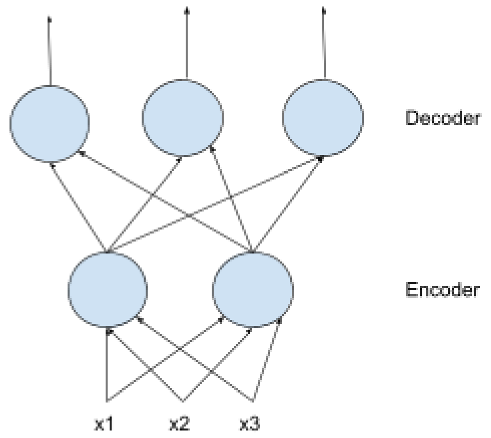
Autoencoders, GANs, and Diffusion Models

Introduction

- Autoencoders are artificial neural networks that can learn dense representations of input data, called latent representations or codings, unsupervised.
- Autoencoders can be used for feature detection, dimensionality reduction, and unsupervised pre-training.
- Some autoencoders are generative models: they can randomly generate new data that looks similar to the training data.
- Generative adversarial networks (GANs) can generate data convincingly identical to the training data.
- Lastly, diffusion models can generate more diverse and higher-quality images than GANs but run more slowly.
- These are all unsupervised learning models, but they work differently.
    - Autoencoders learn to copy their inputs to their outputs. They are forced to learn efficient ways of representing data.
    - GANs are composed of two neural networks: a generator that tries to generate data similar to the training data and a discriminator that tries to tell real data from fake data.
    - Diffusion models are trained to remove noise from images.

Efficient Data Representations

- Models cannot memorize long data sequences, so they must identify patterns that help them store information efficiently.
- An autoencoder has an encoder that converts inputs to latent representations and a decoder that converts the internal representations to the outputs.

- 

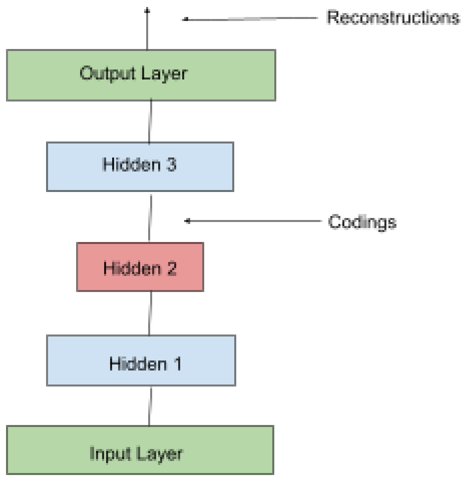
- The outputs are called reconstructions because the autoencoder tries to reconstruct the inputs.
  - The cost function contains a reconstruction loss that penalizes the model when the reconstructions differ from the inputs.
- Since the internal representation has a lower dimensionality than the inputs (2D instead of 3D), the autoencoder is undercomplete.

Performing PCA with an Undercomplete Linear Autoencoder

- If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), it performs principal component analysis.
  - The encoder and decoder have their layers, and when we build the autoencoder, we can simply use a Sequential() model with the encoder and decoder as inputs.
  - The autoencoder's number of outputs equals its number of inputs.
  - We use the autoencoder's fit() method and the encoder's predict() method.

Stacked Autoencoders

- A stacked autoencoder contains several layers, allowing it to learn complex codings.
  - We should not make the autoencoder too powerful, or it may reconstruct the training data perfectly and won't generalize well to new instances.
  - The architecture of a stacked autoencoder is typically symmetrical about the central hidden layer (the coding layer).



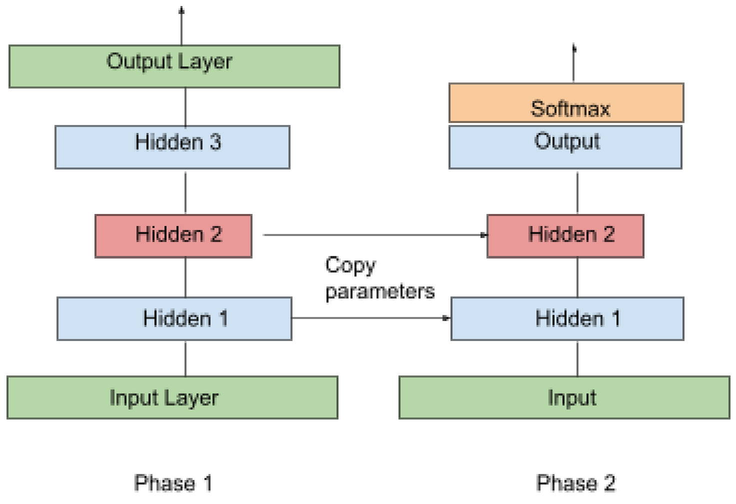
■

- We can implement a stacked autoencoder in Kera like an MLP.
  - Stacked autoencoders typically use the MSE loss and Nadam optimization.
- We can compare an autoencoder's inputs and outputs (e.g., plotting them) to ensure it was trained properly.

Autoencoders for Dimensionality Reduction

- Autoencoders can handle large datasets with many instances and features.
- Thus, we can use autoencoders to reduce the dataset’s dimensionality and then use another dimensionality reduction algorithm for visualization.
  - For example, we use the encoder’s predict() method to get the compressed dataset. Then, we use Scikit-Learn’s **TSNE()** class for further reduction.

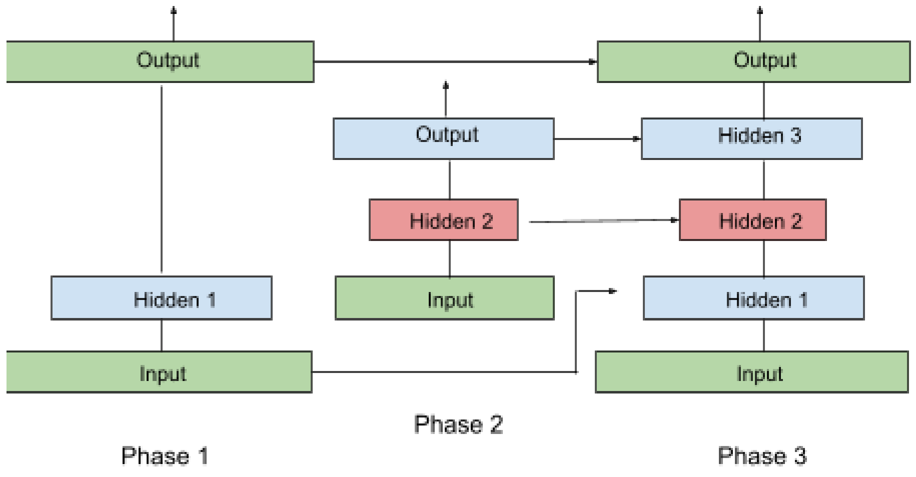
Unsupervised Pretraining Using Stacked Autoencoders

- We can use a stacked autoencoder for unsupervised learning.
  - We can train a stacked autoencoder using all the data if we have a largely unlabeled dataset.
  - Then, we reuse the lower layers to create a neural network for the actual task and train it using the labeled data.
  - 

Tying Weights

- When an autoencoder is symmetrical, we can tie the weights of the decoder layers to the weights of the encoder layers.
  - This technique halves the number of weights in the model, speeding up training and limiting the risk of overfitting.
  - An autoencoder has N layers (excluding the input layer) and W_L, representing the L^{th} layer's connection weights.
    - Layer 1 is the first hidden layer, layer N/2 is the coding layer, and layer N is the output layer.
    - The decoder layer weights are $W_L = W_{N-L+1(T)}$.
  - To implement this in Keras, we can build a custom Dense layer that uses another Dense layer's weights. Then, we can build a Sequential model as normal.

Training One Encoder at a Time

- We can train one shallow encoder at a time and then stack all of them into a single stacked autoencoder.
  - This is also known as greedy layerwise training. This training method is used less.
  - The first autoencoder learns to reconstruct the inputs during the first training phase.
  - Then, we encode the whole training set using this first autoencoder, giving 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 sandwich using all these autoencoders.
  - 

Conventional Autoencoders

- Convolutional neural networks work better than dense networks when handling images.
- We can build an autoencoder for images.

    - The encoder is a regular CNN composed of convolutional and pooling layers. It reduces the inputs' spatial dimensionality while increasing the depth.
    - The decoder must work in reverse: upscale the image and reduce the depth.
        - We can use Keras's **Conv2DTranspose()** layer for this task.
- We can use many other constraints, including ones that allow the coding layer to be as large as the inputs or even larger. This is an overcomplete autoencoder.

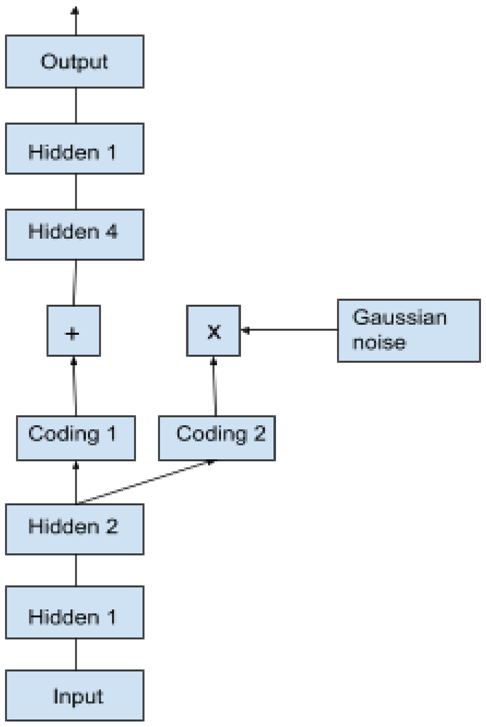
Denoising Autoencoders

- 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, or they can be dropped out.
    - Thus, a denoising autoencoder is a regularly stacked autoencoder with an additional Dropout() layer or GaussianNoise() layer.
    - These autoencoders can efficiently remove noise from images.

Sparse Autoencoders

- Another constraint is sparsity. We add an appropriate term to the cost function, forcing the autoencoder to reduce the number of active neurons in the coding layer.
    - Thus, each neuron in the coding layer typically represents a useful feature.
- We can use a sigmoid function in the coding layer to constrain the codings to values between 0 and 1.
    - The coding layer is large (300 neurons), and we can add ℓ1 regularization to its activations.
    - To add the regularization, we can use Keras's **ActivityRegularization()** layer or set the previous layer's *activity_regularizer* argument to **tf.keras.regularizers.l1()**.
    - Using the ℓ1 norm rather than the ℓ2 norm will push the neural network to preserve the most important codings while eliminating the ones not needed for the input image.
- Another approach is to measure the actual sparsity of the coding layer at each training iteration and penalize the model when the measured sparsity differs from a target sparsity.
    - We compute the mean activation per neuron and penalize neurons that are too active or not active enough by adding a sparsity loss to the cost function.
    - We can use the Kullback_Leibler Divergence equation as the sparsity loss.
        - $D_{KL}(p||q) = p\,log\frac{p}{q} + (1-p)log\frac{1-p}{1-q}$
        - $D_{KL}(p||q)$ is the divergence between two discrete probability distributions p, the probability that a neuron will activate, and q, the actual probability.
    - We can use Keras's **losses.kullback_leibler_divergence** for the loss function. We can create a custom class for the KL divergence regularizer.

Variational Autoencoders

- The variational autoencoder (VAE) is among the most popular variants.
  - VAEs are probabilistic encoders, meaning their outputs are partly determined by chance during and after training.
  - They are generative encoders, generating new instances similar to the training ones.
  - The encoder produces two codings instead of one: a mean coding $\boldsymbol{\mu}$ and a standard deviation $\boldsymbol{\sigma}$.
  - Then, each coding is sampled randomly from a Gaussian distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$.
  - The decoder decodes the sampled coding normally.
  - 

- The latent loss pushes the codings to the latent space to look like a cloud of Gaussian points.
  - This limits the information transmitted to the coding layer, encouraging the autoencoder to learn useful features.
  - The latent loss equation is $\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{n} 1 + \gamma_i - exp(\gamma_i) - \mu_i^2$ where
    - $\mathcal{L}$ is the latent loss, n is the codings' dimensionality, μ_i is the mean of the ith component, and $\gamma = log(\sigma^2)$.
  - We can make a custom function for the sampling. Since it's not entirely sequential, we can use the functional API (e.g., **tf.keras.Model()**) for the encoder and the Sequential() model for the decoder.
  - To generate images, we feed noise to the decoder.

Generative Adversarial Networks

- Adversarial learning tries to trick the model by providing deceitful input to improve the model's prediction accuracy.
- GANs use this concept through two neural networks.
    - The generator outputs new images with the random distribution input it was given.
    - The discriminator must guess whether the input is real data or fake.
- We train the discriminator in the first phase of each training iteration.
    - The training set feeds real data into the discriminator while the generator feeds fake data.
    - The discriminator outputs a zero for fake data and one for real data. This also means that it uses binary cross-entropy loss.
    - Backpropagation only optimizes the weights of the discriminator during this phase.
- We train the generator in the second phase.
    - First, we use the generator to produce fake data and feed it to the discriminator.
    - The generator never sees real data; it learns to produce convincing fake data. We want the generator to produce data that the discriminator will wrongly believe to be real.
    - The discriminator's weights are frozen during this step. Backpropagation only affects the weights of the generator.
- A GAN's architecture is similar to a regular encoder's.
    - The generator is similar to an autoencoder's decoder. The discriminator is a binary classifier using a Dense() layer with a sigmoid function.
- Training the GAN is tricky.
    - First, we compile the discriminator only. After compilation, we make it untrainable. Then, we compile the GAN.
    - We cannot call the GAN's fit() method since the discriminator is not trainable during this phase. We will need a custom loop that splits training into two phases.
- To randomly sample codings, we use the generator's predict() method.

The Difficulties of Training GANs

- Ideally, the GAN would reach Nash equilibrium during training.
    - This equilibrium is when no player would be better off changing their strategy, assuming the other players do not.
    - The GAN reaches this equilibrium when the generator produces perfectly realistic data, and the discriminator must guess (50/50). Reaching this equilibrium is difficult for GANs.
- GANs suffer from mode collapse when the generator's outputs become less diverse.
    - This occurs when the generator gets good at producing data from one class.
    - It will continue producing data from this class since it is likely to fool the discriminator.
    - Eventually, the generator will forget how to produce data from other classes. The discriminator will forget how to discriminate fake data of other classes.

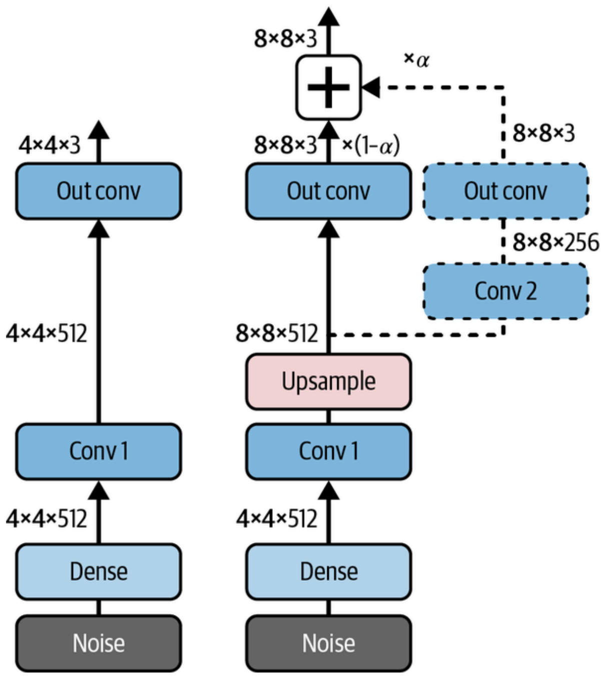
- Since the generator and discrimination are constantly pushing against each other, their parameters may oscillate and become unstable.
    - GANs are very sensitive to hyperparameters; fine-tuning those parameters is costly.
    - When compiling the models, we may need to change the optimization (e.g., from Nadam to RMSProp).
- One possible solution for these problems is experience replay.
    - We store the data produced by the generator at each iteration in a replay buffer, gradually dropping older generated data.
    - We train the discriminator using real data from the training set and fake data from the buffer.
- Another possible solution is mini-batch discrimination.
    - This measures how similar images are across the batch and gives this statistic to the discriminator.
    - The discriminator can reject a batch of fake data that lacks diversity.

Deep Convolutional GANs

- Deep convolutional GANs (DCGANs) are among the most successful GAN architectures.
- DCGANs have some guidelines.
    - We replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
    - We use batch normalization or dropout in the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
    - We remove fully connected hidden layers for deeper architectures.
    - We use ReLU activation in the generator for all layers except the output layer, which should use tanh.
    - We use leaky ReLU activation in the discriminator for all layers.

Progressive Growing of GANs

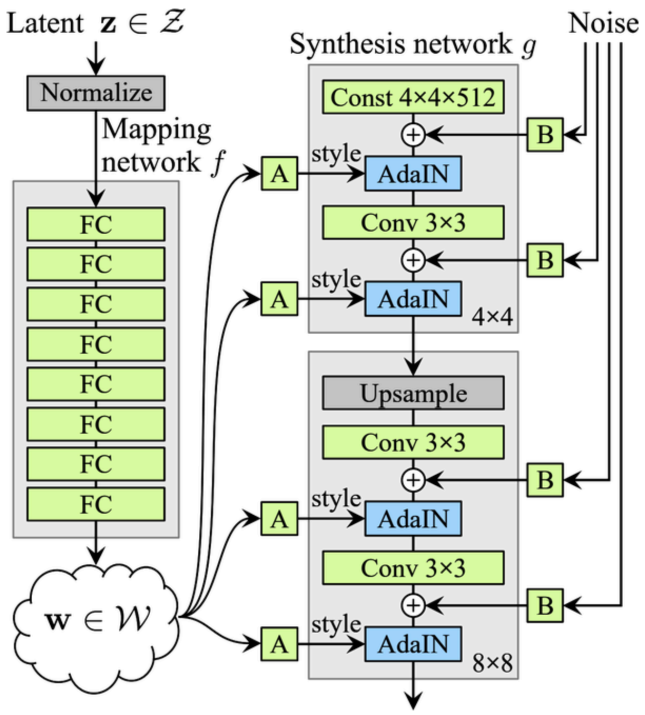
- Some important GAN techniques have been developed.
    - Tero Kerras et al. suggested generating small images at the beginning of training, then gradually adding convolutional layers to both the generator and the discriminator to produce larger and larger images.

■ 

- ○ Mini-batch standard deviation layer: uses standard deviation as a measure that the discriminator can use to distinguish between real and fake data.
- ○ Equalized learning rate: rescales the weights, ensuring 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.
- ○ Pixelwise normalization layer: normalizes each activation based on all the activations in the same image and at the same location across all channels. This avoids explosions in the activations due to excessive competition between the generator and discriminator.

StyleGANs

- The StyleGAN is a popular GAN architecture introduced in 2018.
  - It uses style transfer techniques 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.
- A StyleGAN generator is composed of two networks.
  - Mapping network: maps the codings z to a vector w, sent through multiple affine transformations (Dense layers without activation functions), producing style vectors.
  - Synthesis network: generates images by adding noise to the input and all the outputs of the convolutional layers. Each noise layer is followed by an adaptive instance normalization (AdaIN) layer, standardizing each feature map independently.

- 

    - The A's represent the affine transformations. The B's are noise layers.

- StyleGAN uses mixing regularization where two different codings produce a percentage of the generated images.
    - This ensures that each style vector only affects a limited number of traits in the generated image.

Diffusion Models

- Diffusion models learn the reverse process of a denoising autoencoder: The goal is to start from a mixed state and gradually unmix it.
- Denoising diffusion probabilistic models (DDPMs) are easier to train than GANs but run more slowly.
    - If we have an image x0, at each time step t, we add some noise to the image.
        - The Gaussian noise has mean 0 and variance □t. The noise is independent for each pixel; it is isotropic.
    - We obtain the image x1, x2, and so on until the noise completely hides the image. The last time step is noted T.
    - At each step, the pixel values are rescaled by a factor of $\sqrt{1 - \beta_t}$, ensuring that the mean of the pixel values gradually approach zero. It also ensures that the variance converges to one.
    - The equation for the forward diffusion process is $q(x_t | x_0) = N(\sqrt{\alpha_t} x_0, (1 - \alpha_t) I)$ where q is the probability distribution.
    - The reverse process involves removing some noise from the image at each time step until the noise is gone.

    - We will likely have to make custom functions that prepare the data in batches, adding noise gradually (and in reverse as well).
    - Once we have those functions, we can use any model as a diffusion model.