7.4e-06
cb_len = 2048	1.6e-01	2.8e-02	4.7e-03	7.6e-04	2.7e-05	1.0e-06	3.8e-08

d = 32:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	8.5e-01	7.2e-01	6.1e-01	5.1e-01	3.7e-01	2.6e-01	1.9e-01
cb_len = 64	7.8e-01	6.1e-01	4.8e-01	3.8e-01	2.3e-01	1.4e-01	8.7e-02
cb_len = 256	6.7e-01	4.4e-01	3.0e-01	2.0e-01	9.0e-02	4.1e-02	1.8e-02
cb_len = 1024	4.6e-01	2.2e-01	1.0e-01	4.7e-02	9.9e-03	2.3e-03	5.1e-04
cb_len = 2048	2.7e-01	7.1e-02	1.9e-02	5.1e-03	4.2e-04	3.6e-05	2.6e-06

d = 64:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	9.1e-01	8.4e-01	7.6e-01	7.0e-01	5.8e-01	4.8e-01	4.0e-01
cb_len = 64	8.7e-01	7.6e-01	6.7e-01	5.8e-01	4.5e-01	3.5e-01	2.7e-01
cb_len = 256	7.9e-01	6.6e-01	5.5e-01	4.5e-01	3.2e-01	2.2e-01	1.5e-01

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 1024	5.8e-01	3.4e-01	2.0e-01	1.2e-01	4.2e-02	1.5e-02	5.6e-03
cb_len = 2048	3.5e-01	1.2e-01	4.3e-02	1.5e-02	2.4e-03	3.6e-04	5.2e-05

d = 128:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	9.5e-01	9.1e-01	8.6e-01	8.2e-01	7.4e-01	6.7e-01	6.0e-01
cb_len = 64	9.3e-01	8.6e-01	8.0e-01	7.4e-01	6.4e-01	5.6e-01	5.1e-01
cb_len = 256	8.6e-01	7.9e-01	7.2e-01	6.5e-01	5.4e-01	4.4e-01	3.6e-01
cb_len = 1024	6.5e-01	4.7e-01	3.4e-01	2.5e-01	1.3e-01	6.0e-02	2.6e-02
cb_len = 2048	4.1e-01	2.0e-01	9.3e-02	4.4e-02	8.3e-03	1.8e-03	8.4e-05

d = 256:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	9.7e-01	9.4e-01	9.2e-01	8.9e-01	8.4e-01	8.0e-01	7.5e-01
cb_len = 64	9.5e-01	9.1e-01	8.7e-01	8.4e-01	7.8e-01	7.4e-01	7.1e-01
cb_len = 256	8.9e-01	8.3e-01	7.6e-01	7.0e-01	5.9e-01	5.0e-01	4.2e-01
cb_len = 1024	6.9e-01	5.2e-01	3.8e-01	2.8e-01	1.5e-01	7.8e-02	4.4e-02
cb_len = 2048	4.4e-01	2.2e-01	1.1e-01	5.2e-02	1.2e-02	2.3e-03	2.6e-04

d = 512:

	K = 1	K = 2	K = 3	K = 4	K = 6	K = 8	K = 10
cb_len = 25	9.8e-01	9.7e-01	9.5e-01	9.3e-01	9.0e-01	8.7e-01	8.4e-01
cb_len = 64	9.7e-01	9.4e-01	9.3e-01	9.1e-01	8.8e-01	8.4e-01	8.0e-01
cb_len = 256	9.1e-01	8.6e-01	7.9e-01	7.3e-01	6.3e-01	5.4e-01	4.6e-01
cb_len = 1024	7.1e-01	5.3e-01	3.9e-01	2.9e-01	1.5e-01	8.2e-02	4.5e-02
cb_len = 2048	4.6e-01	2.2e-01	1.1e-01	5.0e-02	1.2e-02	2.3e-03	6.1e-04

> Clarification: RVQ with K=1 is the same thing a regular VQ.

▶ Show the code

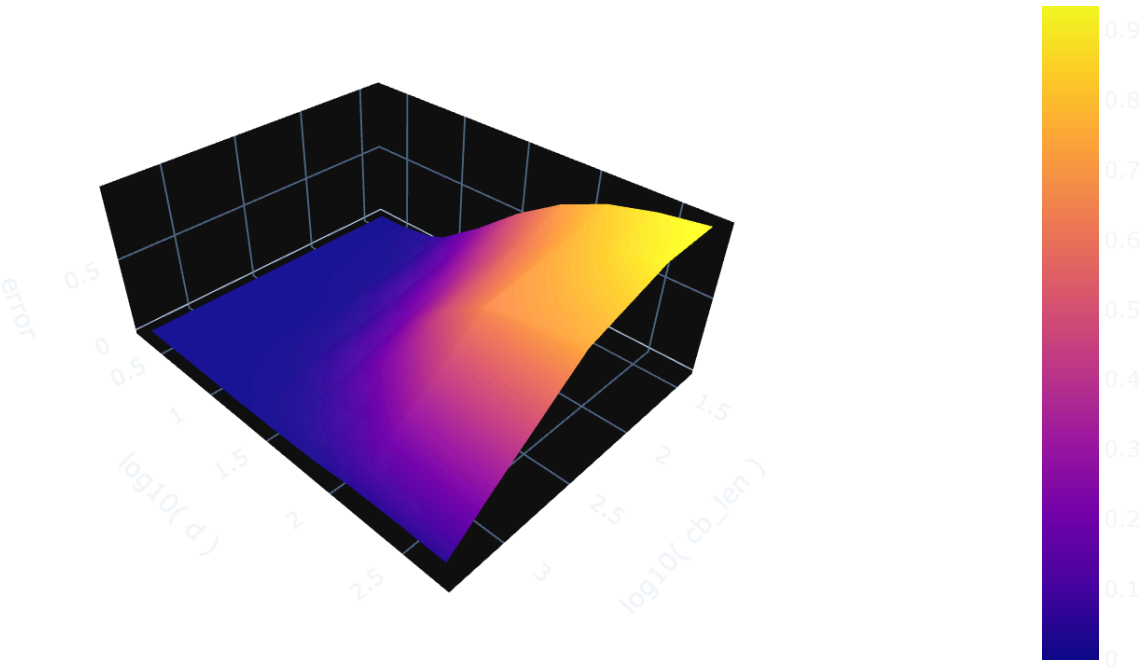


Figure 18. (Interactive) Surface plot of reconstruction error for 4 codebooks with various data dimensions and codebook lengths.

So as we expect, longer codebooks help, and at higher dimensions, error tends to be larger.

Here's the same thing but with a logarithm axis for the error:

▶ Show the code

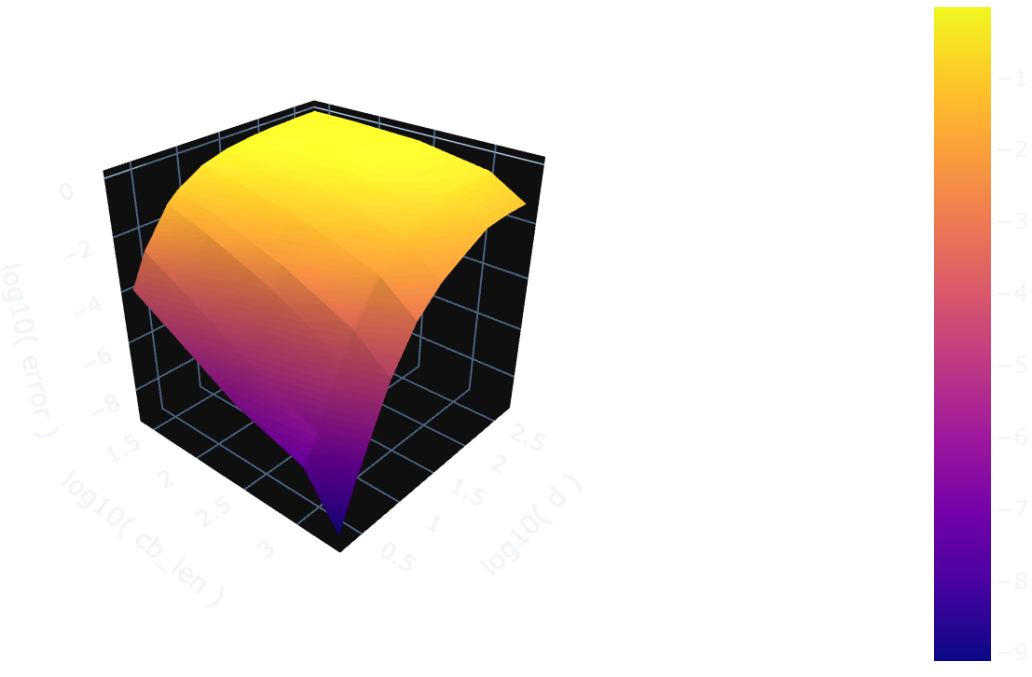
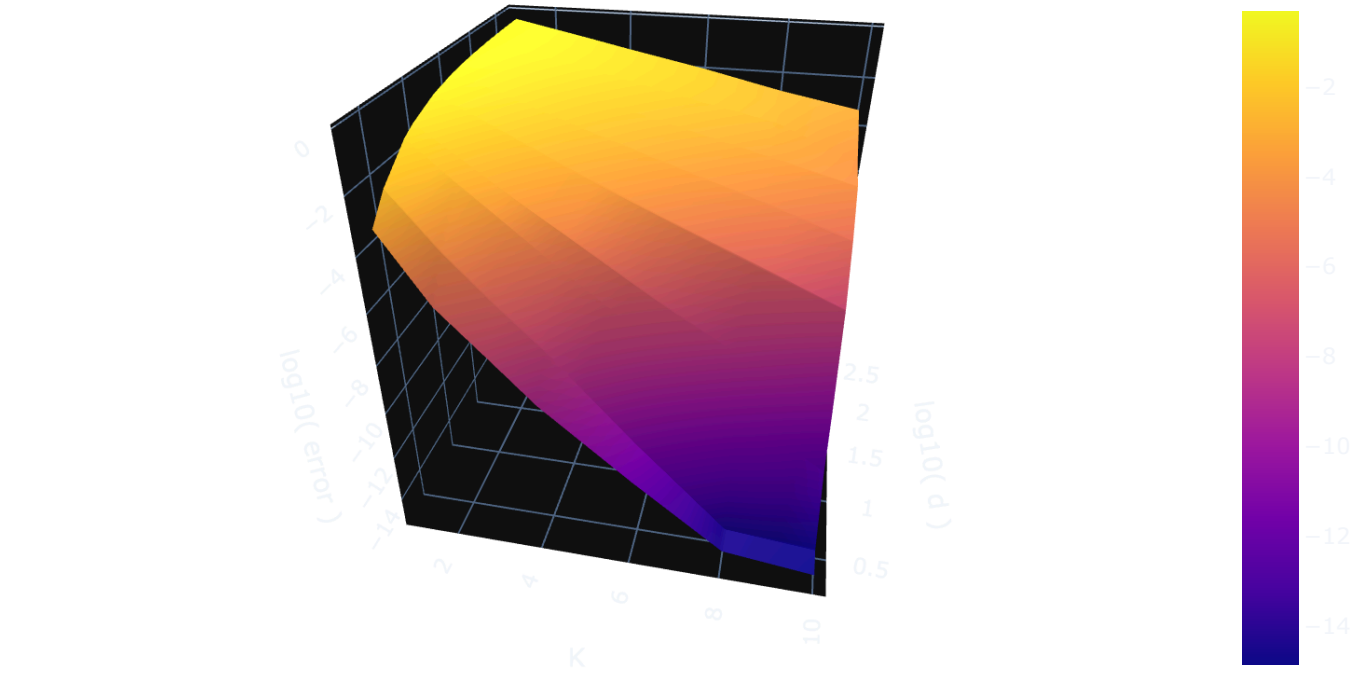


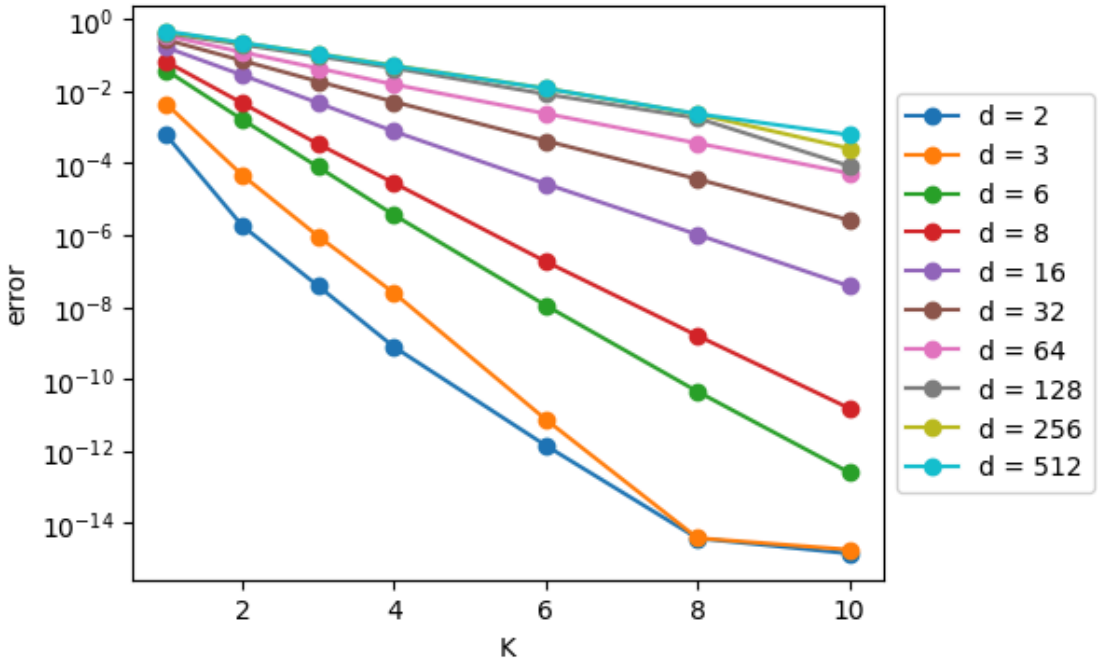
Figure 19. (Interactive) Surface plot of reconstruction error for 4 codebooks with various data dimensions and codebook lengths. (Logarithmic z-axis)

And now let's look at the error for a codebook length of 2048, where we vary the number of codebooks:

▶ Show the code



(a) (Interactive) 3D Surface Plot



(b) Line plot

Figure 20. Reconstruction error for codebook length of 2048 with various data dimensions and number of codebooks (K). (Logarithmic z-axis)

Note that in the last figure, we see **straight lines**[1] across as we vary K, and the K-axis is linear and the error axis is logarithmic.* What this means is – and this is a major takeway:

> **Big Idea**
>
> **As we add codebooks to the RVQ algorithm, the error decreases *exponentially*!**

This is a big "selling point" for the RVQ method: you get an *exponential* payoff in error reduction for a *linear* increase in computational cost.

Addendum: Difficulty at Very High Dimensions

However, the above results also indicate that, for very high dimensions (say, $d \geq 128$), (R)VQ doesn't offer nearly as big of a payoff as is does for lower dimensions – adding more and larger codebooks doesn't have *that much* of an effect on the errors.

For this reason, in their *brand new*[2] paper "High-Fidelity Audio Compression with Improved RVQGAN", the Descript team choose to project down to a lower-dimensional space ($d = 8$) first, using Linear layers, and *then* perform RVQ, and then project back up using another set of Linear Layers.

...Pausing here for now. There's more that we could say and do – for example, "How do you do backpropagation with RVQ?" – but this seems like a good place to pause this discussion.

Acknowledgement

Thanks to Zach Evans of Harmonai for the impetus to look into this topic, and for helpful discussions and corrections while I was writing.

Footnotes

1. Yes, the lines flatten out when we reach machine precision of 10^{-15}; don't worry about that.↩
2. Brand new" as in "came out while I was finishing this post"!↩