Lecture 9: Reinforcement Learning

Emile van Krieken
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

REINFORCEMENT LEARNING

Reinforcement learning

- Train **agent** to act in an **environment** by maximizing **reward**

Comparison

- *Supervised* learning: Exact action given
- *Unsupervised* learning: No reward given

Deep Reinforcement Learning

- Agents use neural network policies

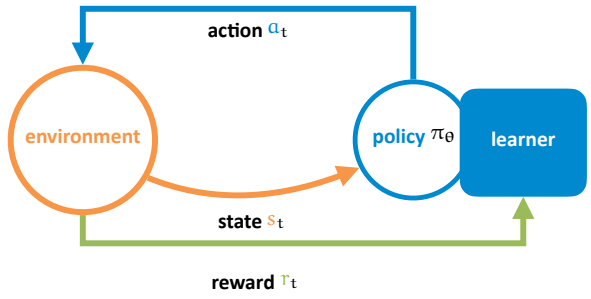
VU

In Reinforcement Learning, we try to learn a policy of an agent that acts in an environment. The agent should act 'optimally', in the sense that it should maximize rewards it receives from the environment.

This is different from supervised learning, where agents are told exactly what is the best action to perform. The agent just receives a reward from the environment, but no feedback as to what action leads to good rewards.

Furthermore, it's also not like unsupervised learning, where no explicit feedback on how to act is given. We do receive rewards!

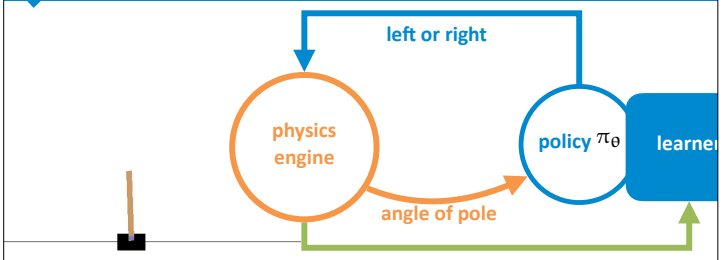
REINFORCEMENT LEARNING LOOP



VU

- This is the standard reinforcement loop that most approaches follow. We have an environment, like a game, or the real world in which a robot has to act. We assume it is black-box, which means that we have no information about what the environment looks like. The environment receives actions from the agent, which follows a policy. A policy is a probability distribution that suggests which action to take in some state. Of course, it receives the state again from the environment.
- This loop happens for every timestep t. The environment presents a state s_t to the policy \pi_\theta, which then chooses an action a_t. The environment does some magic, and chooses the next state s_{t+1}. It also returns a reward r_{t+1} for that time step, which could be received because the agent achieves some goal in the environment. This reward is then used in the learner to update the policy parameters.

CART POLE



reward: 1 if poll upright
0 otherwise

https://medium.com/@tuzzer/cart-pole-balancing-with-q-learning-b54c6068d947

VU

Here, we present an example of an RL environment: Cart pole balancing. Our agent is the funny object on the left of the screen, a cart, has to balance the wooden stick, the pole, so that it remains upright.

To encode the state, it is sufficient to pass the angle of the pole, though additional features can be thought of! Our agent uses its policy to decide whether the cart should be moved to the left or the right, so the pole remains upright.

It receives a positive reward if the pole is upright, and otherwise receives no reward. The reward is used in the learner to update the parameters.

THIS LECTURE

Simplest (Deep) RL setting

- Only neural network policies
- **Episodic**
- **Online**
- **Model-free**

Gradient estimation focus

VU

In this lecture, we will focus on the simplest of Deep Reinforcement Learning settings. We will only use neural networks to model our policies, and not look at the tabular reinforcement learning setting or different machine learning models.

We will assume our environment is **episodic.** This means that our agent acts in the environment for a finite amount of timesteps. There is some state that is the terminal state: From this point, nothing happens anymore! There are also non-episodic environments in which there is never an end to the agent acting and receiving rewards. This is however much less common in practice.

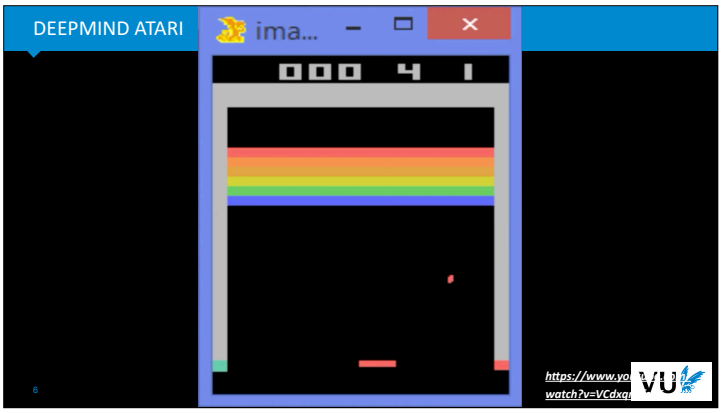
Secondly, we will be assuming an **online RL** setting: Here, we train the agent while it interacts with the environment**.** Other settings, like offline RL, exist, where instead we receive a batch of data of another agent acting in the environment, and we should find an agent that acts optimally just from inspecting this batch of data. This is a very challenging setting, and an active research area!

We will also only look at **model-free** methods for now. There are also model-based methods to RL, where in addition to learning the policy, we also learn a model of the environment! This is also an exciting and active research area.

In this lecture, we will attempt to explain Deep RL with a focus on gradient estimation. We will go into more details later what exactly this is, but we have seen an example of gradient estimation before in our course: The reparameterization in VAEs!

This lecture will be introducing RL and explaining the basic Deep RL algorithm, while the second lecture on RL, lecture 12) will focus on more recent popular methods for Deep RL.

DEEPMIND ATARI

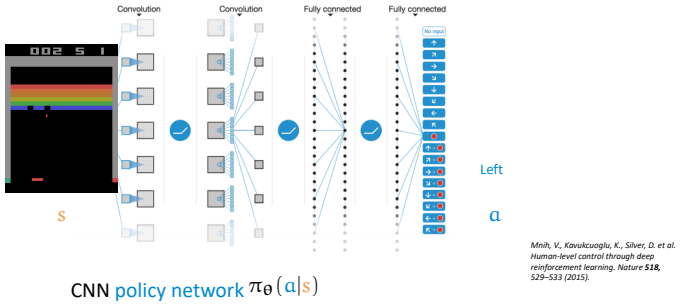


One benefit of RL is that a single system can be developed for many different tasks, so long as the interface between the world and the learner stays the same. Here is a famous experiment by DeepMind, the company behind AlphaGo. The environment is an Atari simulator. The state is a single image, containing 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

ATARI POLICY



*Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015).*

CNN policy network $\pi_\theta(a|s)$

Here, we see an example of what a Deep RL policy might look like. Like we mentioned, the states in our atari game are simple images, so, a sensible idea is to use a CNN! This CNN policy takes the current state of the game, does a lot of hard neural network computation, and computes a probability distribution over actions! Since we have a finite set of actions, we can use a softmax output layer to create a categorical distribution over actions. Then, we sample an action to perform from this distribution.

We won't discuss policy network architectures any further in this lecture. For choosing network architecture, generally the same recommendations apply as for normal deep learning: Use CNNs for states represented as images, Graph Neural Networks for states represented as graphs, RNNs or transformers for text, etc.

Figure from the original DQN paper in Nature: Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).

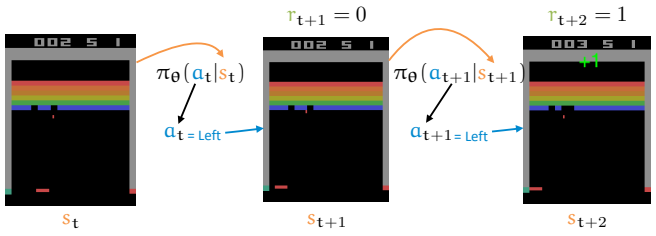
TRAJECTORIES

- **Components of RL:**
  - **Actions** a_t
  - **States** s_t
  - **Rewards** r_t
  - These are **random variables!**
- **Trajectories** $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2 \dots a_{T-1}, r_T, s_T$
- **Initial state** s_0
- **Terminal state** s_T

As we have seen before, reinforcement learning settings are modelled using actions, states and rewards. Note that RL settings are stochastic: This means these three components are **random variables.**

Furthermore, we have **trajectories**, which are a full rollout of the agent interacting with the environment and receiving rewards. It starts at the initial state s_0, and ends in the terminal state s_T, from which the agent stops interacting with the environment.

ATARI TRAJECTORY



Credit assignment:
What action causes reward?

https://becominghuman.ai/lets-build-an-atari-ai-part-0-intro-to-rl-9b2c5336e0ec

To show this setup, let's again look at the atari example. We see three states, represented using images. These go into our CNN policy, and we sample an action from them, in this case left (we see the bat go to the left). The ball hits a block on the third picture, which increases the score by one! That is a positive reward.

This brings into question the **credit assignment problem**: what was it that caused this positive reward? It probably was not bat moving left, but rather, several actions that were taken quite a few timesteps back, where the bat reflected the ball. We need to assign credit to exactly the action(s) that caused the positive reward because it allows us to 'reinforce' these actions: These were good choices!

MARKOV DECISION PROCESSES

Actions a_t, **states** s_t and **rewards** r_t

Trajectories $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2 \dots a_{T-1}, r_T, s_T$

Markov decision processes (MDPs) model $p(\tau|\theta)$

$$p(\tau|\theta) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}, r_{t+1}|s_t, a_t)$$

- **Policy** $\pi_\theta(a_t|s_t)$
- **State transition** distribution $p(s_{t+1}, r_{t+1}|s_t, a_t)$
- **Initial state** distribution $p(s_0)$

Markov decision processes model the RL loop. An MDP is defined as a distribution over trajectories: What trajectories are most likely, given that our agent acts in the environment according to the policy \pi_\theta?

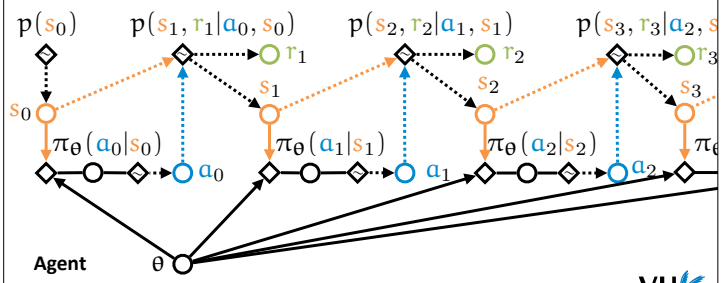
An MDP defines the probability of a trajectory, when interacting using policy \pi_theta. This probability is given with the long equation. We will explain in a lot of detail in the coming slides what exactly this means.

The probability of a trajectory is build up using three probability distributions: The first is the policy, we have seen this one before. It represents our agent, and defines a distribution over actions given states.

The second distribution is the state transition distribution, which represents the environment. This one is more complicated: It defines a distribution over what the next state of the environment is, given our current state of the environment and the action performed by the agent. It is jointly distributed with the reward, which is also dependent both on the previous state and action.

MARKOV DECISION PROCESSES

Environment



MDPs can be illustrated using a variant of computation graphs. These represent the dependencies between how the different components of the MDP are generated step by step. This is done by following the lines in the graph. Dotted lines denote non-differentiable computation. A node with the tilde (~) operation is a sampling step.

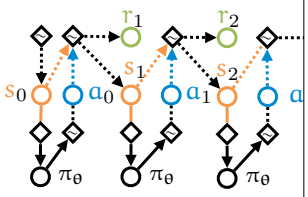
First, we generate the initial state s_0 from the initial state distribution. This is not conditioned on any input, which is represented by the fact that there are no incoming arrows. Next, we pass the generated initial state through the policy network \pi_\theta to get a probability distribution over possible actions. From this distribution, we generate the first action a_0.

Following the arrows, we pass the initial state s_0 and the first action a_0 to the environment and generate the next state s_1 and the associated reward r_1 of this state.

The process repeats from this point: Generate an action from the policy, and use it together with the state to generate the next state and reward.

OBSERVABILITY

- Full **observability** of state



- Partial observability: POMDP
  - Out of scope!

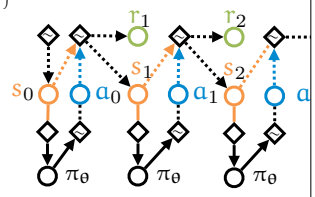
MDPs contain two important assumptions. These assumptions are used to develop efficient algorithms, but are often not a good model of the real world! The first one is **full observability** of states. We assume that we receive from the environment all there is to know about the current state of the environment. An example of an environment with full observability is chess: Both our agent and their opponent know everything there is to know about the current state of the game by just observing the game board. This is often a wrong assumption: Consider the game of poker, where our agent only knows their hand and the open cards, and not the hands of other players. Poker is an example of a **partially observable MDP (POMDP)**, where our agent can only observe a small part of the environment, or its observations are noisy. POMDPs are much more complex to work with than MDPs, but there is a lot of literature out there! For this lecture, it's out of scope, however.

MARKOV ASSUMPTION

Markov assumption: s_t independent of history given s_{t-1}:

$$p(s_t|a_{t-1}, s_1, ..., s_{t-1}) = p(s_t|a_{t-1}, s_{t-1})$$

- Used to derive strong algorithms!
- Fundamental assumption behind RL



The second key assumption, which is maybe even more important than the previous one, is the **Markov assumption**. It says that the distribution over the next state s_t is independent of the history, if we have complete information of the current state s_{t-1}. Or, in other words, we can reliably reconstruct the next state using just the current state and the chosen action. This condition is very easy to violate: For example, a static image doesn't contain the velocity of objects on the pixel! This requires multiple images, or a different feature space. Often, we can expand the features of the state, for example by taking the last few observations as part of the state, to solve such issues.

The Markov assumption is what distinguishes RL from black-box optimization algorithms, like evolutionary algorithms: These do not assume anything about the environment and thus their algorithms don't make use of this structure.

REWARDS

We introduced RL with MDPs

The goal is to maximize reward!

What does this mean in Deep RL?

So far, we have introduced the structure of RL using MDPs. In RL, we want to maximize reward. What exactly does this mean? And particularly, what does this mean in the context of Deep RL?

EXPECTED RETURN

Total reward: $R = r_1 + r_2 + \cdots + r_T$

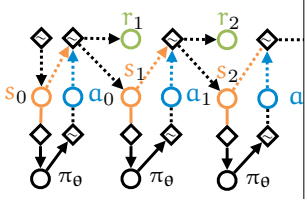
Total **discounted reward** $0 \leqslant \gamma \leqslant 1$:
$R_\gamma = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{T-1} r_T$

→Prefer rewards soon

Goal: *maximize* total discounted reward.

- Rewards and states are **stochastic!**
- So, maximize **expected return**
  $J(\theta) = \mathbb{E}_{p(\tau|\theta)}[R_\gamma]$

$\theta^* = \arg\max_\theta J(\theta)$



VU

The total reward is simply the sum of all rewards the agent receives during one trajectory. Note that a sum of random variables is also a random variable, so the total reward is a random variable!

We will often deal with **discounted** rewards. These encode that we prefer our rewards sooner than later, and is encoded using the **discount factor** \gamma. Usually, this is a number very close to 1, like 0.999. Using the discount factor, rewards decay exponentially, such that rewards very far in the future are weighted infinitely low. This might seem like an unnatural way to develop reward functions. Unfortunately, some algorithms require a discount factor smaller than 1!

To maximize rewards, we have to note that again that total (discounted) reward is a random variable: So instead, we want to maximize the **expected total (discounted) return** given a policy \pi_\theta!

We will explain how exactly this expectation works and how to compute it. To maximize it, we should find the policy for which the expected return is maximized. As we assume our policies are parameterized neural networks, we can turn this into optimizing the parameters of our policy network instead!

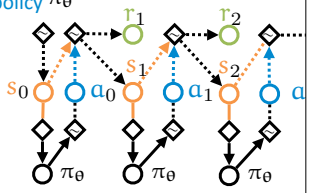
EXPECTED RETURN

$J(\theta) = \mathbb{E}_{p(\tau|\theta)}[R_\gamma]$

Expectation over **trajectories** by following policy π_θ

Requires summing over *all* trajectories!

→Monte Carlo (sampling) estimation



VU

So what exactly is this expectation over returns? It is an expectation over trajectories that result from the agent following the policy \pi_\btheta. Remember that, given a policy, the probability of a trajectory is defined using the MDP formalism. We will use this to compute our expectation!

Unfortunately, computing an expectation requires summing (or integrating) over all possible trajectories that could be sampled. There are a huge amount of possible trajectories, however! So, clearly, we cannot hope to precisely compute the expected return. Therefore, we will do what we usually do when we see an expectation: Sample! This is called Monte Carlo estimation, which sounds fancy but just means estimating an expectation using sampling.

ESTIMATE EXPECTED RETURN

Input: Parameters θ, discount γ

$s_0 \sim p(s_0)$ ⇐

$t = 0$

while s_t **is not** terminal:

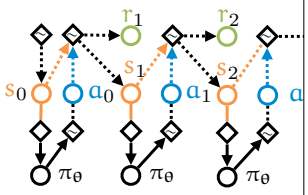
$a_t \sim \pi_\theta(a_t|s_t)$ ⇐

$s_{t+1}, r_{t+1} \sim p(s_{t+1}, r_{t+1}|s_t, a_t)$ ⇐

$t = t + 1$

return $\sum_{i=1}^{t} \gamma^{i-1} r_i$

Monte Carlo estimate of expected return
$J(\theta) = \mathbb{E}_{p(\tau|\theta)}[R_\gamma]$



VU

Here, we show the monte carlo algorithm for computing an estimation of the expected return given a policy \pi_\theta. The algorithm is simple: We generate a trajectory from the MDP, and compute the total discounted return.

The tildes used in this algorithm are an operation that denotes sampling from a distribution.

In the second line, we sample an initial state s_0. Then, we loop until we find a terminal state:

1: we use our policy network \pi_\theta to compute a distribution over actions, from which we sample our next action to take

2: use the sampled action and the previous state in our environment to sample the next state and reward.

Step 2 is a bit hard to follow: Environments are black-box functions that simulate some complicated process, like a game or a physics engine. It need not be stochastic, but it's easiest to assume it is. Sampling from the environment simply means to simulate the environment and create the next state, whether this is done deterministically or stochastically.

After encountering a terminal state, we return the monte carlo estimate: The discounted sum of rewards found in the trajectory.

Note that algorithms described in this way are also called generative stories or generative processes: They describe how the data is generated. It is an alternative and intuitive formulation for probabilistic graphical models to describe complicated joint probability distributions like MDPs! We have seen a generative story before in the VAE lecture.

MAXIMIZING EXPECTED RETURN

How to find $\theta^* = \arg\max_\theta J(\theta)$?

→**Policy gradient methods**: Use $\nabla_\theta J(\theta)$ in gradient *ascent*
$\theta_{i+1} = \theta_i + \alpha \nabla_{\theta_i} J(\theta_i)$

VU

We just introduced how we can estimate the expected return of a policy \pi_\theta. Now, we want to know how to maximize this expected return!

Maximizing an objective in Deep RL amounts to finding the optimal parameters \theta^* that maximize the expected return. We will do this using **policy gradient** methods: Estimate the gradient of expected return, then use this gradient to update the parameters using gradient *ascent*. Note that we do gradient *ascent* and not gradient *descent* like in most other Deep Learning: We want to maximize the return, not minimize it! Alternatively, we can *minimize* the negative expected return :) Like in SGD, we choose a step size (often called learning rate) \alpha with which to update the parameters.

That is all nice, but this requires that we have access to the gradient of the expected return. Unfortunately, this is not an easy quantity! This is because of two problems:

1. The environment is assumed to be black-box (unknown) and stochastic. This means that we cannot compute gradients through the environment! This is a huge downside to RL: A differentiable environment could give us a lot of information on how to cleverly update the policy parameters.
2. Sampling an action from the policy is also not differentiable, because sampling from a distribution in general is not. Luckily, there are several techniques from the **gradient estimation** literature to circumvent this problem.

RECAP PART 1

Reinforcement Learning

- Agent acts in environment to receive rewards

MDPs

- Full observability
- Markov assumption

Find parameters maximizing *expected* reward

- **Policy gradient** methods
- Estimate gradient for gradient ascent

VU

This was it for the first part of the RL lecture! We introduced the reinforcement learning framework through the MDP. The MDP is a decision process with two strong assumptions: States are fully observable, with no information hidden, and the Markov assumption: The next state is independent of the history given the current state.

We then looked at the fundamental problem of RL: Maximizing the expected (discounted) reward. We showed a simple algorithm that estimates this reward by simply sampling a trajectory and summing over the rewards. We want to find parameters for our policy network that maximize this reward. How do we do this? We use gradient ascent, and estimate the gradient of the expected reward using **policy gradient** methods. We will explain those in the next part!

MAXIMIZING EXPECTED RETURN

How to find $\theta^* = \arg\max_\theta J(\theta)$?

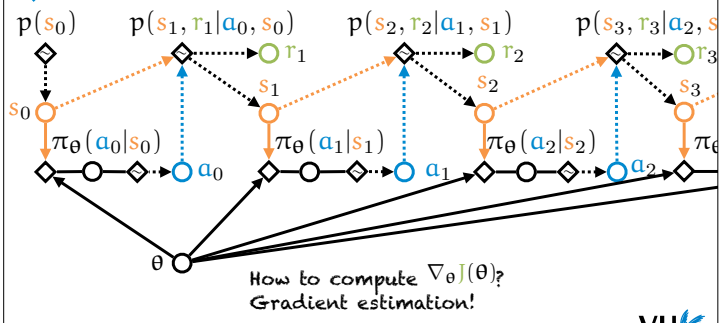
- **Policy gradient methods**: Use $\nabla_\theta J(\theta)$ in gradient *ascent*
$\theta_{i+1} = \theta_i + \alpha \nabla_{\theta_i} J(\theta_i)$

VU

We just introduced how we can estimate the expected return of a policy \pi_\theta. Now, we want to know how to maximize this expected return!

Maximizing an objective in Deep RL amounts to finding the optimal parameters \theta^* that maximize the expected return. We will do this using **policy gradient** methods: Estimate the gradient of expected return, then use this gradient to update the parameters using gradient *ascent*. Note that we do gradient *ascent* and not gradient *descent* like in most other Deep Learning: We want to maximize the return, not minimize it! Alternatively, we can *minimize* the negative expected return :) Like in SGD, we choose a step size (often called learning rate) \alpha with which to update the parameters.

MARKOV DECISION PROCESSES



VU

The problem with finding the gradient is that the computation graph of MDPs are not differentiable. What we mean with this is that there is no differentiable path from the rewards to the model parameters \theta.

SIMPLE REINFORCE

Simple **REINFORCE**:

1. $\tau \sim p(\tau|\theta)$
2. $\theta \leftarrow \theta + \alpha R_\gamma \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)$

Sample trajectory

Gradient ascent

Let's derive algorithm!

VU

We will first look at the simplest method to estimate the policy gradient: The simple REINFORCE algorithm.

The simple REINFORCE algorithm starts by sampling a trajectory from the agent interacting with the environment. This happens just like how we sampled a trajectory when estimating the expectation of the return.

Then, we estimate the gradient using the simple REINFORCE estimate. This might look a bit like magic! Where does it come from? We'll derive it in the next few slides to ensure it's not some equation thrown out of nowhere :)

This estimate gradient is finally used in the gradient ascent step.

JOINT DISTRIBUTION OF MDP

$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma]$

Expectation is over *all* trajectories τ :(

$\nabla_\theta J(\theta) = \sum_\tau R_\gamma \nabla_\theta p(\tau|\theta)$

To sample, we need an expression like

$$\sum_\tau p(\tau|\theta) f(\tau)$$

Solution: The **score function!**

VU

Let's first recap what we know. We want to estimate the gradient of the expected return. This is the derivative of an expectation! We can write this derivative out by expanding the expectation: This results in a sum over all trajectories. We can move the derivative inwards, however. Still, it is obviously impossible to compute this: The amount of possible trajectories is likely infinitely big, so we cannot sum over it.

Like usually when we see an expectation, we want to estimate it using monte carlo estimation. However, to be able to do monte carlo estimation, we need an expression in the form of a sum of the probability times some quantity. Note that the gradient is not in this form: It is the *gradient* of the probability times a quantity (R_\gamma).

How can we write the gradient in a form that allows sampling? For that, we can use the **score function**! Let's show what it is.

THE SCORE FUNCTION

$$\nabla_\theta J(\theta) = \sum_\tau R_\gamma \nabla_\theta p(\tau|\theta)$$

$$= \sum_\tau R_\gamma \nabla_\theta p(\tau|\theta) \frac{p(\tau|\theta)}{p(\tau|\theta)}$$

Multiply by 1

$$= \sum_\tau R_\gamma p(\tau|\theta) \frac{\nabla_\theta p(\tau|\theta)}{p(\tau|\theta)}$$

This is an expression like $\sum_\tau p(\tau|\theta) f(\tau)$!

VU

Again, we remind ourselves that the goal is to put the gradient of the expectation in a form so that we can sample from it. We will be using the score function trick to do this.

First, we multiply the expression by 1: p(\tau|\theta)/p(\tau|\theta). Next, we simply move the probability out of the numerator and the gradient of the probability into the numerator.

We know have an expression that we can sample from! There is a probability term, and a function (R times the fraction). This fraction is called the score function, and has a very useful representation that we will look at next.

THE SCORE FUNCTION

$$\nabla_\theta J(\theta) = \sum_\tau R_\gamma \nabla_\theta p(\tau|\theta)$$

$$= \sum_\tau R_\gamma \nabla_\theta p(\tau|\theta) \frac{p(\tau|\theta)}{p(\tau|\theta)}$$

$$= \sum_\tau R_\gamma p(\tau|\theta) \frac{\nabla_\theta p(\tau|\theta)}{p(\tau|\theta)}$$

$$= \sum_\tau p(\tau|\theta) R_\gamma \nabla_\theta \log p(\tau|\theta)$$

$$= \mathbb{E}_{p(\tau|\theta)}[R_\gamma \nabla_\theta \log p(\tau|\theta)]$$

?

Score function:

$$\nabla_\theta \log p(\tau|\theta)$$

$$= \frac{\partial \log p(\tau|\theta)}{\partial p(\tau|\theta)} \frac{\partial p(\tau|\theta)}{\partial \theta}$$

$$= \frac{1}{p(\tau|\theta)} \nabla_\theta p(\tau|\theta)$$

$$= \frac{\nabla_\theta p(\tau|\theta)}{p(\tau|\theta)}$$

VU

As it happens, we can rewrite the fraction, the score function, into the derivative of the log probability! I'm sure you've seen this quantity a lot before: Losses like the cross entropy minimize log-probabilities by taking their derivatives.

We can clearly see now that this is an expectation: A sum over a probability times a function. So let's rewrite it as such!

So how did we find that that fraction is equal to the derivative of the log-probability? It is easier to show this the other way around. Note that, by the chain rule, this derivative is equal to the 2nd line on the right. Note that the derivative of log x is 1/x, so we find on the left side 1/p(\tau|\theta). The right side is just the derivative of the probability wrt the parameters. This recovers the fraction!

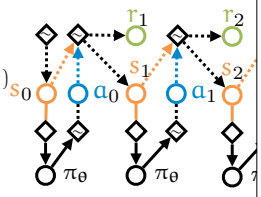
WORKING OUT MDP

$$\nabla_\theta J(\theta) = \mathbb{E}_{p(\tau|\theta)}[R_\gamma \nabla_\theta \log p(\tau|\theta)]$$

How to compute $\nabla_\theta \log p(\tau|\theta)$?

MDP distribution over trajectories:

$$p(\tau|\theta) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}, r_{t+1}|s_t, a_t)$$



VU

So we found a way to sample trajectories, and we can use the discounted reward times the derivative of the log-probability of the trajectory: This is an unbiased estimate of the derivative of the expected return.

Still, we haven't quite figured out how to compute the derivative of the log-probability over trajectories yet.

Recall that we defined the probability of a trajectory using MDPs. This was the long equation some slides back. Now with more knowledge of the structure of how the RL loop works, see if you can follow this equation: It follows exactly the generative process denoted in the graphical representation.

WORKING OUT MDP

$$p(\tau|\theta) = p(s_0)\prod_{t=0}^{T-1}\pi_\theta(a_t|s_t)p(s_{t+1}, r_{t+1}|s_t, a_t)$$

$$\log p(\tau|\theta) = \log p(s_0) + \sum_{t=0}^{T-1}\log\pi_\theta(a_t|s_t) + \log p(s_{t+1}, r_{t+1}|s_t, a_t)$$

$$\nabla_\theta \log p(\tau|\theta) = \nabla_\theta \log p(s_0) + \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t) + \nabla_\theta \log p(s_{t+1}, r_{t+1}|s_t, a_t)$$

$$\nabla_\theta \log p(\tau|\theta) = \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)$$

Gradient of environment wrt policy parameters θ is 0!

Recall how the probability of a trajectory is defined in an MDP. First, we take the logarithm of this probability: This allows us to change multiplication into summation (because \log(ab)=\log(a) + \log(b)). We thus have a big sum of different log-probabilities. The first and the last are probabilities representing the dynamics of the environment, and the middle one represents the policy.

Now remains one step: We are interested in the derivative of this log-probability! By linearity of the gradient, we simply put a gradient symbol in front of every log-probability. In the last line, we remove the terms associated with the environment. Why? The parameters of the policy only influence the policy (the agent), and not the environment! So, changing the value of the parameters of the policy changes nothing about the environment itself. This leaves us with a sum over gradients of log-policy probabilities.

BASIC REINFORCE

$$\nabla_\theta \log p(\tau|\theta) = \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)$$

Fill into $\mathbb{E}_{p(\tau|\theta)}[R_\gamma \nabla_\theta \log p(\tau|\theta)]$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{p(\tau|\theta)}[R_\gamma \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)]$$

$$\approx R_\gamma \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t), \quad \tau \sim p(\tau|\theta)$$

Monte Carlo (sample) estimate

Now that we can compute the derivative of the log-probability of the trajectory, we can fill this into the expression of the gradient of the expected return.

This gives us the expression on the third line: An expectation over trajectories, again, but now we multiply the policy probabilities over time with the discounted reward.

https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#deriving-the-simplest-policy-gradient

CREDIT ASSIGNMENT

$$\nabla_\theta J(\theta) = \mathbb{E}_{p(\tau|\theta)}[R_\gamma \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)]$$

$$R_\gamma = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{T-1} r_T$$

Reinforce actions with high *total* return

- Reinforce a_{T-1} when r_1 is high?
- Only reinforce actions with good *consequences*!

This algorithm is very simple, and also very poor! One important reason for this is that it doesn't do any **credit assignment**: Whenever we get a high *total* reward, the log-probability of **all** actions performed in that trajectory are increased evenly. Increasing the probability of performing an action because we got a good reward is called **reinforcing** that action.

We don't really want this: Consider the situation where we got a high *total* reward because the very first reward r_1 was very high. All actions taken are equally reinforced, so the action a_T taken at the last timestep T will be reinforced as much as the first action a_0, although only the first action a_0 could have influenced the value of the first reward!

CREDIT ASSIGNMENT

Recall $R_\gamma = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{T-1} r_T$

Consider gradient of reward at $t' + 1$:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[\gamma^{t'} r_{t'+1}] = \mathbb{E}_{p(\tau|\theta)}[\gamma^{t'} r_{t'+1} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

$$= \mathbb{E}_{p(\tau|\theta)}[\gamma^{t'} r_{t'+1} \sum_{t=0}^{t'} \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

Only influenced by actions until t'

VU

So can we improve upon this simple REINFORCE algorithm? Yes!

Let's consider what the gradient of the discounted reward at timestep t+1' is. So instead of taking the *total* expected reward, we only look at the expected reward at timestep t'+1. We do the same trick as before to find the first expression on the right.

It turns out this is equal to the second expression: We only need to some over the first t' action log-probabilities! The intuition behind this is that the actions taken in timesteps t'+1 to T do not influence the outcome of the reward at t'+1.

In the RL assignment, you'll prove this formally :)

https://youtu.be/oPGVsoBonLM?t=1523

BETTER REINFORCE

Sum over timesteps:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t'=0}^{T-1} \gamma^{t'} r_{t'+1} \sum_{t=0}^{t'} \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

Equivalent: Update actions based on following rewards

- **Discounted reward to go**

$$G_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'+1}$$

Gradient estimate:

$$\nabla_\theta \mathbb{E}_{p(\tau|\theta)}[R_\gamma] = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

VU

In the last step, we found that for a single reward, we only need to update the probability of actions that came before the reward.

Let's use this to find the gradient of the *total* expected reward (that is, over all timesteps)! We see that, again, for all rewards, we also should only consider the actions that came before to update. Those are the actions that could have led to a positive reward! The actions after receiving each reward couldn't possibly have influenced it.

We can rewrite this to an equivalent formulation: for the update of each action, we should consider only the rewards received *after* performing the action. We will also proof this relation in the assignment.

To do this, let's first define the **discounted reward to go**. This is the discounted reward received after timestep t, but only counting the discounting from that point. You might wonder why! It turns out this quantity represents something very useful in RL, and we will be going back to it in the next lecture on RL.

Finally, we find a nice expression for the policy gradient: Each action is reinforced based on the discounted reward to go it receives: The reward after executing that action.

https://youtu.be/oPGVsoBonLM?t=1523

REINFORCE

$$G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_{k+1}$$

REINFORCE:

1. $\tau \sim p(\tau|\theta)$
2. $\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$

Sample trajectory

Gradient ascent

VU

We will first look at the simplest method to estimate the policy gradient: The REINFORCE algorithm.

First, we define G_t, the total discounted reward received after timestep t. This quantity will turn out to be very useful in future algorithms! Note that the discounting also happens from timestep t onward: the reward r_{t+1} is not discounted, because \gamma^{k-t}=\gamma^{t-t}=1

The REINFORCE algorithm start by sampling a trajectory from the agent interacting with the environment. This happens just like how we sampled a trajectory when estimating the expectation of the return.

Then, we estimate the gradient using the REINFORCE estimate. This might look a bit like magic! Where does it come from? We'll derive it in the next few slides to ensure it's not some equation thrown out of nowhere :)

This estimate gradient is finally used in the gradient ascent step.

REINFORCE ALGORITHM FOR PYTHON

```
1  discount = 0.9999
2  sgd = SGD(NN.parameters(), lr=1e-4)
3  for tau in range(100000):
4      s = env.reset()
5      sgd.zero_grad()
6      R, log_pi_as = [], []
7      for t in range(100000):
8          pi = Categorical(logits=NN(s))
9          a = pi.sample()
10         s, r, terminal, _ = env.step(a)
11         R.append(r)
12         log_pi_as.append(pi.log_prob(a))
13         if terminal:
14             break
15     surrogate_l = 0.
16     for t in range(len(log_pi_as)):
17         Gt = 0.
18         for t' in range(t, len(R)):
19             Gt += discount**(t' - t) * R[t']
20         surrogate_l += discount**t * Gt * log_pi_as[t]
21     (-surrogate_l).backward()
33
```

Initialize

Sample a trajectory

REINFORCE update

$$\sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Update parameters

VU

Here, we see some python code for a simple REINFORCE implementation. It uses PyTorch and the OpenAI Gym library, which implements many RL environments to play around with.

In the first loop (line 3), we loop over the trajectories we sample. In line 4, we sample the initial state s from the environment.

Next, we start looping over the timesteps (line 7). We use the NN to create a categorical distribution over actions from the state s. This is the policy distribution \pi_\theta(a|s)! From this distribution, we sample the next action a to perform. We then use env.step(a) to transition to the next state s, and we receive a reward r in the process. We save the reward and the log-probability of the action. We also get back a boolean terminal which tells us if the sampled state is a terminal state. If so, we break the loop and go to the REINFORCE update.

In the next lines, we compute the REINFORCE update. For each timestep, we compute a "loss" component, which is often called the **surrogate loss**. Don't consider this loss as a meaningful quantity: We don't want to maximize the loss, we want to maximize the expected reward! The REINFORCE update is straightforwardly computed, just like the equation suggests, with a loop in the sums.

REINFORCE

Policy gradient is the gradient of expected return

REINFORCE

- Simplest method to approximate policy gradient
- General and unbiased :)
- *Very* high **variance**! :(
  - Not **sample efficient**

Next part: General **gradient estimation**

VU

Recap: We discussed the policy gradient, which is the gradient with respect to the policy parameters of the expected return.

The simplest policy gradient algorithm is REINFORCE, which is general and *unbiased*, but has *very* high variance. In the next part, we will discuss gradient estimation in general, and will also discuss what it means for an estimator to be unbiased and to have high variance.

GRADIENT ESTIMATION

Policy gradient methods:

Estimate gradient of *expected* return

Gradient estimation:

Estimate gradient of *any* expectation

$$\arg\max_\theta \mathbb{E}_{p_\theta(z)}[f(z)]$$

VU

In this final part of the lecture, we will be discussing gradient estimation in general. As we have seen in the previous part, policy gradient methods estimate the gradient of the *expected* return in an MDP. Gradient estimation as a field generalizes this: We want to estimate the gradient of *any* expectation, and not just over MDPs. This allows us to optimize problems of the form of the formula.

GRADIENT ESTIMATION

Derivative of expectation:

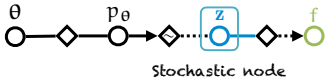
$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f(z)]$$

Continuous distributions:

$$\nabla_\theta \int p_\theta(z) f(z) dz$$

Discrete distributions:

$$\nabla_\theta \sum_z p_\theta(z) f(z)$$



VU

We want to find the derivative of an expectation. This is represented using the graph on the right: We use the parameter \theta to form a distribution p_\theta. Then, we sample z from this distribution. This is represented by the tilde (~) operation. Finally, z is used in the function f to optimize. The problem? There is no differentiable path from the function f to the parameters theta, so we cannot directly optimize this. This is because sampling isn't differentiable, and because the function f might also not be!

The derivative of an expectation can mean different thing, depending on whether the distribution p_\btheta is a continuous or a discrete distribution. For continuous distributions, we compute an integral over all options of z, while for discrete distributions, it is a sum over all options. Both are normalized by the probability density.

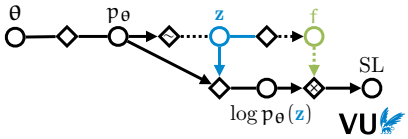
SCORE FUNCTION

Recall **score function**:

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f(z)] = \mathbb{E}_{p_\theta(z)}[f(z)\frac{\nabla_\theta p_\theta(z)}{p_\theta(z)}] = \mathbb{E}_{p_\theta(z)}[f(z)\nabla_\theta \log p_\theta(z)]$$

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

- *All* distributions $p_\theta(z)$
- *All* functions f
- But very high **variance**



VU

Recall the score function. We used it when deriving the REINFORCE estimator: The derivative of an expectation turns out to be equal to the expected value of the function times the gradient of the log-probability of the sample z. This is represented by the graph: The distribution is used to create the log-probability, which is multiplied with the function output to create the **surrogate loss**, which is named this way because it is like a fake loss.

The score function is very general: It can be used for any distribution and any function f! However, it has very, very high **variance**. So what exactly is variance?

VARIANCE

Score function estimate:

$$g_{SF} = f(z)\nabla_\theta \log p_\theta(z), \quad z \sim p_\theta(z)$$

Variance:

$$\mathbb{V}[g_{SF}] = \mathbb{E}_{p_\theta}\left[\sum_{i=1}^{D}(g_{SFi} - \mathbb{E}_{p_\theta}[g_{SFi}])^2\right]$$

High variance

→More samples needed

→Unstable training

Deep RL lecture: Variance reduction for policy gradients!

VU

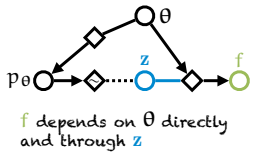
Variance is the average squared distance from mean of an estimator., per dimension. In this case, it is the score function estimator. The D represents the dimensionality of the gradient, which is equal to the dimensionality of the parameters. Note that variance increases with dimensionality: With more dimensions, the variance usually increases.

High variance is pretty bad! It usually means that we will need **way** more samples to get a good estimate of the expected gradient. This also destabilizes the training loop: If the gradient has very high variance, the training algorithm has to deal with a lot of noise, and the training can jump everywhere!

This is in particular problematic in deep learning applications, because the dimensionality of the parameters is so large. And, as we mentioned, the variance scales with dimensionality…

In the next lecture on deep reinforcement learning, we will be looking into variance reduction techniques for policy gradient methods. This is essential to get them working on practice.

MULTIPLE DEPENDENCIES

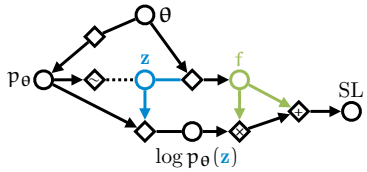


f depends on θ directly and through z

Now, consider gradient estimation on the following graph. Unlike before, the parameter \theta isn't just used to compute the probability p_\theta, but also used in the computation of the function f, that is, f(z, \theta). Furthermore, the function f is a differentiable function now!

The previous gradient will not do here: It'd not take into account how the parameter \theta influences f directly.

MULTIPLE DEPENDENCIES



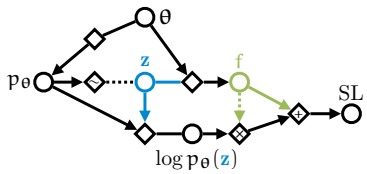
$$\mathrm{SL} = f(z)\log p_\theta(z) + f(z)$$

$$\nabla_\theta \mathrm{SL} = f(z)\nabla_\theta \log p_\theta(z) + \boxed{\nabla_\theta f(z)\log p_\theta(z)} + \nabla_\theta f(z) \not\approx \nabla_\theta \mathbb{E}_{p_\theta(z)}[f(z)]$$

The corresponding surrogate loss for this graph uses both the log probability and the function itself. Walk through the non-broken lines backwards from SL to see what paths to \theta exist. Does that work?

Looking at the derivative, no it doesn't! There is an extra term in the middle which we didn't expect. It comes from using the product rule on the function times log-probability term. So, this surrogate loss is unfortunately wrong.

MULTIPLE DEPENDENCIES



$$\mathrm{SL} = \bot(f(z))\log p_\theta(z) + f(z)$$

$$\nabla_\theta \mathrm{SL} = f(z)\nabla_\theta \log p_\theta(z) + \nabla_\theta f(z) \approx \nabla_\theta \mathbb{E}_{p_\theta(z)}[f(z)]$$

The correct surrogate loss uses the detach function. We have seen this before in lecture 2. This is an operation that blocks gradients from flowing backwards, but leaves forward computation intact.

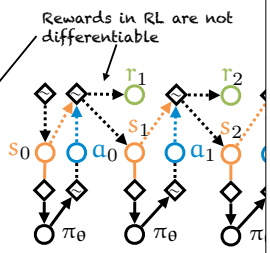
Taking the derivative of the surrogate loss, we see that it uses both the score function and a direct derivative of the function with respect to the parameter \theta.

GENERAL SURROGATE LOSS

$$G_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'+1}$$

$$SL = \mathbb{E}_{p(\tau|\theta)}[\sum_{t=0}^{T-1} \gamma^t G_t \log \pi_\theta(a_t|s_t)]$$

$$SL = \mathbb{E}_Z \left[\sum_{z \in Z} \log p(z) \sum_{z \prec r} \perp(r) + \sum_{r \in R} r \right]$$



Schulman, John, et al. "Gradient estimation using stochastic computation graphs." Advances in Neural Information Processing Systems. 2015.

VU

Can we generalize this? Yes! Schulman et al, 2015 introduces an algorithm for gradient estimation on any computation graph with stochastic nodes. Say we have a set of stochastic nodes Z and a set of rewards R. We will also use the notation z < r, which means that the reward r is influenced by the node z. A node is influenced by another if there is a directed path between them through the computation graph.

The correct surrogate loss then sums over each stochastic node to get their log-probability. This is multiplied by the sum of rewards that the stochastic node influences. Finally, we also have a sum over all rewards to ensure that parameters that directly influence the rewards are also taken into account for computing the gradient.

Compare this with the REINFORCE surrogate loss. We note that reward functions in RL are not differentiable, so we won't need the second sum over rewards. Then, the automatic surrogate loss corresponds perfectly with the REINFORCE estimator: It sums over actions to compute their log-probability, and this is multiplied by the sum of rewards that the action influences!

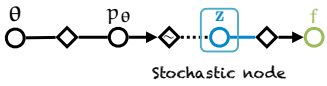
CAN WE DO BETTER?

Score function has high variance…

Can we do better?

Lecture 6 (VAEs):

- VAEs also use gradient estimation!
- "Reparameterization trick"



VU

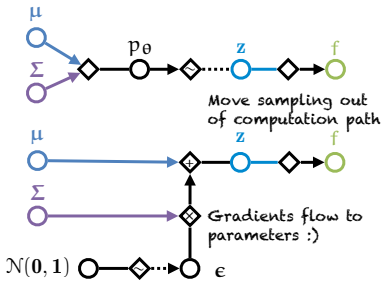
Can we do better than the score function, which has very high variance? Yes, but not always. Recall lecture 6, in which Jakub discussed VAEs. He also talked about reparameterization. The reparameterization is actually also a gradient estimator! Let's dive in.

GAUSSIAN REPARAMETERIZATION

Gaussian $\mathcal{N}(\mu, \Sigma)$

$\epsilon \sim \mathcal{N}(0, 1)$

$z = \mu + \epsilon\Sigma$

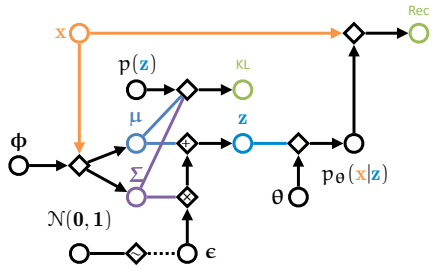


VU

Assume we have a Gaussian distribution with mean parameter \mu and covariance matrix \Sigma (often diagonal).

The idea behind reparameterization is to move the sampling step out of the computation graph, so that gradients can flow directly through the graph to the parameters. This is very easy to do for the Gaussian distribution: We simply sample noise epsilon from a normal gaussian (with mean 0 and identity covariance matrix), and transform this noise by multiplying it with the covariance matrix \Sigma and adding the mean \mu. We can now do a normal backwards pass from f directly to the parameters!

EXAMPLE: VAE ELBO



$$\text{ELBO} = \log p_\theta(x \mid z) - D_{KL}[q_\phi(z|x)||p(z)]$$

Rec KL

VU

For Gaussian posteriors, we can use reparameterization to train this model well! Using the same trick as before, we sample noise \epsilon, multiply it with the covariance matrix of the approximate posterior q(z|x) and then add the result to the mean of the approximate posterior. This produces the sample z in a differentiable way!

PATHWISE DERIVATIVE

Reparameterization:

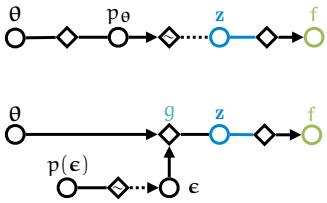
$$\mathbb{E}_{p_\theta(z)}[f(z)] = \mathbb{E}_{p(\epsilon)}[f(g(\theta, \epsilon))]$$

- Noise distribution $p(\epsilon)$

$z = g(\theta, \epsilon) \sim p_\theta(z)$

Pathwise derivative (=backprop):

$$g_{PD} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial \theta}, \quad \epsilon \sim p(\epsilon)$$

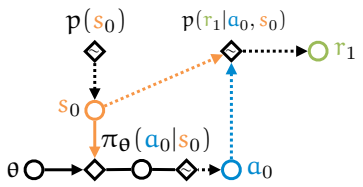


VU

In general, the reparameterization trick works as follows: Instead of sampling directly from p_\theta, we sample from some noise distribution p(\epsilon) and transform it using a function g. Importantly, this function should have the property that, after transforming the noise \epsilon, its results have the same distribution as the original distribution p_\theta. So, if z=g(\theta, \epsilon) is the result of the transformation, and \epsilon ~ p(\epsilon), then z ~ p_\theta(z).

After a reparameterization, we can use the pathwise derivative. This is a simple application of the chain rule: First compute the derivative of z with respect to the function f, then of z with respect to the parameters \theta through the function g. Although it sounds fancy, the pathwise derivative, this happens automatically when using backpropagation!

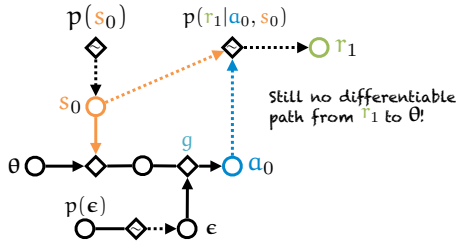
REPARAMETERIZATION IN RL?



VU

Reparameterization sounds pretty great! So why didn't we just use it for Reinforcement Learning?

REPARAMETERIZATION IN RL?



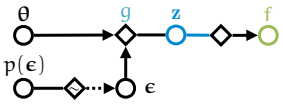
VU

Unfortunately, reparamterization doesn't work in Reinforcement Learning. Although we can transform the action sampling step to make it differentiable, this doesn't solve the non-differentiability of the environment! As you can see, there is still no differentiable path from the parameters to the reward.

PATHWISE DERIVATIVE

Pathwise derivative has low variance :)

- Uses extra info: $\frac{\partial f}{\partial z}$



Requires:

- Differentiable function f :(
- Appropriate *continuous* distribution $p_\theta(z)$:(

No reparameterization for *discrete* distributions…

But **continuous relaxations** exist!

VU

The pathwise derivative using reparameterization has low variance. This is because it uses extra information compared to the score function: It has access to the derivative of the function f, which gives much more information on how to properly change the sampled value z to improve the result. This means that, when using the pathwise derivative, we will have much more stable and quick training.

However, reparameterization requires a differentiable function f. This unfortunately means we cannot use it in applications like Reinforcement Learning. It also requires a continuous distribution which is 'reparameterizable', that is, there is a transformation g on noise epsilon so that the result has the same distribution as p_\theta. This is particularly annoying because it means we cannot use the pathwise derivative for discrete distributions!

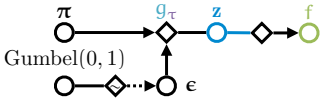
GUMBEL SOFTMAX

Gumbel Softmax: *Continuous relaxation* for categorical distribution

Probabilities $\pi_1, \ldots, \pi_K$, temperature $\tau > 0$

$\epsilon_1, \ldots, \epsilon_K \sim \text{Gumbel}(0, 1)$

$z = g_\tau(\epsilon, \pi) = \text{Softmax}((\log \pi + \epsilon)/\tau)$



VU

Although reparameterization doesn't exist for discrete distributions, we can use so-called continuous relaxations. The **Gumbel Softmax** is a method that allows us to use a pathwise derivative for a continuous distribution that is almost like a categorical distribution.

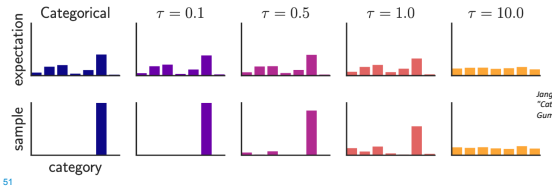
We first sample noise from the Gumbel distribution. Then, we transform this noise. First, we add the log-probability for the i-th class. We divide this by the **temperature** parameter, which should be larger than 0. The result is transformed by the softmax! What does this look like?

GUMBEL SOFTMAX

Probabilities $\pi_1, ..., \pi_K$, temperature $\tau > 0$

$\epsilon_1, ..., \epsilon_K \sim \text{Gumbel}(0, 1)$

$\mathbf{z} = g_\tau(\boldsymbol{\epsilon}, \boldsymbol{\pi}) = \text{Softmax}((\log \boldsymbol{\pi} + \boldsymbol{\epsilon})/\tau)$



Jang, Eric, Shixiang Gu, and Ben Poole. "Categorical Reparameterization with Gumbel-Softmax." (2016).

VU

In this image, you'll see what the gumbel softmax distribution looks like. Samples (bottom row) look a lot like samples from a categorical distribution, for lower temperatures. For higher temperatures, it looks more like a uniform distribution over classes. However, note that it is not a distribution: It is a sample! So where the categorical distribution samples 'one-hot' vectors, the gumbel-softmax samples "kinda one-hot vectors, but not really".

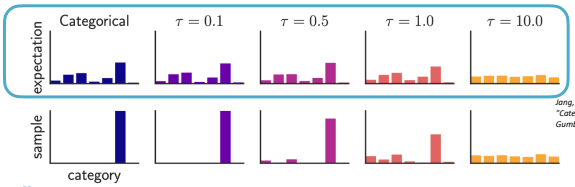
This is similar for the expectation of the distribution. For low temperatures, the expectation looks very similar to the expectation of the categorical distribution. For higher temperatures, the expectation again looks rather more like a uniform distribution over classes.

BIASED ESTIMATORS

If the temperature τ increases

- Variance decreases :)
- **Bias** increases :(

$\text{Bias} = \mathbb{E}_{\mathbf{p}(\boldsymbol{\epsilon})}[\nabla_{\boldsymbol{\pi}} f(g_\tau(\boldsymbol{\epsilon}, \boldsymbol{\pi}))] - \nabla_{\boldsymbol{\pi}} \mathbb{E}_{\mathbf{p}_{\boldsymbol{\pi}}(\mathbf{z})}[f(\mathbf{z})]$



Jang, Eric, Shixiang Gu, and Ben Poole. "Categorical Reparameterization with Gumbel-Softmax." (2016).

VU

So clearly, there's something with this temperature parameter. When it increases, the variance decreases, so that's good! This is quite easy to see: Samples will look more like very similar uniform distributions.

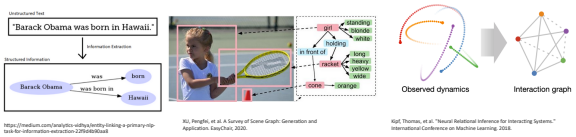
However, when the temperature increases, the **bias** also increases! What is the bias? The bias is the difference between the true gradient and the *expected* gradient using the gumbel softmax. So far, we have only looked at methods that are **unbiased**, so that the true gradient and expected gradient are equal. However, that's not the case for the gumbel softmax: In essence, you could say that its approximation is wrong, and gets even more wrong when the temperature increases.

Does that mean that we cannot use it? Actually, the gumbel-softmax works remarkably well. It requires a continuous, differentiable function, but it can optimize to good performance. This is because, although it is biased, it becomes unbiased when the temperature approaches zero. When balancing bias and variance properly through controlling the temperature (called bias-variance trade-off), we can do well!

DISCRETE DISTRIBUTIONS

Discrete distributions: Make *crisp choices*

- Example: Graphs from text or images
- Transform neural representations to **symbolic representation**



https://medium.com/analytics-vidhya/entity-linking-a-primary-nlp-task-for-information-extraction-229fd4b90aa8

XU, Pengfei, et al. A Survey of Scene Graph: Generation and Application. EasyChair, 2020.

Kipf, Thomas, et al. "Neural Relational Inference for Interacting Systems." International Conference on Machine Learning. 2018.

VU

So why do we even care so much about discrete distributions? Discrete distributions allow us to make crisp choices within our neural network: We can, for example, choose to only do one computation path instead of all of them. Or we can have intermediate **symbolic representations** of what the neural network is thinking of. For instance, we can transform text into a graph using a neural network, and then reason on this graph using a GCN to answer questions about the graph. The same thing goes for images: Transform them first into a graph representation and then answer questions!

Intermediate symbolic representations also are much more interpretable than the very high-dimensional continuous representations of neural networks. If we can transform these representations between each other, this could help us understand the neural network.

NEURO-SYMBOLIC AI

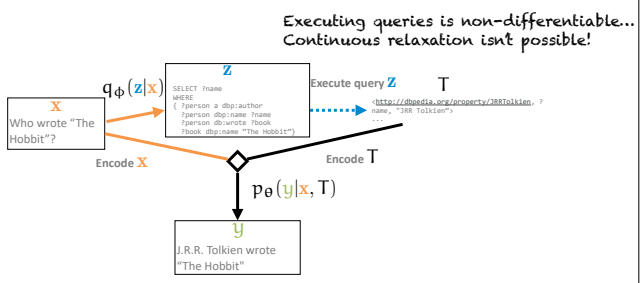
Neuro-Symbolic AI: Combine Deep Learning and Logic

- Deep Learning: Continuous
- Logic: Discrete
- Gradient estimation allows **integrating** these fields!

The field that tries to combine symbolic reasoning with Deep Learning is called **neuro-symbolic AI**. A fundamental challenge of this field is that Deep Learning does continuous reasoning, while symbolic reasoning, like logic, reasons about discrete data. This mismatch makes it hard to integrate the two fields.

However, gradient estimation turns out to be a general and principled way to integrate the two fields!

QUESTION ANSWERING WITH QUERIES



As an example, we present a neuro-symbolic approach to question-answering. Assume we have a dataset of question-answer pairs written in natural language. Given a question x, we encode it to a distribution over queries q_\phi(z|x). From this distribution, we sample a query that represents the question.

Next, we execute the query on the knowledge base to find the extracted data. Note that executing this query is not a differentiable operation!

Finally, we answer the question both by using the question and the retrieved data.

This model is very hard to train. This is because of the non-differentiability of executing the query on the dataset. However, we need to find the parameters \phi so we can learn how to extract queries from an answer. We can use gradient estimation here, although this is hard: We cannot use continuous relaxations either because executing queries is non-differentiable.

STORCHASTIC

Storchastic: Define computation graph with sampling steps.
Compute gradient estimators *automatically*!

- PyTorch library with easy API
- Many low-variance estimators implemented
- Focus on discrete distributions

https://github.com/HEmile/storchastic

We have been working on a PyTorch library for gradient estimation. With this library, you can define a computation graph with sampling steps (~), and it computes correct gradient estimators automatically. This allows people to develop stochastic deep learning models with a simple API. We have implemented state of the art gradient estimators with low variance, that can be plugged and played with. In particular, we have focused on gradient estimators for discrete distributions to allow people to develop neuro-symbolic models as discussed on the previous slide.

THESIS PROJECTS

Master thesis projects with **Storchastic**

- Designing new gradient estimators
- Knowledge Graph walking
- Question answering with querying
- Model-based RL with discrete latent space
- Logical reasoning as a stochastic hidden layer
- Neural network pruning
- … (*your idea here*)

Contact e.van.krieken@vu.nl if interested

https://github.com/HEmile/storchastic

VU

RECAP

Score function: general, but high variance

Reparameterization: move sampling out of computation graph

- Uses **pathwise derivative** with low variance
- Requires appropriate continuous distribution
- Requires differentiable functions

Gumbel softmax: Continuous approximation for discrete distributions

But biased!

Next lecture: Variance reduction for policy gradients

VU

THANK YOU!

e.van.krieken@vu.nl

https://github.com/HEmile/storchastic

https://github.com/HEmile/storchastic

VU