# Module 6
# Autoencoders and GANs

# Introduction

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called latent representations or codings, without any supervision (i.e., the training set is unlabeled).

These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction, especially for visualization purposes.

# Introduction

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called latent representations or codings, without any supervision (i.e., the training set is unlabeled).

These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction, especially for visualization purposes.

Autoencoders also act as feature detectors, and they can be used for unsupervised pre-training of deep neural networks.

# Introduction

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called latent representations or codings, without any supervision (i.e., the training set is unlabeled).

These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction, especially for visualization purposes.

Autoencoders also act as 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.

# Introduction

Generative adversarial networks (GANs) are also neural nets capable of generating data. In fact, they can generate pictures of faces so convincing that it is hard to believe the people they represent do not exist.

See https://thispersondoesnotexist.com -> a website that shows faces generated by a GAN architecture called StyleGAN.
Also see https://thisrentaldoesnotexist.com to see some generated Airbnb listings.

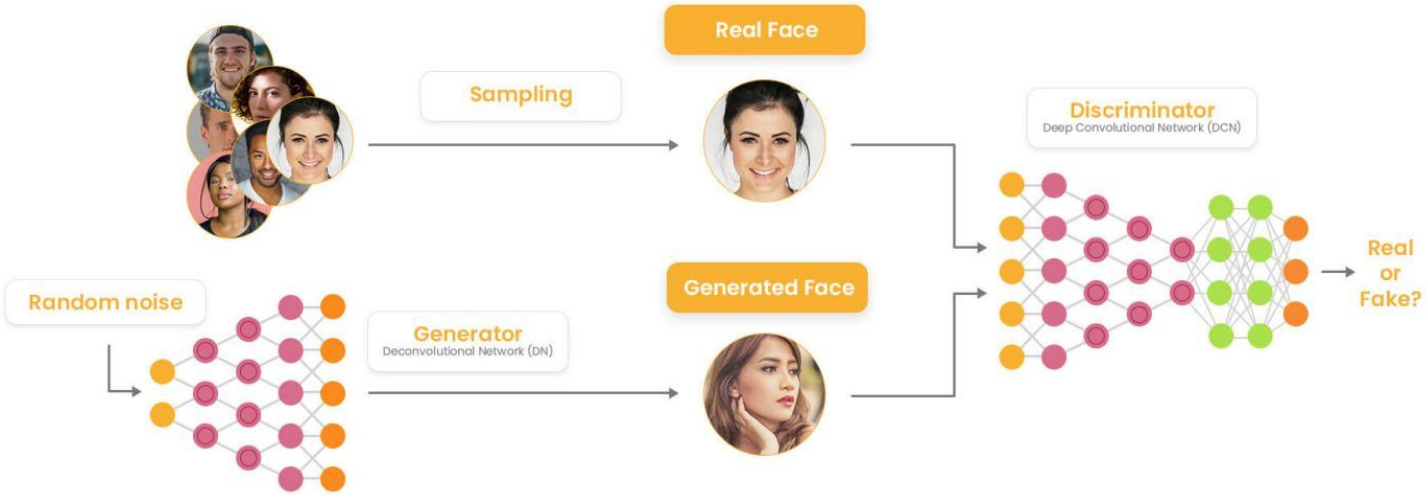
# Introduction

Autoencoders simply learn to copy their inputs to their outputs.

For example, you can limit the size of the latent representations, 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.

# Introduction

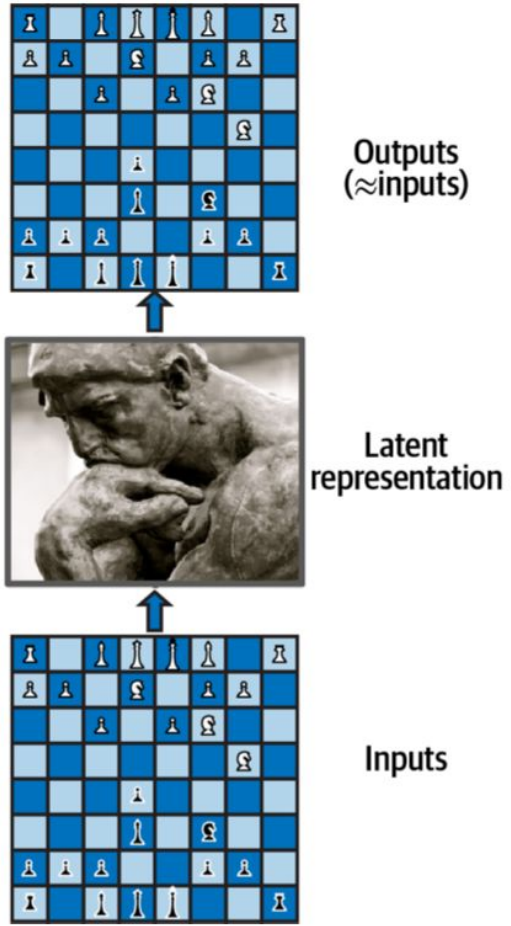
GANs are composed of two neural networks: a *generator* that tries to generate data that looks similar to the training data, and a *discriminator* that tries to tell real data from fake data. The generator and the discriminator compete against each other during training: the generator is often compared to a criminal trying to make realistic counterfeit money, while the discriminator is like the investigator trying to tell real money from fake.



# Efficient Data Representations

An autoencoder typically has the same architecture as a multilayer perceptron (MLP), except that the number of neurons in the output layer must be equal to the number of inputs.

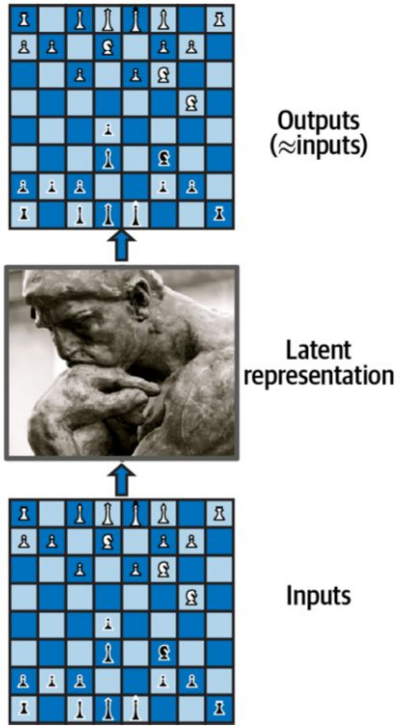
The cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

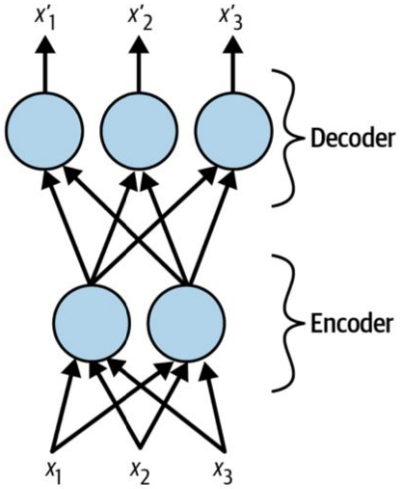


# Efficient Data Representations

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***.

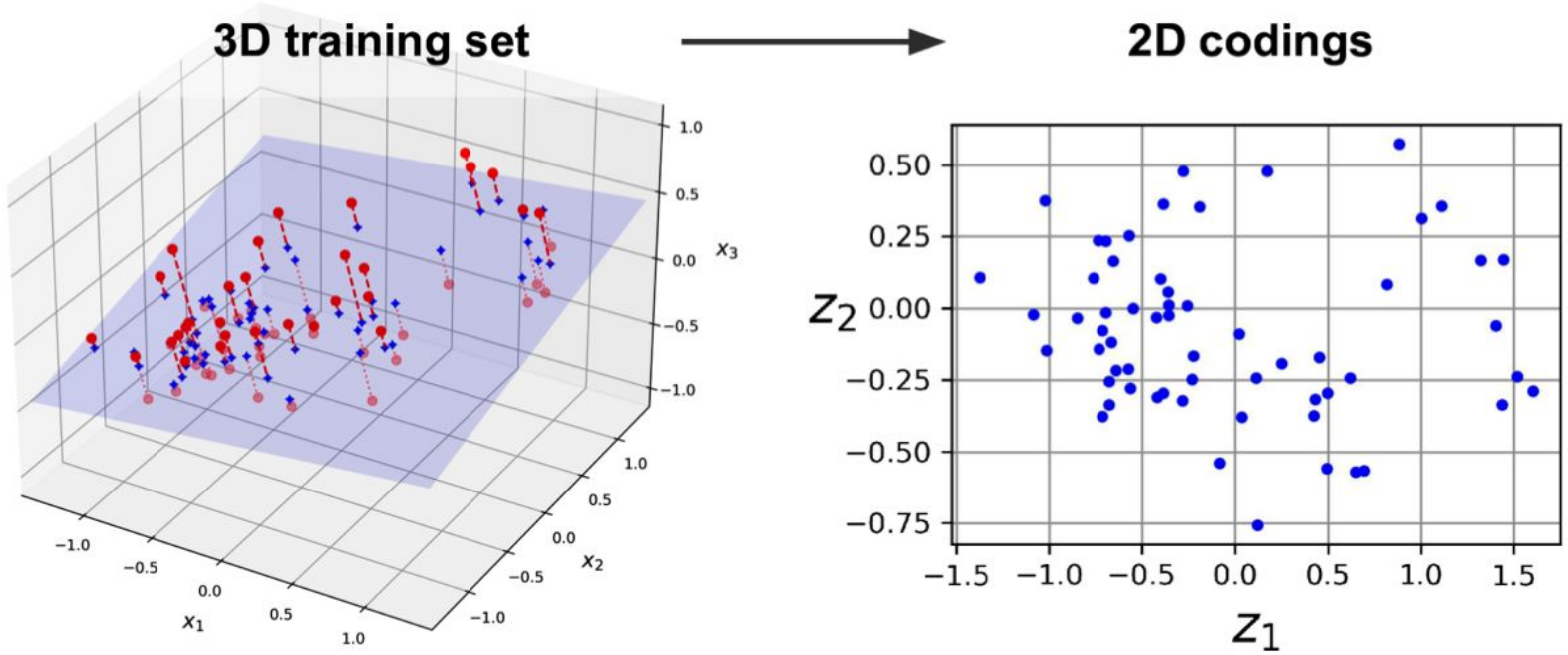
An undercomplete autoencoder cannot trivially copy its inputs to the codings, it must find a way to output a copy of its inputs. It is forced to learn the most important features in the input data (and drop the unimportant ones).



# Performing PCA with an Undercomplete Linear Autoencoder

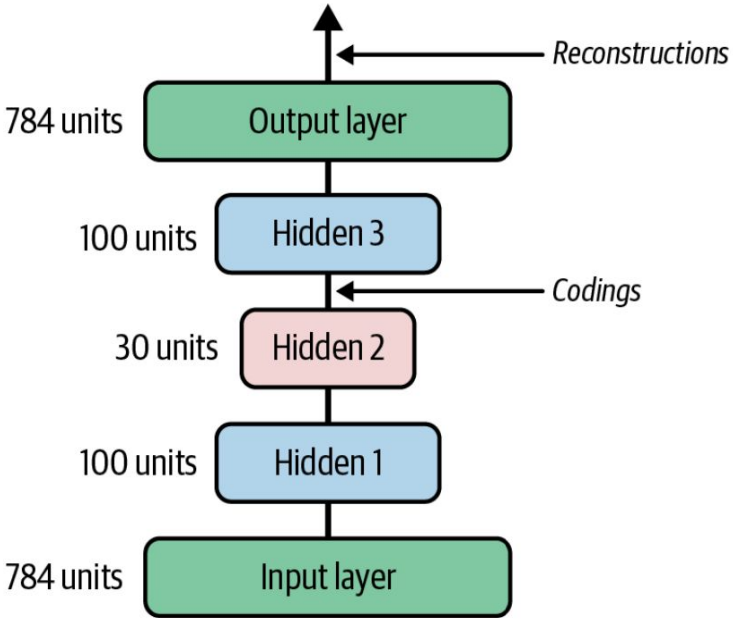
Original 3D dataset (on the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, on the right). As you can see, the autoencoder found the best 2D plane to project the data onto.



# Stacked Autoencoders

Just like other neural networks, autoencoders can have multiple hidden layers => called *stacked autoencoders* (or *deep autoencoders*).

Adding more layers helps the autoencoder learn more complex codings. That said, 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.

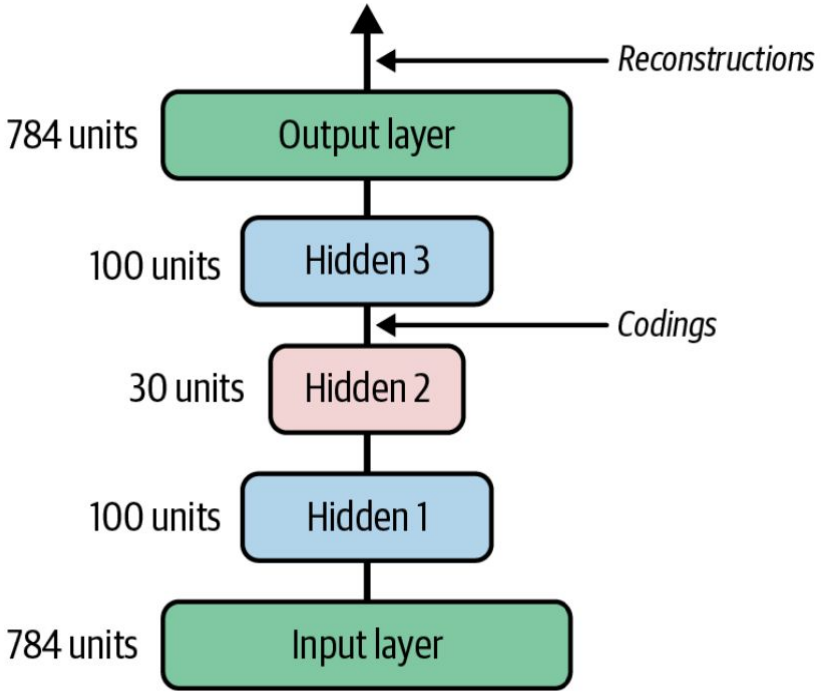


# Stacked Autoencoders

The architecture of a stacked autoencoder is typically symmetrical with regard to the central hidden layer (the coding layer).

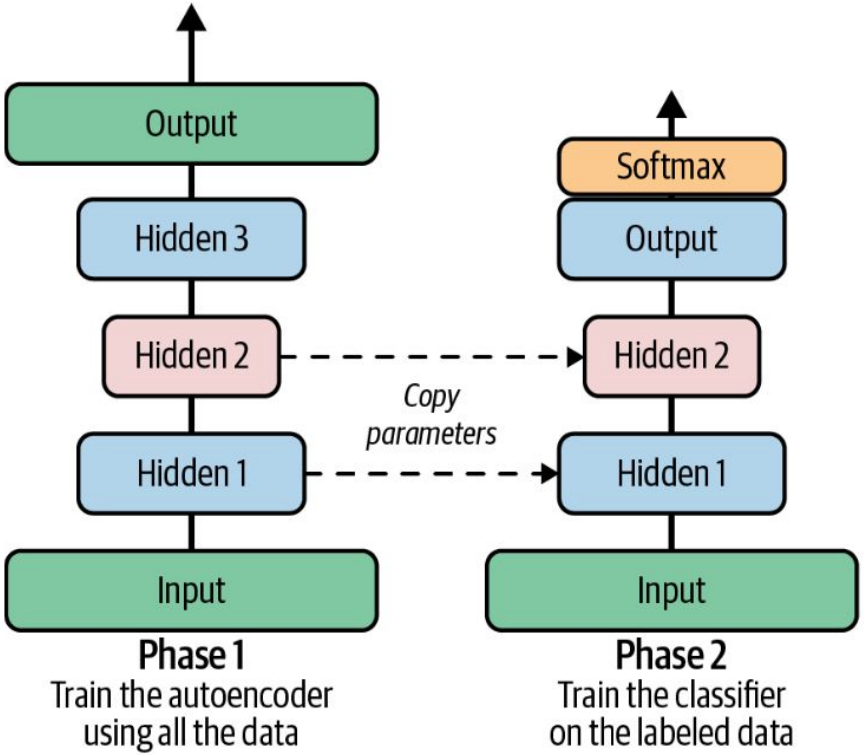
It looks like a sandwich.

For example, an autoencoder for Fashion MNIST may have 784 inputs, followed by a hidden layer with 100 neurons, then a central hidden layer of 30 neurons, then another hidden layer with 100 neurons, and an output layer with 784 neurons.



# Unsupervised Pretraining Using Stacked Autoencoders

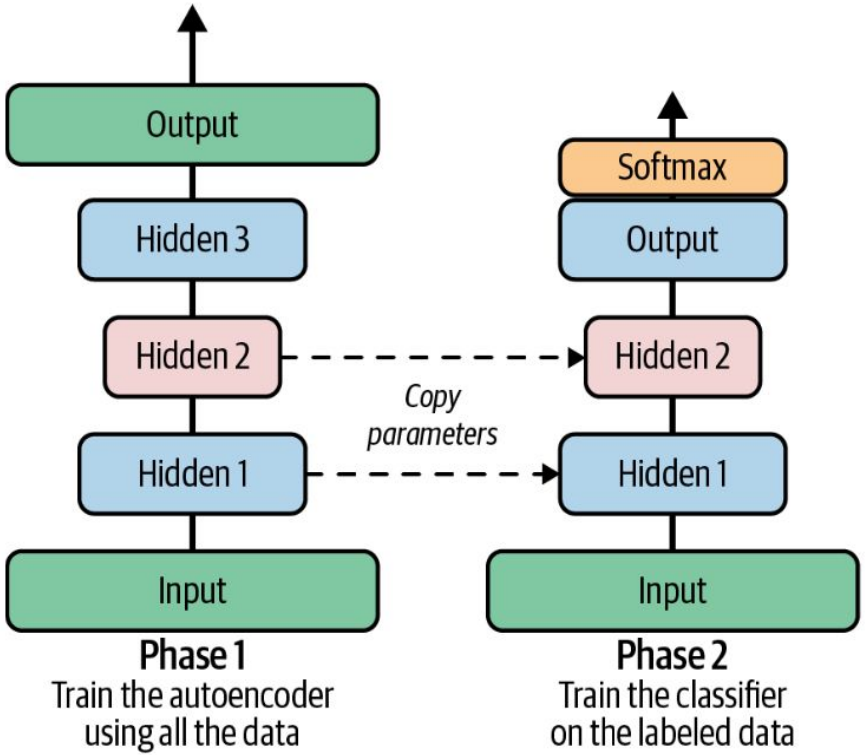
If you are tackling a complex supervised task but you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. This makes it possible to train a high-performance model using little training data because your neural network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



# Unsupervised Pretraining Using Stacked Autoencoders

If you have a large dataset but most of it is unlabeled, you can first train a stacked autoencoder using all the data, then reuse the lower layers to create a neural network for your actual task and train it using the labeled data.

Figure shows how to use a stacked autoencoder to perform unsupervised pre-training for a classification neural network. When training the classifier, if you really don't have much labeled training data, you may want to freeze the pretrained layers (at least the lower ones).



# Unsupervised Pretraining Using Stacked Autoencoders

Having plenty of unlabeled data and little labeled data is common. Building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet), but labeling those images can usually be done reliably only by humans. Labeling instances is time-consuming and costly, so it’s normal to have only a few thousand human-labeled instances, or even less.

Autoencoders are not limited to dense networks: you can also build convolutional autoencoders.

# Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well (unless the images are very small)

Convolutional neural networks are far better suited than dense networks to working with images.

So if you want to build an autoencoder for images (e.g., for unsupervised pre-training or dimensionality reduction), you will need to build a *convolutional autoencoder*.

# Convolutional Autoencoders

The encoder is 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., the number of feature maps).

The decoder must do the reverse (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

It’s also possible to create autoencoders with other architecture types, such as RNNs

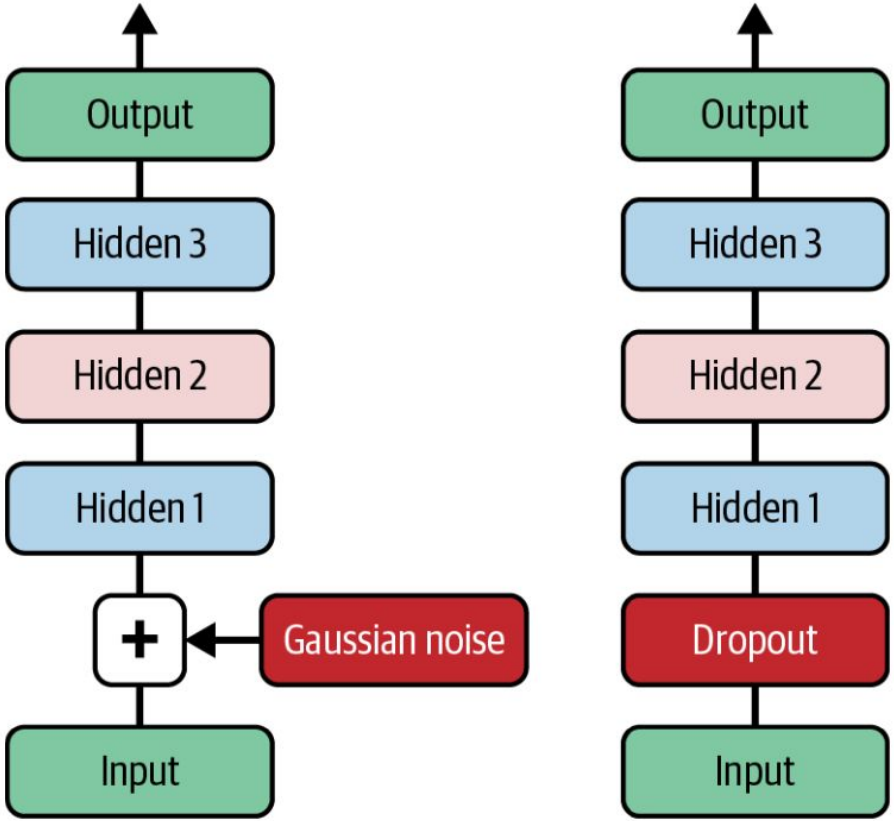
So far we have looked at various kinds of autoencoders (basic, stacked, and convolutional), and how to train them. We also looked at a couple of applications: data visualization and unsupervised pre-training.

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.

The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout. Figure shows both options.



# Generative Adversarial Networks

Generative adversarial networks were proposed by Ian Goodfellow et al., and although the idea got researchers excited almost instantly, it took a few years to overcome some of the difficulties of training GANs.

Idea: Make neural networks compete against each other in the hope that this competition will push them to excel.

# Generative Adversarial Networks

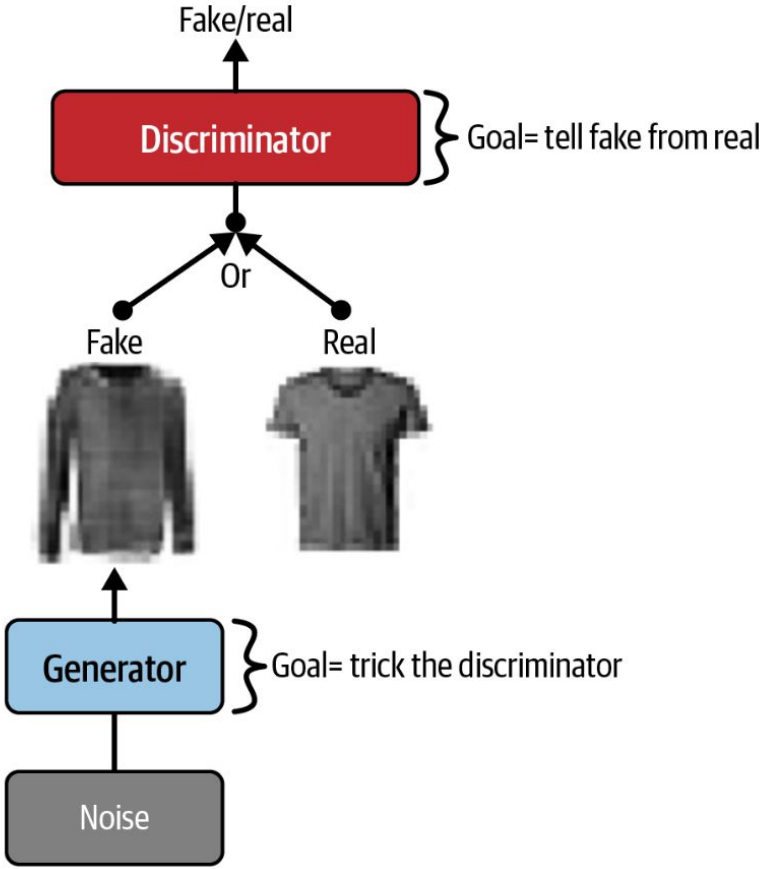
GAN is composed of two neural networks:

**Generator**

Takes a random distribution as input (typically Gaussian Noise) and outputs some data - typically, an image.

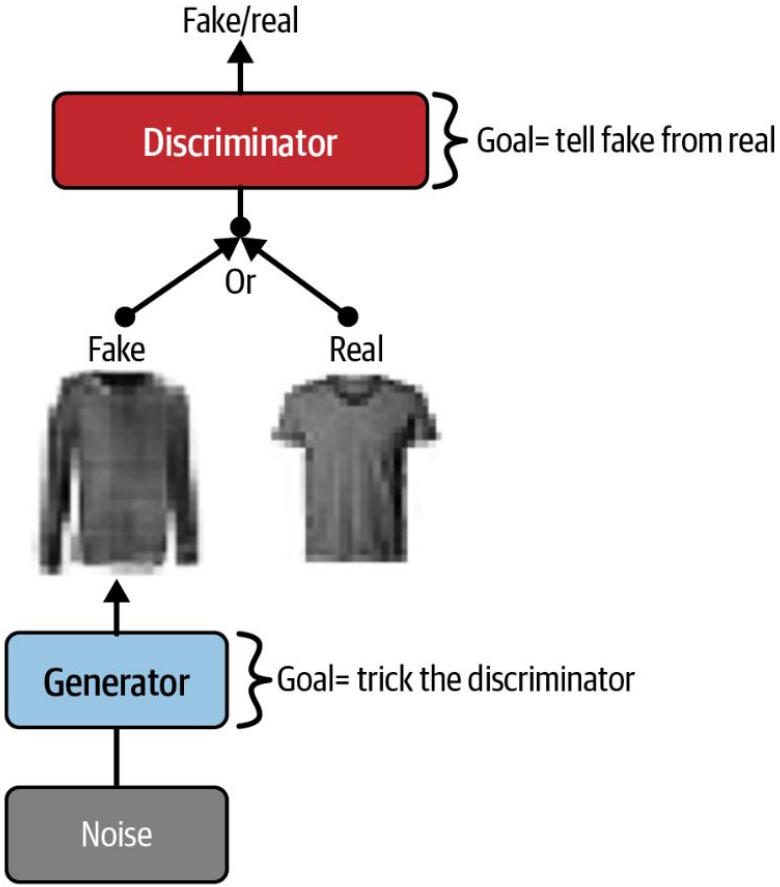
**Discriminator**

Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real.



# Generative Adversarial Networks

During training, the generator and the discriminator have opposite goals: the discriminator tries to tell fake images from real images, while the generator tries to produce images that look real enough to trick the discriminator.



# The Difficulties of Training GANs

The biggest difficulty is called ***mode collapse***: this is when the generator's outputs gradually become less diverse.

**How can this happen?**

Suppose that the generator gets better at producing convincing shoes than any other class. It will fool the discriminator a bit more with shoes, and this will encourage it to produce even more images of shoes. Gradually, it will forget how to produce anything else. Meanwhile, the only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes. Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to move to another class. It may then become good at shirts, forgetting about shoes, and the discriminator will follow. The GAN may gradually cycle across a few classes, never really becoming very good at any of them.

# The Difficulties of Training GANs

Moreover, because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable.

Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities.

GANs are very sensitive to the hyperparameters: you may have to spend a lot of effort fine-tuning them.

# The Difficulties of Training GANs

This is still a very active field of research, and the dynamics of GANs are still not perfectly understood.

But the good news is that great progress has been made, and some of the results are truly astounding!

So let’s look at some of the most successful architectures, deep convolutional GANs.

# Deep Convolutional GANs

The authors of the original GAN paper experimented with convolutional layers, but only tried to generate small images. Soon after, many researchers tried to build GANs based on deeper convolutional nets for larger images.

This proved to be tricky, as training was very unstable, but Alec Radford et al. finally succeeded in late 2015, after experimenting with many different architectures and hyperparameters.

They called their architecture **deep convolutional GANs (DCGANs)**.

# Deep Convolutional GANs

Guidelines they proposed for building stable convolutional GANs:

- Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
- Use batch normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator input layer.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except the output layer, which should use tanh
- Use leaky ReLU activation in the discriminator for all layers.

  These guidelines will work in many cases, but not always, so you may still need to experiment with different hyperparameters. In fact, just changing the random seed and training the exact same model again will sometimes work.

# Deep Convolutional GANs

DCGANs aren’t perfect, though. For example, when you try to generate very large images using DCGANs, you often end up with locally convincing features but overall inconsistencies, such as shirts with one sleeve much longer than the other, different earrings, or eyes looking in opposite directions.

# Deep Convolutional GANs