CAPS Notes - Lecture 4
Model Deployment Evaluation

Leo Klenner, Henry Fung, Cory Combs

Last updated: 11/18/2019

Overview

In this session, we have two broad goals:

- understand the importance and challenges of **auditing an algorithm** once it has been deployed into the real world,
- grasp foundational aspects of **reinforcement learning**, a machine learning technique concerned with enabling agents to select optimal actions in an uncertain environment. Reinforcement learning often requires strong model deployment evaluation to mitigate against specific performance trade-offs.

We will start with a case that connects both of these themes:

- an investigation of the Wall Street Journal into Google's search algorithms, specifically into how Google handles auto-complete suggestions.

How does this case combine audits and reinforcement learning?

Audits:

- In its investigation, the Journal wants to test if Google interferes with how its algorithms handle auto-completes for sensitive topics; specifically, the Journal wants to identify if the algorithms contain human-created blacklists of certain auto-complete suggestions (like "is dumb") for certain input search terms (like "").
- The Journal wants to audit Google's algorithms to find evidence of human interference with how the algorithms handle auto-completes and present search results to users.

Reinforcement learning:

- What technique makes auto-complete suggestions possible? Auto-complete suggestions are enabled through a version of **Markov chains**. Markov chains are sequence models, meaning they constitute a probabilistic model of a sequence of transitions from one state (like "") to another state (like "is dumb").
- A concept closely related to Markov chains, **Markov processes**, constitutes the foundation of reinforcement learning. We will introduce the basics of reinforcement learning and then look at a simple reinforcement learning framework called **multi-armed bandits** that powers recommender systems. Recommender systems perform a function very similar to auto-complete suggestions.

There's another theme present in the case that we want to parse out, the design of interactions between humans and algorithms to create controlled decision-making processes. We refer to this design as **human-in-the-loop learning**, which has become a prominent framework for controlling reinforcement learners.

Human-in-the-loop learning:

- According to the Wall Street Journal, Google uses humans in two ways to control its search algorithms:
    - Google employees can blacklist certain auto-complete suggestions
    - Google contractors review the rankings of search results and re-rank them based on a set of factors; the aggregated re-ranked results are then used to inform Google's search algorithms
- Interventions into the autonomous learning of algorithms are the idea behind human-in-the-loop learning. Providing algorithms with expert advice in the form of a human that can interfere with the (learning) loop of the algorithm brings with it benefits (robustness, faster learning, etc.) but also bears the risk of adding bias. We will conclude today by discussing human-in-the loop reinforcement learning.

1 Auditing Algorithms

Algorithms make decisions that impact humans, from search algorithms to navigation to other domains. Understanding how these algorithms work and ensuring their **alignment** with human values is critical. Those tasks are the concern of "algorithm auditing", a fast emerging field.

A recent article on the topic of bias in algorithms, featured a quote from the CEO of Primer, an AI company:

> Primer's Chief executive, Sean Gourley, said vetting the behavior of this new technology would become so important, it will spawn a whole new industry, where companies pay specialists to audit their algorithms for all kinds of bias and other unexpected behavior.
>
> "This is probably a billion-dollar industry," he said.

What are the core challenges that these specialists who audit algorithms may face? Three challenges come to mind:

- lack of interpretability of algorithms
- lack of access to the algorithms' **code, or "inside"**

We can summarize these challenges in the following table that states the levels of difficulty of an audit:

	Interpretable	**Not interpretable**
Access to the code	Easy	Difficult
No access to the code	Difficult	Hard

These challenges can be divided between cooperative and non-cooperative environments. Auditing in a cooperative environment, where the developer of the algorithm grants access to the code, is *relatively* straightforward compared to audits in non-cooperative environments.

Auditing in Non-Cooperative Environments

We can easily think of scenarios where there is interest from the public or another stakeholder to audit algorithms developed by an entity (company, organization, or state) that has no interest in revealing the code of the algorithms (ie. for reasons of competitiveness).

This in fact often the case, for instance the Wall Street Journal did not receive access to the code of Google's search algorithms to perform its audit. Therefore, a core problem for auditing algorithms is working with the algorithm's **observed behavior, or "outside"**.

Observed Behavior

Working with the "outside" of an algorithm presents the following challenges:

- methods of inquiry into the algorithm are inherently fuzzy,
- as a consequence, results of the audit will not be definitive but imperfect,
- this means judgements about the algorithm have to be made based on imperfect information

What are the fuzzy methods of inquiry available to us? To audit algorithms in non-cooperative environments, we are constrained to **black-box testing**.

Black-Box Testing

Black-box testing is comprised of one core step:

- sending inputs to the algorithm and analyzing the corresponding outputs

```
# example of simple black-box test
# we want to audit the algorithm MathOp that takes two numbers (x, y) as input
and
# performs an unknown mathematical operation on them

MathOp(3, 3)
6  # operation could be x + y or x + 3, or multiple other alternatives
MathOp(3, 2)
5  # operation seems like x + y
MathOp(1, 0)
Error  # unclear what the source of the error is, needs further investigation
```

This means we're simulating how users would interact with the algorithm. Although this seems simple, black-box testing can provide valuable insights into what an algorithm does and how it might work.

Often, once we have collected a set of input-output pairs from the algorithm that we want to audit, we can compare those to other algorithms and, through analyzing the differences between input-output pairs, further narrow down how the algorithm works.

One of the challenges with black-box testing is **how we interpret the results of the test**. This challenge is what we'll discuss next in the case of the Wall Street Journal's investigation into Google's search algorithms.

2 Case: Auditing Google's Search Algorithms

The case is based on the Wall Street Journal's investigation into Google's search algorithms, "How Google Interferes with its Search Algorithms and Changes Your Results".

You can find the case here.

3 Auto-Completion and Recommendation

Overview

One of the functionalities that is at the core of the Journal's investigation into Google is auto-completion.

How do auto-completion algorithms work?

- A user provides the beginning of a search query and the auto-complete algorithm provides the user with a number of suggested alternatives for completing the query

Before we think about solving this problem, let's consider a second, related problem, recommendations. We receive recommendations everywhere online (movies, items, etc.)

How do recommender algorithms work?

- A user makes a choice among alternatives (movies, items, etc.) and, based on features of the choice, the recommender algorithm generates new alternatives for the user

Formalization

To gain a better understanding of the problems and think about how we can solve them, we need to give the problems a formal structure.

In our case, both cases are similar and can be formalized with the same structure. What we'll do though is formalize each problem differently. We'll *frame*:

- **auto-completion** as a problem of **sequential probabilistic state transitions**
- **recommendation** as a problem of **optimal action selection under uncertainty**

Note, that each problem could be formalized either as sequential probabilistic state transitions or optimal action selection under uncertainty.

We'll start with auto-completion and then turn to recommendation.

Sequential Probabilistic State Transitions

Here, we formalize auto-completion. Consider the following sentences as our training data:

- "I like policy"
- "I like strategy"
- "I love computation"

Our task is to perform auto-completion based on this data. How can we formalize the problem?

We can formalize the problem as sequential probabilistic state transitions, for example:

- given the state "I", what is the probability that the next word will be "like"?

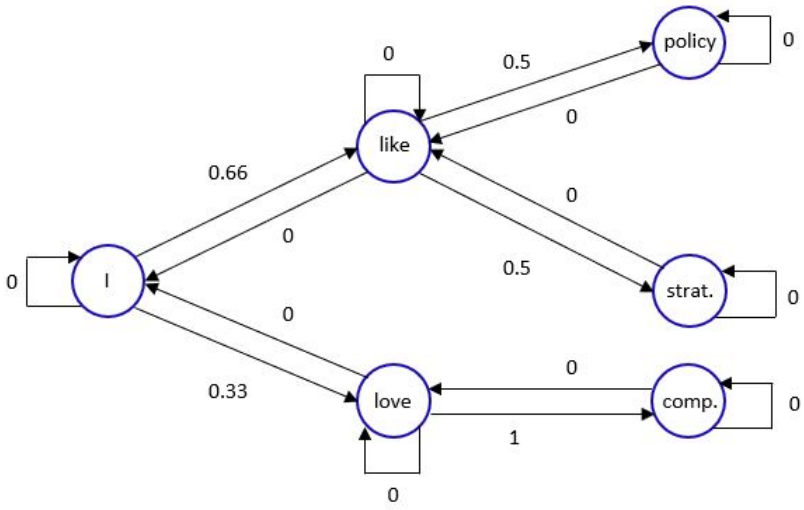


Figure 1. State tranisition diagram for our sample data

This graph of conditional probabilities is a sparse form of a **Markov chain**, which is a sequential model of probabilistic state transitions. There are three states in this chain: state 1 is "I", state 2 is either "like" or "love", state 3 is either "policy", "strategy", or "computation".

In this model, the **Markov property** holds:

- the probability of being in a next state depends only on the previous state, not on the states before that.

The Markov property might not seem intuitive but if you consider how we calculate conditional probabilities, ie. P(state 3 | state 2), it makes sense to say assume that the probability of being in a next state depends only on the previous state and not the ones before that.

Putting this together as auto-completion, if we had recommend only one word after the user has typed "I", we would recommend "like" its probability of occurring after "I" in our training data is higher than for "love".

Concepts to remember: **state**, **state transition**, **state transition probabilities**, **Markov property**

Optimal Action Selection Under Uncertainty

Here, we formalize recommendation. Let's review the example of movie recommendations as optimal action selection under uncertainty:

- **Actions:** We have a set of actions, the movies we can recommend to the user ("we" refers to the algorithm, or **agent**, that performs the recommendations in an **environment**, say the user's Netflix account).
- **Goal:** Our goal is to recommend movies that the user will watch.
- **Reward:** We keep track of our goal by issuing a reward of 1 if a movie that we recommended is watched and a reward of 0 if the movie is not watched; maximizing rewards means reaching our goal.
- **Uncertainty:** For each movie we have, we don't know whether the user will watch the movie or how often the user will watch it; in other words, we don't know the *true reward* associated with each action.
- **Estimation:** To maximize rewards, we need to estimate the *true reward* of each action; we will then select the actions, or recommend the movies to the user, that have the highest estimated reward.

This optimal action selection under uncertainty is called a **multi-armed bandit**. If you've ever used a one-armed bandit slot machine in a casino, think of this as a slot machine with multiple levers.

Each of these levers has an unknown true reward and each time we pull a lever, we receive a reward that is sampled from an underlying statistical distribution with a mean and variance (again, what the distribution looks like is unknown to us). Each lever has its own distribution from which our rewards are sampled.

Concepts to remember: **agent**, **environment**,**action**, **goal**, **reward**, **true reward**, **estimated reward**

A Shared Problem

For both sequential probabilistic state transitions and optimal action selection under uncertainty, we have a similar problem:

- how can we **estimate the probability** of transitioning into a specific state given a state, or receiving a specific reward given an action?

How can we arrive at these estimates? Let's continue the movie recommendation example and test our knowledge of supervised learning.

Solution

Supervised Approach

Continuing movie recommendation example, consider the following question:

- Can estimating the probability that a given movie will be watched by the user be solved as a problem of supervised learning?

To answer this, let's recap our knowledge of supervised learning:

- Is this a problem of classification or regression?
    - Classification because we are interested in the probability of a categorical outcome, whether or not a movie will be watched.
- Do we have training data?
    - We can use the movies that the user has watched in the past as our training data.
- Do we know what features we're looking for?
    - Features of interest in the data could be "genre", director", "lead actress", etc.

So, based on this quick check we can solve movie recommendation as a supervised learning problem.

This is indeed true, but for a number of reasons a supervised approach won't lead to the best solution.

Limits of a Supervised Approach

Taking a supervised approach to the movie recommendation problem faces several limitations:

- **Batch data:** We cannot deal with new users who do not have a history of watched movies as training a classifier requires a substantial batch of training data.
- **Offline learning:** We cannot adapt fast to shifting user tastes; classifiers are often trained offline so updated preferences of the user might not be taken into account.
- **Path dependency:** We cannot deliver content to the user that is an "unknown unknown" for the user but might be the user next favorite movie or genre, etc. because our classifiers

positively reinforce previous choices of the user, which might not have been perfectly informed.

Towards an Alternative Approach

Let's return to the concept of a multi-armed bandit. Wouldn't it be feasible to estimate the *true reward* of each lever simply through a **trial-and-error** experiment in which we pull each lever multiple times over time and thus build up an estimate of the reward we received for each lever? If we continue this experiment for a long-enough time, we should be able to obtain solid estimates and narrow down the best levers.

Taking this approach of **learning from interaction with an environment** should also solve the limitations of a supervised approach:

- **Starting from zero:** To start our experiment, we do not need training data, as we build up our training dataset over time by showing the user movies (pulling levers) and keeping track of which movies were watched (estimating the true reward of each lever).
- **Online learning:** As the user changes her preferences, this is immediately reflected in what movies she decides to watch, which in turn immediately changes our reward estimates and we can in turn start to pull the levers that return the highest currently known reward.
- **Exploration:** Instead of always pulling the lever with the highest estimated reward (reinforcing the users revealed preferences), we can occasionally pull a lever that currently has a lower estimated reward but a weak estimate and which might over time, as the estimate improves, also return a reward that is even higher than the current highest estimated reward.

This approach is called **reinforcement learning**. In situations where you only care about starting from scratch, adaptability and innovation, reinforcement learning is your go to framework.

4 Reinforcement Learning

Reinforcement learning is best described in the words of two founders of the field, Richard Sutton and Andrew Barto:

> Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics —trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

Trial-and-error search over delayed rewards is an incredibly powerful framework for solving problems; especially when no optimal solution to the problem is known.

Prominent applications of reinforcement learning include:

- having a toy helicopter perform sophisticated stunts
- beating world-champions in games like Chess, Go, and StarCraft
- any task that requires sophisticated navigation or strategy creation

To apply reinforcement learning algorithms to a problem, we need to first formalize the problem as a **Markov decision process**.

Markov Decision Process

A Markov decision process is a model of decision-making under uncertainty, where the outcomes of decisions are partly determined by the agent and partly random.

Drawing on the formalization we have established so far, you can think of a Markov Decision Process as the combination of Markov chains, in which we care about estimating the probability of a transition from one state to the next, and multi-armed bandits, in which we care about estimating the unknown rewards associated with a set of actions.

In a Markov decision process:

- an **agent** interacts with an **environment**, which can be exhaustively decomposed into a set of **states**
- the agent has to achieve a **goal**, which can be exhaustively decomposed into a numerical **reward** signal
- the agent is guided by a **policy**, which describes the method that the agent uses to estimate the rewards associated with each **state-action pair** (reward given an action performed in a given state).

For Markov decision processes, we again assume that the Markov property holds, we assume that the future is independent of the past given the present.

We can visualize a Markov decision process as follows:

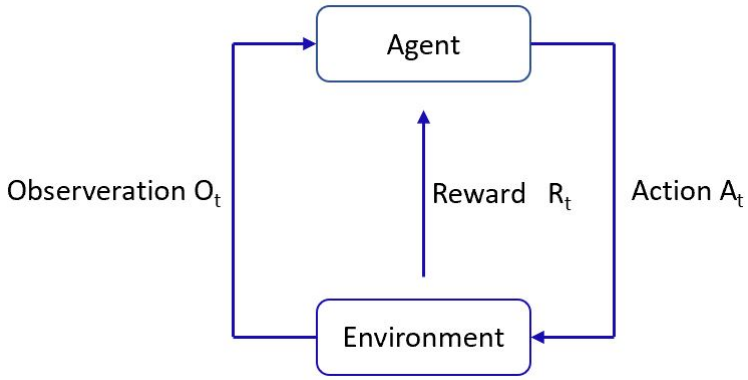


Figure 2. A Markov decision process

When we look at reinforcement learning problems, this is the fundamental process underlying the problem.

A Simple Reinforcement Learning Example

Let's look at a simple example of a reinforcement learning problem: pathfinding in a 15-grid world.

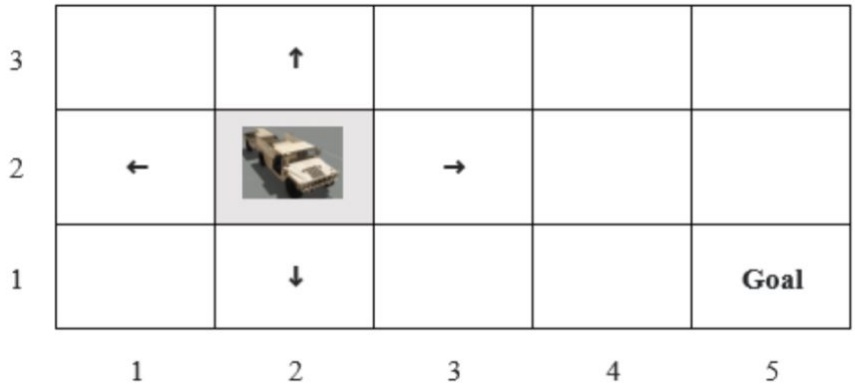


Figure 2. A 15-grid world

Here's the problem specification:

- Agent: the agent is an autonomous car that has four actions available to it (up, down, left, right)
- Environment: the environment encompasses 15 grids; the agent moves from grid to grid
- Goal: the agent's goal is to reach the grid marked as "Goal"
- Reward: when entering the grid marked as "Goal", the agent receives a reward of 100
- Policy: here we ignore how the agent evaluates the state-action pairs to evaluate its optimal actions

Through reinforcement learning, we want the agent to figure out the shortest path to the goal.

Policies

When we talk about different reinforcement learning algorithms, we're talking about different policies, or ways in which the agent can estimate the goodness of a state-action pair.

There are many different ways of evaluating state-action pairs and all of them lie beyond the scope of this course.

Instead, we'll look at a simple policy that can be used to solve multi-armed bandit problems, which are Markov decision processes with *one* state.

Specifying a Multi-Armed Bandit

Let's define a multi-armed bandit with eight levers (actions) as follows:

Action	0	1	2	3	4	5	6	7
Mean	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
Variance	2	2	2	2	2	2	2	2

All of these actions have the same variance but different mean rewards. We can see that action 7 is the optimal action as it has the highest mean reward (0.8).

Solving with Epsilon-Greedy

Epsilon-greedy (or e-greedy) is the most basic policy for solving a multi-armed bandit problem. This policy is defined around the variable epsilon, which states the agent's **exploration rate**, or the probability that the agent will not **exploit** the action with the current known best reward but **explore** another random action. This the most basic way to solve the trade-offs between exploiting a local optimum and discovering the global optimum that we discussed in the first lecture.

We estimate the true rewards of each action through a simple average: we take the cumulative reward received for an action and divide by the number of times that the action was performed.

Epsilon-Greedy in Action

Let's look at what happens when we apply different epsilon-greedy agents (with a different exploration rate) to the multi-armed bandit we specified.

We let the agents perform 5000 actions.

Figure 3. Actions selected by two different epsilon-greedy agents

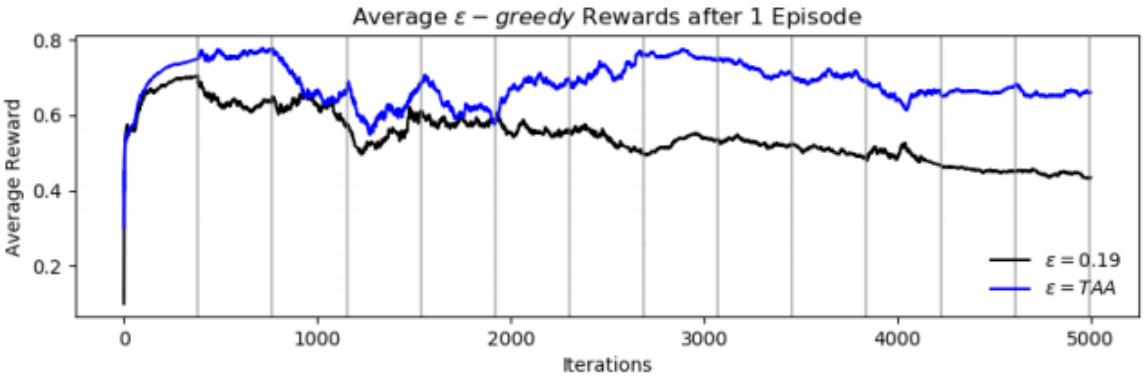


Figure 4. Average reward received by each agent for each of the 5000 actions

Figure 5. Average reward received by each agent after 5000 episodes

5 Human-in-the-Loop Reinforcement Learning

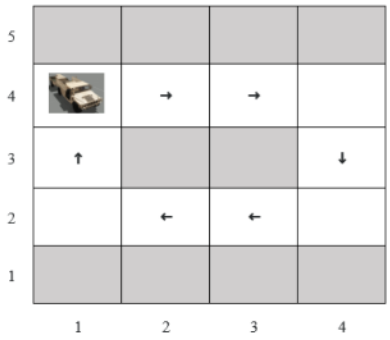
Reinforcement learning is a powerful technique but it comes with many unexpected failure modes that we often need human supervisors to fix.

In this section, as an exercise, we look at two of these failure modes and discuss how humans can fix them.

We look at **reward gaming** and **negative side effects**.

Reward Gaming

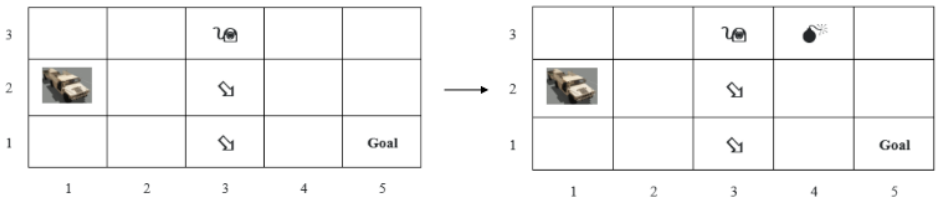
> Agent exploits an unintended loophole in the reward specification, to get more reward than deserved



> Desired outcome: clockwise completion of race
> Arrows are checkpoints associated with a reward of 3

Negative Side Effects

> Reward function does not fully capture all the properties of the test environment



Training environment *Test environment*

> Desired outcome: reach goal state
> (spotted by enemy) = -1, (bad terrain) = -3, (land mine) = - 100, **Goal** = 10