Genetic vs. Gradient Algorithms for Neural Network Training

Andrew Martin
Department of Computer Science
University of New Hampshire
Durham, NH 03824
ajm1272@usnh.edu

Abstract

We will be using OpenAI's gymnasium to compare two different algorithms for reinforcement learning, Genetic and Gradient Descent. Gradient Descent is the usual training algorithm for neural networks, and focuses on small, calculated adjustments to the model, which add up over time to create a suitable network. The Genetic Algorithm simulates evolution by creating generations, then "breeding" the highest-performing specimens to eventually create a working model. To compare these algorithms, we test them with the Cart Pole environment. From this environment, we use Weights and Biases to collect data from our training and compare the two algorithms. In the end, the genetic algorithm trained the fastest and was the more general solution, while the gradient descent model offered a more consistent improvement from episode to episode.

Introduction

Our paper seeks to compare the effectiveness of two algorithms solving the same task, training neural networks using reinforcement learning. This is a near direct offshoot from our classes talking about unsupervised learning and neural networks. In class, we learned about backpropagation, the most popular method for training neural networks. We also briefly mentioned the genetic algorithm in class, but not within this context. We figured it would be interesting to compare the genetic algorithm, which isn't typically used with neural networks, and see how it stacked up against the more popular gradient descent algorithm.

Initial Problem: QWOP

The problem that we originally wanted to solve was teaching a neural net to play QWOP using our two algorithms. However, after some experimentation, it quickly became clear that QWOP was not feasible. For one, the game is entirely Javascript based, which is a language that neither Jon nor I have familiarity with. Second, the game is highly locked-down, meaning we could not interact with it through simple browser commands, and would instead have to simulate keypresses and read data directly from the screen. We also couldn't find any reasonable clones of the game, and figured it would be too tedious to program the quirky physics ourselves.

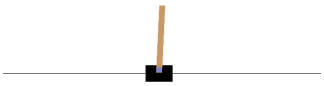

Figure 1: The Cart Pole environment (Farama Farama2022)

Cart Pole

After discovering that QWOP was a no-go, we turned to OpenAI's gymnasium, an open-source python library with many AI testing tools already included. One of the major benefits of this is that the testing API was already set up, allowing us to easily interact with the environment. We decided to use the cart-pole environment, as it was rather simple to set up and would be a good fit for both of our algorithms.

The goal of the cart-pole environment is to apply forces to a cart such that the pole balanced on it stays upright. This is a continuous-state problem with discreet outputs (Farama Farama2022). At each time step, we receive the following description of our state:

- Cart Position
- Cart Velocity
- Pole Angle
- Pole Angular Velocity

From this, our policy should be able to determine what the best action is. Our action space is very simple, with the agent being able to apply a constant force to either the left or right side. Not pushing the cart is not an option.

Another major benefit to cart pole is that it has a failure state. If the pole reaches an angle of greater than 12° in either direction, the program terminates. The agent also fails if it propels the cart off-screen. To ensure our training doesn't go on forever, we have a hard cut-off at 1,000 frames. If the agent makes it to the cut-off, they win.

Rewards Since the ultimate goal of the agent is to survive as long as possible, our reward function is rather simple. For

every time step (frame) that our agent survives, it gets 1 point added to its score. Therefore, the total reward at the end of a run is simply how many frames the agent survived.

Policy Optimization

Our ultimate goal is to "teach" our agent how to balance the pole reliably through trial and error. The output of this reinforcement learning process is called a policy, and represents what action our agent will take given a state. In order to optimize our policy, we simply want to run through many iterations, collecting data and improving our model to increase our rewards. We chose a neural network to represent our model, as the physics involved makes the environment nonlinear.

Neural Networks

Neural networks are collections of virtual "neurons" called perceptrons split up into layers. Each of these neurons are connected to every neuron in the previous layer and to every neuron in the next layer. These connections allow the neurons to propagate signals through the layers of the neural network (IBM IBM2024).

Each neuron represents a complex linear function that takes an input from each of the neurons in the previous layers. This means that each perceptron spans a space with $n+1$ dimensions where n is the number of neurons in the previous layer.

The most basic neuron has one input and one output, and can be represented by the linear function $y = wx + b$. Here, our output y is a sum of our input x times our weight (or slope) w with bias b. Each additional input has its own weight and bias. Because the neuron represents a linear function, no matter how many inputs there are, it only produces one output, which is then sent to an activation function.

Activation functions add a nonlinear influence to the neural network, which allows it to represent much more complex problems. In fact, without activation functions a neural network would behave completely linearly and there would be no point in having more than one layer (Brodtman Brodtman2021).

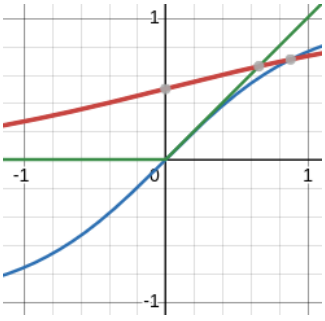

Figure 2: Different Activation Functions

The most basic activation function is a sigmoid (red), and nearly all activation functions such as tanh (blue) follow a similar shape to sigmoids. We decided on the RELU (green) function, as after testing, we determined that it converged faster than other activation functions.

Approaches

Our goal is to compare two possible training algorithms, and we settled on a basic gradient descent method and the genetic algorithm. However, there are many other ways to train a neural network, many of them based on gradient descent. A few of these algorithms will be mentioned later in the paper.

Early on in the project, I knew that I wanted to try implementing a more typical training methodology, while Jon was interested in the Genetic Algorithm. We decided I would implement gradient descent while Jon built our Genetic Algorithm.

My initial plan was to write my own Proximal Policy Optimization, which was a bit too complicated with our time constraints. Instead, I wrote a Vanilla Policy Gradient algorithm, the most basic backpropagation algorithm.

Gradient Descent

In most reinforcement learning situations, gradient descent or some variant is used. The most basic form of Gradient Descent is known as Vanilla Gradient Descent. Below is the pseudo-code for this algorithm (OpenAI OpenAI):

Algorithm 1 Vanilla Gradient Descent

Require: Parameters θ are initialized

while Step $s <$ Maxsteps **do**
  Collect rewards r and actions O by running policy on environment
  Discount rewards to give higher weights for decisions made near the end of the run (Optional)
  Compute Advantage estimates $\hat{A}$, representing how "good" each action in O was
  Create Target $T = O + \hat{A}$
  Compute Loss Function Between T and O
  Backpropogate Loss Function (Find Gradients $\hat{g}$)
  Adjust policy with $\hat{g}$ using learning rate α:

$$\theta = \theta + \alpha\hat{g}$$

end while

The Loss Function

Loss functions represent the difference between our policy and some better policy. The idea of gradient descent is to minimize the loss function, which is typically some inverse of the reward. This way, as our policy decreases its loss function, it nets more rewards (Microsoft Microsoft).

There are many different loss functions, but after some trial and error, I determined the optimal loss function is Mean Squared Error, the function taught in class. The name is rather self-explanatory, but below is the formula:

$$MSE = \frac{\sum_i (O - T)^2}{i}$$

We tested some different loss functions, and below are the results:

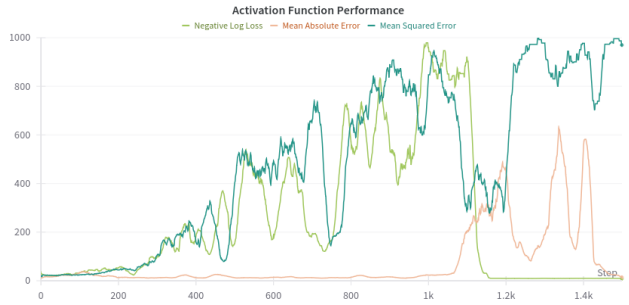

Figure 3: Activation Function Results

Backpropagation

One of the most important steps of gradient descent is backpropogation. During this portion of the algorithm, we determine how much each of our weights and biases contributed to the loss function. This gives a good estimation of how we can change each parameter to improve the policy. In this way, backpropogation gives us what are called gradients, which represent a direction and distance we should move our parameters to decrease the loss function. This is where the word ”gradient” comes from in gradient descent. By repeatedly moving down this gradient, our network can eventually land in a minima where we can no longer improve our policy from. Usually, this will give our policy good performance, but it will hardly ever be perfect (IBM IBM2024).

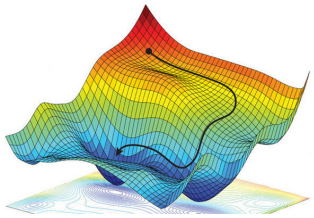

Figure 4: Gradient Descent Visualization (Amini Amini)

In order to backpropogate loss, the algorithm must take a partial derivative of the whole neural network relative to the weight or bias it is calculating. Because we have many layers, each with a nonlinear function, the algorithm uses chain rule to separate the loss into each individual parameter’s effect on loss.

Step Size and Variants of Gradient Descent

Once gradients have been calculated, the next step is to adjust the weights. In vanilla gradient descent, a constant step size is used, meaning weights and biases are adjusted using a constant portion of the gradient. One of the biggest shortcomings of gradient descent are what is called local minima. This is when a ”valley” is found in the data and the step size is too small to climb out. This can be fixed by increasing the step size, but that can result in the algorithm stepping over possible solutions.

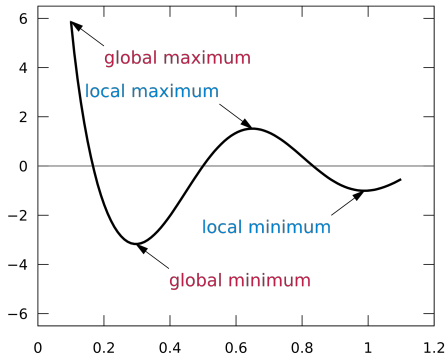

Figure 5: A Local Minima (Wikimedia Wikimedia)

One of the more common solutions to this problem is to employ an optimizer. Most algorithms based on gradient descent use optimizers to tweak the magnitude of changes to the policy, allowing the algorithm to be sensitive enough to find nearly invisible minima while also being broadly aware enough to try exploring outside of any minimum it does find.

One of the more common variants of gradient descent, Adam, does just that. It uses a moment-based optimizer that estimates the rate of change of each parameter. This way, it can adjust its learning rate for each individual parameter as it learns, keeping the step size consistent and avoiding steps that could put it in a local minima (Kingma and Ba Kingma and Ba2015). In the future, it could be interesting to implement an Adam Optimizer.

A rather new algorithm to the scene is PPO, or Proximal Policy Optimization. This algorithm was proposed by OpenAI in 2017, and has quickly become one of the more popular algorithms due to its ease of use on highly varying problems. I was originally planning on implementing PPO, but time constraints led me towards the vanilla gradient algorithm. PPO uses many combined tactics to adjust its step size, but its biggest contribution is called a clipping function. This limits the maximum step size that can be made to the algorithm, allowing it to not step over minima. In addition to the clipping function, PPO uses an adapting learning rate to start by taking large steps, then slowing down as it gets closer and closer to a minima, as well as many other optimizations (Schulman, Wolski, Dhariwal, Radford, and Klimov Schulman et al.).

Genetic Algorithm

The genetic algorithm takes a completely different approach to training a neural network. Rather than calculating the path towards an optimized policy, it simulates evolution. First, a population is created by creating many neural networks with randomized parameters. Next, the population is run through the simulation, with each child being given a reward. From these, the best are taken to combine their parameters or mutate, while the weak are destroyed (MATLAB MATLAB).

This repeats for a set amount of generations. After the last generation, the best policy is taken.

The genetic algorithm is not frequently used in neural network training. This is because of a few different reasons. First, the genetic algorithm typically takes many steps before it starts improving. Second, there is no guarentee that the algorithm will ever improve, since if it doesn't find suitable policies, its only hope is to randomly generate one. Finally, there is no way to prove that the algorithm will converge on a minimum, or even generate a policy that works better than taking random actions. One benefit of the genetic algorithm is that because it randomly samples from all possibilities, it is harder for it to get stuck in a local minima (Chaudhary Chaudhary2022). We expected the genetic algorithm to be able to solve the cart-pole problem, we figured it would be very inefficient compared to the more typical gradient approach.

Evaluation

We ran our tests on the Cart-pole problem using OpenAI's Gymnasium in Python. I programmed a vanilla gradient descent algorithm, and Jon programmed a genetic algorithm. I also included pytorch's implementation of the Adam algorithm to see how a properly optimized gradient descent algorithm compares. We collected data using Weights And Biases, so we could compare our algorithm's training speed over both time and number of iterations. All of our data is based on training through 1500 episodes with a max episode length of 1000.

One thing we've noticed is that both algorithm's raw test scores are very unstable. The unoptimized gradient algorithm is unstable as it tends to , so in addition to the raw data, we will show a rolling average over the last 25 episodes. The genetic algorithm's results are unstable because it is constantly throwing in new randomized networks into the tests, so in addition to its raw score, we will be tracking the best child from each generation.

In addition to our training data, we collected the average of 10 runs of each of the algorithms to determine which algorithm was the most robust. We also did a visual comparison of the two algorithms' strategies.

Data

Through visual inspection, it is clear that the genetic algorithm is more stable than the gradient algorithm. This is likely because the gradient algorithm is unoptimized and gets stuck in a local minimum.

In order to quantitatively test our algorithms' stability, we ran each algorithm with a maximum episode length of 100,000 frames. Below is a graph of the average of 10 runs of each of the algorithms.

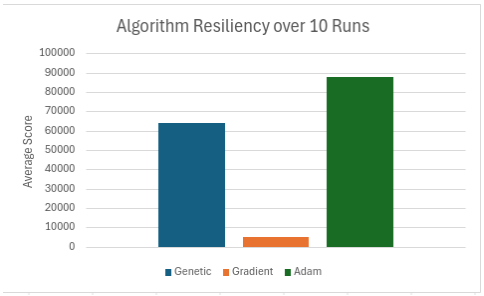

Figure 6: Algorithm Stability

We then ran each of our algorithms a few times and picked the most "typical" performance from both. Below is the raw data combined with some normalization factor, a rolling average for gradient descent, and the max from each generation for genetic.

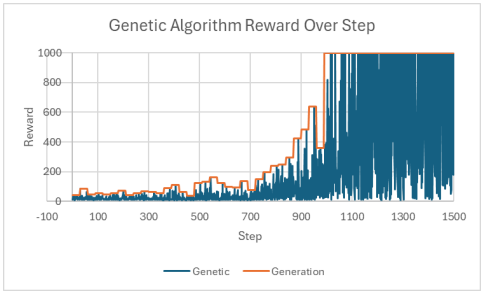

Figure 7: Genetic Results

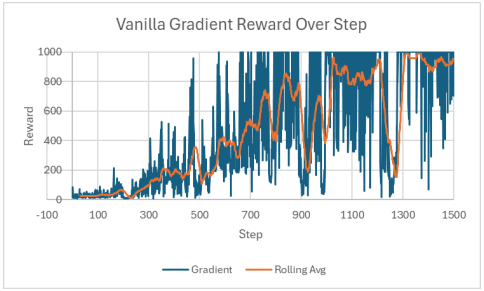

Figure 8: Vanilla Gradient Results

Figure 9: Adam Results

Below is a direct comparison of our three algorithms. Note how both Gradient algorithms have a ”hiccup” around episode 900.

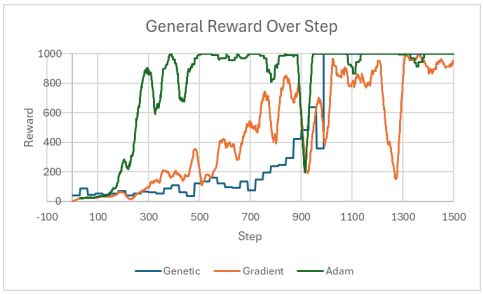

Figure 10: All Algorithms Per Step

We also wanted to compare the training times of each algorithm. Typically, algorithms that take longer to complete 1500 episodes means that their agent spent more time alive.

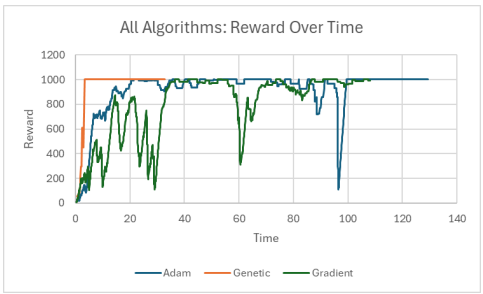

Figure 11: All Algorithms Training Time

Gradient Performance

My gradient algorithm didn't quite perform how I hoped it would. In its current state, it trains slowly, isn't stable, and doesn't generalize very well. However, with the proper optimizer, it holds lots of promise.

Genetic Performance

The genetic algorithm performs surprisingly well. Its high failure rate allows it to try many approaches very quickly, leading to the fastest learning rate over time. However, it does take about 700 episodes, or 14 generations, before it even starts learning. Once it does start improving, it quickly skyrockets and learns the optimal solution by generation 20.

Summary

In conclusion, while the Genetic Algorithm is likely not the optimal general algorithm for training neural networks, it performs the best on this problem. This is likely due to a failure state allowing it to very quickly skip over its weaker children, allowing the strong to learn from their mistakes. My gradient descent algorithm does work rather well, but due to a lack of optimization and the ability to change learning rates it was very bad at generalizing. In the future, I would love to try adding an optimizer to this project, as well as test the algorithms on other gymnasium environments. Overall, it was very interesting to see the two algorithms go head-to-head.

References

Amini, Alexander. *Gradient Descent Visualization*.

Brodtman, Zack. “The importance and reasoning behind activation functions.” Nov. 2021 *Medium*.

Chaudhary, Srishti. “Genetic algorithm applications in machine learning.” Nov. 2022 *Genetic Algorithm Applications in Machine Learning*.

Farama. “Gymnasium documentation.” 2022 *Cart Pole - Gymnasium Documentation*.

IBM. “What is a neural network?.” Feb. 2024 *IBM*.

Kingma, Diedrik P and Jimmy Lei Ba. “Adam: A method for stochastic optimization.” 2015 *Arxiv*.

MATLAB. “What Is the Genetic Algorithm?.” *MathWorks Help Center*.

Microsoft. “Microsoft/AI-for-beginners: 12 weeks, 24 lessons, AI for all!.” *GitHub*.

OpenAI. “Vanilla policy gradient: Spinning Up.” *Vanilla Policy Gradient - Spinning Up documentation*.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Openai baselines PPO - Proximal Policy Optimization.” *OpenAI*.

Wikimedia. *Extrema Example*.