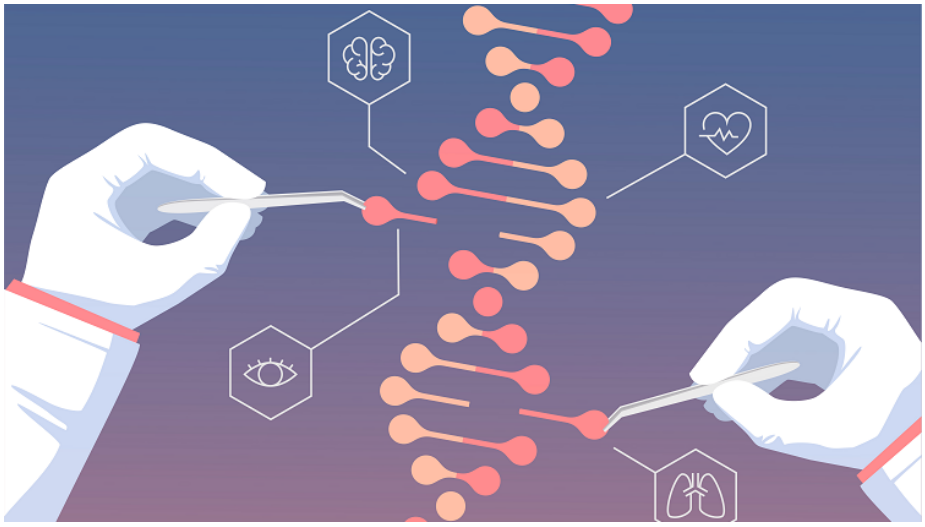
MULANN

Genetic Algorithm Mutation Learning with Artificial Neural Network

Ben-Gurion University of the Negev

Amir Bachar
Raz Lapid
Snir Vitrack Tamam

Spring 2021



1 Introduction

Genetic algorithms are inspired by natural biological processes. In order to approximate the best solution to a given problem, they start with a population of possible solutions, and use their selection functions to choose parents that would generate new offspring for the next generation. Mutation is also a fundamental part of the optimization process, and is usually done at random, without any prior knowledge of the fitness of the individual, and the alleles that contributed to said fitness.

Artificial Neural Networks (ANN or simply NN) is a common machine learning method to learn patterns. They have been proven to be able to approximate any function [4], and also work well in practice. We suggest a new mutation, which is based on learning a neural network model to improve the choice of mutation. A parallel work by H. Zhang et al. was done in a paper called "Learning to Mutate for Differential Evolution" yet to be published, which would be presented in 2021 IEEE. To our knowledge, our proposed method is the first attempt at learning non-random genetic algorithm mutations using an embedding of the problem phenotype.

We chose to showcase our mutation on the Travelling salesman problem (TSP). Although going through all possible swap mutations for an individual and calculating their fitness is possible in this case, this is only a demonstration of a more general framework, which could be useful in practice even more for other problems. Our algorithm would be especially useful in cases when access to the calculation of the fitness function is restrictive or expensive, hence having more generations or a bigger population is infeasible.

2 Problem

TSP is a commonly used problem in the context of genetic algorithms. It is known to be an NP-hard problem. The input of the problem is a list of cities and their pairwise distances. The expected output is the shortest possible route that visits each city exactly once. In order to generate multiple equivalent sets of problems, we have generated 50 random cities on a square, and defined the distance as the standard Euclidean distance.

	Baseline	MULANN
Problem	TSP with 50 random cities	TSP with 50 random cities
Representation	Permutation	Permutation
Crossover	Order 1 Crossover	Order 1 Crossover
Crossover Probability	100%	100%
Mutation	Random Swap	0.3-greedy NN Model Based Swap
Mutation Probability	100%	100%
Elitism	No	No
Fitness Function	Route Length (minimize)	Route Length (minimize)
Parent Selection	Tournament, $k = 2$	Tournament, $k = 2$
Population Size	1000	1000
Generations	15	15
Initialization	Random	Random
Total Fitness Evaluations	15000	15000
Embeddings	-	*node2vec* with $dim = 5$ number of walks=2000 walk of length = 10
NN Model	-	Bidirectional LSTM with hidden layers of size 32

Table 1: Hyper parameters used for the experiment

3 Methods and Algorithms

We have generated random TSP problems (50 cities on a 2D square), and then implemented a genetic algorithm for optimizing a solution for them using genetic algorithms. The mutation used was the simplest one for a permutation-based GA - the swap mutation, that exchanges two random genes. This would also serve as the baseline model for comparison. For implementing our proposed method, we generated an artificial neural network model that learns which mutations are useful. This learning is done online (during the normal execution of the genetic algorithm), so no extra fitness evaluations are done. The input of the network (X) is an embedding of the permutations, that would be described more thoroughly in the Embeddings dedicated section. The hyperparameters used for the experiment are shown in Table 1.

3.1 Embeddings

In order to make the problem "learnable", we needed a way to encode a solution (permutation) in an informative manner. Each individual is encoded with a node2vec embedding algorithm [3], [7], which uses random walks on the graph to generate random paths (sentences). These sentences are then learned using a skip-gram language model [2], [5] to generate a representation for each node. The architecture can be seen in Figure 1.

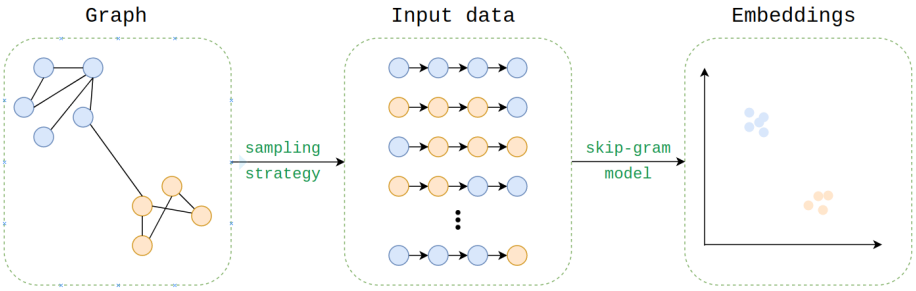


Figure 1: The diagram shows the whole process of embedding a graph

3.2 Neural Network Training

Now we can gather examples of improved solutions as the X's and the mutations as the y's. This data will go to a neural network which will learn how to choose the mutations in a smarter way. For each generation, half of the best mutations were used in a 1 batch training and 1 epoch.

In order to avoid overfitting, we have added noise to the y vector in a way that instead of having just two ones, it might have three (added an extra one value in a random position). Furthermore, two dropout layers were added to the model, both with 0.5 probability.

We chose to use bidirectional LSTM as the learning model, illustrated in Figure 2. Unidirectional LSTM only preserves information of the past because the only inputs it has seen are from the past. Using bidirectional will run the inputs in two ways, one from past to future and one from future to past. Using the two hidden states combined we are able, at any point in time, to preserve information from both past and future.

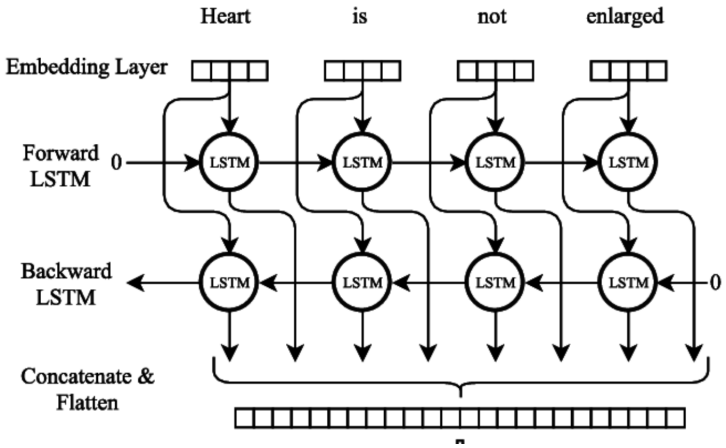


Figure 2: Bidirectional LSTM Flow diagram

4 Software Overview

The project contains 2 different parts, the genetic algorithm and our suggested approach, MULANN. The code can be accessed through github:
https://github.com/snirvt/Derandomized_GA

In the genetic algorithm, we implemented a framework which randomly creates a TSP problem. It essentially creates N randomly distributed cities, on a plane and the main assumption is that between every two cities there exists a road connecting them. Then, we execute the genetic algorithm, with the hyperparameters listed above, and return the fittest evolved individual, which is the shortest route that the algorithm has found where it visits each city exactly once.

In MULANN, most of the processes are the same as the regular GA, except that the mutation has some probability being taken from the trained NN (keras deep learning framework) prediction for the best swap. Before swap is performed on all individuals for a given generation, each individual is also going through embedding (with the library node2vec). Then after generating the offspring using the swap mutations, the NN is trained on the embedding and mutations, to be used for next generations.

5 Experiments and Results

First, we did a proof of concept outside the genetic algorithm framework. For this purpose, we have implemented the learning algorithm of the best cities to swap positions in the sequence, and learned its performance offline on a training set of random generated permutations and swaps.
We managed to learn successfully mutations that are better than random on average — it is shown in Figure 3. The *pval* for that hypothesis is 0.0001.

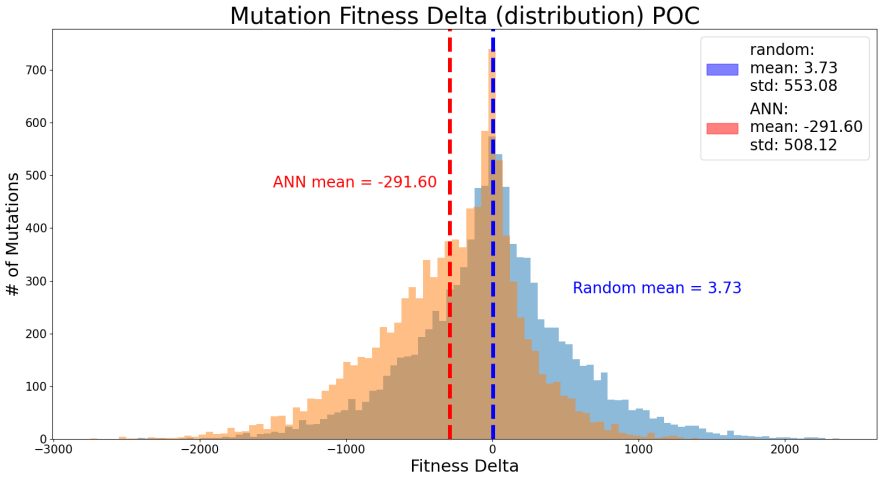


Figure 3: Mutation fitness delta graph

After successfully learning mutations that improved the fitness of a solution, we have initialized a new model, so that all the learned parameters of the NN will start from scratch on each run of the genetic algorithm. We executed the baseline algorithm and the MULANN algorithm 30 times. In order to reduce the variance of the comparison, each time both algorithms were run on the same random graph of cities and the same starting population.

We first show (Figure 4) the distribution of differences of fitness before and after each mutation, and the *pval* for the hypothesis that MULANN generates better mutations on average. Using harmonic mean p-value analysis we can combine all the *pvals* (for all 15 generations) into one *pval* and get an upper bound for the *pval* of the hypothesis that the learning model managed to improve the mutation on average [6] - overall *pval* is 0.0096, hence we get that the model manages to repeatedly produce statistically significant smarter

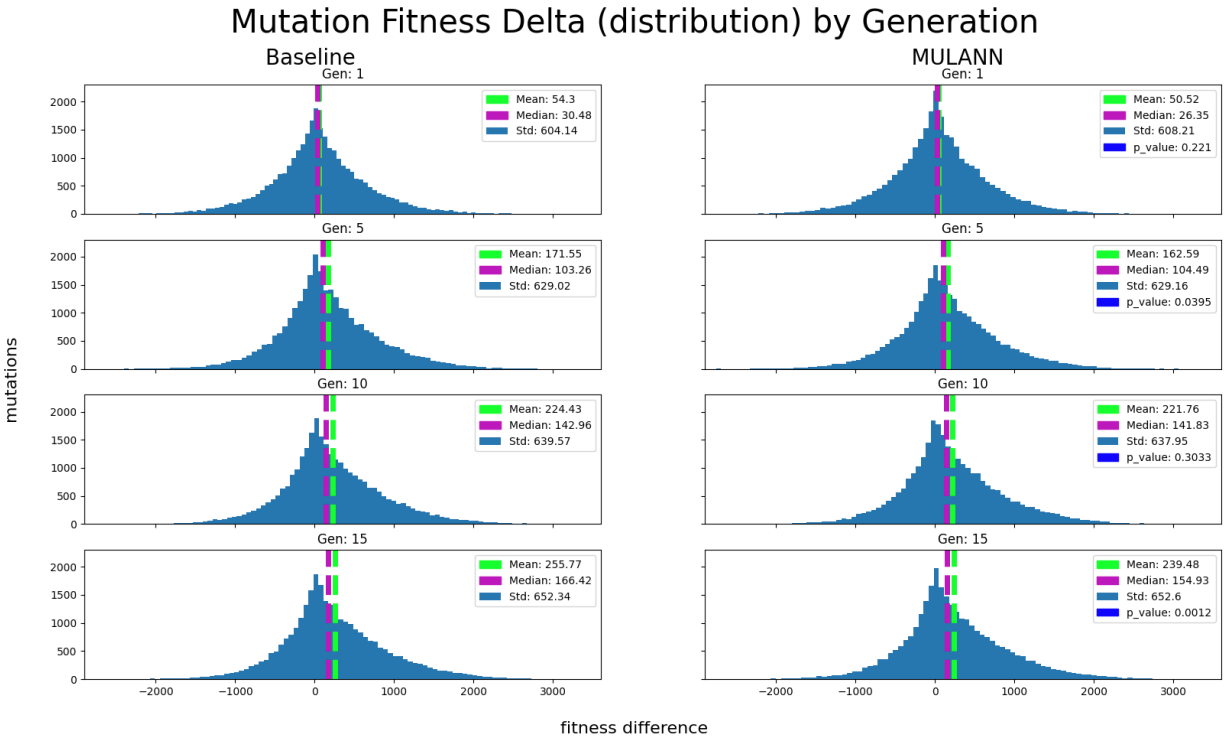


Figure 4: Mutation fitness delta graph by generation

mutations. Notice that since at generation i, the population of the MULANN algorithm is better on average than the base algorithm, it should have been even harder to produce better mutations, however it still manages to produce better mutations.

In a more qualitative analysis, it is interesting to note that the standard deviation per generation is similar in both algorithms, and that the means of MULANN are lower, unlike the median which is not always the case. We will now look at the performance of the entire MULANN algorithm Figure 5.

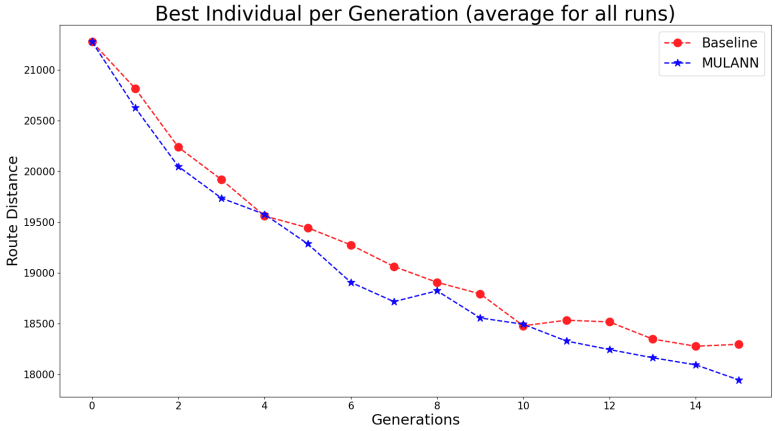


Figure 5: Best individual per generation graph

We can see that on average, MULANN got lower route distance than the baseline (hence better fitness) for almost all generations. And finally, the best individual's fitness per experiment (the experiment indices were sorted by the best individual of the MULANN algorithm in order to visualize the results better), which can be seen on Figure 6 below.

As we can see MULANN has won in 17 experiments, lost in 13. The *pval*

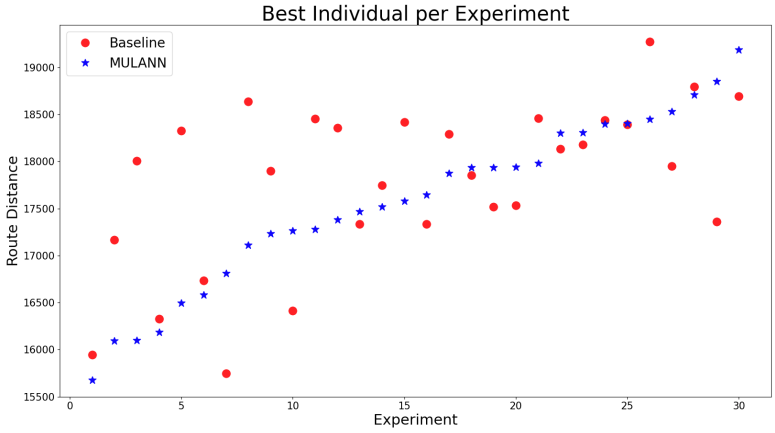


Figure 6: Best individual per experiment graph

using paired samples t-test is 0.077 (permutation test is not suitable here, since the experiment index is the main confounder).

- The experiment was executed also with an ϵ-greedy mutation rate of 0.5 and 0.7, and produced less conclusive results overall.

5.1 Difficulties

Everything was working as expected when we trained the proof of concept NN - the model, without a doubt, has learned how to make smarter mutations, but in practice it didn't work right away in the context of the entire genetic algorithm. The accuracy of the neural network got very high in the training which implies overfitting, probably because of a super individual which was stuck in a local minimum for many generations. This was solved with three ways:

- Dropout layers were added to the model.
- Noise was added to the y vector.
- ϵ-greedy mutation rate of 0.3.

6 Future Work

- In order to avoid overfitting and training on the most accurate data it might be a good idea to actively preserve diversity.
- Try different types of neural networks (regular, attention, convolution, GNN, etc) and see what captures the information the best way.
- We can try to use the node embedding separately or with the edge embedding as X.
- Different graph embedding methods.
- Learning a more complicated mutation.
- Applying the same idea on crossovers.
- Study the best distribution of mutation deltas (similar to the One-Fifth Success Rule [1]).

7 Conclusions

Using artificial neural networks in order to learn smart mutations was proven to successfully produce better results for the TSP problem. This work serves as an initial framework in what will hopefully be further researched in the future, using real dataset as well. There is still much more research to be done about how to handle the learning process.

On a personal note, this was a very interesting project for us.

References

[1] Anne Auger. Benchmarking the (1+ 1) evolution strategy with one-fifth success rule on the bbob-2009 function testbed. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2447–2452, 2009.

[2] Sergey Bartunov, Dmitry Kondrashkin, Anton Osokin, and Dmitry Vetrov. Breaking sticks and ambiguities with adaptive skip-gram. In *artificial intelligence and statistics*, pages 130–138. PMLR, 2016.

[3] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[4] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[5] Chris McCormick. Word2vec tutorial-the skip-gram model. *Apr-2016.[Online]. Available: http://mccormickml. com/2016/04/19/word2vec-tutorial-the-skip-gram-model*, 2016.

[6] Vladimir Vovk and Ruodu Wang. Combining p-values via averaging. *Biometrika*, 107(4):791–808, 2020.

[7] Changping Wang, Chaokun Wang, Zheng Wang, Xiaojun Ye, and Philip S Yu. Edge2vec: Edge-based social network embedding. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(4):1–24, 2020.