https://openai.com/blog/requests-for-research-2/#cartpole

JANUARY 31, 2018 • 4 MINUTE READ

Requests for Research 2.0

We're releasing a new batch of **seven unsolved problems** which have come up in the course of our research at OpenAI. Like our original Requests for Research(which resulted in several papers), we expect these problems to be a fun and meaningful way for new people to enter the field, as well as for practitioners to hone their skills (it's also a great way to get a job at OpenAI). Many will require inventing new ideas. Please email us with questions or solutions you'd like us to publicize!

(Also, if you don't have deep learning background but want to learn to solve problems like these, please apply for our Fellowship program!)

Warmups

If you're not sure where to begin, here are some solved starter problems.

☆ Train an LSTM to solve the XOR problem: that is, given a sequence of bits, determine its parity. The LSTM should consume the sequence, one bit at a time, and then output the correct answer at the sequence's end. Test the two approaches below:

- Generate a dataset of random 100,000 binary strings of length 50. Train the LSTM; what performance do you get?
- Generate a dataset of random 100,000 binary strings, where the length of each string is independently and randomly chosen between 1 and 50. Train the LSTM. Does it succeed? What explains the difference?

☆ Implement a clone of the classic Snake game as a Gym environment, and solve it with a reinforcement learning algorithm of your choice. Tweet us videos of the agent playing. Were you able to train a policy that wins the game?

Requests for Research

☆☆ **Slitherin'.** Implement and solve a multiplayer clone of the classic Snakegame (see slither.io for inspiration) as a Gym environment.

- Environment: have a reasonably large field with multiple snakes; snakes grow when eating randomly-appearing fruit; a snake dies when colliding with another snake, itself, or the wall; and the game ends when all snakes die. Start with two snakes, and scale from there.

- Agent: solve the environment using self-play with an RL algorithm of your choice. You'll need to experiment with various approaches to overcome self-play instability (which resembles the instability people see with GANs). For example, try training your current policy against a distribution of past policies. Which approach works best?
- Inspect the learned behavior: does the agent learn to competently pursue food and avoid other snakes? Does the agent learn to attack, trap, or gang up against the competing snakes? Tweet us videos of the learned policies!

☆☆☆ **Parameter Averaging in Distributed RL.** Explore the effect of parameter averaging schemes on sample complexity and amount of communication in RL algorithms. While the simplest solution is to average the gradients from every worker on every update, you can save on communication bandwidth by independently updating workers and then infrequently averaging parameters. In RL, this may have another benefit: at any given time we'll have agents with different parameters, which could lead to better exploration behavior. Another possibility is use algorithms like EASGD that bring parameters partly together each update.

☆☆☆ **Transfer Learning Between Different Games via Generative Models.**Proceed as follows:

- Train 11 good policies for 11 Atari games. Generate 10,000 trajectories of 1,000 steps each from the policy for each game.
- Fit a generative model (such as the Transformer) to the trajectories produced by 10 of the games.
- Then fine-tune that model on the 11th game.
- Your goal is to quantify the benefit from pre-training on the 10 games. How large does the model need to be for the pre-training to be useful? How does the size of the effect change when the amount of data from the 11th game is reduced by 10x? By 100x?

☆☆☆ **Transformers with Linear Attention.** The Transformer model uses soft attention with softmax. If we could instead use linear attention (which can be converted into an RNN that uses fast weights), we could use the resulting model for RL. Specifically, an RL rollout with a transformer over a huge context would be impractical, but running an RNN with fast weights would be very feasible. Your goal: take any language modeling task; train a transformer; then find a way to get the same bits per character/word using a linear-attention transformer with different hyperparameters, without increasing the total number of parameters by much. Only one caveat: this may turn out to be impossible. But one potentially helpful hint: it is likely that transformers with linear attention require much higher dimensional key/value vectors compared to attention that uses the softmax, which can be done without significantly increasing the number of parameters.

☆☆☆ **Learned Data Augmentation.** You could use a learned VAE of data, to perform "learned data augmentation". One would first train a VAE on input data, then each training point would be transformed by encoding to a latent space, then applying a simple (e.g. Gaussian) perturbation in latent space, then decoding back to observed space. Could we use such an approach to obtain improved generalization? A potential benefit of such data augmentation is that it could include many nonlinear transformations like viewpoint changes and changes in scene lightning. Can we approximate the set of transformations to which the label is invariant? Check out the existing work on this topic if you want a place to get started.

☆☆☆☆ **Regularization in Reinforcement Learning.** Experimentally investigate (and qualitatively explain) the effect of different regularization methods on an RL algorithm of choice. In supervised deep learning, regularization is extremely important for improving optimization and for preventing overfitting, with very successful methods like dropout, batch normalization, and L2 regularization. However, people haven't benefited from regularization with reinforcement learning algorithms such as policy gradients and Q-learning. Incidentally, people generally use much smaller models in RL than in supervised learning, as large models perform worse — perhaps because they overfit to recent experience. To get started, here is a relevant but older theoretical study.

☆☆☆☆☆ **Automated Solutions of Olympiad Inequality Problems.**Olympiad inequality problems are simple to express, but solving them often requires clever manipulations. Build a dataset of olympiad inequality problems and write a program that can solve a large fraction of them. It's not clear whether machine learning will be useful here, but you could potentially use a learned policy to reduce the branching factor.

Want to work on problems like these professionally? Apply to OpenAI!