RL in the Real World: From Chip Design to LLMs

Anna Goldie
Anthropic & Stanford
(Includes work done at Google Brain)
Structure of this Talk

- RL for Chip Design
  - RL for AI Accelerators
- RL for Large Language Models
  - RL from Human Feedback
  - RL from AI Feedback
Structure of this Talk

- **RL for Chip Design**
  - RL for AI Accelerators

- **RL for Large Language Models**
  - RL from Human Feedback
  - RL from AI Feedback
RL-Generated Chip Floorplans used in TPU! (Now for multiple generations, including the latest)
A graph placement methodology for fast chip design

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhor, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter & Jeff Dean

Nature 594, 207–212 (2021) | Cite this article

21k Accesses | 1 Citations | 1552 Altmetric | Metrics
Google Proposes AI as Solution for Speedier AI Chip Design

Google invents AI that learns a key part of chip design

AI helps design AI chip that might help an AI design future AI chips

By Samuel K. Moore

Google is using AI to design chips that will accelerate AI

Google trains chips to design themselves

by Peter Grad, Tech Xplore
In the past decade, systems and hardware have transformed ML.
In the past decade, systems and hardware have transformed ML. Now, it’s time for ML to transform systems and hardware.
Demand for Compute Outpacing Supply (Moore’s Law)

Implications of achieving performance on the computation, carbon emissions, and economic costs from deep learning on projections from polynomial models. *The Computational Limits of Deep Learning, Thompson et al., 2020*

Since 2012, the amount of compute used in the largest AI training runs doubled every 3.4 months, *OpenAI, 2019*
Scaling Laws: Compute Fuels Progress in ML

![Graphs showing relationships between compute, dataset size, and parameters, with equations for each relationship.

Figure 1  Language modeling performance improves smoothly as we increase the model size, dataset size, and amount of compute used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

Value of Machine Learning for Chip Design

- Enabling cheaper, faster, and more environmentally friendly chips

- Potential to reduce the design cycle from 1.5-2 years to weeks
  - Today, we design chips for the NN architectures of 2-5 years from now
  - Shortening the chip design cycle would enable us to be far more adaptive to the rapidly advancing field of machine learning

- New possibilities emerge if we evolve NN architectures and chips together
  - Discovering the next generation of NN architectures (which would not be computationally feasible with today’s chips)
Chip Floorplanning Problem

- A form of graph resource optimization
- Place the chip components to minimize the latency of computation, power consumption, chip area and cost, while adhering to constraints, such as congestion, cell utilization, heat profile, etc.
Complexity of Chip Placement Problem

Chess

Number of states $\sim 10^{123}$

Go

Number of states $\sim 10^{360}$

Chip Placement

Number of states $\sim 10^{9000}$
Prior Approaches to Chip Placement

<table>
<thead>
<tr>
<th>Partitioning-Based Methods (e.g. MinCut)</th>
<th>Stochastic/Hill-Climbing Methods (e.g. Simulated Annealing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytic Solvers (e.g. RePIAce)</td>
<td></td>
</tr>
</tbody>
</table>
Prior Approaches to Chip Placement

<table>
<thead>
<tr>
<th>Partitioning-Based Methods (e.g. MinCut)</th>
<th>Stochastic/Hill-Climbing Methods (e.g. Simulated Annealing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytic Solvers (e.g. RePlAce)</td>
<td>Learning-Based Methods</td>
</tr>
</tbody>
</table>
Chip Placement with Reinforcement Learning

**State:** Graph embedding of chip netlist, embedding of the current node, and the canvas.

**Action:** Placing the current node onto a grid cell.

**Reward:** A weighted average of total wirelength, density, and congestion.
Our Objective Function

\[ J(\theta, G) = \frac{1}{K} \sum_{g \sim G} E_{g,p \sim \pi_\theta}[R_{p,g}] \]

Set of training graphs \( G \)

\( K \) is size of training set

Reward corresponding to placement \( p \) of netlist (graph) \( g \)

\[ R_{p,g} = \ -Wirelength(p, g) \]

\(-\lambda \ Congestion(p, g) - \gamma \ Density(p, g)\)
We Take a Hybrid Approach to Placement Optimization
Results on a TPU-v4 Block

White area are macros and the green area is composed of standard cell clusters. Our method finds smoother, rounder macro placements to reduce the wirelength.

<table>
<thead>
<tr>
<th>Human Expert</th>
<th>ML Placer</th>
</tr>
</thead>
</table>

Time taken: 6-8 weeks
Total wirelength: 57.07m
Route DRC violations: 1766

DRC: Design Rule Checking

Time taken: 24 hours
Total wirelength: 55.42m (-2.9% shorter)
Route DRC violations: 1789 (+23 - negligible difference)
Moving Towards Generalized Placements

Before: Training from scratch for each chip netlist

1. New Netlist → Placer Policy → Placements
   - 10,000s of iterations

Now: Pre-training the policy and fine-tuning on new netlists

2. Netlist 1 → Policy → Placements
3. Netlist 2 → Policy → Placements
4. Netlist N → 10,000s of iterations → Policy → Placements

Inference

- New Netlist → Pre-Trained Policy → Placements
  - 100s of iterations

Final result
First Attempts at Generalization

Using the previous RL policy architecture, we trained it on multiple chips and tested it on new unseen chips.
First Attempts at Generalization

Using the previous RL policy architecture, we trained it on multiple chips and tested it on new unseen chips. -> Didn’t work!
First Attempts at Generalization

Using the previous RL policy architecture, we trained it on multiple chips and tested it on new unseen chips. -> Didn’t work!

Freezing different layers of the RL policy and then testing it on new unseen chips
First Attempts at Generalization

Using the previous RL policy architecture, we trained it on multiple chips and tested it on new unseen chips. -> Didn’t work!

Freezing different layers of the RL policy and then testing it on new unseen chips -> Didn’t work either!
First Attempts at Generalization

Using the previous RL policy architecture, we trained it on multiple chips and tested it on new unseen chips. \( \rightarrow \) Didn’t work!

Freezing different layers of the RL policy and then testing it on new unseen chips \( \rightarrow \) Didn’t work either!

What did work?
First Attempts at Generalization

Using the previous RL policy architecture, we trained it on multiple chips and tested it on new unseen chips. -> Didn’t work!

Freezing different layers of the RL policy and then testing it on new unseen chips -> Didn’t work either!

What did work? Leveraging supervised learning to find the right architecture!
Achieving Generalization by Training Accurate Reward Predictors

Key observation: A value network trained only on placements generated by a single policy is unable to accurately predict the quality of placements generated by another policy, limiting the ability of the policy network to generalize.
Achieving Generalization by Training Accurate Reward Predictors

**Key observation:** A value network trained only on placements generated by a single policy is unable to accurately predict the quality of placements generated by another policy, limiting the ability of the policy network to generalize.

To decompose the problem, we trained models capable of accurately predicting reward from off-policy data.
Compiling a Dataset of Chip Placements

To train a more accurate predictor, we generated a dataset of 10k placements.

Each placement was labeled with their wirelength and congestion, which were drawn from vanilla RL policies.

Each color represents a different netlist.
Searching for Effective Neural Architecture for Encoder

Input Features

- Node Features (x, y, w, h, type*)
- Graph (macro, standard cells, clusters)
- Netlist Metadata (Total number of wires and macros, name of netlist)

*Node type: One-hot category {Hard macro, soft macro}

Predictions
- Wirelength
- Congestion

fc
fc
Searching for Effective Neural Architecture for Encoder

Input Features

Node Features
(x, y, w, h, type*)

Graph (macro, standard cells, clusters)

Netlist Metadata
(Total number of wires and macros, name of netlist)

*Node type: One-hot category {Hard macro, soft macro}

Graph Conv

while Not converged do
  Update edge: \( e_{ij} = f_{c1}(\text{concat}|f_{c0}(v_i)|f_{c0}(v_j)|w_{ij}|) \)
  Update node: \( v_i = \text{mean}_{j \in N(v_i)}(e_{ij}) \)
end

Predictions

Wirelength

Congestion

fc

fc
Edge-based Graph Convolution: Node Embeddings
Edge-based Graph Convolution: Edge Embedding
Edge-based Graph Convolution: Edge Embedding
Edge-based Graph Convolution: Propagate
Edge-based Graph Convolution: Repeat
Final Step: Get Graph Embedding

reduce mean
Discovered Reward Model Architecture and Features

Input Features

- Node Features (x, y, w, h, type)
- Graph (macro, standard cell)
- Netlist Metadata (Total number of wires and macros, name of netlist)

*Node type: One-hot category {Hard macro, soft macro}*

Graph Conv

Predictions

- Wirelength
- Congestion

fc

fc
Label Prediction Results on Test Chips

![Graph 1: Predicted wirelength vs. wirelength](image1)

![Graph 2: Predicted congestion vs. congestion](image2)
Overall RL Policy/Value Network Architecture
Comparisons with Manual and SOTA Baselines

<table>
<thead>
<tr>
<th>Name</th>
<th>Method</th>
<th>Timing</th>
<th>Area</th>
<th>Power</th>
<th>Wirelength</th>
<th>Congestion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>WNS (ps)  TNS (ns)</td>
<td>Total (μm²)</td>
<td>Total (W)</td>
<td>(m)</td>
<td>H (%)</td>
</tr>
<tr>
<td>Block 1</td>
<td>RePlAce</td>
<td>374      233.7</td>
<td>1693139</td>
<td>3.70</td>
<td>52.14</td>
<td>1.82</td>
</tr>
<tr>
<td></td>
<td>Manual</td>
<td>136      47.6</td>
<td>1680790</td>
<td>3.74</td>
<td>51.12</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>84       23.3</td>
<td>1681767</td>
<td>3.59</td>
<td>51.29</td>
<td>0.34</td>
</tr>
<tr>
<td>Block 2</td>
<td>RePlAce</td>
<td>97       6.6</td>
<td>785655</td>
<td>3.52</td>
<td>61.07</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td>Manual</td>
<td>75       98.1</td>
<td>830470</td>
<td>3.56</td>
<td>62.92</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>59       170</td>
<td>694757</td>
<td>3.13</td>
<td>59.11</td>
<td>0.45</td>
</tr>
<tr>
<td>Block 3</td>
<td>RePlAce</td>
<td>193      3.9</td>
<td>867390</td>
<td>1.36</td>
<td>18.84</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>Manual</td>
<td>18       0.2</td>
<td>869779</td>
<td>1.42</td>
<td>20.74</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>11       2.2</td>
<td>868101</td>
<td>1.38</td>
<td>20.80</td>
<td>0.04</td>
</tr>
<tr>
<td>Block 4</td>
<td>RePlAce</td>
<td>58       11.2</td>
<td>944211</td>
<td>2.21</td>
<td>27.37</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>Manual</td>
<td>58       17.9</td>
<td>947766</td>
<td>2.17</td>
<td>29.16</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>52       0.7</td>
<td>942867</td>
<td>2.21</td>
<td>28.50</td>
<td>0.03</td>
</tr>
<tr>
<td>Block 5</td>
<td>RePlAce</td>
<td>156      254.6</td>
<td>1477283</td>
<td>3.24</td>
<td>31.83</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>Manual</td>
<td>107      97.2</td>
<td>1480881</td>
<td>3.23</td>
<td>37.99</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>68       141.0</td>
<td>1472302</td>
<td>3.28</td>
<td>36.59</td>
<td>0.01</td>
</tr>
</tbody>
</table>

- We freeze the macro placements generated by each method and report the place opt results by the commercial EDA.
Ariane (RISC-V) Placement Visualization

Training policy from scratch

Finetuning a pre-trained policy

Ariane is an open-source RISC-V processor. See: https://github.com/pulp-platform/ariane
Convergence Curve: Training from Scratch vs. Finetuning
Open-Source Release of RL Framework ("Circuit Training")
Structure of this Talk

● RL for Chip Design
  ○ RL for AI Accelerators

● RL for Large Language Models
  ○ RL from Human Feedback
  ○ RL from AI Feedback
What to do in domains where reward is hard to specify?

- One solution is to ask humans to provide feedback - however, this is prohibitively expensive in the naive formulation, as RL typically requires thousands to millions of labels to learn an effective policy (depending on the complexity of the task)
- But what if you train a model to predict human judgments and then use this predictive model as the reward signal?
Deep RL from Human Preferences

- Without access to the true reward function and labeling <1% of the environment interactions, able to perform complex tasks, including Atari games and MuJoCo.

RL from Human Feedback in LLMs (aka RLHF)

- “Secret sauce” behind powerful LLMs like ChatGPT and Anthropic’s Claude!
- Humans rank-order pairs of behavior, train a preference model, use preference model as reward, and RL-finetune to optimize “good” behavior
- Performing RLHF on top of pretrained large language models (LLMs) greatly improves instruction-following / in-context learning / prompting.

Bai et al. Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. 12 Apr 2022.
How to Perform RLHF

Bai et al. Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. 12 Apr 2022.
Step 1: Collect Human Judgments
Step 2: Train Preference Models (PMs)

- Train PM to assign a higher score to the response preferred by a human rater
- Base models from 13M through 52B parameters (in increments of 4x)
Step 3: Perform RL-Finetuning with PM as Reward Signal

- Extract all prompts from the previous steps, prompt the base LM to respond, and then use the PM score as the reward signal
- Train with Proximal Policy Optimization (PPO) with an auxiliary KL penalty

\[ r_{total} = r_{PM} - \lambda_{KL} D_{KL}(policy \parallel policy_0) \]
Takeaways

- Alignment tax for small models but alignment bonus for 13B+ models
- Tradeoff between helpfulness and harmlessness, but performance improves on both distributions as model scale up
- RLHF improves programming ability for models pretrained on code
- RLHF boosts performance on MMLU, Lambada, Hellaswag, OpenBookQA, and ARC, but hurt performance on TriviaQA compared to a base models
Next Step: RL from AI Feedback (RLAIF)!

- **Motivation**: Scaling supervision - as models approach or exceed human-level performance, it becomes difficult for humans to supervise them.
- **RLAIF**: Perform RL-finetuning using AI feedback derived from a “constitution” describing desired behavior. Humans don’t need to be in the loop, except to write the constitution!

Bai et al. [Constitutional AI: Harmlessness from AI Feedback](https://example.com). 15 Dec 2022.
Benefits of Supervised Learning + Reinforcement Learning

- **Supervised Learning**: Improves initial model, which helps with exploration and sample efficiency
- **Reinforcement Learning**: Significantly boosts performance and reliability of the final policy
Supervised Phase

1. Sample from an initial policy
2. Generate “self-critiques” and revisions
3. Finetune the original model with the revised responses
Reinforcement Learning Phase

1. Sample from a finetuned model
2. Use a model to evaluate which of two responses is “better”
3. Train a preference model on the AI-labeled data
4. Perform RL-finetuning with the PM as the reward signal (just like RLAIF)
Takeaways

- Finetuning with AI-generated feedback can generate results that match or exceed models that are finetuned with human feedback
Questions?

- RL for Chip Design
  - RL for AI Accelerators
- RL for Large Language Models
  - RL from Human Feedback
  - RL from AI Feedback
Bonus Content: RL for Device Placement!
Structure of this Talk

- **RL for Chip Design**
  - RL for AI Accelerators

- **RL for Large Language Models**
  - RL from Human Feedback
  - RL from AI Feedback

- **RL for Systems Optimization**
  - RL from Device Placement / Model Parallelism
Hierarchical Learning for Device Placement
Azalia Mirhoseini¹, Anna Goldie², Hieu Pham, Benoit Steiner, Quoc V. Le, Jeff Dean

(*) Equal contribution

SUMMARY
We propose a Reinforcement Learning algorithm that learns to automatically design model parallelism for TensorFlow graphs.

PROBLEM
- Given:
  - TensorFlow computational graph G with N ops
  - List of computing devices D (GPUs, CPUs, etc.)
- Find:
  - Placement \( P = \{p_1, p_2, ..., p_D\} \) with \( p_i \in D \)
  - Minimizes the running time of G

A REINFORCEMENT LEARNING APPROACH
- Using policy gradient to learn a policy \( \pi \) that:
  - Proposes placement and then measures runtime
  - Minimizes expected runtime \( J(\pi, \theta) = \mathbb{E}_{P(\pi, \theta)}[R_d] = \sum_{g \in G} \sum_{d \in D} p(g; \theta) p(d; \theta) R_d \)

TRAINING WITH REINFORCEMENT
The goal is to minimize the expectation of runtime:

\[
J(\theta, \theta_d) = \mathbb{E}_{P(\theta, \theta_d)}[R_d] = \sum_{g \in G} \sum_{d \in D} p(g; \theta) p(d; \theta) R_d
\]

\[
\nabla_{\theta_d} J(\theta_d, \theta_d) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\theta_d} \log p(g_i; \theta_d) \sum_{d \in D} p(d; \theta) R_d
\]

\[
\nabla_{\theta} J(\theta, \theta_d) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\theta} \log p(g_i; \theta) \sum_{d \in D} p(d; \theta) R_d
\]

EXAMPLE PLACEMENTS
- Each color is a GPU; transparent is the CPU.
- Neural Machine Translation with 2 layers
- Classification from Inception-V3

UNDERSTANDING THE PLACEMENTS
- Our method learns to optimize for different objectives for different models.
  - For RNNLM: learns that it is best to put all ops on a single GPU.
  - For NMT: learns to balance computation across devices.
  - For Inception-V3: learns to mitigate the time spent on inter-device memory copy.

RESULTS

<table>
<thead>
<tr>
<th>Task</th>
<th>CPU Only</th>
<th>GPU Only</th>
<th>XPU Only</th>
<th>Human Expert</th>
<th>Scratch</th>
<th>MinCIF</th>
<th>Hierarchical Place</th>
<th>Runtime Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNNLM</td>
<td>0.22</td>
<td>0.15</td>
<td>0.12</td>
<td>0.05</td>
<td>0.02</td>
<td>0.13</td>
<td>0.14</td>
<td>19.5%</td>
</tr>
<tr>
<td>RNNLM (2 GPUs)</td>
<td>0.22</td>
<td>0.15</td>
<td>0.12</td>
<td>0.05</td>
<td>0.02</td>
<td>0.13</td>
<td>0.14</td>
<td>19.5%</td>
</tr>
<tr>
<td>RNNLM (3 GPUs)</td>
<td>0.22</td>
<td>0.15</td>
<td>0.12</td>
<td>0.05</td>
<td>0.02</td>
<td>0.13</td>
<td>0.14</td>
<td>19.5%</td>
</tr>
<tr>
<td>RNNLM (4 GPUs)</td>
<td>0.22</td>
<td>0.15</td>
<td>0.12</td>
<td>0.05</td>
<td>0.02</td>
<td>0.13</td>
<td>0.14</td>
<td>19.5%</td>
</tr>
</tbody>
</table>

On the left, we show the computational load profiling of NMT model for RL-based and expert-designed placements. Smaller blocks of each color correspond to forward pass and same-color upper blocks correspond to back-propagation. On the right, we show memory copy time profiling. All memory copy activities in Synchronous towers are between GPUs and a CPU, which are in general slower than GPU copies that take place in the RL-based placement.
What is device placement and why is it important?

Trend towards many-device training, bigger models, larger batch sizes

Google neural machine translation'16
300 million parameters, trained on 128 GPUs

Sparsely gated mixture of experts'17
130 billion parameters, trained on 128 GPUs

BigGAN'18
355 million parameters, trained on 512 TPU cores
Standard practice for device placement

- Often based on greedy heuristics
- Requires deep understanding of devices: nonlinear FLOPs, bandwidth, latency behavior
- Requires modeling parallelism and pipelining
- Does not generalize well
ML for device placement

- ML is repeatedly replacing rule based heuristics
- We show how RL can be applied to device placement
  - Effective search across large state and action spaces to find optimal solutions
  - Automated learning from underlying environment only based on reward function (e.g. runtime of a program)
Posing device placement as an RL problem

Input
- Neural model
- Set of available devices
  - CPU
  - GPU

RL model
- Policy

Output
- Assignment of ops in neural model to devices
Posing device placement as an RL problem

Input
- Neural model
- Set of available devices
  - CPU
  - GPU

RL model
- Policy
- Evaluate runtime
- Assignment of ops in neural model to devices

Output
An end-to-end hierarchical placement model
Learned placement on NMT

White represents CPU (Ixion Haswell 2300)
Each other color represents a separate GPU (Nvidia Tesla K80)
Searching over a space of $5^{280}$ possible assignments
Profiling placement on NMT
Learned placement on Inception-V3

White represents CPU (Ixion Haswell 2300)
Each other color represents a separate GPU (Nvidia Tesla K80)
Searching over a space of $5^{83}$ possible assignments
Profiling placement on Inception-V3
Profiling placement on Inception-V3
## Results (runtime in seconds)

<table>
<thead>
<tr>
<th>Tasks</th>
<th>CPU Only</th>
<th>GPU Only</th>
<th>#GPUs</th>
<th>Human Expert</th>
<th>Scotch</th>
<th>MinCut</th>
<th>Hierarchical Planner</th>
<th>Runtime Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception-V3</td>
<td>0.61</td>
<td>0.15</td>
<td>2</td>
<td>0.15</td>
<td>0.93</td>
<td>0.82</td>
<td>0.13</td>
<td>16.3%</td>
</tr>
<tr>
<td>ResNet</td>
<td>-</td>
<td>1.18</td>
<td>2</td>
<td>1.18</td>
<td>6.27</td>
<td>2.92</td>
<td>1.18</td>
<td>0%</td>
</tr>
<tr>
<td>RNNLM</td>
<td>6.89</td>
<td>1.57</td>
<td>2</td>
<td>1.57</td>
<td>5.62</td>
<td>5.21</td>
<td>1.57</td>
<td>0%</td>
</tr>
<tr>
<td>NMT (2-layer)</td>
<td>6.46</td>
<td>OOM</td>
<td>2</td>
<td>2.13</td>
<td>3.21</td>
<td>5.34</td>
<td>0.84</td>
<td>60.6%</td>
</tr>
<tr>
<td>NMT (4-layer)</td>
<td>10.68</td>
<td>OOM</td>
<td>4</td>
<td>3.64</td>
<td>11.18</td>
<td>11.63</td>
<td>1.69</td>
<td>53.7%</td>
</tr>
<tr>
<td>NMT (8-layer)</td>
<td>11.52</td>
<td>OOM</td>
<td>8</td>
<td>3.88</td>
<td>17.85</td>
<td>19.01</td>
<td>4.07</td>
<td>-4.9%</td>
</tr>
</tbody>
</table>
Summary

- Deep RL for resource allocation optimization
- **Papers:**
  - ICLR’18: A Hierarchical Model for Device Placement, 
    - Azalia Mirhoseini*, Anna Goldie*, Hieu Pham, Benoit Steiner, Quoc V. Le and Jeff Dean
  - ICML’17: Device Placement Optimization with Reinforcement Learning, 
    - Azalia Mirhoseini*, Hieu Pham*, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, Jeff Dean
- **Open-source TensorFlow code:**
  https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/grappler
Comparing Models with Elo Scores

Win Fraction = \frac{1}{1 + 10^{\frac{\Delta(\text{Elo Score})}{400}}} \quad \text{and} \quad \Delta(\text{Elo Score}) \approx 174 \times \Delta(\text{PM Score})