INTRODUCTION TO EVOLUTION STRATEGY ALGORITHMS

James Gleeson
Eric Langlois
William Saunders
**REINFORCEMENT LEARNING CHALLENGES**

- **Credit assignment problem**
  - Bob got a great bonus this year!
  - ...what did Bob do to earn his bonus?
  - Time horizon: 1 year
  - [+] Met all his deadlines
  - [+] Took an ML course 3 years ago
  - Sparse reward signal

- **IDEA:**
  - Lets just treat $f$ like a black-box function when optimizing it.
    - “Try different $\theta$’s, and see what works.”
  - If we find good $\theta$’s, keep them, discard the bad ones.
  - Recombine $\theta_1$ and $\theta_2$ to form a new (possibly better) $\theta_3$

- **$f(\theta)$ is a discrete function of theta...**
  - How do we get a gradient $\nabla \theta f$?

- **Local minima**

- **Backprop**
**EVOLUTION STRATEGY ALGORITHMS**

- **The template:**
  
  **“Sample” new generation**
  Generate some parameter vectors for your neural networks.

  **Fitness**
  Evaluate how well each neural network performs on a **training set**.

  **“Prepare” to sample the new generation:**
  Given how well each “mutant” performed...

  Natural selection!  ➔ Keep the good ones

  The ones that remain “recombine” to form the next generation.

**MNIST ConvNet parameters**

\[ \theta_1, \theta_2, \theta_3 \]
Rastrigin function
Test function

Lots of local optima; will be difficult to optimize with Backprop + SGD!
SCARY “TEST FUNCTIONS” (2)

Schaffer function
WHAT WE WANT TO DO; “TRY DIFFERENT θ “

Algorithm: CMA-ES
CMA-ES; HIGH-LEVEL OVERVIEW

Step 1: Calculate fitness of current generation $g(1)$

Step 2: Natural selection!
Keep the top 25%. (purple dots)

Discrepancy between mean of previous generation and top 25% will cast a wider net!

Step 3: Recombine to form the new generation:

Generate mutants: $g(i) + \epsilon$
$\epsilon \sim N(\mu_{i+1}, \Sigma_{i+1})$
**IDEA:**
Sample neural-network parameters from a multi-variate gaussian with **diagonal covariance** matrix. Update $N(\theta=[\mu, \Sigma])$ parameters using REINFORCE gradient estimate.

$$
\mathbb{J}(\theta) = E_\theta[F(z)] = \int F(z)p(z|\theta)dz
$$

$$
\nabla_\theta \mathbb{J}(\theta) = \nabla_\theta E_\theta[F(\theta)] = E_\theta[F(z)\nabla_\theta \log p(z|\theta)]
\approx \sum_{i=1}^{N} F(z_i)\nabla_\theta \log p(z_i|\theta)
$$

$$
\theta \rightarrow \theta + \eta \nabla_\theta \mathbb{J}(\theta)
$$

Adaptive $\sigma$ and $\mu$
IDEA:
Just use the same $\sigma$ and $\mu$ for each parameter.

\[ \text{Sample neural-network parameters from "isotropic gaussian" } \]
\[ = N(\mu, \sigma^2 I) \]

Seems suspiciously simple... but it can compete!

OpenAI ES paper:
- $\sigma$ is a hyperparameter
- 1 set of hyperparameters for Atari
- 1 set of hyperparameters for Mujoco
- Competes with A3C and TRPO performance
EVOLUTION STRATEGIES AS A SCALABLE ALTERNATIVE TO REINFORCEMENT LEARNING

James Gleeson
Eric Langlois
William Saunders
**TODAY’S RL LANDSCAPE AND RECENT SUCCESS**

- **Discrete action tasks:**
  - Learning to play Atari from raw pixels
  - Expert-level go player

- **Continuous action tasks:**
  - “Hopping” locomotion

**Q-learning:**

Learn the action-value function: \( Q(s,a) \)

Approximate the function using a neural-network, train it using gradients computed via backpropagation (i.e. the chain rule)

**Policy gradient; e.g. TRPO:**

Learn the policy directly: \( \pi(a|s,\theta) \)
Backpropagation isn’t perfect:
- GPU memory requirements
- Difficult to parallelize
- Cannot apply directly to non-differentiable functions
  - e.g. discrete functions $F(\theta)$ (the topic of this course)
- Exploding gradient (e.g. for RNN’s)

You have a datacenter, and cycles to spend
AN ALTERNATIVE TO BACKPROPAGATION: EVOLUTION STRATEGY (ES)

Claim:

$$\frac{\partial}{\partial \theta} F(\theta) \approx \frac{1}{\sigma^2} E_{\epsilon \sim N(0, \sigma^2)} [\epsilon F'(\theta + \epsilon)]$$

Proof:

$$F(\theta + \epsilon) \approx f(\theta) + f'(\theta) + f''(\theta)\epsilon^2/2$$

$$E_{\epsilon}[\epsilon F'(\theta + \epsilon)] \approx E_{\epsilon}[\epsilon F'(\theta) + \epsilon^2 F''(\theta) + \epsilon^3 F'''(\theta)/2]$$


$$\epsilon \sim N(0, \sigma^2)$$

$$= \mu$$

$$= 0$$

$$= E_{\epsilon}[(\epsilon - \mu)^2] = \sigma^2$$

$$E_{\epsilon}[(\epsilon - \mu)^i] = 0$$

for \(N(\mu, \sigma^2)\) and odd \(i\)

$$E_{\epsilon}[\epsilon F'(\theta + \epsilon)] = \sigma^2 F''(\theta)$$

$$\frac{\partial}{\partial \theta} F(\theta) = \frac{1}{\sigma^2} E_{\epsilon}[\epsilon F(\theta + \epsilon)]$$

Gradient of objective \(F(\theta)\)

No derivates of \(F(\theta)\)

No chain rule / backprop required!

And have it be embarrassingly parallel?

Relevant to our course:

\(F(\theta)\) could be a discrete function of \(\theta\)

2\textsuperscript{nd} order Taylor series approximation

\(F(\theta)\) independent of \(\epsilon\)
THE MAIN CONTRIBUTION OF THIS PAPER

- Criticisms:
  - Evolution strategy aren’t new!
  - **Common sense:** The variance/bias of this gradient estimator will be too high, making the algorithm unstable on today’s problems!

- This paper aims to **refute your common sense**:
  - Comparison against state-of-the-art RL algorithms:
    - **Atari**: Half the games do better than a recent algorithm (A3C), half the games do worse
    - **Mujoco**: Can match state-of-the-art policy gradients on continuous action tasks.

Linear speedups with more compute nodes: 1 day with A3C → 1 hour with ES

\[
\frac{\partial}{\partial \theta} F(\theta + \epsilon) \approx \frac{1}{\sigma^2} E_{\epsilon \sim N(0, \sigma^2)} [\epsilon F(\theta + \epsilon)]
\]
Gradient estimator needed for updating $\theta$:

$$\frac{\partial}{\partial \theta} F(\theta + \epsilon) \approx \frac{1}{\sigma^2} E_{\epsilon \sim N(0, \sigma^2)} [\epsilon F(\theta + \epsilon)]$$

Sample:

$$\frac{1}{\sigma^2} \epsilon F(\theta + \epsilon)$$

In RL, the fitness $F(\theta)$ is defined as:

$$F(\theta) = E_{\tau}[R_{\tau}]$$

Where:

- $\tau$ = An episode of state (s) action (a) pairs
- $R_{\tau}$ = Sum of rewards received over episode $\tau$

Embarassingly parallel!

for each Worker↓ $i=1..n$: Worker↓ $i$ computes $F↓i$ in parallel

**Algorithm 1** Evolution Strategies

1. **Input**: Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
2. **for** $t = 0, 1, 2, \ldots$ **do**
3. Sample $\epsilon_1, \ldots, \epsilon_n \sim \mathcal{N}(0, I)$
4. **Compute returns** $F_i = F(\theta_t + \sigma \epsilon_i)$ for $i = 1, \ldots, n$
5. Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n \sigma} \sum_{i=1}^{n} F_i \epsilon_i$
6. **end for**

Generate n random perturbations of $\theta$

**Sequentially** run each mutant

Compute gradient estimate
SECOND ATTEMPT: THE PARALLEL ALGORITHM

**KEY IDEA:** Minimize communication cost avoid sending $\text{len}(\epsilon) = |\theta|$, send $\text{len}(F\downarrow_i) = 1$ instead.

*How?* Each worker reconstructs random perturbation vector $\epsilon$

*...How?* Make initial random seed of $\text{Worker}\downarrow_i$ globally known.

---

**Algorithm 2**: Parallelized Evolution Strategies

1. **Input:** Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
2. **Initialize:** $n$ workers with known random seeds, and initial parameters $\theta_0$
3. for $t = 0, 1, 2, \ldots$ do
   4. for each worker $i = 1, \ldots, n$ do
      5. Sample $\epsilon_i \sim \mathcal{N}(0, I)$
      6. Compute returns $F_i = F(\theta_t + \sigma \epsilon_i)$
   7. end for
8. Send all scalar returns $F_i$ from each worker to every other worker
9. for each worker $i = 1, \ldots, n$ do
10. Reconstruct all perturbations $\epsilon_j$ for $j = 1, \ldots, n$ using known random seeds
11. Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} F_j \epsilon_j$
12. end for
13. end for

*Embarassingly parallel!*

With $F\downarrow_j$ and $\epsilon\downarrow_j$ known by everyone, each worker compute the same gradient estimate

*Tradeoff:* redundant computation over $|\theta|$ message size
EXPERIMENT: HOW WELL DOES IT SCALE?

Actual speedup  Ideal speedup (perfectly linear)

\[
\begin{align*}
\frac{657\text{min}}{60\text{min}} & \approx 11.0 \times \\
\frac{200\text{cores}}{18\text{cores}} & \approx 11.1 \times \\
\frac{657\text{min}}{10\text{min}} & \approx 65.7 \times \\
\frac{1440\text{cores}}{18\text{cores}} & \approx 80.0 \times
\end{align*}
\]

Criticism:
Are diminishing returns due to:
• increased communication cost from more workers
• less reduction in variance of the gradient estimate from more workers

Figure 1: Time to reach a score of 6000 on 3D Humanoid with different number of CPU cores. Experiments are repeated 7 times and median time is reported.

▶ Linearly!
With diminishing returns; often inevitable.
**INTRINSIC DIMENSIONALITY OF THE PROBLEM**

**Argument:**

- The number of update steps in ES scales with the *intrinsic dimensionality* of $\theta$, needed for the problem, **not with the length of $\theta$.**

**Justification:**

- E.g. Simple linear regression: Double $|\theta| \rightarrow |\theta'|$

- After adjusting $\eta$ and $\sigma$.

- Update step has the same effect.

- Same number of update steps.

\[
\frac{\partial}{\partial \theta} F(\theta) \approx \frac{1}{\sigma^2} E_{\epsilon \sim N(0, \sigma^2)} [\epsilon F(\theta + \epsilon)]
\]

\[
E_{\epsilon}[\frac{F(\theta) \epsilon}{\sigma^2}] = 0, \text{ since } E_{\epsilon}[\epsilon] = 0
\]

\[
\frac{\partial}{\partial \theta} F(\theta) \approx E_{\epsilon \sim N(0, \sigma^2)} \left[ \frac{F(\theta + \epsilon) - F(\theta)}{\sigma^2} \epsilon \right]
\]

$\approx$ finite differences in some random direction $\epsilon$

$\Rightarrow$ number of update steps scales with $|\theta|?$. 

---

**Diagram:**

- Double $|\theta| \rightarrow |\theta'|$

- $\epsilon_1 \sim \epsilon_2 \sim N(\mu, \sigma^2)$

- After adjusting $\eta$ and $\sigma$, update step has the same effect.

- Same number of update steps.

\[
y = x\theta
\]

\[
x' = (x, x)
\]

\[
y' = x' \theta'
\]

\[
\approx 2x\theta
\]

\[
\frac{\partial}{\partial \theta} F(\theta)
\]

\[
\eta' = \frac{\eta}{2}, \sigma' = \frac{\sigma}{2}
\]
WHEN IS ES A BETTER CHOICE THAN POLICY GRADIENTS?

How do we compute gradients?

Policy gradients:
Policy network outputs a softmax of probabilities for different discrete actions, and we sample an action randomly.

$$\nabla_{\theta} F_{PG}(\theta) = \mathbb{E}_e \{ R(a(e, \theta)) \nabla_{\theta} \log p(a(e, \theta); \theta) \}$$

Evolution strategy (ES):
We randomly perturb our parameters: \( \theta \rightarrow \tilde{\theta} \)
then select actions according to \( \tilde{\theta} \)

$$\nabla_{\tilde{\theta}} F_{ES}(\theta) = \mathbb{E}_\xi \{ R(a(\xi, \theta)) \nabla_{\tilde{\theta}} \log p(\tilde{\theta}(\xi, \theta); \theta) \}$$

ASIDE: In case you forget; for independent X & Y:

\[
\begin{align*}
\text{Var}[XY] &= E[X^2Y^2] - E[XY]^2 \\
\#	ext{ Since } E[X^2] &= \text{Var}[X] + E[X]^2 \\
&= (\text{Var}[X] + E[X]^2)(\text{Var}[Y] + E[Y]^2) - E[X]^2E[Y]^2 \\
&= \text{Var}[X]\text{Var}[Y] + \text{Var}[X]E[Y]^2 + \text{Var}[Y]E[X]^2 \\
&\approx \text{Var}[X]\text{Var}[Y]
\end{align*}
\]

Credit assignment problem
ES makes fewer (potentially incorrect) assumptions

Independent of episode length.
Variance of gradient estimate grows linearly with the length of the episode.
\( \gamma \) only fixes this for short-term returns!
EXPERIMENT: ES ISN’T SENSITIVE TO LENGTH OF EPISODE $\tau$

- **Frame-skip $F$:**
  - Agent can select an action every $F$ frames of input pixels.
  - E.g. $F = 4$
    - Frame 1: agent selects an action
    - Frame 1-3: agent is forced to take Noop action

**IDEA:**
artificially inflate the length of an episode $\tau$

**Argument:**
Since the ES algorithm doesn’t make any assumption about time horizon $\gamma$ (decaying reward), it is less sensitive to long episodes $\tau$ (i.e. the **credit assignment problem**)

---

*Figure 2: Learning curves for Pong using varying frame-skip parameters. Although performance is stochastic, each setting leads to about equally fast learning, with each run converging in around 100 weight updates.*
The authors looked at:

- discrete action tasks -- Atari
- continuous action tasks -- Mujoco
Paper’s claim: “Given the same amount of compute time as other algorithms, compared to A3C, ES does better on 21 games, worse on 29”

Slightly misleading claim if you aren’t reading carefully:

A3C still does better on most games across all algorithms

⇒ ES is still beaten by other algorithms when it beats A3C
EXPERIMENT: CONTINUOUS ACTION TASKS -- MUJOCO

**Sampling complexity:**
How many steps in the environment were needed to reach X% of policy gradient performance?

\[
\frac{\text{# ES Timesteps}}{\text{# TRPO Timesteps}} < 1 \quad \Rightarrow \text{Better sampling complexity}
\]

\[
\frac{\text{# ES Timesteps}}{\text{# TRPO Timesteps}} > 1 \quad \Rightarrow \text{Worse sampling complexity}
\]

Table 1: MuJoCo tasks: Ratio of ES timesteps to TRPO timesteps needed to reach various percentages of TRPO’s learning progress at 5 million timesteps.

<table>
<thead>
<tr>
<th>Environment</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>HalfCheetah</td>
<td>0.15</td>
<td>0.49</td>
<td>0.42</td>
<td>0.58</td>
</tr>
<tr>
<td>Hopper</td>
<td>0.53</td>
<td>3.64</td>
<td>6.05</td>
<td>6.94</td>
</tr>
<tr>
<td>InvertedDoublePendulum</td>
<td>0.46</td>
<td>0.48</td>
<td>0.49</td>
<td>1.23</td>
</tr>
<tr>
<td>InvertedPendulum</td>
<td>0.28</td>
<td>0.52</td>
<td>0.78</td>
<td>0.88</td>
</tr>
<tr>
<td>Swimmer</td>
<td>0.56</td>
<td>0.47</td>
<td>0.53</td>
<td>0.30</td>
</tr>
<tr>
<td>Walker2d</td>
<td>0.41</td>
<td>5.69</td>
<td>8.02</td>
<td>7.88</td>
</tr>
</tbody>
</table>

**Harder tasks:** at most 10x more samples required

**Simpler tasks:** as few as 0.33x samples required
SUMMARY: EVOLUTION STRATEGY

- ES are a viable alternative to **current RL algorithms**:

  Q-learning: **Learn** the action-value function:

  \[
  Q(s, a)
  \]

  Policy gradient; e.g. TRPO: **Learn** the policy directly

  \[
  \pi(a|s, \theta)
  \]

- **ES:**

  Treat the problem like a black-box, perturb \( \theta \) and evaluate fitness \( F(\theta) \):

  \[
  F(\theta) = E_\tau[R_\tau]
  \]

  Where:
  
  - \( \tau = \text{An episode of state (s) action (a) pairs} \)
  - \( R_\tau = \text{Sum of rewards received over episode } \tau \)

- No potentially incorrect assumptions about **credit assignment problem** (e.g. time horizon \( \gamma \))

- No backprop required
  - Embarrassingly parallel
  - Lower GPU memory requirements