# ECSE 541 Final Report

Patrick Ohl
*261182154*
patrick.ohl@mail.mcgill.ca

Aman Sidhu
*260885556*
aman.sidhu@mail.mcgill.ca

## I. Introduction

Convolution Neural Networks (CNN) have become an increasingly popular tool due to their exceptional results in performing computer vision tasks, like object classification and object localization [1]–[3]. However, a constraint on the design and performance of CNN's is identifying the hyperparameters needed to create high-performing models, especially in terms of accuracy [1], [3]. Hyperparameters can include architecture parameters, like the number of layers in the CNN and the activation functions as well as training parameters, such as the learning rate, both of which need to be specified by the designer. With so many different parameters to tweak, finding the optimal hyperparameters involves exploring a large design space which traditionally requires an experienced researcher to facilitate the search by hand [1]–[3]. This process can become very costly on design time since models can take long periods to train before they can be evaluated, meaning that unguided exploration will waste valuable time experimenting on bad hyperparameter configurations [1]. Hand-made CNNs are also not guaranteed to be the best configuration as automated searches have been shown to create models that outperform custom CNNs [1]. Thus, this motivates research of new methods that can guide the search for optimal hyperparameters to be more efficient, saving time, while still yielding good results, measured by model accuracy.

## II. Related Works

Common automated hyperparameter optimization techniques involve Grid Search, Random Search, and Bayesian Optimization [1]. Grid Search follows a trial-and-error approach by uniformly sampling new solutions across the whole hyperparameter space. Random Search explores the design space by performing evaluations on random solutions which can both speed up the search process and more effectively survey solutions over Grid Search. Lastly, Bayesian Optimization conducts an informed exploration using priors and Bayes' Theorem to decide where in the solution neighbourhood to sample next. However, these methods tend to perform poorly when the solution space becomes too large to search effectively [1].

With metaheuristics' success in finding good sub-optimal solutions in large design spaces, recent work has explored its use for hyperparameter optimization of CNN in the context of image classification and object detection with promising results [1]–[3]. In [3], they explore two different approaches to optimize CNN architecture parameters through Memetic Algorithms, the combination of Evolutionary Algorithms and Local Search (LS), and Simulated Annealing. Both metaheuristic techniques were able to surpass the state-of-the-art hyperparameter optimization methods for CNNs on image recognition datasets [3]. Other papers like [1] and [2] utilize population-based algorithms for generating accurate CNN for image recognition including: GA, Particle Swarm Optimization, Bat Algorithm, Artificial Bee Colony, and Ant Lion Optimization. With the broad range of possible metaheuristics, these papers tend to focus on similar approaches to the same problem.

## III. Approach

Inspired by our class discussion on [4], our project aims to optimize the architectural hyperparameters of an image classification CNN, by exploring the potential of trajectory-based metaheuristics algorithms. We will evaluate how well different meta-heuristics can find candidate hyperparameters, comparing the performances of the resulting CNNs in both accuracy and evaluation latency. We would like to extend past work by investigating trajectory-based algorithms like Tabu Search (TS), Simulated Annealing (SA), and Iterated Local Search (ILS) (replacing Guided Local Search). Going further, we will explore using a neural net predictor to estimate CNN performance based on CNN hyperparameters as input. In more detail, the project will entail the following tasks:

1) Build a parametrizable CNN for image recognition.
2) Build optimization algorithms for the CNN using different meta-heuristics.
3) Optimize the parameters of the CNN using the optimization algorithms and compare the test accuracies for the resulting sets of parameters.
4) Build and train a NN-predictor that, given the parameter configuration of the CNN, predicts its accuracy
5) Use the predictor in the optimization algorithms and evaluate how it may affect their runtime, as well as the accuracy of the resulting CNNs

## IV. Experimental Setup

### A. Image Classification CNN

For this project, the CNN we will use has seven layers, consisting of two pairs of 2D convolutional layers with a 2D max pooling layer, another 2D convolutional layer, and two fully connected layers. The CNN input is images from the CIFAR10 dataset, broken into 32x32 pixels divided into separate RGB channels, and the output is the classification for one of 10 object categories (bird, cat, truck, etc.). For our purposes, the number of layers will stay constant, and we seek

to optimize only the hyperparameters of the convolutional and pooling layers. The hyperparameters for both are:

- *2D Convolutional Layer* - Filters, Kernel Size, Activation Function, Padding,
- *2D Max Pooling Layer* - Pool Size, Strides, Padding

To create the CNN, we will be using the TensorFlow library, and program the CNN and metaheuristics in Python. We perform our evaluations on Google Collab's GPUs and locally on Aman's NVIDIA GeForce RTX 2070.

### B. Metaheurisitics

To map the CNN hyperparameter optimization problem onto a metaheuristic algorithm, we had to uniquely define: the solution representation, solution neighbourhood, and fitness/objective function. The solution is expressed as a dictionary whose keys are the five layers, and values are also dictionaries containing the relevant hyperparameters keys and their assigned attributes. Using a specified solution as a reference, to find neighbouring solutions we stochastically choose three layers and again stochastically choose one or two hyperparameters in each, depending on the layer type, to change. The new hyperparameter value is randomly sampled from a predetermined set of values for that specific hyperparameter. Lastly, we use the accuracy and latency of the trained CNN model on an evaluation dataset to define the following objective function:

$$ObjectiveFunction = \frac{Accuracy^2}{Latency} \tag{1}$$

We chose this definition as we want to optimize the trade-off between evaluation latency and the accuracy of the optimized CNN. Accuracy is squared to ensure that changes in accuracy are dominant over changes in latency. This also is the reason why we use the predictor and evaluate its impact on those two metrics.

As an initial solution, we use the default hyperparameters of the CNN as a universal starting point for all metaheuristics since we assume that the optimal hyperparameters will be nearby.

### C. Tabu Search

The TS algorithm uses short-term memory to keep track of trained past solutions in the tabu list to avoid repeating the same solutions. We also include tabu conditions which in this case will be a list of past changes to layers and hyperparameters that generated poor-performing models. In our implementation, the tabu tenure for short-term memory is 100 and the tenure for tabu condition is 5. The length of tabu conditions is kept much smaller to prevent negatively impacting the neighbouring solution generation and thus finding potential candidate solutions. The main loop of the program finds a candidate solution by performing a best improvement local search on five, non-tabu neighbours around the best-known solution. On each iteration, the algorithm checks if the best-known solution needs to be updated, and then updates the tabu list and tabu conditions. The search terminates after 100 iterations, or if there has been no new best solution in the past 20 iterations. Additionally, as an aspiration condition, if there has been no improvement after 10 iterations, the tabu tenure for past solutions doubles and the tabu conditions are ignored. This change aims to help encourage diversification, and to allow the algorithm to find new potential solutions before ending the search.

### D. Simulated Annealing

SA is a probabilistic approach that, at each iteration, generates a random solution in the neighbourhood of the current best solution. If the new solution is better than the old one, it is accepted. Otherwise, it is accepted with probability

$$exp{-}\frac{i(f(old) - f(new))}{cf(old)} \tag{2}$$

where *f(old)* and *f(new)* are the objective values of the respective solutions, *c* is a constant and *i* refers to the iteration count. As a result, new solutions with a slightly worse objective value are more likely to be adopted than solutions that are far worse than the current one. Moreover, as the optimization progresses, the probability that a worse solution is adopted decreases. As a result, the focus of the optimization process gradually shifts from exploration to exploitation. Our version of SA runs for at most 500 iterations, or breaking after 20 iterations of no improvement, and evaluates one model per loop. We wanted to explore the trade-off between the intensification provided by TS's local search and a diversification-focused approach through SA.

### E. Iterated Local Search

The ILS utilizes a combination of the prior two metaheuristics. It implements short-term memory along with the tabu conditions from TS, and it accepts solutions based on SA. The salience behind ILS is that perturbs the solution neighbourhood to another area of the design space to find another local minima in each iteration. Thus, we sample new neighbours around a starting point solution whose hyperparameters are randomly modified after each best improvement local search. Each local search evaluates 5 potential solutions. Like TS, the tabu tenure for short-term memory is 100 and the tenure for tabu condition is 10. The algorithm also utilizes the same diversification methods if no improvement is found after several iterations. The overall goal of this approach is to combine the diversification and intensification techniques of TS and SA, on top of perturbing the solution neighbourhood.

### F. NN Predictor

The predictor is a neural network consisting only of fully connected layers. It predicts the accuracy of the seven-layered CNN model we are trying to optimize, as well as its evaluation latency for the given test set. This involves preprocessing of the hyperparameters since they include non-numerical values such as activation functions. The goal is to use this predictor in the optimization process of the CNN. By replacing the training and testing of every possible solution with a prediction of its

metrics (accuracy and latency), the runtime of every iteration can be reduced significantly. This enables a far more extensive exploration of the design space. To get an accurate predictor, we are using metaheuristics to optimize the predictor itself. The objective is to optimize the function

$$obj = \frac{1}{Latency * (CnnAccMAE * CnnLatMAE)^2} \quad (3)$$

since we want the predictor to be as accurate, but also as fast, as possible.

The implementation of such a predictor involves the exploration of a vast design space that includes not only the tuning of the NN parameters, but also embedding, pre- and post-processing methods. This means that in addition to the metaheuristic exploration, a lot of tuning has to be done by hand. For example, the CNN hyperparameter model includes activation functions that need to be mapped to numerical input features. To transport a notion of distance between different activation functions to the NN input, we based the mapping on similarities in the shapes of the different functions. For example, the sigmoid and tanh functions both have an s-like shape, so their mappings are clustered together.

## V. Results

### A. Evaluation

We evaluate candidate CNN solutions based on their accuracy and their test latency, reflected in the objective function shown in equation 1. CNNs are compiled in TensorFlow using the "adam" optimizer to minimize "SparseCategoricalCrossentropy" loss. CNNs are then trained on 50,000 32x32 images for 10 epochs from the CIFAR10 dataset, and tested on 10,000 unseen images. We run each metaheuristic 5 times and select the best-performing model in terms of accuracy and latency for our comparison. We do not include training time, since at the end of the optimization, we have a trained model that we can use directly. Even when having to retrain the CNN after the optimization, either because the predictor was used in the optimization or because a larger dataset for training is available, the training time will be negligible compared to the optimization time. We ran each of the metaheuristics five times and selected the best results across all runs.

We mostly care about the trade-off between optimization runtime and the performance of the optimized CNN. This is why we use the predictor and evaluate its impact on those two metrics. We evaluate our predictor based on the objective function (3) described earlier. The main focus is to minimize the MAE's for the CNN accuracy and latency (scaled between 0 and 1), since inaccurate predictions will render the predictor useless. We used the MAE instead of the loss as the evaluation metric since the loss is an NN-hyperparameter that might change during the optimization and thus make the results impossible to compare. We also included the test latency of the predictor in the optimization objective since a faster predictor enables us to explore a larger design space in a shorter amount of time. We ran multiple SA optimizations for different start configurations that we configured by hand to get the best possible results.

### B. Testing

*Metaheuristics:* When we started designing the metaheuristics, we took a layered approach that slowly added new levels of complexity. In the beginning, we created TS around solving a simple example of sorting an array as we have seen in class. This allowed us to identify what code abstractions we needed to make, like the function used to generate neighbours and design the general structure of the algorithm. Once we solved this example, it was straightforward to map the CNN hyperparameter optimization problem by creating and using the appropriate function calls. To ensure the code was working properly, we logged results to a text file and printed updates to the terminal as the metaheuristics performed their search to understand if everything was working as expected.

*Predictor:* Since we were implementing the predictor at the same time as we were gathering data using the metaheuristics, it was difficult to get good evaluations on the hyperparameters of the predictor due to the lack of data. For example, a larger data set makes larger layers more effective, so we can't effectively evaluate the layer size until we gathered all our data. So, the testing process involved a lot of static analysis of the different models that could be explored and modifying the design space accordingly. Once we gathered enough data, we could run the metaheuristic optimization and check the predictor performance as we go.

### C. Metaheuristics

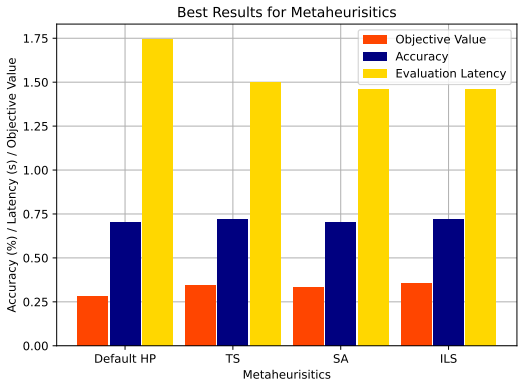


Fig. 1. Comparison of the final best candidate solutions from each metaheuristic and the default hyperparameters

TABLE I
TIMING COMPARISON OF METAHEURISTICS

| Metaheuristic | Iterations | Models Evaluated | Time (hrs) |
|---|---|---|---|
| TS | 31 | 155 | 8.7 |
| SA | 113 | 113 | 6.35 |
| ILS | 38 | 190 | 9.37 |

As shown in figure 1, each metaheuristic can find a better hyperparameter solution that improves both test latency and

accuracy over the default hyperparameters that initially achieve 68% accuracy and 1.74 s latency. ILS provides the best overall improvement, with the highest accuracy improvement of 4%, and latency drop by 16.3% over the default hyperparameters. For reference, TS provides 72% accuracy and 1.5 s latency, SA provides 70% accuracy and 1.4614 s latency, and ILS provides 72% accuracy and 1.4598 s latency.

We also compared the wall clock time for running these searches through Table I which highlights the number of iterations, models evaluated, and wall clock time required for each metaheuristic. As expected, SA runs the shortest and ILS and TS run for much longer due to performing more evaluations. It is also worth noting that throughout these searches, improvements to latency were more common than accuracy. Even though SA evaluates fewer models, it can find a latency comparable to ILS and better than TS, while still improving test accuracy. Thus, this suggests a tradeoff between intensification-focused approaches like ILS and TS which is needed to find higher accuracy solutions, but not necessary for improving latency where SA is sufficient.

Lastly, we looked at the differences and similarities of the hyperparameter solutions found by the metaheuristics to each other and the initial default hyperparameters. Some interesting observations include how all solutions use a linear activation function in the second convolutional layer. Each solution tended to increase the number of filters in the first convolutional layer from 32 to either 100 or 80. ILS uses larger pool sizes for each pooling layer in contrast to TS and SA. TS's solution out of the three was the one that had the most number of hyperparameters in common with the initial solution, sharing 11/18. Between pairs of metaheuristic solutions, SA and ILS had 8 hyperparameters that were the same, which was the most.

### D. NN Predictor

The best results that we achieved for the NN predictor were a MAE of 0.17 for the accuracy and a MAE of 0.19 for the latency (with the latency output scaled between 0 and 1). These predictions are not good enough to be used in the CNN-optimization process, since an optimization based on such inaccurate assumptions would be rendered useless.

With most of the NN configurations we explored leading to MAE's of around 0.25 for both metrics, it is clear that the optimization process needs to be improved in order to obtain a usable predictor. We will discuss possible improvements in the next section.

## VI. Conclusion

### A. Future Work

The project has been a great opportunity to learn more about CNN's and metaheuristics, though there are alternative approaches and ideas we would like to investigate if there was more time. For the metaheuristics, we wanted to look at other ways of generating candidate solutions from the solution neighbourhood that did not emphasize diversification as heavily. Our current approach can be described as performing a localized Random Search, effective at sampling the range of possible solutions but ultimately suffers the same shortcomings as Random Search. Specifically, we currently do not take advantage of the information from prior evaluations to inform how we should sample for new solutions, causing the search to be inefficient. If we did, this could be particularly helpful for TS and ILS where we have access to the known prior solutions through the tabu list. To address this, we could instead use Bayesian Optimization which uses past evaluations as Bayesian priors to infer which hyperparameter to change next. Since we assume that a better solution is in the neighbourhood of the default hyperparameters, perhaps using this intensification-based method would converge to better solutions faster. This could also improve the effectiveness of the tabu list as we no longer generate and keep track of random solutions which are unlikely to repeat given the immense size of the design space. There are already Python libraries we could use to implement Bayesian Optimization such as HyperOpt.

For the predictor, a more extensive design space exploration is needed to get results that can be used in the optimization process. Getting a better understanding of the effects of specific parameters is also needed to constrain the search to a smaller, but viable design space. Different ideas for further improvements involve

1) improved embedding and preprocessing of the CNN-hyperparameter models
2) separation of the predictor into 2 neural networks: One for accuracy and one for latency
3) finding a more suitable learning schedule whose specific parameters can be included in the optimization process
4) collecting more (and better / more balanced) training data

### B. Summary

Throughout the past few weeks, we gained experience using practical tools like TensorFlow, and grew our knowledge on the many *layers* of CNNs. As the two of us did not have extensive experience with either CNN or metaheuristics, we found this project to be a great chance to challenge ourselves with a topic that has applications in the design of neural networks.

## References


[1] A. Gaspar, D. Oliva, E. Cuevas, D. Zaldívar, M. Pérez, and G. Pajares, "Hyperparameter optimization in a convolutional neural network using metaheuristic algorithms," in *Metaheuristics in Machine Learning: Theory and Applications*. Springer, 2021, pp. 37–59.

[2] D. B. Prakash, K. A. Kumar, and R. P. Kumar, "Hyper-parameter optimization using metaheuristic algorithms," *CVR Journal of Science and Technology*, vol. 23, no. 1, pp. 37–43, 2022.

[3] V. Bibaeva, "Using metaheuristics for hyper-parameter optimization of convolutional neural networks," in *2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP)*, 2018, pp. 1–6.

[4] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM computing surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.