Optimizing Optimizations: a Two-Stage Neural Network Approach for Leveraging Optimality in Time-Sensitive Solutions

Matthew McLaughlin
Fires Battle Lab
Fort Sill, Oklahoma
matthew.b.mclaughlin4.civ@mail.mil

ABSTRACT

Time-sensitive optimization problems challenge the balance of optimality and the speed of relevance—including several commercial and military operations. In this research, we explore a use case on an optimized artillery aimpoint algorithm. This aimpoint algorithm is time sensitive because as targets move, the optimal aimpoint solution for engaging these targets changes. With any software, algorithms and their parameters add complexity and may be hard to interpret. In optimization, these parameters include gene pool size, selection, mutation rates, swarm spread and density, step increment, the type of convergence algorithm, and so on. To prevent susceptibility of human misconception about optimization parameters and algorithms, meta-optimizations offer techniques to select ideal optimization parameters. Even the computer hardware, network architecture, and version numbers of software libraries influence the computational performance of generating solutions. Therefore, meta-optimizers should be trained using an operational profile and be adaptable to rapidly changing software by training them within the software release pipeline. This research explores the process and benefits of creating a tunable meta-optimizer. In the first stage, we start with a designed experiment around a resource intensive artificial intelligence or optimization application. This experiment collects responses (primarily processing time and local optimums) associated with a collection of input factors (the inputs of the objective function) and control factors (the optimization parameters). The goal of this data collection is to train a meta-model for the real optimization software. In the second stage, a meta-optimizer is trained. It takes the input factors joined with tuning parameter(s) to leverage our control factors. The meta-model is used to evaluate the tunable meta-optimizer in a reinforcement learning environment. The primary benefit for this tunable meta-optimizer design is to provide the operator control over the trade-off between optimality and time-sensitivity—a critical feature for managing solution quality and deadlines.

ABOUT THE AUTHOR

Matthew McLaughlin is a Lead Computer Scientist at the Fires Battle Lab in Fort Sill, OK. He has a B.A. in Mathematics, and B.S. in Computer Science from Cameron University in Lawton, Oklahoma. He has also earned a Masters of Engineering in Modeling and Simulation in May of 2020 from Arizona State University. He applies mathematical solutions to complex modeling problems which supports combat analysis and constructive simulation experiments. He received the Army Individual M&S Award in 2017 for his software innovations supporting terrain optimizations.

Optimizing Optimizations: a Two-Stage Neural Network Approach for Leveraging Optimality in Time-Sensitive Solutions

Matthew McLaughlin
Fires Battle Lab
Fort Sill, Oklahoma
matthew.b.mclaughlin4.civ@mail.mil

INTRODUCTION

Optimization and artificial intelligence are just a few tools to improve decision making. Both have been around for many years, yet they seem to continue only being emerging concepts. This is likely because greater computing power and algorithms are needed to make any use of these concepts. In response, we typically employ the use of GPUs, supercomputers, and crowd processing. However, if we look at how we are training artificial intelligence or converging on optimal solutions, we may find that the computation and processes are not used efficiently. Academia has a wide area of research focused on education and training for humans with great efficiency. However, on the computation side, we tend to add more processors or employ heuristics. Meta-optimization is a concept used to optimize on optimizations by reducing the resource requirement—such as time and number of processors—needed for the optimization process. Every optimization problem (including the training of artificial intelligence systems), uses an algorithm to solve the problem. Algorithms introduce complexity. This necessary complexity can help or hinder finding a solution efficiently. Some algorithms are good for some types of problems and poor at others. With each algorithm often comes a collection of parameters. Often humans will create assumptions and misconceptions about what these parameters mean and how they impact the speed and quality solutions.

Consider some types of optimization processes. Most include some form of convergence threshold and repetition to avoid getting trapped around local solutions. By increasing the number of repetitions, we will find solutions closer to the absolute optimal. Genetic algorithms usually have a mutation rate, gene pool size, a proportion to select from each generation, and a number of generations. Other mathematical convergence algorithms use a step size to "walk" toward solutions. These parameters usually mean nothing to the end user. They will typically desire 100% optimal solutions instantly. However, processing time and reduced optimality are both artifacts of computation and the limitations of current algorithms. The proposed two-stage neural network approach to tune meta-optimization allows the end-user to control a tuning parameter: such as the amount of processing time or the amount of lost optimality they are willing to accept as shown by the tuning parameter(s) input in Figure 1. The optimization parameters—also called control factors in this research—are generated based on input factors from the problem being solved and are therefore hidden from the end-user.

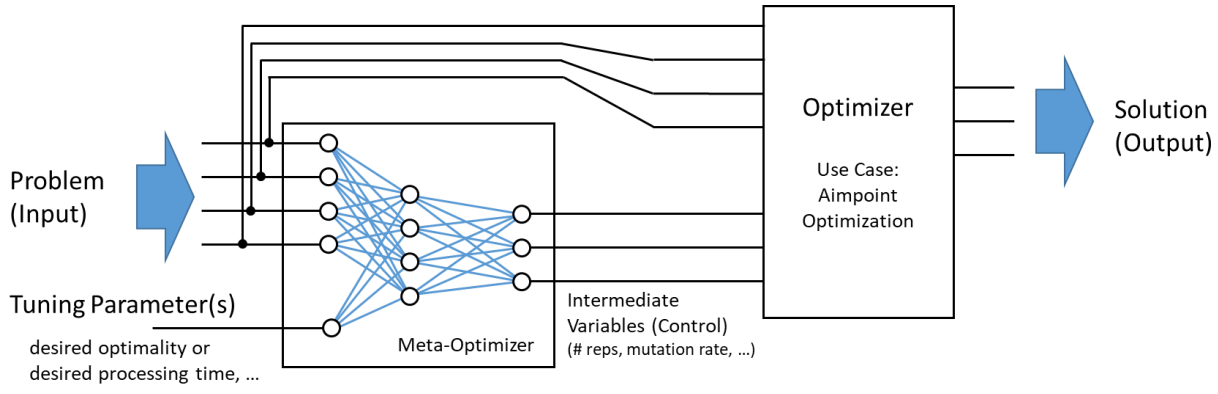

Figure 1. Operational Use for the Proposed Meta-Optimizer

BACKGROUND

Many problems can be solved on a super computer and their results are received within a reasonable amount of time. If solutions are needed faster, more processors can be added. However, some solutions are needed without vast computing power. Consider computers within warfare. Communication, hardware, and processing are limited. The artificial intelligent systems that ingest sensor readings amongst other data elements must interpret the data, find patterns, and advise commanders; they must do so without vast resources. The ability to consider trade-off in meta-optimization is critical for most types of short-term planning in a world that constantly changes around us, from optimizing the next five minutes of manufacturing as orders come in, to optimizing the next five minutes of airport activities, to routing and logistics as weather and road conditions change, even the software and platforms these solutions are generated on are subject to change. Our use case focuses on a military application that optimizes aimpoints to maximize the effective destruction in a target area. It demonstrates the reality that optimal solutions are constantly challenged by the speed of relevance. As targets move, the optimal way of engaging them changes. A perfect solution delivered an hour late is likely useless to the commander.

All meta-optimizer training should consider an operational profile—how the operators are using the software and the environment the software is expected to run on. The type of processor, memory, and availability of graphics processor units (GPUs) or single instruction/multiple data (SIMD) instruction sets, also impact the speed and technique at which solutions are generated. As software and third-party libraries change, efficiencies within the software changes. Therefore, retraining the meta-optimizer should be done regularly within the software release pipeline.

The end-users for optimization software generally do not need access to optimization parameters. If they did, consider how they would use them. Our approach assumes they would use them to leverage optimality versus processing time (and/or other resources) and that their adjustments would be more-or-less arbitrary and inefficient. This is why we reduce all optimization parameters to one or a few tuning parameters. Given that optimal solutions for a time-sensitive optimization problem degrades over time, another benefit for this tuning parameter may be to further optimize when the degradation of optimal solutions is quantifiable. Then, even these tuning parameters can be hidden from the end-user. In our aimpoint optimization use case, this degradation could be made proportional to the speed and unpredictable maneuvering of the target(s).

Meta-optimization is not a new topic, but little research has been spent on leveraging it to meet the needs of the end-user. Our approach trains a collection of neural networks to understand the relationships between inputs of the objective function and the optimization parameters by running an automated designed experiment with the optimizer software. The two-stages of this process are:
1. Use supervised training to train a **cost model** from data collected on an operational profile and automated experimentation. The purpose of this model is so that we can quickly score the meta-optimizer networks without employing the use of the real software.
2. Use reinforcement learning (or transfer learning) to train a **meta-optimizer** on the previous cost model to produce effective optimization parameters.

RELATED WORK

Lang and Stanley (2013) propose that the meta-optimizer can be an evolving neural network. Nezhad and Mahlooji (2014) use a meta-model to approximate expensive and stochastic objectives functions. We use these concepts to develop our cost model in the first stage of this research, which is intended to replace the real optimizer in the second stage. The fitness function for our meta-optimizer has two goals, minimize resource requirements and maximize optimality. These objectives are meld into one fitness function (Łapa, 2014), such that the meta-optimizer network should be punished against either one or both metrics depending on the desired optimality.

BUILDING THE COST MODEL

Before we can train a meta-optimizer model, we need a cost model. The inspiration for using as the cost model as a meta-model for the real optimizer was based on the assumption that expected processing time and optimality would be simple enough to train and that leveraging the meta-optimizer on the actual software would not be feasible due to its stochasticity and the amount of processing time it would realize.To extract this meta-model from the real optimizer,

we employ a design of experiments. For this effort, we need to identify our input and control factors, measurements of effectiveness (response factors), and an experiment plan:

1) Identify input factors – these factors feed the objective function. These can be numerical or categorical, but for more complex inputs such as text, audio, images, and video, these should be aggregated metrics suspected of having an impact on any of the responses. If the inputs draw from scenarios, the scenario should be a blocking factor and measurable nuisance factors should be extracted from these scenarios. To prevent confounding, the degrees of freedom introduced by the scenario should exceed the total degrees of freedom of these measurements. Our use-case employs an image to describe the target area. We have six target scenarios and measure a couple of properties about the target area.
2) Identify control factors – these are the optimization parameters offered by the optimizer software.
3) Identify responses – outputs that we want to minimize or maximize. Our use case considers optimality (the degree of closeness to a global optimum) and processing time. Control factors are selected with a goal to minimize processing time and maximize optimality.
4) Identify the response we are trying to leverage. This will serve as the tuning parameter for both the cost model and meta-optimizer.
5) Identity the scoring function. This function combines the tuning and response factors into a single score. These are used to evaluate meta-optimizers in the next stage. Some scoring options are discussed later.

The definition of optimality depends on the context of the optimization problem, but it usually measures the degree of closeness to a global optimum, such as a ratio of local maximum to global maximum. Our use-case uses the percent of target area destroyed in a nine-count logarithm. $\#9s = -\log_{10}(1 - local/global)$. A nine-count literally counts the leading 9s in a proportion: 0.9 is a 1-nine solution, 0.99 is a 2-nine solution, 0.999 is a 3-nine solution, and so on. If a proposed aimpoint solution destroys 12% of a target, but the best possible solution destroys 13%, we would have $\#9s = -\log_{10}(1 - .12/.13) = -\log_{10}(1 - .923) = 1.114$ nine-count optimality. This definition of optimality creates a unique problem in optimization because we assume that we know global solutions.

If global optimums are not known, consider if they can be obtained using brute-force algorithms. If not, we need to track local optimums and retain the best solution in a global optimums database. Global optimums represent the best solution we can ever hope to obtain given a set of input factors and regardless of control factors. To accomplish this efficiently, we need to limit the number of input configurations we are training in the cost model. As we experiment with multiple control configurations (the optimization parameters), we may encounter local solutions that outperform the current global solution. When this inevitably happens, we simply update the global optimum for that input configuration and continue. The following outlines this strategy:

1) Create a design for input configurations: for a few input factors, this can be a full factorial design, but for several input factors, a partial or custom design should be used.
2) Turn each input configuration from this design into a level of a new multi-level factor and generate a new design using it and all the control factors. Expand the design to include the assigned input configuration. The effect of steps 1 and 2 limits the number of input configurations and provides a sizable population to be genetically selected from for each input configuration.
3) Run the experiment to collect local optimums.
4) For each input configuration, save the best local optimum in the global optimum database.
5) Genetically select the best local optimums for each input configuration to repopulate the next design. This has the goal of focusing strictly on optimization parameters capable of producing database updates.
6) Run the refined design. If any local optimum out-performs the global optimum, update the database.
7) Repeat steps 4 through 6 until updates to the database become negligible and rare.
8) Add a column to all of the observations collected to represent optimality. It should combine the local observation against the global data base entry.

Software changes inside the objective function invalidate the global optimums database because the stored global result of the objective function may no longer be obtainable given the same input. However, software changes isolated outside the objective function and its inputs can reuse the existing database. For instance, when new convergence algorithms or heuristics are introduced, we can test their significance and applicability without painstakingly rebuilding the global optimums database. If optimality is not a response variable, we do not need a global database, and can therefore simply design and run an experiment across all input and control factors.

The data at this point could train a model to predict optimality and other responses such as processing time for any configuration of input and control factors. This is helpful for informing users what to expect in terms of processing time and optimality for their input. Now that results have been collected, we need to score all of the observations. Some options for scoring are:

$$\text{Score} = (\text{Desired\_Optimality} - \text{Observed\_Optimality})^{+} + \alpha \cdot log(\text{Observed\_Proc\_Time})$$

or

$$\text{Score} = [log(\text{Observed\_Proc\_Time}/\text{Desired\_Proc\_Time})]^{+} - \alpha \cdot \text{Observed\_Optimality}$$

The notation $(X)^{+}$ returns the value of X if X is positive, or 0 if X is not positive. This welcomes better than requested responses, but does not improve the overall score. For instance, if a 99% solution is requested but a 90% solution is observed, this would have a difference of 1-nine optimality. In contrast, if a 99.9% solution is observed, the score is unchanged. If observed optimality satisfies the desired optimality, our goal is to minimize processing time. Likewise the second equation maximizes optimality if the requesting processing time is met. The α coefficient describes the trade-off we accept between the tuning term and the remaining response. A logarithm on all processing times will scale up differences with larger numbers and scale down with smaller numbers. For example, a five second difference in processing time is worse if the target time frame was fifteen seconds than if the time frame was fifteen days. These are just some examples of scoring, but other optimization problems may need to leverage other types of resources, such as memory, hard-drive space, and the cost of running web-service modules. Other scoring options may also only care to reduce processing times above a few seconds and ignore time less than this threshold.

Scoring functions introduce the tuning parameter(s). Each tuning parameter requires an additional input column in our results table. To efficiently perform this operation, the results collected from experimentation should be crossed with each level of each tuning parameter we are interested in. A score response column is then added to the results table using the desired scoring function and required inputs. To improve the quality of the data, we should focus on the lowest scores for each input configuration. After all, the meta-optimizer will converge around these lower scores, so they should be as accurate as possible. We do this by generating more experimental runs for each input configuration using a genetic algorithm to select the best control factor levels that minimize the score for each input configuration and refine the experiment and collect data around minimal scores.

All observations collected thus far are used to train the cost model shown in Figure 2. This will serve as a meta-model for the real optimizer in Figure 1. However, if we analyze this training data and screen for main effects and interactions, we may find that some input/control factors have no effect on the score. We can therefore choose to remove them and/or hard-code them. If a control factor has main effects, but no interaction terms, we may select the level that produces the best score and exclude it from the cost model. However, if interaction terms are found or there is a significant lack of fit, we should proceed with supervised training of the proposed cost model. If the model works as an accurate predictor of score, we will proceed to the second stage.

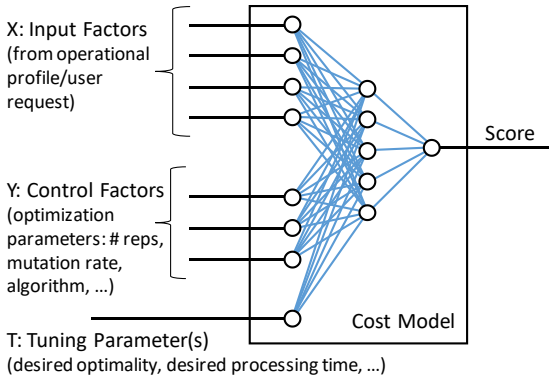

Figure 2. Neural Network for the Cost Model

BUILDING THE META-OPTIMIZER

In the second stage, our efforts shift to training the meta-optimizer. If we substitute the optimizer in Figure 1 with the cost model from Figure 2, we have the training environment shown in Figure 3. Training the meta-optimizer depends on the machine learning tools one has available and the operational use of the meta-optimizer. If the trained network can be combined with an untrained network, transfer learning could be utilized. If not, then one can fall back on reinforcement learning. Our research will focus primarily on reinforcement learning with a deterministic fitness function. We train the meta-optimizer by feeding just the input factors and tuning parameter(s) from the cost model training data into the untrained meta-optimizer. We then employ non-gradient optimization on the network's weights until the total score is minimized.

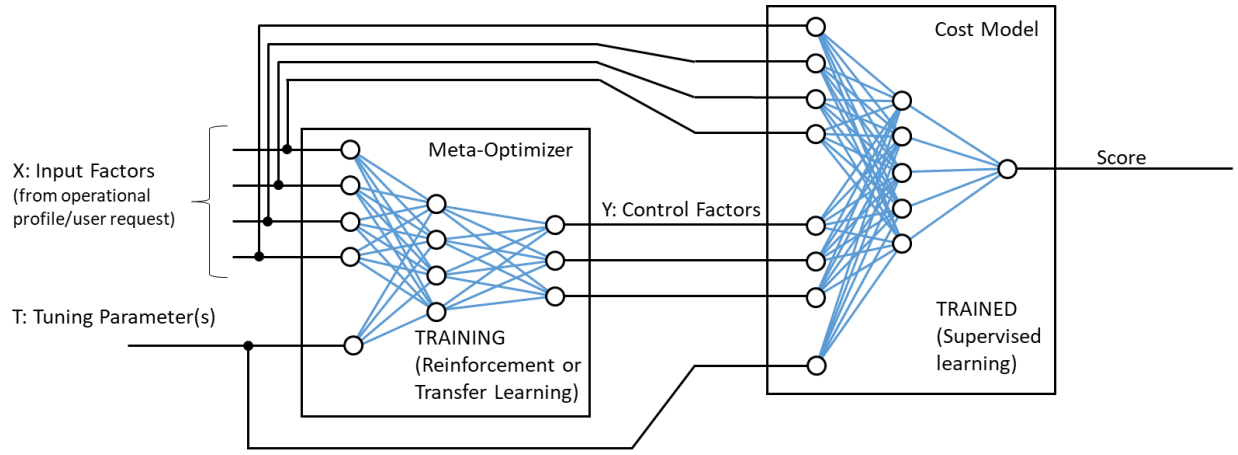

Figure 3. Training Environment for the Tunable Meta-Optimizer

The meta-optimizer output must be adapted for operational use before it is ingested into the cost model (or actual software). The numerical control factors should be clamped to their defined limits and categorical control factors should be selected by setting the highest category to one and setting all others to zero. This way, the meta-optimizer will be better trained on its operational use.

APPLICATIONS

Optimizing Training of Neural Networks

There are a lot of heuristics in the topic of supervised training of neural network. For instance, smaller partitions of the dataset might be fed into the network and cycled through each iterative step toward the final solution. Different convergence algorithms exists, such as simulated annealing and backpropagation. On the network side, there is structure, sizes of layers, and activation functions. These all create complexity, new options, and new configurable parameters. Using an operational profile and tunable meta-optimization may give insight into the trade-off between what works fast and what works well.

Optimizing Compiler Optimizations

Compilers have a wide range of techniques to increase the performance of resulting executables including: scheduling order of independent tasks, prefetching, register allocation, function inlining, and loop unrolling. Stephenson et al. (2003) warns that compiler writers must rely on inaccurate abstractions for the intricacies of modern architectures and compilers and that meta-optimization creates a better model. The compiled code will generate the same output given the same input, so optimality—difference between local and global optimums—as a tunable parameter does not fit. However, we can tune between other resources such as executable size, number of instructions, and processing time.

USE CASE—AIMPOINT OPTIMIZATION

Aimpoints are traditionally used in artillery to spread effects over a target area (McLaughlin, 2019). A target in this context covers a geographic area, a collection of enemy detections, or a single inaccurate detection that we will use multiple rounds to destroy. One can imagine, if all rounds were aimed at the center of the target, it will obliterate one spot, but leave much of the surrounding target untouched. On the other extreme, if we spread aimpoints too far, it will spill effects outside the target area. The placement of each round to balance overlap and spilling of effects is an optimization problem.

Modeling

Before diving into finding the solutions, we must first understand how artillery fires are modeled in the software for this use case. Start off with where we want to shoot and add a couple of error components. Some error is unique to the

round, such as slight manufacturing defects and air vortexes each round may experience in flight. Some error is shared, say the wind suddenly changes direction and all rounds are effected the same direction. We treat both of these components as independent samples from bi-variate normal distributions. These two error parameters relate to precision and accuracy. Individual error relates to precision and shared error relates to accuracy.

Once round impact error is factored in, a damage function is used to define the probability that an effect—casualty, disability, obscuration—was made surrounding the detonation. When considering multiple detonations, an effect was successful if at least one detonation resulted in the desired effect. The combination of probabilities describes the loss of effects caused by overlap. The target area weighs the effects area to determine the objective function.

Optimization

We can imagine that poor accuracy and precision will result in optimal aimpoints pushed to the center of the target. This will increase the probability that a round will hit the target when factoring in aimpoint-detonation deviation. Overlap has a diminished added effect from the combination of probabilities, but spilling effects outside of geometric targets has zero effect on the objective. Therefore overlap is not an enemy of poor precision. Poor precision also spreads detonations around the target even when aimpoints converge on a single point. These behaviors have all been observed from aimpoint optimization research and are common knowledge amongst experienced artillerymen.

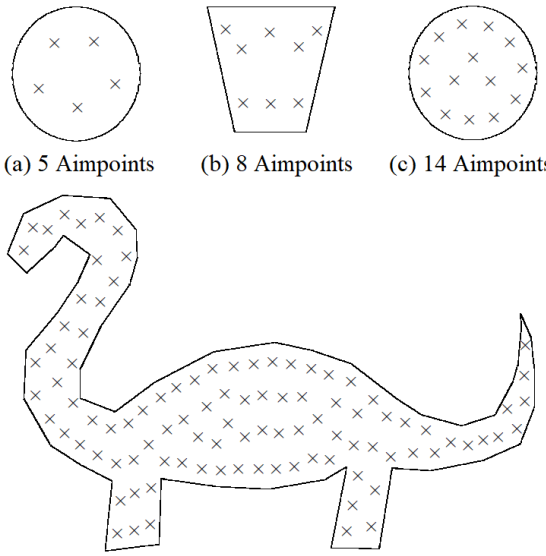

Figure 4. Simple and Complex Solutions

The inputs into the aimpoint optimization algorithm are the number of aimpoints, the size and shape of the target, the size and shape of the detonation, and the impact error distributions expected for some hypothetical weapon system and round. From these inputs, a solution is generated. This solution provides the location that each round should be aimed for maximum overall effects. For the purpose of this research, the actual pattern is discarded from our analysis. Instead, we focus on percent coverage and processing time.

Some solutions are demonstrated in Figure 4. They were generated by allowing the aimpoint optimizer to run for several minutes. When analyzed, solutions captured within the first second were only slightly less optimal. In real-world optimizations, we must ask the question, "when is the solution good enough?" Targets move, objectives change, and windows of opportunity close. Solutions become non-optimal quickly as solutions are only relevant at the time the optimization process started. Enabling the operator the ability to leverage time-sensitive solutions is critical for meeting mission objectives.

Meta-Optimization Strategy

The inputs for the meta-optimizer and optimization parameters are shown by Table 1 and Table 2, respectively. Five scenarios were generated to represent the operational profile plus a sixth very irregular target to explore the potential effect estimate for this factor. A total of 108 input configurations were analyzed. The shape complexity and stage utilization were measured instead of providing a high resolution surface function as neural inputs. Damage is Carleton—or, bivariate normal distribution in shape, scaled by a peak probability of kill. The size of the damage pattern was reduced to a percent coverage over target area per one detonation. This would allow detonation sizes and target areas to scale together because scaling measurements in the optimization inputs produce the same scaled solution.

Table 1. Aimpoint Request (Input Factors)

Factor	Levels	Description
Target Scenario	6 targets (Blocked)	The weight function described by a two-dimensional image: ellipse, rectangle, dinosaur, and three preprogrammed polygons.
Target Complexity	(Measured)	A simple geometric measurement, $p^2/(4\pi A)$, on the complexity of a shape using its perimeter, p, and area, A, scaled relative to a circle.
Stage Utilization	(Measured)	Measures the weight function as a ratio between all pixels in the target area and the total number of pixels in the bounding image.
%-Cov/Det	5%, 40%, and 80%	Ratio between the total area damaged from one detonation and the target area.
# of Aimpoints	1, 6, and 24	Number of rounds to shoot.
Peak Pr(Kill)	70% and 100%	Peak probability of kill at the center of a detonation. Defined as D_0 in the Carleton damage function.
Scaled Precision Error	0% and 50%	A ratio between the product of individual bivariate error, $\sigma_d\sigma_r$, and the target area, A.
Precision Eccentricity	0 and 0.75	Defines the relation between spread of precision error along the range and deflection axes. $\sqrt{1-(\sigma_r/\sigma_d)^2}$
Detonation Eccentricity	0 and 0.75	Defines the relation between spread of effects along the range and deflection axes. $\sqrt{1-(R_r/R_d)^2}$
Desired Optimality	0, 4, 7	The desired nine-count optimality of the result.

Table 2. Optimization Parameters (Control Factors)

Factor	Values	Description
Algorithm	{Simplex, Simplex2, BGFS, and BGFS2}	Used to converge on solutions. Other GNU Scientific Library (GSL) optimization algorithms are available, but were selected less often in a preliminary analysis of the dataset collected.
Resolution	0.01 to 1.0	How much to scale down the number of sampled pixels for optimization.
Stop Threshold	0.01 to 10.0	Magnitude of improvement between steps to stop a repetition.
Step Size	0.01 to 10.0	How far the initial step in a repetition should be relative to the square root of the target area.
# of Repetitions	1 to 500	Number of repetitions to perform
% Gene Pool Size	5 to 50	Number of the top performing solutions to maintain as a percentage of the number of repetitions.
Genetic Rate	0.01 to 1.0	The rate to pull two solutions from the gene pool for merging—opposed to individual randomly sampled aimpoints. Pool must fill up first.

As we can see, the inputs from Table 1 are analyzed from the user input. A design of 108 input configurations (excluding desired optimality) was generated and this sample set of input configurations is used for the remainder of this study. A global optimum database was generated as discussed in a previous section. After the global data was captured, optimality was computed as described in Table 3. Desired optimality was selected as the tuning parameter and the score was calculated using the following equation:

$$\text{Score} = (\text{Desired\_Opt} - \text{Observed\_Opt})^{+} + 0.1 \cdot [\log_{10}(\text{Observed\_Proc\_Time}) - \log_{10}(.1)]^{+}$$

This score ignores any processing times less than one tenth second. This prevents extremely fast configurations from scaling up and overpowering the desired optimality. Otherwise, the coefficient of 0.1 allows a full trade-off of 1-nine optimality for a solution that takes $10^{1/0.1} = 1 \times 10^{10\text{th}}$ of the processing time.

Table 3. Response Variables

Response	Values	Description
Logarithmic Processing Time	[-1, ∞)	The base 10 logarithm of the time taken to obtain an "optimal" solution. It is clamped to -1 for stability in the score.
Nine-Count Optimality	[0, M). The upper-bound, M, is defined by the computational limit into equality.	How the local optimal solution compares to the global solution for an input. The figure is represented by the number of leading nine in the ratio: $\#9s = -\log_{10}(1 - local/global)$.
Score	[0, ∞)	A score based on logarithmic processing time, desired nine-count optimality, and observed nine-count optimality.

Cost-Model Training

Scripts were written to generate training data for the cost model. This process was allowed to converge on minimized scores for each input configuration before completing in 2.8 days with a total of 50,677 records. A sensitivity analysis was performed on the score response. Considering main effects only, the strongest control factor to be selected was resolution, accounting for 4.2% of the total sum of squares. The strongest input factors were desired optimality, coverage per detonation, and precision error accounting for 59.7%, 5.9%, and 4.4% of the total sum of squares, respectively. Before adding the scenario blocking factor, the measurements of stage utilization and target complexity accounted for 0.49% and 0.0067% of the total sum of squares, respectively. Adding the blocking factor accounted for 0.25%. This means stage utilization it is a good candidate to replace a sizable portion of the blocking factor variance for any target area. The linear model on main effects has a coefficient of determination of R^2=0.801.

The 50,677 data records were used to train the cost model. For this neural network, we need a network of 19 input neurons and one output neuron. The input neurons represent 8 input factors (dropping the preprogrammed target shapes in lieu of its measurements), 7 control factors, and the tuning parameter. The input factors are numerical, so 8 neurons will be assigned to represent the input configuration. The convergence algorithm has been reduced to 4 options: BGFS, BGFS2, Simplex, and Simplex2. We will assign a neuron to each one. The remaining control factors are numerical so a total of 10 neurons represent the control configuration. The turning parameter is numeric and is also assigned a neuron. The output neuron represents the score. We will give the network a feed forward structure of 2 hidden layers of size 6 and 3. All hidden neurons will be given a logistic activation function and the output will be linear. The trained neural network gives us a model with a coefficient of determination of R^2=0.942. A significant improvement from the linear regression model.

Meta-Optimization Training

Using reinforcement learning and all of the input configurations from the training data in the last model, we will train the meta-optimizer neural network in an environment similar to Figure 3. The meta-optimizer network will have 9 input neurons and 10 output neurons. The input factors and tuning parameter all get one input neuron. The control factors account for all of the output neurons. We do not expect many principle components, so we will give this network a feed-forward encoder structure with one hidden layer of size 5. All hidden and output neurons will be given a logistic activation function.

The meta-optimizer was trained to leverage the desired optimality and minimize the expected processing time obtained from the cost meta-model. After the meta-optimizer was trained, we explored the average control factor selection as a function of desired optimality. In Figure 5, the vertical axis represents the range of each factor's low and high value from Table 2. We found that some control factors were constant, such as the number of repetitions, step-size, and genetic rate. We also found that BGFS2 and Simplex were never selected. We observed that BGFS and Simplex2 trade-off depending on the quality of solution desired. In the future, we could consider just these two convergence algorithms.

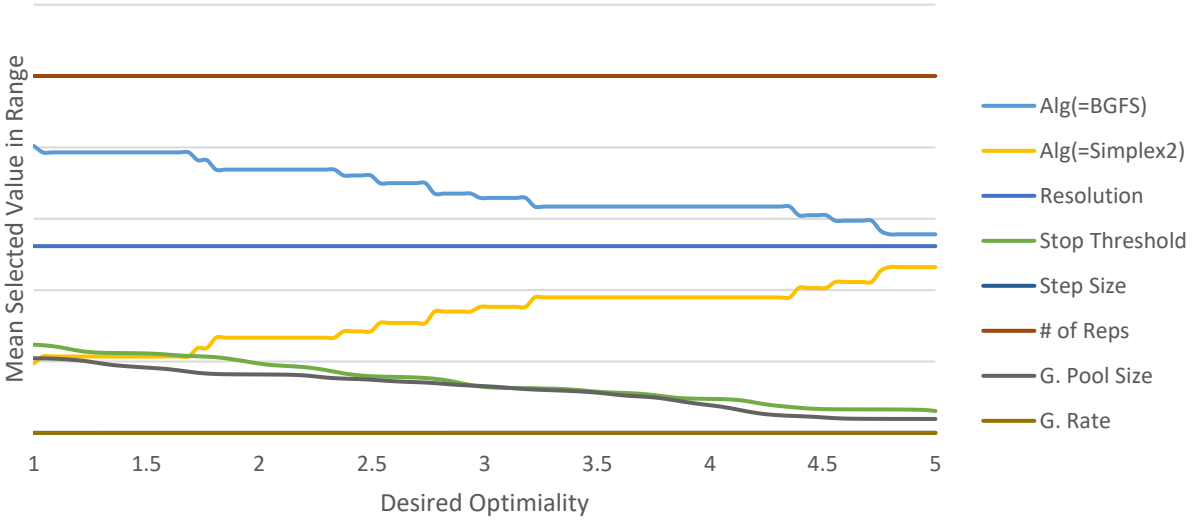

Figure 5. Average Control Factor Output as a Function of Desired Optimality

FUTURE WORK

As with any machine learning model, the degrees of freedom should be low to prevent overfitting error (Lang and Stanley, 2013). This research has not explored overfitting issues with reinforcement learning, but it is an important area of research to improve our meta-optimizer results. Also, the cost of retraining the meta-optimizer is high, especially when a global optimum database is required for measuring optimality. Techniques to reduce data collection and neural network training should be explored to address speedy retraining of the meta-optimizer by using past training data (Feurer et al., 2018). The process of optimizing meta-optimization can be employed when considering a limited computation budget (Branke and Elomari, 2012).

CONCLUSION

This research illustrates the impact that meta-optimization has for aiding time-sensitive optimization algorithms. The two-stage network approach has several benefits. First, the cost model is also meta-model for the real optimizer. This diminishes its stochasticity, thereby speeding up the training of the meta-optimizer. It also allows the meta-optimizer to leverage multiple objectives in a single scoring function: reduce processing time, reduce resources requirements, and maximize optimality. Second, algorithms and optimization parameters no longer need to be left susceptible to human misconception. With regular retraining of a meta-optimizer in the software pipeline, we can also adapt to software changes that impact the resource requirement, level of fidelity in the objective function, and optimality of solutions. Third, the neural network approach in both stages allows a fully trained cost model to back-propagate information into the untrained meta-optimizer using transfer learning. Finally, the end-user is given control over obtaining relevant solutions now or better solutions later—a critical feature for managing scalability and deadlines. The aimpoint optimizer use case demonstrated the process of meta-optimization. With the meta-optimizer integrated into the software, it will allow the warfighter to reap the benefits of both speed and efficiency in their use of munitions.

REFERENCES

Branke, J., & Elomari, J. (2012). Meta-optimization for parameter tuning with a flexible computing budget. In *Proceedings of the 14th annual conference on genetic and evolutionary computation* (pp. 1245-1252). ACM.

Feurer, M., Letham, B., & Bakshy, E. (2018). Scalable meta-learning for bayesian optimization.

Karpenko, A., & Svianadze, Z. (2011). Meta-optimization based on self-organizing map and genetic algorithm. *Optical Memory and Neural Networks, 20* (4), 279-283.

Lang, A., & Stanley, K. O. (2013). NeuroEvolutionary meta-optimization. In *The 2013 international joint conference on neural networks (IJCNN)* (p. 1-8).

McLaughlin, M. (2019). *Aimpoint Solutions on Complex Area Targets*. In Proceedings of the Interservice/Industry Training Simulation and Education Conference (I/ITSEC). Orlando, FL.

Nezhad, A. M., & Mahlooji, H. (2014). An artificial neural network meta-model for constrained simulation optimization. *Journal of the Operational Research Society, 65* (8), 1232-1244.

Stephenson, M., Amarasinghe, S., Martin, M., & O'Reilly, U. (2003). Meta optimization: Improving compiler heuristics with machine learning. *Acm Sigplan Notices, 38* (5), 77-90. New York, NY, USA: Association for Computing Machinery. doi: 10.1145/781131.781141

Łapa, K. (2019). Meta-optimization of multi-objective population-based algorithms using multi-objective performance metrics. *Information Sciences*, 489, 193-204.