CSCI 6521

Advanced Machine Learning I

Chapter 03: Representation Learning and Generative Learning Using Autoencoders and GANs

Md Tamjidul Hoque

Autoencoders and GANs

- 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 pretraining 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 and GANs

- In contrast, faces generated by generative adversarial networks (GANs) are now so convincing that it is hard to believe that the people they represent do not exist.

- We can judge so for ourselves by visiting https://thispersondoesnotexist.com/, a website that shows faces generated by a recent GAN architecture called ***StyleGAN***.

- We can also check out https://thisrentaldoesnotexist.com/ to see some generated Airbnb bedrooms).

… Autoencoders and GANs

- GANs are now widely used for
  - super resolution (increasing the resolution of an image),
  - Colorization (or see deOldify),
  - powerful image editing (e.g., replacing photobombers with realistic background using befunky or MakeiT),
  - turning a simple sketch into a photorealistic image, or check gauGAN,
  - predicting the next frames in a video,
  - augmenting a dataset (to train other models),
  - generating other types of data (such as text, audio, and time series),
  - identifying the weaknesses in other models and strengthening them,
- and more.

… Autoencoders and GANs

Autoencoders and GANs are both unsupervised,

they both learn dense representations, they can both be used as generative models, and

they have many similar applications.

However, they work very differently:

- ❑ **Autoencoders** simply learn 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.
- ➢ For example, we can limit the size of the latent representations, or we can add noise to the inputs and train the network to recover the original inputs.

… Autoencoders and GANs

- 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 learning the identity function under some constraints.

- **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.
- This architecture is very original in Deep Learning in that 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 -

… Autoencoders and GANs

- the discriminator is like the police investigator trying to tell real money from fake.
- Adversarial training (training competing neural networks) is widely considered as one of the most important ideas in recent years.

Chapter Overview

- We will start by exploring in more depth
  - how autoencoders work and
  - how to use them for dimensionality reduction,
  - feature extraction,
  - unsupervised pretraining, or
  - as generative models.
- This will naturally lead us to GANs.
  - We will start by building a simple GAN to generate fake images, but
  - we will see that training is often quite difficult.
  - We will discuss the main difficulties we will encounter with adversarial training, as well as
  - some of the main techniques to work around these difficulties.

Autoencoder

- An autoencoder looks at the inputs,
- converts them to an efficient latent representation, and
- then spits out something that (hopefully) looks very close to the inputs.
- An autoencoder is always composed of two parts:
  - an encoder (or recognition network) that converts the inputs to a latent representation,
  - followed by a decoder (or generative network) that converts the internal representation to the outputs (see Figure 1, (next slide)).

… Autoencoder

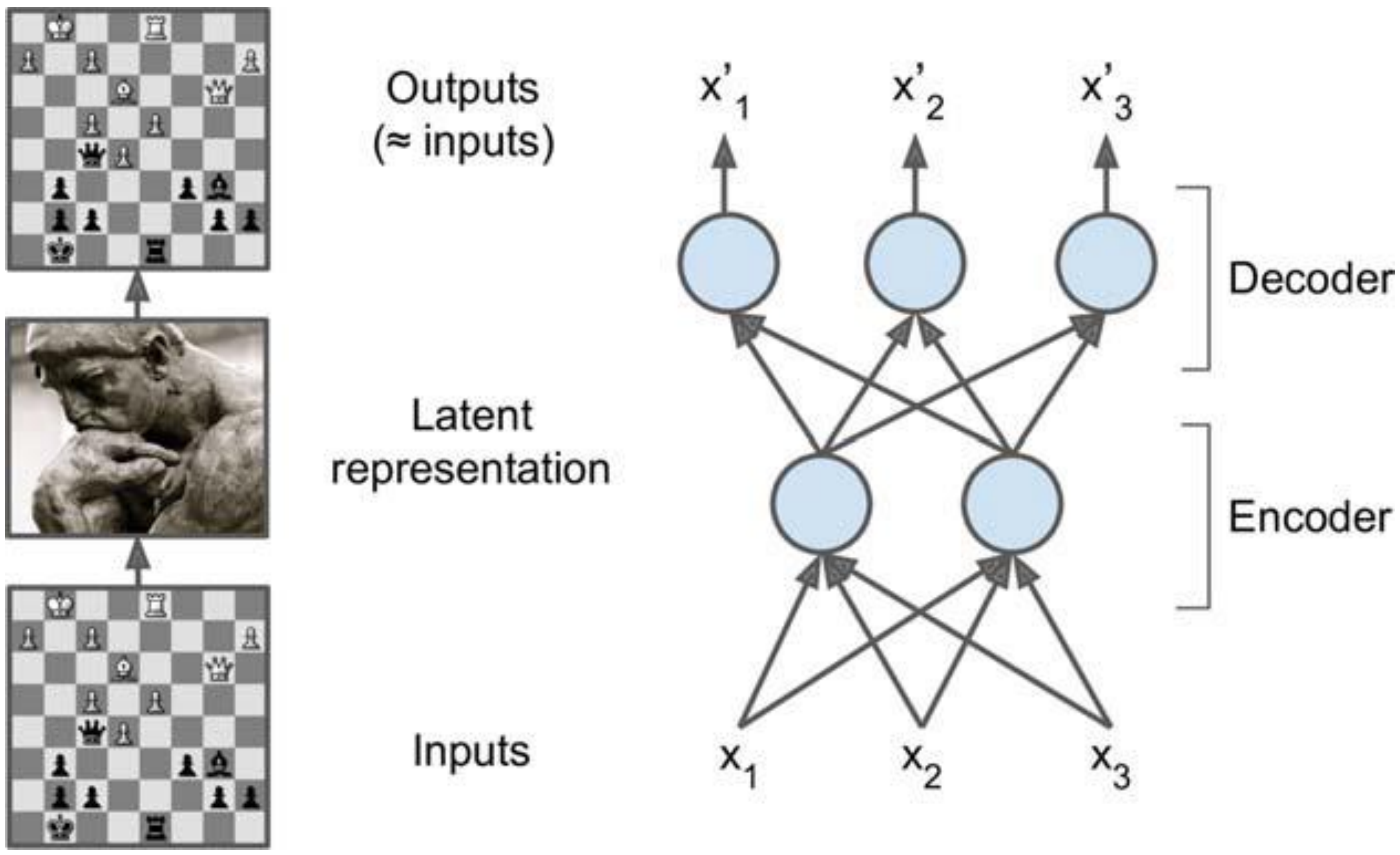


Figure 1: The chess memory experiment (left) and a simple autoencoder (right).

… Autoencoder

- As we can see, an autoencoder typically has the same architecture as a Multi-Layer Perceptron (MLP),
- except that the number of neurons in the output layer must be equal to the number of inputs.
- In this example, there is just one hidden layer composed of two neurons (the encoder), and one output layer composed of three neurons (the decoder).
- The outputs are often called the *reconstructions* because 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.

… 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*.
- An *undercomplete* autoencoder cannot trivially copy its inputs to the codings,
- yet it must find a way to output a copy of its inputs.
- Thus, 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

- If the autoencoder uses only linear activations and
- the cost function is the mean squared error (MSE),
- then it ends up performing *Principal Component Analysis* (PCA).
- See exercise: “PCA with a linear Autoencoder”

Stacked Autoencoders

- 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.
- 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).
- To put it simply, it looks like an hourglass -
- For example, an autoencoder for 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.
- This stacked autoencoder is represented in Figure 3 (next slide).

… Stacked Autoencoders

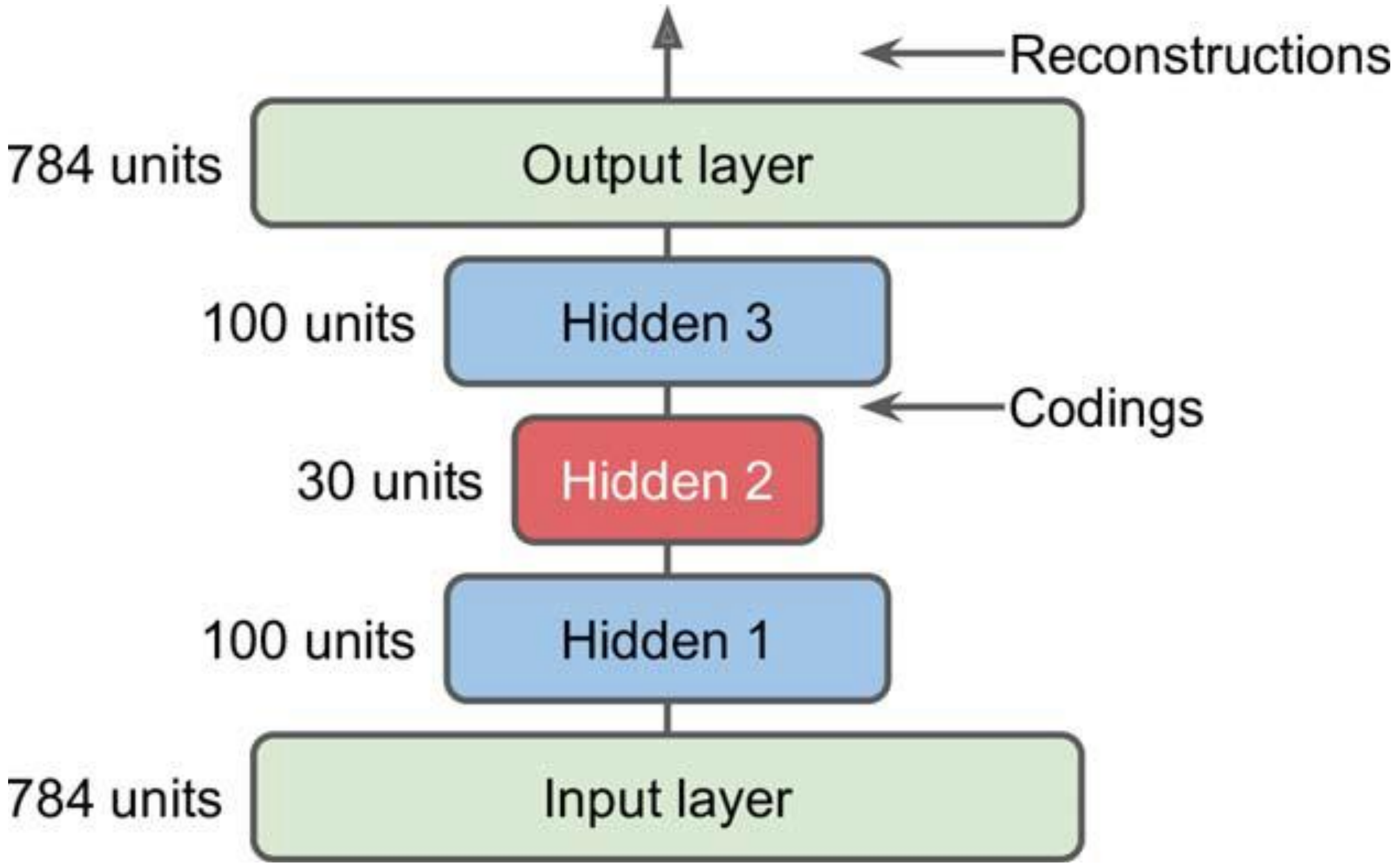


Figure 3: Stacked autoencoder.

- See exercise: Implementing a Stacked Autoencoder Using Keras.

Unsupervised Pretraining Using Stacked Autoencoders

- If we are tackling a complex supervised task but we 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.
- Similarly, 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.

… Unsupervised Pretraining Using Stacked Autoencoders

- For example, Figure 6 shows how to use a stacked autoencoder to perform unsupervised pretraining for a classification neural network.

- When training the classifier, if we really don't have much labeled training data, we may want to freeze the pretrained layers (at least the lower ones).

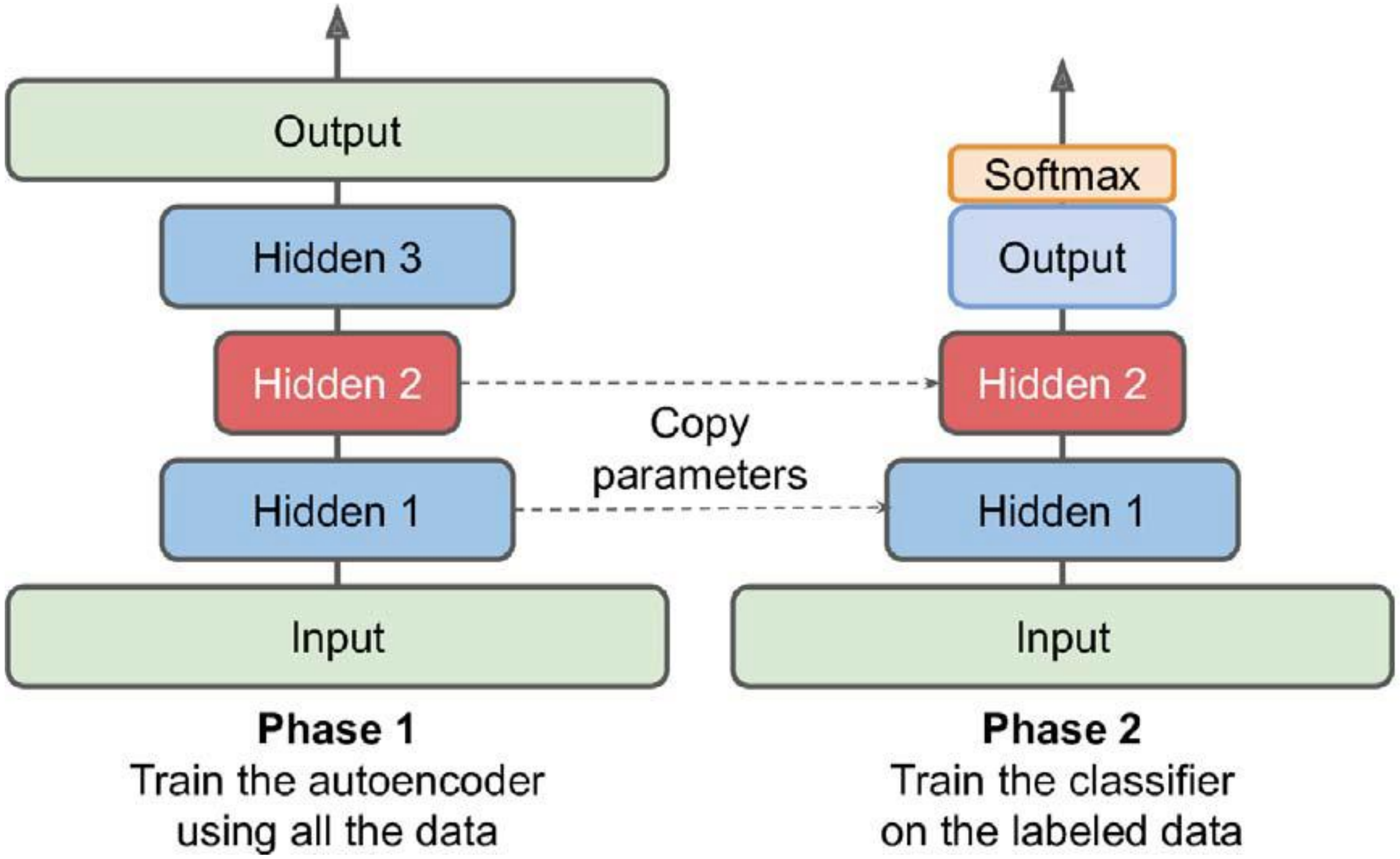


Figure 6: Unsupervised pretraining using autoencoders.

… Unsupervised Pretraining Using Stacked 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.
- 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 (e.g., classifying them as cute or not) 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.

Tying Weights

- When an autoencoder is neatly symmetrical (like the one in the exercise), a common technique is to tie the weights of the decoder layers to the weights of the encoder layers (Excellent Idea!!!).
- This halves the number of weights in the model,
- speeding up training and
- limiting the risk of overfitting.
- Specifically, if the autoencoder has a total of *N* layers (not counting the input layer), and
- $\mathbf{W}_L$ represents the connection weights of the L^{th} layer (e.g., layer 1 is the first hidden layer, layer N/2 is the coding layer, and layer N is the output layer),
- then the decoder layer weights can be defined simply as: $\mathbf{W}_{N-L+1} = \mathbf{W}_L^T$ (with L = 1, 2, …, N/2).
- **See exercise**: Tying Weights.

Training One Autoencoder at a Time

- Rather than training the whole stacked autoencoder in one go like we just did, 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 in Figure 7.

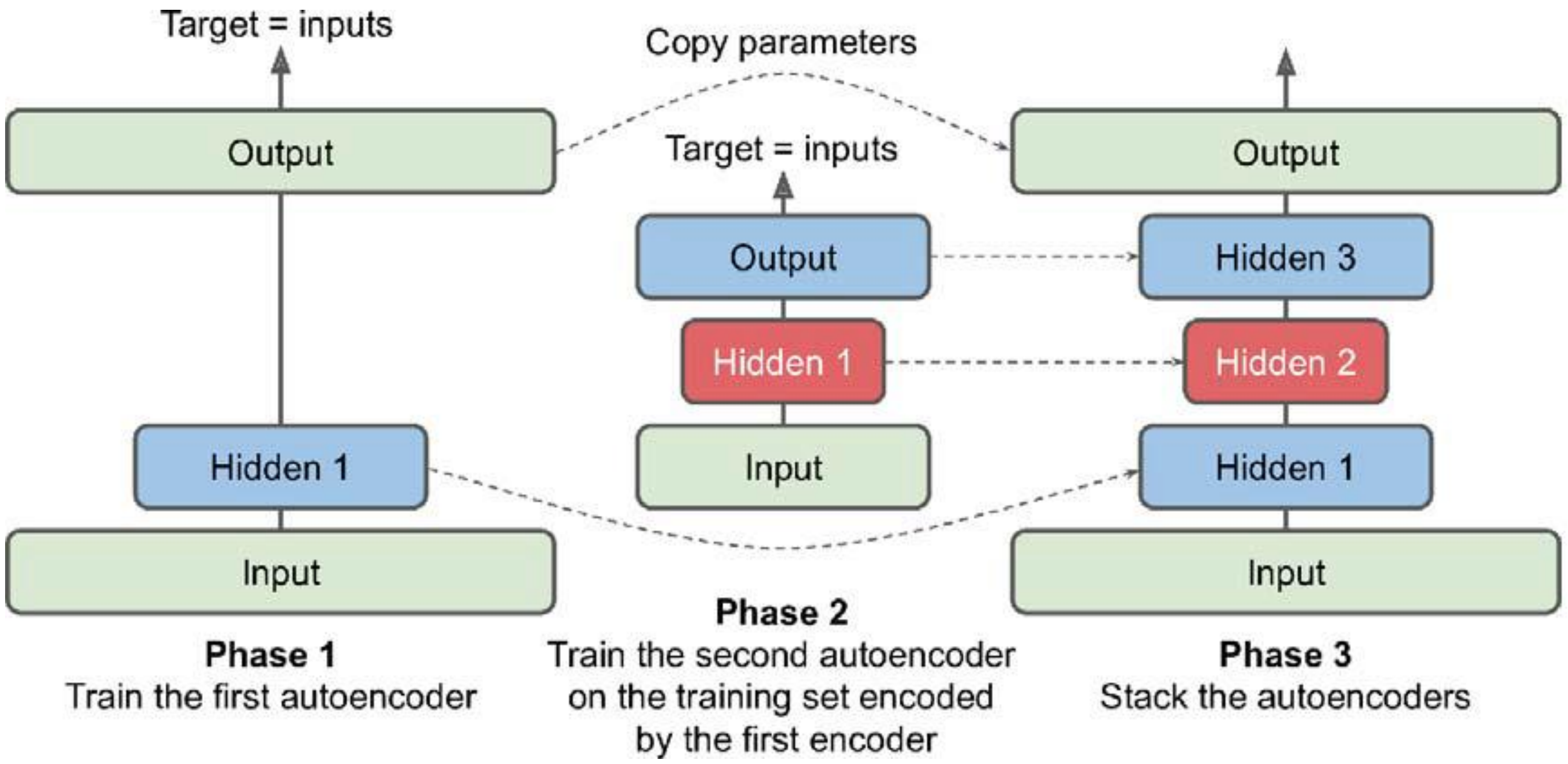


Figure 7: Training one autoencoder at a time.

… Training One Autoencoder at a Time

- This technique is not used as much these days, but we may still run into papers that talk about “greedy layer wise training,” so it’s good to know what it means.
- During the first phase of training, the first autoencoder learns to reconstruct the inputs.
- Then we encode the whole training set using this first autoencoder, and this gives us a new (compressed) training set.
- We then train a second autoencoder on this new dataset.
- This is the second phase of training.
- Finally, we build a big sandwich (or hourglass) using all these autoencoders, as shown in Figure 7,
- i.e., we first stack the hidden layers of each autoencoder, then the output layers in reverse order.

… Training One Autoencoder at a Time

- This gives us the final stacked autoencoder
  - **see exercise**: “Training One Autoencoder at a Time”.
- We could easily train more autoencoders this way, building a very deep stacked autoencoder.

Convolutional Autoencoders

- Autoencoders are not limited to dense networks:
  - we can also build convolutional autoencoders, or even recurrent autoencoders.

- If we 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 work with images.
  - So, if we want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), we will need to build a convolutional autoencoder (see paper #1).

- 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).

… Convolutional Autoencoders

- The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and
- for this we can use transpose convolutional layers (alternatively, we could combine upsampling layers with convolutional layers).
- **See exercise**: Using Convolutional Layers Instead of Dense Layers

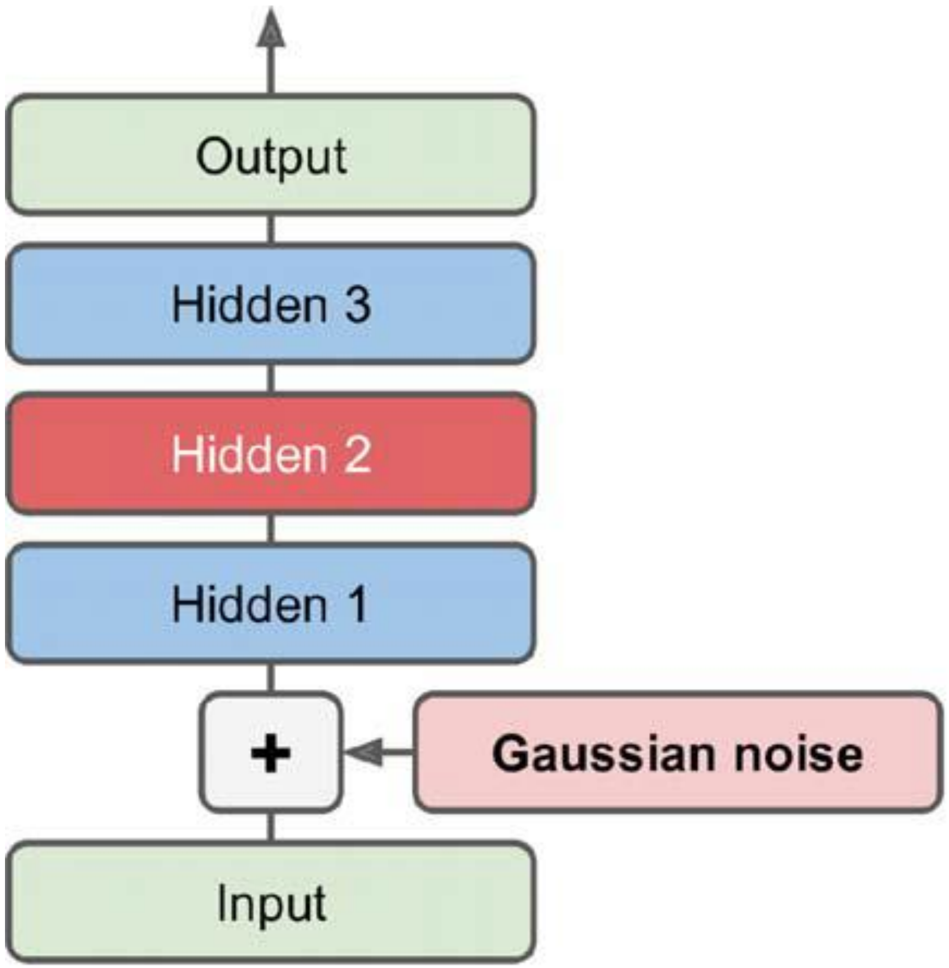
Recurrent Autoencoders

- If we want to build an autoencoder for sequences,
- such as time series or text (e.g., for unsupervised learning or dimensionality reduction),
- then recurrent neural networks (RNN) may be better suited than dense networks.
- Building a recurrent autoencoder is straightforward:
  - the encoder is typically a sequence-to-vector RNN which compresses the input sequence down to a single vector.
  - The decoder is a vector-to-sequence RNN that does the reverse.
- **See exercise**: Recurrent Autoencoders

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.
- This idea has been around since the 1980s (e.g., it is mentioned in Yann LeCun's 1987 master's thesis).
- Autoencoders could also be used for feature extraction (see paper #1).
- Stacked denoising autoencoder was introduced in paper #2.
- The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout.
  - Figure 8 (see next slide) shows both options.

… Denoising Autoencoders



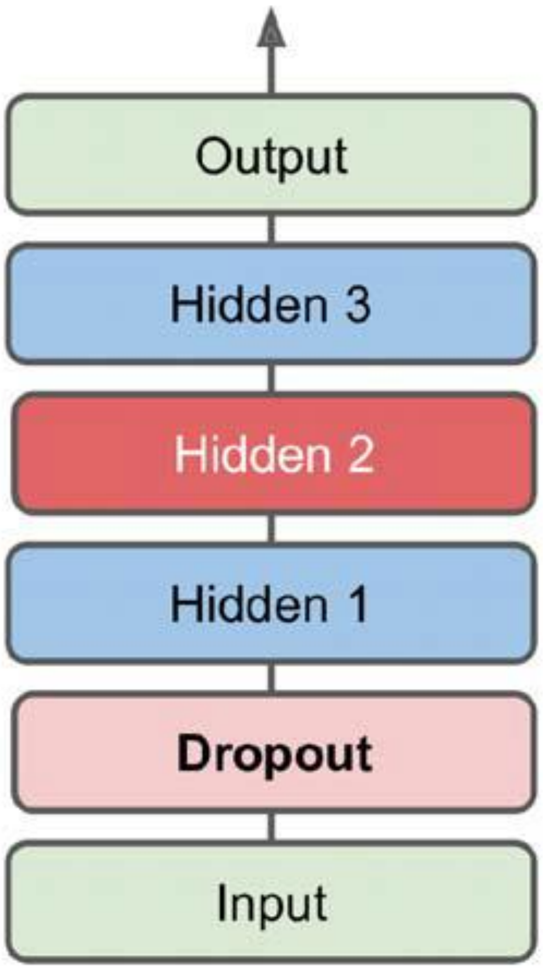


Figure 8: Denoising autoencoders, with Gaussian noise (left) or dropout (right)

… Denoising Autoencoders

- The implementation is straightforward:
  - it is a regular stacked autoencoder with an additional Dropout layer applied to the encoder’s inputs or
  - we could use a GaussianNoise layer instead.
- Recall that the Dropout layer is only active during training and so is the GaussianNoise layer.
- **See exercise**: Stacked denoising Autoencoder.

Sparse Autoencoders

- Another kind of constraint that often leads to good feature extraction is sparsity:
  - by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer.
  - For example, it may be pushed to have on average only 5% significantly active neurons in the coding layer.
  - This forces the autoencoder to represent each input as a combination of a small number of activations.
  - As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to).
- A simple approach is to use the sigmoid activation function in the coding layer (to constrain the codings to values between 0 and 1),
- use a large coding layer (e.g., with 300 units), and
- add some ℓ1 regularization to the coding layer's activations (the decoder is just a regular decoder). [**See exercise**]

Variational Autoencoder

- Variational autoencoders, introduced in 2013 (see paper #1), are quite different from all the autoencoders:
  - They are probabilistic autoencoders, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
  - Most importantly, they are generative autoencoders, meaning that they can generate new instances that look like they were sampled from the training set.
- they are easier to train, and the sampling process is much faster.
- Variational autoencoders perform variational Bayesian inference, which is an efficient way to perform approximate Bayesian inference.
- Figure 12 (left) shows a variational autoencoder (see next slide).

… Variational Autoencoder

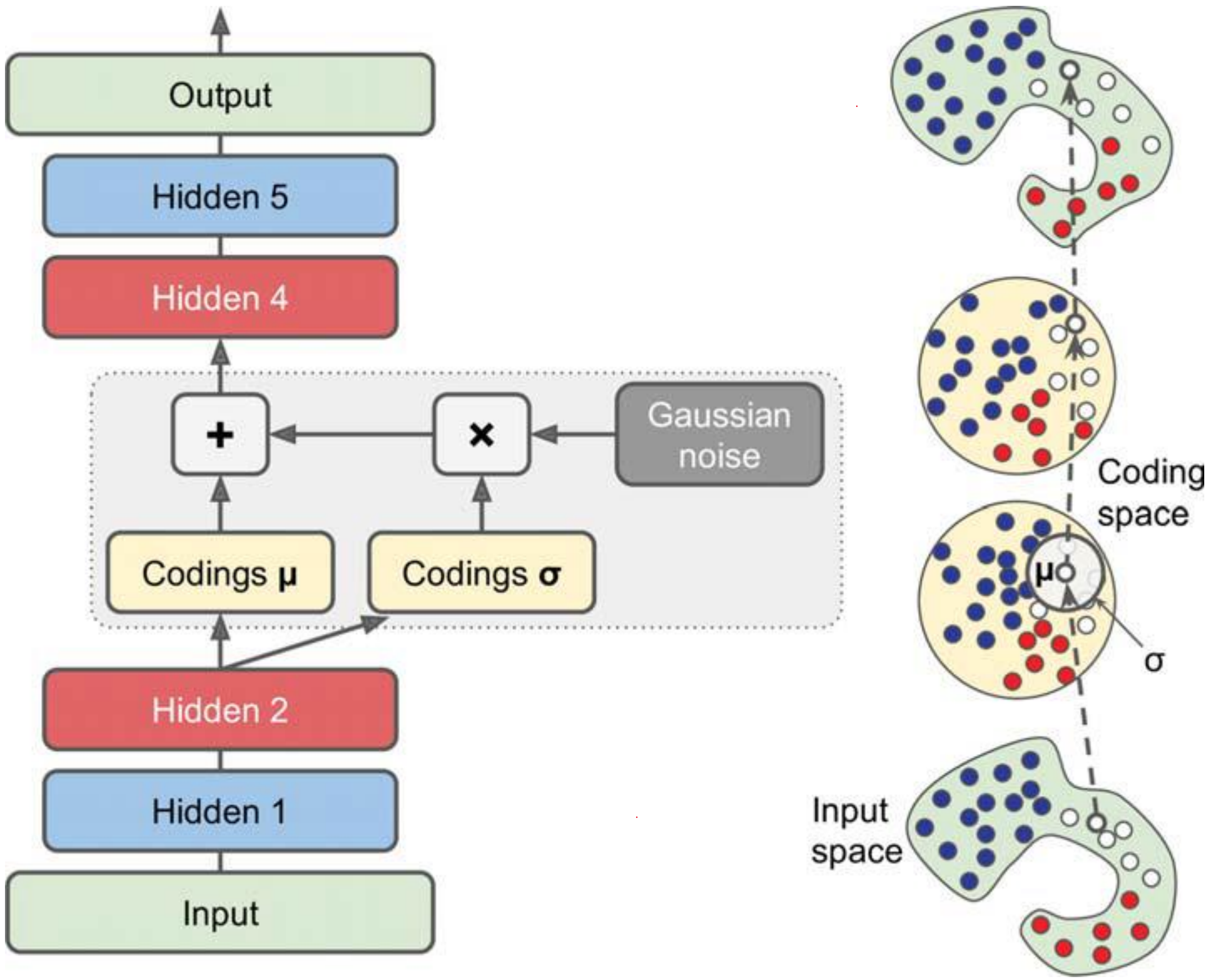


Figure 12: Variational autoencoder (left) and an instance going through it (right).

… Variational Autoencoder

- We can recognize the basic structure of all autoencoders, with an encoder followed by a decoder (in Figure 12, they both have two hidden layers), but there is a twist:
  - instead of directly producing a coding for a given input, the encoder produces a mean coding **μ** and a standard deviation **σ**.
  - The actual coding is then sampled randomly from a Gaussian distribution with mean **μ** and standard deviation **σ**.
  - After that the decoder decodes the sampled coding normally.
- The right part of the diagram (Figure 12) shows a training instance going through this autoencoder.
  - First, the encoder produces **μ** and **σ**, then a coding is sampled randomly (notice that it is not exactly located at **μ**), and
  - finally, this coding is decoded;
  - the final output resembles the training instance.

… Variational Autoencoder

- Although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution†:
  - during training, the cost function pushes the codings to gradually migrate within the coding space (also called the ***latent space***) to end up looking like a cloud of Gaussian points.
  - One great consequence is that after training a variational autoencoder, we can very easily generate a new instance:
    - just sample a random coding from the Gaussian distribution, decode it.
- The **cost function** is composed of **two parts**:
  - The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs (we can use cross entropy for this).

… Variational Autoencoder

- The second is the latent loss that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution:
  - it is the KL divergence between the target distribution (i.e., the Gaussian distribution) and the actual distribution of the codings.
  - The latent loss can be computed simply using **Equation 3** (Variational autoencoder's latent loss) †:

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \log\left(\sigma_i^{\,2}\right) - \sigma_i^{\,2} - \mu_i^{\,2}$$

  - In this equation, $\mathcal{L}$ is the latent loss, *K* is the codings' dimensionality, and μ_i and σ_i are the mean and standard deviation of the i*th* component of the codings.
  - The vectors **μ** and **σ** (which contain all the μ_i and σ_i) are output by the encoder, as shown in Figure 12 (left).

… Variational Autoencoder

- A common tweak to the variational autoencoder's architecture is to make the encoder output **γ** = log(**σ**^{2}) rather than **σ**.
- The latent loss can then be computed as shown in Equation 4. This approach is more numerically stable and speeds up training.
- **Equation 4**: Variational autoencoder's latent loss, rewritten using **γ** = log(**σ**^{2}):

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \gamma_i - \exp\left(\gamma_i\right) - {\mu_i}^2$$

- **See exercise:** Variational Autoencoder.

Generative Adversarial Network (GAN)

- Generative adversarial networks were proposed in 2014 (paper #1).
- The idea was to make neural networks compete against each other in the hope that this competition will push them to excel.
- As shown in Figure 15, a GAN is composed of two neural networks: (1) Generator and (2) Discriminator.

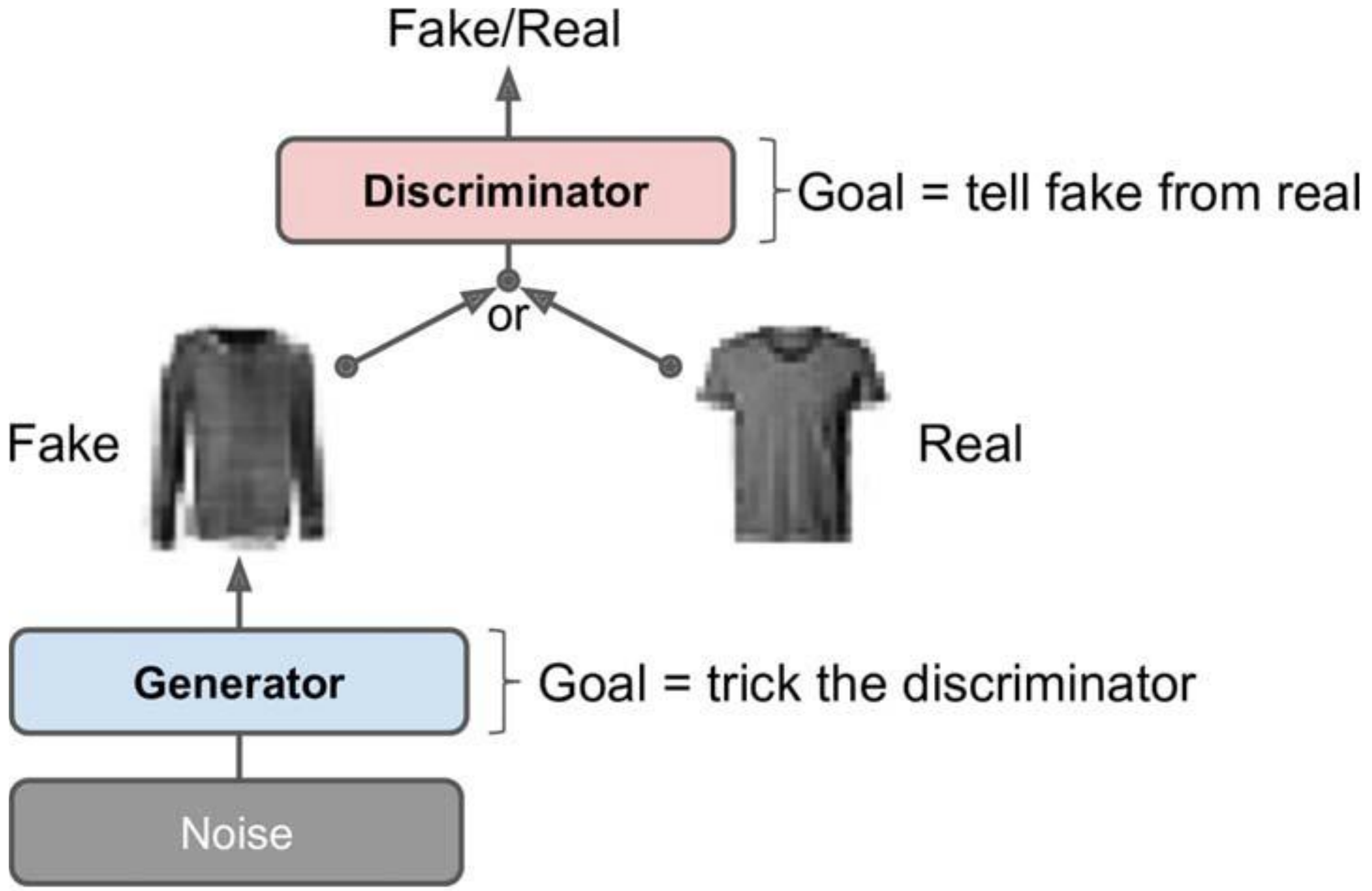


Figure 15: A generative adversarial network.

… Generative Adversarial Network (GAN)

Generator:

- Takes a random distribution as input (typically Gaussian) and outputs some data typically, an image.
- We can think of the random inputs as the latent representations (i.e., codings) of the image to be generated.
- So, as we can see, the generator offers the same functionality as a decoder in a variational autoencoder, and
- it can be used in the same way to generate new images (just feed it some Gaussian noise, and it outputs a brand-new image).
- However, it is trained very differently.

… Generative Adversarial Network (GAN)

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 Network (GAN)

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.
- Because the GAN is composed of two networks with different objectives, it cannot be trained like a regular neural network.
- Each training iteration is divided into two phases:
- In the first phase, we train the discriminator:
  - A batch of real images is sampled from the training set and
  - is completed with an equal number of fake images produced by the generator.

… Generative Adversarial Network (GAN)

- The labels are set to 0 for fake images and 1 for real images, and
- the discriminator is trained on this labeled batch for one step,
- using the binary cross-entropy loss.
- Importantly, backpropagation only optimizes the weights of the discriminator during this phase.

- In the second phase, we train the generator:
  - We first use it to produce another batch of fake images, and
  - once again the discriminator is used to tell whether the images are fake or real.
  - This time we do not add real images in the batch, and all the labels are set to 1 (real) –

… Generative Adversarial Network (GAN)

- in other words, we want the generator to produce images that the discriminator will (wrongly) believe to be real!
- Crucially, the weights of the discriminator are frozen during this step, so backpropagation only affects the weights of the generator.

- The generator never actually sees any real images, yet it gradually learns to produce convincing fake images!
- All it gets is the gradients flowing back through the discriminator.
- Fortunately, the better the discriminator gets, the more information about the real images is contained in these secondhand gradients,
- So, the generator can make significant progress.
- **See exercise**: Generative Adversarial Networks.

The Difficulties of Training GANs

- During training, the generator and the discriminator constantly try to outsmart each other, in a zero-sum game.
- As training advances, the game may end up in a state that game theorists call a *Nash equilibrium*, named after the mathematician John Nash:
  - this is when no player would be better off changing their own strategy, assuming the other players do not change theirs.
- GAN can only reach a single *Nash equilibrium*:
  - that's when the generator produces perfectly realistic images, and the discriminator is forced to guess (50% real, 50% fake).
  - This fact is very encouraging:
    - It would seem that we just need to train the GAN for long enough, and

… The Difficulties of Training GANs

- it will eventually reach this equilibrium, giving us a perfect generator.
- Unfortunately, it’s not that simple:
  - nothing guarantees that the equilibrium will ever be reached.

- 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.

… The Difficulties of Training GANs

- 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.

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

… The Difficulties of Training GANs

- Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities.
- And since many factors affect these complex dynamics, GANs are very sensitive to the hyperparameters:
  - we may have to spend a lot of effort fine-tuning them.

- Some proposed new cost functions (see # 1, 2) or techniques to stabilize training or to avoid the mode collapse issue:
  - a popular technique called *experience replay* consists in storing the images produced by the generator at each iteration in a replay buffer (gradually dropping older generated images) and
    - training the discriminator using real images plus fake images drawn from this buffer (rather than just fake images produced by the current generator).

… The Difficulties of Training GANs

      - This reduces the chances that the discriminator will overfit the latest generator’s outputs.

  - Another common technique is called *mini-batch discrimination*:

  - it measures how similar images are across the batch and provides this statistic to the discriminator, so it can easily reject a whole batch of fake images that lack diversity.

  - This encourages the generator to produce a greater variety of images, reducing the chance of mode collapse.

- GAN is still a very active field of research, having some great progresses, such as using deep convolutional GANs architecture.

Deep Convolutional GANs

- ***Deep convolutional GANs*** (DCGANs) was the first successful GANs with deeper convolutional nets (paper #1),
- with the following main guidelines 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's 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 we may still need to experiment with different hyperparameters.

Progressive Growing of GANs

- An important technique was proposed in 2018 (paper #1):
  - they suggested generating small images at the beginning of training,
  - then gradually adding convolutional layers to both the generator and the discriminator
  - to produce larger and larger images (4 × 4, 8 × 8, 16 × 16, …, 512 × 512, 1,024 × 1,024).
    - This approach resembles greedy layer-wise training of stacked autoencoders.
  - The extra layers get added at the end of the generator
  - and at the beginning of the discriminator, and
  - previously trained layers remain trainable.
- For example, see Figure 19 (in the next slide).

… Progressive Growing of GANs

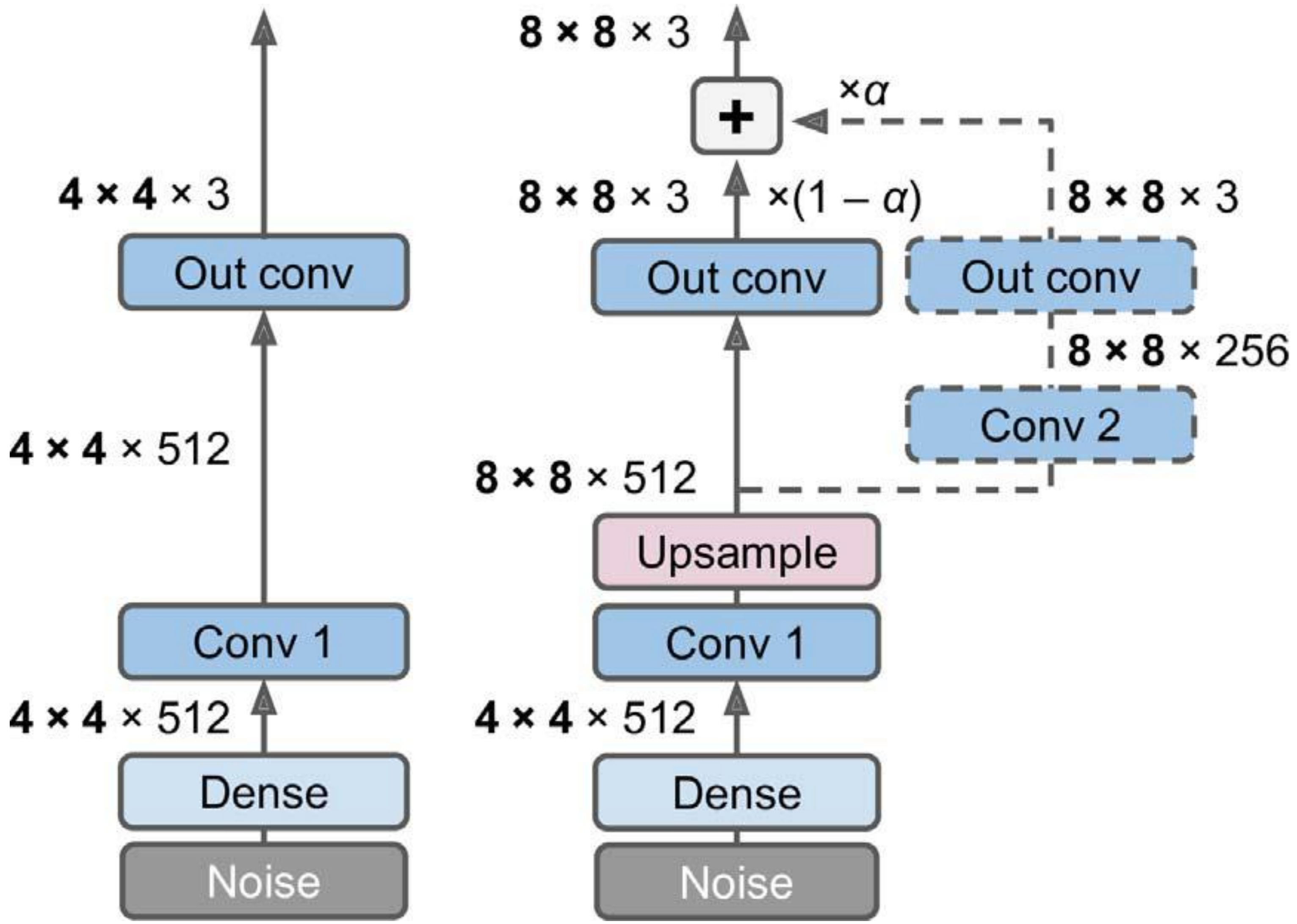


Figure 19: Progressively growing GAN - a GAN generator outputs 4 × 4 color images (left); we extend it to output 8 × 8 images (right).

… Progressive Growing of GANs

- When growing the generator's outputs from 4 × 4 to 8 × 8 (see Figure 19), an upsampling layer (using nearest neighbor filtering) is added to the existing convolutional layer,
- so it outputs 8 × 8 feature maps, which are then fed to the new convolutional layer (which uses "same" padding and strides of 1, so its outputs are also 8 × 8).
- This new layer is followed by a new output convolutional layer:
    - this is a regular convolutional layer with kernel size 1 that projects the outputs down to the desired number of color channels (e.g., 3).
- To avoid breaking the trained weights of the first convolutional layer when the new convolutional layer is added, the final output is a weighted sum of the original output layer (which now outputs 8 × 8 feature maps) and the new output layer.

… Progressive Growing of GANs

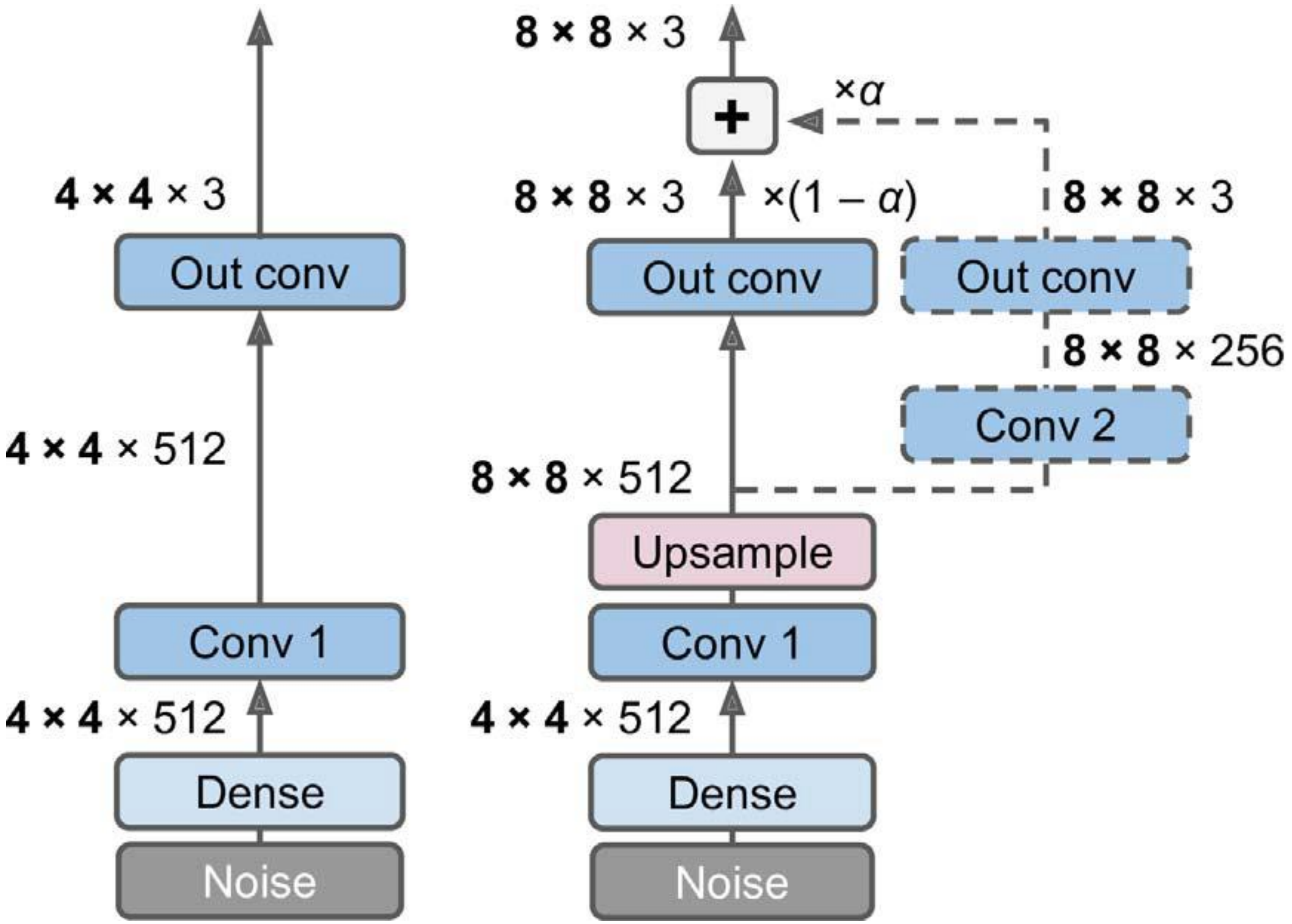


Figure 19: Progressively growing GAN - a GAN generator outputs 4 × 4 color images (left); we extend it to output 8 × 8 images (right).

… Progressive Growing of GANs

- The weight of the new outputs is α, while the weight of the original outputs is 1 – α, and α is slowly increased from 0 to 1.
- In other words, the new convolutional layers (represented with dashed lines in Figure 19) are gradually faded in, while the original output layer is gradually faded out.
- A similar fade-in/fade-out technique is used when a new convolutional layer is added to the discriminator (followed by an average pooling layer for downsampling).
- The paper (#1) also introduced several other techniques aimed at increasing the diversity of the outputs (to avoid mode collapse) and making training more stable:
  - *Minibatch standard deviation layer*: Added near the end of the discriminator.
    - For each position in the inputs, it computes the standard deviation across all channels and all instances in the batch (S = tf.math.reduce_std(inputs, axis=[0, -1])).

… Progressive Growing of GANs

- These standard deviations are then averaged across all points to get a single value (v = tf.reduce_mean(S)).
- Finally, an extra feature map is added to each instance in the batch and filled with the computed value (tf.concat([inputs, tf.fill([batch_size, height, width, 1], v)], axis=-1)).
- **How does this help?** Well, if the generator produces images with little variety, then there will be a small standard deviation across feature maps in the discriminator.
- Thus, the discriminator will have easy access to this statistic, making it less likely to be fooled by a generator that produces too little diversity.
- This will encourage the generator to produce more diverse outputs, reducing the risk of mode collapse.

- *Equalized learning rate*:
  - Initializes all weights using a simple Gaussian distribution with mean 0 and standard deviation 1 rather than using He initialization.

… Progressive Growing of GANs

- However, the weights are scaled down at runtime (i.e., every time the layer is executed) by the same factor as in He initialization:
  - they are divided by $\sqrt{2/n_{\text{inputs}}}$, where n_{inputs} is the number of inputs to the layer.
- this technique significantly improved the GAN's performance when using RMSProp, Adam, or other adaptive gradient optimizers.
- Indeed, these optimizers normalize the gradient updates by their estimated standard deviation, so parameters that have a larger dynamic range will take longer to train, while parameters with a small dynamic range may be updated too quickly, leading to instabilities.
- By rescaling the weights as part of the model itself rather than just rescaling them upon initialization, this approach ensures that the dynamic range is the same for all parameters, throughout training, so they all learn at the same speed.
- This both speeds up and stabilizes training.

… Progressive Growing of GANs

- *Pixelwise normalization layer*: Added after each convolutional layer in the generator.

- It normalizes each activation based on all the activations in the same image and at the same location, but across all channels (dividing by the square root of the mean squared activation).

- In TensorFlow code, this is inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8) (the smoothing term 1e-8 is needed to avoid division by zero).

- This technique avoids explosions in the activations due to excessive competition between the generator and the discriminator.

- The combination of all these techniques allowed the authors to generate very convincing high-definition images of faces.

- Evaluation is one of the big challenges when working with GANs:

  - although it is possible to automatically evaluate the diversity of the generated images, judging their quality is a much trickier and subjective task.

… Progressive Growing of GANs

- One technique is to use human raters, but this is costly and time-consuming.
- So, the authors proposed to measure the similarity between the local image structure of the generated images and the training images, considering every scale.
- This idea led them to another groundbreaking innovation: **StyleGANs**.

StyleGANs

- Popular StyleGAN architecture was introduced in paper #1, see Figure 20:

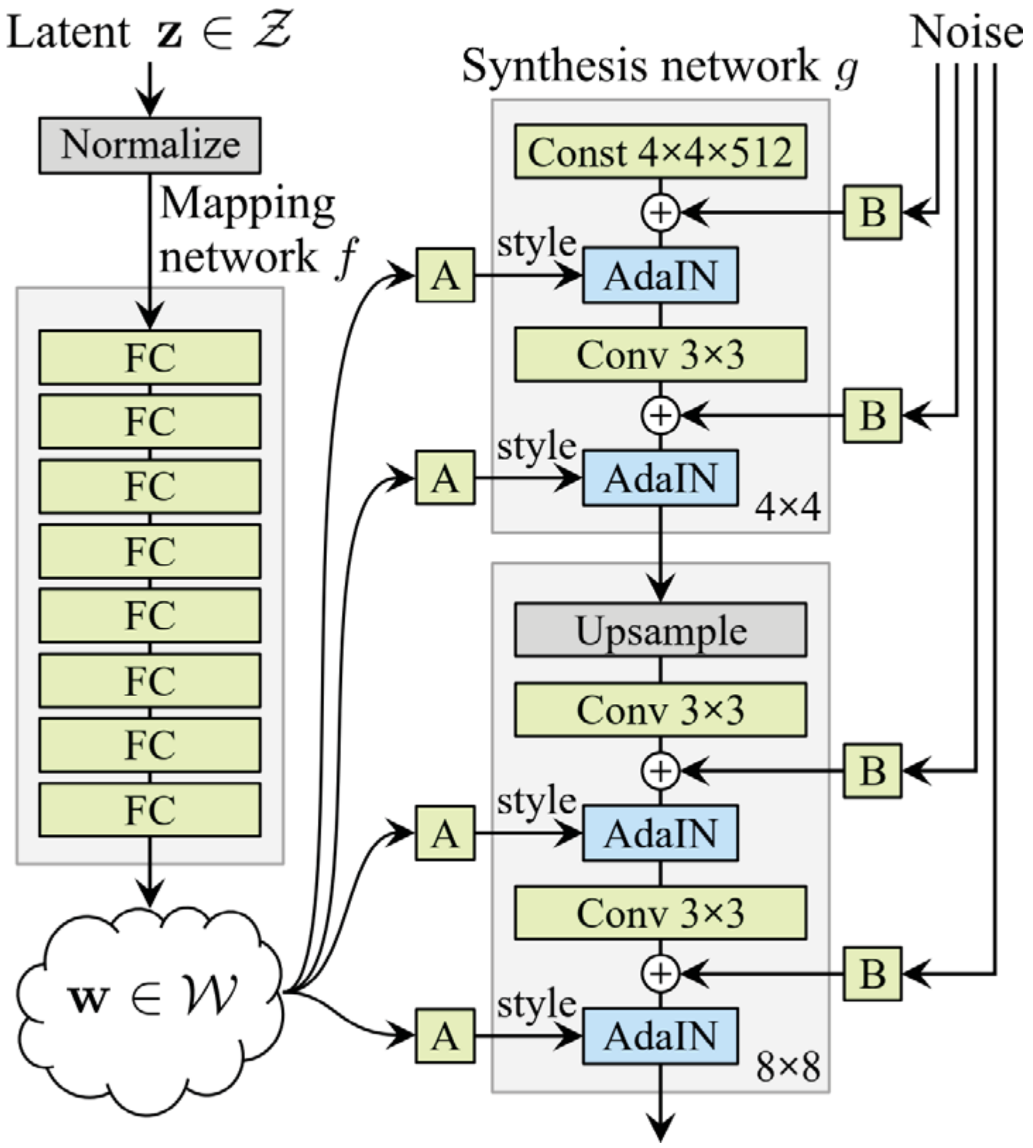


Figure 20: StyleGAN's generator architecture.

… StyleGANs

- *Style transfer* techniques was used in the generator to ensure that the generated images have the same local structure as the training images, at every scale, greatly improving the quality of the generated images.
- The discriminator and the loss function were not modified, only the generator.
- StyleGAN is composed of two networks:
- *Mapping network*: An eight-layer MLP that maps the latent representations **z** (i.e., the codings) to a vector **w**.
  - This vector is then sent through multiple affine transformations (i.e., Dense layers with no activation functions, represented by the "A" boxes in Figure 20), which produces multiple vectors.
  - These vectors control the style of the generated image at different levels, from fine-grained texture (e.g., hair color) to high-level features (e.g., adult or child).
  - In short, the mapping network maps the codings to multiple style vectors.
- *Synthesis network*: Responsible for generating the images.
  - It has a constant learned input (to be clear, this input will be constant after training, but during training it keeps getting tweaked by backpropagation).

… StyleGANs

- It processes this input through multiple convolutional and upsampling layers, as earlier, but there are two twists:
  - first, some noise is added to the input and to all the outputs of the convolutional layers (before the activation function).
  - Second, each noise layer is followed by an *Adaptive Instance Normalization* (AdaIN) layer:
    - it standardizes each feature map independently (by subtracting the feature map's mean and dividing by its standard deviation),
    - then it uses the style vector to determine the scale and offset of each feature map (the style vector contains one scale and one bias term for each feature map).
- The idea of adding noise independently from the codings is very important.
- Some parts of an image are quite random, such as the exact position of each freckle or hair.

… StyleGANs

- In earlier GANs, this randomness had to either come from the codings or be some pseudorandom noise produced by the generator itself.

- If it came from the codings, it meant that the generator had to dedicate a significant portion of the codings' representational power to store noise:

  - this is quite wasteful.

  - Moreover, the noise had to be able to flow through the network and reach the final layers of the generator:

    - this seems like an unnecessary constraint that probably slowed down training.

- And finally, some visual artifacts may appear because the same noise was used at different levels.

- If instead the generator tried to produce its own pseudorandom noise, this noise might not look very convincing, leading to more visual artifacts.

StyleGANs

- Popular StyleGAN architecture was introduced in paper #1, see Figure 20:

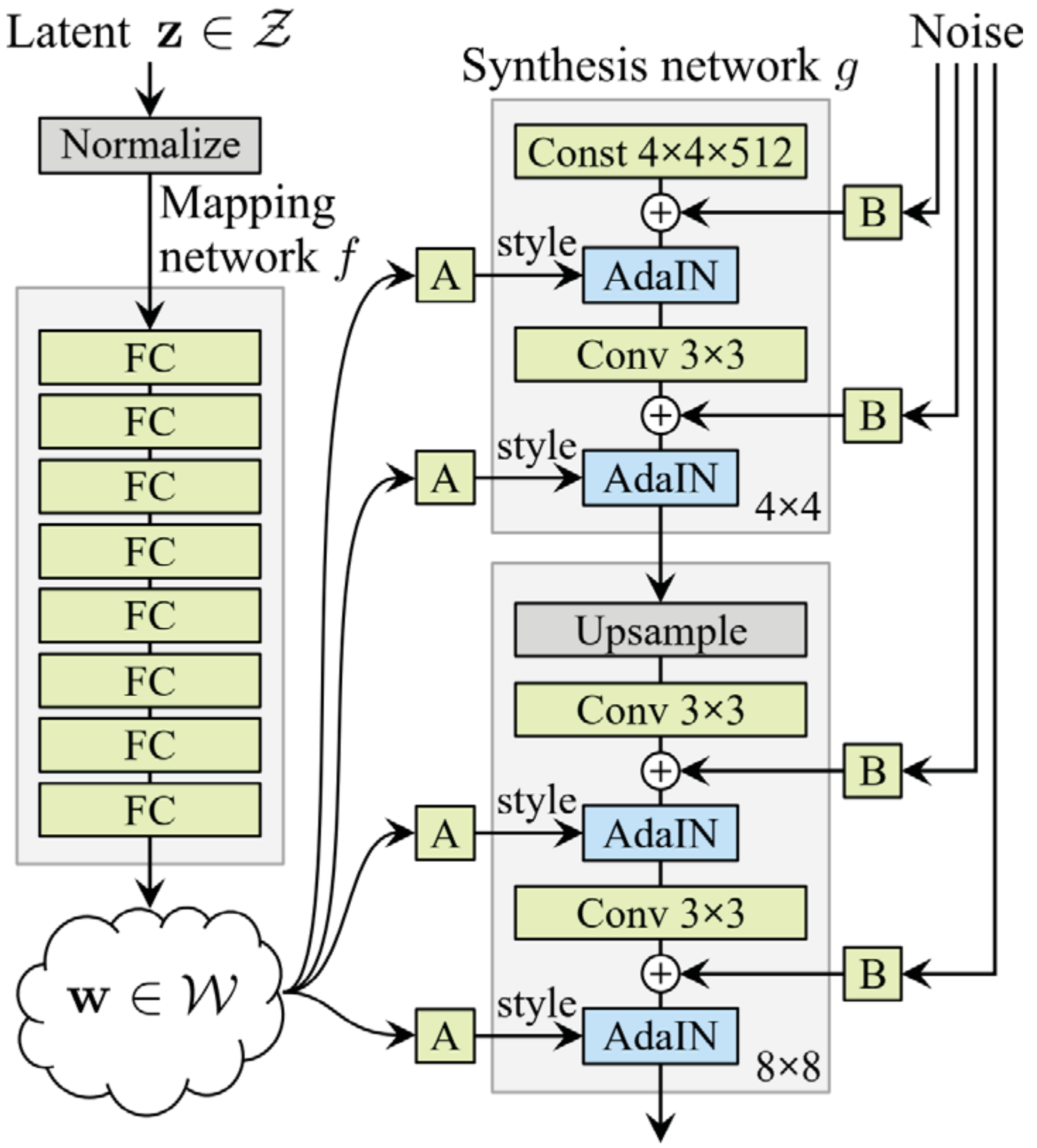


Figure 20: StyleGAN's generator architecture.

… StyleGANs

- Plus, part of the generator's weights would be dedicated to generating pseudorandom noise, which again seems wasteful.
- By adding extra noise inputs, all these issues are avoided;
- the GAN is able to use the provided noise to add the right amount of stochasticity to each part of the image.
- The added noise is different for each level.
  - Each noise input consists of a single feature map full of Gaussian noise, which is broadcast to all feature maps (of the given level) and
  - scaled using learned per-feature scaling factors (this is represented by the "B" boxes in Figure 20) before it is added.
- Finally, StyleGAN uses a technique called *mixing regularization* (or *style mixing*), where a percentage of the generated images are produced using two different codings.
  - Specifically, the codings $\mathbf{c}_1$ and $\mathbf{c}_2$ are sent through the mapping network, giving two style vectors $\mathbf{w}_1$ and $\mathbf{w}_2$.

… StyleGANs

- Then the synthesis network generates an image based on the styles $\mathbf{w}_1$ for the first levels and the styles $\mathbf{w}_2$ for the remaining levels.
- The cutoff level is picked randomly.
- This prevents the network from assuming that styles at adjacent levels are correlated, which in turn encourages locality in the GAN, meaning that each style vector only affects a limited number of traits in the generated image.

➢ There is such a wide variety of GANs out there that it would require a whole book to cover them all.

➢ If we just want to get some amazing results quickly, then we can just use a pretrained model (e.g., there are pretrained StyleGAN models available for Keras).

END.