Reinforcement Learning and the Future of Software

Danny Goodman
Software is eating the world.

Marc Andreessen
Software is eating the world.

Marc Andreessen

SOFTWARE IS EATING THE WORLD, BUT AI IS GOING TO EAT SOFTWARE

Jensen Huang / Nvidia CEO
Economic Implications

• Software **adds** to the US today
  – $1.14T of GDP
  – 10.5M jobs

• AI will **add** to the world in 2030
  – $13T to GDP
  – 10% of all jobs
Goals

• Explain Reinforcement Learning
• Convince you RL will change software
• Applications in software
• Proof of concept in software synthesis
About Me

• Founder  Switchback Ventures
  – AI for Enterprise  Prudential  McDonalds  Lionsgate

• Technical Background
  – Math  Machine Learning  
  – 4 papers, 6 patents

• Operator at Startups
  – VP of Platform  Mind  salesforce research
  – Director of Data Science  metromile

• Social Media
  – https://www.linkedin.com/in/danthmangoodman/
Software Runs Everything
Software Is Hard

![Fire image](image1.png)

![Medical image](image2.png)

![Error message](image3.png)

![Map image](image4.png)
Reinforcement Learning Will Fix Coding
RL Is Really Powerful

• Winning Human Games
  – Atari: 2013
  – Go: 2016
  – Chess: 2017
  – Dota2: 2018
RL Is Really Powerful

• Winning Human Games
  – Atari: 2013
  – Go: 2016
  – Chess: 2017
  – Dota2: 2018

• Other applications
  – Designing neural nets
  – Robot control
  – Cooling data centers
What is RL?

- Way to train an **Agent**
RL Agent Needs a NN to Work

- RL is a general framework
  - The hard part is Agent: reward history $\Rightarrow$ action
RL Agent Needs a NN to Work

- RL is a general framework
  - The hard part is Agent: reward history $\Rightarrow$ action
- Neural Networks solve the pattern recognition problem
  - Assuming sufficient data and compute
RL Requirements

• Negligible cost for agent to play
  – AlphaGo played billions of Go games

• Reward function not too sparse
  – Learn through exploration

• Tremendous compute
  – RL is sample inefficient (b/c NNs are at present)
RL Current Limitations

• Understanding limited to “on-policy” states
  – NNs not yet good at extrapolation & abstraction
  – Example: AlphaGo-Lee Sedol game 4
RL Current Limitations

- Understanding limited to “on-policy” states
  - NNs not yet good at extrapolation & abstraction
  - Example: AlphaGo-Lee Sedol game 4
  - Similar to human chess cognition
RL Current Limitations

• Understanding limited to “on-policy” states
  – NNs not yet good at extrapolation & abstraction
  – Example: AlphaGo-Lee Sedol game 4
  – Similar to human chess cognition

• No “self-awareness”
  – Doesn’t know what it doesn’t know
  – AI Safety is ongoing area of research

OpenAI  future of life institute  DeepMind
Flavors of RL

- Policy Iteration
- Value Iteration
- GANs
- MCTS
Policy Iteration Example: AutoML

- **Action**: output RNN or CNN component
- **State**: NN (partial) architecture
- **Reward**: performance of NN on training task
- **Policy**: RNN outputs NN architecture
- **Example**: **Efficient Neural Architecture Search**
Flavors of RL

• Policy Iteration
• Value Iteration
• GANs
• MCTS
Value Iteration Example: Atari

- Action: game player action
- State: game display pixels
- Reward: real-time score changes
- Value CNN: game pixels $\Rightarrow$ action values
- Example: Playing Atari with Deep Reinforcement Learning
Flavors of RL

• Policy Iteration
• Value Iteration
• GANs
• MCTS
GAN Example: Image Generation

- Action: create an image
- State: image
- Reward: score from fake-image discriminator
- Discriminator CNN, Generator NN
- Example: Generative Adversarial Nets
Flavors of RL

- Policy Iteration
- Value Iteration
- GANs
- MCTS
MCTS Example: AlphaGo

- Monte-Carlo Tree Search to play go
- Policy CNN: which tree branches to follow
- Value CNN: evaluate final positions
AlphaGo: RL Is Advancing Rapidly

- **2016**: janky
- Pre-trained on 15M human games
- Random rollouts to aid in evaluation
- Performance judged sub-human

- **2017**: clean
- No human learning
- Superhuman performance
Deep Learning Is the Soul of RL

• 2016 $\Rightarrow$ 2017 improvement due to NNs
• CNN $\Rightarrow$ ResNet
• Value and Policy networks share convolution weights
Software Is Ideal for RL

- Negligible cost to “play”: compile & run code cheaply
- Reward functions:
  - Input/Output behavior
  - Resource usage
  - Test cases
  - Code analysis
Classical Methods Improve with RL

• AlphaZero Analogy
  – MCTS for go was “strong amateur” level
  – Guiding tree search with RL was superhuman

• Many classical software analysis techniques
Classical Software Tools Ready

- IDEs
- Compiler optimization
- Automatic Deploy
- SCM & CICD
Opportunities for Improvement

- Code Generation
- Parameter Tuning
- Security
- Code Quality
Opportunities in Code Generation

• Autocomplete**: prefix → code
• Higher level language: description → code
• Code from examples: (input, output) → code
• Visual coding: mockups → code
• Optimization: code → better code
Opportunities in Parameter Tuning

- AutoML**
- Process scheduling
- Resource provisioning
- Database optimization
Opportunities in Security

• NN-guided dynamic analysis for
  – Bug finding
  – Formal verification
• Automatic software patching**
• Detecting malicious code**
Opportunities in Code Quality

- Fixing broken tests automatically
- Smarter refactoring tools that modify tests and code together**
- RL for TDD: test → code to make it pass
- Visual bug-fixing in interfaces
“Classical” NN Capabilities

• Gather data ⇒ train model, eg:
  – Code examples ⇒ predict next code snippets
  – Labelled network traffic ⇒ identify intrusions

• Limitations:
  – Gathering & labeling data is time consuming
  – NNs need >thousands of human data points
  – Need a new NN for every application
    • NNs reproduce a labeling process without generalization
Research Frontier: Neural Nets

**Tree-to-Tree NNs for Program Translation**

CoffeeScript Program: \( x=1 \) if \( y==0 \)

- Parse Tree
  - Block
    - If
      - Op: \( == \)
        - Value
          - Identifier: \( y \)
        - Value
          - Number Literal: \( 0 \)
      - Block
        - Assign
          - Value
            - Identifier: \( x \)
          - Value
            - Number Literal: \( 1 \)

JavaScript Program: if \( (y === 0) \) \{ x = 1; \}

- Parse Tree
  - Program
    - IfStatement
      - BinaryExpression
        - Identifier: \( y \)
        - Op: \( === \)
        - Literal: \( 0 \)
      - AssignExpression
        - Identifier: \( x \)
        - Op: \( = \)
        - Literal: \( 1 \)
Research Frontier: Neural Nets

- **Tree-to-Tree NNs for Program Translation**
- **Neural Turing Machines**
Research Frontier: RL

- **Priority Queue Training**

Table 1: The eight commands in the BF programming language (c.f. [https://esolangs.org/wiki/Brainfuck](https://esolangs.org/wiki/Brainfuck)).

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Move the data pointer to the right</td>
</tr>
<tr>
<td>&lt;</td>
<td>Move the data pointer to the left</td>
</tr>
<tr>
<td>+</td>
<td>Increment the value at the current memory position</td>
</tr>
<tr>
<td>-</td>
<td>Decrement the value at the current memory position</td>
</tr>
<tr>
<td>.</td>
<td>Output the value at the current memory position</td>
</tr>
<tr>
<td>,</td>
<td>Get next value from the input stream and write it to the current memory position</td>
</tr>
<tr>
<td>[</td>
<td>Jump past the matching ] if the cell under the pointer is 0</td>
</tr>
<tr>
<td>]</td>
<td>Jump back to the matching [ if the cell under the pointer is nonzero</td>
</tr>
</tbody>
</table>

Table 3: Number of successes (out of 25) of synthesis methods on all tasks when the maximum number of programs executed (max NPE) is 20M. In each cell we report two numbers separated by a forward slash. The number of successes on training test cases is first, and the number of successes on held-out eval test cases is second. For many task-method combinations no program was found which satisfies the training cases, and we mark these cells with just a dash.

<table>
<thead>
<tr>
<th>Task</th>
<th>Uniform</th>
<th>GA</th>
<th>PG</th>
<th>PQ1</th>
<th>PG+PQ1</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse</td>
<td>2 / 2</td>
<td>15 / 15</td>
<td>3 / 2</td>
<td>20 / 20</td>
<td>17 / 17</td>
</tr>
<tr>
<td>remove-char</td>
<td>- / -</td>
<td>21 / 0</td>
<td>2 / 0</td>
<td>18 / 0</td>
<td>10 / 0</td>
</tr>
<tr>
<td>count-char</td>
<td>- / -</td>
<td>4 / 4</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
</tr>
<tr>
<td>add</td>
<td>2 / 2</td>
<td>19 / 18</td>
<td>6 / 6</td>
<td>25 / 25</td>
<td>25 / 25</td>
</tr>
<tr>
<td>bool-logic</td>
<td>- / -</td>
<td>19 / 18</td>
<td>19 / 19</td>
<td>15 / 15</td>
<td>17 / 17</td>
</tr>
<tr>
<td>print-hello</td>
<td>- / -</td>
<td>12 / 16</td>
<td>- / -</td>
<td>25 / 25</td>
<td>25 / 25</td>
</tr>
<tr>
<td>echo-twice</td>
<td>- / -</td>
<td>11 / 2</td>
<td>- / -</td>
<td>3 / 1</td>
<td>5 / 1</td>
</tr>
<tr>
<td>echo-thrice</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
</tr>
<tr>
<td>copy-reverse</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
</tr>
<tr>
<td>zero-cascade</td>
<td>- / -</td>
<td>1 / 0</td>
<td>- / -</td>
<td>21 / 1</td>
<td>22 / 0</td>
</tr>
<tr>
<td>cascade</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>11 / 1</td>
<td>9 / 0</td>
</tr>
<tr>
<td>shift-left</td>
<td>13 / 0</td>
<td>25 / 2</td>
<td>13 / 2</td>
<td>25 / 0</td>
<td>25 / 0</td>
</tr>
<tr>
<td>shift-right</td>
<td>- / -</td>
<td>3 / 0</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
</tr>
<tr>
<td>riffle</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
</tr>
<tr>
<td>unriffle</td>
<td>- / -</td>
<td>4 / 0</td>
<td>- / -</td>
<td>8 / 0</td>
<td>8 / 0</td>
</tr>
<tr>
<td>middle-char</td>
<td>- / -</td>
<td>1 / 0</td>
<td>- / -</td>
<td>17 / 0</td>
<td>22 / 0</td>
</tr>
<tr>
<td>remove-last</td>
<td>1 / 1</td>
<td>19 / 13</td>
<td>- / -</td>
<td>25 / 9</td>
<td>25 / 5</td>
</tr>
<tr>
<td>remove-last-two</td>
<td>- / -</td>
<td>2 / 0</td>
<td>- / -</td>
<td>25 / 9</td>
<td>25 / 8</td>
</tr>
<tr>
<td>echo-alternating</td>
<td>- / -</td>
<td>4 / 0</td>
<td>- / -</td>
<td>25 / 0</td>
<td>25 / 0</td>
</tr>
<tr>
<td>echo-half</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>1 / 0</td>
<td>- / -</td>
</tr>
<tr>
<td>length</td>
<td>22 / 10</td>
<td>25 / 5</td>
<td>17 / 2</td>
<td>25 / 12</td>
<td>25 / 10</td>
</tr>
<tr>
<td>echo-second-seq</td>
<td>25 / 10</td>
<td>25 / 5</td>
<td>25 / 9</td>
<td>25 / 8</td>
<td>25 / 7</td>
</tr>
<tr>
<td>echo-nth-seq</td>
<td>- / -</td>
<td>13 / 13</td>
<td>- / -</td>
<td>24 / 23</td>
<td>25 / 25</td>
</tr>
<tr>
<td>substring</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
</tr>
<tr>
<td>divide-2</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
</tr>
<tr>
<td>dedup</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
<td>- / -</td>
</tr>
<tr>
<td>Average</td>
<td>2.5 / 1.0</td>
<td>8.6 / 4.3</td>
<td>3.3 / 1.5</td>
<td>13.0 / 5.7</td>
<td>12.9 / 5.4</td>
</tr>
</tbody>
</table>
Research Frontier: RL

- **Priority Queue Training**
- Program Synthesis ([survey](#))
- **Input-Output Examples**
- **RL Guided Tree Search**
- **Leveraging Grammar**
RL & Software Experiment

• Goal: demonstrate RL understands software

• Making is understanding ⇒ software synthesis

• Problem hard enough to be relevant

• Avoid data issues

```python
quine = "r='r=%r;print(r%%r)';print(r%r)\n"
assert quine == self.execute(compile(quine, '<string>', 'exec'))
```
Experiment: Learning a Quine

• Quine: Program with output identical to code

```python
quine = "r='r='r;print(r%r)';print(r%r)\n"
assert quine == self.execute(compile(quine, '<string>', 'exec'))
```

— Or, a fixed point of the code evaluation operator

• Exists by general (long, ugly) construction in every Turing-complete language

• Hard for humans

```python
>>> print(); print('print'); print('print('print'))
```

```python
print
print('print')
```
Experiment: Anatomy of a Quine

```python
quine = "r='r=%r;print(r%%r)';print(r%r)\n"
assert quine == self.execute(compile(quine, '<string>', 'exec'))
```

- 2 Statements:
   - `r='r=%r;print(r%%r)'`
   - `print(r%r)\n`

- Key is self-interpolation: `r%r`

```
>>> r%r
"r='r=%r;print(r%%r)';print(r%r)"
```

- Python `print` ⇒ trailing newline
Experiment: Learning a Quine

• State: sequence of tokens (code)
  – <START> r = ' % ; print ( ) \n n <END>

```python
quine = "r='r=%r;print(r%r)';print(r%r)\n"
assert quine == self.execute(compile(quine, '<string>', 'exec'))
```

  – P("print" | ASCII chars) < 0.00000000001

• Action: output token or run code
Experiment: Reward for a Quine

• Options
  – Edit_distance(code, output)
  – Common prefix/suffix

• Partial credit for:
  – Successful compilation
  – Execution without error
  – Producing output

```python
>>> r%%%%'print
File "<stdin>", line 1
  r%%%%'print
  SyntaxError: invalid syntax
```
```python
>>> print('print')
print
```
Experiment: Policy Iteration

• RNN policy outputs 1 token at a time
  – Random off-policy “exploration”

• Iterate
  – Sample batch of codes
  – Execute and score each code
  – Upweight good codes, downweight bad codes

• Policy iteration gets stuck
  – eg training can converge to: print(print)
  – Most changes are “typos”
  – Probabilities are “zero sum”/winner take all
Experiment: Value Iteration

- $V(s) = E(\text{reward} \mid \text{code prefix})$
- Policy: lookahead to select best next character
- Not “zero sum”, so can learn off-policy values
- Converges to local optimum depending on reward
  - Edit distance: `print('[super duper long string]')`
  - Common prefix:

```python
>>> print('print(\'print(\"print(\\\"print(\\\\\\\")
print('print(\'print(\\
>>>  
```
Experiment: Memory

- Store priority queue of best unique rewards
- Randomly sample 10% of a training batch from them
- Significantly reduces training time for sparse rewards; makes things learnable that would be too sparse, killed by regularization otherwise
- Similar to “priority queue training”
Experiment: Sketch Completion

- Value Iteration misses key idea
- Sketches are programs with placeholders

```python
sketch = "\"%s(r%%r)\n""
```

- Specify key idea, self-interpolation
  - Learn the rest

```python
quine = self.sketch % "r='r=%r;print(r%%r)';print" 
assert quine == self.execute(compile(quine, '<string>', 'exec'))
```
Experiment: Sketch Completion

- It works

```
>>> placeholder = "r='%r';"
>>> exec("r='%r';print(r%r)\n")
'\n' + r
```

```
>>> sketch % placeholder
"r='%r';(r%r)\n"
>>> exec(sketch % placeholder)
```

```
>>> quine = self.sketch % "r='r=%r;print(r%r);print"
assert quine == self.execute(compile(quine, '<string>', 'exec'))
```
Experiment: Takeaways

• Human and machine do better together
  – Humans better at concepts & ideas; RL missed self-interpolation
  – RL filled in the sketch, which humans find tedious

• Reduce an enormous code search space with sensible heuristics/structure
Practical RL Advice

- Use open source TensorFlow and PyTorch
- Start small, e.g., 1 GPU
- Debugging is hard; plot things and look at agent behavior
- Validate your deep learning
- Consider regularization and memory
Summary: RL and Software

• Reinforcement Learning, Neural Net $\Rightarrow$ agent

• Software is ideal RL domain

• Most applications barely touched

• Software synthesis PoC shows RL can handle simple tasks autonomously, can solve hard tasks with human input
SOFTWARE IS EATING THE WORLD, BUT AI IS GOING TO EAT SOFTWARE

Jensen Huang / Nvidia CEO
SOFTWARE IS EATING THE WORLD, BUT AI IS GOING TO EAT SOFTWARE

Jensen Huang / Nvidia CEO
Switchback Ventures is Looking for Development Partners

contact us

✉️ sales@switchback.ventures   ☎️ +1 (415) 799 - 1265

www.switchback.ventures

Danny Goodman
danny@switchback.ventures
+1 (646) 331 - 0984

Trevor z Hallstein
trevor@switchback.ventures
+1 (510) 747 - 8149
QUESTIONS?
National Security Implications

• RL can provably secure code OR find exploits
• Governments that oppose human rights and cyber conventions are making enormous investments in this technology.
  — The free world needs to step up to be secure.