# CIFAR-10 Classification

X-Ray

COMP 540 – Machine Learning

Xilin Liu (xal1) – Danwei Luo (dl36)

**Abstract**

*We ran multiple different classification methods on the CIFAR-10 dataset with varying success. At first, we tried using one vs. all (OVA) logistic regression. This yielded an accuracy of about 34%. Our next method was to use support vector machines (SVM) with an RBF kernel, which yielded an accuracy of 55%. Finally, our best results came from running a convolutional neural network (CNN). Our final and best accuracy was 86%.*

## I. Introduction

The CIFAR-10 dataset is a well-known, widely used set of 60,000 32x32 RGB images. These images are split equally amongst ten classes, including truck, cat, airplane, etc. The set is split into 50,000 training images and 10,000 test images, which was augmented with 290,000 unscored images for a total of 300,000 test images. Our goal was to correctly classify as many of the test images as we could.

As the class progressed, we tried new methods that we had learned in class. Specifically, three methods we used, in increasing classification accuracy, were OVA logistic regression, SVMs, and CNNs.

Thankfully, Python has a vast library of tools that we could import and use

## II. Considerations

There are a number of important features that needed to be considered before diving straight into finding a solution. Here we cover a few factors that affected how we approached the problem. We will refer back to this section often when we describe our methods and how they address these issues.

### I. Search Space

First and foremost, the search space for this problem is massive. The initial position of the duel is predetermined, so we do not have to

worry about picking a starting location or direction to face. However, each species is composed of 50 genes, each of which has 4 possibilities resulting in a search space of $4^{50}$. It is computationally infeasible to search every possible solution.

Furthermore, we use a hill climbing algorithm to find a solution. What is the shape of this search space? If it is very hilly, our algorithm will be caught in local optima very often, and we will not find good results. However, if it is mostly flat with one mountain, then our hill climbing method will be exceptional. If the search space is of the former configuration, is there any way to smooth out the space so our hill climbing algorithm needs fewer random restarts or mutations?

## II. Fitness and Scoring

Is there truly a "fittest" species? At first, we thought that there was one "fittest" species out there to find. However, upon closer scrutiny, this may not be the case. It is very easy to imagine species A dominating species B, which dominates species C. Species C may, in turn, also dominate species A, resulting in a rock-paper-scissors scenario. How can we pick which one is the most fittest? Although the evaluation of a duel between two genes is fairly specific, allowing us to determine which species is more fit simply by their score, we are unable to determine with certainty which gene is more powerful in a complex rock-paper-scissors scenario of hundreds or thousands of solutions.

Even if we take the species that beats out the largest number of other species, we have not necessarily found the best. If species A beats out 1000 weak species, and species B beats out only 5 other species who can also beat 1000 other species, it will be equally difficult to assign a "fittest" label to either A or B. While we may say A is stronger because it defeats 200 more species than B, we may decide to submit B as our final species anyways, because it is more likely that our competitors will be more similar to B than A.

From the scoring rules in the project description, we see that the outcome of a duel between two species is not only determined by beating the other opponent in 500 rounds, but also is related to how many rounds it takes to eliminate the other one. So the best gene in the Pacwar should be the gene that can ?kill? its opponent in the most efficient way.

## III. Subgenes

We must remember that a species' genes are divided into six genetic functions, or subgenes, three of which are further divided into four more subgenes. All subgene groups are either three or four genes long. The location of each subgene is important because breaking up these subgenes may result in a drastic change in behavior for the pacmite. For example, during the crossover step of a GA, it would be wise to crossover at a division between two subgenes, instead of in the middle of two subgenes. This way, we maintain the optimal subgene. In a sense, we are actually looking for sets of optimal subgenes. Below, we hypothesize some configurations of the gene that may be more likely to succeed.

That being said, it is equally important to note that an optimal subgene faces the same problem as above, in that there may not be a single 'fittest" subgene. The situation is exacerbated by the idea that individual subgenes may interact either positively or negatively with other subgenes, so it becomes even more difficult to tell whether a specific subgene is stronger or weaker.

- U – The effect of this subgene is more obvious in the earlier rounds, since there are a lot of empty cells in the beginning. It will affect the speed of population growth in the early rounds.

- V – This is also a very important subgene since it will determine a pacmite's efficiency at killing other species and its ability to survive early in its lifetime.

- W –It is obvious that this subgene should not be 0, because when it runs into a wall, turning to another direction is always a better idea.

- X – This gene will also have great effect on the speed of population growth in the early steps.

- Y – For younger mites, since they have a harder time protect allies, it should choose to face other directions. For older pacmites, who are more powerful, facing the same direction to protect its ally is a better choice.

- Z – For younger mites, since their power is low, choosing another direction to generate a newborn mite is a better choice. For older pacmites, facing the same direction to attack the enemy in the next step is a better choice.

## IV. Age and Rounds

These are two separate but somewhat related factors that affect the functionality of a pacmite. Older pacmites will have different goals than younger pacmites, and all pacmites will have different goals in the late game as opposed to the early or late game of a duel. Since younger mites have less power, their main purpose should be generating newborn mites. On the other hand, the older ones should be more aggressive, trying to eliminate enemies and protect friendly mites. Initially, we assumed that the rate of population growth would be a crucial key to victory, since having more troops would result in a better chance to win the duel. The fastest growth is approximately exponential growth, like 1, 2, 4, 8, 15 (where one dies of age). But after we created some species with fast growth, we found that these species still could not beat the example species of only 1s. Since the mites of all the species of only 1s will always face the same direction, their combat strength seems very high. We conclude that although the initial population growth is important, it may be more crucial to have your mites stay together and face the enemy together to win the fight.

## III. Methods

While our GA was the basis for our methods, we also considered other algorithms. GA is a hill climbing algorithm which is not guaranteed to find the optimal solution, so many of our initial methods revolved around trying to solve the problem instead of finding an approximate solution.

### I. Random Generation

Obviously not a real solution. However, this was our initial attempt when first testing out the system, simply to get a feel for how Pacwar works. As expected, it did not work well. Using random generation of genes, we were not even able to beat the provided test genes of all 1s or all 3s. However, from this discovery, we were able to conclude that our GA should be more based on crossovers and less on mutations in order to find an optimal solution, as strong random variations quickly break down the fitness of a species.

### II. Brute Force

The simplest solution to any problem is a brute force solution. This was very briefly considered in our earlier stages. The algorithm basically ran every possible DNA against the rest and

found the species with the most wins. Unfortunately, the search space for the solutions is $4^{50}$ combinations of the possible genes. A brute force algorithm was clearly not a computationally feasible solution. The idea was quickly scrapped.

## III. Pruning

While a brute force solution on the entire search space was computationally infeasible, we considered the possibility of running a brute force solution on a subset of the search space. Using pruning, we could remove entire regions of the search space that we knew with relative certainty would be comparatively unfit. For example, we were given four example genes to start with, each a set of 50 genes of the same value. We found that the genes of only 0s and of only 2s were significantly weaker than only 1s and only 3s, because the pacmites were limited to one dimension. Thus, we could safely remove species which had genes consisting of mostly 0s or 2s. However, this idea was also scrapped for a number of reasons. First, it would not have been trivial to determine which regions we could remove because just as there is no universal "fittest" solution, there is likely no universal "unfittest" solution as well. Second, we had no idea how small we would be able to reduce the search space to, so this solution may have been just as computationally infeasible as the brute force method. Finally, GAs seemed like a better and more natural solution to this problem, and also a more worthy solution to spend more time on.

## IV. Genetic Algorithm

We now come to the magnum opus of our discoveries. At first, our GA was very simple and basic, but we continued to improve it by adding new features or tweaking the parameters in order to overcome the limits of previous versions.

### IV.1 Version 1

Our first GA was designed by following the steps introduced in the lecture. The parameters we chose were as follows:

- Population size: 50 genes
- Selection: 20% elitism, which means the top 10 genes will enter the next population unmodified
- Mating Pool: all 50 genes in the current population will enter the mating pool, and will be picked as parents by a probability proportionate to their evaluation scores

- Crossover: one point, which is randomly chosen from position 1 to 49
- Mutation rate: 2%, which means only 1 value in 50 is changed in each iteration
- Iterations: 10 iterations for one trial (10 was the initial pick for version 1 because we did not know what reasonable number was, and we did not want our algorithm to time out)

We evaluate every species by running a duel between this species and 5 randomly generated genes, and use the sum score as this gene's score. The pseudo code of our GA is as follows:

**Algorithm 1** Base algorithm for Pacwar genetic algorithm

```
Randomly generate 50 genes
for iteration times do
    Evaluate 50 genes in the population and sort them by their score
    Choose the top 10 genes and put them in the next iteration's population
    for i = 0 to (population_size − 10)/2 do
        Randomly choose 2 genes from mate pool as mom and dad
        Randomly choose a crossover point position k.
        Generate two new children: mom[0 : k] + dad[k + 1, 49] and dad[0 : k] + mom[k + 1, 49]
        Randomly mutate 1 value of gene for both children
        Create a new population with these new genes
    end for
end for
```

After building this primary GA, we ran a few trials to see how it works. While it did generate some better species, at least better than those randomly generated ones, we also found some problems. First, the genes we used to evaluate our target gene are randomly generated, so they very weak and easily beaten. Our threshold for which genes are considered "good" is too low, and thus it is difficult to find truly good genes. As a result, the gene we found from this GA was still not strong enough to beat the example genes. Secondly, when one trial finishes, we found that the genes in the population are quite different from each other.

However, our algorithm should only end when the population becomes very similar and no improvement is made upon each iteration. This meant that we had too few iterations.

### IV.2 Version 2

Version 2 attempted to solve the problems we found in version 1.

To solve the former problem of being unable to beat the example genes, we put the example genes into our evaluating gene pool, because that at least guarantees that our algorithm will output a gene that can beat the example genes. Our evaluation gene pool now consists of 3 randomly generated genes and the example genes of only 1s and only 3s. We kept 3 randomly generated genes to prevent the evaluation system from becoming too specific. In the final Pacwar competition, we will not just be competing against niche genes like the examples, but against a group of unpredictable genes. To solve the latter problem, we ran some experiments and found that generally after 100 iterations, all the genes in the population become essentially the same. We chose this new limit as our iteration number.

The result of our second version is excellent. We were able to discover genes that could beat both example genes in under 100 rounds, achieving the max score in our fitness function. However, we discovered two new problems. First, although each trial will end after 100 iterations, we do not have a way to connect each trial. For each trial, we should be able to find a gene that can beat the result found from previous trial. This way, we can continuously improve the quality of our best gene. In addition, our criterion for finding a best gene, beating only the example genes, is not strong enough if we want to compete in the final tournament. Connecting each trial will ensure that every new trial will output an even stronger gene than we had ever produced. Secondly, for each trial, we always start with a population of 50 randomly generated genes. It generally takes a large number of iterations to find a good gene that can beat almost all the genes in the evaluation pool because a random gene is weak. With a growing evaluation pool size, this becomes extraordinarily inefficient. For particularly large evaluation pools, our algorithm may fail to find a champion gene within our iteration limits.

### IV.3 Version 3

This is almost our most recent version, which solves all the previous problems we have discovered.

To connect genes from different trials, we put the best genes we found from each trial into the evaluation gene pool. This forces each the new trial to find a new best gene that can beat all the previous ones. In addition, this also increases the threshold for what we consider a "good" gene at every trial. Initially, a "good" gene only needs to beat the example genes. Subsequent trial thresholds require the best gene to be able to beat the example genes, and all genes that were previously shown to beat the example genes as well. However, continuously adding better genes to the evaluation gene pool exacerbates our second problem because the evaluation gene pool now continually grows in size with more trials. After each trial, our evaluation gene pool will grow by one more gene. We somewhat combat this problem by removing the example genes of only 1s and only 3s, as well as older genes which are weaker than new ones.

To solve the second problem, we start each trial with a population that has 49 randomly generated gene, and 1 gene that is actually the optimal gene found from previous trial. Since optimal best gene is usually better than randomly generated ones, it will help the whole population to find a good gene very quickly via crossover. In addition, another benefit of adding this gene to the population is that it guarantees that we will not get worse result in each trial, even if we cannot find the a better one. This is because we keep the top 10 genes unmodified for each iteration, so the best gene from previous trial will likely come out on top, especially relative to the initial randomly generated genes. Thus, if we cannot find a better gene that has higher evaluation score the original best gene, our algorithm will still output the original gene. At the end of each trial, our GA will either find a better one which we will add to the evaluation gene pool, or simply the current best gene from previous trials, which we will ignore. In this way, our GA is guaranteed to make progress in the right direction after each trial, or at least not make progress in the opposite direction.

This version is near completion. However, one glaring problem that still exists is the speed at which our GA completes given larger and larger evaluation gene pools. In addition, there may be no improvement for many trials in a row, because it becomes more difficult to find a gene that can beat all the other genes.

### IV.4 Version 4

We now present our final version, which fixes essentially all of the problems we have found.

To solve the large evaluation pool runtime problem from version 3, now we divide our procedure into two stages.

1. We run our own brute force round robin tournament in our evaluation gene pool, and pick up the top 10 genes as a miniature evaluation gene pool. The tournament is exactly the same as the final one in class, which means each gene duels against all the other genes in the evaluation gene pool, and ranked by the sum of the scores.

2. We repeat the same algorithm from previous versions, except now we use the mini evaluation gene pool to evaluate the score. Then after we find about 10 better successor genes in this stage, we add these new genes back to our main evaluation gene pool, and repeat back to stage 1.

This was the final version of our GA, which we just keep running continuously it until the deadline. Now that we are on the final leg of our journey, all the successor genes look identical, with only one or two bit differences. Mutation is actually the key in the final stage searching. Our final gene is selected by running one last tournament in our main evaluation pool and picking the winner.

## IV. Reflections

Pacwar is a very difficult problem to solve. There seems to be no limit to how good your gene may be, as each new solution we provided significantly improved our algorithm. At the same time, while we initially placed 6th in the class rankings, we later only placed 13th, despite having grown very much since the previous rankings. However, in the final tournament, we placed 4th, which we were both surprised and overjoyed about! While we missed out on the top three champion spots, we believe that if we had slightly more time to run our final version, we may have found the best gene via mutation. Alternatively, we could have brute forced the final step to find which mutation resulted in the absolute best.

One problem we faced throughout was that each trial's best gene was indirectly based on beating the example genes. This led us to believe that our solutions were niche solutions that could only beat the examples and not other ones. However, it seems that after enough iterations, mutations, and crossovers, we were able to overcome this hurdle and produce a gene that could beat more than just the example genes. Having more trials and greater mu-

tation/crossover rates probably significantly helped us escape the niche.

One impressive feature of this project is how easily it can be extended past simply artificial intelligence in computer science. We are essentially directly modeling a real world battle between various but similar species. Clearly, Pacwar is a simplification of the forces of natural selection, but we can easily extend what we have learned here in biology.

In general, these solution finding algorithms can be applied to more than just biology. We can use it to query the search space of any problem with competing solutions and use the common traits of the algorithms we used in this project, such as crossover or mutations, to approximate an optimal solution.