Speaker Info. Today’s lecture is a guest lecture by Aranyak Mehta from the Market Algorithms team at Google Research, Mountain View. The team applies algorithms, machine learning, optimization, and auction theory to problems related to Google’s marketplaces. He’s interested in game theory, online algorithms, and reinforcement learning; and has done some work on the relationship between AdWords and bipartite matching in online setting.

Background. Deep Learning has achieved a lot of success on several tasks, but our focus is on optimization work (particularly combinatorial optimization). The broad question is:

Can machine learning solve classic algorithms problems?

In order to make this question more tractable, we focus on a more specific version of this:

Can deep reinforcement learning design worst case, uniform online optimization algorithms?

Here, worst case means that the algorithm performs well on every version of the problem, not just the best case or average case. Additionally, uniform means that the algorithm’s behavior should not vary based on the size of the problem.

1 Three Problems and Algorithms

1.1 AdWords

This problem is an abstracted version of the real advertising setting at Google. Advertisers come to Google wanting to advertise based on certain keywords (e.g. a florist wants to advertise to users who are searching for "flowers in Cambridge"), and give Google a bid, or a certain dollar amount that they’re willing to pay per click. This means several advertisers are bidding on different queries, and there’s an auction where the highest bid ad might get shown. But advertisers also have a budget – how much they’re willing to pay in total.

Definition. There are \( N \) advertisers, where advertiser \( a \) has budget \( B(a) \). We assume that \( M \) search queries arrive online, and advertiser \( a \) has a bid \( v(a, q) \) for query \( q \). This can be viewed as a bipartite graph between advertisers and queries (edges here denote bid amounts), and you have to pick at most one advertiser for each query. In particular, the algorithm must allocate \( q \) to zero or one of the advertisers irrevocably, and the goal is to maximize the sum of the values over all allocations. To simplify, we can assume for now that the value is the bid.
This problem generalizes online bipartite matching: we only know the advertisers (the left-hand side vertices) a priori, and the queries (right-hand side vertices) appear one at a time. Once each vertex arrives, we see its incident edges and can match it to an advertiser, but we don’t know what might appear in the future.

**Simple solution.** Give next query to advertiser with highest bid. But this greedy allocation might prevent you from allocating future queries by exhausting some particular advertiser’s budget.

**MSVV Algorithm.** For advertiser $a$, define $\text{spent}(a)$ to be the fraction of their budget already exhausted. The MSVV algorithm allocates query $q$ to an advertiser that maximizes $\text{bid}(a, q) \ast \Psi(\text{spent}(a))$, where $\Psi(x) = 1 - e^{-(1-x)}$.

This is equivalent to the greedy solution, but where we scale each advertiser’s bid based on how much of their budget is left. This trades off on the immediate gain from the bid and the value you might get from the future by keeping the advertiser around. Turns out that this achieves the optimal competitive ratio $1 - \frac{1}{e}$ or around 0.632, as shown in [MSVV07]. Additionally, the $\Psi$ curve is the best estimate of the optimal dual variable.

A quick followup note: if the sequence that is arriving is not worst case but comes from a known distribution, then there exists follow up works which show that we can achieve competitive ratio close to 1.

### 1.2 Knapsack

**Definition.** We have a knapsack with size $B$, and items arrive over time where item $t$ has $\text{value}(t)$, $\text{size}(t)$. The algorithm must make a decision to take the item in knapsack or not, and the goal is to maximize $\sum \text{value}(t)$ such that $\sum \text{size}(t) \leq B$.

Note that you cannot take any items out (decision is irrevocable). We will consider i.i.d. case instead of worst case, where every item’s value and size is chosen i.i.d. from some known distribution $F$.

**Online Optimal: Bang-per-Buck.** Pick item $t$ if and only if $\frac{\text{value}(t)}{\text{size}(t)}$ exceeds some threshold $T^*$, chosen based on the distribution $F$. Here, again, $T^*$ is the optimal value of the dual variable of the problem.

### 1.3 Secretary Problem (Optimal Stopping)

**Definition.** Again, $N$ items arrive one by one where $t$ has $\text{value}(t)$. You know a priori the number $N$ of items that will arrive. You need to pick exactly one item, so the goal is to pick the item with highest value, i.e. to maximize the probability of picking the item with highest value.

- **Adversarial Version** [D63] – Here there is no constraint on the values, but items arrive in a random order – the values are chosen deterministically before the problem starts. So, for example, the adversary can’t see you pick 50 and then make the rest of the values 51. In this case, the worst case numbers are defined as binary values (is this the largest number seen
so far) or as percentiles of the maximum value, to bound the size of the problem. The best known algorithm is "Wait-then-Pick" which discards the first $\frac{1}{e}$ of the items and then picks the first item better than the max so far. In other words, you use the prefix of the sequence as an estimate of the maximum. The optimal ratio is $\frac{1}{e}$ in this case.

- **I.I.D Version (Gilbert Mosteller ’66)** – Here the items values are picked i.i.d. from some known distribution $F$. The best algorithm is "Decreasing Thresholds": we construct a sequence of decreasing thresholds $T(1) \geq T(2) \geq ... \geq T(n)$ and pick the first item which meets or exceeds its threshold. You always pick the last item if you get that far, so $T(n) = 0$.

Again, the algorithms here have a primal-dual interpretation.

### 1.4 Why these problems?

There are a few similarities between these problems, like:

- In all of these problems, simple formulations lead to quite concise algorithms.
- Natural ideas like greedy do not work, so that even though the algorithms are simple they’re also subtle, and the algorithms for different problems are very different from each other.
- They have a nice structure for testing input length independence.
- They have useful interpretations for industry applications (AdWords).

Also, an additional conjecture: All these problems have a natural primal-dual interpretation.

Apart from these properties, if RL can solve these problems, we can hope it may be able to solve a much larger class of problems.

### 2 Reinforcement Learning Setting

At a high level, reinforcement learning attempts to trade-off short term gains with long term gains. This seems extremely applicable to all three problems we’ve seen so far. Each of these problems has a natural MDP (Markov Decision Process) representation - at every state, you can make some decisions (e.g. take the item or not) that determines what state you go to next. This is a property of the problem and not the algorithm – additionally, the MDP doesn’t give you any hints towards the algorithm.

#### 2.1 AdWords MDP

- **States:** When query arrives, agent sees bid of each advertiser for the query (n-tuple) and fraction of budget spent for each advertiser (also a n-tuple). Here we use fraction to make the problem scale-free, although this is a little bit of a hint to the algorithm.

- **Actions:** Choice of advertiser to allocate query to, OR "Not-Allocated"
• Reward: bid of the chosen advertiser if they have budget left, 0 otherwise

• Transition: Fraction of budget spent increased by \( \text{bid/budget} \) for chosen advertiser. Then, the next query is generated and a new state vector is produced.

• Goal: learn agent’s policy function that maps vector of (value, fractional spend) pairs to index of an advertiser (or n/a)

We need deep learning here because MDP is very large; we can’t store a Q-value table at every state. We trained a 5-layer 500-neuron-per-layer network with ReLU non-linearity, which took a few hours on a standard Linux computer. In particular, we didn’t do anything special or unique in terms of the machine learning approach – it was all very standard.

2.2 Knapcksack and Secretary MDP

Let \( v(i) \) be the value of the \( i \)-th item, \( s(i) \) be the size of the \( i \)-th item, and \( \text{spent} \) be the amount of space used in the bag before receiving the \( i \)-th item. Then we have:

Knapsack state:
\[
\left( v(i), s(i), \frac{i}{n}, \text{spent} \right)
\]

Secretary state:
\[
\left( v(i), \frac{i}{n} \right)
\]

Note that this is strictly not a MDP because rewards are not Markovian - reward is only given at the end-state. It actually works as is though in this setting!

This basic version works for some settings (iid for Knapsack, iid and binary/percentile for Secretary). For truly worst case algorithms, we will need to augment the state by some "memory." In particular, RL has to learn to maintain some data structure, like a histogram of value-by-size, so that the algorithm can approximate the underlying distribution.

3 Adversarial Training Sets

Question. How do we train the RL agent to find these standard algorithms? Typical ML approach: train it on a whole bunch of distributional data and hope it generalizes. But this isn’t sufficient for the worst case scenario.

A typical TCS paper always has a concluding section that gives an Upper Bound - no algorithm can achieve competitive ratio better than our’s. This is obtained via carefully crafted distribution. In that case, Yao’s Lemma gives a bound on randomized algorithms as well.

Idea. These distributions capture all the hardness of the problems, so we should use them as training sets: as universal distributions. If these distributions are not available for the problem at end, we can find high-entropy distributions instead.
3.1 Universal Distribution for AdWords

Consider the following figure (figure 4.) from the paper [KLMS19] which shows two different examples of universal distributions for the AdWords problem.

The first case shows the “adversarial” case where the adjacency matrix of the graph is an upper triangular matrix; in this case, the randomized algorithm is optimal. In the second case randomized assignment has a very small probability of picking the “correct” assignment.

Notice that the scaling algorithm would not achieve the best result if, instead of 100 copies of each request, there were only one copy of each request. The scaling algorithm is optimal when the budget and number of requests is big compared to the number of advertisers.

A good question is whether this scaling algorithm is unique from a theoretical perspective. We do not know for sure yet. But from a primal/dual perspective, there seems to be no other solutions, so it might be unique.

3.2 Goal

The “right way” to do this would be “YaoGAN”, which is a work in progress. In Machine Learning, GAN is the right way to solve the min-max problem. We set up a zero-sum game between two agents:

- Generator $G$ that generates “hard” instances, and
- An algorithm $A$ that tries to perform well on those instances.

If you can solve this GAN, you can have some theoretical guarantee.
4 Results

4.1 AdWords

We trained the RL model only on inputs of length 100, but the learned algorithm achieves a reward that is very close to the optimal algorithm across an increasing number of ad slots. But did it actually learn the algorithm? How do we evaluate it?

The network is succinct, so it must represent a succinct logic. So we can probe the network as a black box - suppose you were in this situation, what would you have done? And what would the optimal algorithm do? Do this across many situations and see if the decisions are similar. Consider the following:

- Pretend we’re in the middle of execution for an instance, and an item has just arrived.
- Assume all advertisers have same bid of 1, and that all advertisers except \( i \) have spend of 0.5.
- Plot the probability that \( i \) wins the item against the spend of \( i \)

The following plot from [KLMS19] shows the result of that setting.

![Plot](image)

Figure 1: The algorithm learned by the agent. Each curve in Figure 1a plots the probability that advertiser \( i \) (as seen by the network) is allocated as a function of their spend when all other advertisers have spend 0.5 and all advertiser have value 1. plots the following curves. Figure 1b is obtained by averaging the curves in Figure 1a.

Now the general case:

- All advertisers except 0 have bid=1, spend=0.5.
- Plot the minimum bid 0 needs to win the item against spend(0) for learned agent and MSVV.


4.2 Achieving Truly Input-Length Independence

MSVV is a simple "policy" that can be used on inputs with different lengths, or even without knowing the length of the input sequence. How could we achieve such true input length independence from our RL algorithms?

**Idea.** Discretize the action space even further by giving to the algorithm a rough sketch of the advertiser space. So, when a new request $q$ arrives, the algorithm only sees a $100 \times 100$ state vector $s$, where $s(c, d)$ equals the fraction of advertisers with value $c$ for $q$, and $d\%$ of their budget spent. **Action space** consists of picking a $(c, d)$ pair within the set $\{1, 2, \cdots, 100\} \times \{1, 2, \cdots, 100\}$.

The environment reacts with assigning the query to a random agent within the $(c, d)$ bucket if the bucket is non-empty.

Surprisingly, after discretization of space, it still works! Can train on smaller input instances and it generalizes to larger ones.

4.3 Knapsack

Trained on i.i.d. setting where (value, size) drawn randomly from $U[0, 1] \times U[0, 1]$. Graphs clearly show agent picks those items whose value-by-size (which is not given to network) is greater than

---

Figure 7: The algorithm learned by the agent. Figure 7a plots the following curves. Fix advertiser $i$. Then all advertisers except $i$ has value 1 for the ad slot and their fractional spend is 0.5. We then let the fractional spend of bidder $i$ vary from 0 to 1 and plot the minimum value that advertiser $i$ needs to be allocated the item with probability at least 0.5. The dotted curve corresponds to the threshold given by MSVV. Figure 7b is obtained by averaging the curves for all the advertisers.
some threshold - has learned the classical algorithm. In fact, with discretization, the network achieves 96% of optimal value.

In the adversarial setting, the network itself is not enough - we need to augment it with memory, as mentioned earlier (histogram of seen value-by-size).

4.4 Secretary Problem, Binary/Percentile Setting and i.i.d. settings

The first plot shows that, similar to the optimal algorithm against the worst-case adversary, the RL algorithm learns to wait for approximately the first $N/e$ steps before picking the candidate. The second plot shows that over time, the threshold for picking a value goes down, similar to the decreasing thresholds algorithm for this problem.

5 Conclusion

Wanted to learn algorithms, and we did! Three key ideas/results:

- Architecture sweet spot: online optimization problems for symmetric problems\(^1\)
- Use of TCS-style "hard distributions" on training instances leads to robust algorithms
- Black-box analysis of learned algorithms reveals behaviors of classic algorithms

\(^{1}\)Problems that involve learning functions whose value doesn’t change under permutations of inputs.
References

