Various Population Models with Feed-Forward Neural Networks

Turner Haugen TURNER.HAUGEN712@GMAIL.COM
Machine Learning Student
Montana State University
Bozeman, MT 59718, USA

Jackson Thorn JTHORN2448@GMAIL.COM
Machine Learning Student
Montana State University
Bozeman, MT 59718, USA

Abstract

This paper aims to provide multiple population based algorithms as an alternate way to tune neural networks. The genetic algorithm selects networks to create offspring, mutate weights in those offspring, and replace the worst performing algorithms to explore the potential best weights we can have while trying to find the best solution. Differential evolution also uses crossover and mutation, but crosses over every network with a combination of networks that makes a donor vector. Particle swarm optimization has each network communicate with each other to travel in the direction of their personal best as well as toward the best performing model. All 3 of these methods take the best neural network after the algorithms converge enough, which will then be used for cross validation. These can also take classification and regression methods, and each has their own set of hyperparameters to tune.

1 Introduction

Population based methods are heavily based on the concept of evolutionary programming. These methods all incorporate a population of models that are trying to improve over time by using information from other models in the population. How each algorithm in a population obtains and uses this information is how each population method differs from each other. Some have algorithms communicate with each other and share where the best solution is, like particle swarm optimization doing it directly or ant colony optimization doing it indirectly through pheromone trails. Others have algorithms create new algorithms by combining information from pre-existing models and replacing a population through Darwinist concepts. Because of so many different ways of selecting models, there are constantly new population based methods being discovered that make slight changes to the process. As this is our first time working with population algorithms, we will be working with three classic population based methods, those being the genetic algorithm, differential evolution, and particle swarm optimization.

The genetic algorithm is one of the oldest evolutionary programming methods, but re-

mains useful to this day. First discussed in 1962, the genetic algorithm originally consisted of a a population of bit strings that represented algorithm parameters. We are using a real valued genetic algorithm, where each member of the population consists of a neural network. These networks start with randomly initialized weights and biases. A tuned number of networks are selected to create offspring, where the models in the population with the best fitness are more likely to be selected. These selected models are then used to create offspring, which share a mix of information from each parent. A crossover rate is used to control the rate at which this happens. Then, children are put through mutation, where a random proportion of the weights and biases in each child get randomly deviated. This is done to encourage more variation to explore the sample space more. Lastly, each child replaces the current worst member of the population through steady state replacement, which has a lower sampling pressure. If a child is worse than the worst member of the population, it is removed instead. This process is repeated until the best model remains the same for a set number of repeats, which is also tuned. The setup of this algorithm contains many hyperparameters, which means tuning on it may take a long time.

Differential evolution shares many processes with the genetic algorithm, but makes many tweaks to improve its ability to search the possible options in the sample space. The primary element that makes differential evolution unique is the creation of a donor vector, which is the combination of a few distinct randomly selected members in the population. This donor vector is used to create children with each member of the population. For our donor vector, we decided to pick a DE/each/2/z strategy. This means we iterate through each member of the population procedurally for crossover and pick 2 random members of the population to create a donor vector. We only replace each member of the population by their child if the child performs better than the parent. This process is continued for a set number of (tuned) iterations.

Particle swarm optimization is unique from these 2 other algorithms as it does not involve crossover, replacement, or mutation at all. Instead, each member of the population communicates with each other about what weights and biases gave them the best fitness. Some versions of PSO will restrict which models can communicate with each other to have a low sampling pressure, but we decided to use a star topology (every model communicates with each other) due to its simplicity and increased communication. Each model in the population stores its velocity, which determines which direction it will update its weights in. This velocity is updated on each iteration by its prior velocity (intertia), personal best (cognitive component), and global best (social component). Random weights are generated for the cognitive and social components to make sure the influence of these components varies on each iteration. This process is continued for a set number of (tuned) iterations.

For all of these algorithms, we will be comparing the results of these three algorithms to each other, as well as to the results from the backpropagated neural net from project 3. In general, we hypothesize that all of the population based methods as a whole will result in models that perform better than the backpropagated neural net, as the backpropagated neural nets suffered from overfitting. The increased emphasis on exploring the search space for evolutionary methods should help with this, especially on the machine and forestfires

datasets. For the genetic algorithm, we hypothesize that it will suffer the least from overfitting due to the use of steady state replacement, which is supposed to reduce sampling pressure. Particle swarm optimization will likely run the fastest due to all models communicating with each other, but may result in overfitting. Differential evolution will likely not find the best solution due to our methods not using the best algorithm to find the donor vector, but will likely result in quicker solutions and low overfitting.

2 Design

2.1 Genetic Algorithm: Probabilistic Selection

After initializing the population of networks, the process of making a new generation of networks starts with probabilistic selection. Each model's fitness is used as a probability distribution weight. For a given model, x_i, its probability of being selected is:

$$P(select = x_i) = \frac{f(x_i)}{\sum_{j=1}^{|P|} f(x_j)}$$

Where $f(x)$ indicates the fitness evaluated for model x and $|P|$ is the number of models in the population. We select 2 models from the population and ensure that the first model is not the same as the second model. The specific fitness used for this process is the inverse loss, which is $\frac{1}{cross-entropy}$ for classification problems and $\frac{1}{MSE}$ for regression problems. This selection process is repeated the same number of times as the steady state replacement count, which is a hyperparameter to be tuned.

2.2 Genetic Algorithm: Uniform Crossover

For each combination of parents selected for crossover, we first generate a random number to see if we actually perform crossover. If this number is lower than the crossover rate (hyperparameter to be tuned), we actually perform crossover. If not, then the children produced from this process will be identical to the parents. When uniform crossover is actually performed, a random number generator will randomly determine whether the corresponding weight/bias will come from parent 1 or parent 2 for child 1. The same weight/bias will be from the other parent for child 2.

2.3 Genetic Algorithm: Mutation

Each child generated from crossover (regardless of if crossover actually happens or not) will be subject to mutation. Mutation iterates through each weight/bias and changes their value based on a normal curve, with a mean of 0. This normal curve's standard deviation is a hyperparameter to be tuned. Due to the normal curve being centered at 0, each weight/bias has an equal chance of becoming higher or lower. Weights/biases are only effected by mutation based randomly on the model's mutation rate, which is another hyperparameter to be tuned.

2.4 Genetic Algorithm: Steady State Replacement

Steady state replacement is used to switch new models in for older models in the population. We updated our neural nets to be compared based on their fitness to make replacement easier. The number of models that may be replaced by steady state replacement is based on another hyperparameter that is the same as the number of models selected during probabilistic selection. We start by sorting both the list of new child models and older models so that the worst models are earliest in each model list. We compare the worst new model to the worst old model, and the new model is only added to the population if it is better than the old model. Every time a new model is added, the next old model compared is whatever the current worst old model is at the time. This process is continued until we have looked at all models.

2.5 Differential Evolution: Donor Vector

The donor vector in differential evolution consists of a linear combination of different vectors. In our case, we decided to use DE/each/2/z. This means our donor vector is calculated with the following equation:

$$v_j = x_i + \beta x_1 - x_2$$

Where x_1 and x_2 are randomly selected models such that $x_1 \neq x_2$, β is the learning rate of the algorithm which is a tuned hyperparameter between 0 and 2, and x_i is the model currently being mutated by differential evolution. Since we are looking at each model, we iterate through what vector is x_i systematically to effect each individual model.

2.6 Differential Evolution: Binomial Crossover

Each model gets selected for crossover between itself and the current donor vector. This process is similar to uniform crossover, except the chance that a weight comes from the current model or the current donor vector is not even. Instead, an additional hyperparameter, α is used to compare a randomly generated number to. If the randomly generated number is higher than α, then that weight will be pulled from the donor vector. Otherwise, that weight will be pulled from the current model. Different values of α can result in crossover selections preferring the donor vector or the current model. From there, the result of crossover replaces the model it originally resulted from if the crossover model has a better fitness. This process repeats for a tuned number of generations.

2.7 Particle Swarm Optimization: Velocity Updates

In particle swarm optimization, velocity is everything, as a model's velocity dictates how its weights will change. Velocity is affected by three different components, those being the social component, cognitive component, and inertia. Inertia is the model's prior velocity, which each model individually keeps track of. When the model is initialized, inertia gets set to random values, similar to the weights and biases of the neural network being tuned with this process. The cognitive component is the model's current best collection of weights. Every time a model's position changes, it checks to see the fitness of its current set of weights

and updates its current best weights if its current fitness is larger than its personal best fitness. The social component is similar, but instead keeps track of the set of weights that provided the best performance of every model in the group. Both the social and cognitive components have the current model's current position subtracted from them so that these components measure the direction for the current model to take. These three components are used to update velocity through the following calculation:

$$v(x_t) = \omega v(x_{t-1}) + c_1 r_1 (pbest(x_t) - x_{t-1}) + c_2 r_2 (gbest(x_t) - x_{t-1})$$

Where ω, c_1, and c_2 are all hyperparameter weights for the inertia, cognitive, and social components, r_1 and r_2 are randomly generated weights to ensure variance is maintained in velocity updates, and x_{t-1} represents the model's current position. This process repeats for a tuned number of generations.

3 Results

We will present a summary of the model after grid search along with the model architecture for each dataset. Additionally, the average loss from cross validation will be discussed. Due to the extensive number of hyperparameters present with each model, we will exclusively be looking at a neural network with 2 hidden layers and 5 nodes per layer to save time. Each backpropagated model discussed will be the best performing model from project 3. We will start by discussing the results for classification datasets.

Loss	**Backprop**	**Genetic**	**DiffEv**	**PSO**
Validation Loss	2.3832	0.0403	0.1661	0.0349
Avg. Cross-val Loss	2.3897	0.3187	0.2510	0.4263
Layer Architecture	[]	[5, 5]	[5, 5]	[5, 5]

Table 1: Breast Cancer Wisconsin (Classification)

The genetic algorithm had the following hyperparameters: Population size: 100, crossover rate: 0.8, mutation rate: 0.2, mutation variance: 0.1, steady state count: 10, patience: 20

Differential evolution had the following hyperparameters: Population size: 20, α: 0.8, scaling factor: 0.5, generations: 50

Particle swarm optimization had the following hyperparameters: Population size: 50, inertia weight: 0.5, cognitive weight: 1.5, social weight: 2.0, generations: 50

Of the 4 models used on the Breast Cancer dataset, the best performing model is differential evolution, as it seems that it was the only model that didn't succumb to overfitting. Both the genetic algorithm and PSO show clear signs of overfitting due to the substantial gap between their validation loss and cross-validation loss.

Loss	Backprop	Genetic	DiffEv	PSO
Validation Loss	0.7429	0.1623	0.2596	0.2483
Avg. Cross-val Loss	0.8215	0.1927	0.2670	0.2804
Layer Architecture	[]	[5, 5]	[5, 5]	[5, 5]

Table 2: Glass (Classification)

The genetic algorithm had the following hyperparameters: Population size: 50, crossover rate: 0.8, mutation rate: 0.1, mutation variance: 0.1, steady state count: 10, patience: 40

Differential evolution had the following hyperparameters: Population size: 50, α: 0.8, scaling factor: 0.5, generations: 50

Particle swarm optimization had the following hyperparameters: Population size: 50, inertia weight: 0.6, cognitive weight: 2.0, social weight: 2.0, generations: 25

The genetic algorithm performed the best this time, but all three algorithms performed fairly similarly. No overfitting occurred this time either, as all three validation losses are similar to their cross validation losses. However, all three easily beat the best backpropagated network.

Loss	Backprop	Genetic	DiffEv	PSO
Validation Loss	0.0	0.0	0.1989	0.0034
Avg. Cross-val Loss	0.1799	0.0	0.0	0.0
Layer Architecture	[]	[5, 5]	[5, 5]	[5, 5]

Table 3: Soybean (Classification)

The genetic algorithm had the following hyperparameters: Population size: 50, crossover rate: 0.8, mutation rate: 0.1, mutation variance: 0.1, steady state count: 5, patience: 40

Differential evolution had the following hyperparameters: Population size: 50, α: 0.8, scaling factor: 0.5, generations: 50

Particle swarm optimization had the following hyperparameters: Population size: 20, inertia weight: 0.8, cognitive weight: 1.5, social weight: 2.0, generations: 25

Soybean small is always a strange dataset to work with due to its very small size. It seems like our models were not able to handle it without encountering errors, as a cross validation loss of 0 shouldn't happen with neural networks. No models necessarily performed better here, as all 4 models reported a loss of 0 at some point. The genetic algorithm broke half of the time when run on this dataset too, which may be due to 0 losses not working well with the genetic algorithm's fitness of 1/loss.

Loss	Backprop	Genetic	DiffEv	PSO
Validation Loss	4.3978	4.4210	7.3756	6.3145
Avg. Cross-val Loss	5.0108	4.9774	16.3673	30.8110
Layer Architecture	[10, 5]	[5, 5]	[5, 5]	[5, 5]

Table 4: Abalone (Regression)

The genetic algorithm had the following hyperparameters: Population size: 50, crossover rate: 0.8, mutation rate: 0.2, mutation variance: 0.1, steady state count: 10, patience: 40

Differential evolution had the following hyperparameters: Population size: 50, α: 0.8, scaling factor: 0.5, generations: 50

Particle swarm optimization had the following hyperparameters: Population size: 50, inertia weight: 0.6, cognitive weight: 2.0, social weight: 1.5, generations: 25

On abalone, both the backpropagated neural network and genetic algorithm performed similarly well. Both differential evolution and PSO may have prematurely converged, as both reached solutions that are noticeably worse than the others and show signs of overfitting.

Loss	Backprop	Genetic	DiffEv	PSO
Validation Loss	895.5429	907.1134	948.5081	898.6347
Avg. Cross-val Loss	4240.0446	4376.4862	4473.9688	4480.1821
Layer Architecture	[]	[5, 5]	[5, 5]	[5, 5]

Table 5: Forest Fires (Regression)

The genetic algorithm had the following hyperparameters: Population size: 50, crossover rate: 0.8, mutation rate: 0.2, mutation variance: 0.1, steady state count: 10, patience: 40

Differential evolution had the following hyperparameters: Population size: 10, α: 0.8, scaling factor: 0.5, generations: 50

Particle swarm optimization had the following hyperparameters: Population size: 50, inertia weight: 0.6, cognitive weight: 1.5, social weight: 1.5, generations: 50

The forest fire data had all algorithms performing relatively the same. Each model got a mean square error of around 900 during the tuning process, and seemed to overfit with an average cross validation mean square error of around 4,400. This may be due to the skewed distribution of a forest fire's area or the large number of features present in this dataset.

Loss	Backprop	Genetic	DiffEv	PSO
Validation Loss	410.3156	1283.0828	8273.7780	3509.4898
Avg. Cross-val Loss	3584.4021	6021.4119	36739.5994	22450.8216
Layer Architecture	[]	[5, 5]	[5, 5]	[5, 5]

Table 6: Machine (Regression)

The genetic algorithm had the following hyperparameters: Population size: 50, crossover rate: 0.8, mutation rate: 0.2, mutation variance: 0.1, steady state count: 10, patience: 40

Differential evolution had the following hyperparameters: Population size: 50, α: 0.8, scaling factor: 0.5, generations: 50

Particle swarm optimization had the following hyperparameters: Population size: 50, inertia weight: 0.4, cognitive weight: 1.5, social weight: 2.0, generations: 50

The machine dataset performed even worse with differential evolution and particle swarm, as both resulted in clear premature convergence and overfitting. This may also be due to the algorithms struggling to handle with the one hot encoded model names. The genetic algorithm struggled to run this algorithm quickly, as it took a long time to find where it actually converged.

4 Discussion

All of the population models seemed to perform in a predictable manner; they each performed much better than the backpropagated network on classification data but failed to run well on the trickier regression datasets. Signs of premature convergence seemed to come up on differential evolution and particle swarm optimization with their relatively quick run time and a massive gap being present between validation and cross-validation losses. The genetic algorithm produced similar results on those datasets despite running for much longer, so it may have just overfit instead.

As for our hypothesis, backpropagation didn't necessarily perform worse than the population models. Its poor performance on the classification datasets may more likely be due to an error than anything else. The genetic algorithm did not perform better with overfitting than the other methods, as it seemed to have similar gaps between its cross validation loss and validation loss. The lowered sampling pressure may have actually led to more overfitting due to the drastically higher run time that led to. Particle swarm optimization did run the fastest and provided fairly similar results to the other algorithms, despite the increased pressure that results from using a star topology. Our hypothesis on differential evolution ended up not being true outside of the quicker solutions, as it sometimes performed the best and sometimes had the worst overfitting.

5 Summary

After project 3, it was interesting working with methods of tuning a neural net that could be applied to any model. The population models seem to excel in terms of their flexibility, as you could use them with architectures besides a neural network. At the same time, using these with a neural network made for a good point of comparison for all of the population methods. Unfortunately, none of the population methods discussed above performed as expected. Forestfires, machine, and soybean small remain datasets that are yet to have a good solution to due to their strange behaviors. However, these models may perform better with further tuning that we were unable to act upon due to time constraints.

6 Appendix

Jackson wrote the following:

- Particle swarm optimization
- Differential evolution
- Grid search + cross-validation reorganization
- Video recording + editing

Turner wrote the following:

- Initial codebase
- Genetic algorithm
- Paper