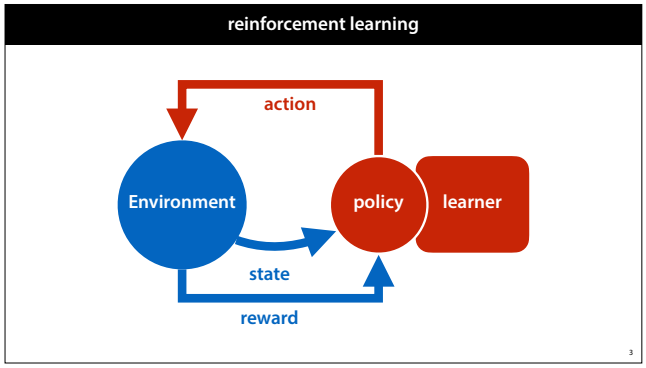
In this lecture we'll look at reinforcement learning. Reinforcement learning is first and foremost an abstract task: like regression, classification or recommendation.

The first thing we did, in the first lecture, when we first discussed the idea of machine learning, was **to take it offline**. We simplified the problem of learning by assuming that we have a training set from which we learn a model once. We reduced the problem of adaptive intelligence to a function from a dataset to a model by removing the idea of interacting with an outside world, and by removing the idea of continually learning and acting at the same time.

Sometimes those aspects can not be reduced away. In such cases we can use the framework of r**einforcement Learning**. Reinforcement learning is the practice of training agents (e.g. robots) that interact with a dynamic world, and to train them to learn *while* they're interacting.

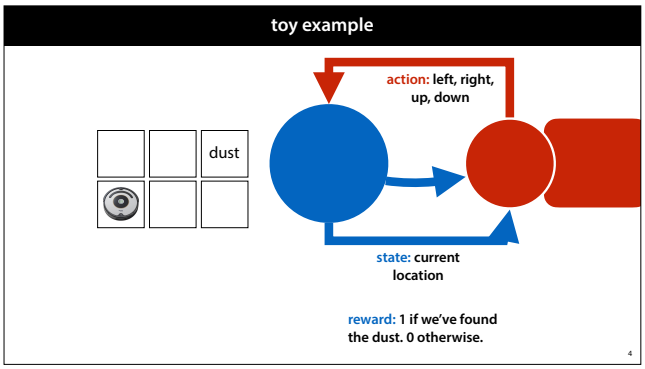


Reinforcement learning (RL) is an **abstract task**, and it is one of the most generic abstract tasks available.. Almost any learning problem you encounter can be modelled as a reinforcement learning problem (although better solutions will often exist).

The source of examples to learn from in RL is the environment. The **agent** finds itself in a **state**, and takes an **action**. In return, the environment tells the agent its new state, and provides a **reward** (a number, the higher the better). The agent chooses its action by a **policy**: a function from states to actions. The policy is essentially the model that we learn. As the agent interacts with the world the **learner** adapts the policy in order to maximise the expectation of future rewards.

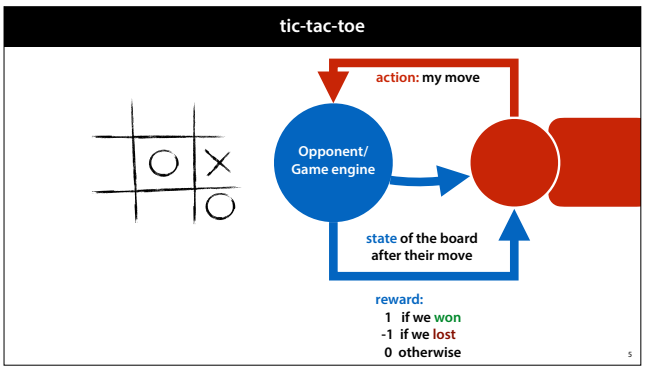
In order to translate your problem to the RL abstract task you must decide what your states and actions are, and how to learn the policy.

The only true constraint that RL places on your problem is that for a given state, the optimal policy may not depend on the states that came before. Only the information in the current state counts. This is known as a **Markov decision process**.

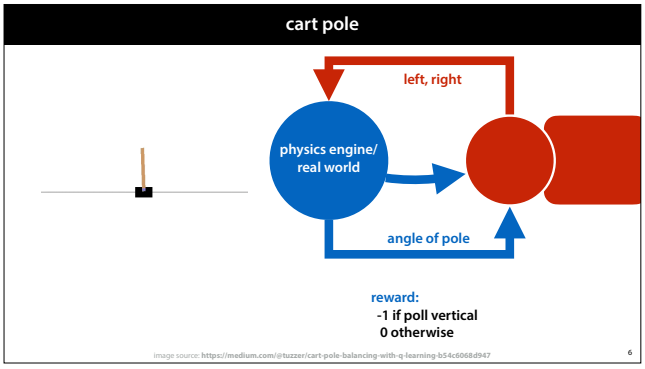


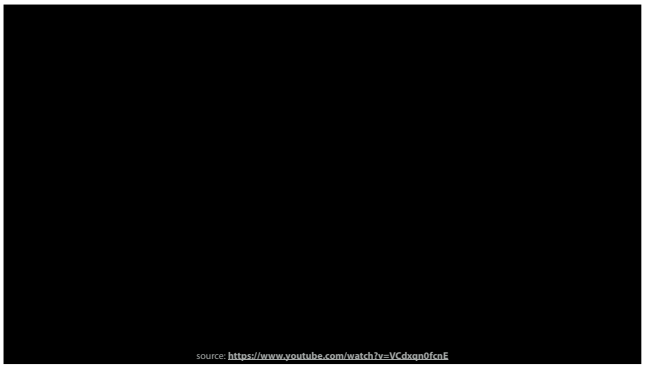
Here is a very simple example for the floor cleaning robot. The room has six positions for the robot to be in, and in one of these, there is a pile of dust. For now we, assume that the position of the dust is fixed, and the only job of the robot is to get to the dust as quickly as possible. Once the robot find the dust, the world is reset, and the robot is placed in a random position. This is not a very realistic example, but it pays to start very simple. (If you really wanted to solve this specific problem you would simply use A*)

The environment has six states (the six squares). The actions are: up, down, left and right. Moving to any state yields a reward of zero, except for the G state, in which case it gets a reward of 1.



Games can also be learned through RL. In the case of a perfect information, turn-based two player game like tic-tac-toe (or chess or Go). The states are simple board positions. The **available actions** are the moves the player is allowed to make. After an action is chosen, **the environment chooses the opponent's move**, and returns the resulting state to the agent. All states come with reward 0, except the states where the game is won by the agent (reward = 1) or the game is lost (reward -1). A draw also yields reward 0.





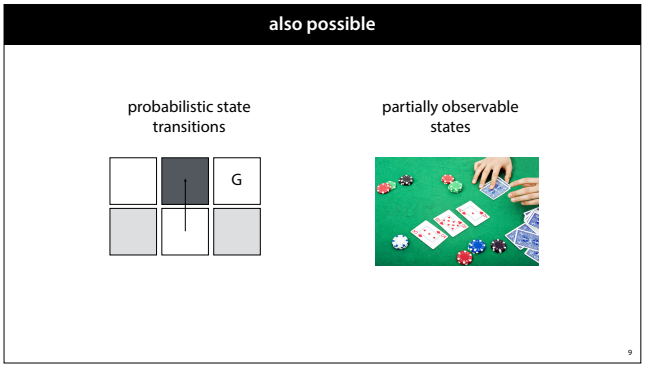
Here is an example with a slightly faster rate of actions chosen: controlling a helicopter. The helicopter is fitted with a variety of sensors, telling it which way up it is, how high it is, it's speed and so on. The combined values for all these sensors at a given moment form the state. The actions following this state are the possible speeds of the main and tail rotor. The rewards, again, are zero unless the helicopter crashes, in which case it gets a negative reward. To train the helicopter to do specific tricks (like flying upside down), we can give certain states a positive reward depending on the trick

source: **https://www.youtube.com/watch?v=VCdxqn0fcnE**

One benefit of RL is that a single system can be developed for many different tasks, so long as the interface between the world and the learner stays the same. Here is a famous experiment by DeepMind, the company behind AlphaGo. The environment is an Atari simulator. The state is a single image, containing everything that can be seen on the screen. The actions are the four possible movements of the joystick and the pressing of the fire button. The reward is determined by the score shown on the screen.

The amazing thing here is that the system was not pre-programmed with any knowledge of any of the games. For several of the games the system learned play the game better than the top human performance.

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

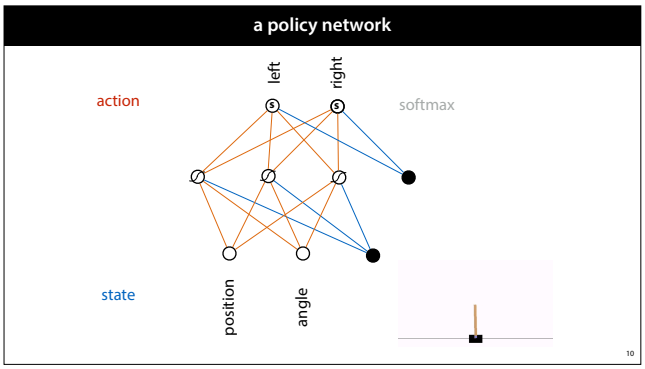


Here are some extensions to RL that we won't go into (too much) today.

Sometimes the state transitions are probabilistic. Consider the example of controlling a robot: the agent might tell its left wheel to spin 5 mm, but on a slippery floor the resulting movement may be anything from 0 to 5 mm.

Another thing you may want to model is partially observable states. For example, in a poker game, there may be five cards on the table, but three of them might be face down.

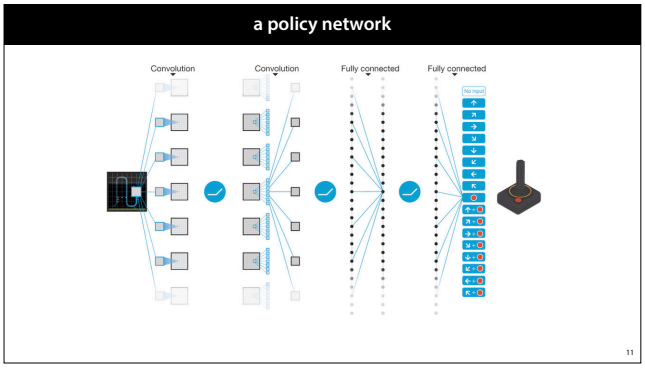
image source: ***http://www.dwaynebaraka.com/blog/2013/10/03/why-most-csr-budgets-are-wasted/***



Before we decide how to train our model, let's decide what it **is**, first. There are many ways to represent RL models, but most of the recent breakthroughs have come from using neural networks. Our job is to map states to actions, to states to a distribution over actions. We represent the state by two numbers (the position of the cart and the angle of the pole) and we use a softmax output layer to produce a probability distribution over the two possible actions.

If we somehow figure out the right weights, this is all we need to solve the problem: for every state, we simply feed it through the network and either choose the action with the highest probability, or sample from the outputs.

So now all we need is a way to figure out the weights.



episodic learning

Play short stretches of learning activities.

One episode:

- A game of tic-tac-toe, chess, go
- 3 minutes of similated helicopter flight
- 1 game of breakout until death

Train for one episode, observe reward, learn, repeat

Result is a fixed, non-updating policy for production

12

Even though reinforcement learning agents can theoretically learn in an online mode, where they continuously update their model while they explore the world, this can be a very difficult setting to control, and it may lead to very unpredictable behaviors. In practice this is rarely how agents are trained.

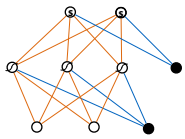
A more common setting is that of **episodic learning**. We define a particular activity that we'd like the robot to learn, and call that an episode. This could be one game of chess, one helicopter flight of a fixed length, or one Atari game for as long as the agent can manage to stay alive.

We then let the agent act one episode, based on its current policy. We observe the **total reward** at the end of the policy, use that to update the parameters of the policy (to learn) and then start another episode.

Often after training like this for a while, when we are convinced we have a good policy, we keep it fixed when we roll it out to production. That is, when you buy a robot vacuum cleaner, it may contain a policy trained by reinforcement learning, but it almost certainly won't update its weights as it's vacuuming your floors.

---

the four problems of RL



Sparse loss

Delayed reward, credit assignment

Non-differentiable loss

Exploration vs. exploitation

13

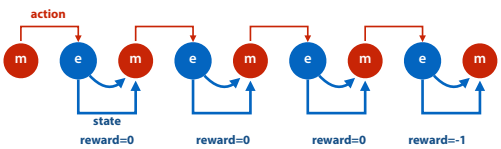
So what's the big problem? We know how to find good weights for a neural network already: we use minibatch gradient descent and use back propagation to work out the gradient.

There are three problems that make it difficult to apply gradient descent as we know it.

If your problem has any of these properties, it can pay to tackle it in a reinforcement learning setting, even if the problem doesn't look like a reinforcement learning task to begin with. For instance, an autoencoder architecture with a non-differentiable sampling step in the middle could be trained using reinforcement learning methods, even though the task doesn't really

---

sparse loss



14

The problem of **sparse loss** is the issue that often very few states have a meaningful reward.

In a chess game, there are three states with meaningful loss: lost, won or a draw. All other states, while the game is still in progress, provide no meaningful reward. We could of course estimate the value of these states to help our model learn (more about that later), but we might estimate these values wrong. If we can learn purely from the sparse reward signal (the rewards that we *know* to be correct), we can be sure that we're not inadvertently sending the model in the wrong direction.

The problem of **delayed reward** refers to the property

that a high

sparse loss

ad-hoc solutions:

Start with imitation learning:
Supervised learning, copying human action

Reward shaping:
Guessing the reward for intermediate states, or states near to good states.

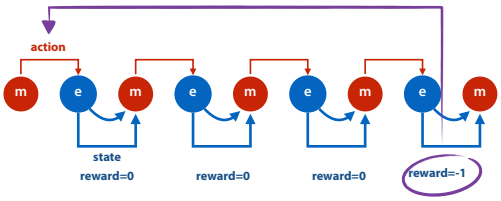
Auxiliary goals:
Curiosity, maximum distance traveled

15

Even the best reinforcement learning systems have trouble learning some tasks purely from sparse loss. Some tricks can be employed to help the model along.

Good explanation of reward shaping: **https://www.youtube.com/watch?v=xManAGjbx2k**

delayed reward (aka credit assignment)



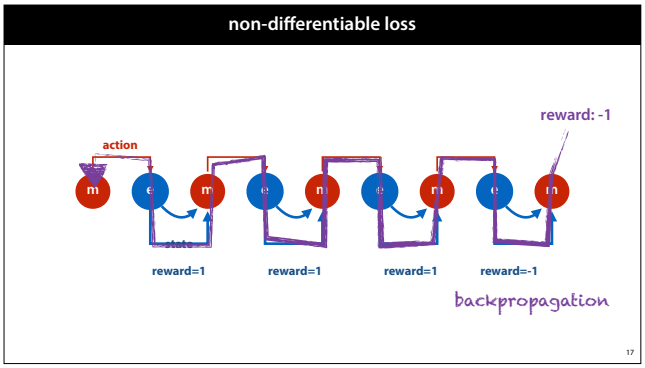
16

The problem of **delayed reward** is that we have to decide on our immediate *action*, but we don't get immediate *feedback*.

In the cart pole task, the pole falls over, it may be because we made a mistake 20 timesteps ago, but we only get the negative reward when the pole finally hits the ground. Once the pole started tipping over to the right, we may have moved right twenty times: these were good actions, that should be rewarded, they were just too late to save the situation.

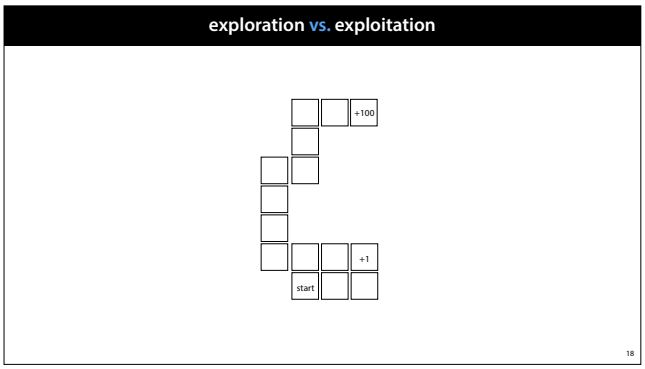
Another example is crashing a car. If we're learning to drive, this is a bad outcome that should carry a negative reward. However most people brake just before they crash. These are good actions that led to a bad outcome. We shouldn't learn not to brake before a crash, we should work backward to where we went wrong (like taking a turn at too high a speed) and apply the negative feedback to only those actions.

This is also called the **credit assignment problem**, and it's what reinforcement learning is all about.



If we draw a run of our policy like this, we are essentially unrolling the execution of the network over time, much like we did with the recurrent neural nets. If we could apply backpropagation through time, we could let the backpropagation algorithm deal with credit assignment for us. We take the reward at each point, and backpropagate it through the run to compute the gradients over the weights.

Here unfortunately, a lot of parts of the model *aren't differentiable*, chief among them **the environment**. We don't usually know exactly how the environment computes the next state: we need to *pay* in exploration to find out how the environment works.

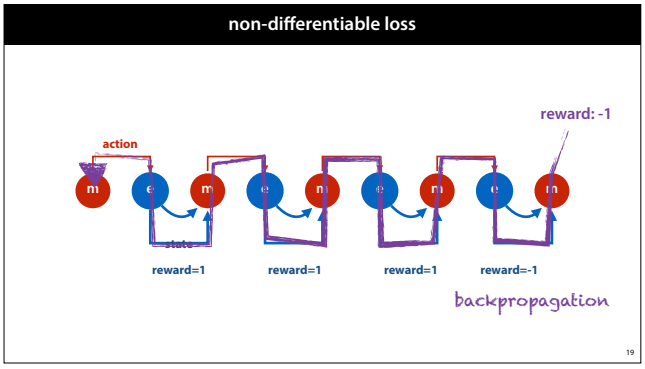


The final problem in RL is the **exploration vs. exploitation** tradeoff.

In this scenario drawn here the squares are states, and the agent moves from state to state to find a reward. Each time the agent finds a reward it is reset to the start state. This is like the floor cleaning robot example.

An agent stumbling around randomly will most likely find the reward top right first. After a few resets it will have figured out how to return to the +1 reward. If it exploits only the things it has learned so far, it will keep coming back for the +1 reward, never reaching the +100 reward at the end of the long tunnel. An agent that follows a more random policy, that sometimes moves away from known rewards will explore more and eventually find the bigger treasure. At first, however, the exploring agent does markedly worse than the exploiting agent.

There is no definite answer to how to optimise this tradeoff, although a few best practices exist.



Of course, if we could solve the problem of non-differentiable loss, we could update our weights in a much more *directed* fashion. We could follow the gradient of the reward rather than take random steps in model space. The two main algorithms to do this are policy gradients and Q-learning. We'll look at both in the next two videos.

coming up

- Random search
- Policy gradients
- (Deep) Q learning

20

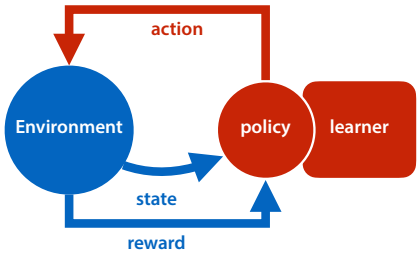
Reinforcement Learning
Part 2: Random search and policy gradients

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this video, we'll look at two simple methods for training a policy network: random search and policy gradients.

---

reinforcement learning



22

---

random search

pick a random point **m** in the model space

loop:

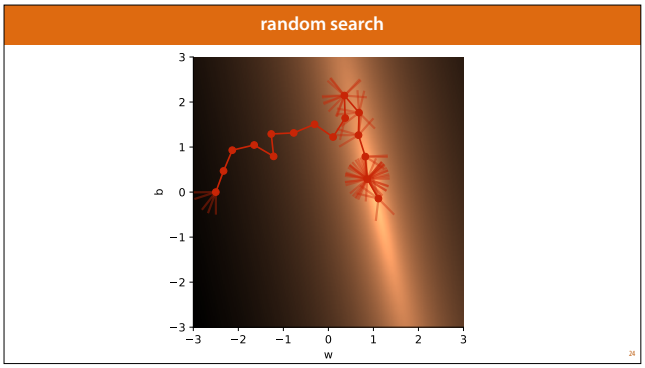
  pick a random point **m'** close to **m**

  if loss(**m**) < loss(**m'**):

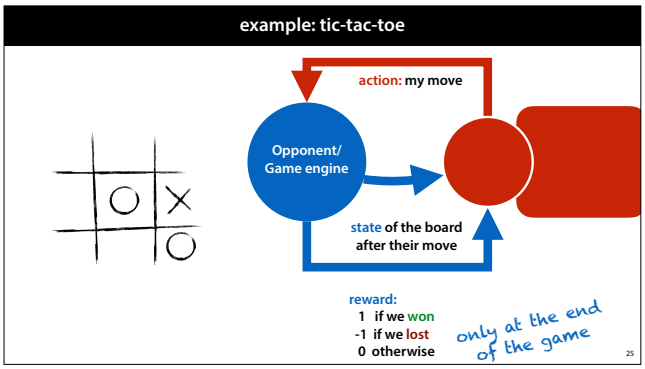
    m <- **m'**

23

Let's start with a very simple example: random search.

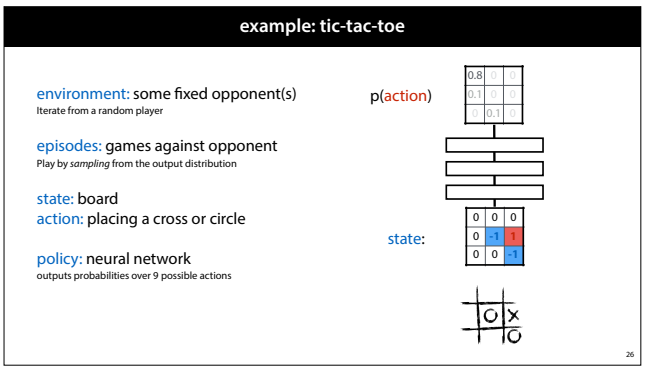


Here is random search in action. The transparent red offshoots are successors that turned out to be worse than the current point. The algorithm starts on the left, and slowly (with a bit of a detour) stumbles in the direction of the low loss region.



As an example, let's look at how we might learn to play tic-tac-toe by framing it as a reinforcement learning problem and

We'll return to this example for every reinforcement learning algorithm we introduce.



The first thing we need to decide on is what the environment will be. To keep things simple, we'll assume that we have some fixed opponent that we are going to try to beat. This could be some algorithmic player we've built based on simple rules. If we don't have any such algorithm available, we could just start with an opponent that plays random moves, train a policy network that beats that player, and iterate: we make our trained network the new opponent and train a new network to beat it.

It's often good to maintain a pool of opponents and sample one at random for each episode, so that the network doesn't overfit to beating one opponent, but maintains a good general strategy.

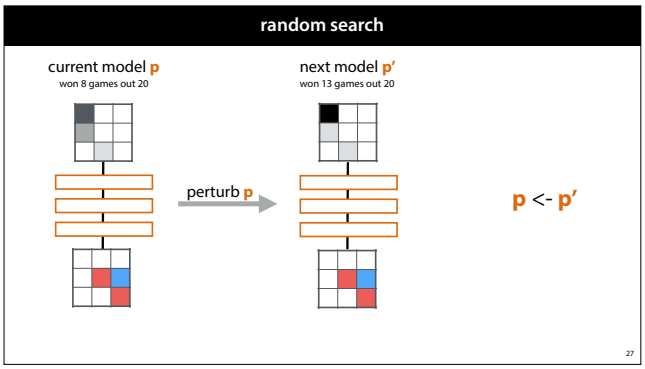
The episodes are simple games against the opponent. We play a game by taking our current policy, and sampling from the output distribution over the possible actions.

The state is simply the game board at any given moment, and an action is placing our symbol in one of the squares.

Finally, we model our policy by a neural network, a policy network. We represent the board as a 3x3 matrix, containing 0s for empty squares, -1s for squares occupied by the opponent, and 1s for squares we occupy. We can flatten this matrix into a vector and feed it to a fully connected network, or keep it a 2 tensor and feed it to a convolutional neural net. The

output layer of the network also contains 9 units that represent the squares of the board. We softmax the output layer and interpret the resulting probabilities as a probabilitstic policy: the probability on the first output node is the probability that we'll place a cross in the top left corner square.

We needn't worry about forcing the neural network not to make illegal moves, like placing a cross where there already was a symbol. If we simply ensure that an illegal move always results in instant loss of the game, then it's likely that the neural network learns not to place crosses in those squares. In short, we don't hardcode the rules, we let the network *learn* the rules.



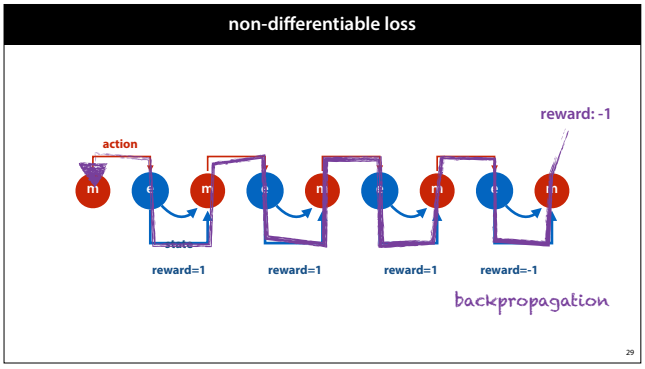
Training this model by random search is very simple. We gather up all the weights of the policy network into a vector **p** and call that our current model. We run a few episodes, that is games against the opponent, and we see how many it wins. More precisely, what it's average reward is over all the episodes. We then apply a small perturbation to **p**, like some random noise, and call the resulting policy **p'**. We check the average reward for **p'**, and if it's higher that that for **p**, we call **p'** the new current model.

If it isn't higher, we discard **p'** and keep p as the current model.

We iterate for as long as we have patience and see if the

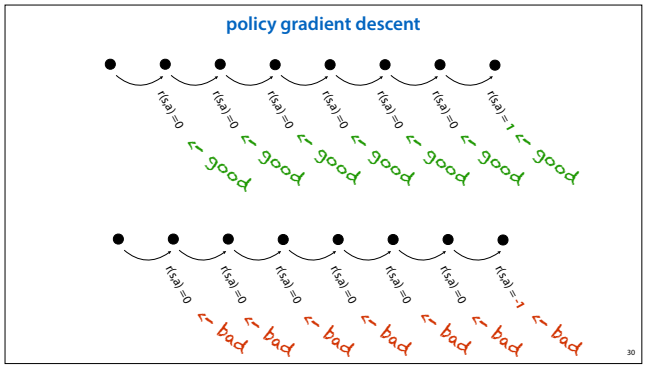
This is an exceedingly simple method, but for some games, like the Atari game Frostbite, it already works very well.

Note that the Atari challenge is "from pixels". That means we're looking at a convolutional neural net that was trained not by gradient descent but by random search.



In the last video, we noted two things:

1. We can **unroll** the computation of an episode in our learning process.
2. We cannot backpropagate through this unrolled computation graph, because parts of it are not differentiable. Specifically the sampling of actions is not differentiable, and the computation of the environment is not even accessible to us: it's essentially secret that we are meant to discover by learning.

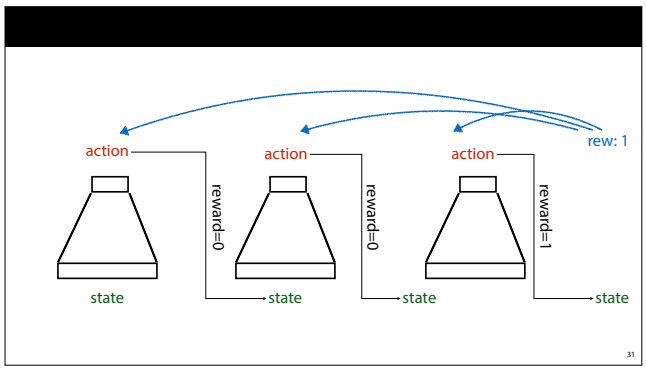


Policy gradient descent is a simple solution. We run an episode, compute the total reward at the end, and apply that as the feedback for all the steps in our policy. If we had a high reward at the end we compute the gradient for a high value and follow that, and if we had a low reward we compute the gradient for a low value and follow that.

This essentially completely ignores the problem of credit assignment. Many of the actions that led to a bad outcome may in fact have been good, and only one of the actions led to disaster. In policy gradient descent, we don't care. If the episode ended badly we punish the network blindly for all of its actions, good or bad.

The idea is that if some of the actions we punish the network for are good, then *on average* they will occur more often in episodes ending with a positive reward, and on average they will be labeled bad more often than good. We let the averaging effect over many episodes take care of the

In the case of the car crash, we should make sure the agent investigates the sequences where it doesn't brake before a crash as well (preferably in a simulated environment). Averaging over all sequences, braking before the crash results is less damage than not braking so the agent will eventually learn that braking is a good idea. Of course, we also have to make sure the reward is scaled according to the severity of the crash.



Here's what that looks like for an episode with a policy network. We compute the actions from the states, sample an action, and observe a reward and a new state. We keep going until the episode ends and then we look at the total reward.

Now, the question is how exactly do we apply the reward to each network? Once we have a loss for each instance of the network we can backpropagate based on the values form the forward pass. But is is best to just backpropapagate the reward? Should we scale it somehow? How should it ineract with the different probabilities that the network produced for the actions? If we sampled a low probability action, should we apply less of the reward. Do we backpropagate only from the node corresponding to the action we chose, or from all output nodes?

All of these approaches may or may not work. As we've seen in the past, it can help to derive an intuitive approach more formally, to help us make some of these decisions. Luckily, such a formal derivation exists from policy gradients, and it's relatively simple.

policy gradients: the math

$$\nabla \mathbb{E}_a r(a) = \nabla \sum_a p(a) r(a)$$
$$= \sum_a \nabla p(a) r(a)$$
$$= \sum_a p(a) \frac{\nabla p(a)}{p(a)} r(a) \qquad \nabla \ln(z) = \frac{1}{z} \nabla z$$
$$= \sum_a p(a) \nabla \ln p(a) r(a)$$
$$= \mathbb{E}_a r(a) \nabla \ln p(a) \approx \frac{1}{k} \sum_i r(a_i) \nabla \ln p(a_i)$$

backprop

32

We'll look at a single action a that was taken at some point during the trajectory.

We call r the final reward at the end of the episode. Our network produces probabilities from which we sample, so the final reward is a probabilistic value. Two episodes with the same initial state may lead to different total rewards due to different actions sampled. The expectation of that reward is what we're really interested in maximizing.

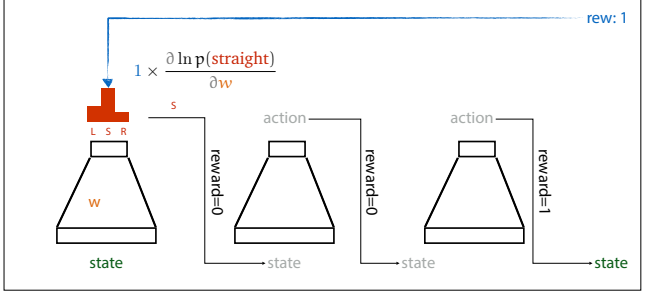
We start by writing out the reward. Here p(a) is the probability that our neural network gives to action a. The expectation is a sum over all actions, and we can freely move the gradient inside the sum.

Multiplying by p(a)/p(a) gives us a factor in the middle which we can recognize as the derivative of the natural logarithm. Filling this in, and rewriting, we see that the derivative of the expected reward after taking action a is equal to the expectation of the reward after taking a times the derivative of the log-probability of a.

And this expectation we can approximate by sampling k actions from the output distribution of our network, and averaging this quantity. The derivative of ln $p(a_i)$ is simply the derivative of the logarithm of one of the outputs of the neural net with respect to the weights. This we know how to work out by backpropagation

---

k=1



To simplify things, let's approximate the expectation with a single sample: k= 1 and look only at the gradient for the first instance of the policy network.

Imagine that our network outputs three actions: left, straight and right. And we sample the action straight. We complete the episode, and observe a total reward of 1.

If we look at the derivative for a single weight w in the neural net, we see that its gradient is one times a derivative that we can work out by backpropagation.

---

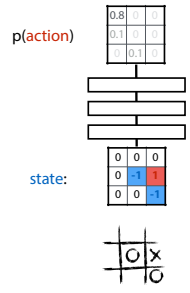
example: tic-tac-toe

environment: some fixed opponent(s)
Iterate from a random player

episodes: games against opponent
Play by *sampling* from the output distribution

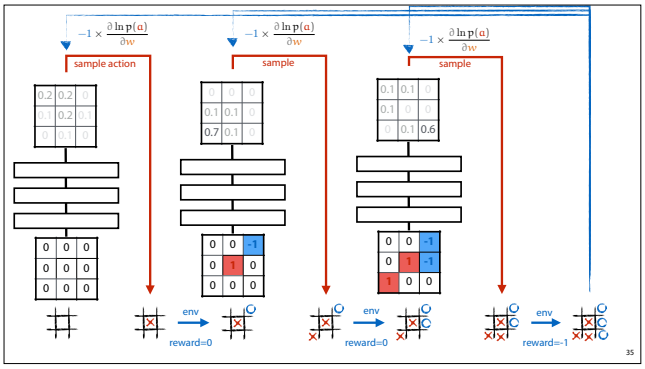
state: board
action: placing a cross or circle

policy: neural network
outputs probabilities over 9 possible actions



34

To finish up, let's see what this looks like in our our tic-tac-toe example.



policy gradients

Train an episode, save all instances of policy network.

Observe total reward.

Distribute the total reward back to all instances in the episode.

Backpropagate down the network.

36

policy gradients: final notes

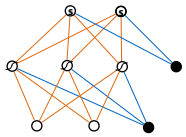
Gradient estimates are unbiased, but high variance

Many variance reduction methods exist

Control variates, actor-critic

37

the four problems of RL



- ✓ Sparse loss
- ✓ Delayed reward, credit assignment
- ✓ Non-differentiable loss
- ✗ Exploration vs. exploitation

38

So, which parts of our problem do random search and policy gradients solve? We have a solution to some extent for the sparse loss, credit assignment and non-differentiable, since the we only focus on the total loss over an episode. We may need many episodes for the effects to average out properly, but in principle, this is the start of a solution.

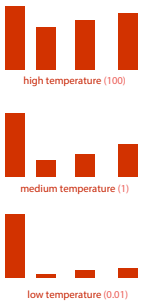
It's important to note, however, that we haven't solved the exploration vs. exploitation problem. If we always follow our currently best policy, we are still very likely to be seduced by early successes and end up just repeated a known formula for a quick and low reward, rather than finding a more complex path towards a higher reward. Put simply, we quickly get stuck in local

minima.

The optimal tradeoff between exploration and exploration is not easy to define, and in some sense, it's a subjective choice: how much immediate gain are we willing to trade off against long-term gain. Nevertheless, there are simple ways to at least give yourself control over the behavior of your learner.

Boltzmann exploration

$$\text{softmax}(x_i, t) = \frac{\exp \frac{x_i}{t}}{\sum_j \exp \frac{x_j}{t}}$$



39

One trick is Boltzmann exploration. Instead of sampling from the normal Softmax output, we introduce a new softmax with a **temperature** parameter. The temperature is divided by the elements before they are exponentiated and normalized.

---

epsilon-greedy sampling

with probability ε:

pick an action at random, uniformly

otherwise:

pick the action with the highest probability

40

The simplest algorithm to balance exploration and exploitation i

---

Reinforcement Learning
Part 3: Deep Q-Learning

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

notation

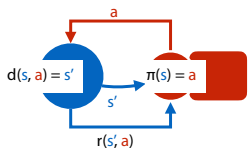
the environment

reward function:
r(s, a) = 0.01 (the higher the better)

state transitions
d(s, a) = s'

our model

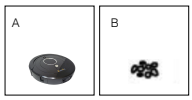
policy
π(s) = a or p(a|s) = 0.2



42

Here is some basic notation for the elements of Reinforcement learning. In most cases, the agent will not have access to the reward function, or the transition function and it will have to learn them. Sometimes the agent will learn a deterministic policy, where every state is always followed by the same action. In other cases it's better to learn a **probabilistic policy** where all actions are possible, but certain ones have a higher probability.

Q-Learning



43

While policy gradient descent is a nice trick, it doesn't really get to the heart of reinforcement learning. To understand the problem better let's look at Q-learning, which is what was used in the Atari challenge.

The example we'll use is the robotic hoover, also used in the first lecture. We will make the problem so simple that we can write out the policy explicitly: The room will have two states, the hoover can move left or right, and one of the states has dust in it. Once the hoover finds the dust, we reset. (The robot is reset to state A, and the dust is replaced, but the robot keeps its learned experience).

---

what do we want to optimize?

discounted reward:
$r(s_0, a_0) + \gamma\, r(s_1, a_1) + \gamma^2\, r(s_2, a_2) + \gamma^3\, r(s_3, a_3) + \ldots$

with $\gamma = 0.99$
or something similarly close to 1

44

If we fix our policy, then we know for a given policy what all the future states are going to be, and what rewards we are going to get. The discounted reward is the value we will try to optimise for: we want to find the policy that gives us the greatest discounted reward for all states. Note that this can be an infinite sum.

Note also that we are limiting ourselves here to deterministic policies: for a fixed policy we always do the same thing in the same state.

If our problem is finished after a certain state is reached (like a game of chess) the discounted reward has a finite number of terms. If the problem can (potentially) go on forever (like the cart pole) the sum has an infinite number of terms. In that case the discounting ensures that the sum still converges to a finite value.

definitions

policy: $\pi(s)$

value function:
$V^{\pi}(s_0) = r(s_0, a_0) + \gamma\, r(s_1, a_1) + \gamma^2\, r(s_2, a_2) + \ldots$

$V^{\pi}(s_0) = r(s_0, a_0) + \gamma\, V^{\pi}(s_1)$

optimal policy:
π^* : the π such that for all states s, $\pi^* = \underset{\pi}{\text{argmax}}\, V^{\pi}(s)$

optimal value function:
$V^*(s) = V^{\pi^*}(s)$

45

The discounted reward we get from state s for a given policy π is called $V^{\pi}(s0)$, the value function. This represents the **value** of state s: how much we like to be in state s, given that we stick to policy π.

Using the value function, we can define our optimal policy, π*. This the policy that gives us the highest value function for all states. Note that this is always possible if policy A gives us the maximal value in state s but not in state q, and policy B gives us the maximal value in state q but not in state s, we can define a new policy that follows A in state s and B in state q.

We can then define V*(s), which is just the value function for the optimal policy.

definitions

$\pi^*(s) = \underset{a}{\text{argmax}}\, [\text{"discounted reward of V*"}]$

$\pi^*(s) = \underset{a}{\text{argmax}}\, [r(s, a) + \gamma\, V^*(d(s, a))]$

$Q^*(s, a) = r(s, a) + \gamma\, V^*(d(s, a))$

$\pi^*(s) = \underset{a}{\text{argmax}}\, Q^*(s, a)$

$V^*(s) = \underset{a}{\max}\, Q^*(s, a)$

46

Using V* we can rewrite π* as a *recursive* definition. The optimal policy is the one that chooses the action which maximises the future, assuming that we follow the optimal policy. We fill in the optimal value function to get rid of the infinite sum. We've now defined the optimal policy in a way that depends on what the optimal policy is. While this doesn't allow us to compute π*, it does *define* it. If someone gives us a policy, we can recognise it by checking if this equality holds.

To make this easier, we take the part inside the argmax and call it Q(s, a). We then rewrite the definitions of the optimal policy and the optimal value function in terms of Q(s,a).

How has this helped us? Q(s,a) is a function from state-action pairs to a number expressing how good that particular pair is. If we were given Q, we could automatically compute the optimal policy, and the optimal value function. And it turns out, that in many problems it's much easier to learn the Q-function, than it is to learn the policy directly.

making the definition of Q* recursive

$Q^*(s, a) = r(s, a) + \gamma\, V^*(d(s, a))$

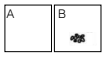
$Q^*(s, a) = r(s, a) + \gamma\, \underset{a'}{\max}\, Q^*(d(s, a), a')$

47

In order to see how the Q function can be learned we rewrite it. Earlier, we rewrote the V functions in terms of the Q function, now we plug that definition back into the Q function. We now get a recursive definition of Q.

Again, this may be a little difficult to wrap your head around. If so think of it this way: If we were given a random Q-function, how could we tell whether it was optimal? We don't know π* or V* so we can't use the original definitions. But this equality must hold true! If we loop over all possible states and actions, and plug them into this equality, we must get the same number on both sides. Let's try it for a simple example.

Is my Q-function optimal?



r(A, R) = 1, all others 0

s	a	r(s,a)	Q(s,a)
A	L	0	1
A	R	1	2
B	L	0	1
B	R	0	-1

$Q(s, a) \stackrel{?}{=} r(s, a) + \gamma \max_{a'} Q(d(s, a), a')$

48

This is the two-state hoover problem again. We have states A and B, and actions left and right. The agent only gets a reward when moving from A to B. On the bottom left we see some random policy, generated by assigning random numbers to each state action pair. Did we get lucky and stumble on the optimal policy? Try it for yourself and see. (take γ = 0.9)

---

solving recurrent equations by *iteration*

$$x = x^2 - 2$$

$$x = 0:$$
$$0 \rightarrow 0^2 - 2 = -2$$
$$x = -2:$$
$$-2 \rightarrow (-2)^2 - 2 = 2$$
$$x = 2:$$
$$2 \rightarrow 2^2 - 2 = 2$$

49

Of course, random sampling of policy functions is not an efficient search method. How do we get from a recursive definition to the value that satisfies that definition? Here is a simple example from a single number: define x as the value for which x = x^2 - 2 holds. This is analogous to the definition above: we have one x on the left and a function of x on the right.

Of course, we all learned in high school how to solve this by rewriting, but we can also solve it by *iteration*. We replace the equals sign by an arrow and write: x <- x^2 - 2. We start with some randomly chosen value of x, compute x^2 - 2, and replace x by the new value. We iterate this and we end up with a functions for which the definition holds. Try it for yourself (start with x = 0)

Note that in this example infinity also counts as a solution, so if you pick the wrong starting state you may end up with larger and larger numbers. For other functions, there may be stable states that jump back from one point to another, or even chaotic states (see **https://en.wikipedia.org/wiki/Logistic_map** for more information if you're interested, but this is not exam material).

This is known as the **iteration method** for solving recurrences (recursive definitions). And it works for function definitions too.

Q-Learning

init Q(s, a) = 0 for all s and a

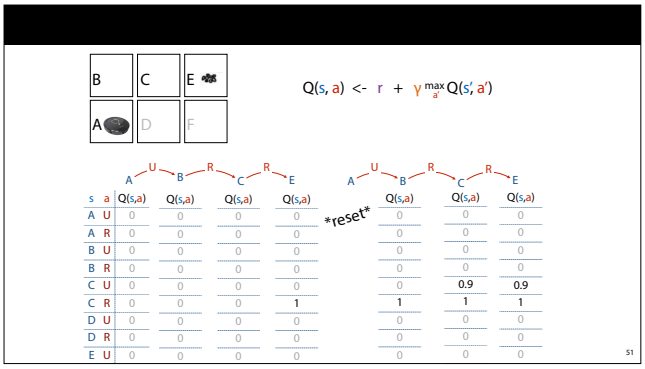
loop:
- in state s, take action a
  - arrive in s'
  - receive immediate reward r
- update Q(s, a) <- r + $\gamma \max_{a'} Q(s', a')$

50

This gives us the Q-learning algorithm shown here.

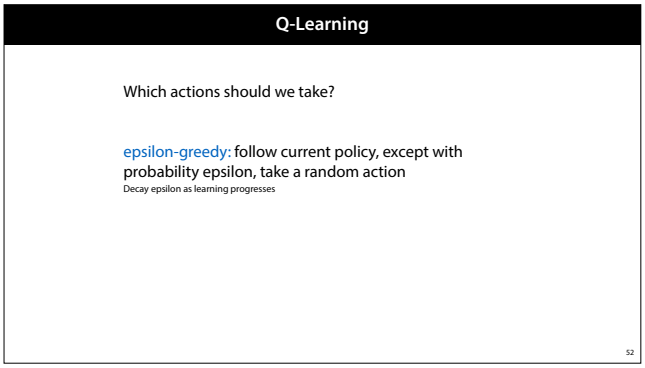
Note that the algorithm does not tell you how to choose the action. It may be tempting to use your current policy to choose the action, but that may lead you repeat early successes with out learning much about the world.

NB:While we are learning a deterministic policy here (the Q function), the function that decides which actions to take can be anything, and should contain some randomness.

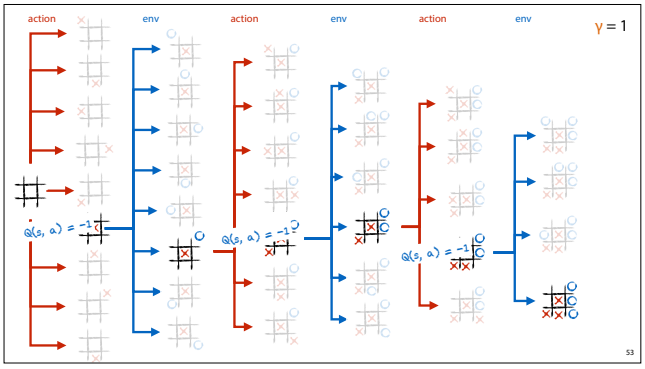


To see how Q learning operates, imagine setting a robot in the bottom-left square (A) in the figure shown and letting it explore. The robot chooses the actions up, right, right and when it reaches the goal state (E) it gets reset to the start state. It gets +1 immediate reward for entering the goal state and 0 reward for any other action.

What we see is that the Q function stays 0 for all values until the robot enters the goal state. At that point Q(C, R) west updated to value one. In the next run, Q(B, R) gets updated to 0.9. In the next run after, Q(A, U) is updated to 0.9 * 0.9. This is how Q-learning updates. In every run of the algorithm the immediate rewards from the previous runs are propagated to neighbouring states.



In contrast to policy gradients, where the standard approach is to follow your current policy, Q-learning completely separate exploration used to learn the q function and exploitation of the q function once it's been learned. Any policy that has a sufficient randomness will converge to us learning the same q function, and once we've learned it, we can use it to explore.



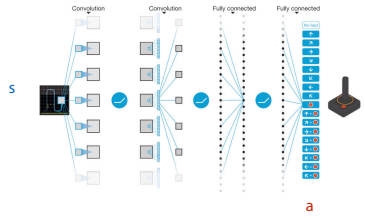
tabular Q-learning

Explicitly stores Q for *all state-action pairs*

- Only feasible for very small state spaces
- No generalization *between states*

54

deep Q-learning

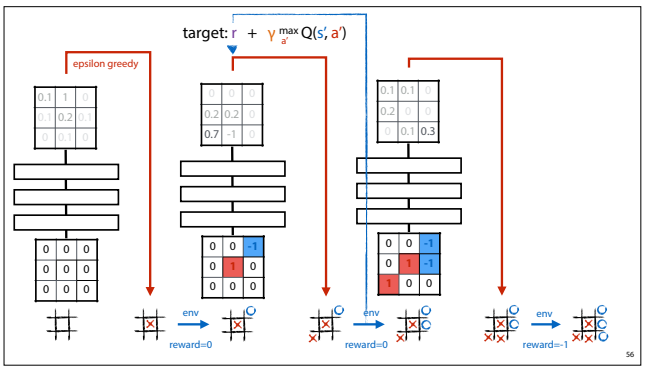


update Q(s, a) <- r + $\gamma \max_{a'} Q(s', a')$

55

The main idea behind deep q learning is that we can take the policy network, and instead of softmaxing the outputs and interpreting them as a distribution on the actions, we can use a linear activation, and interpret it as a prediction of the Q value. For a given input s, we take the output for action a to be an estimate of the value Q(a, s).

The update rule from Q learning is still the same, su



56

Here how that might look in the tic-tac-toe example.

We play a game, usually by an epsilon greedy following of our current policy. After each move we make, we observe a reward, and we compute a target value for our network. This consists of the observed reward r, and the discounted future reward in the new state s'. We add these together and this becomes a target value for the output node corresponding to a, the action we took. We compute the difference between this output and the output we observed (by some loss function) and we can backpropagate.

Note the difference with policy gradients: there we had to keep our intermediate values in memory until we'd observed the total reward for the episode. Here, we can immediately do an update after we've made our move.

more information

DLVU: **dlvu.github.io** Lectures 9 and 11

OpenAI spinning up:
https://spinningup.openai.com/en/latest/

OpenAI Gym:
https://gym.openai.com/

These slides hopefully give you an overview of the basic mechanisms of deep reinforcement learning. However, if you want to use these algorithms to do more than learn to play tic-tac-toe, You'll need to know a lot more tricks of the trade. RL is one of those methods that requires a lot of experience to make it work well.

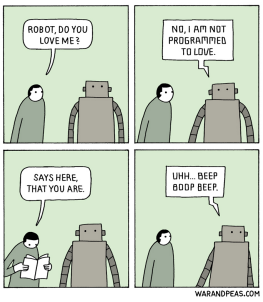
Our lectures in the master course deep learning provide more details on the extra methods you need to use in addition to the base gradient estimators like policy gradients and Q-learning.

OpenAI spinning up is a good website, that goes through all of this information step by step and tells you what you need to know to write implementations yourself.

And finally, OpenAI gym is a nice resource that saves you the considerable effort of writing an environment yourself. You can just download many different environments and focus on writing a policy networks together with a training method.

---

break



source: **https://warandpeas.com/2016/10/09/robot/**

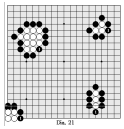
Reinforcement Learning
Part 4: AlphaGo

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In 2016 AlphaGo, a Go playing computer developed by the company DeepMind beat the world champion Lee Sedol. Many AI researchers were convinced that this AI breakthrough was at least decades away.

image source: http://gadgets.ndtv.com/science/news/lee-sedol-scores-surprise-victory-over-googles-alphago-in-game-4-813248

The game of Go



61

First, some intuition about how Go works. The rules are very simple: players (black and white) move, one after the other, placing stones on a 19 by 19 grid. The aim of the game is to have as many stones on the board, when no more stones can be placed. The only way to remove stones is to encircle your opponent. Why is Go so difficult and what has AlphaGo done to finally solve these issues?

---

claims from the media

AlphaGo is an important move towards general purpose AI.

AlphaGo thinks and learns in a human way.

AlphaGo mimics the human brain.

Go has more possible positions than there are atoms in the universe. That's why it's difficult.

What makes Go difficult is the high branching factor.

62

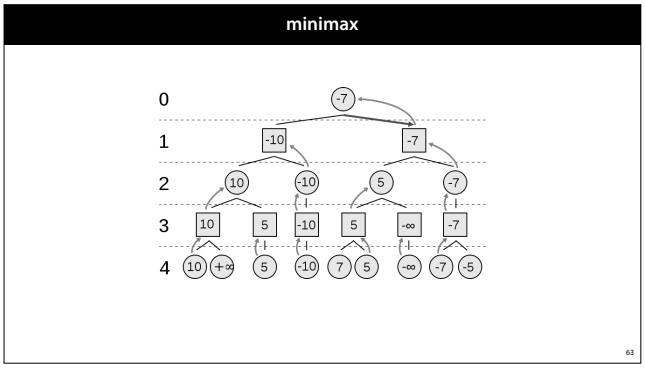
When the win against Lee Sedol was publicised, many claims were made in the media, some by DeepMind themselves. Here are some of them. All of these are dubious for various reasons

From top to bottom:

- AlphaGO is very much purpose-built for Go. It's not an architecture that can be translated 1-on-1 to any other games, and Go has some features that are exploited in a very specific way. However, DeepMind hasher projects that *are* impressive milestones toward general purpose AI. It's also true that projects like Deep Blue (the chess computer that beat Kasparov) were filled with hand-coded chess knowledge, written with the help of experts. This is not true for AlphaGo: the rules of Go were hardcoded into it, and it learned everything else by simply observing existing matches, and playing against itself
- AlphaGo learns. It's thinking is probably more human than Deep Blue's, but we don't understand human thinking well enough to make this claim.
- AlphaGo uses convolutional neural networks, which very loosely inspired by brain architecture.
- This is true, but it's also true of chess. In fact, if you had enough room to store every possible chess position in a single atom, there would be just about enough atoms in the universe for all possible Go positions. But that doesn't actually mean very much.

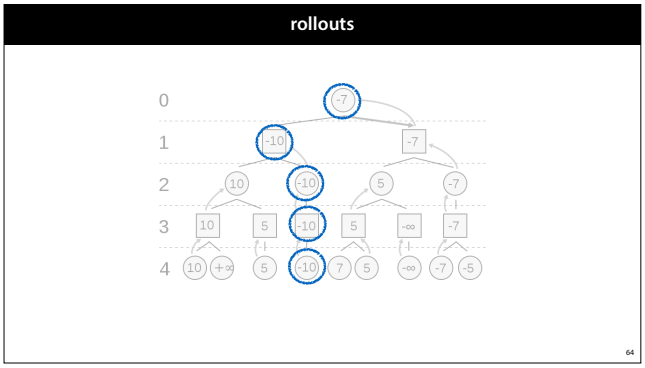
There are even more possible video files of 90 minutes in length, and yet we can easily find the ones representing good movies in that space.

- The branching factor is one of the two aspects that make go so difficult. the second is that in Go you have to wait very long to see a payoff. In chess, a basic tactic will think perhaps 5 or six moves ahead. By that time, you will have captured something and it will be clear that the tactic worked. In Go, if you're encircling a big group of stones, that can takes tens of moves, even if your opponent doesn't get in your way. That means *you need to look very deep into the game tree to see if your current action is going to have a payoff*. That's what makes Go so difficult, and it's an important factor in explaining why DeepMind solved it the way they did.

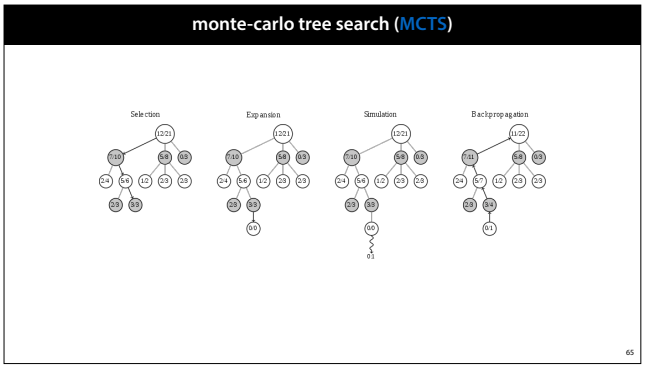


The minimax algorithm is mostly useless when it comes to Go. For each node in the tree there are up to 361 children, compared to about 30 for chess. This means almost 17 billion terminal nodes if we just search two turns deep. And as we discussed, you need to search very deep to find the nodes that show clear rewards.

image source: By Nuno Nogueira (Nmnogueira) - http://en.wikipedia.org/wiki/Image:Minimax.svg, created in Inkscape by author, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=2276653



This simple principle was an early success in playing Go: we simply choose random moves from some fast policy, and play a few full games for each immediate successor. We then average the rewards we got over these as a value for the successor states, and choose the action that lead to the highest values. The **rollout policy** should ideally give good moves high probability, but also be very fast to compute.



Monte Carlo Tree Search (MCTS) is a simple, but effective algorithm, combining rollouts with an incomplete tree search. We start the search with an unexpanded root node labeled 0/0 (this value represents the probability of winning from the given state). We then iterate the following algorithm

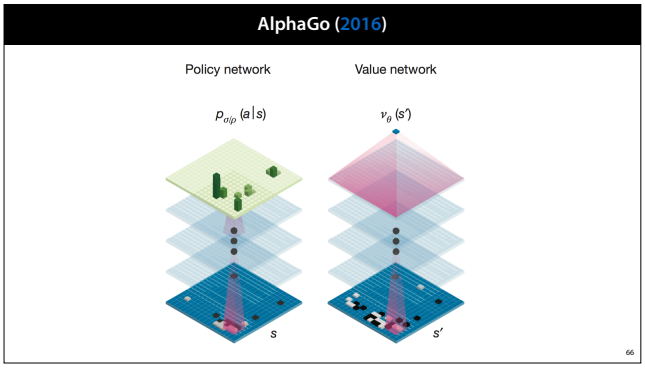
- Selection: select unexpanded node. At first, this will be the root node. But once the tree is further expanded we perform a random walk from the root down to one of the leaves.
- Expansion: Once we hit a leaf (an unexpanded node), we expand it and give its children the value 0/0.
- Simulation: From each expanded child we do a rollout.
- Back propagation (nothing to do with the backpropagation we know from NNs): If we win the rollout let v = 1 otherwise v = 0. For the new child and everyone of its parents update the value. If the old value was a/b, the new value is a+v / b+1. The value is the proportion of simulated games crossing that state that we've won.

The random walk performed in the selection phase should favour nodes with a high value, but also explore the nodes with low values. This is a classic exploration/ exploitation tradeoff. There is no single best way to this,

Incidentally, the phrase *Monte Carlo* is used for any algorithm which uses sampling as its main mechanism.

Note that if you don't roll out fully, but get a probability p of winning (from a heuristic) you can simple use v=p.

image source: By Mciura - Own work, CC BY-SA 3.0, ***https://commons.wikimedia.org/w/index.php?curid=25382061***



the functions that AlphaGo learns are convolutional networks. One type is a policy network (from states to moves) and one is a value network (from states to a numeric value).

Here are the networks it learns

SL: policy network: from a database of games (like ALVINN, watching human drivers)

RL: policy network, start with SL, but refine with reinforcement learning (policy gradient descent)

slow policy network (with softmax layer on top)

fast policy network with (with linear

activations)

V: value network learned from observing older versions of itself playing games. Once the game is finished and the outcome of the game (eg. “black wins”) is used as the label for all the states observed in the game.

The value network predicts the winner form the current board state.

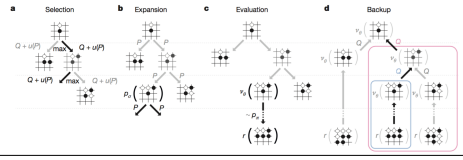
---

putting it all together

Start with imitation learning
Learn to copy human players

Train by playing against previous iterations and self
update weights by policy gradients

Boost network performance by MCTS



67

The networks are trained by reinforcement learning using policy gradient descent.

During actual play, AlphaGO uses an MCTS algorithm. The value on each node (as in basic MCTS) represents the probability that black will win from that state.

- When it comes to rollouts, use the slow policy network to do a rollout for T steps, then finish with the fast policy
- The value v of the newly opened state is the average of the value (computed with the the value network) after T steps, and the win/lose value at the end of the rollout. (image c)
- Backup as with standard MCTS: each node’s value becomes the probability of a win from that state. Precisely the value of node n becomes the sum of the values of all simulations crossing node n, divided by the total number of simulations crossing node n. A simulation is counted as a full game simulated from the root node down to a terminal (win/loss) node. (image(d))

AlphaGo Zero (2017)

Learns from scratch, no imitation learning, reward shaping etc.
Also applicable to Chess, Shogi…

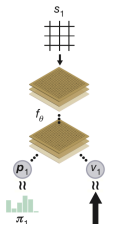
Uses three tricks to simplify/improve AlphaGo

1. Combine policy and value nets
2. View MCTS as a *policy improvement operator*
3. Add residual connections, batch normalization

68

AlphaGo Zero (2017)

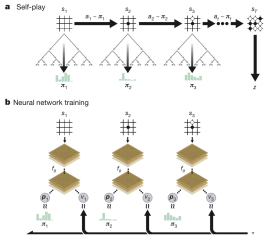
trick 1: combine policy and value nets.



69

AlphaGo Zero (2017)

trick 2: view MCTS as a *policy improvement operator*.

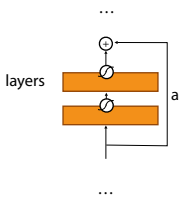


70

image source: Mastering the game of Go without human knowledge, David Silver, Julian Schrittwieser, Karen Simonyan et al.

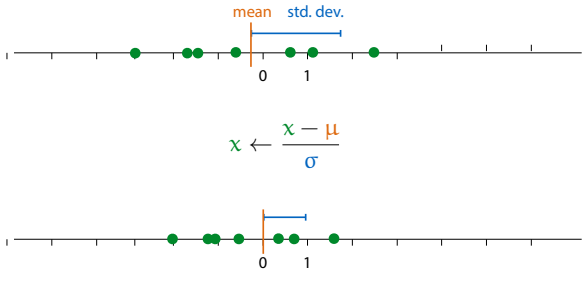
AlphaGo Zero (2017)

trick 3: residual connections and batch normalisation



71

standardisation (Methodology 2)



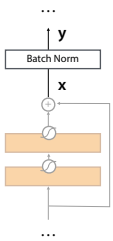
72

Another option is standardization. We rescale the data so that the mean becomes zero, and the standard deviation becomes 1. In essence, we are transforming our data so that it looks like it was sampled from a standard normal distribution (as much as we can with a one dimensional linear transformation).

We can think of the data as being generated from a standard normal distribution, followed by multiplication by sigma, and and adding mu. The result is the distribution of the data. If we then compute the mean and the standard deviation of the data, the formula in the slide is essentially inverting the transformation, recovering the “original” data as sampled from the normal distribution.We will build on

this perspective to explain whitening.

batch normalisation



$x_1, \ldots, x_m$: output of previous layer
$y_1, \ldots, y_m$: result
γ, β : learnable parameters

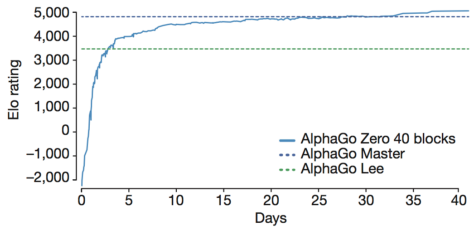
$$\mu = \frac{1}{m}\sum x_i \quad \text{mean over batch}$$

$$\sigma = \sqrt{\frac{1}{m}\sum (x_i - \mu)^2} \quad \text{std. dev. over batch}$$

$$\hat{x}_i = \frac{x_i - \mu}{\sigma + \epsilon} \quad \text{standardize}$$

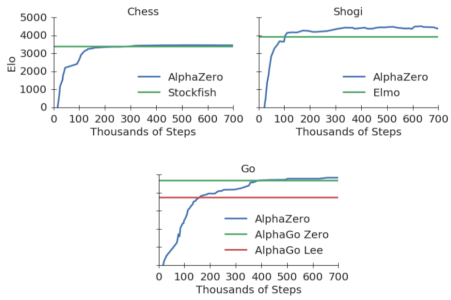
$$y^i = \gamma \hat{x}_i + \beta$$

73



74

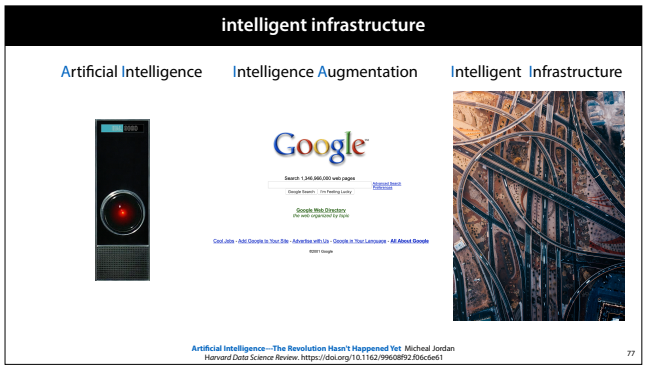
Alpha Zero (2017)



75

In the previous videos on social impact, we've looked primarily at sensitive attributes and parts of the population that are particularly at risk of careless use of machine learning.

In this video, we'll zoom out a bit, and look at the ways society *as a whole* may be at risk.



Since the 1950s, people have been talking about artificial intelligence: the idea that we may build automata that have cognitive abilities, rivalling that of humans. In fiction, examples of this are often very human in appearance: they are embodied in human bodies and they function very much the way humans do, if with a slightly metallic voice. The more more imaginative examples were AI's like 2001's HAL, which embodied a space ship rather than a humanoid body, and spoke with a natural voice. Nevertheless, it still represented a single, quite human intelligence. And the peril in that movie, still came from the intelligence behaving against the interests of the humans in a direct, adversarial way.

As we began to solve some of the problems of intelligence, the intelligent software that entered our lives did not take the form of robotic housemaids, but of simpler tool, far less intelligent tools that could augment our intelligence. A search engine is good example: it has no deep intelligence, but it helps us to use our own intelligence more effectively. This is the use of intelligent software that has rapidly increased since the 1990s.

A few years ago, machine learning researcher Michael Jordan coined the phrase **Intelligent Infrastructure** to capture the era we are entering now. An era when human-level intelligence is still some way off, but more and more components of of our national and international infrastructure are being replaced by semi-autonomous, intelligent components. These include:

- Tax services automatically generating candidates for fraud investigations.
- Hospitals automatically assigning risk profiles to patients to allocate doctor's time.
- Recommender systems highlighting relevant news stories and analysis
- Banks predicting the risk of loan defaults and setting the interest rates accordingly.

Infrastructure here refers not just to the flow of traffic, although that's included, but also to the flow of people in general, the flow of money and most importantly the

flow of of information. Like artificial intelligence, intelligent infrastructure comes with risks. But here the risk is not so simple as the AI not opening the pod bay doors when we tell it to. The risks come primarily from unintended consequences that stay hidden, and are difficult to measure.

An intelligent infrastructure is not a system that is built and tested all at once. It's something that emerges step by step as people replace human decision making with automated decision making. It's not just controlled by engineers, but also by project managers, third parties, company managers. At the largest level, the network doesn't even come under the control of one government. Even if Europe gets ahead of the curve, large parts of the infrastructure we use may be hosted in the US, or in China, where different rules apply and different levels of oversight are possible.

image source: photo by Ian Beckley from Pexels

Feedback loops

Blind optimization

Predictions vs. actions

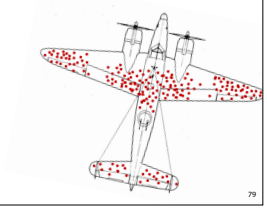
78

These are probably the three main categories of issues to be wary of when you're working on some piece of infrastructure that will make decisions automatically.

We've seen examples of each already, in the lectures in general and in the previous social impact videos. We'll look at some new examples to highlight the risks specifically from the perspective of intelligent infrastructure.

Pittsburgh, 1995

```
if has_asthma(X) then pneumonia_risk(X, low)
```



predictions vs. actions

79

We'll start with a classic example. In the early 90s, Pittsburgh Medical Center started a project to investigate ways to make their health care more cost effective: to achieve better results with the same resources. One thing they decided to focus on was community acquired pneumonia (CAP). A lung infection acquired from other people outside of the hospital.

Pneumonia is sometimes relatively benign and sometimes leads to sudden and quite severe adverse reactions, and even death. The reasoning was that if risk factors could be identified that predicted such highly adverse reactions, patients could be monitored in a more effective way, and perhaps deaths could be

prevented.

The researchers trained a rule based system: a type of machine learning that learns discrete if-then rules that hold for a majority of the data. Such systems are less popular today, since their performance tends to be much lower than that of modern methods, but they do have one advantage. If you keep the number of rules small, the model becomes very easy to inspect. You can see exactly what your model has learned in a very interpretable format.

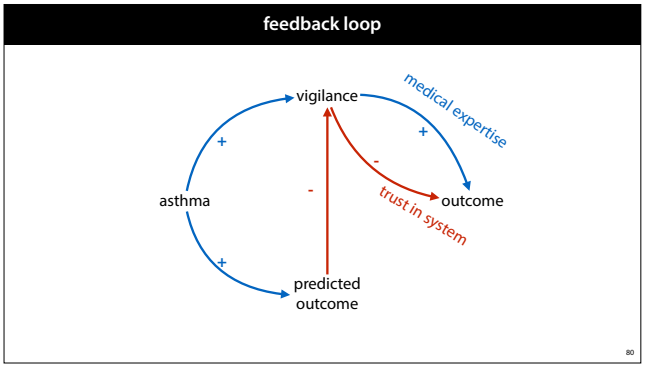
One of the rules the model learned was this one: patients with asthma had a much lower risk of developing strongly adverse affects from pneumonia.

This was a counter-intuitive result. Asthma is a lung condition, and any doctor will tell you that catching pneumonia is much more dangerous for an asthmatic person than for others. What happened here, was that doctors and asthmatic patients were already being much more careful. Patients with asthma know that they should be more watchful for signs of pneumonia, and doctors know that such patients require more active care.

Nevertheless, if we had trusted the system blindly, or if we had used a neural network which would not have allowed us to inspect it in this way, we would end up lowering our vigilance for asthmatic patients presenting with signs of pneumonia.

This is yet another example of our data coming from a biased distribution, like the planes in world war 2: we are not seeing what would happen to asthma patients if their were treated the same as everybody else, so our inference is biased. Here as then our **predictions** are entirely accurate: we can predict very well where planes coming back will be hit, and we can predict very accurately what will happen to asthmatic patients admitted to hospital with signs of pneumonia.

What's going wrong are the implied **actions**, we decide to attach to that prediction. More often than not, this is not a concious choice ans we simply confuse accurate predictions with sound actions.

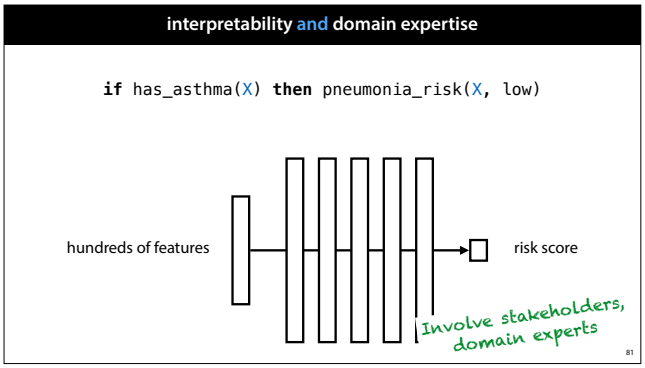


Here's a sketch of how blindly trusting this system, introduces a feedback loop.

Before we introduce our system, there is positive feedback from having Asthma to the hospital staff being more vigilant. This improves outcomes.

When we train our system it pick up on the resulting correlation, and having asthma leads to a better predicted outcome. This is a correct inference.

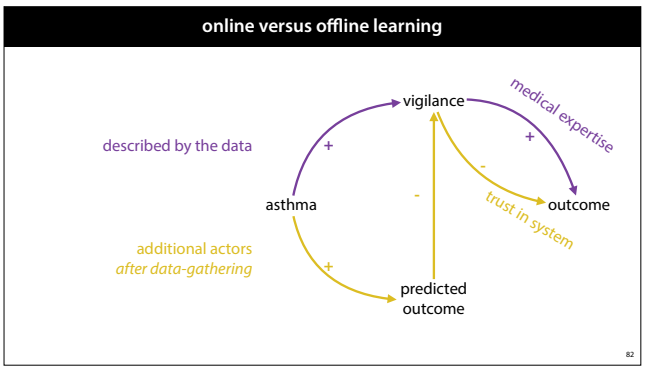
However, if we use that prediction to take action naively, we reduce the vigilance of doctors towards asthma patients, worsening the outcomes.



In this case, there are two important factors stopping this feedback loop from being put into production, and costing lives. First the fact that a rule based system was used. This allowed us to inspect what the system was learning and to pick up on the fact that a counter-intuitive rule was coming out.

Second, the fact that domain experts were consulted, in this case, doctors. A doctor can look at rules like these and tell you that they're wrong, which helps you to see that you've made a fundamental mistake in your reasoning. In this case the people most affected by the system, asthma patients, would also have been to tell you this. We call the people that are affected by the system **stakeholders**, and the people who study the domain of the data **domain experts**. Both should be consulted in the design of a system.

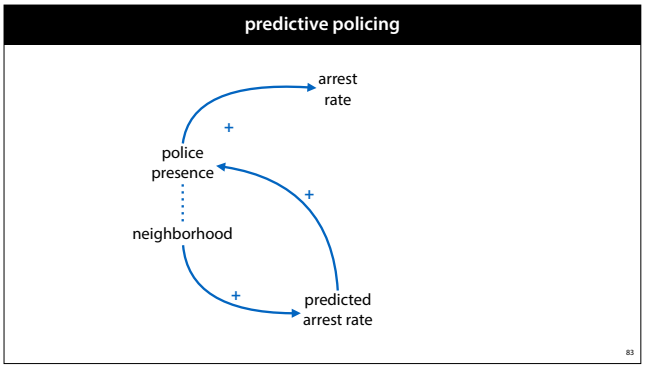
Note that domain experts and stakeholders can only be involved of the system is made interpretable in some way. That doesn't mean your model needs to consist of discrete rules, but it does mean that you somehow need to make the behavior of your system inspectable for people that don't have a machine learning background. How to do this, is an active area of research.



A key problem here is that when we are using offline learning to make predictions, we are taking our data as a static snapshot of the world.

That snapshot accurately describes the world, but only the world without our system in it. Once we take actions based on the predictions, our system becomes an additional actor in the world, and our data does not represent the world with that actor in it.

In theory, we could deploy the system, gather more data, retrain the system. deploy and repeat. If we're lucky, the system will eventually converge to a stable state where it has the right idea about how asthma influences the outcome. But it only learns this after it has reduced vigilance for a sizeable number of patients, likely costing lives.



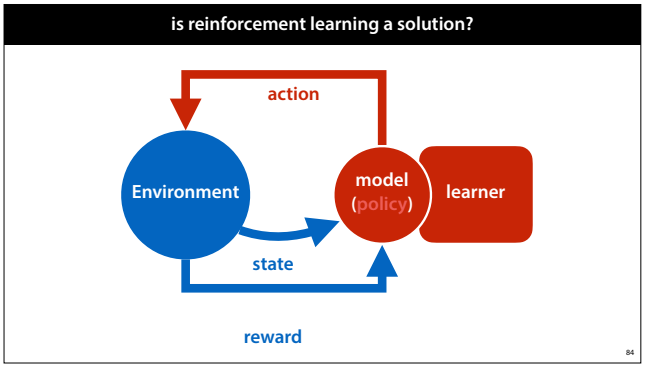
Here is another example, predictive policing. This is strongly related to the profiling question we discussed in the last social impact video, but it shows that we have a problem even if we take race out of the equation.

If we increase the police presence in a particular neighborhood, that will increase the arrest rate. Regardless of whether it's a tranquil or crime-ridden neighborhood, the more police, the more arrests.

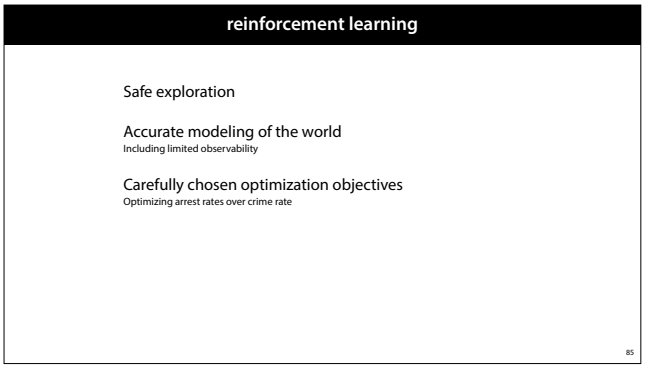
Imagine that we want to train a model to predict where the crime in a city is most likely to appear. We don't know exactly where crime is most likely, since many crimes go undetected. An obvious measure to use instead is the number of arrests in a neighborhood. This is called a **proxy measure**: something that doesn't quite measure what you want to measure, but is close in enough, and is easier to measure accurately. In this case, it's quite likely that people use arrest rate as a proxy for crime rate without realizing that they're doing it.

Of course, you know where this is going. If we build predictor that includes the neighborhood as a feature, and predicts the arrest rate, we end up with a system that, accurately, predicts high high arrest rates for neighborhoods with police presence. If we then interpret arrest rate as crime rate and increase police presence in areas where the predicts predicts more arrests, we just end up increasing the disparity in police presence between neighborhoods. If there is some bias to start with in neighborhood police presence, we end up compounding the problem.

Note that this is a case where gathering more data and retraining will not solve the problem. We only have positive feedback loops, so if we blindly follow the system, every time we retrain, we end up with more concentration of the police presence, until all the police and all the arrests are in one neighborhood in the city. The system will be making perfect predictions at that point, but it's unlikely to lead to a safer city.



All this shows an important difference between reinforcement learning and classical offline learning. A reinforcement learning model has the option to model itself as an agent in the world. It learns to act rather than to predict, so in theory, it can control the consequences of its actions, even if those actions change the distribution of the data.

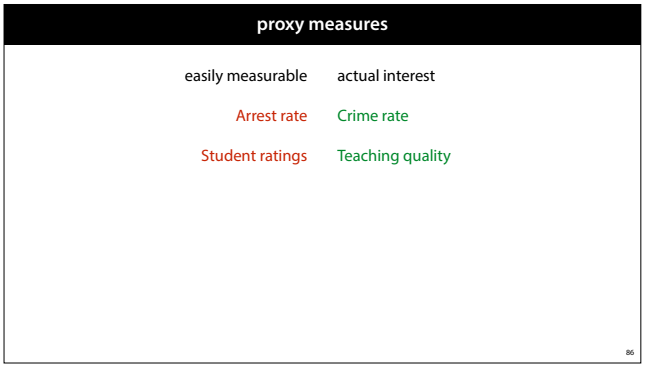


Of course all of this is easier said than done. Just letting a neural network loose in the world and letting it learn by optimization is unlikely to make the world a safer place.

First of all, such an algorithm learns by **exploration**: it needs to try things and observe the consequences. Letting it reduce the vigilance on asthmatic pneumonia patients may eventually lead to sounds actions, but if we let it do this in the real world, it will take a long time before it figures out things that we already know. Safe reinforcement learning is an active field, but it's in its infancy.

Another thing we need to worry about, is that the model works on an accurate model of the world. We may set the objective that a self-driving car should get to its destination without running red lights, but if it's red light detector is a neural network, then we need to make sure that it doesn't optimize its performance by getting worse at detecting red lights. After all, if it's red light detector doesn't fire, it's free to drive as fast as it likes.

And finally, and most importantly, reinforcement learning still optimizes a single metric. If we pick that metric wrongly, we can still build a very dangerous system. Take the predictive policing case. We could build a reinforcement learning system to maximize the arrest rate, but what we're actually interested in minimizing the crime rate. These are not the same thing, and maximizing the arrest rate is much more likely to lead to police nuisance than to actual reductions in crime.



Another case is student evaluations: taken together with other metrics, these can help to paint a complete picture of a teacher's performance. If, however, we look only at evaluations, then we are just pressuring teachers to make students happy. For instance, setting too easy an exam, to reduce the possibility of complaints.

Goodhart's law

Goodhart (1975): "Any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes."

Strathern (1997): "When a measure becomes a target, it ceases to be a good measure."

87

A relevant adage here is Goodhart's law, which was restated in more simple terms by Marilyn Strathern. Often the proxy measure is often a really good way to measure hwat we're interested in. So long as the proxy and the true quantitiy are correlated and we're a bit careful, we can get quite a good idea of what's going on and where the potential improvements are. The problem happens when we put pressure on people to minimize or maximize the proxy measure. Then it stops being a reliable indicator of what we're actually trying to measure.

For instance, in many cases the arrest rate in a neighborhood may be a reasonable proxy for the crime rate. It's not perfect, but it's often close enough, especially when we look at different factors like income, substance abuse and financial security as well. However,often the police is actively pressured to either minimize the arrest rate (to show that crime has reduced), or to maximize it (to show that progress is being made). In that case, the crime rate stops being a usefull proxy, and we need to look to other measurements to see if the pressure is actually working, or people are just fudging the statistics, or even worse arresting people that shouldn't be.

Here's another example: "chuggers" short for charity muggers are the people with clipboards that charities use to get people to donate in the street.

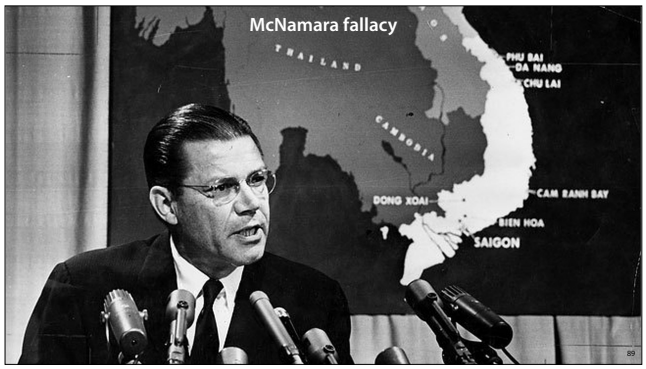
Before deciding to employ these tactics, the charity likely kept an eye on how many people sign up to donate. This is an important measure for a charity: it's indicate how much income they get how able they are to survive, to affect change and also how positive people feel about them.

However, the implementation of a tactic like this is likely based on a violation of Goodhart's law. The charity decides to employ chuggers, and sees an uptick in registrations concluding that the method must work. In doing so, they forget that donations are just a proy measure for what they actually care about. If one in twenty people sign up to donate, we'll get an uptick in donations, but what about the nineteen people that didn't sign up? If they were annoyed by the chugger, their appreciation of the charity will be diminished. They will come away thinking less of the charity, and being less receptive to its message in the long run.

This illustrates an important effect behind many instances of Goodhart's law being violated: we care about many different things that are hard to measure, so we end up optimizing for a single thing that is easy to measure.

In many ways, people with machine learning and computer science backgrounds are at an extra risk of

this, since they are so used to optimizing for single metrics.



A strongly related problem is the **McNamara fallacy**. Robert McNamara was the US secretary of Defense for most of the sixties and oversaw a large part of the Vietnam war.

McNamara was an early pioneer of data driven management, first as a manager at Ford, and then in the US government. During the Vietnam war, the focus was entirely on measurable metrics. Often such gruesome ones as the numbers of Vietcong and US soldiers killed. All processes and policy decisions were shaped around such statistics. This had two detrimental effects.

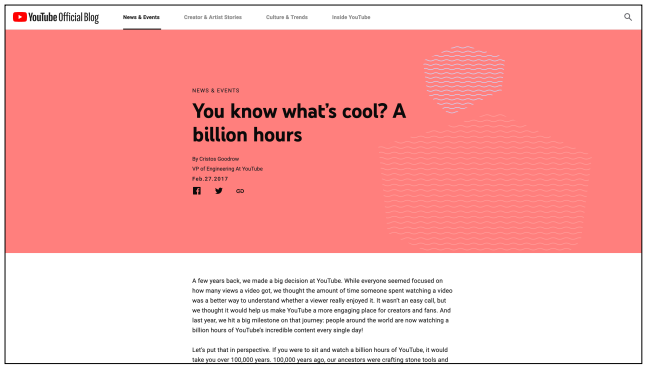
First, it ignored things that were difficult to measure, such as the Vietnamese sentiment towards the US, which turned the general population against the US, and towards the Vietcong.

Second, it incentivised the military to present positive numbers. This lead to generals putting arbitrary caps on what numbers of enemy troops could be reported, and redrawing categories like the army command structure to make the progress of the war look better, so that congress would commit to more sending more troops abroad.

For more than a decade, the data suggested that the US was winning the war. And, then, in the early 70s, around the time Goodhart first formulated his principle, the US withdrew,

In short, since quantitative measurement is such a powerful tool, it can become addictive, to the point where only easily measured quantities are used to guide policies and decisions.

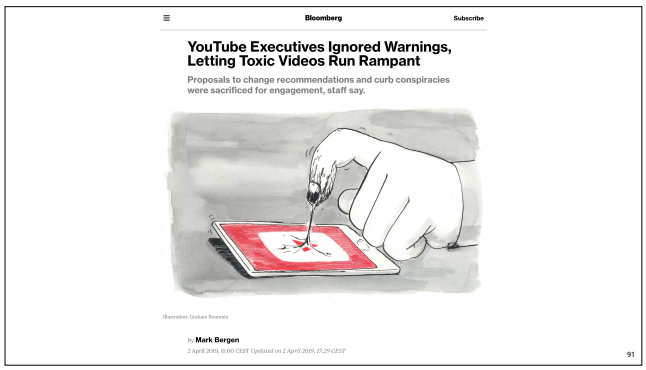
source: **https://www.forbes.com/sites/jonathansalembaskin/2014/07/25/according-to-big-data-we-won-the-vietnam-war/?sh=33dc487f3f21**



Here's another example. Somewhere in the early 2010s, executives at YouTube decided that they shouldn't optimize their recommendation algorithm for clicks on videos, they should instead optimize for total amount of time people spent watching video.

This was in general a good idea: it's a better proxy measure. Optimizing for clicks lead to people using clickbaity titles and thumbnails, with little content behind it. Optimizing for viewing time requires authors to put the work into the content of the video, and to keep users watching, if they want to be promoted by the youtube recommender system. But, it's still a proxy measure.

What's more youtube set an arbitrary goal of a billion hours of video watched per day. They didn't just make hours watched the objective for the recommendation algorithm, they made it the objective for a company as a whole.



Youtube bossted its recommendation engine with deep neural networks and reinforcement learning. All with a single goal: to increase engagement.

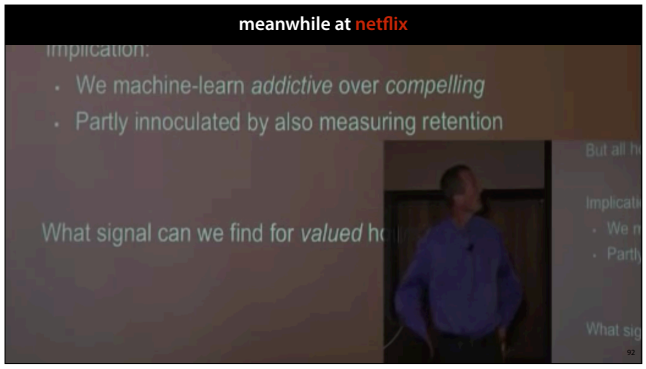
The precise effects are difficult to ascertain. Youtube is not forthcoming with details about its algorithm, and researchers only began to study the system quantitatively from the outside in around 2019, by which time Youtube seemed to have tempered its hunger for engagement.

Nevertheless, in the years between 2012 and 2019, youtube faced an extreme amount of scrutiny from the media. It's recommender system was recommending unsuitable content to children, favouring more politically extreme content, and doing everything it could to keep people hooked on its videos.

sources:

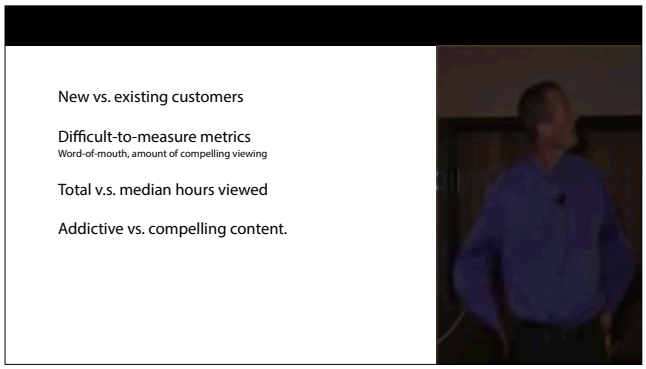
https://www.bloomberg.com/news/features/2019-04-02/youtube-executives-ignored-warnings-letting-toxic-videos-run-rampant

https://www.nytimes.com/2018/03/10/opinion/sunday/youtube-politics-radical.html



By contrast, here is Neil Hunt chief strategy officer at Netflix, in 2014 giving a keynote about the content strategy at Netflix.

source: **https://youtu.be/lYcDR8z-rRY?t=4559**



Note that he explictly talks about:

- The tradeoff between optimizing for new customers and keeping your existing customers happy
- The importance of non-measurable metrics like word-of-mouth.
- That Netflix doesn't optimize for total hours viewed. Since this is proportional to the average, optimizing for it means paying disproportionate attention to outliers: the people who are already spending hours and hours on your service. Optimizing for the mean focuses your attention on those customers who are spending a moderate amount.
- The fact that the recommender system doesn't know the difference between *addictive* and *compelling* content. One hour of of content may be a life changing experience, while six hours of bingeing 1990s sit coms may leave you feeling empty and ready to cancel your subscription
- Making your users addicted to your product makes them unhappy in the long term. For a service like netflix that is ultimately counterproductive.

These strategies were followed by Netflix and Youtube more or less at the same time. The picture of Youtube is one of a single-metric strategy being pushed top-down by management, and shaping the corporate culture as well as the algorithms that it produces.

The Netflix picture is one of a management that actively thinks about the limitations of its algorithms. About the things it cares about that are difficult, or hard to measure, and even about how its technology is affecting the lives of their customers beyond engagement.

Of course, Netflix has the luxury position that it's not driven by advertisements, but then the combination of advertising and recommender systems may itself be part of the problem.

Beware of corporate culture
Go fever, groupthink

Don't optimize for a single metric
Beware of **proxy metrics**

Goodhart's law: optimize for one thing, measure another.

Involve stakeholders, domain experts

Keep humans in the loop

Distinguish between actions and predictions

Many of these effects aren't new. In fact, Goodhart's law was first formulated in 1975 long before automated decision making became prevalent. Most of the biases, feedback loops and blind optimization approches we've discussed during these four videos on social impact predate computers and happen just as often in organizations made up purely of human agents.

The dangers in intelligent infrastructure are in the compounding of all these feedback loops, the lack of human oversight, the monoculture effect of such software being rolled out uniformly across the globe, and finally the danger that each institution optimizes its own systems for local benefits like revenue, without looking at the global effects.

mlcourse@peterbloem.nl

Many of the examples in this video were borrowed from Anna Koop's coursera course optimizing machine learning, which I highly recommend if you're interested in building production machine learning systems responsibly.