Learning Deep Models of Optimization Landscapes

Shumeet Baluja
Google Research
Mountain View, CA

Abstract—**In all but the most trivial optimization problems, the structure of the solutions exhibit complex interdependencies between the input parameters. Decades of research with stochastic search techniques has shown the benefit of explicitly modeling the interactions between sets of parameters and the overall quality of the solutions discovered. We demonstrate a novel method, based on learning deep networks, to model the global landscapes of optimization problems. To represent the search space concisely and accurately, the deep networks must encode information about the underlying parameter interactions and their contributions to the quality of the solution. Once the networks are trained, the networks are probed to reveal parameter combinations with high expected performance with respect to the optimization task. These estimates are used to initialize fast, randomized, local-search algorithms, which in turn expose more information about the search space that is subsequently used to refine the models. We demonstrate the technique on multiple problems that have arisen in a variety of real-world domains, including: packing, graphics, job scheduling, layout and compression. Strengths, limitations, and extensions of the approach are extensively discussed and demonstrated.**

I. Optimization via Search Space Modeling

In the 1990s, a number of researchers [1][2][3][4] independently started employing probabilistic models to guide stochastic search algorithms. The idea was simple, to use the knowledge of the search landscape that may be ascertained by analyzing the points encountered to guide where to look next. This sharply contrasted the procedure of many successful randomized hill-climbing algorithms that made small perturbations stochastically in hopes that a better solution was a close neighbor to the current best solution.

In its basic formulation, probabilistic methods explicitly maintain statistics about the search space by creating models of the good solutions found so far. These models are then sampled to generate the next query points to be evaluated. The model is updated with the newly sampled solutions, and the cycle is continued.

The primary goal of this paper is to demonstrate that while employing any search algorithm (hillclimbing, genetic algorithm [5][6][7], simulated annealing, etc.), it is possible to simultaneously learn a deep-neural network based approximation of the evaluation function. Then, through the use of "deep-network inversion" (employed as a method to sample the deep networks), intelligent perturbations of some/all of the samples generated by the search algorithm can be made. After these perturbations, the new solutions should have an increased probability of higher scores.

A. *Predecessors to Deep Modeling*

One of the early probabilistic model-based optimization approaches was Population-Based Incremental Learning (PBIL) [1]. A defining characteristic of PBIL was the simplicity of the model employed: no inter-parameter dependencies were captured; each parameter was modeled independently. The entire probability model is a single vector [2][8][9][10][11][12]. Although this simple probabilistic model was used, PBIL was successful compared to many competing approaches.

MIMIC [13] extended PBIL by capturing a heuristically chosen set of the pairwise dependencies between the solution parameters. In [14], MIMIC's probabilistic model was further expanded to a larger class of dependency graphs: trees in which each variable is conditioned on at most one parent. This created the optimal tree-shaped network for a maximum-likelihood model of the data [15]. In experimental comparisons, MIMIC's chain-based probabilistic models performed significantly better than PBIL's simpler models. The tree-based graphs performed significantly better than MIMIC's chains.

The trend indicated that more accurate probabilistic models were helpful [16]. The natural extension to pair-wise modeling is modeling arbitrary dependencies. Bayesian networks are a popular method for efficiently representing complex dependencies [17] [18]. Numerous researchers have combined full Bayesian networks with stochastic search [19] [20] [21]. For an overview, see [22]. An alternate model building approach, termed STAGE, was presented in [23]. STAGE mapped a set of user-supplied features of the state space to a single value representing the solution quality. The "value function" is used to select the next point from which to initialize search.

In the next section, we describe the Deep-Opt algorithm and give details of how the probabilistic model is created and sampled. We also describe how the model is integrated with fast-search heuristics, following the work of [23] and [14].

II. Deep Learning for Search Space Modeling

Optimization with probabilistic modeling, at a high level, is simply explained in Figure 1. A large set of randomly generated candidate solutions are created and evaluated with respect to the objective function. The poor-performing candidate solutions are discarded. The set of remaining solutions, usually a small subset of the better performing members from the original set, are then modeled. The model is stochastically sampled to generate new candidates which are then evaluated, and the process repeated.

Algorithm 1 High Level Probabilistic Modeling for Optimization

```
Create set, S, with random solutions from uniform distribution.
while termination condition is not met do
    Create a probabilistic model, PM, of S.
    Stochastically sample from PM, to generate C candidate solutions.
    Evaluate the C candidate solutions.
    Update S with only the N high-evaluation solutions from $C$, where $N \ll |C|$.
```

Fig. 1. Probabilistic Modeling for Optimization Overview.

In this paper, we use a neural network to create a mapping between the solutions sampled thus far and their score (*e.g.* the evaluation function). In the context of describing the algorithm, we also discuss four large differences between our approach and the probabilistic modeling techniques employed earlier.

First, one of the benefits in using a deep neural network (DNN), is that we do not *a priori* specify the form of the dependencies in the probabilistic model (*e.g.* pair or triplet combinations). Although the architecture of the neural network is manually specified, the dependencies that the network encodes need not be the same form for all parameters, nor are they deeply tied to architecture of the network [24].

Second, sampling a deep neural network to generate new samples is very different than sampling a Bayesian-probabilistic model. With previous models, such as the dependency-trees [15], sampling the model is simple: generate a biased random number that is conditioned on the parent variable specified in the tree-based model. With neural networks, however, generating samples is more complex. **Given a trained network**, the network is "inverted" using standard back-propagation to *modify the inputs* rather than the network's weights. The inputs are modified to match a preset and clamped output. This method was first presented in [25] and has recently been popularized within the context of texture and style generation using DNNs [26] [27].

It works as follows: We are given a trained network that maps the input parameters (scaled between 0.0 and 1.0) to their evaluation (also scaled between 0.0 and 1.0). All the weights of the network are frozen; they will not change for the sample generation process. Then, we clamp the output to the desired output — for maximization problems, the desired output is set to 1.0; this indicates that we would like to generate solutions that are as good as the best ones seen so far. The inputs are then initialized (either randomly or by other means such as perturbing the best solution seen thus far) and the network performs a forward propagation step. The error is measured at the output — the error is the standard Least-Mean Squares Error on the target output (clamped to 1.0).

$$E_{LMS} = \sum_{i \in outputs} (targets_i - predicted_i)^2$$

Since we have pinned the target to 1.0, and there is only a single output scaled between 0.0 and 1.0 (that represents the score of the candidate solution). This is simply:

$$E_{LMS} = (1.0 - predicted_{score})^2$$

A process similar to standard training with stochastic-descent back-propagation (or any other neural network training procedure) is then used. However, unlike standard network training, the errors are propagated back to the inputs, and the **inputs are modified – not the weights**. As described in [25], the procedure addresses the following question: "Which input should be fed into the net to produce an output which approximates the given target vector T' (in our case the the target vector T is a simple scalar of 1.0). The error signal for the input i:

$$\delta_i = -\frac{\delta E}{\delta input_i}$$

tells the *input units* how to change (direction and magnitude) to decrease the error. In general, modest learning-rates for the gradient descent algorithm were found to work best for the network-inversion process (we used the Adam Optimizer [28] with learning-rate = 0.001). If the networks are trained well, then the candidates generated will yield solutions with high scores when tested on the actual evaluation function. In summary, the stochastic sampling process of previous systems is replaced with this network-inversion process that sets the inputs to expected high-evaluation settings.

Third, in previous studies with probabilistic models (Figure 1), the low-performance candidate solutions were discarded and the high-performance solutions were kept. Often, the actual evaluation of the high-performance solutions was not used. In contrast, in our procedure, we create an explicit mapping from the candidate solution to its score. This reflects a fundamental difference in the previous and current approaches: since an explicit mapping is created, it is not assumed that the model only represents good solutions; it is capable of representing both high and low quality solutions as well as a mapping to their scores.

Fourth, note that in contrast to many previous studies that represented the solution vectors as binary strings and modeled only the binary parameters, the deep-neural networks used in this study naturally model real-values (the inputs are real values). Extensions to binary and other discrete parameters have been explored in detail [29], though for space reasons are not presented here.

Algorithm 2 Next-Ascent Stochastic Hillclimbing (NASH) (shown for maximization)

```
Initial random candidate, $c$, composed of $|P|$ real-values.
BestEvaluation $\leftarrow$ Evaluate ($c$)
while termination condition is not met do
    $c' \leftarrow c$
    Num-Perturbations $=$ Rand-Integer from $[1, m \times |c|]$
    loop Num-Perturbations
        Randomly select a parameter, $p$, from $c'$
        c' $\leftarrow$ Modify $p$ to a rand value in $[(.75 * p), (1.25 * p)]$
    CandidateEvaluation $\leftarrow$ Evaluate (c')
    if (CandidateEvaluation $\geq$ BestEvaluation) then
        $c \leftarrow c'$
        BestEvaluation $\leftarrow$ CandidateEvaluation
    else
        Discard $c'$
Return c
```

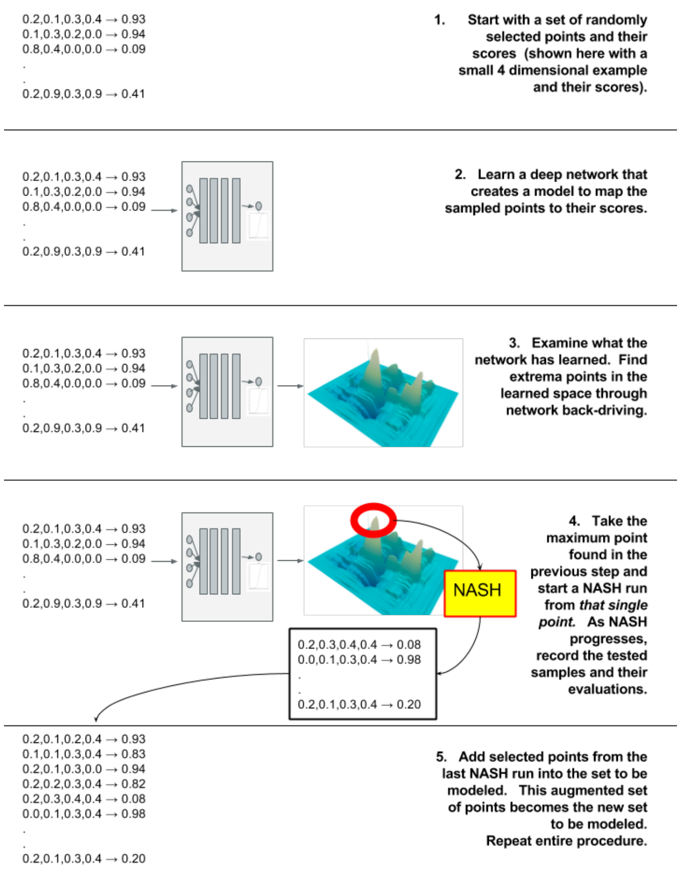
Fig. 2. Next-Ascent Stochastic Hillclimbing (NASH). In our implementation we set $m = 2\%$. Note that the mutation amount ± 0.25 is quite large; however, this was set based on extensive empirical testing on these and similar problems. |P| is dependent on the problem. In the experiments presented throughout this paper, it ranged from 50 to 1600.

A. *Integration with Fast Local Search Heuristics*

In the simplest implementation, the candidate solutions generated by the network inversion are evaluated and the cycle is continued. Although this method will work, there are drawbacks. First, this is a slow process; training a full network to map the sampled points to their evaluations is an expensive procedure, as is sampling the network. Second, a post-processing step of local optimization, where small changes are made to the solutions generated, yields improvement to the solutions found. This is because both the interpolation and extrapolation capabilities of the trained networks are not perfect; there will be discrepancies between the estimated "goodness" (evaluation) of a candidate solution and its actual evaluation.

Two early works in probabilistic model-based optimization [14] [23] suggested using probabilistic models to initialize faster local-search optimization techniques. This technique is used here. A very simple next-ascent stochastic hillclimbing (*NASH*) procedure is shown in Figure 2. It has repeatedly proven to work well in practice when used in conjunction with other optimization algorithms to perform local optimization, and also surprisingly well when used alone in a variety of scenarios [30] [31] [32]. For the majority of the paper, we will use NASH as the underlying search process; the neural modeling will "wrap-around" NASH.

Importantly, note that NASH is initialized with a single candidate solution. This candidate is perturbed until an equal or better solution is found. This fits into the procedures described thus far: all of the candidates evaluated by NASH can be added to S. When the network is trained and the next set of candidates are generated (through network inversion), the single best solution found is used to initialize the NASH algorithm. NASH proceeds as normal, again recording all the candidate solutions it evaluates – which are then used to augment S in the next time step, and the cycle continues. A visual description of the combined procedure is given in Figure 3.



1. Start with a set of randomly selected points and their scores (shown here with a small 4 dimensional example and their scores).

2. Learn a deep network that creates a model to map the sampled points to their scores.

3. Examine what the network has learned. Find extrema points in the learned space through network back-driving.

4. Take the maximum point found in the previous step and start a NASH run from *that single point*. As NASH progresses, record the tested samples and their evaluations.

5. Add selected points from the last NASH run into the set to be modeled. This augmented set of points becomes the new set to be modeled. Repeat entire procedure.

Fig. 3. A visual description of Deep-Opt. (Sample topographic plot by [33])

B. *Implementation Details*

Because this is, to our knowledge, a new approach to probabilistic modeling for optimization, we provide a few of the implementation details required to replicate the results. Figure 3 gives a full description of the algorithm.

- Initializing S. Line 1 (of Figure 3) The simplest method is random sampling; this gives the broadest exploration of the search space but provides no information about the area surrounding the sampled points. Alternatively, we could have performed a single NASH run and used the points explored to initialize S; however, this gives a poor representation of the global landscape. We chose a combination of the two: seed-points are randomly selected and their local neighborhoods are also explored by making small perturbations of the seed points.

- Maintaining S. The size of S is kept constant. To keep S a constant size, after every NASH run, the last 1000 unique solutions from the NASH run are added to S, and S is pruned back by removing the members that have been in S the longest, as in [14]. As members of S improve over time, those that have been in S the longest

Algorithm 3 Deep-Opt at a High-Level

```
 1: Create set, S, with random candidate solutions
 2: Evaluate all solutions in S.

 3: while termination condition is not met do
 4:     Scale all real evaluations of $s$ to range [0.0,1.0], $\forall(s|s \in S)$
 5:     Train a DNN to map $s \rightarrow Evaluation(s), \forall(s|s \in S)$
 6:     Freeze the weights of the deep network.

 7:                                  ▷ Stochastically Generate New Candidates
 8:     Initialize $C$ to be empty.
 9:     repeat
10:         Create $c$ by random perturbations of best solution yet
11:         Append $c$ to $C$.    $C \leftarrow C + c$
12:     until (|C| = Number-To-Generate-From-Model)

13:                                  ▷ Back-Drive the Network to Improve C
14:     Select a fraction, $F$, of $C \rightarrow C', s.t. |C'| \leq |C|$.
15:     Clamp the network's output to 1.0
16:     for each $d \in C'$ do
17:         Initialize the inputs of the DNN with $d$
18:         Back-Drive network until inputs stop changing $\rightarrow d'$
19:         Replace $d$ in C with $d'$

20:                                  ▷ Measure Solution Quality
21:     Evaluate all candidates, $c \in C$.

22:                                  ▷ Further Improve Solution by local optimization
23:                                  ▷ And Obtain New Points to Model
24:     Select $m$ ($m \in C$), where $m$ has the highest evaluation.
25:     Initialize NASH with $m$.
26:     Run NASH. Record all of the unique points (into set $Y$)

27:                                  ▷ Update the data to be modeled
28:     $Y' \leftarrow$ Select a subset of the best solutions from $Y$.
29:     $S \leftarrow S + Y'$
30:     Discard Duplicates in $S$; prune $S$ if necessary
```

Fig. 4. Deep-opt: Creating a model of the search landscape to initialize fast search algorithms (NASH). We set the *Number-to-Generate-From-Model = 50* for all of our tests. $|S|$ was kept at 10,000 samples with the size of Y' set at 1000. For simplicity, F was kept static at 50%.

are often not as good as more recent entrants. Further, overly precise models of low-evaluation areas are not useful to model.

- Network Architecture. Two networks were used in our study. The first, *Deep-Opt-5*, used a 5-fully-connected-layer network with 100 hidden units per layer. The second, *Deep-Opt-10*, used a 10-layer fully-connected network with 20 hidden units per layer and skip connections between every layer and its predecessors. A single output is used which is the estimate of the sample's evaluation. Our goal is to show that using a deep neural network is capable of modeling the search space, not necessarily advocating a particular network or set of parameters.

- Network Training. The use of a validation set (hold-out set) and weight decay is suggested to avoid overfitting, and was found to be vital to good performance. If the error on the validation does not improve through training, training is restarted with random weights.

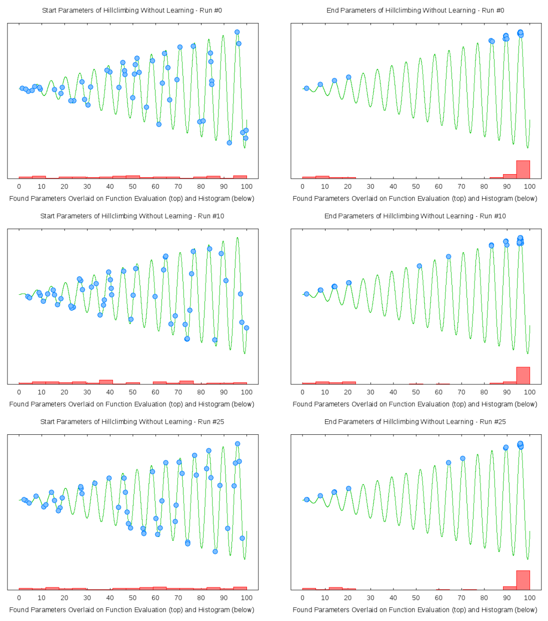


Fig. 5. Sines Problem without Learning. The (green) line is the underlying function to be maximized. The (blue) points are the settings of each the 50 parameters in the beginning (left) and ending (right) of NASH. A histogram showing the distribution of the points is shown in red. Note that in the beginning of each run (left column), the points are uniformly distributed. The results are improved through hillclimbing, though not all points are at the optimum. Top Row: NASH run #1, Middle Row: NASH run #10 (this is the NASH run made after the 9th modeling step), Bottom Row: NASH run #25. About the graph: although each points represents one of the 50 parameters in the same single solution string, they are shown 'overlapped' onto the same graph. This is possible because each parameter is independent and is evaluated with respect to the same function.

C. Visualizing the Learning

Before turning our attention to the empirical tests, we present a motivating example to demonstrate how the local search algorithms utilize the models created. We examine a simple problem in which the evaluation is: $evaluation(\boldsymbol{x}) = \sum_{i=1}^{50}(x_i * sin(x_i))$. We instantiate this simple problem with 50 independent parameters with a range of [0,100].

In Figure 5 (top-row), in the left column are the starting points for all 50 of the parameters before the NASH algorithm is run. They are randomly distributed across the input range. At NASH's completion (right column), many are close to their optimal value. Figure 5(middle row) shows the same for the 10th restart of NASH. Because NASH is still initialized randomly (no learning), we see approximately the same distribution of points in the beginning and the end as before. The 25th restart of NASH is shown in the bottom row.

Next, we repeat the same experiment with Deep-Opt. The difference is in the initialization of the NASH algorithm. The samples that are generated in each NASH run are added to the set of solutions that are modeled by the neural network. From this neural network model, M new samples are drawn. The single best of the M is used to initialize the next NASH run. As expected, in Figure 6, the first run looks similar to the earlier case with no model. This is because there is no

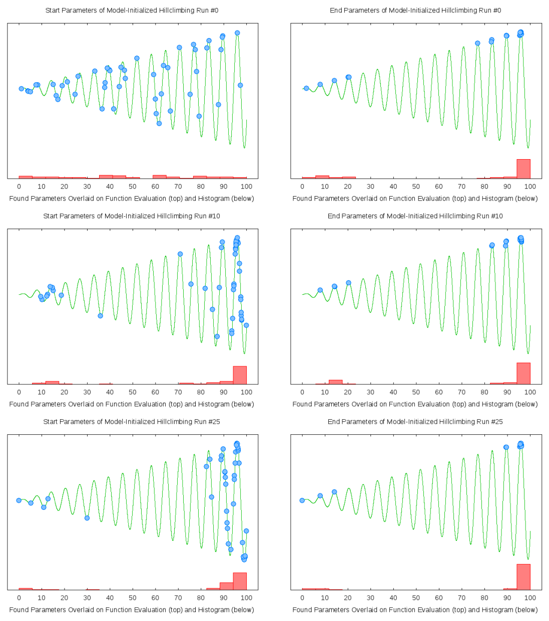


Fig. 6. Sines Problem with Search-Space Modeling / Learning. Note that even at the beginning of the runs, shown in the left column of rows 2 & 3 (run #10 & #25 respectively), the hillclimbing *begins* in regions of high performance – thereby leading to better performance overall by the end of the hillclimbing search. In run 1 (top row, left)- since no learning has yet happened, the points are randomly distributed in the beginning of the run.

information in the model as yet. However, after the 10th NASH restart, (middle-row, Figure 6) there is a distinct difference in the starting values – many more are already in high-evaluation regions. The best solution found through sampling had many of its parameters in the right region of the search space. The likelihood of these reaching the global optimum is increased through NASH (right column). By the 25th run, this trend is even further evident. Sampling the model works as expected: the starting point for the hillclimbing is already in a better region - where more parameters are closer to the global optima.

Though this problem is simple, it demonstrates how the modeling can improve the search results. Interestingly, early on in our studies, modeling *sometimes led to poorer* overall performance. Why? As the probabilistic model improved, exploration decreased – more samples started in a basin of attraction that led to the same local maxima as was seen previously. Improvement slowed because the updates to the model all happened with similar candidate solutions. To address this, the size of the sample set that was used for modeling, $|S|$, was set as shown in Figure 4 to slow the convergence; this vastly improved performance.

III. Empirical Results

Deep-Opt has been tested on numerous of problems drawn from the literature and real-world needs. When the model is used, M samples are generated, the best of which is used to initialize NASH. To ensure that the model is actually providing useful information and that it is not merely the process of examining M samples before initializing NASH that is yielding the improved performance, 3 variants are tested to establish competitive base-lines.

1) **NASH-V1:** This is exactly NASH shown in Figure 2.

2) **NASH-V2:** Before beginning the NASH run, M samples are *randomly* generated and evaluated. The best one found from the M generated is used to initialize the NASH algorithm. No learning is used here. This variant is included to test whether just the process of generating multiple samples and selecting the best for initializing NASH is enough to provide improvement to the final result – even with no modeling.

3) **NASH-V3:** Before beginning the NASH run, M samples are generated by making small perturbations to the best solution found in all the previous NASH runs. (The first NASH run is initialized randomly). Each of the M samples is then evaluated. The best sample found from the M is used to initialize the NASH algorithm. This variant tests whether the neural network models actually capture the shape of the search space, or whether they are only (inefficiently) forcing search around the best solutions seen so far.

The termination condition for each NASH run was either (1) 10,000 evaluations were performed or (2) 500 evaluations were conducted with no-improvement. The latter indicated that the search might be trapped in a local maxima. All approaches were given a total of 500,000 evaluations. The number of NASH runs that were conducted within the 500,000 evaluations was dependent on the problem and how quickly/often the algorithm was unable to escape local optima.

A. Noisy Evaluations

In the previous section, we used a simple Sines maximization problem as an illustrative example of how learning aids search. Because of the problem's simplicity, most search algorithms can perform well on it. However, with the introduction of noise, a clearer separation in performance emerges.

In this version of the problem, *Noisy-Sines*, the evaluation was modified to include significant uniformly distributed random noise. Uniformly chosen random noise between $[0, 0.5]$ was added to the evaluation. For large parts of the search space, the overall evaluation is dominated by noise.

$$eval(\boldsymbol{x}) = \frac{\sum_{i=1}^{50}(x_i * sin(x_i))}{50.0 * 100.0} + uniformNoise(0, 0.5) \quad (1)$$

The performance of each algorithm is judged by the best solution found for the underlying objective function (as determined **without the noise**); no algorithm is privy to the underlying real function. The results are shown in Table I. For each of the 5 algorithms tried (NASH-1,NASH-2,NASH-3, Deep-Opt-5-Layers, Deep-Opt-10-Layers), the best evaluation averaged over trials is listed in the first row. The last two rows

TABLE I
RESULTS FOR NOISY SINES PROBLEM (20 TRIALS FOR EACH APPROACH)

	NASH-1	NASH-2	NASH-3	Deep-5	Deep-10
Avg.	0.690	0.691	0.700	0.731	0.726
Diff. Deep-5 Signif	>99%	>99%	>99%	-	96%
Diff. Deep-10 Signif	>99%	>99%	>99%	96%	-

TABLE II
STABLE MARRIAGE RECEPTION-PARTY SEATING: NUMBER OF TIMES EACH ALGORITHM OUTPERFORMED ALL OTHERS (OUT OF 20)

	NASH-1	NASH-2	NASH-3	Deep-Opt-5	Deep-Opt-10
		Wins=5		***Wins=15***	
Overall Best (Out of 20)	3	2	0	10	5
Deep-Opt-5 Outperformed(ties)	14	16	19	-	11 (1)
Deep-Opt-10 Outperformed(ties)	10	13	15	8 (1)	-

show the significance of the difference between the algorithm's performance and the performance of Deep-Opt-5-Layers and Deep-Opt-10-layers, respectively.

B. *Stable Marriage* Reception-Party Seating

In a canonical version of this problem, G parties are invited to a formal-seated party, such as a wedding reception. Each party can have a variable size. Each member of the party must sit together at one of the T tables, which each have a capacity C_t. The additional twist to this problem is that each G has a preference with whom to sit with, expressed as a real value. The full preference matrix is $|G \times G|$. Preferences can be negative, and not constrained in magnitude. Preferences may not be symmetric. Though this problem shares part of its name with the stable marriage problem, it is more akin to knapsack/packing problems. Real versions of this problem have arisen in topics as diverse as processor scheduling to intern and group seating assignment.

The goal is to find a seating assignment that (1) keeps the members of each group together, (2) does not seat people beyond the capacity of the table, (3) maximizes the summed happiness/preferences over all the tables. For the size of the problems explored here, the reception has 10 tables, each with capacity 12 people. Each party size is randomly chosen between 1 and 3 people. Preferences were expressed as a value between [-100, +100]. The number of groups, $|G|$, was set to 50.

To encode the solution as a vector, each group was assigned T parameters, corresponding to each of T tables; there were a total of 500 (50×10) parameters, $(realValueParameter_{g,t})$. At evaluation time, these 500 parameters were sorted from high to low. Based on the sorted list of $realValueParameter_{g,t}$ assignments were made in order from highest to smallest of group g to table t. Note that the assignment occurred only if the group was (1) as yet unseated and (2) the table could hold the size of the group; otherwise that parameter was ignored and the next one processed. This encoding has the benefit of not only specifying each groups' preferences to tables, but also being able to encode "how important" it is that a particular group be assigned to a particular table.

20 unique problems were created and tested with randomly generated, complete, $|G \times G|$ preference matrices. The random generation of problems led to an extremely large spread of final answers across problems. To summarize the results, we compared the five approaches, and report the numbers of problems (out of 20) on which each algorithm obtained the highest evaluation (highest summed preferences at the tables, with all the constraints being met). The results are shown in Table II. Out of the 20 trials, modeling the search space helped in 15 trials. The next line of the table give the number of trials (out of 20) that Deep-Opt-5-layers outperformed the other 4 methods (including the other Deep-Opt network). The last line does the same for Deep-Opt-10-layers.

C. *Graph Bandwidth*

Given a graph with V vertices and E edges, the graph bandwidth problem is to label the vertices of the graph with unique integers so that the difference between the labels between any two connected vertices is minimized. Formally, as described in [34] [35], label the p vertices v_i of a graph G with distinct integers $f(v_i)$ so that the quantity $\max\{\,|f(v_i) - f(v_j)| \,:\, v_i v_j \in E\,\}$ is minimized (E is the edge set of G). Interest in this problem stems from a variety of sources, including constraint satisfaction [36] and minimizing propagation delay in the layout of electronic cells.

The solution is encoded as: each vertex is assigned a real-valued parameter (full solution is $|V|$). The vertices are sorted according to their respective assigned values. The integers $[1..V]$ are then assigned to the vertices in their respective sort position. Once each vertex has an integer assignment, the maximum difference between the assignments of connected edges is returned. The results are shown in Table III. This is a particularly difficult problem; ties are shown in parentheses.

D. *Graph-Based Constraint Satisfaction*

Constraint Satisfaction has numerous real-world applications. We recently used it for resource allocation and job scheduling. Is is presented here in its simplest form. There are $P = 100$

TABLE III
GRAPH BANDWIDTH:NUMBER OF TIMES EACH ALGORITHM OUTPERFORMED ALL OTHERS (OUT OF 20). TIES SHOWN IN PARENTHESES.

	NASH-1	NASH-2	NASH-3	Deep-Opt-5	Deep-Opt-10
		Wins=6		***Wins=18***	
Overall Best	0	0	6	10	8
Deep-Opt-5 Outperformed	20	20	12 (3)	-	6 (6)
Deep-Opt-5 Difference Significant?	>99%	>99%	96%	-	no
Deep-Opt-10 Outperformed	20	20	12 (2)	8 (6)	-
Deep-Opt-10 Difference Significant?	>99%	>99%	96%	no	-

TABLE IV
REAL VALUED GRAPH-BASED CONSTRAINT SATISFACTION: NUMBER OF TIMES EACH ALGORITHM OUTPERFORMED ALL OTHERS (OUT OF 20)

	NASH-1	NASH-2	NASH-3	Deep-Opt-5	Deep-Opt-10
		Wins=2		***Wins=18***	
Overall Best	0	0	2	13	5
Deep-Opt-5 Outperformed	20	20	16	-	14
Deep-Opt-5 Difference Significant?	>99%	>99%	>99%	-	no
Deep-Opt-10 Outperformed	20	20	17	6	-
Deep-Opt-10 Difference Significant?	>99%	>99%	>99%	no	-

TABLE V
DISCRETE VALUED GRAPH-BASED CONSTRAINT SATISFACTION: NUMBER OF TIMES EACH ALGORITHM OUTPERFORMED ALL OTHERS (OUT OF 20)

	NASH-1	NASH-2	NASH-3	Deep-Opt-5	Deep-Opt-10
		Wins=3		***Wins=17***	
Overall Best	0	0	3	13	4
Deep-Opt-5 Outperformed	19	19	16	-	14
Deep-Opt-5 Difference Significant?	>99 %	>99 %	98%	-	no
Deep-Opt-10 Outperformed	19	19	14	6	-
Deep-Opt-10 Difference Significant?	>99 %	>99 %	92%	no	-

real-value parameters in the range [0,1.0]. These parameters are assigned to the vertices in a graph. The graph contains 2,000 randomly chosen, directed, edges which specify a constraint that the origination-node must hold a value greater than the destination-node. The optimization problem is to assign values to the nodes such that as many of the 2,000 constraints are satisfied as possible. If the constraint is not met, the error is the absolute difference in the two values. The error, to be minimized, is summed over all constraints. The results are shown in Table IV.

E. Graph-Based Discrete Constraint Satisfaction

In this variant of the previous graph-based constraint satisfaction problem, the exact same setup is used as in Section III-D, however, each node may only take on 1 of 16 letters – $A..P$. In terms of the real-world application of job scheduling mentioned above, in this version of the problem, jobs can enter the system only at specific, synchronized times. This makes the problem closer to a selection problem (where one of the 16 values is selected for each of the nodes) as compared to the previous instantiation where a real value was assigned to each node.

Though conceptually a small difference from the above encoding, discretization has enormous ramifications in the solution encoding. The simplest encoding is to use 100 real-valued outputs (one for each node) and divide the [0,1] space into 16 evenly spaced regions, each assigned to a single letter. However, this encoding does not work — discretizing the real values in this way does not allow derivatives/small changes in training to effect actual changes to the evaluation [29]. Instead, the encoding employed is similar to the Reception-Party-Seating task (Section III-B). Each vertex in the graph is assigned 16 real-valued parameters, each corresponding to a letter $A..P$ (In contrast, recall that with the encoding described in Section III-D, each vertex was assigned a real-value). In each set of 16, the maximum value is found and the corresponding letter assigned to the vertex. In sum, for a 100 node graph, 1,600 parameters are used. Once the nodes are assigned values, the evaluation proceeds as described in Section III-D.

F. Two Dimensional Layout Problems

This section highlights the limitations of the Deep-Opt approach. A number of problems which broadly encompassed

TABLE VI
RESULTS FOR MINIMIZING CROSSING ON 20 RANDOMLY GENERATED PROBLEMS

	NASH-1	NASH-2	NASH-3	Deep-Opt-5	Deep-Opt-10
	Wins=9			***Wins=11***	
Overall Best	1	5	3	8	3
Deep-Opt-5 Outperformed	16	11	14	-	15
Deep-Opt-5 Difference Significant?	>99%	no	97%	-	no
Deep-Opt-10 Outperformed	14	6	13	5	-
Deep-Opt-10 Difference Significant?	>97%	no	no	no	-

the task of two dimensional layout did not statistically improve with search space modeling. Two problems are detailed here.

1) Minimizing Crossings: The goal is to find a planar layout of a graph's nodes that minimizes edge crossings. See [37] for more details. In general, the edges can be drawn in any shape. For simplicity, here, the edges are drawn only with straight lines, this is termed the *rectilinear crossing number*.

For our tests, each node was represented with two parameters (x,y coordinates). Small graphs were tried with 25 nodes. This yielded a solution encoding of 50 real-values, which specified the coordinates of each point on a plane. Each graph had 50 randomly chosen connections. 20 randomly generated problem instantiations were attempted.

One of the peculiar findings is that NASH-2 outperformed NASH-3. In most previous experiments, this was reversed. NASH-2 received a higher score in 12 out of the 20 problems (the scores, as measured by a standard t-test were statistically different with $p = 0.96$). Although left for future exploration, it is worth investigating what insight this gives about the search space? If searching around the current best does not yield as good results as randomly starting over, does the search space have more or less optima, or are the local optima further spread apart, deep, etc? Returning to the experiments, because Deep-Opt can just as easily be applied to NASH-2 as NASH-3, here, we used it to "wrap" NASH-2.

2) Image Approximation via Triangle Covering: This is the only problem in the paper for which the parameters for NASH were changed. Accordingly, Deep-Opt also used the same parameters. For this problem, there is an intensity target image, I, that is $N \times N$ pixels. The goals is to find T triangles and their intensities that approximate the image. Specifically, each triangle must specify three vertices between [0,N] in both X&Y and an intensity value. The triangles are drawn onto an initially empty canvas. The triangles may overlap; their intensities are additive. After all T triangles are drawn, the resulting image is scaled back into the appropriate space (0 .. 255 pixel intensities), and compared, pixel-wise, to the original image. The L_2 distance is to be minimized. This is a particularly difficult/interesting problem when T is small.

To set the parameters for NASH and Deep-Opt, we tested this problem with 50 triangles, trying to approximate an intensity based crop of "The Scream" by Edvard Munch. Each triangle was encoded as 7 parameters: 6 for the (x,y) coordinates of three vertices and 1 for the intensity. With 50 triangles, there were a

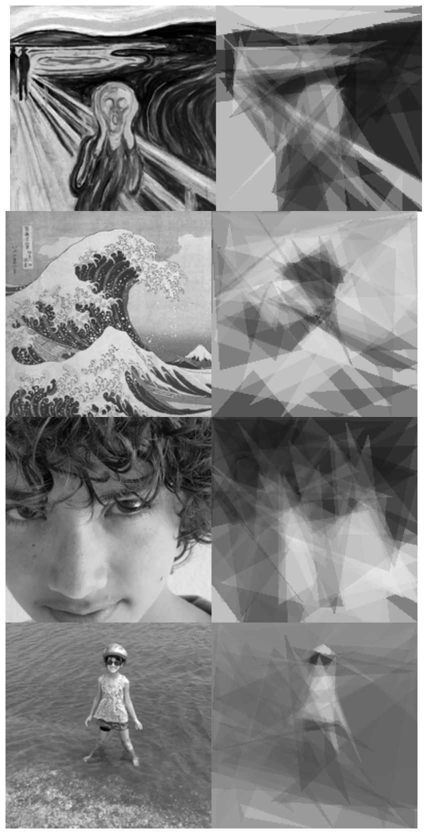


Fig. 7. In each pair of images, the left is the original image. The right is the reconstruction of the image using the 50 triangles found by Deep-Opt.

total of (50×7) 350 parameters in the solution encoding. Once the parameters were set, 3 other images, shown in Figure 7, were also attempted with the same parameter settings. Averaged over 10 trials on each image, the results were not statistically significantly different for any algorithm; although all results were surprisingly detailed (Figure 7).

IV. ALTERNATIVE UNDERLYING SEARCH ALGORITHMS

In this paper, we have coupled the use of deep-net modeling with an extremely simple, fast, localized search algorithm, NASH. Let us also consider the possibility of using alternate search algorithms with the same modeling procedures. Alternative search heuristics such as simulated annealing [38] and TABU search [39] can easily be substituted as they search

TABLE VII
GA PERFORMANCE ON GRAPH-BASED CONSTRAINTS PROBLEM (20 INSTANTIATIONS)

Approach	Performance	Wins
Standard GA (population 50)	4817	0
Deep-Opt-GA (population 50)	5120	20
Standard GA (population 100)	4825	0
Deep-Opt-GA (population 100)	5131	20

neighborhoods around a single point in a manner similar to NASH. More interesting is the use of probabilistic models with very different search paradigms, such as genetic algorithms.

Recall that, unlike NASH, genetic algorithms work from a population of points. Members of the population are created with recombination operators (crossover) that combine the elements of two or more candidate solutions. The newly created solutions are further randomly perturbed (mutated) to reveal the 'children' solutions that are the candidate solutions to evaluate next [5]. Numerous variations of GAs and task-specific operators are possible and have been explored in the research literature. Next, we perform a tests using a a simple-GA with typical parameter settings for static optimization problems: population-size 50/100, crossover: uniform, mutation rate: 2%, generational populations, elitist selection: on.[1]

We test the GA with two population sizes (50 & 100) run for an equivalent number of function evaluations as all of the previous runs with hillclimbing (500,000). Additionally, as with the previous runs, the GA was restarted after 10,000 evaluations. In the standard GA, the initial population is comprised of candidate solutions that were randomly generated. In the Deep-Opt-GA, the initial population of candidate solutions is entirely generated from the back-driven neural network model.

This approach was tested on the same real-valued constraints problems described in Section III-D. The results are shown in Table VII (problem is formulated as a *maximization* problem). Deep-Opt-GA outperformed the random initialization on all 20 instantiations attempted, for both sets of trials (with population size 50 and 100). To summarize the findings in this section, using models to guide search can help even in search heuristics that operate from more than a single point – those that are population-based, such as genetic algorithms.

V. DISCUSSION AND FUTURE WORK

We have presented a novel method to incorporate deep-learning with stochastic optimization. It is the next instantiation of intelligent model-based stochastic optimization and follows in the tradition of the probabilistic model based optimization approaches from the last two decades of research. An important aspect of this work is that *a priori* information about the problem is minimal in setting the form of the model. In this study, two multi-layer feed-forward networks with 5 and 10 hidden layers were used on all of the problems with no problem-specific modifications. With the judicious use of early-stopping in training, the potential downsides for overtraining were overcome.

Two other extensions not presented in this paper due to space restrictions have also been explored: (1) discrete parameter encodings: In contrast to the real-valued parameters explored in this paper, discrete parameter encodings can be handled through techniques used to binarize hidden layers in neural networks, such as stochastic sigmoid units [40]. Work towards this is shown in [29]. (2) the role of scaling the target outputs during training [29]: One can imagine that instead of scaling the target outputs to values in the range [0.0,1.0], they were scaled to [0.0,Z], where Z < 1.0. In this approach, the highest scoring $s \in S$ will have a value of Z. When the network is back-driven, it is still driven to find solutions that produce a 1.0 in the output. This effectively attempts to create new solutions that are *better than*, not just equal to, those seen so far. The success of this approach is pinned on the network's successful extrapolation of the underlying search surface to regions of better performance. Many versions of this were tried, where Z was set to [0.2,0.5,0.8,0.9,0.95, and 0.99] in various experiments. Although the results are preliminary, setting the Z to 0.95 and higher had little effect on the results, when compared to setting Z = 1.0. Setting Z in the low range often hurt performance.

Many of the advances from the rapidly evolving field of deep learning can be directly incorporated into this work (such as network shrinking, rapid training, regularization schemes, etc.), as continuously training networks is the core of the learning components. Outside of the deep-learning advancements, three avenues for future explorations are given below.

First, the overarching goal of this paper was to concretely demonstrate the integration of neural network learning with optimization, not to promote this system as finalized optimization system. To make a finalized optimization system, further exploration of the algorithm's robustness and behavior is warranted. For example, there are many parameters in this system. In this study, we have found that the results are most sensitive to the size of $|S|$ and to the decision of when to restart training the network from scratch – this happens when new samples are added to $|S|$ and the network fails to accommodate them in learning (i.e. the error on the samples does not reduce). The beneficial effects of regularization in training may be especially pronounced in this application as networks are constantly being incrementally trained with changing data. Also, we have found a class of problems for which we have not observed a benefit of using the modeling (see Section III-F). Are the problems too easy or too hard, or is an alternate representation needed?

Second, an alternative to using the network back-driving technique to generate candidate initialization points is to simply use the network as a proxy evaluation function for hillclimbing.

[1]Alternate operators and operator application rates may yield improved optimization algorithms for each problem. Our intent is only to show that the Deep-Opt framework can be as easily wrapped around multiple-point search-based algorithms, such as GAs, as well as single-point search based algorithms, such as Hillclimbing.

In this approach, hillclimbing (NASH) is conducted directly on the model's output. After every perturbation, the new candidate-solution is passed through the network to measure it's *estimated* performance. This, unlike network back-driving, does not take advantage of the fact that the network is differentiable, and rather only uses it as a proxy for the real evaluation function. However, it may reveal parts of the search space that back-driving does not.

The third, and perhaps the most speculative, direction is to determine if there are transferable features that are learned between different instantiations of the same problem. For example, with respect to the triangle-covering problem presented in Section III-F2, once we have learned how to evaluate how well a set of triangles reproduces an image, is learning the evaluations for the next image easier? One can imagine that low level primitives of how to draw triangles, if they are indeed learned by the network, may be reusable. Even if there is little transference in the current implementation, exploring problem transference has enormous potential to make this system automatically more intelligent with time.

References

[1] S. Baluja, "Population-based incremental learning. a method for integrating genetic search based function optimization and competitive learning," CMU-CS-94-163. Carnegie Mellon University, Dept. of Computer Science, Tech. Rep., 1994.

[2] A. Juels, *Topics in black-box combinatorial optimization*. University of California, Berkeley, 1996.

[3] H. Mühlenbein and G. Paass, "From recombination of genes to the estimation of distributions i. binary parameters," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1996, pp. 178–187.

[4] G. R. Harik, F. G. Lobo, and D. E. Goldberg, "The compact genetic algorithm," *IEEE transactions on evolutionary computation*, vol. 3, no. 4, pp. 287–297, 1999.

[5] D. E. Goldberg, *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Publishing Company, 1989.

[6] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[7] K. De Jong, "Genetic algorithms: a 30 year perspective," *Perspectives on Adaptation in Natural and Artificial Systems*, vol. 11, 2005.

[8] V. Kvasnicka, M. Pelikán, and J. Pospichal, "Hill climbing with learning (an abstraction of genetic algorithm)," in *Neural Network World, 6*. Citeseer, 1995.

[9] M. Hohfeld and G. Rudolph, "Towards a theory of population based incremental learning," in *Proceedings of the International Conference on Evolutionary Computation*, 1997.

[10] R. Rastegar, A. Hariri, and M. Mazoochi, "A convergence proof for the population based incremental learning algorithm," in *17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'05)*. IEEE, 2005, pp. 387–391.

[11] C. Gonzalez, J. A. Lozano, and P. Larrañaga, "The convergence behavior of the pbil algorithm: a preliminary approach," in *Artificial Neural Nets and Genetic Algorithms*. Springer, 2001, pp. 228–231.

[12] C. González, J. A. Lozano, and P. Larranaga, "Analyzing the population based incremental learning algorithm by means of discrete dynamical systems," *Complex Systems*, vol. 12, pp. 465–479, 2000.

[13] J. S. De Bonet, C. L. Isbell, P. Viola *et al.*, "Mimic: Finding optima by estimating probability densities," *Advances in neural information processing systems*, pp. 424–430, 1997.

[14] S. Baluja and S. Davies, "Fast probabilistic modeling for combinatorial optimization," in *AAAI/IAAI*. Madison, WI, USA, 1998, pp. 469–476.

[15] C. Chow and C. Liu, "Approximating discrete probability distributions with dependence trees," *IEEE transactions on Information Theory*, vol. 14, no. 3, pp. 462–467, 1968.

[16] S. Baluja and S. Davies, "Using optimal dependency-trees for combinatorial optimization," in *International Conference on Machine Learning (ICML)*, 1997, pp. 30–38.

[17] D. Heckerman, "A tutorial on learning with bayesian networks," in *Innovations in Bayesian networks*. Springer, 2008, pp. 33–82.

[18] J. Pearl and S. Russell, "Bayesian networks," *Department of Statistics, UCLA*, 2000.

[19] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "Boa: The bayesian optimization algorithm," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*. Morgan Kaufmann Publishers Inc., 1999, pp. 525–532.

[20] J. Yao, Y. Kong, and L. Yang, "Bayesian optimization algorithm based on incremental model building," in *International Symposium on Intelligence Computation and Applications*. Springer, 2015, pp. 202–209.

[21] R. Etxeberria and P. Larranaga, "Global optimization using bayesian networks," in *Second Symposium on Artificial Intelligence (CIMAF-99)*. Habana, Cuba, 1999, pp. 332–339.

[22] M. Pelikan, K. Sastry, and E. Cantú-Paz, *Scalable optimization via probabilistic modeling: From algorithms to applications*. Springer, 2007, vol. 33.

[23] J. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *Journal of Machine Learning Research*, vol. 1, no. Nov, pp. 77–112, 2000.

[24] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[25] A. Linden and J. Kindermann, "Inversion of multilayer nets," in *Neural Networks, 1989. IJCNN., International Joint Conference on*. IEEE, 1989, pp. 425–430.

[26] L. Gatys, A. S. Ecker, and M. Bethge, "Texture synthesis using convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 262–270.

[27] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *arXiv preprint arXiv:1508.06576*, 2015.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[29] S. Baluja, "Deep learning for explicitly modeling optimization landscapes," *CoRR*, vol. abs/1703.07394, 2017. [Online]. Available: http://arxiv.org/abs/1703.07394

[30] A. Juels and M. Wattenberg, "Stochastic hillclimbing as a baseline method for evaluating genetic algorithms," in *Proceedings of the 1995 Conference on Neural Information Processing Systems (NIPS)*, vol. 8, 1996, p. 430.

[31] S. Baluja, "An empirical comparison of seven iterative and evolutionary function optimzation heuristics," Computer Science Department, Tech. Rep. CMU-CS-95-193, September 1995.

[32] M. Mitchell, J. H. Holland, and S. Forrest, "When will a genetic algorithm outperform hill climbing," in *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds. Morgan-Kaufmann, 1994, pp. 51–58. [Online]. Available: http://papers.nips.cc/paper/836-when-will-a-genetic-algorithm-outperform-hill-climbing.pdf

[33] Chriddyp. (2017) Asyncho from 0.00 to 0.05. [Online]. Available: https://plot.ly/~chriddyp/

[34] Wikipedia. (2016) Graph bandwidth. [Online]. Available: https://en.wikipedia.org/wiki/Graph_bandwidth

[35] P. Z. Chinn, J. Chvátalová, A. K. Dewdney, and N. E. Gibbs, "The bandwidth problem for graphs and matrices—a survey," *Journal of Graph Theory*, vol. 6, no. 3, pp. 223–254, 1982.

[36] R. Zabih, "Some applications of graph bandwidth to constraint satisfaction problems." in *AAAI*, 1990, pp. 46–51.

[37] Wikipedia. (2016) Crossing number (graph theory). [Online]. Available: https://en.wikipedia.org/wiki/Crossing_number_%28graph_theory%29

[38] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[39] F. Glover, "Tabu search-part i," *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989.

[40] T. Raiko, M. Berglund, G. Alain, and L. Dinh, "Techniques for learning binary stochastic feedforward neural networks," *arXiv preprint arXiv:1406.2989*, 2014.