PARAMETER OPTIMIZATION AND EMERGENT BEHAVIOUR IN DEFI

Tom McLean, Marc Sabate-Vidales and David Siska
vega.xyz
WHAT WILL WE COVER?

- What is Vega Protocol?
- Case for large-scale agent based simulations
- Vega Nullchain and Vega market simulator
  - Code: Setting up the Vega market simulator
- Scenarios, Agents and Environments
  - Code: Building a basic agent
- Reinforcement Learning
  - Code: Building a smarter agent
WHAT IS VEGA PROTOCOL

github.com/vegaprotocol/vega
INTRODUCTION TO VEGA PROTOCOL

- Layer 1 blockchain, PoS, Tendermint for consensus
- Optimised for trading margined products
- Price discovery is through order books (LOBs) and auctions
- Permissionless market creation
- If there is an oracle there can be a Vega market
- Bespoke liquidity mechanism for LOBs
- Assets bridged from Ethereum
WHY RUN OWN L1?

- No fees on transactions that aren’t trades
  - Limit orders are liquidity and information - why penalise?
- Atomic closeouts
- “Bare metal” for risk computations
- Fairness: Wendy
- Latency optimisation
CASE FOR LARGE-SCALE AGENT BASED SIMULATIONS

Economy DeFi needs large agent based modelling (Nature)
DeFi protocols are becoming more complex
Simple rules can lead to complex behaviours
With complexity, we often lose the ability to thoroughly understand how a system behaves in every possible situation
There are many parameters set by governance which fine-tune protocol behaviours (Uniswap fees, Aave liquidation thresholds, Vega network and market parameters, risk parameters)
DeFi is interoperable; as protocols connect and automate complexity will increase

CASE FOR AGENT BASED SIMULATIONS

“Emergence occurs when an entity is observed to have properties its parts do not have on their own, properties or behaviors that emerge only when the parts interact in a wider whole.”

Wikipedia
“A computational model for simulating the actions and interactions of autonomous agents (both individual or collective entities such as organizations or groups) in order to understand the behavior of a system and what governs its outcomes.”

Wikipedia
TYPES OF AGENTS

- Zero intelligence: hard coded actions based on state, no optimization, no learning
- Optimizing agents: full knowledge of how environment and others work, solving control problem / game theory problems; no learning.
- Reinforcement learning agents: State, Action, Reward, State, Action - SARSA
AGENT BASED SIMULATIONS WITH VEGA MARKET SIMULATOR

https://github.com/vegaprotocol/vega-market-sim/
VEGA MARKET SIMULATOR

- Runs a full Vega stack, but with the Tendermint layer stripped away
- Replaced with a ‘null’ chain, a consensus layer which accepts whatever is sent and forwards time on command
- On top of this, an API layer allowing trading behaviour expression without (much) concern for the underlying blockchain
VEGA MARKET SIMULATOR

- Using this API, build:
  - Robust Scenarios covering a range of environments
  - Composable, configurable agents who can be slotted in or taken out at will
VEGA MARKET SIMULATOR

- We interact with a Vega instance through a ‘Service’ class, entered either in a context or with a .start() method in a notebook
  - VegaServiceNull
    - A ‘Nullchain’ instance spun up locally
  - VegaServiceNetwork
    - Connect to an existing remote network
Prerequisites:
- make
- Go 1.19
- Python 3.10
- Ideally poetry

Optional:
- For UI:
  - yarn
  - nvm
- For learning:
  - pytorch

Clone https://github.com/vegaprotocol/vega-market-sim/
Follow README.md#setup
Try at least `python -m examples.nullchain` if you skipped `make test_integration`
Once the Sim is running, we have three main routes to inspect:

- **API**
- **GraphQL**
  - Always runs, port logged on startup
  - Start VegaServiceNull with `launch_graphql=True` to automatically launch a browser
- **Console**
  - `run_with_console=True` to launch console + browser
VIEWING THE MARKET

- For a more interesting scenario run:
  - python -m vega_sim.scenario.adhoc \\ -s historic_shaped_market_maker \ 
    --console \ 
    --graphql \ 
    --pause

- GraphQL Docs:
  - https://docs.vega.xyz/docs/testnet/graphql
VEGA MARKET SIMULATOR - VEGASERVICENULL

- Core
  - Process transactions, maintains state, produces events
- Datanode
  - A storage layer allowing query of historic data from a Vega instance, consumes events
- Vegawallet (Optional)
  - Signs transactions and web interaction
- Console (Optional)
  - A frontend web GUI for Vega networks
ENvironments, Agent State, Action $\rightarrow$ State Step

- Scenario
  - Environment
    - Number of Steps
    - Vega Components
    - Logging
  - Agent 1
    - Initialisation
    - Action at each step
  - Agent 2
    - Initialisation
    - Action at each step
  - Agent N
    - Initialisation
    - Action at each step
VEGA SIM - AGENTS & SCENARIOS

- What is a Scenario?
  - Agents
    - Take actions at each
  - Environment
    - Number of steps
    - Vega instance config
    - Logging
What is an agent?

Class with three interfaces:

- **initialise**
  - Called at start of a scenario

- **step**
  - Called once each scenario step

- **finalise**
  - Called at end of a scenario

- **wait_for_total_catchup**
  - Keeps things in sync
SIMPLE AGENT: A WALKTHROUGH

Starting from a skeleton, we’ll build a basic agent
class Agent(ABC):
    def step(self, vega: VegaService):
        pass

    def initialise(self, vega: VegaService):
        self.vega = vega

    def finalise(self):
        pass

class AgentWithWallet(ABC):
    def __init__(
        self,
        wallet_name: str,
        wallet_pass: str,
        key_name: Optional[str] = None,
    ):
        """Agent for use in environments as specified in environment.py."
        To extend, the crucial function to implement is the step function which will be called on each timestep in the simulation.

        Additionally, the initialise function can be added to. This function is called once before the main simulation and can be used to create assets, set up market faucet assets to the agent etc.

        Args:
            wallet_name:
                str, The name to use for this agent's wallet
            wallet_pass:
                str, The password which this agent uses to log in to the wallet
            key_name:
                str, optional, Name of key in wallet for agent to use. Defaults to value in the environment variable "VEGA_DEFAULT_KEY_NAME".
        """
        super().__init__()
        self.wallet_name = wallet_name
        self.wallet_pass = wallet_pass
        self.key_name = key_name

    def step(self, vega: VegaService):
        pass

    def initialise(self, vega: VegaService, create_wallet: bool = True):
        super().initialise(vega=vega)
        if create_wallet:
            self.vega.create_wallet(
                name=self.wallet_name,
                passphrase=self.wallet_pass,
                key_name=self.key_name,
            )
        else:
            self.vega.login(name=self.wallet_name, passphrase=self.wallet_pass)

# A simple agent framework which you can extend with some custom logic.

As-is, this agent will faucet itself some tokens in the setup phase and then do nothing for the rest of the trading session.

Fill in your own logic into the `step` function to make them trade however you'd like.

Below, we have a range of building blocks, copy and paste these into your code to get started.

# Pull best bid/ask prices
best_bid, best_ask = self.vega.best_prices(self.market_id)

# Pull market depth (up to a specified number of levels)
market_depth = self.vega.market_depth(self.market_id, num_levels=5)
- The agent itself:
  - `vega_sim.reinforcement.agents.simple_agent`

- To run the Scenario
  - `python -m vega_sim.reinforcement.run_simple_agent`
EXISTING AGENTS

- Market makers: Ideal MM v1 and v2, Curve market maker (optimising)
- Liquidity taker (no int)
- Informed trader (no int)
- Momentum traders (no int)
Simulations, with agents performing (mostly) logical, real world actions

With a stable of agents, and some initial parameters, investigate the metrics you care about as the system evolves

Note: The agents don’t have to actually make money!
Agent testing allows the system to be evaluated far more thoroughly than can ever be done manually.

But we still have limitations:

- What initial conditions do we start from?
  - Test a range
  - Look at the real world

- What agents do we use?
  - Agents with set logic are a great starting point, but limit the range of states we investigate.
EFFECT OF RISK METRICS ON MM PROFITABILITY
BUILDING RL AGENTS
WHY DO WE WANT RL AGENTS?

- Zero intelligence and optimizing agents are “statistically” very similar even if each run is different
- Once you’ve run an environment 10-100 times you’ve seen it all
- RL agents explore and learn, stressing the system in new ways
Outline

Markov Decision Process (MDP)

Reinforcement Learning
  Q-learning
  Policy Gradient
Environment Dynamics

Finite MDP consists of:

- Finite sets of states $S$, actions $A$.
- Environment dynamics. Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space. For $a \in A$, $y, y' \in S$ we are given $p^a(y, y')$ of a discrete time Markov chain $(X_n^\alpha)_{n=0,1,...}$ so that

$$\mathbb{P}(X_{n+1}^\alpha = y' | X_n^\alpha = y) = p^a_n(y, y')$$

- $\alpha$ is the control process. $\alpha$ is measurable with respect to $\sigma(X_k^\alpha, k \leq n)$. In other words, $\alpha$ can’t look into the future.
Value Function

- Let $\gamma \in (0, 1)$ be a fixed discount factor
- Let $f : S \times A \to \mathbb{R}$ be a running reward.
- Our aim is to maximize the expected return

$$J^\alpha(x) = \mathbb{E}^x \left[ \sum_{n=0}^{\infty} \gamma^n f(\alpha_n, X_n^\alpha) \right]$$

over all controlled processes, where $\mathbb{E}^x := \mathbb{E}[\cdot | X_0^\alpha = x]$

- For all $x \in S$, we define the value function and the optimal value function as

$$v^\alpha(x) = J^\alpha(x), \quad v^*(x) := \max_{\alpha \in \mathcal{A}} J^\alpha(x)$$
Dynamic Programming for controlled Markov Processes

Theorem (DPP)
Let $f$ be bounded. Then for all $x \in S$ we have

$$v^*(x) = \max_{a \in A} E^x [f^a(x) + \gamma v^*(X^a_1)]$$

Corollary
Among all admissible control processes, it is enough to consider the ones that depend only on the current state.
Policy Iteration

Start with initial guess of $\alpha^0(x_i)$ for $i = 1, \ldots, |S|$. Let $V^k(x_i), \alpha^k(x_i)$ be defined through the iterative procedure.

1. **Evaluate** the current policy

   $$V^{k+1}(x_i) = f(x_i, \alpha^k(x_i)) + \gamma \mathbb{E} \left[ V^k(X_{1}^{\alpha^k}) | X_{0}^{\alpha^k} = x_i \right]$$

   $p^{\alpha^k(x_i)}(y, y')$ needed!

2. **Improve** the policy

   $$\alpha^{k+1}(x_i) \in \arg \max_{a \in A} f(x_i, a) + \gamma \mathbb{E} \left[ V^{k+1}(X_{1}^{a}) | X_{0}^{a} = x_i \right]$$

   $p^a(y, y')$ needed!
Outline

Markov Decision Process (MDP)

Reinforcement Learning
  Q-learning
  Policy Gradient
Remark
In policy iteration, we need to know the transition probabilities $p^a(y, y')$, $f$, and $g$! This is not the usual case. The alternative is to learn the policy from data, collected from interacting with the environment.
Q-learning

Definition (Q-function)

\[ Q^\alpha(x, a) := r(x, a) + \gamma \mathbb{E}[v^\alpha(X_1^a)] \]

\[ Q^*(x, a) := r(x, a) + \gamma \mathbb{E}[v^*(X_1^a)] \]

From DPP, we know that, \( \max_a Q^*(x, a) = v^*(x) \), therefore

\[ Q^*(x, a) = r(x, a) + \gamma \mathbb{E}[\max_{b \in A} Q^*(X_1^a, b)]. \]

Re-arranging,

\[ 0 = r(x, a) + \gamma \mathbb{E}[\max_{b \in A} Q^*(X_1^a, b)] - Q^*(x, a) \]
Q-learning Algorithm - Stochastic Approximation

Stochastic approximation arises when one wants to find the root $\theta^*$ of the following expression

$$0 = C(\theta) := \mathbb{E}_{X \sim \mu}(c(X, \theta))$$

If we have access to unbiased approximations of $C(\theta)$, namely $\tilde{C}(\theta)$, then the following updates

$$\theta \leftarrow \theta - \delta_n \tilde{C}(\theta)$$

with $\delta_n \in (0, 1)$ satisfying

$$\sum_n \delta_n = +\infty, \quad \sum_n \delta^2_n < +\infty$$

will converge to $\theta^*$

Going back to Q-learning, we want to find an unbiased approximation of

$$r(x, a) + \gamma \mathbb{E}_{b \in A}[\max_{X_1^a} Q^*(X_1^a, b)] - Q^*(x, a)$$
Q-learning Algorithm

Recall $S, A$ are finite (they can be big). Transition probabilities, running cost and final cost are unknown, but we can observe tuples $(x_n, a_n, r_n, x_{n+1})$ from interacting with the environment.

1. Make initial guess, for $Q^*(x, a)$ denoted by $Q(x, a)$ for all $x, a$.
2. We select and perform an action $a$ (either by following the current policy, or by doing some sort of exploration).
3. We select the state we landed in, denoting it by $y$. If it is not terminal, adjust

$$Q(x, a) \leftarrow Q(x, a) + \delta_n \left( r(x, a) + \gamma \max_{b \in A} Q(y, b) - Q(x, a) \right)$$

Note: we are doing Stochastic Approximation using $\max_{b \in A} Q(y, b)$ as an unbiased approximation of $\mathbb{E}^x[\max_{b \in A} Q(X_1^a, b)]$.

4. Go back to (2)
Q-learning Algorithm - Function approximation

In practice, the state space might be very large (or continuous). It is then infeasible to sample \((x_n, a, r, x_{n+1})\) to explore all the space.

Alternatively, \(Q\) can be approximated with a Neural Network with parameters \(\theta\).

The optimal policy will be defined as \(\alpha(x) = \max_{b \in A} Q_{\theta^*}(x, a)\) for some optimal parameters \(\theta^*\).

1. Initialise network’s parameters \(\theta\).
2. Sample tuples \((x_n, a_n, r_n, x_{n+1})_{n=1,...,M}\) from the environment, using some exploration-exploitation heuristics.
3. Find \(\theta^*\) that minimise the \(L_2\)-error

\[
J(\theta) = \frac{1}{2} \mathbb{E}_{x,a \sim \mu} \left( Q_{\theta}(x, a) - (r(x, a) + \gamma \mathbb{E}^x_{b \in A} \max_{b \in A} Q_{\theta}(X, b)) \right)^2
\]

where \(\mu\) is the empirical measure of the visited action-states, using gradient ascent. We use the following approximation of the gradient

\[
\nabla_{\theta} J = \mathbb{E}_{x,a \sim \mu} \left( Q_{\theta}(x, a) - (r(x, a) + \gamma \mathbb{E}^x_{b \in A} \max_{b \in A} Q_{\theta}(X, b)) \right) \nabla_{\theta} Q_{\theta}(x, a)
\]
Soft Policies

From DPP it follows that the optimal policy is a deterministic function of the state. In practice, since the environment and the running cost/reward function are unknown, we will use **soft policies**,

$$\pi : \mathcal{S} \rightarrow \mathcal{P}(A)$$

where $\mathcal{P}(A)$ is the space of probability measures on $A$. I will abuse the notation, and I will indistinctively use $\pi(\cdot|x)$ for the distribution, the probability mass function (or the density) of $\pi(x)$.

**Remark (Relationship between the value function and the Q-function)**

$$v^\pi(x) = \mathbb{E}_{A \sim \pi(\cdot|x)} Q(x, A)$$
Consider a soft (random) policy with probability mass function $\pi_{\theta}(\cdot|\mathbf{x})$ parametrised by some parameters $\theta$. Let $\rho$ be some initial state distribution. Instead of finding the optimal policy through the Q-function, we directly maximise the expected return for all $\mathbf{x} \in S$.

$$J^{\pi_{\theta}}(\theta) = \mathbb{E}_{A_n \sim \pi(\cdot|X_n)} \left[ \sum_{n=0}^{\infty} \gamma^n r(A_n, X_n^\alpha) \bigg| X_0 \sim \rho \right]$$

Assume we know an expression for $\nabla_{\theta} J^{\pi_{\theta}}$ (next slide). Then $\arg \max_{\theta} J^{\pi_{\theta}}(\theta)$ is found using gradient ascent using a learning rate $\tau$

$$\theta \leftarrow \theta + \tau \cdot \nabla_{\theta} J^{\pi_{\theta}}$$
We need to find an expression for $\nabla_\theta J^{\pi_\theta}$. This is given by The Policy Gradient Thm, Section 13.2 in [Sutton and Barto, 2018]

Theorem (Policy Gradient Theorem)

\[
\nabla_\theta J^{\pi_\theta}(\theta) \propto \sum_{x \in S} \mu(x) \sum_{a \in A} \nabla_\theta \pi_\theta(a|x) Q_{\pi_\theta}(x, a)
\]

\[
\propto \mathbb{E}_{X_n \sim \mu} \left[ \mathbb{E}_{\pi_\theta(\cdot|X_n)} \nabla_\theta \log(\pi_\theta(A_n|X_n)) Q_{\pi_\theta}(X_n, A_n) \right]
\]

where $\mu$ is the visitation measure.

We need to approximate $Q_{\pi_\theta}$!
Policy Gradient for Deterministic Policies

If we have a deterministic policy with continuous actions $\alpha_\alpha : S \rightarrow A$, then the Deterministic Policy Gradient for Reinforcement Learning with continuous actions is given by Theorem 1 in [Silver et al., 2014]

Theorem

$$\nabla_\theta J^{\alpha, \theta}(\theta) = \mathbb{E}_{X_n \sim \mu} \left[ \nabla_\theta \alpha_\theta(x) \nabla_a Q_{\alpha, \theta}(X_n, \alpha_\theta(s)) \right]$$

We need to approximate $Q_{\alpha, \theta}$
Actor-Critic type Algorithms

Policy Gradient theorems include the Q-function. In practice, one can either

- approximate it using Monte Carlo (i.e. by simulating several games starting from 
  \((x, a)\) and approximate it with the average). This is expensive and might have a
  high variance.

- Using a function approximation \(Q_\psi(x, a)\) with parameters \(\psi\). This motivates
  actor-critic algorithms:

  1. **Policy evaluation**: approximate the Q-function (the critic) using for example the
     Bellman equation.

     \[ \psi^* = \arg \max_\psi \frac{1}{2} \mathbb{E}_{x, a \sim \mu} \left( Q_\psi(x, a) - (r(x, a) + \gamma \mathbb{E}^x \nu_\psi(X)) \right)^2 \]

     where we recall that \(\nu_\psi(X) = \mathbb{E}_{a \sim \pi_\theta(\cdot | X)}[Q_\psi(X, a)]\)

  2. **Policy improvement** improve the policy (the actor) with gradient ascent using the
     Policy Gradient theorems.

Simple RL Agent - A walkthrough

- Start in `/vega_sim/reinforcement`
- run_rl_agent.py --rl-max-it 100
- run_rl_agent.py --evaluate 10

Do:
1. Collect SARSA from policy
2. Update q-function approximation (critic)
3. Update policy (actor)

While: error > threshold
Collect SARSA data from fixed policy

Policy is fixed (initially random neural network weights)
Collect SARSA data from fixed policy

```python
def _step(self, vega_state: LAMarketState) -> Action:
    # learned policy
    state = vega_state.to_array().reshape(1, -1)  # add
    state = torch.from_numpy(state).float()  # .to(self).
    with torch.no_grad():
        c = self.sample_action(state=state, sim=True)
    choice = int(c.item())
    return Action(buy=choice == 0, sell=choice == 1)

def states_to_sarsa(
    states: List[Tuple[LAMarketState, AbstractAction]],
    inventory_penalty: float = 0.0,
) -> List[Tuple[LAMarketState, AbstractAction, float, LAMarketState, AbstractAction]]:
    res = []
    for i in range(len(states) - 1):
        pres_state = states[i]
        next_state = states[i + 1]
        if next_state[0].full_balance <= 0:
            reward = -1e12
            res.append(
                (pres_state[0],
                 pres_state[1],
                 reward,
                 next_state[0] if next_state is not np.nan else np.nan,
                 next_state[1] if next_state is not np.nan else np.nan,
             )
            break
```
```python
def state(self, vega: VegaServiceNull) -> LAMarketState:
    position = self.vega.positions_by_market(self.wallet_name, self.market_id)
    position = position[0].open_volume if position else 0
    account = self.vega.party_account(
        wallet_name=self.wallet_name,
        asset_id=self.tdai_id,
        market_id=self.market_id,
    )
    book_state = self.vega.market_depth(
        self.market_id, num_levels=self.num_levels
    )
```
Improve Q-function estimate

def policy_eval(self, batch_size: int, n_epochs: int):
    toggle(self.policy_discr, to=False)
    toggle(self.q_func, to=True)
    dataloader = self.create_dataloader(batch_size=batch_size)
    pbar = tqdm(total=n_epochs)
    for epoch in range(n_epochs):
        for i, 
            (batch_state, batch_action_discr, batch_reward, batch_next_state,)
        in enumerate(dataloader):
            next_state_terminal = torch.isnan(batch_next_state).float()  # shape (batch_size, dim_state)
            batch_next_state[~next_state_terminal.eq(True)] = batch_state[~next_state_terminal.eq(True)]
            self.optimizer_q.zero_grad()
            pred = self.q_func(batch_state),
            dim=1,
            index=batch_action_discr,
        }
    with torch.no_grad():
        v = self.v_func(batch_next_state)
        target = {
            batch_reward
            + (1 - next_state_terminal.float().mean(1, keepdim=True))
            * self.discount_factor
            * v
        }
        loss = torch.pow(pred - target, 2).mean()
        loss.backward()
        self.optimizer_q.step()
        self.losses[f"q"].append(loss.item())
        # logging loss
        with open(self.logfile_pol_eval, "a") as f:
            f.write(  "{}\n\t\nepoch = self.learningIteration + n_epochs, loss.item(), self.coef_discr, self.coef_cont,"  )
        pbar.update(1)
    return 0
Improve policy

```python
def policy_improvement(self, batch_size: int, n_epochs: int):
    toggle(self.policy_discr, to=True)
    toggle(self.q_func, to=False)

dataloader = self.create_dataloader(batch_size=batch_size)

pbar = tqdm(total=n_epochs)
for epoch in range(n_epochs):
    for i, (batch_state, _, _, _) in enumerate(dataloader):
        self.optimizer_pol.zero_grad()
        d_kl = self.D_KL(batch_state).mean()
        d_kl.backward()
        # nn.utils.clip_grad_norm_(self.policy_volume,
        self.optimizer_pol.step()
        self.losses['d_kl'].append(d_kl.item())
    with open(self.logfile_pol_imp, "a") as f:
        f.write("{},{{:.4f}}\n".format(
            epoch + n_epochs * self.learingIteration, d_kl.item())
    )
pbar.update(1)
```

\[ \pi^*_{\text{MaxEnt}}(a_t|s_t) = \exp\left(\frac{1}{\alpha}(Q^*_\text{soft}(s_t,a_t) - V^*_\text{soft}(s_t))\right). \]

---

Reinforcement Learning with Deep Energy-Based Policies

Tuomas Haarnoja *1  Haoran Tang *2  Pieter Abbeel 134  Sergey Levine 1
What to expect?

Policy improvement

Q-function estimation
Evaluation

$1.6619368484134334e-05 < PnL < 2.1934163854174126e-05$
THANK YOU!

Tom McLean  
Vega Protocol

vega.xyz

tom@vegaprotoctol.io  
Twitter: @TomMcLn

Marc Sabate-Vidales  
Simtopia

simtopia.ai

marc@simtopia.ai  
Twitter: @msabvid

David Siska  
Vega Protocol

vega.xyz

david@vegaprotoctol.io  
Twitter: @dsiska

vega.xyz