Autoencoders

Autoencoders

- **Autoencoders** are
  - Artificial neural networks
  - Capable of learning efficient representations of the input data, called **codings**, without any supervision
  - The training set is unlabeled.
- These codings typically have a much lower dimensionality than the input data, making autoencoders useful for **dimensionality reduction**

Autoencoders

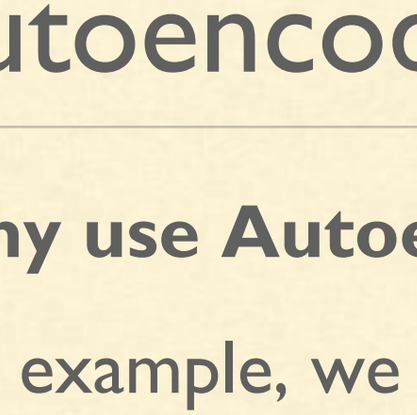
Why use Autoencoders?

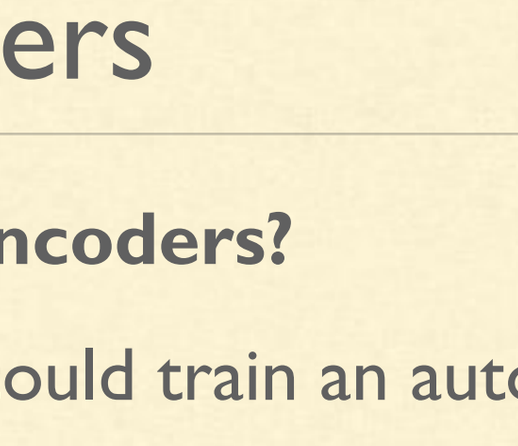
- Useful for dimensionality reduction
- Autoencoders act as powerful feature detectors,
- And they can be used for unsupervised pre-training of deep neural networks
- Lastly, they are capable of randomly generating new data that looks very similar to the training data; this is called a **generative model**

Autoencoders

Why use Autoencoders?

For example, we could train an autoencoder on pictures of faces, and it would then be able to generate new faces





Autoencoders

- Surprisingly, autoencoders work by simply learning to copy their inputs to their outputs
- This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult

Autoencoders

For example

- You can limit the size of the internal representation, or you can add noise to the inputs and train the network to recover the original inputs.
- These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data
- In short, the codings are byproducts of the autoencoder's attempt to learn the identity function under some constraints

Autoencoders

What we'll learn ?

- We will explain in more depth how autoencoders work
- What types of constraints can be imposed
- And how to implement them using TensorFlow, whether it is for
  - Dimensionality reduction,
  - Feature extraction,
  - Insupervised pretraining,
  - Or as generative models

Efficient Data Representations

Let's try to understand

"why constraining an

autoencoder during training pushes it to discover and exploit

patterns in the data"

With an example

Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

Efficient Data Representations

- At first glance, it would seem that the first sequence should be easier, since it is **much shorter**
- However, if you look carefully at the second sequence, you may notice that it follows two simple rules:
  - Even numbers are followed by their half,
  - And odd numbers are followed by their triple plus one

This is a famous sequence known as the **hailstone sequence**

Efficient Data Representations

- Once you notice this pattern, the second sequence becomes much easier to memorize than the first because you only need to memorize the two rules,
  - The first number,
  - And the length of the sequence

Efficient Data Representations

- Note that if we could quickly and easily memorize very long sequences, you would not care much about the **existence of a pattern** in the second sequence
- You would just learn every number by heart, and that would be that
- It is the fact that it is **hard to memorize long sequences** that makes it useful to **recognize patterns**,
- And hopefully this clarifies **why constraining an autoencoder during training pushes it to discover and exploit patterns in the data**

Efficient Data Representations

The relationship between memory, perception, and pattern matching was famously studied by **William Chase and Herbert Simon** in the early 1970s

William Chase

Herbert Simon

Efficient Data Representations

Let's see how their study of chess players is similar to an Autoencoder

- They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just **5 seconds**,
- A task that most people would find impossible
- However, this was only the case when the pieces were placed in realistic positions from actual games, not when the pieces were placed randomly.

Efficient Data Representations

Let's see how their study of chess players is similar to an

Autoencoder

- Chess experts don't have a much better memory than you and I,
- They just see chess patterns more easily thanks to their experience with the game
- Noticing patterns helps them store information efficiently

Efficient Data Representations

Let's see how their study of chess players is similar to an

Autoencoder

- Just like the chess players in this memory experiment, **an autoencoder**
  - looks at the inputs,
  - converts them to an efficient internal representation,
  - and then spits out something that (hopefully) looks very close to the inputs

Efficient Data Representations

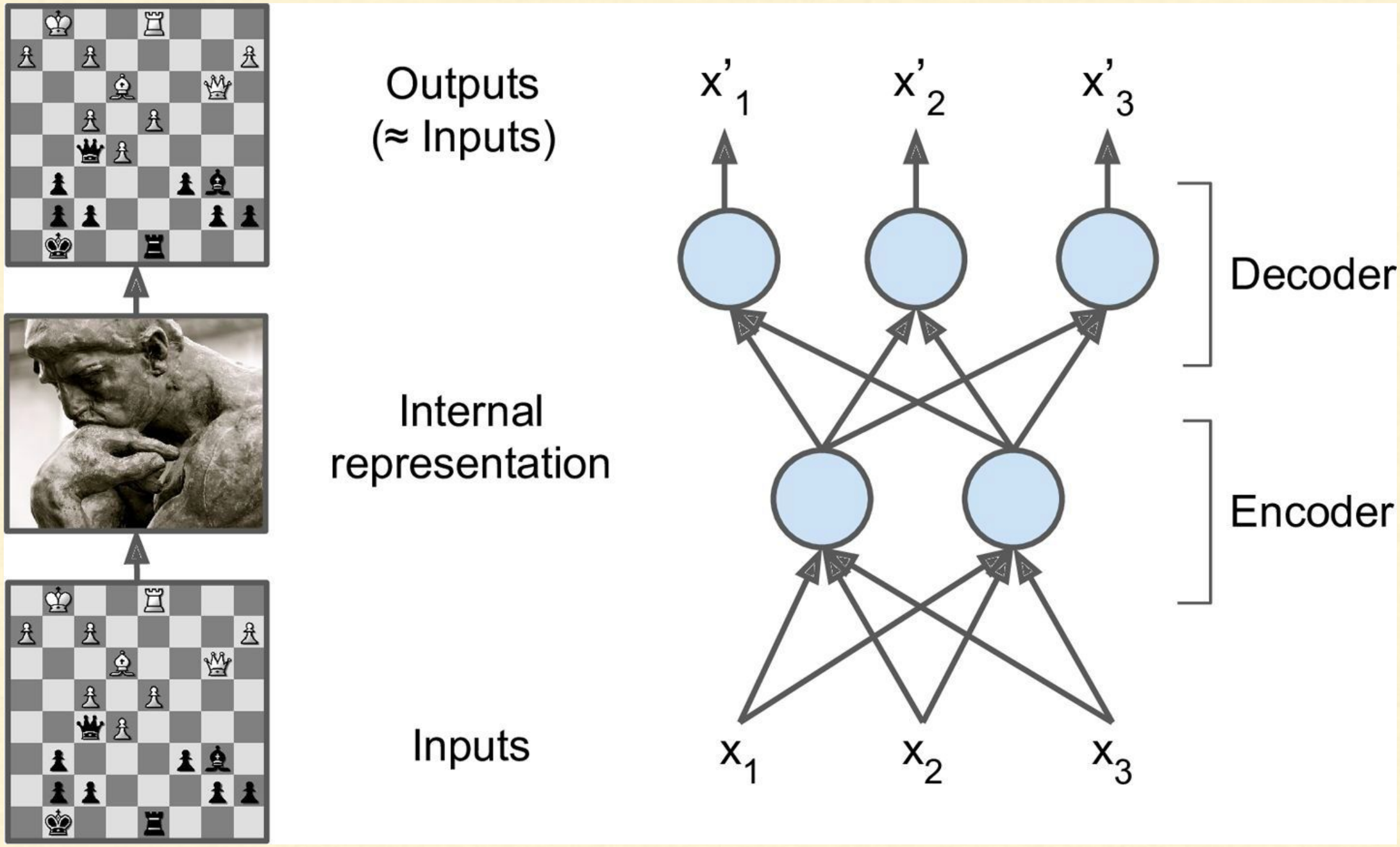
Composition of Autoencoder

An autoencoder is always composed of **two parts**:

- **An encoder** or **recognition network** that converts the inputs to an internal representation,
- Followed by **a decoder** or **generative network** that converts the internal representation to the outputs

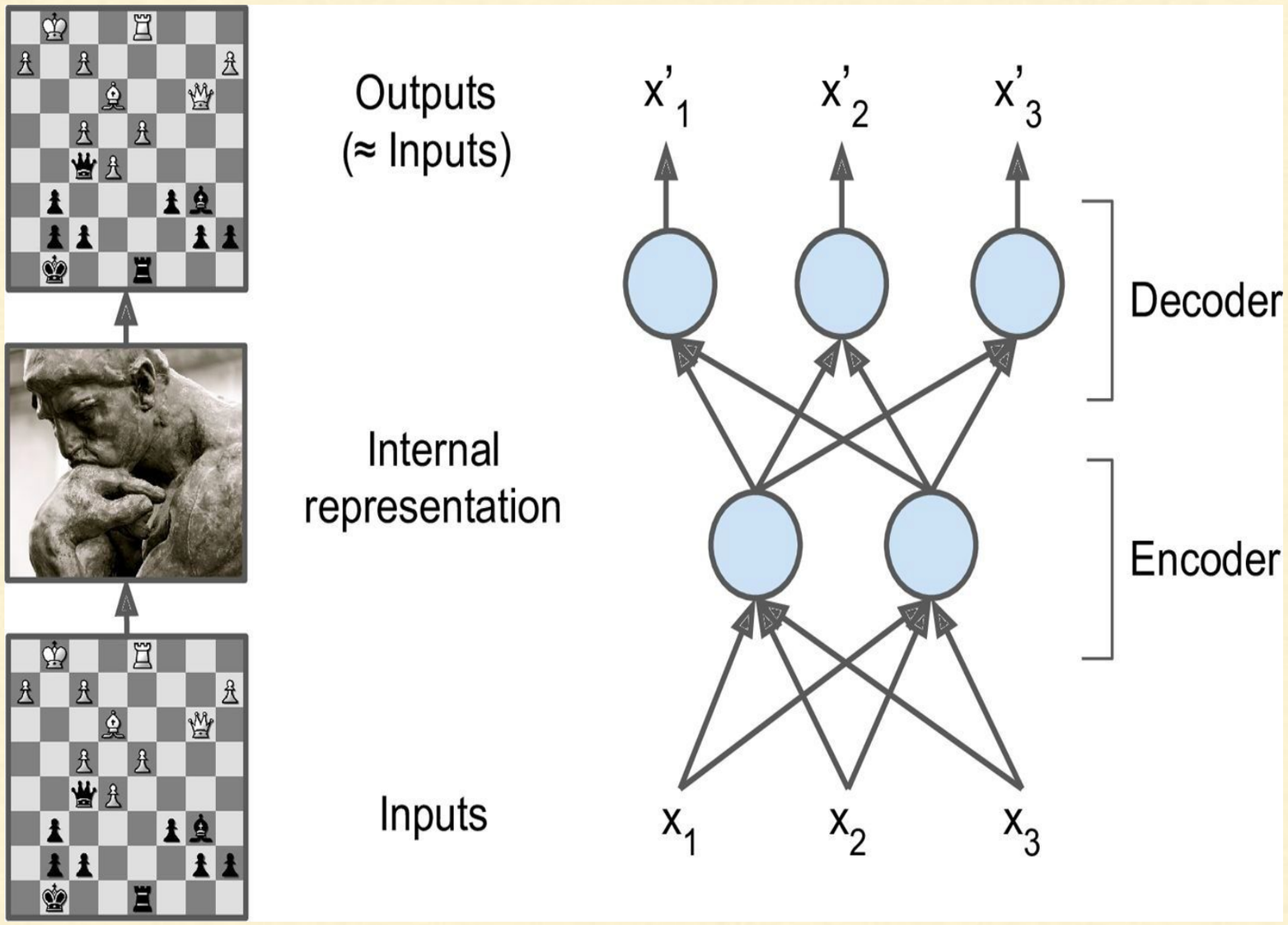
Efficient Data Representations

Composition of Autoencoder



Efficient Data Representations

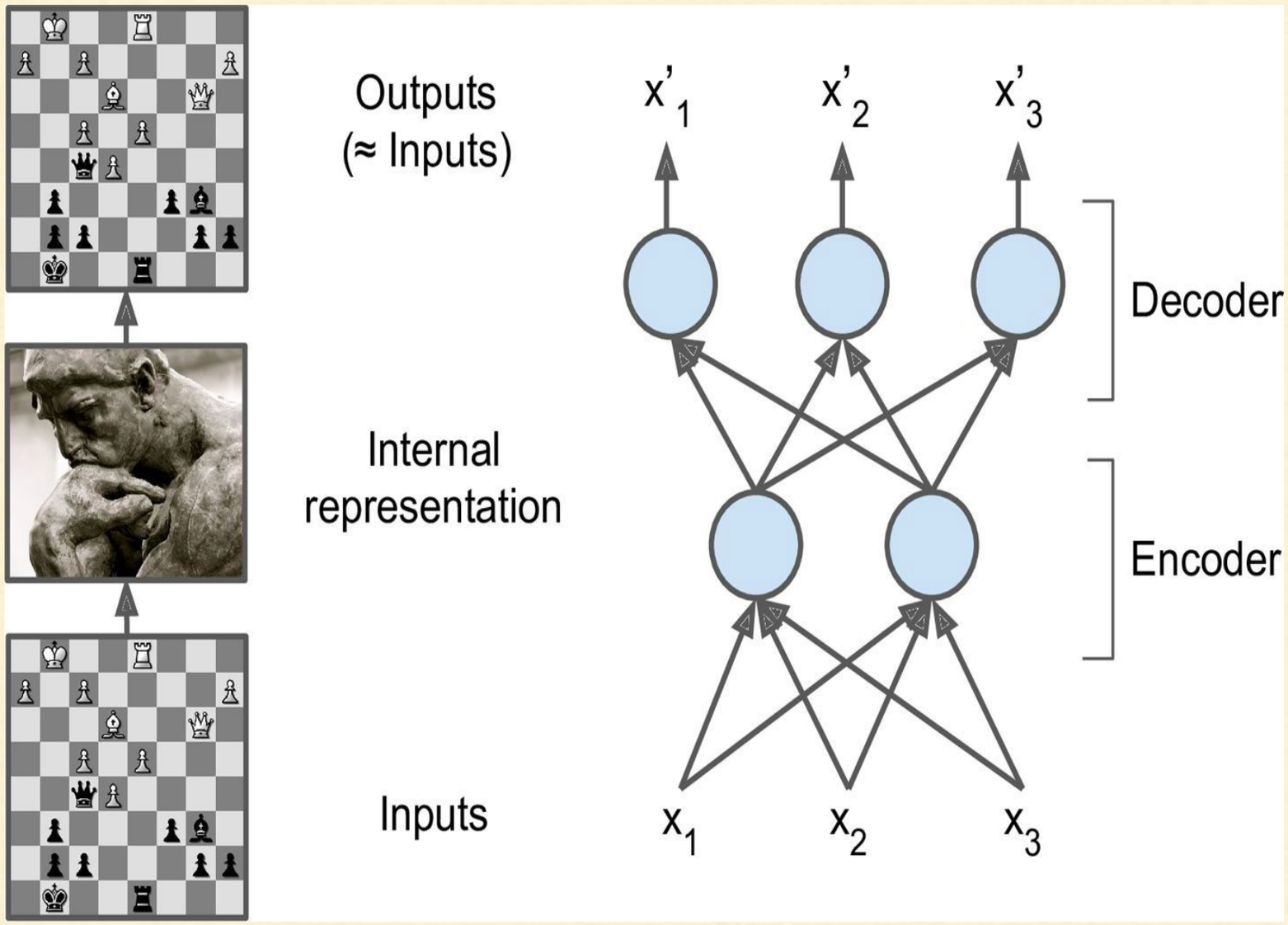
Composition of Autoencoder



An **autoencoder** typically has the same architecture as a **Multi-Layer Perceptron**, except that the **number of neurons** in the output layer must be **equal to the number of inputs**

Efficient Data Representations

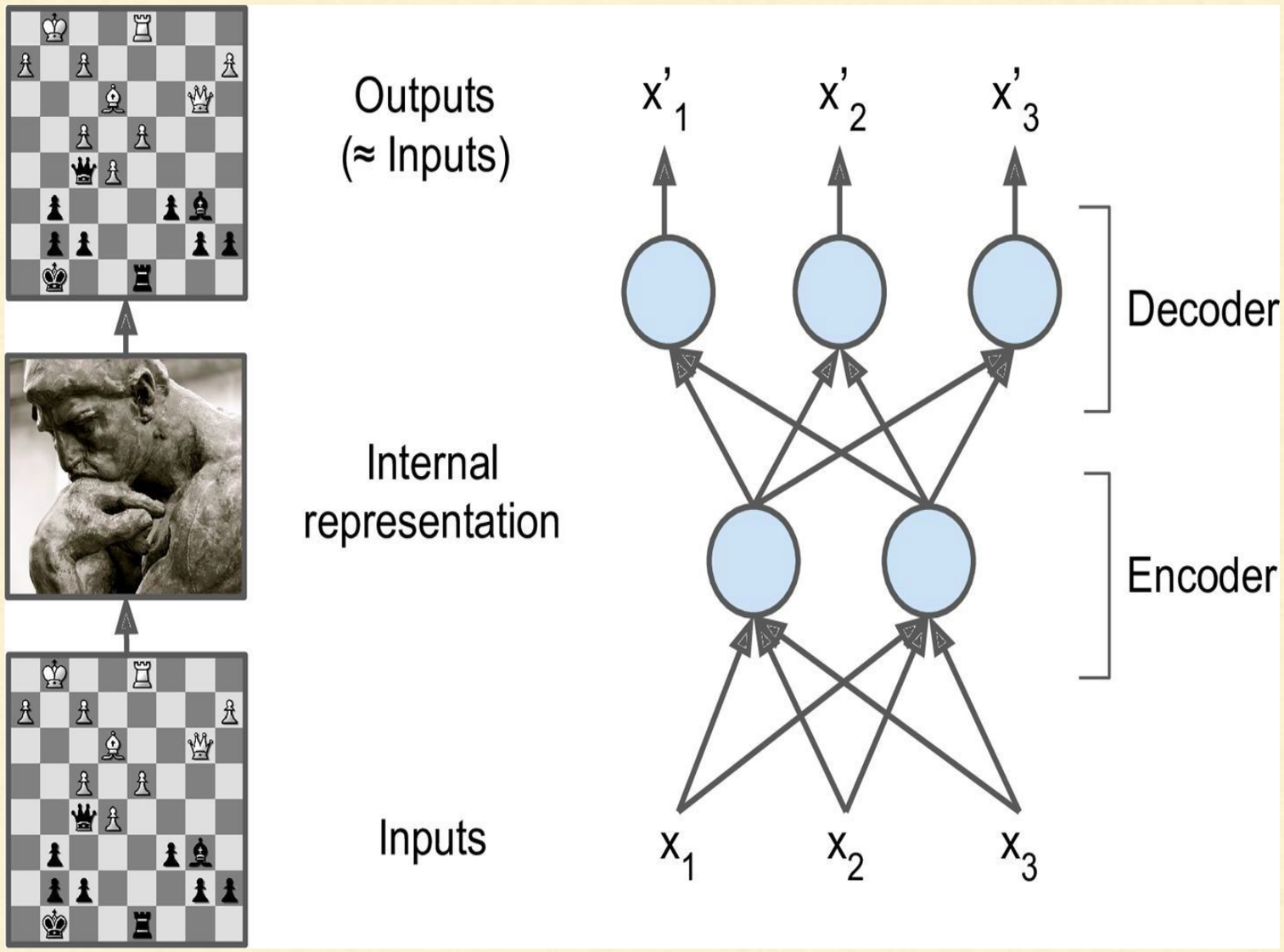
Composition of Autoencoder



There is just one hidden layer composed of two neurons **(the encoder)**, and **one output layer** composed of three neurons **(the decoder)**

Efficient Data Representations

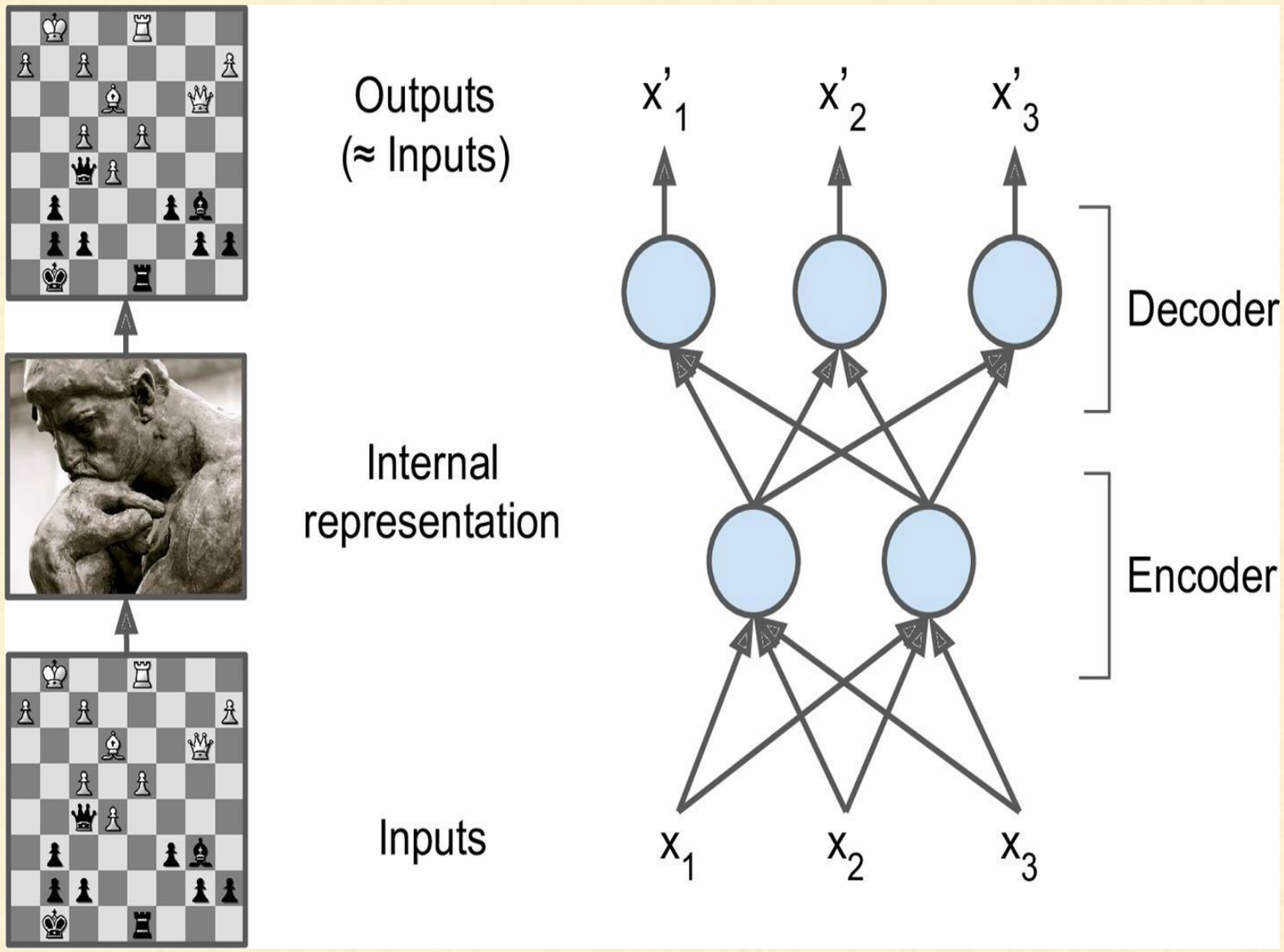
Composition of Autoencoder



The outputs are often called **the reconstructions** since the autoencoder tries to reconstruct the inputs, and the cost function contains a **reconstruction loss** that penalizes the model when the reconstructions are different from the inputs.

Efficient Data Representations

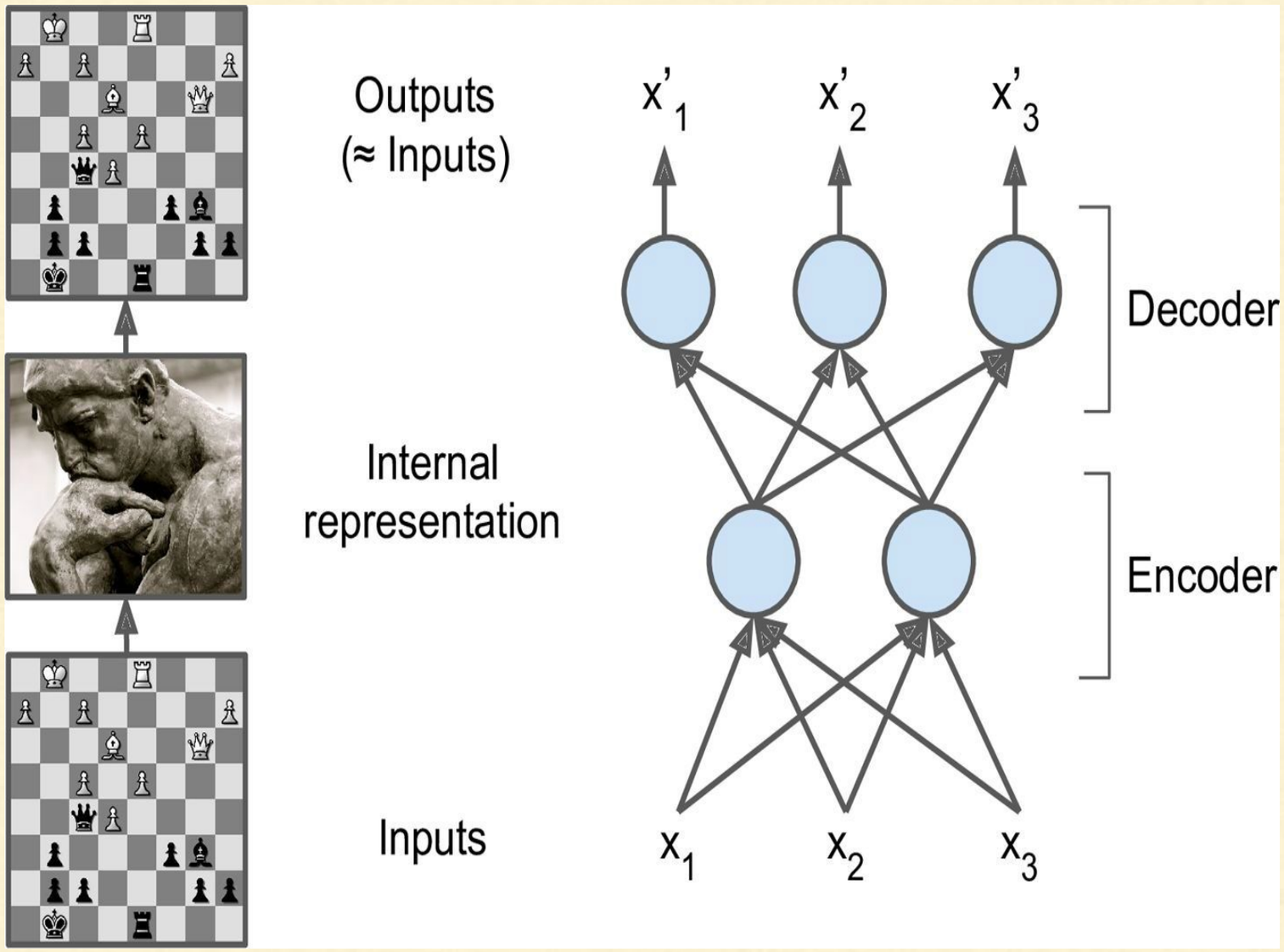
Composition of Autoencoder



Because the internal representation has a lower dimensionality than the input data, **it is 2D instead of 3D**, the autoencoder is said to be **undercomplete**

Efficient Data Representations

Composition of Autoencoder



- An **undercomplete** autoencoder cannot trivially copy its inputs to the **codings**, yet it must find a way to output a copy of its inputs
- It is forced to learn the most important features in the input data and drop the unimportant ones

Let's implement a very simple undercomplete autoencoder for dimensionality reduction

PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses **only linear activations** and the cost function is the **Mean Squared Error (MSE)**, then it can be shown that it ends up performing **Principal Component Analysis**

Now we will build a simple linear autoencoder to perform PCA on a 3D dataset, projecting it to 2D

PCA with an Undercomplete Linear Autoencoder

```
>>> import tensorflow as tf
>>> from tensorflow.contrib.layers import fully_connected
>>> n_inputs = 3 # 3D inputs
>>> n_hidden = 2 # 2D codings
>>> n_outputs = n_inputs
>>> learning_rate = 0.01
>>> X = tf.placeholder(tf.float32, shape=[None, n_inputs])
>>> hidden = fully_connected(X, n_hidden, activation_fn=None)
>>> outputs = fully_connected(hidden, n_outputs, activation_fn=None)
>>> reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
>>> optimizer = tf.train.AdamOptimizer(learning_rate)
>>> training_op = optimizer.minimize(reconstruction_loss)
>>> init = tf.global_variables_initializer()
```

Run it on Notebook

PCA with an Undercomplete Linear Autoencoder

The two things to note in the previous code are are:

- The number of **outputs** is equal to the number of **inputs**
- To perform simple PCA, we set **activation_fn=None** i.e., all neurons are linear, and the cost function is the MSE.

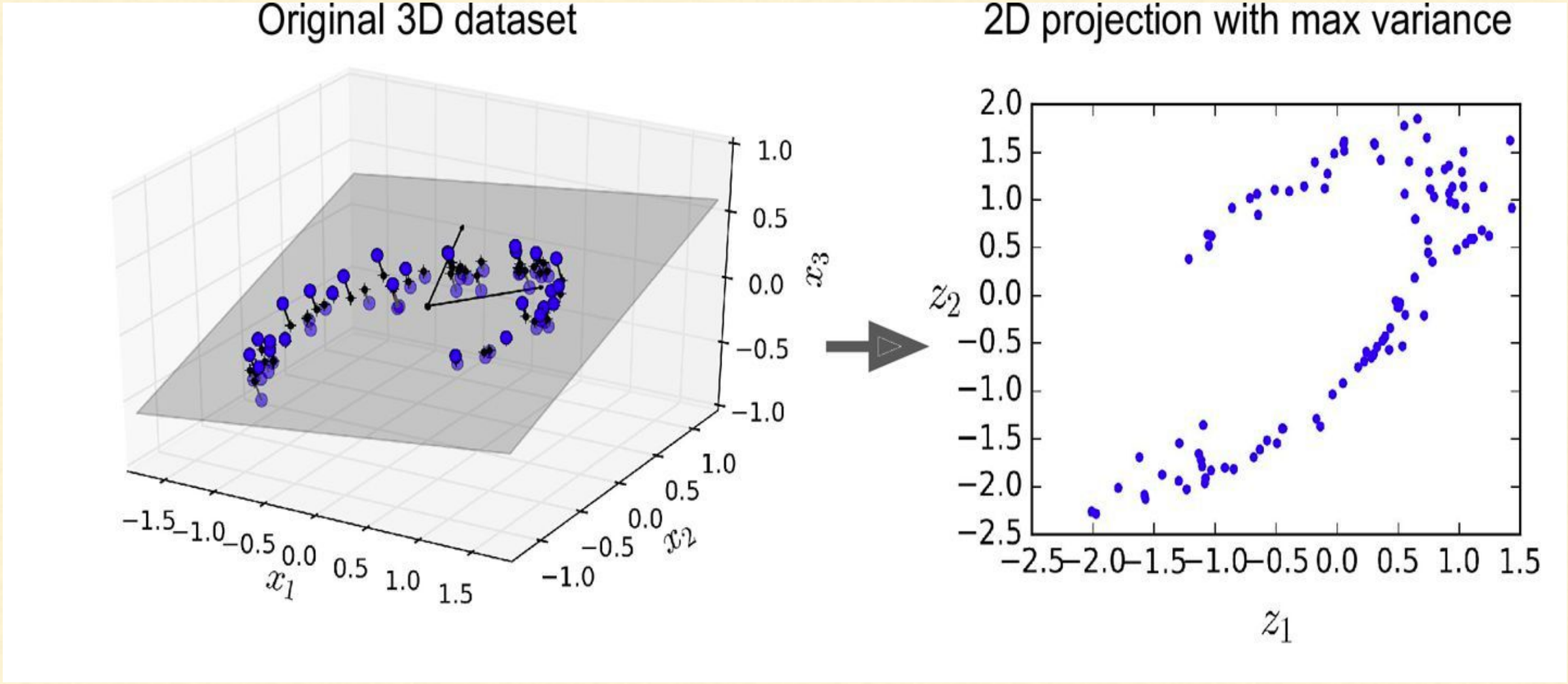
PCA with an Undercomplete Linear Autoencoder

Now let's load the dataset, train the model on the training set, and use it to encode the test set i.e., project it to 2D

```
>>> X_train, X_test = [...] # load the dataset
>>> n_iterations = 1000
>>> codings = hidden # the output of the hidden layer provides the
codings
>>> with tf.Session() as sess:
        init.run()
        for iteration in range(n_iterations):
            training_op.run(feed_dict={X: X_train}) # no labels
        codings_val = codings.eval(feed_dict={X: X_test})
```

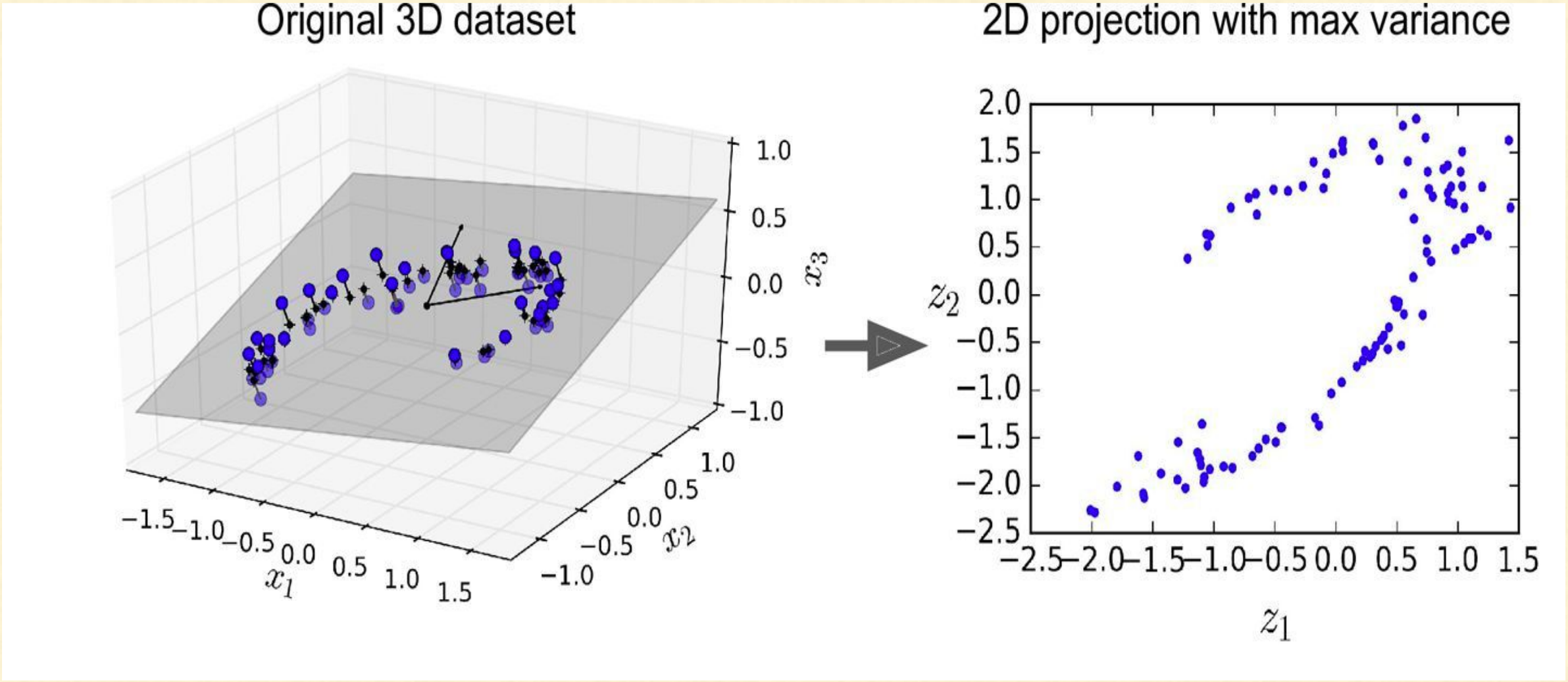
Run it on Notebook

PCA with an Undercomplete Linear Autoencoder



Above figure shows the original 3D dataset (at the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, at the right)

PCA with an Undercomplete Linear Autoencoder



As you can see, the autoencoder found the best 2D plane to project the data onto, preserving as much variance in the data as it could just like PCA

Questions?

https://discuss.cloudxlab.com

reachus@cloudxlab.com