Reinforcement Learning Using Polymatrix Competitive Gradient Descent

William Beard

11/27/2023

Outline

Background

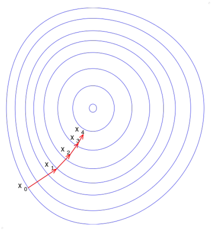
For a differentiable function F, we can use gradient descent to find local minima. We can think of F as the objective function of a single-agent game where the agent tries to minimize its objective.

Single-agent Game Formulation

Agent acts to find $\min_{x \in \mathbb{R}^m} F(x)$

Gradient Descent Update

$x_{n+1} = x_n - \gamma \nabla F(x_n)$



Background

That's great, but how do we generalize this to two-agent games? Perhaps the most natural generalization is Simultaneous Gradient Descent (SimGD), where each agent performs gradient descent on their own objective during each step.

Two-agent Game Formulation

Agents act to find $\min_{x \in \mathbb{R}^m} F(x, y)$ and $\min_{y \in \mathbb{R}^n} G(x, y)$ respectively.

SimGD Update

$$x_{n+1} = x_n - \gamma_1 \nabla_x F(x_n, y_n)$$
$$y_{n+1} = y_n - \gamma_2 \nabla_y G(x_n, y_n)$$

What does this mean intuitively?

The players ignore the fact that their opponent's strategy can change and optimize their own objective as if the opponent's strategy remains constant.

Background

As you can imagine, ignoring the fact that your opponent's strategy can change is less than ideal. What does this simplification look like in practice?

Nash Equilibria

A *Nash equilibrium* of a two-player game is a strategy pair where neither player can improve her strategy given the opponent's strategy remains constant.

A Simple Two-agent Game

Consider the game with objective functions given by

$$F(x, y) = \alpha xy = -G(x, y)$$

where $x, y \in \mathbb{R}$ and $\alpha \in \mathbb{R}^+$.

SimGD performs horribly!

Avoiding Cycling

But what if we want to avoid this cycling problem?

Enter *Competitive Gradient Descent (CGD)*

How to Beat SimGD

Traditional approaches to stabilizing SimGD lack a clear game-theoretic foundation and rely on stepsizes inversely proportional to the interactions between the players. What if we tried to approximate the underlying game and formulate our update in terms of it's Nash equilibria?

A New Update Rule

We want: $(x_{k+1}, y_{k+1}) = (x_k, y_k) + (x^*, y^*)$ where (x^*, y^*) is the Nash equilibrium of our approximated game.

A Choice of Approximation

SimGD effectively linearized the game as in $\mathbb{R}^m \times \mathbb{R}^n \longrightarrow \mathbb{R}$ but this failed to capture player interactions. But since a linear approximation works well for one agent, we choose a bilinear approximation for two agents.

A Bilinear Approximation

We take inspiration from gradient descent.

Gradient Descent Revisited

$$x_{k+1} = \text{argmin}_{x \in \mathbb{R}^m} (x^T - x_k^T) \nabla_x F(x_k) + \frac{1}{2\eta} \|x - x_k\|^2$$

The New Game

Let's formulate a new game with such a linearization structure.

$\min_{x \in \mathbb{R}^m} x^T \nabla_x F + x^T D^2_{xy} f y + y^T \nabla_y f + \frac{1}{2\eta} x^T x$

$\min_{y \in \mathbb{R}^n} y^T \nabla_y G + y^T D^2_{yx} g x + x^T \nabla_x g + \frac{1}{2\eta} y^T y$

A Unique Nash Equilibrium

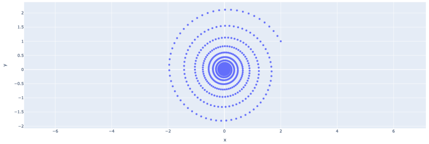
$$x = -\eta(\text{Id} - \eta^2 D^2_{xy} f D^2_{yx} g)^{-1} (\nabla_x f - \eta D^2 xy f \nabla_y g)$$
$$y = -\eta(\text{Id} - \eta^2 D^2_{yx} g D^2_{xy} f)^{-1} (\nabla_y g - \eta D^2 yx g \nabla_x f)$$

Competitive Gradient Descent

This description yields Schäfer's Competitive Gradient Descent algorithm.

Competitive Gradient Descent (CGD) Update

$$x_{n+1} = x_n - \eta(\mathrm{Id} - \eta^2 D^2_{xy} f D^2_{yx} g)^{-1}(\nabla_x f - \eta D^2 xyf \nabla_y g)$$
$$y_{n+1} = y_n - \eta(\mathrm{Id} - \eta^2 D^2_{yx} g D^2_{xy} f)^{-1}(\nabla_y g - \eta D^2 yxg \nabla_x f)$$



Great success!

Semantic Strategies

Recall our earlier game.

A Simple Two-agent Game

Consider the game with objective functions given by

$$F(x, y) = \alpha xy = -G(x, y)$$

where $x, y \in \mathbb{R}$ and $\alpha \in \mathbb{R}^+$.

What if the strategies x and y semantically represented something like time or length. We want these quantities to be non-negative, but CGD has no way of understanding or enforcing this constraint.

Constrained Two-agent Game Formulation

Agents act to find $\min_{x:\tilde{f}(x)\in\mathcal{C}} f(x, y)$ and $\min_{y:\tilde{g}(y)\in\mathcal{K}} g(x, y)$ respectively.

Adapting CGD

The most straightforward way to adapt CGD to this constrained setting is to interleave its updates with projections onto the constraint set, but this leads to the issue of *empty threats*.

Mirror Descent

We encounter empty threats because the CGD update doesn't contain any information about the global structure of the problem. To fix this, we adapt the mirror descent framework, which uses a *Bregman potential* to change our local update rule to take global geometry into account.

Bregman Divergence

We obtain the next iterate x_{k+1} as the minimizer of our approximation of the objective, regularized by the Bregman divergence $\mathbb{D}_\psi(x_{k+1}||x_k)$ for some ψ.

Number of Agents

But what if we want more than two agents?

Enter *Polymatrix Competitive Gradient Descent (PCGD)*

Another Choice of Approximation

Both CGD and CMD exploited a bilinear approximation of the underlying game when working with two agents. So why not a multilinear approximation?

A Multilinear Game

We would like an n-th order multilinear approximation of the underlying game to capture all the interactions between players, but doing so requires that solving an $(n-1)$-th order system. For $n > 2$, this is computationally intractable.

A Polymatrix Game

What if rather than trying to capture all player interactions, we only explicitly account for interactions between pairs of players. This allows us to use well-developed linear algebra tools to efficiently compute the solution.

A Polymatrix Game

What does this game look like?

Polymatrix Game Formulation

We have: $\forall i \in \{1, \ldots, n\}, \min_{\Theta^i \in \mathbb{R}^{d_i}} L^i(\Theta^1, \ldots, \Theta^n)$

The Game Hessian

$$H(\Theta) = \begin{pmatrix} \nabla_{11} L^1(\Theta) & \ldots & \nabla_{1n} L^1(\Theta) \\ \vdots & \ddots & \vdots \\ \nabla_{n1} L^n(\Theta) & \ldots & \nabla_{nn} L^n(\Theta) \end{pmatrix} \text{ where } \Theta = (\Theta^1, \ldots, \Theta^n)$$

A Unique Nash Equilibrium!

The unique Nash equilibrium for this game is given by $\Theta = -\eta(I + \eta H_o)^{-1}\zeta$, so we can apply the same update rule behind CGD to this multi-agent case.

Reinforcement Learning

Multi-agent Reinforcement Learning

Agents in an n-player MDP attempt to maximize expected reward in the form $\forall i \in \{1, \ldots, n\}, \max_{\Theta^i \in \mathbb{R}^{d_i}} J^i(\Theta^1, \ldots, \Theta^n)$, which is easily tranformed into the corresponding minimization problem.

From a big-picture perspective, reinforcement learning is a way of solving this complex optimization problem where we don't have immediate access to derivatives by generating samples and analyzing their performance.

In practice, we can abstract many of the details of the RL system by adopting an automatic differentiation framework that let's us focus on improving policies and not the minutiae of MARL.

Reinforcement Learning Is Hard

How do we know that we've implemented PCGD correctly?

We verify with classic game theory.

Matching Pennies

We test our implementation by investigating a classic game: *matching pennies*.

Matching Pennies Setup

Two players each have a penny. Each player secretly decides to flip her penny to either heads or tails. The choices are revealed simultaneously and rewarded as follows:

1. The pennies match $\longrightarrow (+1, -1)$
2. The pennies mismatch $\longrightarrow (-1, +1)$

The payoff matrix for this game is given as

$(+1, -1)$	$(-1, +1)$
$(-1, +1)$	$(+1, -1)$

Solving Matching Pennies With PCGD

As a first test of our PCGD implementation, we want to understand the behavior of our algorithm on matching pennies to establish a baseline.

Gradient Computation

During the first epoch, each policy is sampled and rewards are doled out accordingly. We find the sample:

$$A^{(1)} = A^{(2)} = \texttt{HEADS}$$

so $R^{(1)} = 1 = -R^{(2)}$. We know the expression for the gradients, that is

$$\nabla_{\theta^{(i)}} L^{(i)} = \nabla_{\theta^{(i)}} \log_{\pi^{(i)}}(A^{(i)}) \cdot R^{(i)}(A^{(1)}, A^{(2)})$$
$$\nabla_{\theta^{(i)},\theta^{(j)}} L^{(i)} = \nabla_{\theta^{(i)}} \log_{\pi^{(i)}}(A^{(i)}) \cdot \nabla_{\theta^{(j)}} \log_{\pi^{(j)}}(A^{(j)}) \cdot R^{(i)}(A^{(1)}, A^{(2)})$$

since we have no state or trajectory to process. The reward is a constant so

$$\nabla_{\theta^{(i)}} \log_{\pi^{(i)}}(A^{(i)}) = \begin{cases} \begin{pmatrix} \frac{e^{\theta_2^{(i)}}}{e^{\theta_1^{(i)}} + e^{\theta_2^{(i)}}} \\ -\frac{e^{\theta_2^{(i)}}}{e^{\theta_1^{(i)}} + e^{\theta_2^{(i)}}} \end{pmatrix}, & A^{(i)} = \texttt{HEADS} \\ \ldots \end{cases}$$

which we evaluate for both agents.

A Single Iteration For Matching Pennies

We can use this gradient computation to compute an iteration PCGD given some sample and initialization.

$$\nabla_{\theta^{(1)}} \log_{\pi^{(1)}}(A^{(1)}) = \begin{pmatrix} \frac{e^{\theta_2^{(1)}}}{e^{\theta_1^{(1)}}+e^{\theta_2^{(1)}}} \\ -\frac{e^{\theta_2^{(1)}}}{e^{\theta_1^{(1)}}+e^{\theta_2^{(1)}}} \end{pmatrix} = \begin{pmatrix} \frac{e}{e+e} \\ -\frac{e}{e+e} \end{pmatrix} = \begin{pmatrix} 1/2 \\ -1/2 \end{pmatrix}$$

Which lets us calculate the mixed products as

$$\nabla_{\theta^{(i)}} \log_{\pi^{(i)}}(A^{(i)}) \cdot \nabla_{\theta^{(j)}} \log_{\pi^{(j)}}(A^{(j)}) = \begin{pmatrix} 1/2 \\ -1/2 \end{pmatrix} \begin{pmatrix} 0.7311 \\ -0.7311 \end{pmatrix}^T = \Pi$$

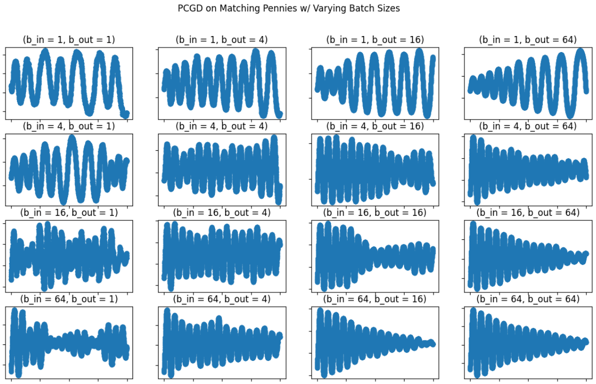
Accounting for the rewards given to each agent, we find

$$\zeta = \begin{pmatrix} 0.5 \\ -0.5 \\ -0.7311 \\ 0.7311 \end{pmatrix}, \quad I + \eta H_o = \begin{pmatrix} I & \eta\Pi \\ -\eta\Pi & I \end{pmatrix}$$

as desired. We solve for $[I + \eta H_o]^{-1}\zeta$ and apply our update to the parameters.

PCGD Batch Sizes

Here we note the importance of using large batch sizes with PCGD. The bias present in gradient samples when using reinforcement learning is difficult for the algorithm to overcome. Consider the following result.



Let's Make PCGD Fast

In the above example we explicitly constructed the matrix of second derivatives and used it as part of the update. This easily becomes the bottleneck of the algorithm's update rule.

Matrix-Free Version

Rather than computing the matrix of second derivatives explicitly, we can solve the linear system using a Krylov subspace method. This allows us to only have to evaluate Hessian-vector products, which are far cheaper than computing the Hessian itself.

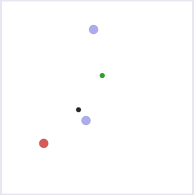
Code Sample

```
def compute_loss_mat_update_iterative(self, loss_mat, zeta):
    mv = lambda v: self.mvp(loss_mat, torch.tensor(v)).detach().numpy()
    A = LinearOperator((zeta.shape[0], zeta.shape[0]), matvec=mv)
    b = zeta.detach().numpy()
    return self.eta * torch.tensor(gmres(A, b)[0])
```

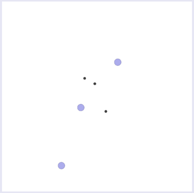
Empirical Tests

We consider a collection of environments from OpenAI for multiagent cooperative-competitive games.

Simple Adversary



Simple Spread



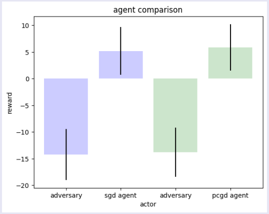
Agent Comparisons

Fair Comparisons

Note that it is hard to compare the results of two optimizers given only the reward achieved by each in a competitive environment. Since both agents may improve simultaneously, reward is a poor metric for measuring success. Instead we propose the following:

1. Select a canonical adversary from either optimizer.
2. Sample trajectories for agents trained using both optimizers from the same initialization against the canonical adversary.
3. Compare the differences in reward achieved by both agents.

Comparison Results



Future Work

While we haven't been able to show superiority of PCGD for the collection of reinforcement tasks shown here, there are still many open lines of investigation.

1. Using larger batch sizes to improve update quality.
2. Adapting to actor-critic methods to improve sample efficiency.
3. Trying more games / environments.

$\vdots$

Thanks

Thanks for listening!