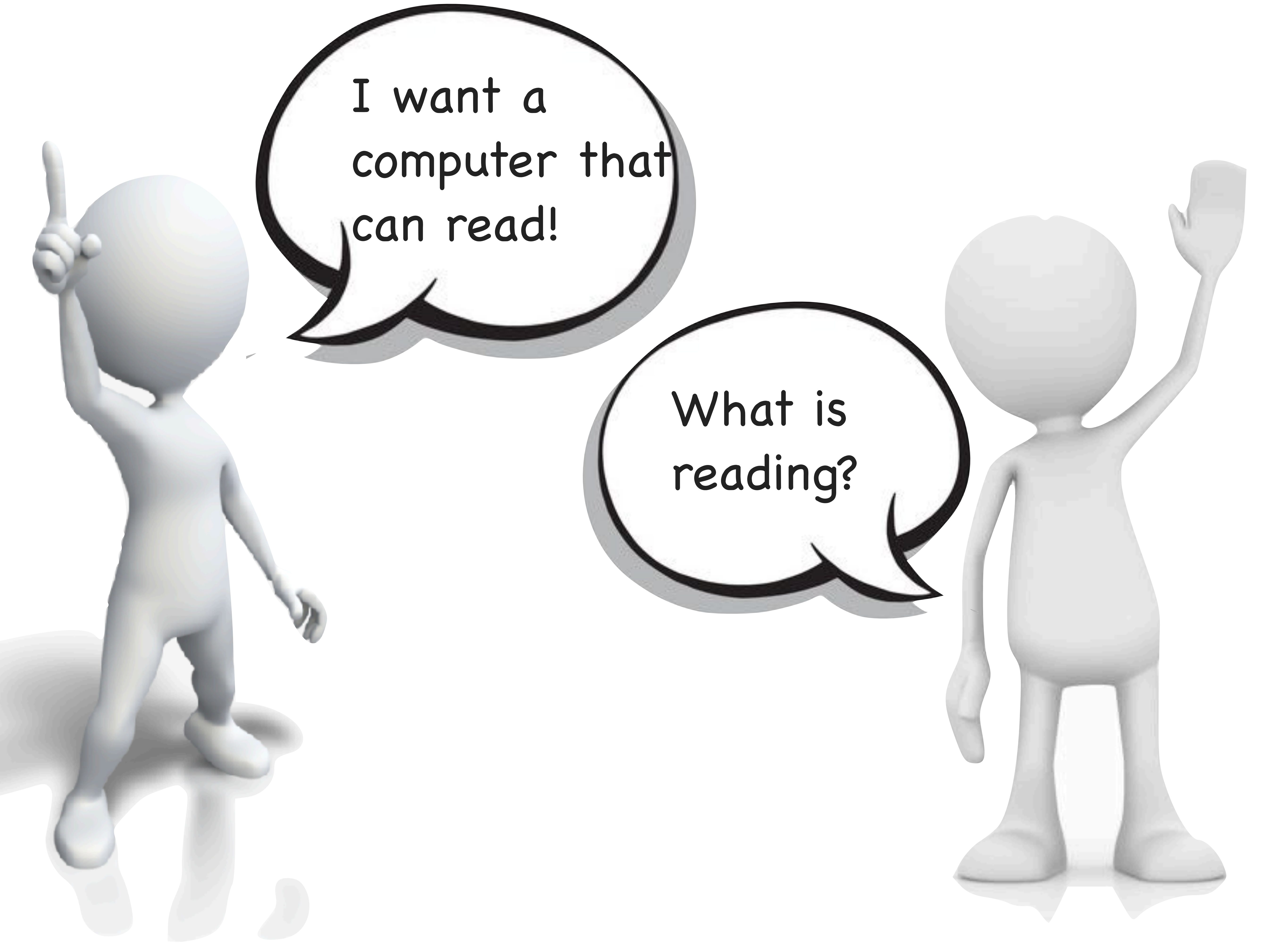
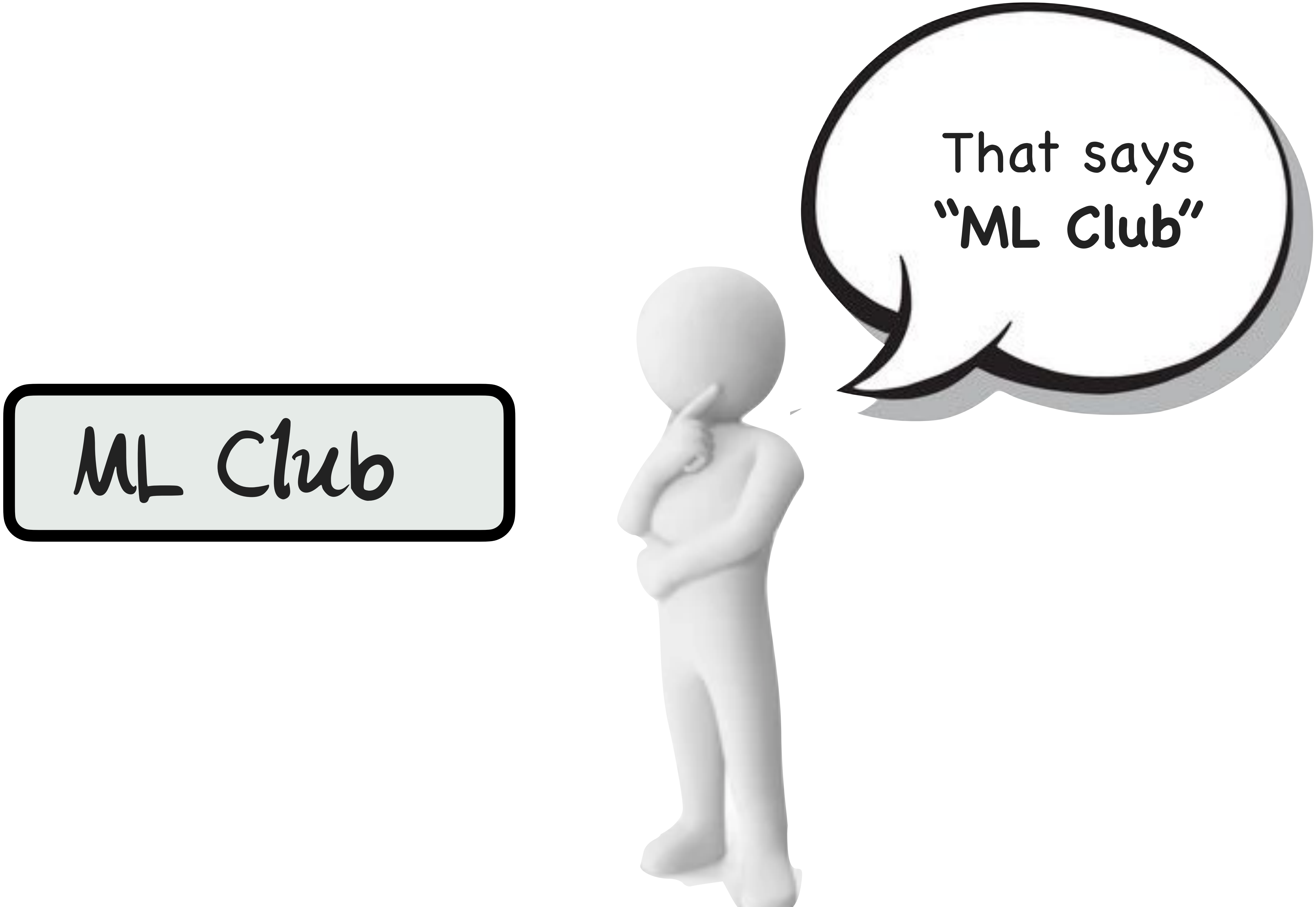
Discovering the Idea of Neural Networks

Or, teaching computers to read.


I want a computer that can read!


I want a computer that can read!
What is reading?


ML Club
That says "ML Club"

Handwritten Sign

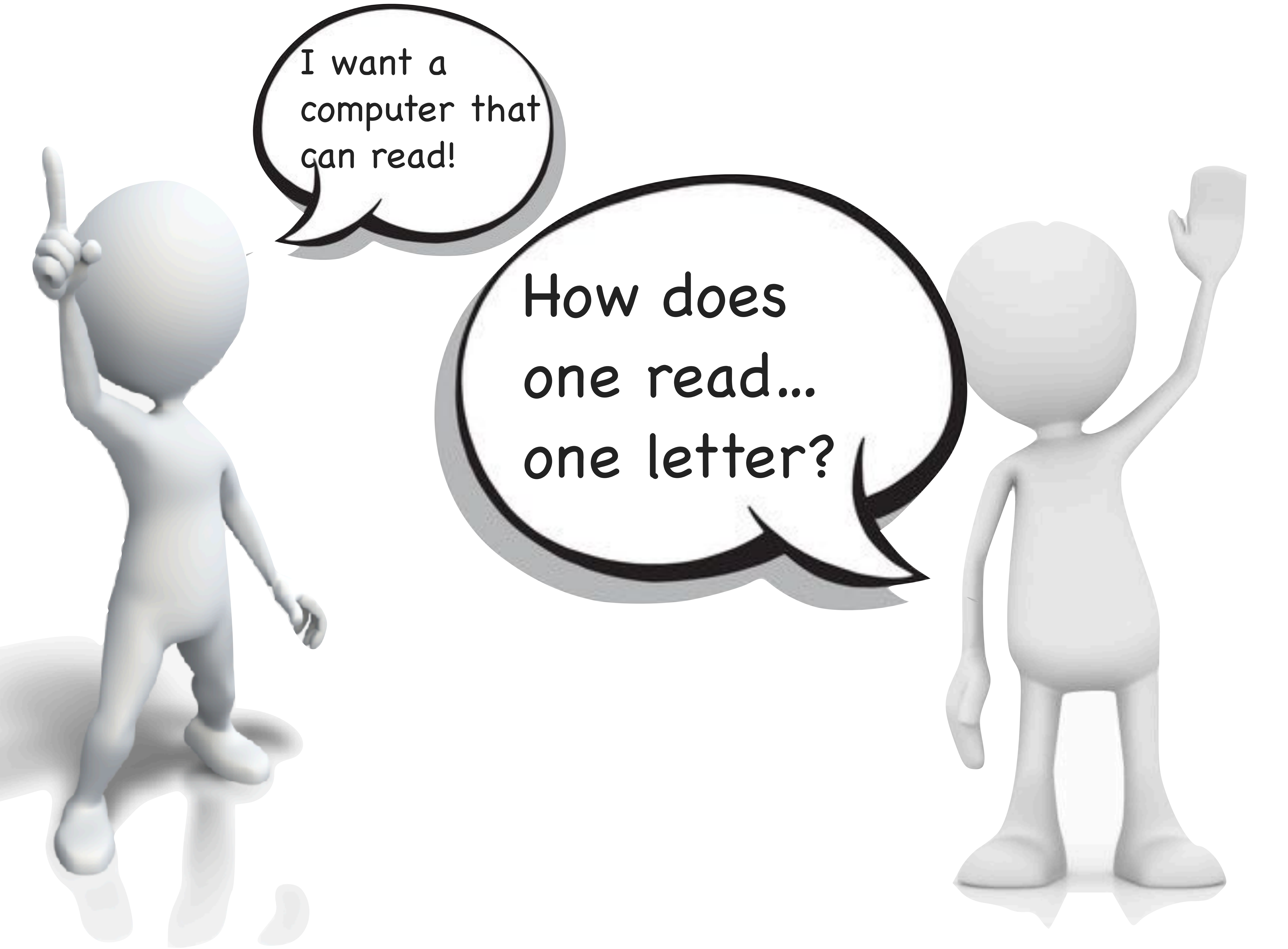
Messy, real-world, external data.

Characters

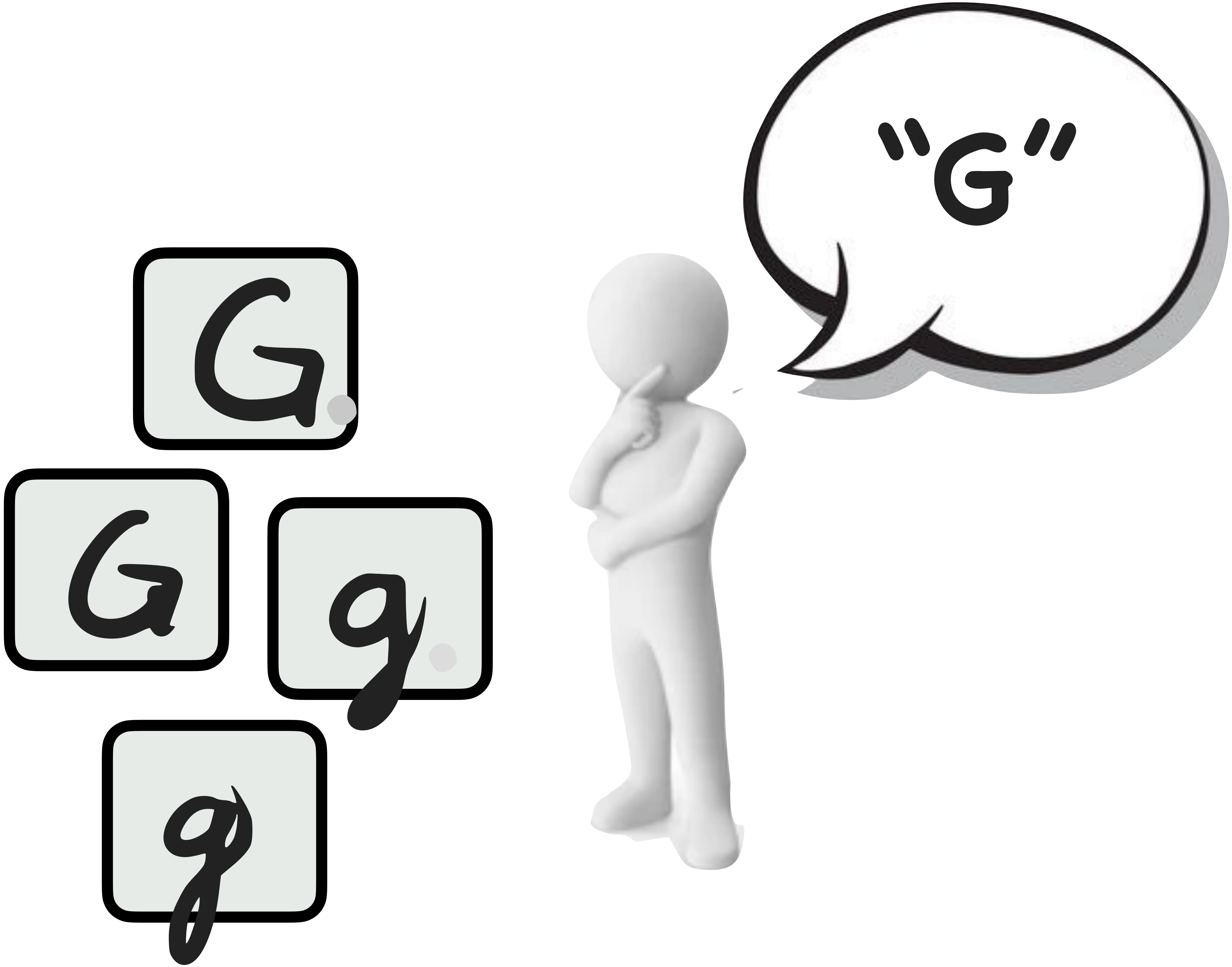
Discrete, abstract categories.

ML Club

ML Club

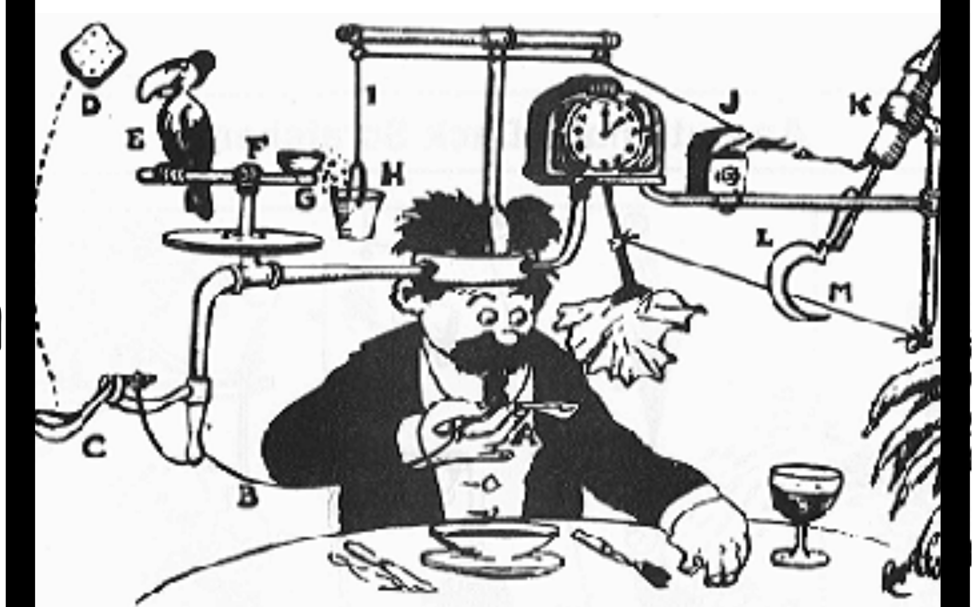

I want a computer that can read!
How does one read... one letter?


A
A
a
a
"A"

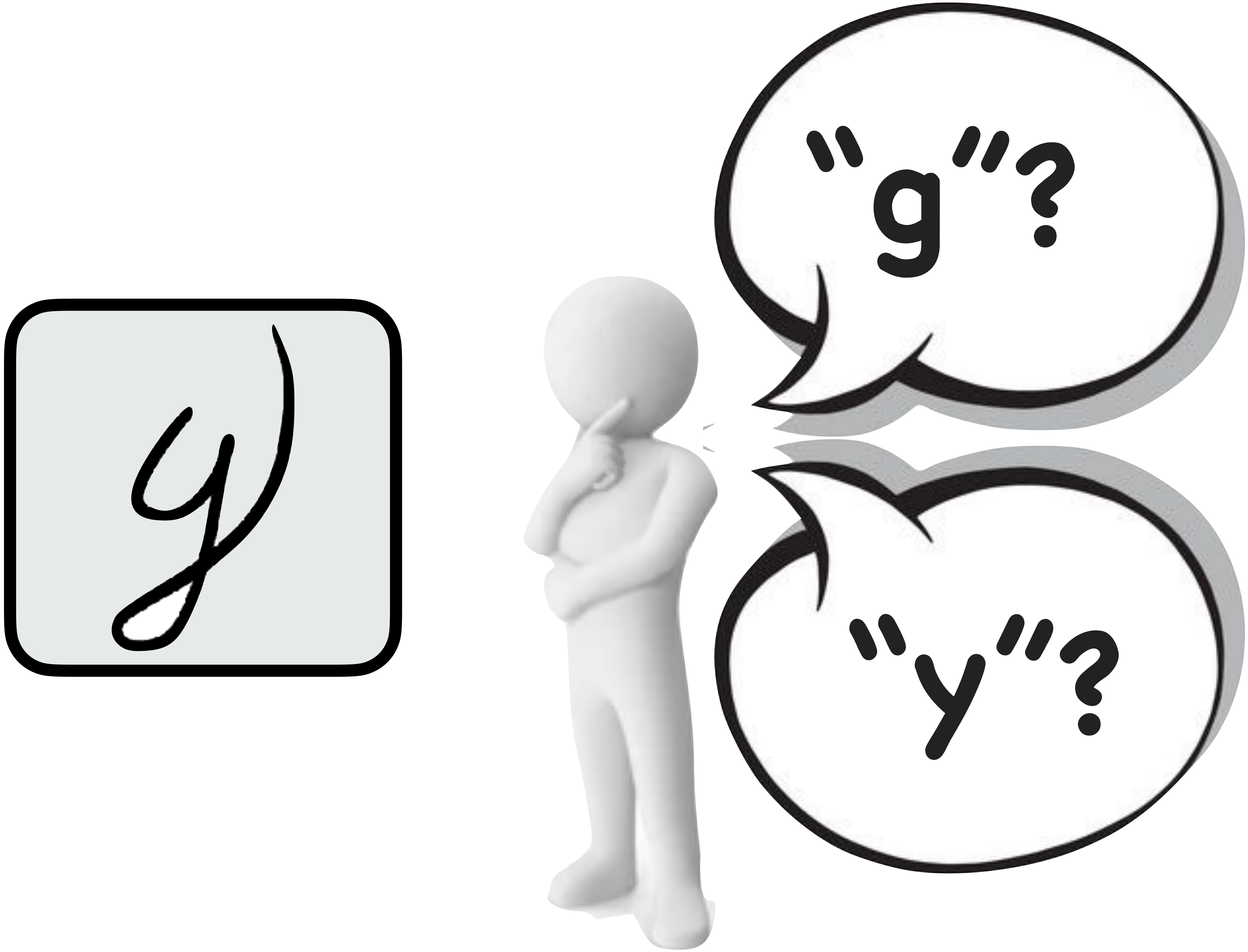

G
G
g
g
"G"

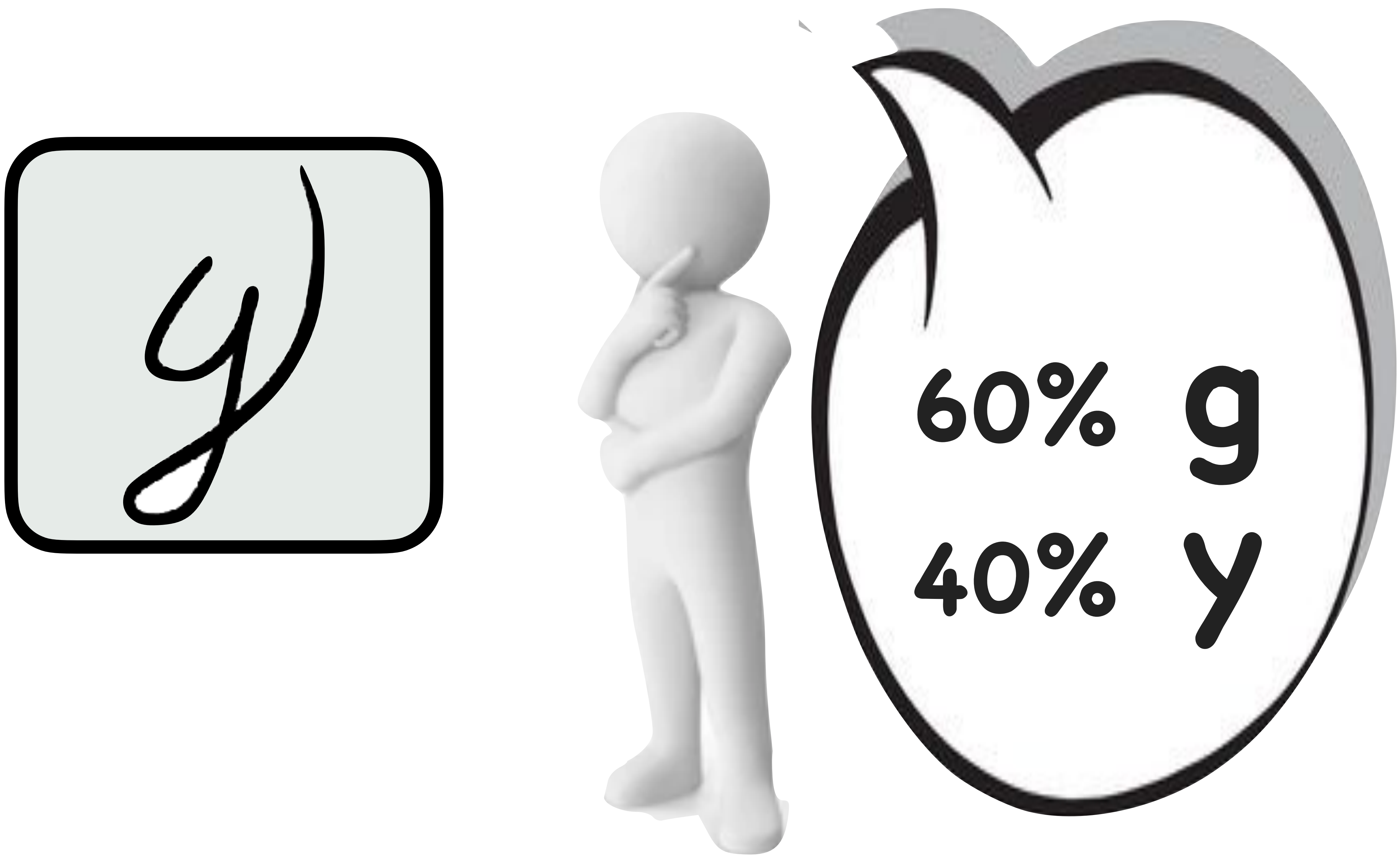
A Reading Machine

An image of a hand-drawn letter



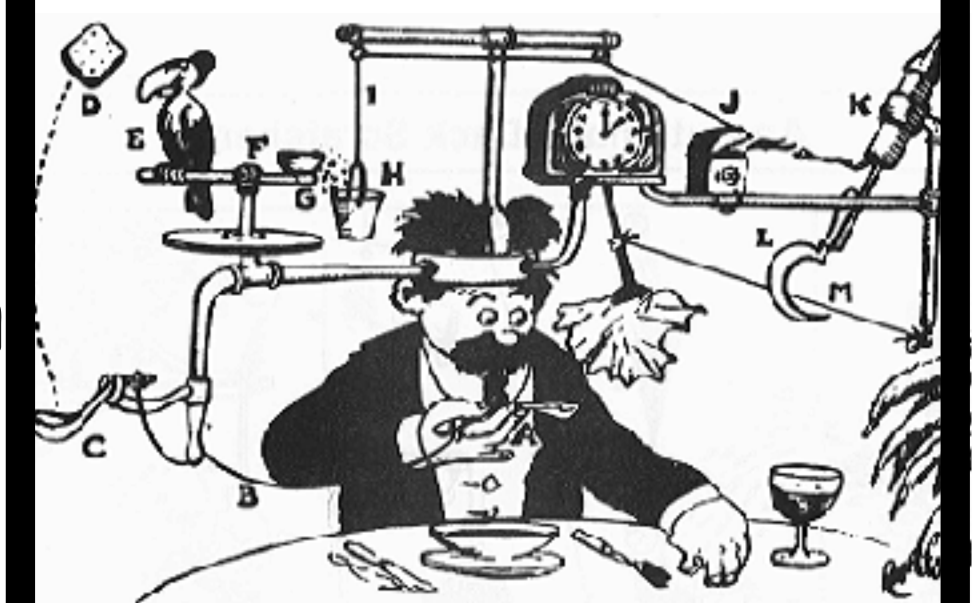
A letter of the alphabet


"g"?
"y"?

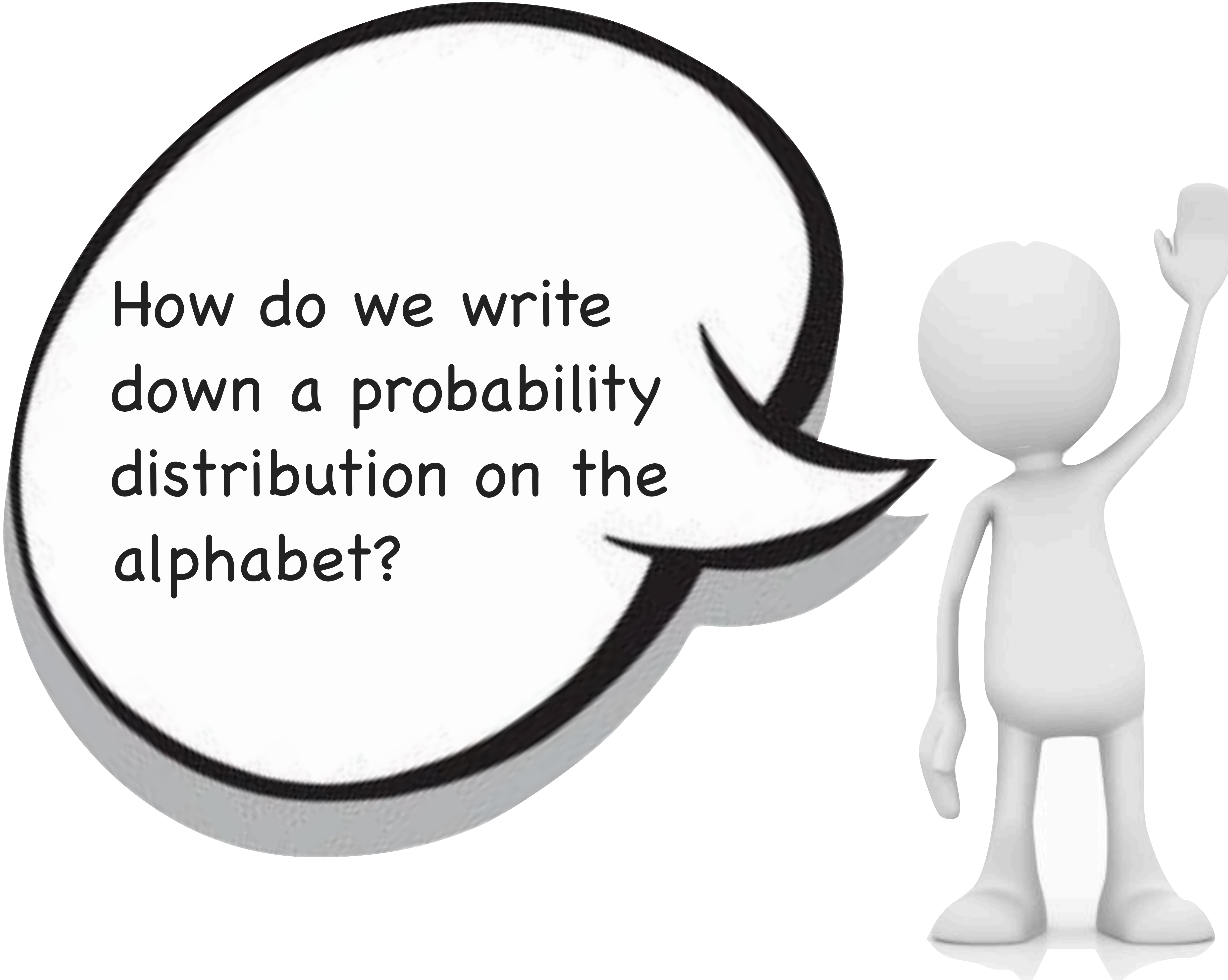

60% g
40% y

A Reading Machine

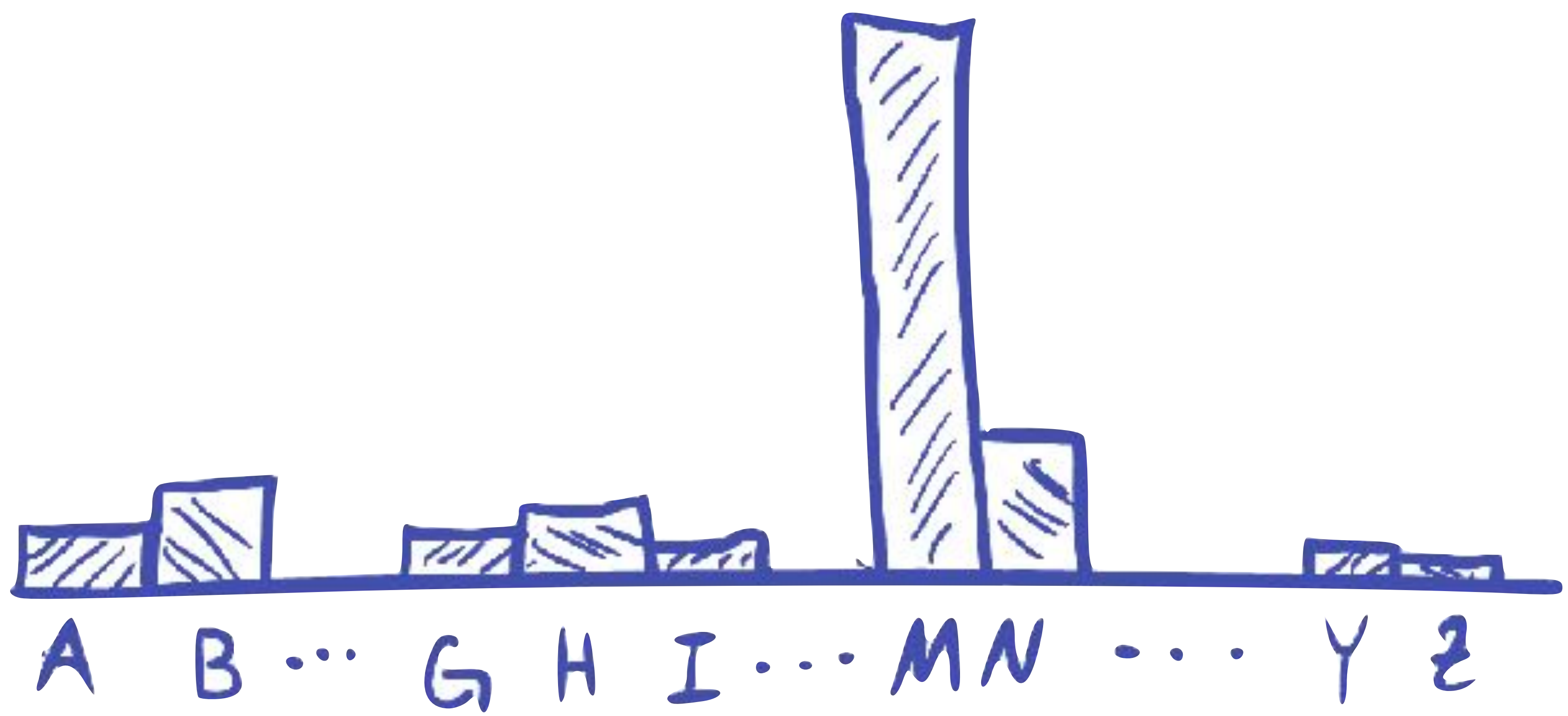
An image of a hand-drawn letter



A probability distribution on the alphabet


How do we write down a probability distribution on the alphabet?

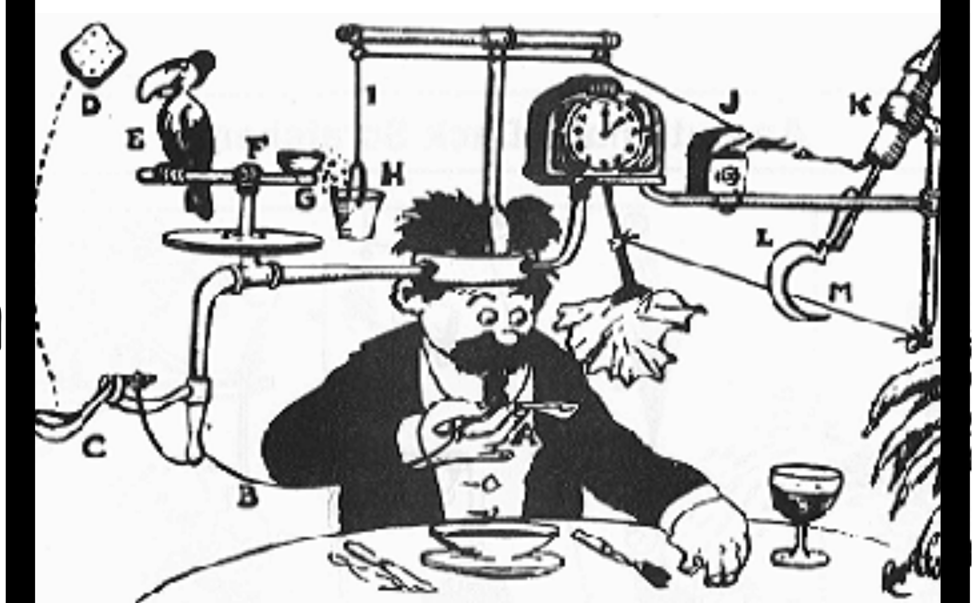
The alphabet has 26 letters, so we should expect to output a probability for each letter.



This means our output should be a list of 26 nonnegative numbers which sum to 1.

A Reading Machine

An image of a hand-drawn letter



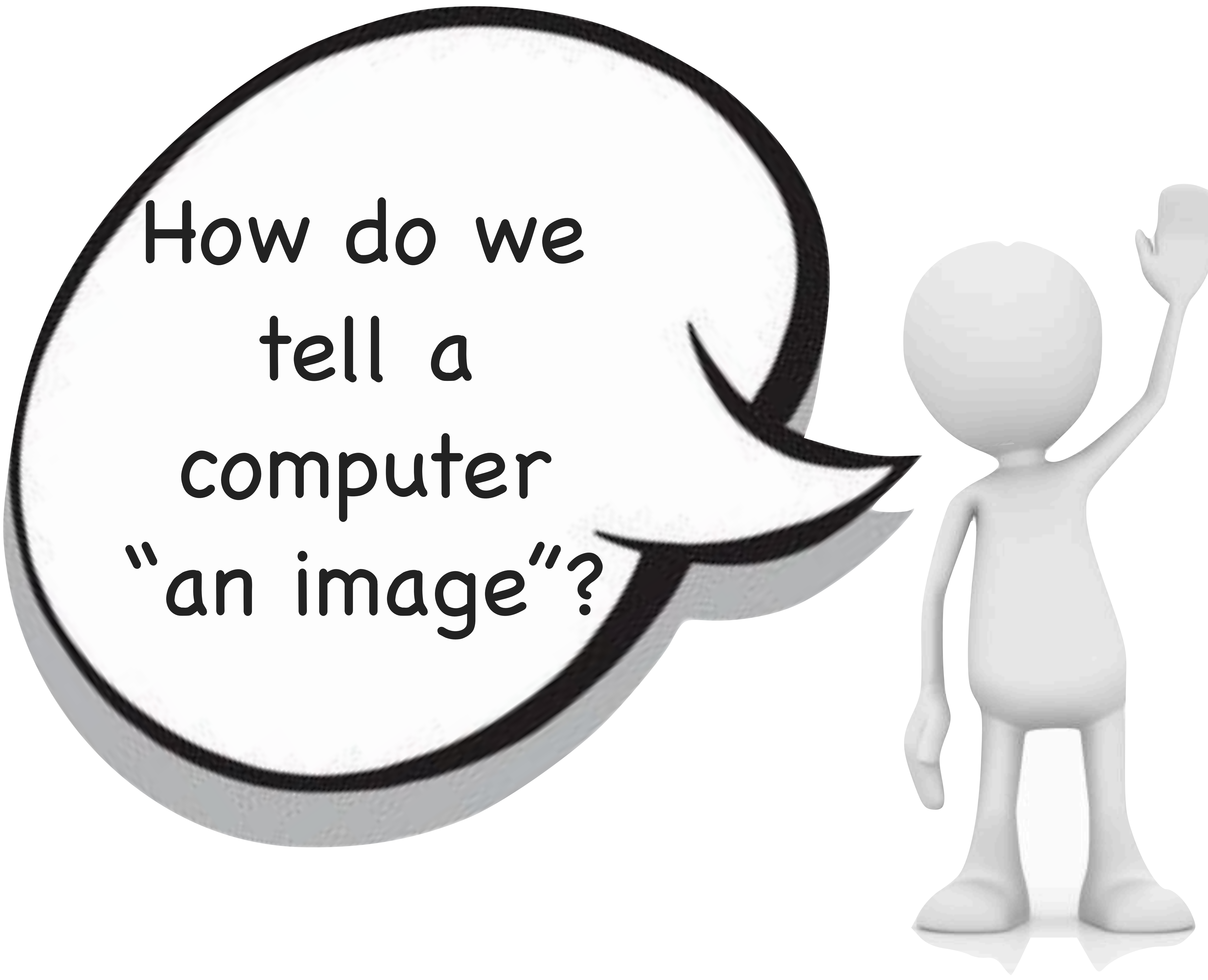
A vector

$$(p_1, p_2, p_3, \ldots, p_{26})$$

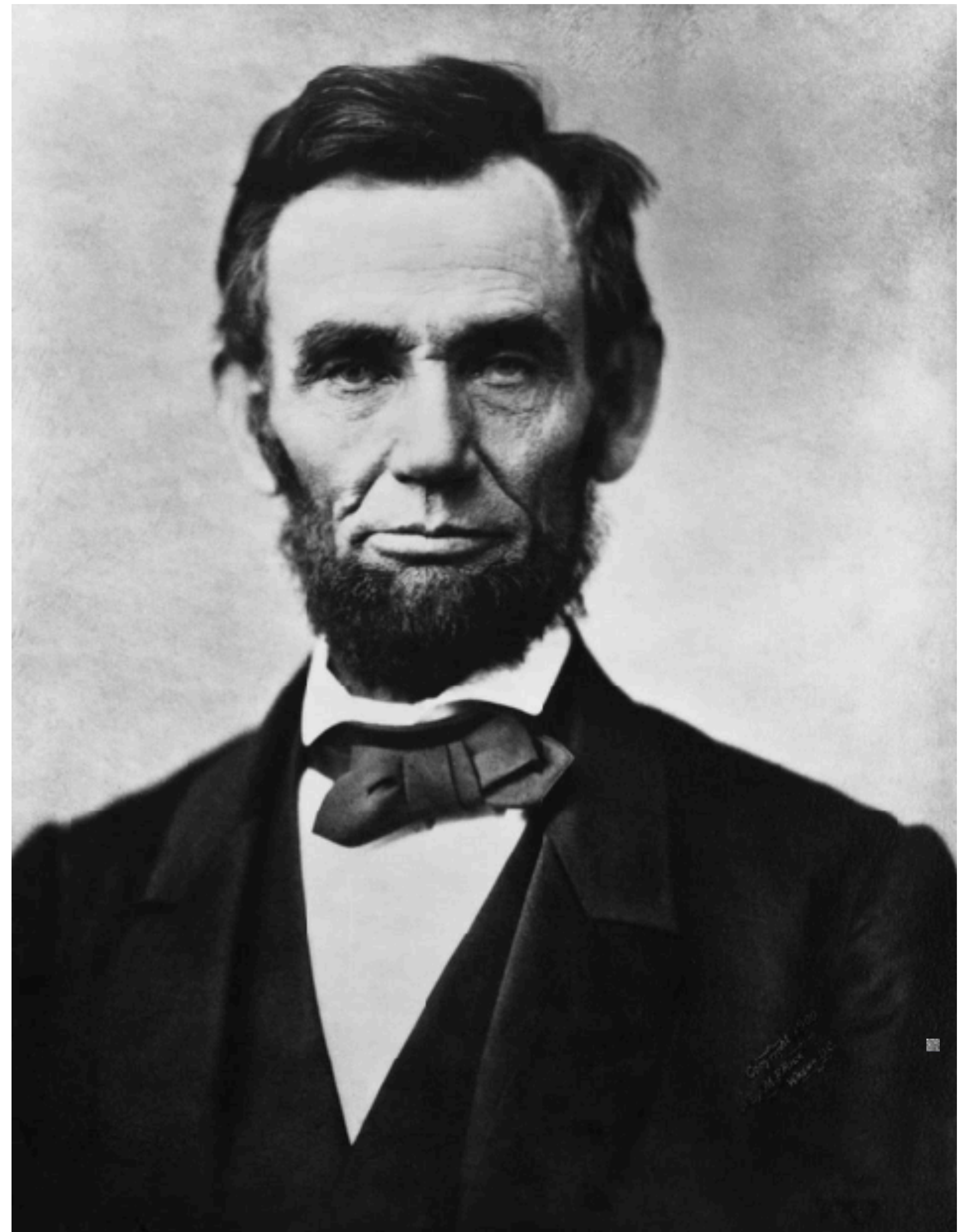
with $p_i \geq 0$

and

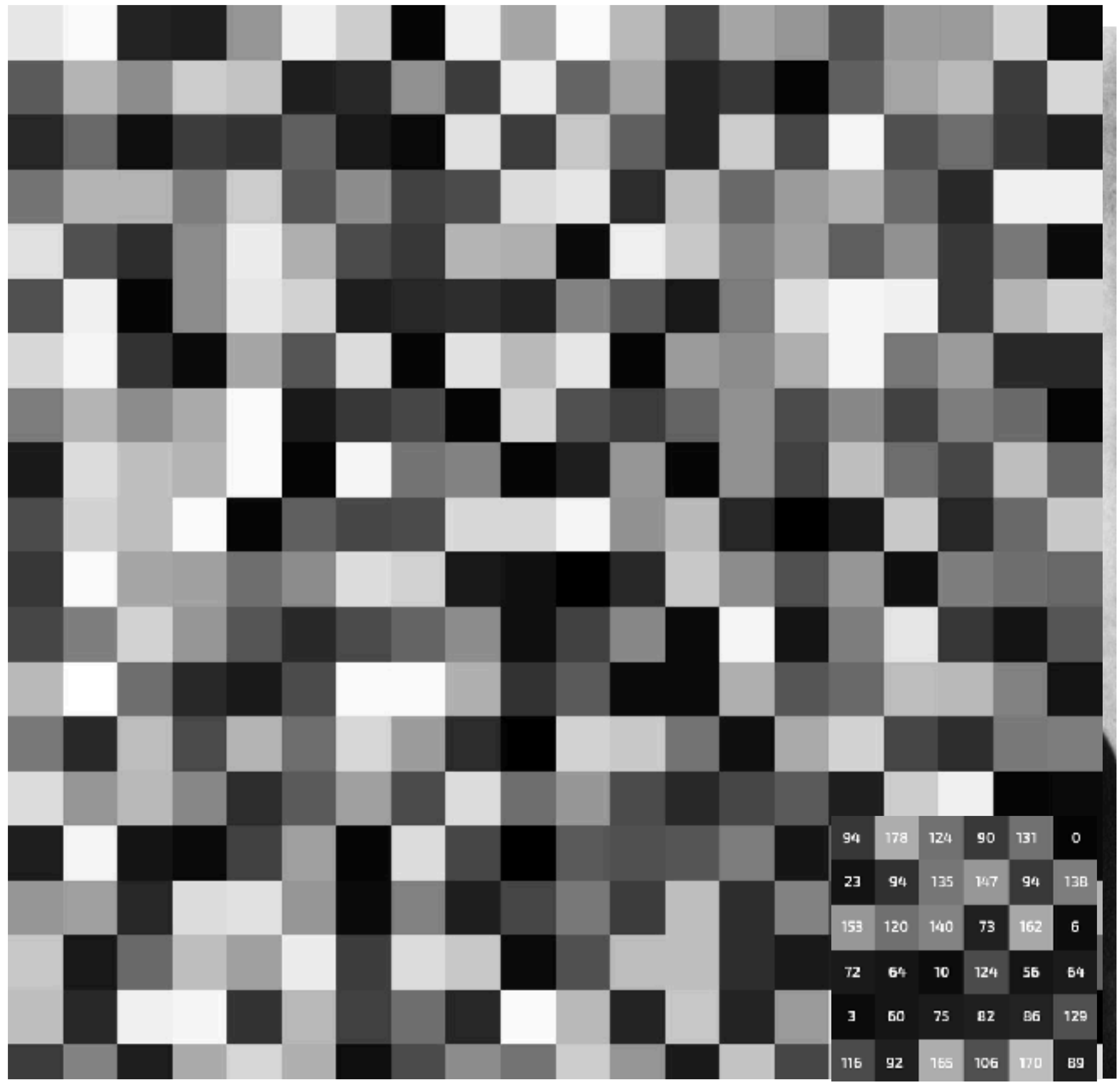
$$\sum_{i=1}^{26} p_i = 1$$


How do we tell a computer “an image”?

If you look really close…



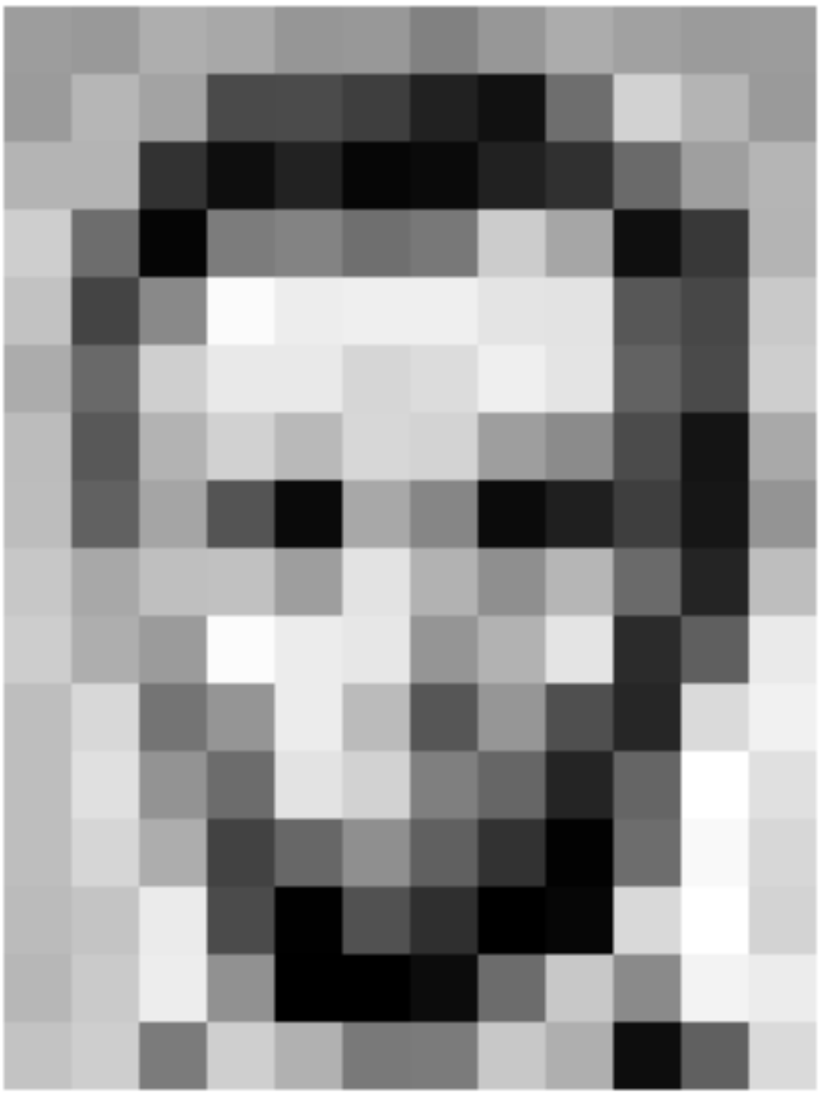
Images are just arrays of pixels

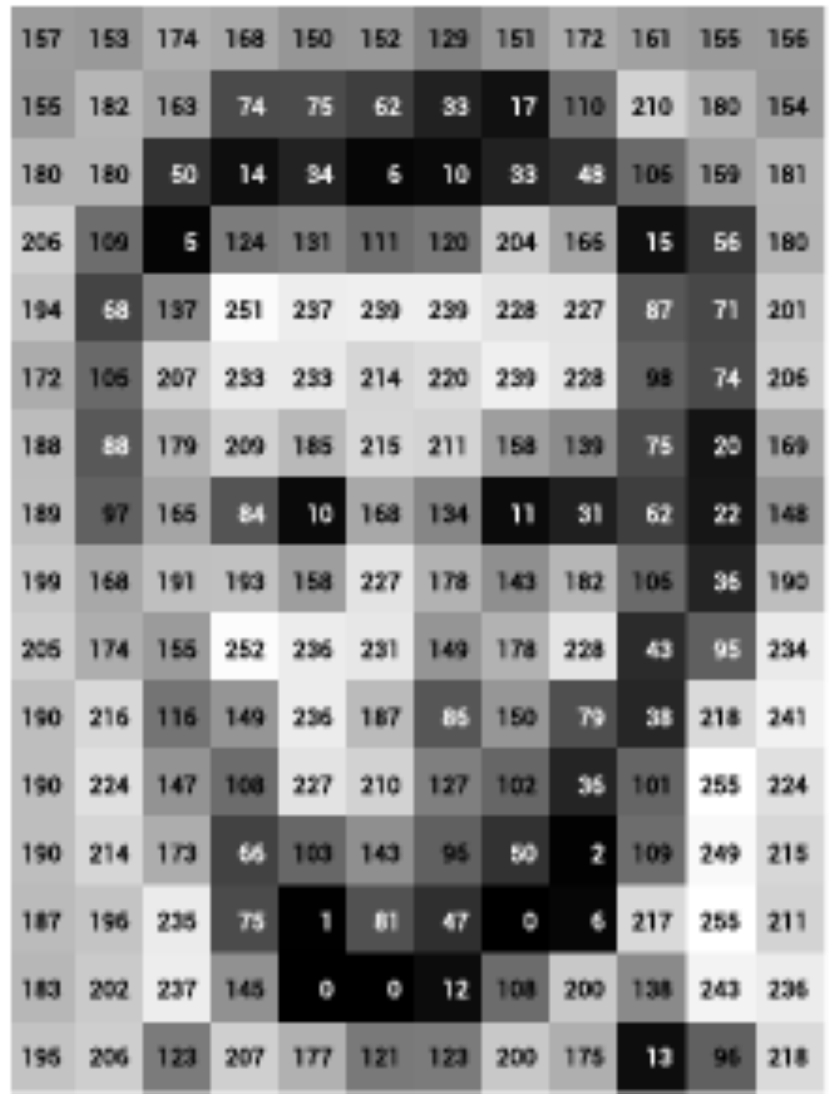


And,
Pixels
are
just
numbers

94	178	124	90	131	0
23	94	135	147	94	138
153	120	140	73	162	6
72	64	10	124	56	64
3	60	75	82	86	129
116	92	165	106	170	89

If an image is p pixels tall and q wide





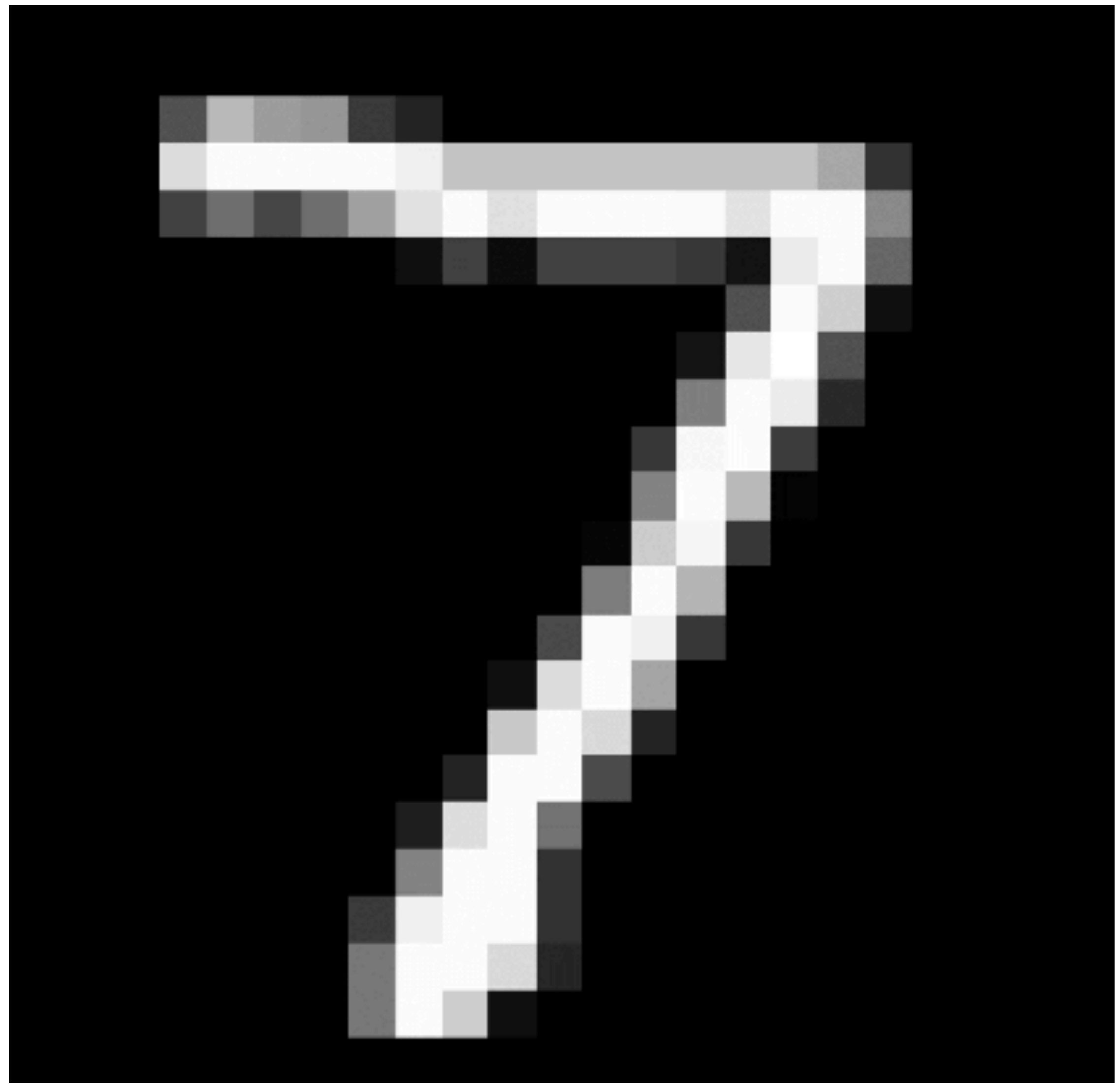
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

It's just an ordered list of pxq numbers!

For example...an HD image is an ordered list of 1920x1080, or 2,073,600 numbers

This is a vector in 2 million dimensional space!

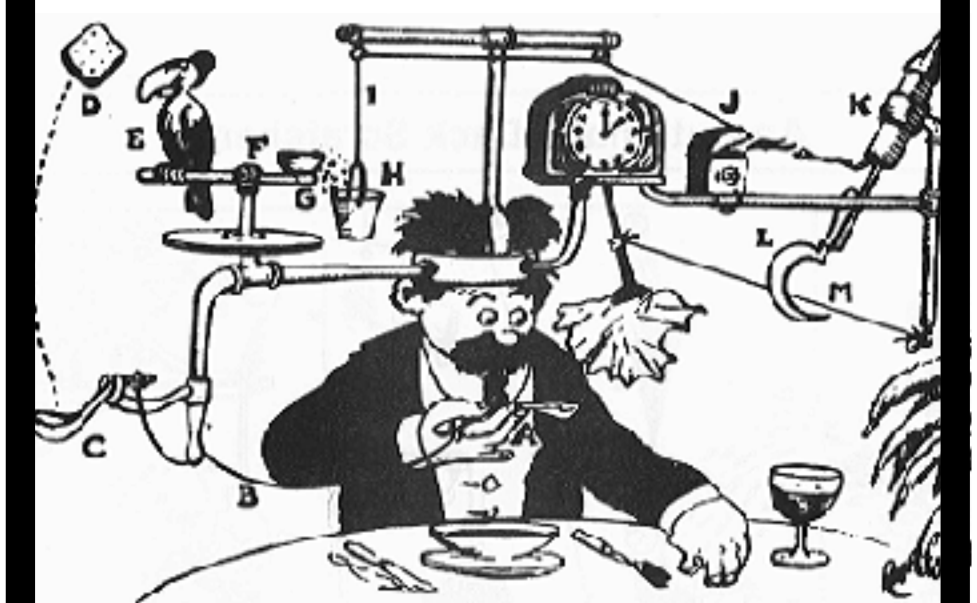
An image in the handwriting dataset is 28x28 pixels



This is a vector in 784-dimensional space.

A Reading Machine

A 784-dimensional vector



A vector

$$(p_1, p_2, p_3, \ldots, p_{26})$$

with $p_i \geq 0$

and

$$\sum_{i=1}^{26} p_i = 1$$

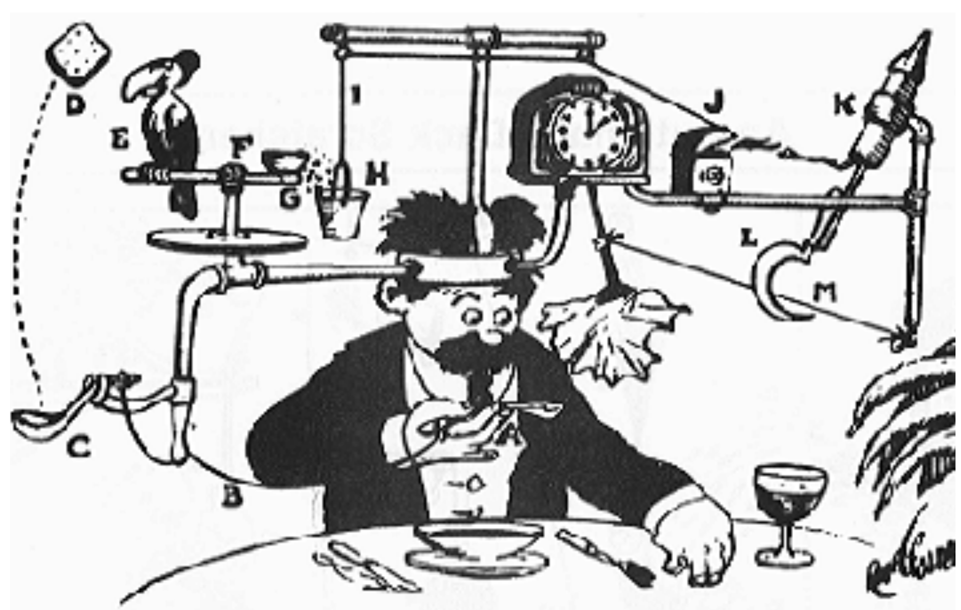
This sounds really daunting: there are so many high dimensions floating around.

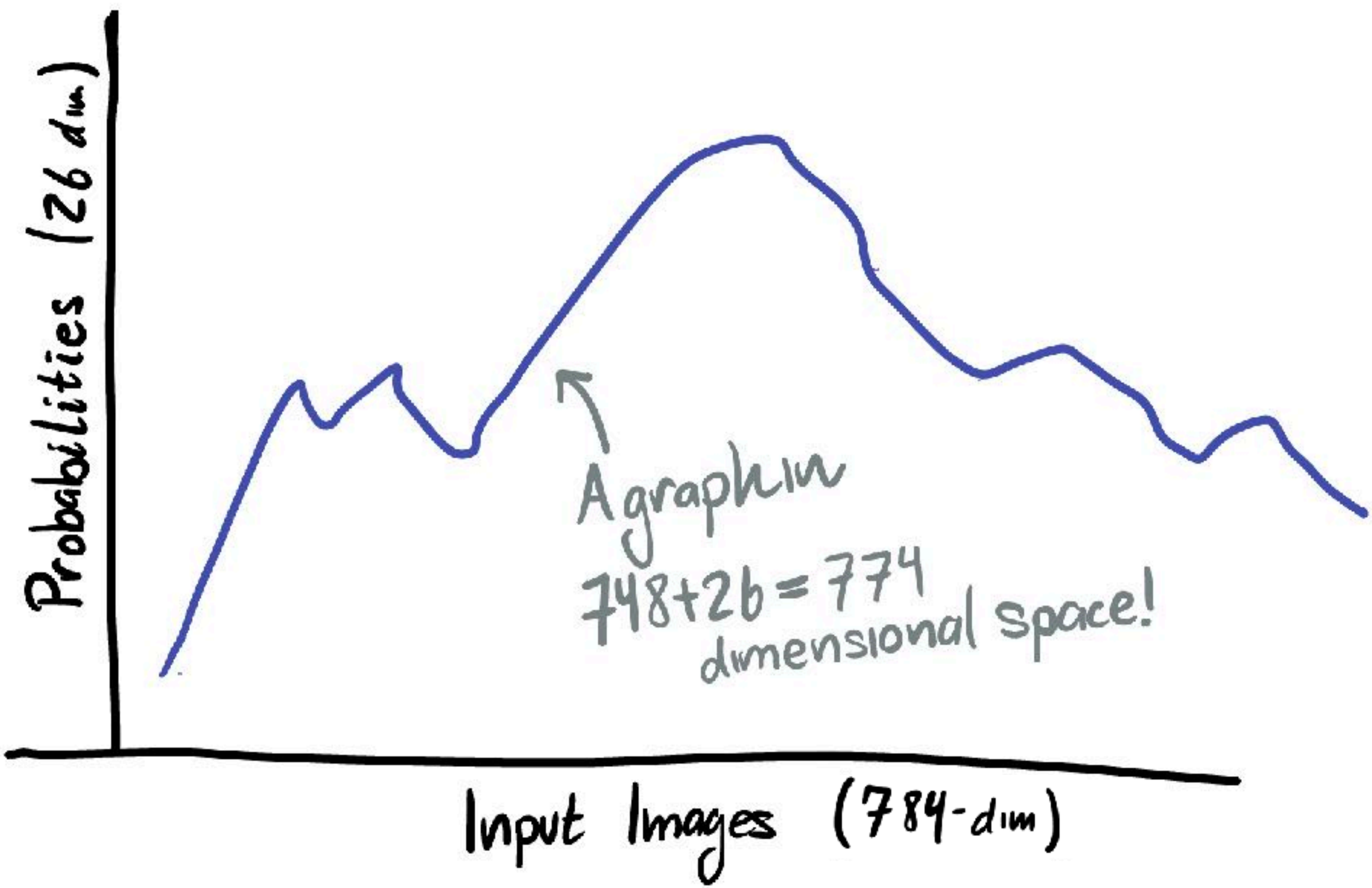
But your brain performs this function without trouble!

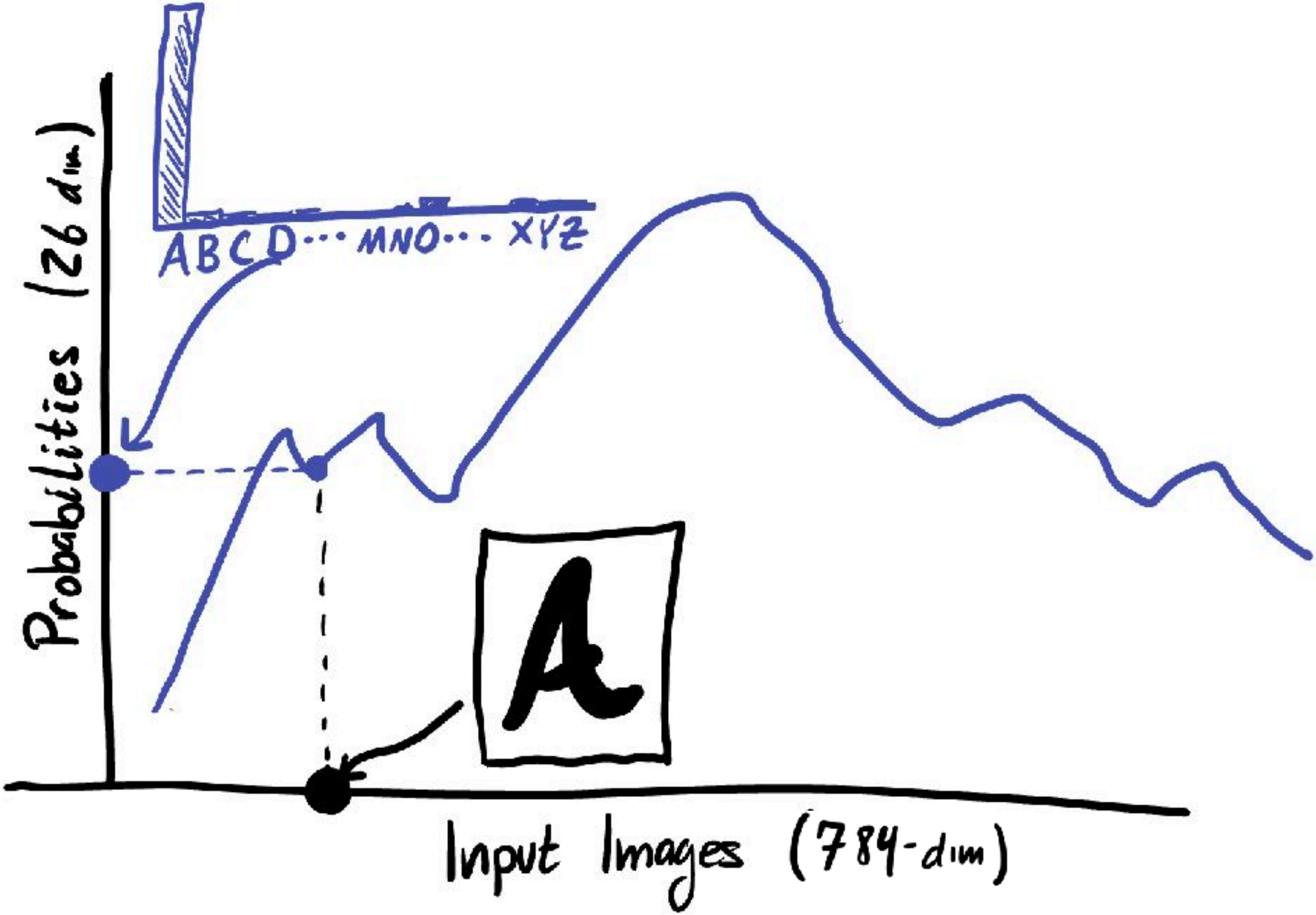
How can we draw a schematic of

The Reading Function

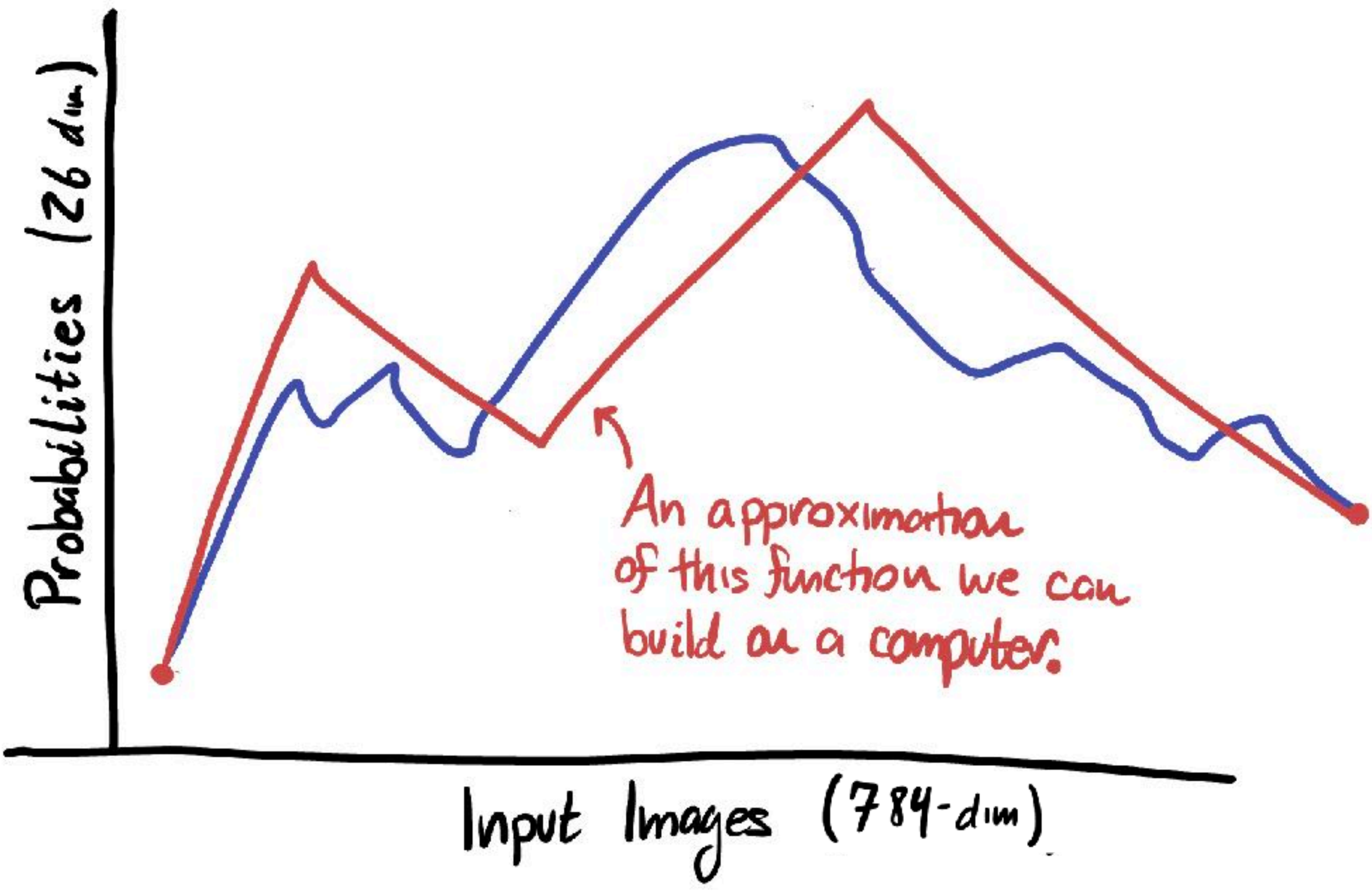
to reason about?

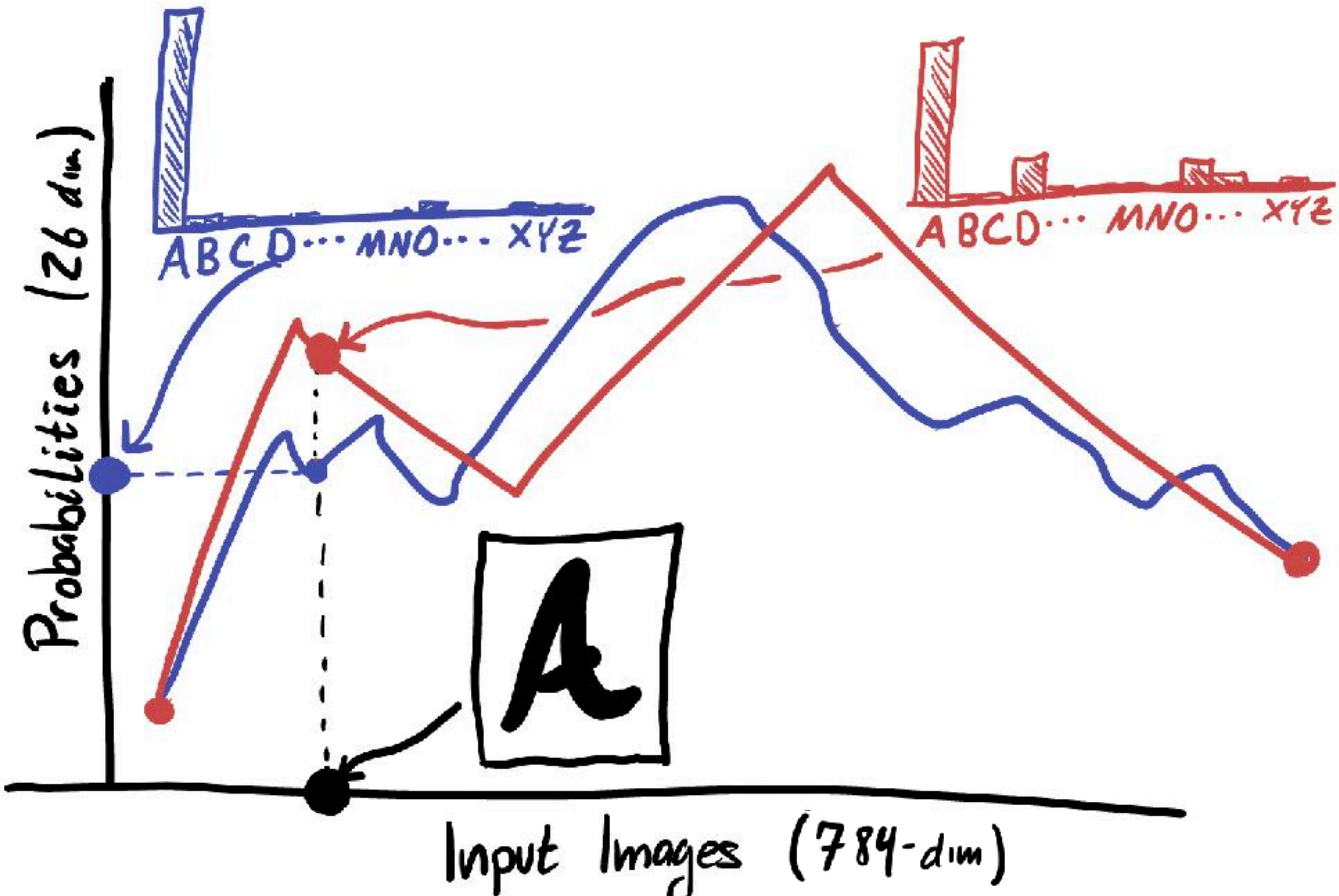



Probabilities (26 dim)
Agraphin
748+26 = 774
dimensional space!
Input Images (784-dim)



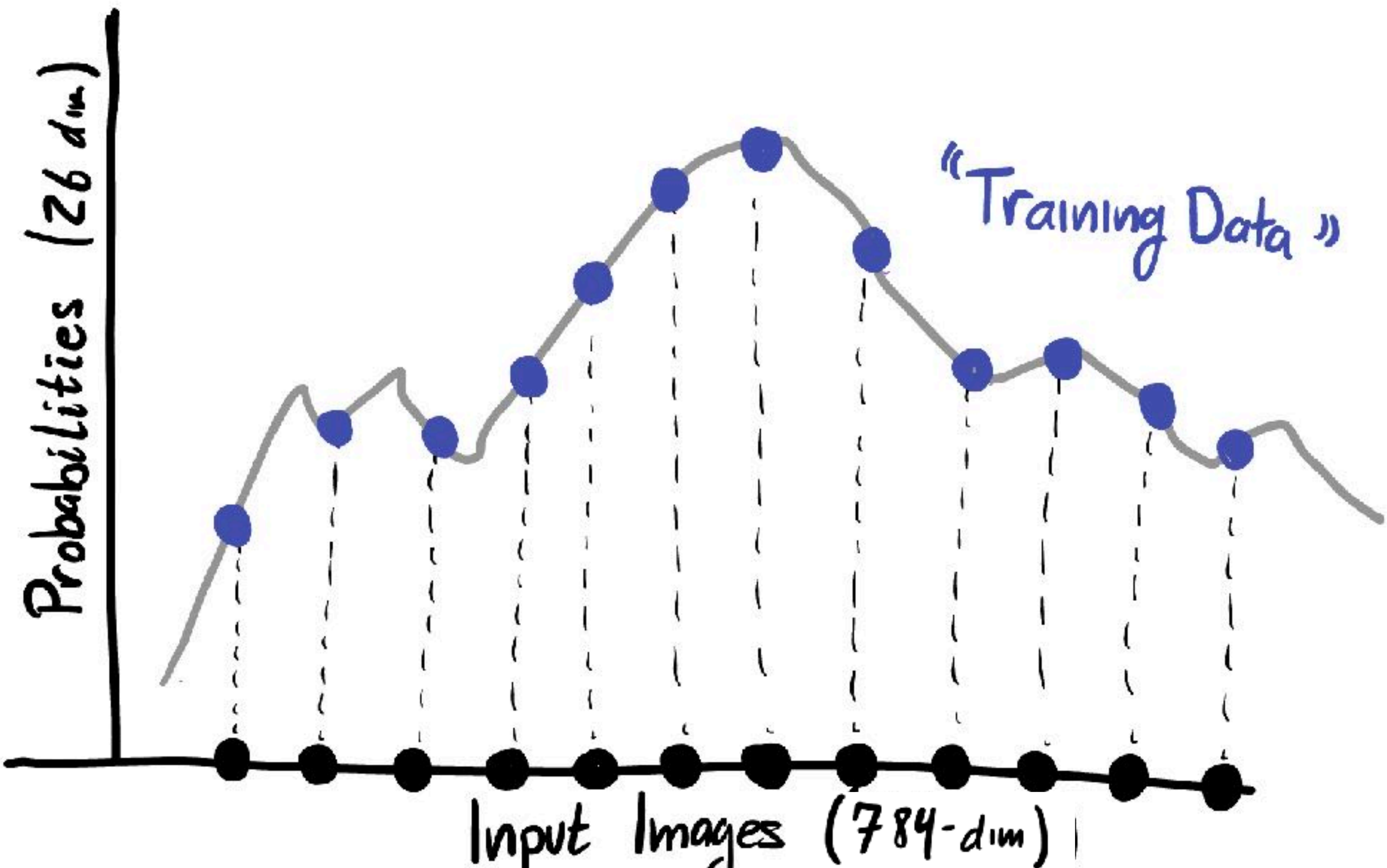
Probabilities (26 dim)
ABCD··· MNO··· XYZ
Input Images (784-dim)

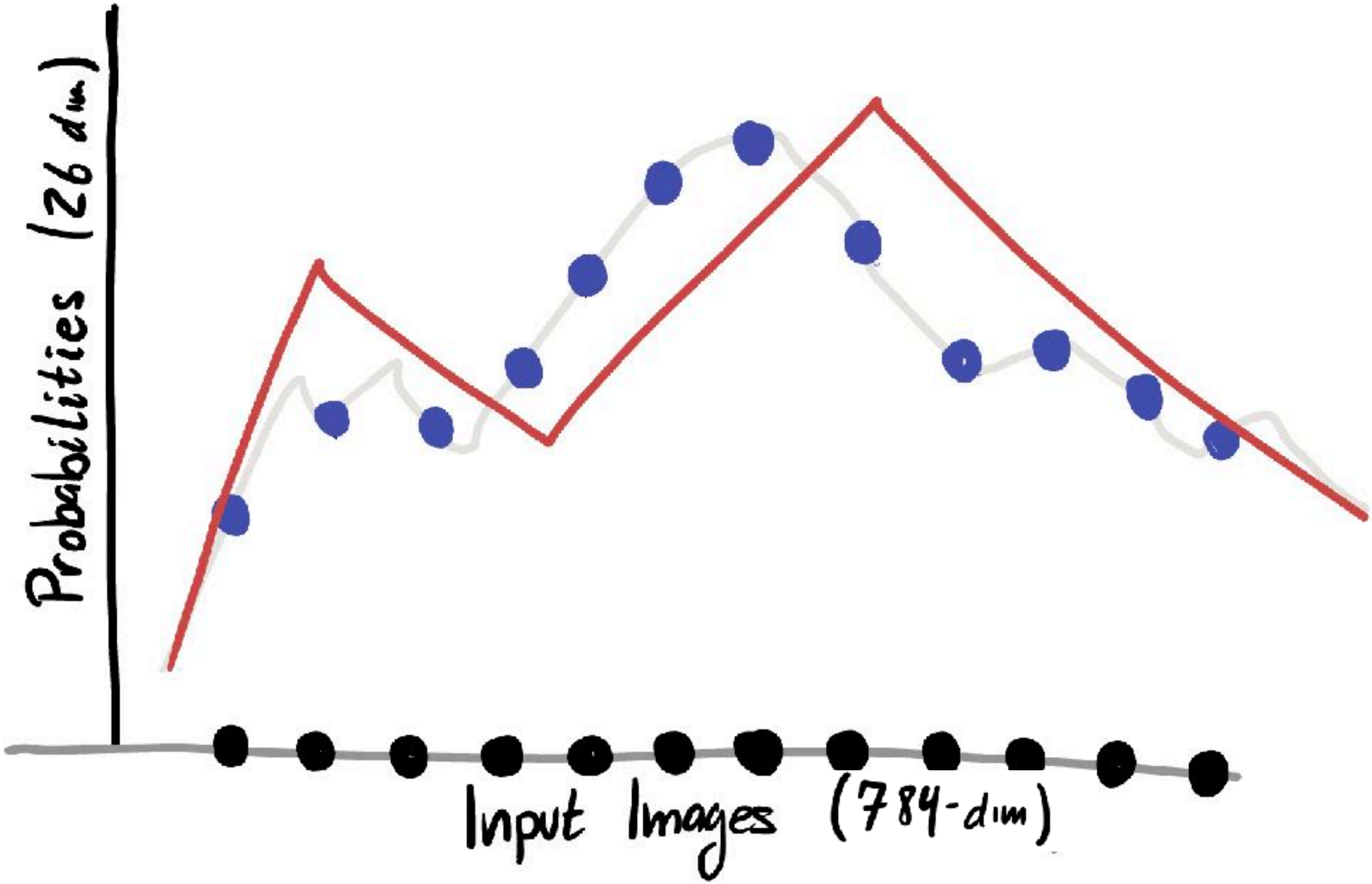

Probabilities (26 dim)
An approximation of this function we can build on a computer.
Input Images (784-dim)

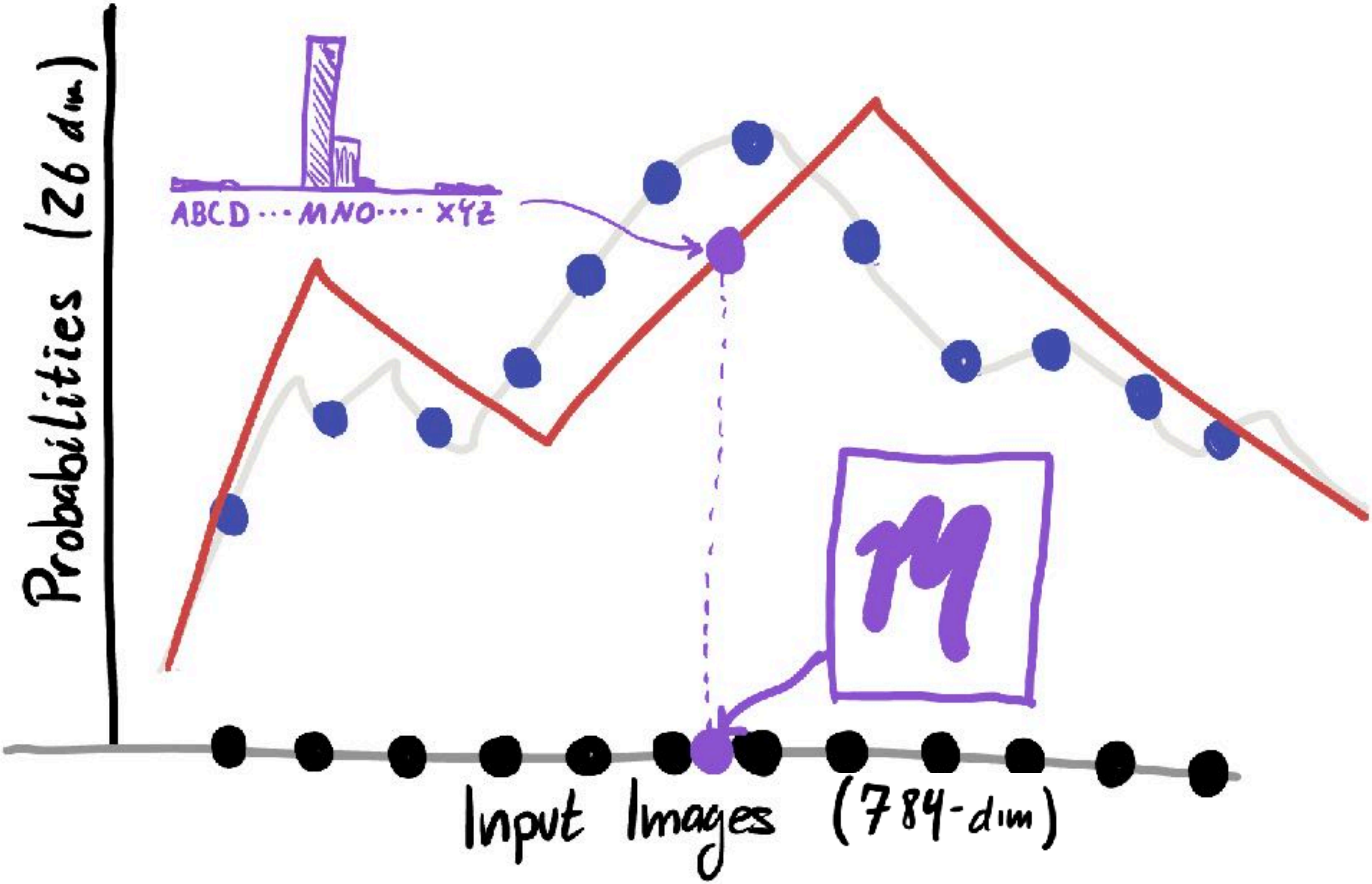


Probabilities (26 dim)
ABCD··· MNO··· XYZ
ABCD··· MNO··· XYZ
A
Input Images (784-dim)


Wait….we don't actually have access to the entire "true" function though, do we?

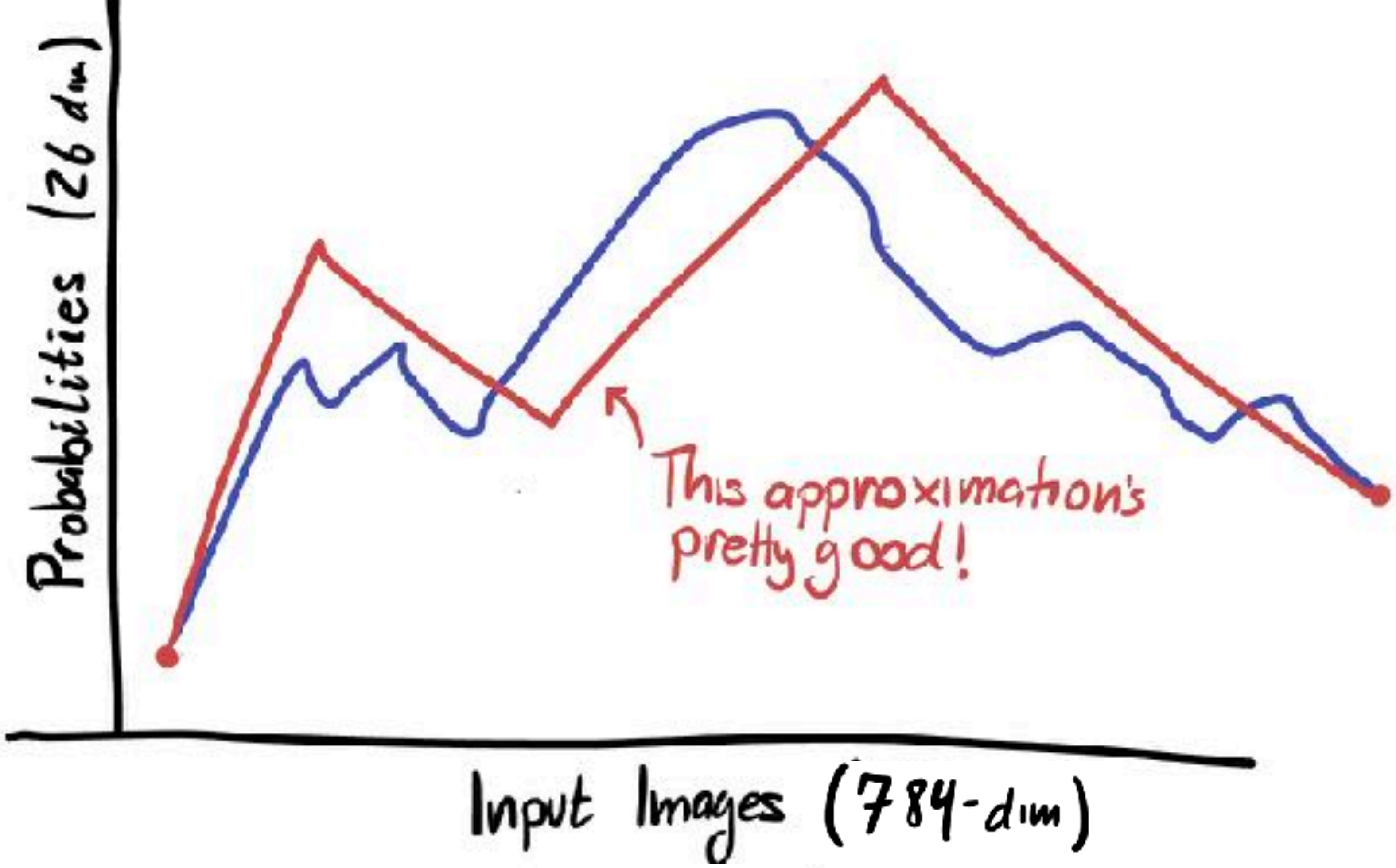

Probabilities (26 dim)
"Training Data"
Input Images (784-dim)

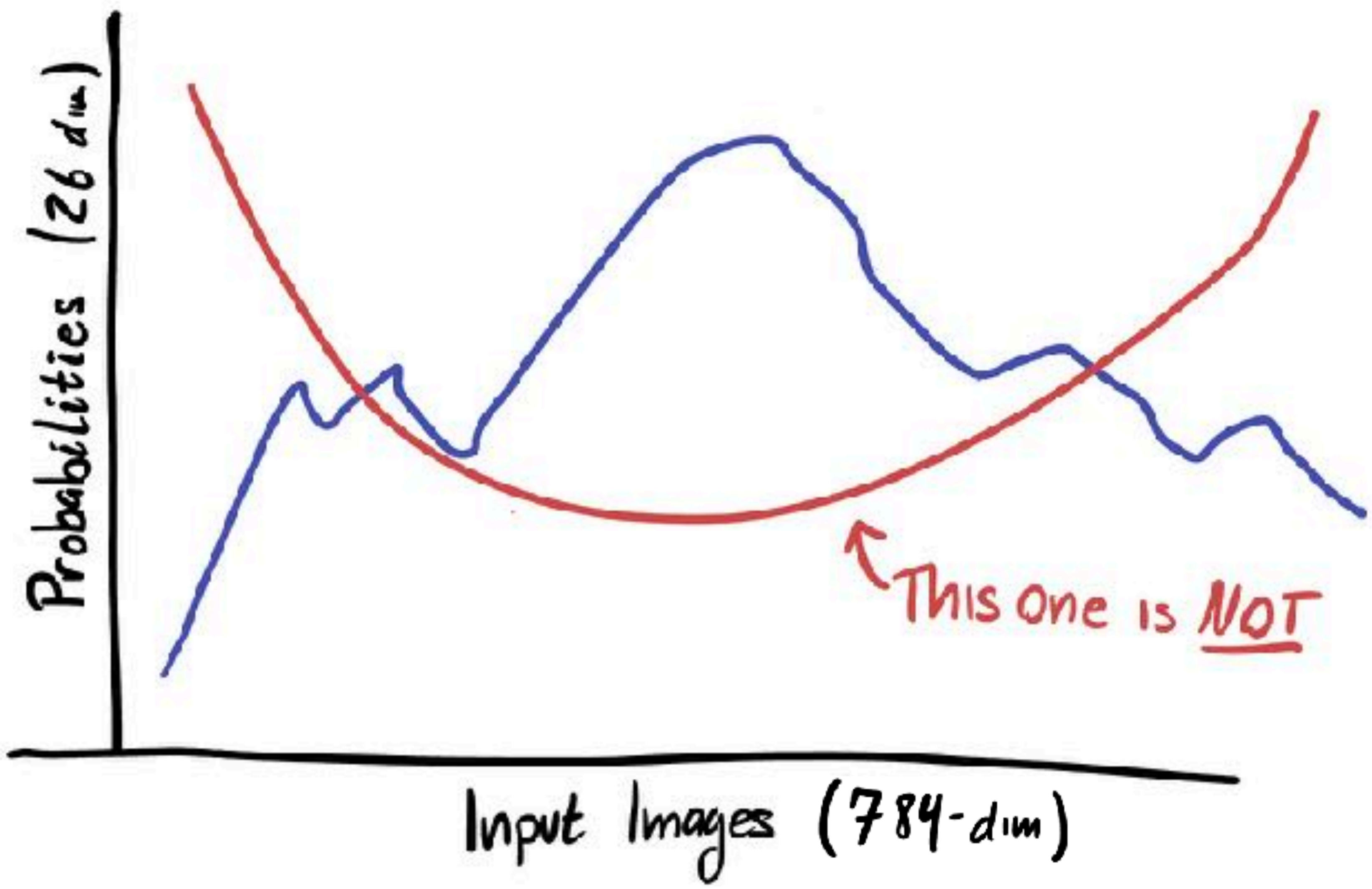

Probabilities (26 dim)
Input Images (784-dim)

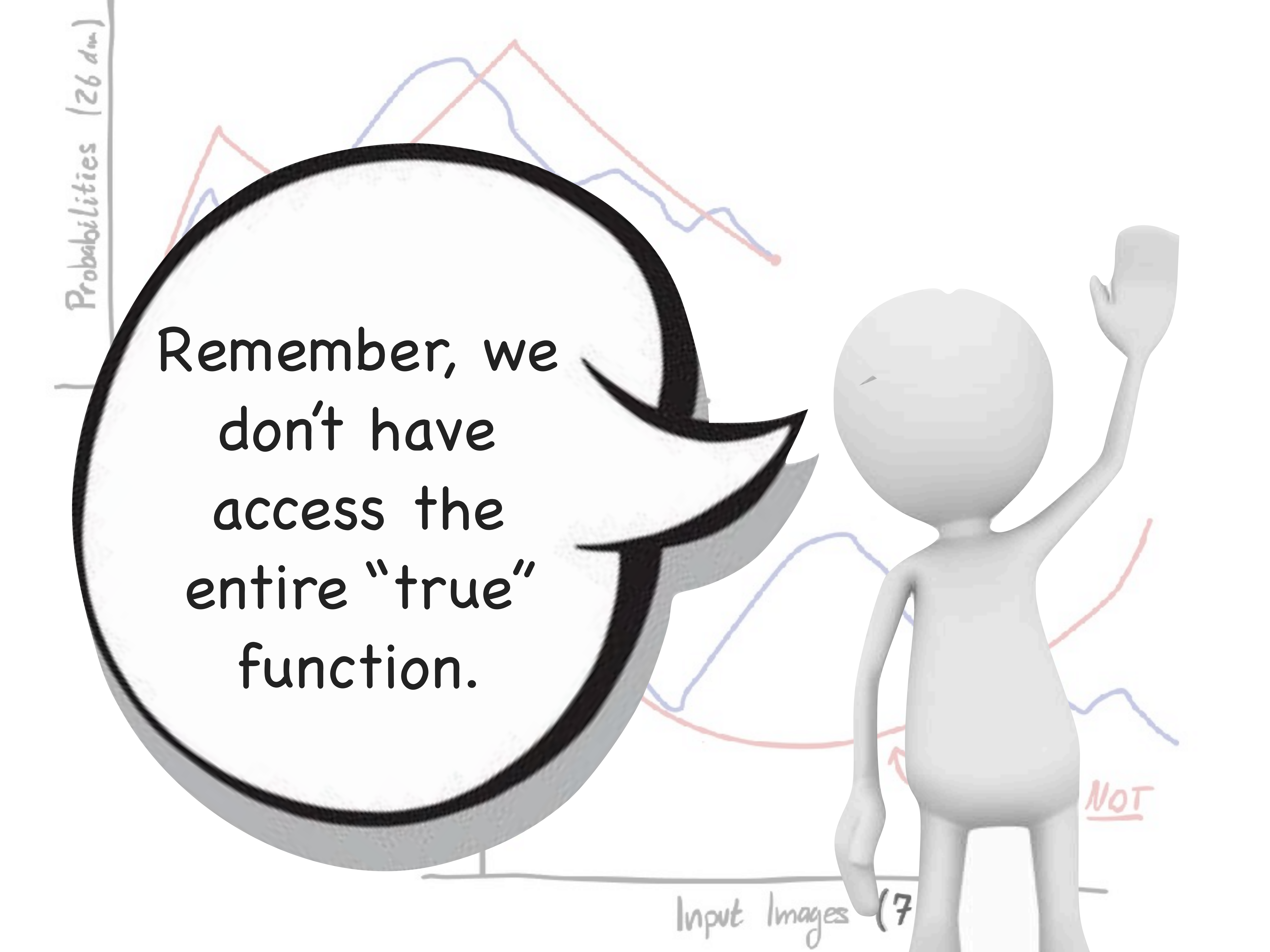

Probabilities (26 dim)
ABCD ··· MNO ···· XYZ
M
Input Images (784-dim)


How do we build an approximation?
If we make some approximation, how can we tell if its any good?

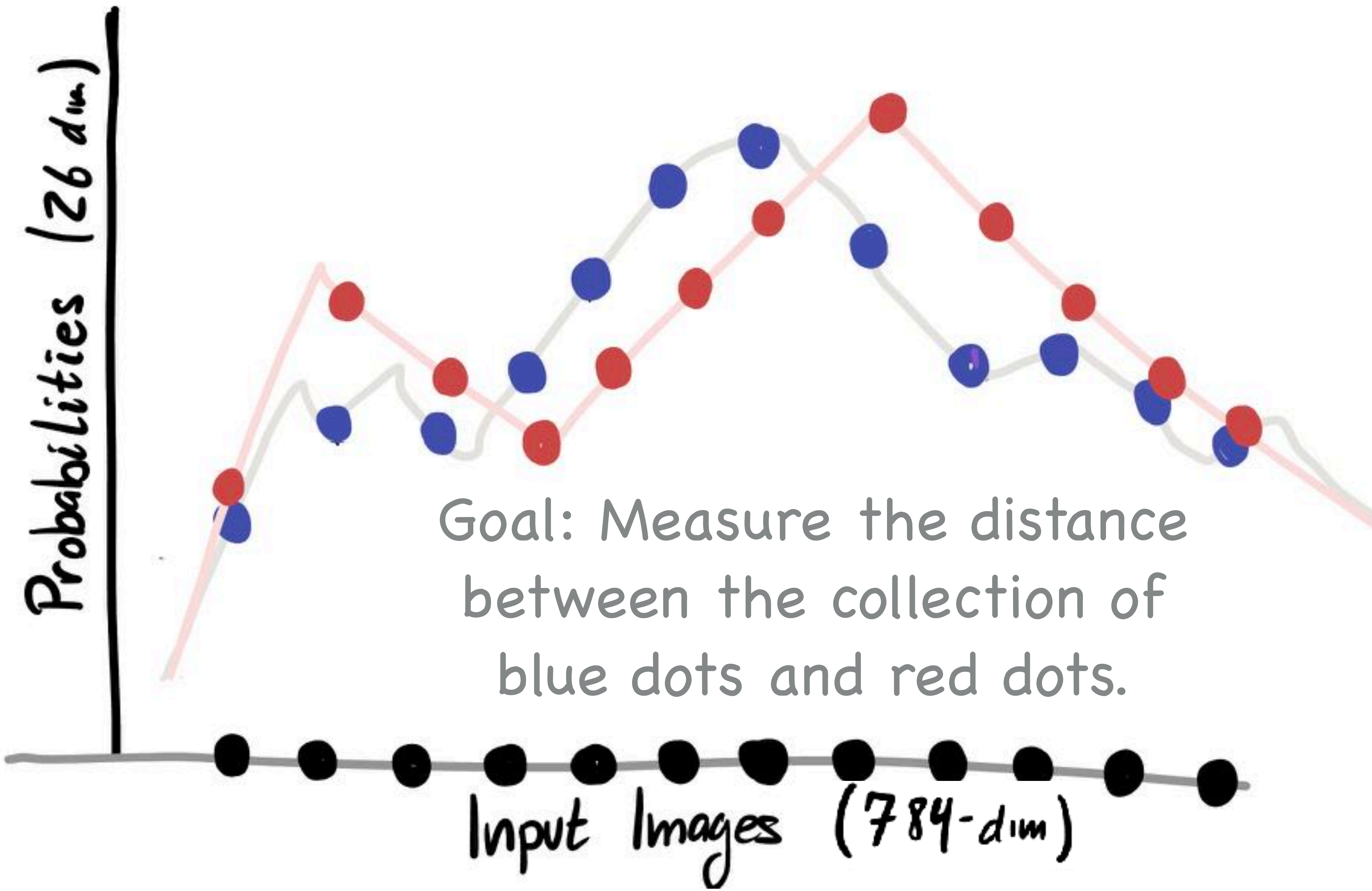

How do we build an approximation?
If we have some approximation, how can we tell if its any good?


Probabilities (26 dim)
This approximation's pretty good!
Input Images (784-dim)


Probabilities (26 dim)
This one is NOT
Input Images (784-dim)

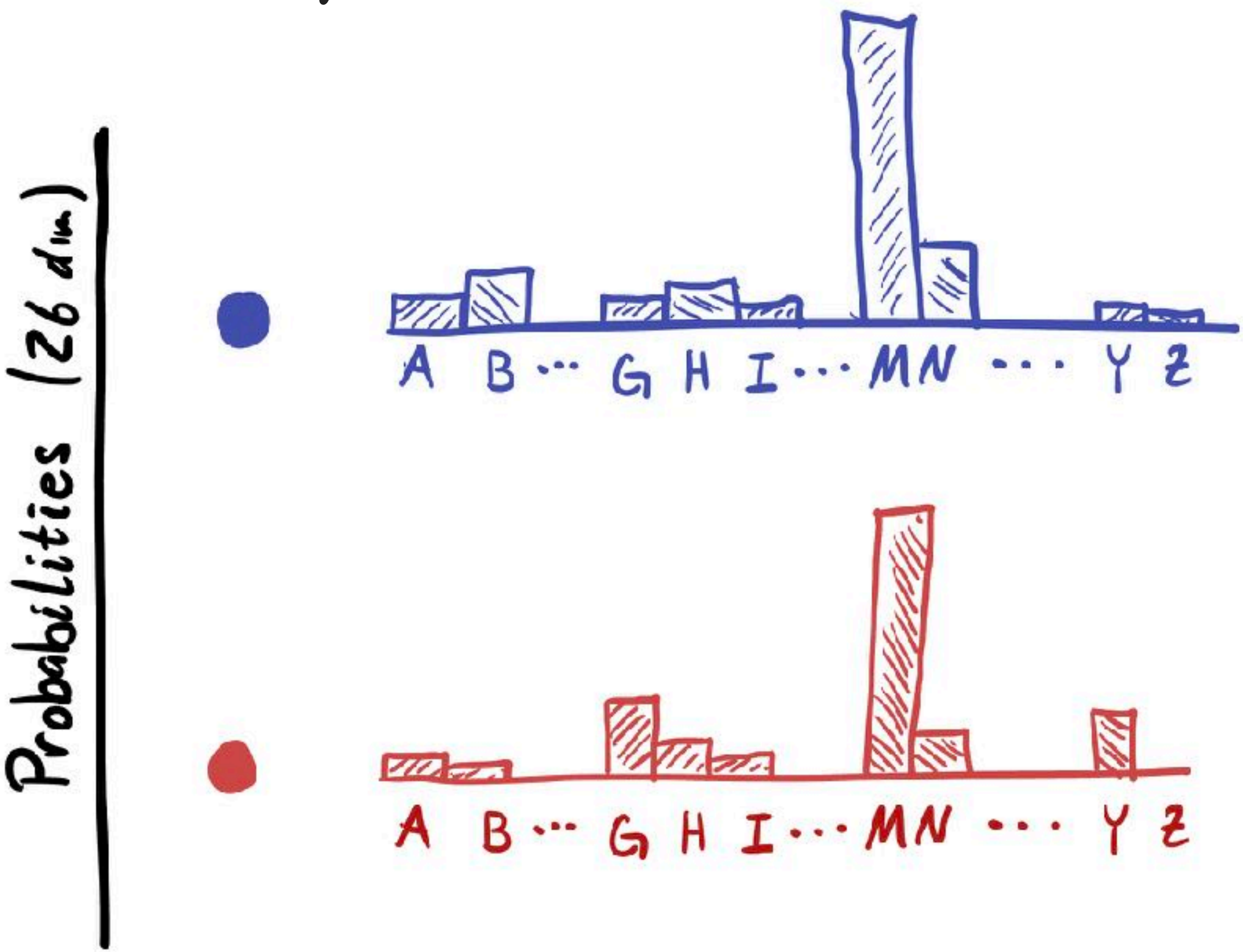

Remember, we don't have access the entire "true" function.
Probabilities
Input Images
NOT

The Loss Function

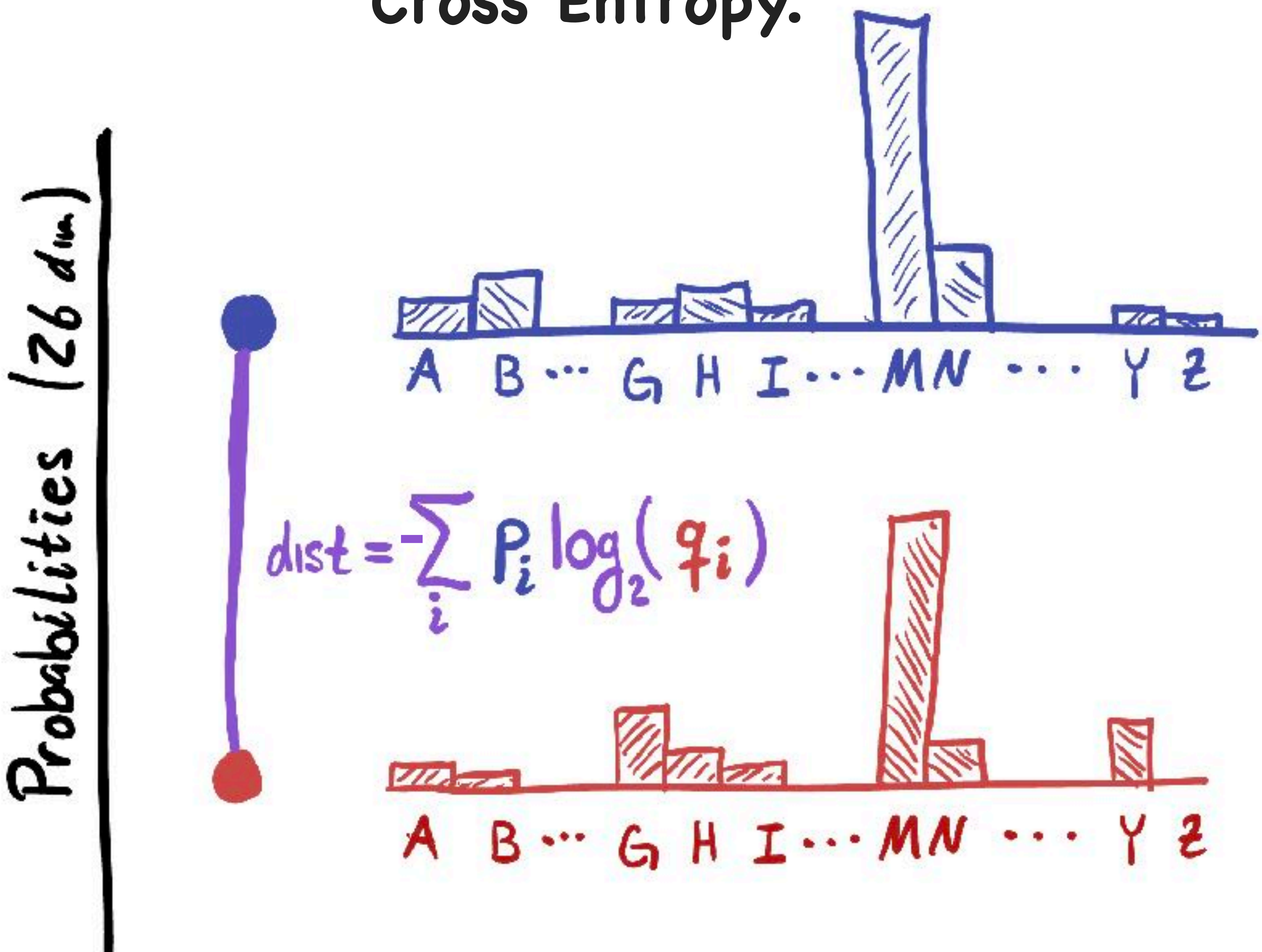


Each dot is a

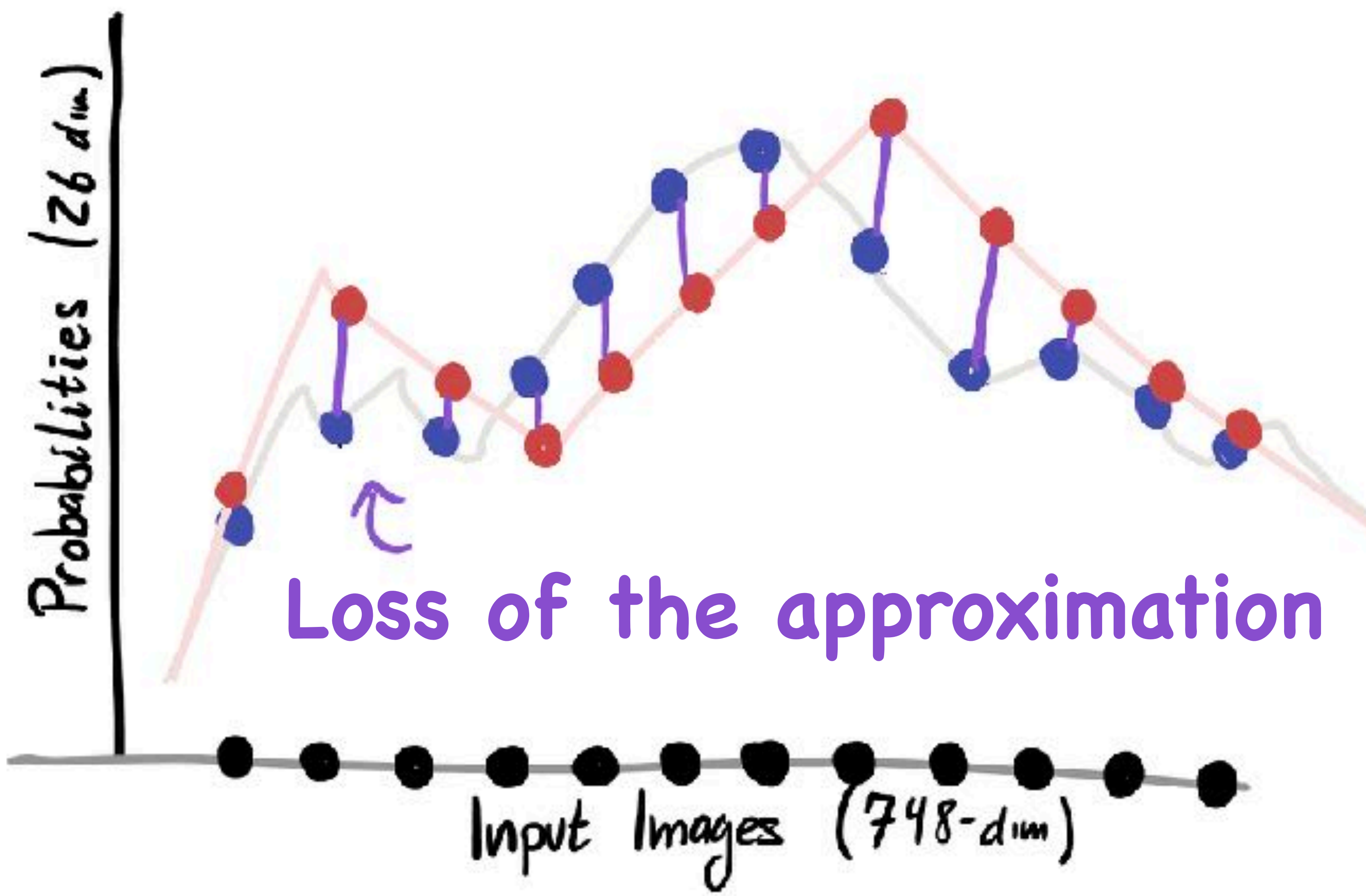
Probability Distribution!

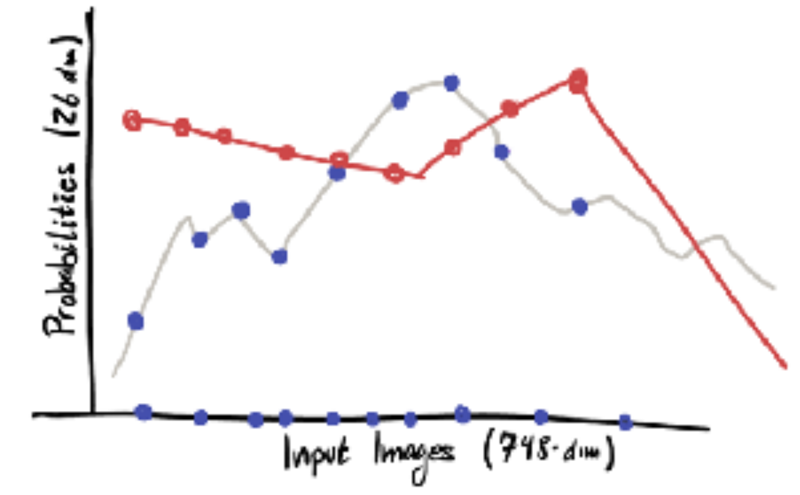


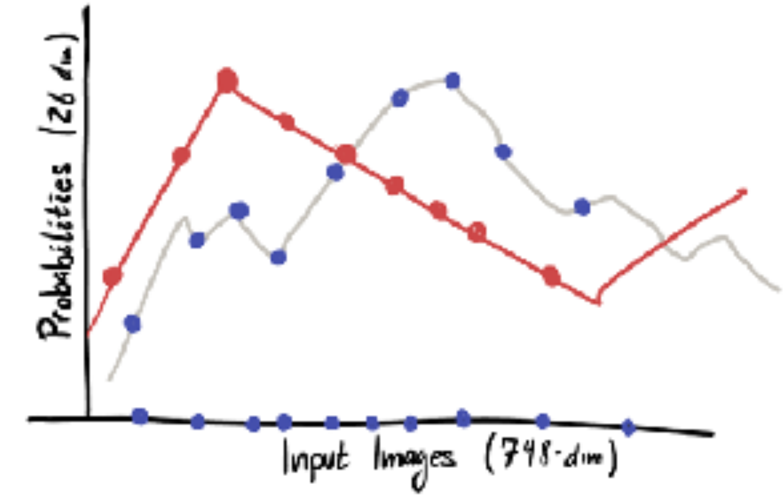
Luckily, mathematicians know how to compare probability distributions. One option is the **Cross Entropy.**

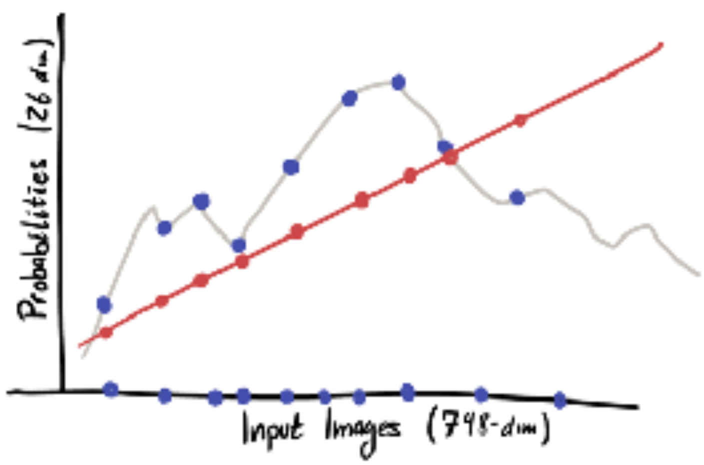


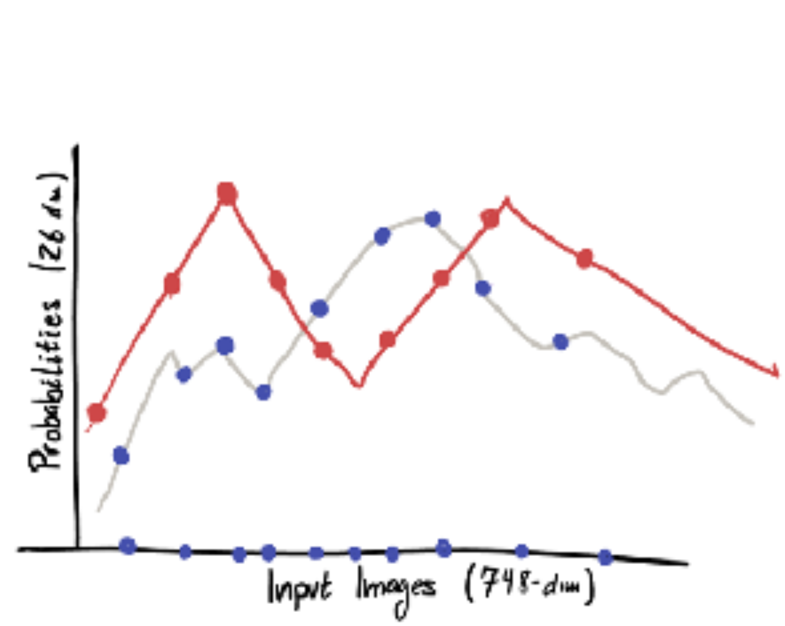
The difference between our "true function" and the approximation we are considering is called the "loss" - its how much info our approximation has lost.

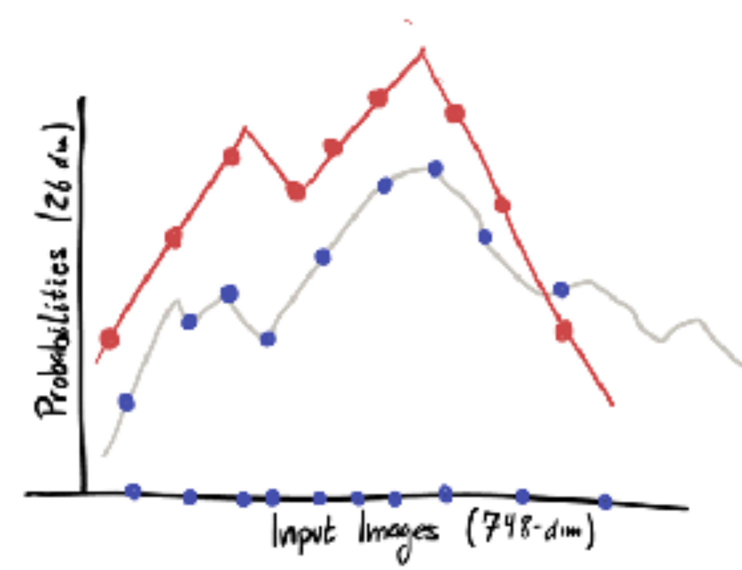


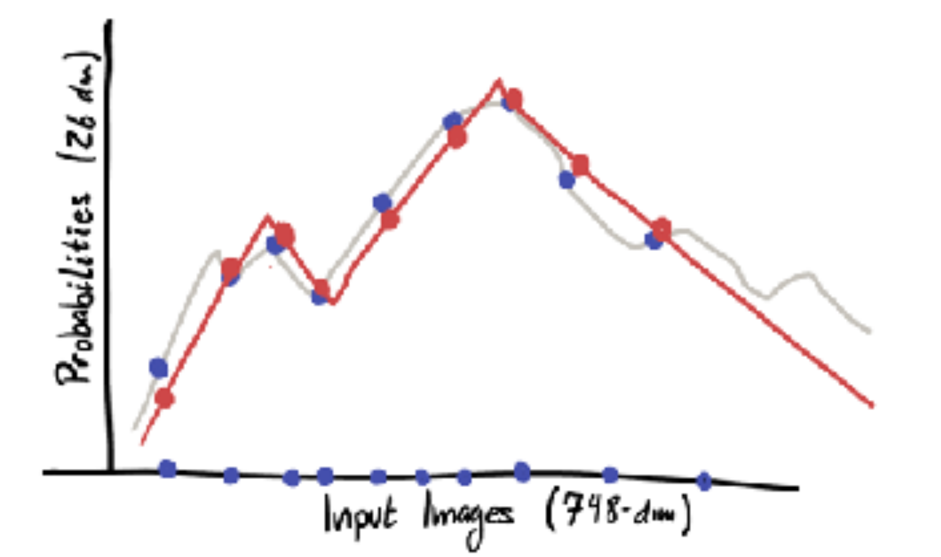

Probabilities (26 dim)
Input Images (748-dim)

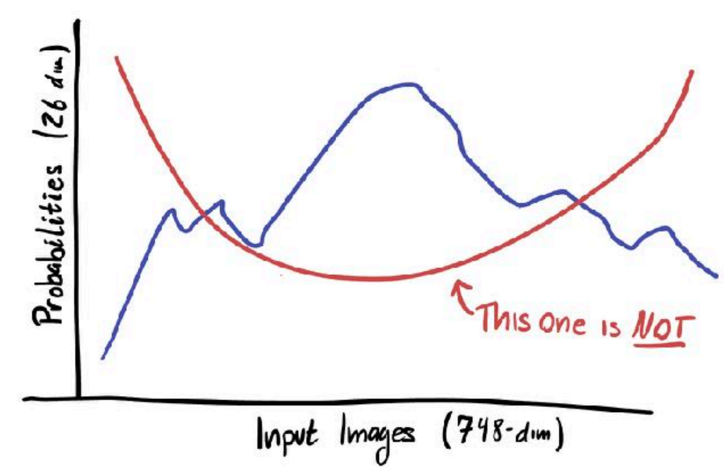

Probabilities (26 dim)
Input Images (748-dim)

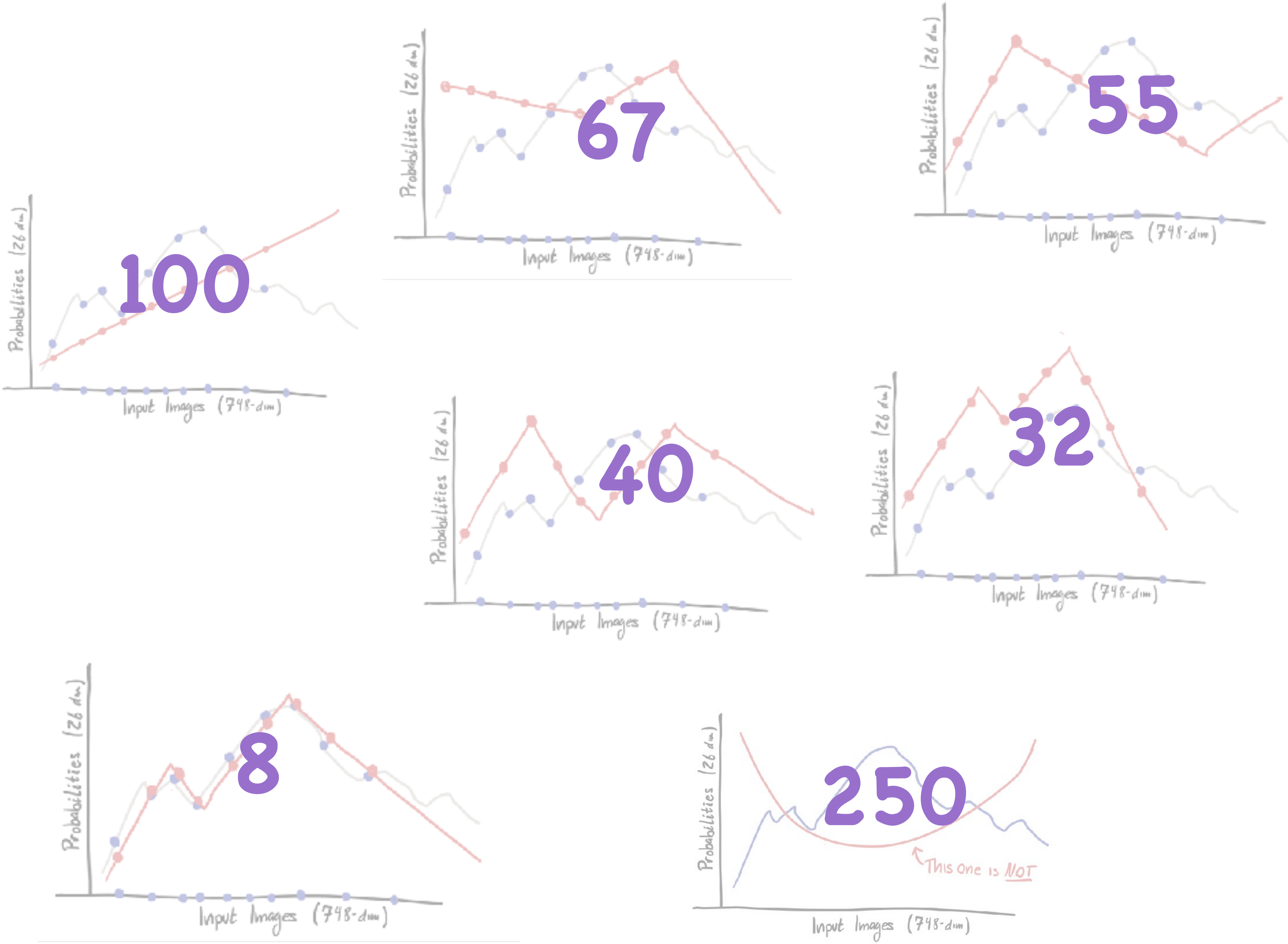

Probabilities (26 dim)
Input Images (748-dim)


Probabilities (26 dim)
Input Images (748-dim)


Probabilities (26 dim)
Input Images (748-dim)


Probabilities (26 dim)
Input Images (748-dim)

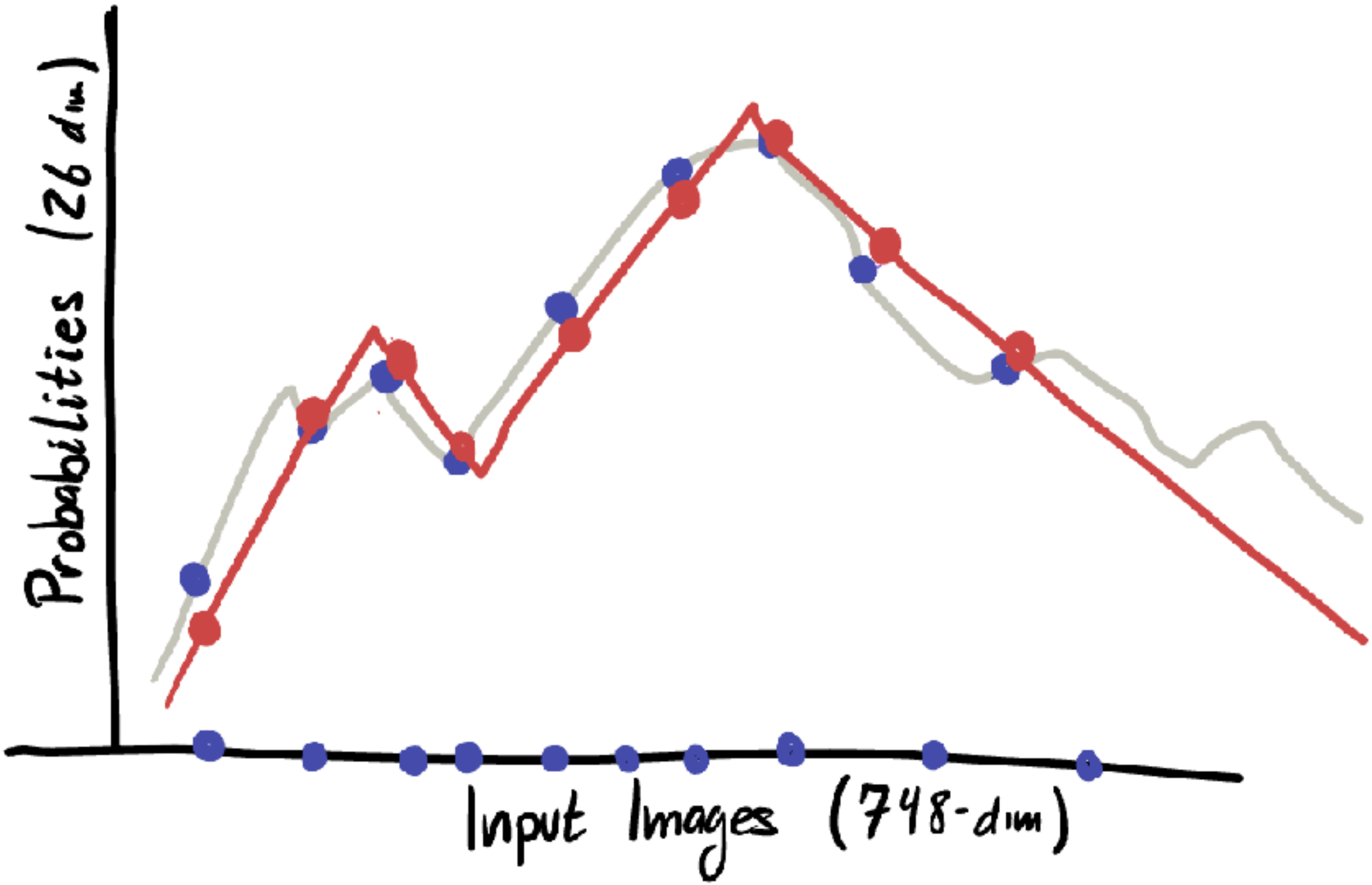

Probabilities (26 dim)
This one is NOT
Input Images (748-dim)

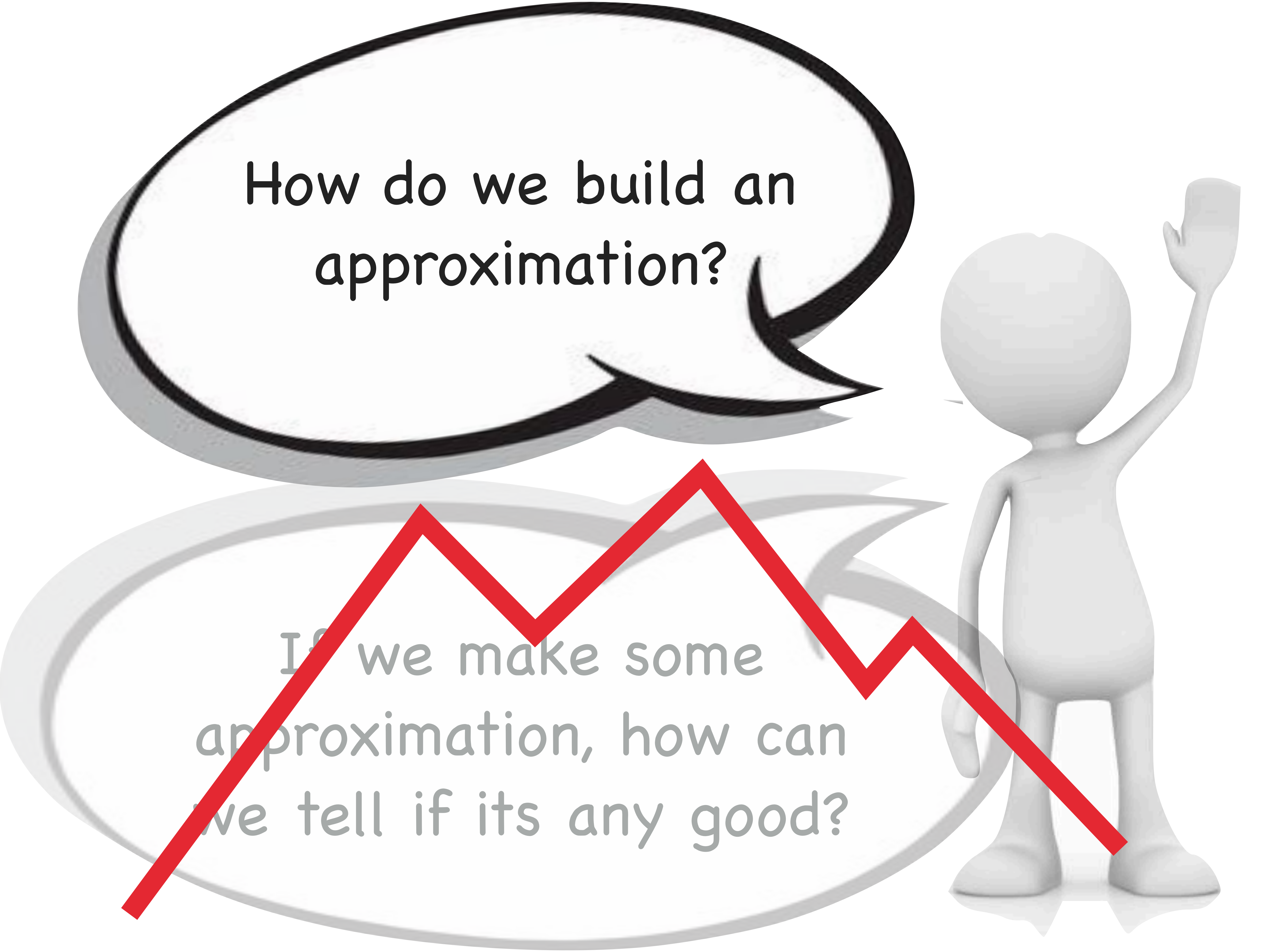


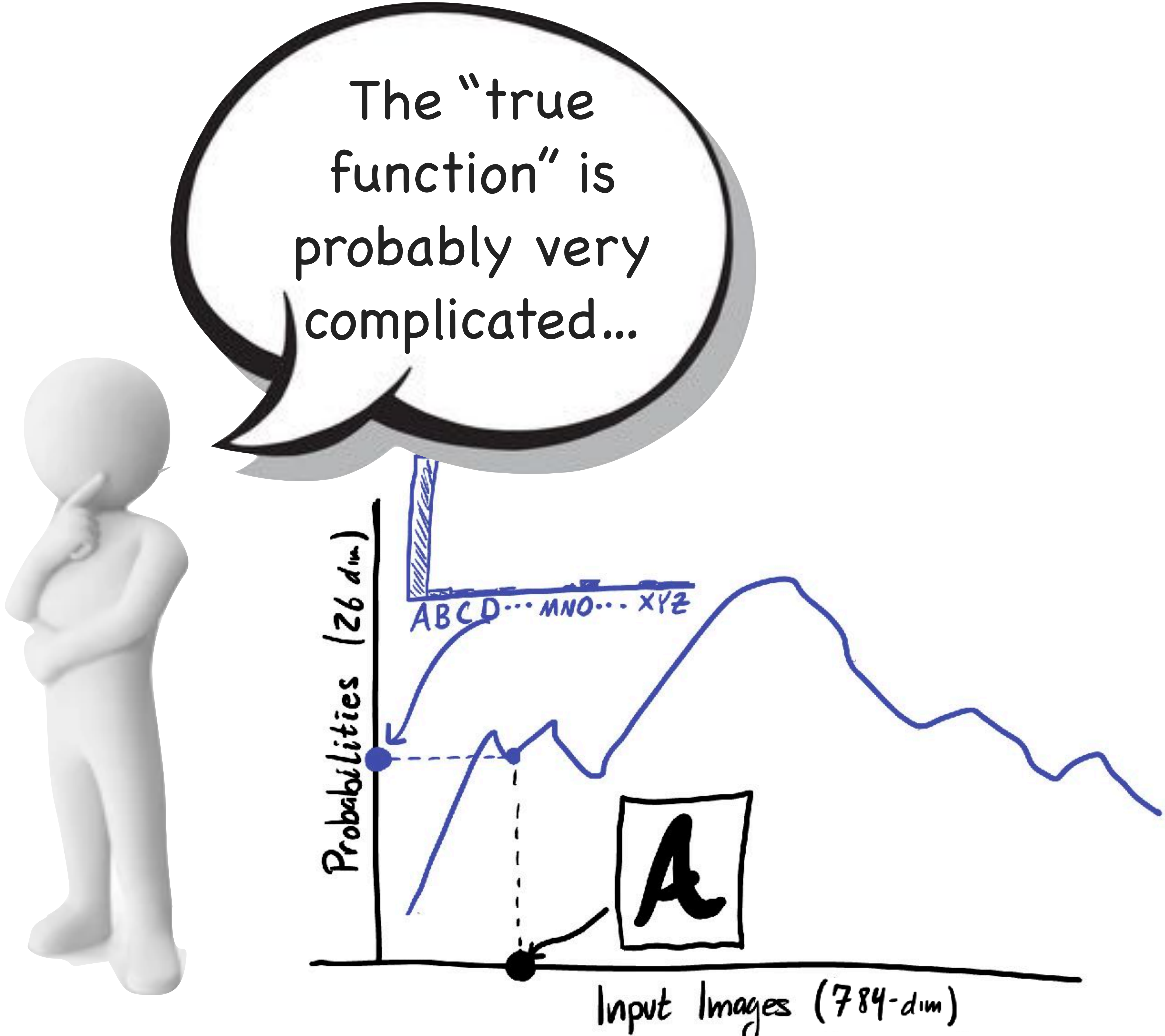
100
67
55
40
32
8
250
Probabilities (26-dim)
Input Images (748-dim)
This one is NOT

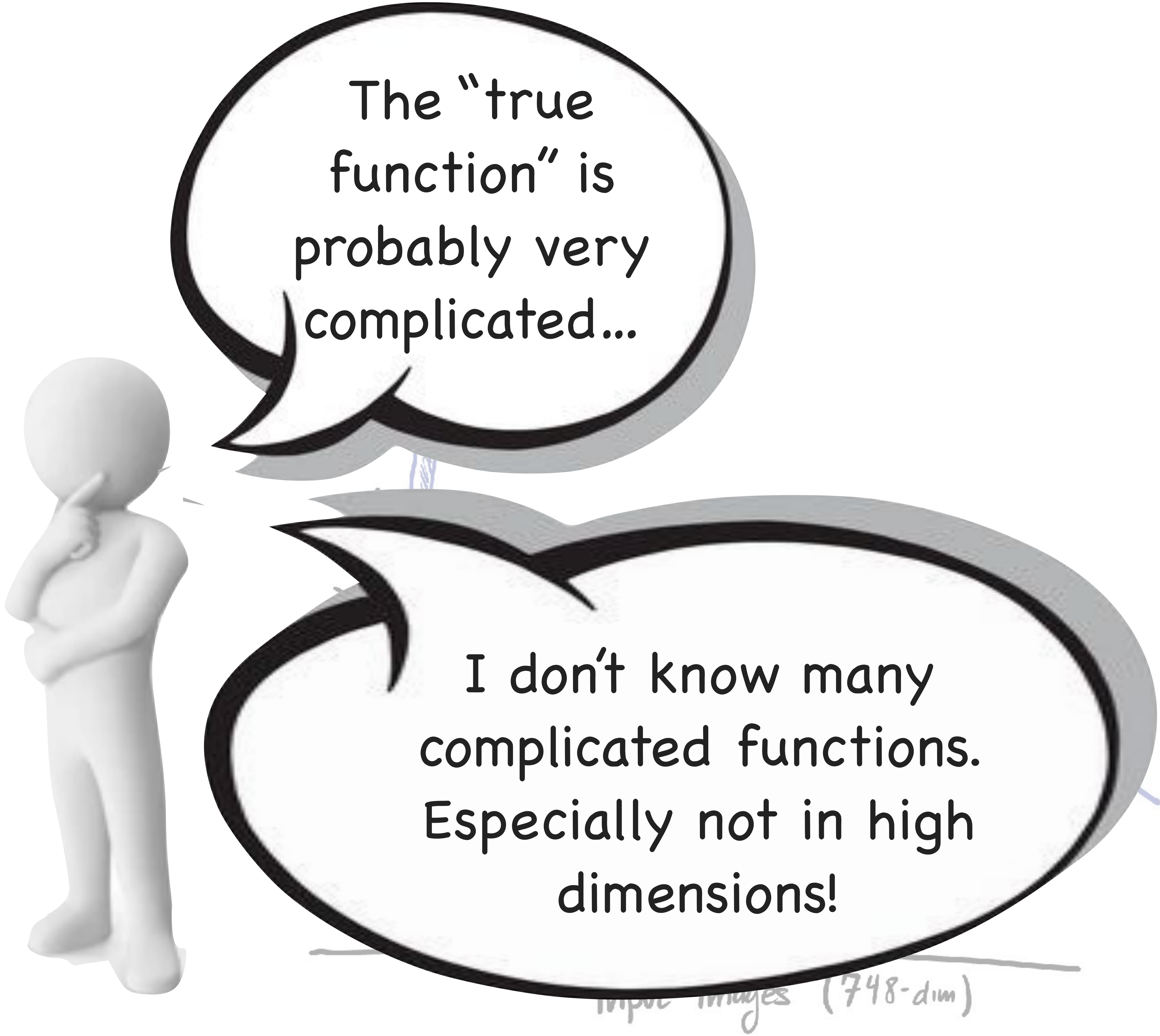
The best approximation is the one that minimizes the loss function.

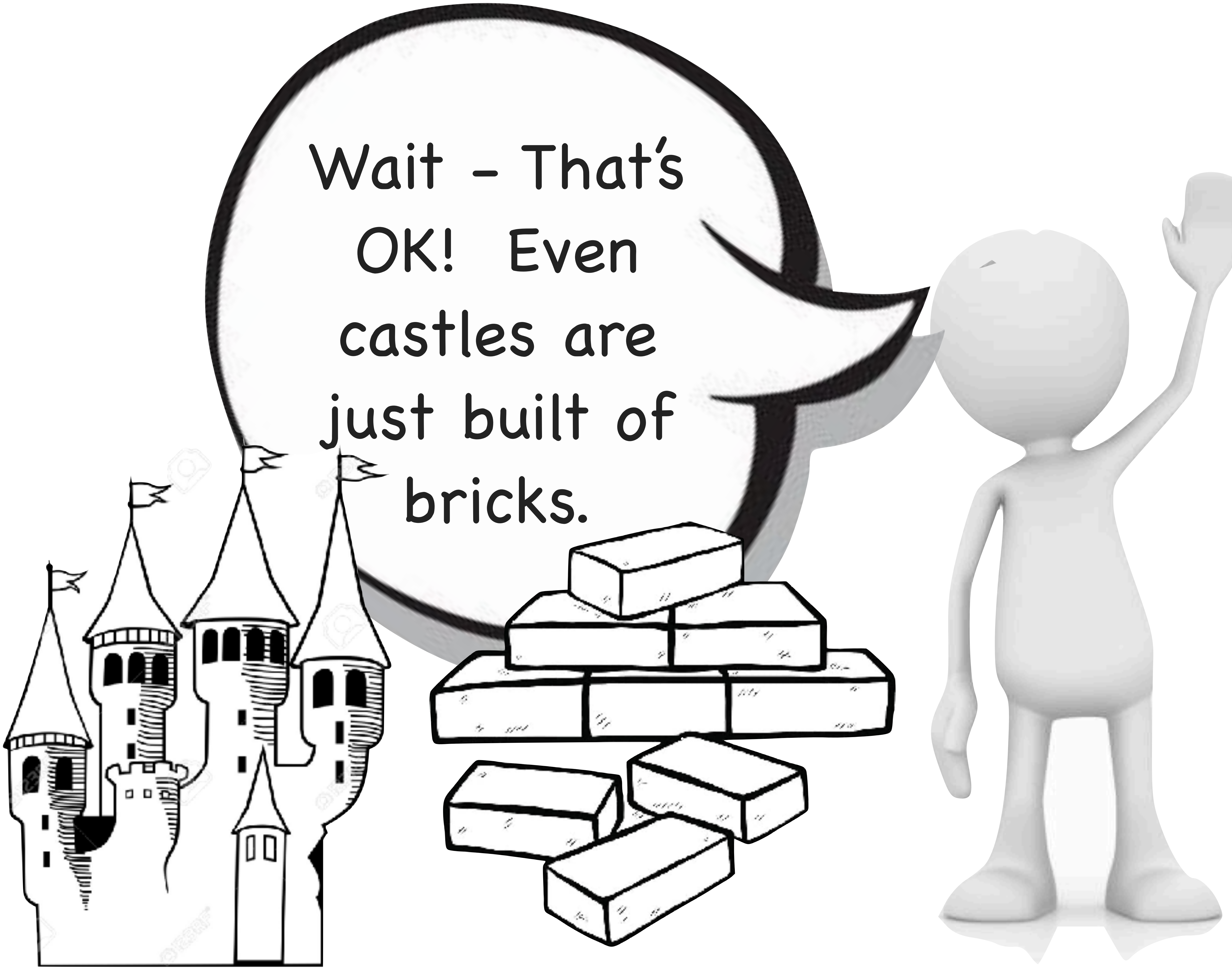
8



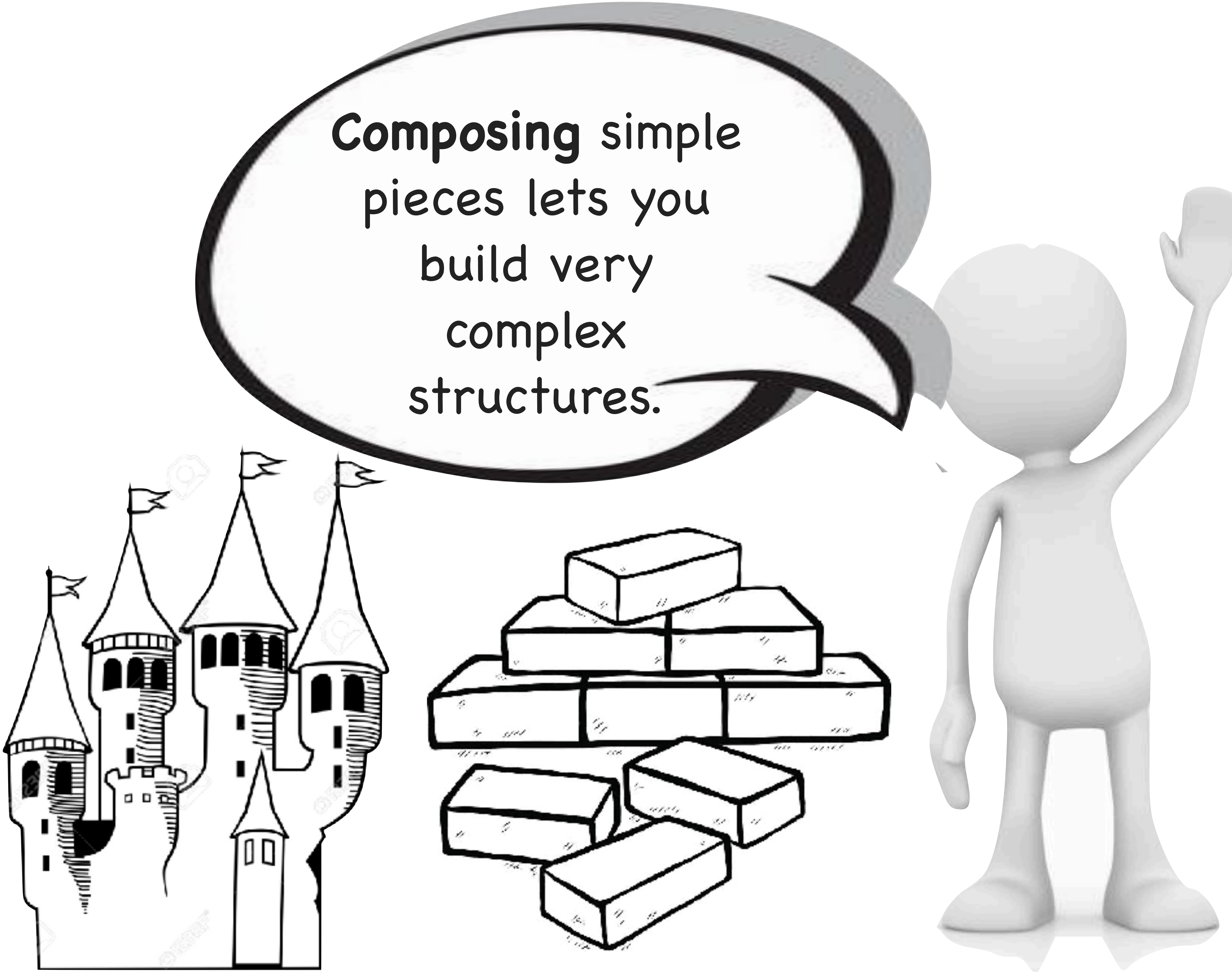

How do we build an approximation?
If we make some approximation, how can we tell if its any good?


The “true function” is probably very complicated…
Probabilities (26 dim)
ABCD··· MNO··· XYZ
Input Images (784-dim)


The "true function" is probably very complicated...
I don't know many complicated functions. Especially not in high dimensions!

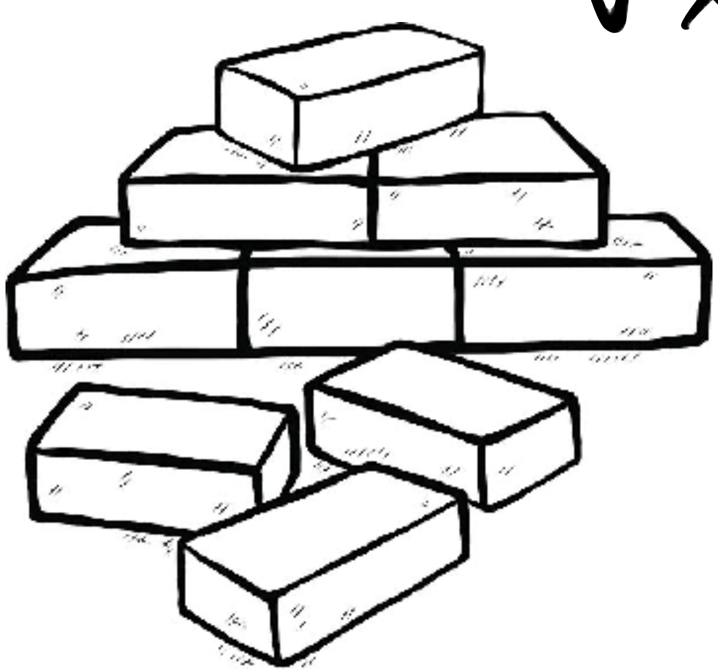

Wait – That's OK! Even castles are just built of bricks.

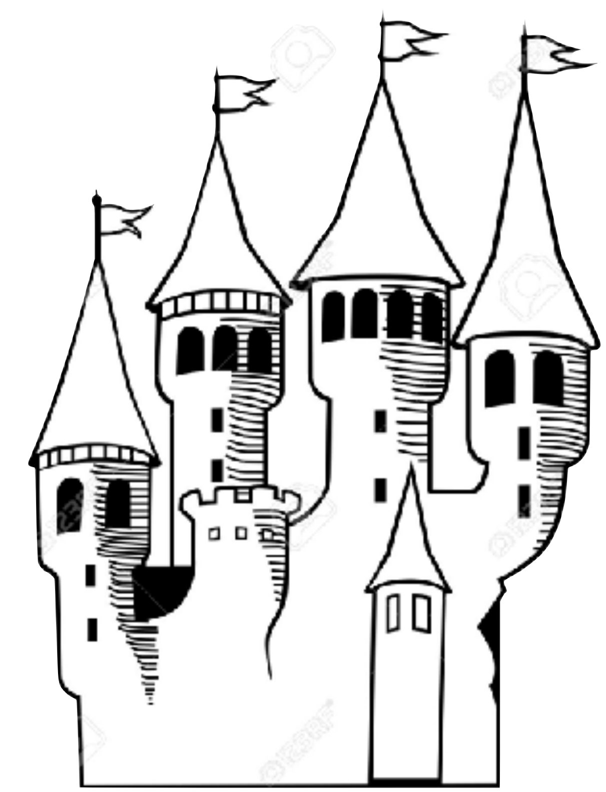
Wait – That's
OK! Even
are
?


Composing simple pieces lets you build very complex structures.

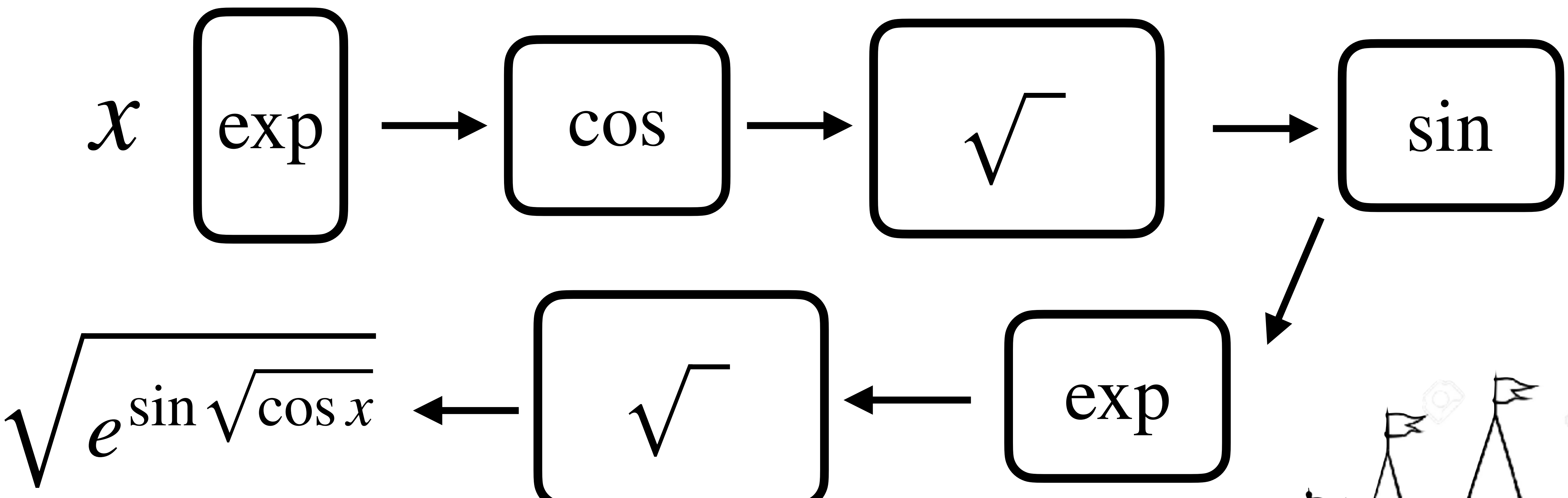
Function composition is a way to build complicated functions from easy pieces:

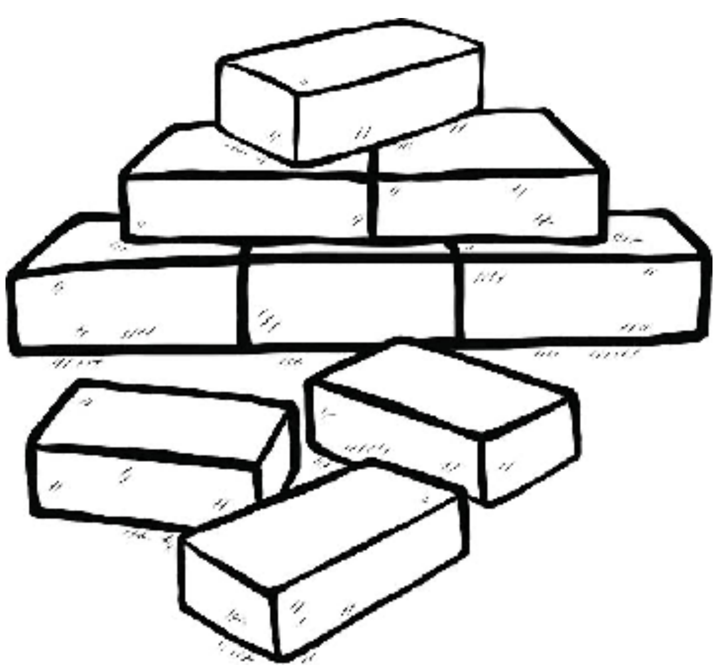
$$\left.\begin{matrix} e^x \\ \sin x \\ \cos x \\ \sqrt{x} \end{matrix}\right\} \longrightarrow \sqrt{e^{\sin\left(\sqrt{\cos e^x}\right)}}$$



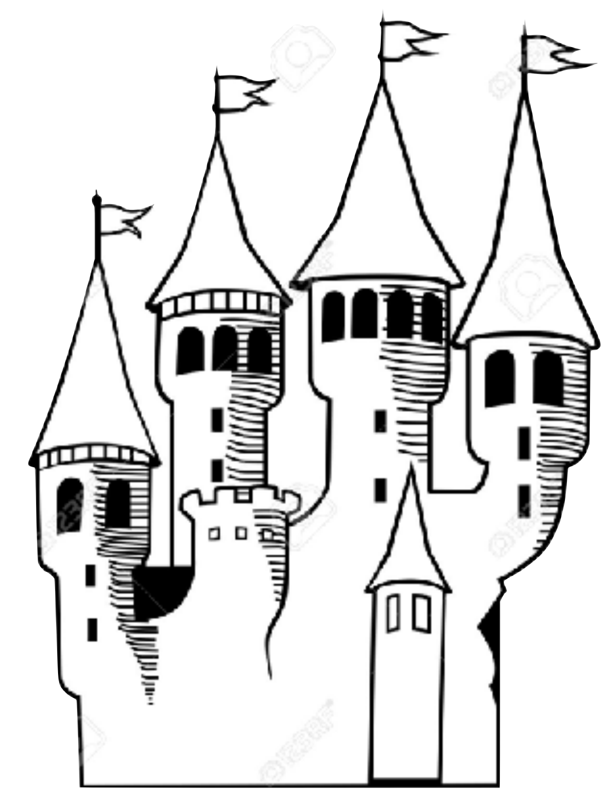


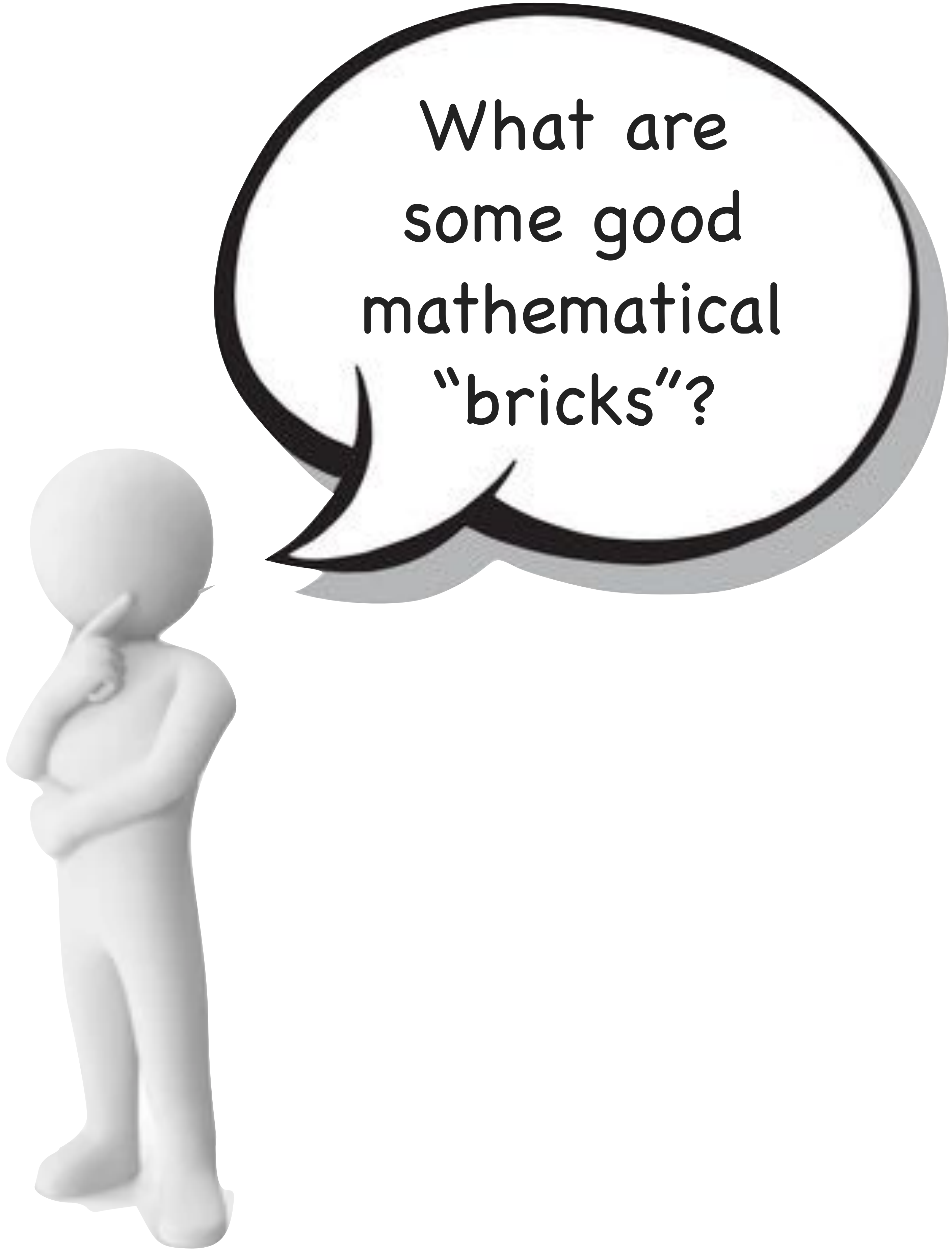
Function composition is a way to build complicated functions from easy pieces:

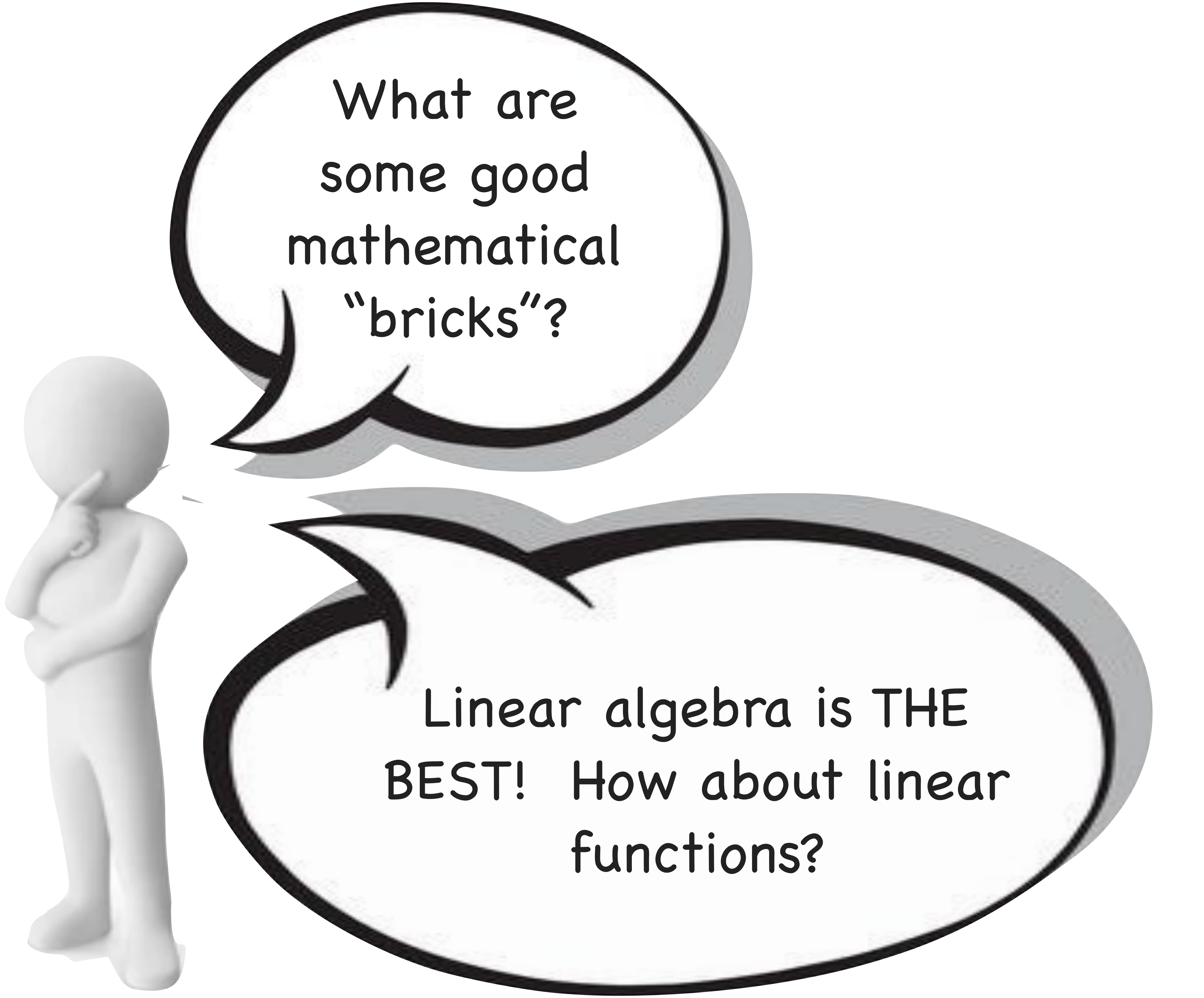




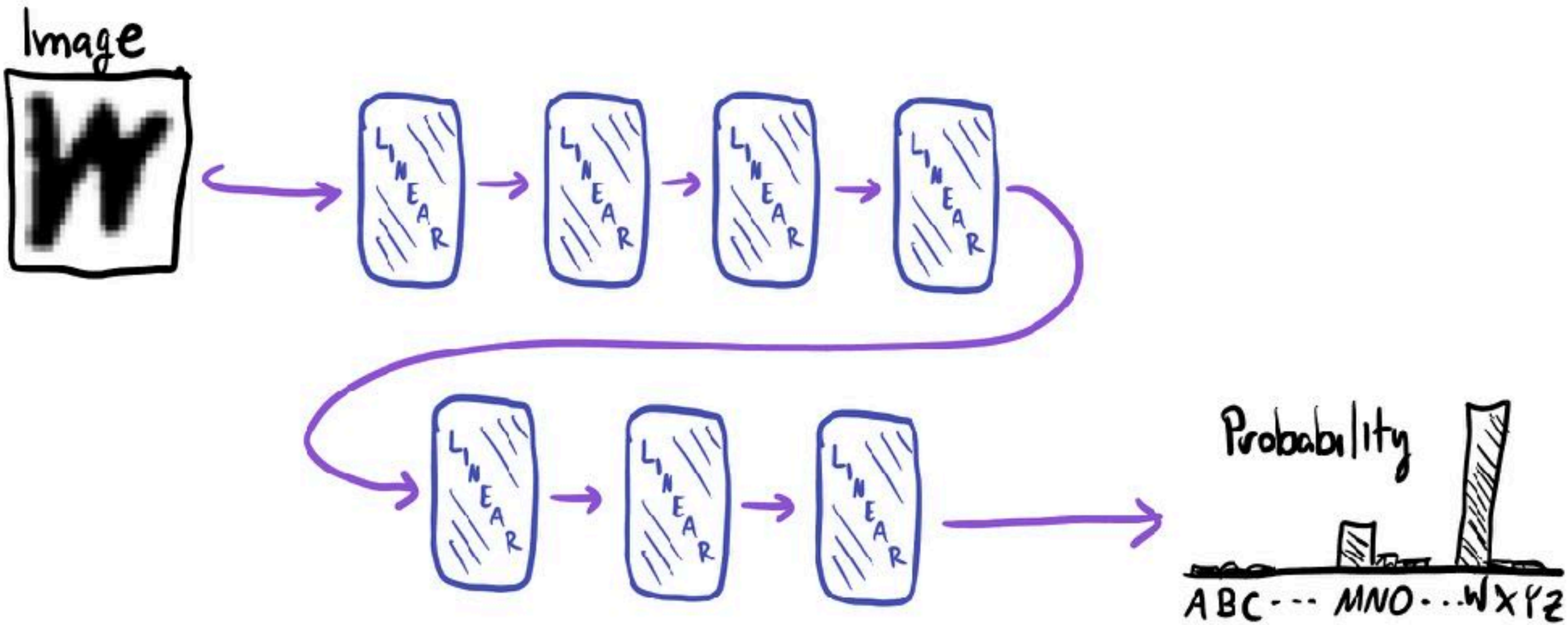
Because composition "strings functions together" we call the result a **network.**



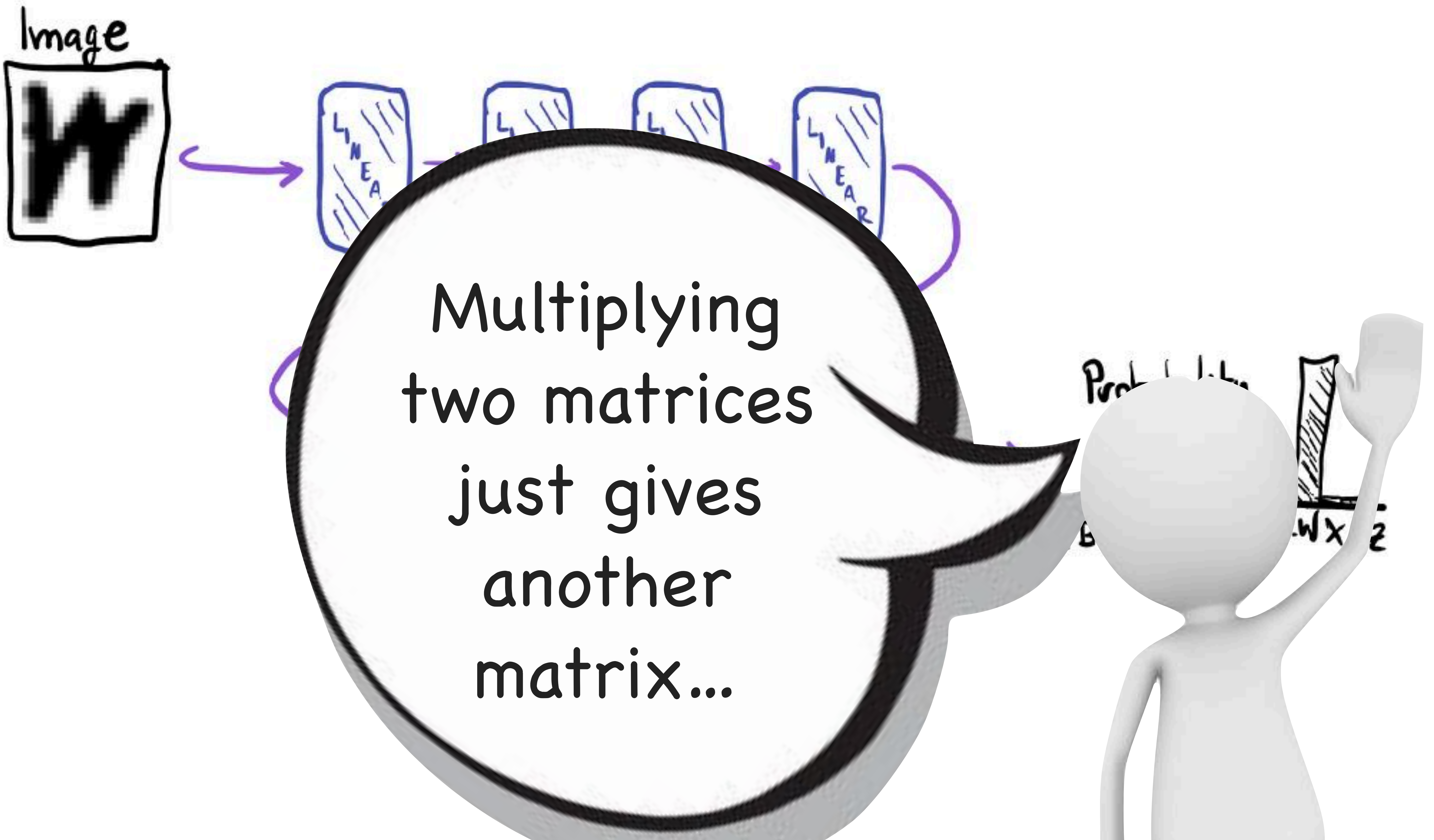

What are some good mathematical "bricks"?


What are some good mathematical "bricks"?
Linear algebra is THE BEST! How about linear functions?

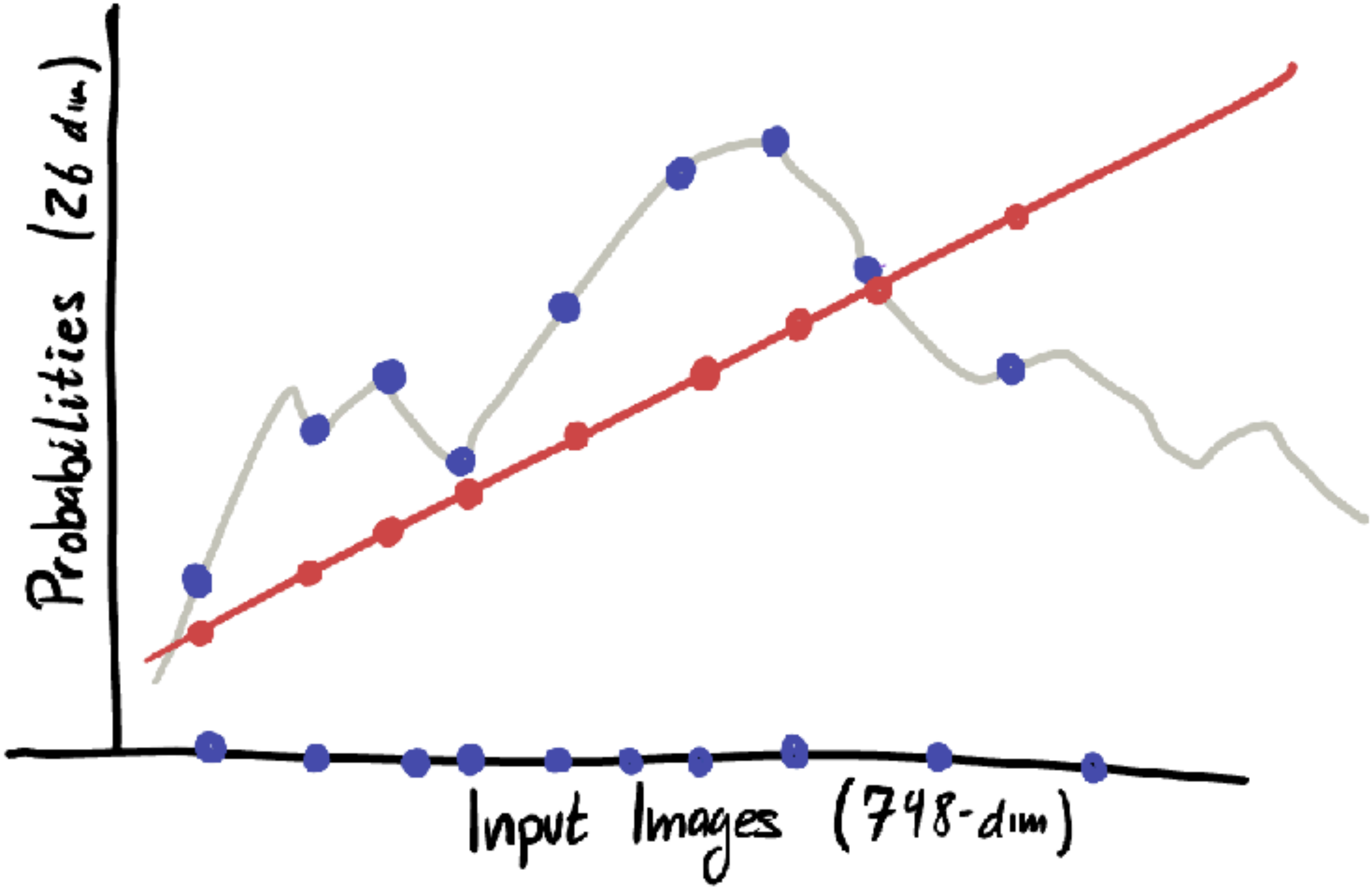
A network of linear pieces:



A network of linear pieces:



A network of linear pieces:


Probabilities (26 dim)
Input Images (748-dim)

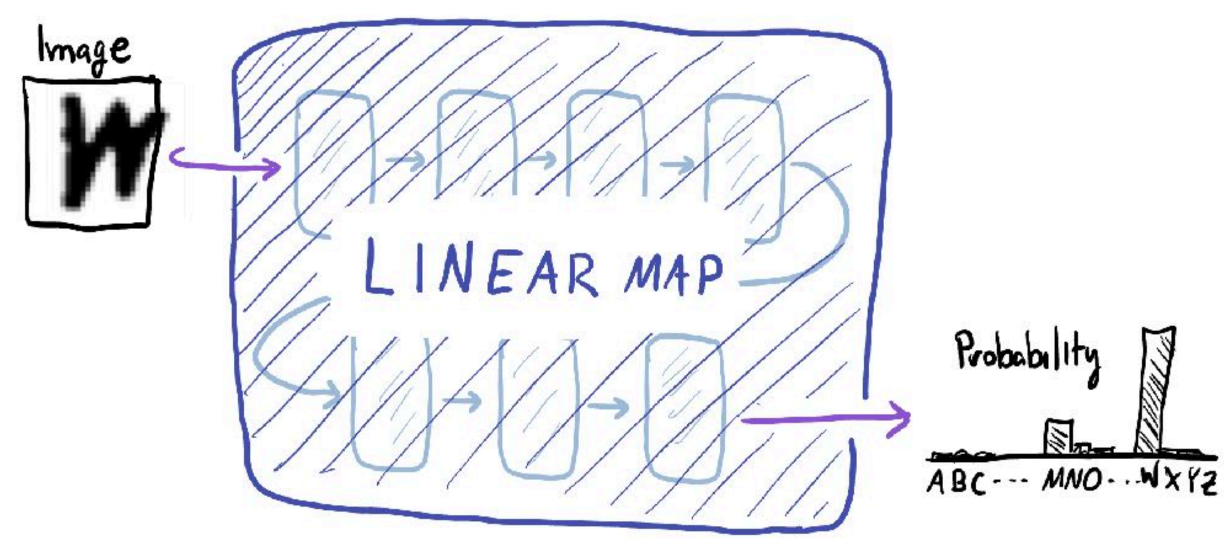
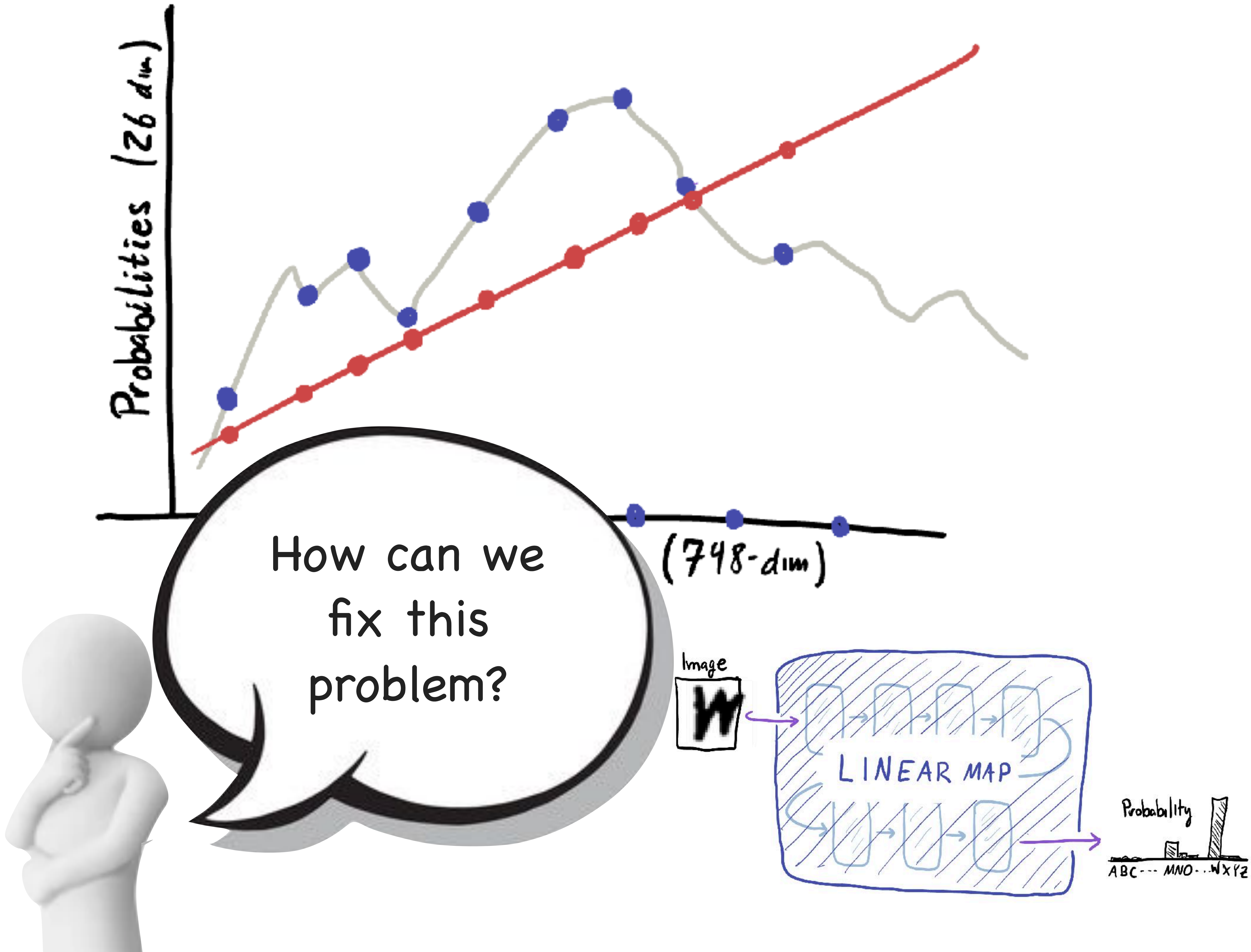
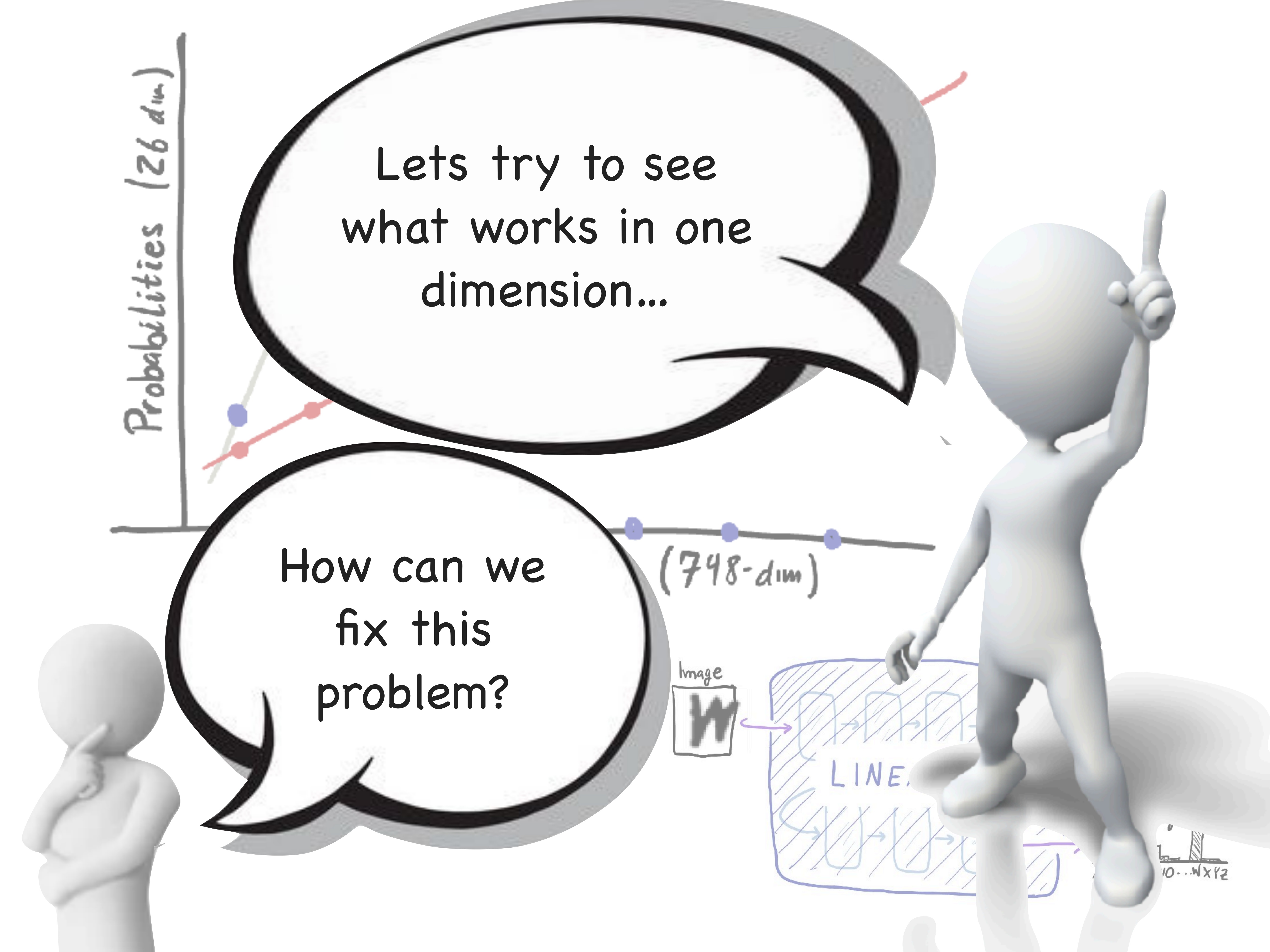
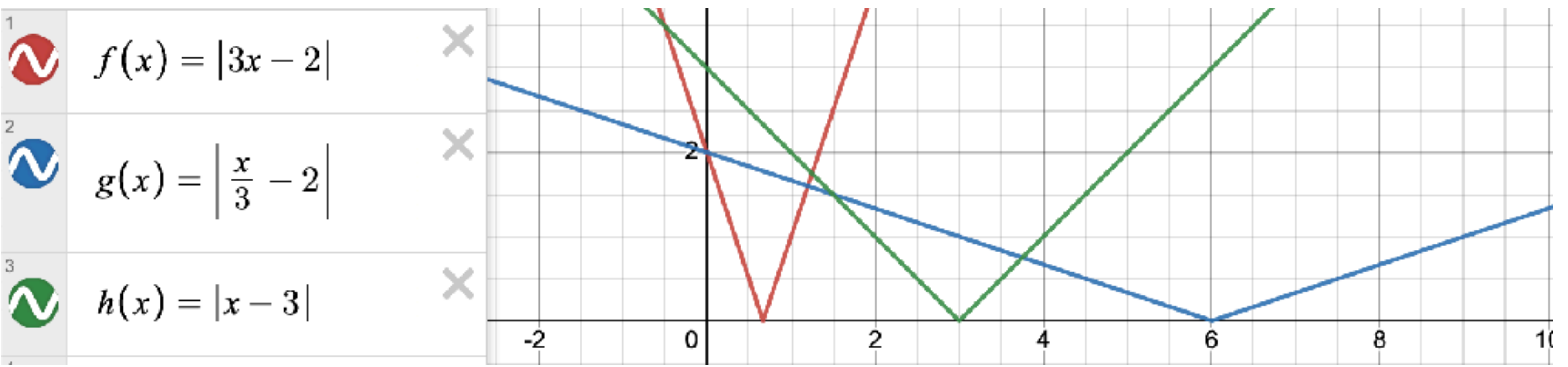


Image
LINEAR MAP
Probability
ABC --- MNO ... WXYZ

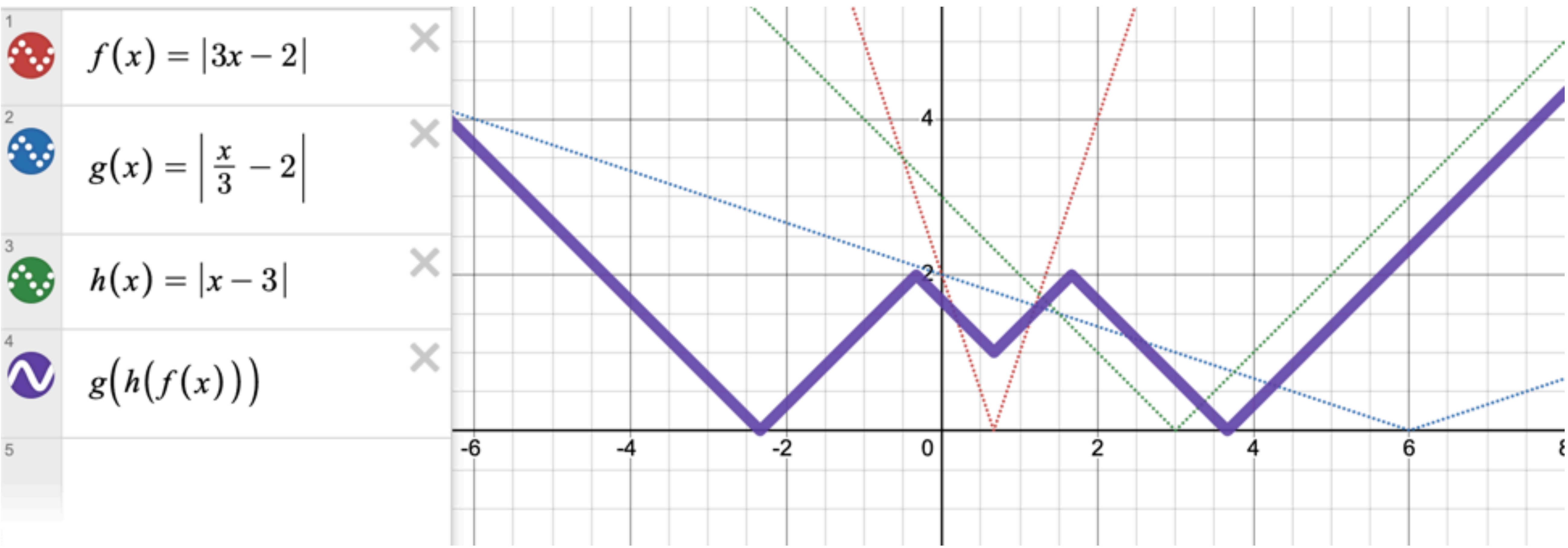

Probabilities (26 dim)
(748-dim)
How can we fix this problem?
Image
LINEAR MAP
Probability
ABC --- MNO ... WXYZ


Lets try to see what works in one dimension...
How can we fix this problem?
Probabilities (26 dim)
(748-dim)
Image

What happens if we surround some linear functions with the absolute value, before composing them?

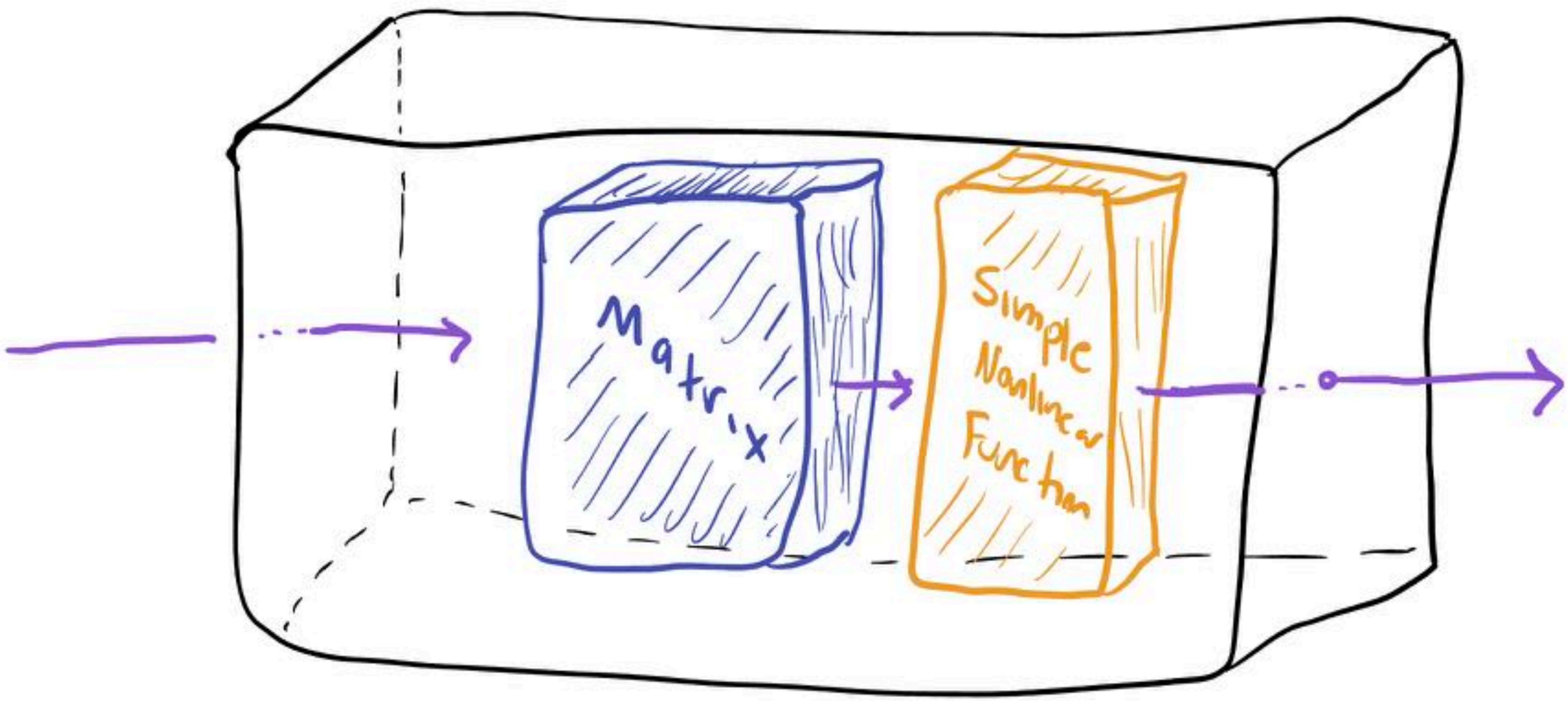


The result is much more complicated than any of the individual building blocks!




The result is
d than
any of t
s!
We should sandwich together a linear and nonlinear function to build our bricks!
$f(x) = |3x - 2|$
$g(x)$
$h(x) =$
$g(h(f(x)))$
-6
-4
-2
0
2
4
6

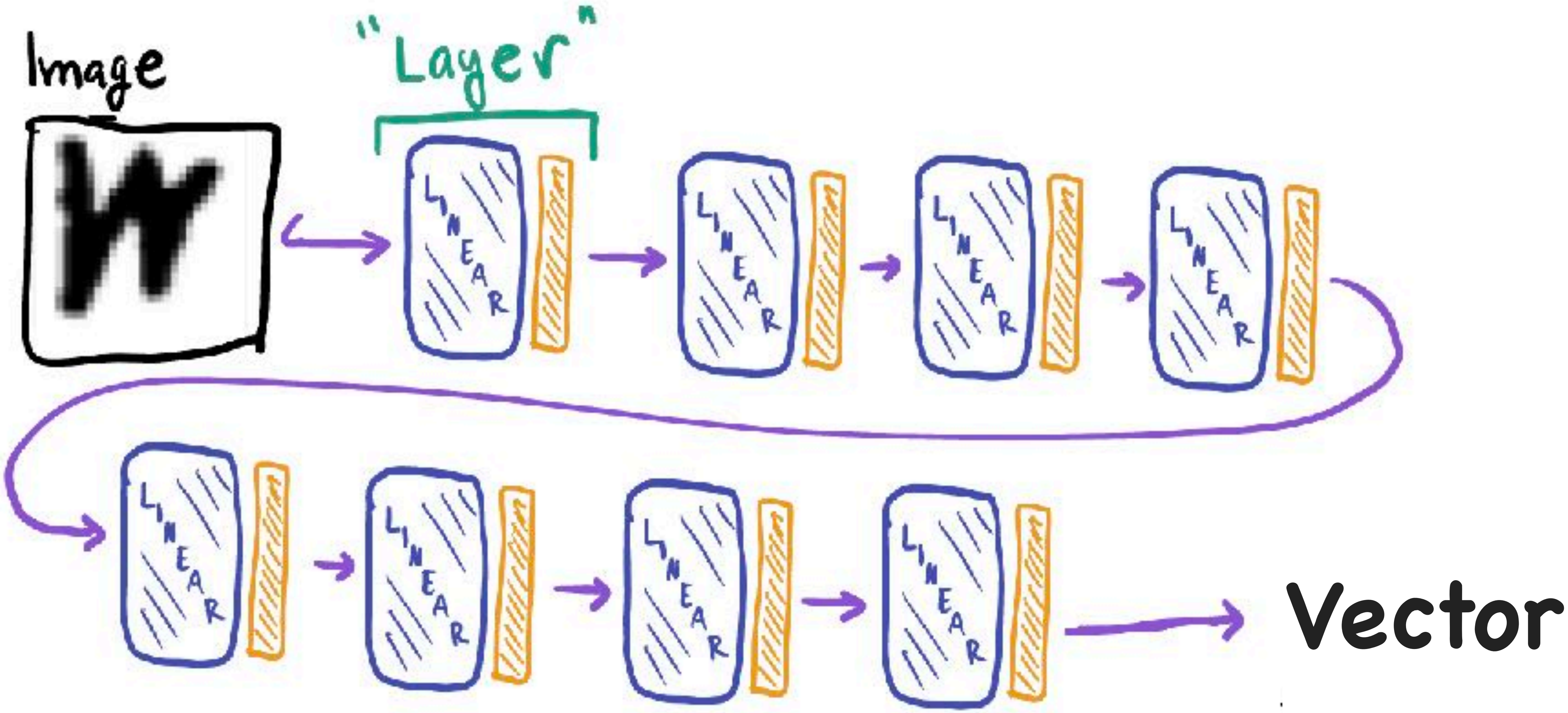
A proposed "brick"



$$\sigma = \max\{x, 0\}$$

$$\sigma = |x| \qquad \sigma = e^x/(1 + e^x)$$

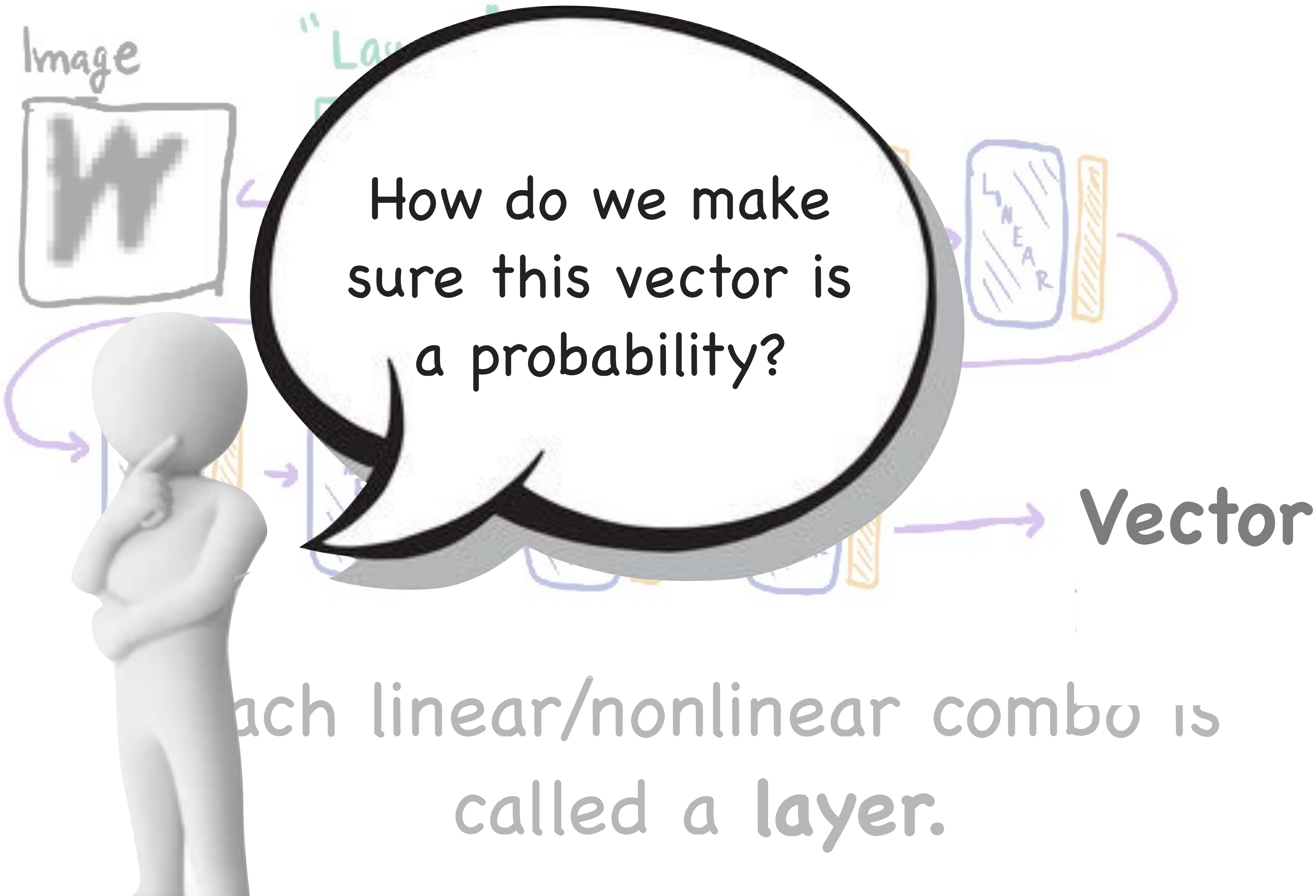
Building our Castle out of Bricks:
The architecture of a Neural Network



Each linear/nonlinear combo is called a **layer.**

Building our Castle out of Bricks:

The architecture of a Neural Network



First: make sure the final linear map is of the right dimension! So for us, it needs to map into 26-dimensional space.

Second: vectors can have any real numbers as entries, but probabilities must be positive. We much choose a way to make all entries positive.

Third: Probabilities add to 1: we must rescale the vector so that its entires add to 1.

The Softmax Function

$$\vec{v} = \langle v_1, v_2, \ldots, v_{26} \rangle$$

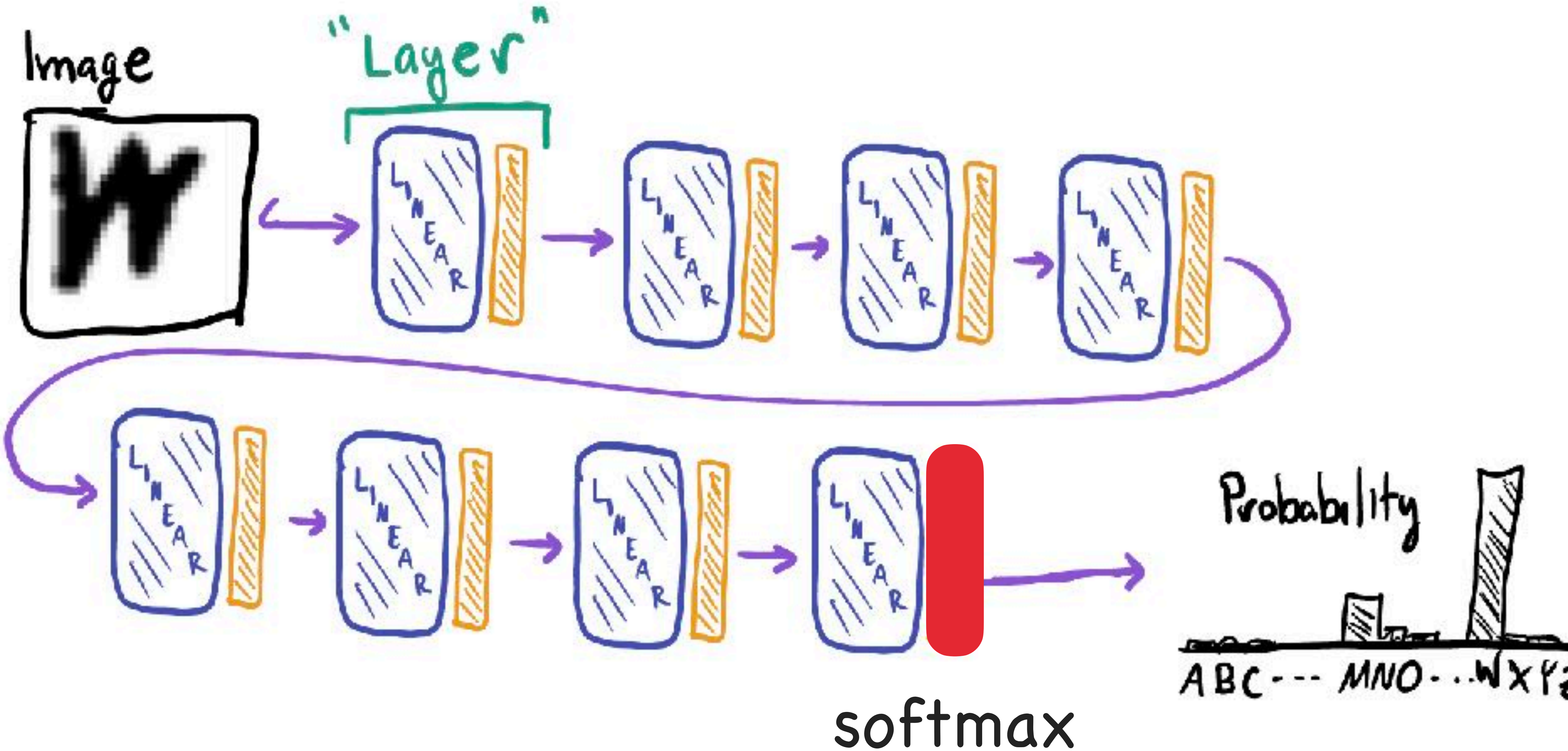
$$\downarrow$$

$$\langle e^{v_1}, e^{v_2}, \ldots, e^{v_{26}} \rangle$$

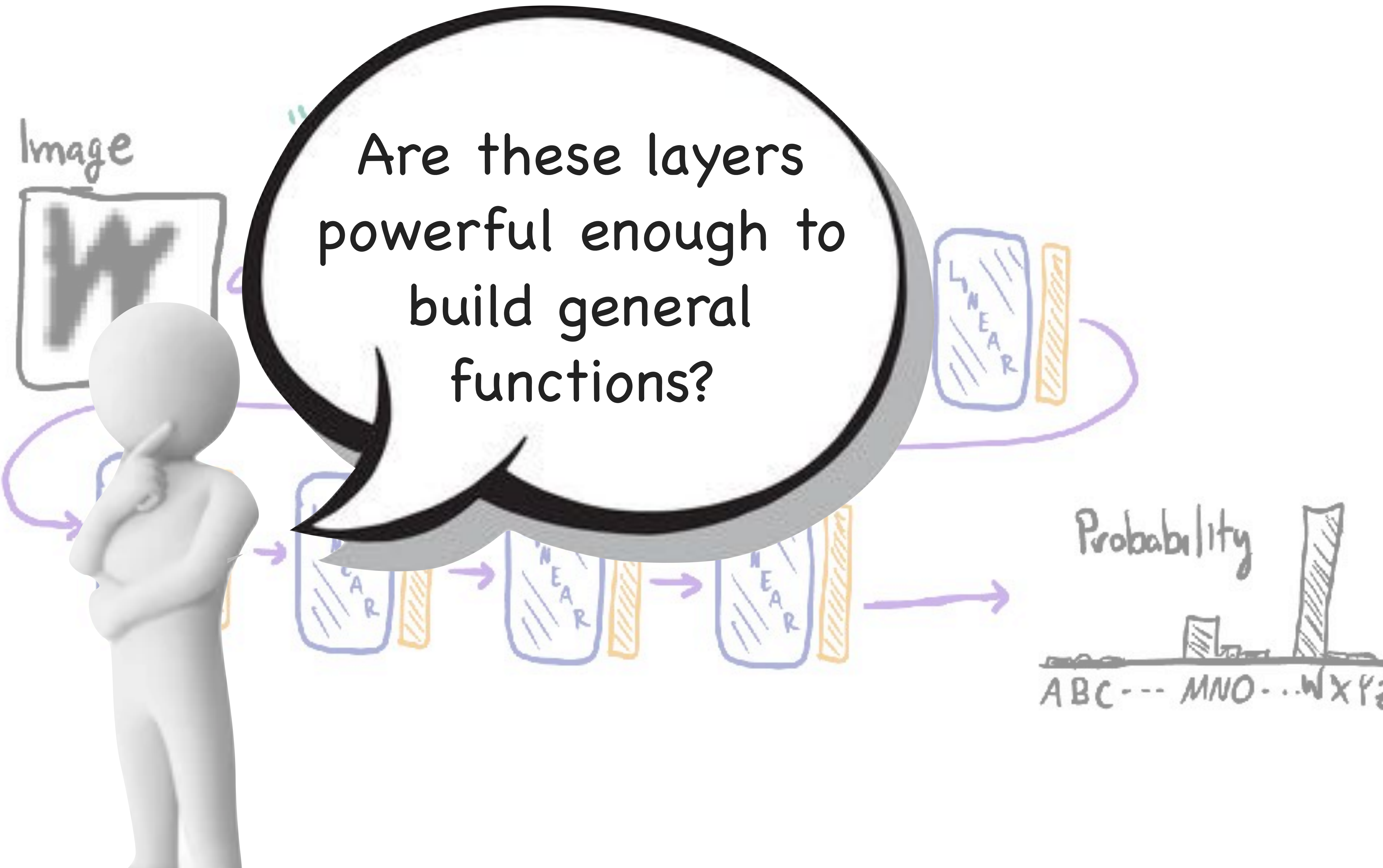
$$\downarrow$$

$$\text{softmax}(\vec{v}) = \frac{\langle e^{v_1}, e^{v_2}, \ldots, e^{v_{26}} \rangle}{e^{v_1} + e^{v_2} + \cdots + e^{v_{26}}}$$

Building our Castle out of Bricks:
The architecture of a Neural Network



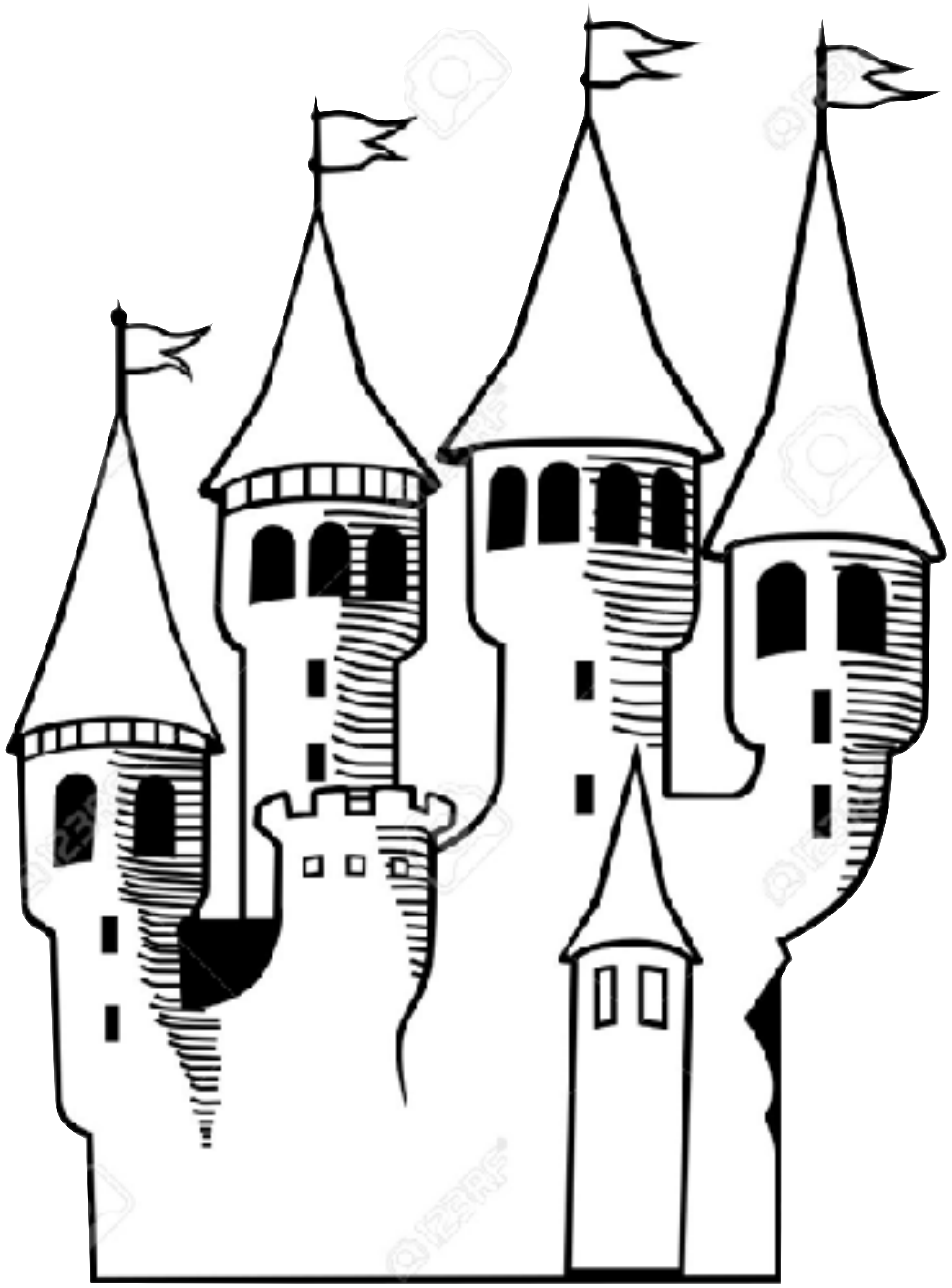
Building our Castle out of Bricks:
The architecture of a Neural Network



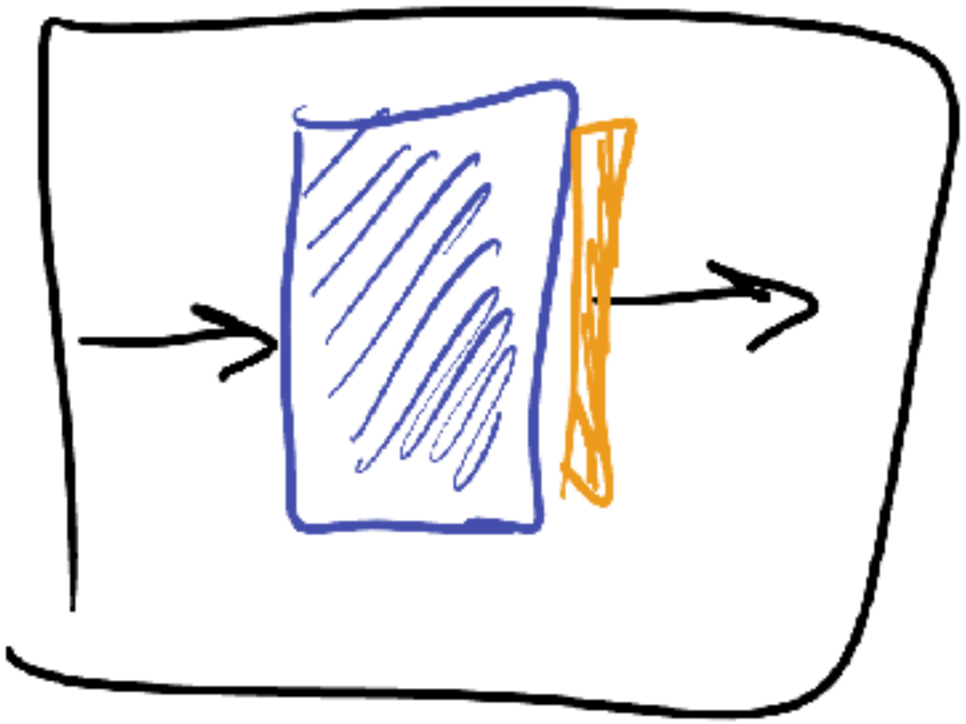
Universal Approximation

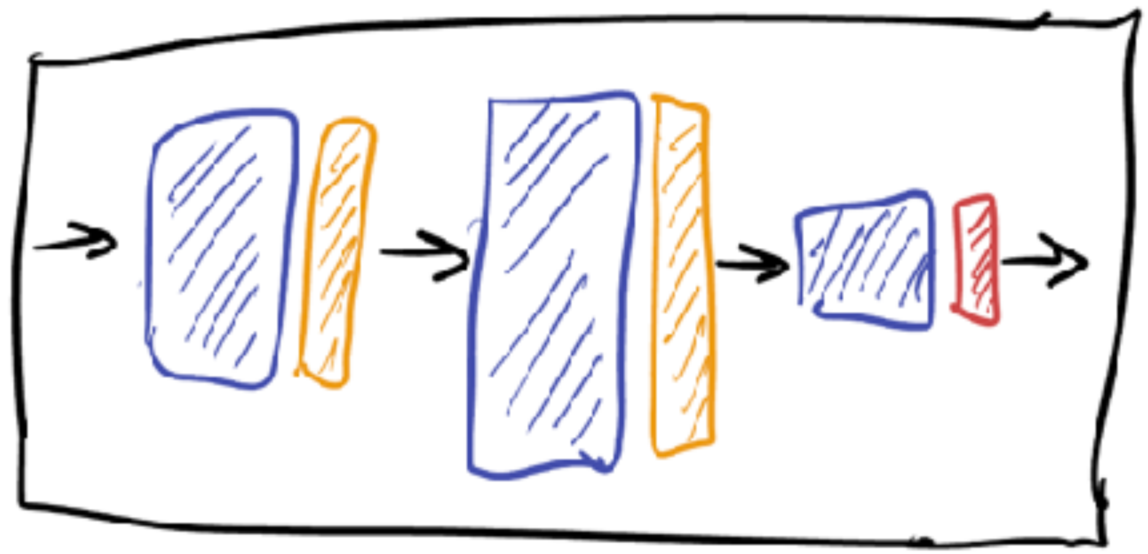
Theorem:

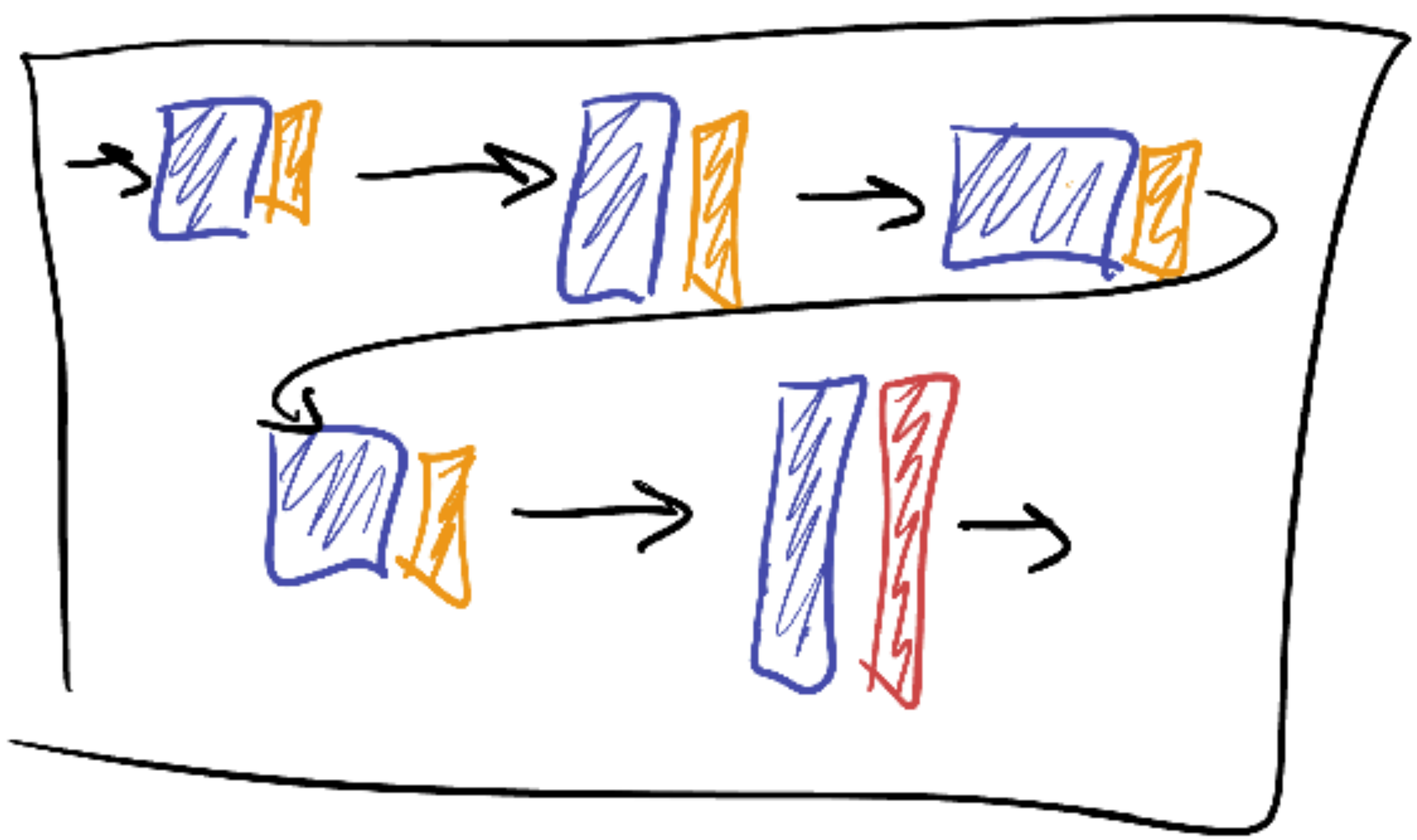
Using enough* layers and large enough* matrices, you can approximate any* function to whatever accuracy** you need

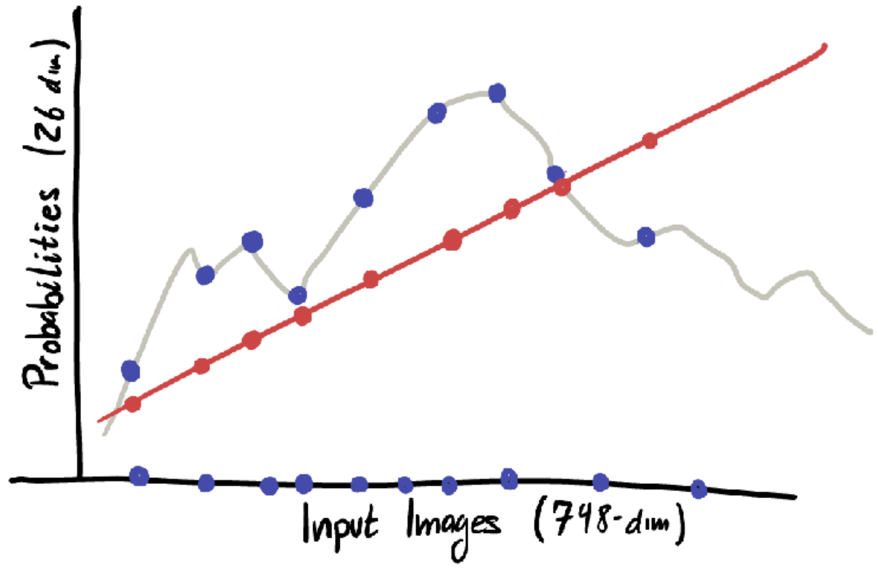


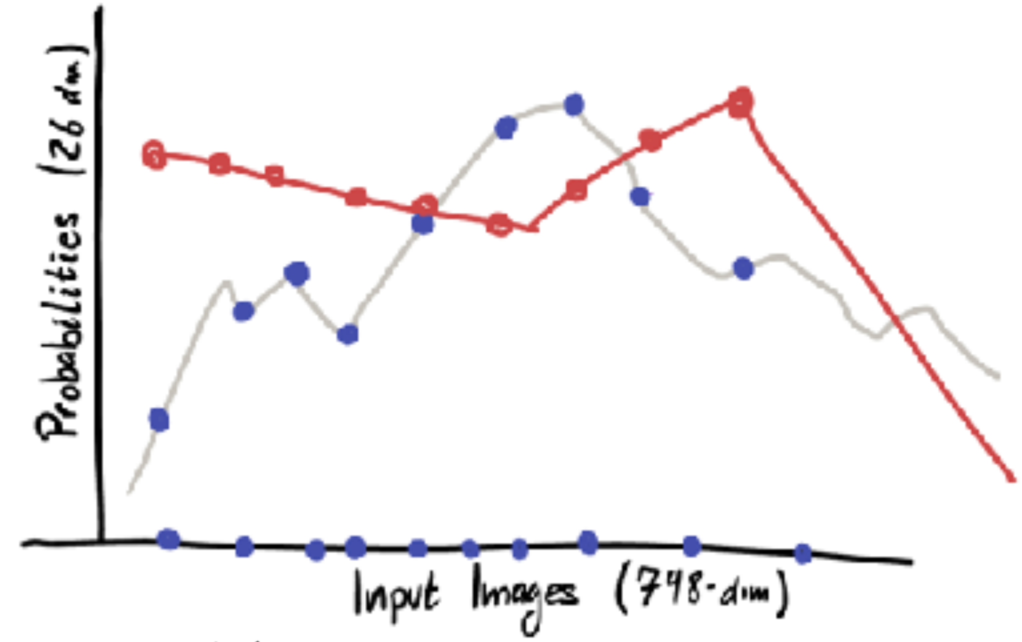
*Warning: I'm ignoring a TON of technical details here...
**With accuracy measured by the Loss function!

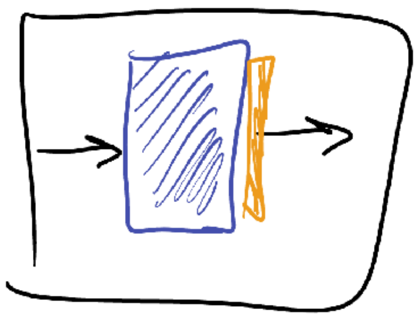


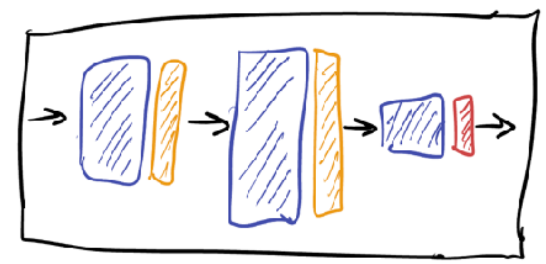


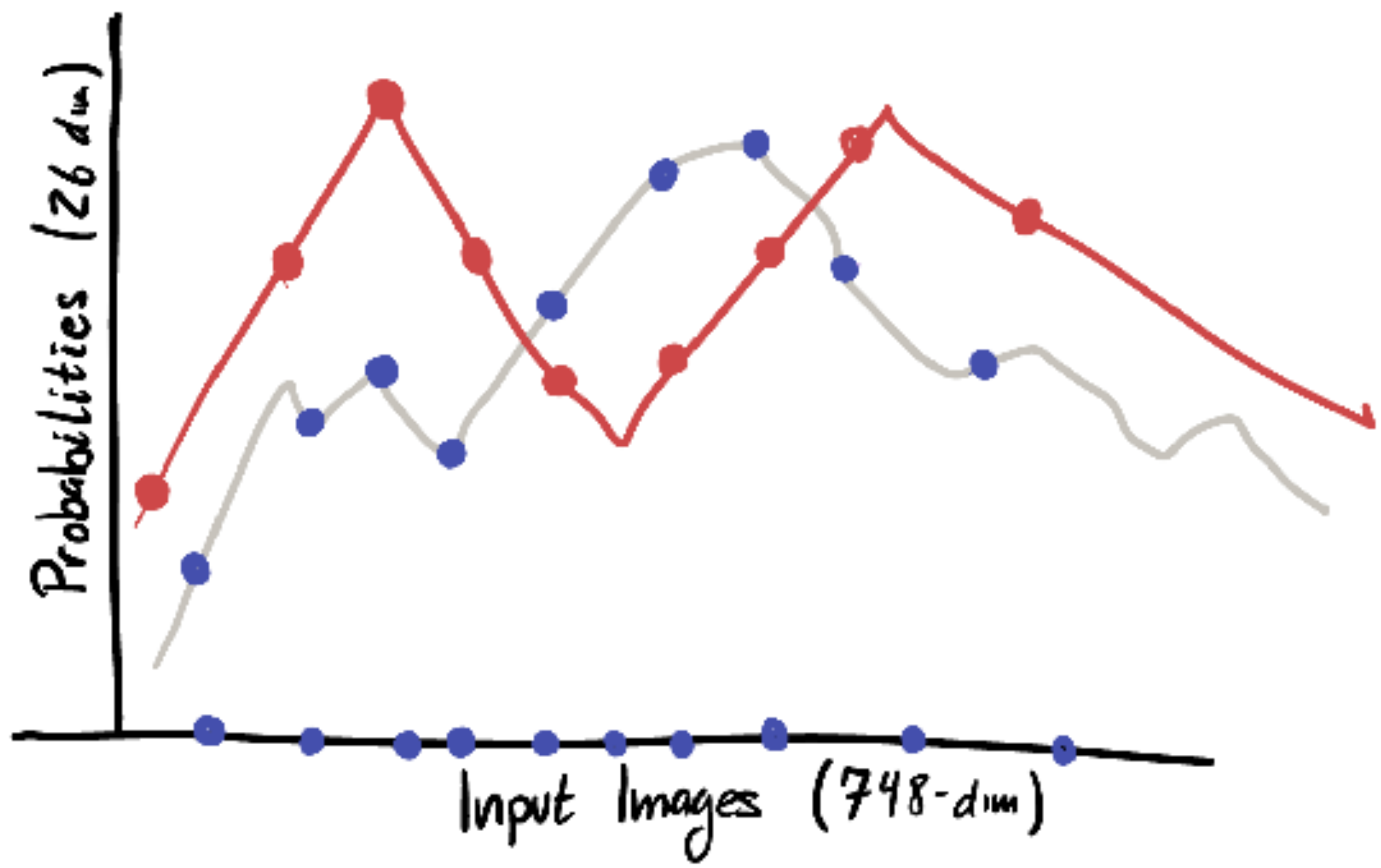


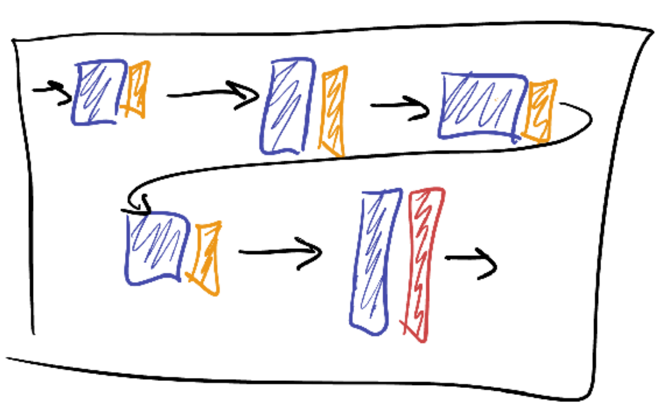

Probabilities (26 dim)
Input Images (748-dim)


Probabilities (26 dim)
Input Images (748-dim)

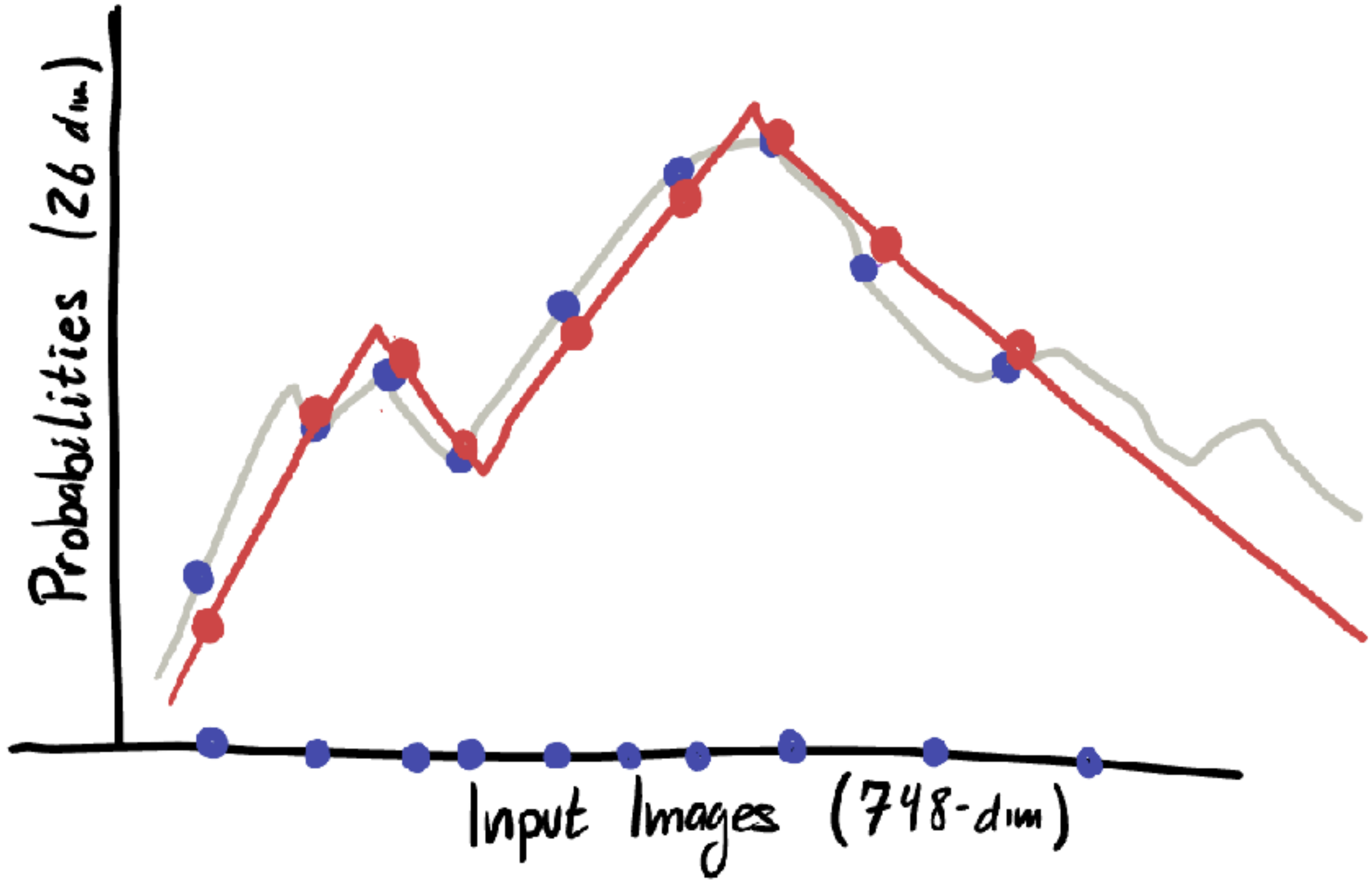


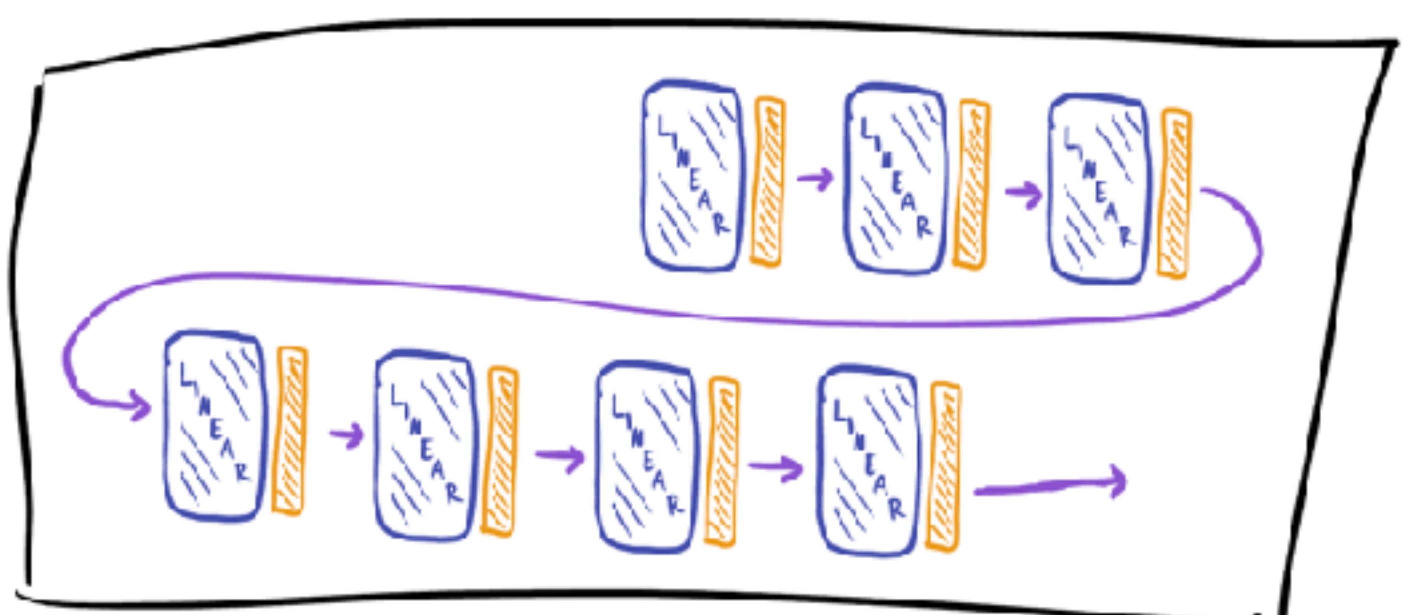



Probabilities (26 dim)
Input Images (748-dim)



Somewhere in the space of all such functions, there are ones that approximate as accurately as you wish.

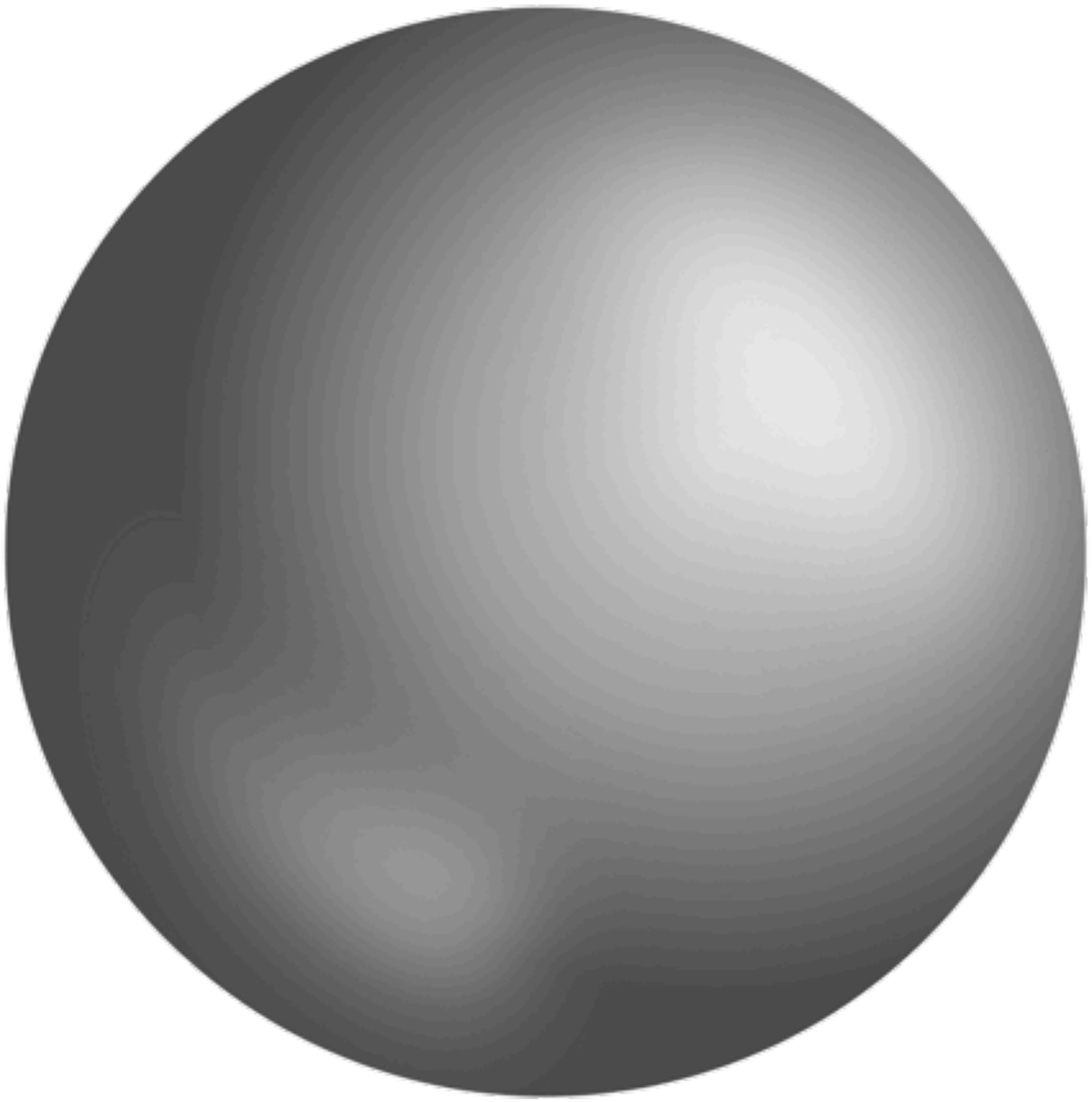




A geometric perspective

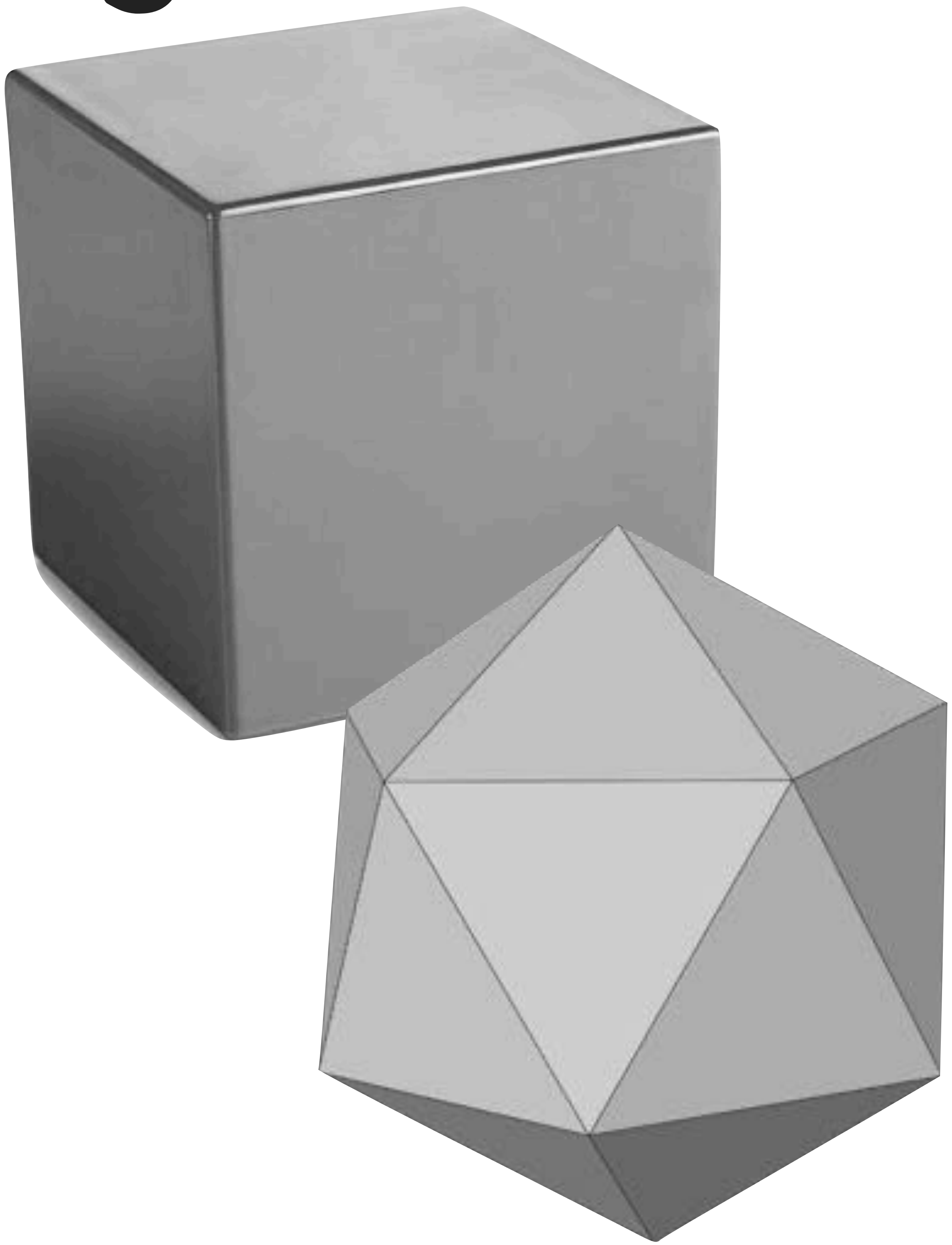
Neural Networks are Like Disco Balls

A geometric perspective



To approximate a complicated shape, like the graph of a classifier function,

A geometric perspective



You can use simple linear pieces joined together nonlinearly..

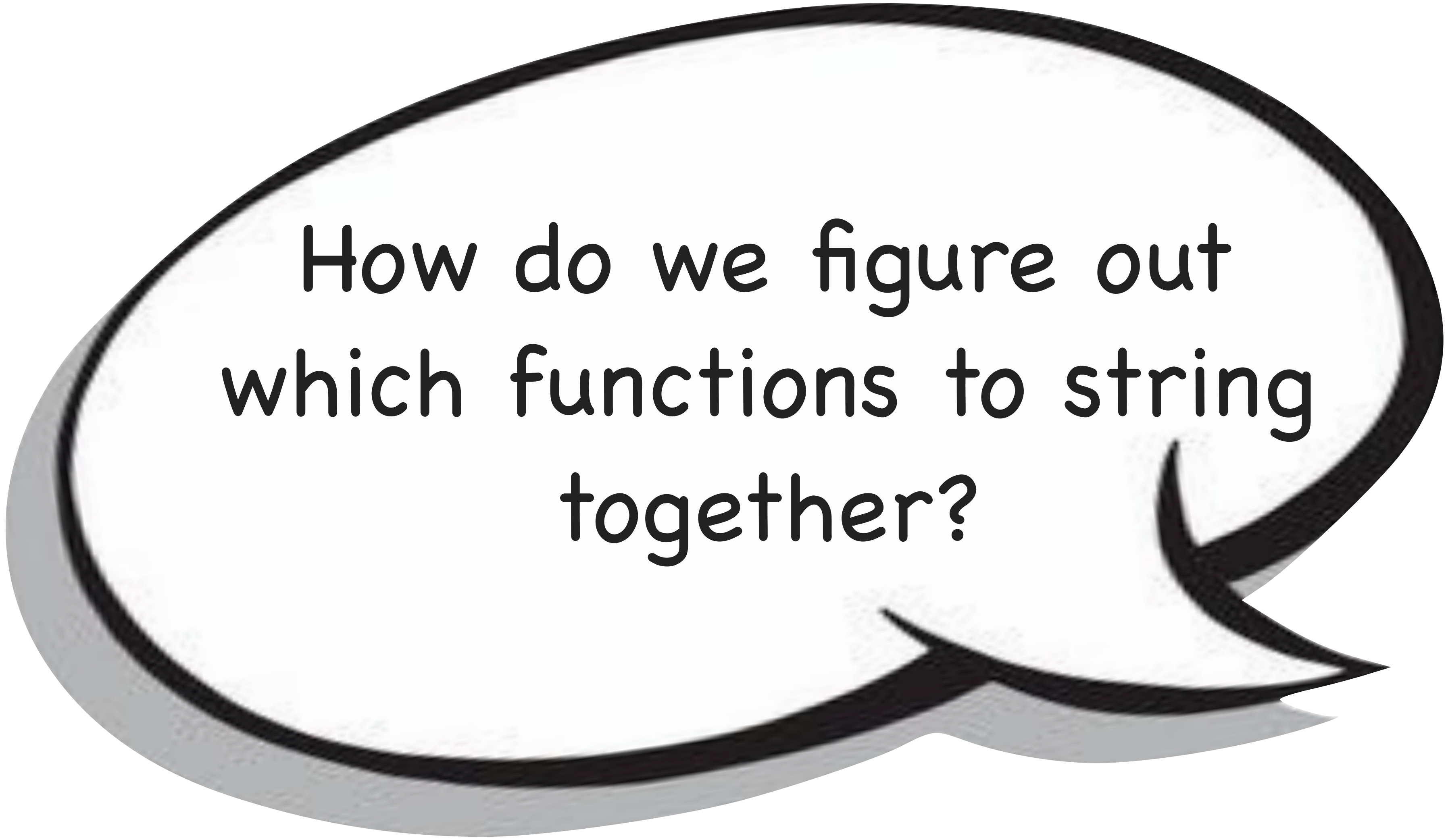
A geometric perspective

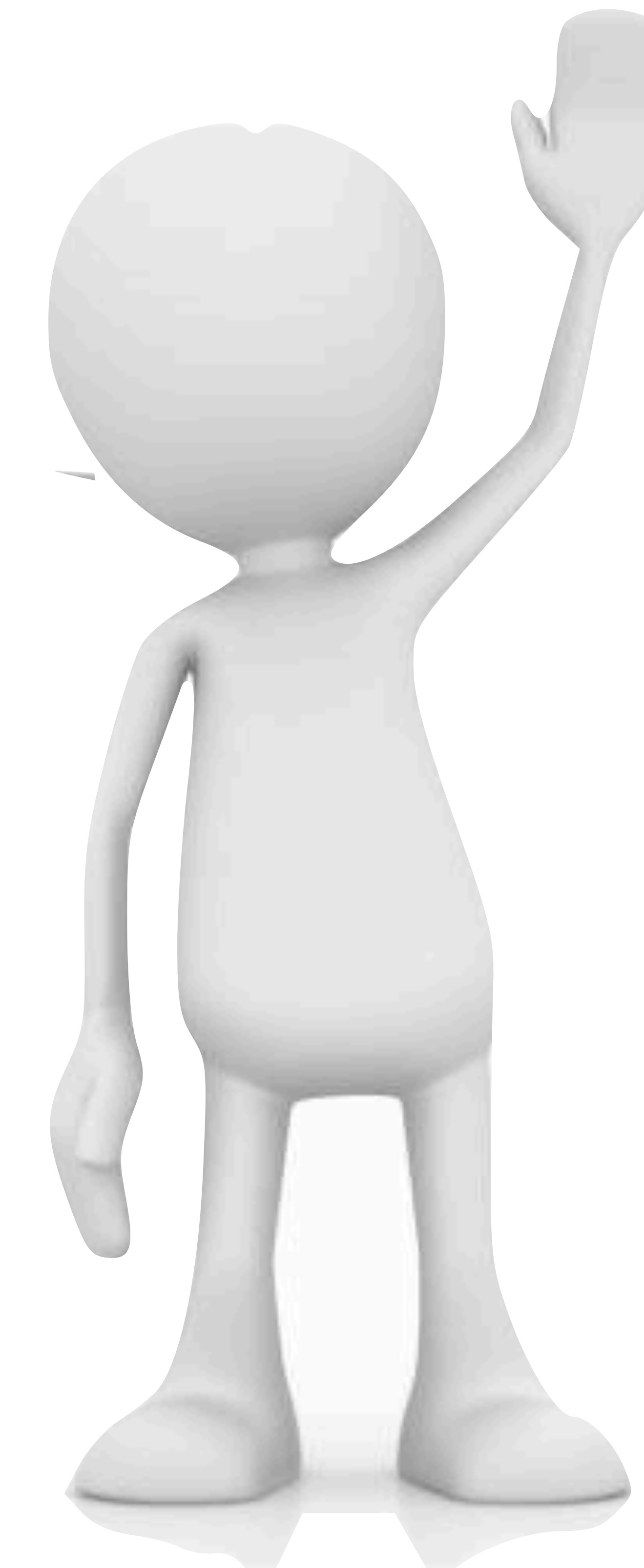
And doing this enough times gives a really good approximation!

This approximation scheme is **universal** so it can approximate any shape you like!

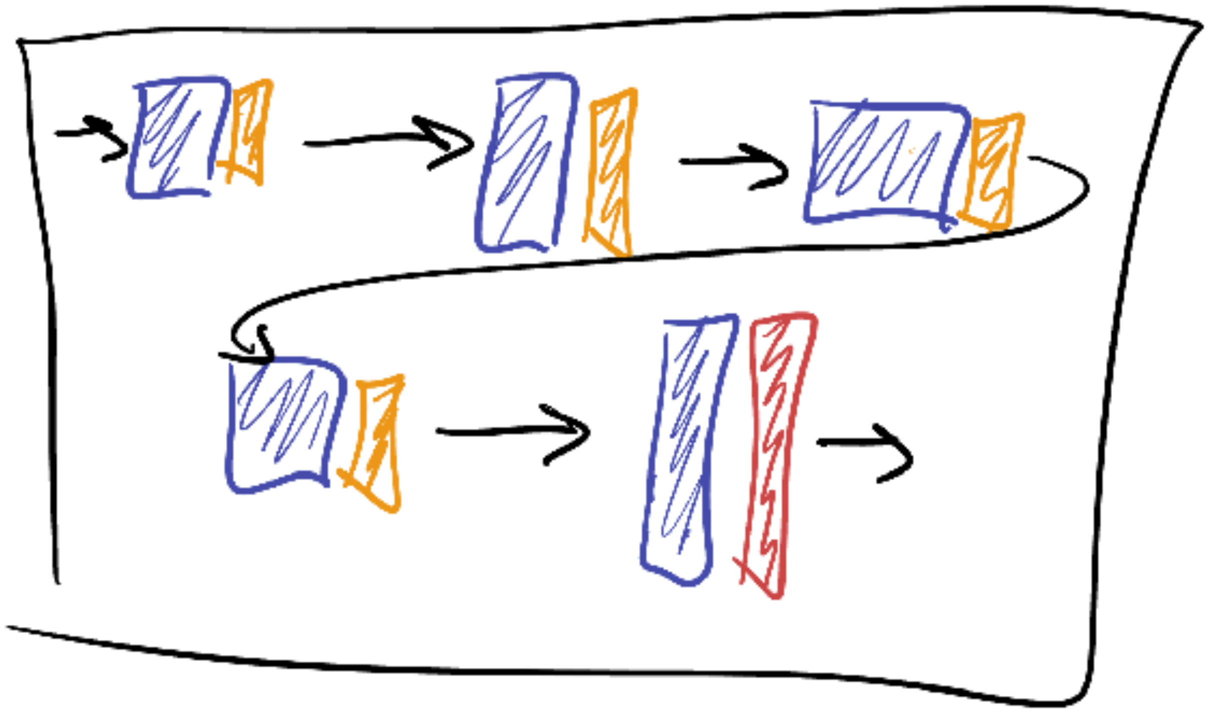
It's possible for computers to learn to read!!!!!!

By universal approximation, there is some sequence of layers that approximates the actual "reading function" to arbitrary precision.

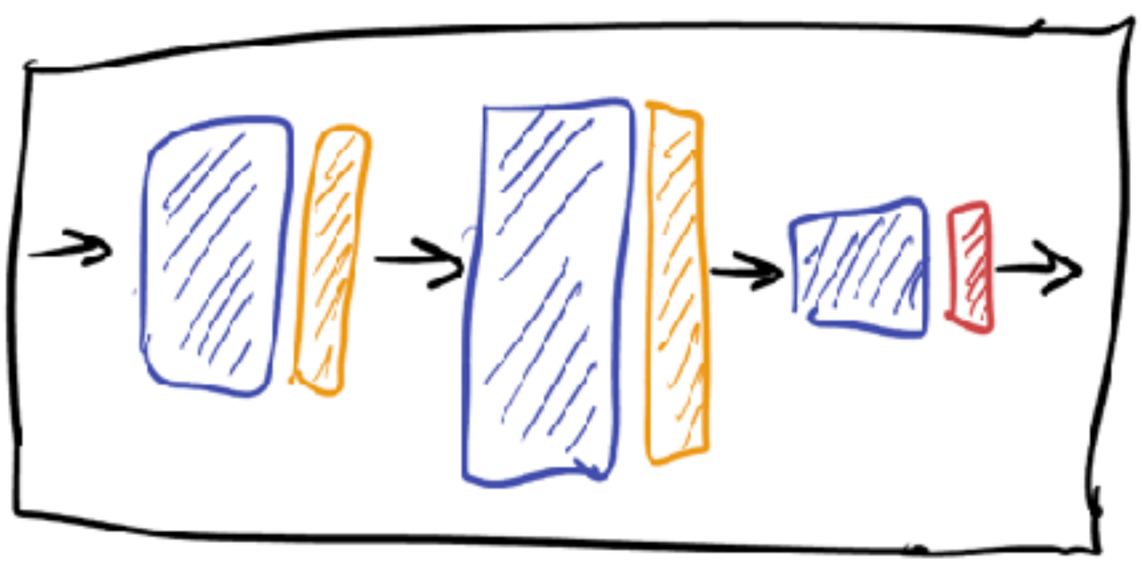




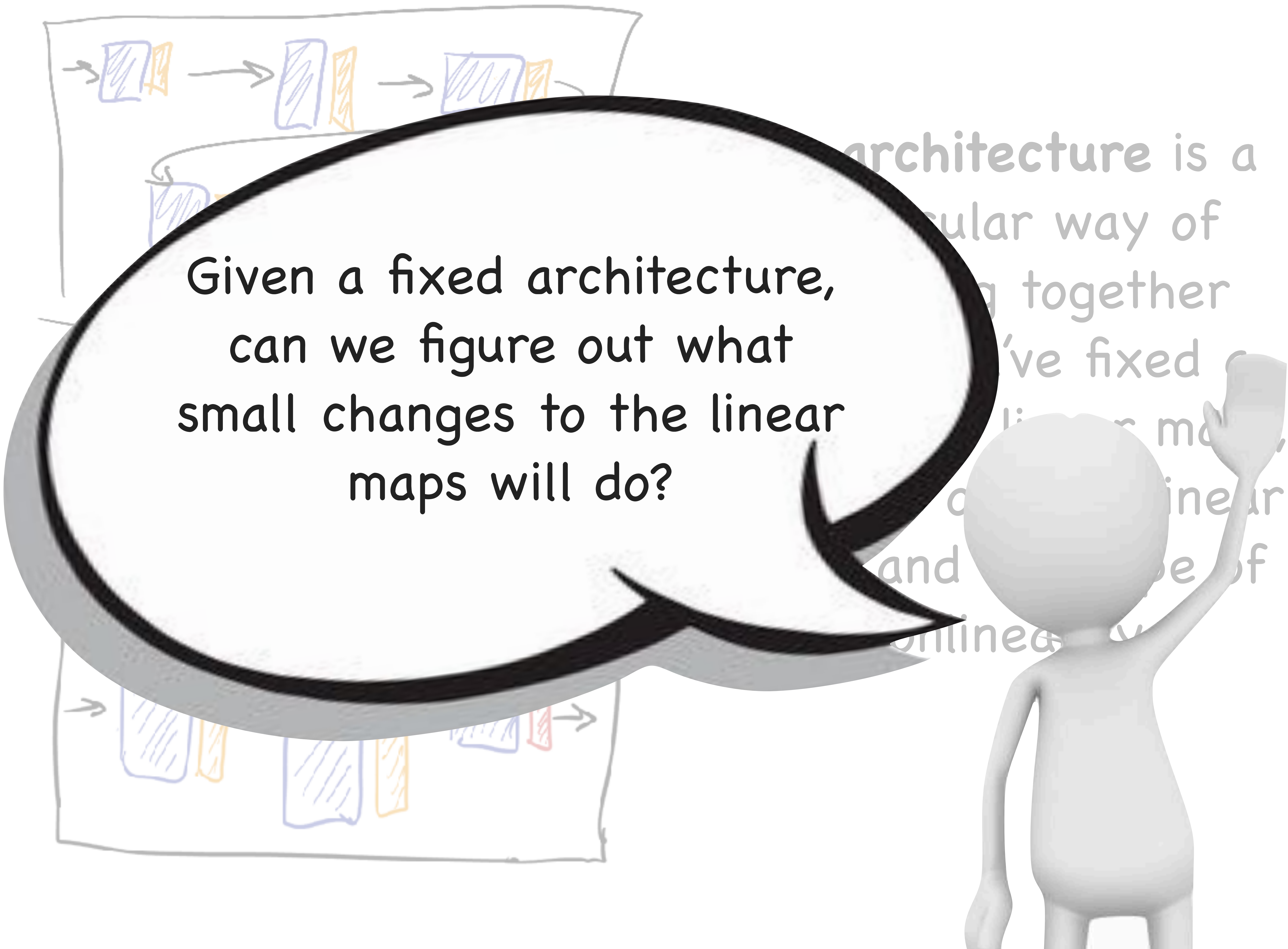
This sounds hard….can we ask an easier question?

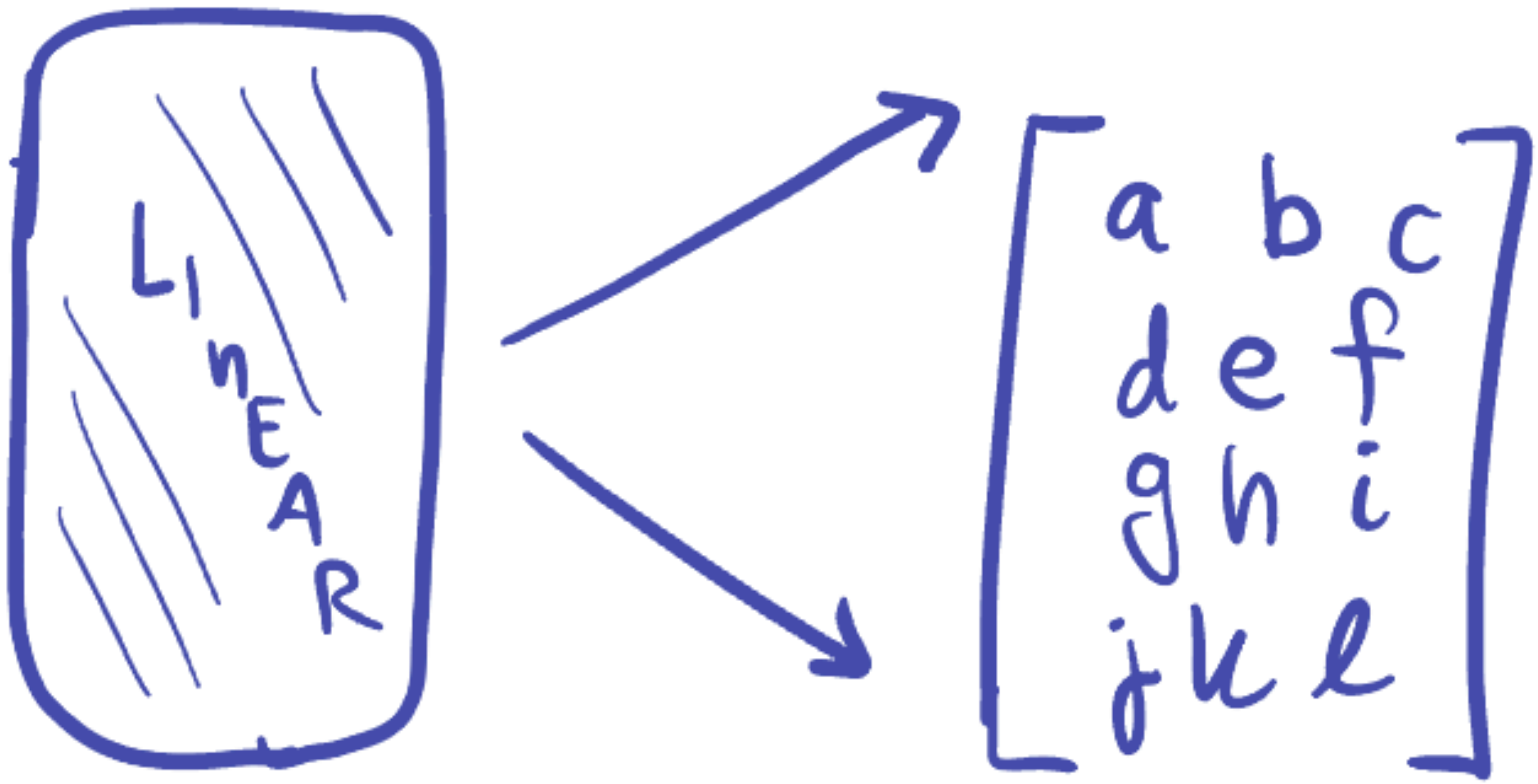


Two different architectures.

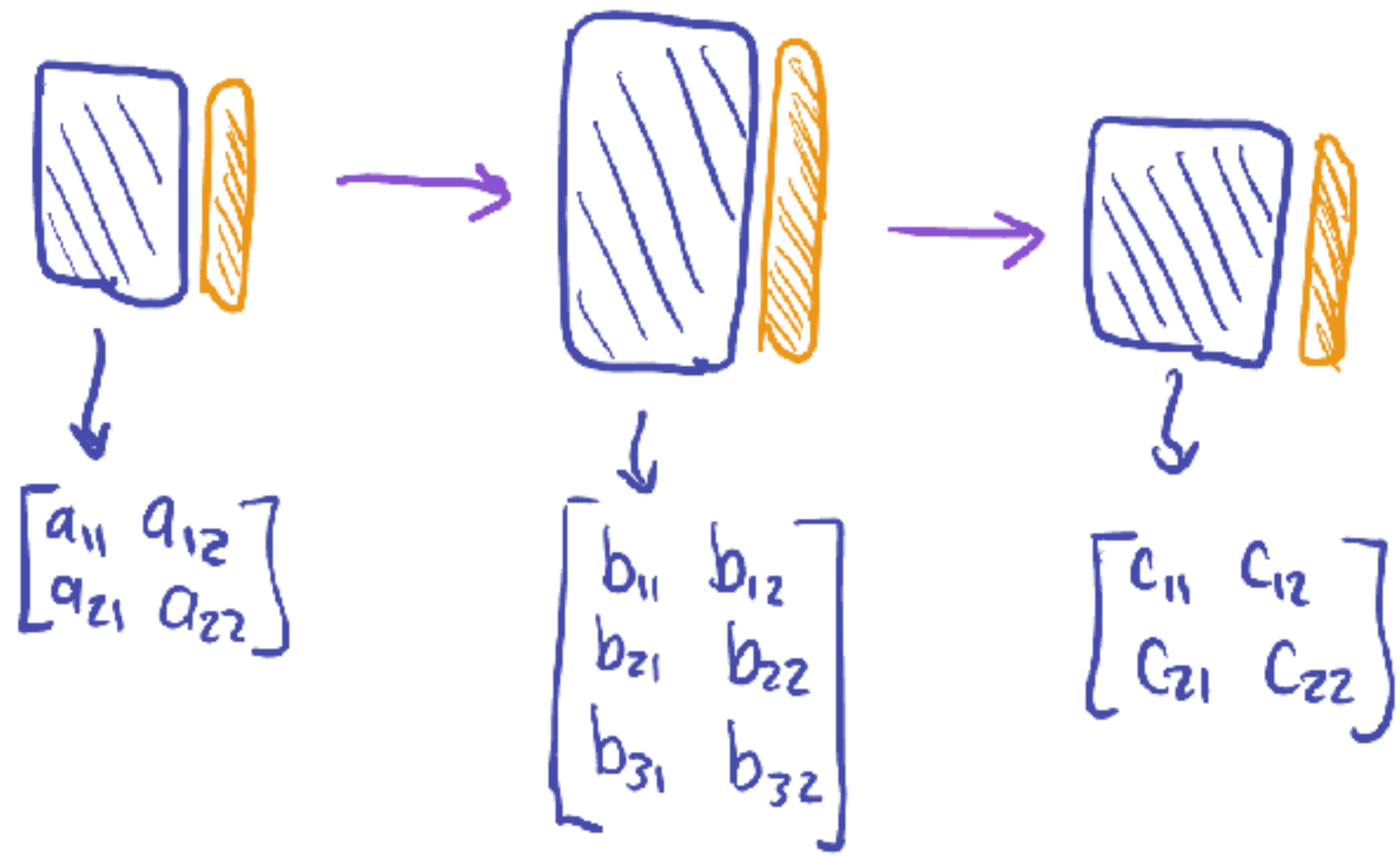


An **architecture** is a particular way of stringing together maps: you've fixed a number of linear maps, the size of each linear map, and the type of nonlinearity.


Given a fixed architecture, can we figure out what small changes to the linear maps will do?



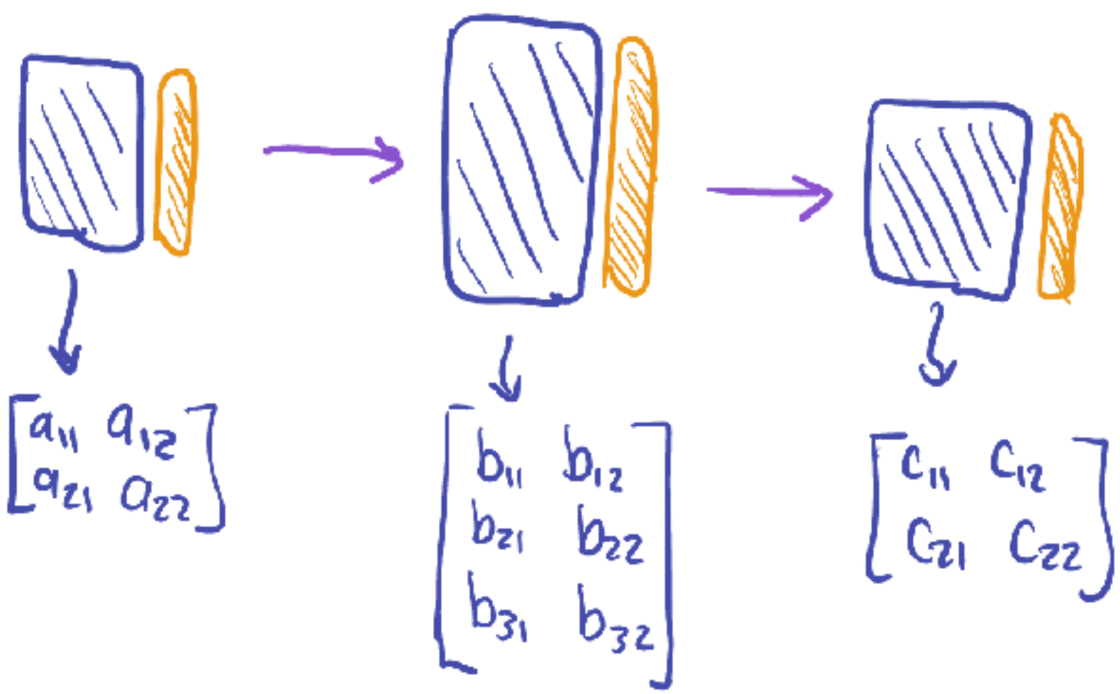
A linear map can be represented by a **matrix**. So each linear map (layer) of our function (network) can be specified by a list of numbers.



Doing this to all the linear maps, an entire network can be specified by a list of numbers. These are called the **weights of the network.**

$$[a_{11}\ a_{12}\ a_{21}\ a_{22}\ b_{11}\ b_{12}\ b_{21}\ b_{22}\ b_{31}\ b_{32}\ c_{11}\ c_{12}\ c_{21}\ c_{22}]$$

Amazing idea! The **space of all networks of a fixed architecture** is a vector space!



For example, this architecture is specified by 14 numbers, so is a point in 14-dimensional space.

$$[a_{11}\ a_{12}\ a_{21}\ a_{22}\ b_{11}\ b_{12}\ b_{21}\ b_{22}\ b_{31}\ b_{32}\ c_{11}\ c_{12}\ c_{21}\ c_{22}]$$

An example architecture we will actually build!

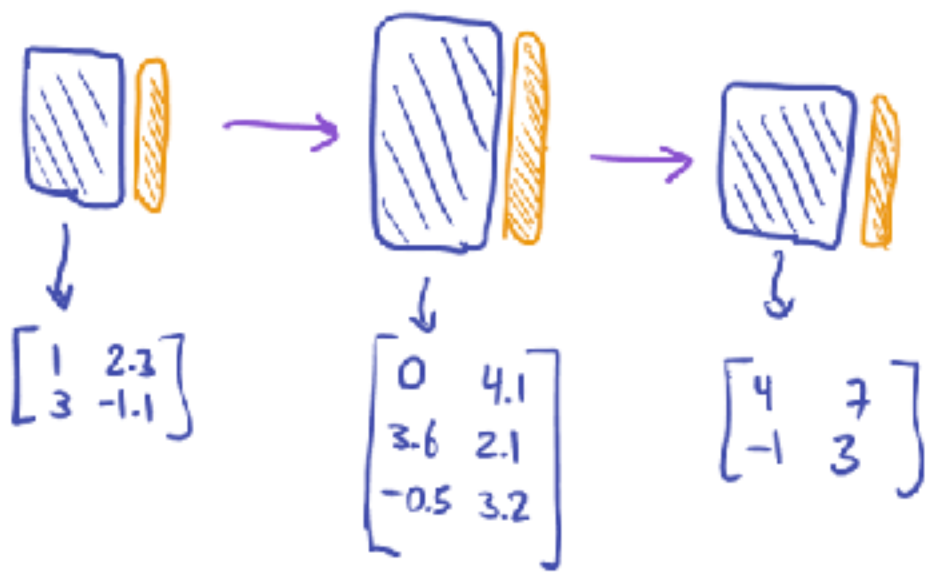
```
class SimpleNeuralNetwork(torch.nn.Module):
 def __init__(self):
    super().__init__()
    self.dense1 = torch.nn.Linear(784,100)
    self.dense2 = torch.nn.Linear(100,300)
    self.dense3 = torch.nn.Linear(300,10)
```

This has 784x100+ 100x300 + 300x10

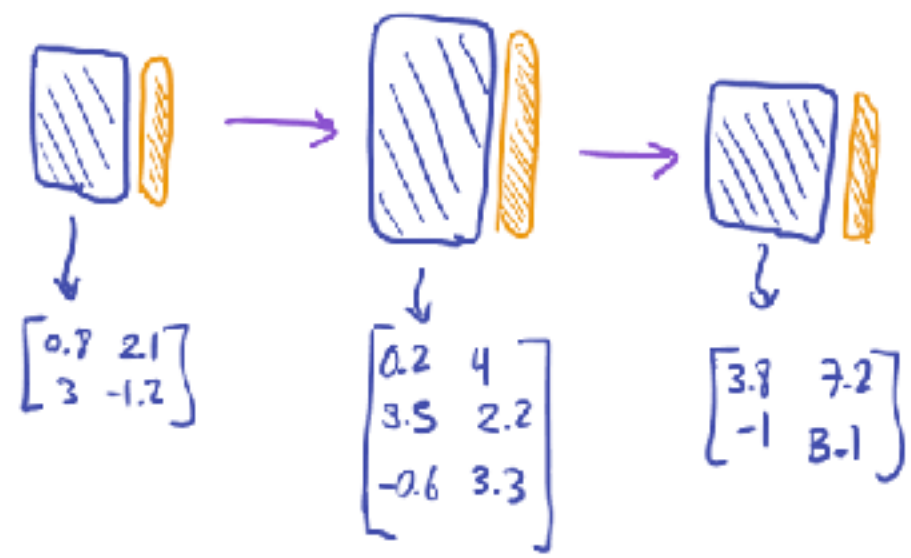
= 111,400 weights!

Slightly tweaking a neural network is as easy as slightly changing the numbers that determine it!

Weights B1

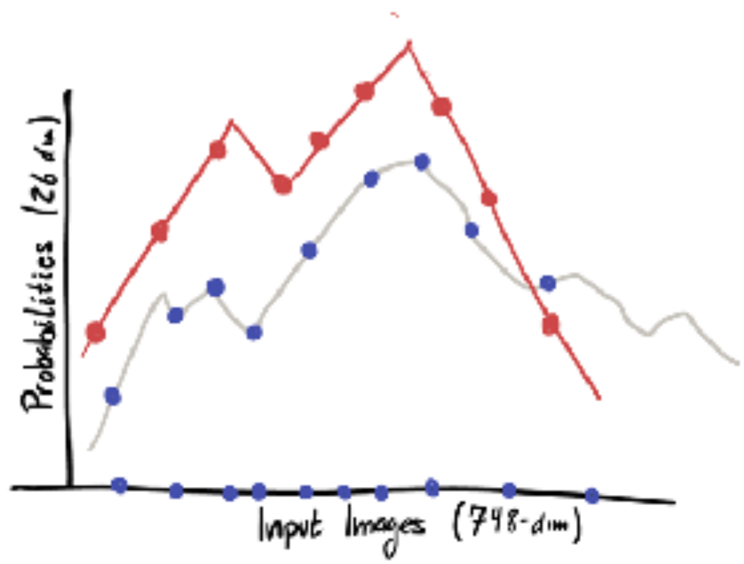


Weights B2

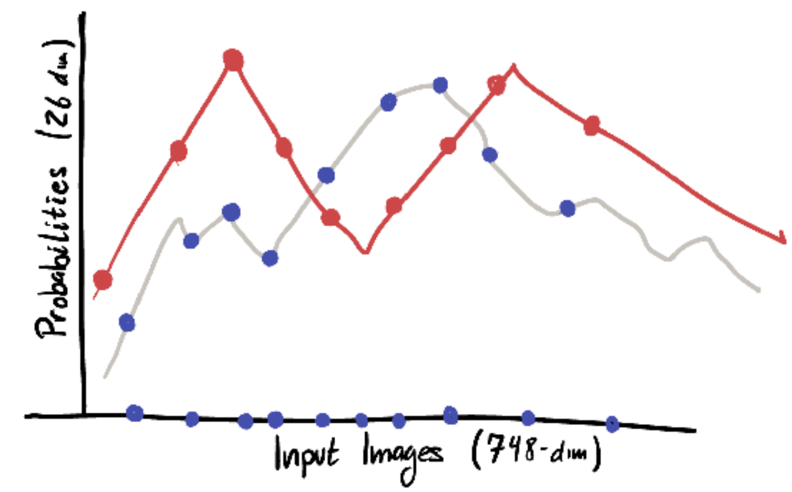


So, we can try to change the numbers a bit, and see if the new network does better or worse with respect to our loss function.

Slightly tweaking a neural network is as easy as slightly changing the numbers that determine it!



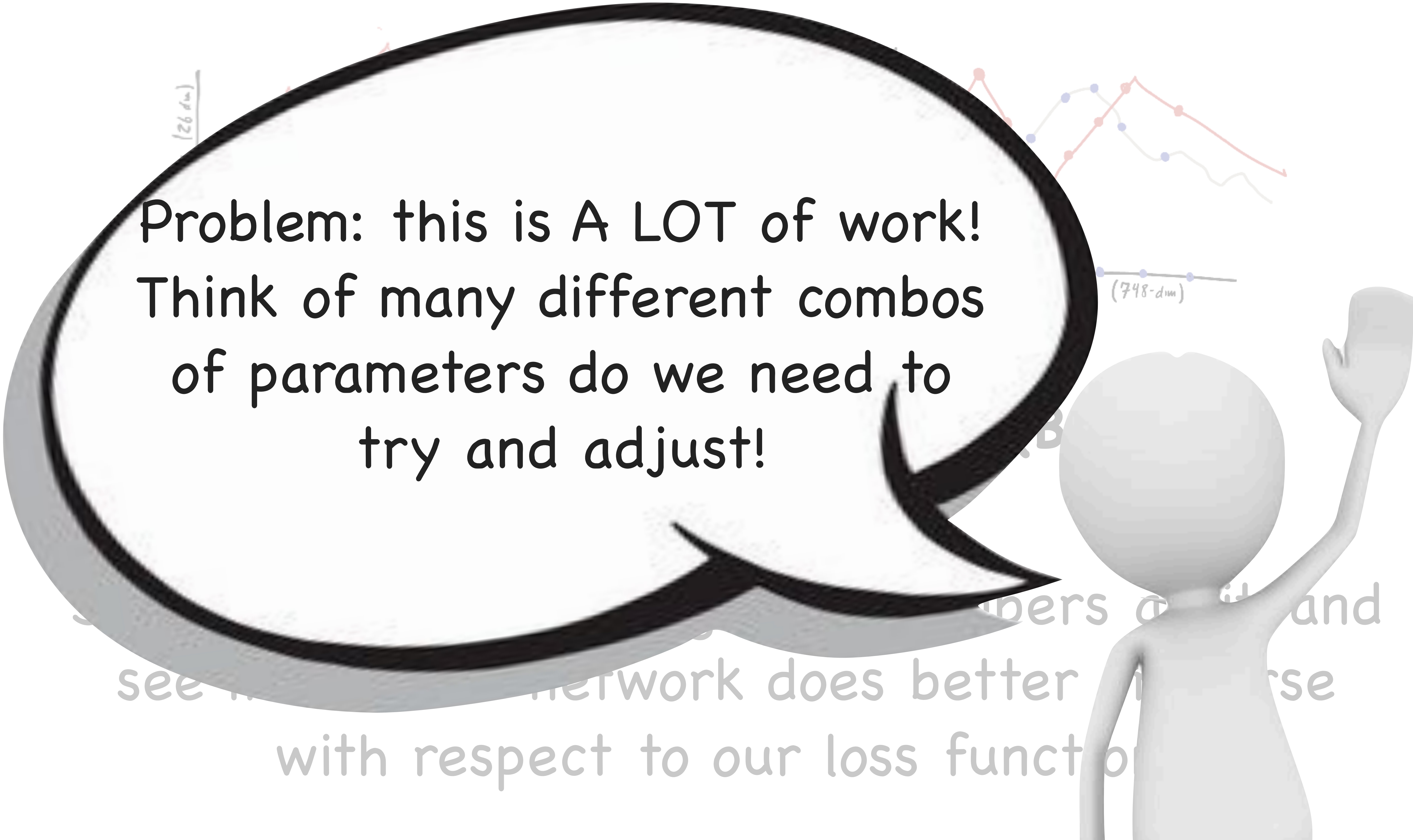
loss(B1) = 6

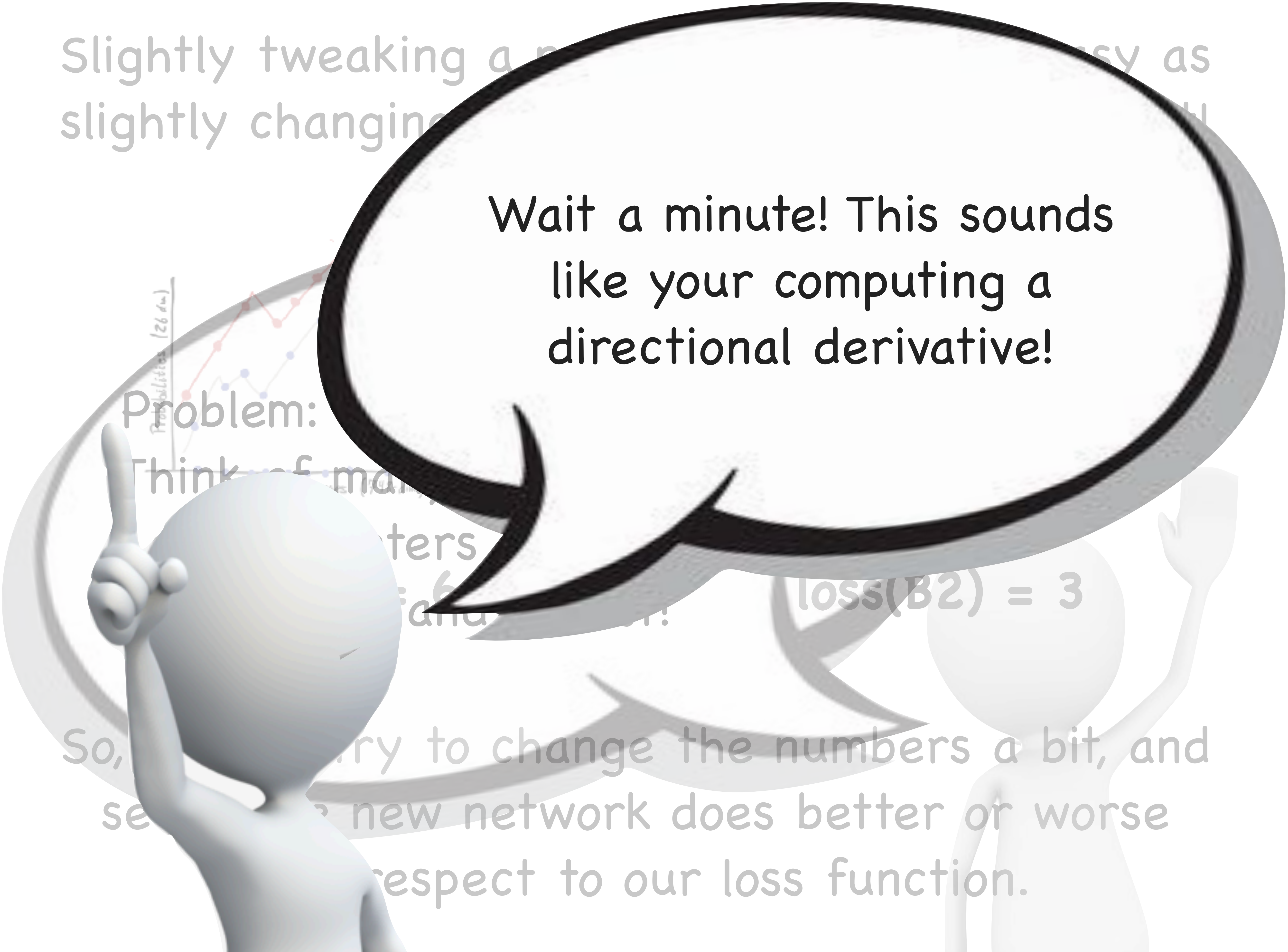


loss(B2) = 3

So, we can try to change the numbers a bit, and see if the new network does better or worse with respect to our loss function.

Slightly tweaking a neural network is as easy as slightly changing the numbers that determine it!




Wait a minute! This sounds like your computing a directional derivative!
Slightly tweaking a
sy as
slightly changin
Problem:
ters
loss(B2) = 3
ry to change the numbers a bit, and
new network does better or worse
espect to our loss function.

Original
Network

B

Slightly adjusted
Network

B+h

Original Network

B

Slightly adjusted Network

B+h

Original Loss

loss(B)

Slightly adjusted Loss

loss(B+h)

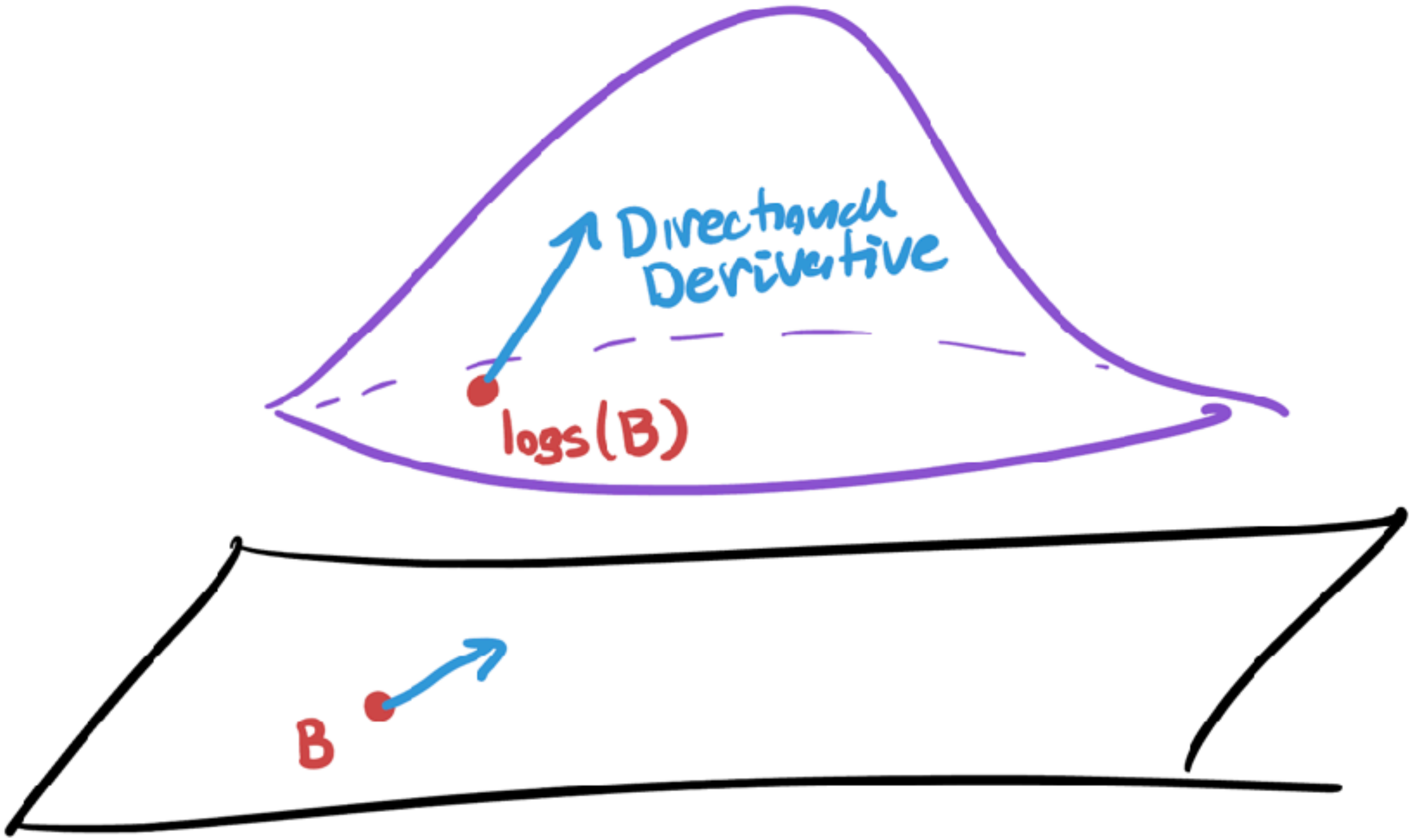
Net improvement of the change
(normalized by size of change).

$$\frac{\text{loss}(B+h)-\text{loss}(B)}{h}$$

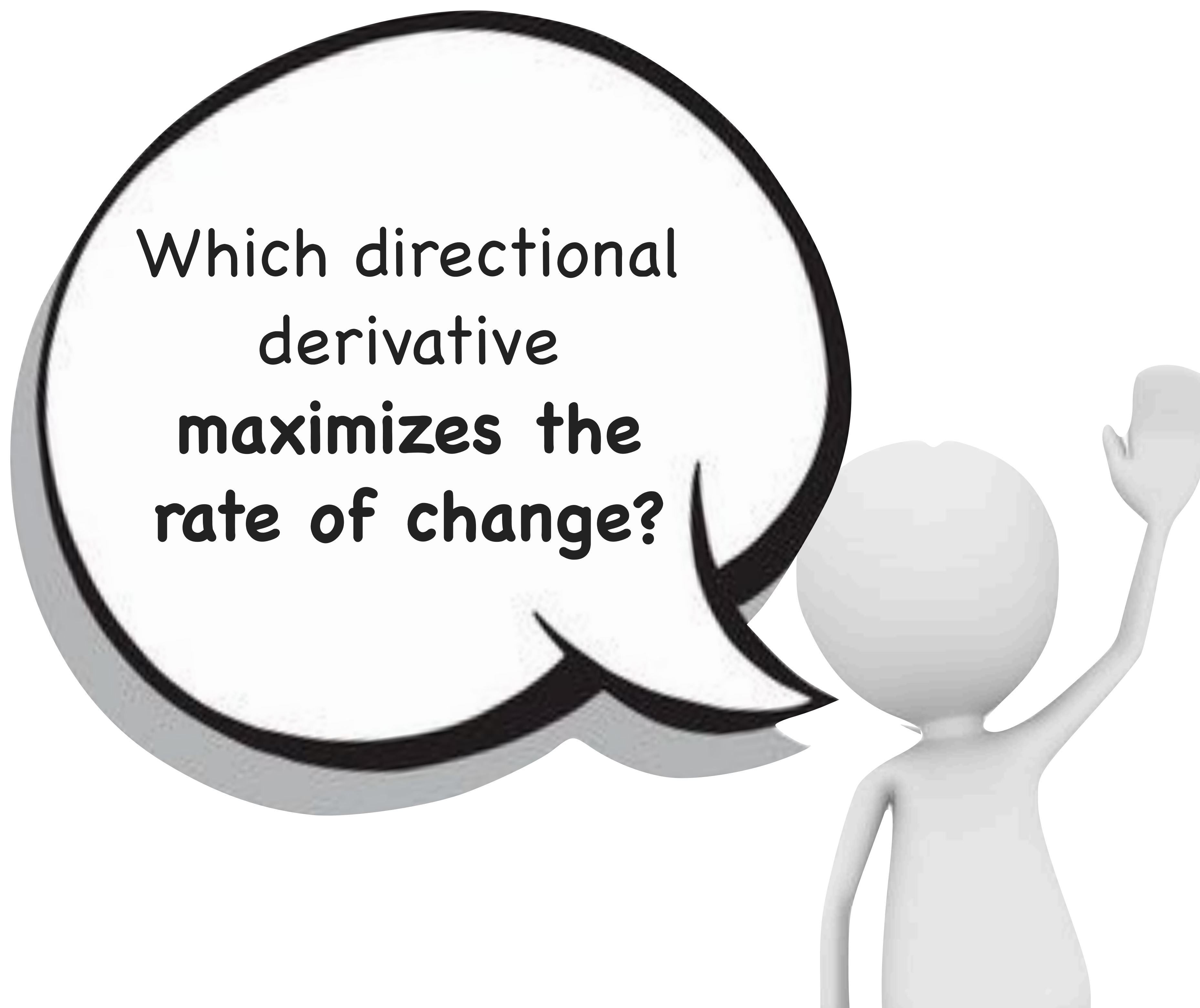
Net improvement of the change
(normalized by size of change).

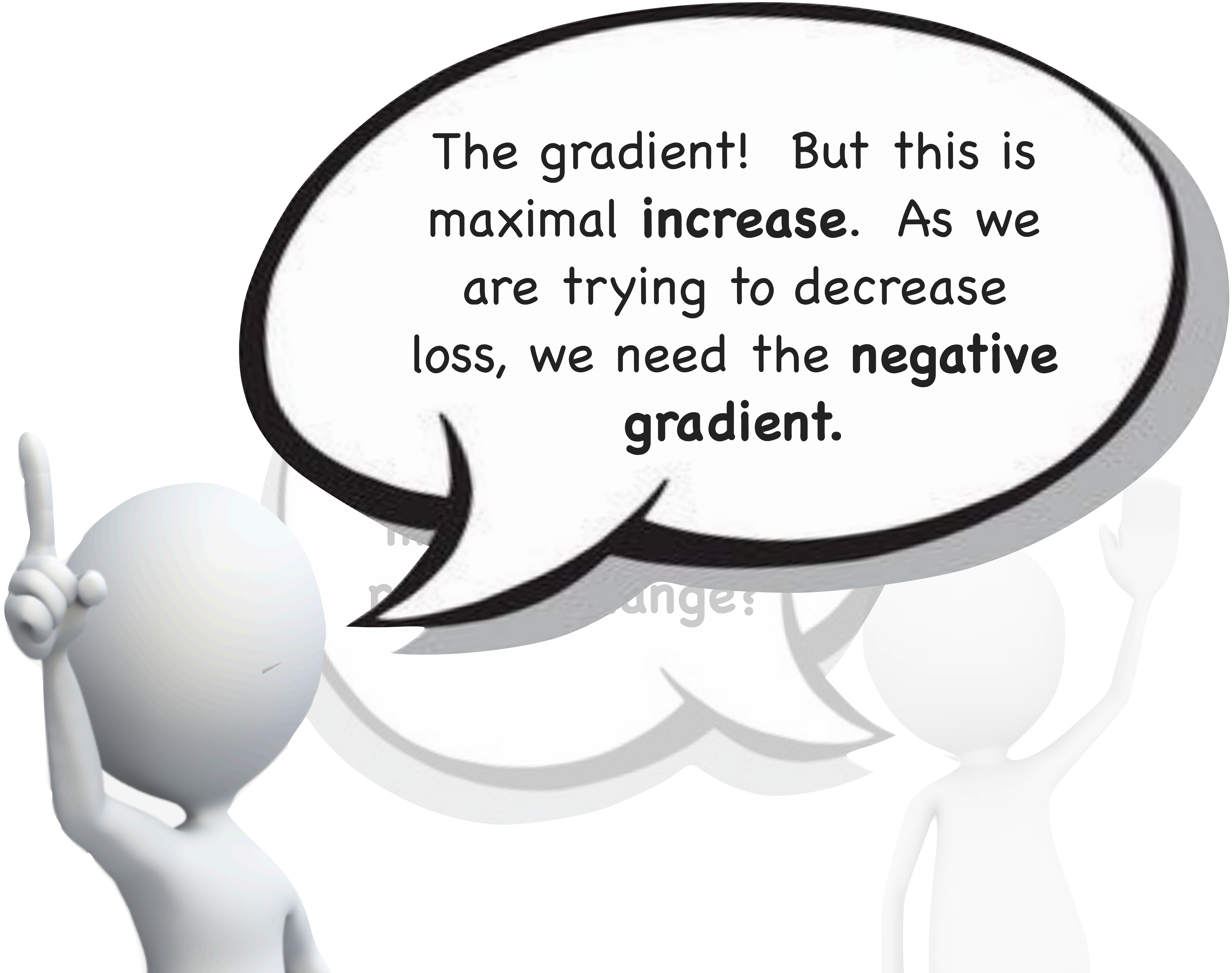
$$\frac{\text{loss}(B+h)-\text{loss}(B)}{h}$$

This is the **directional derivative of loss in the direction h,** on the space of networks!

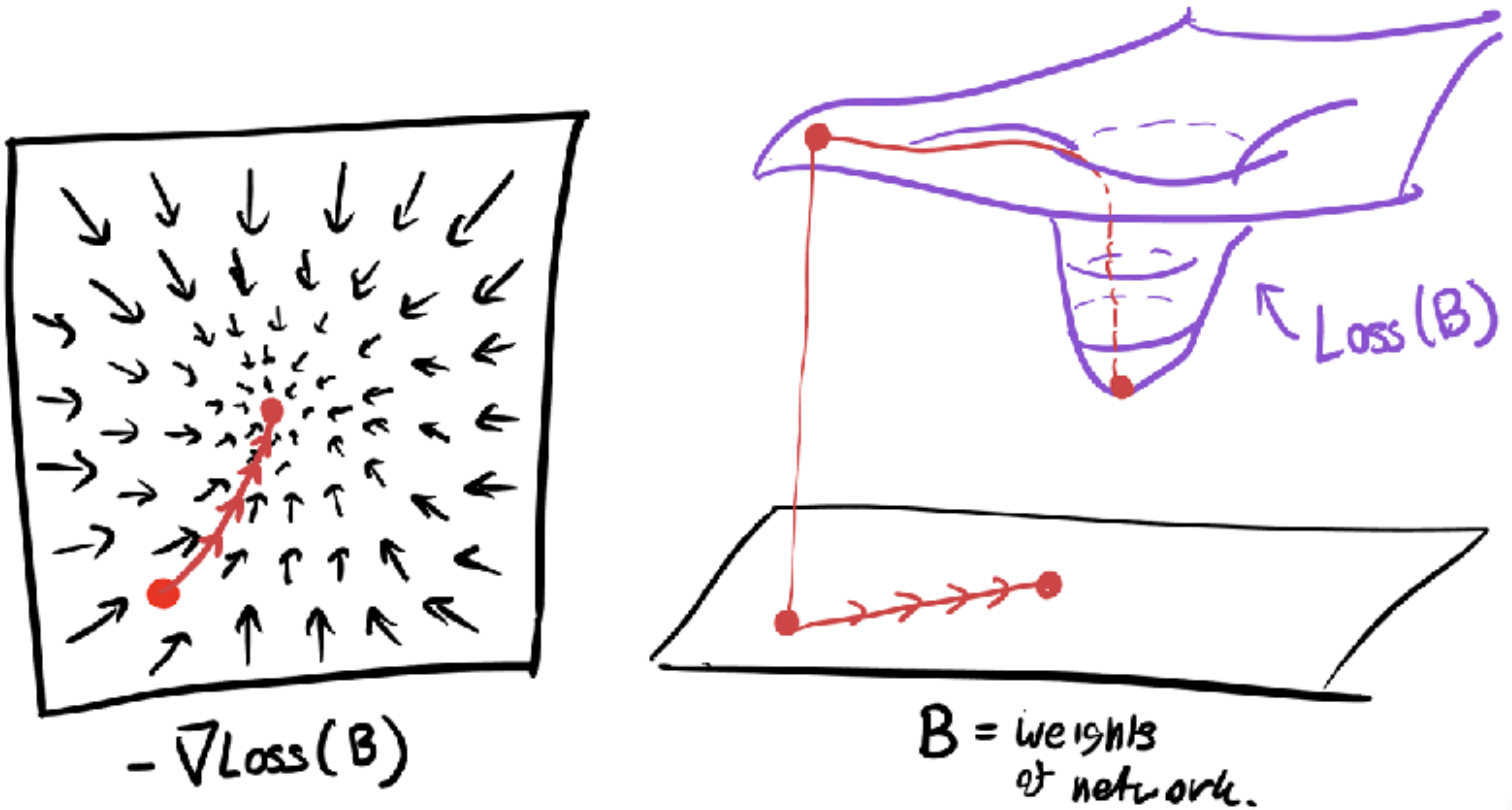


This is the **directional derivative of loss in the direction h**, on the space of networks!


Which directional derivative **maximizes the rate of change?**


The gradient! But this is maximal **increase**. As we are trying to decrease loss, we need the **negative gradient.**

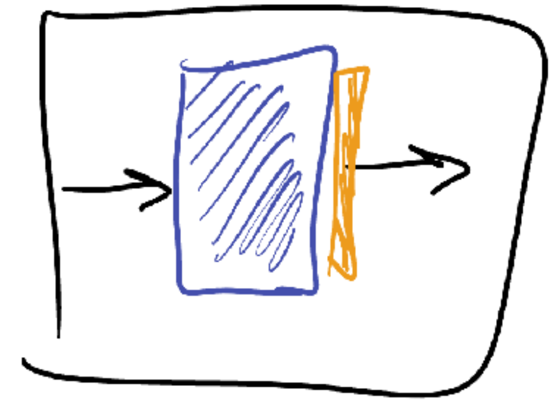
Gradient Descent: Following the negative gradient leads to a minimum - a set of weights which minimize the loss!

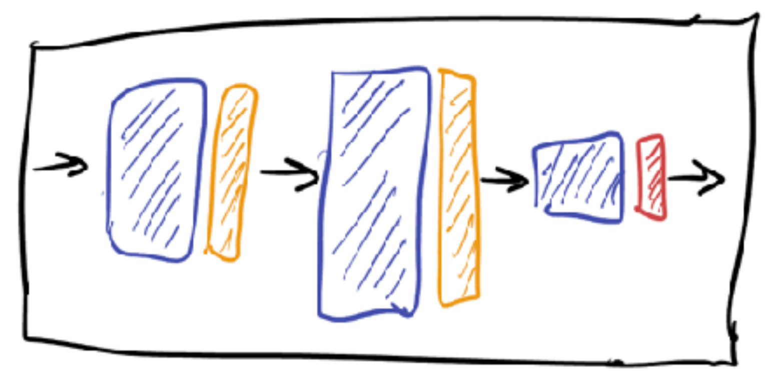


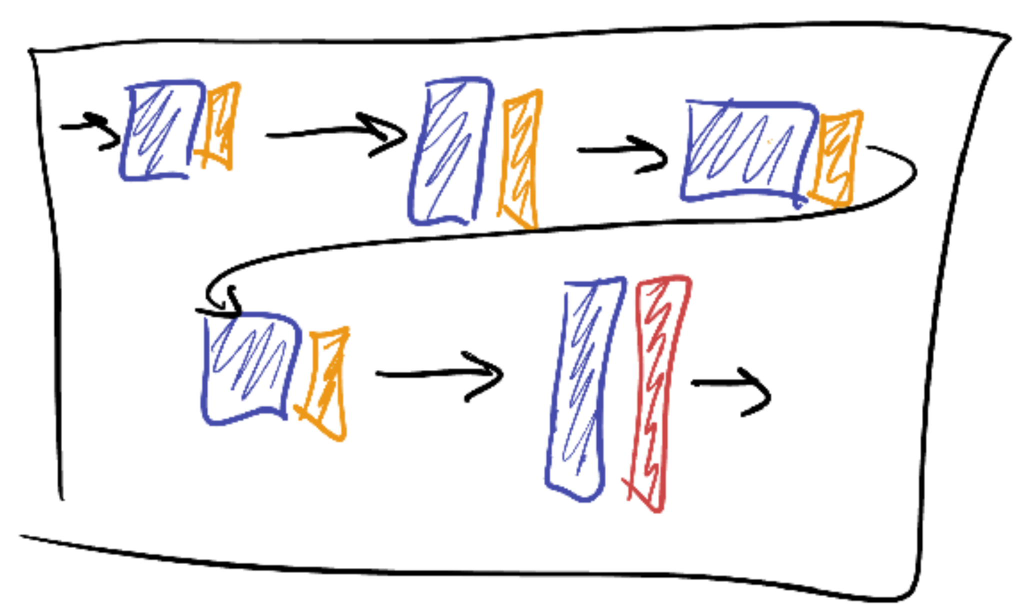
*Warning: I'm ignoring a TON of technical details here... the space we have to search through is huge, and there could be many local minima that are not global minima...

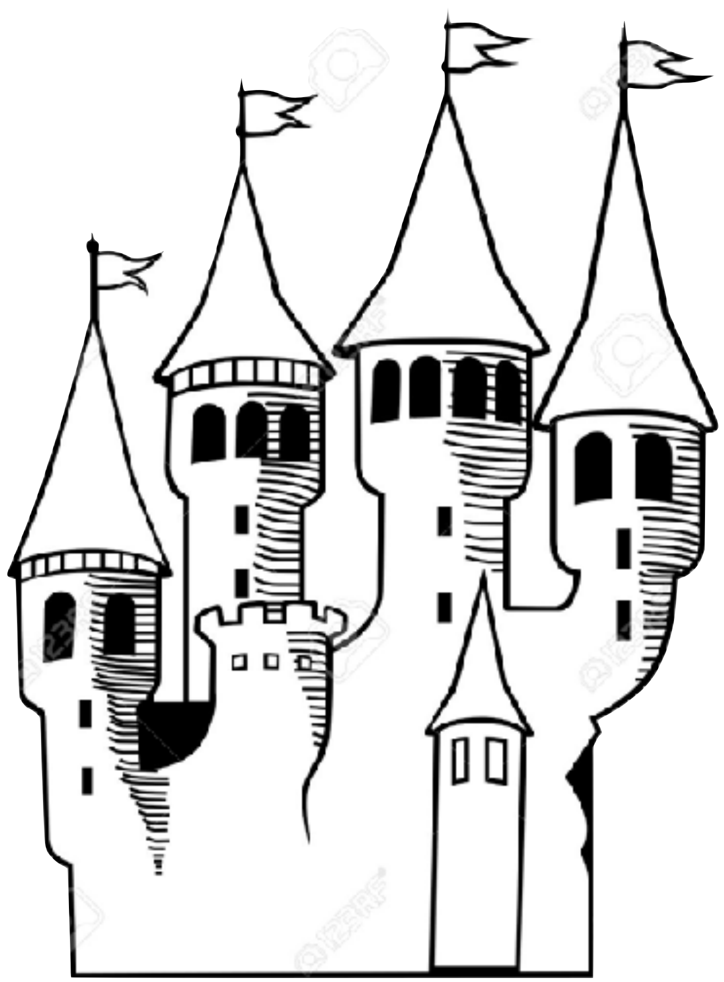

This tells us how to get the best network of a given architecture….but how do we find the best architecture?

Universal Approximation doesn't tell us how many bricks (layers) or what size (number of weights) are needed…just that there is some combination that works.









Universal A

doe

b