Personal Coverage Notes on Machine Learning

Authored by Dun-Ming Huang; Department of Computer Science, UC Berkeley

Preface(s)

0.1 Preface

In this personal coverage note, I try to record some thought processes and learning results I obtained from watching lectures and videos about numerous machine learning literature or applications. For now, the personal coverage note covers the following sections:

0.1.1 COMPSCI 189: Introduction to Machine Learning

This part covers the basic concepts of machine learning, including supervised learning, unsupervised learning, and focuses specifically on statistical learning techniques.

0.1.2 COMPSCI 285: Deep Reinforcement Learning

This part covers fundamental knowledge regarding deep reinforcement learning, sampled from Sergey Levine's Deep Reinforcement Learning lecture.

0.1.3 COMPSCI 294-158: Deep Unsupervised Learning

This part covers fundamental knowledge regarding deep unsupervised learning, sampled from Pieter Abbeel's Deep Unsupervised Learning lecture.

0.1.4 Mu Li's Videos on YouTube

This part covers the notes I took from watching Mu Li's videos on YouTube, which covers a wide range of topics in machine learning.

Contents

Part I

COMPSCI 189: Introduction to Machine Learning

Chapter 1

Fundamentals of Machine Learning

In this chapter, we cover the introductory lecture of COMPSCI 189.

Learning Goals:

- Understand the fundamental workings of machine learning.
- Learn about classification as an example machine learning task.

Machine learning is a popular topic over the recent centuries. It is a subset of artificial intelligence that focuses on the development of algorithms that allow computers to learn from and make predictions based on data. The study of machine learning per se is a long journey, even disregarding the participation of research activities. In this section of the note, we concentrate on statistical learning, which involve fundamental techniques of machine learning that are popularized before the current pulvinar of deep learning. Deep learning techniques will be addressed in later sections of the entire note.

1.1 The Framework of Machine Learning

Machine learning is the use and development of computer systems that can learn without explicit instructions; that is, they learn a specific pattern of the provided data via statistical measures, in an autonomous and algorithmic manner. The learning process is largely valuable on the ability of machine learning algorithms to draw insights, or **inferences**, upon the provided training data. Fundamentally, statistical learning is all about finding patterns in data, and using them to make predictions. An abstraction of this will be issued in Lecture 5 of the section.

Machine learning is a data-driven approach. As mentioned before, all that a machine learning can learn from is what the distribution of a provided training data provides. This is an important insight in the future. Just as how humans cannot learn what a cat is if they have never learned anything about a cat, a machine cannot learn about cat if the data we provide to its algorithm never describes what a cat is. In summary, what an algorithm learns is largely dependent on the data we provide to it.

1.2 Classification as Example Machine Learning Task

Classification, as you may have learned in highschool biology, is the process of categorizing things based on their properties. In machine learning, classification is a task that involves predicting the category of a given data point. For example, provided a picture that may entail a cat or a dog, a machine learning algorithm would be asked to classify it as either a cat or a dog. This is convenient in that humans do not have to process this judgment manually, and can instead automate this task with a fairly accurate algorithm.

How do we really decide if a given data point is a cat or a dog? For example, suppose the datapoint I am provided is an image, how do I transform this image into a decision's label (a cat, versus a dog)? In classification, we usually use

numbers to denote the label of a category (hereby we call it a "class"). A classifier h, therefore, is a function that is provided a datapoint $\vec{x}$ and outputs a numeric label for the representing class:

$$h(\vec{x}) = \begin{cases} 1 & \text{if the algorithm considers } \vec{x} \text{ is a cat} \\ 0 & \text{if the algorithm considers } \vec{x} \text{ is a dog} \end{cases}$$

The question now comes down to:

1. $\vec{x}$: How is the colorful image we see in human eyes represented as a vector?
2. "if the algorithm considers $\vec{x}$ as a something": how is this rule implemented programatically?

For question (1), the image's pixels can be converted into color-representing numeric values, then flattened into a vector based on the spatial ordering of pixels. For question (2), the algorithm learns a function h that outputs the above mapping. The takeaways of above questions are as follows:

1. The machine learning algorithm receives data in numeric form, such as a list of numbers (vectors), but not qualitatively.
2. The machine learning algorithm learns a function that is tailored to our need.

1.3 The Train-Validate-Test Framework

In machine learning, an algorithm usually follows the framework of train, validate, test. These aspects of the paradigm are summarized as follows.

1.3.1 Aspects of the T V T Framework

Training a Model. Recall that any machine learning algorithm produces a function h, which we also call a model, by having the algorithm detect patterns in our datapoints $\vec{x}$'s. The act of learning an appropriate function h that behaves well on our given datapoints is called **training** a model. That is, we are training a machine learning model on a provided dataset, and the resulting model should learn a function h that accurately predicts the class label (dog vs. cat) of a given datapoint $\vec{x}$ (an image). In this phase, models are provided a labeled dataset; that is, a set of images that are labeled either as a cat or a dog. Such dataset we use to train the model is otherwise known as a **training set**. Usually, we continue with the training phase until the algorithm's model has reached a satisfying accuracy for the training set.

Validating a Model. We have trained a model with images of cats and dogs, and now it's time to evaluate the model. More precisely, it's time to evaluate the model on datapoints it has not seen yet. After all, when a model is deployed into the real world, it is expected for the model to be able to classify cats and dogs from pictures immediately before us, mostly unseen to anyone, rather than just known images that are already labeled in a dataset. The dataset that we use to validate the model, entirely unseen during the training phase, is known as the **validation set**. We will stay in this phase until our model has reached a satisfying accuracy for the validation set.

Testing a Model. At last, we evaluate our model again using another unseen dataset, called the **testing set**. The testing set is used to evaluate the model's performance on a dataset that is entirely unseen during the training and validation phases. This is the final phase of the train-validate-test framework, and the model's performance on the testing set is the final metric of the model's performance.

1.3.2 Justification: Overfitting and Underfitting

Hi

1.3.3 A Summary of Questions up to This Point

Hi

Chapter 2

Linear Classifiers

Chapter Description.

2.1 section title

Section.

Theorem 2.1.1. Tested Theorem

I am the bone of my sword.

Definition 2.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 2.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 3

Gradient Descent

Chapter Description.

3.1 section title

Section.

Theorem 3.1.1. Tested Theorem

I am the bone of my sword.

Definition 3.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 3.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 4

Support Vector Machine

Chapter Description.

4.1 section title

Section.

Theorem 4.1.1. Tested Theorem

I am the bone of my sword.

Definition 4.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 4.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 5

It Is All About The Layers of Abstraction

Chapter Description.

5.1 section title

Section.

Theorem 5.1.1. Tested Theorem

I am the bone of my sword.

Definition 5.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 5.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 6

Decision Theory, Bayesian Decision Rule

Chapter Description.

6.1 section title

Section.

Theorem 6.1.1. Tested Theorem

I am the bone of my sword.

Definition 6.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 6.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 7

Gaussian Discriminant Analysis and Maximum Likelihood Estimation

Chapter Description.

7.1 section title

Section.

Theorem 7.1.1. Tested Theorem

I am the bone of my sword.

Definition 7.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 7.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 8

Eigendecomposition of Symmetric Matrices

Chapter Description.

8.1 section title

Section.

Theorem 8.1.1. Tested Theorem

I am the bone of my sword.

Definition 8.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 8.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 9

Abstractions of a Regression Problem

Chapter Description.

9.1 section title

Section.

Theorem 9.1.1. Tested Theorem

I am the bone of my sword.

Definition 9.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 9.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 10

Newton's Method and Logistic Regression

Chapter Description.

10.1 section title

Section.

Theorem 10.1.1. Tested Theorem

I am the bone of my sword.

Definition 10.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 10.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 11

Statistical Justifications for Regressions

Chapter Description.

11.1 section title

Section.

Theorem 11.1.1. Tested Theorem

I am the bone of my sword.

Definition 11.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 11.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 12

Ridge Regression and Regularization

Chapter Description.

12.1 section title

Section.

Theorem 12.1.1. Tested Theorem

I am the bone of my sword.

Definition 12.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 12.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 13

Decision Trees

Chapter Description.

13.1 section title

Section.

Theorem 13.1.1. Tested Theorem

I am the bone of my sword.

Definition 13.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 13.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 14

The Kernel Trick

Chapter Description.

14.1 section title

Section.

Theorem 14.1.1. Tested Theorem

I am the bone of my sword.

Definition 14.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 14.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 15

Introduction to Neural Networks

Chapter Description.

15.1 section title

Section.

Theorem 15.1.1. Tested Theorem

I am the bone of my sword.

Definition 15.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 15.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 16

Tricks and Heuristics for Neural Networks

Chapter Description.

16.1 section title

Section.

Theorem 16.1.1. Tested Theorem

I am the bone of my sword.

Definition 16.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 16.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 17

Convolutional Neural Network

Chapter Description.

17.1 section title

Section.

Theorem 17.1.1. Tested Theorem

I am the bone of my sword.

Definition 17.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 17.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 18

Unsupervised Learning: PCA

Chapter Description.

18.1 section title

Section.

Theorem 18.1.1. Tested Theorem

I am the bone of my sword.

Definition 18.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 18.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 19

Unsupervised Learning: Clustering Algorithms

Chapter Description.

19.1 section title

Section.

Theorem 19.1.1. Tested Theorem

I am the bone of my sword.

Definition 19.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 19.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 20

Unsupervised Learning: Clustering Algorithms

Chapter Description.

20.1 section title

Section.

Theorem 20.1.1. Tested Theorem

I am the bone of my sword.

Definition 20.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 20.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 21

Geometry of High-Dimensional Space

Chapter Description.

21.1 section title

Section.

Theorem 21.1.1. Tested Theorem

I am the bone of my sword.

Definition 21.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 21.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 22

Learning Theory

Chapter Description.

22.1 section title

Section.

Theorem 22.1.1. Tested Theorem

I am the bone of my sword.

Definition 22.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 22.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 23

AdaBoost

Chapter Description.

23.1 section title

Section.

Theorem 23.1.1. Tested Theorem

I am the bone of my sword.

Definition 23.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 23.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 24

k-Nearest Neighbor Approaches

Chapter Description.

24.1 section title

Section.

Theorem 24.1.1. Tested Theorem

I am the bone of my sword.

Definition 24.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 24.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Part II

COMPSCI 285: Deep Reinforcement Learning

Chapter 25

Introduction to Deep Reinforcement Learning

Chapter Description.

25.1 Motivation to Reinforcement Learning

Let us begin a motivating problem: how can we let a robot hand pick up something?

In classical robotics, the problemsolving process is to: (1) Define the problem in modeling perspectives, (2) Model the problem using mathematical equations, and (3) Solve the problem via a designed algorithm. However, as we accumulate technological knowledge, now we have a second option, which is to set it up as a machine learning problem. With the knowledge we have currently learned in statistical learning, we are inclined to use supervised learning; that is, provided some data of the robot and the environment, we train some model that can provide the robot an action to comply with. However, this approach is not well-informed from human experience, and the crafting of such data is still difficult. So, instead, we follow the line of thought of letting robots earn their own experiences via trial and error. This approach develops into a **reinforcement learning** setting.

In a reinforcement learning seting, robots collect examples of their own behavior, and label its success (significance as well) based on a state-derived reward signal (function). The robot then learns to maximize the reward signal by adjusting its behavior. Eventually, we obtain a **policy** (a function $\pi : \mathcal{S} \to \mathcal{A}$ that provides an action in response to a seen state) that the robot can follow to pick an object up.

So, reinforcement learning is really an experience-collecting framework: it allows for a freedom of trial, a disregard for a pre-existing dataset (although in most situations we will still have one), and inherits from machine learning approaches the waive of need to manually design solutions for each specific problem. Like statistical learning, reinforcement learning is also a massively scaled process of density estimations for underlying distributions (say, $p_\theta(x)$, or $p_\theta(y|x)$) of the training data. However, reinforcement learning is different in that it enables learning via agent-environment interactions, which provides a new source of information; and, it allows for new applications like evolutionary algorithms, controls, and optimizations. Reinforcement learning is mainly an approach for a design of behavior that does not require human intervention. These behavior are impressive because it is unthought of, as well as because of its delicate mimicry for human results (say, artistic outcomes).

25.2 Introduction to Reinforcement Learning

Reinforcement learning is both a mathematical ofmralism for learning-based decision making, and an approach for learning decision making and control from experience.

In supervised learning, the framework follows as a provided dataset $\mathcal{D} = \{(x_i, y_i)\}$ that contributes to the learning of a function $f : \mathcal{X} \to \mathcal{Y}$, which itself is the ultimate fruit of a supervised learning paradigm. Recall that supervised learning makes several assumptions regarding its paradigm. One, the data provided to us are i.i.d. samples from an underlying distribution. Two, we have known ground truth outputs in training.

In reinforcement learning, however, we ignore both of these assumptions. One, data is not i.i.d., becuase previous outputs influence future inputs (in a Markovian fashion). Two, ground truth answer is not known, and we are only provided a reward signal that notifies us whetehr a demonstration from the agent is successful or not. Reinforcement learning is therefore ran on reward-labeled data, rather than ground-truth-labeled data.

To summarize the paradigm of reinforcement learning, it is comprised of the following aspects:

1. An **agent** that interacts with the world to achieve a specific task.
2. An **environment** that the agent interacts with. This can be a Minecraft flat world for an agent that tries to learn walking on. This environment can be both in-real-life and simulated. In most occassions, it is simulated.
3. The input of learning is a **state** s_t that represents the current situation of the agent.
4. The output of learning, generally, is a **action** a_t that the agent takes in response to the state.
5. A **reward** r_t that the agent receives after taking an action.
6. A **policy** π that the agent follows to take an action.

The data we receive for reinforcement learning is therefore a sequence of (s_t, a_t, r_t) tuples that the agent collects from the environment as it interacts with it.

25.3 Motivation Towards Deep Reinforcement Learning

The fusion of data-driven AI and reinforcement learning provides us a complementary approach. In data-driven AI (deep learning), while we extract valuable inferences about the real world from data, we don't actively attempt to perform better than the data. Meanwhile, in reinforcement learning, while we extract emergent behavior to do better than existing data, we are not prepared with a way to extract inferences regarding the environment, and are not provided a means of using data at scale. That is, Data-Driven AI is about using data, while reinforcement learning is about using optimizations. Therefore, deep reinforcement learning is expected to excel at both learning and searching: learning from data and searching for (discovering) better ways to interact with the environment provided the data.

Noted, we have deep neural network architectures that extracts inference well, and RL algorithms that are compatible with these approaches. However, at the current stage, learning-based control in truly real-world settings remains a major open problem. We will discuss these topics at lengths with later sections of the note.

In the current state, we face the following open challenges:

1. We don't yet have amazing methods that both use data and Reinforcement Learning
2. Humans can learn incredibly quickly, but deep RL methods are usually slow, even in simulators.
3. Humans can reuse past knowledge, but domain transfer is a problem to deep reinforcement learning.
4. The role of prediction and design of reward functions are still not very clear in reinforecment learning.

Chapter 26

Supervised Learning of Behaviors

Chapter Description.

26.1 Terminology and Notation in DRL

In this section, we will cover several temrinologies and notations that the community uses regarding learning situations.

In our paradigm, we concern an input, a system that processes the input, and the output. Decision-making problems consider these as respectively observations o_t, policy $\pi_\theta(a_t|o_t)$, and actions a_t. These symbols are subscripted by time because the context of a decision-making problem is usually a chronological sequence of events. Note that, the production of next observation, o_{t+1}, is based on the impacted state o_t upon the transpiration of a_t. Note that the format of a_t, for example, is not limited to a vector; it can also be a continuous distribution, as we would sometimes like to sample actions to take rather than using an almost deterministic policy. The policy π_θ is parameterized by θ, which is a vector of parameters that the policy uses to make decisions (that is, to provide action provided observation). They, therefore, assign the probability to all possible actions provided a specific state.

We also introduce the notion of a state, s_t, which is a partial observation. This notion is introduced by the fact that the entirety of an environment is not always observable (and in most situations, we do not observe the entirety of an environment). Therefore, realistically, the policy we learn is $\pi_\theta(a_t|s_t)$, which we call a partially observed policy. A very appropriate analogy is perhaps our visual-neural system: our eyes guide our decisions based on what objects are posed in our environment, particularly what is in front of our eyes. But, we do not gain information from our eyes regarding what is behind us, simply because the environment we are situated in is partially observable: the sensors we have (eyes) simply do not detect what is behind. Therefore, in reinforcement learning paradigms, we work with state-action pairs rather than observation-action pairs.

We develop this into what we call a Markov Decision Paradigm. In this paradigm, we assume that the state s_t is sufficient to make decisions, and that the future state s_{t+1} is independent of the past states $s_{t-1}, s_{t-2}, \ldots$ given the current state s_t and the action a_t. The paradigm per se contains Markov-ness; that is, s_t only depends on s_{t-1}, and the connection of states is only nonzero for $p(s_{t+1}|s_t, a_t)$. The involvement of only states and actions is a reflection of the partial observability we suffer in environments, which produces a Partially Observable Markov Decision Process (POMDP).

Note: in some literature, states will be issued as x_t, and action u_t instead, due to the involvement of background in contorl theory from some influential figures.

26.2 Imitation Learning

Imitation learning concerns the learning of a policy π_θ from a dataset of expert demonstrations. **Behavioral cloning**, an approach of imitation learning, is the idea that, via supervised learning, we learn a policy that regresses an underlying function $f : \mathcal{S} \to \mathcal{A}$ that maps the expert-induced states to experr-performed actions. These expert demonstrations are, as assumed, provided to the algorithm. However, behavioral cloning is usually not a good solution to general problems. Therefore, we expect the agent to practically clone the behavior (state-action reaction) of the expert.

The reason why is because, although expert demonstrations provide us many trajectories, once we receive a state outside of the provided trajectories, the actions our agent provides are not well-defined in nature. It is moreso that, once the agent is exposed to an unseen state, the action it provides via its policy does not guarantee a continued cloning of the expert-demonstrated trajectory. This is because the nature of a trajectory destructs the i.i.d. assumption of supervised learning; that is, because of temporal dependencies, we encounter problems in behavioral cloning that does not appear in conventional supervised learning problems. (A spam classifier generalizes well to unseen emails, but a behavioral cloning agent does not generalize well to unseen states). We can propose ad-hoc solutions, such as data augmentation that increases familarity of the agent to unseen states, but these solutions are not always guaranteed to work.

To make behavioral cloning work, we must make modifications to the existing paradigm. Let us begin with lessons of the story:

1. Different from conventional supervised studying problems, imitation learning via behavioral cloning is not guaranteed to work.
2. To work against it, we can use data augmentation as well as modify data collection methods.
3. An exotic solution, like a multi-task learning formulation, may help to generalize to unseen states and perform good imitation learning.
4. The most intuitive approach is perhaps modifying the algorithm along which we do imitation learning.

26.3 Theoretical Analysis of Failure in Behavioral Cloning

The main culprit of behavioral cloning's failure is the dsitributional shift problem. Suppose that we have some policy $\pi_\theta(a_t|o_t)$ trained on a dataset of expert demonstrations. Here, let us say that the distribution provided by expert dataset is $p_{data}(o(t))$, but the distribution of observation the policy faces is $p_{\pi_\theta}(o_t)$. Note that, since our policy is trained under $p_{data}o(t)$, the objective of that training is $\max_\theta \mathbb{E}_{o_t \sim p_{data}(o(t))} \log \pi_\theta(a_t|o_t)$. However, the policy is evaluated under $p_{\pi_\theta}(o_t)$, which is not the same as the training distribution. This problem is otherwise known as **distributional shift**, and this occurs due to the policy's own deviation from the training distribution.

The lesson of such problem is that we should perhaps define more precisely what we want to define as "well-learned". A policy that is good would perhaps not be a point-estimate for actions, since it can easily lead to deviations in policy behavior. Perhaps we may define a cost otherwise. Suppose that π^* is the optimal policy for us to clone:

$$c(s_t, a_t) = \begin{cases} 0 & \text{if } a_t = \pi^*(s_t) \\ 1 & \text{otherwise} \end{cases}$$

Now, then, our training objective becomes:

$$\min \mathbb{E}_{s_t \sim p_{\pi_\theta}(s_t)}[c(s_t, \pi_\theta(s_t))]$$

Assume that we have an upper bound of the policy mistake, $\pi_\theta(a \neq \pi*(s)|s) \leq \epsilon$. Then, we observe an incurred cost of:

$$\begin{aligned} &\mathbb{E}[\sum_t c(s_t, \pi_\theta(s_t))] \\ &\leq \epsilon T + (1-\epsilon)(\epsilon(T-1) + (1-\epsilon)(\dots)) \in 0(\epsilon T^2) \end{aligned}$$

Therefore, the error of behavioral cloning is quadratic to the length of its trajectory (T).

It turns out that an analysis composed by Ross et al. (2011) shows that the error of behavioral cloning is quadratic to the length of the trajectory. That is, imitation learning is prone to failure at any timestep, and lacks a means of recovery from a policy's small failures.

26.4 Addressing Problem of Imitation Learning

To address the problem of imitation learning, we can consider the following approaches:

1. Data Augmentation and collection
2. Powerful models that make very few mistakes
3. Multi-task learning formulation of imitation learning
4. Changing the algorithm of use (where we discuss DAgger)

Data Augmentation and Collection. Behavioral cloning is difficult because the model doesn't generalize well to mistakes. What is we involve data regarding mistakes instead? If the dataset involves mistakes, due to additional steps in data collection and augmentation process, although the training set will be diluted, we can now access corrections in our BC paradigm.

Powerful Models. Failures from fitting the expert can stem from several reasons. First, the expert's behavior is non-Markovian. While the policy we train is Markovian-ly conditioned, the expert's behavior are not formulated on a Markovian approach. That is, provided exposure to states $s_t = s_{t'}$, it is likely that the action elicited from these states are different. Therefore, human demonstrators post a very unnatural circumstance for a Markovian policy. Perhaps one remedy is to use the entire history of a trajectory, and a sequence model is capable of processing it as a temporal sequence of frames. However, the exploitation of entire history may still work poorly, simply because including the history may still be harmed by incomplete information, which can lead to causal confusion. Second, the expert's behavior may be multimodal. That is, the expert may have multiple ways of solving a problem, and the policy we train may not be able to capture all of these modes. This is specifically unhelpful for a continuous distribution of actions, which rely on the use of mean and variance that is difficultly characterizing for bimodal distributions. Although we can instead choose expressive continuous distributions for actions (namely, use other classes of distributions), or use an autoregressive discretization with high-dimensional action spaces, they do pose higher computational costs to this procedure.

Multi-task Learning Formulation. Training a policy that has one sole destination of trajectory can be difficult, but perhaps a multi-task learning formulation that attempts to let policies reach multiple different destinations. This approach can be summarized as "goal-conditioned behavioral cloning", which can provide more opportunities to learn corrections despite distributional shift; that is, to maximize $\log \pi_\theta(a_t^i | s_t^i, g = s_T^i)$. However, this approach actually introduces a secondary source for distributional shift, making the approach theoretically worse.

Changing the Algorithm of Use: DAgger. The idea of DAgger, Dataset Aggregation, is to make $p_{data}(o_t) = p_{\pi_\theta}(o_t)$. That is, we collect training data from $p_{\pi_\theta}(o_t)$ by running the policy $\pi_\theta(a_t|o_t)$ in the environment. Then, we aggregate the dataset of expert demonstrations and the dataset of the policy's own data, and train the policy on the aggregated dataset. The algorithm is as follows:

1. Collect a dataset of expert demonstrations $\mathcal{D} = \{(o_t^i, a_t^i)\}_{i=1}^N$.
2. For $k = 1, 2, \ldots, K$:
   (a) Train the policy π_θ on $\mathcal{D}$.
   (b) Collect a dataset of the policy's own data $\mathcal{D}_{\pi_\theta} = \{(o_t^i, \pi_\theta(o_t^i))\}_{i=1}^N$.
   (c) Aggregate the datasets $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_{\pi_\theta}$.

26.5 A Hint at the Need of Reward and Cost Signals

Deep learning works best when data is plentiful, but humans are finite sources of data (that is, humans cannot provide all data and label all data). Therefore, to resolve the disruptive demand of large ground truths, we expect our reinforcement learning algorithm to learn autonomously. To enable algorithms to evaluate their experiences, and exceeding their own performances, we offer a reward signal called **reward function**, $r(s, a)$, which provides a numeric evaluation for an observed pair of state and action. For example, for imitation learning, we can propose the reward function $r(s, a) = \log p(a = \pi^*(s)|s)$.

Chapter 27

Introduction to Reinforcement Learning

Chapter Description.

27.1 Foundations, in A Comprehensive Manner

In prior, we have learned that reinforcement learning focuses on learning a policy $\pi_\theta(a_t|o_t)$ that provides an action provided an observation. Here, a_t is an action at timestep t, while o_t is an observation at timestep t. Note that, in a not fully observed context, we instead discuss s_t as a state at timestep t in place of the observation. And, that, the decision paradigm we adopt here is Markovian: the only determining factor of the current timestep is the previous timestep.

Different from imitation learning, a supervised learning setting, we use the reward function to notate the success of a performance. A reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a function that provides a scalar reward for a state-action pair; as the name suggests, a higher reward resembles a better performance. However, reinforcement learning is not about short-term maximizations of reward functions, but long-term maintenance of high reward values. A temporal abstraction is therefore also imposed upon reinforcement learning formulations.

27.1.1 Mathematical Objects

A Markov Chain is a stochastic object defined as $\mathcal{M} = \{\mathcal{S}, \mathcal{P}\}$ where $\mathcal{S}$ is the set of all states (state-space), and $\mathcal{T}$ is a transition operator that encodes $p(s_{t+1}|s_t)$ for all possible pairs (s_t, s_{t+1}). The transition operator is expressable as a matrix, where each row sums to 1, and each element is a probability of transitioning from one state to another. That is, $\mathcal{T}_{i,j} = p(s_{t+1} = i|s_t = j)$.

A reinforcement learning paradigm, then, can be framed as a variation of Markov Chain, called a Markov Decision Process. This process is formulated as $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$, where:

- $\mathcal{S}$ is the state-space, as before.
- $\mathcal{A}$ is the action-space, a set of all possible actions.
- $\mathcal{T}$ is the transition operator, as before.
- r is the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, as before.

A partially observed Markov decision process is just an augmented MDP: $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{E}, r\}$, where $\mathcal{E}$ is the emission probability $p(o_t|s_t)$.

27.1.2 Objective of Reinforcement Learning

In reinforcement learning, we learn a policy-representing object $\pi_\theta(a|s)$. In the chain rule of states within some trajectory, we may discover that:

$$p_\theta(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

We denote the trajectory $s_1, a_1, \ldots, s_T, a_T$ as τ. The objective of reinforcement learning is to maximize the expected return, or the expected sum of rewards:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta} \left[\sum_{t=1}^{T} r(s_t, a_t) \right]$$

and we would like a policy equipped with the parameter θ that maximizes this return:

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

Let us attempt to define the finite horizon case of the expected return using an augmented state-space for markov chain on state-action pairs. Therefore, by linearity of expectation,

$$\begin{aligned}
\theta^* &= \arg\max_\theta J(\theta) \\
&= \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta} \left[\sum_{t=1}^{T} r(s_t, a_t) \right] \\
&= \arg\max_\theta \sum_{t=1}^{T} \mathbb{E}_{\tau \sim p_\theta} \left[r(s_t, a_t) \right]
\end{aligned}$$

That also means, in an infinite-horizon case, where $T = \infty$, our objective may become ill-defined unless we obtain a finite mean of reward.

27.2 Value Functions

We can also express our expected reward objective in a recursive manner:

$$\mathbb{E}_{s_1 \sim p(s_1)} \left[\mathbb{E}_{a_1 \sim p(a_1)} \left[r(s_1, a_1) + \mathbb{E}_{s_2 \sim p(s_2|s_1, a_1)} \left[\mathbb{E}_{a_2 \sim p(a_2|s_2)} \left[r(s_2, a_2) + \cdots \right] |s_1, a_1 \right] |s_1 \right] \right]$$

To simplify this expression, we express:

$$Q(s_1) = r(s_1, a_1) + \mathbb{E}_{s_2 \sim p(s_2|s_1, a_1)} \left[\mathbb{E}_{a_2 \sim p(a_2|s_2)} \left[r(s_2, a_2) + \cdots \right] |s_1, a_1 \right]$$

such that,

$$\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right] = \mathbb{E}_{s_1 \sim p(s_1)} \left[\mathbb{E}_{a_1 \sim p(a_1)} \left[Q(s_1) | s_1 \right] \right]$$

Now, with this concise notation, we may easily modify π_θ based on the function Q. The question, then, is how do we know the function Q, otherwise called the Q-function?

So, as we have defined in prior, the Q-function is expressed as:

$$Q^\pi(s_t, a_t) = \sum_{t=t'}^{T} \mathbb{E}_{\pi_\theta} \left[r(s_t, a_t) | s_t, a_t \right]$$

The value function, then, is defined as:

$$V^{\pi}(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[Q^{\pi}(s_t, a_t) \right]$$

Now, we can also express the reinforcement learning objective as $\mathbb{E}_{s_1 \sim p(s_1)}[V^{\pi}(s_1)]$.

How can we use Q-functions and value functions? One immediate application of them is to formulate a policy that only outputs the maximum-Q-inducing action. Another idea would be to compute the gradient of Q^{π} to increase the probability of a good action a, building on the precondition that $Q^{\pi}(s, a) > V^{\pi}(s)$ implies a as better than an average action, so that we modify $\pi(a|s)$ to increase the probability of the aforementioned condition.

27.3 Algorithms in Reinforcement Learning

Reinforcement learning algorithms share the following high-level anatomy:

- Collect data from the environment (via running the policy).
- Update the policy using the collected data (to fit the policy's model).
- Improving the policy (this is some form of optimization).

Here, generating samples has an expense that depends on the task of reinforcement learning; for example, under a robotics setting, the sample generation process is very costly without a simulator. Fitting a model and improving policies also has an expense dependent on the approach.

27.3.1 Diversity in RL Algorithms

Here is a general categorization of reinforcmenet learning algorithms.

Policy Gradient algorithms directly differentiate the expected-reward objective. While the gradient can be estimated, the improvement of policy relies on a gradient descent step that will be introduced in the next lecture.

Value-based algorithms estimate a value function or Q function of the optimal policy, rather than having an explicit policy. In this family of algorithms, we use a model to fit $V(s)$ or $Q(s, a)$, and use this model to plan actions. The policy is simply $\pi(s) = \arg\max_a Q(s, a)$.

Actor-Critic algorithms combine the two, where the critic is an estimator of the value function, and the improvement of policy relies on a gradient descent step.

Model-based algorithms learn a model of the environment, and use this model to plan actions. For model-based algorithm, fitting a model also involves learning a transition operator $\mathcal{T}$, while the policy improvement aspect has many options: (1) using the learned transition to plan eventual actions; (2) backpropagating gradients into the policy network; (3) using the transiton model to learn a value function, and use dynamic programming to accelerate the improvement progress.

Different families of algorithms have tradeoffs of performance on several fields, ranging across the following examples:

Sample Efficiency. Sample efficiency is the matter of "how many samples do we need to get a good policy", and one important factor of this is whether the algorithm is on-policy (online, new samples are needed upon any update of policy) or off-policy (the algorithm improves the policy without generating new samples from that policy). On the spectrum of sample efficiency, on-policy algorithms generally host less sample efficiency. However, having a less sample efficiency is acceptable because of the tradeoff it can bring, such as the need of online-ness that is saved by on-policy algorithms' online-ness in exchange of smaller sample efficiency.

Stability. Stability and ease of use discusses whether the algorithm converges, how often does it converge, and what does the algorithm converge to (if at all). This is a question because reinforcement learning might not be using gradient-based optimizations. For example, Q-learning is a fixed-point iteration scheme, while model-based RL concentrates on a transition model not optimized for expected reward.

Assumptions from Algorithms. Algorithms pose assumptions about our reinforcement learning paradigm. Some common assumptions include: full observability, episodic learning, and continuity or smoothness of specific objectives.

Chapter 28

Policy Gradients

Chapter Description.

28.1 Foundations of Policy Gradient

Remember that the objective of reinforcement learning is to maximize the expected return of our policy π_θ over time. In deep reinforcement learning, we would formulate that the policy is a neural network taking in some state s_t and outputting some action a_t. And, we may also define a trajectory distribution:

$$p_\theta(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

In this note, the shorthand of such distribution is $p_\theta(\tau)$.

Crucially, in development of model-free algorithms, we do not assume that we know the transition probabilities $p(s_{t+1}|s_t, a_t)$ or the initial state probability $p(s_1)$, but treat real-world interaction as an act of sampling per se. And, the objective of our algorithm is to find parameter θ where:

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

where

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

for which the finite horizon case can be simplified into a sum of expectations via the linearity of expectation.

Let us attempt to evaluate the objective of our algorithm before discussing its optimization. We may find an unbiased estimator of $J(\theta)$ by sampling trajectories via policy-world interactions:

$$J(\theta) \sim \frac{1}{N} \sum_i \sum_t r(s_{i,t}, a_{i,t})$$

However, we do want to improve the objective beyond estimating it. The estimate of the derivative of objective also needs to be feasible without knowledge of transition probability and state prior. To satisfy the above demands, let us

propose as follows:

$$\begin{aligned}
J(\theta) &= \mathbb{E}_{\tau \sim p_\theta(\tau)}[r(\tau)] \\
&= \int p_\theta(\tau) r(\tau)\, d\tau \\
\nabla_\theta J(\theta) &= \int \nabla_\theta p_\theta(\tau) r(\tau)\, d\tau \\
&= \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau)\, d\tau \\
&= \mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)] \\
&= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right]
\end{aligned}$$

To evaluate the policy gradient, we can run the policy to generate samples from $p_\theta(\tau)$, and then multiply them by the gradients. Then, we improve the policy by taking a step in the direction of the gradient:

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

The sketch of the algorithm, otherwise known as REINFORCE, is as follows:

- Sample $\{\tau^i\}$ from running the policy π_θ.
- Compute the policy gradient: $\nabla_\theta J(\theta) = \sum_i (\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}))(\sum_t r(s_{i,t}, a_{i,t}))$.
- Update the policy: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$.
- Repeat.

Policy gradients are then reward-weighted versions of maximum likelihood objective. This may be observed from their expressions:

$$\begin{cases}
\nabla_\theta J(\theta) & \sim \frac{1}{N} \sum_i (\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}))(\sum_t r(s_{i,t}, a_{i,t})) \\
\nabla_\theta J_{ML}(\theta) & \sim \frac{1}{N} \sum_i (\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}))
\end{cases}$$

That is, in policy gradient, high reward trajectories have increased log probabilities, while low reward trajectories have decreased log probabilities.

Under the constraint of continuous actions, on the other hand, we select a representation of policy π that outputs a representation of continuous distribution, such as a Gaussian policy: $\pi_\theta(a_t|s_t) = \mathcal{N}(f(s_t), \Sigma)$. Then, the log probability of an action can be written in terms of an anisotropic Gaussian distribution's, or some other continuous distribution's PDF.

Note that, the implementation of policy gradient deamnds automatic differentiation, such that the policy gradient can be computed by a library within reasonable computation time. Typically, this is inefficiency, becuase neural networks have more parameters within itself than samples that it uses.T To calculate the gradient in a neural network, we would often instead employ back-propagation, which the automatic differentiation will set up a computational graph for us as well when it comes to computation of poicy gradient. We may, fortunately, construct a computational graph for which the gradient of it is the policy gradient. This is often by the involvement of Q values, and having an alternative objective $\tilde{J}(\theta)$ which grants us the policy gradient on its computational graph of gradients.

28.2 Reducing Variance in Policy Gradient

However, the property of policy gradients to skew away from bad trajectories and towards good ones, and this can introduce pathological changes led by high variance of its estimator. The policy gradient estimator we described before has very high variance depending on the acquired samples.

28.2.1 Causality Trick

The first trick we can use to reduce variance is causality, which is that a policy at time t' cannot affect the reward at time t when $t < t'$. This allows us to rewrite the policy gradient as:

$$\nabla_\theta J(\theta) \sim \frac{1}{N} \sum_i \sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}) \right)$$

Which allows us to discard rewards from the past that our reward policy cannot impact. Then, we obtain a lower-variance estimate, because the total sum is a smaller number (which accompanies a smaller variance). This is otherwise known as a reward-to-go formulation.

28.2.2 Baseline Trick

The second trick we can use to reduce variance is the baseline trick. Becuase subtracting a baseline from the reward aspect in policy gradient does not change its expectation (a proof is provided in lecture), but changes its variance, for any baseline b, the following trick reduces the variance of the policy gradient while maintaining the estimator as unbiased. Let $b = \frac{1}{N} \sum_{i=1}^{N} r(\tau_i)$. Then, we use the baselined policy gradient instead:

$$\nabla_\theta J(\theta) \sim \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log p_\theta(\tau)[r(\tau) - b]$$

Furthermore, via derivation, we find the optimal variance-reducing baseline to have value:

$$b^* = \frac{\mathbb{E}[g(\tau)^2 r(\tau)]}{\mathbb{E}[g(\tau)^2]}$$

28.3 Off-Policy Policy Gradient

Because the policy gradient is a sample-based estimator, the requirement of sampling from the policy upon every policy update becomes a large operational cost. This is a disadvantage of the REINFORCE algorithm's online-ness (it being on-policy). This immediately attracts us to convert the algorithm into an off-policy one, via an access to some $\bar{p}(\tau)$ that is not the policy we are optimizing for, so that we can update the policy while being offline (that is to be off-policy, then).

This may be operated along an importance sampling trick:

$$\begin{aligned}
\mathbb{E}_{x \sim p(x)}[f(x)] &= \int p(x) f(x)\, dx \\
&= \int \frac{p(x)}{q(x)} q(x) f(x)\, dx \\
&= \mathbb{E}_{x \sim q(x)} \left[\frac{p(x)}{q(x)} f(x) \right]
\end{aligned}$$

Then, we obtain our reinforcement learning objective as:

$$J(\theta) = \mathbb{E}_{\tau \sim \bar{p}(\tau)} \left[\frac{p_\theta(\tau)}{\bar{p}(\tau)} r(\tau) \right]$$

where

$$\begin{aligned}
\frac{p_\theta(\tau)}{\bar{p}(\tau)} &= \frac{p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)}{p(s_1) \prod_{t=1}^{T} \bar{\pi}(a_t|s_t) p(s_{t+1}|s_t, a_t)} \\
&= \prod_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\bar{\pi}(a_t|s_t)}
\end{aligned}$$

The policy gradient is then re-expressed to be:

$$\begin{aligned}\nabla_{\theta'} J(\theta') &= \mathbb{E}_{\tau\sim\bar{p}(\tau)}\left[\frac{p_\theta(\tau)}{\bar{p}(\tau)}\nabla_{\theta'}\log\pi_{\theta'}(\tau)r(\tau)\right] \\ &= \frac{1}{N}\sum_i(\sum_t\prod_{t=1}^{T}\frac{p_\theta(\tau)}{\bar{p}(\tau)})(\sum_t\nabla_\theta\log\pi_\theta(a_{i,t}|s_{i,t}))(\sum_t r(s_{i,t},a_{i,t}))\end{aligned}$$

This expression can be further advanced using the causality trick.

28.4 Covariant Policy Gradient

There exist other problems with policy gradient algorithms. For instance, suppose we have a one-dimensional action space, where the policy is Gaussian. Then, as the gradient with respect to the variance becomes larger due to the reduction of variance over time development, we will find the policy gradient not quite pointing the agent towards an optimum (destination). Therefore, following the policy gradient takes a long time until the optimal parameter is reached.

One approach to prevent this problem is known as covariant/natural policy gradient. To choose the learning rate α of our policy gradient step, we take into account that some parameters change probabilities a lot more than others, and therefore give learning rates corresponding to the sensitivity of each parameter. That is, we instead follow the step:

$$\theta' \leftarrow \arg\max_{\|\theta'-\theta\|_2^2\leq\epsilon}(\theta'-\theta)^T\nabla_{\theta'}J(\theta)$$

We are free to replace the constraint $\|\theta'-\theta\|_2^2\leq\epsilon$ with a different parameterization-independent divergence measure, such as $D_{KL}(\pi'_\theta,\pi_\theta)\leq\epsilon$. The quadratic Taylor expansion of divergence constraints further enable different optimization program formulations. Specifically, the quadratic Taylor expansion of KL divergence stated above grants us a policy fradient step:

$$\theta\leftarrow\theta+\alpha F^{-1}\nabla_\theta J(\theta)$$

where F is resulted from the use of KL divergence and known as the Fisher Information matrix:

$$F=\mathbb{E}_{\pi_\theta}\left[\nabla_\theta\log\pi_\theta(a|s)\nabla_\theta\log\pi_\theta(a|s)^T\right]$$

And we may solve for an optimal learning rate α while solving $F^{-1}\nabla_\theta J(\theta)$.

Chapter 29

Actor-Critic Algorithms

Chapter Description.

29.1 Policy Evaluation

Recall from policy gradient that we have considered such a value:

$$\hat{Q}_{i,t'} = \sum_{t=t'}^{T} r(s_{i,t}, a_{i,t})$$

As this is a single-step estimate of the reward to go, we would rather compute an expectation over all possible state-action pairs to capture this reward-to-go estimation more closely. This problem is also directly related to the high variance of policy gradient methods. That is, the policy's computation fo reward-to-go would have a high variance due to it being a single-sample estimate of a quite complex expectation. Thus, we phrase the Q-function as:

$$Q(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi} \left[r(s_{t'}, a_{t'} | s_t, a_t) \right]$$

Meanwhile, we can phrase a baseline that is the expectation of reward-to-go over all possible trajectories starting from some specific state, forming what we call a value fucnction:

$$V(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}[Q(s_t, a_t)]$$

And therefore, the Q-function provides us an insight on how good our action is than the average that is presented by the value function. This $Q(s_t, a_t) - V(s_t)$ expression is otherwise called an advantage function.

Specifically, we would express the Q-function and Value function, which are respectively the total reward from taking a_t in s_t and the total reward from s_t, as Q^π and V^π. Then, the advantage value $A^\pi(s_t, a_t)$ presents how much better a_t is as an action. We would therefore have the following objective gradient:

$$\nabla_\theta J(\theta) \sim \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) A^\pi(s_{i,t}, a_{i,t})$$

where the better this estimate, the lower the variance of such estimate.

Then, the objective of our algorithm naturally turns to attempting to fit the value function. Why the value function? This is becuase the Q-function is in fact form-able from the value function as:

$$Q^\pi(s_t, a_t) \sim r(s_t, a_t) + V^\pi(s_{t+1})$$

Note that, the value function is only dependent on the state, making it easier to fit $V^\pi(s)$ than other functions. This fitting process is otherwise called policy evaluation.

To estimate our value function, we can perform Monte Carlo methods, which is to run the policy and sample value functions, then averging them to make our estimator. This method is especially relevant in a reset-able context, like a simulator. Furthermore, we can approximate V^π by using a neural network, which reduces our variance because the neural network can identify similar states and therefore estimate their values with mutual considerations. Particularly, the neural network would be capable of noticing the locality of space (that similar states would have similar rewards). Note that for the above process, we can perform a bootstrap estimate of the ideal target as:

$$y_{i,t} \sim r(s_{i,t}, a_{i,t}) + \hat{V}^\pi_\phi(s_{i,t+1})$$

29.2 Actor-Critic Algorithm

A batch actor-critic algorithm, based on policy evaluation, follows the outline below roughly:

1. Sample $\{s_i, a_i\}$ from $\pi_\theta(a_i|s_i)$ by running it on the agent.
2. Fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums
3. Evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \hat{V}^\pi_\phi(s_{i+1}) - \hat{V}^\pi_\phi(s_i)$
4. Compute the policy gradient:

$$\nabla_\theta J(\theta) \sim \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \hat{A}^\pi(s_{i,t}, a_{i,t})$$

5. Update θ by a gradient step: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$.

To account for infinite-horizon settings, we use a discount factor that is exponentially applied across timesteps to prevent an infinitely large value function. This makes intuitive sense: we generally do not mind rewards that come in 100 timesteps compared to what will come in 10. The mathematical formulation this discount factor-involving equation is:

$$y_{i,t} \sim r(s_{i,t}, a_{i,t}) + \gamma \hat{V}^\pi_\phi(s_{i,t+1})$$

where $\gamma \in [0, 1]$ is the discount factor. Popular choices of it lie in $[0.9, 0.99]$. This changes our policy gradient formulation into:

$$\nabla_\theta J(\theta) \sim \frac{1}{N} \sum_i \sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) \right)$$

or **in**equivalently with more importance at earlier timesteps' decision:

$$\nabla_\theta J(\theta) \sim \frac{1}{N} \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t'=1}^{T} \gamma^{t'-1} r(s_{i,t'}, a_{i,t'}) \right)$$

the latter option is more naturally suited towards a discounted problem, but is also not necessarily the better solution. When applying discounts, we just replace the third step of batch actor-critic algorithm as:

$$A^\pi(s_i, a_i) = r(s_i, a_i) + \gamma \hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i)$$

An online version of the actor-critic algorithm is also possible, where we update the policy after each timestep:

1. Take an action $a \sim \pi_\theta(a|s)$, and obtain (s, a, r, s').

2. Update the value function $\hat{V}_\phi^\pi$ using the target $y = r + \gamma \hat{V}_\phi^\pi(s')$.
3. Compute the advantage $A^\pi(s,a) = r + \gamma \hat{V}_\phi^\pi(s') - \hat{V}_\phi^\pi(s)$.
4. Construct, then, a policy gradient: $\nabla_\theta J(\theta) \sim \nabla_\theta \log \pi_\theta(a|s) \hat{A}^\pi(s,a)$.
5. Update θ by a gradient step: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$.

29.3 Actor-Critic Design Decisions

To instantiate our algorithm as a deep reinforcement learning one, we need to consider choices for our value fucntion and policy function. The most fundamental design is to have the fitted value function and the policy as separate neural networks. This is a simple and stable option, but there will be no shared features between the actor (policy) and the critic, which we could have benefitted from. However, having shared layers for their networks will also cause shared gradients at different scales, which would then require more hyperparameter tuning to stabilize.

Another point of discussion is batch-sizes, which related to the online version of our algorithm as it learns from occurring datapoints. Updating deep neural networks using SGD usually results in an overly high variance, so an online actor-critic algorithm functions best with batches, which can be simply enabled by parallelization. That is to have parallel simulated environments under different random seeds to collect data with. Therefore, across each parallel environment, we maintain the transition acuqiring-policy updating two-step procedure. This design can be either synchronized or asynchronous, with the latter granting higher flexibility. Asynchronous threads may cause slight out-of-distribution updates for each policy, but in a theoretical ground, it would perform finely with respect to the original actor-critic standard.

Yet another design decision discusses whether we can remove the on-policy assumption entirely. This is the principle of an off-policy actor-critic, which provides a replay buffer for transition sampling from. This allows us to update our policy using data we logged into the replay buffer, which may have came from an older version of the policy.

29.4 Critics as Baselines

The critic in an actor-critic algorithm can be used as a baseline for the policy gradient, and this provides for an interesting tradeoff with respect to the actor-critic algorithm we usually know. The actor-critic policy gradient is a low-variance estimate but biased, while the original policy gradient is unbiased but has a high variance. To keep the estimator unbiased while using the fitted value function $\hat{V}_\phi^\pi$, we may use the following estimator for policy gradient:

$$\nabla_\theta J(\theta) \sim \frac{1}{N} \sum_i \sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) \right) - \hat{V}_\phi^\pi(s_{i,t}) \right)$$

which is low-variance and unbiased.

Methods that use state and action-dependent baselines are also known as control-variates. They are another way for variance reduction and unbiased-ness:

$$\nabla_\theta J(\theta) \sim \frac{1}{N} \sum_i \sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) (\hat{Q}_{i,t} - Q_\phi^\pi(s_{i,t}, a_{i,t})) + \frac{1}{N} \sum_i \sum_t \nabla_\theta \mathbb{E}_{a_t \sim \pi_\theta}[Q_\phi^\pi(s_{i,t}, a_t)]$$

Some other interesting tricks, like GAE and n-step return estimators, are not to be written in this chapter but considered as separate paper reviws.

Chapter 30

Value Function Methods

Chapter Description.

30.1 section title

Section.

Theorem 30.1.1. Tested Theorem

I am the bone of my sword.

Definition 30.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 30.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 31

Deep RL with Q-Functions

Chapter Description.

31.1 section title

Section.

Theorem 31.1.1. Tested Theorem

I am the bone of my sword.

Definition 31.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 31.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 32

Advanced Policy Gradients

Chapter Description.

32.1 section title

Section.

Theorem 32.1.1. Tested Theorem

I am the bone of my sword.

Definition 32.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 32.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 33

Optimal Control and Planning

Chapter Description.

33.1 section title

Section.

Theorem 33.1.1. Tested Theorem

I am the bone of my sword.

Definition 33.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 33.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 34

Model-Based Reinforcement Learning

Chapter Description.

34.1 section title

Section.

Theorem 34.1.1. Tested Theorem

I am the bone of my sword.

Definition 34.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 34.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 35

Model-Based Policy Learning

Chapter Description.

35.1 section title

Section.

Theorem 35.1.1. Tested Theorem

I am the bone of my sword.

Definition 35.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 35.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 36

Exploration (Part 1)

Chapter Description.

36.1 section title

Section.

Theorem 36.1.1. Tested Theorem

I am the bone of my sword.

Definition 36.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 36.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 37

Exploration (Part 2)

Chapter Description.

37.1 section title

Section.

Theorem 37.1.1. Tested Theorem

I am the bone of my sword.

Definition 37.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 37.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 38

Offline Reinforcement Learning (Part 1)

Chapter Description.

38.1 section title

Section.

Theorem 38.1.1. Tested Theorem

I am the bone of my sword.

Definition 38.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 38.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 39

Offline Reinforcement Learning (Part 2)

Chapter Description.

39.1 section title

Section.

Theorem 39.1.1. Tested Theorem

I am the bone of my sword.

Definition 39.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 39.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 40

Reinforcement Learning Theory

Chapter Description.

40.1 section title

Section.

Theorem 40.1.1. Tested Theorem

I am the bone of my sword.

Definition 40.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 40.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 41

Variational Inference and Generative Models

Chapter Description.

41.1 section title

Section.

Theorem 41.1.1. Tested Theorem

I am the bone of my sword.

Definition 41.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 41.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 42

Connection between Inference and Control

Chapter Description.

42.1 section title

Section.

Theorem 42.1.1. Tested Theorem

I am the bone of my sword.

Definition 42.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 42.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 43

Inverse Reinforcement Learning

So far, we manually designed reward functions to define a task, but what if we may instead learn a reward function from observing an expert, and then use reinforcement learning? In this algorithm's class, we apply an approximate optimality model from what we learned in last lecture, but now learn a reward. In today's lecture, we attempt to:

- Understand the inverse reinforcement learning problem definition.
- Understand how proabbilastic models of behavior can be used to derive inverse reinforcmeent learning algorithms.
- Understand a few practical inverse reinforcement learning algorithms that one can use.

43.1 Inverse Reinforcement Learning Problem

A definition of rational behavior is a framed act of reward-maximizing strategies, such as that modeled by DPO. Where, an irrational strategy outlines a behavior that does not maximize the reward, and therefore cannot be defined over a set of scalar values. Therefore, the modeling of human decision relies on the following framework:

$$a_1, \ldots, a_T = \arg \max_{a_1,\ldots,a_T} \sum_{t=1}^{T} r(s_t, a_t)$$

or in a stochastic case,

$$\pi = \arg \max_{\pi} \mathbb{E}_{s_{t+1} \sim p(s_{t+1}|s_t,a_t), a_t \sim \pi(s_t,a_t)}[r(s_t, a_t)]$$

We optimize the above prompt to explain the data of human motions we obtained. However, because humans are not perfectly optimal (in other words, they can be irrational), we want to use soft optimality models rather than something like above.

On the other hand, learning reward functions is helpful in many ways. While in imitation learning, we provide expert demonstrations for a robot to replicate the behavior via BC. But when humans imitate other humans, we don't try to approximate $\pi(s)$. We watch something, figure out the intention behind it, and emulate the intentions rather than the actions per se. This intention is signaled by the unknown reward function, but once we learn it well, we can understand the intentions of an action and therefore act more naturally.

An inverse reinforcement learning problem, however, is underspecified: there can be multiple possible reward functions for any given set of demonstrations. What makes the demonstrations all the more incomprehensible to an algorithm is the lack of semantics that an algorithm has access to, such as what is a traffic light.

43.1.1 Forward vs. Inverse

To clarify inverse reinforcement learning, we can look at a comparison of it and forward reinforcement learning.

Forward Reinforcement Learning: Given states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, and rewards $r(s,a)$, we want to find the optimal policy π^* that maximizes an objective.

Inverse Reinforcement Learning: Given states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, and demonstrations $\tau = (s_1, a_1, \ldots, s_T, a_T)$, we want to find the reward function $r_\phi(s,a)$ that explains the demonstrations. Usually, we use some linear reward function: $r_\psi(s,a) = \sum_i \psi_i f_i(s,a) = \psi^T \vec{f}(s,a)$, or a neural network that takes s, a to output a scalar.

43.1.2 Feature Matching IRL

In classic perspectives, people have approached IRL as such: given a linear reward function, if the features $\vec{f}$ are important, what if we can match its expected values? Let π^{r_ψ} be an optimal policy for r_ψ, we can pick ψ such that $\mathbb{E}_{\pi^{r_\psi}}[\vec{f}(s,a)] = \mathbb{E}_{\pi^*}[\vec{f}(s,a)]$. However, multiple ψ vectors can still satisfy the above equation, causing ambiguity in the problem again. To specify a unique solution, we choose a maximum margin principle solution as in SVM:

$$\max_{\psi, m} m \quad \text{s.t.} \quad \psi^T \mathbb{E}_{\pi^*}[\vec{f}(s,a)] \geq \max_{\pi \in \Pi} \psi^T \mathbb{E}_{\pi}[\vec{f}(s,a)] - m$$

This however still provides ambiguity across similar continuous policies, but to counter it people use a KL divergence in place of the variable m.

The issues of this approach is that:

1. Maximizing the margin is a bit arbitrary in terms of why the learned policy knows the intention of the reward, the assumption about expert behavior is not explicit.
2. There is no clear model of expert suboptimality.
3. This is a messy constrained optimization problem for largely parametrized functions.

43.2 Learning Reward Functions

We may learn the optimality variable of a trajectory to learn the reward function, which should follow optimality. Provided that $p(\mathcal{O}_t | s_t, a_t, \psi) = \exp\big(r_\psi(s_t, a_t)\big)$, as we formulate that $p(\tau | \mathcal{O}_{1:T}, \psi) \propto p(\tau) \exp\big(\sum_t r_\psi(s_t, a_t)\big)$, when we sample from this unknown optimal policy (expert behavior), we use maximum likelihood learning to extract the optimal policy:

$$\max_\psi \frac{1}{N} \sum_{i=1}^{N} \log p(\tau_i | \mathcal{O}_{1:T}, \psi) = \max_\psi \frac{1}{N} \sum_{i=1}^{N} r_\psi(\tau_i) - \log Z$$

The log normalizer says that you cannot make all rewards larger just because they are seen trajectories. This log normalizer makes IRL difficult.

We call Z the partition Functions:

$$Z = \int p(\tau) \exp\big(r_\psi(\tau)\big)\, d\tau$$

Then we may obtain that the derivative of our objective is:

$$\nabla_\psi \mathcal{L} = \mathbb{E}_{\tau \sim \pi^*(\tau)}[\nabla_\psi r_\psi(\tau_i)] - \mathbb{E}_{\tau \sim p(\tau | \mathcal{O}_{1:T}, \psi)}[\nabla_\psi r_\psi(\tau)]$$

Our gradient is therefore the difference between expected expert policy return and expected current policy return. Therefore, we can follow the iterative approach of finding softly optimal policies, and as we sample trajectories from the current policy, we increase the reward of expert and decrease that of the current policy that is soft optimal.

In practice, to estimate the above expectation, er may first understand that the second term of our objective is equivalently:

$$\mathbb{E}_{\tau\sim p(\tau|\mathcal{O}_{1:T},\psi)}[\nabla_\psi r_\psi(\tau)] = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim p(s_t,a_t|\mathcal{O}_{1:T},\psi)}[\nabla_\psi r_\psi(s_t,a_t)]$$

Then, using last lecture, we find that $p(a_t|s_t,\mathcal{O},\psi)p(s_t|\mathcal{O},\psi) \propto \beta(s_t,a_t)\alpha(s_t)$. Let a state-action marginal $\mu_t(s_t,a_t) \propto \beta_t(s_t,a_t)\alpha_t(s_t)$, then we can estimate the expectation as $\sum_{t=1}^{T} \vec{\mu}_t^T \nabla_\psi \vec{r}_\psi$. This is quite feasible for a smaller state-space.

Then, we arrive at the Maximum Entropy IRL algorithm:

1. Given ψ, compute the backward message $\beta(s_t,a_t)$.
2. Given ψ, compute the forward message $\alpha(s_t)$.
3. Compute $\mu_t(s_t,a_t) \propto \beta_t(s_t,a_t)\alpha_t(s_t)$.
4. Evaluate $\nabla_\psi \mathcal{L}$.
5. Gradient ascent step: $\psi \leftarrow \psi + \eta\nabla_\psi\mathcal{L}$.
6. Repeat until convergence.

In the case where $r_\psi(s_t,a_t) = \psi^T \vec{f}(s_t,a_t)$, we can show that it optimizes:

$$\max_\psi \mathcal{H}(\pi^{r_\psi}) \quad \text{s.t.} \quad \mathcal{E}_{\pi^{r_\psi}}[\vec{f}] = \mathcal{E}_{\pi^*}[\vec{f}]$$

That is to say, do not make any inference or assumptions other than what is validated by expert demonstrations.

43.3 Approximated IRL in High Dimensions

Maximum Entropy IRL requires: (1) Solving for soft optimal policy in the inner loop, and (2) enumerating all state-action tuples for visitation frequency and gradient. However, it is not practical because we don't know yet how to handle: (1) large and continuous state-action spaces, (2) states obtained via sampling only, and (3) unknown dynamics; all of which are relevant in reinforcement learning.

43.3.1 Unknown Dynamics and Large State/Action Spaces

Assume that we don't know the dynamics, but we can sample it. Then, what if we try to learn $p(a_t|s_t,\mathcal{O},\psi)$ with a max-ent RL algorithm, then run that policy to sample our trajectories? While this is viable, it would require us to run the forward problem until convergence, which is difficult. So, instead of this expensive procedure, let's use lazy policy optimization. In this case, rather than learning $p(a|s,\mathcal{O},\psi)$, we improve it a little. To deal with the now biased estimator representing a wrong distribution, we can use importance sampling, with the particular design of weights per trajectory to be:

$$w_j = \frac{\exp\left(\sum_t r_\psi(s_t,a_t)\right)}{\prod_t \pi(a_t|s_t)}$$

such that each policy update concerning r_ψ will bring us closer to the target distribution.

43.4 IRLs and GANs

As we revisit the framework of IRL, we can see that it's quite like a game. Provided two players, an initial policy π and a set of human demonstrations, we change the initial policy based on a specific objective. Specifically, again, expert demonstartions are made more likely, while samples are made less likely.

This is quite like a GAN, which attempts to capture a particular data distribution. A GAN follows a two-party architecture as well: (1) A generator that takes in a random noise z to turn it into a sample x that comes from the target of capture, and (2) a discriminator that classifies "True" to all demonstration-originated data and "False" to generator-produced data. With the generator data sampled from $p_\theta(x)$ and data demonstrations from $p^*(x)$, the objective of the discriminator is to maximize the log-likelihood of data being from the demonstrators, and minimize the log-likelihood of data being from the generator. The generator, on the other hand, is trained to fool the discriminator. This is very much like the IRL procedure we framed before. Essentially, we expect $p_\theta(x) = p^*(x)$ upon convergence, allowing us to capture $p^*(x)$.

For IRL as GAN, we choose a parametrization for discriminator to be:

$$D_\psi(\tau) = \frac{\frac{1}{Z}\exp\big(r(\tau)\big)}{\prod_t \pi_\theta(a_t|s_t) + \frac{1}{Z}\exp\big(r(\tau)\big)}$$

Note that this discriminator will only be equal to 0.5 when the policy converges as noted above. We would optimize this objective with respect to ψ:

$$\psi \leftarrow \arg\max_\psi \mathbb{E}_{\tau\sim p^*}[\log D_\psi(\tau)] + \mathbb{E}_{\tau\sim\pi_\theta}[\log\big(1 - D_\psi(\tau)\big)]$$

The importance weights are subsumed into the partition function Z.

Chapter 44

Reinforcement Learning with Sequence Models

Chapter Description.

44.1 section title

Section.

Theorem 44.1.1. Tested Theorem

I am the bone of my sword.

Definition 44.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 44.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 45

Meta-Learning and Transfer Learning

Chapter Description.

45.1 section title

Section.

Theorem 45.1.1. Tested Theorem

I am the bone of my sword.

Definition 45.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 45.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 46

Challenges and Open Problems

Chapter Description.

46.1 section title

Section.

Theorem 46.1.1. Tested Theorem

I am the bone of my sword.

Definition 46.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 46.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 47

Guest Lecture: Reinforcement Learning from Human Feedback

A language model is, generally, an autoregressive model over tokens. A Language Model π maps a token sequence $X = \{x_1, \ldots, x_n\}$ to some distribution over $V : x_{n+1} \sim \pi(x_{n+1}|x_1, \ldots, x_n)$. In a reinforcement leraning paradigm, then, imagine that the token sequences per se are states, and the action of a model is a token, or sequence of tokens, that it outputs. In this guest lecture, we discuss the use of RLHF (Reinforcement Learning from Human Feedback) for language models, and many relevant works for it.

47.1 Reinforcement Learning from Human Feedback

Reinforcement Learning from Human Feedback is a procedure with four main steps:

1. Step 0: Unsupervised pre-training via tons of Internet data
2. Step 1: A supervised fine-tuning of model based on human demonstrations
3. Step 2: Fitting a reward model to human preferences.
4. Step 3: Optimize a policy to maximize learned rewards

In a reinforcement learning step, how do we reward the agent? Note that ideally, we would assign high reward to things that humans prefer, and vice versa. Therefore, the reward is not hard-coded; rather, it should be elicited from human data. This manner is slightly similar to inverse reinforcement learning. The question, then, is how the human data should be (1) elicited, (2) processed. With some considerations, the elicitation of human data comes down to the form of pairwise preference judgment, providing $\mathcal{D} = \{x^{(i)}, y_w^{(i)}, y_l^{(i)}\}$. From the pairwise preference judgments, we further elicit scores out of each responses via economic models, such as Bradley-Terry Model (and specifically that model, for the discussion of this guest lecture).

Now, we obtain a probabilistic model that can be converted into a loss function for the model,

$$\mathcal{L}_R(\phi, \mathcal{D}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))\right]$$

which we may train in negative maximum likelihood. We also call this model a "reward model". We therefore attempt to learn some model that acheives a high reward in the model. However, note that this approach can suffer from trajectories where the reward model is inaccurate. Therefore, we would require a constraint for our model to prevent it from overoptimization:

$$\max_{\pi_\theta} \mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(y|x)}[r_\phi(x, y)] - \beta\mathbb{D}_{KL}[\pi_\theta(y|x)||\pi_{ref}(y|x)]$$

Then, we optimize this objective via PPO (or other reinforcement learning algorithms, PPO choice is just conducted by the author of related paper).

47.2 Direct Preference Optimization

RLHF with PPO suffers from (1) Implementation complexity, (@) Resource requirements, and (3) Stability of training, which comes from the inter-prompt shift of reward function that gives an additional degree of freedom to the model. Direct Preference Optimization serves as an algorithm to mitigate this problem with the core principle that, by parametrizing the reward model, it is possible to extract the optimal policy for learned reward model in closed form.

How so? Note that the RLHF Objective is:

$$\max_{\pi_\theta} \mathbb{E}_{x\sim\mathcal{D},y\sim\pi_\theta(y|x)}[r_\phi(x,y)] - \beta\mathbb{D}_{KL}[\pi_\theta(y|x)||\pi_{ref}(y|x)]$$

while the closed-form optimal policy is:

$$\pi^*(y|x) = \frac{1}{Z(x)}\pi_{ref}(y|x)\exp\left(\frac{1}{\beta}r_\phi(x,y)\right)$$

where $Z(x) = \sum_y \pi_{ref}(y|x)\exp\left(\frac{1}{\beta}r_\phi(x,y)\right)$ is a normalizing constant. However, note again that we cannot immediately use the normalizing constant due to intractability of its computation. Therefore, we rearrange the closed-form optimal policy to some parametrization of a reward function:

$$r(x,y) = \beta\log\frac{\pi^*(y|x)}{\pi_{ref}(y|x)} + \beta\log Z(x)$$

such that the ratio $\frac{\pi^*(y|x)}{\pi_{ref}(y|x)}$ results in a positive value when the response of π^* triumphs thta of π_{ref}. Then, combining the above insights together, we obtain a loss function for the policy as:

$$\mathcal{L}_{DPO}(\pi_\theta;\pi_{ref}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)})\right]$$

which is a tractable loss function to optimize. Furthermore, as π_θ is normalized, we managed to remove a degree of freedom. A concern regarding expressiveness and a note to resolve it exists in the paper discussed at this section.

We can further decompose its gradient to understand the optimization process. Observe the gradient of DPO:

$$\nabla_\theta\mathcal{L}_{DPO}(\pi_\theta;\pi_{ref}) = -\beta\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\sigma(\hat{r}_\theta(x,y_l) - \hat{r}_\theta(x,y_w))\left[\nabla_\theta\log\pi(y_w|x) - \nabla_\theta\log\pi(y_l|x)\right]\right]$$

which, like policy gradient, suppresses unpreferred responses (trajectories) by suppressing the likelihood of its occurrence (and vice versa, just like policy gradient).

In methodological summary, (1) DPO removes the RL training loop from RLHF; (2) DPO is simple, stable, and computationally cheaper than PPO; (3) DPO optimizes the same objective as RLHF.

Part III

COMPSCI 294-158: Deep Unsupervised Learning

Chapter 48

Introduction to Deep Unsupervised Learning

48.1 Introduction to the Subject

Deep unsupervised learning concerns capturing rich patterns in raw data wtith deep networks in a label-free way. This involves generative models, which recreate a raw data distribution, as well as self-supervised learning, which are puzzle tasks that require semantic understanding.

The expectation of required unsupervised learning (and a massive amount of it) comes from the large number of "synapses" that a brain must use to learn (which we equate to parameters). As expected, then, foundational models need a tremendous amount of information to build a generalization of semantic understanding, "common sense". A notable model "LeCake" was proposed in this perspective, stating that a "cake" of machine learning is constructed as follows:

- Unsupervised/Predictive Learning: The body of cake, the most important part of the cake
- Supervised Learning: The icing of cake, for a generally smaller bit per sample.
- Reinforcement Learning: The cherry on top, for small bit per sample that finishes up an application.

An Ideal Intelligence that we refer to in this discipline discusses compression of dataset into a simple expression, and finding a pattern as a short description of raw data (which we call a low Kolmogorov Complexity), such as a summary vector. That is, we aim to present datasets in a compressed, efficient form. And, we also aim for optimal inference, which we know by Solomonoff Induction and demands induction via shortest code-length (with the model being referred to as a program of codes). We also what something that is extensible to optimal action making agents (AIXI). The ultimate demand, and a major assumption, follows as:

> Assume we pretrain unsupervised on distribution $\mathcal{D}_1$ and then finetune on $\mathcal{D}_2$. If $\mathcal{D}_1$ and $\mathcal{D}_2$ are related, then compressing $\mathcal{D}_2$ conditioned on $\mathcal{D}_1$ should be more efficient than compressing $\mathcal{D}_2$ directly. Therefore, pretraining accelerates learning.

Aside from theoretical interests, DUL has made powerful applications across generating novel data, conditional synthesis technology, compression of data, improving downstream tasks via pretraining, as well as being flexible building blocks.

Chapter 49

Autoregressive Models

The problems we'd like to solve with likelihood-based models involve (1) data generation, where we synthesize image, videos, speech, text; (2) data compression, where we construct efficient codes; (3) anomaly detection, where we detect out-of-distribution data. The way in which likelihood-based models resolve such problem is by estimating a distribution p_{data} from samples $x_1, \ldots, x_n \sim p_{data}(x)$. That is to say, likelihood-based models learn a distribution p that allows the computation of $p(x)$ for an arbitrary datapoint x, and sampling $x \sim p(x)$ from the learned distribution p.

The desiderata of our model is an estimated distribution of complex, high-dimensional data, such as a high-resolution image. On the other hand, we also want computational and statistical efficiency, across operations of training, generalization, sampling quality, and compression rate.

49.1 1-Dimensional Distributions

An easy model that can fit a one-dimensional distribution is a histogram. Suppose that the samples, $x_1, \ldots, x_n$, take on values in a finite set of natural numbers, $\{1, \ldots, k\}$. A model, like a histogram, effectively counts the amount of datapoint in each bin of natural numbers, and is therefore described by k nonnegative numbers that describe the frequency of each numbers. The "training" process of such model is merely counting the frequency of each number in the dataset. The inference step of such model, which is to query p_i for an arbitrary i, is simply a lookup into the array $p_1, \ldots, p_k$. On the other hand, the sampling step of the model can be performed via inverse transform sampling, which is to first come up with an empirical CDF F, then draw a uniform random number $u \in [0, 1]$ and return $i^* = \min_i\{u \leq F(i)\}$.

However, a histogram lacks generalization. Although our learned histogram describes the training data distribution well, it often has poor generalization onto non-training data. The solution towards such problems is to learn a parametrized distribution, which often generalizes better. In such context, we introduce some parametrized model $p_\theta(x)$ such that p_θ is close to p_{data}. To learn θ, we can pose an optimization problem over possible values of it:

$$\theta^* = \arg\min_\theta L(\theta, x_1, \ldots, x_n)$$

Such optimization problem must be able to:

- Work with large datasets
- Yield a θ that solves the optimization problem
- Note that the training procedure can only see the empirical distribution, but need to generalize towards unseen datapoint.

An instant solution of such approach would be maximum likelihood, where:

$$\hat{\theta}_{MLE} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} -\log p_\theta(x_i)$$

which is equivalent to minimizing KL divergence between the empirical data distribution and the model:

$$KL(\hat{p}_{data}||p_\theta) = \mathbb{E}_{x\sim\hat{p}_{data}}\left[-\log p_\theta(x)\right] - H(\hat{p}_{data})$$

Its combination with stochastic gradient descent works with large dataset and is compatible with neural networks, and is thus a popular choice with respect to use of MLE.

49.2 High-Dimensional Distributions

High-dimensional data are difficult to work with, because it has a large space of possible values. Even upon flattening (such that we reduce the problem of fitting high-dimensional distributions into fitting a one-dimensional distribution), as each image influences only one parameter (despite having a large amount of values), it doesn't provide a feasible range and also does not allow for generalization.

However, by paying attention to the fact that any multi-variable distribution can be written as a product of conditionals:

$$p_\theta(x) = \prod_{i=1}^{d} p_\theta(x_i|x_{1:i-1})$$

we can build a class of models called "autoregressive model", which relies on a sequence of existing datapoint. This allows us to fit a chain of one-dimensional distributions, but as it is conditioned on high-dimensional things, we need to think further on how to efficiently represent and learn each conditional distribution.

Here, four families of autoregressive solutions are presented in the following subsections.

Bayes' Nets. This model sparsifies the conditioning by writing the chain rule as:

$$p_\theta(x) = \prod_{i=1}^{d} p_\theta(x_i|Parents(x_i))$$

where $Parents(x_i)$ is a subset of the previous variables, and we define the relationship of variables as a directed graph of variables that are related to each other if directly connected by an edge. Then, "parent" acquires a graph-based meaning. If the data has such underlying causal structure, then this architecture can provide great inductive bias. However, the sparsification of variable relationship imposes strong assumptions on inter-variable relationship, which limits the expressiveness of such model. Furthermore, in standard form, there is a limited parameter sharing between conditional distributions.

MADE. In MADE (Masked Autoencoder for Distribution Estimation), our approach is to paraetrize conditional distributions using neural networks. This approach proceeds as a process of defining relationship between data via the use of inter-node connections. This model uses a left-to-right causal structure where each hidden layer node can be vector-valued, and each edge can be its own MLP connection. Furthermore, the model uses a masking strategy to ensure that each node can only depend on previous nodes, as well as covering inter-node connections that should not occur. However, the model is limited by finite parameter sharing, which limits its learning efficiency as well.

Causal Masked Neural Models. In this model, we parametrize conditional distributions wiht a neural network as well, but we have parameter-sharing mechanisms across conditional distributions, and also add coordinating coding to individualize conditional distributions. In this model, an output is synthesized one sample at a time via taking an input raw signal. Such manner invovles several updates from MADE:

- Every hidden layer node can be vector-valued
- Same parameters when shifting from column i to $i+1$
- Adds coordinate coding to retain positional information across a sequence.
- Full uniformization via padding.

One application of these philosophies was WaveNet, which included the receptive field via dilated convolution with exponential dilation. It also allows for better expressivity, using gated residual blocks and skip connections.

Recurrent Neural Network This model is characterized by its parameter sharing mechanism and infinite look-back opportunities. In an RNN, the probability of a sequence is given by: $p_\theta(x) = \prod_{i=1}^{d} p_\theta(x_i|x_{1:i-1})$. That is, $\log p(x) = \sum_{i=1}^{d} \log p(x_i|x_{1:i-1})$. Although the lookback is theoretically infinite, in practice, the lookback is limited by the capacity of the RNN. This is quite an expressive model, but in practice, it is not as amenable to parallelization due to dependence of past information, and cannot really propagate signals from long history. Back propagation through time can also having exploding or vanishing gradients. Note that, an RNN is in fact a very deep vausal masked model with very sparse connectivity, which uses the main diagonal to synthesize data, and therefore is also limited in expressiveness when compared to Causal Masked Model.

49.3 Deeper Dive into Causal Masked Neural Models

49.3.1 Convolutional

In PixelCNN, people have proposed a masked spatial convolution spanning on 2D images. This is a masked variant of CNN, which first imposes an autoregressive ordering on 2D images. Upon that, we must design some masking method to obey our proposed ordering. In this method, we use softmax sampling on all pixels, but the masking technique used leads to a blind spot in the receptive field that we cannot condition current outputs on. This is a limitation of the model, as it cannot condition on the entire receptive field.

49.3.2 Attention

A recurring problem for convolution was the limited receptive field, which leads to difficulties in capturing long-range dependencies. An alternative of such method would be self-attention, which provides for an unlimited recpetive field, a constant-time parameter scaling with respect to data dimension, and a parallelized computation (while RNN is not parallelized). Specifically, attention mechanisms are formalized as scaled dot products:

$$A(q,K,V) = \sum_i \frac{\exp\big(s(q,k_i)\big)}{\sum_j \exp\big(s(q,k_j)\big)} v_i, s(q,k) = \frac{qk}{\sqrt{d}}$$

Such that $A(Q,K,V) = \mathrm{softmax}(QK^T)V$. Then, a masked attention would be a self-attention mechanism that makes all non-wanted elements very negative values. Let M resemble the binary mask, then a masked attention presents itself as:

$$A(Q,K,V) = \mathrm{softmax}(QK^T - (1-M)\times 10^{10})$$

The flexibility of such softmax mechanism also provides us flexibility in the composition of M.

This self-attention mechanism is used in Transformer Blocks, which involve a Multi-Head Self-Attention (MHSA) and an MLP. Collectively,

$$h = h + \mathrm{MHSA}(\mathrm{LayerNorm}(h))$$
$$h* = h + \mathrm{MLP}(\mathrm{LayerNorm}(h))$$

49.3.3 Tokenization

Autoregressive transformers are general-purpose, modality-agnostic models that can model any complex distribution as long as the data is tokenizable. To tokenize a data is to make a discrete, such that softmax operations are feasible. The means of tokenization for each modality are slightly different from each other.

Tokenization in Language. For instance, tokenizing a sentence into words is a difficult task for the non-finite-ness of vocabularies and high possibilities of seeing out-of-distribution vocabularies. A Byte-Pair Encoding (BPE) is a method that tokenizes the sentence by groupings of characters instead, and prioritizing them by frequency, which also enables bag-of-words encoding. Then, we have 1 to 2 tokens per word, and can therefore generalize to novel combinations of characters. Although, mappings are language-specific. GPT models are examplar of such tokenization method. This model uses unsupervised pre-training followed by supervised fine-tuning, and is a 100M parameter transformer model. Finetuning a model pretrained on general language ends up outperforming models designed for each task, showcasing the generalizability of GPT-1.

Tokenizaiton in Images. We may consider each RGB encoding as a token, but this introduces too many tokens per image (particularly, this leads to quadratic scaling of attention). Therefore, we would instead use a sparse transformer, which trains a causal transformer via special hard-coded masking structures. The sparse masks are designed such that these causal masks can be performed with lower-than-quadratic scaling via hardware engineering, at the sacrifice of some pairwise interactions. This produces a SoTA on perplexity of images, but the sample quality is still low. Meanwhile, a model called iGPT creates a custom 9-bit color palette by clustering RGB pixel values with k-means clustering. This approach is more concerned with representational learning, and finds a strong 3-times reduction in sequence length from the original 24-bit color palette. Finally, a popular approach now is to use a discrete autoencoder, which can compress an image into a smaller image discretely encoded and upsampled by the decoder. This method of tokenization is lossy, provides very strong compression and reduction from its approach. Then, the tokenization of video is processed in training of a large autoregressive transformer on sequential visual data.

49.3.4 Caching

The most naive method of sampling would perhaps be, provided datapoints x_1 and x_2, we compute keys and values for such tokens, feeding it down attention layers, then obtain x_3 from softmax and attention. This process continues throughout $x_4, x_5, \ldots, x_T$. However, this also requires computation of all keys and values from prior timestep, which is why caching these key-value queries per sampling step would be helpful in the described situation.

49.4 Other Miscellaneous Topics

49.4.1 Decoder-only vs. Encoder-Decoder Models

An encoder-decoder model involves another encoder (which is a transformer) that is bidirectional, encodes all inputs, and makes the decoder become conditioned on the encoder via information-sharing mechanisms (in this case cross-attention mechanism). This is a good archtiecture choice for clear conditional distribution to model. However, in chat-style models where the inputs and outputs are both uncertain, it becomes hard to structure such archtiectures. Compared to decoder-only models, encoder-decoder models to scale similarly to it, but learn somewhat more useful representations. Meanwhile, many existing text-image and video generation model condition on features from an encoder, but similar success has not been massively replicated from decoder-only models.

Chapter 50

Flow Models

When we use a generative model, we want it to (1) have a good fit to the training data, (2) be able to evaluate the probability of an unseen datapoint occurring, (3) be able to sample from a particular learned distribution, and (4) a meaningful latent representation space. Autoregressive models, as we have introduced in last lecture, lacks a latent representation embedding space that is interpretatively meaningful. This is an edge of flow models over autoregressive models, although its methodology does not provide equally competent performances.

50.1 Foundations of Flows

The objective of a flow model is to fit a density model $p_\theta(x)$ with some continuous $x \in \mathbb{R}^n$ This fitting occurs via maximum likelihood estimation, which seeks an optimal parameter for maximizing a likelihood-associated objective. To fit a density model, we instead formulate the learned distribution as $p_\theta(x) = \sum_{i=1}^{k} \pi_i \mathcal{N}(x; \mu_i, \sigma_i^2)$, where we learn the means and variacnes of components. This is otherwise known as a Gaussian mixture, which can form complicated distributions using a weighted sum, but doesn't quite work for high-dimensional data because the sampling process of such mixtures is just picking a cluster center and adding gaussian noises, rather than looking at the mixture holistically.

Provided that mixtures are not naturall fits, we would instead train some transformation that lets limitedly expressive distributions succeed. This is the core principle of flow models. A flow model would take an original input x, process it, and generating some embedding $z = f_\theta(x)$ (where therefore $x = f_\theta^{-1}(z)$). Although, the implication of f's invertibility is that x and z must have the same dimensions, as well as an inherenet constraint on the neural network structure.

Observe that, for an invertible and differentiable f_θ (and it requires so that):

$$\begin{aligned} z &= f_\theta(x) \\ p_\theta(x)\,dx &= p_\theta(z)\,dz \\ p_\theta(x) &= p(f_\theta(x)) \left| \frac{\partial f_\theta(x)}{\partial x} \right| \end{aligned}$$

Then, we may find that:

$$\max_\theta \sum_i \log p_\theta(x) = \max_\theta \sum_i \log p_z(f_\theta(x)) + \log \left| \frac{\partial f_\theta}{\partial x} \right|$$

such that, with an expression for p_z, we can optimize the objective via SGD. Then, there are two general choices for z: either we choose some invertible function class for z, or we use an embedding space density $z \sim p_Z(z)$ (which papers call a normalizing flow) (some examples are uniform distributions, Gaussian mixtures). Under the choice of some normalizing flow, sampling can be conducted via inverse transform sampling. Notably, in the special case where

$p_Z(z)$ is $Uniform[0,1]$ and the flow is defined as a parametrized CDF, the objective we described above directly recoevers the original objective for fitting the corresponding parametrized CDF.

Can every distirbution be represented by a normalizing flow? Note that, as CDFs can convert any density into a uniform distirbution under inverse transform sampling scheme, we see that it is possible to ocnvert any smooth distribution $p(x)$ to smooth $p(z)$ via an intermediate uniform distribution.

To recap, **flows** are differentiable, invertible mappings from data (x) to noise (z). We train so that the data distribution becomes a base distribution $p(z)$, under common choices like uniform or standard normal distributions, using the MLE objective we described above. Therefore, we achieve an inference step via transformation f, and sampling via f^{-1}.

50.2 2-Dimensional Flows

In a 2-dimensional circumstance, we formulate the original data and noise as follows:

$$x_1 \to z_1 = f_{\theta_1}(x_1), x_2 \to z_2 = f_{\theta_2}(x_1, x_2)$$
$$z_1 \to x_1 = f_{\theta_1}^{-1}(z_1), z_2, x_1 \to x_2 = f_{\theta_2}^{-1}(x_1, z_2)$$

Note that the dependence on x_1 in f_{θ_2} can be arbitraily complex, and there are no invertibility requirements for so. This dependence of x_1, rather than on z_1, is an inverse autoregressive flow.

The training objective is to be rephrased as follows:

$$\begin{aligned}\log p_\theta(x_1, x_2) &= \log_{p_{Z_1}(z_1)} + \log \left| \frac{\partial z_1(x_1)}{\partial x_1} \right| + \log_{p_{Z_2}(z_2)} + \log \left| \frac{\partial x_2(x_1, x_2)}{\partial x_2} \right| \\ &= \log_{p_{Z_1}(f_{\theta_1}(x_1))} + \log \left| \frac{\partial z_1(x_1)}{\partial x_1} \right| + \log_{p_{Z_2}(f_{\theta_2}(x_1, x_2))} + \log \left| \frac{\partial x_2(x_1, x_2)}{\partial x_2} \right|\end{aligned}$$

50.3 n-Dimensional Flows

In this section, we discuss possible ways to generate n-dimensional flows.

50.3.1 Autoregressive Flows

Recall that the inference step follows from f_θ, and sampling its inverse. The sampling process of a Bayes net is similarly a flow, and if it is autoregressive, we call it an autoregressive flow:

$$\begin{aligned} x_1 &\sim p_\theta(x_1), & x_1 &= f_\theta^{-1}(z_1) \\ x_i &\sim p_\theta(x_i | x_{<i}), & x_i &= f_\theta^{-1}(z_i, x_{<i}) \end{aligned}$$

To fit autoregressive flows, we still use the objsecitve:

$$p_\theta(x) = p(f_\theta(x)) = \left| \det \frac{\partial f_\theta(x)}{\partial x} \right|$$

Note here that the computation of $x \to z$ is similar to the log-likelihood computation of an autoregressive model, while the inverse is the sampling procedure of that.

In an inverse autoregressive flow, we instead observe that:

$$\begin{aligned} z_1 &= f_\theta^{-1}(x_1), & x_1 &= f_\theta(z_1) \\ z_i &= f_\theta^{-1}(x_i; z_{<i}), & x_i &= f_\theta(z_i; z_{<i}) \end{aligned}$$

In the case of IAF, $x \to z$ is instead the sampling procedure of an autoregressive model, while $z \to x$ is the log-likelihood computation. Therefore, IAF sampling is faster. For a neater comparison, autoregressive flows have parallel training and serial sampling, while the opposite occurs for inverse autoregressive flows. Several models exploit the fast sampling of inverse autoregressive flows, such as Parallel Wavenet and IAF-VAE.

50.3.2 RealNVP-like Architectures

In this approach, our sampling process f^{-1} instead linearly transforms a small cube dz into a small parallelepiped dx:

$$p(x) = p(z)\frac{\text{vol}(\,dz)}{\text{vol}(\,dx)} = p(z)\left|\det \frac{\mathrm{d}z}{\mathrm{d}x}\right|$$

such that x is likely if it maps to a large region in the z-space. We train such model with the MLE objective as follows:

$$\arg\min_{\theta} \mathbb{E}_x[-\log p_\theta(x)] = \mathbb{E}_x\left[-\log p(f_\theta(x)) - \log\det\left|\frac{\partial f_\theta(x)}{\partial x}\right|\right]$$

But, a new requirement comes that the Jacobian determinant must be easy to calculate and differentiate for f_θ.

Because flows can be composed, we can use simplistic f to resemble a complex and much expressive distribution. That is, suppose the transformation:

$$x \to f_1 \to f_2 \to \cdots \to f_k \to z$$

Then we may observe,

$$\begin{aligned} z &= f_k \circ f_{k-1} \circ \cdots \circ f_1(x) \\ x &= f_1^{-1} \circ \cdot \circ f_k^{-1}(z) \\ \log p_\theta(x) &= \log p_\theta(z) + \sum_{i=1}^{k} \log\left|\det \frac{\partial f_i}{\partial f_{i-1}}\right| \end{aligned}$$

For the simplistic flows f_i, let us consider the following examples:

1. Affine flows (multivariate Gaussian), where $f(x) = A^{-1}(x - b)$ for paramteres A and b.
2. Element-wise flows, where $f_\theta((x_1, \ldots, x_d)) = (f_\theta(x_1), \ldots, f_\theta(x_d))$.

A NICE/RealNVP architecture, for instance, uses affine coupling layer that is both invertible and easily nonlinear upon their design, and there are no restrictions on the neural networks for such architecture.

50.3.3 Brief Introduction of Glow, Flow++, FFJORD

This subsection's approach disucsses the choice of coupling transformation. In a Bayes net, coupling dependency is defined, but choice of invertible transformation f becomes a design question. While NICE, RealNVP, IAF-VAE, and many models use affine transformations, we also find in other studies that more complex, nonlinear transformations do lead to better performances. In Flow++, for instance, The CDFs and inverse CDFs of Gaussian or Logistic mixtures are used, with some modification in neural net architecture such as self-attention. Glow, from OpenAI, instead uses 1×1 convolutions along large-scale training. Continuous time flows (FFJORD), meanwhile, allow for unrestricted architectures and guarantees invertibility as well as fast log probability computation.

50.4 Dequantization

One large assumption of flow models is the use of continuous distributions. In the case of fitting discrete distributions, although it eventually finds the associated peaks of the distribution, it is numerically problematic and not well-defined

as a fitting process. To resolve this problem of degeneracy, we would instead want to consider the probability of some discrete value as:

$$P_{model}(x) := \int_{[0,1]^D} p_{model}(x+u)\,du$$

That is to dequantize the data by adding noise to it, where the noise u is popularly drawn uniformly from $[0,1]^D$.

Chapter 51

Latent Variable Models

Chapter Description.

51.1 section title

Section.

Theorem 51.1.1. Tested Theorem

I am the bone of my sword.

Definition 51.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 51.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 52

GAN and Implicit Models

Chapter Description.

52.1 section title

Section.

Theorem 52.1.1. Tested Theorem

I am the bone of my sword.

Definition 52.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 52.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 53

Diffusion Models

Chapter Description.

53.1 section title

Section.

Theorem 53.1.1. Tested Theorem

I am the bone of my sword.

Definition 53.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 53.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 54

Self-Supervised Learning, Non-Generative Representation Learning

Chapter Description.

54.1 section title

Section.

Theorem 54.1.1. Tested Theorem

I am the bone of my sword.

Definition 54.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 54.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 55

Leage Language Models

Chapter Description.

55.1 section title

Section.

Theorem 55.1.1. Tested Theorem

I am the bone of my sword.

Definition 55.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 55.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 56

Video Generation

Chapter Description.

56.1 section title

Section.

Theorem 56.1.1. Tested Theorem

I am the bone of my sword.

Definition 56.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 56.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 57

Semi-Supervised Learning and Unsupervised Distribution Alignment

Chapter Description.

57.1 section title

Section.

Theorem 57.1.1. Tested Theorem

I am the bone of my sword.

Definition 57.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 57.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 58

Compression

Chapter Description.

58.1 section title

Section.

Theorem 58.1.1. Tested Theorem

I am the bone of my sword.

Definition 58.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 58.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 59

Multimodal Models

Chapter Description.

59.1 section title

Section.

Theorem 59.1.1. Tested Theorem

I am the bone of my sword.

Definition 59.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 59.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 60

Parallelization

Chapter Description.

60.1 section title

Section.

Theorem 60.1.1. Tested Theorem

I am the bone of my sword.

Definition 60.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 60.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 61

AI for Science (Gues Instructor)

Chapter Description.

61.1 section title

Section.

Theorem 61.1.1. Tested Theorem

I am the bone of my sword.

Definition 61.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 61.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Chapter 62

Neural Radiance Fields (Guest Instructor)

Chapter Description.

62.1 section title

Section.

Theorem 62.1.1. Tested Theorem

I am the bone of my sword.

Definition 62.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 62.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Part IV

Mu Li's Videos on YouTube

Chapter 63

Note Name

Chapter Description.

63.1 section title

Section.

Theorem 63.1.1. Tested Theorem

I am the bone of my sword.

Definition 63.1.1. Tested Theorem

Steel is my body and fire is my blood.

Example Question 63.1.1: Rules?

I have created over a thousand blades.

Unknown to death, nor known to life.

Revision Log

- August 22nd: Note is created