Experiments on Mixed Genetic and Gradient-Driven Evolution for Machine Learning

Gavin Cooke, Luca de Alfaro

Introduction:

In classic machine learning, a model is improved by using gradient descent to adapt its parameters and decrease the loss of the model's predictions. This process is repeated for a number of iterations until the model reaches a satisfactory performance on its training data. In this experiment we combined this standard approach with techniques similar to genetic algorithms and examined the results.

The motivation for this approach came from how genes work in nature. For a given population of some species, each member of the population has some set of genes. The values of these genes, alleles, govern all sorts of characteristics about that individual. More importantly for this project, a given population has many possible combinations of alleles for its genes that result in individuals which all perform similarly well and the variety in these individuals provides the population with resilience against hazards such as diseases which might target one allele more effectively than others. We sought to investigate if a population of neural networks developed using a combination of gradient descent and recombination steps using the parameters of the models as genes could receive any similar benefits.

Specifically, we investigated whether this approach produced models which perform well, how much variety there would be in the parameters of the final models, and whether it provided models with any resistance to adversarial attacks. For adversarial attacks, the question was given some population of models trained in such a way, with multiple alleles for its genes, how well would an adversarial attack generated on one member of the final population translate against the other members and how well would an attack generated on a child model of the final population translate to other children.

Method:

The general idea of this method is that, instead of training a single model, a population of models is trained using alternating steps of gradient descent and genetic recombination. For example, the models may be trained using gradient descent for a few epochs before, then the best models after the training are recombined to generate a new population, this new population is trained for a few epochs, and the process is repeated for as many generations as required.

A key part of this is the recombination of parameters from the original generation. This is done with a method called recombine, which takes in two parent models as input and outputs a single child model. In this implementation, each individual layer of a given model is considered as a gene and a child is created by initializing each of its layers to the corresponding layer of one of its parents, chosen randomly. So a child model may be initialized with the first layer of its first parent and the rest from the second parent, or any other potential combination.

Another key part of this operation is how to select the best models from the final population to use for this recombination. In genetic algorithms this is done by defining a fitness function to rank the members of the population in terms of some metric. In our experiment, we defined the fitness of each model as its accuracy on the test data. Using these accuracies, we sorted the population in descending order and selected the best half to perform the recombination steps to generate the next population.

The definition of what is considered a gene is an important consideration for this kind of experiment. As mentioned before, we consider each individual layer of a model to be a gene, but we used two distinct model architectures which will be discussed in the next section.

Model Architecture:

The first and most straightforward architecture used in our experiments is what we refer to as horizontal genes. This implementation uses a three layer fully connected neural network. The first layer receives the image as input and has 256 neurons, the second layer has 128 neurons, the final classification layer has 10 neurons. The name ‘horizontal genes’ refers to the classical picturing of a neural network, where there is a bottom layer, middle layer, and input layer and each layer is a horizontal block of neurons.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
     GeneticLinear-1                [100, 256]         200,960
     GeneticLinear-2                [100, 128]          32,896
     GeneticLinear-3                 [100, 10]           1,290
================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
----------------------------------------------------------------
```

A summary of the horizontal gene model architecture used in our experiments. The 100 in Output Shape refers to the batch size.

The second implementation is what we refer to as vertical genes. This is once again a fully connected neural network which looks very similar to what is shown above, the difference comes from how the second layer is handled.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
     GeneticLinear-1                [100, 256]         200,960
     GeneticLinear-2                 [100, 64]          16,448
     GeneticLinear-3                 [100, 64]          16,448
     GeneticLinear-4                 [100, 10]           1,290
================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
----------------------------------------------------------------
```

A summary of the vertical gene model architecture used in our experiments. The second layer from the horizontal gene implementation has been split into two layers.

In this implementation, the first and last layers are identical to the first and last layers of the horizontal gene implementation. The combination of the middle two layers, labeled 2 and 3 in the image above and referred to as 2a and 2b in the plots later on, corresponds to the middle layer of the horizontal gene implementation. These layers both work by taking the first layer as input and then each giving an output of size 64, both of these outputs are concatenated together to form the size 128 input expected by the classification layer. The name ‘vertical genes’ comes from this method of dividing the middle layer into two halves in parallel, which can be thought of as a vertical division.

Each linear layer is instantiated from a custom extension of PyTorch’s linear layer class. This extension functions identically to a standard linear layer, but contains some additional functions to assist with performing the recombination steps.

For other model hyperparameters, the optimizer used was stochastic gradient descent (SGD), the device used was the CPU provided by Google Colaboratory, and the training and testing was performed with a batch size of one hundred.

Experimental Results:

To get out results we ran experiments for both the horizontal gene implementation as well as the vertical one. For both of these, we ran experiments with the following hyperparameters: the population contained twenty models, the simulation ran for six generations, and each model in the population was trained for one epoch after the recombination step in each generation. The dataset we used for these experiments was MNIST.

For comparison, we used a baseline model initialized with the same network framework as a member of the horizontal gene population and trained using stochastic gradient descent for six epochs. In terms of accuracy, the difference between the performance of the models of the final population and this baseline was negligible. All models generally ended with accuracies in the 96%-97% range.

Naturally, the population simulation takes significantly longer in terms of runtime than training a single model. Simulating six generations takes around thirty-five minutes, using the CPU in Google Colaboratory, while training a single model for six epochs only takes around one minute.

For the question of variety, we measured this using the cosine similarities of each gene across the population. At the beginning, this similarity value is near zero, since each model is a unique initialization, but as the simulation progresses they become more similar overall due to learning from the same data and possibly inheriting from the same parent models.

We found that when simulating beyond six generations, the variety in the models often disappeared completely and the entire population was nearly identical in terms of cosine similarity. At six generations, generally the population divided itself into a few somewhat different groups, similar to alleles that occur in genes in nature. For some reason, this behavior was most pronounced in the output layer of the models.

None of the models in the populations we simulated showed any resistance to adversarial attacks on their own, the accuracy of each model was generally less than 1%. For attacks developed against one model from the final population against the others, the models showed little resistance to this kind of attack either, accuracies were generally around 1%-3%.

For a final experiment, we developed an attack against a child of the final population and then tested it against other children of the final population. In populations where the cosine similarities had not completely converged to 1 yet, this generally achieved accuracies around 5%-15%. In one simulation, accuracies around 33% were achieved, but this was unable to be replicated consistently. In populations where the cosine similarities had converged to 1, the accuracies were around 4%.

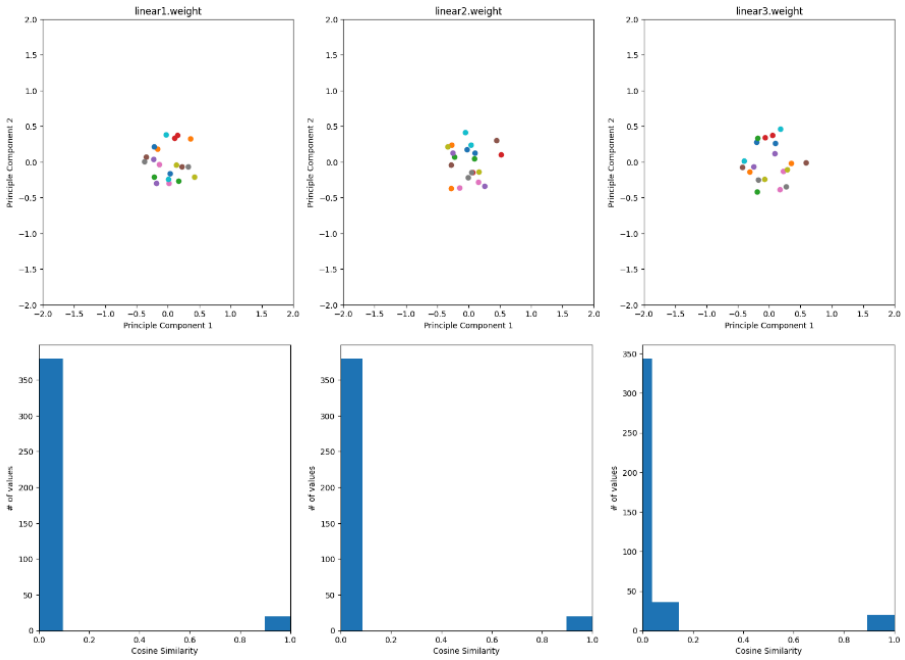
Conclusion:

In conclusion, simply adding genetic crossing like we did in this experiment does not provide any meaningful advantages to a machine learning model compared to training it in a traditional manner using gradient descent. A potential cause of this is that there is not enough incentive for diversity in our implementation, since we only use accuracy to determine fitness. In the adversarial attacks test, the population that retained some diversity did perform slightly better than the others. To test this more thoroughly, there would need to be some motivation given for diversity in the parameter values of the final population. Perhaps by changing the fitness function to model advantages given to genetic hybrids in nature, or by adding something such as mutations to offset the loss in diversity which occurs in each generation.

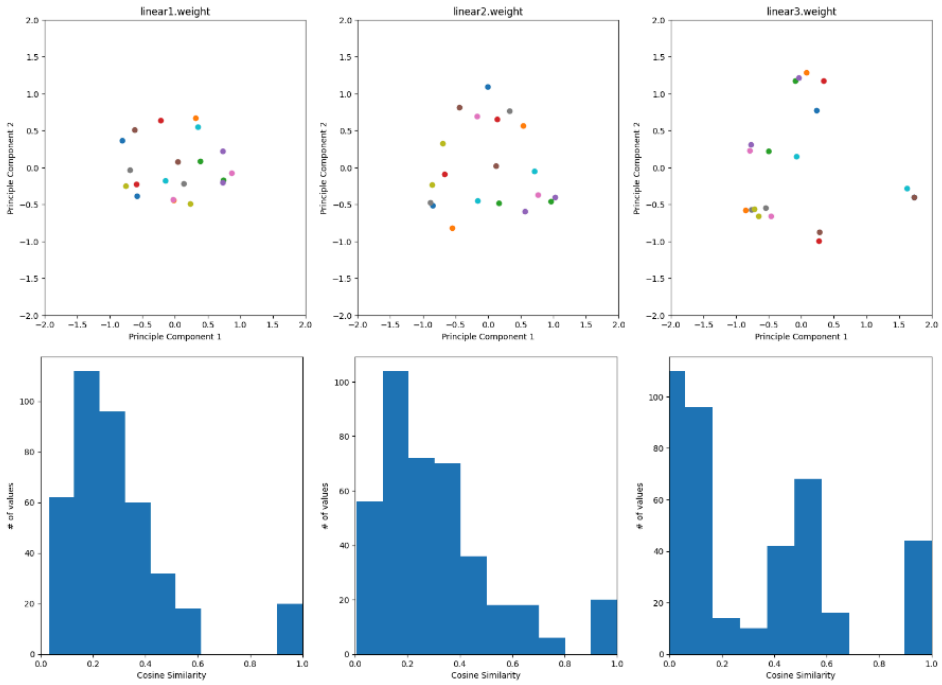
Cosine Similarity Graphs:

Below are some of the plots from the last simulation run as of the time of this report. Each set of two rows of graphs represents a generation in terms of the cosine similarity between the layers in the population of models. The first row of graphs in each generation plots the cosine similarities in a scatter plot by performing principal component analysis on the 20 by 20 cosine similarity matrix for each layer. The proximity of the dots to each other can be interpreted as how similar the layers represented by those dots are to each other, this is most meaningful once they have separated themselves into distinct clusters. The second row of graphs consists of histograms which count the frequency of each cosine similarity value in the matrix mentioned above.

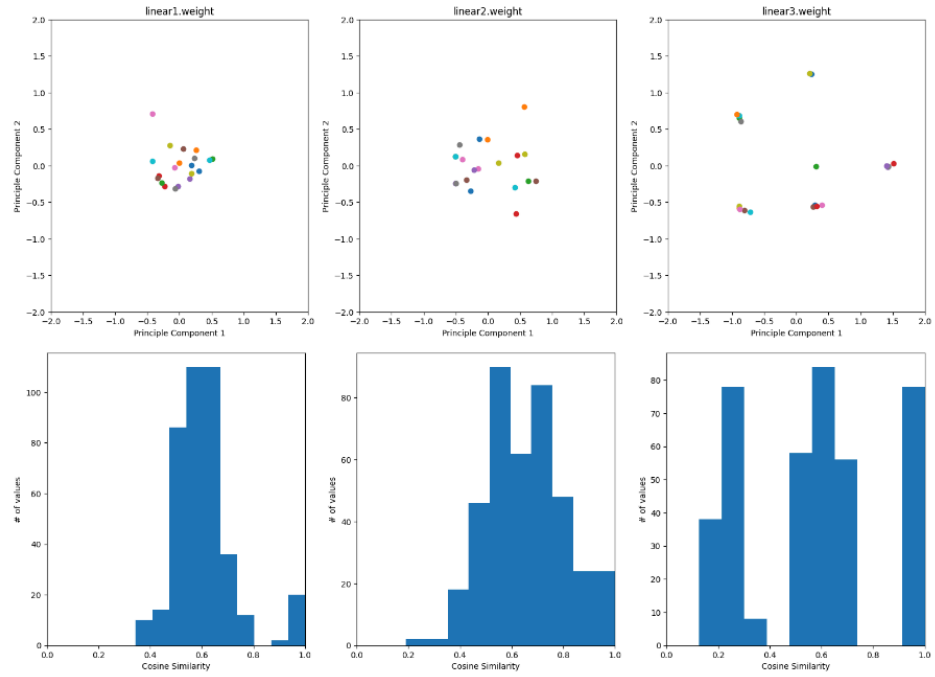
This first set of three generations represents generations zero, three, and six from the last simulation run of the horizontal gene population. In this simulation the cosine similarities of the population did not converge to 1 and there remained some diversity as of the final generation.



Gen 0: Initial generation of horizontal gene population. All models are random initializations and bear no resemblance to each other in terms of cosine similarity.



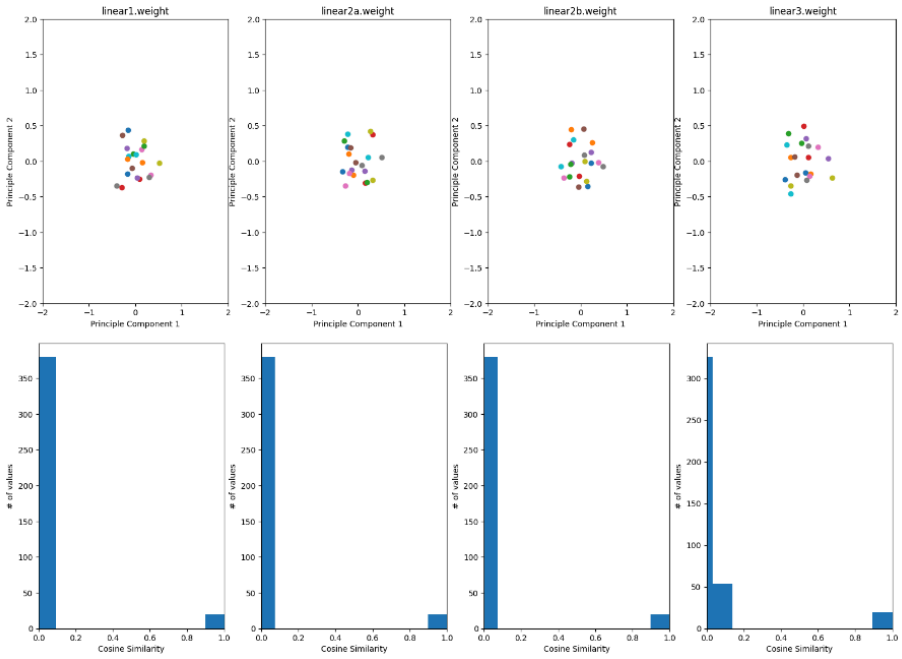
Gen 3: After three generations of the simulation, the similarity scores are higher but the models are still mostly distinct. Clusters are beginning to form in the PCA graph for layer 3 where groups of models likely inherited from the same or similar parents.



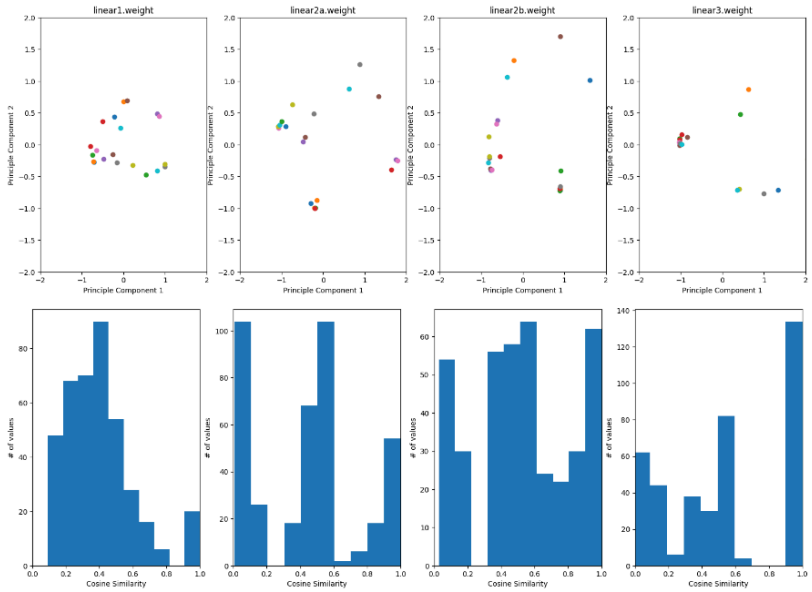
Gen 6: The final population of this simulation. Most models are fairly similar to each other and the clusters in layer 3 are now well defined. There is little to no clear clustering visible in the

other layers, which is standard for a population where layer 3 has not yet been reduced to one or two clusters.

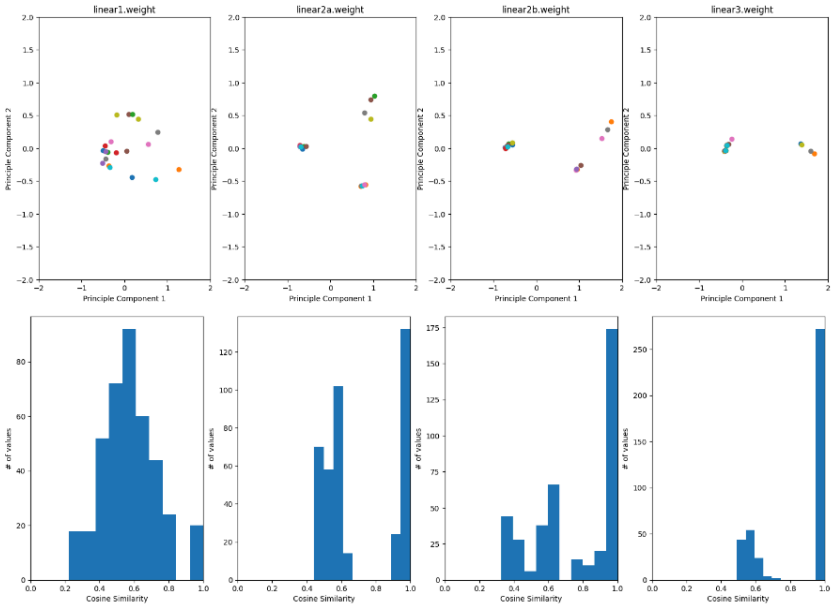
This next set of four generations represents generations zero, four, five, and six from the most recent simulation of the vertical gene population as of the time of this report. In this simulation the cosine similarity values of the final layer converged to 1, and nearly converged to 1 for both halves of the middle layer.



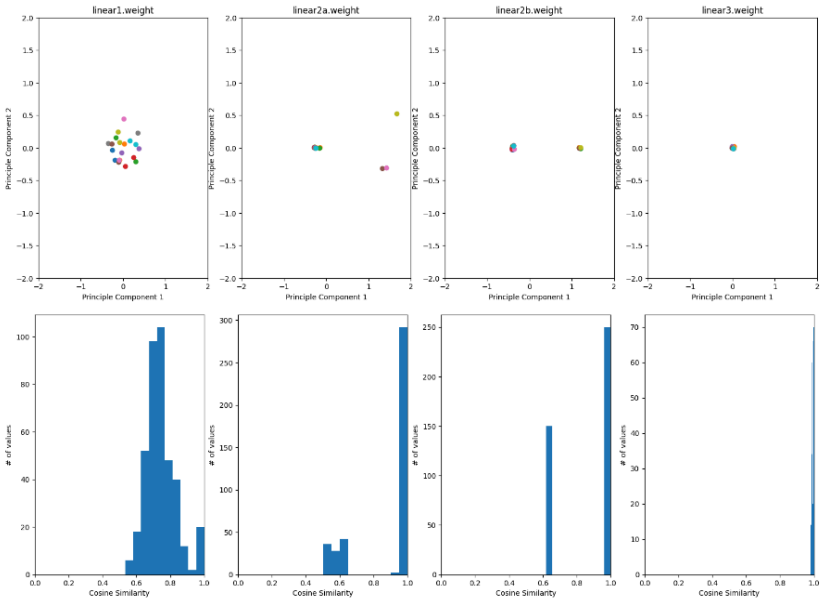
Gen 0: Initial generation of vertical gene population. All models are random initializations and bear no resemblance to each other in terms of cosine similarity.



Gen 4: After four generations similarity scores high for some models but remain low for others, three clusters have formed in the final layer. Some clustering is visible in layers 2a and 2b.



Gen 5: After generation 4, the similarity scores have suddenly increased by a significant amount. One of the clusters has disappeared from the final layer, possibly due to random chance or comparatively lower performance on the test data. Layers 2a and 2b have formed 2-3 clusters each



Gen 6: The final layer has now fully converged to a single cluster, meaning that every member of the population has nearly identical parameter values for that layer. Layers 2a and 2b have nearly converged as well. Layer 1 has not gone through this process, but each model still has somewhat similar values for its parameters.

Resources:

The code for this project, as well as a the complete collection of graphs and other output from the last experiment run using it, can be found in the Jupyter Notebook in the following Github repository: https://github.com/GavinCooke2000/Genetic_ML.