Если вы видите что-то необычное, просто сообщите мне.

Day 10: Beyond Supervised Learning

Ever wondered how machines defeated the best human Go player Lee Sedol in 2016? A historical moment for the game that was previously considered to be very tough. What is reinforcement learning that generates so much buzz recently? Superhuman performance playing arcade Atari games, real-time Tokamak plasma control, Google data center cooling, and autonomous chemical synthesis are all the recent achievements behind the approach. Let's dive to learn what empowers deep reinforcement learning.

Introduction

Reinforcement learning (RL) is a machine learning technique that is all about repetitive decision making. Well, just like when you go camping. During your first trip you realize that you need more food and you lack some tools. Next time, you take some more cereals and dried fruit, a better knife, and a pot. And after yet several more attempts you invest in better shoes or a backpack and learn how to arrange your stuff in a neat way. The journeys become longer and more enjoyable. As you can see, you iterate. You learn from experience. And you get rewarded.

No matter how much you can relate yourself to the experiences above, they contain everything that reinforcement learning has. An agent is learning from experience (or from mistakes if you want) by interacting with its environment. The agent receives some sort of a reward signal in response to its actions. And that is basically the idea. An idea from psychology? Perhaps.

In a supervised learning setting1 you would be instructed what equipment to take with you. And you would need to act exactly as instructed. However, the real life is not like that. You might have watched a camping channel on YouTube and still struggle to start a fire. Trying different kinds of approaches yourself, however, improves your survival skills over time.

> The difference between supervised and reinforcement learning The biggest difference between supervised learning and reinforcement learning is that in reinforcement learning the agent creates its own dataset by interacting with the environment. You may also notice that reinforcement learning is actually more general than supervised learning. Indeed, in supervised learning we usually talk about a single iteration and apply some sort of ground truth or target signal. However, let's take a look at the following example. Imagine, you are a director of AI at Tesla (those hipsterish cars, you know) and your team trains a self-driving car based on videos that were previously recorded. Sounds like a supervised problem with images and labels, right? The car performs well during the same day, but tomorrow it rains and the car refuses to drive well. What do you do? You say, perhaps the distribution of images has changed. Let's collect some more for a rainy weather. And the model training is repeated. The third day is very foggy and your team has to collect the data again. And train the model again. So what happens? Exactly! Iteration. You have an implicit reinforcement learning loop where it is you who acts as an agent trying to outsmart the weather conditions2.

More often than not present decisions will influence the situation, and thus they will influence future decisions too. This is perhaps the biggest difference from supervised learning. Supervised learning conveniently omits to confess that its main goal is to help making decisions in real world. The most boring vision task you have encountered, classifying MNIST images, at some point was actually useful for post offices to automate mail delivery. Similarly, predicting time series might be helpful getting an idea about climate change or predicting economic downturns. Whatever supervised learning benchmark you take, in its origin there is some sort of implication in the real world.

Unlike supervised learning, reinforcement learning is all about decisions. It explicitly states the goal of achieving the best overall consequences -- by maximizing total reward. For instance, to arrive from point A to point B you may take a car. Depending on which route you will take and how frequently you will rest, this will affect your trip duration and how tired you will arrive. Those circumstances may further affect your plans. Decisions almost never occur in isolation. They will inevitably lead to new decisions. That is why real-life challenges are often accompanied by reinforcement learning in some form.

A quick note. People often talk about deep reinforcement learning. This is to highlight the presence of neural networks. The fact does not change the essence of the method. These days almost everyone is doing deep reinforcement learning anyway, so I prefer to omit the term. Just wanted to make sure we stay on the same page. Now, we will focus our attention on the nitty-gritty details of reinforcement learning.

Reinforcement Learning Concepts

By interacting with environment, an RL agent actually creates its own unique dataset, whereas in supervised learning the dataset is predefined. As the agent explores its environment and the dataset is being created we apply backprop to teach our agent similarly to supervised learning. It turns out there is a large variety of ways how to do that. The concepts below will give you a taste about the key ideas from reinforcement learning.

Policies, States, Observations

A policy (denoted by π) is simply a strategy that an agent uses when responding to changes in the environment. The following notation reads "a policy parametrized by parameters ϕ to take an action a given a state s":

$$\pi_\phi(a|s).$$

Typically ϕ signifies weights in a neural network (the deep reinforcement learning story). In the notation above, by s people usually mean "state". And sometimes they actually mean "observations" o. The difference between the two is that observations are a projection of the complete state describing the environment. For example, when learning robotic manipulations

from camera pixels, observations of a robotic hand are only a representation of the actual environment state from a certain angle. Sometimes people talk about "completely observable" environments, then s=o. The distinction between s and o does not affect the RL algorithms we are learning about today.

On-policy vs Off-policy

Reinforcement learning algorithms can be split between on-policy and off-policy. The first ones use trajectories (experiences)3 generated by the policy itself, while the second ones use trajectories from some other policies. For instance, algorithms that use a replay buffer (such as deep Q networks playing Atari games) to store transitions are off-policy algorithms. Typically, off-policy algorithms have better sample efficiency compared to on-policy ones, which makes them attractive when it is costly to generate new experiences. For instance, when training a physical robot. On the other hand, on-policy algorithms are often used when acquiring new experiences are cheap. For instance, in simulation.

Offline Reinforcement Learning

Offline RL is a family of algorithms where an RL agent observes past experiences and tries to figure out a better policy without interaction with an environment. Of course this is a very challenging task. Nevertheless, it has many real-world applications where it could be dangerous or illegal to apply reinforcement learning experiments. For instance, finding the best treatment in the medical setting. A doctor would not experiment on a patient using different drugs because this can lead to health deterioration. Instead, they may use reinforcement learning techniques applied to previous records of other patients to figure out the best prescription (offline setting).

Discrete vs Continuous Actions

Playing an arcade game using a manipulator with four buttons implies a discrete action setting (four possible actions). However, in reality there are settings with infinite or very large number of

actions. For instance, robot joints positions. Certain algorithms can be applicable only to one action type (DQN can be applied only for discrete and DDPG only for continuous actions), whereas certain others can be used for both (e.g. A2C, PPO).

Deterministic vs Stochastic Policies

Certain policies inherently produce deterministic actions (DQN), while others sample from a learned distribution producing stochastic actions (PPO). Of course, it is possible to make a deterministic algorithm to produce stochastic actions (epsilon-greedy exploration) and vice versa. Why use a stochastic policy? Actually, under stochastic environments optimal policies are often stochastic. The simplest example is Rock-Paper-Scissors game.

Model-free vs Model-based

Model-based approaches assume a certain environment model. The advantage of this approach is better sample efficiency. On the other hand, model-free approaches may better generalize to unknown environments. In some cases, environment's model is learned.

Sample Efficiency

By sample efficiency we mean using less examples for training.

Reinforcement Learning Hints

The goal of reinforcement learning is typically to solve a real-life challenge. For that purpose you will need to define some interaction protocol between your agent and its environment and also a reward function. The reward function provides a score how well your agent performs. Typically, it is

beneficial having a "dense" reward so that the agent gets some information at every time step. This is not always possible and there are methods to circumvent this issue. One way to look at reward shaping is that RL is your "compiler" and the reward is your understanding of the problem you are trying to solve.

To gain some expertise, you may want to start with existing environments4 and RL algorithms. If your goal is to train a physically embodied robot, you may want to create your own environment to try out some ideas in simulation. This will save you some time since running a simulator is typically faster than real-time robot training.

The more accurate is the simulator, the better the agent is likely to perform in real-world. For instance, in Sim-to-Real: Learning Agile Locomotion For Quadruped Robots the authors have obtained better results by creating accurate actuator models and performing latency modeling.

However, there is always a difference between running an agent on any simulator and reality. The problem known as reality gap. There are different techniques how to reduce this difference. One of them is domain randomization. The idea is to train the agent using environments with randomized parameters. In active domain randomization, the active learning approach is combined: agents are trained more often on environments where they performed poorly.

Another technique domain adaptation suggests making real data look like those in simulation or vice versa. You may also want to perform parameters fine-tuning on your real robot.

Note also that if you are training an agent in a simulated environment, the agent may "overfit" to the environment. That is, it may exhibit unrealistic (and often undesired) behaviors that still lead to high reward scores. The agent may try to exploit environment glitches if there are any.

Having those practical strategies in mind, we are going to the meat of RL. Below we are discussing an algorithm which constitutes the basis to many modern RL strategies such as TRPO and PPO.

REINFORCE

REINFORCE, also called Monte Carlo Policy Gradient, is the simplest from the policy gradient algorithms family. The advantage of policy gradients is that they can learn stochastic policies. This

also addresses the exploration-exploitation dilemma.

REINFORCE algorithm:

Initialize policy parameters ϕ. Generate trajectories $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$ by interacting with environment. For every step t

$0, 1, \ldots, T-1$, estimate returns R_t

$\sum_{T}^{k}$

$t+1 \gamma^{k-t-1} r_k$. Optimize agent parameters $\phi \leftarrow \phi + \alpha R_t \nabla \log \pi_\phi(a_t | s_t)$. Repeat steps 2-5 until convergence. Note that when using universal function approximators (that is neural networks), convergence is not guaranteed. However, in practice you still have a good chance to

train your agent.

numEpisodes = 400

main :: IO () main = do putStrLn $ "Num episodes " ++ show numEpisodes

-- Seed Torch for reproducibility. -- Feel free to remove. manual_seed_L 10 let seed = 42 -- Environment seed

-- Step 1: Initialize policy parameters agent <- mkAgent obsDim actionDim -- We also initialize the optimizer let trainer = mkTrainer agent

-- Repeat steps 2-4 for the number of episodes (trajectories) (agent', trainer') <- foldM (\at i -> (reinforce conf) seed at i) (agent, trainer) [1..numEpisodes]

return ()

reinforce cfg@Config {..} seed (agent, trainer) i = do -- Step 2: Trajectories generation (rollout) let s0 = Env.reset (seed + i) (_, _, !rs, !logprobs_t) <- rollout maxSteps agent s0 let logprobs' = cat (Dim 0) logprobs_t putStrLn $ "Episode " ++ show i ++ " - Score " ++ show (sum rs)

-- Step 3: Estimating returns let returns_t = asTensor $ returns γ rs returnsNorm = (returns_t - mean returns_t) / (std returns_t + 1e-8)

-- Step4: Optimize optimize cfg logprobs' returnsNorm (agent, trainer) Trajectory Generation The agent generates a trajectory by interacting with the environment. We call this a rollout. By observing the function signature below, we can tell that the function consumes an integer number, an agent, and environment state. This integer is simply the maximal number of steps per episode. As a result, the function will provide the trajectory: observations, actions, and rewards. In addition, it also provides log probabilities, which we will use to compute our objective (or loss) before we can optimize parameters in Step 4.

rollout :: Int -> Agent -> Env.State -> IO ([Observation], [[Int]], [Reward], [Tensor]) -- ^ Observations, actions, rewards, log probabilities Here is how we implement it: we benefit from the excellent unfoldM function that has type Monad m => (s -> m (Maybe (a, s))) -> s -> m [a]. That is we iterate as long as the function in the first argument (s -> m (Maybe (a, s))) provides a Just value (and stop when Nothing).

Haskell libraries provide lots of "pieces of code" or "programming templates": functions like map, foldl, scanr, unfoldr, zip, zipWith, etc. All those are compact versions of loops! Some of those concepts gradually diffuse into imperative languages (such as C++ and Python).

rollout _maxSteps agent s0 = L.unzip4 <$> unfoldM f (_maxSteps, agent, s0) where -- Reached max number of steps. Nothing = stop. f (0, _, _) = pure Nothing

```
f (_maxSteps, _agent@Agent{..}, _s) = do
  if Env.isDone _s
    -- The environment is done: stop.
    then do
       pure Nothing
    else do
      let ob = Env.observations _s
      (ac@(Action ac_), logprob) <- getAction ϕ ob
      let (r, s') = Env.step ac _s

      pure $ Just ((ob, ac_, r, logprob), (_maxSteps - 1, _agent, s'))
```

The heart of iteration is in the end: First, observe the environment and sample actions.

let ob = Env.observations *s (ac@(Action ac*), logprob) <- getAction ϕ ob Then, simulate the environment step:

let (r, s') = Env.step ac _s And finally, prepare the next iteration step by reducing the maximal number of steps and passing the agent and the new environment state.

pure $ Just ((ob, ac_, r, logprob), (_maxSteps - 1, _agent, s')) Here is how we get an action and a log probability.

getAction ϕ obs = do let obs_ = unsqueeze (Dim 0) $ asTensor obs (ac, logprob) <- evaluate ϕ obs_ Nothing

-- Return a single discrete action return (Action [asValue ac], logprob) Evaluate the policy π ϕ : If no action provided, sample a new action from the learned distribution. Also get log probabilities for the action.

evaluate ϕ obs a = do let probs = policy ϕ obs dist = Categorical.fromProbs probs action <- _getAct a dist let logProb = D.logProb dist action pure (action, logProb) where -- Sample from the categorical distribution: -- get a tensor of integer values (one sample per observation). _getAct Nothing dist = D.sample dist [head $ shape obs] _getAct (Just a') _ = pure a' Estimating Returns After a rollout we (retrospectively) compute how good were our decisions during the whole trajectory. The trick is that we start from the last step. That is, our return R at time step t is our current reward plus a discounted future return.

R t

r t + γ R t + 1 .

This expands into

R t

r t + γ (r t + 1 + γ (r t + 2 + γ (. . .))) .

where r t is the reward at time t . The meaning of this expression is the essence of policy gradient: We evaluate how good were our past decisions with respect to the future outcomes.

An intriguing way to look at discount factor γ is by using the concept of terminal state (to put simply, death). At each future step, the agent can get a reward with probability γ and can die with probability 1 − γ . Therefore, discounting the reward is akin to incorporating the "fear of death" into RL. In his lectures, S. Levine gives an example of receiving 1000 dollars today or in million years. In the latter case, the reward is hugely discounted as it is unlikely that we will be able to receive it.

We compute the returns as a function of rewards rs. We also use next terminal states indicators and future value estimators in special cases.

returns γ rs = f rs where f (r:[]) = [r]

```
f (r:xs) =
  let y = f xs
```

```
    -- Discounting a future return
    in r + γ * (head y) : y
```

In policy gradient algorithms we evaluate how good are our past decisions with respect to the future outcomes.

Note that we compute the return recursively as in equation above: reward plus a discounted future return. Since Haskell is a lazy programming language, it is not critical that this value does not exist yet. Essentially, we promise to compute the list of future values y = f xs. Finally, we prepend current return to the list of future returns r + γ * (head y) : y. Fascinating!

f (r:xs) = let y = f xs in r + γ * (head y) : y Optimizing parameters Computing the loss. Running a gradient step. optimize :: Config -> Tensor -> Tensor -> (Agent, Trainer) -> IO (Agent, Trainer) optimize Config {..} logprobs_t returns_t (Agent {..}, Trainer {..}) = do let loss = Torch.sumAll $ -logprobs_t * returns_t (ϕ', opt') <- runStep ϕ opt loss (asTensor lr) pure (Agent ϕ', Trainer opt') Final Bits There are still a few other things to complete the project. Let's define our policy network type Φ . Here we have three fully-connected layers. In other words, two hidden layers.

data Phi = Phi { pl1 :: Linear , pl2 :: Linear , pl3 :: Linear } deriving (Generic, Show, Parameterized) Now we can implement the forward pass in a Policy Network:

policy :: Phi -> Tensor -> Tensor policy Phi {..} state = let x = (linear pl1 ~> tanh ~> linear pl2 ~> tanh ~> linear pl3 ~> softmax (Dim 1)) state in x An agent is simply a policy network in Reinforce. We will update this data type to also accommodate the critic network in improved policy gradient later on:

data Agent = Agent { ϕ :: Phi } deriving (Generic, Show) REINFORCE Trainer type is a single optimizer.

data Trainer = Trainer { opt :: Adam } deriving Generic A new, untrained agent with random weights

mkAgent :: Int -> Int -> IO Agent mkAgent obsDim actDim = do let hiddenDim = 16 ϕ <- samplePhi obsDim actDim hiddenDim pure $ Agent ϕ Parameters ϕ ∈ Φ initialization

samplePhi :: Int -> Int -> Int -> IO Phi samplePhi obsDim actDim hiddenDim = Phi <$> sample (LinearSpec obsDim hiddenDim) <> *sample (LinearSpec hiddenDim hiddenDim)* <> sample

(LinearSpec hiddenDim actDim) Initializing the trainer

mkTrainer :: Agent -> Trainer mkTrainer Agent {..} = let par = flattenParameters ϕ opt = mkAdam 0 0.9 0.999 par in Trainer opt Now, let's train the agent to solve the classic CartPole environment! See the complete REINFORCE project on Github.

Proximal Policy Optimization REINFORCE algorithm is nice because it is simple. On the other hand, in practice it has a problem: finding the best meta-parameters, such as learning rate. Choose a value to low and training needs more samples, too high and it does not converge. To alleviate this training instability multiple variations of this algorithm have been proposed. One popular variation is called Proximal Policy Optimization (PPO). Below we upgrade REINFORCE to PPO.

Actor-Critic Style In REINFORCE we only had a policy network π ϕ . A more advanced version would be not only to use a policy network (so-called actor), but also a value network (critic). This value network would estimate how good is the state we are currently in. Naturally, the output of the critic network is a scalar with this estimated state value.

-- Value (Critic) Network type: Three fully-connected layers data Theta = Theta { l1 :: Linear , l2 :: Linear , l3 :: Linear } deriving (Generic, Show, Parameterized)

-- | Forward pass in a Critic Network critic :: Theta -> Tensor -> Tensor critic Theta {..} state = let net = linear l1 ~> tanh ~> linear l2 ~> tanh ~> linear l3 in net state Here is how we will sample initial network weights:

sampleTheta :: Int -> Int -> IO Theta sampleTheta obsDim hiddenDim = Theta <$> sample (LinearSpec obsDim hiddenDim) <> *sample (LinearSpec hiddenDim hiddenDim)* <> sample (LinearSpec hiddenDim 1) To make our life a bit simpler, we can also use the following wrapper when dealing with raw observations.

-- | Get value: Convenience wrapper around `critic` function. value :: Theta -> [Float] -> Float value θ ob = let ob_ = unsqueeze (Dim 0) $ asTensor ob in asValue $ critic θ ob_ The state value given by the critic network will be used for advantage estimation, instead of simply calculating discounted returns.

Advantage Estimation As you remember, in policy gradients we optimize neural network parameters using A · ∇ log π ϕ (a t | s t) . In REINFORCE, this term A

R t is discounted returns. This totally makes sense. However, this is not the only option5.

Let me introduce advantage A

r − v where r is reward and v is value, i.e. how good is our action. The advantage tells us how current action is better than an average action. Therefore, by optimizing our policy with respect to advantage, we would improve our actions compared to average. This would help to reduce variance of policy gradient.

In PPO, we typically use a fancy way to estimate the advantage called generalized advantage estimator (GAE).

advantages γ γ' rs dones vs = f $ L.zip4 rs (L.drop 1 dones) vs (L.drop 1 vs) -- vs are current values (denoted by v) and -- (L.drop 1 vs) are future values (denoted by v') where -- Not necessary to

reverse the list if using lazy evaluation f :: [(Float, Bool, Float, Float)] -> [Float] -- End of list to be reached: same as terminal (auxiliary value) f ((r, _, v, _):[]) = [r - v]

```
-- Next state terminal
f ((r, True, v, _):xs) = (r - v) : f xs

-- Next state non-terminal
f ((r, False, v, v'):xs) =
  let a = f xs
      delta = r + γ * v' - v
   in delta + γ * γ' * (head a) : a
```

PPO Specifics In REINFORCE, the policy gradient loss function was simply

$$L(\phi)$$

$$-\sum t$$

$$\log \pi \phi (a t | s t) \cdot R t .$$

Note the negation sign in front: without it, the loss (to be minimized) becomes an objective (to be maximized) - earning highest returns6:

$$L(\phi)$$

$$\sum t$$

$$\log \pi \phi (a t | s t) \cdot R t .$$

Now, what distinguishes PPO, it its objective. It consists of three components: a clipped policy gradient objective (actor network), value loss (critic network), and entropy bonus. Since the value function is a loss term, it has a minus sign:

L P P O t

L C L I P t − c 1 L V F + c 2 S ,

where c 1

0.5 and c 2

0.01 are constants. If we are going to minimize loss instead,

L P P O t

− L C L I P t + c 1 L V F − c 2 S

= L P G t + c 1 L V F − c 2 S .

loss = pg + vfC `mulScalar` vLoss - entC `mulScalar` entropyLoss Now, the most

interesting part of PPO. If we denote r t (ϕ) the probability ratio r t (ϕ)

π ϕ (a t | s t) π ϕ old (a t | s t) . Then PPO is maximizing the following objective

L C L I P (ϕ)

^ E t (min (r t (ϕ) ^ A t , clip (r t (ϕ) , 1 − ϵ , 1 + ϵ) ^ A t)) .

First, let's take a look at the clipped objective clip (r t (ϕ) , 1 − ϵ , 1 + ϵ) ^ A t . Here, we try to restrict how far our policy will move during the policy gradient update. If advantage A is positive, the objective is clipped at value 1 + ϵ . If advantage A is negative, then the objective is clipped at 1 − ϵ . Finally, we take a min between a clipped and unclipped objective. Therefore, the final objective is a pessimistic bound on the unclipped objective (see Schulman et al.).

Now, transforming the objective into a loss: we add a minus sign and replace min with max:

L P G (ϕ)

∑ t

max (− r t (ϕ) ^ A t , − clip (r t (ϕ) , 1 − ϵ , 1 + ϵ) ^ A t) .

pgLoss clipC logratio advNorm = let ratio = exp logratio ratio' = clamp (1 - clipC) (1 + clipC) ratio

```
  pgLoss1 = -advNorm * ratio
  pgLoss2 = -advNorm * ratio'
```

in mean $ max' pgLoss1 pgLoss2 Note that for better numerical properties we typically use logratio, instead of ratio (a log of a ratio is the difference of logs):

logratio = newlogprobs - logprobs_t where newlogprobs are calculated by evaluating the new policy.

Next term L V F t is often a squared-error loss (V θ (s t) − V targ t) 2 .

vLoss = mean (newvalues - returns)^2 However, we use clipped value loss.

clippedValueLoss clipC val newval ret = let lossUnclipped = (newval - ret)^2 clipped = val + (clamp (-clipC) clipC (newval - val)) lossClipped = (clipped - ret)^2 lossMax = max' lossUnclipped lossClipped in mean lossMax Finally, the entropy bonus S is used in order to stimulate exploration, i.e. performing the same task by as many ways as possible.

entropyLoss = mean entropy Here is updated evaluate function:

-- | Get action, logProb, and entropy tensors evaluate :: Phi -- ^ Policy weights -> Tensor -- ^ Observations -> Maybe Tensor -- ^ Action -> IO (Tensor, Tensor, Tensor) evaluate ϕ obs a = do let probs = policy ϕ obs dist = Categorical.fromProbs probs action <- _getAct a dist let logProb = D.logProb dist action entropy = D.entropy dist pure (action, logProb, entropy) where _getAct :: Maybe Tensor -> Categorical.Categorical -> IO Tensor _getAct Nothing dist = D.sample dist [head $ shape obs] _getAct (Just a') _ = pure a' Putting it all together, we get this optimize function:

optimize :: Config -> Tensor -> Tensor -> Tensor -> Tensor -> Tensor -> (Agent, Trainer) -> IO (Agent, Trainer) optimize Config {..} obs_t acs_t val_t logprobs_t advantages_t returns_t (Agent {..}, Trainer {..}) = do (_, newlogprobs, entropy) <- evaluate ϕ obs_t (Just acs_t)

let newvalues = critic θ obs_t logratio = newlogprobs - logprobs_t -- Normalized advantages advNorm = (advantages_t - mean advantages_t) / (std advantages_t + 1e-8)

```
  pg = pgLoss clipC logratio advNorm

  vLoss = clippedValueLoss clipC val_t newvalues returns_t

  entropyLoss = mean entropy

  loss = pg + vfC `mulScalar` vLoss - entC `mulScalar` entropyLoss
```

((θ', ϕ'), opt') <- runStep (θ, ϕ) opt loss (asTensor lr)

pure (Agent θ' ϕ', Trainer opt') See the complete code on Github.

AlphaGo vs Lee Sedol We started this article with the defeated Go champion Lee Sedol. He played against software called AlphaGo (version Lee, after him). How actually did AlphaGo win? What is self-play and how does it relate to reinforcement learning? Below we will address those questions.

Overview AlphaGo generates data by playing games against an opponent (since the game of Go is a two-player game). This opponent is another machine player randomly chosen from a pool of opponents. In the end of the game one player wins (reward r

•

1) and another loses (reward r

− 1).

After several games have been played, the player is updated. Then, this new player is compared to the old one. If the new one wins sufficiently often, it is then accepted. Iteration by iteration, and the AlphaGo is improved by playing against itself. This is called a self-play. Below, we are going into mode details.

Monte Carlo Tree Search We should introduce a new concept called Monte Carlo Tree Search. Monte Carlo, if you didn't know, is the area in the city-state of Monaco. It is also the European gambling capital. Some rich people like to waste money in the Monte Carlo casino. The author had a chance to visit it once and can confirm that is absolutely true.

Anyway, I digress. Statisticians simply love calling everything Monte Carlo when there are random simulations involved. For example, have you noticed that REINFORCE is also named Monte Carlo Policy Gradient? This is related to stochastic trajectories generated during rollouts.

Monte Carlo Tree Search. Source: Silver *et al.* Figure 1 Monte Carlo Tree Search. Source: Silver et al.

Monte Carlo Tree Search (MCTS) explores a tree of possible moves and is continuously refining its understanding of the game state by simulating random rollouts (also called playouts because of the game aspect). In Monte Carlo tree, each node is a board state (see Fig. 1). For each board state, we estimate the value of this state Q . That is how good it is or, in other words, whether we believe this board position is leading to a win or loss. The innovation behind AlphaGo was in combining MCTS with convolutional neural networks for state value estimation and for selecting the move to play.

In the first MCTS stage called Selection (Fig. 1a), an action is selected to maximize the value Q plus some bonus u (P) that encourages exploration. This bonus is designed such that in the beginning

the algorithm prefers actions with high prior probability P and low visit count; and eventually it prefers actions with high action value Q . This is achieved by weighting the bonus term by an exploration constant.

After a certain depth of Monte Carlo tree is reached, the second stage Evaluation (Fig. 1c) is performed: current position (leaf node in tree) is evaluated using the value network. And the actions (game moves) are selected according to the policy network π until the end of the game (terminal state), which leads to the outcome ± r .

Finally, the Backup is performed (Fig. 1d): the rollout statistics are updated by adding the outcome in a backward pass through the Monte Carlo tree. In the end, the value estimate Q (s , a) becomes a weighted sum of the value network and just obtained rollout statistics. At the end of the search the algorithm selects an action with the maximum visit count. The authors state that this is less sensitive to outliers as compared to maximizing action value.

One more thing: when the number of certain node visits is frequent, the successor state is added to the tree. This is called Expansion (Fig. 1b).

Coming back to the neural networks you might be pleased to learn that the policy network π was trained using the REINFORCE algorithm you already know. Each iteration consisted of a minibatch of several games played between the current policy network π ϕ and an opponent π ϕ − using parameters from the previous iteration. Every 500 iterations, parameters ϕ were saved to the opponent pool (so that there is always a variety of opponents to choose from).

The value network was trained to approximate the value function of the RL policy network π ϕ . This was a regression task. The network was trained on a dataset of 30 million positions drawn from the self-play. There are many interesting technical aspects I suggest to read about in the original publication.

To summarize, Monte Carlo Tree Search was employed in AlphaGo because an exhaustive search is simply impossible as the game tree quickly grows too large. The idea of MCTS combined with neural networks is conceptually simple and provided sufficient computational resources, it wins.

Instead of Conclusion Beating the strongest human player is an amazing feat by the DeepMind team. However, we should not forget that behind the scenes there was operating a whole data center to support AlphaGo computation. This is about six orders of magnitude more power

consumption compared to the human brain (~20W)! Devising energy-efficient AI hardware is therefore our next milestone we are heading to.

Citation @article{penkovsky2023RL, title = "Beyond Supervised Learning", author = "Penkovsky, Bogdan", journal = "penkovsky.com", year = "2023", month = "May", url = "https://penkovsky.com/neural-networks/beyond/" } If you like the article please consider sharing it.

Goodbye? I have to admit that after all these days we have barely scratched the surface. Yet, I am happy about the journey we have made. We have seen how to create neural networks and to benefit from them; how to estimate model uncertainty; how to generate new things. And today we have learned how to continuously make decisions. There are so many more ideas to explore! Reinforcement learning is a rabbit hole in itself. By the way, did you know that ChatGPT is secretly using PPO to get better answers?

Let me know if you have any remarks.

Learn More D. Silver. Policy Gradient - Lecture Understanding the role of the discount factor Hugging Face Deep RL Course Learning dexterous in-hand manipulation Closing the Sim-to-Real Loop: Adapting Simulation Randomization with Real World Experience Noise and The Reality Gap Policy Gradient with PyTorch Proximal Policy Optimization Algorithms Implementation matters in deep policy gradients Mastering the game of Go with deep neural networks and tree search Mastering the game of Go without human knowledge This is what we were doing all the days before. Supervised learning. That is how it is called. One of my favourite teachers used to quote Molière, "Par ma foi, il y a plus de quarante ans que je dis de la prose, sans que j'en susse rien" ("These forty years now I've been speaking in prose without knowing it!"). ^ Honestly, I have no clue how a director of AI works at Tesla. But if you Andrej Karpathy and you are reading this, please share your past experiences. I would appreciate. ^ A trajectory is a repetitive sequence of observations and actions. ^ In Haskell it is possible to benefit from existing OpenAI Gym environments via Gym HTTP API. ^ See D. Silver's Lecture about Policy Gradients. ^ PyTorch seems to be better at minimizing rather than maximizing. ^

Revision #1
Created 2026-03-02 20:02:33 UTC by gasick
Updated 2026-03-02 20:04:46 UTC by gasick