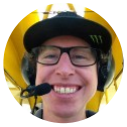
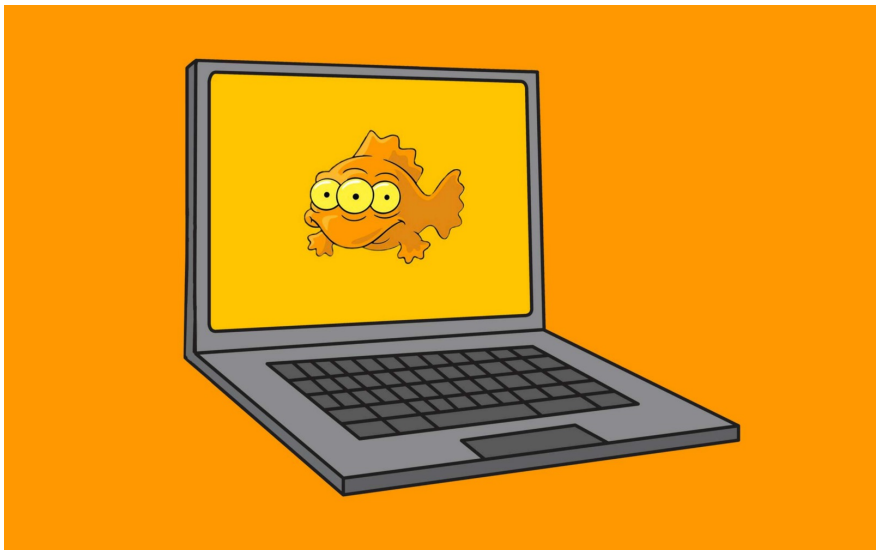


Matt Harvey Follow
Founder of Coastline Automation, using AI to make every car crash-proof.
Apr 6, 2017 · 5 min read

Let's evolve a neural network with a genetic algorithm—code included



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: Try every combination of sensible parameters, send them to your Spark cluster, go about your daily jive, and come back when you have an answer.

But there's gotta be a better way!

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.

How much faster?

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…

Ok Google, what's a genetic algorithm?

> *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.—Wikipedia*

How do genetic algorithms work?

At its core, a genetic algorithm…

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. Sounds like a lot evolution, right? Same deal.

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:

- Number of layers (or the network depth)
- Neurons per layer (or the network width)

- Dense layer activation function
- 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.

Now that we understand how it works, let's actually do it.

Dataset

We'll use the relatively simple but not easy CIFAR10 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.

Note: A convolutional neural network is certainly the better choice for a 10-class image classification problem like CIFAR10. But a fully connected network will do just fine for illustrating the effectiveness of using a genetic algorithm for hyperparameter tuning.

Code explained

Hopefully most of the code is self-explanatory and well-documented. (Famous last words, I know.) Here are parts of the `optimizer.py` module, which holds the meat of the genetic algorithm code. Note that this code is heavily inspired by an excellent post by Will Larson, Genetic Algorithms: Cool Name & Damn Simple.

```
def create_population(self, count):
    """Create a population of random networks.

    Args:
        count (int): Number of networks to generate, aka t
            size of the population

    """
    pop = []
    for _ in range(0, count):
        # Create a random network.
        network = Network(self.nn_param_choices)
```

We start by creating a population. This instantiates "count" networks with randomly initialized settings, and adds them to our "pop" list. This is the seed for all generations.

```
def breed(self, mother, father):
    """Make two children as parts of their parents.

    Args:
        mother (dict): Network parameters
        father (dict): Network parameters

    """
    children = []
    for _ in range(2):

        child = {}

        # Loop through the parameters and pick params for
        for param in self.nn_param_choices:
            child[param] = random.choice(
                [mother.network[param], father.network[par
```

Breeding children of our fittest networks is where a lot of the magic happens. Breeding will be different for every application of genetic algorithms, and in our case, we've decided to randomly choose parameters for the child from the mother and father.

```
def mutate(self, network):
    """Randomly mutate one part of the network.

    Args:
        network (dict): The network parameters to mutate

    """
    # Choose a random key.
    mutation = random.choice(list(self.nn_param_choices.ke

```

Mutation is also really important as it helps us find chance greatness. In our case, we randomly choose a parameter and then randomly choose a new parameter for it. It can actually end up mutating to the same thing, but that's all luck of the draw.

```
def evolve(self, pop):
    """Evolve a population of networks.

    Args:
        pop (list): A list of network parameters
    """
    # Get scores for each network.
    graded = [(self.fitness(network), network) for network

    # Sort on the scores.
    graded = [x[1] for x in sorted(graded, key=lambda x: x

    # Get the number we want to keep for the next gen.
    retain_length = int(len(graded)*self.retain)

    # The parents are every network we want to keep.
    parents = graded[:retain_length]

    # For those we aren't keeping, randomly keep some anyw
    for individual in graded[retain_length:]:
        if self.random_select > random.random():
            parents.append(individual)

    # Randomly mutate some of the networks we're keeping.
    for individual in parents:
        if self.mutate_chance > random.random():
            individual = self.mutate(individual)

    # Now find out how many spots we have left to fill.
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []

    # Add children, which are bred from two remaining netw
    while len(children) < desired_length:

        # Get a random mom and dad
```

The evolve method is where everything is tied together. Each run of this method is a single evolution. Call it enough times, have enough babies and mutations, and... well, evolution!

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 **56.03% accuracy** with the following parameters:

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

This took 63 *hours* to run. Yikes.

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

Final score? **56.56% accuracy!** Time? *Seven hours*. And the resulting network? Almost identical:

- **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 55.65%. Should've set a random seed.)

So what's the big deal? **The genetic algorithm gave us the same result in 1/9th the time!** Seven hours instead of 63. And it's likely that as the parameter complexity increases, the genetic algorithm provides exponential speed benefit.

What's next?

I'm looking forward to applying this type of hyperparameter tuning to a much more complex problem and network. That will probably require distributed training, so perhaps we tackle that next!

Want the code? It's all right here on GitHub.