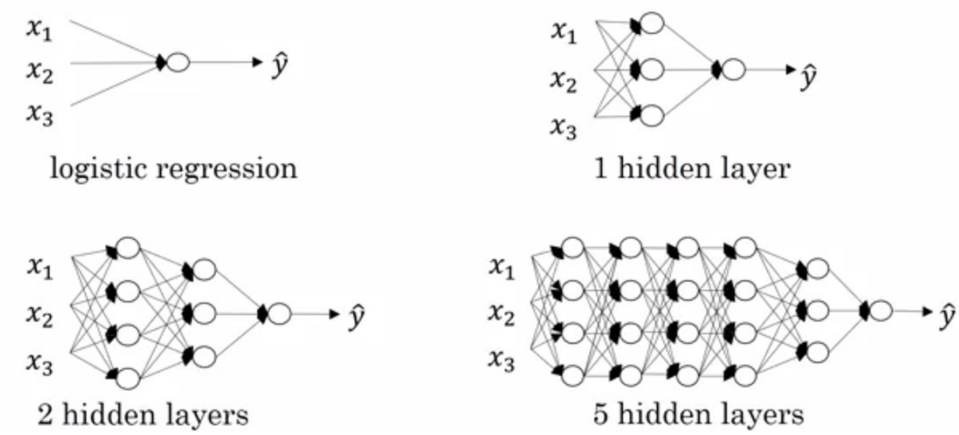
Introduction to Machine Learning

Fun with DNNs

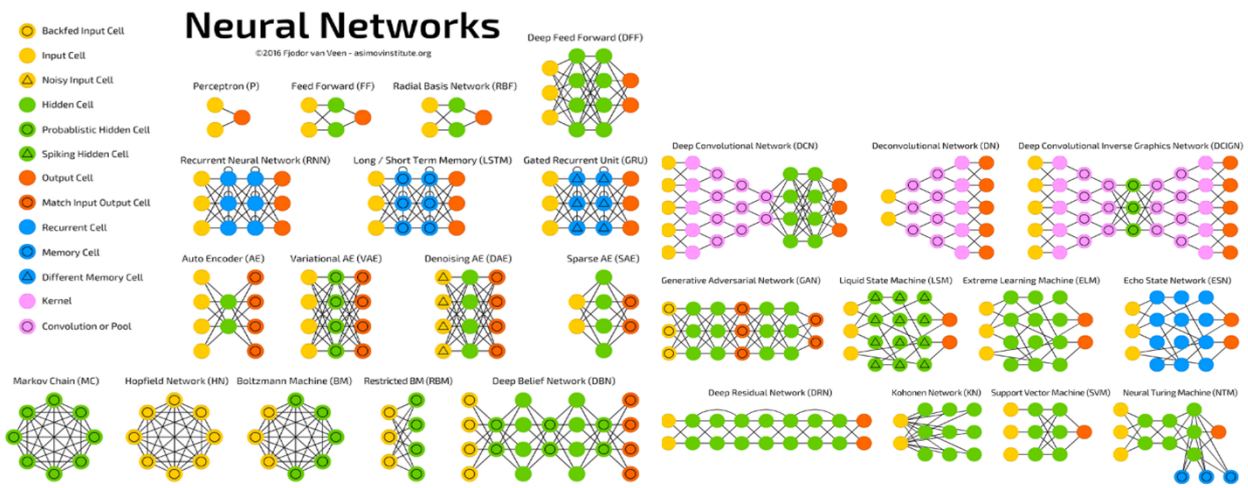
RNNs, Autoencoders, GANs

Deep neural networks



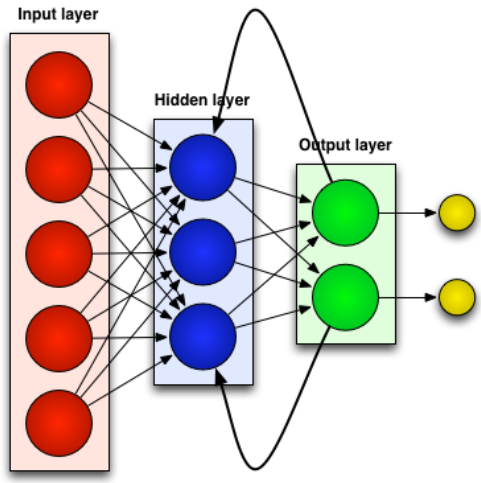
Logistic regression ("shallow")
1 hidden layer ("2 layer" NN)
2 hidden layers
5 hidden layers ("deep")
shallow vs. deep is relative

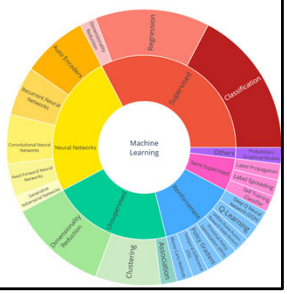
Zoo of neural network types



https://www.asimovinstitute.org/neural-network-zoo/

Recurrent neural networks





Networks that remember and use the past

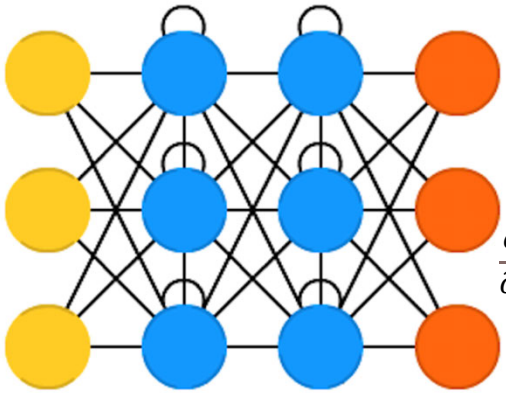
Output from one time step can feed back into the network (input or hidden) in the next time step

Can even add units to remember two steps back, three steps back, any fixed number of steps

hidden unit is $f(x_t, x_{t-1}, x_{t-2}, h_{t-1}, h_{t-2}, o_{t-1}, o_{t-2}, \ldots)$

In this example, h_1 and h_3 are functions of $x_{t,1}, x_{t,2}, x_{t,3}, x_{t,4}, x_{t,5}, o_{t-1,1}, o_{t-1,2}$

Training RNNs



$$L(\hat{y}, y) = \sum_{t=1}^{T} L_t(\hat{y}_t, y_t)$$

$$\frac{\partial L}{\partial w} = \sum_{t}^{T} \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} \frac{\partial h_k}{\partial w}$$

Train with backpropagation through time, BPTT

1. Present sequence to network, k1 timesteps long, of input and output pairs
2. Unroll network and calculate/accumulate errors for k2 timesteps (e.g., f(x_t1, x_t2, .., o_t))
3. Roll up network and update weights
4. Repeat

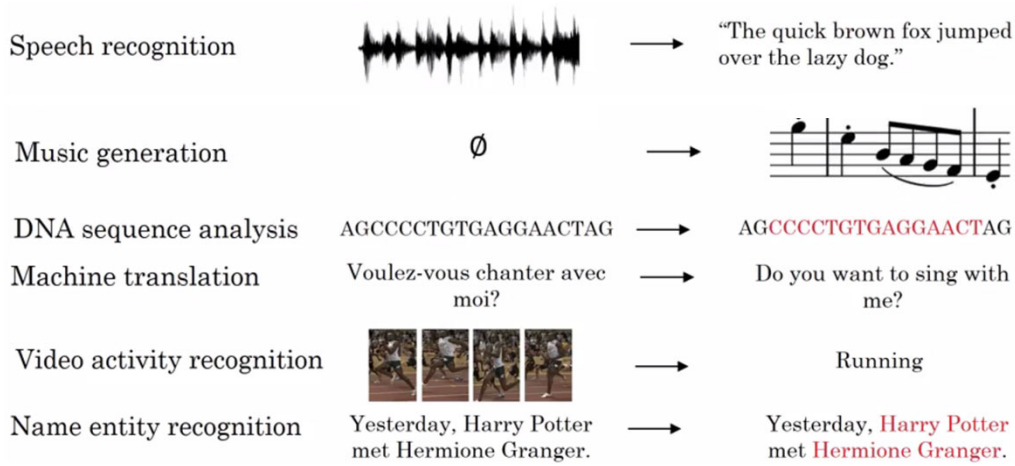
First, compute loss function, first equation
The weights W are shared across the entire time sequence.
Differentiate w.r.t. W at each time step and sum, second equation

d(L w.r.t. output)*d(output wrt hidden values)*
d(hidden values at t+1 wrt hidden values at t)*
d(hidden values at the beginning wrt weights)

$\partial h_{t+1} \partial h_t$ is a chain rule of its own $\partial h_{t+1} \partial h_t = \prod_{i=k+1}^{t} \partial h_{t+1} \partial h_t$
causes vanishing or exploding gradients

Recurrent neural networks

- Sequence models



[Ng 2018]

Given input audio clip x, map to text transcript y.

Both input and output are sequences

Music generation

input could be empty or an integer representation genre

output is a sequence of pitches and durations

Given your DNA sequence of {A,G,C,T}, map a subsequence onto a protein

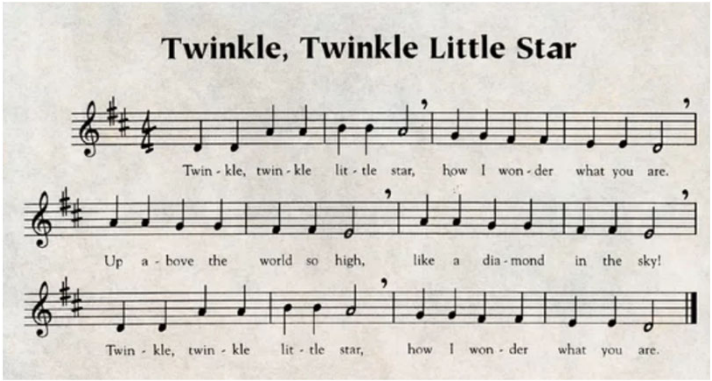
Given phrase in one language, map onto phrase in another language

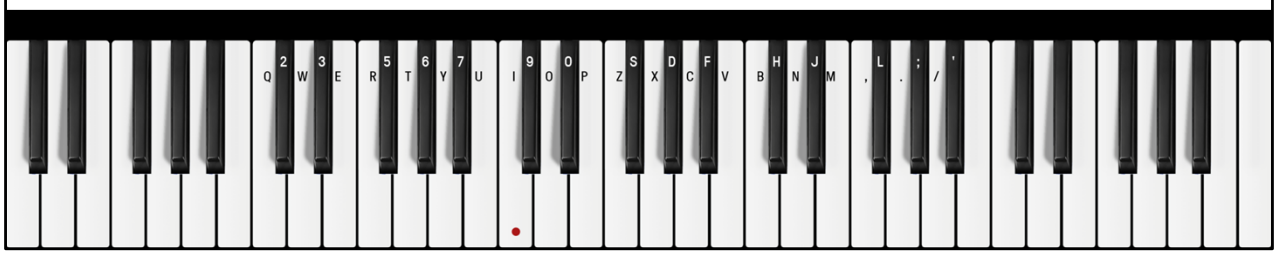
Given video frame sequence, recognize activity

Given sentence, recognize people mentioned in sentence

Given sequence of notes, predict next note

Example

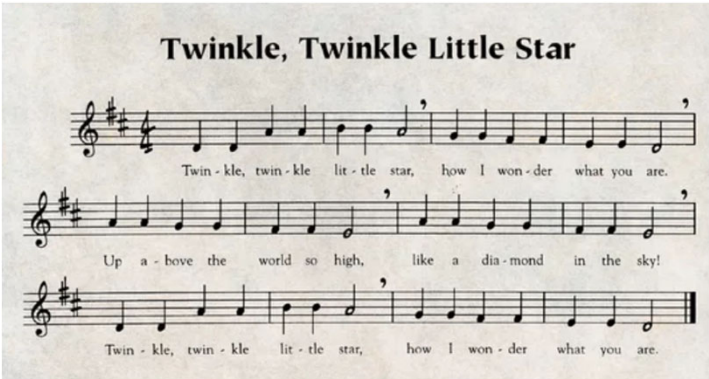




https://www.onlinepianist.com/virtual-piano
Google Drive, ML, Lectures, twinkle.csv

1. Import modules, plot data
2. Reshape data so each data point has 6 notes in, predict note 7, time_step = 6 here

Input		Target	
x0,x1,x2,x3,x4,x5	->	x6	
x6,x7,x8,x9,x10,x11	->	x12	…



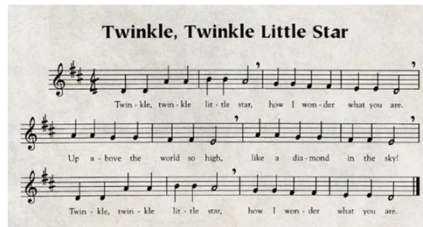
Sequence	Note
1	0
2	0
3	7
4	7
5	9
6	9
7	7
8	7
9	5
10	5
11	4
12	4
13	2
14	2
15	0
16	0

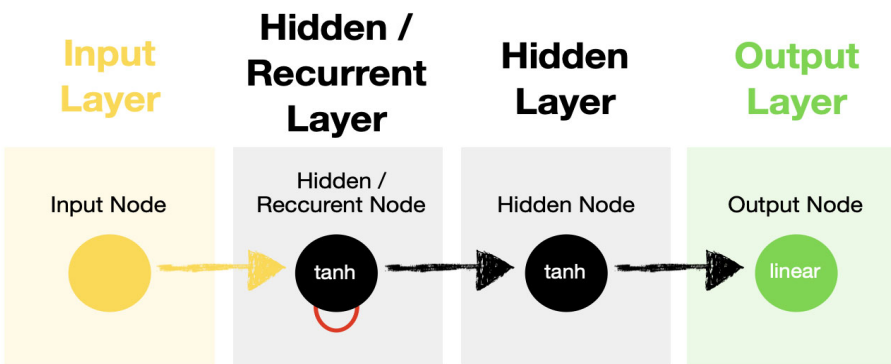
https://www.onlinepianist.com/virtual-piano
Google Drive, ML, Lectures, twinkle.csv

1. Import modules, plot data
2. Reshape data so each data point has 6 notes in, predict note 7, time_step = 6 here

Input		Target	
x0,x1,x2,x3,x4,x5	->	x6	
x6,x7,x8,x9,x10,x11	->	x12	…

Example





3. NN structure, 4 layers, only one node in each layer

Last parameter in each layer is bias, rest are weights

First hidden layer has 3 values (x weight, h_{t-1} weight, bias)

Second hidden layer has 2 values (weight from previous hidden, bias)

Play with different time steps and epochs

SimpleRNN, output is fed back to input

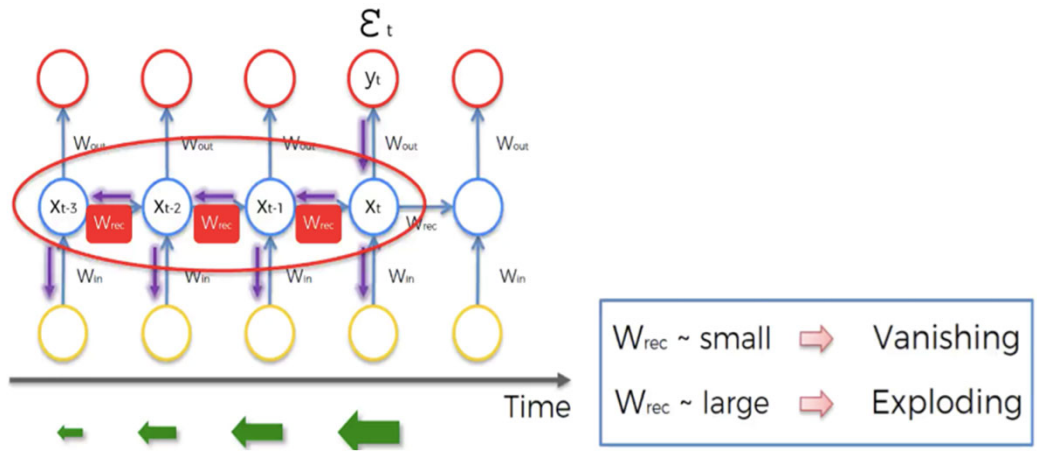
Let's try this out

twinkle.py

Compare 1D CNN, RNN, LSTM

One data point consists of 6 notes, predict note 7

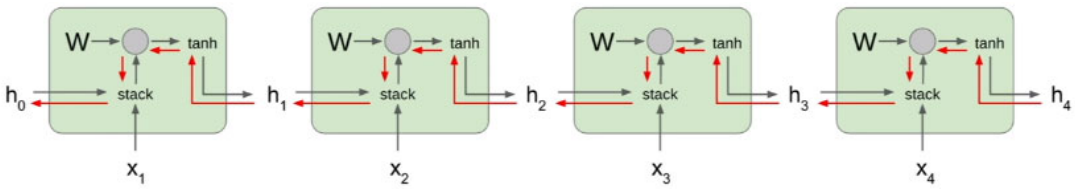
Vanishing Gradient



1D CNN may be better than RNN in some cases, worse here

Traditional NN, information feeds forward, error feeds backward
RNN, information travels through time so using information from previous times
Consider error term e_t
need to update weights for every node that participated in output
also update neurons backward in time (this is the weights on the loops)
to get from x_{t-3} to x_{t-2} multiply x_{t-3} by w_{rec}
to get from x_{t-2} to x_{t-1} multiply x_{t-2} by w_{rec}
multiply by same weight multiple times, when weight is small
the product decreases very quickly
we try for small weights, regularizer even helps with this
the lower the gradient, the harder to update weights, longer to converge
if w_{rec} is small (< 1) result is vanishing gradient
if w_{rec} is big (> 1), exploding gradient, neither converges

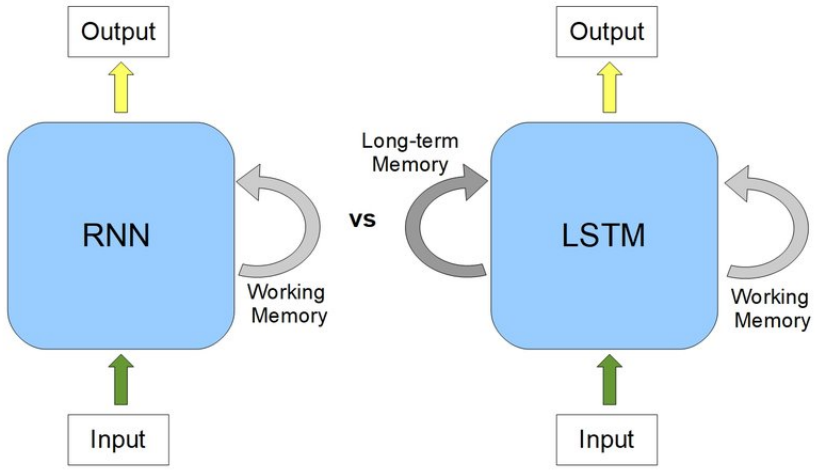
Vanishing Gradient



Bengio et al., ICML 2013

Computing gradient of h0 involves many W components and corresponding repeated tanh
If largest value > 1, exploding gradients; if largest value < 1, vanishing gradients

Long Short-Term Memory Networks



The vanishing gradient means the RNN cannot handle long term dependencies
Long term implies large number of multiplications, gradients get too small

LSTM introduces three gates: input gate, output gate, forget gate
These handle long term dependencies

NLP Example

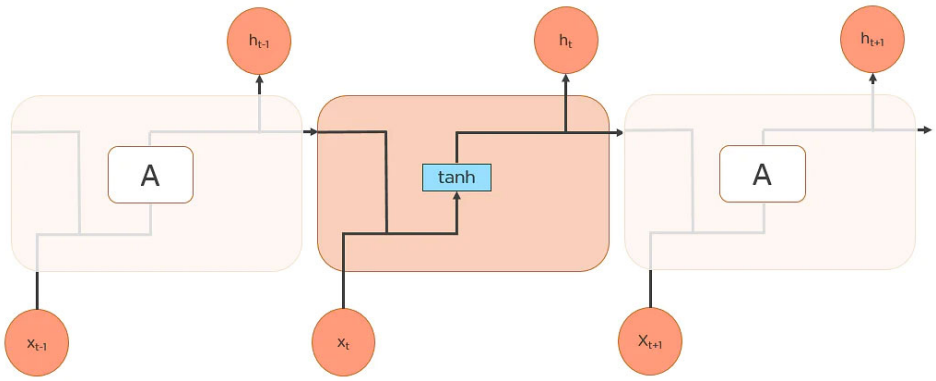
Twinkle twinkle little star, how I wonder what you _____.

The blank is filled with “are”. We do not need more context then this to predict the last word.

If the text was “I often play childhood songs on the piano. My favorite is Twinkle twinkle little star, how I wonder what you are.”
We do not need the earlier text for the prediction task, we can forget that.

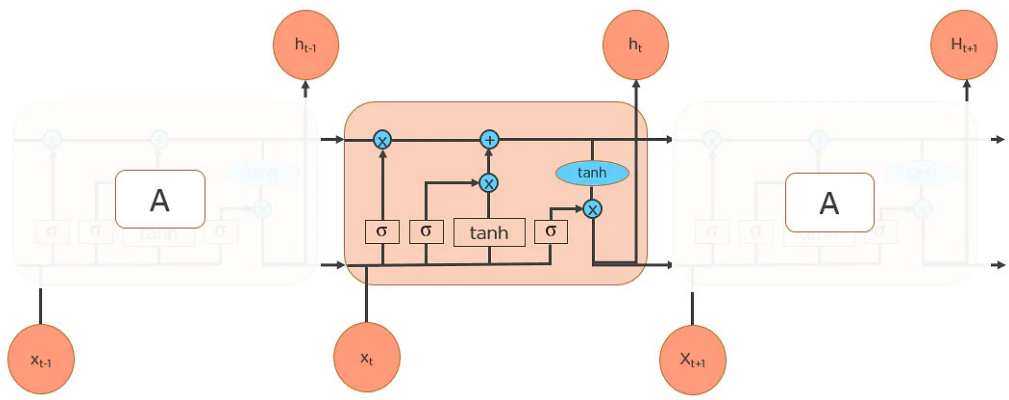
If the text was I often play childhood songs on the piano. My favorite is Twinkle twinkle little star, but I cannot remember what comes after the phrase “wonder what you ___”, the earlier reference to “Twinkle twinkle would be useful but not the text immediately after that. LSTMs help store this information.

RNN Structure



An RNN has a chain of repeating modules like this figure.

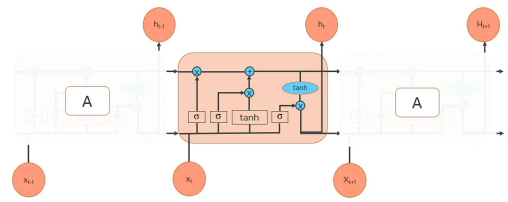
LSTM Structure



Three steps

Instead of a single NN layer, an LSTM has four interacting layers that communicate with each other.

Step 1: Forget gate



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

f_t = forget gate
Decides which information to delete that is not important from previous time step

Alice is good in Physics. John, on the other hand, is good at Chemistry.

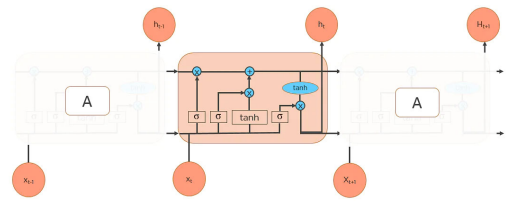
John plays football well. He told me yesterday over the phone that he had served as the captain of his college football team.

Decide how much past data to remember.

Consider the following two sentences:
1) Let the output of h(t-1) be "Alice is good in Physics. John, on the other hand, is good at Chemistry."
2) Let the current input at x(t) be "John plays football well. He told me yesterday over the phone that he had served as the captain of his college football team."

The forget gate realizes there might be a change in context after encountering the first full stop. It compares with the current input sentence at x(t). The next sentence talks about John, so the information on Alice is deleted. Forget gate decides what to keep from prior cell state.
The sigmoid function decides what past data to remember (values 0 or 1)

Step 2: Input gate



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

i_t = input gate
Determines which information to let through based on its significance in the current time step

Alice is good in Physics. John, on the other hand, is good at Chemistry.

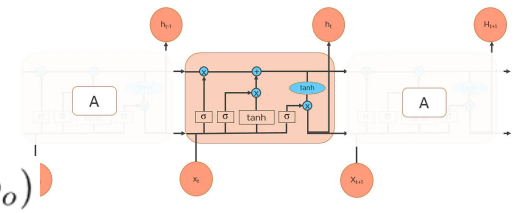
John plays football well. He told me yesterday over the phone that he had served as the captain of his college football team.

Two parts, sigmoid function (first equation) and tanh function (second question). Sigmoid decides which values to let through (0 or 1), tanh weights the values that are let through, indicating their importance (-1 to 1).

Alice is good in Physics. John, on the other hand, is good at Chemistry.
x(t) = John plays football well. He told me yesterday over the phone that he had served as the captain of his college football team.

Input gate determines important information is John plays football and John was captain of college team. He told me yesterday over the phone is less important and is forgotten. This process of adding new information is done via the input gate.
The input gate decides what information is relevant to update in the current cell state.

Step 3: Output gate



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

o_t = output gate
Allows the passed in information to impact the output in the current time step

Alice is good in Physics. John, on the other hand, is good at Chemistry.

John plays football well. He told me yesterday over the phone that he had served as the captain of his college football team.

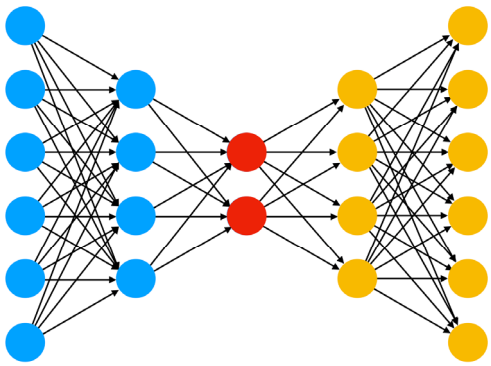
Decide what part of current cell state makes it to the output. First equation is sigmoid layer, decides what parts of cell state make it to output. Second equation is tanh, pushes values to be between -1 and 1, determining importance, and multiply by output of sigmoid gate.

Let's consider this example to predict the next word in the sentence: "John played tremendously well against the opponent and won for his team. For his contributions, brave ____ was awarded player of the match."

There could be many choices for the empty space. The current input brave is an adjective, and adjectives describe a noun. So, "John" could be the best output after brave.

This is how a single node of an LSTM works. Most popular version of RNN.

Autoencoders

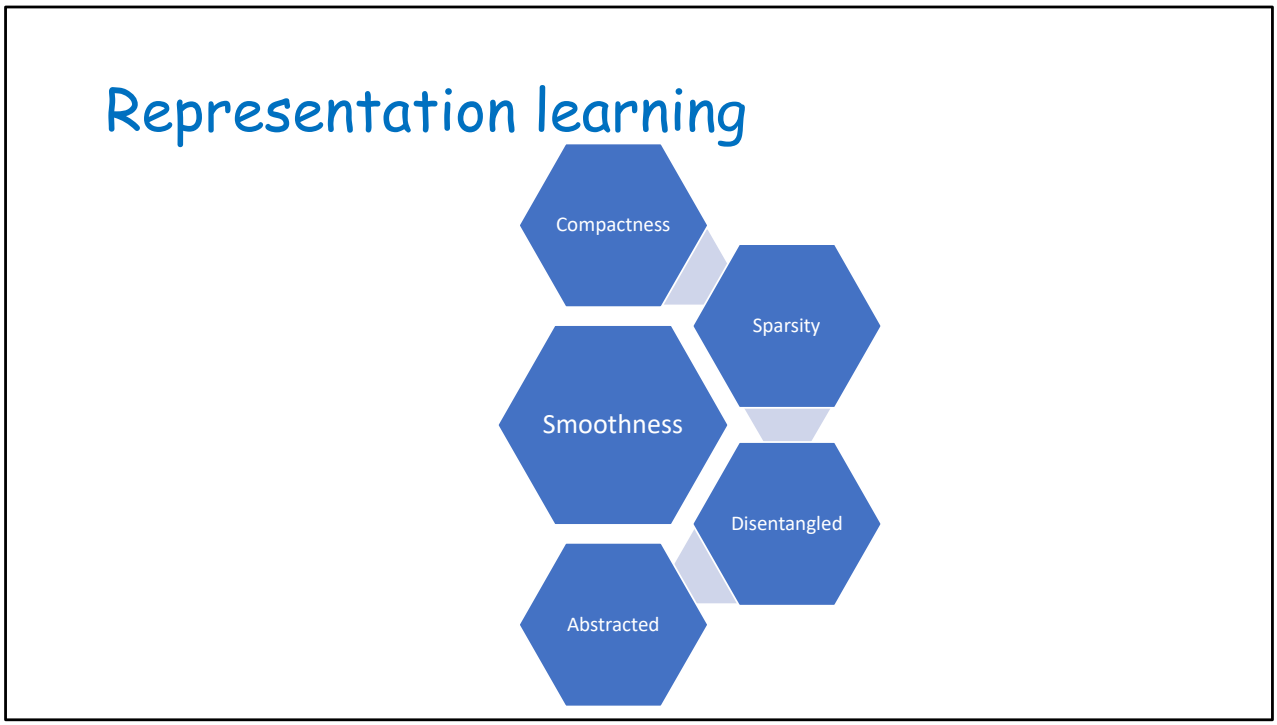


So far, DNNs have commonalities:

- Input layer, quantitative representation
- Hidden layer, apply transformation(s) with nonlinearity
- Output layer: result for classification, regression, translation, sequence prediction

All used for supervised learning

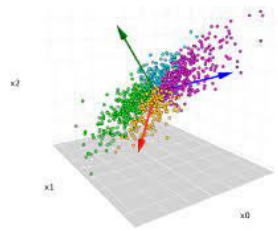
Now we consider dnn for unsupervised learning

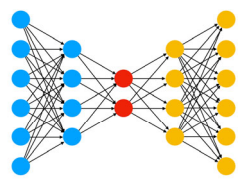


What kind of properties we want for representational learning:

1. Smoothness: small change to input corresponds to small change in rep
2. Compactness: this property may be desirable if you want to remove redundancy in the input
3. Sparsity: want small weights, regularizers
4. Disentangle: find factors that explain the input in a way independent of each other
5. Abstraction: high degree of invariance, recognize higher level properties of inputs that are shared between superficially dissimilar inputs (e.g., face images with different illuminations, poses, angles, still want to recognize high level abstract property of a particular face even though in pixel space they are very different)

PCA vs Autoencoder





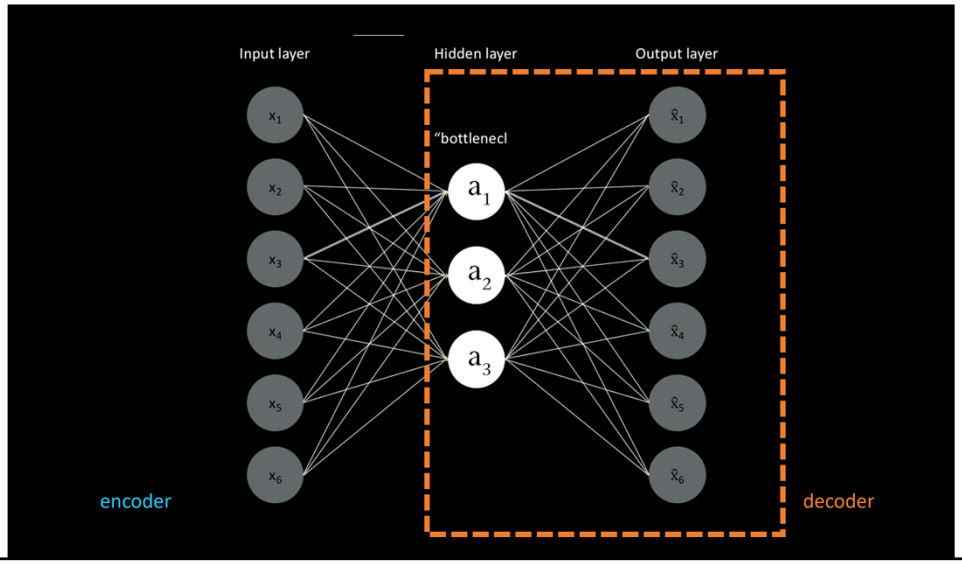
For many tasks, impossible to know what features to extract. Alan Turing deciphering the enigma code observed patterns that regularly appeared in messages. This is where representation learning truly comes into view.

In representation learning, machine is provided with data and learns the representation that dictate how the predictive model will perform.

Representation learning works by reducing high-dimensional data into low-dimensional data, making it easier to find patterns, anomalies, and also giving us a better understanding of the behavior of the data. Reduce the complexity of the data, so anomalies and noise are reduced.

PCA maps N-dimensional data to M-dimensional data, M<<N
AE can mimic PCA. An AE is a NN trained to copy input to output.

Autoencoder visualization



An AE is a NN trained to copy input to output.

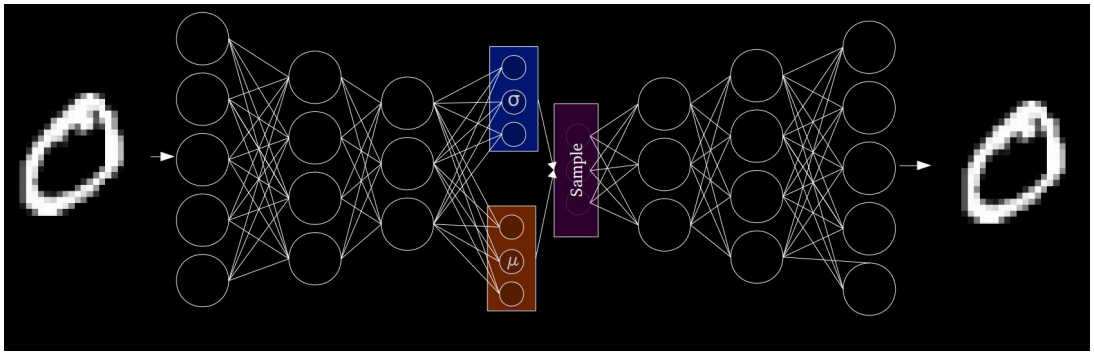
Squish it through "sieve" first, hidden layer. If imitating PCA, N input nodes, N output nodes, M hidden nodes, M<<N. NN learns best way to compress data that minimizes reconstruction loss.

AE learns efficient, compact data representations by training the NN to ignore noise. Used for denoising, compression, learning latent space.

Ae.py

Objective function is $\min|x - \hat{x}|$

Variational autoencoders



A VAE latent space explicitly encodes the data distribution.

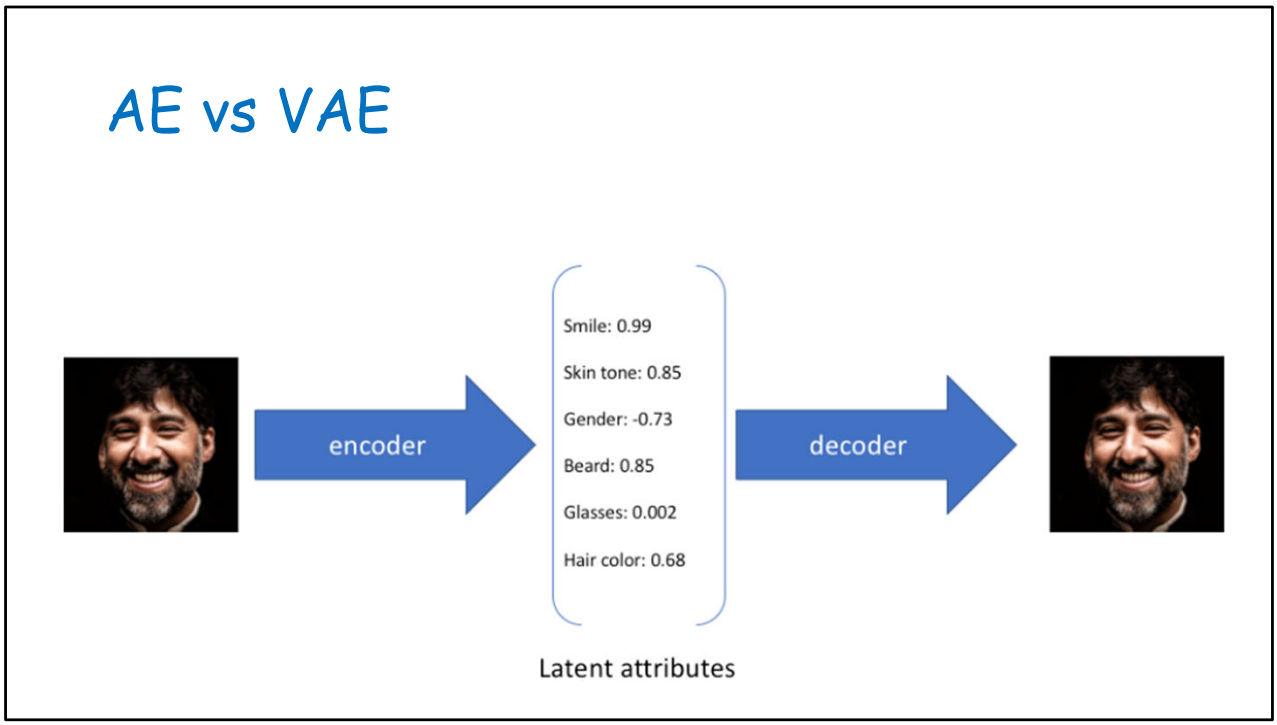
Input -> encode to statistic vectors describing a distribution -> sample a latent vector -> decode for reconstruction

Loss: Reconstruction + KL Divergence

Line 11 of cell 4 for VAE in ae.py

$$\min |x - \hat{x}|$$

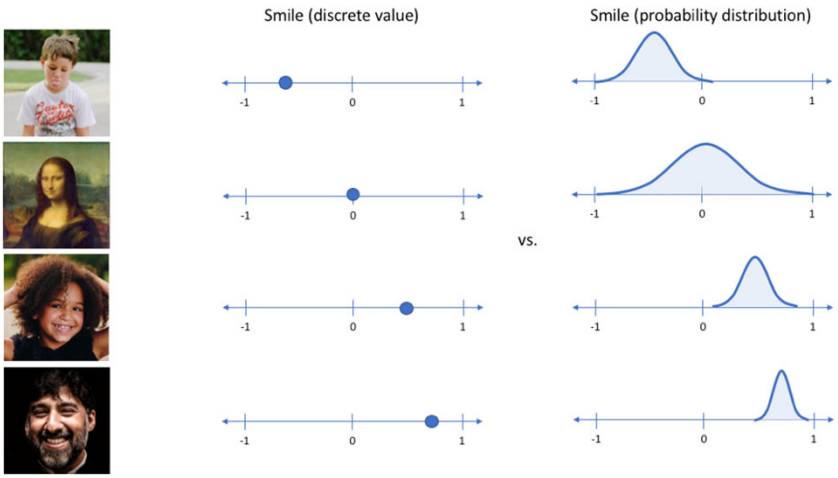
$$KL(p||q) = -\sum p(x) \log q(x) + \sum p(x) \log p(x)$$



AE learns latent attributes

$$\min|x - \hat{x}|$$

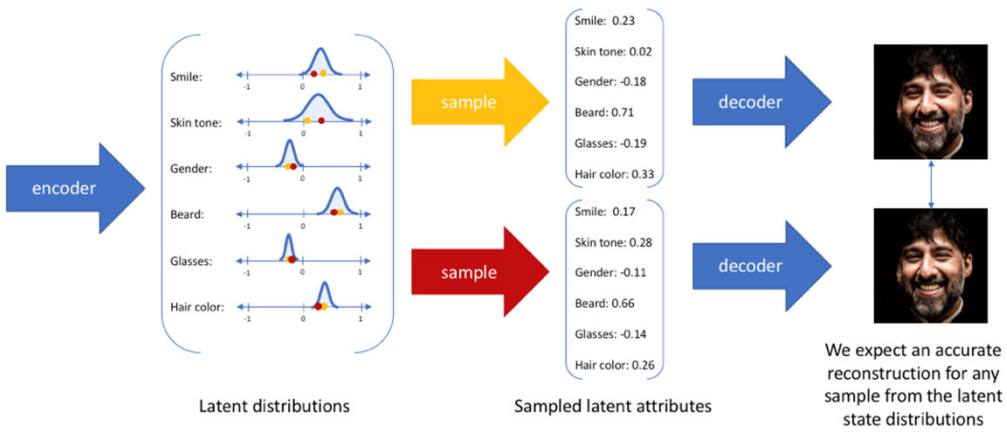
AE vs VAE



VAE learns distribution parameters

$$\min|x - \hat{x}|$$

AE vs VAE



VAE learns distribution parameters

$$\min|x - \hat{x}|$$

VAEs for data generation

Let's say you are developing a video game, and you have an open-world game that has very complex scenery. You hire a team of graphic designers to make a bunch of plants and trees to decorate your world with, but once putting them in the game you decide it looks unnatural because all of the plants of the same species look exactly the same, what can you do about this?

At first, you might suggest using some parameterizations to try and distort the images randomly, but how many features would be enough? How large should this variation be? How computationally intensive?

Use a VAE. Train a NN to learn latent features about the plant, and then every time one pops up in our world, we can take a random sample from our "learned" features and generate a unique plant. This is how many open world games have started to generate aspects of the scenery within their worlds.

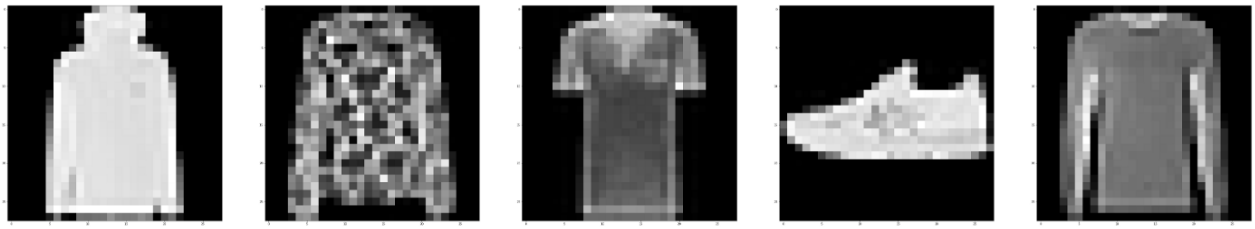
VAEs for data generation

towardsdatascience.com

Imagine we are an architect and want to generate floor plans for a building of arbitrary shape. We can an autoencoder network to learn a data generating distribution given an arbitrary build shape, and it will take a sample from our data generating distribution and produce a floor plan, shown here.

VAEs for data generation

- Input (fashion MNIST)



VAEs for data generation

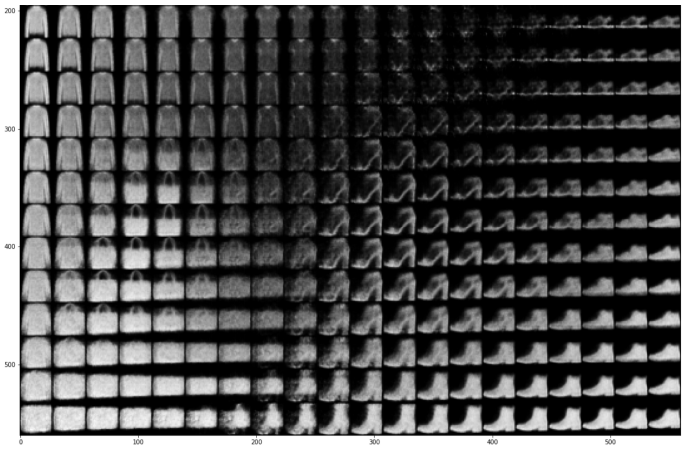
- NN structure

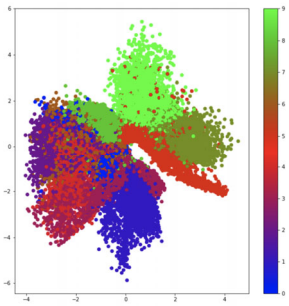
```
input_layer = Input(shape=(258, 540, 1))
encoder = Conv2D(32, (3,3), activation='relu', padding='same')
# 3 more Conv2D layers, then Flatten and Dense layers
output = Dense(latent_dim)
decoder_input = Input(latent)
x = Dense(shape_before_flattening)
x = Reshape()
x = Conv2DTranspose(32, 3)
x = Conv2D(1, 3)
Decoder = Model(decoder_input, x)
```

```
def sampling(z_mu, z_log_sigma):
  epsilon = K.random_normal(shape=(K.shape(z_mu), latent_dim)
  return z_mu + K.exp(z_log_sigma) * epsilon
```

VAEs for data generation

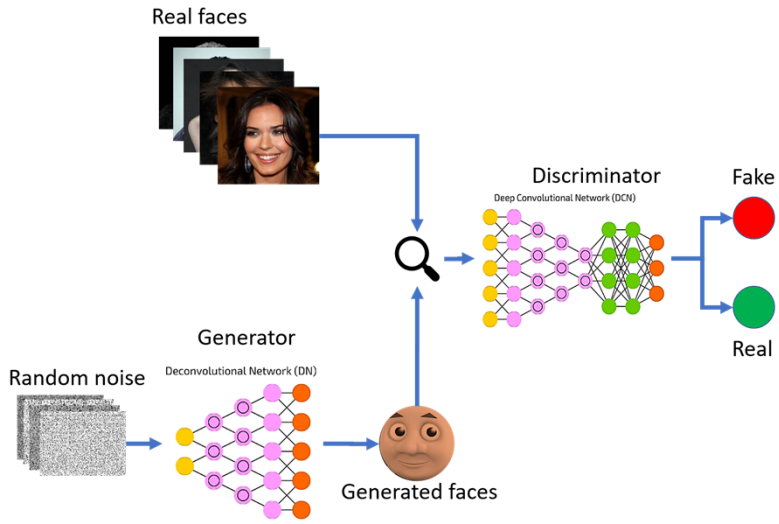
- Output



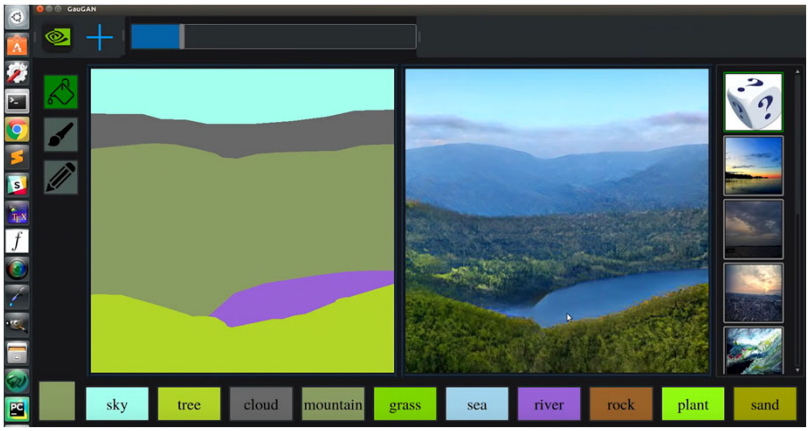


Plot on right shows how items are separated into distinct clusters by the VAE.

Generative Adversarial Networks



Generative adversarial networks

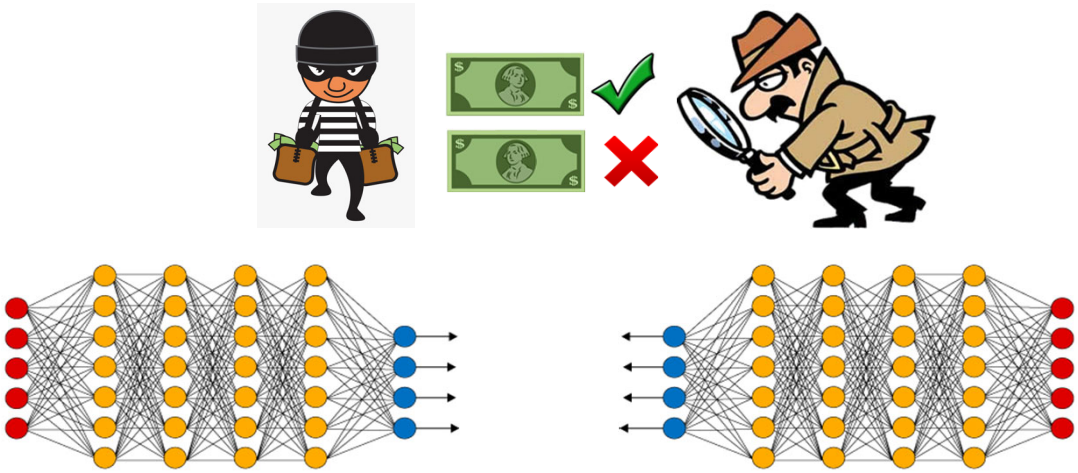


gan1.py

GauGAN
Turn combination of words and drawings into lifelike image.
Type phrase “lake in front of mountain” and generate scene in real time.
Start with your own doodle or existing scene. Areas are labeled with features (e.g., sand, river, grass, flower), providing instructions on how to fill in scene.

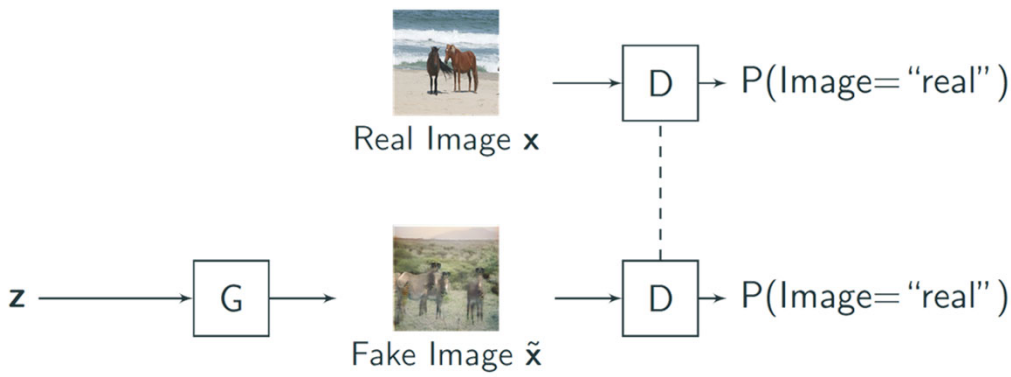
Generative adversarial networks

Generative adversarial networks



gan1.py

Generative adversarial networks



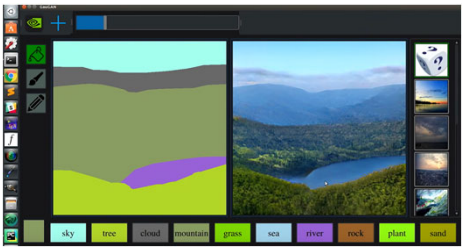
$$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log(1 - D(G(z)))\right]$$

Dashed line between D networks indicates they share weights or are the same network. Top row, real image from training data is fed to discriminator. Bottom row, fake image from generator fed to D

Minimax objective of the game. G wants min, D wants max.
x is real sample, z is input noise sample
log(D(x) is log prob of correctly discriminating real (x) and fake (G(z)) Goal of D
D goal is maximize log D(x) = 1 (correctly classify as real)
and D(G(z))=0 (correctly classify as fake), or maximize log(1 – D(G(z))
G goal is to minimize these terms
Mission is to find parameters for G and D that optimize this function
Discriminator learns to correctly classify input as real or fake while generator tries to fool the discriminator into thinking that its generated output is real.

GANs are powerful .. GANs are hard

- Hard to train
- Hard to evaluate



Hard to train, vanishing gradients, mode collapse (keep generating same sample), non-convergence, instability
Hard to train, hard to formalize what makes one image more realistic than another, some automated, need humans

GAN applications

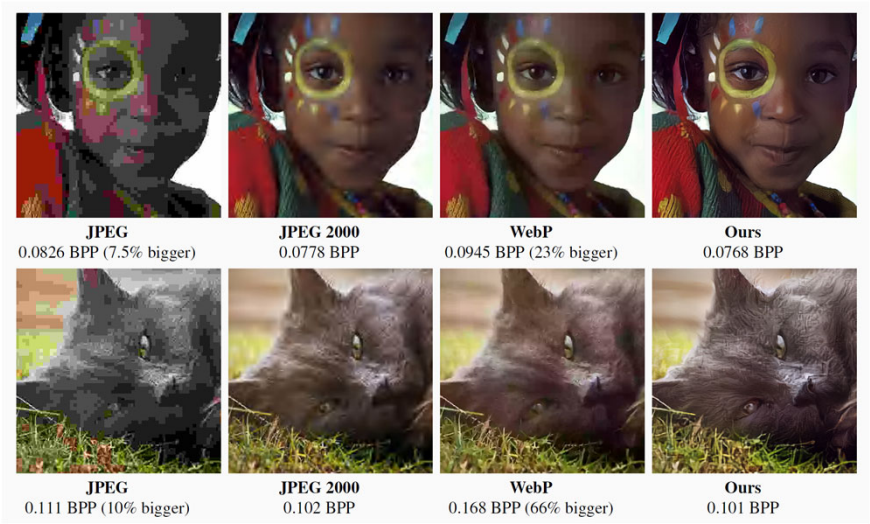


Image-to-image / video-to-video translation

Picture, produce more realistic images with very low bit rates

Video compression

Domain adaptation

Learning by an adversary: object detection, robot grasping, etc.

GAN applications

Image-to-image / video-to-video translation

Picture, produce more realistic images with very low bit rates

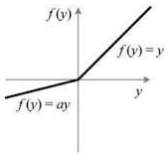
Video compression

Domain adaptation

Learning by an adversary: object detection, robot grasping, etc.

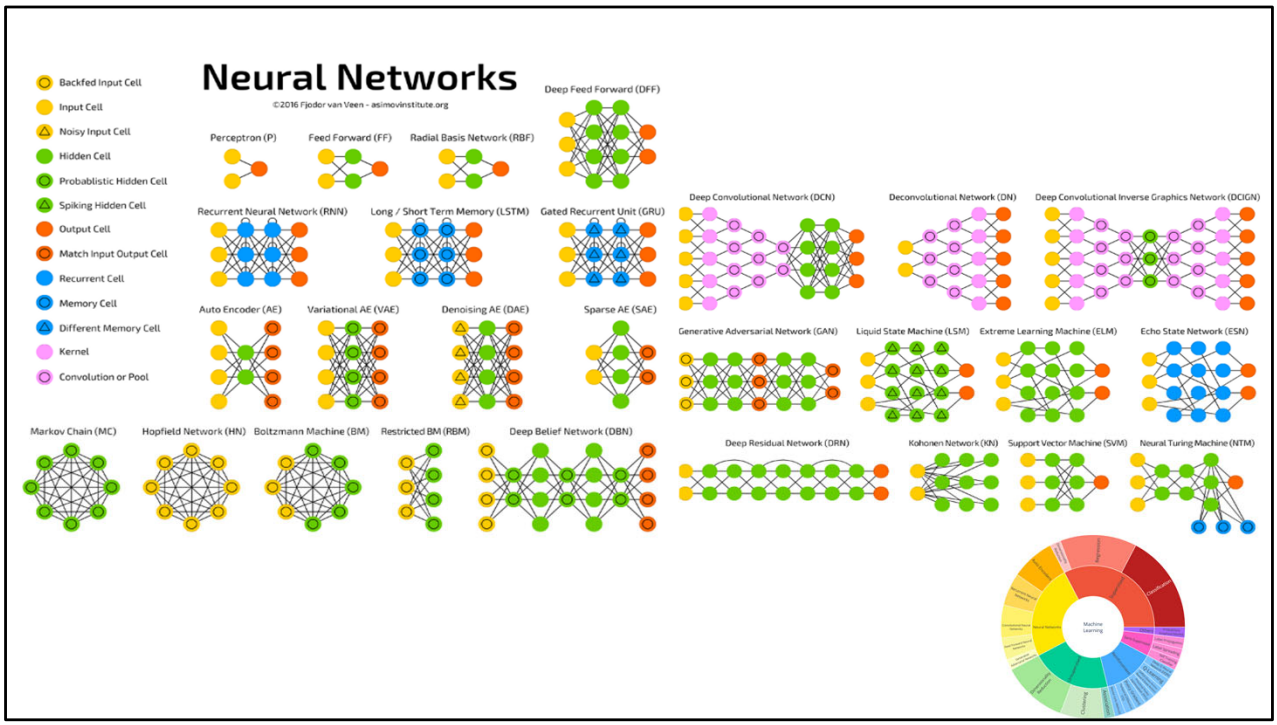
Let's try this out

Gan1.py, compare with gan0.py for 1 epoch



Leaky RelU, small slope for negative values instead of flat slope (RelU), useful when we suffer from sparse gradients (not strong enough feedback) as we often do with GANs.

Define D and G structures, sample real data, start with noise for fake data
Look at train function, alternate between training D and training G.


Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)