10707
Deep Learning: Spring 2021

Andrej Risteski

Machine Learning Department

Lecture 22:
Other topics in deep learning

Too many interesting topics in deep learning

In the course, we touched on the “core” topics in deep learning.

Plenty of other interesting core topics,

plenty of applications we didn’t speak about!

Today+Wednesday: a mélange of topics that we didn’t get to (each very brief !)

Robustness

Robustness in deep learning

Question: how "sensitive" are the trained neural networks to small changes in inputs / input distributions ?

Why care:

Distribution shifts occur all the time: e.g. a self-driving car may need to adapt to new roads / road conditions / lighting / etc.

Safety concerns: a malicious attacker might try to fool trained system, e.g. cheat a facial recognition software.

Robustness in deep learning

Question: how "sensitive" are the trained neural networks to small changes in inputs / input distributions ?

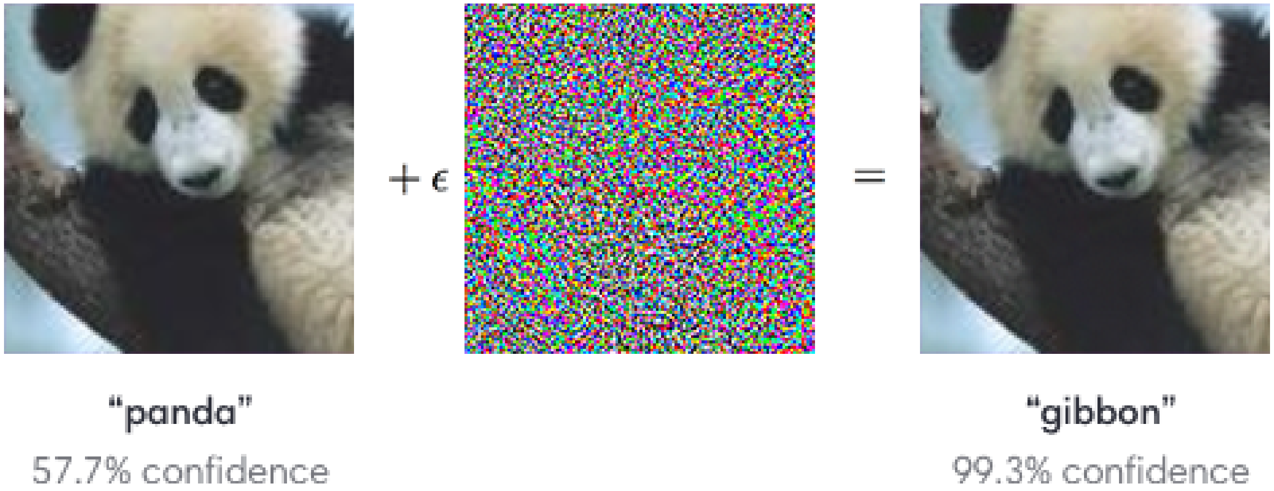
Why care:

Distribution shifts occur all the time: e.g. a self-driving car may need to adapt to new roads / road conditions / lighting / etc.

***Safety concerns*: a malicious attacker might try to fool trained system, e.g. cheat a facial recognition software.**

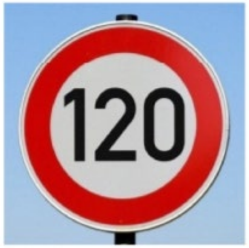
Early observation: neural networks are surprisingly brittle

(*Szegedy et al ‘13*): tiny perturbations *imperceptible* to humans can make a neural network misclassify an image.

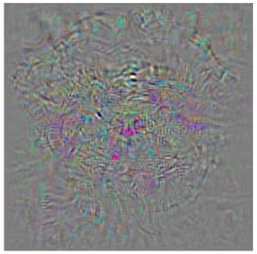


Early observation: neural networks are surprisingly brittle

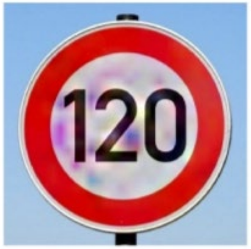
(*Szegedy et al ‘13*): tiny perturbations *imperceptible* to humans can make a neural network misclassify an image.



Speed limit (120)



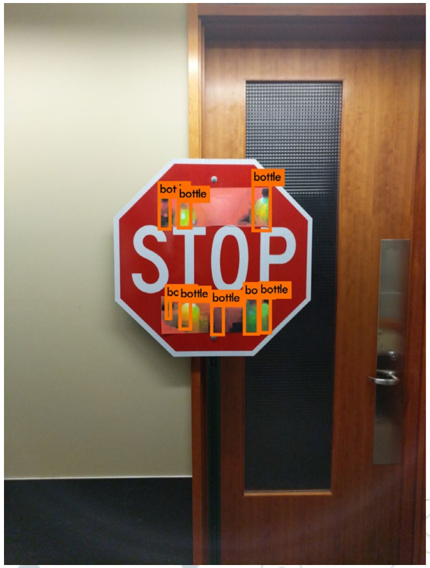
+noise

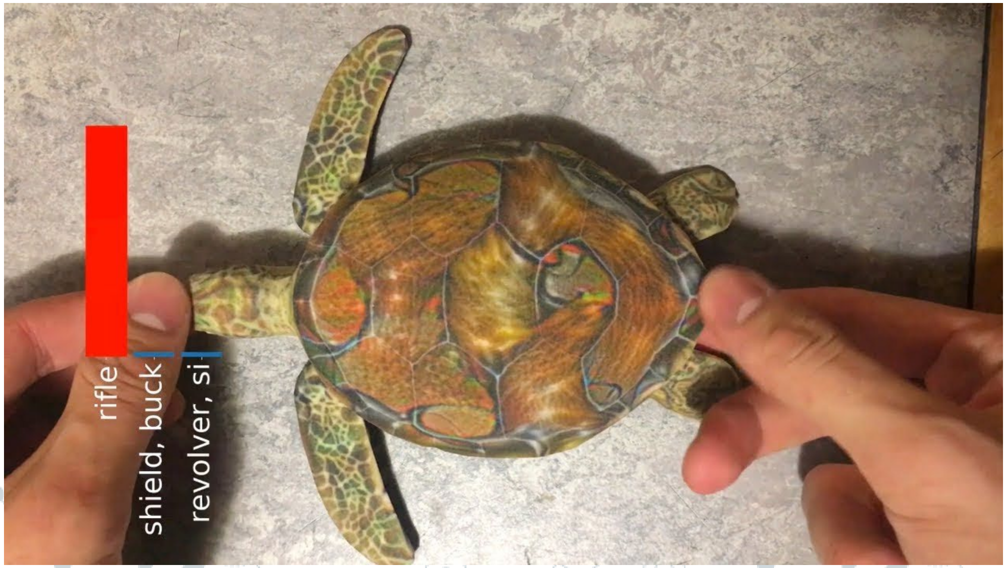


Speed limit (30)

Similar things can be done in a more realistic threat model

Modifying inputs directly may not be realistic, but we can modify objects in the real world. Classic example is sticker on a stop sign.
[Kurakin et al. 2016]





How are these examples produced?

[Szegedy et al. 2013]:
Pass a test input through the network, determine the loss.
Evaluate the gradient of the loss w.r.t. ~~network parameters~~ *the input.*
Take a step in the ~~negative~~ *positive* direction.

$$\mathcal{L}(x) = -y_i \log f_\theta(x)_i$$
$$x = x + \eta \nabla_x \mathcal{L}(x)$$

Turns out this is enough to make the network make an error on this modified input.
These modifications are known as ***Adversarial Examples***.

Why do adversarial examples exist?

Why is this happening? We don’t know (exactly)! A couple theories:

Deep neural networks have very non-linear, non-Lipschitz decision boundaries. In high dimensions, possible to find directions with large gradient. A tiny step in these dimensions means a large change in output.

Deep neural networks don’t “see” like we do. They are machines optimized to extract statistical signal from the noise. So they pick up on patterns that exist but that we just don’t notice. *[Ilyas et al. 2019]*

Mitigation strategies

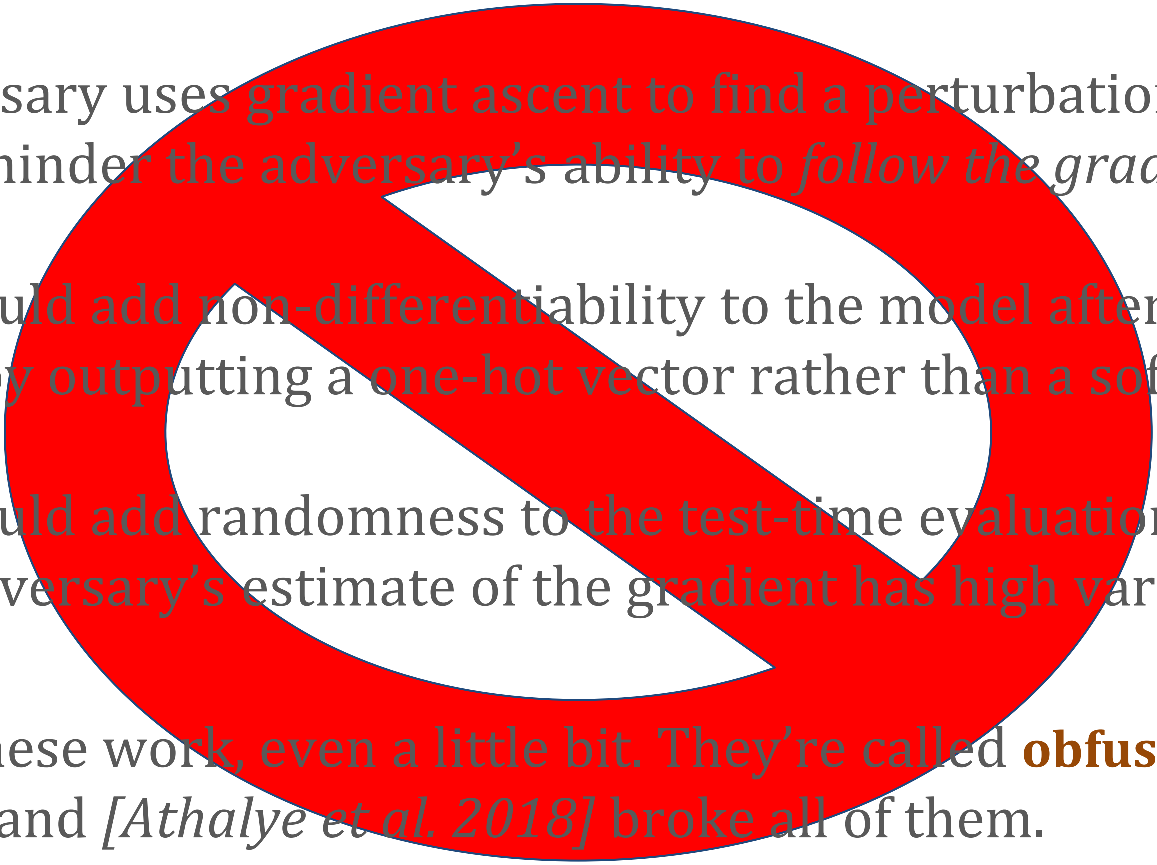
Train neural nets to be robust to a restricted set of adversaries (e.g. adversaries that take a few gradient steps to find an adversarial example).

Reminiscent of crypto: arms race b/w attackers / defenders.

Train neural nets to not have adversarial examples at all (e.g. within a small ball around each example, the classification remains the same.)

A mix of empirical strategies and provable algorithms.
Mixed success on both fronts.

Empirical defense strategies



The adversary uses gradient ascent to find a perturbation. So maybe we could hinder the adversary's ability to *follow the gradient*.

- We could add non-differentiability to the model after training (e.g., by outputting a one-hot vector rather than a softmax).
- We could add randomness to the test-time evaluation, meaning the adversary's estimate of the gradient has high variance.

None of these work, even a little bit. They're called **obfuscated gradients** and *[Athalye et al. 2018]* broke all of them.

So we’ve given up on empirical defenses, right?

Wrong! A whole slew of adaptive defenses have since been released.

k-Winners Take All, Generative Classifiers, ME-Net, Asymmetrical Adversarial Training, Sparse Fourier Transform…

But some of these work, right?

Nope! *[Tramèr et al. 2020]* broke these.

Ok, so now we’re *really* done with empirical defenses, right??

Just one more: “Adversarial Training”

This one actually works (so far, no one has broken it). It’s very simple.

[Goodfellow et al. 2014, Madry et al. 2017]

We can view adversarial examples as a game:

1. Defender learns parameters θ to minimize $E_{x\sim D}L(x;\theta)$ on the test set.
2. Adversary receives input *x~D*, wants to produce δ within an ϵ norm ball to maximize L(x+δ; θ).

Why not modify defender’s objective to allow for this perturbation?
Defender’s objective now becomes:

$$\min_{\theta} \mathbb{E}_{x\sim\mathcal{D}}\left[\max_{\delta:\|\delta\|\leq\epsilon} \mathcal{L}(x+\delta;\theta)\right]$$

Minimax optimization is tough to solve.
Done by attacking *every training point for every batch* during training.
Very slow! A couple follow-up works address this. *[Shafahi et al. 2019]*

Provable Defenses

Clearly, the ML community has a lot to learn from the security community.

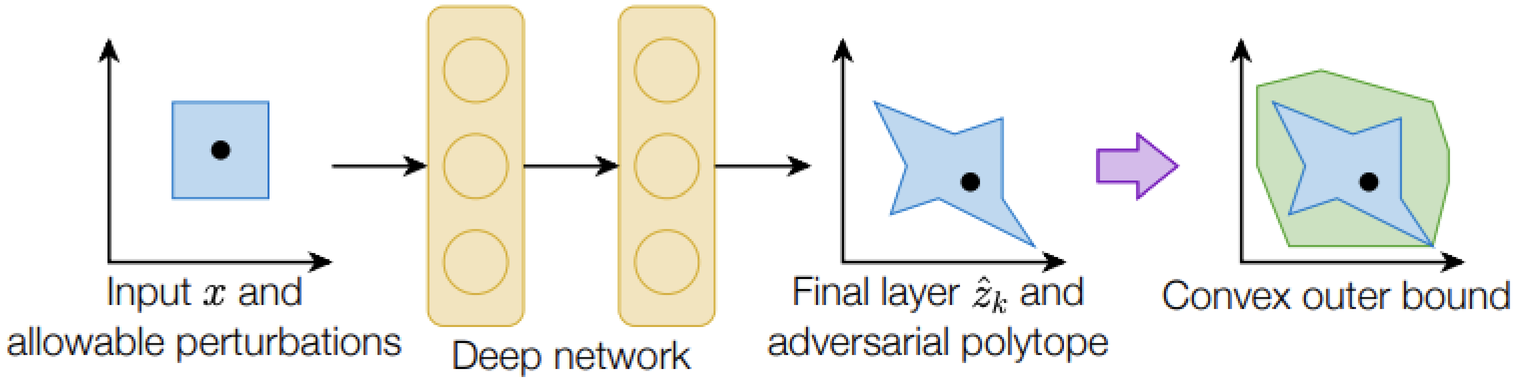
Rule #1: If you can't prove it can't be broken, it can be broken.
Rule #2: Even if you can *prove it can't be broken,* ***it can probably be broken****.*

Obviously we can't guarantee that the network is correct (that's the whole point of machine learning!).
Instead, we can guarantee that the network's decision boundary is not contained within an ϵ-norm ball.

Why is this meaningful?
If we assume adversary is limited to an ϵ-norm perturbation, and decision boundary is not that close, then the input's classification cannot have been modified with a perturbation.
If our classifier is wrong, *it would have been wrong anyways*.

Provable Defenses

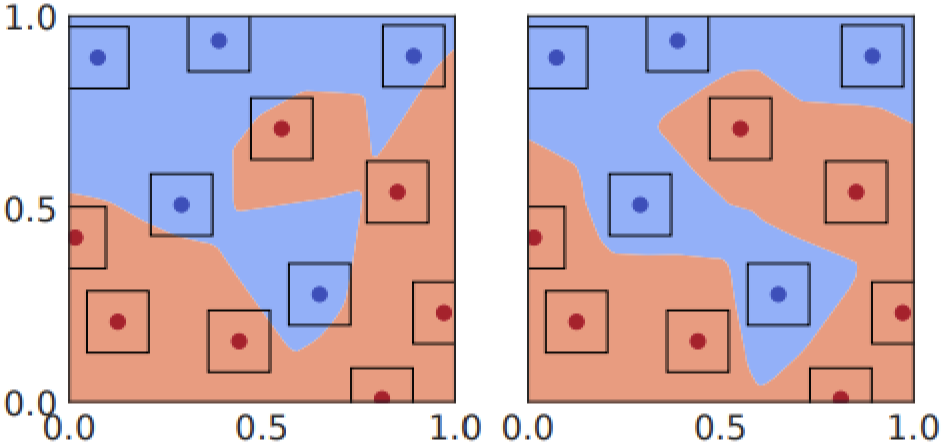
[Wong & Kolter, 2017]: As we pass the input through the network, maintain a convex envelope of possible activations caused by a perturbation. Then train the network to minimize this envelope.


Input x and allowable perturbations
Deep network
Final layer $\hat{z}_k$ and adversarial polytope
Convex outer bound

Called the **Convex Outer Adversarial Polytope**.
How well does this work?

Provable Defenses

[Wong & Kolter, 2017]: As we pass the input through the network, maintain a convex envelope of possible activations caused by a perturbation. Then train the network to minimize this envelope.



Pretty well! What's the catch?

Solving this optimization problem is very expensive, and gets even more so as the network grows. Not really scalable. ☹

Provable Defenses

[Mirman et al., 2018, Zhang et al., 2019]: As we pass the input through the network, maintain an axis-aligned polytope of possible activations caused by a perturbation. Use this to certify the worst possible case

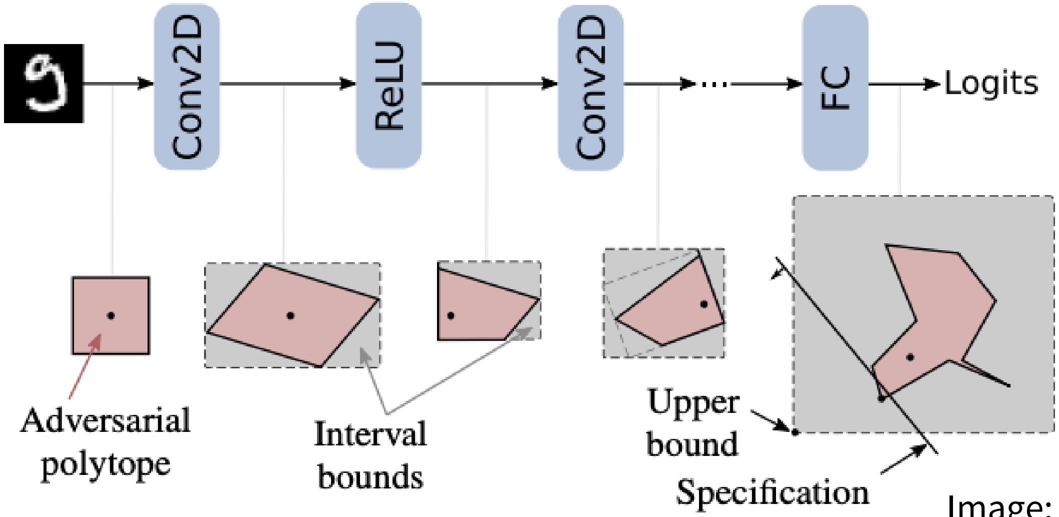


Image: Gowal et al. 2019

This is known as **Interval Bound Propagation**.
Significantly faster, but still can't scale to ImageNet.

Provable Defenses

[Li et al. 2019, Cohen et al., 2019]: Don't try to certify the network f! Instead, define a *new* network *g* whose output is *f* convolved with Gaussian noise:

$$g(x) = \mathbb{E}_{\epsilon \sim \mathcal{N}(0,I)}[f(x+\epsilon)]$$

This new network *g* can be certified as a function of the margin with which the largest softmax class wins.

Known as **Randomized Smoothing**. The decision boundary of *g* is "smoothed" compared to *f*.

Randomized Smoothing

How well does this work? Outperforms others by a large margin. Even better, it's relatively cheap and applies to *any classifier f.* Easily scales to larger networks capable of classifying ImageNet.

A few caveats:

1. You can't actually integrate a neural network's decision regions! Instead, this is approximated via Monte Carlo sampling.

1. Gaussian noise ideal for ℓ_2 norm, but needs modifications to work for other norms. Recent work shows noise must have variance $\mathbf{\Omega}$(d) for ℓ_∞ *[Blum et al. 2020]*.

Handling distribution shifts

Domain generalization/out-of-distribution-generalization: learning models from one (or few) domains that generalize to new (unseen) domains.

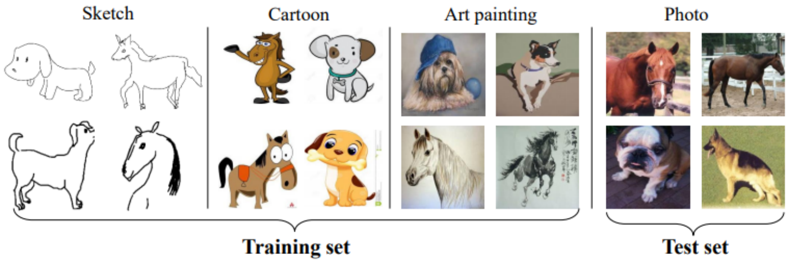


Fig. 1. Examples from the dataset PACS [1] for domain generalization. The training set is composed of images belonging to domains of sketch, cartoon, and art paintings. DG aims to learn a generalized model that performs well on the unseen target domain of photos.

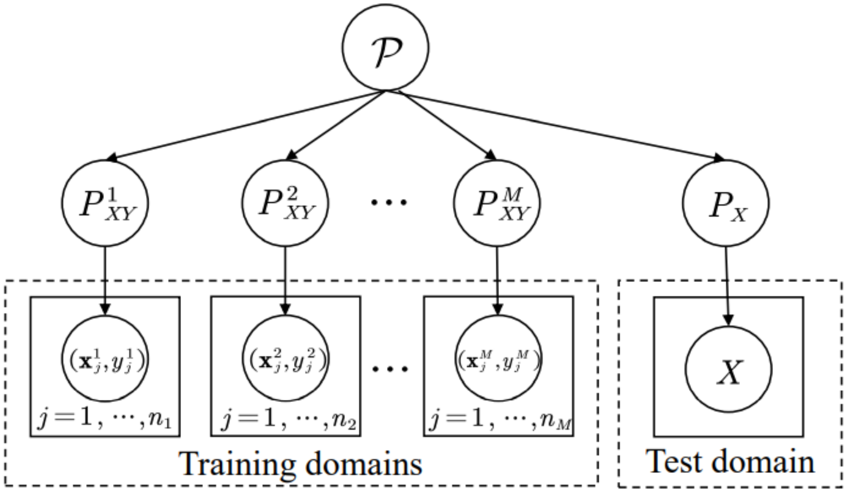
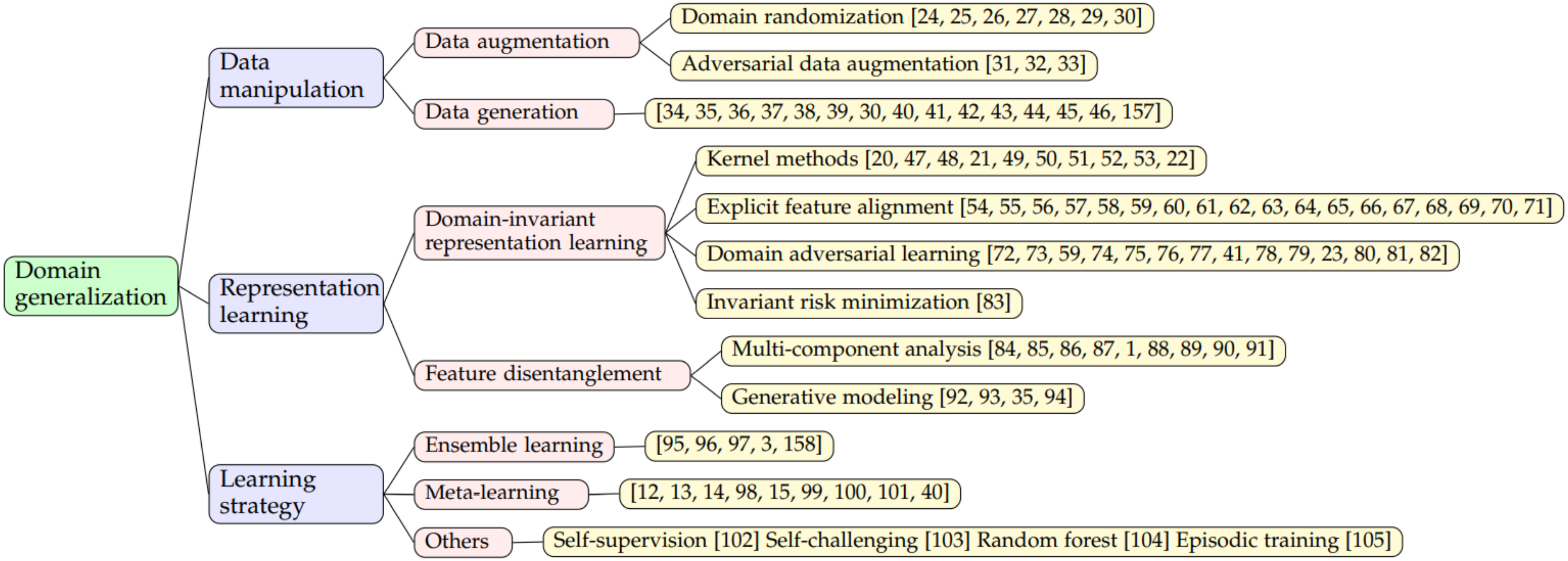


Figure from Wang et al 2021.

Taxonomy of OOD generalization methods



Domain generalization

Data manipulation

Data augmentation

Domain randomization [24, 25, 26, 27, 28, 29, 30]

Adversarial data augmentation [31, 32, 33]

Data generation

[34, 35, 36, 37, 38, 39, 30, 40, 41, 42, 43, 44, 45, 46, 157]

Representation learning

Domain-invariant representation learning

Kernel methods [20, 47, 48, 21, 49, 50, 51, 52, 53, 22]

Explicit feature alignment [54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71]

Domain adversarial learning [72, 73, 59, 74, 75, 76, 77, 41, 78, 79, 23, 80, 81, 82]

Invariant risk minimization [83]

Feature disentanglement

Multi-component analysis [84, 85, 86, 87, 1, 88, 89, 90, 91]

Generative modeling [92, 93, 35, 94]

Learning strategy

Ensemble learning

[95, 96, 97, 3, 158]

Meta-learning

[12, 13, 14, 98, 15, 99, 100, 101, 40]

Others

Self-supervision [102] Self-challenging [103] Random forest [104] Episodic training [105]

Figure from Wang et al 2021.

Quick overview of different approaches

Data manipulation/augmentation: add/modify the data you have to simulate new domains. (e.g. change texture, add simple objects, change viewpoint, rotate, …) Can be combined with GAN-like techniques to create new / potentially hard examples.

Domain-invariant representation learning: learn shared representations across domains. (Ensure the same classifier works on top of this representation, possibly with other constraints (Invariant Risk Minimization, Arjovsky et al 2019); in some versions would like to ensure features are “aligned” in some way.)

Quick overview of different approaches

Feature disentanglement: learn features with “unchanging“ distribution and varying features (across domains). Only use unchanging features.

Ensemble learning: learn multiple “different” predictors for different domains and combine them in some fashion (e.g. weight sharing, averaging, majority vote) to make more robust predictors.

Meta-learning: learn a model which is easy to “finetune” for different domains (e.g. by one or a small number of gradient steps).

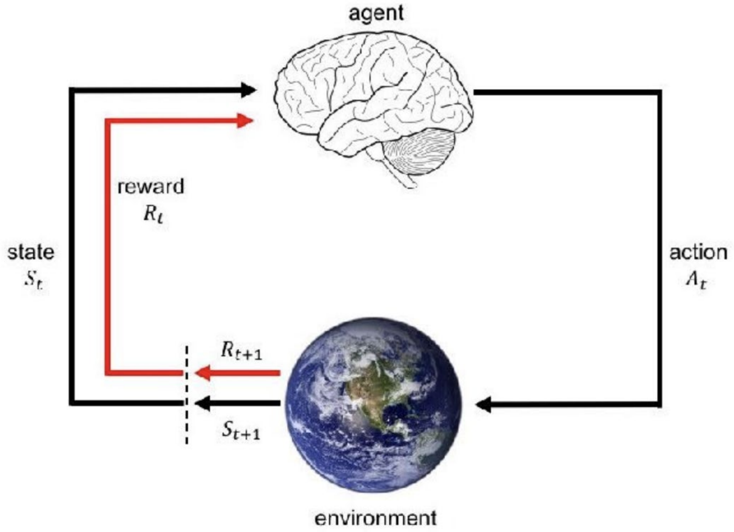
Reinforcement learning

A (very) brief introduction to RL

In most of the machine learning we’ve done in this class, we’ve begun by assuming the data is drawn i.i.d. from some unknown distribution.

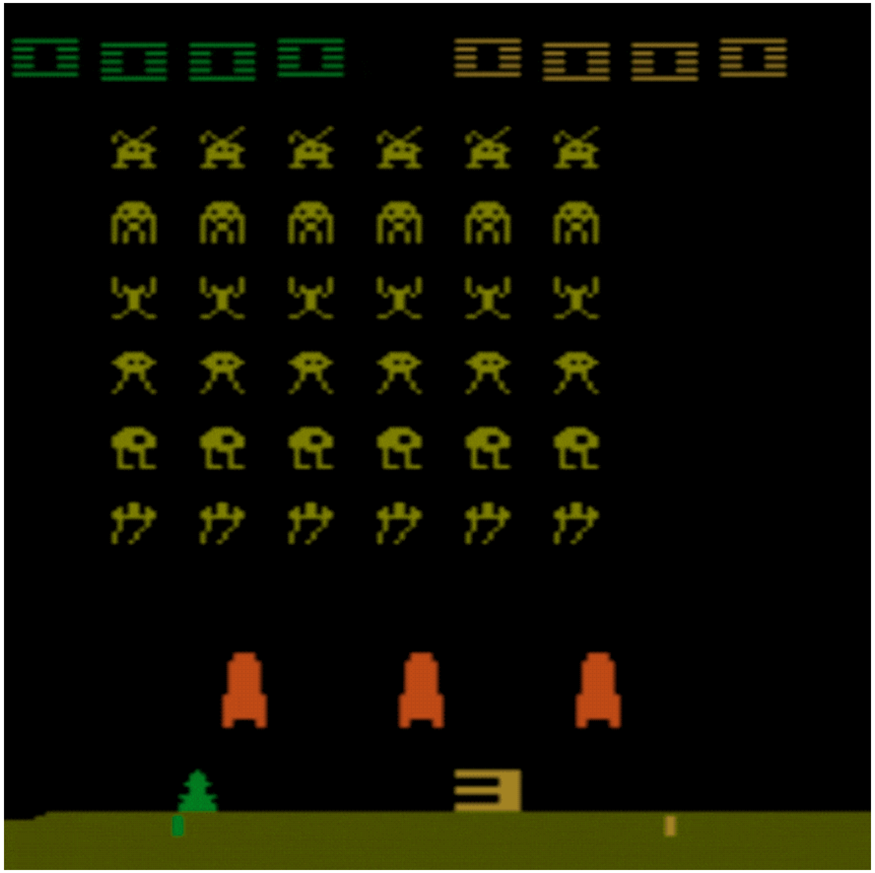
This of course is not how the world (nor our intelligence) works. In reality, we take in data in a continuous stream, and act based on it.

The analog in machine learning is ***Reinforcement Learning***, in which we train an algorithm t experience.



A (very) brief introduction to RL

This paradigm has seen a lot of advances in recent years across many domains:



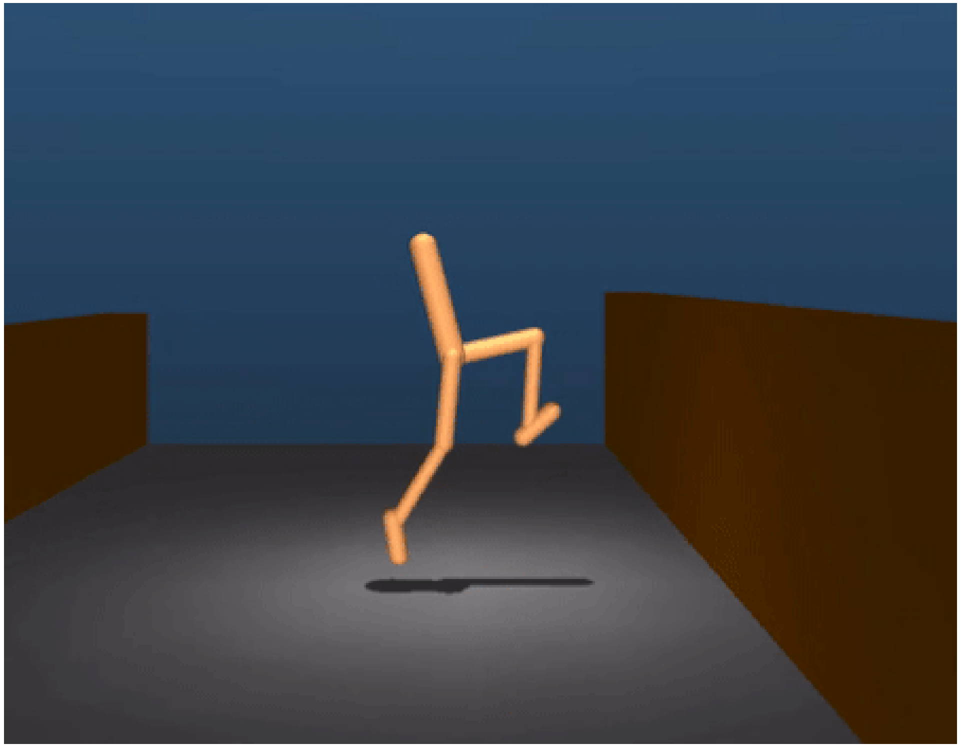
A (very) brief introduction to RL

This paradigm has seen a lot of advances in recent years across many domains:

https://deepmind.com/blog/article/alphago-zero-starting-scratch

A (very) brief introduction to RL

This paradigm has seen a lot of advances in recent years across many domains:

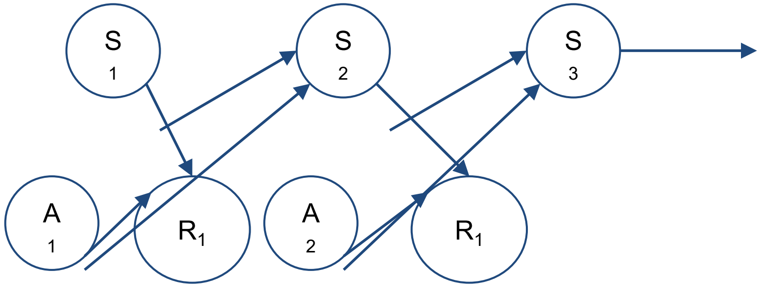


https://deepmind.com/blog/article/producing-flexible-behaviours-simulated-environments

A (very) brief introduction to RL

The formal model of this interaction process is called a ***Markov Decision Process* (MDP)**. This consists of:
A set of (potentially uncountably infinite) possible states $\boldsymbol{S = \{s_1, s_2, \ldots\}}$
A set of (potentially uncountably infinite) possible actions $\boldsymbol{A = \{a_1, a_2, \ldots\}}$
A matrix of transition probabilities $\boldsymbol{T : S \times A \rightarrow P(S)}$

Agent interacts with its environment (modeled as this system) as follows:



Note: We're assuming here that the state is *fully observed*. RL frequently deals with only *partially observed* states. This is called a POMDP. We won't cover POMDPs in this lecture.

...and so on. We also model an agent's **reward**. This is a measure of how successfully the agent is achieving its goal. The reward is a function **r** : $\boldsymbol{S \times A} \rightarrow \mathbb{R}$.

A (very) brief introduction to RL

Markov property: **The only thing that affects the next state is the current state and chosen action.**

Given this system, our goal is to take the actions which maximize our total reward.

The function we use to decide which action to take is the **policy** π.

The policy is a (possibly stochastic) function from state to action.

Given a policy π_θ parameterized by θ, we write $\pi_\theta(a|s)$ to denote the probability of taking action a from state s.

A (very) brief introduction to RL

The policy, combined with the MDP, induces a probability distribution on **trajectories** τ, sequences of state/action pairs.

$$\underbrace{p_\theta(\mathbf{s}_1,\mathbf{a}_1,\ldots,\mathbf{s}_T,\mathbf{a}_T)}_{p_\theta(\tau)} = p(\mathbf{s}_1)\prod_{t=1}^{T}\underbrace{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}_{\text{Markov chain on } (\mathbf{s},\mathbf{a})}$$

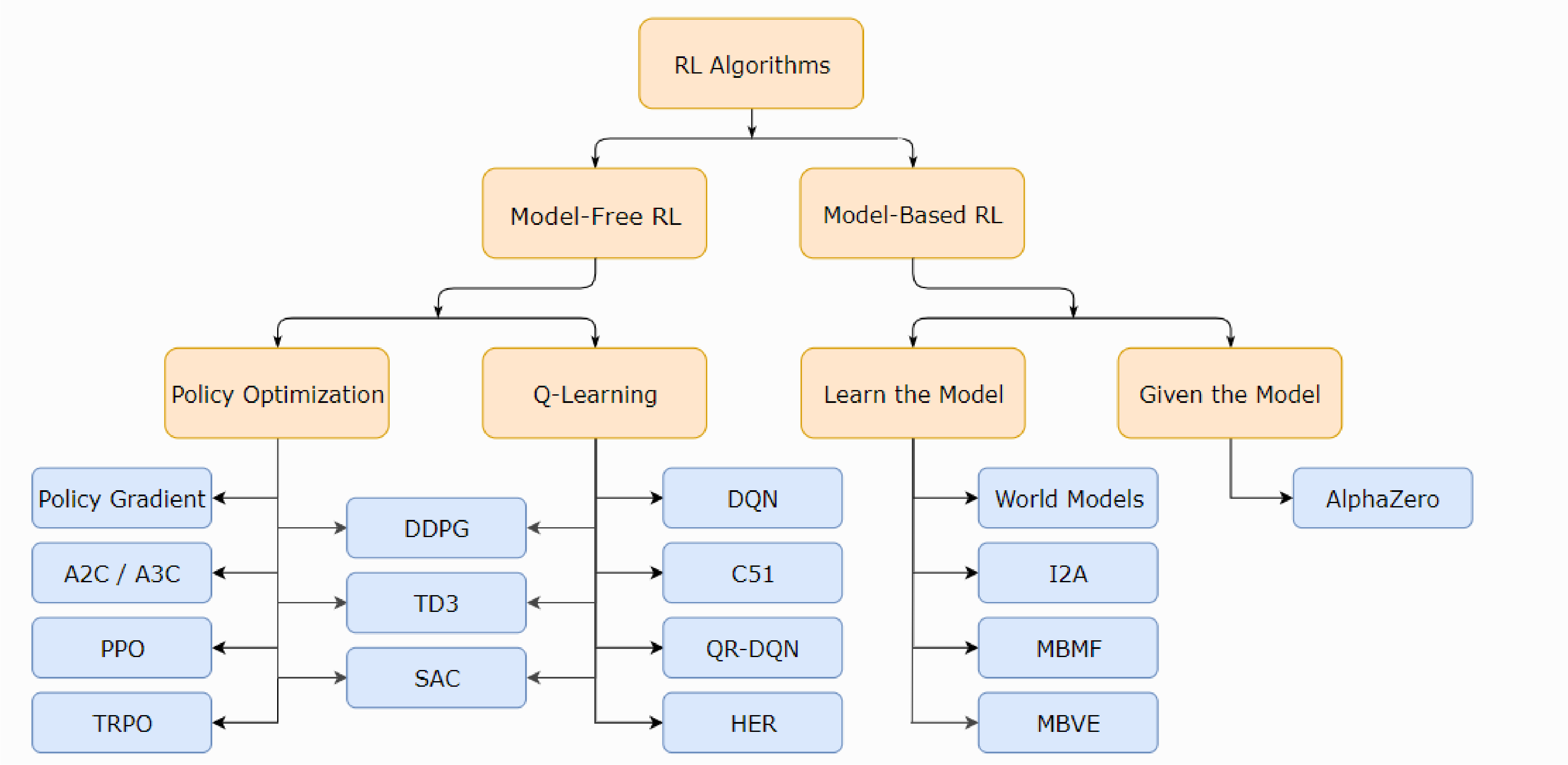
We will sometimes also write π_θ(τ), these mean the same thing.

Our objective is thus to find the parameters θ which will maximize our expected reward, where the expectation is with respect to the distribution over trajectories.

$$\theta^* = \mathrm{argmax}_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)} \sum_{t=1}^{T} r(s_t, a_t)$$

T is called the horizon. One can also handle an infinite horizon by adding a **decay parameter** $\boldsymbol{\gamma}$ and discounting rewards at time t by γ^t

Taxonomy of RL algorithms

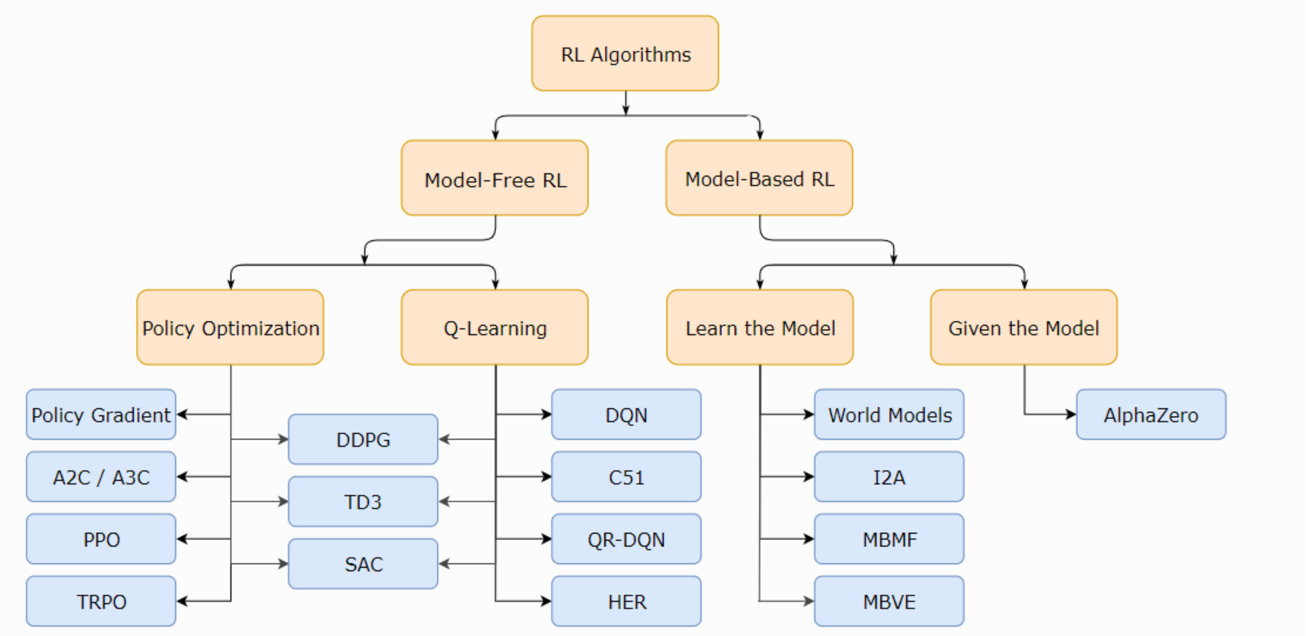


spinningup.openai.com

Model based vs. model free

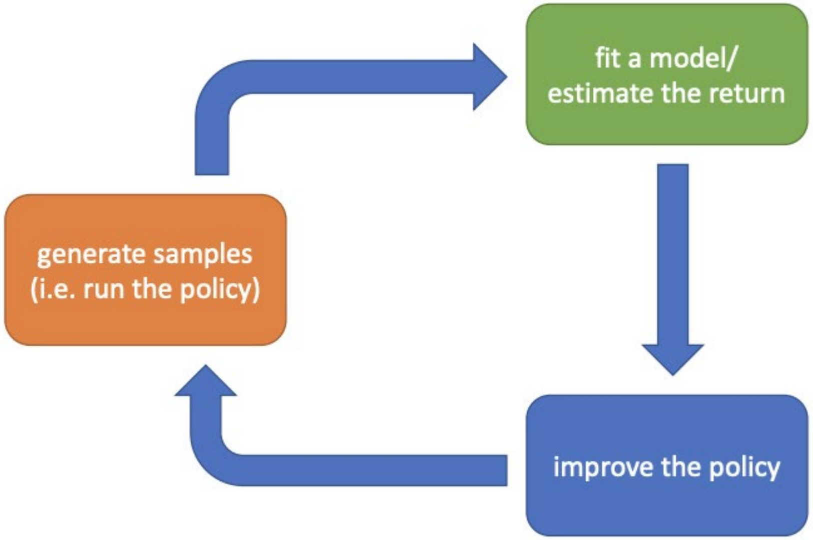
A *model-based* algorithm fits a model to the environment (i.e. learn system dynamics). Allows for efficient planning.

A *model-free* algorithm makes no attempt to learn the dynamics. Instead, learns a set of rules for maximizing reward. Doesn't fail as much under model misspecification.



The anatomy of a reinforcement learning algorithm

Almost every RL algorithm follows the same process:



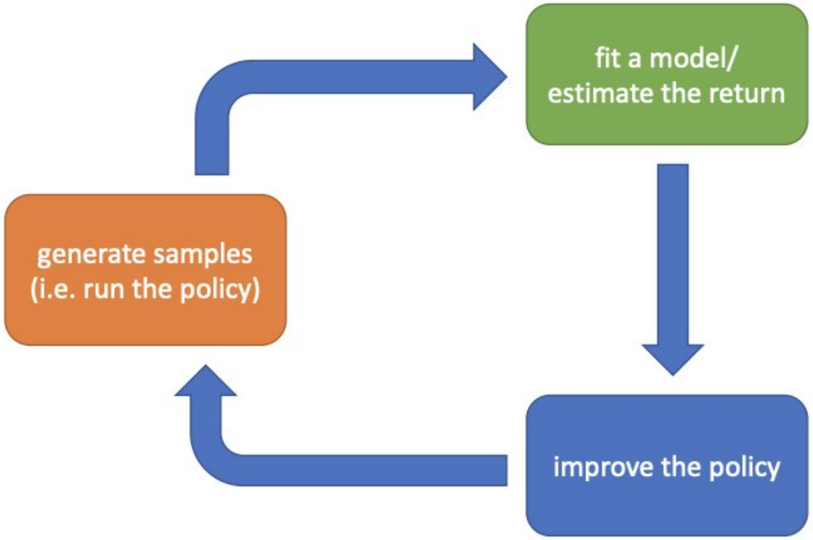
They differ in how they approach each step.

This lecture will *very briefly* touch upon some common approaches.

If you want to learn more, check out 10-703!

Model-free example: Policy gradients

A straightforward example:



$$J(\theta) = E_\pi\left[\sum_t r_t\right] \approx \frac{1}{N}\sum_{i=1}^{N}\sum_t r_t^i$$

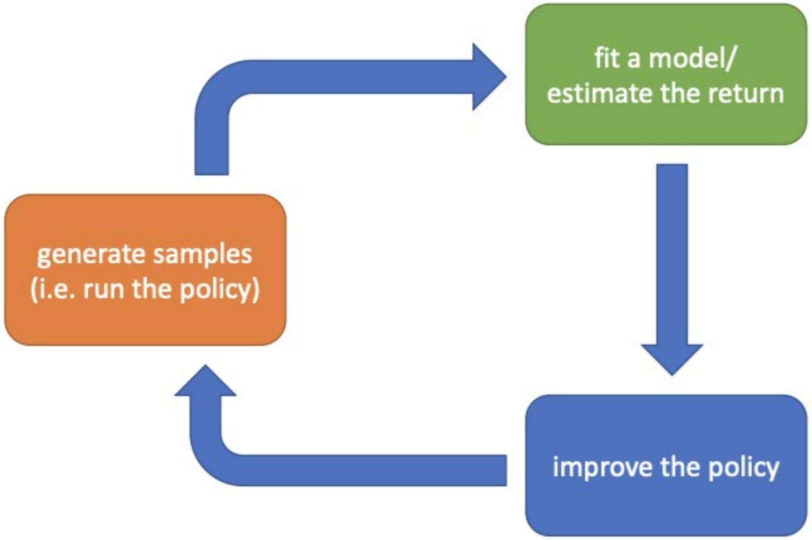
$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Seems straightforward: follow the gradient of the policy parameters to increase the expected reward.

There's a catch!

Model-free example: Policy gradients

Remember that original example we saw for learning a policy?


fit a model/ estimate the return
improve the policy
generate samples (i.e. run the policy)

$$J(\theta) = E_{\pi}\left[\sum_t r_t\right] \approx \frac{1}{N}\sum_{i=1}^{N}\sum_t r_t^i$$

Remember that π here is parameterized by θ!

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

This means we need to take the gradient of the parameters over an expectation *which depends on the parameters*.
Recall this same problem occurs when optimizing a VAE. But here, there's no simple method for reparameterization.

The REINFORCE algorithm

[Williams, 1992]
Recall:

$$\nabla_\theta \log p_\theta(\tau) = \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)}$$

Rearranging,

$$p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) = \nabla_\theta p_\theta(\tau)$$

Our objective:

$$J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\underline{\sum_t r(s_t, a_t)}\right]$$

Pushing through the gradient,

$$\begin{aligned}\nabla_\theta J(\theta) &= \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau \\ &= \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) r(\tau) d\tau \\ &= \mathbb{E}_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)]\end{aligned}$$

The REINFORCE algorithm

As usual, working with log-likelihood makes everything simpler.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\underline{\nabla_\theta \log \pi_\theta(\tau)}r(\tau)]$$

$$\overbrace{\nabla_\theta \left[\log \cancel{p}(\mathbf{s}_1) + \sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log \cancel{p(\mathbf{s}_{t+1}}|\mathbf{s}_t, \mathbf{a}_t)\right]}$$

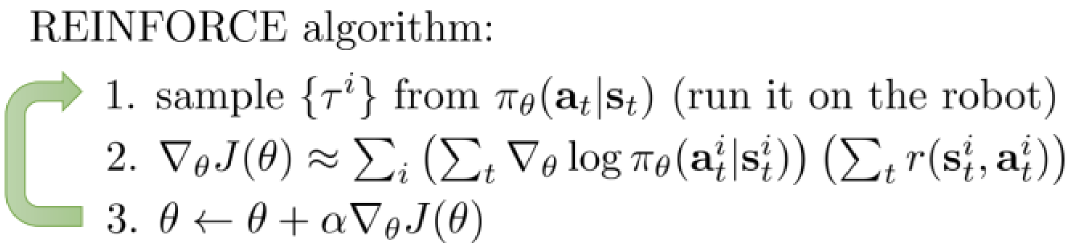
$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)\right)\right]$$

In practice, this is approximated with the empirical estimate.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right)$$

The REINFORCE algorithm

Putting it all together:



REINFORCE algorithm:

```
1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run it on the robot)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right) \left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
```

So what does this accomplish?

$$\text{policy gradient:} \quad \nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

$$\text{maximum likelihood:} \quad \nabla_\theta J_{\text{ML}}(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \right)$$

Looks like MLE, but weighted by the reward of the trajectory!
Makes good trajectories more likely, and bad trajectories less likely.
We’ve effectively formalized “trial and error”.

Variance reduction for PG methods

Unfortunately, this estimator has very high variance.
This is *not* the same as gradient descent on a standard supervised learning problem.
How can we address this?

One common approach is to use a *control variate*.
This is a “baseline” which we subtract from the reward at each step.
The most basic choice is just the average reward.

$$b = \frac{1}{N} \sum_{i=1}^{N} r(\tau_i)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i) [r(\tau_i) - b]$$

Variance reduction for PG methods

Can we just subtract the baseline like that?

$$\begin{aligned}
\mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) b] &= \int \pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau) b d\tau \\
&= b \int \nabla_\theta \pi_\theta(\tau) d\tau \\
&= b\nabla_\theta \int \pi_\theta(\tau) d\tau \\
&= b\nabla_\theta 1 = 0
\end{aligned}$$

This term has expectation 0, so subtracting it is ok!
Average reward is not the best control variate. We can *learn* a better one from data.

Model-free example: Q-learning

[Watkins, 1989]
For an infinite horizon MDP, the *Q* function *Q(s, a)* approximates the expected cumulative reward for taking action *a* in state *s* and then continuing to choose actions based on the Q function

How do you estimate *Q?*

Main idea: *Q* satisfies a sort of "consistency" recurrence equation, called the Bellman equation:

$$Q(s_t, a_t) = r(s_t, a_t) + \max_{a\prime} Q(s_{t+1}, a')$$

Model-free example: Q-learning

We can treat Bellman equation as a regression problem!

Learn a Q in some function family to satisfy the Bellman recurrence:

$$\min_{\phi} \sum_{t} \left(Q_{\phi}(s_t, a_t) - \left(r(s_t, a_t) + \max_{a'} Q_{\phi}(s_{t+1}, a') \right) \right)^2$$

Deep Q-learning: use a deep network to approximate the Q function.

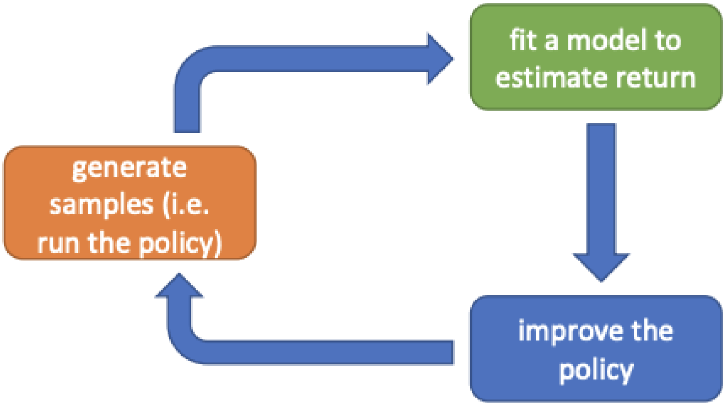
Using the Q function

Given a Q function, policy is obvious:
Take whatever action maximizes expected future reward.
In other words, $\mathbf{a} = \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})$

Note that this is a *deterministic* policy!
Typically this will be implemented in an ϵ-greedy fashion.

Our algorithm looks like this:

$$Q_\phi(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}', \mathbf{a}')$$



fit a model to estimate return

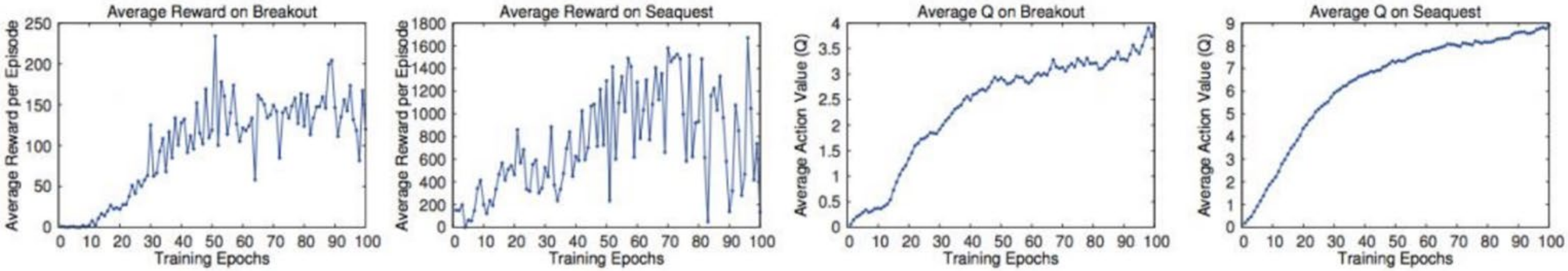
improve the policy

generate samples (i.e. run the policy)

$$\mathbf{a} = \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})$$

Double Q-learning

How accurate are Q-values?



They’re correlated with observed reward. But they overestimate it!

Why? We use the *same Q function* to **choose** the action to take, and to **estimate** the value of said action.
Recall: $\mathbf{a} = \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})$

This means we are biased upwards. We will necessarily have an optimistic estimate of the expected reward.

Double Q-learning

[Hasselt, 2010]: To address this, introduce a *second* Q function. Call the parameters ϕ_A and ϕ_B, so the functions are $Q_{\phi A}$ and $Q_{\phi B}$.

When we update the functions, each estimates the value of the other's maximizing action.

$$Q_{\phi_A}(\mathbf{s}, \mathbf{a}) \leftarrow r + \gamma Q_{\phi_B}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi_A}(\mathbf{s}', \mathbf{a}'))$$

$$Q_{\phi_B}(\mathbf{s}, \mathbf{a}) \leftarrow r + \gamma Q_{\phi_A}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi_B}(\mathbf{s}', \mathbf{a}'))$$

The noise in the two Q functions' approximations are uncorrelated, so they don't work together to bias the estimate.

Actor-Critic Methods

Actor-Critic Methods combine policy-based methods (e.g. policy gradient) and value-based methods (e.g. Q-learning).

Main idea: learn an **actor** who learns a policy and a **critic** who learns how good an action is.

Decoupling policy and value learners decreases bias. Most straightforward incarnation: Modify policy gradient update to

$$\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i \mid s_i) \hat{Q}(s_i, a_i)$$

Check out 10-703

This is a very broad field with tons of interesting stuff going on!