MINI PROJECT REPORT

EVOLVE A NEURAL NETWORK WITH A GENETIC ALGORITHM

Submitted by-
Varun Sharma (19103293)
Samay Gandotra (19103292)
Ayush Raj(19103291)
Arya Raj Singh(19103290)

Department of CSE/IT
Jaypee Institute of Information Technology, Noida

INTRODUCTION

Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.

Building the perfect deep learning network involves a hefty amount of art to accompany sound science. One way to go about finding the right hyperparameters is through brute force trial and error.

Here, we try to improve upon the brute force method by applying a genetic algorithm to evolve a network with the goal of achieving optimal hyperparameters in a fraction the time of a brute force search.

SCOPE

Let's say it takes five minutes to train and evaluate a network on your dataset. And let's say we have four parameters with five possible settings each. To try them all would take (5**4) * 5 minutes, or 3,125 minutes, or about 52 hours.

Now let's say we use a genetic algorithm to evolve 10 generations with a population of 20 (more on what this means below), with a plan to keep the top 25% plus a few more, so ~8 per generation. This means that in our first generation we score 20 networks (20 * 5 = 100 minutes). Every generation after that only requires around 12 runs, since we don't have the score the ones we keep. That's 100 + (9 generations * 5 minutes * 12 networks) = 640 minutes, or 11 hours.

We've just reduced our parameter tuning time by almost 80%! That is, assuming it finds the best parameters.

TOOLS & TECHNOLOGIES USED

- Keras library to build, train and validate.
- Python for writing genetic algorithms code.

DESIGN OF THE PROJECT

1. Creates a population of (randomly generated) members
2. Scores each member of the population based on some goal. This score is called a fitness function.
3. Selects and breeds the best members of the population to produce more like them
4. Mutates some members randomly to attempt to find even better candidates
5. Kills off the rest (survival of the fittest and all), and
6. Repeats from step 2. Each iteration through these steps is called a generation.

Repeat this process enough times and you should be left with the very best possible members of a population.

IMPLEMENTATION DETAILS

Applying genetic algorithms to Neural Networks:

We'll attempt to evolve a fully connected network (MLP). Our goal is to find the best parameters for an image classification task.

We'll tune four parameters:

1. Number of layers (or the network depth)
2. Neurons per layer (or the network width)
3. Dense layer activation function
4. Network optimizer

The steps we'll take to evolve the network, similar to those described above, are:

1. Initialize N random networks to create our population.
2. Score each network. This takes some time: We have to train the weights of each network and then see how well it performs at classifying the test set. Since this will be an image classification task, we'll use classification accuracy as our fitness function.
3. Sort all the networks in our population by score (accuracy). We'll keep some percentage of the top networks to become part of the next generation and to breed children.
4. We'll also randomly keep a few of the non-top networks. This helps find potentially lucky combinations between worse-performers and top performers, and also helps keep us from getting stuck in a local maximum.
5. Now that we've decided which networks to keep, we randomly mutate some of the parameters on some of the networks.
6. Here comes the fun part: Let's say we started with a population of 20 networks, we kept the top 25% (5 nets), randomly kept 3 more loser networks, and mutated a few of them. We let the other 12 networks die. In an effort to keep our population at 20 networks, we need to fill 12 open spots. It's time to breed!

Breeding

Breeding is where we take two members of a population and generate one or more child, where that child represents a combination of its parents.

In our neural network case, each child is a combination of a random assortment of parameters from its parents. For instance, one child might have the same number of layers as its mother and the rest of its parameters from its father. A second child of the same parents may have the opposite. You can see how this mirrors real-world biology and how it can lead to an optimized network quickly.

Dataset

We'll use the relatively simple but not easy MNIST dataset for our experiment. This dataset gives us a big enough challenge that most parameters won't do well, while also being easy enough for an MLP to get a decent accuracy score.

Steps to run the Project :

- To run the brute force algorithm:

  python3 brute.py

- To run the genetic algorithm:

  python3 main.py

The Results will be displayed on brute-log.txt and log.txt

RESULTS-

We'll start by running the brute force algorithm to find the best network. That is, we'll train and test every possible combination of network parameters we provided.

Our top result using brute force achieved 98.03% accuracy with the following parameters:

- **Layers: 2**
- **Neurons: 768**
- **Activation: elu**
- **Optimizer: adamax**

Now we'll run our genetic algorithm, starting with a population of 20 randomly initialized networks, and we'll run it for 10 generations.

Our top result using brute force achieved 97.03% accuracy with the following parameters:

- **Layers: 2**
- **Neurons: 512**
- **Activation: elu**
- **Optimizer: adamax**

The only difference is the genetic algorithm preferred 512 to 768 neurons. In the brute force run, the 512 network achieved 97.65%.

The genetic algorithm gave us the same result in 1/9th the time! And it's likely that as the parameter complexity increases, the genetic algorithm provides exponential speed benefit.