AI Notes

Mateusz Puto

February 2020

Contents

Contents

1 Introduction **1**
- 1.1 Goals . . . 1
- 1.2 And how to accomplish them . . . 1

2 Mathematical optimization **3**
- 2.1 Guess & check . . . 3
- 2.2 Hill climbing . . . 4
- 2.3 Gradient descent . . . 4
- 2.4 Gradient descent + step size . . . 5
- 2.5 Gradient descent + momentum . . . 6
- 2.6 Stochastic gradient descent . . . 7

3 Neural networks **9**
- 3.1 Perceptron . . . 9
- 3.2 Neurons . . . 9
- 3.3 Activation functions . . . 9
- 3.4 Ensemble . . . 9
- 3.5 Backprop . . . 9
- 3.6 Advanced topics in neural networks . . . 9

4 Symbolic AI **11**
- 4.1 Solution by search . . . 11
- 4.2 Search trees . . . 11
- 4.3 Depth and breadth first search . . . 11
- 4.4 Minimax search . . . 11
- 4.5 A* . . . 11
- 4.6 MCTS . . . 11

5 Reinforcement learning **13**
- 5.1 RL problem . . . 13
- 5.2 Multi-armed bandit . . . 13
- 5.3 Markov Decision Process . . . 13
- 5.4 Monte-Carlo learning . . . 13
- 5.5 TD learning . . . 13
- 5.6 TD(lambda) learning . . . 13

6 Games and beyond **15**
- 6.1 AlphaGo (year 2016) . . . 15
- 6.2 AlphaZero (year 2017) . . . 15
- 6.3 MuZero (year 2019) . . . 15
- 6.4 AGI . . . 15

About author

He is the author.

Preface

What will we look at in this course?

The only preliminary is basic understanding of mathematics. Many of the advanced topics will not be covered here, and that was not the goal in mind of an author. Rather than introducing as much of the material as possible, he is of the mind that one is better of understanding deeply basic concepts than having a hazy idea about whole lot of ideas but not being able to reproduce any of them. Author wishes to cover mathematical optimization, neural networks, search trees and reinforcement learning. To the beginner it may seem like not a whole much, but the expert may think that we will cover every subject badly. In the mind of the author all of these objections are valid but subjects were chosen in the pursuit of creating the most complete self contained story while presenting the most powerful existing techniques. In time of writing statistical methods dominated the landscape and symbolic ways were a bit of forgotten beast.

Who this script is for

Conventions

Chapter 1

Introduction

1.1 Goals

The goal is intuitively described as a want, wish or something we move towards. We will call the being with some goal, with accordance with RL literature, **the agent**. Now to move forward we will need to formalize some notions. In this formalization the goal will mean some numerical value which we can access in some number of points in time. We can think of this process of access as of sensing. Agents can sense their situation and goals. We will also say that we wish to get maximal possible value of this goal variable. It is the same notion as in the mathematical maximization, so:

$$g = max(v) \tag{1.1}$$

where $\boldsymbol{g}$ is the goal and $\boldsymbol{v}$ is a variable. We call g the value function. We could say for example that, goal of a being is:

$$g = max(e + \zeta c) \tag{1.2}$$

Where $\boldsymbol{e}$ is value associated with existing, $\boldsymbol{c}$ is reproduction rate and $\boldsymbol{\zeta}$ is a some constant. It is the exchange rate between e and c.

1.2 And how to accomplish them

Now we come to what we will call the central purpose of the AI discipline. That is achieving goals in the sense that we defined, so creating agents that maximize some predefined value. This is very general and powerful definition, because one could imagine defining every problem one faces in his day to day life as this kind of goal. In some deep sense AI is a study which aims at solving all existing problems. But as many solutions that seem easy at the first glance there hides immense complexity in doings of those systems. We will see that formalizing some notions may be very tricky in practice, and AI is in its core the engineering discipline.

Chapter 2

Mathematical optimization

Mathematical optimization is a huge and growing discipline, with many peculiarities and advanced use of mathematical apparatus. It existed before AI study, and has many other applications. We will look at it to the depth in which it accomplishes its uses in our systems and show the key ideas. Most people working on AI use libraries containing known, best versions of the optimiers, but understanding of some of the inner workings is key in successful application of the described techniques.

The basic idea that stands behind use of optimization techniques in AI world is that we often have a diverse set of possibilities from which we cannot clearly choose the best one without a lot of exploration. Let's think about choosing the best place to build a home, or choosing best book to read, or choosing what food we want to eat. Is such occasion we can check only few things before choosing the one that we think we will like the best, but the possibility space is huge. We could build our home near the sea, near the mountains, at the outskirts of a big city, on a stone or sandy bedrock and so on.

Mathematical optimization answers the question what is the best way to explore such spaces if they can be described as some mathematical space. Like in the example of building a house, we can represent location in a coordinate system and manually rank every space on a map that we visit. Than we would be able to use mathematical optimization in order to most efficiently explore that space.

2.1 Guess & check

The setting of a problem is as follow. We have goal variable y which we wish to maximize, we can access the variable by using our sensors. The variable y depends on variable x in some unknown fashion. We influence $\boldsymbol{x}$ by taking actions, so we are allowed to tweak $\boldsymbol{x}$.

$$max(y) = f(x) \tag{2.1}$$

Can reader find the simplest solution to solving this problem? We encourage him to spend a moment thinking of a solution method. The **Guess & check** method proposes that we will guess the value of x and record the associated values of y. Than after some time of trial we will use the x associated with highest value of $\boldsymbol{y}$.

This method given enough time can check every result of every finite space of actions and give us the best possible result. The problem with this approach is that it cannot solve optimally problems set in infinite dimensions, requires a lot of computation time and does not offer us an improvement metric.

2.2 Hill climbing

Let's think how we could guide our search so that it does not check as much of bad solutions. What kind of information we could use? Again we ask a reader to think on his own about all of the questions that will be asked going forward.

One method which could help us is better choice of a points to be checked. If the space is continuous we can use our knowledge of points lying nearby to approximate the value of point in question. We could think of this problem as climbing a hill in the fog. We look for the steepest direction which we can see and move in this direction.The Hill climbing algorithm keeps track of one currently best solution, checks the points nearby and moves in the direction of the biggest improvement, as long as any improvement can be made.

In n dimensions one should check $2 \cdot n$ directions in orthogonal way (which means at right angle but for every number of dimensions) backward and forward the number line by some vector (some displacement in space), before changing current best point. This problem seems very obvious, as number of dimension grows also the number of possibilities we must check grows enormously. Also we don't necessarily move in a best directions since it can be that the best directions lies in between the directions we checked. Finally we do not use any of so called gradient information.

2.3 Gradient descent

We've been talking a lot about moving uphill, maximizing some value, but for some not so clear reasons (probably because of ease of notation) people in optimization community usually speak of moving downhill, minimizing some value. Those ideas are exactly the same if not for the sign. If we conceptually flip the space on it's head we can change between gradient ascent and descent.

There is some information that we leave on a table doing hill climbing. Can you think about some kinds of information that are unused in this setting?

One of the problems is that we don't look at all at the difference between our previous point and current point to determine our best guess of direction that will give us the biggest improvement. We could simply assume that the change in a given direction (dimension) will stay the same, and plan our move with accordance with this assumption.

Let's look at the example in 2-dim (two dimensions):

We start at the point $p_0 = (x_0, y_0)$ and move by the vector $\vec{v} = (1, 2)$. Than we end up in the point $p_1 = (x_0 + 1, y_0 + 2)$ or in another way of description in the point $p_1 = (x_1, y_1)$. Let's now say that we improved from $y = y_0$ to $y = y_1$.

The Gradient descent algorithm tells us that our next direction of a move should be proportional to the improvement (so to $\Delta y = y_1 - y_0$) in our value function (goal) improvement and also proportional to the direction we took (so vector $\vec{v}(1, 2)$ in our example).

Our next move vector will be equal to $\vec{v} = (\Delta y \cdot 1,\ \Delta y \cdot 2)$, and in general:

$$\nabla = \Delta y \cdot \vec{v} \tag{2.2}$$

where ∇ (nabla) is symbol of gradient and $\vec{v}$ is is our previous step vector (move).

This gives us so called recursive formula (which means we can apply it over and over again) for getting better and better points:

$$p_n + 1 = p_n - \nabla \tag{2.3}$$

The biggest idea here is that we can use past information on the slope of the space that we are exploring to guide our guesses of best direction. We are benefiting from the fact that space nearby is often similar, for example if we are in the flat region we expect neighboring regions to be also flat.

This algorithm with some additions, some of which we will describe in following chapters, is the basic and most commonly used algorithm for optimization in artificial intelligence setting.

2.4 Gradient descent + step size

We've written a bit on choosing the direction to move in but we haven't touched on how to choose by what amount to move. Gradient descent methods are almost always used with the step size, which tells us how far should we jump in our next direction. The math changes as follows:

$$\nabla = \Delta y \cdot \vec{v} \tag{2.4}$$

$$p_{n+1} = p_n - step\_size \cdot \nabla \tag{2.5}$$

So we only add step size parameter in our second equation.

Now you may wonder how should we choose the step size. Is there the optimal way of doing so?

And there is no optimal way of choosing step size for every problem. This property comes to being because there are so many different problems that we may be trying to solve. This situation is common in the study of AI and you will find it very often if you

choose to continue studying it. One thing differentiating physics and AI is that there often are no known constants, like $step\_size = 0.234...$ or something like that. It also means that someone trying to train some model will be changing the constant to see if it improves the effects of learning. This is a big part of a job of an AI researcher.

There are however some ways which help guide us in a choice of step size. First of all what do you think in what range should be step size? How much should the gradient influence the answer?

Usual answer is somewhere between 0.2 to 0.000001. Can you think of additional technique of changing step size during training?

There is one technique called **learning rate scheduling**, and what it does is as we train to decrease the step size which is also often called learning rate because it influences how fast our solution changes. The big learning rate will give us fast improvement but may jump over some good places, and small learning rate will give us slower improvement but will make sure we don't miss anything. One of the methods of learning rate scheduling is to halve the step size after we've seen some n number of examples.

There exist also other methods for choosing an adaptive step size which involve using some mathematical methods for approximating functions such as Taylor series.

2.5 Gradient descent + momentum

The problem of gradient descent is that it is often not very efficient in it's progress. To find out why let's imagine very thin ridge we want to move on. To make progress we have to keep going at the bottom of the ridge, but if the gradient will be big compared to distance between walls we will have problem. What direction do you think we will move with or against the direction of a ridge?

Well, will be moving against the direction of the ridge because every time we will be overshooting and getting on the wall, in which case the best direction to move according to gradient descent is to again overshoot in the other direction. To prevent that phenomenon people introduced the momentum.

In physics the momentum is defined as:

$$R = m \cdot v \tag{2.6}$$

where $\boldsymbol{R}$ is momentum, $\boldsymbol{m}$ is mass and $\boldsymbol{v}$ is velocity.

The force is defined as:

$$F = \frac{\delta R}{\delta t} \tag{2.7}$$

Derivative (change) of momentum in time dimension. We can also say that:

$$F = \frac{\delta R}{\delta t} = \frac{\delta mv}{\delta t} = m \cdot \frac{\delta v}{\delta t} = m \cdot a \tag{2.8}$$

So force being equal to $\boldsymbol{m}$ mass multiplied by $\boldsymbol{a}$ which is acceleration.

If the time would be divided in small pieces the math for updating the momentum in n-th time step would be:

$$R_{n+1} = R_n + F \tag{2.9}$$

because the $\boldsymbol{F}$ is the change in $\boldsymbol{p}$.
There is also one other small thing. In the real world we observe friction and every movement losses its momentum to it over time. This friction we would call **alpha (α)** and it describes the percentage of momentum that is not lost in a single time step. So the math changes to:

$$R_{n+1} = \alpha \cdot R_n + F \tag{2.10}$$

This is exact equation for momentum in gradient descent if we substitute gradient vector for R and move for F, so:

$$\nabla_{n+1} = \alpha \cdot \nabla_n + \vec{v} \tag{2.11}$$

And here again alpha is a constant which we can tweak because in general there are no best solutions for it.

Thinking about it in terms of ridges, if we are in this tight junction we will add the previous momentum to the current momentum. They mostly cancel each other out because the go in opposite directions and what is left is only the movement down the ridge which we wanted in a first place.

2.6 Stochastic gradient descent

In gradient descent chapter I hidden away one important piece of information. Namely how to compute gradient. You may have thought that the way to do it is on one single example. Actually such operation is called **stochastic gradient descent**. Deriving it's name from so called stochasticity, which is mathematical way of naming random processes. In this case the name comes from randomness of single examples which may not be representative of a whole data set. But what exactly even is data-set?

Data-set is a set or collection of objects which share some similarity. For example we may talk about the data-set of pictures from the wedding, or a data-set of names of books on my shelf, or the data-set of prices of market securities.

$$D = [x_1, x_2, x_3, \ldots x_n]$$

If we were NOT to do stochastic gradient descent, we would have to compute gradient using all of the examples from the data-set for one single step. You can imagine it could be quite impractical if the data-set consists of thousands or even millions of examples.

There is also a problem with stochastic gradient descent. What would be the difference between computing gradient at a single example and the whole data-set?

Training on the whole data-set is more accurate because differences between examples will cancel out but it is also a lot more computationally expensive. So maybe we can have middle of the road type of compromise? Let's train on few elements from data-set. This is called **mini-batch** stochastic gradient descent. It will offset some of the fluctuations of our value function thus making it easier to learn.

Mini-batch stochastic gradient descent often referred to as SGD is one of the most commonly used algorithms in AI. Even though being one of the first algorithms discovered it is still heavily used for example in training of neural networks.

Chapter 3

Neural networks

3.1 Perceptron

3.2 Neurons

3.3 Activation functions

3.4 Ensemble

3.5 Backprop

3.6 Advanced topics in neural networks

Chapter 4

Symbolic AI

4.1 Solution by search

4.2 Search trees

4.3 Depth and breadth first search

4.4 Minimax search

4.5 A*

4.6 MCTS

Chapter 5

Reinforcement learning

5.1 RL problem

5.2 Multi-armed bandit

5.3 Markov Decision Process

5.4 Monte-Carlo learning

5.5 TD learning

5.6 TD(lambda) learning

Chapter 6

Games and beyond

6.1 AlphaGo (year 2016)

6.2 AlphaZero (year 2017)

6.3 MuZero (year 2019)

6.4 AGI