CITS5508 Machine Learning
Lectures for Semester 1, 2021
Lecture Week 12: Book Chapter 17

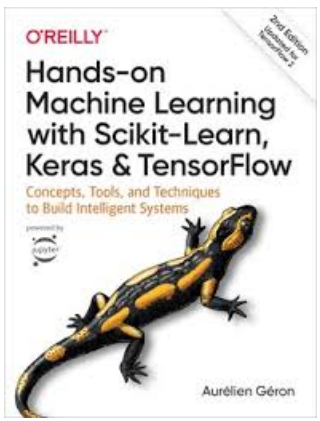
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 17.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Representation Learning and Generative Learning Using Autoencoders and GANs

Here are the main topics we will cover:

- What are Autoencoders?
- Efficient data representations
- Encoder and decoder in an autoencoder
- Performing PCA using an *undercomplete* linear autoencoder
- Stacked autoencoders and how to implement one using Keras in TensorFlow
- Tying the weights in a stacked autoencoder and visualizing the reconstruction
- Unsupervised pre-training using stacked autoencoders
- Convolutional Autoencoders and Denoising autoencoders

What Are Autoencoders?

- Autoencoders are artificial neural networks capable of learning efficient representations of the input data, called *codings*, without any supervision (i.e., the training set is unlabelled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see Chapter 8).
- More importantly, autoencoders act as powerful feature detectors, and they can be used for unsupervised pre-training of deep neural networks.
- Lastly, some autoencoders are *generative models*: they are capable of randomly generating new data that looks very similar to the training data. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces. However, the generated images are usually fuzzy and not entirely realistic.

Autoencoders

To prevent the autoencoders from simply learning to copy their inputs to their outputs, we can constrain the network in various ways to make the task more difficult. For example,

- you can limit the size of the internal representation, or
- 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.

Efficient Data Representations

- Just like the chess players who can easily see chess patterns to help them quickly memorize the positions of all chess pieces in a game, 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.
- An autoencoder is always composed of two parts:
  1. an *encoder* (or *recognition network*) that converts the inputs to an internal representation, followed by
  2. a *decoder* (or *generative network*) that converts the internal representation to the outputs.

Efficient Data Representations (cont.)

- An autoencoder typically has the same architecture as a Multi-Layer Perceptron (MLP; see Chapter 10), except that the number of neurons in the output layer must be equal to the number of inputs.

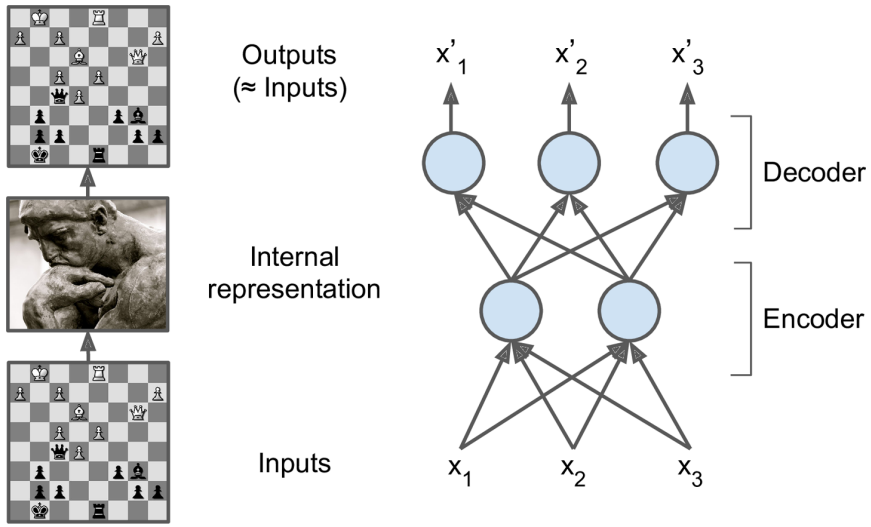


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

- In the example here, there is just one hidden layer composed of two neurons (the *encoder*). 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 (cont.)

- Because the internal representation in Figure 17-1 has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be *undercomplete*.
- 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 see how to implement a very simple undercomplete autoencoder for dimensionality reduction.

Performing 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 (see Chapter 8).
Example: Build a simple undercomplete autoencoder to perform PCA:

```
from tensorflow import keras

encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

Performing PCA with an Undercomplete Linear Autoencoder (cont.)

This code is very similar to the MLP code in a previous chapter, but there are a few things to note:

- We organized the autoencoder into two sub-components: the *encoder* and the *decoder*. Both are regular `Sequential` models with a single `Dense` layer, and the *autoencoder* is a `Sequential` model containing the encoder followed by the decoder.
- The autoencoder's number of outputs is equal to the number of inputs (i.e., 3).
- To perform simple PCA, we do not use any activation function (i.e., all neurons are linear), and the cost function is the MSE.

Performing PCA with an Undercomplete Linear Autoencoder (cont.)

Now let's generate the same 3D data used in a previous chapter:

```
def generate_3d_data(m, w1=0.1, w2=0.3, noise=0.1):
    angles = np.random.rand(m) * 3 * np.pi / 2 - 0.5
    data = np.empty((m, 3))
    data[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * np.random.randn(
    data[:, 1] = np.sin(angles) * 0.7 + noise * np.random.randn(m) / 2
    data[:, 2] = data[:, 0] * w1 + data[:, 1] * w2 + noise * np.random.randn(
    return data

X_train = generate_3d_data(60)
X_train = X_train - X_train.mean(axis=0, keepdims=0)
```

and train the model, and use it for decoding (i.e., project it to 2D):

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

Performing PCA with an Undercomplete Linear Autoencoder

Figure 17-2 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). 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).

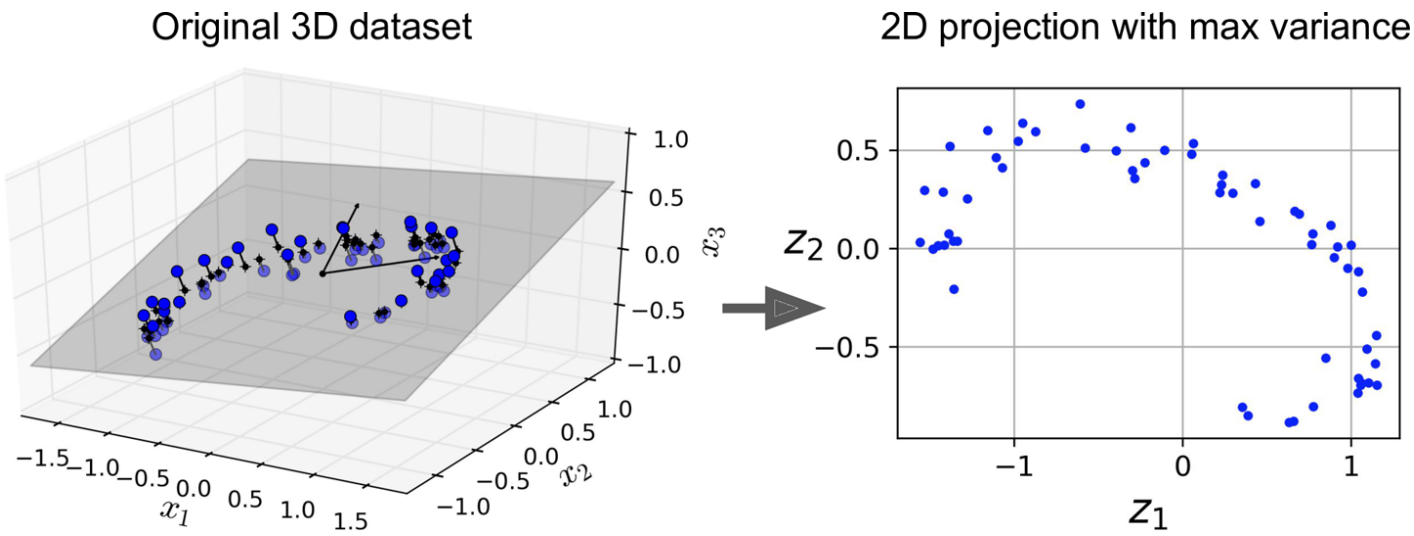


Figure 17-2. PCA performed by an undercomplete linear autoencoder

Stacked Autoencoders

- Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*).
- Adding more layers helps the autoencoder learn more complex codings. However, one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process (and it is unlikely to generalize well to new instances).

Stacked Autoencoders

- The architecture of a stacked autoencoder is typically symmetrical with regards to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich, e.g., an autoencoder for MNIST may have 784 inputs, then a hidden layer with 300 neurons, a central hidden layer of 150 neurons, then another hidden layer with 300 neurons, and an output layer with 784 neurons (Figure 17-3).

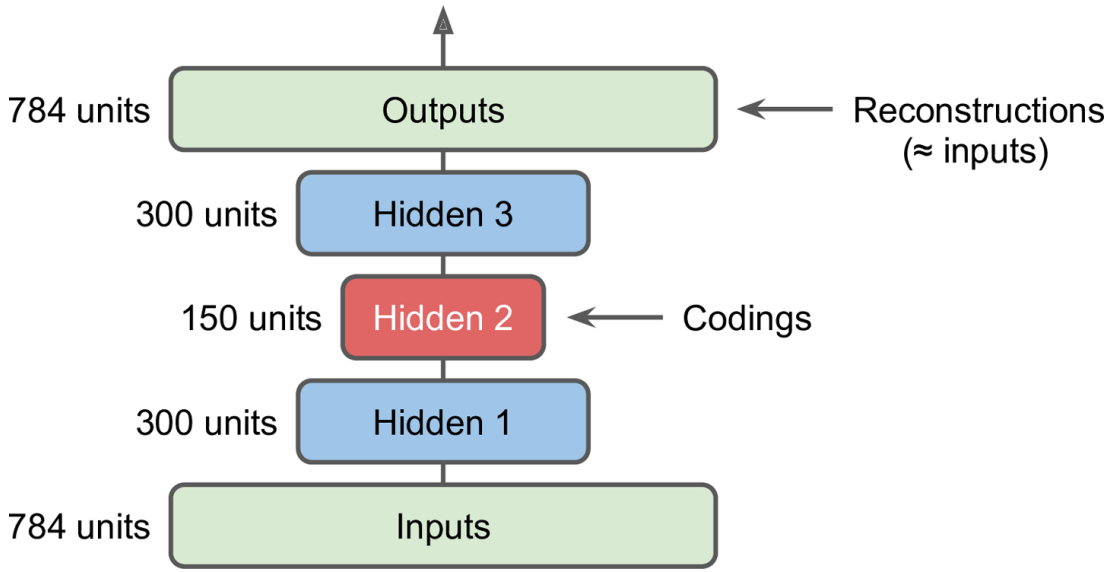


Figure 17-3. Stacked autoencoder

Implementing a Stacked Autoencoder Using Keras

You can implement a stacked autoencoder very much like a regular deep MLP. The same techniques we used for training deep nets can be applied.

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                   optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                         validation_data=[X_valid, X_valid])
```

Implementing a Stacked Autoencoder Using Keras (cont.)

Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant.

Figure 17-4. Original images (top) and their reconstructions (bottom)

The reconstructions are recognizable, but a bit too lossy. We may need to train the model for longer, or make the encoder and decoder deeper, or make the codings larger.

Implementing a Stacked Autoencoder Using Keras (cont.)

Visualizing Using T-SNE

Once we have reduced the datset's dimensionality, we can use another dimensionality reduction algorithm to see how well the lower-dimensional data is for the downstream classification task. T-SNE is a suitable algorithm as it reduces the data down to 2D for visualization.

```
from sklearn.manifold import TSNE
X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Implementing a Stacked Autoencoder Using Keras (cont.)

Visualizing Using T-SNE (cont.)

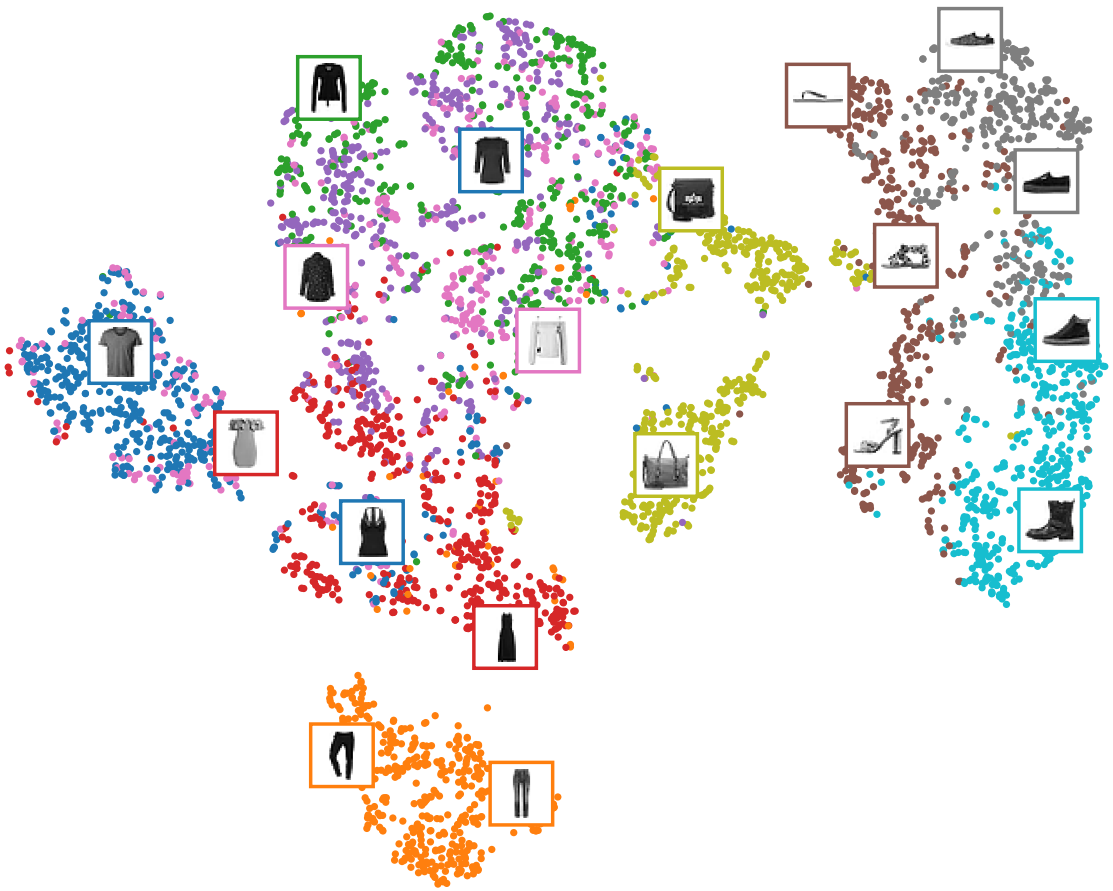


Figure 17-5. Fashion MNIST visualization using an autoencoder followed by t-SNE

Unsupervised Pretraining Using Stacked Autoencoders

If you are tackling a complex supervised task but you do not have a lot of labelled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. This makes it possible to train a high-performance model using little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing network.

Similarly, we can firstly train a stacked autoencoder using all the data. This training process does not require the data to be labelled. Once the stacked autoencoder is trained, we can reuse its lower layers in our neural network for the actual task. As our neural network has fewer layers that require training, it can cope with fewer labelled training data.

Unsupervised Pretraining Using Stacked Autoencoders (cont.)

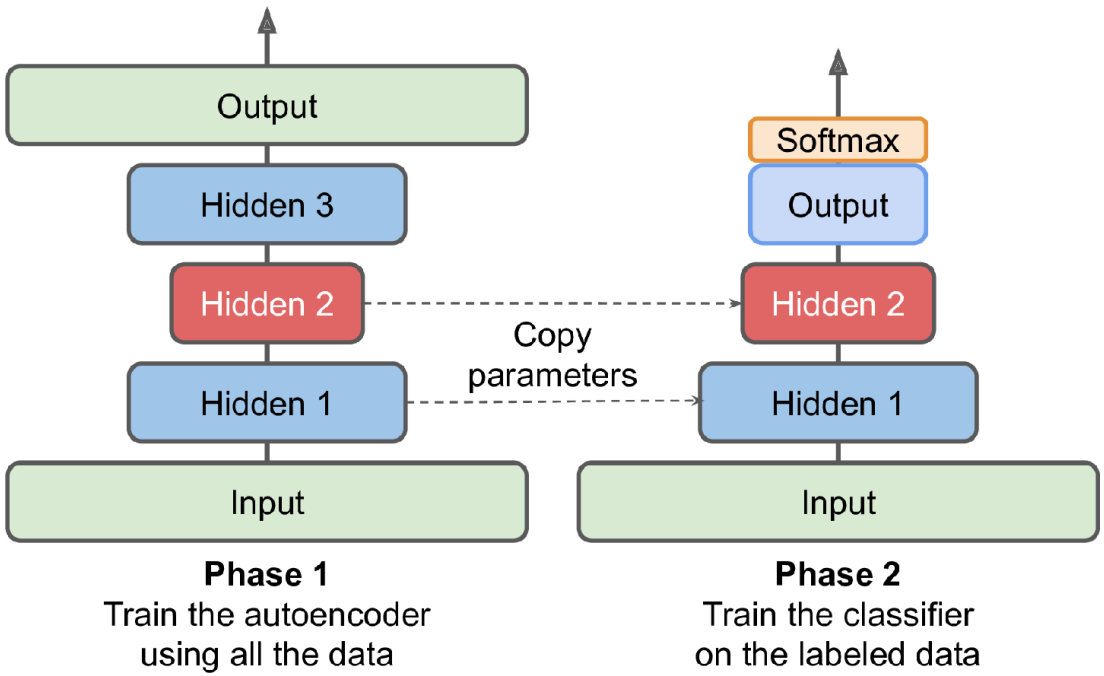


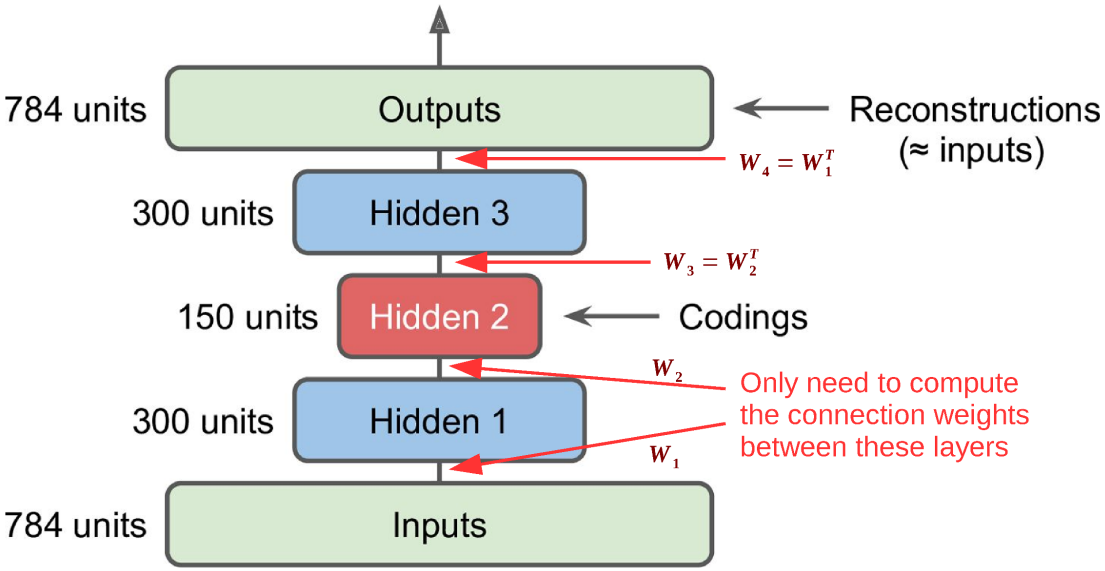
Figure 17-6. Unsupervised pretraining using autoencoders.

There is nothing special about the implementation: just train an autoencoder using all the training data (labeled plus unlabeled), then reuse its encoder layers to create a new neural network.

Stacked Autoencoders: Tying the Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie the weights* of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting.

Stacked Autoencoders: Tying the Weights (cont.)



Tying the weights in a stacked autoencoder ($N = 4$ in this example).

Specifically, if the autoencoder has a total of N layers (not counting the input layer), and $\mathbf{W}_\ell$ represents the connection weights of the ℓ^{th} layer (e.g., layer 1 is the first hidden layer, layer $\frac{N}{2}$ is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined simply as:

$$\mathbf{W}_{N-\ell+1} = \mathbf{W}_\ell^T \quad (\text{with } \ell = 1, 2, \cdots, N/2).$$

Stacked Autoencoders: Tying the Weights (cont.)

```
class DenseTranspose(keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias", initializer="zeros",
        shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

Stacked Autoencoders: Tying the Weights (cont.)

```
dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")
tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])

tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])
```

This model achieves a slightly lower reconstruction error than the previous model, with the number of parameters that require training reduced by almost half.

Training One Autoencoder at a Time

It is possible to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown below:

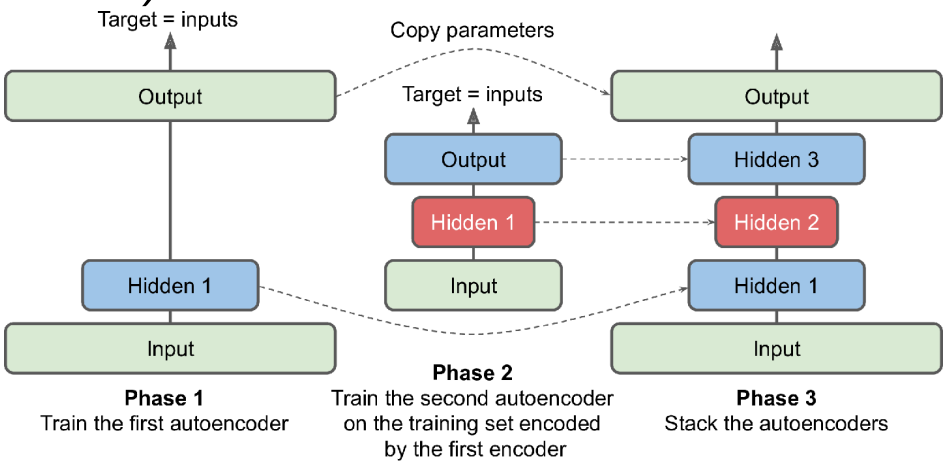


Figure 17-7. Training one autoencoder at a time

- **Phase 1:** The first autoencoder learns to reconstruct the inputs.
- **Phase 2:** (i) we encode the whole training set using this first autoencoder. (ii) We train a second autoencoder using the new (compressed) training set from phase 2.
- **Phase 3:** Finally, we build a big sandwich using all these autoencoders.

Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well. So if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a convolutional autoencoder.
The *encoder* of a convolutional autoencoder is just a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., number of channels).
The *decoder* must do the reverse operation (upscale the image and reduce its depth back to the original dimensions), and for this you can use *transpose convolutional layers* (alternatively, you could combine upsampling layers with convolutional layers).

Convolutional Autoencoders (cont.)

Here is a simple convolutional autoencoder for Fashion MNIST:

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])

conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3, strides=2, padding="valid",
                                 activation="selu", input_shape=[3, 3, 64]),
    keras.layers.Conv2DTranspose(16, kernel_size=3, strides=2, padding="same",
                                 activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding="same",
                                 activation="sigmoid"),
    keras.layers.Reshape([28, 28])
  ])
conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])
```

Other Types of Autoencoders

So far we have seen various kinds of autoencoders:
- basic, stacked, and convolutional.

We have looked at how to train them :
- either in one shot or in several phases.

We have also looked at a couple of applications:
- data visualization and unsupervised pretraining

Up to now, in order to force the autoencoder to learn interesting features, we have limited the size of the coding layer, making it *undercomplete*.

There are actually many other kinds of constraints that can be used, including ones that allow the coding layer to be just as large as the inputs, or even larger, resulting in an *overcomplete autoencoder*.

Denoising Autoencoders

Another way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. In a 2010 paper of Vincent et al.[1], the authors introduced the *stacked denoising autoencoders*. The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in *dropout*:

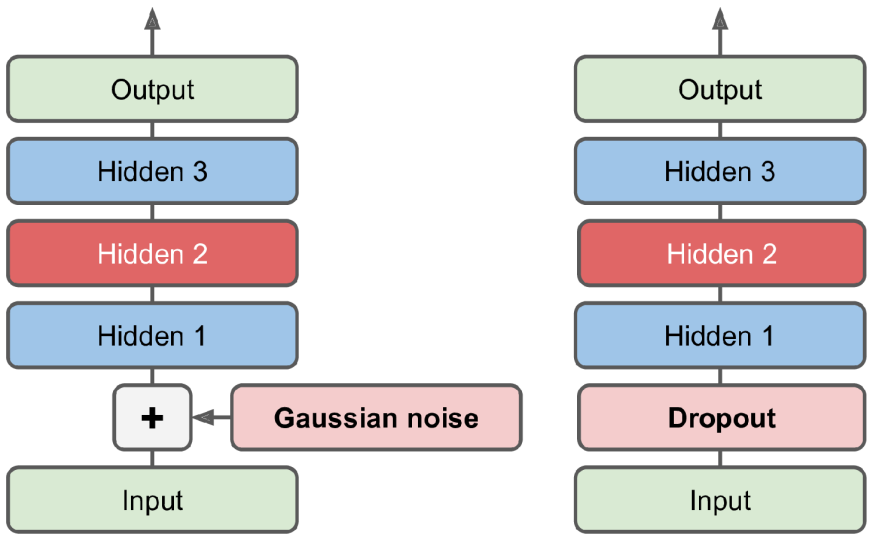


Figure 17-8. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

[1] Pascal Vincent et al., "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," Journal of Machine Learning Research 11 (2010): 3371-3408.

Denoising Autoencoders (cont.)

An Implementation Example

```
dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])
```

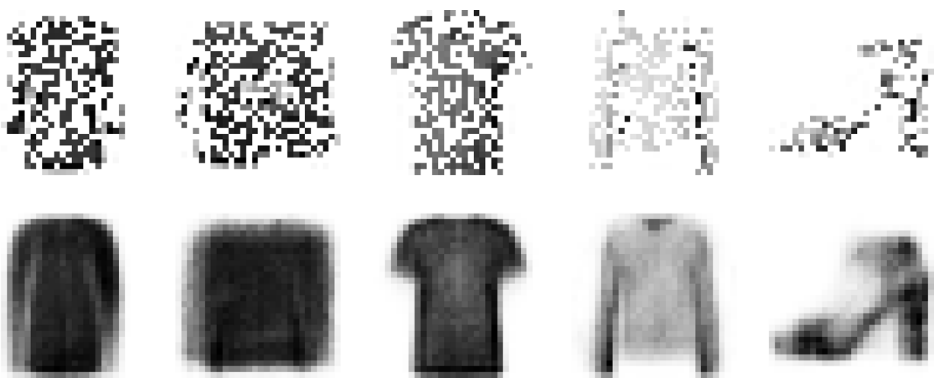


Figure 17-9. Noisy images (top) and their reconstructions (bottom)

Summary

- Understand that autoencoders can be trained and used as an way to efficiently represent data (analogous to PCA).
- Understand the two main parts in an autoencoder: the *encoder* and the *decoder*.
- Know how to implement an *undercomplete* linear autoencoder in TensorFlow to perform PCA on the input data.
- Understand stacked autoencoders and know how to implement one using TensorFlow.
- Understand the concept of tying the weights in a stacked autoencoder and know how to visualize the reconstruction.
- Understand how to use stacked autoencoders in unsupervised pre-training tasks.
- Understand convolutional autoenders and denoising autoencoders.

And that's all for the final week's lecture.
Good luck for the exam!

Have a good week.