CS-GY 6923: Lecture 14
Finish Semantic Embeddings, Modern Image Generation, Reinforcement Learning

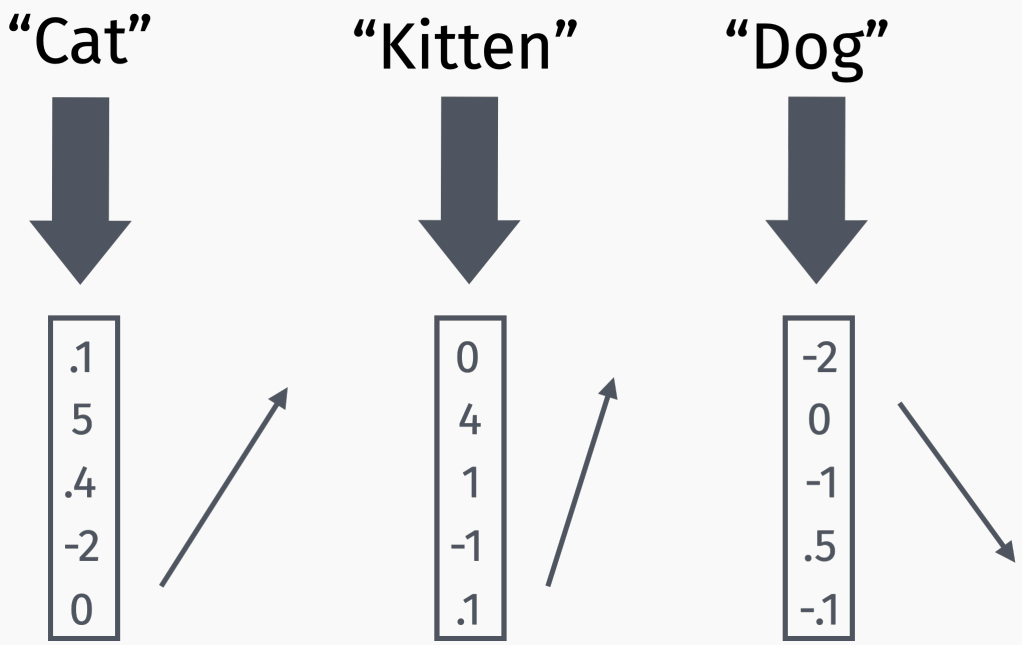
NYU Tandon School of Engineering, Prof. Christopher Musco

SEMANTIC EMBEDDING

Goal: Learn mapping from inputs to numerical vectors such that similar inputs map to similar vectors (e.g., with high inner product).

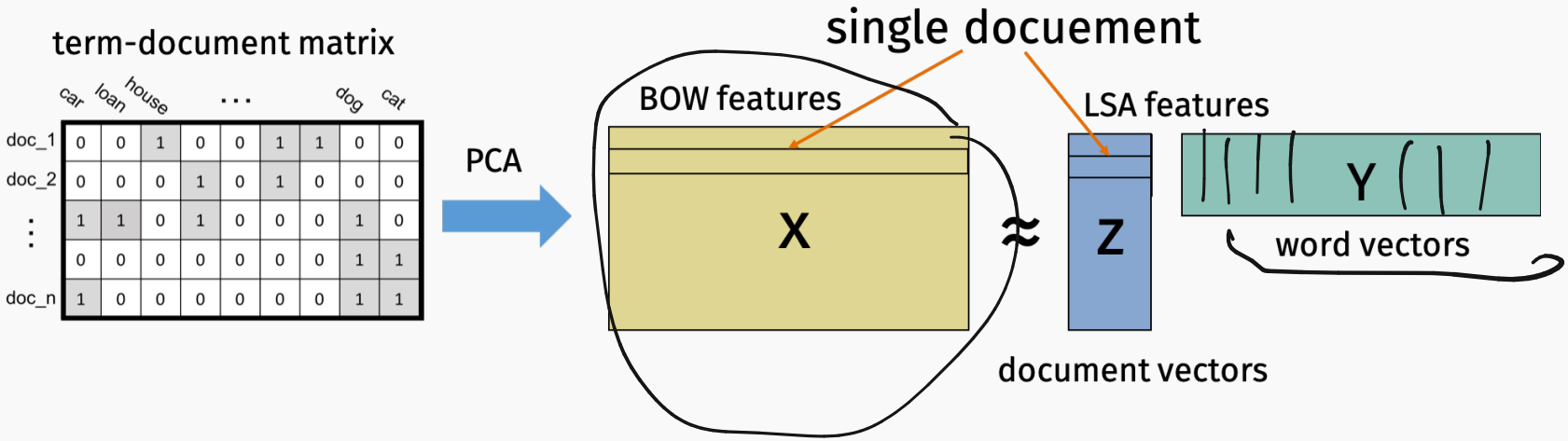
SEMANTIC EMBEDDING

Goal: Learn mapping from inputs to numerical vectors such that similar inputs map to similar vectors (e.g., with high inner product).



For words, the mapping is typically just a lookup table.

HOW TO GET EMBEDDINGS?



For documents or words, earliest approaches were based on latent semantic analysis (PCA on term document matrix).

More modern word embedding recipe:

1. Choose similarity metric $k(word_i, word_j)$ which can be computed for any pair of words.
2. Construct similarity matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ with $\mathbf{M}_{i,j} = k(word_i, word_j)$.
3. Find low rank approximation $\mathbf{M} \approx \mathbf{Y}^T\mathbf{Y}$ where $\mathbf{Y} \in \mathbb{R}^{k \times n}$.
4. Columns of $\mathbf{Y}$ are word embedding vectors.

We expect that $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ will be larger for more similar words.

$M = U \Sigma V^T$ $\quad U \neq V$ even if M is symmetric.

Common choice for similarity metric is to use co-occurence frequency in windows.

The girl walks to her dog to the park.
It can take a long time to park your car in NYC.
The dog park is always crowded on Saturdays.

The girl walks to her dog to the park.
It can take a long time to park your car in NYC.
The dog park is always crowded on Saturdays.

The girl walks to her dog to the park.
It can take a long time to park your car in NYC.
The dog park is always crowded on Saturdays.

	dog	park	crowded	the
dog	0	2	0	3
park	2	0	1	2
crowded	0	1	0	0
the	3	2	0	0

Usually followed by some tranformation or normalization. E.g.,
$k(word_i, word_j) = \log\left(\frac{p(i,j)}{p(i)p(j)}\right)$

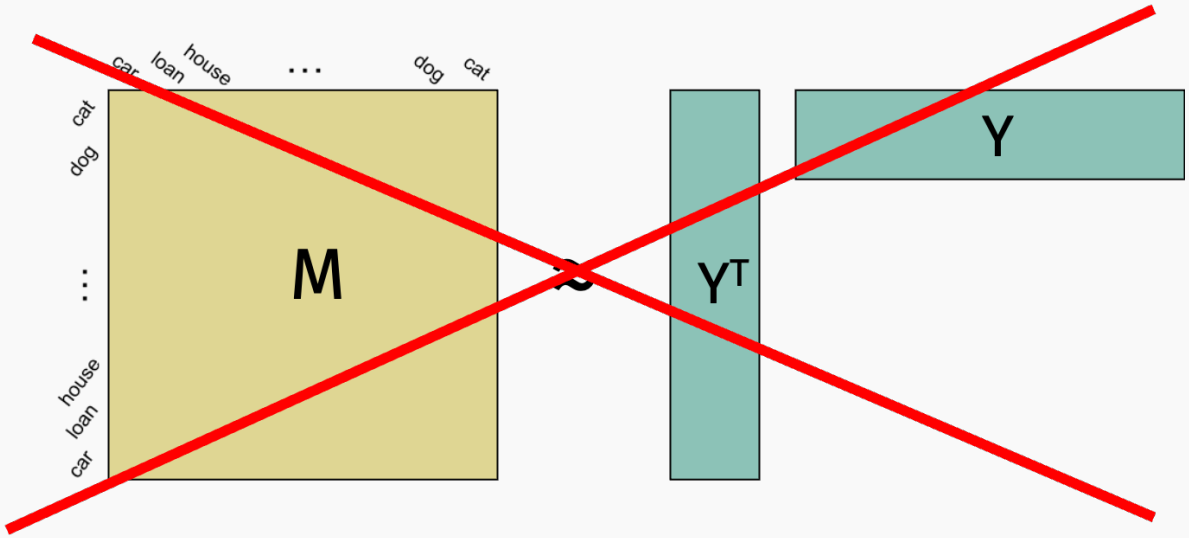
Current state of the art models: `GloVE`, `word2vec`.

- `word2vec` was originally presented as a shallow neural network model, but it is equivalent to matrix factorization method (Levy, Goldberg 2014).
- For `word2vec`, similarity metric is the "point-wise mutual information": $\left(\log \frac{p(i,j)}{p(i)p(j)}.\right)$

Common to use pre-trained word vectors:

- Compilation of many sources: `https://github.com/3Top/word2vec-api`

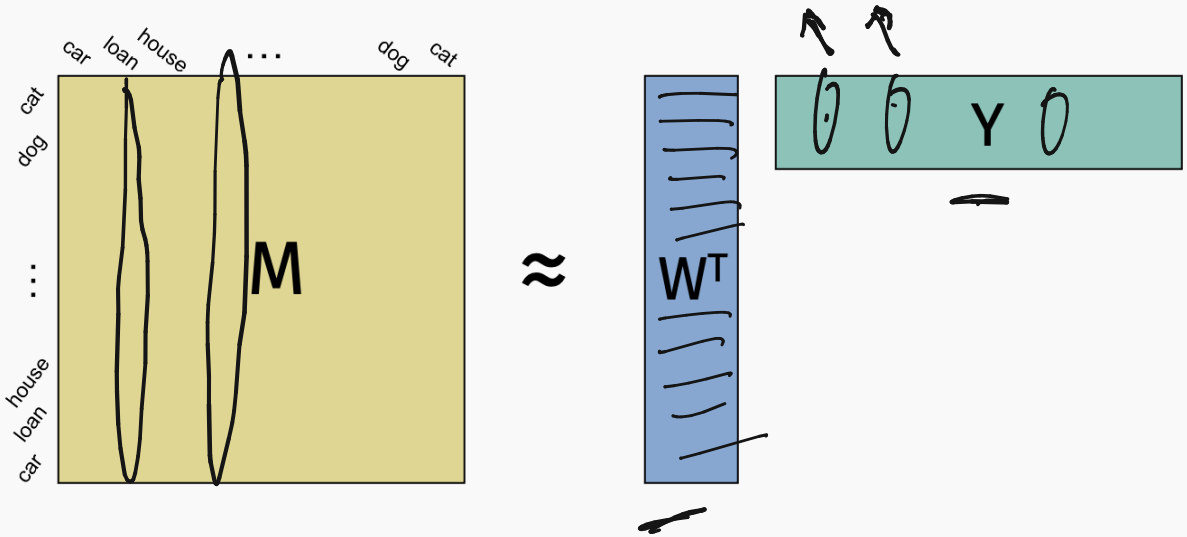
CAVEAT ABOUT FACTORIZATION



SVD will not return a symmetric factorization in general. In fact, if $\mathbf{M}$ is not positive semidefinite[1] then the optimal low-rank approximation does not have this form.

[1] I.e., $k(word_i, word_j)$ is not a positive semidefinite kernel.

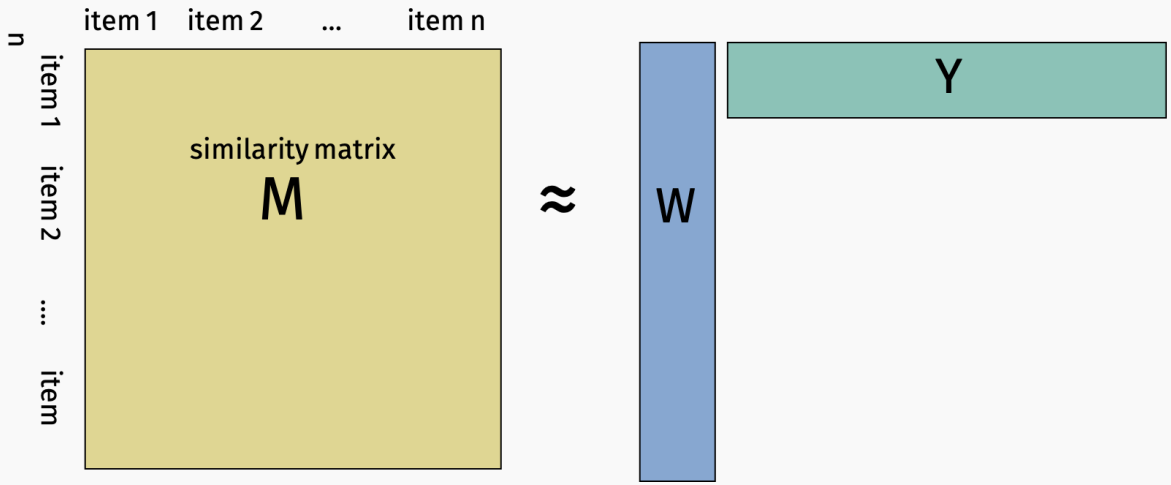
CAVEAT ABOUT FACTORIZATION



- For each word i we get a left and right embedding vector $\mathbf{w}_i$ and $\mathbf{y}_i$. It's reasonable to just use one or the other.
- If $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ is large and positive, we expect that $\mathbf{y}_i$ and $\mathbf{y}_j$ have similar similarity scores with other words, so they typically are still related words.
- Another option is to use as your features for a word the concatenation $[\mathbf{w}_i, \mathbf{y}_i]$

SEMANTIC EMBEDDINGS

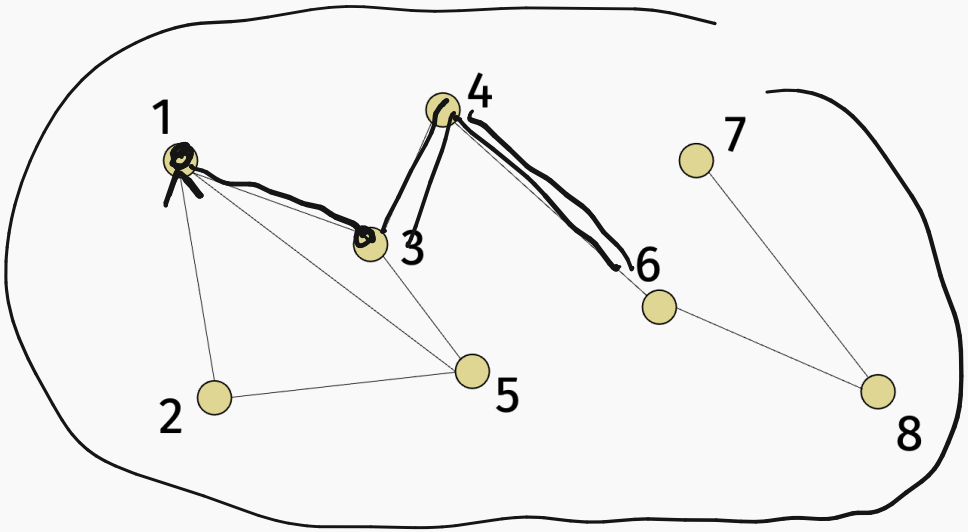
The same approach used for word embeddings can be used to obtain meaningful numerical features for any other data where there is a natural notion of similarity.



For example, the items could be(nodes in a social network) graph. Maybe be want to predict an individuals age, level of interest in a particular topic, political leaning, etc.

NODE EMBEDDINGS

Generate random walks (e.g. “sentences” of nodes) and measure similarity by node co-occurence frequency.

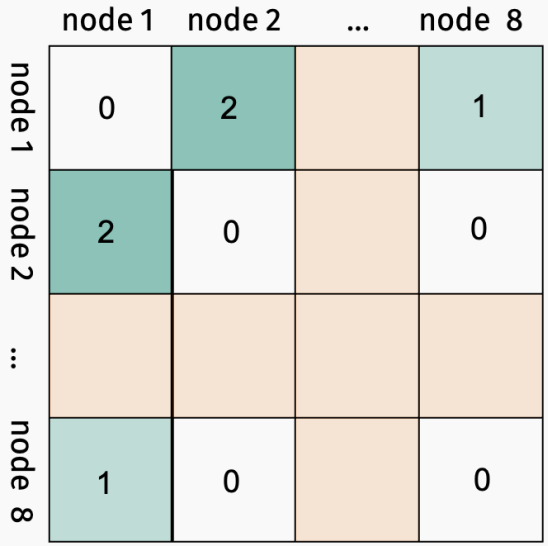


1, 3, 4, 4, 5, 2, 1, 2, 5
6, 8, 6, 4, 3, 1, 5, 3, 4
7, 8, 6, 8, 7, 8, 6, 8, 6
⋮
4, 6, 8, 6, 4, 3, 1, 2, 5

NODE EMBEDDINGS

Again typically normalized and apply a non-linearity (e.g. log) as in word embeddings.

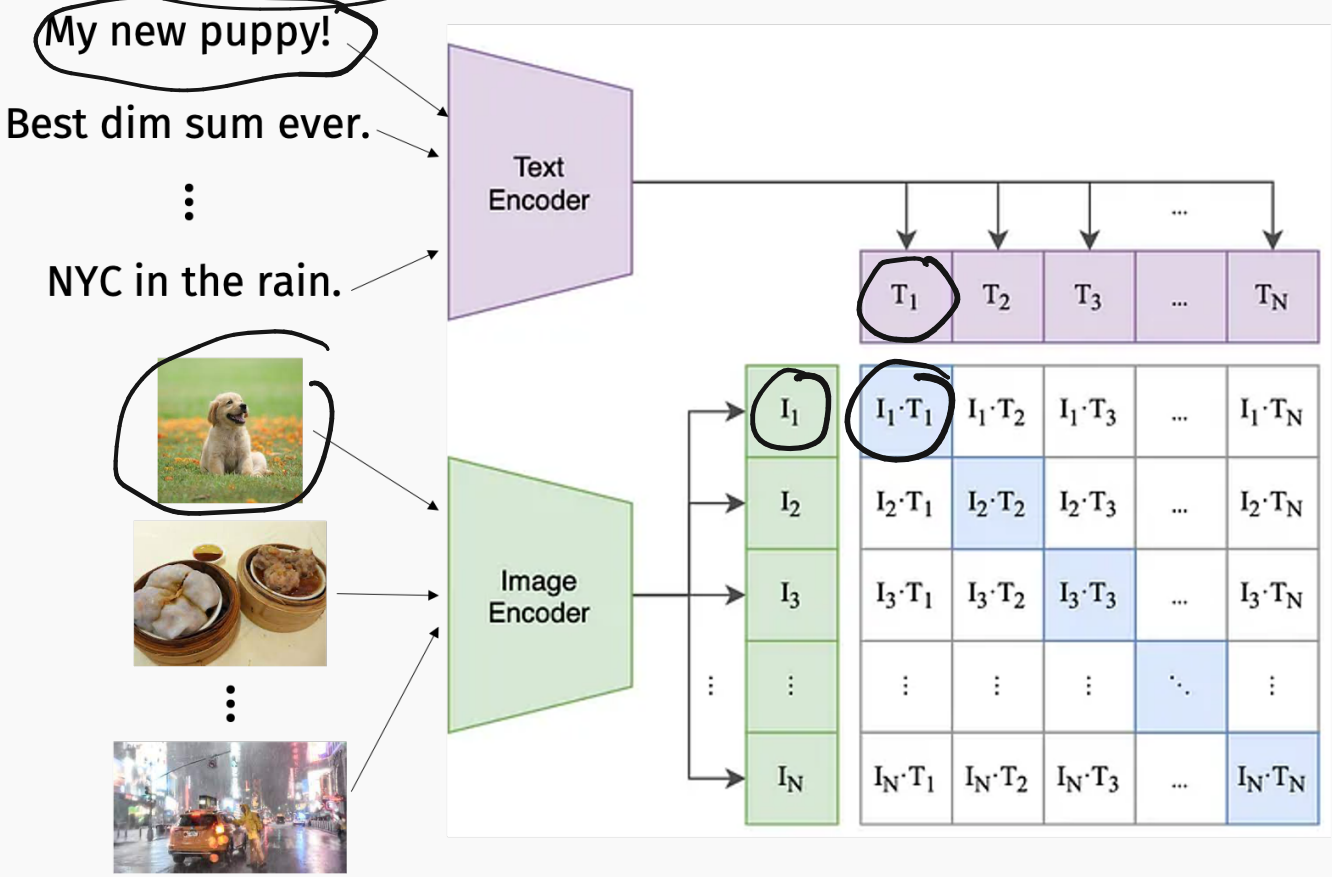
1, 3, 4, 4, 5, 2, 1, 2, 5
6, 8, 6, 4, 3, 1, 5, 3, 4
7, 8, 6, 8, 7, 8, 6, 8, 6
⋮
4, 6, 8, 6, 4, 3, 1, 2, 5



Popular implementations: `DeepWalk`, `Node2Vec`. Again initially derived as simple neural network models, but are equivalent to matrix-factorization (Qiu et al. 2018).

BIMODAL EMBEDDINGS

We can also create embeddings that represent different types of data. OpenAI's CLIP architecture:



Goal: Train embedding architectures so that $\langle \mathbf{T}_i, \mathbf{I}_j \rangle$ are similar if image and sentence are similar.

CLIP TRAINING

What do we use as ground truth similarities during training? Sample a batch of sentence/image pairs and just use identity matrix.



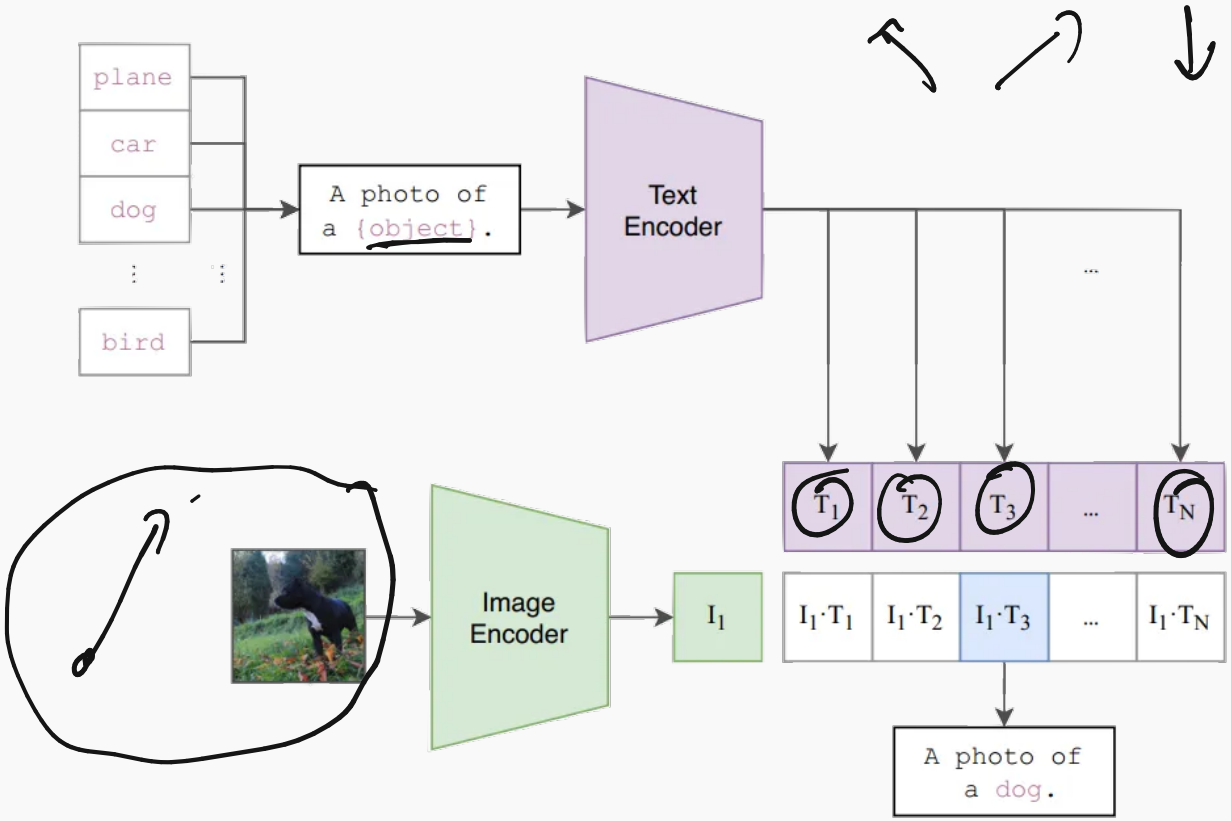
My new puppy!	1	0	0
Best dim sum ever.	0	1	0
NYC in the rain.	0	0	1

This is called contrastive learning. Train unmatched text/image pairs to have nearly orthogonal embedding vectors.

CLIP FOR ZERO-SHOT LEARNING

Learning Transferable Visual Models From Natural Language Supervision

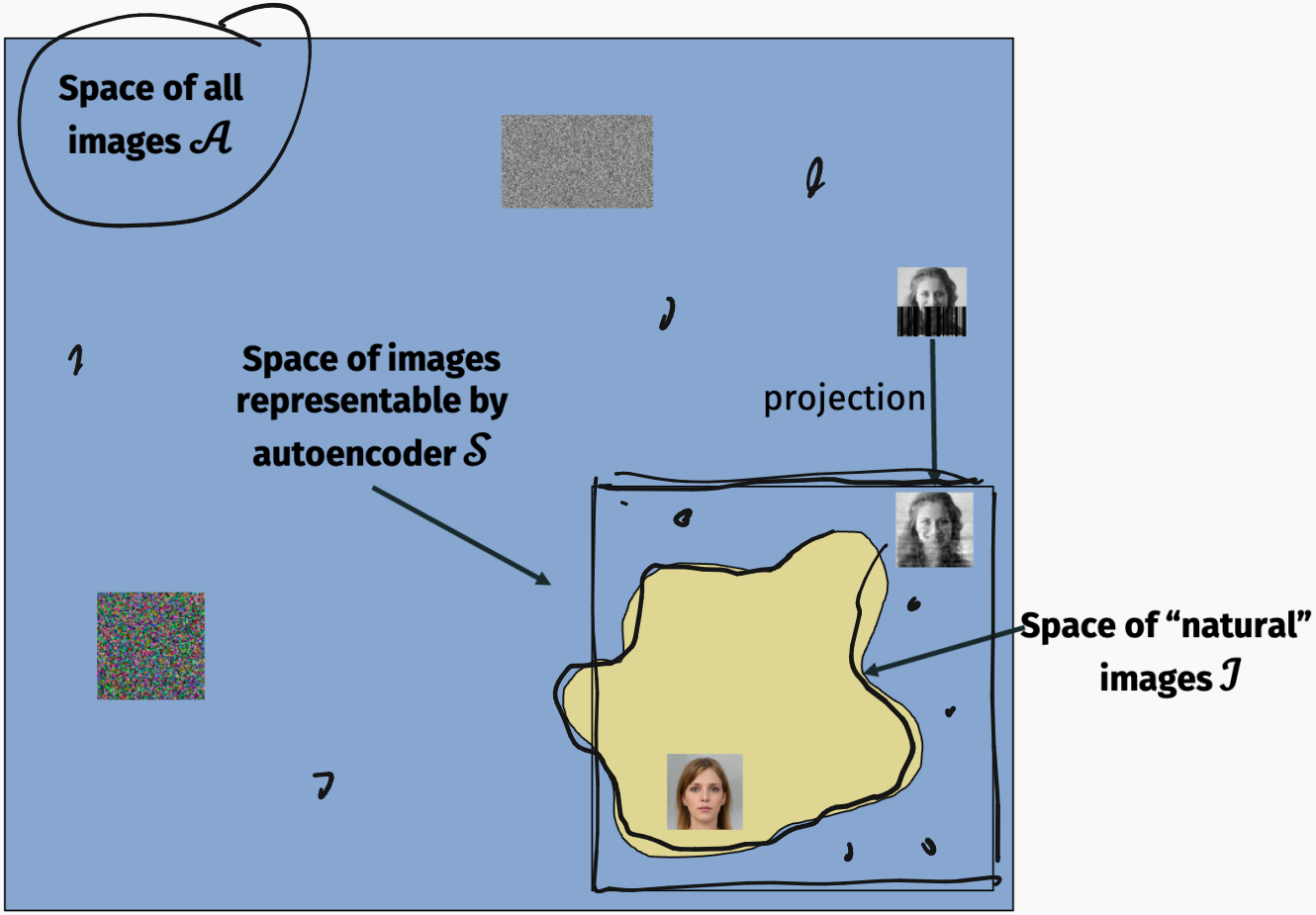
Alec Radford [*1] Jong Wook Kim [*1] Chris Hallacy [1] Aditya Ramesh [1] Gabriel Goh [1] Sandhini Agarwal [1]
Girish Sastry [1] Amanda Askell [1] Pamela Mishkin [1] Jack Clark [1] Gretchen Krueger [1] Ilya Sutskever [1]



2021 result: 76% accuracy on ImageNet image classification challenge with no labeled training data.

IMAGE SYNTHESIS (TEASER)

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS



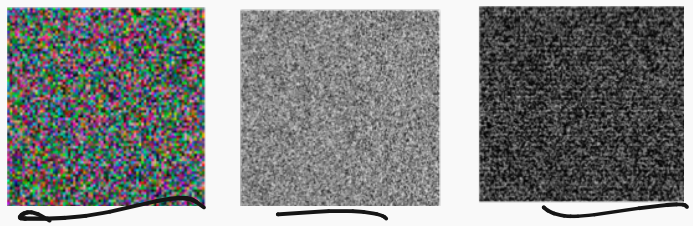
$f(\mathbf{x}) = d(e(\mathbf{x}))$ projects an image $\mathbf{x}$ closer to the space of natural images.

$d(z) \rightarrow$ random

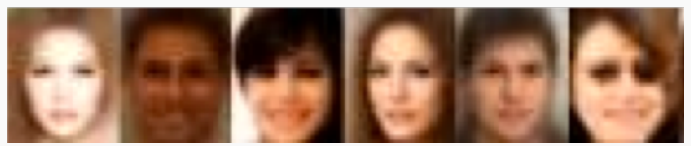
AUTOENCODERS FOR DATA GENERATION

Suppose we want to generate a random natural image. How might we do that?

- **Option 1**: Draw each pixel value in $\mathbf{x}$ uniformly at random. Draws a random image from $\mathcal{A}$.



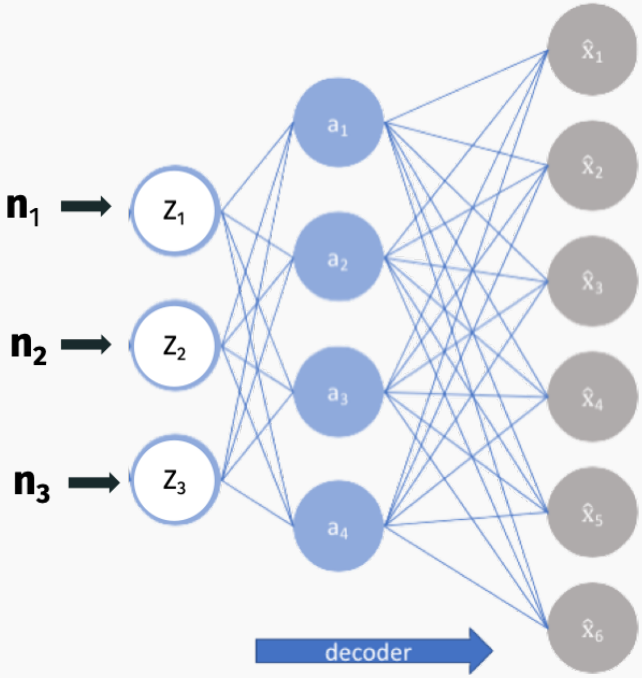
- **Option 2**: Draw $\mathbf{x}$ randomly from $\mathcal{S}$, the space of images representable by the autoencoder.

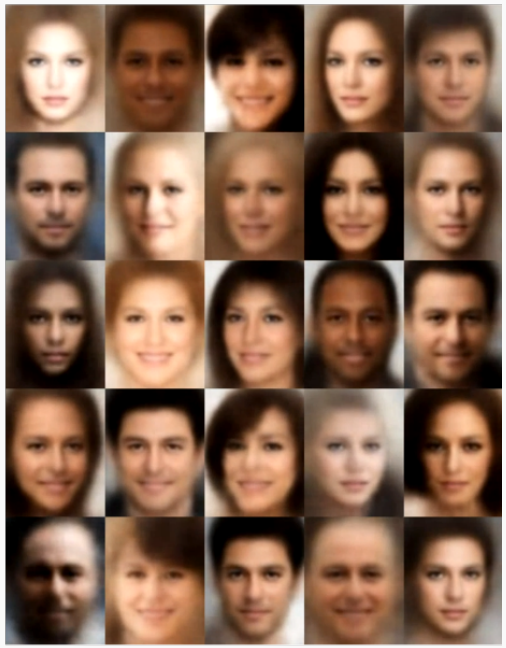


How do we randomly select an image from $\mathcal{S}$?

AUTOENCODERS FOR DATA GENERATION

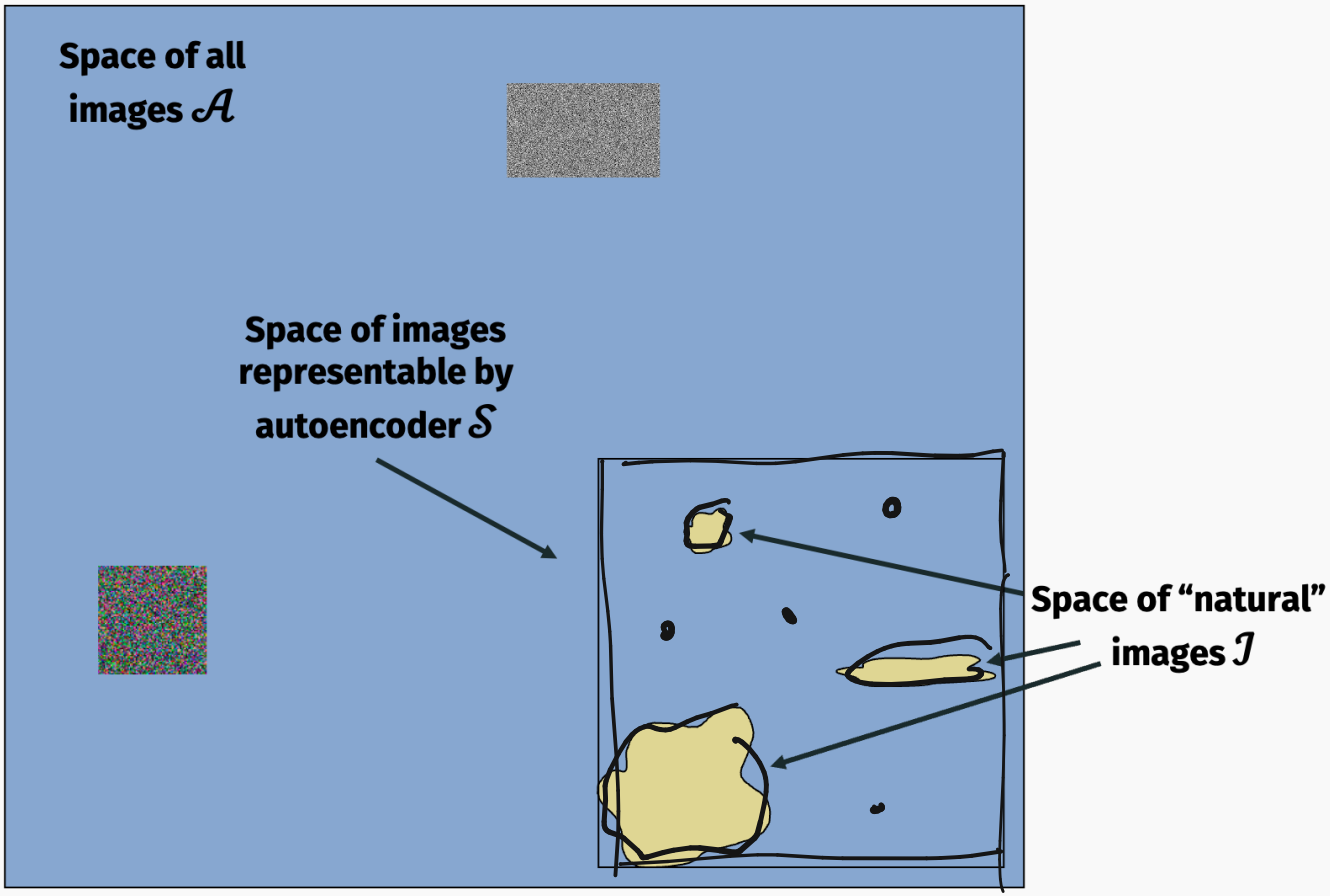
Autoencoder approach to generative ML: Feed random inputs into decode to produce random realistic outputs.





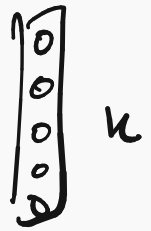
Main issue: most random inputs words will "miss" and produce garbage results.

AUTOENCODERS FOR DATA GENERATION



Variational auto-encoders attempt to resolve this issue.

Developed from a different perspective than regular autoencoders. **Make the data generation goal more explicit.**



- Train a neural network G_θ that takes in a length k code word, $\mathbf{z}$, and outputs an image.
- Assume $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$. I.e., a random Gaussian vector.
- Goal is to maximize probability of producing a "natural image".

$G_\theta(z)$

First attempt: Given training data $\mathbf{x}_1, \ldots, \mathbf{x}_n$,

$$\max_\theta \int \mathbb{1}[G_\theta(\mathbf{z}) = \mathbf{x}_i \text{ for some } i] \cdot p(\mathbf{z}) d\mathbf{z}$$

$$= \max_\theta \mathbb{E}_\mathbf{z} \mathbb{1}[G_\theta(\mathbf{z}) = \mathbf{x}_i \text{ for some } i]$$

Issues: Super brittle, impossible to train.

$z \sim N(0, I)$

Bayesian approach: assume each $\mathbf{x}_i$ is of the form $G_\theta(\mathbf{z}) + \sigma\mathcal{N}(0, \mathbf{I})$ for randomly chosen $\mathbf{z}$. Choose parameters, θ, to maximize the likelihood of the data:

$$\max_\theta \prod_{i=1}^{n} p(\mathbf{x}_i) = \max_\theta \prod_{i=1}^{n} \int p(\mathbf{x}_i \mid \mathbf{z}) \cdot p(\mathbf{z}) d\mathbf{z}$$

$$= \max_\theta \sum_{i=1}^{n} \log \int p(\mathbf{x}_i \mid \mathbf{z}) \cdot p(\mathbf{z}) d\mathbf{z}$$

$$= \min_\theta \sum_{i=1}^{n} -\log \int e^{-\|\mathbf{x}_i - G_\theta(\mathbf{z})\|_2^2 / 2\sigma^2} \cdot p(\mathbf{z}) d\mathbf{z}$$

$\sigma N(0, I) \Rightarrow x_i \sim G_\theta(z)$

$$\max_\theta \prod_{i=1}^n p(\mathbf{x}_i) = \max_\theta \prod_{i=1}^n \int p(\mathbf{x}_i \mid \mathbf{z}) \cdot p(\mathbf{z}) d\mathbf{z}$$

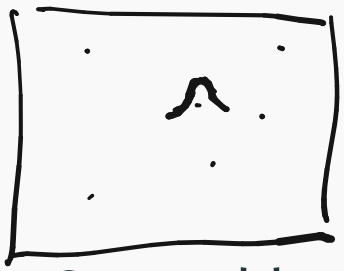
$$= \max_\theta \sum_{i=1}^n \log \int p(\mathbf{x}_i \mid \mathbf{z}) \cdot p(\mathbf{z}) d\mathbf{z}$$

How to deal with the integral? Very common approach in generative modeling (beyond VAEs): Monte Carlo approximation. Draw samples $\mathbf{z}_1, \ldots, \mathbf{z}_m$ and observe that:

$$\approx \max_\theta \sum_{i=1}^n \log \left(\frac{1}{m} \sum_{j=1}^m p(\mathbf{x}_i \mid \mathbf{z}_j) \right).$$

This approach does not work out of the box. The issue is that the integral will be very poorly approximated by sampling:

$$\int p(\mathbf{x}_i \mid \mathbf{z}) \cdot p(\mathbf{z}) d\mathbf{z} \not\approx \frac{1}{m} \sum_{j=1}^{m} p(\mathbf{x}_i \mid \mathbf{z}_j).$$



Second key idea: Importance sampling. For any distribution $q(\mathbf{z})$,

$$p(\mathbf{x}_i) = \int p(\mathbf{x}_i \mid \mathbf{z}) \cdot p(\mathbf{z}) d\mathbf{z} = \int q(\mathbf{z}) \left(\frac{p(\mathbf{x}_i \mid \mathbf{z})}{q(\mathbf{z})} \cdot p(\mathbf{z}) \right) d\mathbf{z}$$

$\mathbb{E}_{z \sim q}\left(\frac{p(x_i|z)}{q(z)} p(z) \right)$

Draw $\mathbf{z}_1, \ldots, \mathbf{z}_m$ from $q(\mathbf{z})$ and estimate:

$$p(\mathbf{x}_i) \approx \frac{1}{m} \sum_{j=1}^{m} \frac{p(\mathbf{x}_i \mid \mathbf{z}_j)}{q(\mathbf{z}_j)} \cdot p(\mathbf{z}_j).$$

Claim: Ideal choice for $q_i(z)$ is $p(z \mid x_i)$.

We can choose a different distribution for each $\mathbf{x}_i$. I.e., choose $(q_1, \ldots, q_n.)$ Ideally, want q_i to be higher for $\mathbf{z}$ that are more likely to generate $\mathbf{x}_i$. Ideal choice is $q_i(\mathbf{z}) = p(\mathbf{z} \mid \mathbf{x}_i)$.

$$p(x_i) \approx \frac{1}{m} \sum_{j=1}^{m} \frac{p(\mathbf{x}_i \mid \mathbf{z}_j)}{q_i(\mathbf{z}_j)} \cdot p(\mathbf{z}_j) = \frac{1}{m} \sum_{j=1}^{m} \frac{p(\mathbf{x}_i \mid \mathbf{z}_j) \cdot p(\mathbf{z}_j)}{p(\mathbf{z}_j \mid \mathbf{x}_i)}$$

$r = p(x_i, z_j)$

$$= \frac{1}{m} \sum_{j=1}^{m} p(x_i)$$

$z_j \sim q$

$p(x_i) \cdot p(z_j \mid x_i) = p(x_i, z_j)$

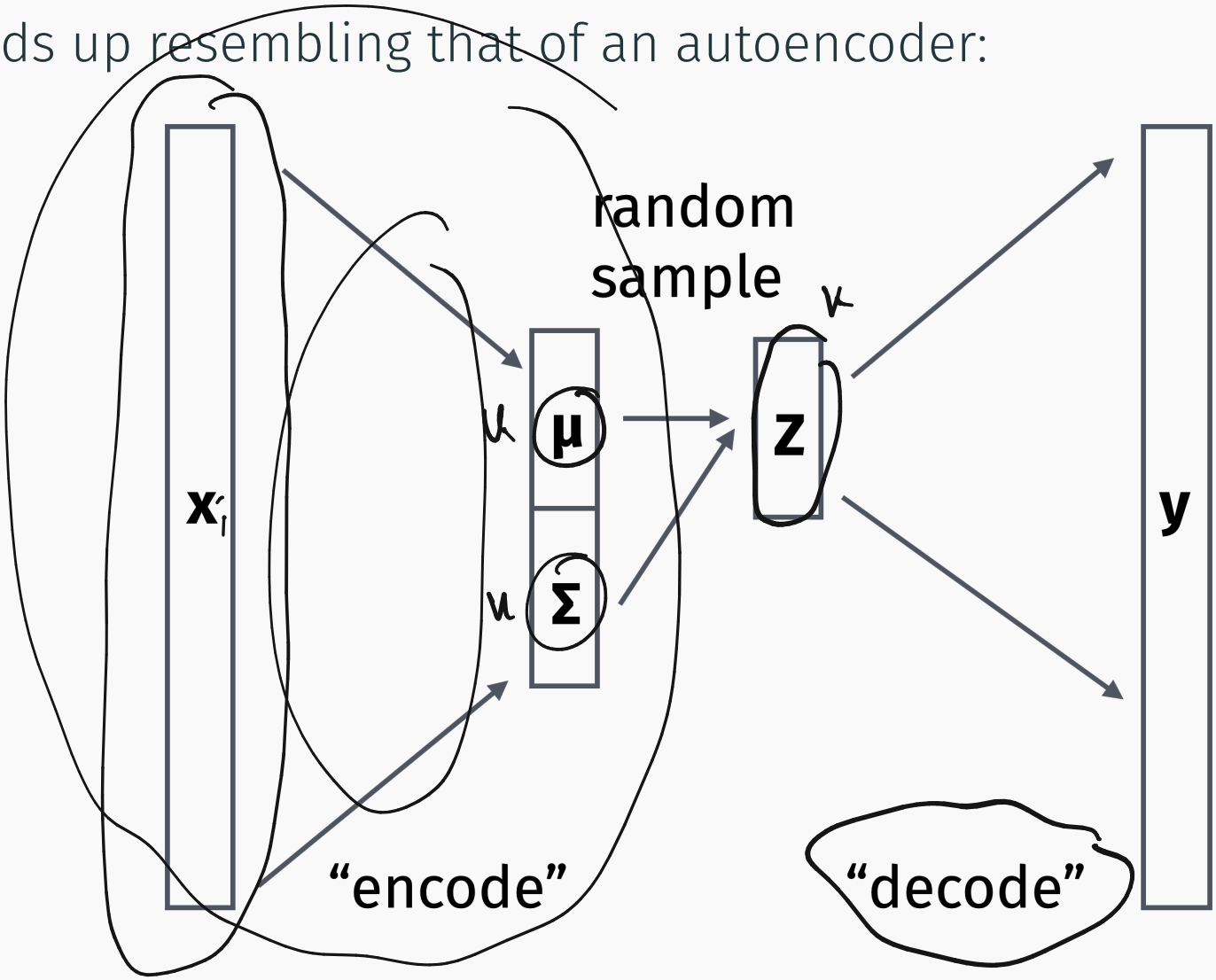
$p(z_j \mid x_i)$

Typical VAE approach: Assume q_i is parameterized as a multivariate Gaussian distribution with mean $\boldsymbol{\mu}_i \in \mathbb{R}^k$ and variances $\boldsymbol{\Sigma}_i = [\sigma_1^2, \ldots, \sigma_k^2]$. Train a model (e.g., neural network) that maps $\mathbf{x}_i$ to $\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i$.

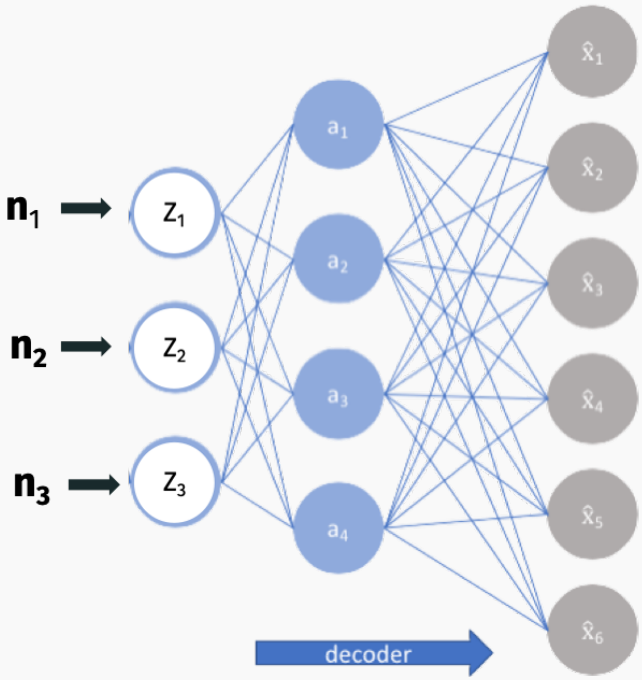
Simulateously minimize distance between q_i and $p(\mathbf{z} \mid \mathbf{x}_i)$ (typically using KL divergence) and maximize $\sum_{i=1}^{n} p(\mathbf{x}_i)$, where $p(\mathbf{x}_i)$ is approximated via importance sampling.

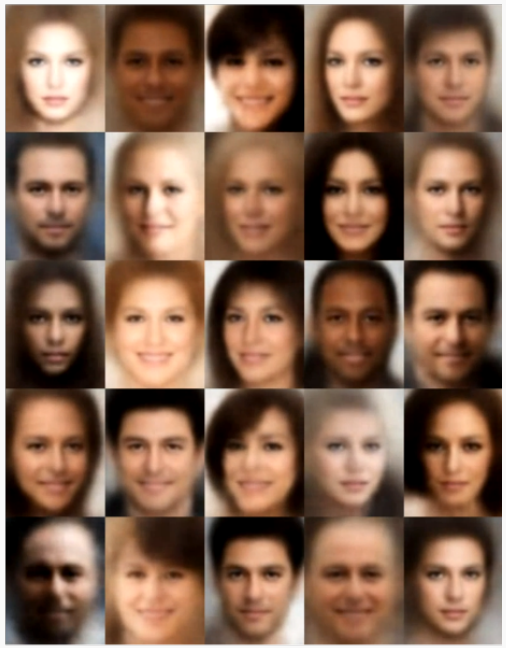
Lots of details here! Link to some good notes by Brian Kang.

VAEs are not really autoencoders. Not designed to map an input $\mathbf{x}$ to an approximation $\tilde{\mathbf{x}}$. But, their final architecture ends up resembling that of an autoencoder:



VAE's give very good results, but tend to produce images with immediately recognizable flaws (e.g. soft edges, high-frequency artifacts).



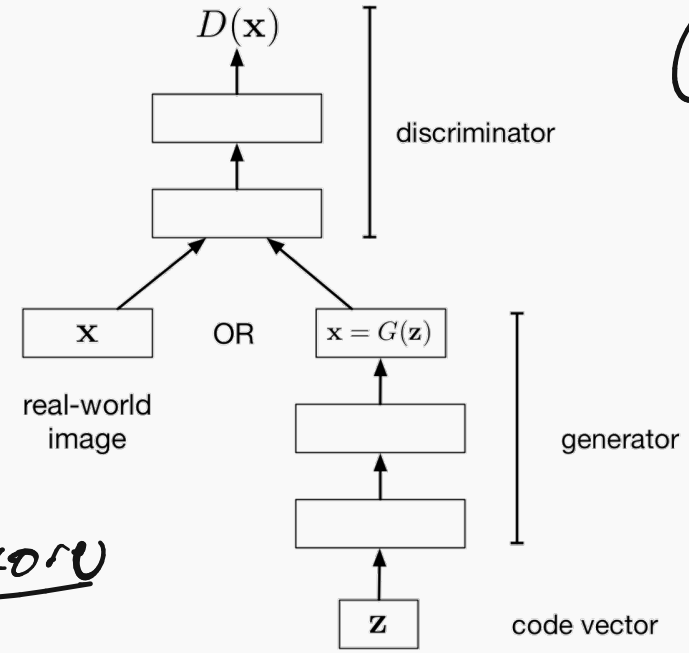


Lots of efforts to hand-design regularizers that penalize images that don't look realisitic to the human eye.

Main idea behind GANs: Use machine learning to automatically encourage realistic looking images.

$$\min_{\theta} L(\theta) + P(\theta)$$

GENERATIVE ADVERSARIAL NETWORKS (GANS)

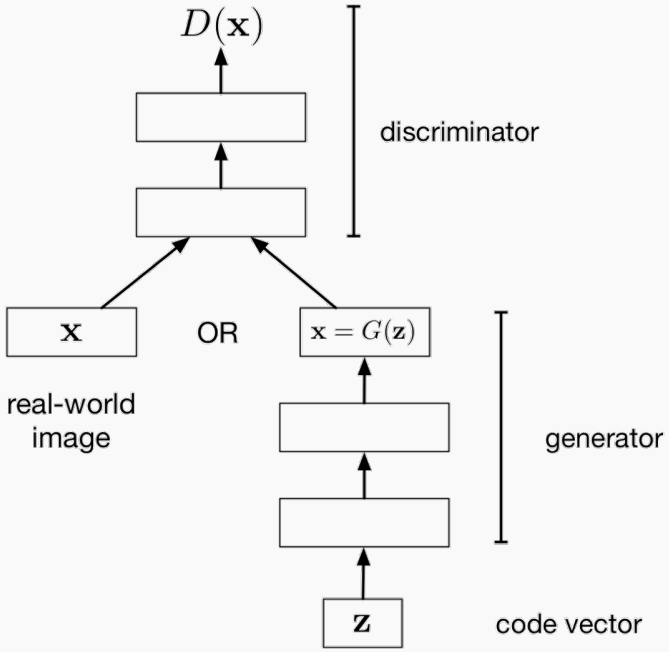


Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be real images and let $\mathbf{z}_1, \ldots, \mathbf{z}_m$ be random code vectors. The goal of the discriminator is to output a number between $[0, 1]$ which is close to 0 if the image is fake, close to 1 if it's real.

Train weights of discriminator $D_{\boldsymbol{\theta}}$ to minimize:

$$\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} -\log\left(D_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + \sum_{i=1}^{m} -\log\left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}'}(\mathbf{z}_i))\right)$$

GENERATIVE ADVERSARIAL NETWORKS (GANS)



Goal of the generator $G_{\theta'}$ is the opposite. We want to maximize:

$$\max_{\theta'} \sum_{i=1}^{m} -\log\left(1 - D_{\theta}(G_{\theta'}(z_i))\right)$$

This is called an "adversarial loss function". D is playing the role of the adversary.

$$\boldsymbol{\theta}^*, \boldsymbol{\theta}'^* \text{ solve } \left(\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\theta}'} \sum_{i=1}^{n} -\log\left(D_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + \sum_{i=1}^{m} -\log\left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}'}(\mathbf{z}_i))\right) \right)$$

This is called a minimax optimization problem. Really tricky to solve in practice.

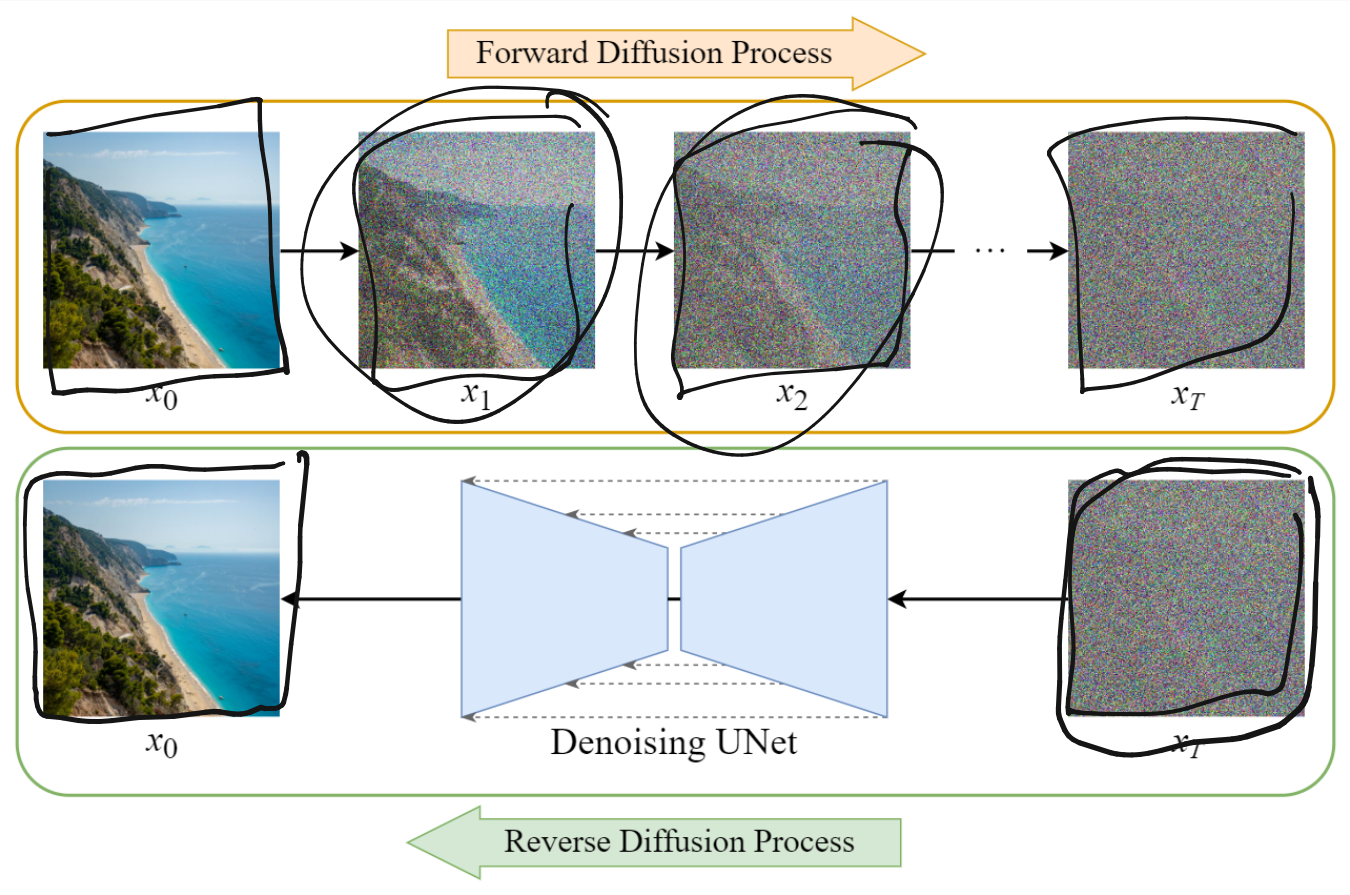
- **Repeatedly play:** Fix one of $\boldsymbol{\theta}$ or $\boldsymbol{\theta}'$, train the other to convergence, repeat.
- **Simultaneous gradient descent:** Run a single gradient descent step for each of $\boldsymbol{\theta}, \boldsymbol{\theta}'$ and update D and G accordingly. Difficult to balance learning rates.

Generative adversarial networks (GANs)

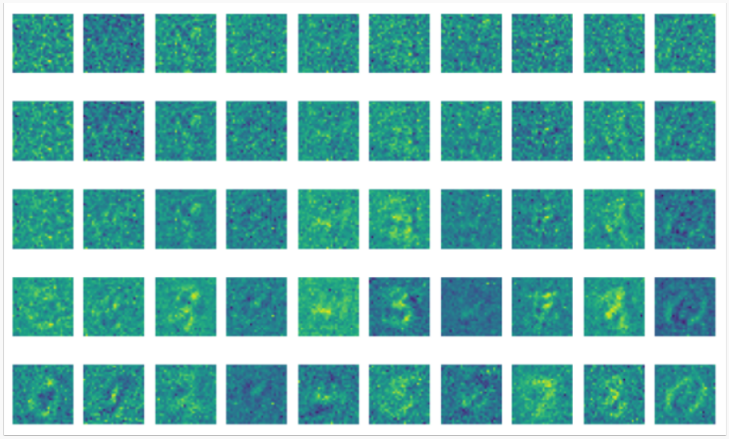
State of the art until a few years ago.

Auto-encoder/GAN approach: Input noise, map directly to image.

Diffusion: Slowly move from noise to image.



We will post a demo for generating MNIST digits via diffusion.

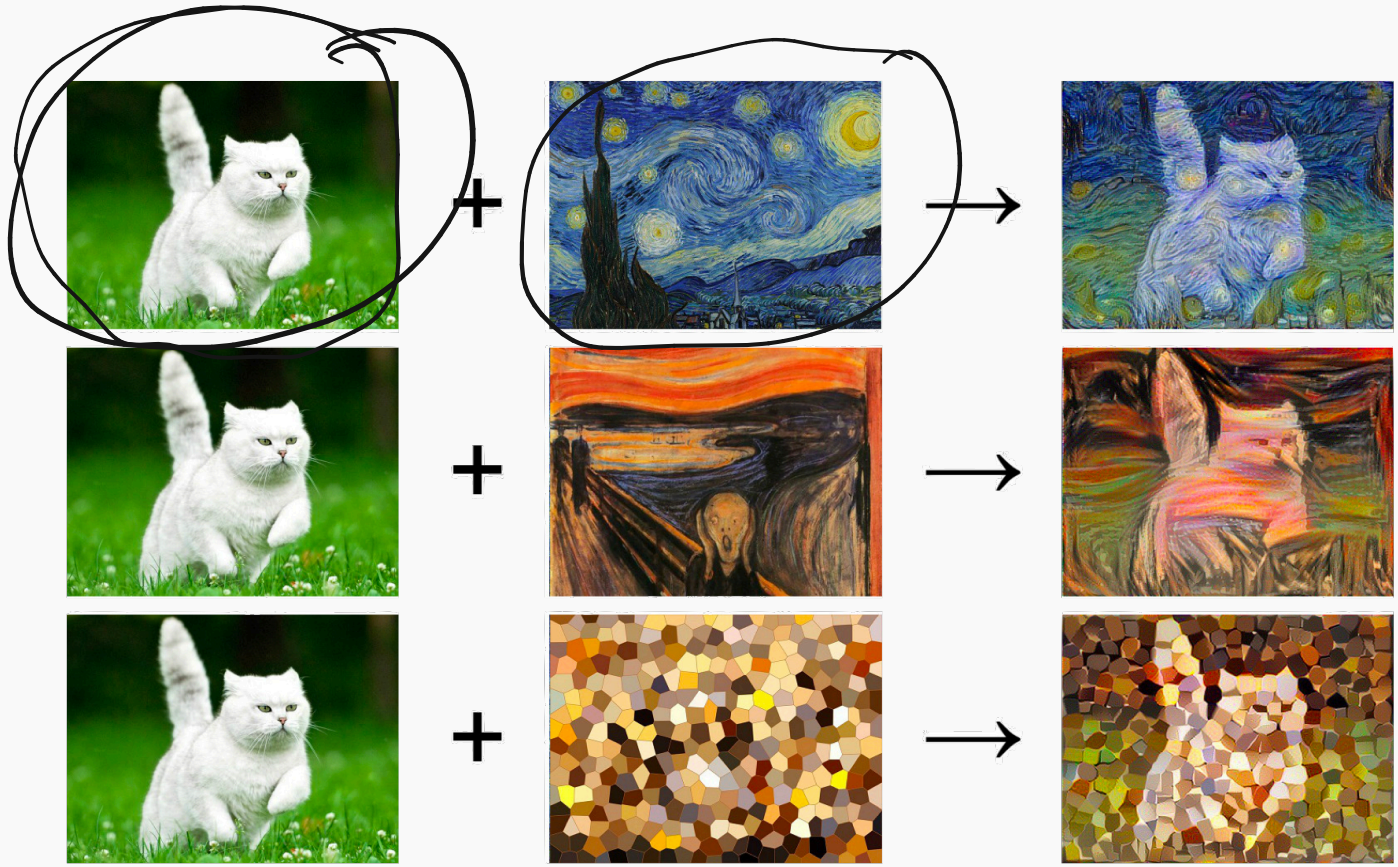


WHAT ELSE?

Tons of other work going on in image generation. One key topic is “class conditioned” generation:

SEMANTIC EMBEDDINGS + GENERATIVE MODELS

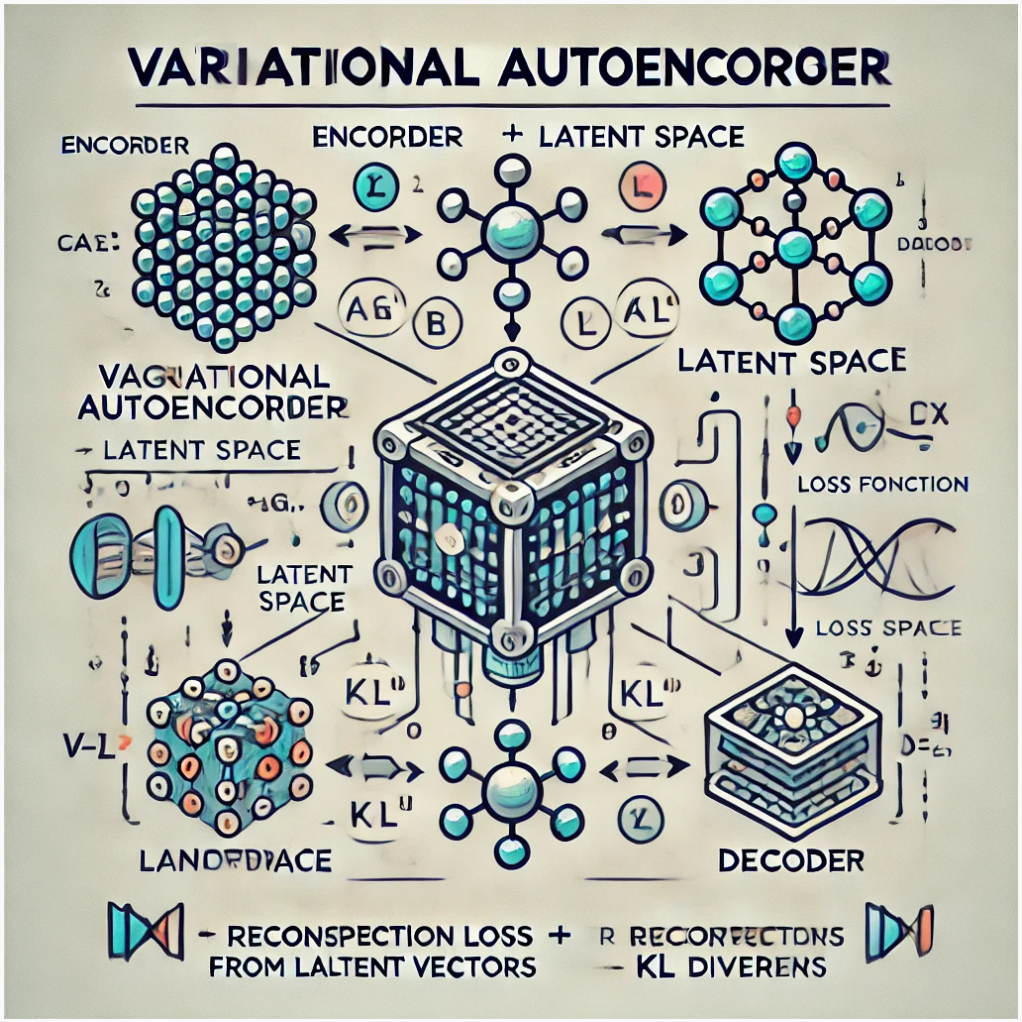
Can also condition on another image...



Or a sentence...

"A chair that looks like an avocado"

Or a sentence...



"A diagram that explains variational autoencoders"

REINFORCEMENT LEARNING (TEASER)

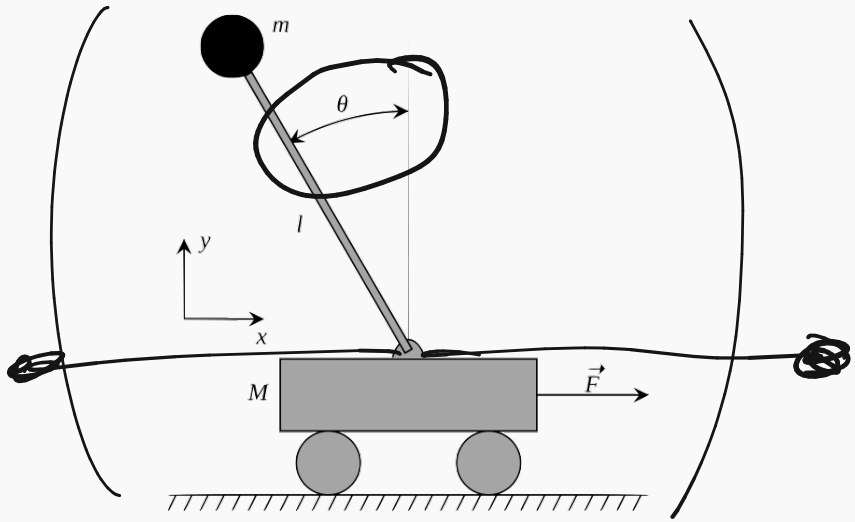
Rest of lecture: Give flavor of the area and insight into one algorithm (Q-learning) which has been successful in recent years.

Basic setup:

- **Agent** interacts with **environment** over time $1, \ldots, t$.
- Takes repeated sequence of **actions**, $a_1, \ldots, a_t$ which effect the environment.
- **State** of the environment over time denoted $s_1, \ldots, s_t$.
- Earn **rewards** $r_1, \ldots, r_t$ depending on actions taken and states reached.
- Goal is to maximize reward over time.

REINFORCEMENT LEARNING EXAMPLES

Classic inverted pendulum problem:

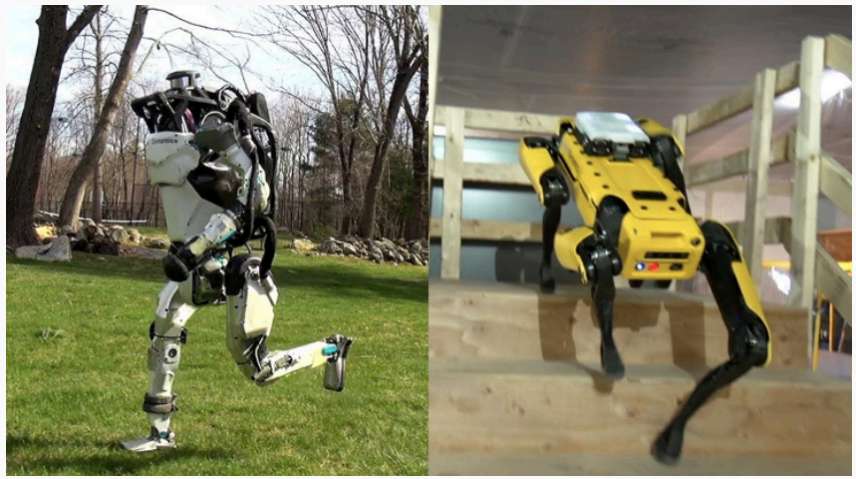


- **Agent:** Cart/software controlling cart.
- **State:** Position of the car, pendulum head, etc.
- **Actions:** Move cart left or move right.
- **Reward:** 1 for every time step that $|\theta| < 90°$ (pendulum is upright). 0 when $|\theta| = 90°$

REINFORCEMENT LEARNING EXAMPLES

This problem has a long history in **Control Theory.** Other applications of classical control:

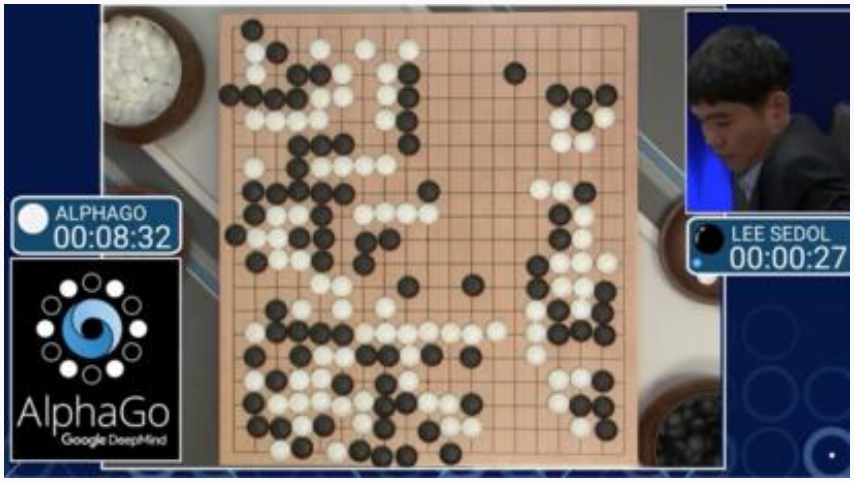
- Semi-autonomous vehicles (airplanes, helicopters, drones, etc.)
- Industrial processes (e.g. controlling large chemical reactions)
- Robotics



control theory : reinforcement learning :: stats : machine learning

REINFORCEMENT LEARNING EXAMPLES

Strategy games, like Go:



- **State:** Position of all pieces on board.
- **Actions:** Place new piece.
- **Reward:** 1 if in winning position at time t. 0 otherwise.

This is a sparse reward problem. Payoff only comes after many times steps, which makes the problem very challenging.

REINFORCEMENT LEARNING EXAMPLES

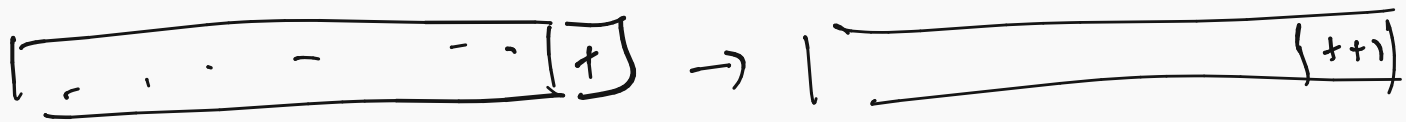
Video games, like classic Atari games:

- **State:** Raw pixels on the screen (sometimes there is also hidden state which can't be observed by the player).
- **Actions:** Actuate controller (up,down,left,right,click).
- **Reward:** 1 if point scored at time t.

Model problem as a **Markov Decision Process (MDP)**:

- $\mathcal{S}$: Set of all possible states. # of states is $|\mathcal{S}|$.
- $\mathcal{A}$: Set of all possible actions. # of actions is $|\mathcal{A}|$.
- $\mathcal{R}$: Set of possible rewards. Could have $\mathcal{R} = \mathbb{R}$.
- **Reward function**
  $R(s, a) : \mathcal{S} \times \mathcal{A} \to$ probability distribution over $\mathcal{R}$. $r_t \sim R(s_t, a_t)$.
- **State transition function**
  $P(s, a) : \mathcal{S} \times \mathcal{A} \to$ probability distribution over $\mathcal{S}$. $s_{t+1} \sim P(s_t, a_t)$.

Why is this called a Markov decision process? What does the term Markov refer to?



Goal: Find a **policy** $\Pi : \mathcal{S} \to \mathcal{A}$ from states to actions which maximize expected cumulative reward.

- Start is state s_0.
- For $t = 0 \ldots, T$
  - $r_t \sim R(s_t, \Pi(s_t))$.
  - $s_{t+1} \sim P(s_t, \Pi(s_t))$.

The **time horizon** T could be short (game with fixed number of steps), very long (stock investing), or infinite. Goal is to maximize:

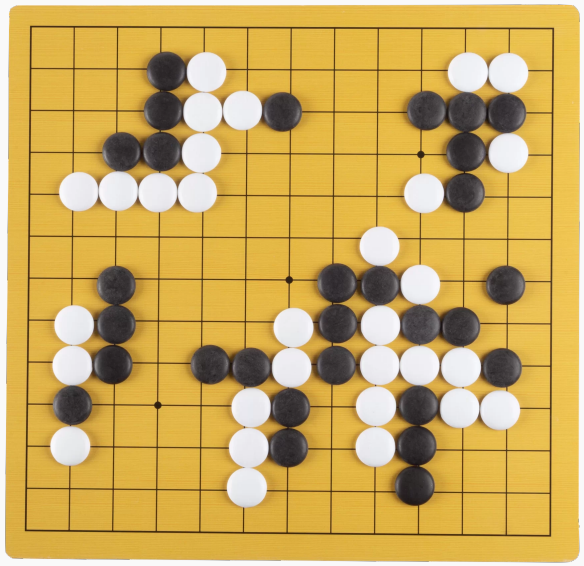
$$reward(\Pi) = \mathbb{E} \sum_{t=0}^{T} r_t$$

$[s_0, a_0, r_0], [s_1, a_1, r_1], \ldots, [s_t, a_t, r_t]$ is called a **trajectory** of the MDP under policy Π.[2]

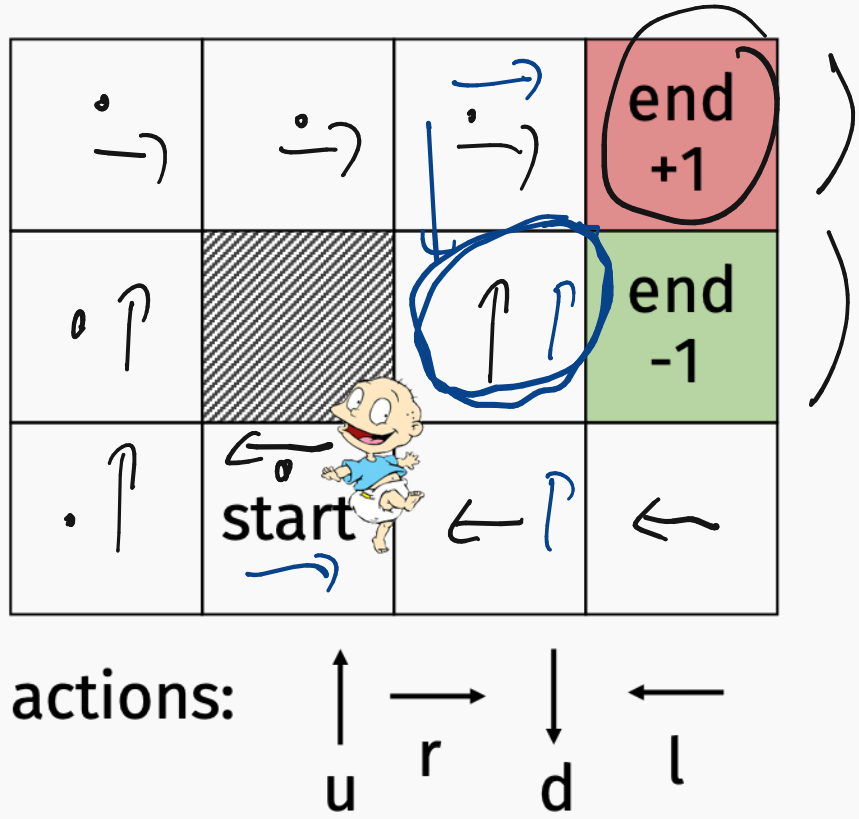
[2] It turns out that it is always optimal to use a fixed policy. There is no benefit to changing Π over time. We will discuss this shortly.

FLEXIBILITY OF MDPS

- Can be used to model time-varying environments. Just add time t to the state vector.
- Can be used to model games where actions have different effect if play in sequence (e.g. combo in a video game). Just add list of previous few actions to state.
- Can be used to model two-player games. Model adversary as part of the transition function.



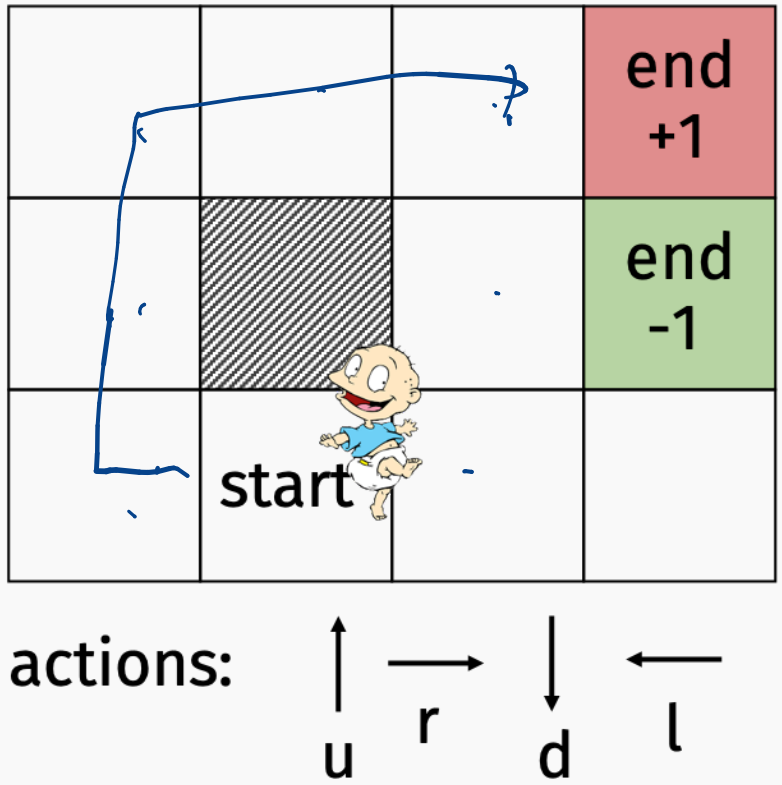
SIMPLE EXAMPLE: GRIDWORLD



- $r_t = -.01$ if not at an end position. ± 1 if at end position.
- $P(s_t, a)$: 70% of the time move in the direction indicated by a. 30% of the time move in a random direction.

What is the optimal policy Π?

SIMPLE EXAMPLE: GRIDWORLD



- $r_t = -.5$ if not at an end position. ± 1 if at end position.
- $P(s_t, a)$: 70% of the time move in the direction indicated by a. 30% of the time move in a random direction.

What is the optimal policy Π?

DISCOUNT FACTOR

For infinite or very long times horizon games (large T), we often introduce a **discount factor** γ and seek instead to take actions which minimize:

$$\mathbb{E} \sum_{t=0}^{T} \gamma^t r_t$$

where $r_t \sim R(s_t, \Pi(s_t))$ and $s_{t+1} \sim P(s_t, \Pi(s_t))$ as before.

$\gamma \to 1$: No discount. Standard MDP expected reward.

$\gamma \to 0$: Care about short term reward more.

From now on assume $T = \infty$. We can do this without loss of generality by adding a time parameter to state and moving into an "end state" with no additional rewards once the time hits T.

Value function: Measures the expected return if we start in state s and follow policy Π.

$$V^{\Pi}(s) = \mathbb{E}_{\Pi, s_0 = s} \sum_{t \geq 0} \gamma^t r_t$$

Let $\Pi_s^* = \arg\max V^{\Pi}(s)$. If we are in state s, at any point, we should always take action $\Pi_s^*(s)$.

Value function:

$$V^{\Pi}(s) = \mathbb{E}_{\Pi, s_0 = s} \sum_{t \geq 0} \gamma^t r_t$$

Claim: Let $\Pi_s^* = \arg\max V^{\Pi}(s)$. If we are in state s, at any point, we should always take action $\Pi_s^*(s)$.

Proof: Suppose we has already taken $j-1$ steps and seen trajectory $[s_0, a_0, r_0], \ldots, [s_j, a_j, r_j]$. Then our expected reward is:

$$\begin{aligned} & r_0 + \gamma r_1 + \ldots + \gamma^{j-1} r_{j-1} + \mathbb{E}_{\Pi} \sum_{t \geq j} \gamma^t r_j \\ &= r_0 + \gamma r_1 + \ldots + \gamma^{j-1} r_{j-1} + \gamma^j \cdot \mathbb{E}_{\Pi} \sum_{t \geq 0} \gamma^t r_{t+j} \\ &= r_0 + \gamma r_1 + \ldots + \gamma^j r_j + \gamma^j \cdot V^{\Pi}(s_j) \end{aligned}$$

Value function:

$$V^{\Pi}(s) = \mathbb{E}_{\Pi, s_0 = s} \sum_{t \geq 0} \gamma^t r_t$$

Claim: Let $\Pi_s^* = \arg\max V^{\Pi}(s)$. If we are in state s, at any point, we should always take action $\Pi_s^*(s)$.

Consequence: there is a single optimal policy Π^* which simultaneously maximizes $V^{\Pi}(s)$ for all s. I.e. $\Pi_1^* = \Pi_2^* = \ldots = \Pi_{|S|}^* = \Pi^*$. We do not need to change the policy over time to maximize expected reward.

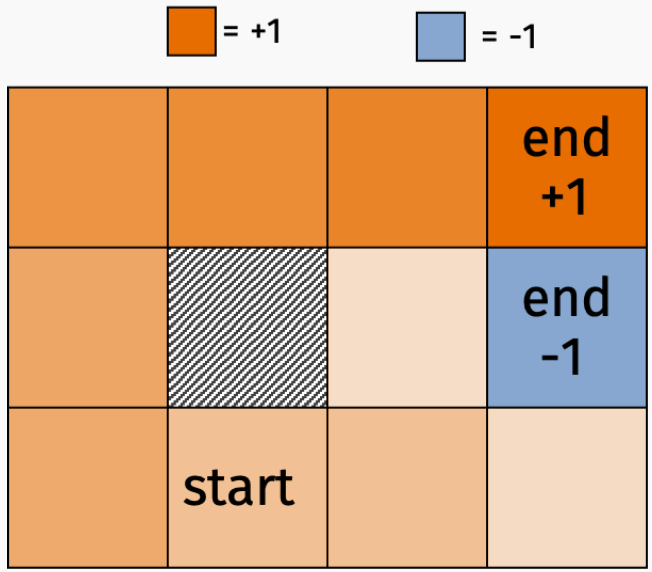
Goal in RL is to find this optimal policy Π^*.

TWO SETTINGS

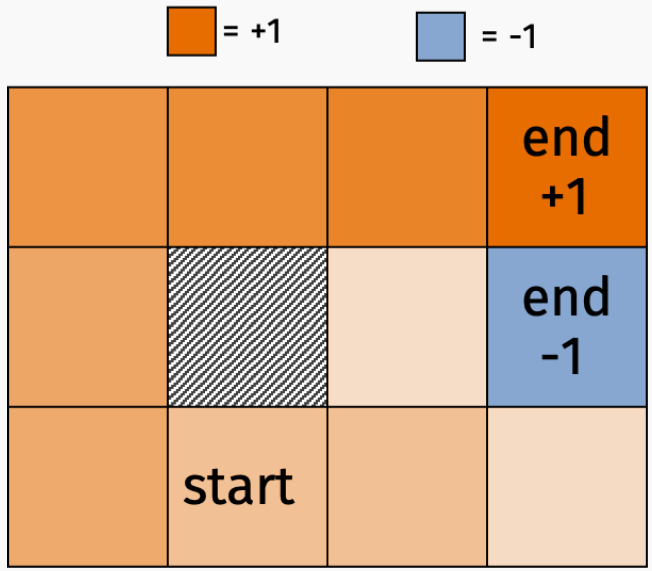
Full information: We know $\mathcal{S}$, $\mathcal{A}$, the transition function P and reward function R. Sometimes called the "planning" problem.

Reinforcement Learning setting: We do not know P or R, but we can repeatedly play the MDP, running whatever policy we like.

Let $V^*(s) = V^{\Pi^*}(s)$. This function is equal to the expected future reward if we play optimally starting in state s.



In the full information setting, if we knew V^* we can easily find the optimal policy Π:



$$\Pi^*(s) = \arg\max_a \sum_{s',r} \cdot \Pr(s', r \mid s, a)[r + \gamma V^*(s')]$$

VALUE ITERATION

$V^*(s)$ satisfies what is called a Bellman equation:

$$V^*(s) = \max_a \sum_{s',r} \cdot \Pr(s', r \mid s, a)[r + \gamma V^*(s')]$$

Run a fixed point iteration to find V^*:

- Start with initial guess V^0.
- For $i = 1, \ldots, z$:
  - For $s \in \mathcal{S}$:
    - $V^i(s) = \max_a \sum_{s',r} \cdot \Pr(s', r \mid s, a)[r + \gamma V^{i-1}(s')]$

Can be shown to converge in roughly $z = \frac{1}{1-\gamma}$ iterations. What is the computational cost of each iteration?

TWO SETTINGS

Full information: We know $\mathcal{S}$, $\mathcal{A}$, the transition function P and reward function R.

Reinforcement Learning setting: We do not know P or R, but we can repeatedly play the MDP, running whatever policy we like.

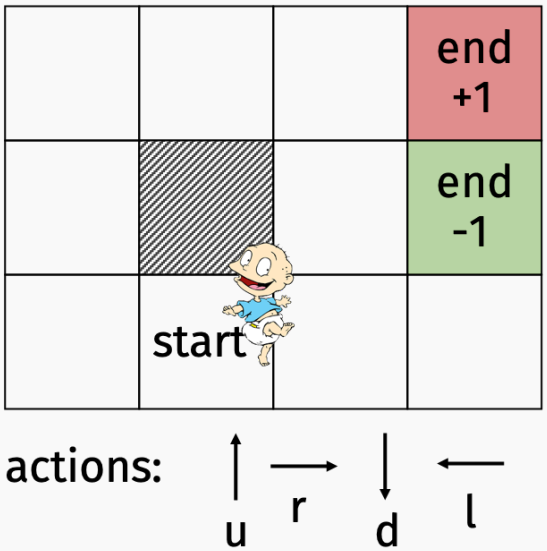
- **Model-based** RL methods essentially try to learn P and R very accurately and then find Π^* via a method like value iteration. Require a lot of samples of the MDP.
- **Model-free** RL methods try to learn Π^* without necessarily obtaining an accurate model of the world – i.e. without explicitly learning P and R.

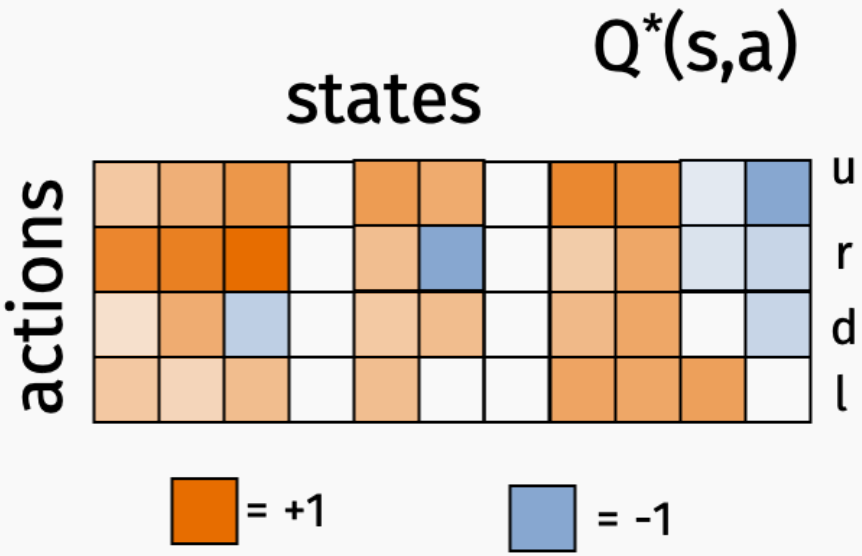
Q FUNCTION

Another important function:

- **Q-function:** $Q^{\Pi}(s,a) = \mathbb{E}_{\Pi, s_0=s, a_0=a} \sum_{t \geq 0} \gamma^t r_t$. Measures the expected return if we start in state s, play action a, and then follow policy Π.

$$Q^*(s,a) = \max_{\Pi} Q^{\Pi}(s,a) = Q^{\Pi^*}(s,a).$$

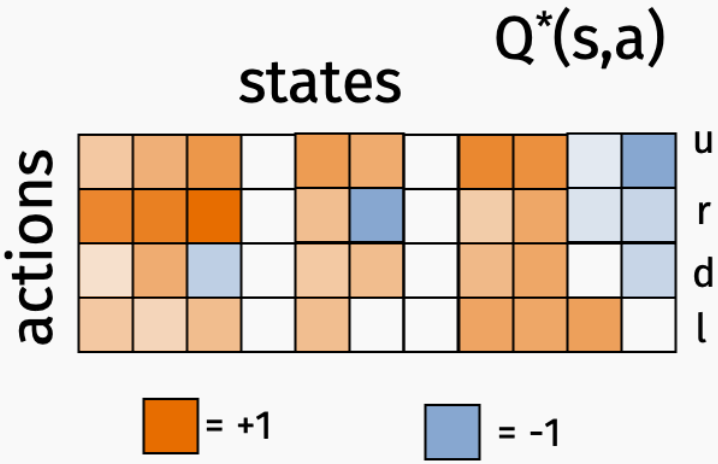




Q FUNCTION

$$Q^*(s,a) = \max_{\Pi} \mathbb{E}_{\Pi, s_0=s, a_0=a} \sum_{t \geq 0} \gamma^t r_t.$$

If we knew the function Q^*, we would immediately know an optimal policy. Whenever we're in state s, we should always play action $a^* = \arg\max_a Q^*(s,a)$.



Q has more parameters than V, but you can use it to determine an optimal policy without knowing transition probabilities.

BELLMAN EQUATION

Q^* also satisfies a Bellman equation:

$$Q^*(s,a) = \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{s' \sim P(s,a)} \max_{a'} Q^*(s',a').$$

Bellman equation:

$$Q^*(s,a) = \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{s' \sim P(s,a)} \max_{a'} Q^*(s',a').$$

Again use fixed point iteration to find Q^*. Let Q^{i-1} be our current guess for Q^* and suppose we are at some state s, a.

$$Q^i(s,a) = \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{s' \sim P(s,a)} \max_{a'} Q^{i-1}(s',a')$$

In reality, drop expectations and use a learning rate α

$$Q^i(s,a) = (1-\alpha) Q^i(s,a) + \alpha \left(R(s,a) + \gamma \max_{a'} Q^{i-1}(s',a') \right)$$

How do we choose states s and a to make the update for? In principal you can do anything you want! E.g. choose some policy Π and run:

- Initialize Q^0 (e.g. all zeros)
- Start at s, play action $a = \Pi(s)$, observe reward $R(s, a)$.
- For $i = 1, \ldots, z$
  - $Q^i(s, a) = (1 - \alpha)Q^i(s, a) + \alpha \left(R(s, a) + \gamma \max_{a'} Q^{i-1}(s', a')\right)$
  - $s \leftarrow P(s, a)$
  - $a \leftarrow \Pi(s)$

  (restart if we reach a terminating state)

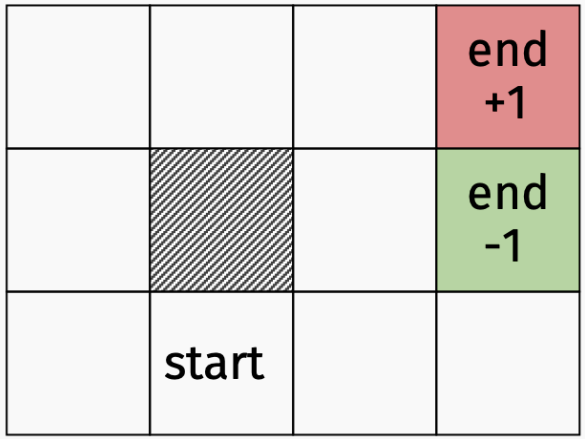
Q-learning is considered an **off-policy** RL method because it runs a policy Π that is not necessarily related to its current guess for an optimal policy, which in this case would be $\Pi(s) = \max_a Q^i(s, a)$ at time i.

EXPLORATION VS. EXPLOITATION

For small enough α, Q-learning converges to Q^* as long as we follow a policy Π that visits every start (s, a) with non-zero probability.

Mild condition, but exact choice of Π matters for convergence rate.

- **Random:** At state s, choose a random action a.
- **Greedy:** At state s, choose $\arg\max_a Q^i(s, a)$. I.e. the current guess for the best action.

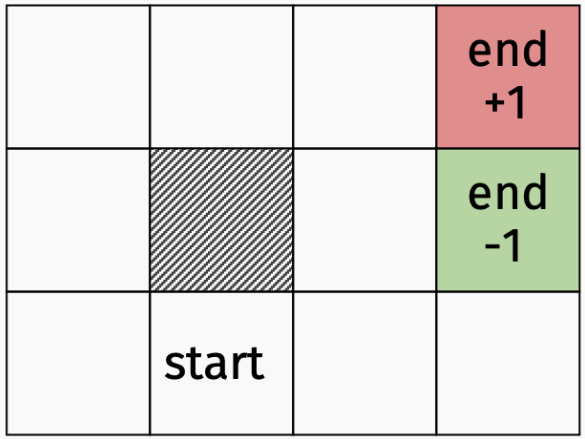


Random can be wasteful. Spend time improving parts of Q that aren't relevant to optimal play. **Greedy** can cause you to zero in on a locally optimal policy without learning new strategies.

EXPLORATION VS. EXPLOITATION

Possible choices for Π:

- **Random:** At state s, choose a random action a.
- **Greedy:** At state s, choose $\arg\max_a Q^i(s,a)$. I.e. the current guess for the best action.
- ϵ-**Greedy:** At state s, choose $\arg\max_a Q^i(s,a)$ with probability $1-\epsilon$ and a random action with probability ϵ.



Exploration-exploitation tradeoff. Increasing ϵ = more **exploration**.

Another issue: Even writing down Q^* is intractable... This is a function over $|\mathcal{S}||\mathcal{A}|$ possible inputs. Even for relatively simple games, $|\mathcal{S}|$ is gigantic...

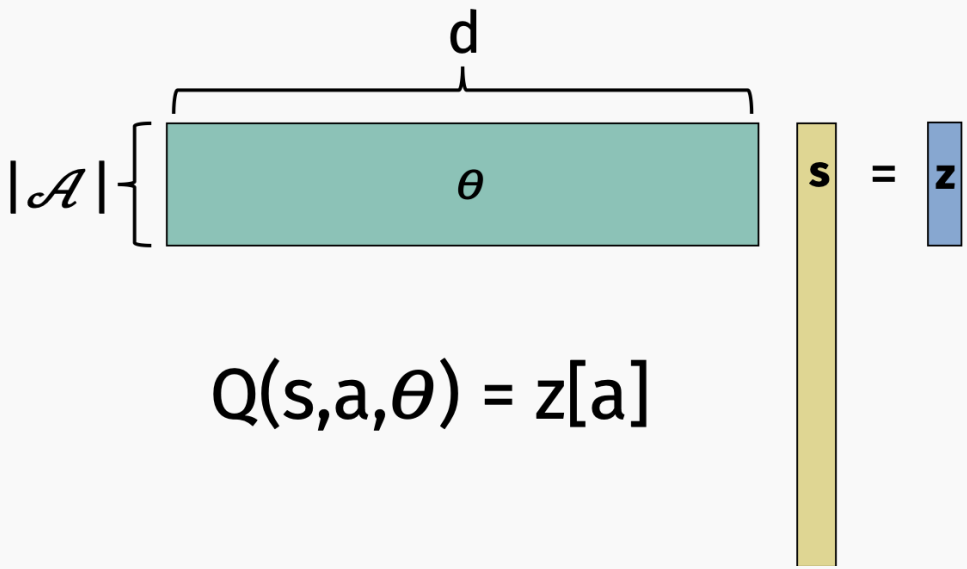
Back of the envelope calculations:

- **Tic-tac-toe:** $3^{(3\times 3)} \approx 20,000$
- **Chess:** $\approx 10^{43} < 28^{64}$ (due to Claude Shannon).
- **Go:** $3^{(19\times 19)} \approx 10^{171}$.
- **Atari:** $128^{(210\times 160)} \approx 10^{71,000}$.

Number of atoms in the universe: $\approx 10^{82}$.

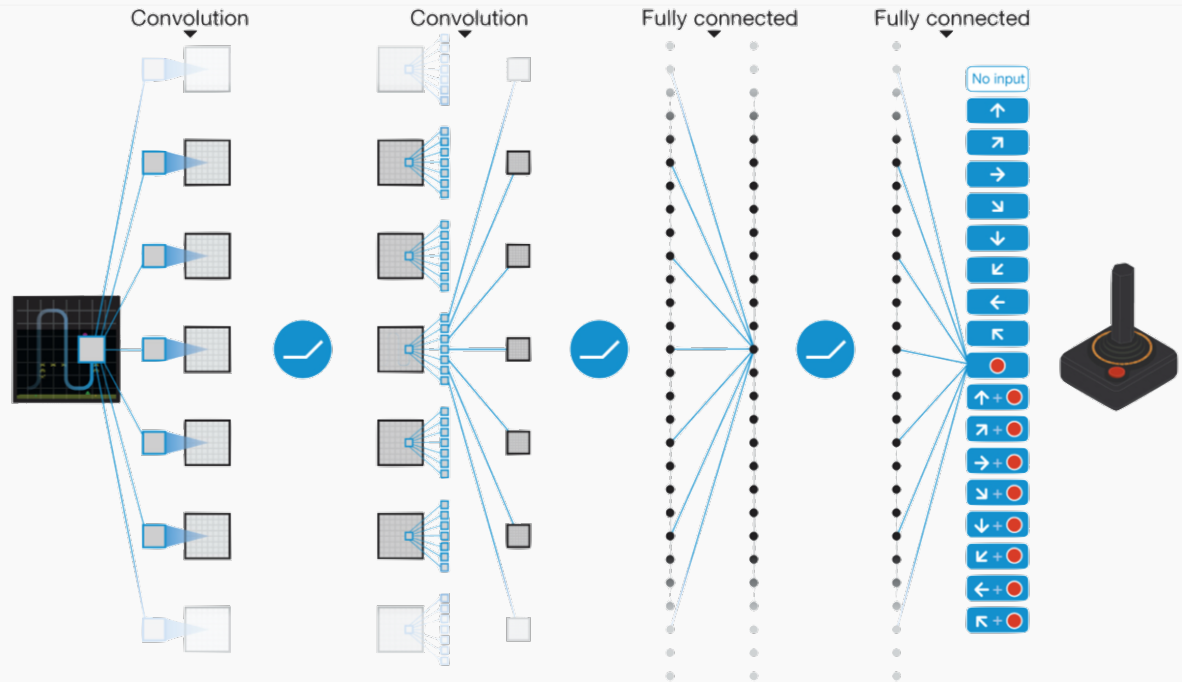
Learn a **simpler** function $Q(s, a, \theta) \approx Q^*(s, a)$ parameterized by a small number of parameters θ.

Example: Suppose our state can be represented by a vector in $\mathbb{R}^d$ and our action a by an integer in $1, \ldots, |\mathcal{A}|$. We could use a linear function where θ is a small matrix:



Learn a **simpler** function $Q(s, a, \theta) \approx Q^*(s, a)$ parameterized by a small number of parameters θ.

Example: Could also use a (deep) neural network.



DeepMind: "Human-level control through deep reinforcement learning", Nature 2015.

If $Q(s, a, \theta)$ is a good approximation to $Q^*(s, a)$ then we have an approximately optimal policy: $\tilde{\Pi}^*(s) = \textbf{arg max}_a\, Q(s, a, \theta)$.

- Start in state s_0.
- For $t = 1, 2, \ldots$
  - $a^* = \textbf{arg max}_a\, Q(s, a, \theta)$
  - $s_t \sim P(s_{t-1}, a^*)$

How do we find an optimal θ? If we knew $Q^*(s, a)$ could use supervised learning, but the true Q function is infeasible to compute.

Q-LEARNING W/ FUNCTION APPROXIMATION

Find θ which satisfies the Bellman equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim P(s,a)} \left[R(s,a) + \gamma \max_{a'} Q^*(s',a') \right]$$

$$Q(s,a,\theta) \approx \mathbb{E}_{s' \sim P(s,a)} \left[R(s,a) + \gamma \max_{a'} Q(s,a,\theta) \right].$$

Should be true for all a, s. Should also be true for $a, s \sim \mathcal{D}$ for any distribution $\mathcal{D}$:

$$\mathbb{E}_{s,a \sim \mathcal{D}} Q(s,a,\theta) \approx \mathbb{E}_{s,a \sim \mathcal{D}} \mathbb{E}_{s' \sim P(s,a)} \left[R(s,a) + \gamma \max_{a'} Q(s,a,\theta) \right].$$

Loss function:

$$L(\theta) = \mathbb{E}_{s,a \sim \mathcal{D}} \left(y - Q(s,a,\theta) \right)^2$$

where $y = \mathbb{E}_{s' \sim P(s,a)} \left[R(s,a) + \gamma \max_{a'} Q(s',a',\theta) \right]$.

Q-learning w/ function approximation

Minimize loss with **gradient descent**:

$$\nabla L(\theta) = \mathbb{E}_{s,a\sim\mathcal{D}}\left[-2\nabla Q(s,a,\theta)\cdot[y - Q(s,a,\theta)]\right]$$

In practice use stochastic gradient:

$$\nabla L(\theta, s, a) = -2\cdot\nabla Q(s,a,\theta)\cdot\left[R(s,a) + \gamma \max_{a'} Q(s',a',\theta) - Q(s,a,\theta)\right]$$

- Initialize θ_0
- For $i = 0, 1, 2, \ldots$
  - Run policy Π to obtain s, a and $s' \sim P(s,a)$
  - Set $\theta_{i+1} = \theta_i - \eta\cdot\nabla L(\theta_i, s, a)$

η is a learning rate parameter.

Again, the choice of Π matters a lot. **Random play** can be wastefully, putting effort into approximating Q^* well in parts of the state-action space that don't actually matter for optimal play. ϵ-greedy approach is much more common:

- Initialize s_0.
- For $t = 0, 1, 2, \ldots,$
  - $a_i = \begin{cases} \arg\max_a Q(s_t, a, \theta_{curr}) & \text{with probabilty } (1 - \epsilon) \\ \text{random action} & \text{with probabilty } \epsilon \end{cases}$

REFERENCES

Lots of other details we don't have time for! References:

- Original DeepMind Atari paper: `https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf`, which is very readable.
- Stanford lecture video: `https://www.youtube.com/watch?v=lvoHnicueoE` and slides: `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf`

Important concept we did not cover: **experience replay**.

ATARI DEMO

https://www.youtube.com/watch?v=V1eYniJ0Rnk