Suite

INDUSTRY-GRADE FINANCIAL ANALYTICS™

Using Reinforcement Learning to Accelerate Real-Time Risk

December 2022

Abstract

Real-time risk computation loads can be notoriously heavy. Distributing them in the cloud seems appealing. But determining the optimal distribution strategy is more challenging than it sounds. This white paper shows how a reinforcement learning agent can help.

James Baker

jbaker@suitellc.com

OVERVIEW

This white paper covers work we presented at the recent QuantMinds International 2022 conference.

One of our recent research directions has been to explore how advances in Reinforcement Learning could accelerate risk management computations.

A trading operation requires a variety of analyses to be performed in real time. Examples include P&L, hedging, stress testing, VAR and margining. These are each composed of many calculations that are dependent on a mix of ticking market data and other calculations.

An easy place to see this is in looking at the vast number of non-trivial zero curve building computations that are involved. This article shows how the required number of such computations quickly becomes vast and may not be efficiently and effectively supported even with a large amount of cloud computing power. But modern Reinforcement Learning techniques can transform the problem into something less onerous.

Although the discussion here is framed in terms of building zero rate curves, the approach is agnostic to the domain. It may be applied wherever large numbers of heavy, interdependent calculations need to be maintained in real time.

PROBLEM DOMAIN

If we consider a modern trading context – with the focus here being Rates – then we typically have a series of zero rate curves to compute. The list of curves increases with the set of currencies traded. As well as the "base" market curves, we must also build a series of shifted curves. These are used in everything from:

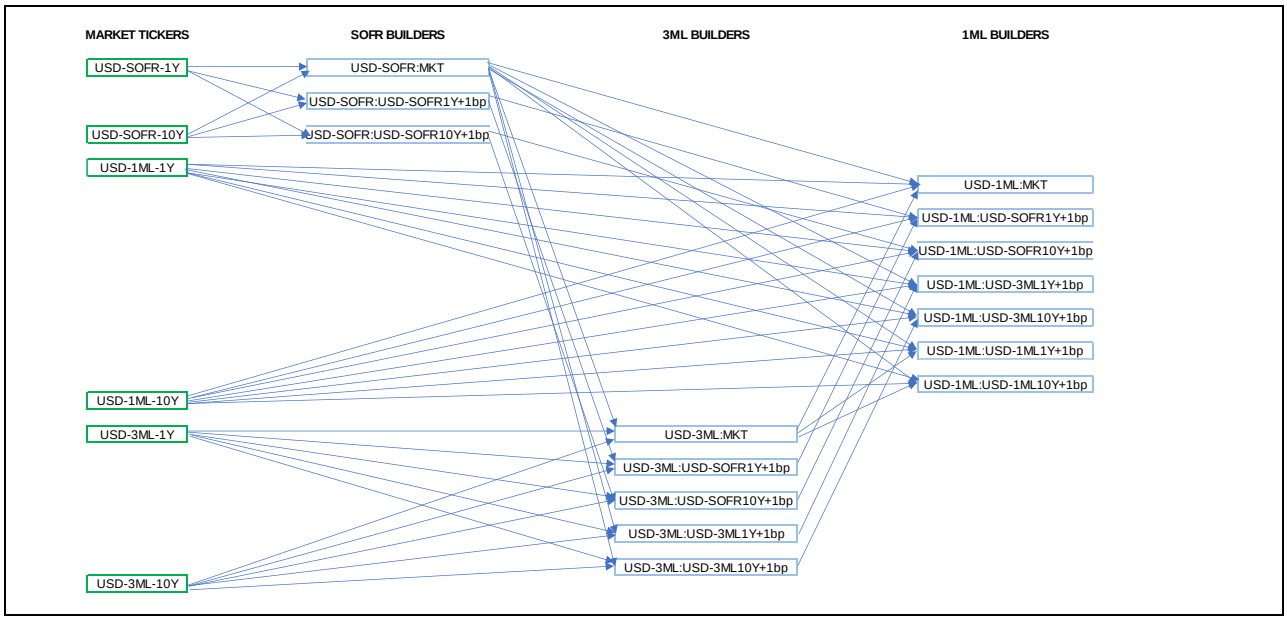
- Risk / sensitivity analysis
- Stress testing
- Historical VAR
- Margining
- CVA

All told, there will be hundreds of curves. Each curve build is computationally expensive. The market data driving those curves tick frequently – so to get accurate results, these curves must be built quickly.

At first glance, this seems simple enough, so let's consider a toy example where:

- The only analysis we need is a simple bucketed PV01
- We only have one currency: USD
- There are only three base curves: USD-SOFR, USD-3ML and USD-1ML
- To make things simpler still: there are only two tickers/market rates on each curve

Even performing only this trivial bucketed PV01 analysis on our trivial set of market data requires a fairly intricate dependency graph:



The nodes are grouped into columns representing, from left to right, Market Tickers, SOFR builders for the various scenarios, 3ML builders and 1ML builders. Not every builder is required for every scenario – often a scenario will use some curves from a base/market builder.

The builders are labeled CurveName:ScenarioName. So USD-3ML:USD-SOFR-1Y+1bp is the USD-3ML curve that is built for the scenario where the 1Y ticker on the SOFR curve is perturbed by a basis point.

While this tree could be laid out in many ways, it's clear that moving from a toy example to a real-world problem will result in a big tree with a huge number of dependencies. And between increasing the number of currencies/curves, increasing the number of tickers per curve, and increasing the set of analyses we wish to perform (hedging, stress testing, VAR), the growth in scale and complexity rapidly become significant.

Returning to our toy example: given any set of ticker changes, Alib's dependency graph determines both (1) the minimal set of recomputations required – since the tickers don't all change at the same time, not all builders need to be recomputed every time – and (2) an order for those computations to guarantee consistent computation. (e.g. if USD-SOFR-1Y changes, we cannot recompute USD-1ML:USD-3ML1Y+1bp until we have recomputed USD-SOFR:MKT and USD-3ML:USD-3ML1Y+1bp)

SCALING THINGS UP

In the real world, that ordered list contains hundreds of items, each requiring a non-trivial computation. As a result, this workload is not suited to a single processor. Execution in the cloud is often indicated.

It is possible, although far from guaranteed, that the most performant solution is to deploy more and more cloud computing power. With a one-to-one mapping of physical compute nodes to nodes in our graph, and with the right communication protocol between them, we might achieve something very fast. But such an approach is expensive and arguably wasteful (since only a fraction of the graph needs recomputing at any time). It is also not adaptive to changes in either network performance or compute node performance.

This brings us to the key question: what is the most effective strategy for distributing this workload in the cloud? What at first appears trivial, looks more complicated when we consider that:

- Not all market tickers change at the same time, so different subsets of tasks need computing at different moments
- There are dependencies between tasks
- Different compute nodes run at different speeds – and those speeds vary over time
- There are costs to serializing / communicating / deserializing – and while some of those costs are effectively fixed per call, some are linear with the number of computations. As a result, batching computations together can often be more efficient
- While all the tasks are computationally expensive, some are more expensive than others

Consequently the ideal solution needs to be complex and dynamic.

REINFORCEMENT LEARNING

If we take a step back to consider our problem, we continually:

- Receive a list of tasks to distribute
- Apply some sort of policy to schedule those tasks on a set of compute nodes in the cloud
- Observe the time taken to perform those computations, then adjust and optimize our policy
- Move on to the next batch of work

Classically, in Reinforcement Learning, an Agent observes its current State, selects an Action and submits it to the Environment. The Environment then returns a numeric Reward and moves the Agent to some new State. The Agent learns and the process continues. So perhaps our problem may usefully be formulated as a Reinforcement Learning problem.

One way to think of this is to imagine that our agent is writing a small computer program or script for us one step at a time, based on a dynamic problem. Once it has figured out a program for efficiently scheduling the tasks in the current worklist, then it runs the program, dispatching the scheduled work for execution in the cloud. It learns from the time taken to produce the results and adapts its policy to be able to come up with a more efficient strategy for similar situations in the future.

Among the first things to consider in Reinforcement Learning are the nature of the actions and of the states.

Our actions will have one of two basic forms:

- Schedule <TaskList> to Compute Node X: prepares the listed tasks for dispatching to X
- Dispatch: dispatches all scheduled tasks to the relevant compute nodes in the cloud

This means that an episode will contain a series of Schedule actions and at least one Dispatch action. If we had the most trivial example imaginable, a cloud containing two compute nodes, and one curve containing one ticker, and a PV01 analysis, our agent's program might be the following sequence of actions:

1. Schedule [USD-SOFR] to Compute Node 1
2. Schedule [USD-SOFR:USD-SOFR-1Y+1bp] to Compute Node 2
3. Dispatch

Our state is an encoded representation of the list of tasks that need to be computed plus the list of tasks already scheduled (by Actions) for each node. It also contains the current speed of each compute node. So, after the first action above has been added, then the state would become:

- Worklist: [USD-SOFR, USD-SOFR:USD-SOFR-1Y+1bp]
- WorkScheduledToComputeNode1: [USD-SOFR]
- WorkScheduledToComputeNode2: []
- ComputeNodeSpeeds: {1:x,2:y}

Where x and y are the current observed speeds of the two machines.

The problem is structured to be episodic. An episode begins with a worklist determined by the dependency graph based on the most recent ticker changes. It ends when that entire worklist has been computed, whereupon the next episode commences.

Rewards will be a small fixed positive bonus for completing the work (to encourage the agent to get the job done) plus the negative of elapsed time at every step (to encourage the agent to get it done faster).

POLICIES AND VALUES

In Reinforcement Learning, we work in terms of the following:

- Policy: $\pi(a|s)$, the probability with which we must select action a, given that we are in state s.
- State Value: $v(s)$, the expected value in terms of discounted rewards when starting in state s and following our policy to the end of the episode

The choice of policy can often be trivial. One could imagine using an argmax over the actions – hence taking the action at every step which leads us to the state which has the highest estimated state value. The limitations of this are well known and are referred to as the Exploration-Exploitation Dilemma. Effectively, if we start out with incomplete knowledge of the environment, then if we always follow our highest current estimate, we may never learn that there is a better choice, and hence always take a sub-optimal path. Conversely we could always choose our actions at random. This guarantees that we do a thorough job of exploring our environment and hence build up good estimated state values. But if we always choose our actions at random, we never take advantage of that knowledge.

The most common response to the dilemma is to us an Epsilon Greedy policy, where we choose the "best" action (from current estimates) part of the time and a random action the rest of the time. How much time is spent doing each is determined by the probability epsilon.

In practice, this does not provide a good solution to our problem. This is partly because our environment is dynamic – the relative speeds of some of the compute nodes may change, for example – so we always need to keep learning. It is also because even a small number of random actions at the wrong time can lead far from the optimal schedule.

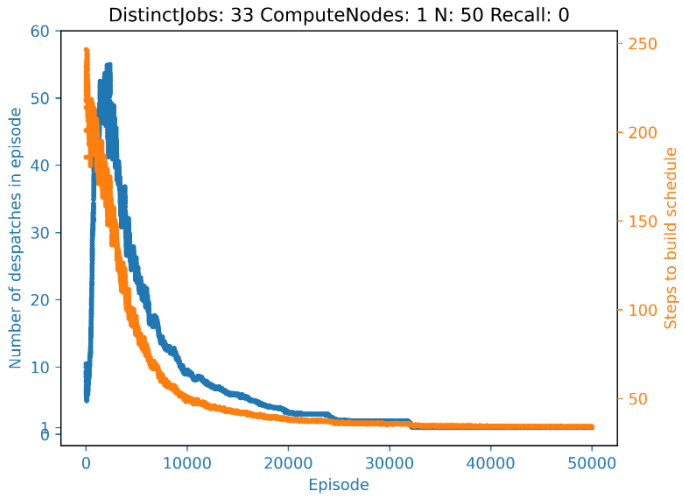
Since we can't easily know what the ideal policy is, we need an algorithm that learns both the state value function and the optimal policy function. Even with, say, 200 tasks and 5 compute nodes, actions encode to a vector of 1,000 binary elements – so there are 2^{1000} possible action configurations, very roughly 10^{300}, a truly vast number. So both our action space and state space are too large for tabular representations of these functions, so we approximate them with two neural networks.

A common family of Reinforcement Learning algorithms which meet these requirements are the Actor-Critic and its various extensions. In this case, the Actor is a function approximator for the policy and the Critic is a function approximator for the state value.

The full details of our algorithm and architecture will be presented in a forthcoming paper.

SOME INITIAL RESULTS

Applying this approach to another toy problem – 33 tasks and a single compute node – we see these results:



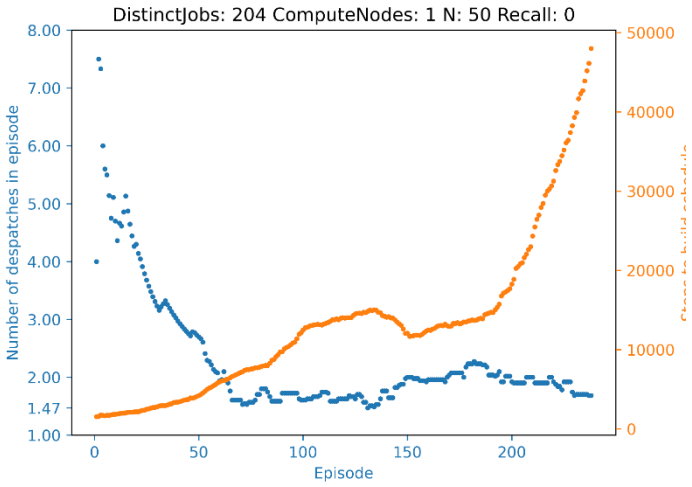
The orange graph shows, episode by episode, the number of steps taken to build schedules in order to complete the worklist. At first, the agent takes 250 steps to schedule just 33 tasks. This is because the agent knows nothing about the problem, so effectively tries random combinations of actions. Some of those actions are either duplicated or invalid and do not move the agent forward. As the episodes pass, the agent learns to build the schedule in fewer and fewer steps.

The agent has no knowledge of the dependency graph. It learns that there are relationships between tasks by trial and error. So in learning to build the schedule in this small number of steps, it is not just "learning" about scheduling but also learning these dependencies.

The blue graph shows the number of dispatches the agent performs in each episode. The agent can construct an arbitrary sequence of Schedules and Dispatches – Dispatch executes the currently scheduled tasks. So [ScheduleTask1,ScheduleTask2,Dispatch] and [ScheduleTask1,Dispatch,Scheduletask2,Dispatch] are both valid ways to compute those two tasks. In the first case the two tasks are batched together and dispatched as a single transaction. In the second case, they are treated separately. The difference between the two partly comes down to the extent to which the time taken on a Dispatch is fixed versus growing linearly with the size of the batch.

In our example, the agent determined that the fixed cost per Dispatch is large enough to justify scheduling all tasks in a single Dispatch. As a result, the values on the blue graph decrease to 1 per episode.

This is promising, but when we scale our problem to be closer to the real world we find the following:



The agent rapidly requires thousands of steps per episode to schedule the tasks. We haven't even considered what happens when there are multiple compute nodes available or when the speeds for those nodes vary.

DIGGING DEEPER

The challenge is the sparsity of the rewards. Our state and action spaces are vast – and increase dramatically as we scale our application. It's only at the end of an episode comprising potentially hundreds of steps that the agent receives material rewards. It then faces a Credit Assignment problem – although the reward was given on performing the final Dispatch action, it "belongs" to the various steps taken to get us to that point. We need to find a mechanism for sharing that reward.

To improve performance and adaptability we undergo a series of extensions where we:

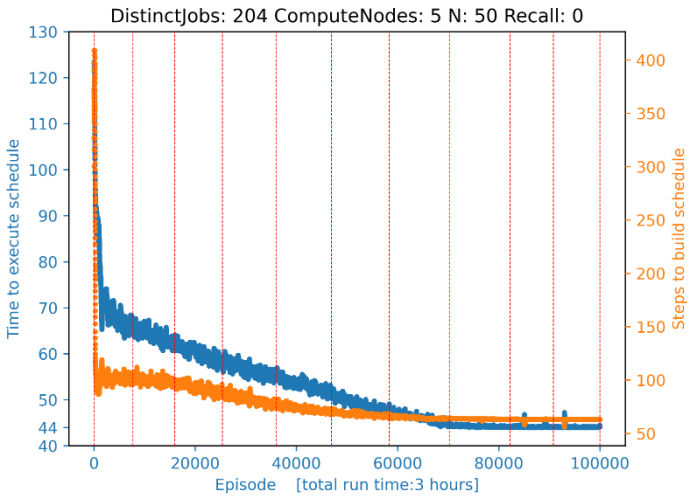
- Deepen our analysis of the reward function
- Implement a form of Eligibility Trace
- Add an element of Experience Replay in order to cope better with changing regimes
- Make some changes to the environment to make things easier for our agent

This last point sounds like cheating. But the changes are such that we are only giving the agent the benefit of things that we know about the problem. This allows our agent to focus on learning the things that we don't know about our problem – which is after all its purpose.

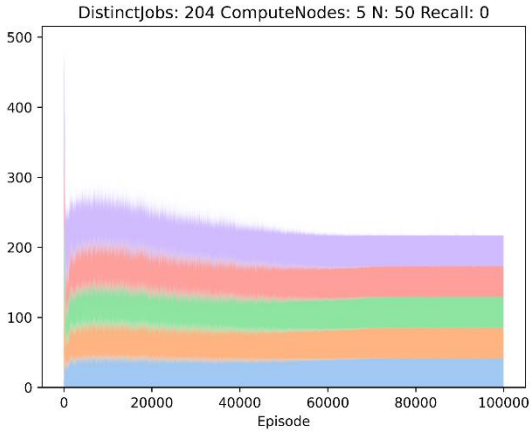
Putting this all together we arrive at the structure that we will share in our forthcoming technical paper.

TRYING IT OUT

If we run our new agent – this time with multiple compute nodes – we see the following performance.



This shows that the agent learns to solve each episode in a reliably small number of steps, resulting in a relatively fast execution of the tasks in the cloud. We also see that the agent is learning not just to minimize total work but also to keep each of the compute nodes fully utilized:



IN CONCLUSION

Our extended Actor Critic agent finds an optimal dispatching schedule to compute a set of non-trivial tasks. It achieves this in a dynamic environment where compute node performance varies over time and where random subsets of tickers update in each episode. To do this, the agent implicitly learns the dependencies between tasks and the work effort requirements for each task. Some work remains before this can be deployed in production but results so far are encouraging.

Although the framework was applied here to the building of zero rate curves, it is quite agnostic to the domain. Indeed, it may be applied wherever large numbers of heavy, interdependent calculations need to be maintained in real time.

Fuller information will be presented in our paper.