Learning to superoptimize programs

Anonymous Author(s)
Affiliation
Address
email

Abstract

Superoptimization requires the estimation of the best program for a given computational task. In order to deal with large programs, superoptimization techniques perform a stochastic search. This involves proposing a modification of the current program, which is accepted or rejected based on the improvement achieved. The state of the art method uses uniform proposal distributions, which fails to exploit the problem structure to the fullest. To alleviate this deficiency, we learn a proposal distribution over possible modifications using Reinforcement Learning. To demonstrate the efficacy of our approach, we provide convincing results on the superoptimization of "Hacker's Delight" programs.

1 Introduction

Superoptimization requires us to obtain the optimal program for a computational task. While modern compilers implement a large set of rewrite rules, they fail to offer any guarantee of optimality. An alternative approach is to search over the space of all possible programs that are equivalent to the compiler output, and select the one that is the most efficient. If the search is carried out in a brute-force manner, we are guaranteed to achieve superoptimisation. However, this approach quickly becomes computationally infeasible as the number of instructions and the length of the program grows.

In order to efficiently perform superoptimisation, recent approaches have started to use a stochastic search procedure, inspired by Markov Chain Monte Carlo sampling [13]. Briefly, the search starts at an initial program, such as the compiler output. It iteratively suggests modifications to the program, where the probability of a modification is encoded in a proposal distribution. The modification is either accepted or rejected with a probability that is dependent on the improvement achieved. Under certain conditions on the proposal distribution, the above procedure can be shown to eventually sample from a distribution over programs, where the probability of a program is related to its quality. In other words, the more efficient a program, the more times it is encountered, thereby enabling superoptimisation.

One of the main factors that governs the efficiency of the above stochastic search is the choice of the proposal distribution. Surprisingly, the state of the art method, called Stoke [13], uses an almost uniform proposal distribution. We argue that this choice fails to fully exploit the power of stochastic search. For example, consider the case where we are interested in performing bitwise operations, as indicated by the compiler output. In this case, it is more likely that the optimal program will contain bitshifts than floating point opcodes. Yet, Stoke will assign an equal probability of use to those.

In order to alleviate the aforementioned deficiency of Stoke, we build a reinforcement learning framework to estimate a more suitable proposal distribution for the task at hand. The quality of the distribution is measured as the expected quality of the program obtained via stochastic search. Using training data, which consists of a set of input programs, the parameters are learnt via the REINFORCE algorithm [16]. We demonstrate the efficacy of our approach on a set of "Hacker's Delight" [15] programs. Preliminary results indicate that a learnt proposal distribution outperforms the uniform one on novel tasks that were previously unseen during training.

Submitted to the workshop on Neural Abstract Machines and Program Induction (NIPS 2016). Do not distribute.

2 Related Works

The earliest approached for superoptimization relied on brute-force search. By sequentially enumerating all programs in increasing length orders [4, 11], the shortest program meeting the specification is guaranteed to be found. As expected, this approach scales poorly to longer programs or to large instruction sets. The longest reported synthesized program was 12 instructions long, on a restricted instruction set [11].

Trading off completeness for efficiency, stochastic methods [13] reduced the number of programs to test by guiding the exploration of the space, using the observed quality of programs encountered as hints. However, using a generic, unspecific exploratory policy made the optimisation blind to the problem at hand. We propose to tackle this problem by learning the proposal distribution.

Similar work was done in the restricted case of finding efficient implementation of computation of value of degree k polynomials [17]. Programs were generated from a grammar, using a learned policy to prioritise exploration. This particular approach of guided search looks promising to us, and is in spirit similar to our proposal, although applied on a very restricted case.

Another approach to guide the exploration of the space of programs was to make use of the gradients of differentiable relaxation of programs. Bunel et al. [2] attempted this by simulating program execution using recurrent Neural Networks. This however provided no guarantee that the optimum found was going to correpond to a real program. Additionally, this method only had the possibiliy of performing very local moves, limiting the kind of discoverable transformations.

Outside of program optimisation, applying learning algorithms to improve optimisation procedures, either in terms of results achieved or time taken, is a well studied subject. Doppa et al. [3] proposed imitation learning based methods to deal with structured output spaces, in a "Learning to search" framework. This is however not useful to our problem as we always have a valid cost function.

More relevant is the recent litterature on learning to optimize. Li and Malik [10] and Andrychowicz et al. [1] learns how to improve on first-order gradient descent algorithms, making use of neural networks. Our work is similar, as we aim to improve the optimisation process. We differ in that our initial algorithm is a MCMC sampler, on a discrete space, as opposed to gradient descent on a continuous, unconstrained space.

The training of a Neural Network to generate a proposal distribution to be used in sequential Monte-Carlo was also proposed by Paige and Wood [12] as a way to accelerate inference in graphical models. Additionally, similar approaches were successfully employed in computer vision problems where data driven proposals allowed to make inference feasible [7, 9, 18].

3 Learning Stochastic Superoptimization

Stoke performs black-box optimisation of a cost function on the space of programs, represented as a series of instructions. Each instruction is composed of an opcode, specifying what to execute, and some operands, specifying the corresponding registers. Each given input program $\mathcal{T}$ defines a cost function. For a candidate program $\mathcal{R}$ called a rewrite, the associated cost is given by:

$$\text{cost}\left(\mathcal{R}, \mathcal{T}\right) = \omega_e \times \text{eq}(\mathcal{R}, \mathcal{T}) + \omega_p \times \text{perf}(\mathcal{R}) \tag{1}$$

The term $\text{eq}(\mathcal{R}; \mathcal{T})$ measures how well do the outputs of the rewrite match with the outputs of the reference program when executed. This can be obtained either by running a symbolic validator or by running test cases, and accepting partial definition of correctness. The other term, $\text{perf}(\mathcal{R})$ is a measure of the execution time of the program. An approximation can be the sum of the latency of all the instructions in the program. Alternatively, timing the program on some test cases can be used.

To find the optimum of this cost function, Stoke runs an MCMC sampler, using the Metropolis algorithm. This allows to sample from the probability distribution induced by the the cost function:

$$p(\mathcal{R}; \mathcal{T}) = \frac{1}{Z} \exp(-\text{cost}\left(\mathcal{R}, \mathcal{T}\right))), \tag{2}$$

where $\mathcal{R}$ is the proposed rewrite, $\mathcal{T}$ is the input program.

The sampling is done by proposing random moves $\mathcal{R} \to \mathcal{R}^\star$, sampled from a proposal distribution $q(\mathcal{R}^\star|\mathcal{R})$. An acceptance criterion is computed, and used as the parameter of a Bernoulli distribution,

to decide whether or not the move is accepted.

$$\alpha(\mathcal{R} \to \mathcal{R}^\star, \mathcal{T}) = \min\left(1, \frac{p(\mathcal{R}^\star; \mathcal{T})}{p(\mathcal{R}; \mathcal{T})}\right). \tag{3}$$

This criterion is justified at the condition that the proposal distribution is symmetric, that is, $q(\mathcal{R}^\star|\mathcal{R}) = q(\mathcal{R}|\mathcal{R}^\star)$. In that case, in the limit, the distribution of states visited by the sampler will be p, making the optimal program the most sampled. In practice, the proposal distribution used is not symmetric but the whole process can still be understood as a stochastic search.

The proposal distribution q originally used in [13] is a hierarchical model. A type of move is initially sampled from a probability distribution. Depending on the move that was sampled, additional samples are drawn to specify which part of the current program should be modified or which new operand or opcode should be used. The detailed structure of the probability distribution can be found in the attached supplementary material.

Existing methods use uniform distributions for each of the elementary probability distributions the model sample from. This corresponds to a specific instantiation of the general approach. We propose to learn those probability distribution so as to maximize the probability of reaching the best programs.

The cost function defined in equation (1) corresponds to what we want to optimize. Under a fixed computational budget to perform program superoptimization in less than T iterations, we are interested in having the lowest possible cost at the end. Given that our optimisation procedure is stochastic, we will need to consider the expected cost as our loss. This expected loss is a function of the parameters θ of our proposal distribution. The objective function of our "meta-optimisation" problem is therefore:

$$\mathcal{L}(\theta) = E_\theta\left(\min_{t=1..T} \text{cost}\left(\mathcal{R}_t, \mathcal{T}\right)\right), \tag{4}$$

Our chosen parameterisation of q is to keep the hierarchical structure of the original work of Schkufza et al. [13], and parameterise all separate probability distributions (over the type of move, the opcodes, the operands, and the lines of the program) independently. In order to learn them, we will make use of unbiased estimators of the gradient. These can be obtained using the REINFORCE algorithm [16]. A helpful way to derive them is to consider the execution traces of the search procedure under the formalism of stochastic computation graphs [14]. This graph used can be found in the supplementary materials, as well as the derivation of the gradients associated with it.

By instrumenting the Stoke system of Schkufza et al. [13], we can collect the execution traces so as to compute gradients over the outputs of the probability distributions, which can then be backpropagated. In that way, we can perform Stochastic Gradient Descent (SGD) over our objective function 4.

4 Experiments

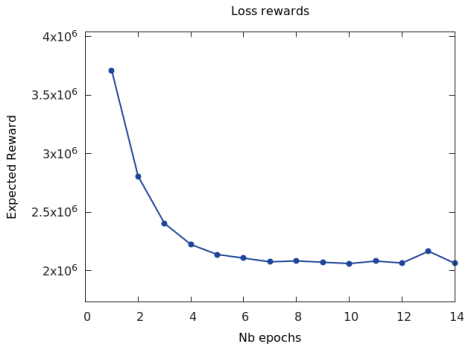
We ran our experiments on the Hacker's delight [15] corpus, a collection of 25 bit-manipulation programs, used as benchmark in program synthesis [6, 8, 13]. A detailed description of the task is given in the appendix. Some examples include identifying whether an integer is a power of two from its binary representation, counting the number of bits turned on in a register or computing the maximum of two integers.

In order to have a larger corpus than the twenty-five programs initially obtained, we generate various starting points for each optimisation. This is accomplished by running Stoke with a cost function where $\omega_p = 0$ in (1), keeping only the correct programs and filtering out duplicates. This allows us to create a larger dataset.

We divide the Hacker's Delight tasks into two sets. We train on the first set and only evaluate performance on the second so as to evaluate the generalisation of our learned proposal distribution. We didn't attempt to learn the probability distribution over the operands and the program position, only learning the ones over opcodes and type of move to perform.

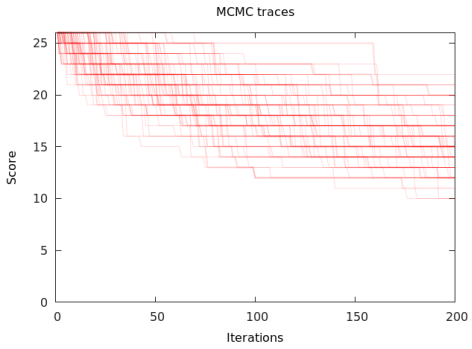
The probability distribution is here a simple categorical distribution. We learn the parameters of each separate distribution jointly, using a Softmax transformation to enforce that they are proper probability distribution. In our current experiment, the proposal distributions are not conditioned on the input program. Optimising them corresponds to finding an ideal proposal distribution for Stoke. Figure 2 shows the results. Both the training and the test loss decreases and it can be observed that the optimisation of program happens faster and that more programs reach the observed minimum.

(a) Training Loss

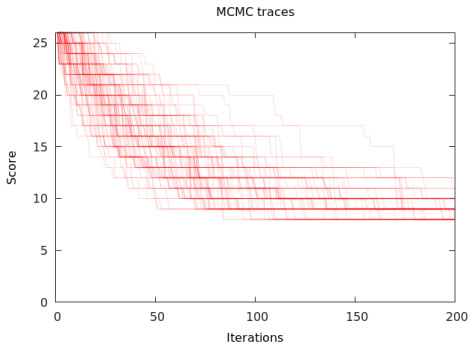


(b) Testing Loss

Figure 1: Training of the proposal distribution. (1a) corresponds to the (unnormalized) objective function of Eq.(4), respectively on the Training dataset and on the Testing dataset.



(a) Before Training



(b) After Training

(c) Learned proposal distribution on the type of move

Figure 2: (2a) and (2b) are superposition of the plot of the lowest energy achieved using respectively the initial proposal distribution / the trained proposal distribution. The learned proposal distribution over the type of moves to do is shown in (2c). Moves corresponding to instruction deletion (second cluster) or instruction swap with same mnemonics (fourth cluster) become more likely than instruction permutations. The proposal distribution over the specific assembly instructions can be found in the supplementary materials.

5 Conclusion

Within this paper, we have shown that learning the proposal distribution of the stochastic search can lead to significant performance improvement. It is interesting to compare our approach to the synthesis-style approaches that have been appearing recently in the Deep Learning community [5] that aim at learning programs directly using differentiable representations of programs. We find that the stochastic search-based approach yields a significant advantage compared to those types of approaches, as the resulting program can be run independently from the Neural Network that was used to discover them.

Several improvements are possible to the presented approach. Making the probability distribution a Neural Network conditioned on the initial input or on the current state of the rewrite would lead to a more expressive model, while essentially having similar training complexity. It will however be necessary to have a richer, more varied dataset to make any evaluation meaningful.

References

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. *CoRR*, 2016.

[2] Rudy Bunel, Alban Desmaison, Pushmeet Kohli, Philip HS Torr, and M Pawan Kumar. Adaptive neural compilation. In *NIPS*. 2016.

[3] Janardhan Rao Doppa, Alan Fern, and Prasad Tadepalli. Hc-search: A learning framework for search-based structured prediction. *JAIR*, 2014.

[4] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. *ACM SIGPLAN Notices*, 1992.

[5] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, 2014.

[6] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[7] Varun Jampani, Sebastian Nowozin, Matthew Loper, and Peter V Gehler. The informed sampler: A discriminative approach to bayesian inference in generative computer vision models. *Computer Vision and Image Understanding*, 2015.

[8] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*, 2010.

[9] Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *CVPR*, 2015.

[10] Ke Li and Jitendra Malik. Learning to optimize. *CoRR*, 2016.

[11] Henry Massalin. Superoptimizer: A look at the smallest program. In *ACM SIGPLAN Notices*, 1987.

[12] Brookes Paige and Frank Wood. Inference networks for sequential Monte Carlo in graphical models. In *ICML*, 2016.

[13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *SIGPLAN*, 2013.

[14] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *NIPS*, 2015.

[15] Henry S Warren. Hacker's delight. 2002.

[16] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992.

[17] Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. In *NIPS*. 2014.

[18] Song-Chun Zhu, Rong Zhang, and Zhuowen Tu. Integrating bottom-up/top-down for object recognition by data driven markov chain monte carlo. In *CVPR*, 2000.

Learning to superoptimize programs Supplementary Materials

Anonymous Author(s)
Affiliation
Address
email

1 Generative model of the program transformations

In Stoke [2], the program transformation are sampled from a generative model. This process was analysed from the publicly available code [1].

First, a type of transformation is sampled uniformly from the following proposals method.

1. **Add a NOP instruction** Add an empty instruction at a random position in the program.
2. **Delete an instruction** Remove one of the instruction of the program.
3. **Instruction Transform** Replace one existing line (instruction + operands) by a new one (New instruction and new operands).
4. **Opcode Transform** Replace one instruction by another one, keeping the same operands. The new instruction is sampled from the set of compatible instructions.
5. **Opcode Width Transform** Replace one instruction by another one, with the same memonic. This means that those instructions do the same thing, except that they don't operate on the same part of the registers (for example, will replace `movq` that move 64-bit of data of the registers by `movl` that will move 32-bit of data)
6. **Operand Transform** Replace the operand of a randomly selected instruction by another valid operand for the context, sampled at random.
7. **Local swap Transform** Swap two instructions in the same "block".
8. **Global Swap transform** Swap any two instructions.
9. **Rotate transform** Draw two positions in the program, and rotate all the instructions between the two (the last one becomes the first one of the series and all the others get pushed back).

Then, once the type of move has been sampled, the actual move has to be sampled. To do that, a certain numbers of sampling steps need to happen. Let's take as example 3.

To perform an **Instruction Transform**,

1. A line in the existing programs is uniformly chosen.
2. A new instruction is sampled, from the list of all possible instructions.
3. For each of the arguments of the instruction, sample from the acceptable value.
4. The chosen line is replaced by the new line that was sampled.

Submitted to the workshop on Neural Abstract Machines and Program Induction (NIPS 2016). Do not distribute.

```
def proposal(current_program):
    move_type = sample(categorical(1-9))
    if move_type == 1: % Add NOP Instruction
       pos = sample(categorical(all-positions(current_program)))
       return (ADD_NOP, pos)
    if move_type == 2: % Delete an Instruction
       pos = sample(categorical(all-positions(current_program)))
       return (DELETE, pos)
    if move_type == 3: % Instruction Transform
       pos = sample(categorical(all-positions(current_program)))
       instr = sample(categorical(set-of-all-instructions))
       arity = nb_args(instr)
       for i = 1, arity:
           operands[i] = sample(categorical(possible-arguments(instr, i))) % get one of the arguments
                                                                           % that can be used as i-th
                                                                           % argument for instr
       return (TRANSFORM, pos, instr, operands)
    if move_type == 4: % Opcode Transform
       pos = sample(categorical(all-positions(current_program)))
       args = arguments_at(current_program, pos)
       instr = sample(categorical(possible-instruction(args))) % get one instruction that can
                                                               % be used with the arguments that
                                                               % are in the program at line pos.
       return(OPCODE_TRANSFORM, pos, instr)
    if move_type == 5: % Opcode Width Transform
       pos = sample(categorical(all-positions(current_program))
       curr_instr = instruction_at(current_program, pos)
       instr = sample(categorical(same-memonic-instruction(curr_instr)) % get one instruction with the
                                                                        % same memonic that the
                                                                        % instruction in the program
                                                                        % at line pos.
       return (OPCODE_TRANSFORM, pos, instr)
    if move_type == 6: % Operand transform
       pos = sample(categorical(all-positions(current-program))
       curr_instr = instruction_at(current_program, pos)
       arg_to_mod = sample(categorical(1-nb_args(curr_instr)))
       new_operand = sample(categorical(possible-arguments(curr_instr, arg_to_mod)))
       return (OPERAND_TRANSFORM, pos, arg_to_mod, new_operand)
    if move_type == 7: % Local swap transform
       block_idx = sample(categorical(all-blocks(current-program)))
       pos_1 = sample(categorical(all-positions-in-block(current_program, block_idx)))
       pos_2 = sample(categorical(all-positions-in-block(current_program, block_idx)))
       return (SWAP, pos_1, pos_2)
    if move_type == 8: % Global swap transform
       pos_1 = sample(categorical(all-positions(current_program)))
       pos_2 = sample(categorical(all-positions(current_program)))
       return (SWAP, pos_1, pos_2)
    if move_type == 9: % Rotate transform
       pos_1 = sample(categorical(all-positions(current_program)))
       pos_2 = sample(categorical(all-positions(current_program)))
       return (ROTATE, pos_1, pos_2)
```

Figure 1: Generative Model of a Transformation

The sampling process of a move is therefore a hierarchy of sampling step. One way to thing of it is that we have a generative model for the moves. Depending on what type we sample, we may have differents series of sampling steps to perform. For a move, all the probabilities are sampled independently so the probability of proposing the move is the product of the probability of picking each of the sampling steps. The generative model is defined in Figure 1. It is going to be parameterized by the the parameters of each specific probability distribution it samples from. The default Stoke version uses uniform probabilities over all of those elementary distributions.

2 Metropolis algorithm as a Stochastic Computation Graph

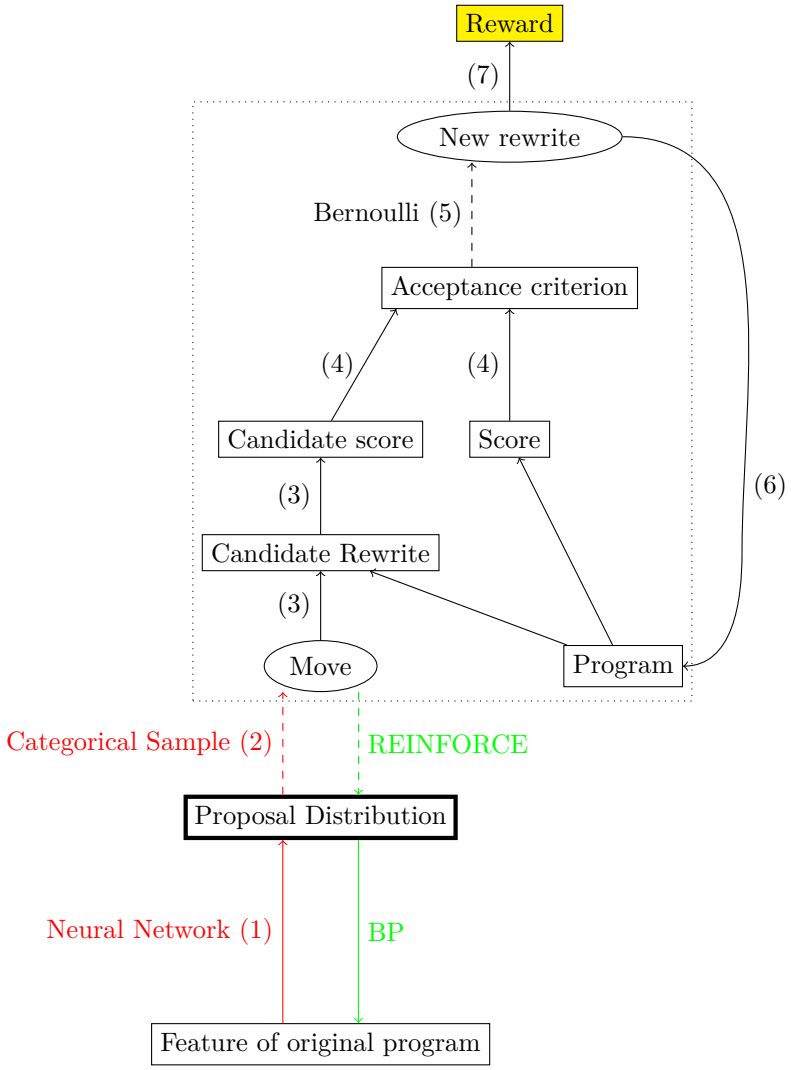


Figure 2: *Stochastic Computation Graph of the Metropolis algorithm used for program superoptimization. Round nodes are stochastic nodes and square ones are deterministic. Red arrows corresponds to computation done in the forward pass that needs to be learned while green arrows correspond to the backward pass. Full arrow represent deterministic computation and dashed arrow represent stochastic ones. The different steps of the forward pass are:*
(1) Based on features of the reference program, the proposal distribution q is computed.
(2) A random move is sampled from the probability distribution and we keep track of the probability of taking this move.
(3) The score of the rewrite that would be obtained by applying the chosen move is measured experimentally.
(4) The acceptance criterion for the move is computed.
(5) The move is accepted with a probability equal to the acceptance criterion.
(6) Move 2-7 are repeated N times.
(7) The reward is observed, corresponding to the best program obtained during the search.

3 Hacker's delight tasks

The 25 tasks of the Hacker's delight [3] datasets are the following:

1. Turn off right-most one bit
2. Test whether an unsigned integer is of the form $2^(n-1)$
3. Isolate the right-most one bit
4. Form a mask that identifies right-most one bit and trailing zeros
5. Right propagate right-most one bit
6. Turn on the right-most zero bit in a word
7. Isolate the right-most zero bit
8. Form a mask that identifies trailing zeros
9. Absolute value function
10. Test if the number of leading zeros of two words are the same
11. Test if the number of leading zeros of a word is strictly less than of another work
12. Test if the number of leading zeros of a word is less than of another work
13. Sign Function
14. Floor of average of two integers without overflowing
15. Ceil of average of two integers without overflowing
16. Compute max of two integers
17. Turn off the right-most contiguous string of one bits
18. Determine if an integer is a power of two
19. Exchanging two fields of the same integer according to some input
20. Next higher unsigned number with same number of one bits
21. Cycling through 3 values
22. Compute parity
23. Counting number of bits
24. Round up to next highest power of two
25. Compute higher order half of product of x and y

4 Learned distribution over the instruction

In addition to the probability distribution over the type of transformation to make that is shown in the main paper, we also learn jointly a distribution over the assembly instruction. The learned version is shown in Figure (3).

Figure 3: Learned proposal distribution over the opcodes. Each pixel correpsonds to a different opcode. Light one correspond to high probability, while black ones correspond to opcodes that are never going to get sampled.

References

[1] Berkeley Churchll, Eric Schkufza, and Stefan Heule. Stoke. `https://github.com/StanfordPL/stoke`, 2016.

[2] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *SIGPLAN*, 2013.

[3] Henry S Warren. Hacker's delight. 2002.