Motivation. Computer systems rely on many algorithms, such as congestion control, traffic engineering (how to spread traffic), cluster scheduling (scheduling and load balancing), adaptive video streaming (decide what resolution of video to request over network with variable network) and internet telephony (routing). There are two paradigms in algorithm design for such systems:

1. Classical paradigm (1960 – now): “one size fits all” approach, where the algorithm works pretty well in all cases.
   (a) Specify model of operating environment and low-level goals
   (b) Design one algorithm that achieves goals in (most) cases of interest

2. Emerging paradigm: “instance-specific algorithms”
   (a) Learn operating environment and low-level goals
   (b) Learn algorithms that can adapt to system/workload conditions

In the classical paradigm, the trade-off between algorithms is universality vs performance. As shown in Figure 1), classical algorithms can either achieve high performance in homogeneous environments or lower performance in heterogeneous environments. In contrast, the goal of the emerging paradigm is to build learning-augmented systems which can achieve high performance in both types of environments.

Figure 1: Trade-off between algorithms.
1 Job Scheduling

One area where learned algorithms can be useful is efficiently scheduling jobs on computing clusters. These algorithms are applied in Cluster managers (e.g. Kubernetes, Mesos, Borg) and App. frameworks (e.g. Spark, Hadoop).

The specific problem we’ll focus on is job scheduling with dependencies. Each job is represented as a directed acyclic graph (DAG) where each node is a set of parallel tasks that contains information about them (e.g. number of tasks, average task duration, CPU and memory requirements). The edges of the DAG represent data dependencies; a node in the DAG cannot start executing until its parents have finished executing.

This is an online task. As jobs arrive, the scheduler must be able to analyze their DAGs and assign runnable tasks to free executors. The metric used to assign these tasks depends on what the designer is trying to optimize (e.g. average completion time, fairness).

1.1 Obstacle: Designing optimal schedulers is intractable

There are many factors that must be considered so that optimal performance can be achieved. A few examples are:

- Job dependency structure
- Degree of parallelism
- Job sizes
- Data locality (prefer to schedule jobs where the data resides)
- Placement constraints

![Figure 2: Weakness of critical path heuristic](image)
No “one-size-fits all” algorithm is able to perform optimally under such variety of constraints. Different strategies work better in different situations. To clarify this, let’s take a look at two examples that an efficient algorithm would have to deal with.

**Exploiting DAG structure.** What is the best way to schedule the tasks in Figure 2? A reasonable heuristic would be the "critical path” heuristic: find the path in the DAG which would take the longest time to execute, and schedule the first node on that path. Repeat until all nodes have been scheduled. Unfortunately this can cause choke-points, where one task runs for a while on a single machine and there’s no other tasks to schedule at the same time.

If we follow this heuristic in Figure 2 we end up scheduling all nodes of the right-hand path before any nodes of the left-hand path. But to more efficiently exploit the structure of the DAG, it’s necessary to schedule the (1,10) task and (40,1) tasks together.

This is the first challenge. For one DAG, this problem is actually solvable. For multiple DAGs, it’s NP-hard and the best we can hope for are approximate solutions.

**Job-specific Parallelism.** The second challenge is that there are many system-specific parameters that can affect job running time and that can be hard to model. For example, more parallelism allows jobs to finish faster, but there are diminishing returns (Figure 3). Furthermore, the returns to parallelism can depend on the system and on the job. Different jobs have different inherent parallelism. Beforehand, these parameters might be unknown or modelling their effect on running time could be intractable.

This is where machine learning comes in. Rather than hand-designing a scheduling algorithm to somehow compute the inherent parallelism of jobs, can we learn it based on previous executions?

1.2 **Performance of classical heuristics**

Three classical heuristics for scheduling are FIFO, shortest-job-first, and fair scheduling. Fair scheduling attempts to give all uncompleted jobs approximately the same number of machines.

In Figure 4 we see example schedules produced by these algorithms. Red lines represent when jobs finish and each job is represented by a shade of blue/green/yellow; purple is idle time. In this example, the fair scheduler turns out to be the most efficient, finishing all jobs 100 seconds faster than the other algorithms and achieving a lower average job completion time.
The Decima scheduler, as proposed in [1], uses reinforcement learning to learn workload-specific scheduling algorithms. See Figure 5 for the high-level architecture. The input state consists of the incoming jobs and the states of the executors. The scheduling agent consists of a graph neural network and a policy network that outputs which nodes should be scheduled. The environment simulates the schedule and outputs both a new state and a reward, i.e. the negative of the number of incomplete jobs in the system at that time, which is used to update the scheduling agent parameters.

Figure 6 shows example schedules produced by Decima throughout the training process. During the first iterations it is only learning to pack things more efficiently, but later on it starts to reorder things, finishing short jobs early on as is shown by the red lines shifting to the left. The final average completion time, 89 sec, is significantly faster than the fair scheduler shown in Figure 4.

3 Decima System Design

3.1 System Specification

The input consists of a sequence of jobs arriving online, each associated with a DAG. There is a set of executors to which each job must be assigned.

The node features are:
The last feature can be used to improve performance, since it is more efficient to keep jobs running on the same executors if possible, rather than switching a job between executors, which incurs a cost for transferring job data between machines.

The executor features are simpler; it is assumed that all executors are identical. Thus, the only relevant information is the set of free executors.

3.2 Processing Job DAGs with Neural Networks

The first challenge we face is how to process job DAG information. The solution is to apply a neural network to the job DAGs by converting the graphs into vectors. A naive attempt would be to ignore the DAG structure, and simply concatenate the node vectors into a job vector. Unsurprisingly, this doesn’t work. Instead, we use a ”graph neural network”. The motivation is that we want the neural network to be able to simulate classical DAG functions that might be useful—for example, the critical path. More generally, for each node \( v \), we want the neural network to be expressive enough that it could compute functions of the subgraph of the DAG that is ”downstream” from \( v \).

Each node \( v \) of the DAG, with feature vector \( x_v \), corresponds to a node in the neural network with vector \( e_v \) defined by

\[
e_v = F(x_v; \{e_w\}_{w \in \xi(v)}; \theta).
\]

Here \( F \) is a nonlinear function with learned parameters \( \theta \), and \( \xi(v) \) is the set of children of \( v \) in the DAG. Thus, information is propagated from the leaves of the DAG all the way to the root.

For our purposes, we construct \( F \) as follows. Let \( g \) and \( f \) be nonlinear functions (in particular, small neural networks) with parameters to be learned. Then the vector corresponding to node \( v \) in
Figure 6: Schedules produced by Decima after training for 0, \ldots, 30000 iterations
the neural network is
\[ e_v = g \left( \sum_{w \in \xi(v)} f(e_w) \right) + x_v. \]

The second challenge is how to encode scheduling decisions as actions. Some options are:

- Assign all executors in one action. Problem: huge action space.
- Assign one executor per action. Problem: long action sequences, making it hard to assign credit to “good” actions.
- Assign groups of executors per action.

Decima takes the third approach. Each action has two outputs: which stage to schedule and how many executors to use.

### 3.3 Training the Scheduler: Policy Gradient Method

To train the neural network we use the policy gradient method from reinforcement learning. The neural network defines a policy \( \pi_\theta(s, a) \) with learned parameters \( \theta \), where \( s \) is an environment state and \( a \) is a possible action – for this application, a schedule of a group of executors to a job. The goal is to maximize the reward of the policy. In the following equation, \( r_t \) is the reward at time \( t \), which is the negative number of incomplete jobs for this particular application.

\[
\max J(\theta) = \mathbb{E}_{\pi_\theta} \left[ \sum_{t=0}^{T-1} r_t \right].
\]

The traditional policy gradient method consists of the following steps:

1. Collect trajectories with the policy
2. Estimate gradient of \( J(\theta) \)
3. Update parameters in direction of gradient, and repeat

Here’s the traditional gradient update:

\[ \theta \leftarrow \theta + \alpha \Delta \theta \log \pi_\theta(s_t, a_t) \left( \sum_{t'=t}^{T-1} r_{t'} - b(s_t) \right) \]

where \( \alpha \) is a step size parameter. The term \( \sum_{t'=t}^{T-1} r_{t'} \) is the return from step \( t \), and \( b(s_t) \) is the baseline expected return from state \( s_t \). Thus, \( \theta \) is moved towards actions which perform “better than average” for state \( s_t \).

However, the above gradient update doesn’t work very well in our scenario. Some problems with it are the following:
1. Early in training, learned policy is poor, so a huge backlog of jobs accumulates with a long input sequence.


The first problem has a fairly straightforward solution: start by training on short input sequences and increase the length gradually, as the system learns. The second problem is a more fundamental obstacle to applying the traditional policy gradient method to problems where the environment has a stochastic component.

### 3.4 Training the Scheduler: RL in Input-Driven Environments

**Definition 1.** An input driven markov decision process is a MDP in which dynamics are driven by a stochastic input process. That is, the new state depends on not only the previous state and the action taken, but also the state of the input process.

Consider a training step in which the policy makes an action $a_t$ at time $t$. The score for action $a_t$ is traditionally

$$
\text{score for action } a_t = (\text{return after } a_t) - (\text{baseline for } s_t) \\
= \sum_{t'=t}^{T-1} r_{t'} - b(s_t)
$$

But the return after $a_t$ might be very high if the input process is very lenient in the future; or it might be depressed if the input process sends a burst of large jobs in the near future. So there is high variance in score.

To decrease variance, the baseline should be $b(s_t, z_t, z_{t+1}, \ldots)$, the expected reward from start $s_t$ where the future input process is $z_t, z_{t+1}, \ldots$. Then

$$
\text{score for action } a_t = \sum_{t'=t}^{T-1} r_{t'} - b(s_t, z_t, z_{t+1}, \ldots).
$$
Figure 7: Decima versus hand-optimized algorithms
4  Results: Decima vs. Classical Scheduling

Decima matches hand-optimized algorithms and outperforms them during bursts with lots of jobs (see Figure 7). This is because Decima finishes short jobs faster and has a slightly different profile of number of executors assigned to short jobs, with a little more CPUs than hand-crafted algorithms.

5  Applications of the Input-Dependent Baselines

![Figure 8: The half-cheetah on floating tiles problem](image)

The idea of Input-Dependent Baselines is very general. An example application can be seen in Figure 8. The goal of this problem is to make the "half-cheetah" run over the tiles floating on the water as fast as possible, knowing that each tile has a different buoyancy. The typical solution is to have a fixed sequence of buoyancy parameters, but Input-Dependent Baselines allows us to solve this problem for a random sequence of buoyancy parameters and learn a policy that is robust to that.

6  Conclusion

The key takeaways from this Lecture were:

- Opportunity to move past one-size-fits all algorithms in system design
- Networks and systems are a challenging domain for ML and present new problems that sometimes require new techniques
- Success hinges on systematic and reliable design methodologies, such as safe training and interpretability
References