Representation Learning & Generative Learning Using Autoencoders and GANs

Autoencoders and GANs

- In this chapter we will learn about
    - Fundamental concepts in Autoencoders
    - Various types of Autoencoders
    - Implementing Autoencoders using Keras

- Finally, we will take a look at GANs
    - Implement GAN for Fashion MNIST
    - Difficulties of Training GANs
    - DCGAN and StyleGAN

Autoencoders

What are Autoencoders?

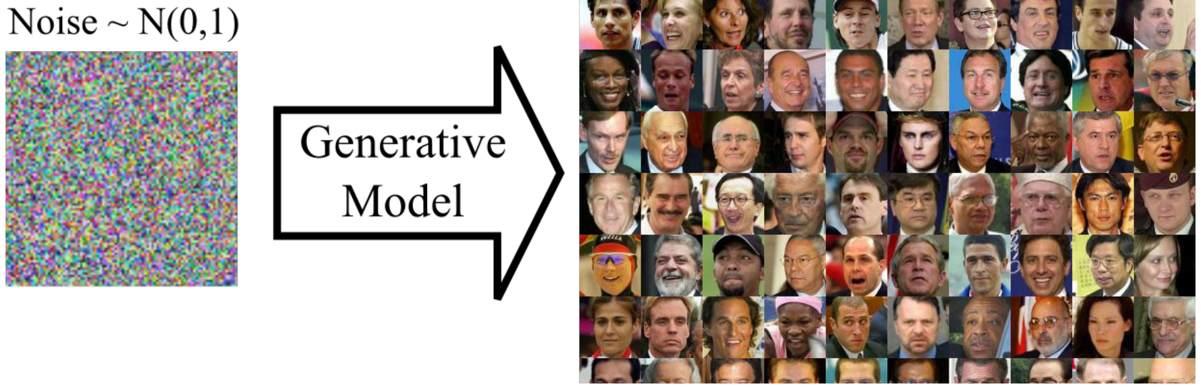
- Artificial neural networks
- Capable of learning efficient representations of the input data, called **codings**, without any supervision
- The training set is unlabeled
- Codings typically have a lower dimensionality than the input data
- These makes autoencoders useful for dimensionality reduction

Why 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**.

Why Autoencoders?

- For ex., we could train an Autoencoder on pictures of faces, and it would then be able to generate new faces



Autoencoders

- 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 size of internal representation, or add noise to inputs and train the network to recover the original inputs.
- These constraints prevent it from copying inputs directly to outputs
- This forces it to learn efficient ways of representing the data

Autoencoders

In short, codings are byproducts of autoencoder’s attempt to learn

Identity function under some constraints

Autoencoders

What we will 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,
  - Unsupervised pre training,
  - 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 easier 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

- 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

- The **shorter** sequence i.e. the first one**?**

Efficient Data Representations

- 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

- The **shorter** sequence i.e. the first one**?**
- But second sequence follows two simple rules:
  - Odd ⇒ 3N+1
  - Even ⇒ N/2
- This is **hailstone sequence**

Efficient Data Representations

- 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

Now which one is easier?

Second sequence. you only need to memorize

- The two rules. Odd ⇒ 3N+1, Even ⇒ N/2
- The first number
- How many numbers you want
- Not the entire sequence

Efficient Data Representations

- If we could easily memorize a sequence,
    - we won't figure out a pattern
    - Instead, we'd byheart every number

Efficient Data Representations

- If we could easily memorize a sequence,
  - we won't figure out a pattern
  - Instead, we'd byheart every number
- ***Good memory is not always a boon!!***

Efficient Data Representations

- If we could easily memorize a sequence,
  - we won't figure out a pattern
  - Instead, we'd byheart every number
- ***Good memory is not always a boon!!***
- So, constraining a Neural Network, pushes it to
  - Discover patterns in data
  - That's why **Autoencoders**

Efficient Data Representations

Relationship between memory, perception, and pattern matching was studied by **William Chase** and **Herbert Simon** in the early 1970s. They observed ....

William Chase

Herbert Simon

Efficient Data Representations

*"Expert chess players can memorize board in just **5 seconds**."*

Provided

- The pieces were placed in **realistic positions**,
- Not **randomly**

-- William Chase and Herbert Simon, 1970

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

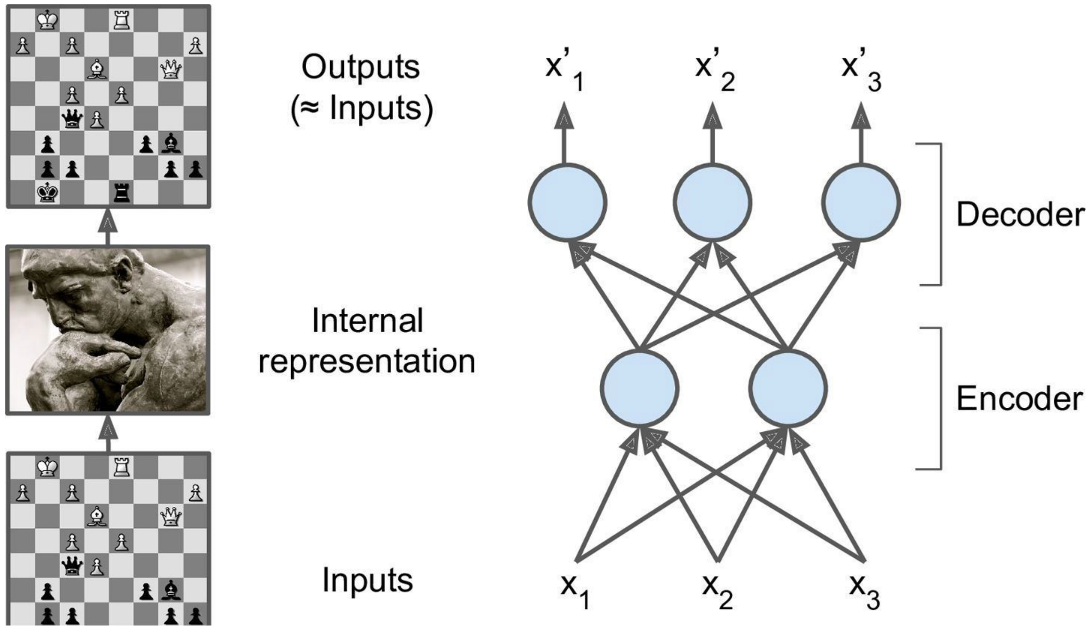
Composition of Autoencoder

Composition of Autoencoder

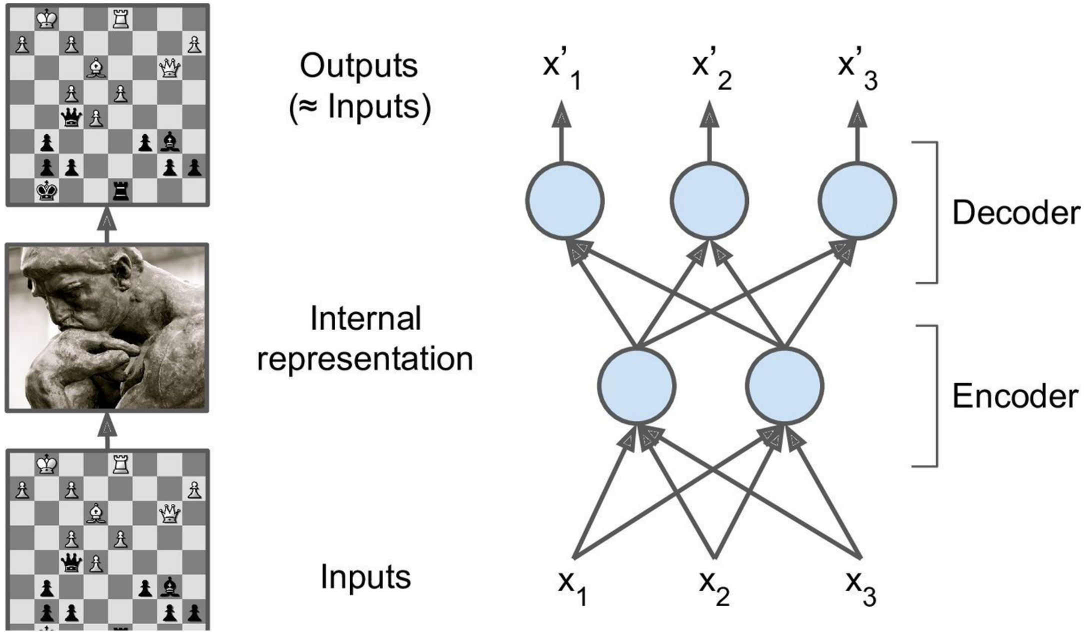
An autoencoder is always composed of two parts:

- 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

Composition of Autoencoder



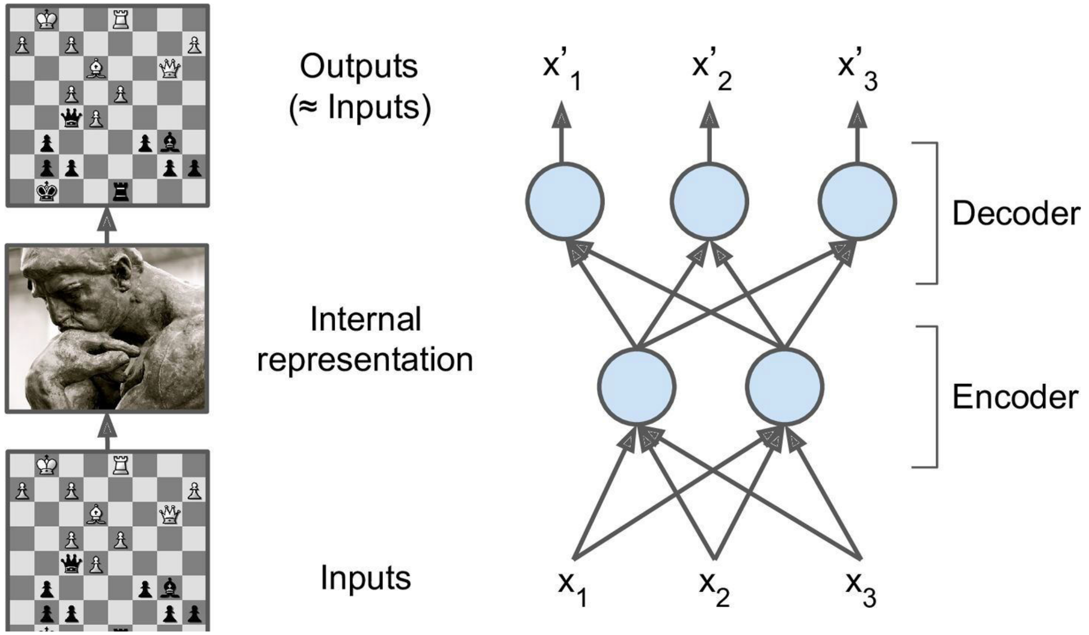
Composition of Autoencoder



An **autoencoder** typically has the same architecture as a **Multi-Layer Perceptron**,

except that the **no. of neurons** in the output layer must be **equal to the no. of inputs**

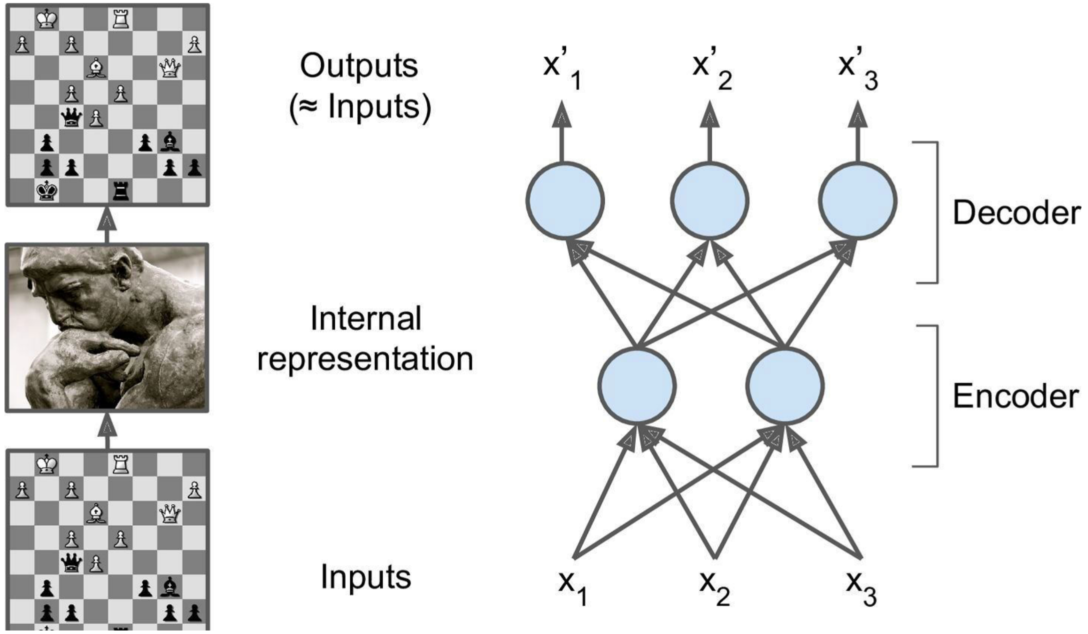
Composition of Autoencoder



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

And **one output layer** composed of three neurons (**the decoder**)

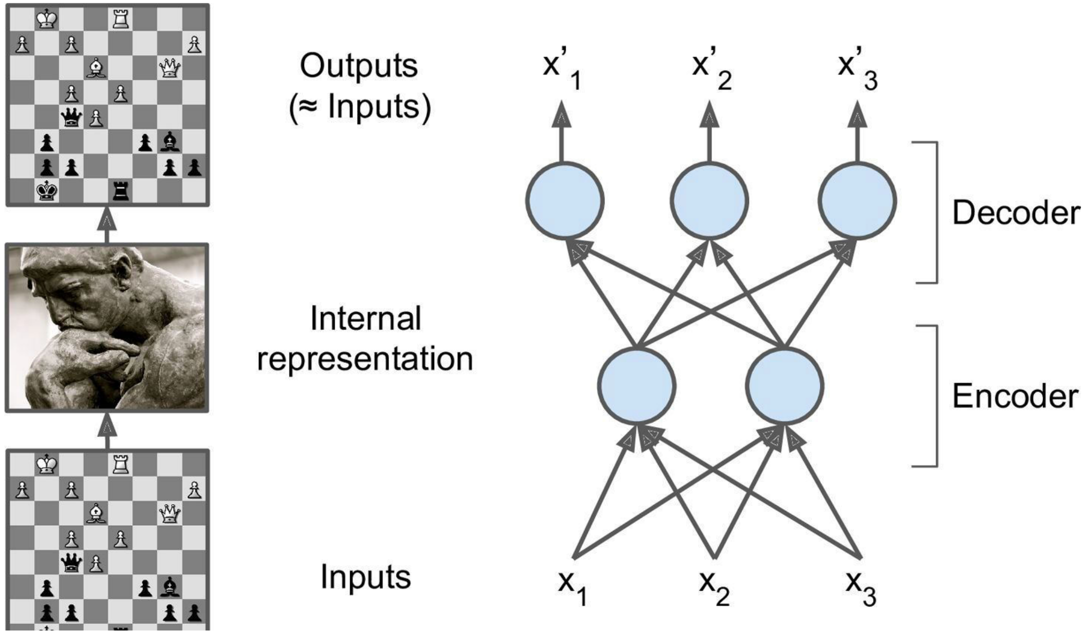
Composition of Autoencoder



The outputs are called **Reconstructions**

And cost function contains a **Reconstruction loss** that penalizes the model

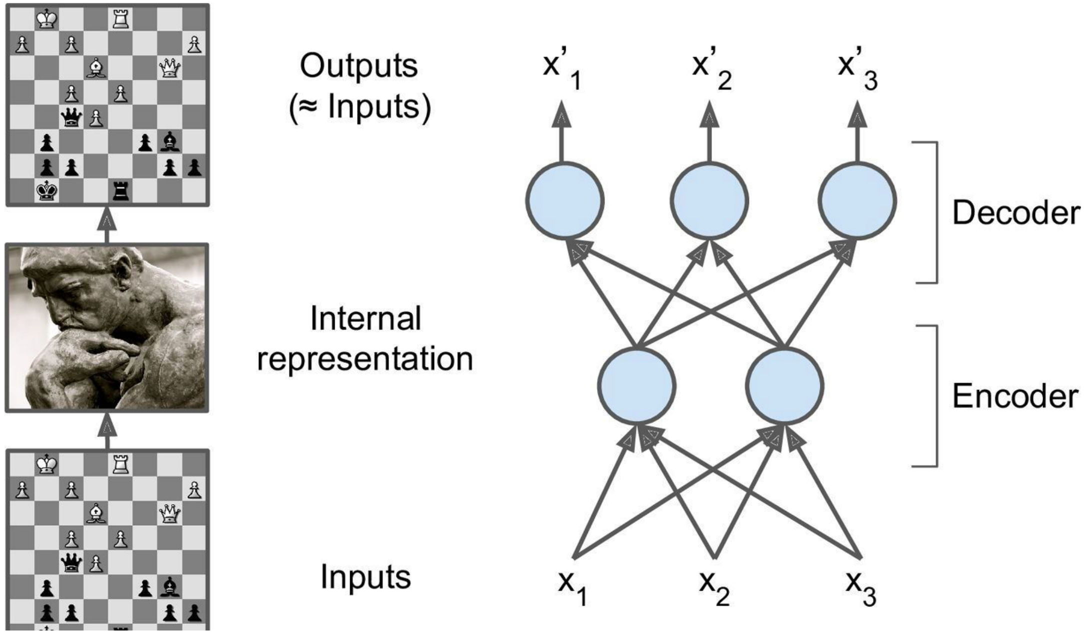
Composition of Autoencoder



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

So, the autoencoder is said to be **undercomplete**

Composition of Autoencoder



An **undercomplete** autoencoder cannot trivially copy its inputs to the **codings**.

It is forced to learn the most important features in the input data

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 (PCA)**

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

Switch to Notebook

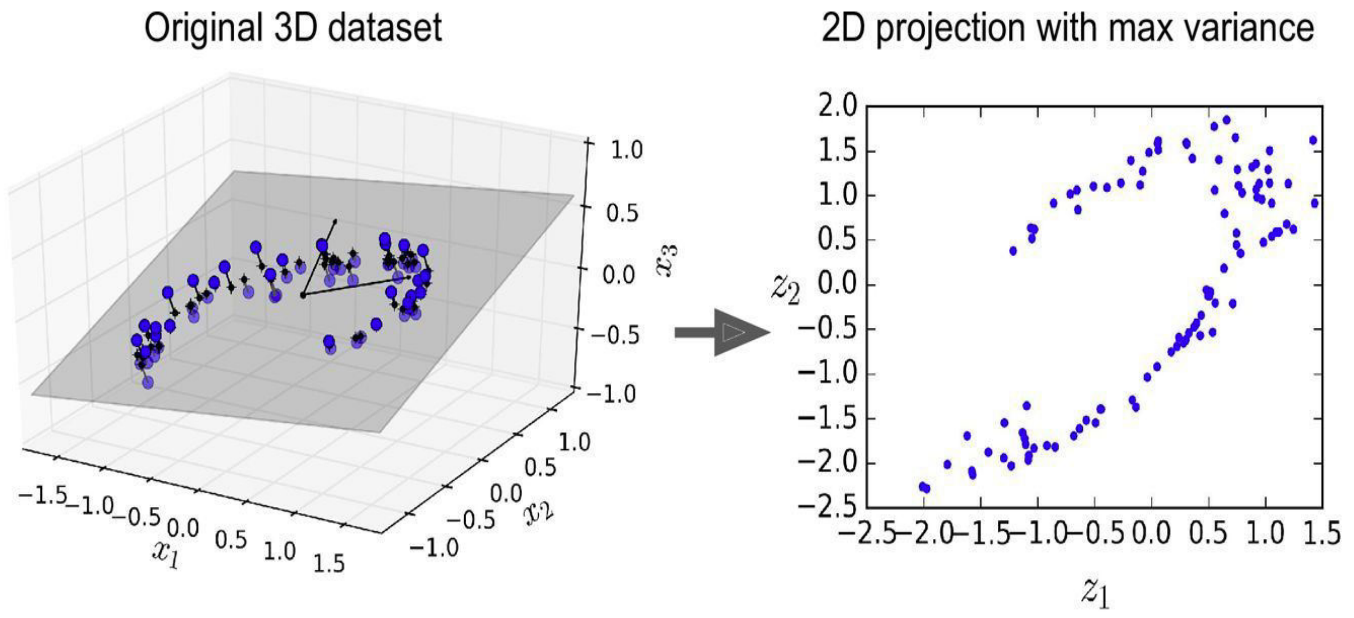
`autoencoders_and_gans.ipynb`

PCA with an Undercomplete Linear Autoencoder

The two things to note in the previous code are:

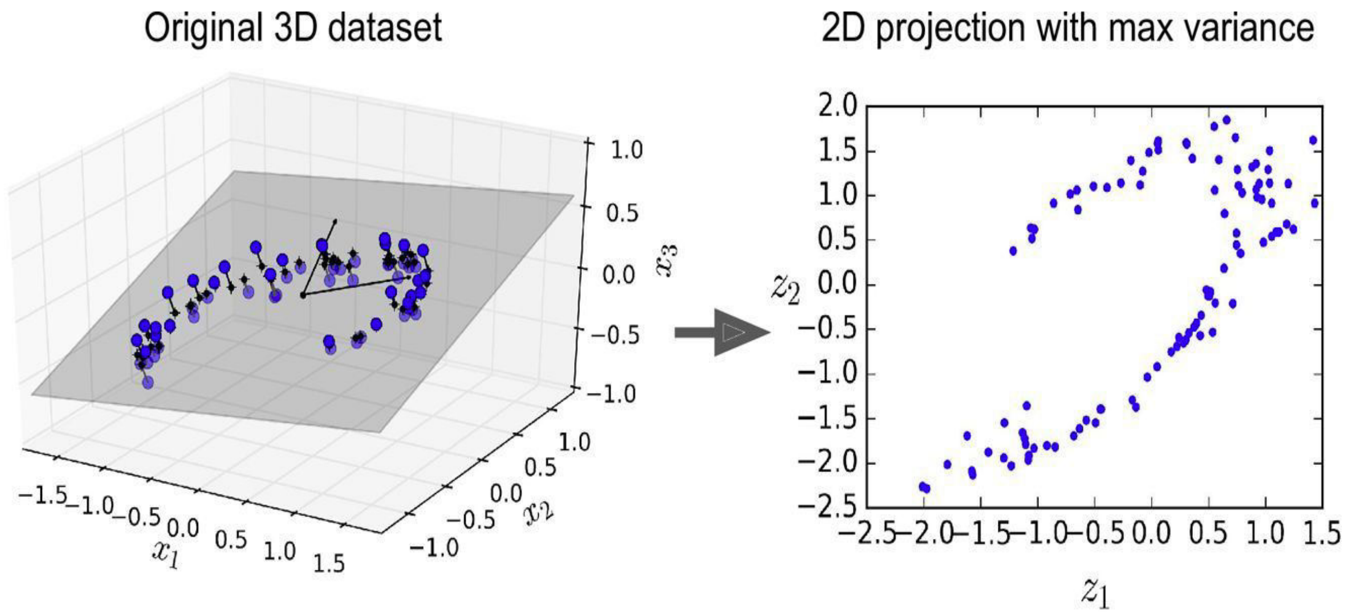
- The number of **outputs** is equal to the number of **inputs**
- To perform simple PCA, we set **activation_fn=None** i.e., all neurons are linear, and the cost function is the MSE.

PCA with an Undercomplete Linear Autoencoder



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

Stacked Autoencoders

Stacked Autoencoders

- Autoencoders can have multiple hidden layers
  - They are called **Stacked Autoencoders** (or **Deep Autoencoders**)
  - They can learn more complex codings

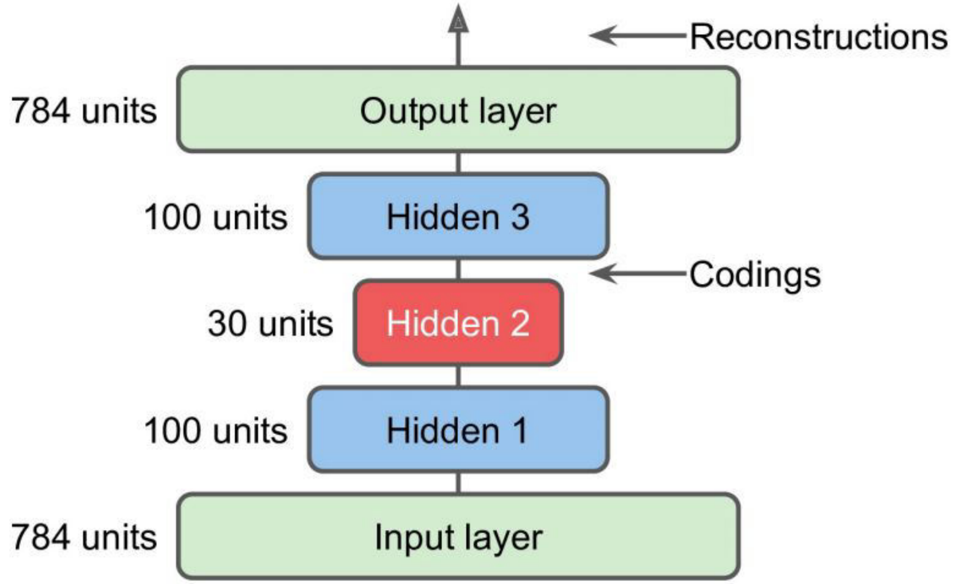
Stacked Autoencoders

- But if autoencoder is too powerful
  - It will map each input to a single arbitrary number
  - And decoder learns reverse mapping
  - It will reconstruct the training data perfectly
  - But it will not have learned any useful data representation

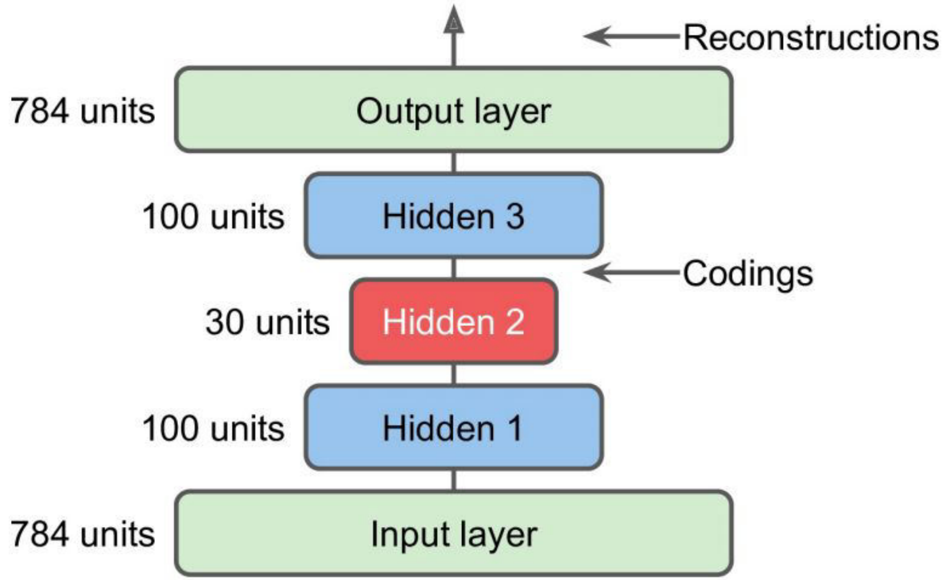
Architecture of Stacked Autoencoders

Architecture of Stacked Autoencoders

- It is typically symmetrical with regard to central hidden layer (coding layer)



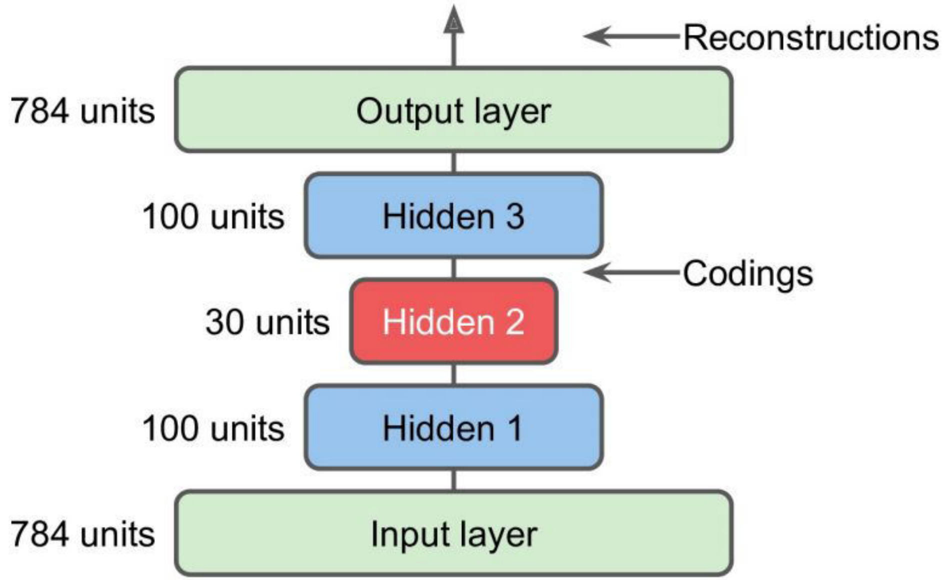
Architecture of Stacked Autoencoders



We split autoencoder model into two submodels:

Encoder & **decoder**

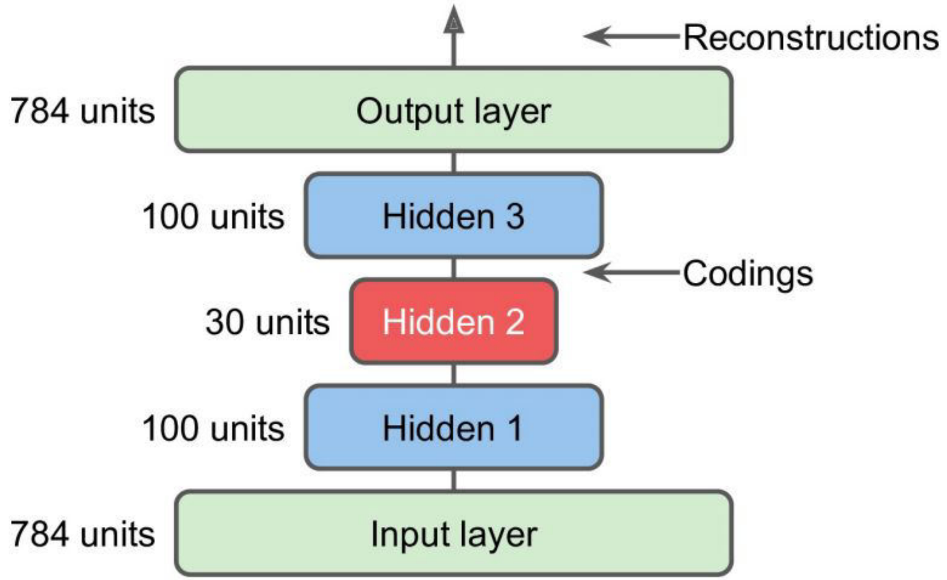
Architecture of Stacked Autoencoders



Encoder takes 28 × 28 px grayscale images

Flattens them into vector of size **784**

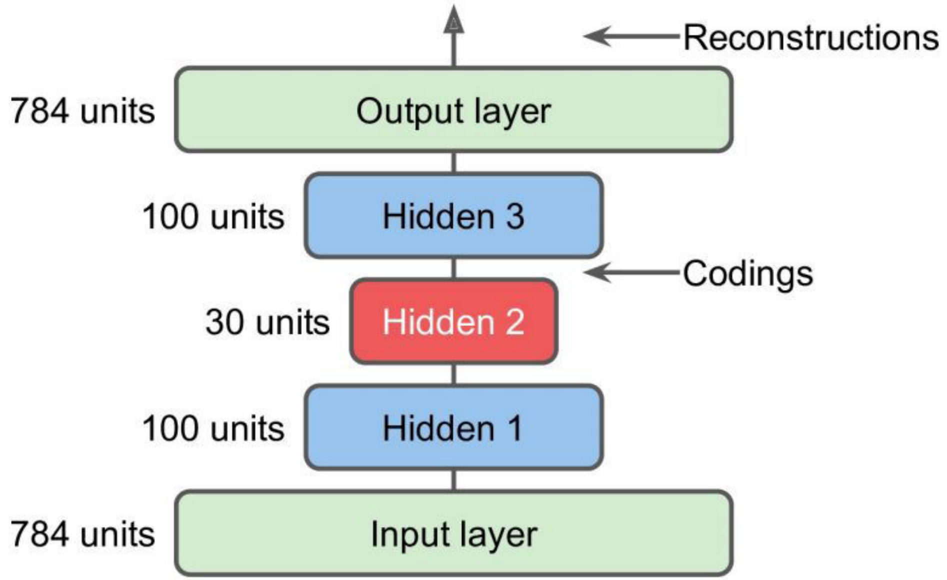
Architecture of Stacked Autoencoders



Two Dense layers of diminishing sizes of 100 units then 30 units

Both use **SELU** activation function

Architecture of Stacked Autoencoders

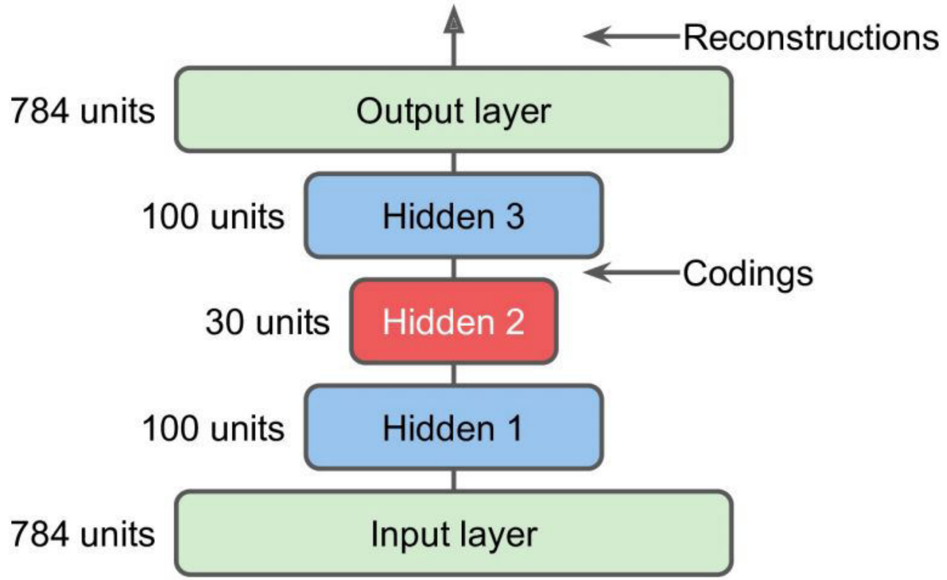


Decoder takes codings of size 30

Processes them through **two Dense layers** of increasing sizes (100 units then 784 units),

And it reshapes the final vectors into 28 × 28 arrays

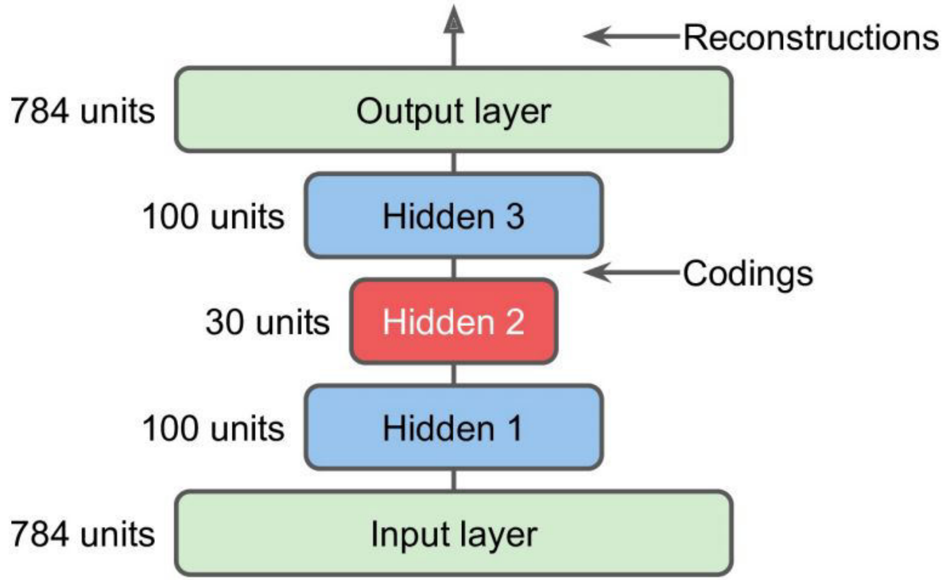
Architecture of Stacked Autoencoders



When compiling, we use **Binary cross-entropy loss** instead of **Mean squared error**

It is treated as a **multilabel binary classification** problem

Architecture of Stacked Autoencoders



Finally, we train the model using **X_train** as both inputs & targets

Similarly, we use **X_valid** as both validation inputs & targets.

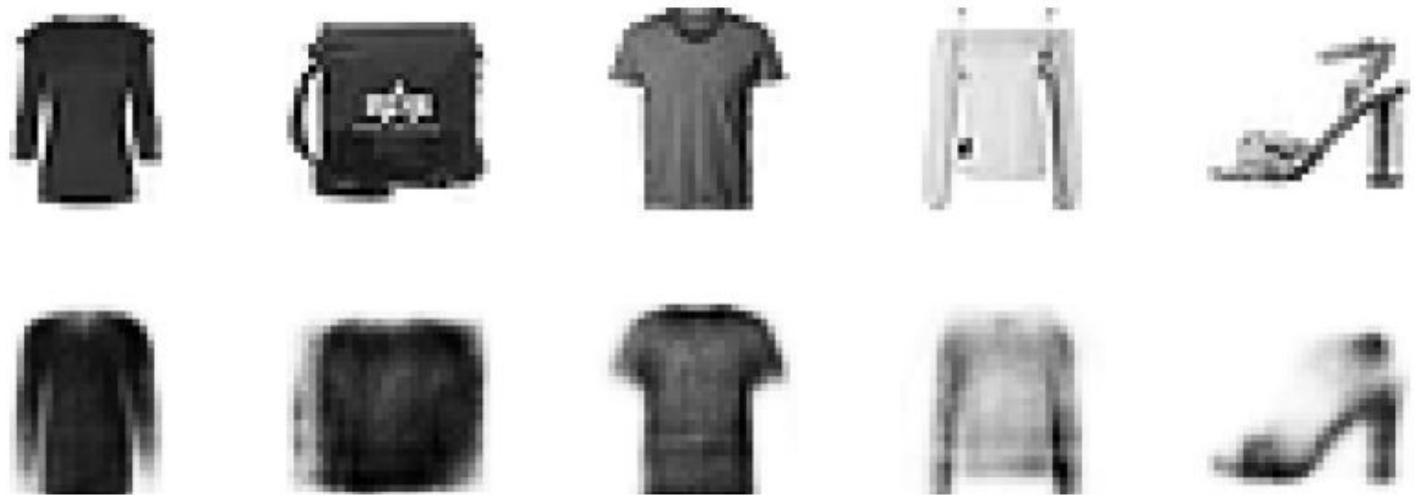
Implementing a Stacked Autoencoder Using Keras

Switch to Notebook

`autoencoders_and_gans.ipynb`

Visualizing the Reconstructions

- Let’s plot a few images from the **validation set**, as well as their reconstructions to ensure it is properly trained



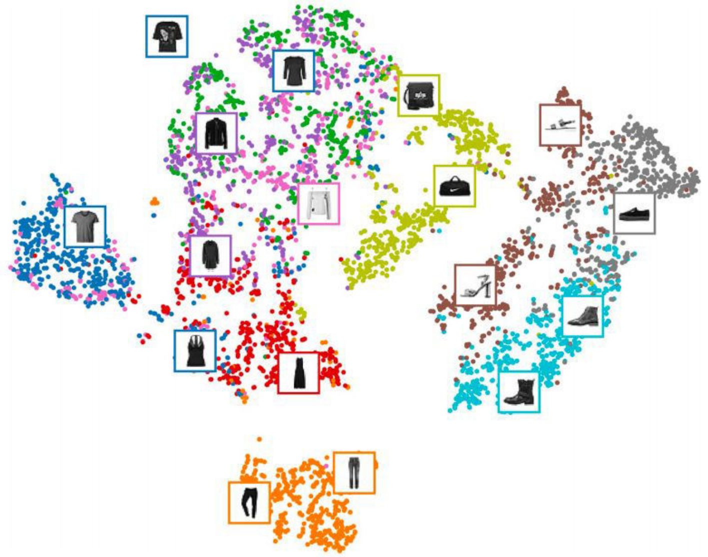
Visualizing the Fashion MNIST Dataset

Visualizing the Fashion MNIST Dataset

- We can use autoencoder to reduce the dataset's dimensionality
- It's not better compared to other dimensionality reduction algorithms for visualization
- **Advantage**: They can handle
  - large datasets,
  - with many instances
  - and many features

Visualizing the Fashion MNIST Dataset

- Encoder from stacked autoencoder is used to reduce the dimensionality down to 30
- **t-SNE algorithm** is used to reduce dimensionality down to 2 for visualization



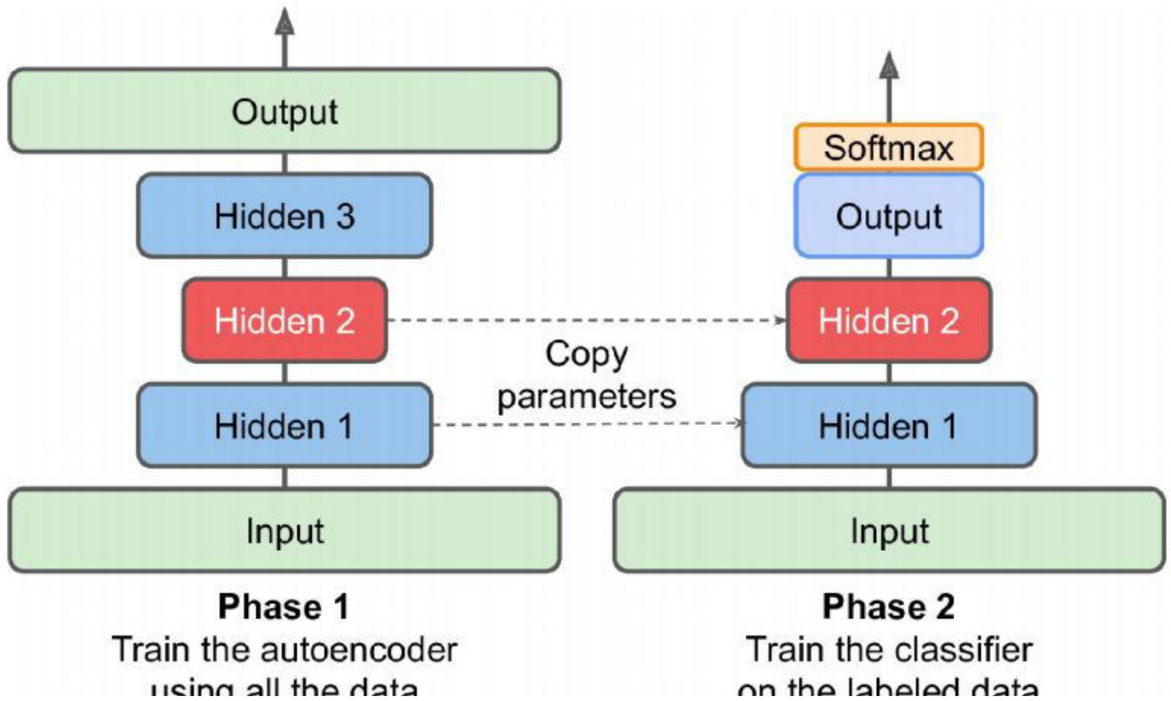
Unsupervised Pretraining Using Stacked Autoencoders

Unsupervised Pretraining Using Stacked Autoencoders

- For a large mostly unlabeled dataset,
  - First train a stacked autoencoder using all the data,
  - Then reuse lower layers for our actual NN
  - Train it using the labeled data

- Makes possible to train a high-performance model using little training data because our NN won't have to learn all the low-level features

Unsupervised Pretraining Using Stacked Autoencoders

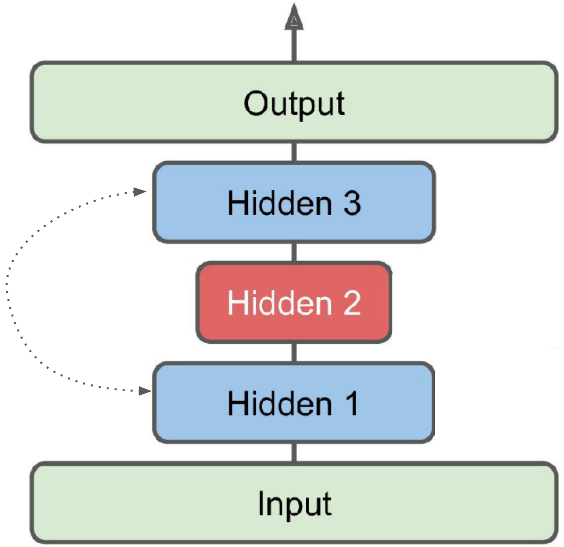


Tying Weights

Tying Weights

- When an autoencoder is **neatly symmetrical**
  - Weights of decoder layers can be tied to weights of encoder layers

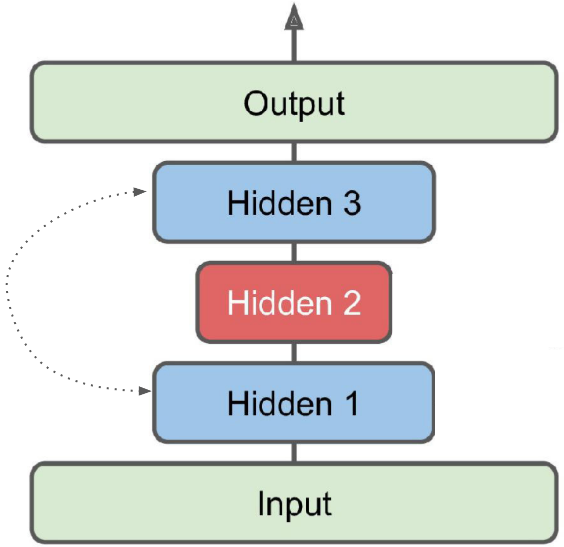
- This halves the no. of weights in the model
- Speeds up training
- And limits the risk of overfitting



Tying Weights

- If autoencoder has **N** layers (excl. input layer)
- $\mathbf{W}_L \rightarrow$ Connection weights of the **L** layer
- Then, Decoder layer weights is:

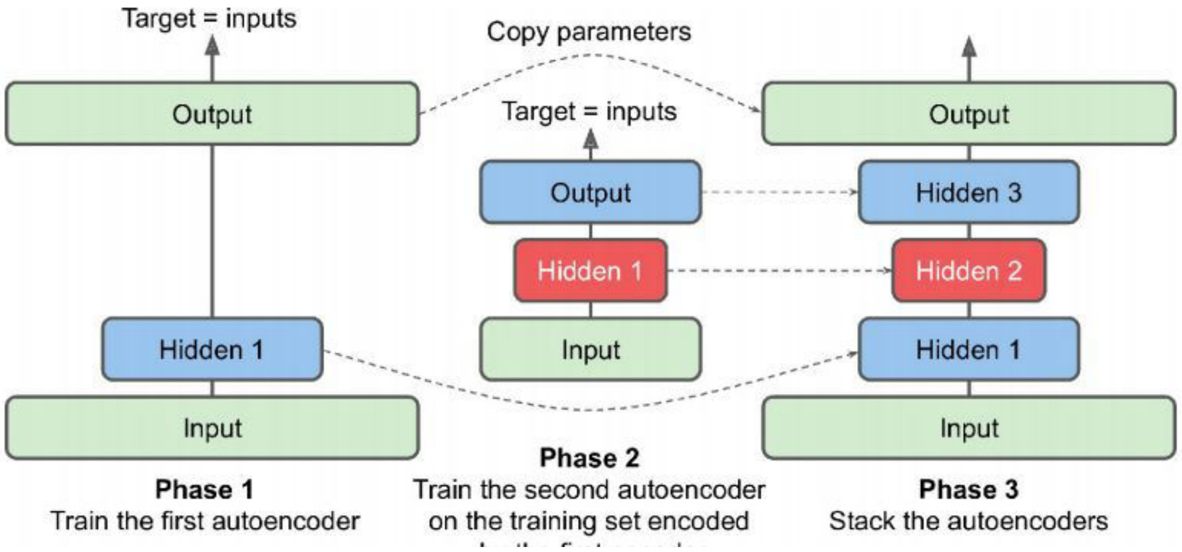
$$\mathbf{W}_{N-L+1} = \mathbf{W}_L^{\top} \quad \text{With } L = (1, 2, \ldots, \tfrac{N}{2})$$



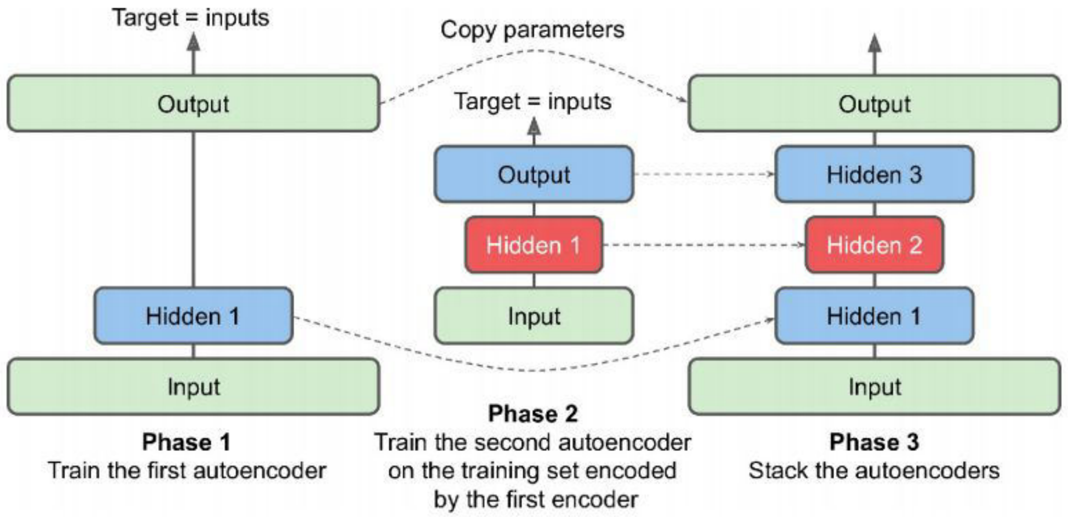
Training One Autoencoder at a Time

Training One Autoencoder at a Time

- It is possible to train one shallow autoencoder at a time
- Then stack them into a single stacked autoencoder



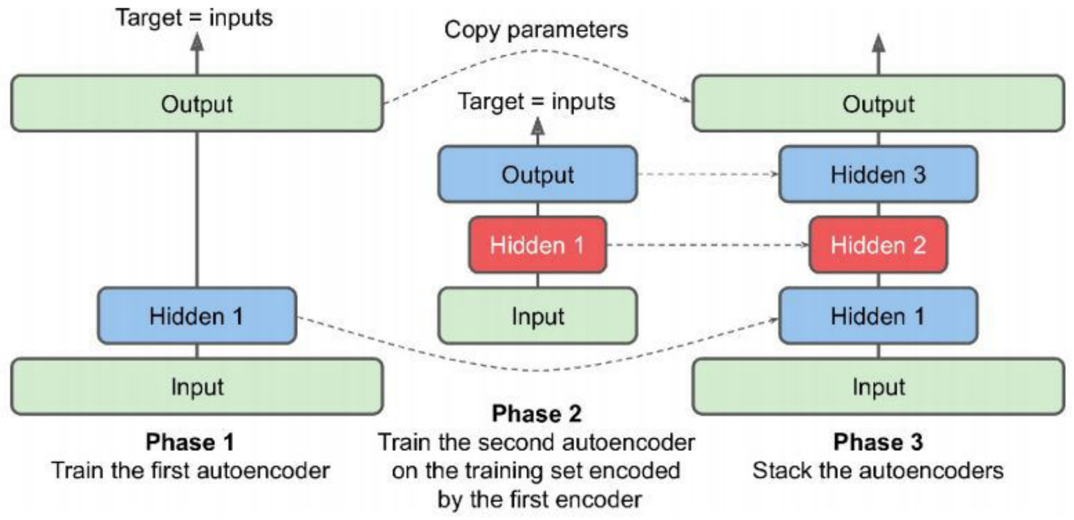
Training One Autoencoder at a Time



- **First phase of training**

First autoencoder learns to reconstruct the inputs.

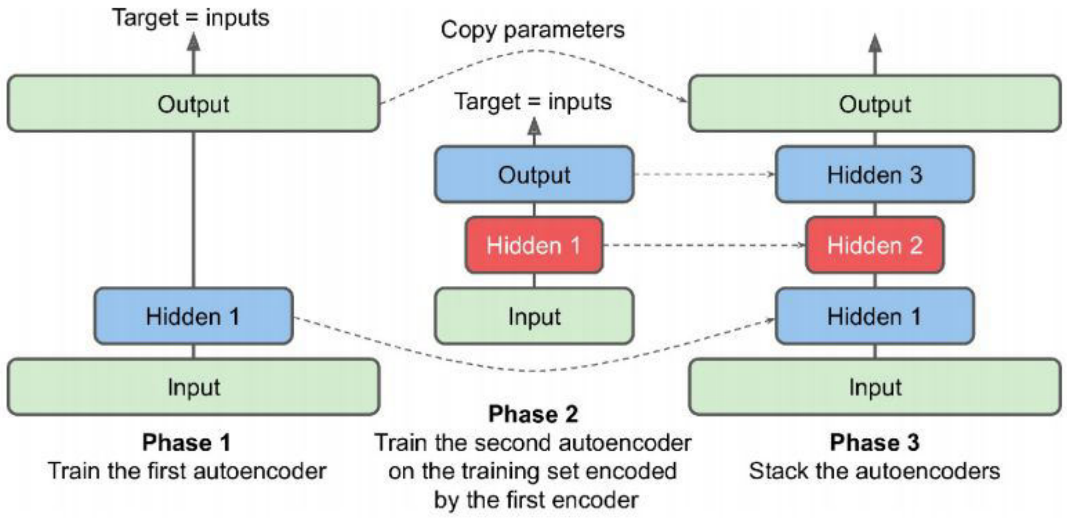
Training One Autoencoder at a Time



Then we encode the whole training set using this first autoencoder

This gives us a new **(compressed) training set**.

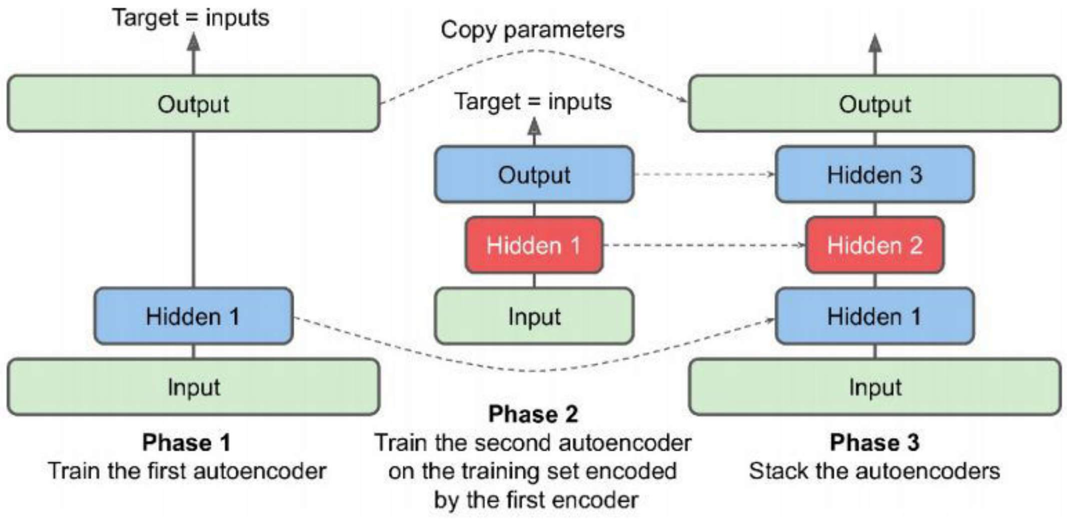
Training One Autoencoder at a Time



- **Second phase of training**

We train a second autoencoder on this new dataset

Training One Autoencoder at a Time



This gives us the final stacked autoencoder

Convolutional Autoencoders

Convolutional Autoencoders

- They are more suited than dense networks to work with **images**

- Encoder is a regular CNN composed of:
    - **Convolutional layers** and
    - **Pooling layers**

Convolutional Autoencoders

- It reduces **spatial dimensionality** of inputs (i.e., height and width)
- Increases **depth** (i.e., the number of feature maps).
- Decoder does reverse and uses transpose convolutional layers

Recurrent Autoencoders

Recurrent Autoencoder

- They are better suited for **sequences**, such as **time series** or **text** (e.g., for unsupervised learning or dimensionality reduction)
- Encoder is a **sequence-to-vector RNN**
- It compresses input sequence down to a single vector
- Decoder is a vector-to-sequence RNN that does the reverse

Recurrent Autoencoder

- It can be used for any kind of sequence
- It can process Fashion MNIST images by treating each image as a sequence of rows
- We use a **RepeatVector** layer as the first layer of the decoder, to ensure that its input vector gets fed to the decoder at each time step

Undercomplete vs Overcomplete

Undercomplete vs Overcomplete

- To force autoencoder to learn interesting features, size of coding layer was limited
- This made it **undercomplete**

- There are other kinds of constraints:
  - Allow coding layer to be just as large as the inputs, or even larger
  - This results in **overcomplete** autoencoder

Let's look at some of these approaches

Denoising Autoencoders

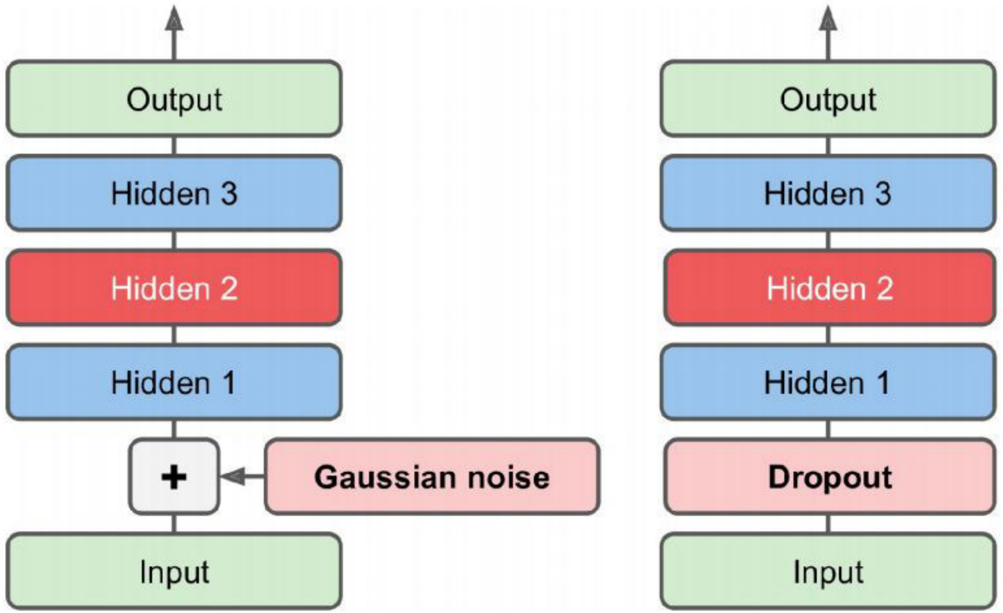
Denoising Autoencoder

- To learn useful features
  - We can add noise to its inputs
  - And train it to recover 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**

Denoising Autoencoder

- The figure shows both autoencoders, with Gaussian noise or dropout



Denoising Autoencoder

- It is a regular stacked autoencoder
- Gaussian Noise or Dropout layer is applied to the encoder’s inputs
- Gaussian Noise or Dropout layer is only active during training

Sparse Autoencoders

Sparse Autoencoder

- **Sparsity** can be used as a constraint for feature extraction
- It’s added to **cost function**
- Autoencoder is pushed to reduce no. of active neurons in coding layer
- For ex. on average only 5% significantly active neurons in coding layer

Sparse Autoencoder

- This forces autoencoder to represent each inputs a combination of a small no. of activations
- As a result, each neuron in coding layer typically ends up representing a useful feature

First Approach to

Sparse Autoencoders

First Approach to Sparse Autoencoder

- Use **sigmoid activation function** in coding layer
- Use a **large coding layer**
- Add **ℓ_1 regularization** to the coding layer’s activations
- Decoder is regular decoder

First Approach to Sparse Autoencoder

- **ActivityRegularization** layer adds a training loss equal to the sum of absolute values of its inputs
- This penalty encourages it to produce codings close to 0

Second Approach to Sparse Autoencoders

Second Approach to Sparse Autoencoder

- Measure actual **sparsity** of the coding layer at each training iteration
- **Penalize** the model when measured sparsity differs from a target sparsity

How are neurons penalized?

How are neurons penalized?

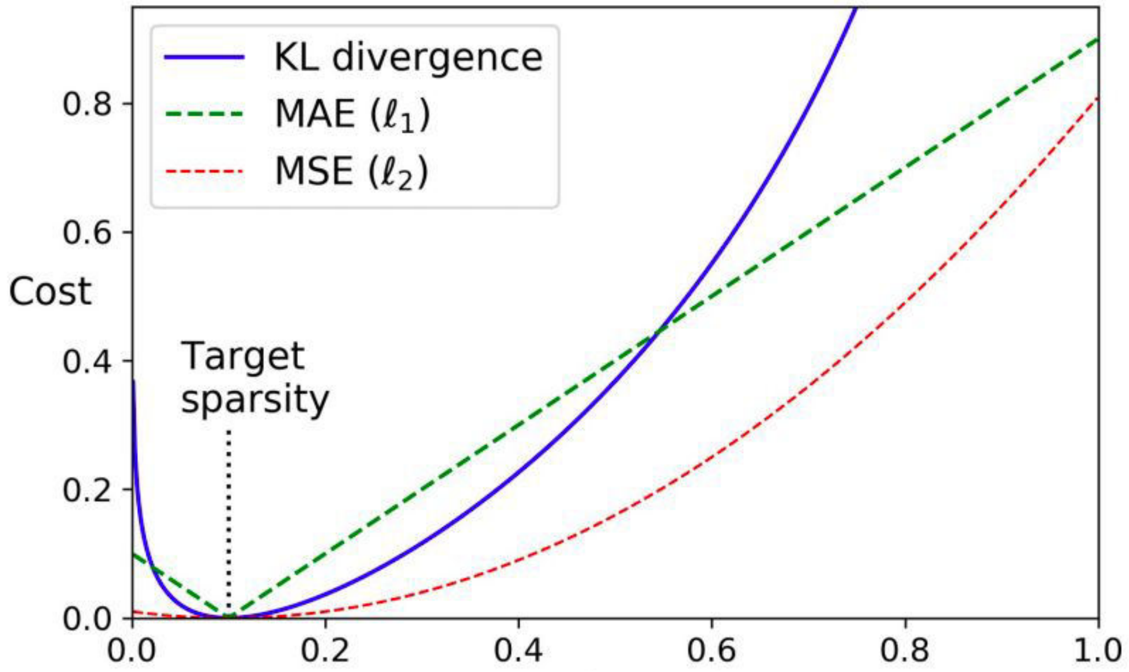
- Compute average activation of each neuron in the coding layer, over the whole training batch
- **Penalize** the neurons that are too active, or not active enough
- By adding a **sparsity loss** to the cost function

How are neurons penalized?

- **MSE** can be used
- **Kullback–Leibler (KL) divergence** is better suited

Sparsity Loss

Sparsity Loss



Sparsity Loss

- Given two discrete probability distributions P and Q,
- The KLdivergence is

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

- For target probability **p** and actual probability **q**

$$D_{\mathrm{KL}}(p \parallel q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

Sparsity Loss

- Compute sparsity loss for each neuron in coding layer
- Sum up these losses
- Add result to cost function
- For relative importance of **sparsity loss** & **reconstruction loss**
    - multiply **sparsity loss** by **sparsity weight** hyperparameter

Variational Autoencoders

Variational Autoencoder

- They are **probabilistic autoencoders**
  - Outputs are partly determined by chance, even after training

- They are **generative autoencoders**
  - Can generate new instances that look like they were sampled from the training set

Variational Autoencoder

- Similar to **RBMs**, but
  - Easier to train, and
  - Sampling process is much faster

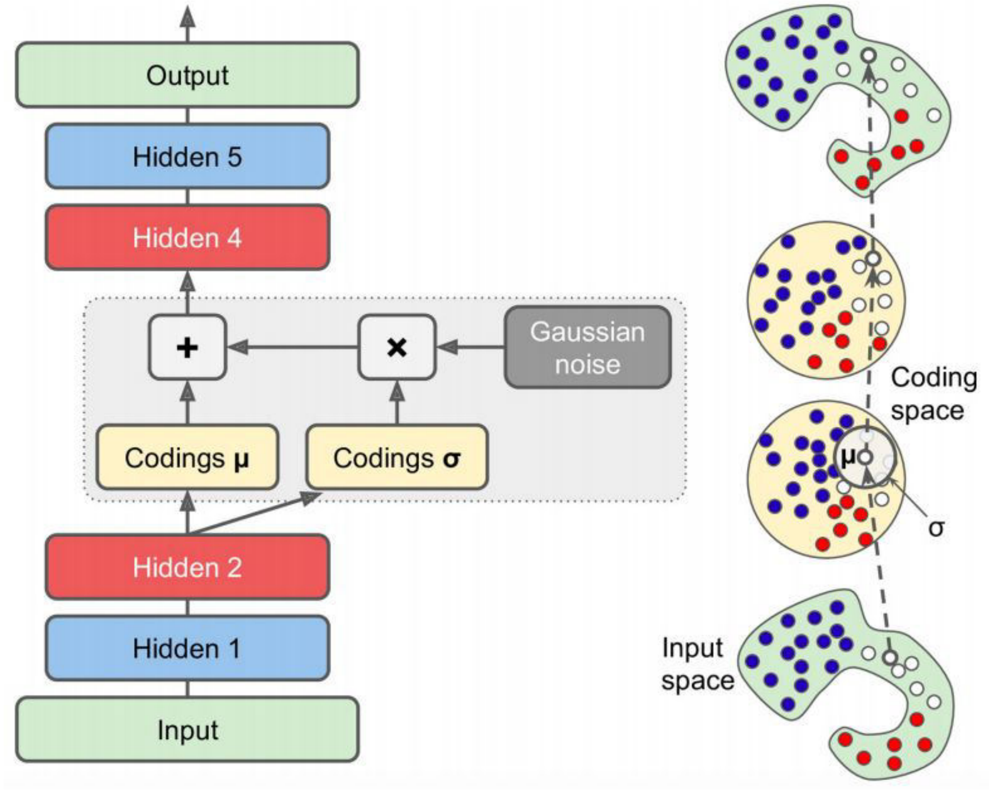
- It performs **Variational Bayesian inference**
  - Efficient way to perform approx. Bayesian inference

Structure of Variational Autoencoder

Structure of Variational Autoencoder

- Encoder & decoder have **2 hidden layers**
- Encoder produces mean coding **μ** & std. deviation **σ**
- Actual coding is **sampled** randomly from Gaussian distribution with mean **μ** and std. deviation **σ**
- Decoder decodes the sampled coding

Variational Autoencoder



Variational Autoencoder

- Inputs have a very convoluted distribution
- But variational autoencoder produce codings that look as though they were sampled from a simple Gaussian distribution

Training a Variational Autoencoder

Training a Variational Autoencoder

- 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

- To generate a new instance:
  - Sample a random coding from the Gaussian distribution
  - Decode it

Cost Function

Cost Function

Cost Function has two parts

- **Reconstruction loss**
  - Pushes autoencoder to reproduce its inputs
- **Latent loss**
  - Pushes autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution

Cost Function

Latent Loss

- KL divergence between 8 target distribution (i.e., Gaussian distribution) & actual distribution of the codings

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

where **L** = latent loss,

n = codings' dimensionality,

$\boldsymbol{\mu}_i$ & $\boldsymbol{\sigma}_i$ = mean & std. dev. of **i**th comp. of codings

Cost Function

Common Tweak to Architecture

- Make encoder output **γ = log(σ)** rather than **σ**

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

This is more numerically stable & speeds up training

Autoencoder - Semantic Interpolication

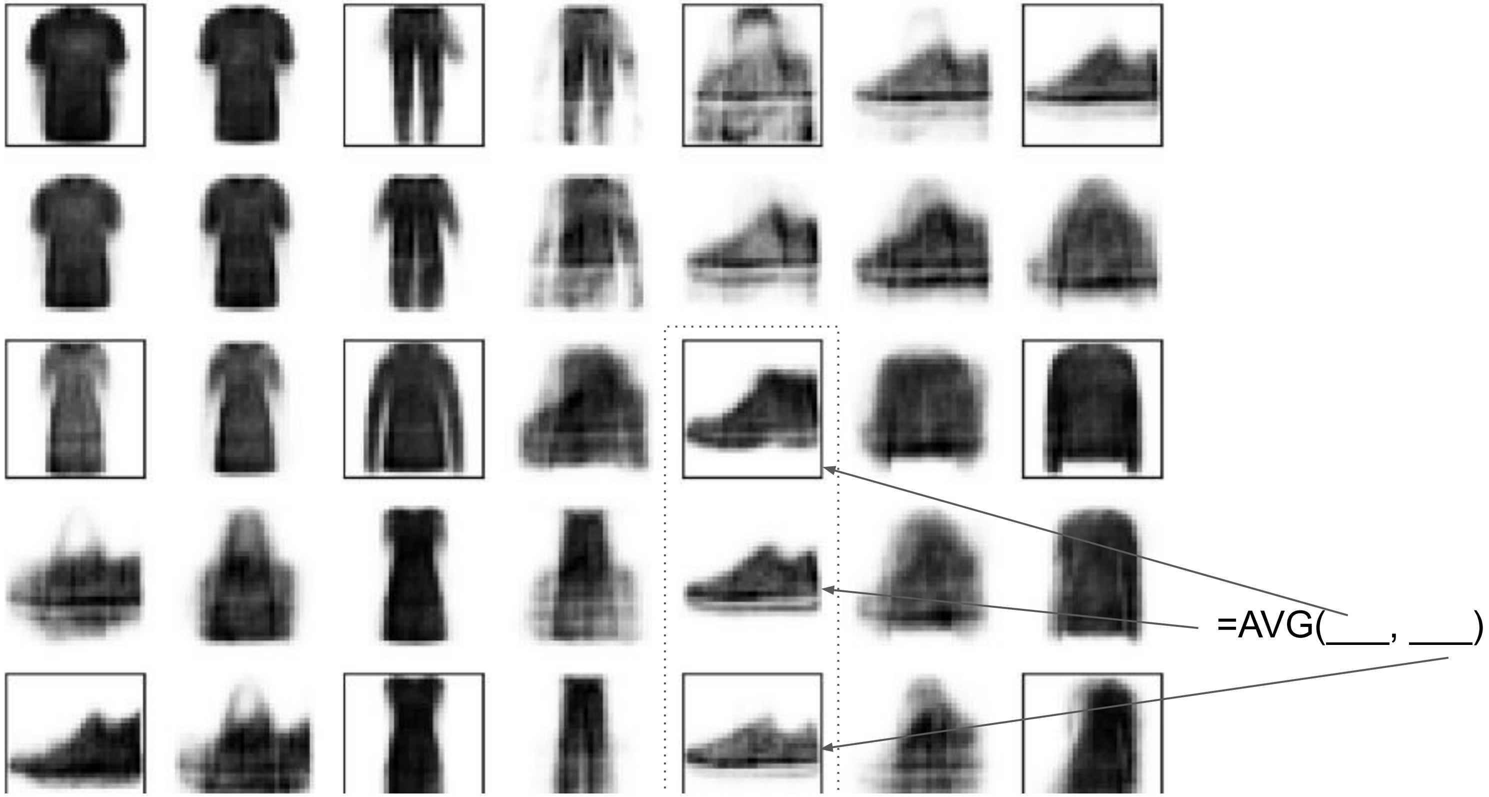
Generating Fashion MNIST Images

- They make it possible to perform **semantic interpolation**
- Instead of interpolating two images at the pixel level (which would look as if the two images were overlaid)
- We can interpolate at the codings level

Autoencoder - Semantic Interpolication

Bordered images are actual.

The images between bordered are interpolation of the bordered images.



Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GANs)

- **Generative adversarial networks** were proposed in a 2014 paper by Ian Goodfellow et al.
- The idea is simple: **make neural networks compete against each other in the hope that this competition will push them to excel**

Uses of GANs

Uses of GANs

- Super resolution (increasing the resolution of an image)
- Colorization
- Powerful image editing (e.g., replacing photobombers with realistic background)
- Turning a simple sketch into a photorealistic image
- Predicting the next frames in a video

Uses of GANs

- 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

Structure of GANs

Structure of GAN

GANs are composed of two neural networks:

- **Generator**
  - Takes a random distribution as input (typically Gaussian)
  - And outputs some data—typically, an image.

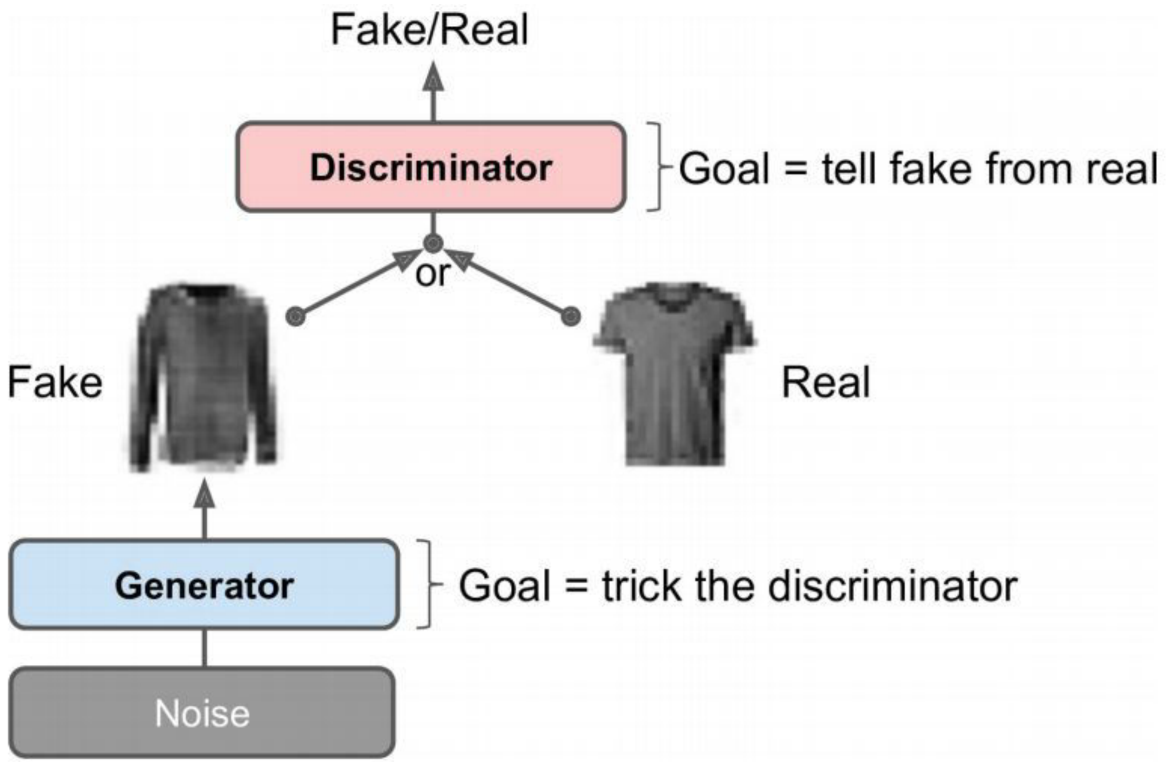
- You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated

Structure of GAN

GANs are composed of two neural networks:

- **Discriminator**
  - Takes either a fake image from **generator**
  - Or a real image from **training set** as input
  - And must guess whether input image is **fake** or **real**

Generative Adversarial Networks (GANs)



Training a GAN

Training a GAN

During training

- Generator and Discriminator have opposite goals
- **Discriminator** tries to tell fake images from real images,
- **Generator** tries to produce images that look real enough to trick the discriminator

- Each training iteration have two phases:

Training a GAN

First phase

- Discriminator is trained
- A batch of **real** images is sampled from **training set**
- Equal number of **fake** images produced by **generator**
- The labels are set: fake images$\rightarrow$ 0 and real images$\rightarrow$ 1

Training a GAN

First phase

- Discriminator is trained on this labeled batch for one step
- **Binary cross-entropy loss** is used

Backpropagation only optimizes the weights of discriminator during this phase.

Training a GAN

Second phase

- Generator is trained
- Another batch of fake images is produced
- Discriminator predicts whether the images are fake or real
- Real images are not added in the batch
- All the labels are set to 1 (real)

Training a GAN

Second phase

- We want the generator to produce images that the discriminator will (wrongly) believe to be real!
- Weights of discriminator are frozen

Backpropagation only optimizes the weights of generator during this phase.

Let’s build a simple GAN for Fashion MNIST

Switch to Notebook

`autoencoders_and_gans.ipynb`

Building Generator and Discriminator

Building Generator and Discriminator

- Generator is similar to an autoencoder’s **decoder**
- Discriminator is a regular **binary classifier**
  - It takes an image as input and ends with a Dense layer containing a single unit and using the sigmoid activation function
- So, naturally use **binary cross-entropy loss**

Training Process

Training process

Phase One

- Gaussian noise is fed to generator to produce fake images
- Equal no. of real images are concatenated to the batch
- The targets y1 are set to 0 for fake images and 1 for real images
- Discriminator is trained on this batch

Training process

Phase One

- Discriminator’s trainable attribute is set True
- To get rid of warning that Keras displays when it notices that trainable is now False but was True when the model was compiled (or vice versa).

Training process

Phase Two

- Gaussian noise is fed to GAN
- Generator will start by producing fake images
- Then discriminator will try to guess whether these images are fake or real

Training process

Phase Two

- We want the discriminator to believe that the fake images are real
- So the targets y2 are set to 1
- We set the trainable attribute to False, once again to avoid a warning.

Difficulties of Training GANs

Difficulties of Training GANs

- During training, the generator and the discriminator constantly try to outsmart each other, in a **zero-sum** game
- The game may end up in a state that game theorists call a **Nash equilibrium**, named after the mathematician **John Nash**

Nash Equilibrium

Nash equilibrium

- When no player would be better off changing their own strategy
- Assuming the other players do not change theirs
- Ex. When everyone drives on the left side of the road
  - No driver would be better off being the only one to switch sides

How does this apply to GANs?

How does this apply to GANs?

- Authors of the paper demonstrated
  - A GAN can only reach a single Nash equilibrium
- Then Generator produces perfectly **realistic** images,
- And Discriminator is forced to **guess** (50% real, 50% fake)
- **But nothing guarantees that the equilibrium will ever be reached**

Difficulties of Training GANs

Mode Collapse

- The generator’s outputs gradually become less diverse

How can Mode Collapse happen?

How can Mode Collapse happen?

- Suppose, generator produces more **convincing** shoes than other classes
- It will **fool** the discriminator a bit more with shoes
- So it will produce even more images of shoes

How can Mode Collapse happen?

- Gradually, Generator will **forget** how to produce anything else
- Only fake images that discriminator sees will be shoes
- So, it will also forget how to discriminate fake images of other classes

How can Mode Collapse happen?

- Eventually, when discriminator manages to discriminate fake shoes from real ones
- Generator will be forced to move to another class
- It may then become good at shirts, forgetting about shoes, and the discriminator will follow.
- GAN may gradually cycle across a few classes
- It never becomes very good at any of them

How can Mode Collapse happen?

- Because generator & discriminator are constantly pushing against each other
- Their parameters may oscillate & become unstable
- Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities
- GANs are very sensitive to the hyperparameters

Few techniques to stabilize training

Few techniques to stabilize training

Experience replay

- Store images produced by generator at each iteration in a **replay buffer**
- Gradually **drop** older generated images
- Train discriminator using real images & fake images drawn from this buffer
- Rather than just fake images produced by current generator
- This reduces the chances that discriminator will **overfit** latest generator’s outputs

Few techniques to stabilize training

Mini-batch discrimination

- Measure how similar images are across the batch
- Provide this **statistic** to the discriminator
- **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

Deep Convolutional GANs

Deep Convolutional GANs

- Training GANs based on deeper convolutional nets for larger images was very unstable
- Alec Radford et al. finally succeeded in late 2015
- This architecture is called **Deep Convolutional GANs**

Guidelines for stable convolutional GANs

Guidelines for stable convolutional GANs

- Replace any **pooling** layers with
  - **Strided convolutions** (in the discriminator) and
  - **Transposed convolutions** (in the generator).
- Use Batch Normalization in both generator & discriminator
- Except in generator’s output layer & discriminator’s input layer

Guidelines for stable convolutional GANs

- 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

Deep Convolutional GANs

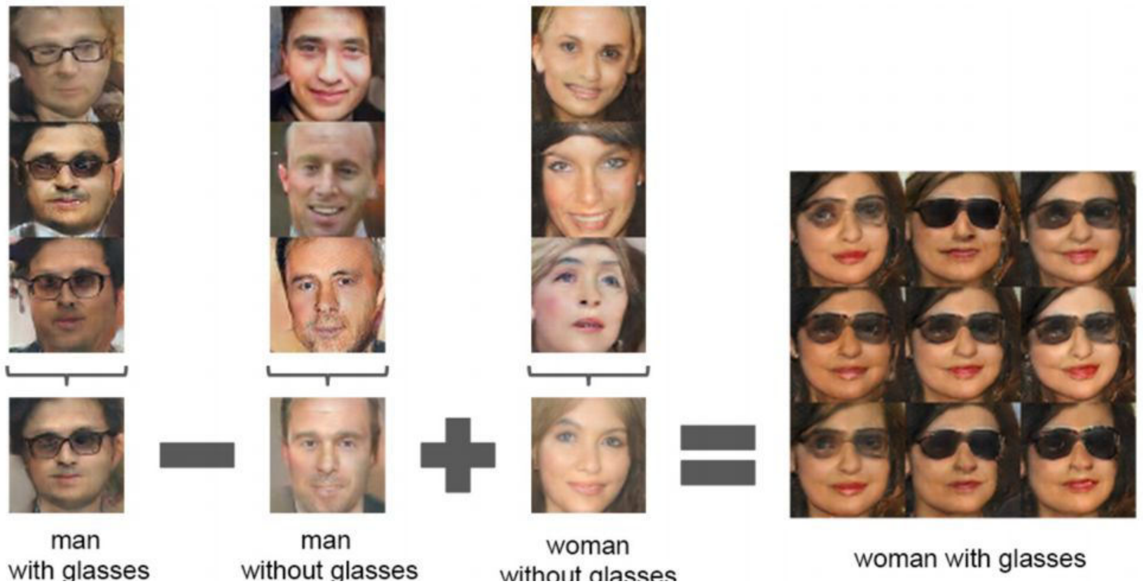
- DCGANs can learn quite meaningful latent representations
- Many images were generated, and nine of them were picked manually
- For each of these categories, codings that were used to generate images were averaged
- And image was generated based on the resulting mean codings

Deep Convolutional GANs

- This is not a **simple mean** computed at the pixel level
- This would result in three overlapping faces
- It is a **mean** computed in the **latent space**
- So the images still look like normal faces.

Deep Convolutional GANs

- Men with glasses – minus men without glasses + plus women without glasses =



Switch to Notebook

`autoencoders_and_gans.ipynb`

Progressive Growing of GANs

Progressive Growing of GANs

- Progressive Growing of GANs was proposed in 2018 by Nvidia researchers Tero Karras et al.
- They generated small images at the beginning of training
- Then gradually added convolutional layers to both generator & discriminator
- And produced larger & larger images
- (4×4, 8×8, 16×16, ..., 512×512, $1{,}024 \times 1{,}024$)

Progressive Growing of GANs

- This resembles greedy layer-wise training of stacked autoencoders
- Extra layers get added at the end of generator and beginning of discriminator
- Previously trained layers remain trainable

Let’s see an example

Progressive Growing of GANs

- When growing the generator’s outputs from 4 × 4 to 8 × 8
- An upsampling layer (using nearest neighbor filtering) is added to existing convolutional layer
- So it outputs 8 × 8 feature maps
- It is fed to new convolutional layer (which uses "same" padding and strides of 1, so its outputs are also 8 × 8)

Progressive Growing of GANs

- 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 desired no. of color channels
- To avoid breaking the trained weights of the first convolutional layer when new convolutional layer is added
- The final output is a weighted sum of original output layer (which now outputs 8 × 8 feature maps) & new output layer

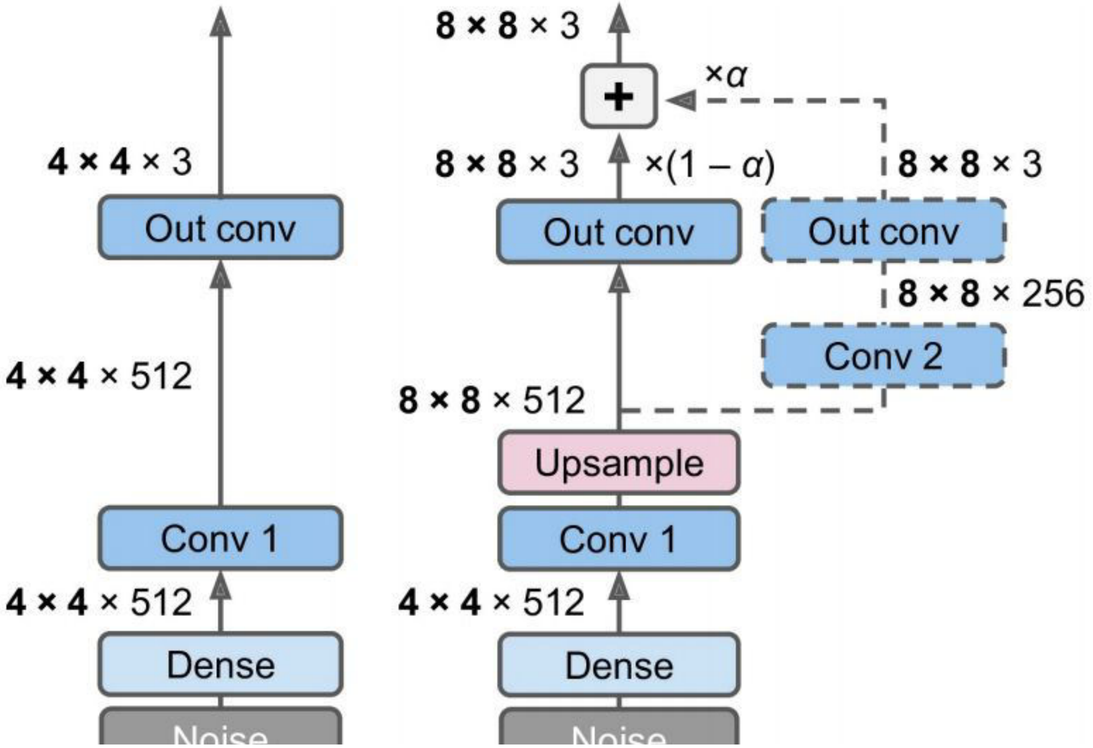
Progressive Growing of GANs

- Weight of new outputs is α, while weight of original outputs is 1 – α
- α is slowly increased from 0 to 1
- New convolutional layers are gradually faded in
- Original output layer is gradually faded out
- Similar fade-in/fade-out technique is used when a new convolutional layer is added to discriminator (followed by an average pooling layer for downsampling).

Progressive Growing of GANs

Architecture

Progressive Growing of GANs Architecture



Progressive Growing of GANs

- The paper 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
- Equalized learning rate
- Pixelwise normalization layer

Minibatch standard deviation layer

Minibatch standard deviation layer

- Added near the end of discriminator
- For each position in inputs, it computes std. deviation across all channels and all instances in batch
- **S = tf.math.reduce_std(inputs, axis=[0, -1])**

- These std deviations are then averaged across all points to get a single value
- **v = tf.reduce_mean(S)**

Minibatch standard deviation layer

- Finally, an extra feature map is added to each instance in batch
- And filled with computed value
- **tf.concat([inputs, tf.fill([batch_size, height, width, 1], v)], axis=-1)**

How does this help?

Minibatch standard deviation layer

- If generator produces images with little variety
- There will be small std. deviation across feature maps in discriminator
- Now discriminator will have easy access to this statistic, making it less likely to be fooled by a generator that produces too little diversity.
- This encourage generator to produce more diverse outputs, reducing risk of mode collapse.

Equalized learning rate

Equalized learning rate

- Initializes all weights using a simple Gaussian distribution with mean 0 & std. deviation 1 rather than using He initialization
- However, weights are scaled down at runtime by same factor as in He initialization
- They are divided by $\sqrt{2/n_{\text{inputs}}}$, where $\mathbf{n}_{\textbf{inputs}}$ is the no. of inputs to layer
- It improved GAN's performance when using **RMSProp**, **Adam**, or other **adaptive gradient optimizers**

Equalized learning rate

- Indeed, these optimizers normalize 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

Equalized learning rate

- By rescaling weights as part of model itself rather than just rescaling them upon initialization
- This approach ensures that dynamic range is the same for all parameters, throughout training
- So they all learn at same speed
- This both speeds up and stabilizes training

Pixelwise normalization layer

Pixelwise normalization layer

- Added after each convolutional layer in generator
- It normalizes each activation based on all activations in same image & at same location
- But across all channels (dividing by square root of mean squared activation)
- **inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)**
- This technique avoids explosions in activations due to excessive competition between generator & discriminator.

Evaluation of convincing

Evaluation of convincing

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

- But what exactly do we call "**convincing**"?
- Evaluation is one of the big challenges when working with GANs

Evaluation of convincing

- It is possible to automatically evaluate diversity of generated images
- But judging their quality is a much trickier and subjective task
- One technique is to use human raters, but this is costly and time-consuming

Evaluation of convincing

- So authors proposed to measure similarity between local image structure of generated images & training images, considering every scale

- This idea led them to another groundbreaking innovation: **StyleGANs**

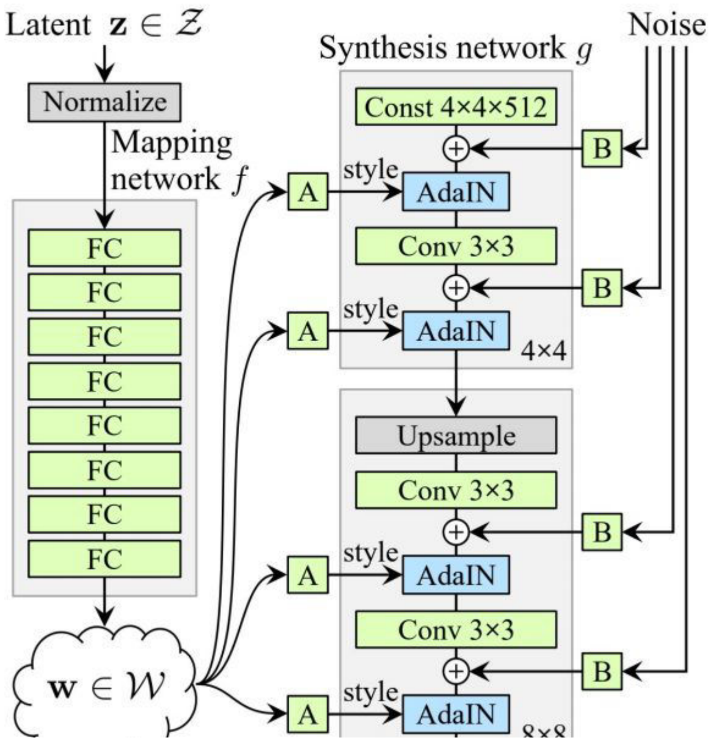
StyleGANs

StyleGANs

- StyleGAN architecture was introduced in 2018
- **Style transfer** techniques are used in the generator
- They ensure that generated images have same local structure as the training images at every scale
- This greatly improving the quality of the generated images.
- The discriminator and the loss function were not modified, only the generator

StyleGAN Architecture

StyleGAN Architecture



StyleGAN Architecture

Mapping network

- An eight-layer MLP
  - Maps the codings **z** to a vector **w**
- This vector is then sent through multiple **affine transformations**
- They are Dense layers with no activation functions
- This produces multiple vectors.

StyleGAN Architecture

Mapping network

- 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)
- The mapping network maps codings to multiple style vectors.

StyleGAN Architecture

Synthesis network

- Responsible for generating the images
- It has a constant learned input
- During training it is tweaked by backpropagation
- It processes this input through multiple convolutional and upsampling layers, as earlier

StyleGAN Architecture

Synthesis network

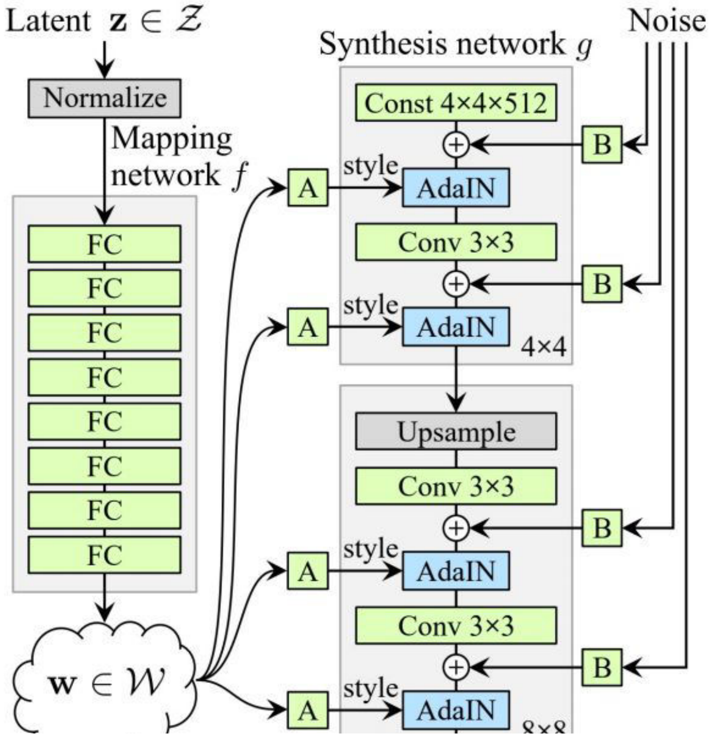
- First, some noise is added to the input and to all the outputs of the convolutional layers (before the activation function)
- Second, each noise layer is followed by an **Adaptive Instance Normalization** (AdaIN) layer
- It **standardizes** each feature map independently

StyleGAN Architecture

Synthesis network

- Then it uses the style vector to determine the scale & offset of each feature map
- Style vector contains one scale & one bias term for each feature map

StyleGAN Architecture



StyleGAN Architecture

- **Noise is added independent of codings**
- Some parts of an image are quite random, such as the exact position of each freckle or hair
- In earlier GANs, randomness came from
  - **Codings** or
  - **pseudorandom noise** produced by generator

StyleGAN Architecture

If noise came from codings

- Generator have to dedicate representational power
- This is quite wasteful
- Moreover, noise has to flow through the network and reach final layers
- This seems unnecessary constraint
- It probably slowed down training

StyleGAN Architecture

If generator produces its own pseudorandom noise

- This noise might not look very convincing
- This leads to more visual artifacts
- Part of the generator’s weights would be dedicated
- This also seems wasteful

StyleGAN Architecture

- By adding extra noise inputs, all these issues are avoided
- The GAN is able to add the right amount of **stochasticity** to each part of the image.

Mixing Regularization

Mixing Regularization

- StyleGAN uses **mixing regularization** (or style mixing)
- Percentage of generated images are produced using two different codings
- Codings $\mathbf{c}_1$ and $\mathbf{c}_2$ are sent through mapping network
- It gives two style vectors $\mathbf{w}_1$ and $\mathbf{w}_2$
- Then synthesis network generates an image based on
  - Styles $\mathbf{w}_1$ for the first levels and
  - Styles $\mathbf{w}_2$ for the remaining levels.

Mixing Regularization

- The cutoff level is picked randomly
- This prevents network from assuming that styles at adjacent levels are correlated
- Which encourages locality in the GAN
- So each style vector only affects a limited number of traits in the generated image

SEE MORE

- https://medium.com/@jonathan_hui/gan-stylegan-stylegan2-479bdf256299
- https://arxiv.org/abs/1812.04948

Questions?