CMPN206: Multimedia

Lecture 5: Quantization

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Spring 2014

Agenda

- Quantization
- Scalar Quantization
  - Uniform
  - Nonuniform
- Vector Quantization

- Next: Transform Coding

- More information: Chapter 9, 10 [**IDC**]

Acknowledgments: Most slides are adapted from Richard Ladner, from Li and Drew, and from Khaled Sayood.

Lossless vs. Lossy

- Lossless: no information loss in the compression and decompression process $y = x$

- Lossy: some information is lost in the compression and decompression process $y \neq x$

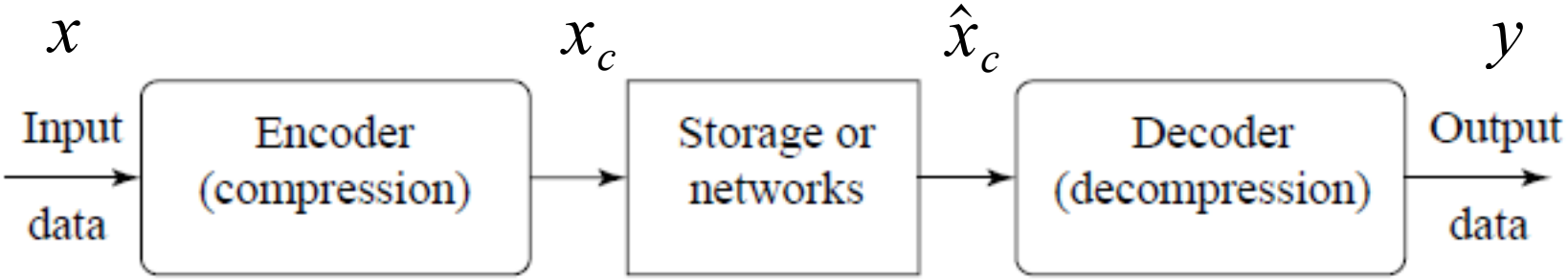


Fig. 7.1: A General Data Compression Scheme.

Lossy Compression

- Compression ratios from *lossless* compression is *not* enough for multimedia applications
- Recall that the *rate* of any *lossless* algorithm is bounded by *entropy* of the source
- Most multimedia compression algorithms are *lossy* to achieve rates *lower* than the entropy
- We need to have some measure of *distortion* i.e. how much different the reconstructed data is from the original data
- For a lossy algorithm we need to minimize the *rate* while also minimizing the *distortion*

Distortion Criteria

- We need to measure the *fidelity* of the compressed data i.e. how close is it to the source data
- This depends on the application:
  - When compressing a painting, we need to ask artists
  - When compressing an audio piece, we need to ask listeners
  - When compressing a house image to be used in ads, we need to ask real estate agents
- All these answers will be *subjective* i.e. depend on the person being asked
- We also need some *objective* measures that don't depend on the subjects

Distortion Measures

- *Mean Squared Error* (MSE):

$$\sigma_d^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - y_n)^2$$

where N is the number of samples

- *Signal-to-Noise Ratio* (SNR):

$$\text{SNR} = 10\log_{10}\frac{\sigma_x^2}{\sigma_d^2}\ \text{dB}$$

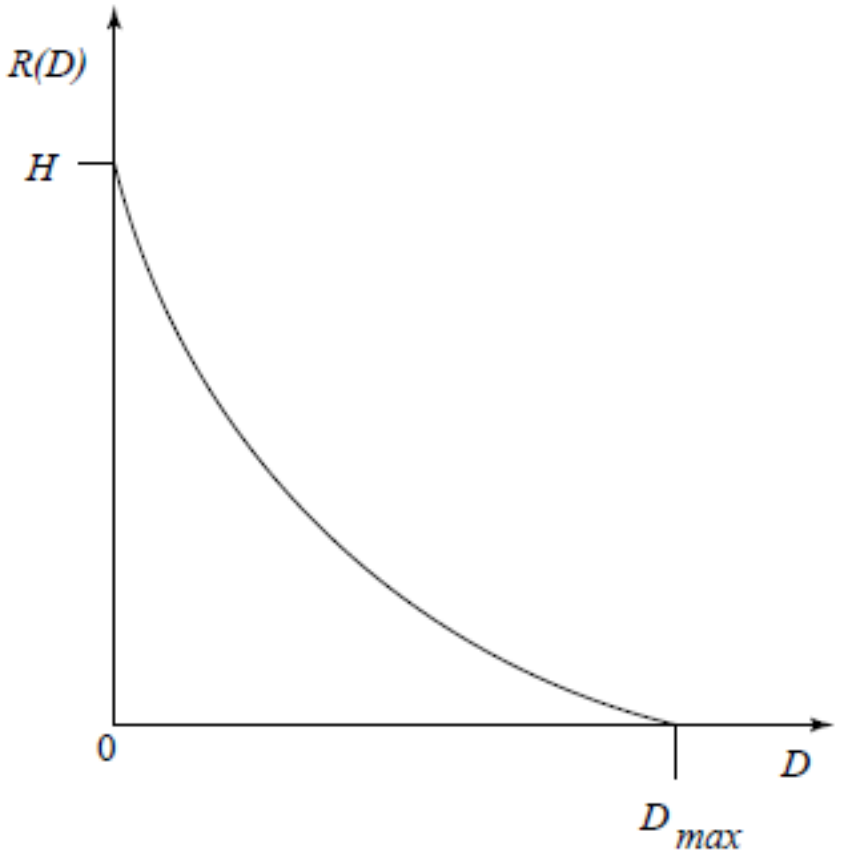
where σ^2_x is the *average squared value* and σ^2_d is the MSE

- *Peak Signal-to-Noise Ratio* (PSNR):

$$\text{PSNR} = 10\log_{10}\frac{x_{peak}^2}{\sigma_d^2}\ \text{dB}$$

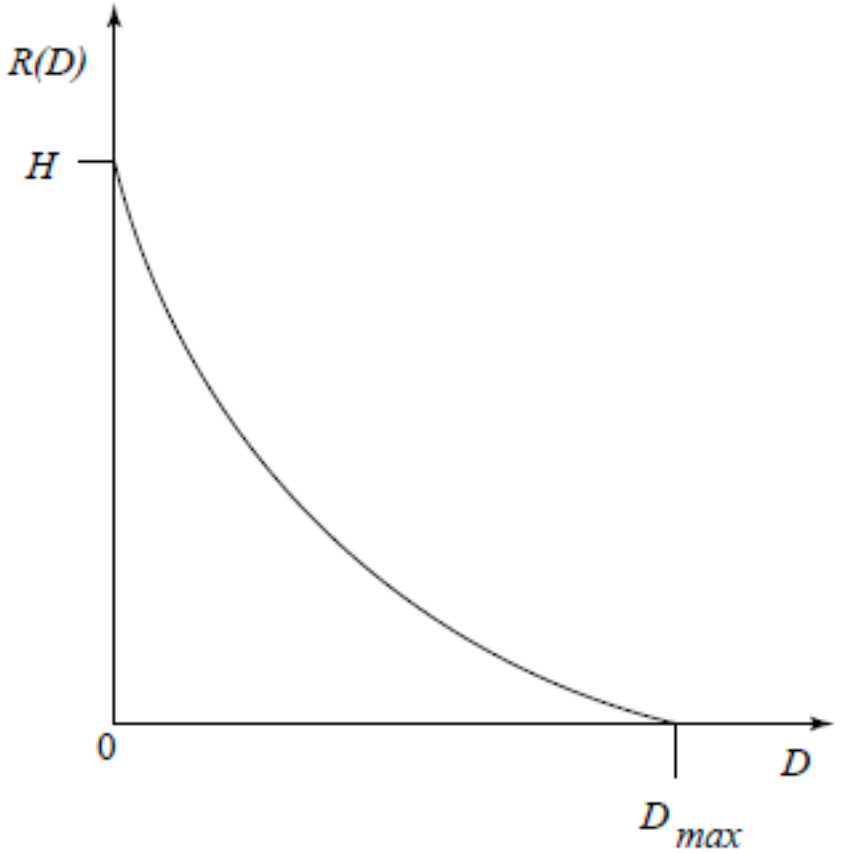
Rate Distortion Theory

- Studies the *tradeoff* between *rate* and *distortion*
  - To decrease the rate, distortion increases
  - To decrease the distortion, the rate increases
- Define the *rate-distortion function* $R(D)$



Rate Distortion Function

- Defines the *rate-distortion function* $R(D)$ which specifies the lowest rate achievable for the rate to stay below a certain value D.
  - When $D = 0 \rightarrow R(D) = H$
  - When $D > 0 \rightarrow R(D) < H$



Quantization

- At the heart of most *lossy* compression techniques
- The main source of information loss
- Main Idea: represent the *source* outputs using a *smaller* number of codewords, where the number of source symbols is much larger the number of codewords
- Two broad types:
  - Scalar Quantization: works on scalar values
  - Vector Quantization: works on vectors

Example

- A source that emits real values between [–10, 10] can be quantized by *rounding* to the nearest integer
- The infinite number of source outputs can be *quantized* to only 21 values

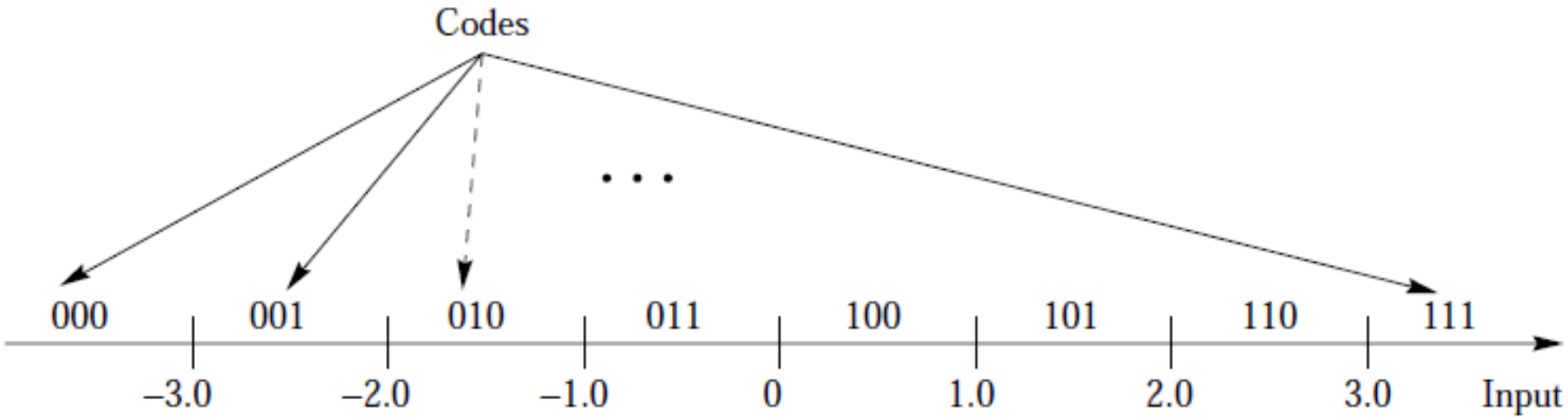
Quantizer

The quantizer consists of two *mappings*:

- Encoder:
  - Divides the range of the outputs of the source into a number of *intervals*.
  - Assigns a *codeword* for every interval.
  - All source outputs that fall in the same interval will be represented by the same codeword.
- When the input is analog, this can also be called A/D converter.

Example

- Divides the range $[-\infty, \infty]$ into 8 intervals
- All values that fall in the same interval e.g. $(-3.0, -2.0]$ will be assigned the codeword 001
- All values below –3.0 will be assigned the codeword 000



Quantizer

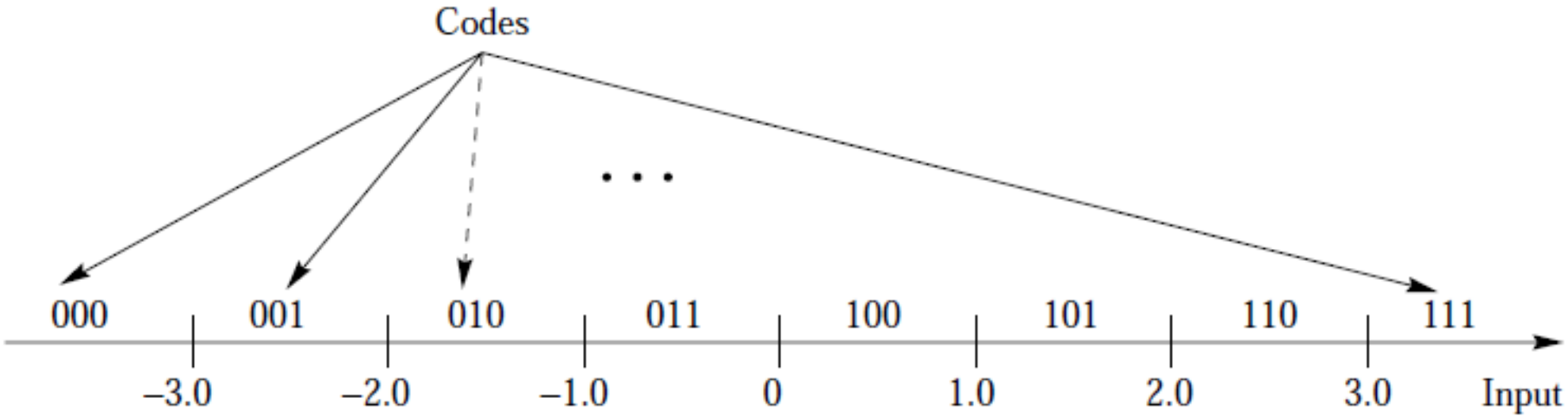
The quantizer consists of two *mappings*:

- Decoder:
  - *Reconstructs* the source output from the input *codewords*.
  - Each interval is represented by one value, called *reconstruction level*, and the codeword is replaced by this value.
- When the reconstruction is analog, this can also be called D/A converter.

Example

- *Codebook* of codewords and corresponding values

Input Codes	Output
000	−3.5
001	−2.5
010	−1.5
011	−0.5
100	0.5
101	1.5
110	2.5
111	3.5



Example

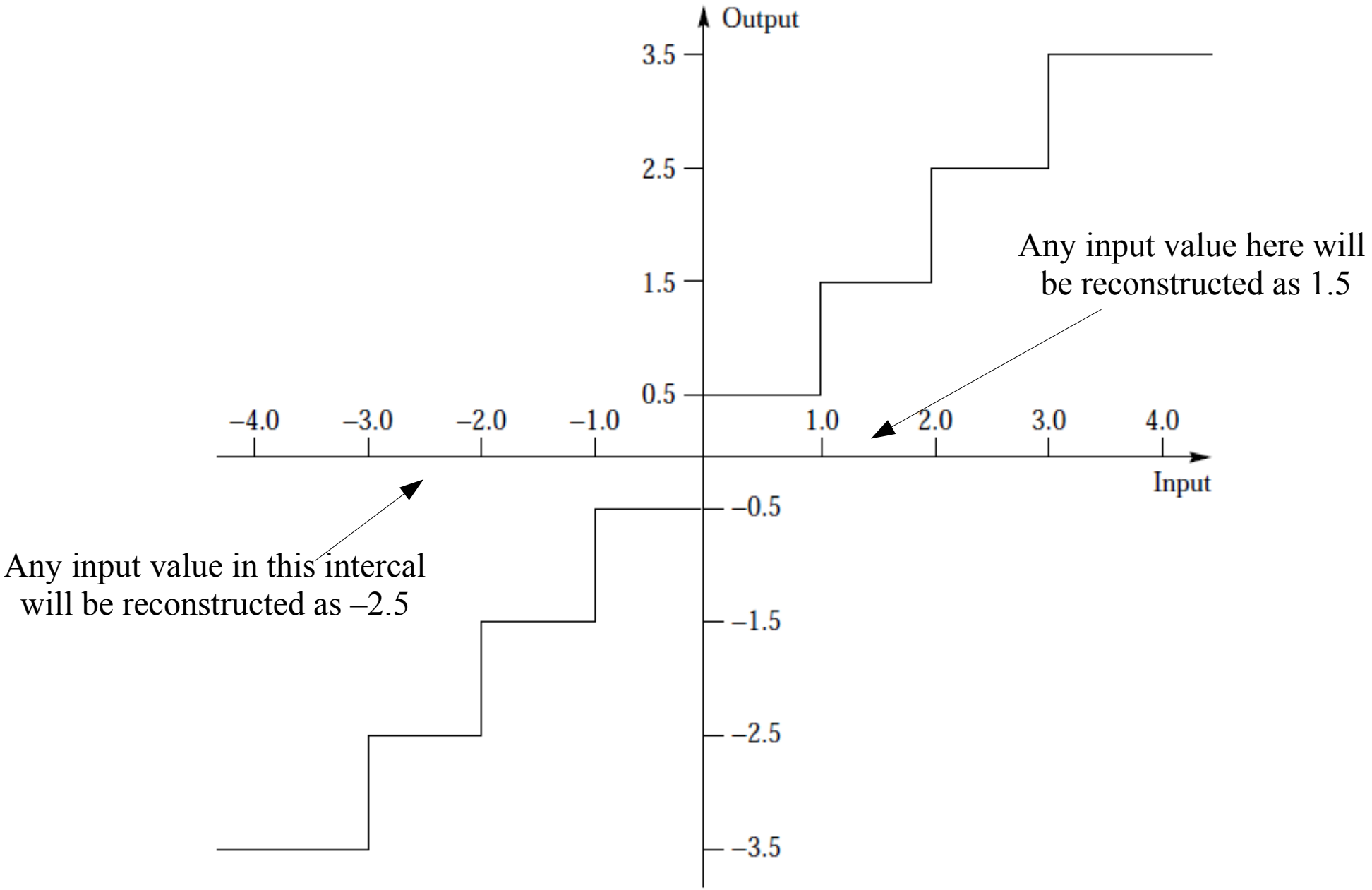


FIGURE 9. 3 **Quantizer input-output map.**

Quantizer Design

- Let the *decision boundaries* b_i for $i = 0, \ldots, M$ and the *reconstruction levels* y_i for $i = 1, \ldots, M$ and the *quantization* operation Q such that:

$$Q(x) = y_i \text{ iff } b_{i-1} < x \leqslant b_i$$

- The *mean-squared quantization error* is:

$$\begin{aligned}\sigma_q^2 &= \int_{-\infty}^{\infty} \left(x - Q(x)\right)^2 f_X(x)\, dx \\ &= \sum_{i=1}^{M} \int_{b_{i-1}}^{b_i} \left(x - y_i\right)^2 f_X(x)\, dx\end{aligned}$$

where $f_X(x)$ is the *pdf* of the source output x.

- The *rate* or expected codeword length is:

$$R = \sum_{i=1}^{M} l_i \int_{b_{i-1}}^{b_i} f_X(x)\, dx$$

where l_i is the length of the codeword for level y_i.

Quantizer Design

- To design a quantizer we:
  - Need to decide the *boundaries* of the intervals: if we want M output codewords, we need to specify $M+1$ *boundary* values b_i
  - Need to decide the *reconstruction levels* for each interval
  - Need to decide the *codeword* for each interval
- The criteria for these decisions is to minimize the *mean squared quantization error* given a *rate* upper limit, or minimize the *rate* given a maximum allowable *distortion*.

$$R \leqslant R^* \quad \text{or} \quad \sigma_q^2 \leqslant D^*$$

Uniform Scalar Quantization

- Partitions the input into *equally spaced* intervals, except possibly at the two outer intervals
  - The *reconstruction level* is taken to be the mid-point of the interval
  - Δ is the length of each interval, called *step size*
- Two types:
  - Midrise quantizers have an *even* number of levels
  - Midtread quantizers have an *odd* number of levels, including *zero*.

Midrise vs. Midtread

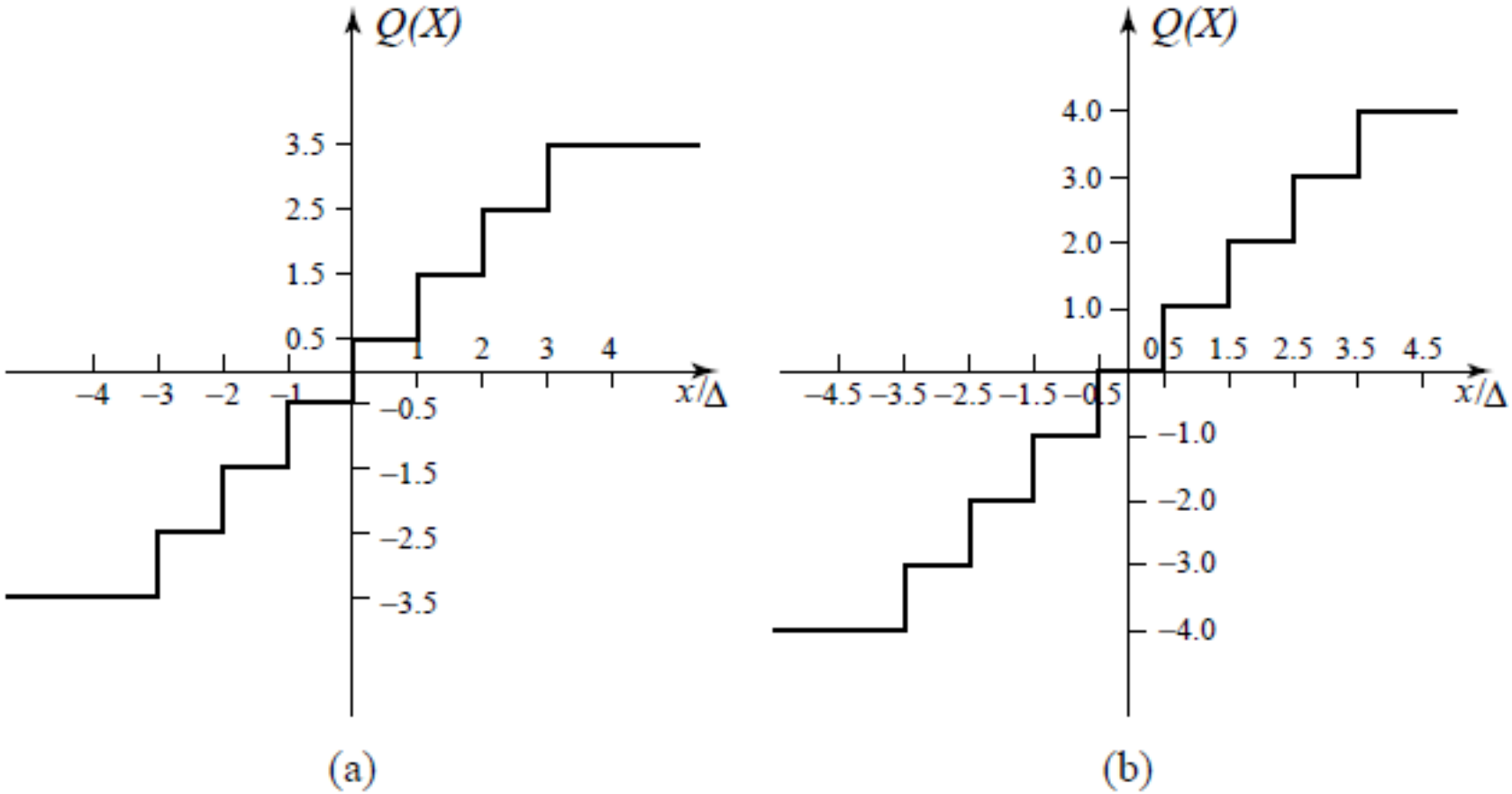


Fig. 8.2: Uniform Scalar Quantizers: (a) Midrise, (b) Midtread.

$$\Delta = 1$$

Uniform Midrise Quantizer

- Assume we have a source that outputs values *uniformly* in the range $[-X_{\max}, X_{\max}]$ i.e. any value x in this interval is equally likely
- We need to divide this interval into M equally spaced intervals

$$\Delta = \frac{2X_{\max}}{M}$$

- The *boundary* values b_i for positive x for $i = 1 \ldots M/2$:

$$b_i = i\Delta$$

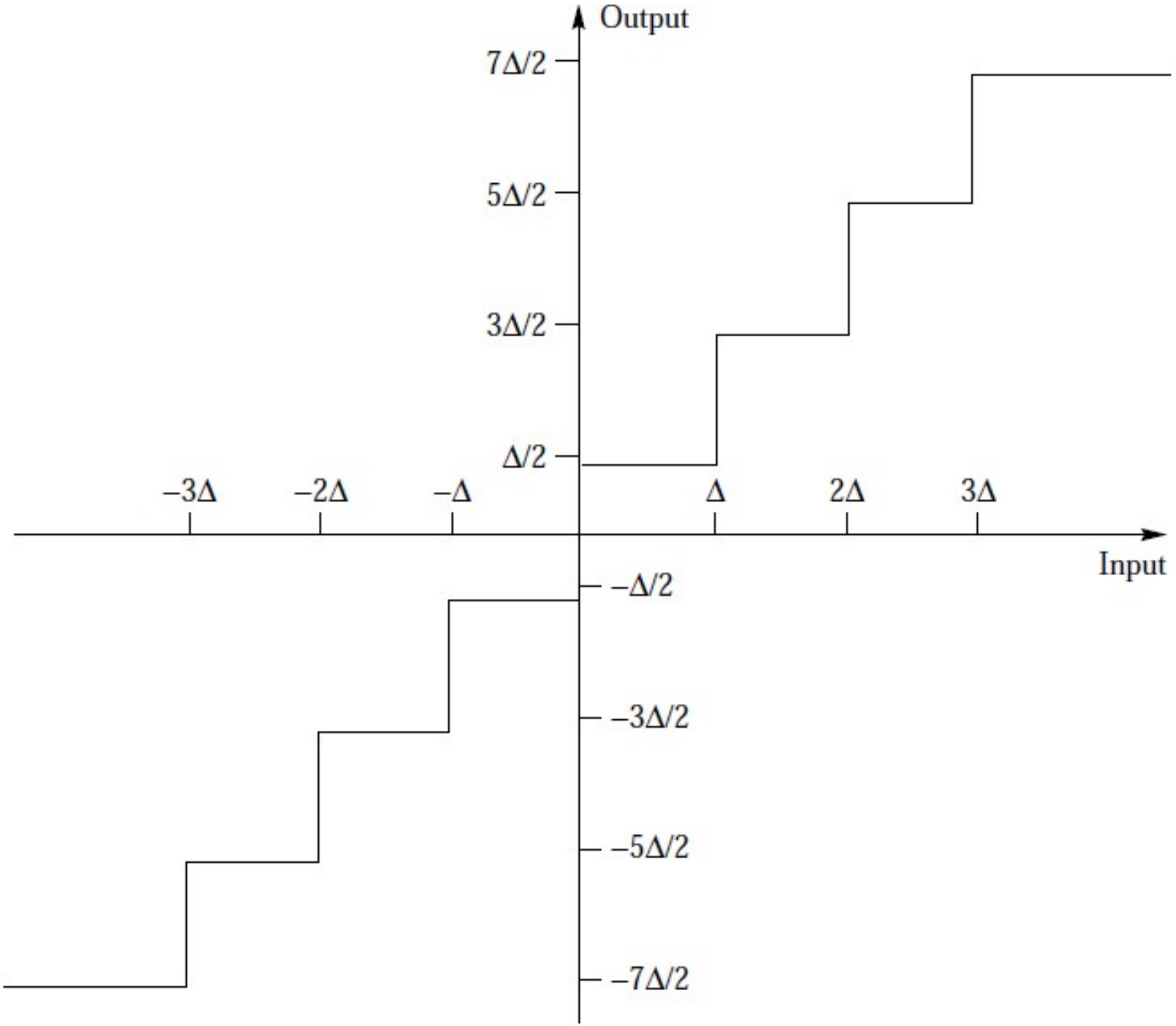
- The *reconstruction* levels y_i for positive x for $i = 1 \ldots M/2$:

$$y_i = i\Delta - \frac{\Delta}{2}$$

- The *distortion* is twice that for the positive part:

$$\sigma_q^2 = 2\sum_i \int_{(i-1)\Delta}^{i\Delta} \left(x - \frac{2i-1}{2}\Delta\right)^2 \frac{1}{2X_{\max}} dx = \frac{\Delta^2}{12}$$

Uniform Midrise Quantizer



Uniform Midrise Quantizer

- The variance of the uniformly distributed input values x:

$$\sigma_s^2 = \frac{\left(2X_{\max}\right)^2}{12}$$

- Signal-to-Noise Ratio

$$\begin{aligned}
\text{SNR(dB)} &= 10\log_{10}\left(\frac{\sigma_s^2}{\sigma_q^2}\right) \\
&= 10\log_{10}\left(\frac{(2X_{\max})^2}{12}\cdot\frac{12}{\Delta^2}\right) \\
&= 10\log_{10}\left(\frac{(2X_{\max})^2}{12}\frac{12}{(\frac{2X_{\max}}{M})^2}\right) \\
&= 10\log_{10}(M^2) \\
&= 20\log_{10}(2^n) \\
&= 6.02n\,\text{dB}.
\end{aligned}$$

where n is the number of bits for each codeword

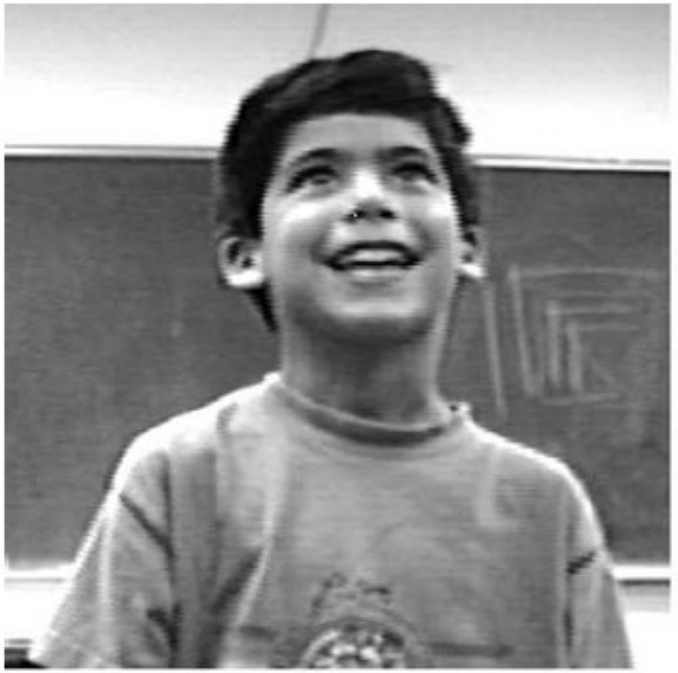
- This means that for every 1 extra bit in the codeword length, we get 6dB of increase in the SNR.

Example

- Quantizing pixel values: assume we have an input *grayscale* image with 8 bits/pixel
- We want to quantize these values using 1 bit/pixel:
  - $M = 2 \rightarrow \Delta = 256/2 = 128$
  - pixels [0, 127] will map to pixel value 64
  - pixels [128, 255] will map to pixel value 196
- We want to quantize using 2 bits/pixel
  - $M = 4 \rightarrow \Delta = 256/2 = 64$
  - pixels [0, 63] will map to pixel value 32
  - … etc

Example

Original Image



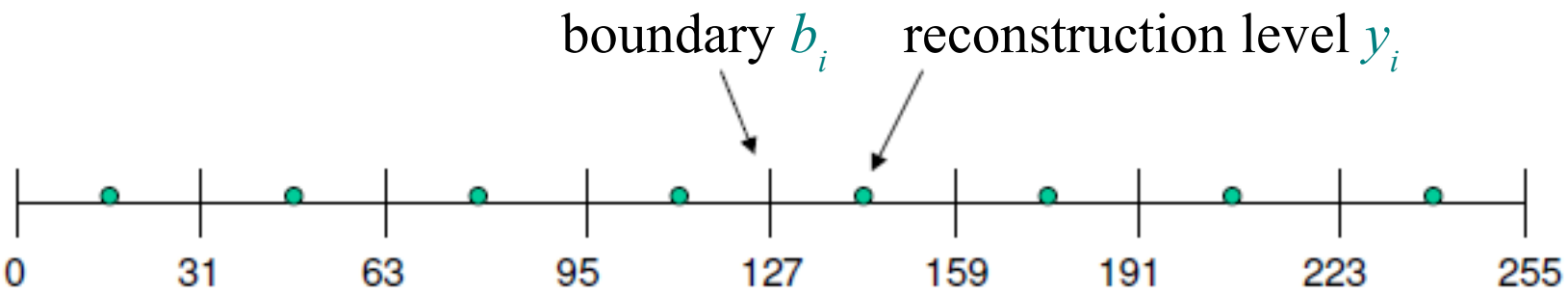
1 bit/pixel quantizer

2 bits/pixel quantizer

3 bits/pixel quantizer

Another Example

- 3-bit quantization → 8 levels



Codebook

Index	0	1	2	3	4	5	6	7
Codeword	16	47	79	111	143	175	207	239

Another Example

- Encoder:

input	0-31	32-63	64-95	96-127	128-159	160-191	192-223	224-255
code	000	001	010	011	100	101	110	111

- Decoder:

code	000	001	010	011	100	101	110	111
output	16	47	79	111	143	175	207	239

- Rate = 3 bits/pixel
- Compression ratio = 8 / 3

Improving Rate

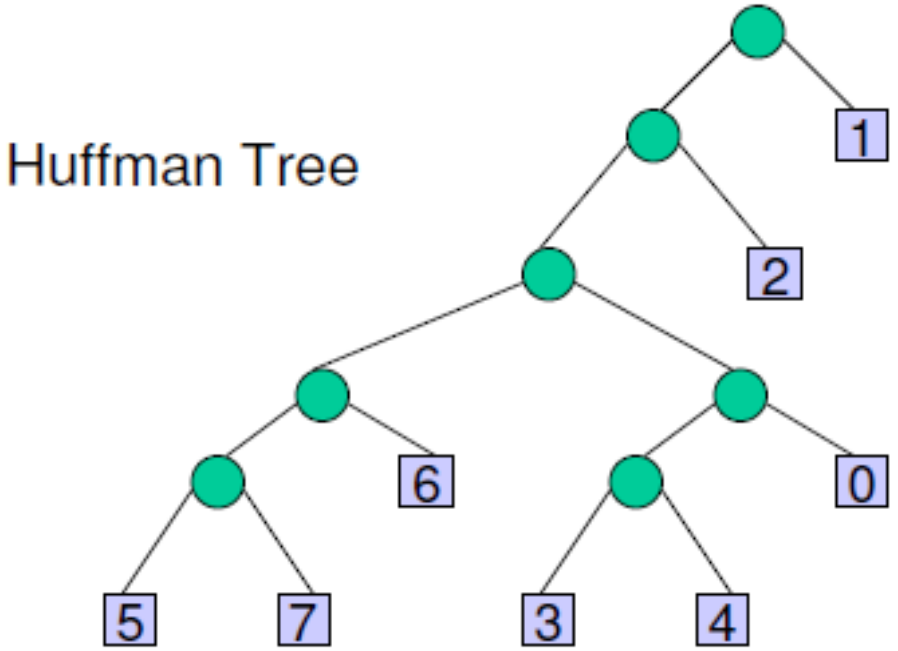
- If we know the probability of each input value, e.g. pixel, we can use *variable-length* coding to encode the output reconstruction levels
- Instead of assigning the same number of bits for each level, we assign *fewer* bits to *more frequent* levels
- Can use Huffman Coding or Arithmetic Coding for that

Example

- 512x512 image = 256K pixels
- Assume we have the following *histogram*:

index	0	1	2	3	4	5	6	7
input	0-31	32-63	64-95	96-127	128-159	160-191	192-223	224-255
frequency	25,000	100,000	90,000	10,000	10,000	10,000	18,000	9,144

- We can construct a Huffman Tree for the codes and achieve a lower rate:

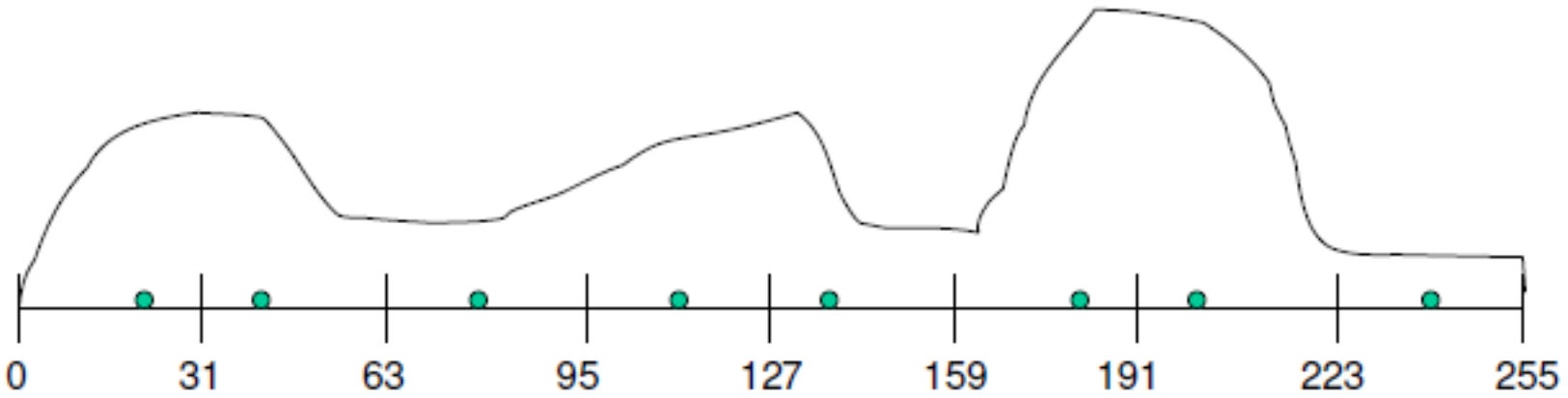


Improving Distortion

- If we know the *pdf* of the input values x, instead of just choosing the reconstruction level y as the midpoint of the interval, we can choose the *optimal* value that *minimizes* the *mean-squared quantization error*.

- Let $[L_i, R_i)$ be the i^{th} interval, the *optimal* reconstruction level y_i is *mean* (*expectation*) of x in the interval:

$$y_i = \int_{L_i}^{R_i} x f_X(x) dx$$



Example

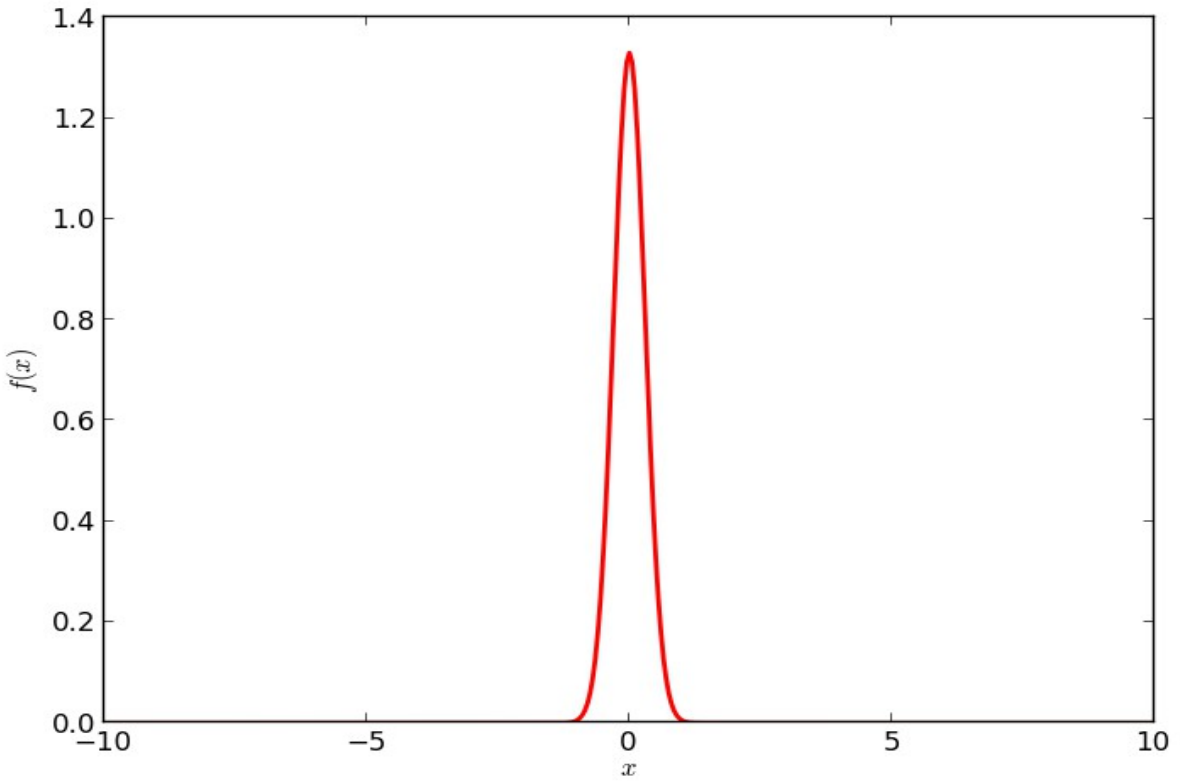
- Assume we have the following distribution for the pixels in the interval [8, 15]:

pixel value	8	9	10	11	12	13	14	15
frequency	100	100	100	40	30	20	10	0

- The mid-point:
  - $y = 11$
  - $\text{MSE} = \frac{130 \times 1^2 + 120 \times 2^2 + 1100 \times 3^2}{400} = 40$
- The mean:
  - $y = \frac{8 \times 100 + 9 \times 100 + 10 \times 100 + 11 \times 40 + 12 \times 30 + 13 \times 20 + 14 \times 10 + 15 \times 0}{400} = 10$
  - $\text{MSE} = \frac{140 \times 1^2 + 130 \times 2^2 + 20 \times 3^2 + 10 \times 4^2}{400} = 25$

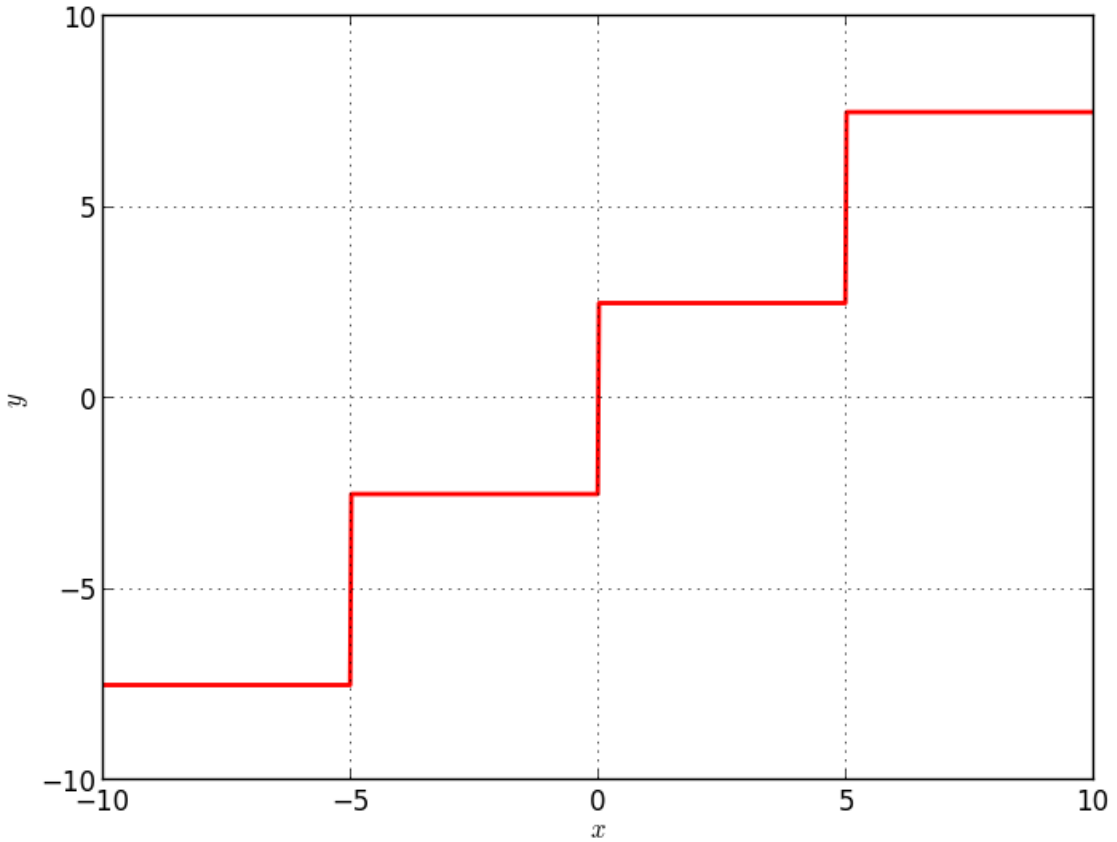
Nonuniform Scalar Quantization

- When the input is not uniformly distributed, uniform quantization is not efficient
- For example, assume we have a source that emits values in the range [– 10, 10], and 95% of the time the values are within [–1, 1]



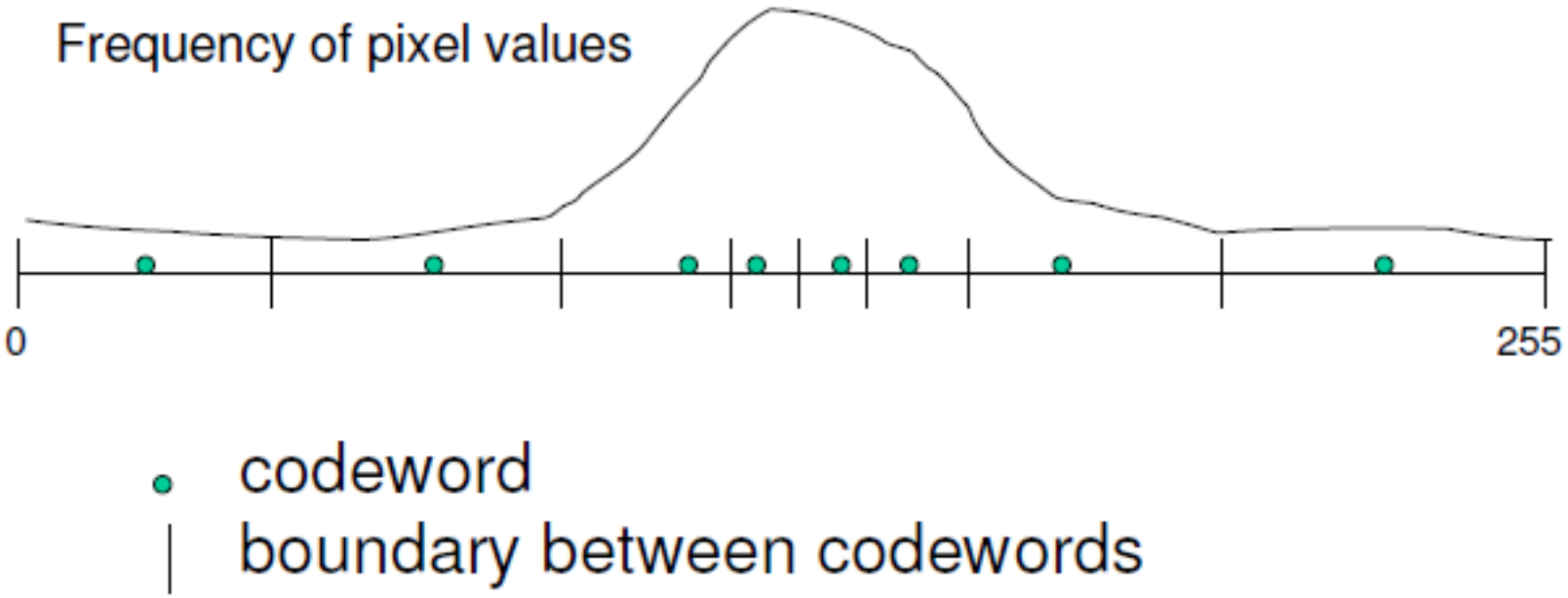
Nonuniform Scalar Quantization

- Using a 4 level quantizer: 95% of the time, the *minimum* quantization error will be 1.5
- It would be better to have *more* levels in the range that is *more probable* to reduce the quantization error or improve *approximation*, at the cost of more error in less probable area



Nonuniform Scalar Quantization

More levels in the dense area



- How do we find the *decision boundaries* b_i and *quantization levels* y_i?

Lloyd-Max Quantizer

- The distortion to be *minimized* is defined by:

$$\sigma_q^2 = \sum_{i=1}^{M} \int_{b_{i-1}}^{b_i} (x - y_i)^2 f_X(x)\, dx$$

- To find y_i we can set the derivative with respect to y_i to 0:

$$y_i = \frac{\int_{b_{i-1}}^{b_i} x\, f_X(x)\, dx}{\int_{b_{i-1}}^{b_i} f_X(x)\, dx}$$

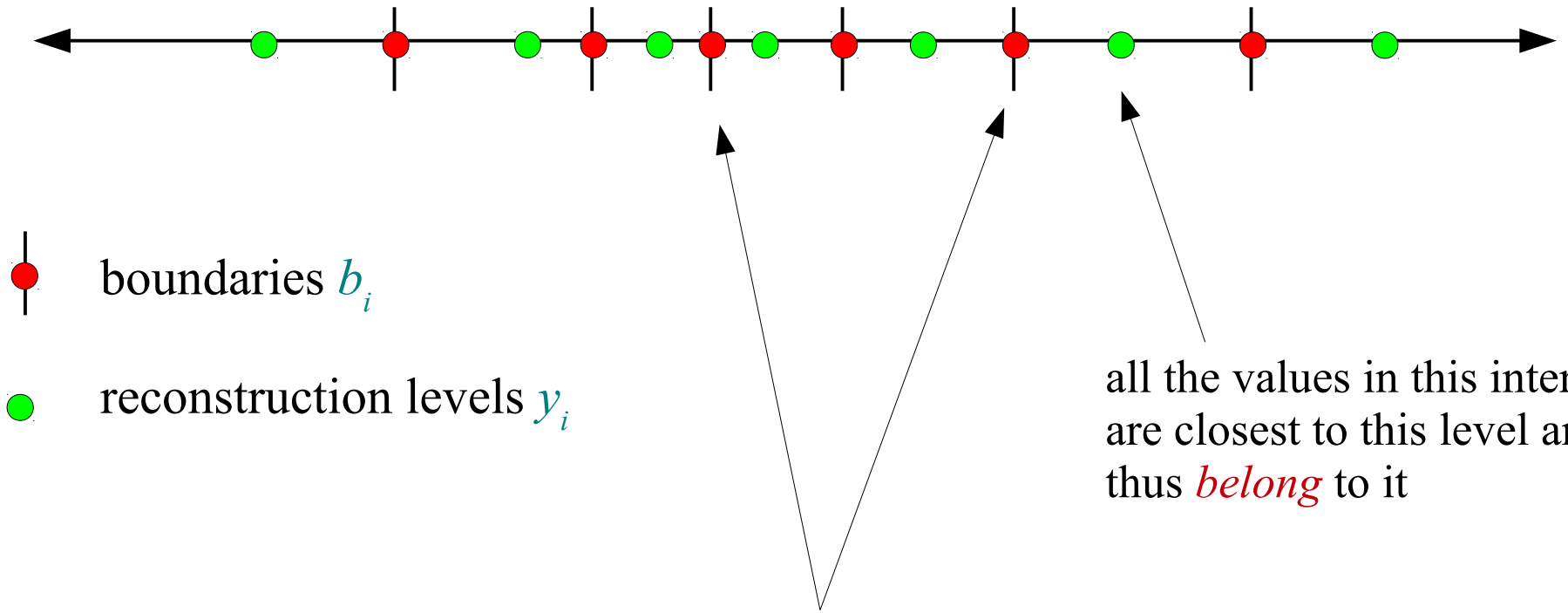
  i.e. it's the *centroid* (or mean) in that interval

- To find b_i we can set the derivative with respect to b_i to 0:

$$b_i = \frac{y_{i+1} + y_i}{2}$$

  i.e. it's the *mid-point* between every two levels. This means every point x *belongs* to the *nearest* level y_i.

Example


boundaries b_i
reconstruction levels y_i
all the values in this interval are closest to this level and thus *belong* to it
decision boundaries are halfway between reconstruction levels

Lloyd-Max Quantizer

- The problem now is that if we know the y_i's we can compute the b_i's, and if we know the b_i's we can compute the y_i's
- How do we find both simultaneously?
- Iterative solution:
  - Start with an initial guess for the reconstruction levels $y_i^{(0)}$
  - Repeat:
    - Compute the decision boundaries $b_i^{(t)}$ using: $b_i^{(t)} = \frac{y_{i+1}^{(t-1)} + y_i^{(t-1)}}{2}$
    - Compute the new levels $y_i^{(t)}$ using: $y_i^{(t)} = \frac{\int_{b_{i-1}^{(t)}}^{b_i^{(t)}} x f_X(x) dx}{\int_{b_{i-1}^{(t)}}^{b_i^{(t)}} f_X(x) dx}$
    - Until convergence

Lloyd-Max for Images

```
Choose a small error tolerance ε > 0.
Choose start levels y_i
Compute V_i = {x: x is a pixel value
   closest to y_i}
Compute distortion D for y_i
Repeat
   Compute new levels y_i' using:
```

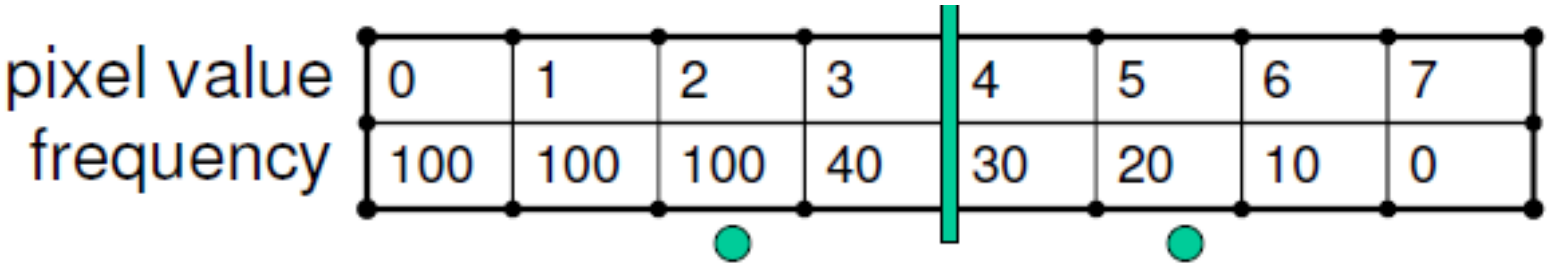
$$y_i' = \text{round}\left(\frac{\sum_{x \in V_i} x\,P(x)}{\sum_{x \in V_i} P(x)} \right)$$

```
   Compute V_i' = {x : x is a pixel value
      closest to y_i'}
   Compute distortion D' for y_i'
   if |(D - D')/D| < ε then quit
   else y_i = y_i'; V_i = V_i', D = D'
End
```

Set of all pixels belonging to y_i

$P(x)$ is the probability of pixel value x

Example



$y_0 = 2$ and $y_1 = 5$

$V_0 = \{0, 1, 2, 3\}$ and $V_1 = \{4, 5, 6, 7\}$

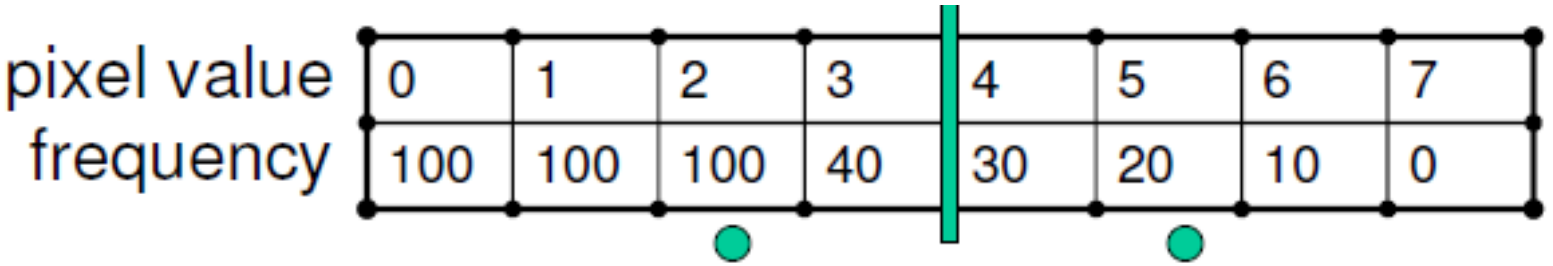
$D = 140\text{x}1 + 100\text{x}4 + 40\text{x}1 = 580$

```
Choose a small error tolerance ε > 0.
Choose start levels y_i
Compute V_i = {x: x is a pixel value
    closest to y_i}
Compute distortion D for y_i
Repeat
    Compute new levels y_i' using:
```

$$y_i' = \text{round}\left(\frac{\sum_{x \in X_i} x\,P(x)}{\sum_{x \in X_i} P(x)}\right)$$

```
    Compute V_i' = {x : x is a pixel value
        closest to y_i'}
    Compute distortion D' for y_i'
    if |(D - D')/D| < ε then quit
    else y_i = y_i'; V_i = V_i', D = D'
End
```

Example



$y_0 = 2$ and $y_1 = 5$

$V_0 = \{0, 1, 2, 3\}$ and $V_1 = \{4, 5, 6, 7\}$

$D = 140\text{x}1 + 100\text{x}4 + 40\text{x}1 = 580$

$$y_0' = \text{round}((100\text{x}0 + 100\text{x}1 + 100\text{x}2 + 40\text{x}3) / 340) = 1$$

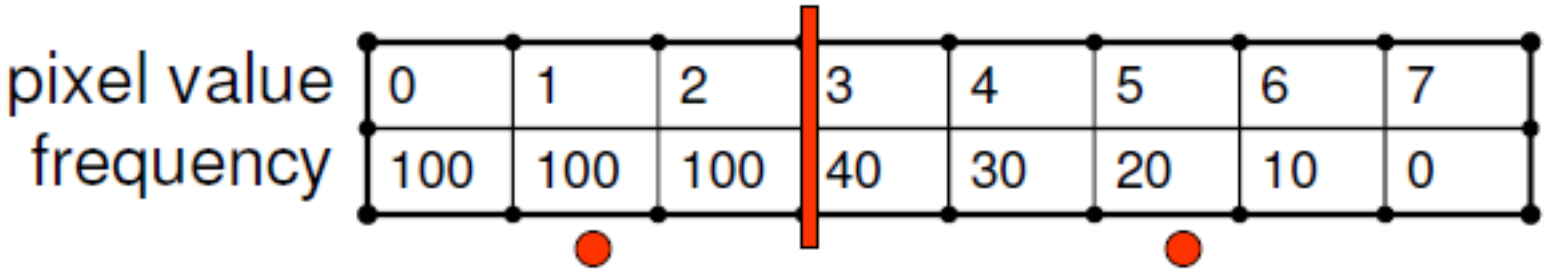
$$y_1' = \text{round}((30\text{x}4 + 20\text{x}5 + 10\text{x}6) / 60) = 5$$

```
Choose a small error tolerance ε > 0.
Choose start levels y_i
Compute V_i = {x: x is a pixel value
    closest to y_i}
Compute distortion D for y_i
Repeat
    Compute new levels y_i' using:
```

$$y_i' = \text{round}\left(\frac{\sum_{x \in X_i} x P(x)}{\sum_{x \in X_i} P(x)}\right)$$

```
    Compute V_i' = {x : x is a pixel value
        closest to y_i'}
    Compute distortion D' for y_i'
    if |(D - D')/D| < ε then quit
    else y_i = y_i'; V_i = V_i', D = D'
End
```

Example



$y_0' = 1$ and $y_1' = 5$

$V_0' = \{0, 1, 2\}$ and $V_1' = \{3, 4, 5, 6, 7\}$

$D = 200\text{x}1 + 40\text{x}1 + 40\text{x}4 = 400$

$|D - D'| / D = (580 - 400) / 400 = 0.31$

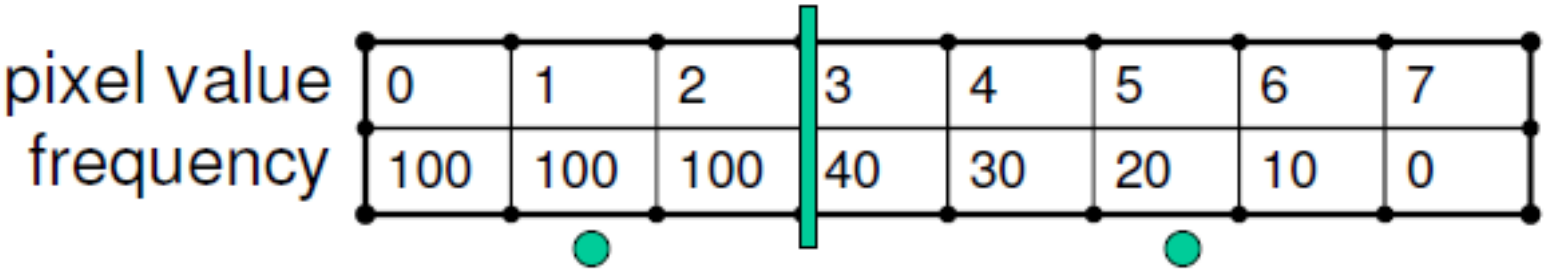
$y = y', \; V = V', D = D'$

```
Choose a small error tolerance ε > 0.
Choose start levels y_i
Compute V_i = {x: x is a pixel value
    closest to y_i}
Compute distortion D for y_i
Repeat
    Compute new levels y_i' using:
```

$$y_i' = \text{round}\left(\frac{\sum_{x \in X_i} x P(x)}{\sum_{x \in X_i} P(x)}\right)$$

```
    Compute V_i' = {x : x is a pixel value
        closest to y_i'}
    Compute distortion D' for y_i'
    if |(D - D')/D| < ε then quit
    else y_i = y_i'; V_i = V_i', D = D'
End
```

Example



$y_0 = 1$ and $y_1 = 5$

$V_0 = \{0, 1, 2\}$ and $V_1 = \{3, 4, 5, 6, 7\}$

$D = 400$

$y_0' = \text{round}((100\text{x}0 + 100\text{x}1 + 100\text{x}2) / 300) = 1$

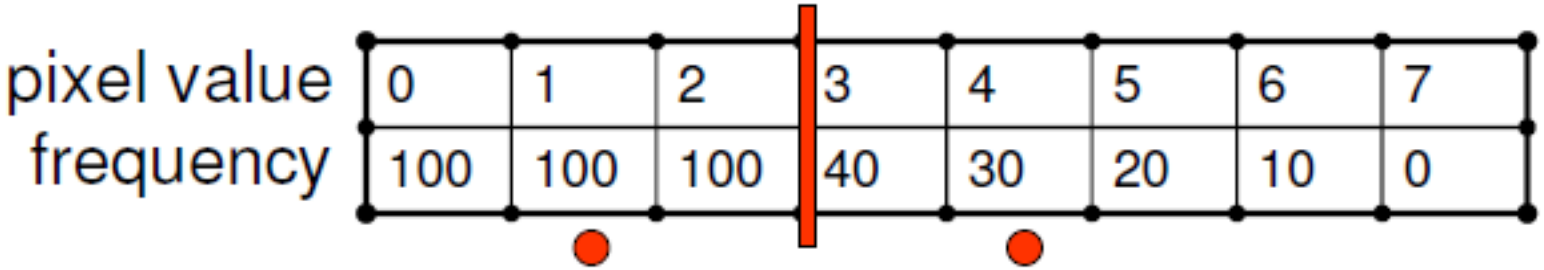
$y_1' = \text{round}((40\text{x}3 + 30\text{x}4 + 20\text{x}5 + 10\text{x}6) / 100) = 4$

```
Choose a small error tolerance ε > 0.
Choose start levels y_i
Compute V_i = {x: x is a pixel value
    closest to y_i}
Compute distortion D for y_i
Repeat
    Compute new levels y_i' using:
```

$$y_i' = \text{round}\left(\frac{\sum_{x \in X_i} x P(x)}{\sum_{x \in X_i} P(x)}\right)$$

```
    Compute V_i' = {x : x is a pixel value
        closest to y_i'}
    Compute distortion D' for y_i'
    if |(D - D')/D| < ε then quit
    else y_i = y_i'; V_i = V_i', D = D'
End
```

Example



$y_0' = 1$ and $y_1' = 4$

$V_0' = \{0, 1, 2\}$ and $V_1' = \{3, 4, 5, 6, 7\}$

$D = 200\text{x}1 + 60\text{x}1 + 10\text{x}4 = 300$

$|D - D'| / D = (400 - 300) / 300 = 0.17$

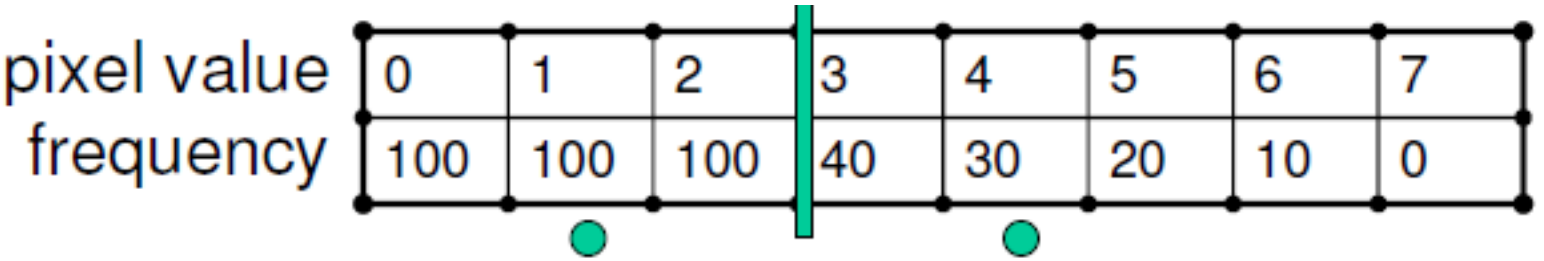
$y = y', \; V = V', D = D'$

```
Choose a small error tolerance ε > 0.
Choose start levels y_i
Compute V_i = {x: x is a pixel value
    closest to y_i}
Compute distortion D for y_i
Repeat
    Compute new levels y_i' using:
```

$$y_i' = \text{round}\left(\frac{\sum_{x \in X_i} x P(x)}{\sum_{x \in X_i} P(x)}\right)$$

```
    Compute V_i' = {x : x is a pixel value
        closest to y_i'}
    Compute distortion D' for y_i'
    if |(D - D')/D| < ε then quit
    else y_i = y_i'; V_i = V_i', D = D'
End
```

Example



$y_0 = 1$ and $y_1 = 4$

$V_0 = \{0, 1, 2\}$ and $V_1 = \{3, 4, 5, 6, 7\}$

$D = 300$

$$y_0' = \text{round}((100x0 + 100x1 + 100x2) / 300) = 1$$

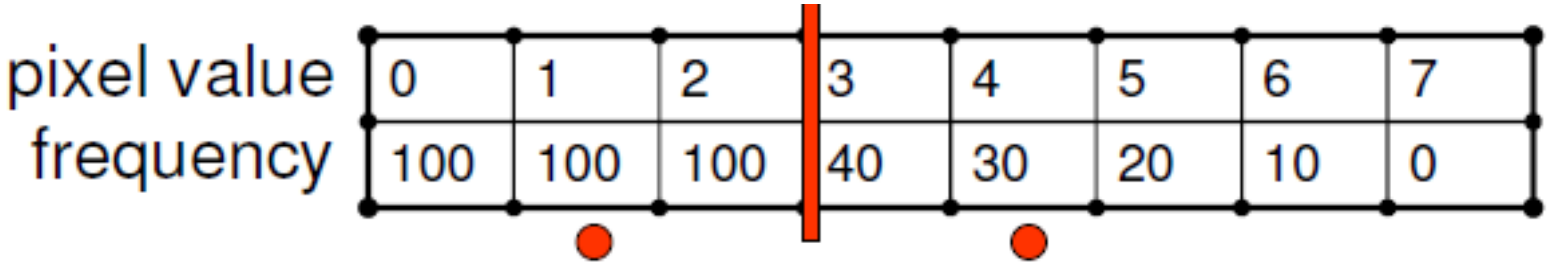
$$y_1' = \text{round}((40x3 + 30x4 + 20x5 + 10x6) / 100) = 4$$

```
Choose a small error tolerance ε > 0.
Choose start levels y_i
Compute V_i = {x: x is a pixel value
    closest to y_i}
Compute distortion D for y_i
Repeat
    Compute new levels y_i' using:
```

$$y_i' = \text{round}\left(\frac{\sum_{x \in X_i} x P(x)}{\sum_{x \in X_i} P(x)}\right)$$

```
    Compute V_i' = {x : x is a pixel value
        closest to y_i'}
    Compute distortion D' for y_i'
    if |(D - D')/D| < ε then quit
    else y_i = y_i'; V_i = V_i', D = D'
End
```

Example



$y_0' = 1$ and $y_1' = 4$

$V_0' = \{0, 1, 2\}$ and $V_1' = \{3, 4, 5, 6, 7\}$

$D = 200\text{x}1 + 60\text{x}1 + 10\text{x}4 = 300$

$|D - D'| / D = (300 - 300) / 300 = 0$

Quit with: $y_0 = 1$ and $y_1 = 4$

```
Choose a small error tolerance ε > 0.
Choose start levels y_i
Compute V_i = {x: x is a pixel value
    closest to y_i}
Compute distortion D for y_i
Repeat
    Compute new levels y_i' using:
```

$$y_i' = \text{round}\left(\frac{\sum_{x \in X_i} x P(x)}{\sum_{x \in X_i} P(x)}\right)$$

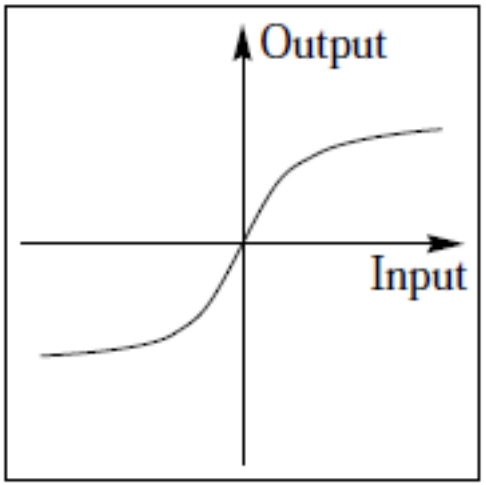
```
    Compute V_i' = {x : x is a pixel value
        closest to y_i'}
    Compute distortion D' for y_i'
    if |(D - D')/D| < ε then quit
    else y_i = y_i'; V_i = V_i', D = D'
End
```

Companding

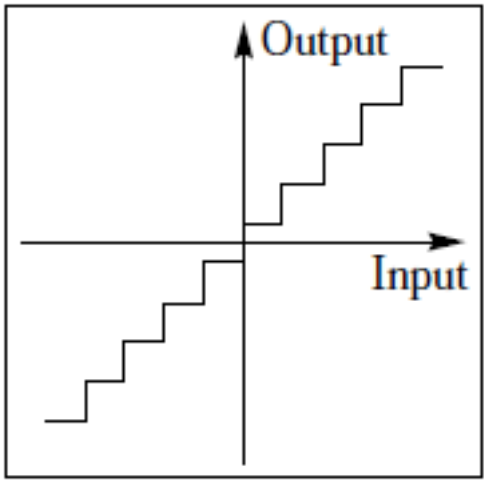
- Another way to achieve *nonuniform* quantization
- Creates more quantization levels (smaller intervals) at values closer to zero
- Useful and mostly used in audio compression

Companding

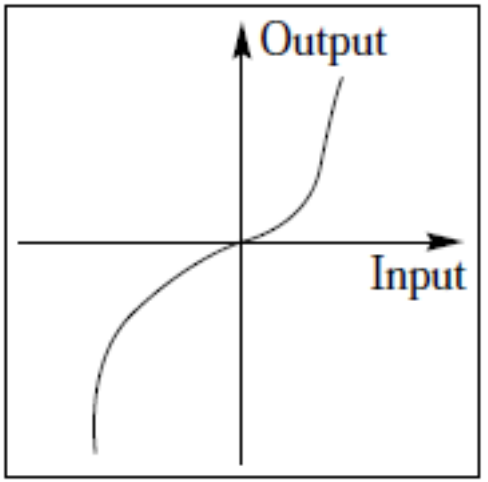
- Idea:
  - Compress signal using a non-linear function that *stretches* the high probability values close to zero and *compresses* the low probability values
  - Perform uniform quantization on the output
  - The decoder then performs the *inverse* of the compression (*expander*) and reconstructs the original input



Compressor



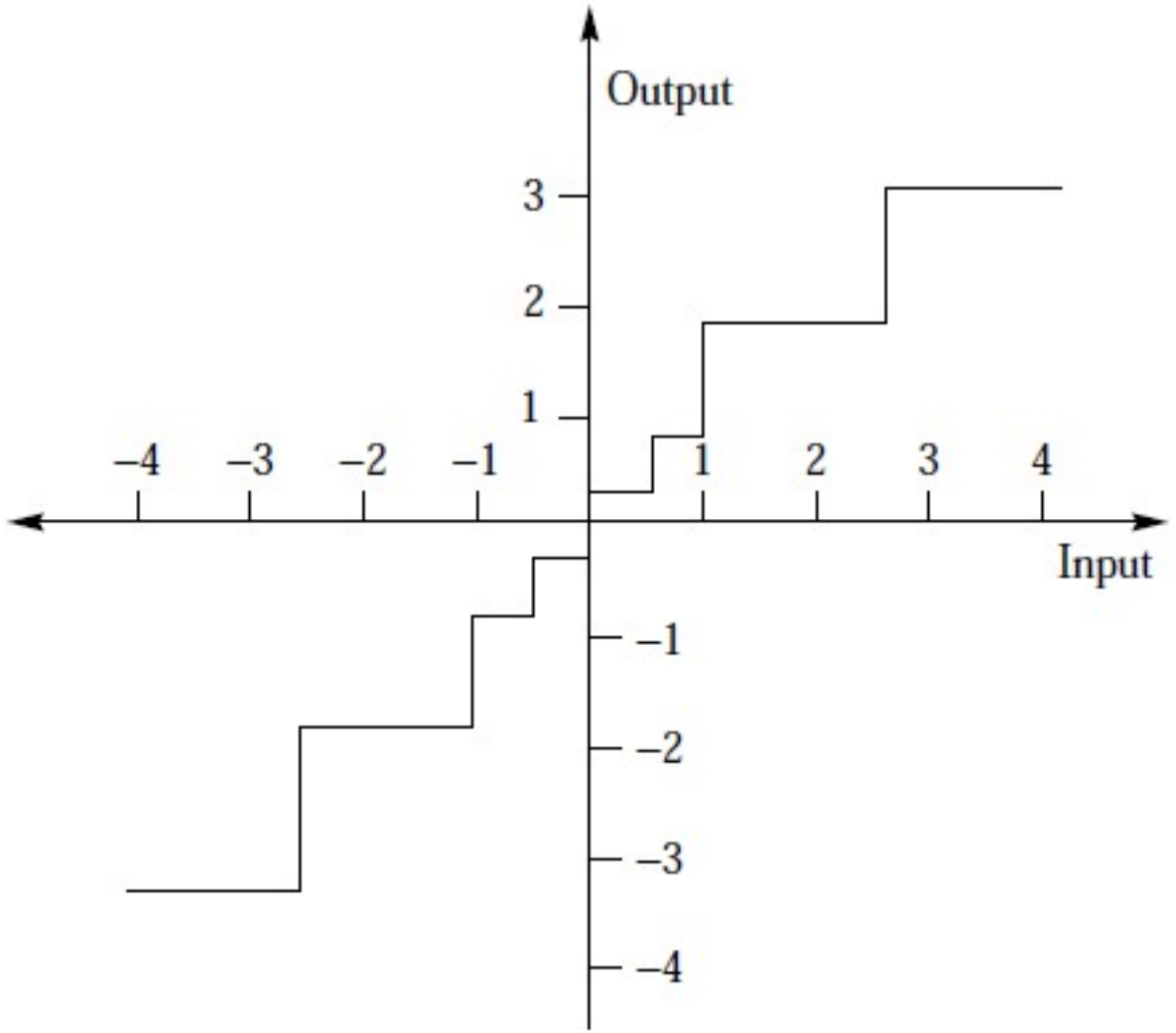
Uniform quantizer



Expander

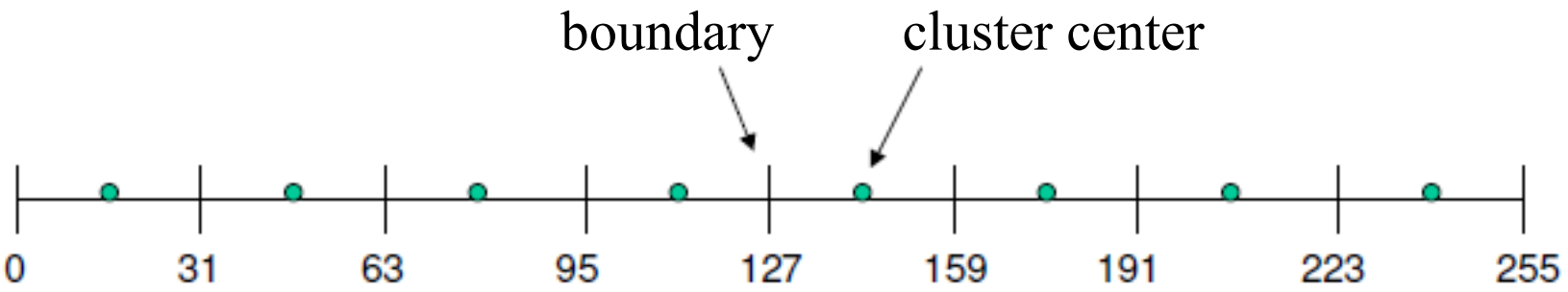
Companding

Is equivalent to:



Quantization as Clustering

- Can view the quantization process as some sort of *clustering* where:
  - the *encoder* maps every input to the *nearest cluster center* (reconstruction level) and transmits its index
  - the *decoder* replaces the index with the reconstruction level from the *codebook*



Codebook

Index	0	1	2	3	4	5	6	7
Codeword	16	47	79	111	143	175	207	239

Scalar Quantization Summary

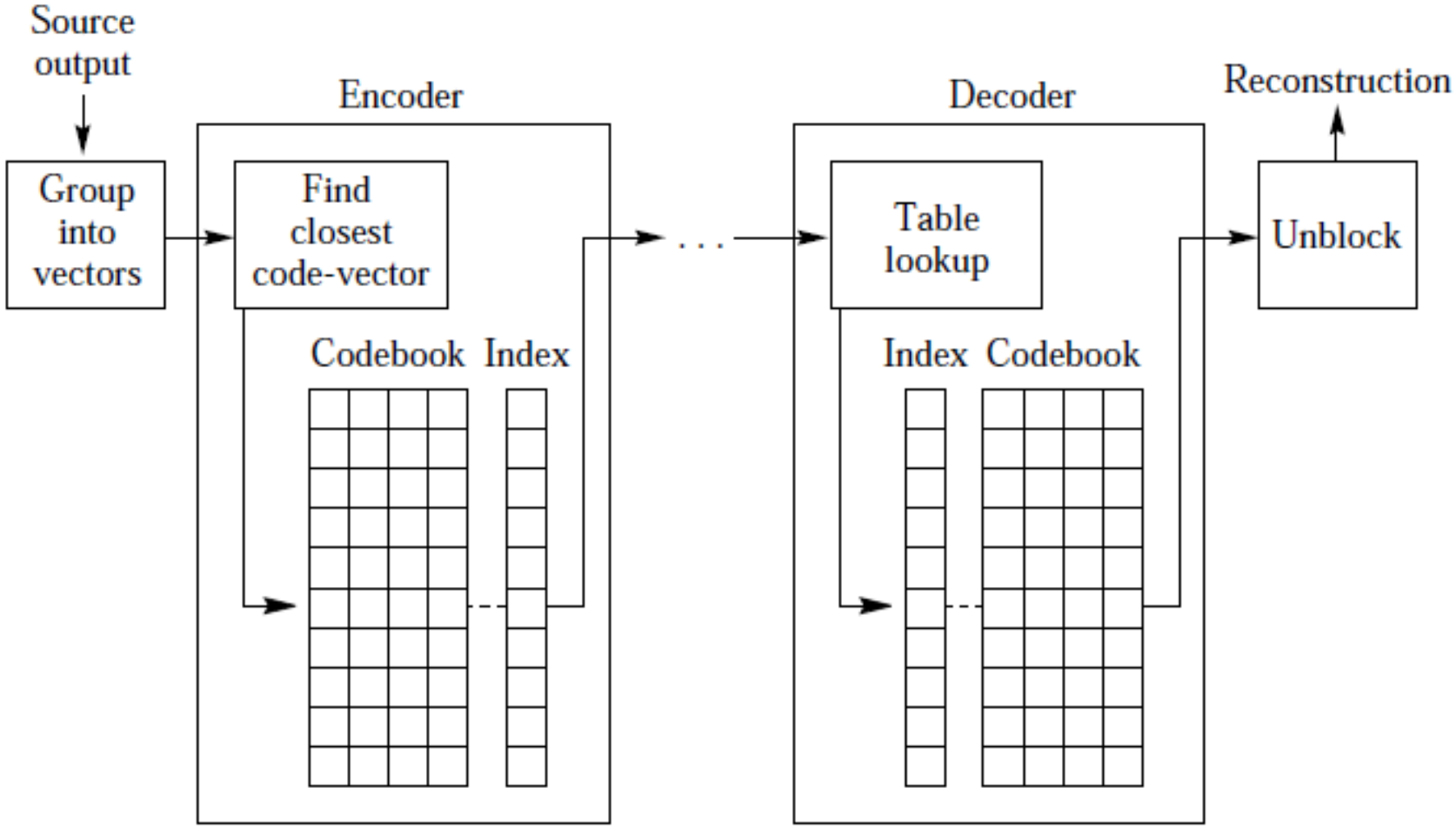
- Useful for A/D for audio compression
- Useful for image quantization
- Can be combined with lossless coding for more efficiency of coding the reconstruction level indices
- Lloyd-Max algorithm works very well in practice, will be generalized later
- Can view it as a clustering process

Vector Quantization (VQ)

- Shannon's theory proved that working on *vectors* (blocks) of samples is more *efficient* than working on *individual* samples, as we did with Huffman and Arithmetic Coding
- We can achieve *lower distortion* and *rate* using vectors of inputs
- Scalar Quantization works on one symbol at a time and maps all values that lie in an interval to one reconstruction level
- Vector Quantization works on *L-dimensional vectors* formed by concatenating L consecutive symbols
- Vectors that lie within a region in the *L-dimensional* space are mapped to the same reconstruction vector, called *code vectors* or *codewords*
- The collection of code vectors is called a *codebook*

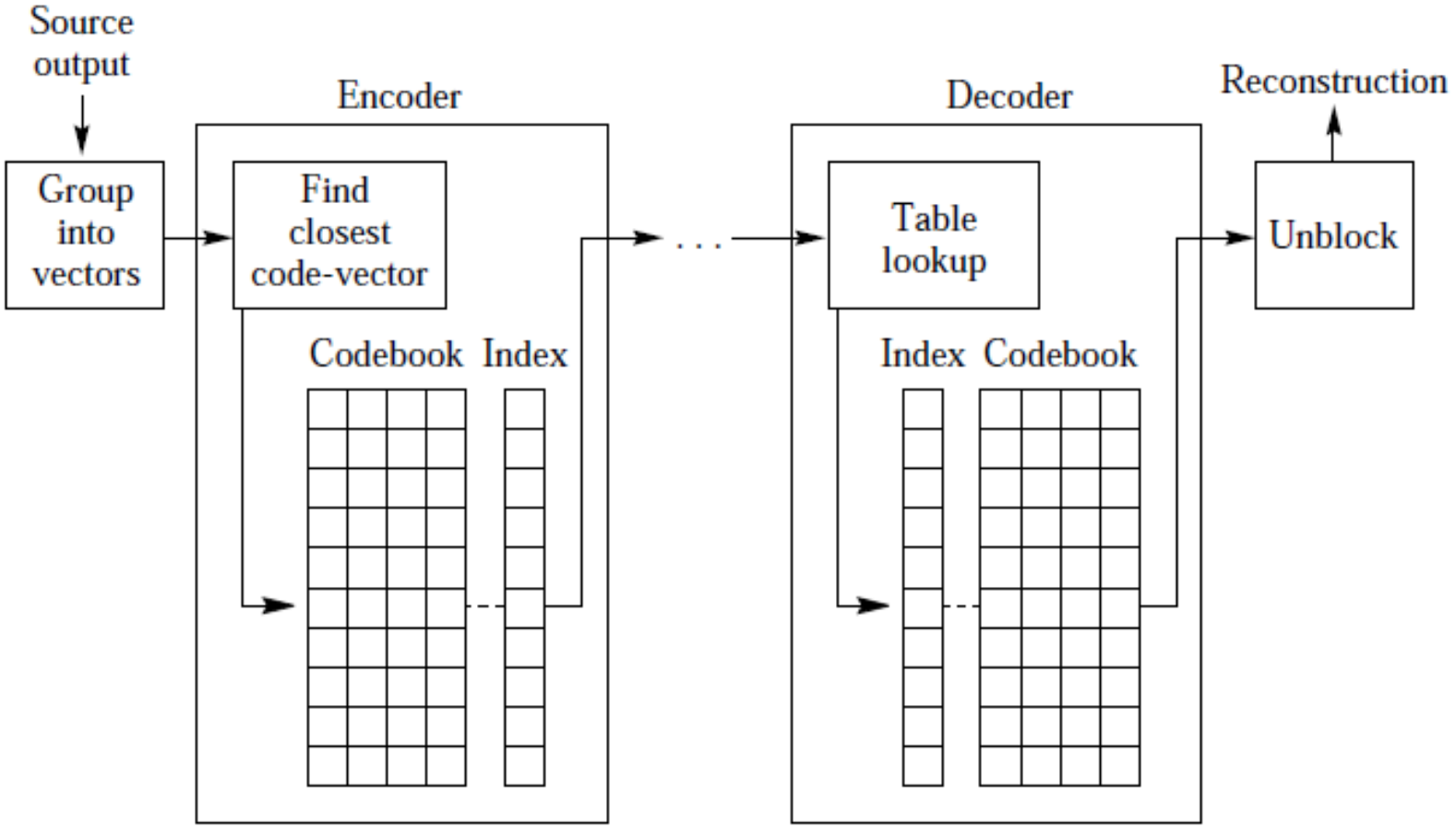
Vector Quantization

- *Encoder*: for every input vector, finds the closest *code vector* in the *codebook* and transmits its *index*
- *Decoder*: looks up the index in the *codebook* and outputs the corresponding *code vector*



Vector Quantization

- Examples:
  - Take L consecutive samples of audio and form a vector
  - Take L pixels in a block of the image and form a vector



Vector Quantization

- Assume we have vectors of length L i.e. L-dimensional vectors of input samples
- Assume we have a codebook with K *code vectors* each with L dimensions
- The number of bits required to represent the index of a code vector is $\lceil \log_2 K \rceil$ *bits per vector*
- So the number of *bits per sample* is $\lceil \log_2 K \rceil / L$ bits
- We will measure distortion using the MSE
- Each input vector X is closest to a code vector Y_j which means that $\| X - Y_j \|^2 \leq \| X - Y_i \|^2$ for all i where $\| X \|^2 = \sum_i x_i^2$ is the norm of vector X

Why VQ Helps?

- Assume we have an 8-level uniform scalar quantizer with *step* Δ
- All values in the same *interval* (i.e. within the decision boundaries), map to the same *reconstruction level*
- Another way we can look at this is that every value is mapped to the *closest reconstruction level* which is then output from the quantizer

Another view: points map to the closest output value

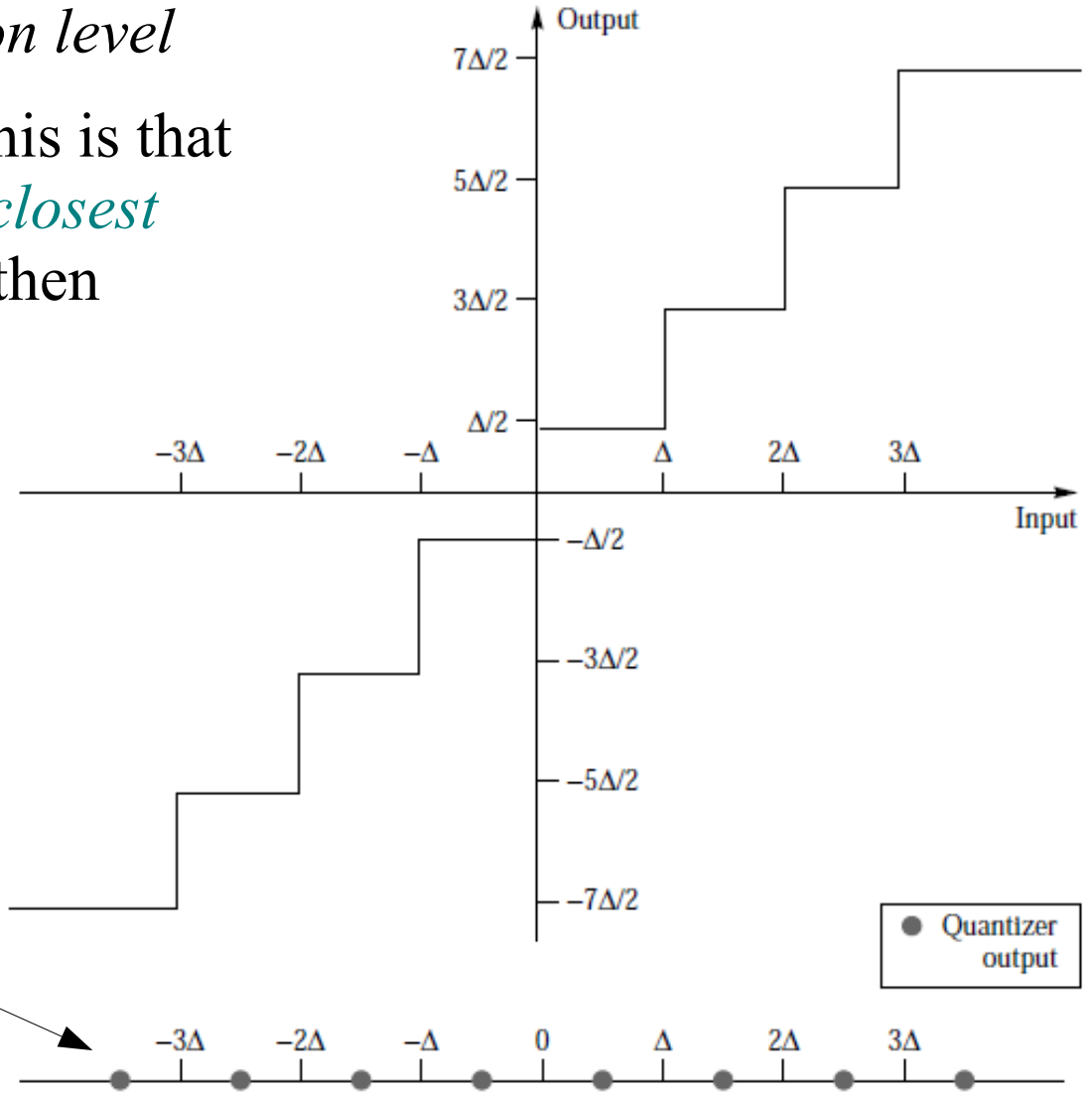


FIGURE 10. 4 Two representations of an eight-level scalar quantizer.

Why VQ Helps?

- Assume we quantize every *two* consecutive values together x_1 and x_2
- For example, this quantizer output is sent when x_1 is within the interval $[\Delta, 2\Delta]$ and x_2 is within the interval $[2\Delta, 3\Delta]$
- The *decision boundaries* in this case are all parallel to the coordinate axes and are not as *flexible*

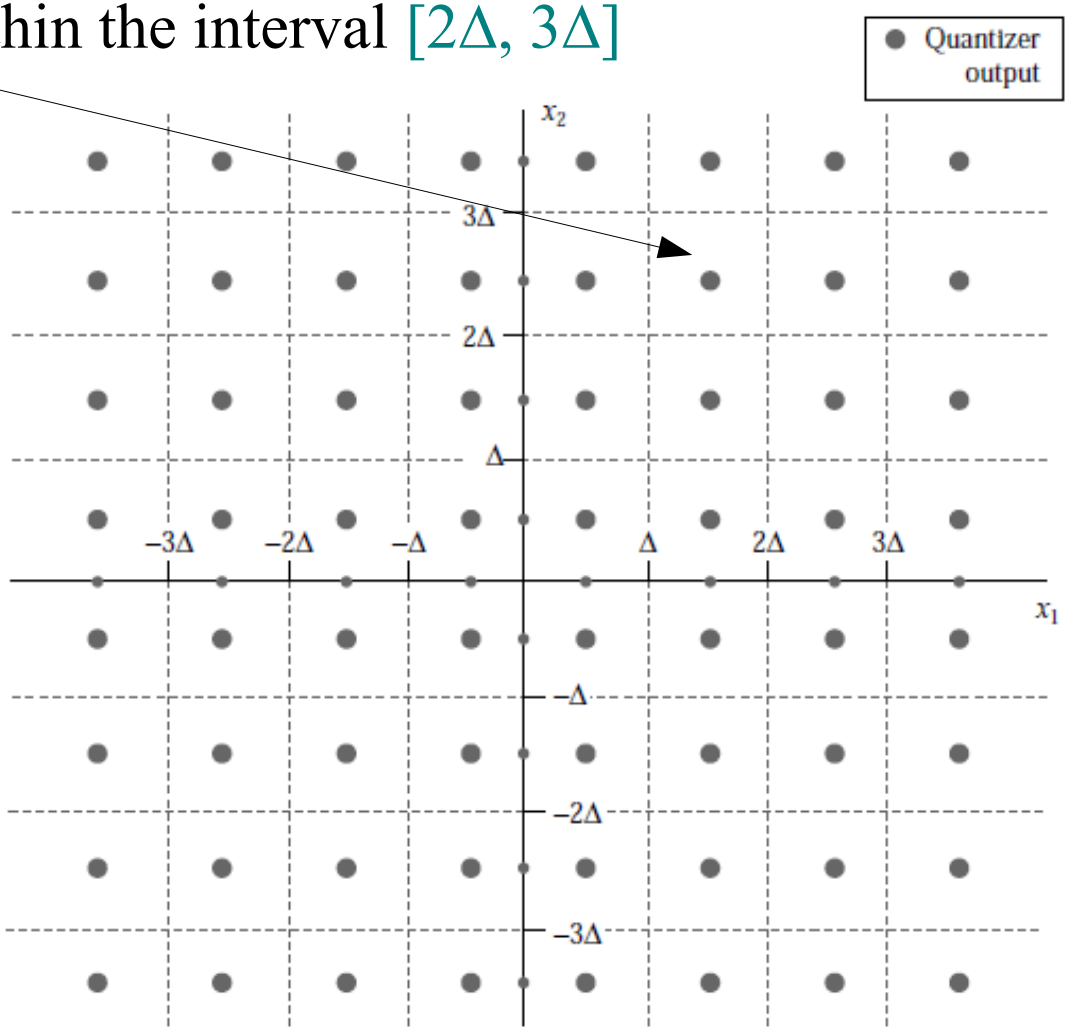


FIGURE 10. 5 Input-output map for consecutive quantization of two inputs using an eight-level scalar quantizer.

Why VQ Helps?

- Suppose we move the top-right point (that is not used that often) to the *origin*
- This modified quantizer has the same number of codewords *but* has *higher* SNR
- This is because it is now more *flexible* since the *decision boundaries* are not restricted to be parallel to the coordinat axes
- How do we design this *codebook*?

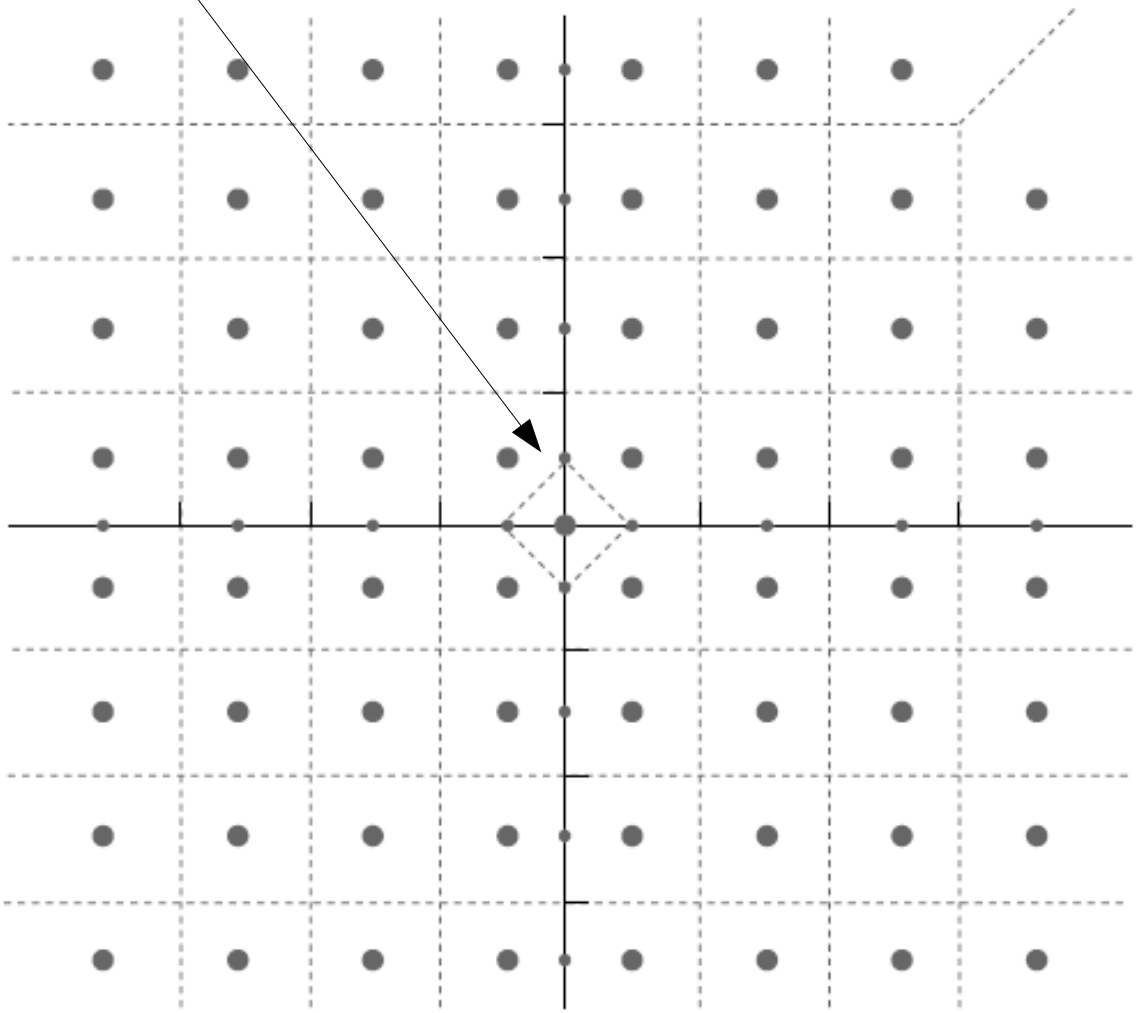


FIGURE 10. 6 Modified two-dimensional vector quantizer.

Linde-Buzo-Gray (LBG) Algorithm

- A generalization of the Lloyd-Max algorithm for higher dimensions
- Also known as *k–means* algorithm or the Generalized Lloyd Algorithm (GLA)
- Requires a *training set* instead of relying on computing or estimating the probabilities as in Lloyd-Max

LBG Algorithm

Choose a small error tolerance $\varepsilon > 0$.
Choose initial codewords Y_i, $i = 1 \ldots K$
Compute $V_i = \{X: d(X, Y_i) < d(X, Y_j) \ \forall \ j\}$
Compute distortion D:

$$D = \sum_{i=1}^{K} \sum_{X \in V_i} \left\| X - Y_i \right\|^2$$

Repeat
  Compute new codewords Y_i':

$$Y_i' = \frac{1}{|V_i|} \sum_{X \in V_i} X$$

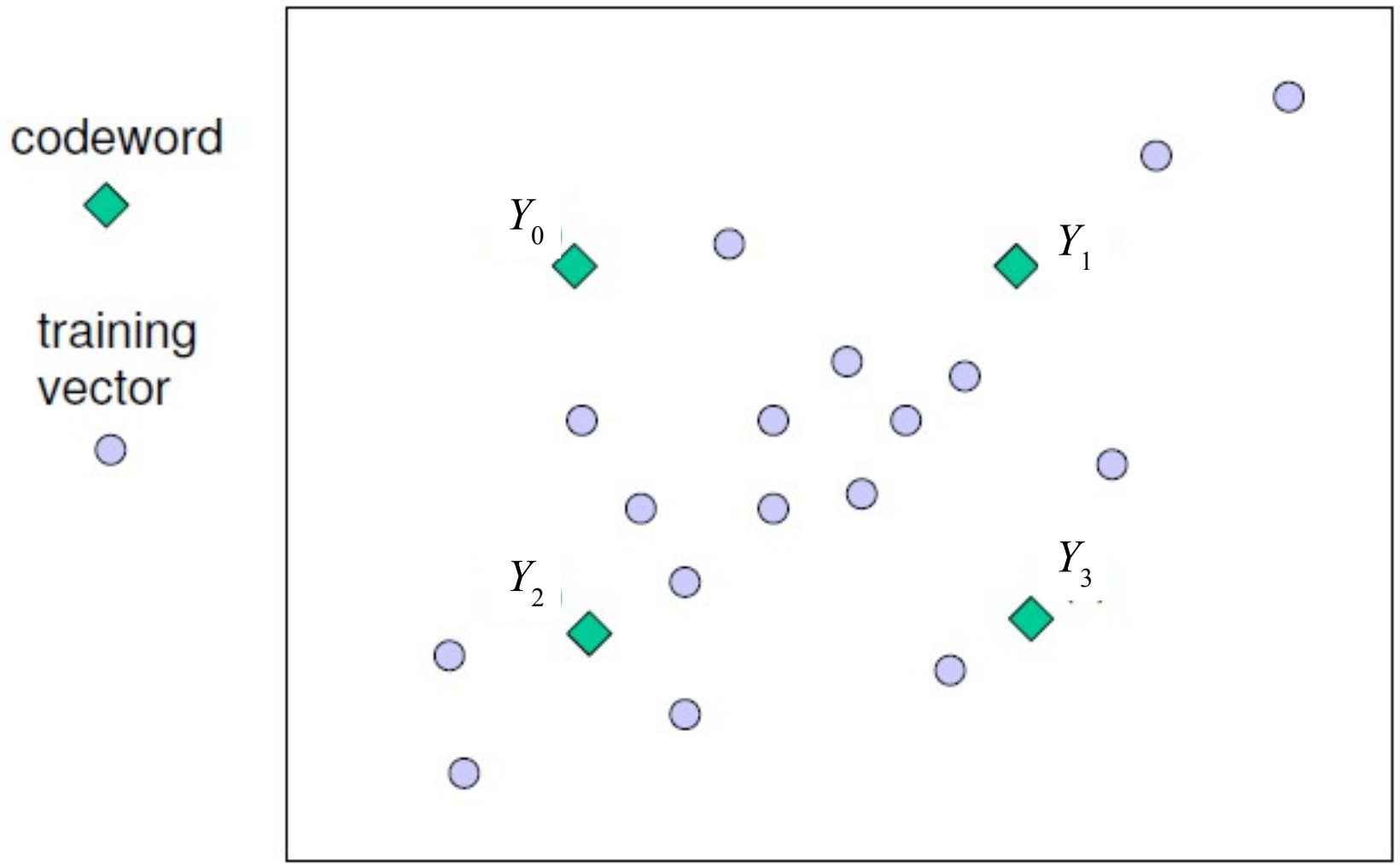
  Compute $V_i' = \{X: d(X, Y_i) < d(X, Y_j) \ \forall \ j\}$
  Compute distortion D':
  if $(D' - D)/D < \varepsilon$ then quit
  else $Y = Y'$; $V = V'$, $D = D'$
End

All training vectors that are closest to code vector Y_i

For every code word, compute the squared error of points closer to Y_i

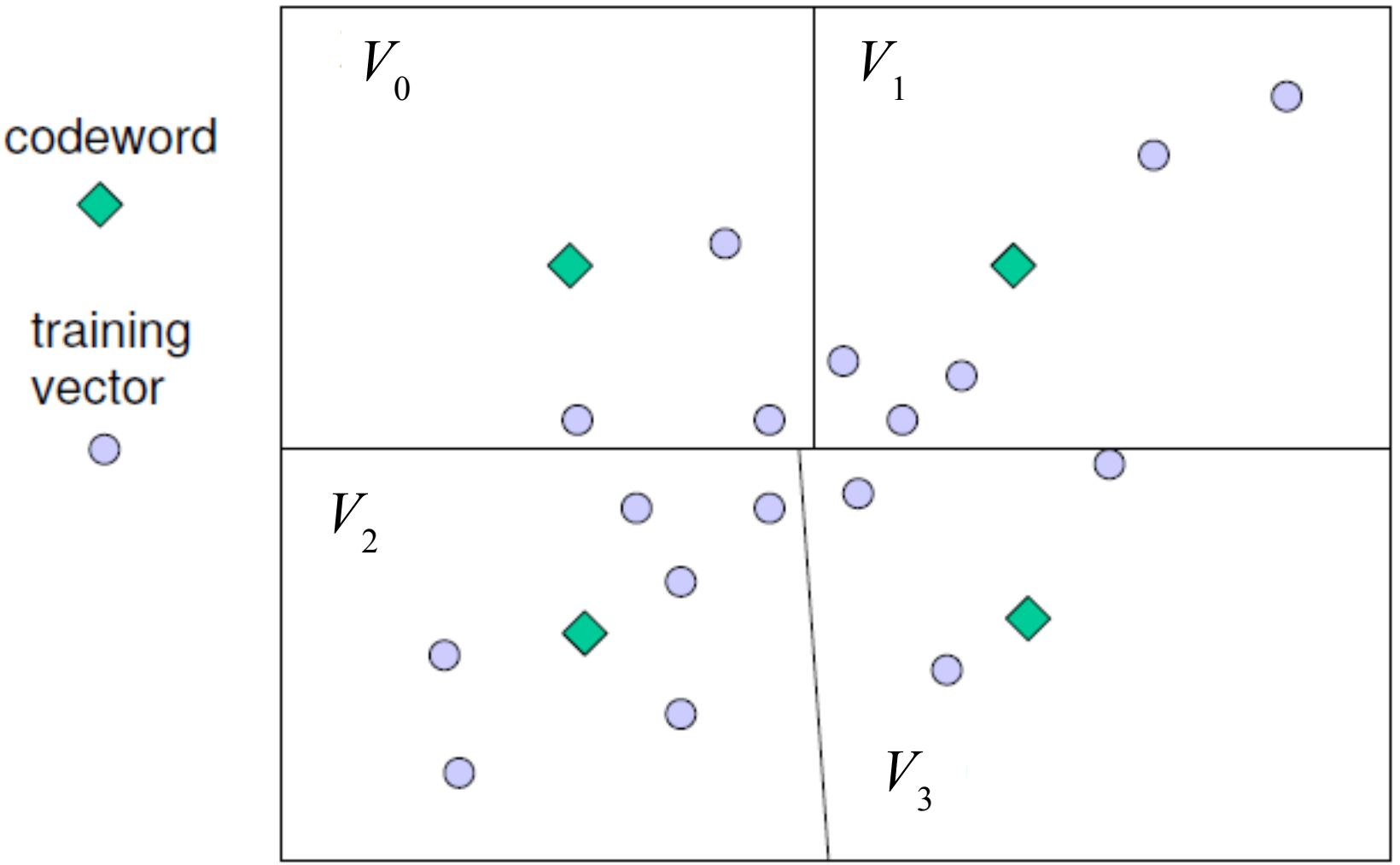
Compute the new codeword as the centroid of all points in V_i

Example



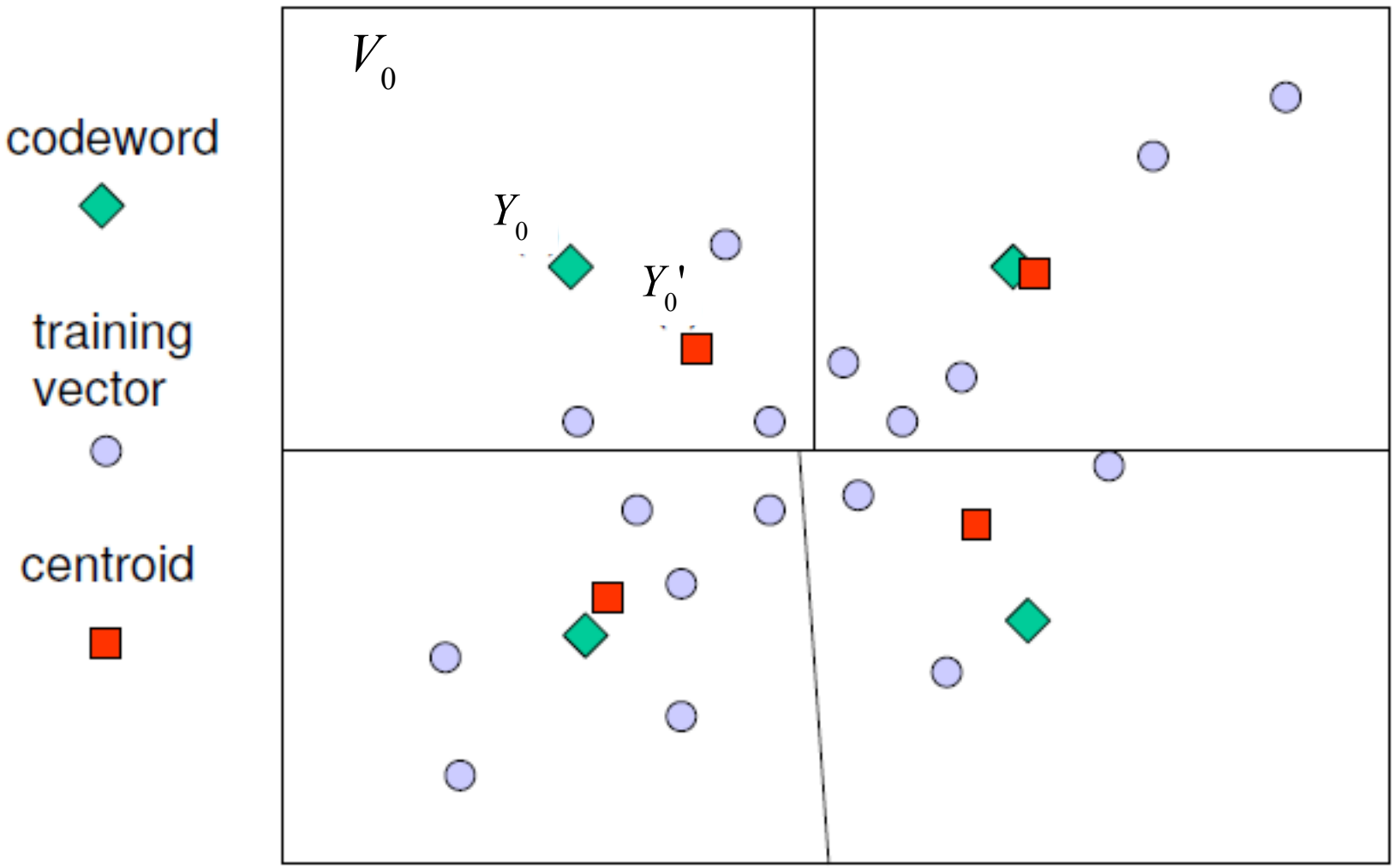
Choose initial code vectors

Example



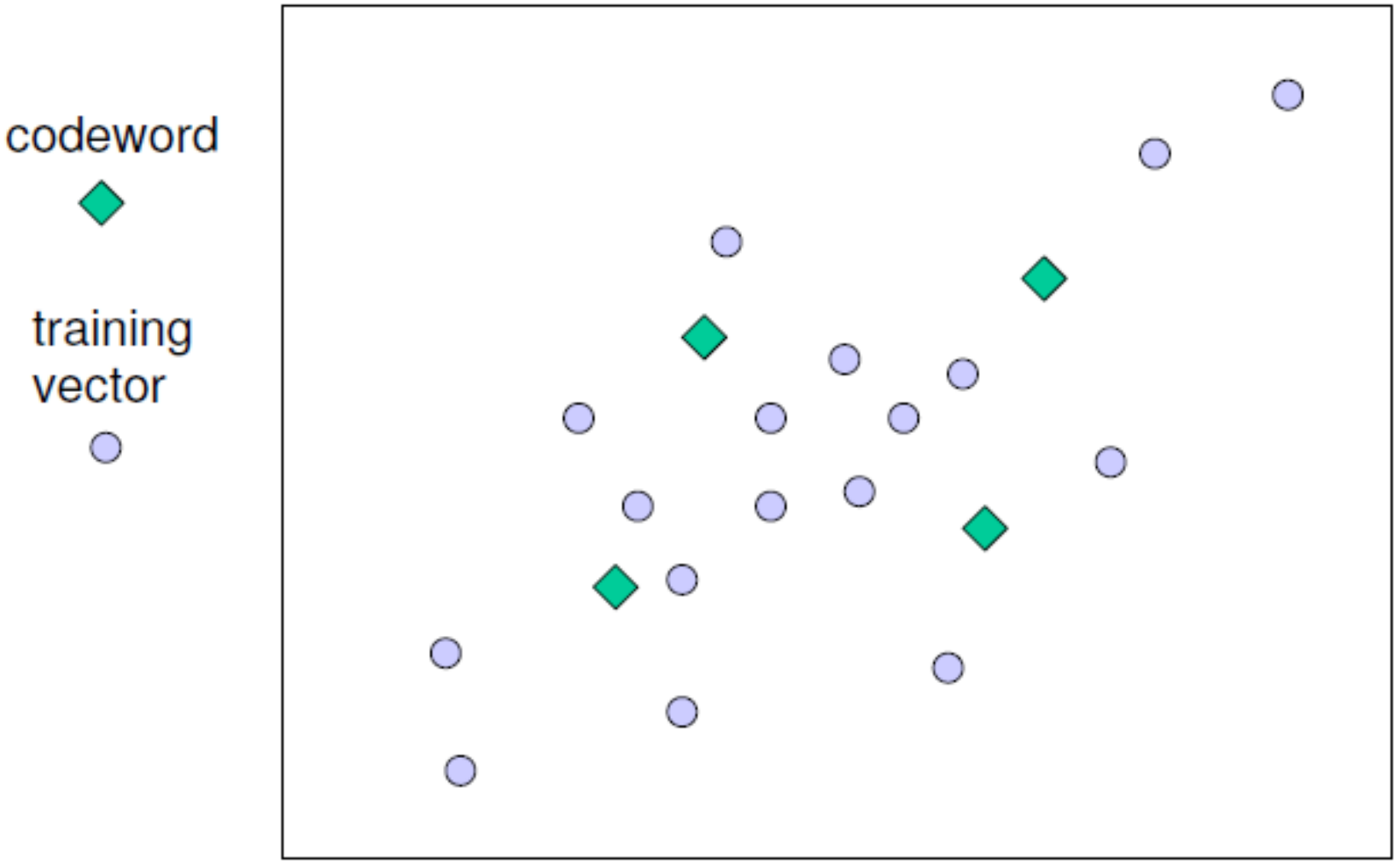
Compute decision regions

Example



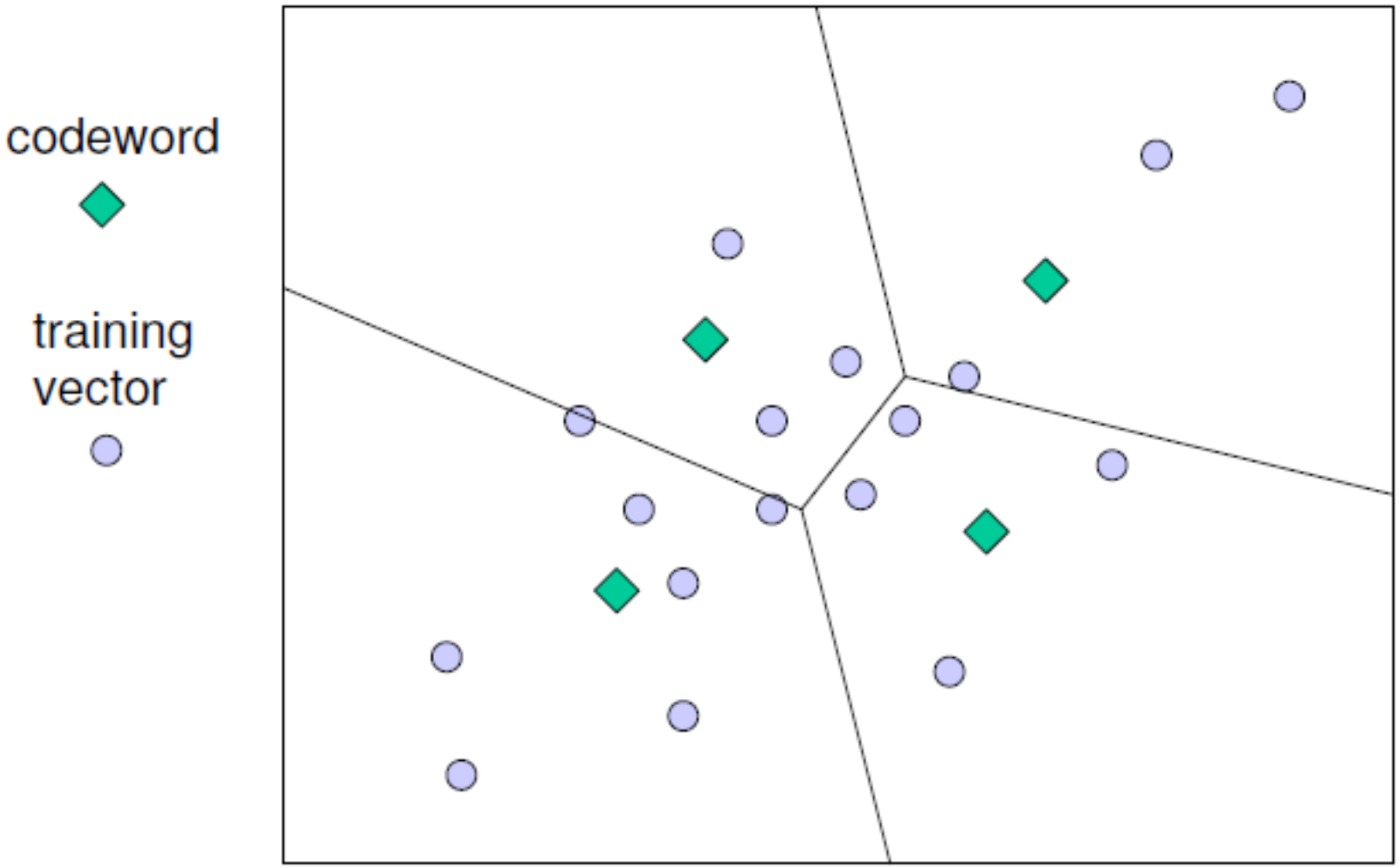
Compute centroids of each region

Example



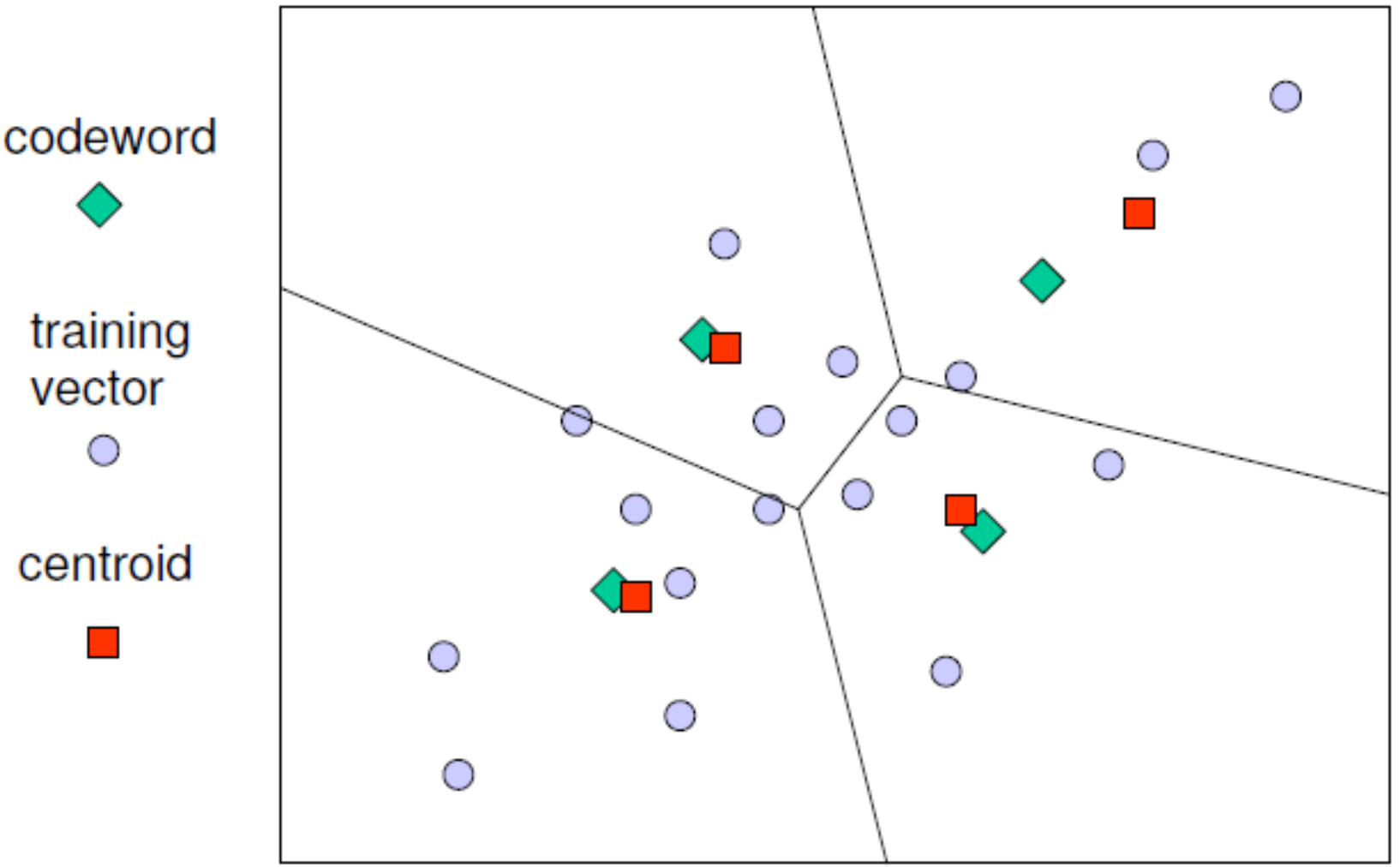
Start a new iteration

Example



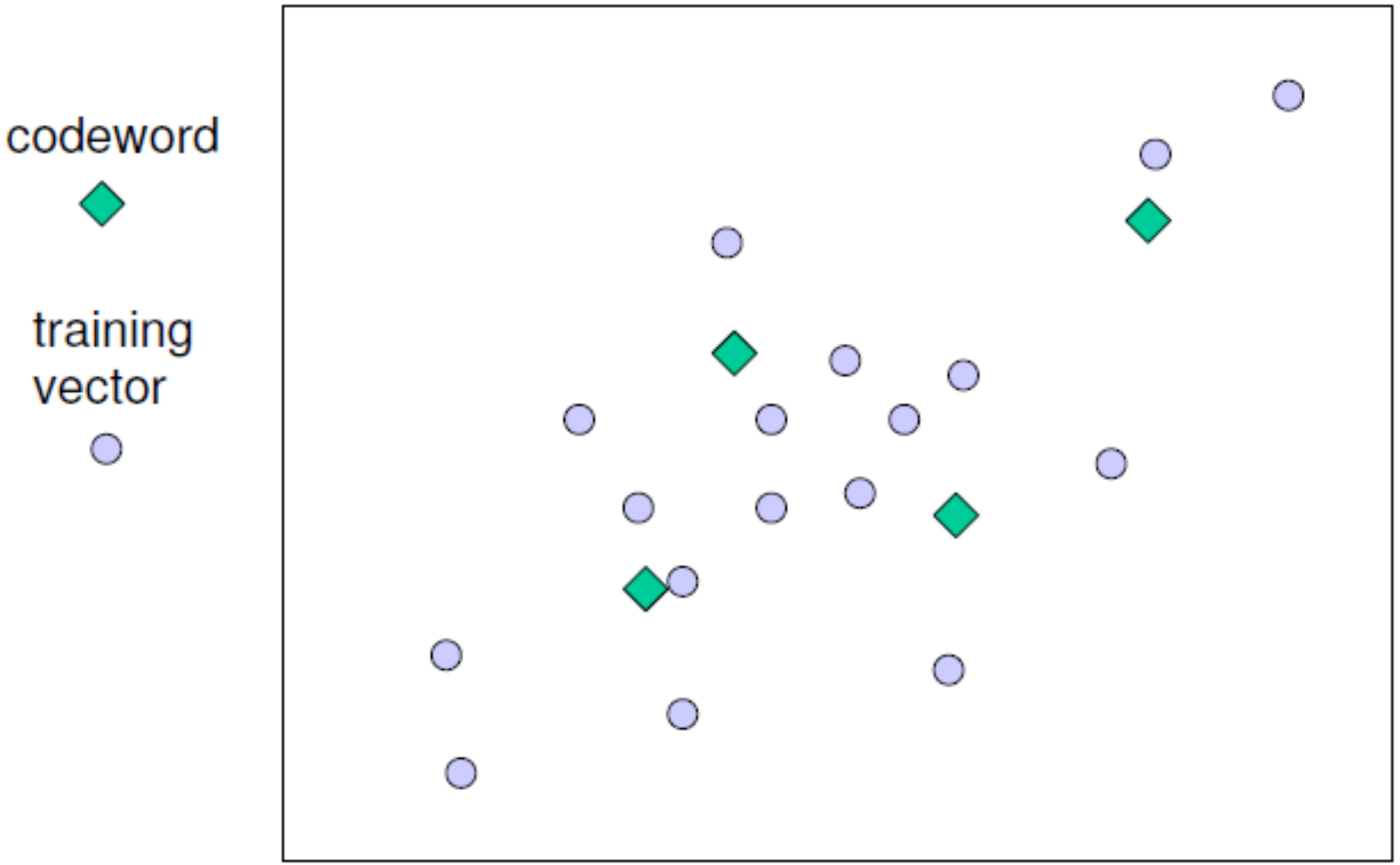
Compute regions

Example



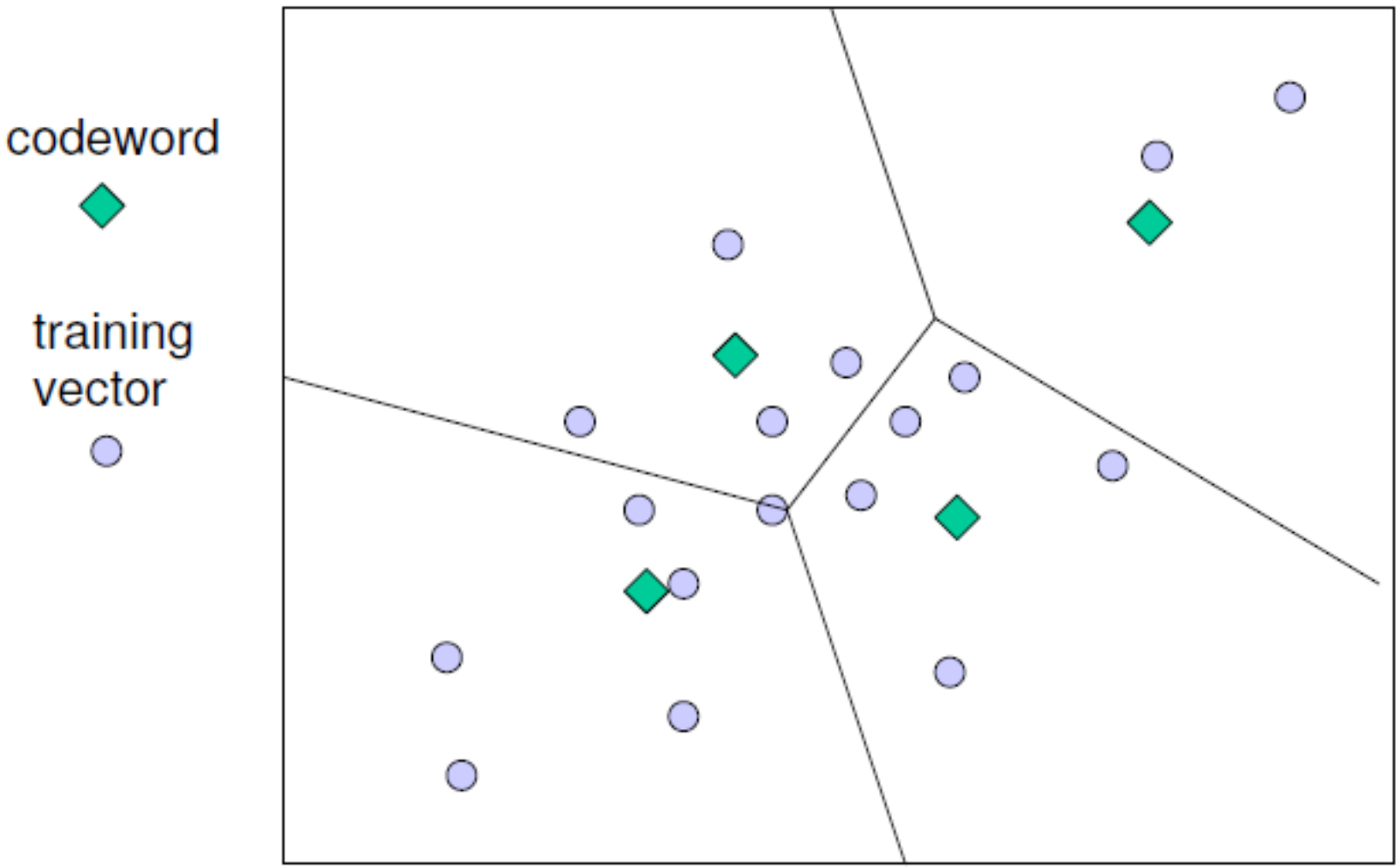
Compute centroids

Example



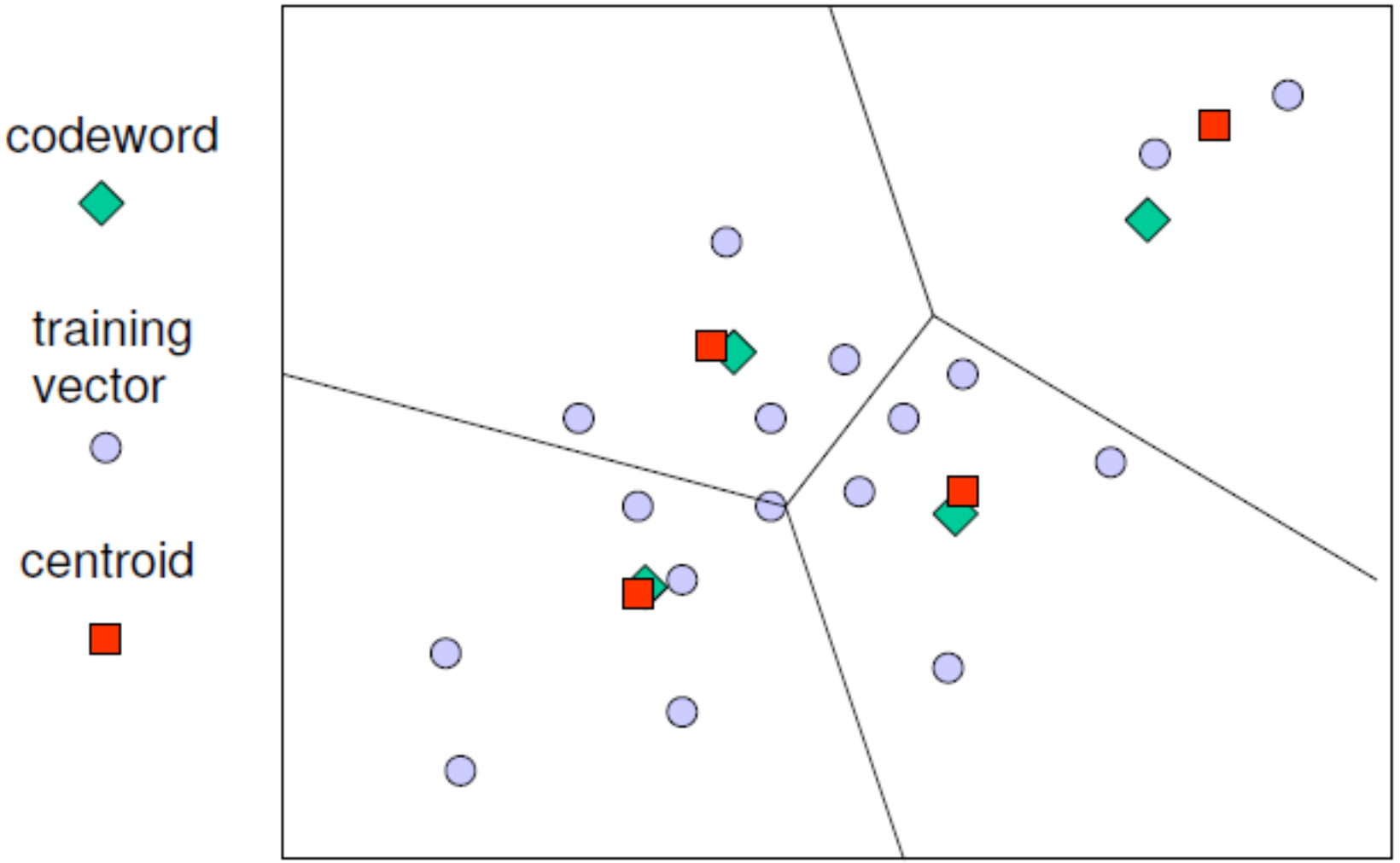
New iteration

Example



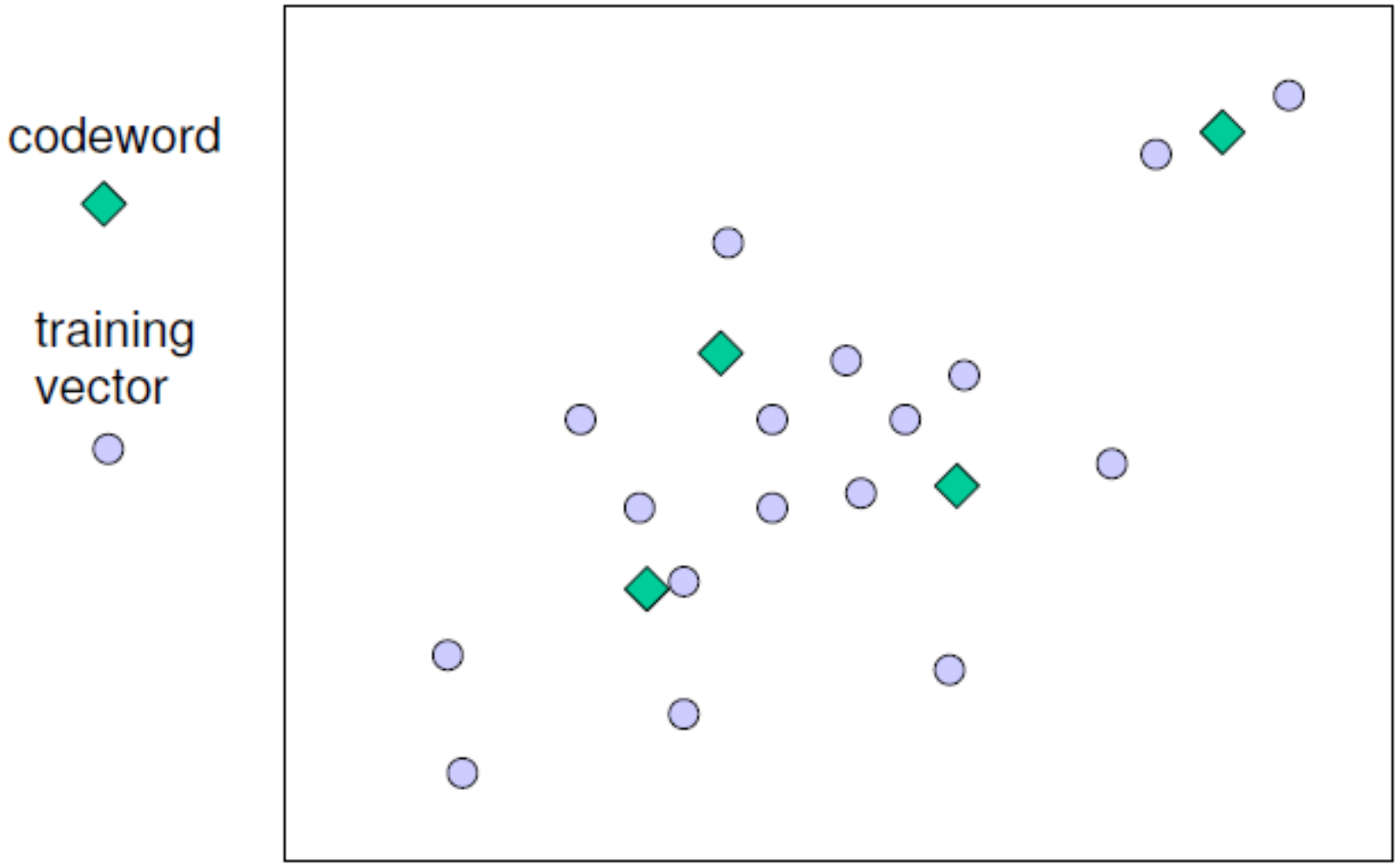
Compute regions

Example



Compute centroids

Example

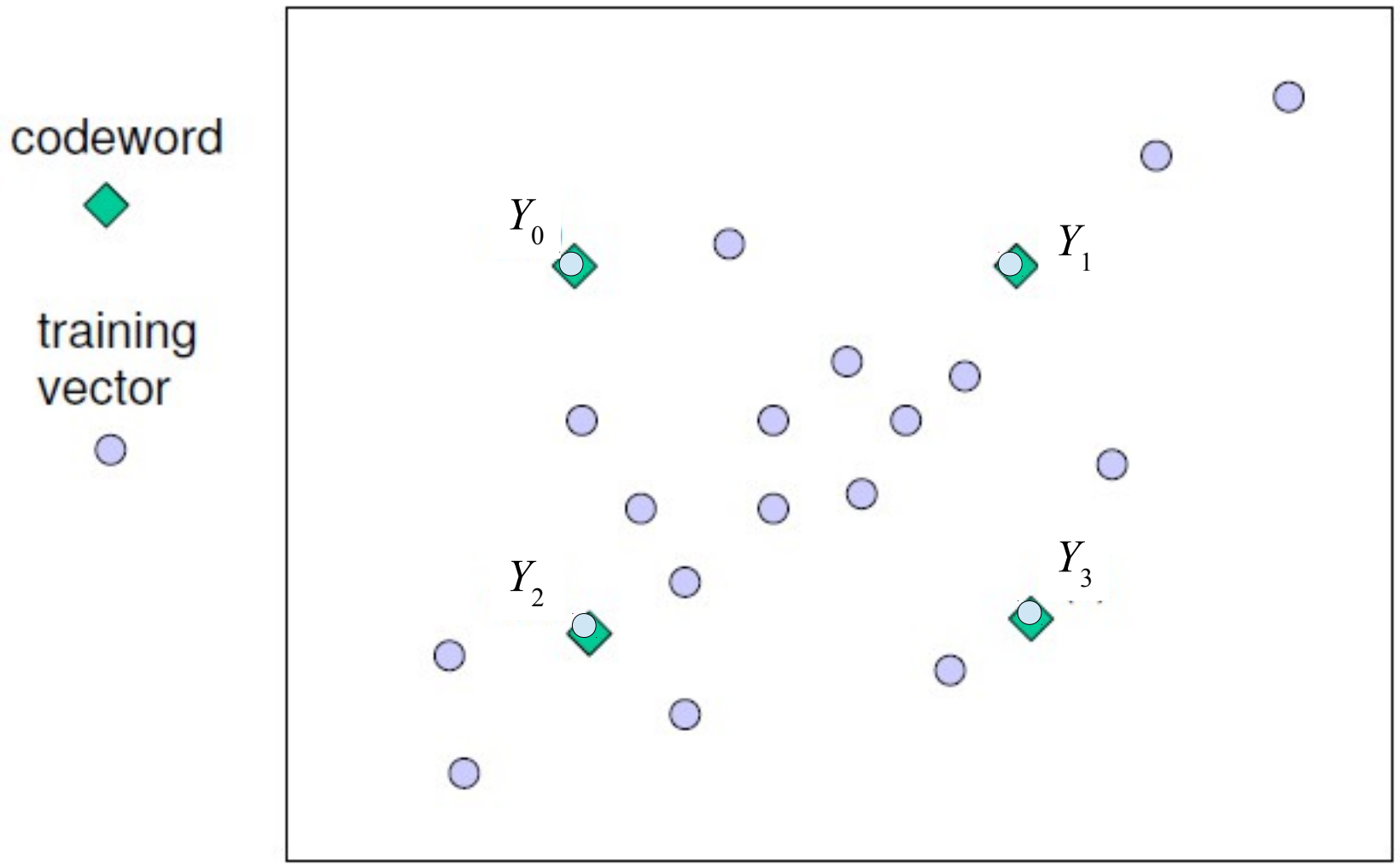


Final result

LBG Issues: Initialization

- For the LBG Algorithm to work, we need to start with *initial values* for the code vectors Y_i's
- There are a variety of ways to perform this initialization
- The simplest way is to choose K sample vectors from the *training set* as the initial set of code vectors
- This guarantees that every initial region is non-empty

Example

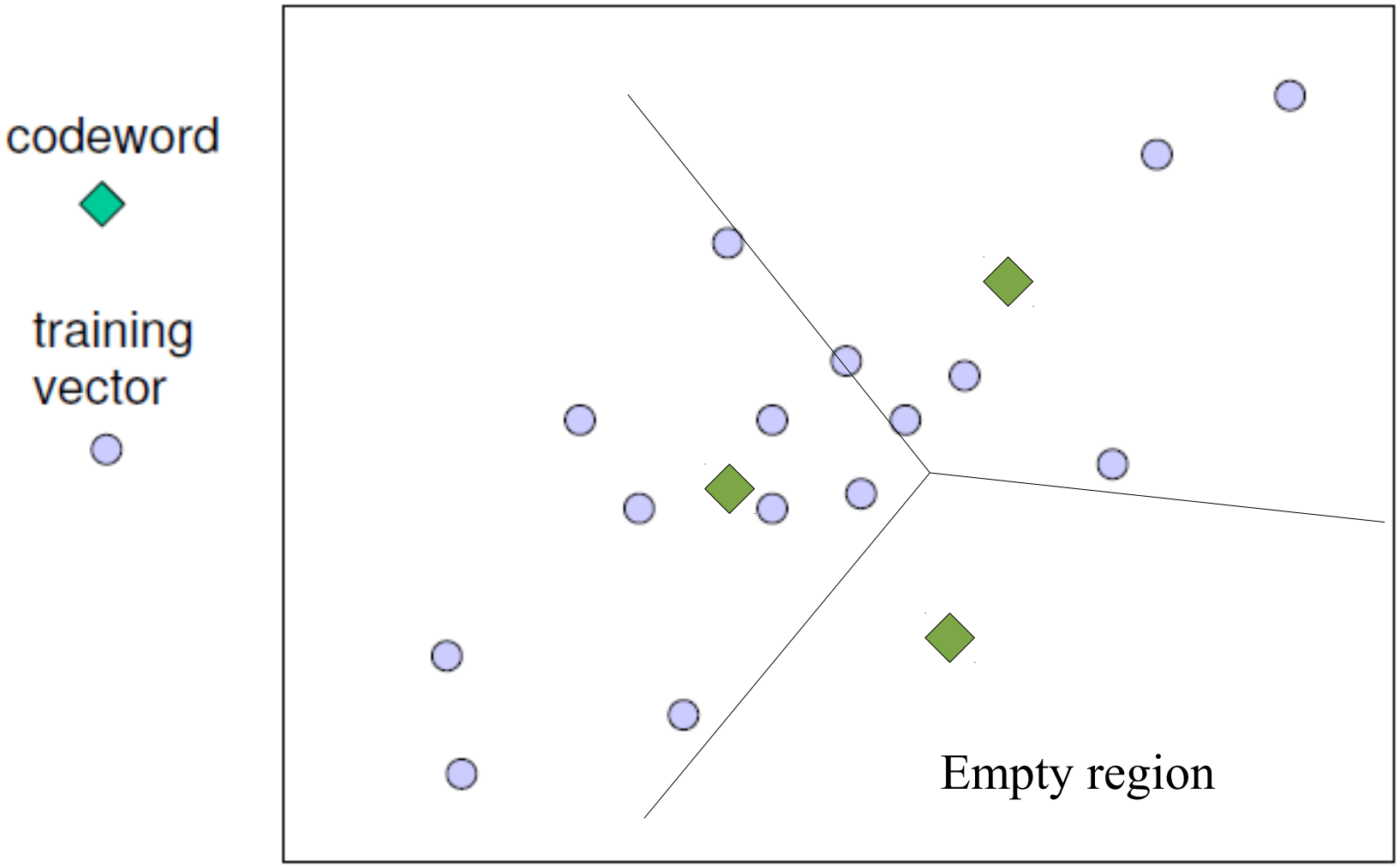


Initial code vectors are samples from the training set

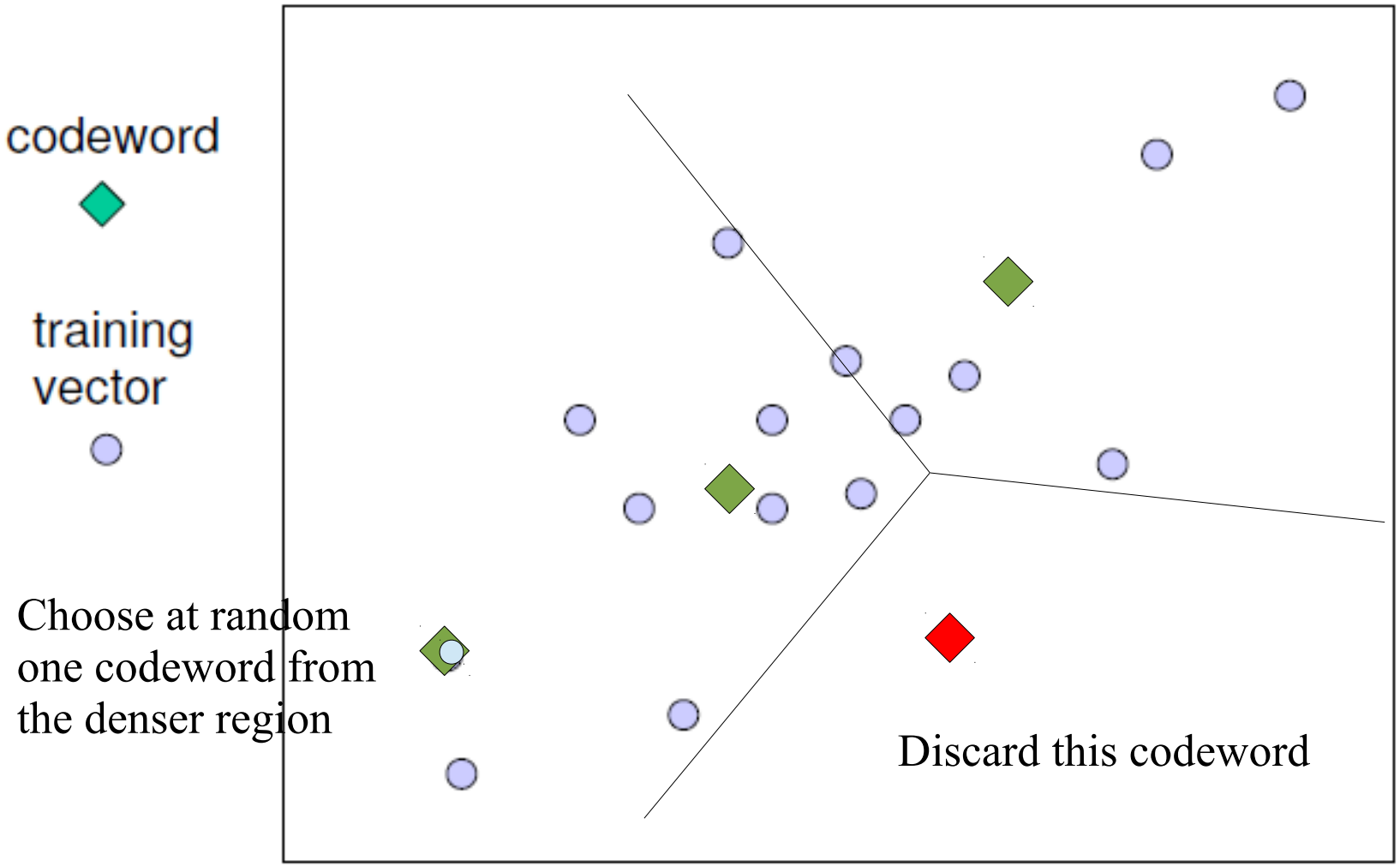
LBG Issues: Empty Regions

- When updating the regions around each code vector, it can happen that one region V_i becomes *empty* i.e. all samples X are closer to other code vectors
- Many ways to deal with this, e.g.
  - Leave this code vector and don't update it further
  - Replace this code vector with a random sample X from the region V_j with the most samples in it

Example



Example


codeword
training vector
Choose at random one codeword from the denser region
Discard this codeword

Application: Image Compression

- We can take blocks of pixels in the image of size NxN and treat them as $L = N^2$ – dimensional vectors by concatenating the rows together in one long vector
- For example, we can compress this image by taking blocks of size 4x4 pixels i.e. 16-dimensional vectors
- The codebook was trained on the image

Application: Image Compression

Original

16 codewords

64 codewords

256 codewords

1024 codewords

Application: Image Compression

TABLE 10.7 Summary of compression measures for image compression example.

Codebook Size (# of codewords)	Bits Needed to Select a Codeword	Bits per Pixel	Compression Ratio
16	4	0.25	32:1
64	6	0.375	21.33:1
256	8	0.50	16:1
1024	10	0.625	12.8:1

- Since the codebook was trained on the image itself, we need to transmit the codebook to the receiver, which causes more overhead and reduces the compression ratio

TABLE 10.8 Overhead in bits per pixel for codebooks of different sizes.

Codebook Size K	Overhead in Bits per Pixel
16	0.03125
64	0.125
256	0.50
1024	2.0

- Will look later at how to deal with this problem

Summary

- Vector quantization more flexible than scalar quantization
- Many ways to perform vector quantization:
  - LBG
  - Structured Vector Quantizers
  - Trellis Coded Quantization
  - …

Recap

- Quantization
- Scalar Quantization
  - Uniform
  - Nonuniform
- Vector Quantization

- Next: Transform Coding

- More information: Chapter 9, 10 [**IDC**]