Evolution

Last updated on November 20 to include GUI enhancements in Part 3.

Help Session: Thursday, November 21st, 2:30 PM (in class)

Design Section: Tuesday, December 3rd - Thursday, December 5th

On-Time Handin: Saturday, December 14th, 11:59 P.M.*
*Please note that although the due date is 11:59pm, the SunLab will close at 10:00pm

Late Handin: Monday, December 16th, 5 P.M.
You cannot use a late pass on this assignment.

Flappy Bird demo: cs0150_runDemo EvolutionPart1
Full Evolution demo: cs0150_runDemo EvolutionPart2
Demos may not work over ssh!

To install: cs0150_install Evolution
To handin: cs0150_handin Evolution

Evolution is a capped project this semester since we are piloting it for the first time.

Introduction

Machine learning is one of the most rapidly growing branches of computer science. Unlike many positions in the software engineering industry, where comprehensive foundational skills offered in our core curriculum are sufficient experience for an entry level job, most ML-focused positions require prior experience developing ML projects. For more background information about the context and motivation for this project, see the Background and Motivation document!

Your task is to design a reinforcement learning algorithm that can learn how to play the 2013-hit Flappy Bird. You’ll be implementing a style of learning called Neuroevolution, which uses concepts from evolution in nature to make agents “evolve” into effective game players.

This is a brand spankin’ new final project, and there are many new concepts that have not been covered in class. Therefore, it is important that you have a firm grasp of the material covered in the course, feel comfortable with researching some concepts independently, and work well with slightly less structure than has been provided on previous projects. While we do not want to
discourage people from choosing Evolution as a final project, we will be screening applicants in order to make sure this is the right project for you. If you have questions or concerns about this policy, please reach out to the Evolution HTAs, Noah and Julie!

**Table of Contents**

*Introduction*

*Table of Contents*

*Assignment Specifications*

**Part I: FlappyBird Game - DUE BY SECTION**

*Bird*

*Pipe Generation and Movement*

*Collisions*

*Sidebar*

*Coding Part I Incrementally*

**Part II: Thinking and Learning**

*Thinking*

*Neural Networks*

*Input Nodes*

*Forward Propagation*

*Weights*

*Activation Function*

*Learning*

*Neuroevolution*

*Fitness Tracking*

*Population Class*

*Selection*

*Mutation*

*Coding Part II Incrementally*

**Part III: Optimizations**

*Write-up*

*Network Visualizer*

*Evolution Minimum Functionality Requirements*
Assignment Specifications

Before reading this handout, it will help to understand how the game works. You may want to play the game Flappy Bird for a bit, and should definitely run the demo a few times!

In this program, you will first implement a clone of the game Flappy Bird. This game features a bird whose goal is to fly as far as possible through a series of never-ending pipes while avoiding collision. The bird accelerates downwards due to gravity and jumps when the user presses the spacebar.

Once you have created a working version of Flappy Bird, your program should generate an entire population of birds at once that decide when to jump using a neural network. These birds will play the game very poorly at first, but will evolve after each generation using the Neuroevolution algorithm. Eventually, a handful of birds from every generation will be able to fly through many pipes without collision.

After achieving this, you should try to optimize your program. How can you achieve a working solution more quickly, or generate a population with more birds that have found an optimal strategy? You will need to describe these optimizations in your writeup, so try different strategies and see how they affect your game. Feel free to be creative with this part, as there are many ways to improve your program!

**In order to make sure you have enough time for Parts II and III, you will have to complete Part I by the design section. This will be 5% of your final grade.**

To meet full functionality, you will need to complete the following:

- **Part I: Flappy Bird Game**
  - A Bird that jumps when the spacebar is pressed.
  - A collection of Pipes that continue to generate and move until the bird has collided into one. The horizontal gap between each pipe should be constant, while the vertical gap’s location should be semi-random.
  - A floor and ceiling that the bird cannot pass through.
  - Upon collision with a pipe or the floor, the game should automatically restart.
  - A sidebar with at least two different buttons that control the speed of the timeline.

- **Part II: Thinking and Learning**
  - An entire Population of no more than 50 birds is generated at the start of the game, all of which fly and move through pipes on their own.
    - Birds should have a relatively low opacity, in order to better visualize.
  - A NeuralNetwork which is able to return an output (which is translated into an action) based on the values of the input nodes and the hidden layer — more on this later.
Bird populations are initialized randomly and utilize concepts of selection and mutation to create a stable increase in the max/average fitness over time.

The game restarts when every bird in the current generation is dead, and a new generation is created.

The sidebar should display:
- the current generation
- the number of birds alive in the current generation
- the average fitness from the previous generation
- the best fitness from the previous generation

Part III: Optimization and Writeup

A detailed write-up that explains the optimization process you went through that includes:
- The optimal configuration you reached
- A summary of design for Parts I and II
- Approach to optimizing design and training parameters
- Visualizations of metrics

Explore how different parameters affect convergence to an optimal solution, trying to improve your program as much as possible.

Be able to explain key differences in each approach, the effectiveness of these differences, and possible reasons why they worked better, worse, or similarly. This explanation should be thorough.
Part I: FlappyBird Game - DUE BY SECTION

Bird

A bird is constructed much like the doodle from DoodleJump, and can be represented using just a Circle. Apart from the properties of Circle you’re used to using, make sure to use Circle’s setOpacity method to adjust the transparency of the bird.

The bird should fall based on gravity and jump in response to keyboard input (spacebar). The new position of the bird can be calculated using the physics equation from DoodleJump:

\[ velocity_{new} = velocity_{old} + Gravity \times Time \]

Pipe Generation and Movement

When initially generating pipes, you should begin by creating a leftmost pipe, then new pipes should be generated at a specific x distance from previous pipes. Your Game class should keep track of all the pipes on the screen with a data structure.

A pipe is composed of two rectangles, the top pipe and bottom pipe, that have a semi-randomly generated gap between them. This gap should have a fixed height, but must be reachable by the bird passing through a previous pipe. Therefore, you should be able to generate a gap location that changes, but is reachable.

The pipes should scroll at a constant rate across the screen. Make sure to remove pipes both graphically and logically when they’ve scrolled off the screen. Feel free to refer back to DoodleJump, since platform scrolling was similar!

Collisions

Just like in DoodleJump, you should utilize the intersects method for collisions. At every timeframe, make sure to check if the bird intersects any part of the pipe, or if the bird hits the top or bottom of the screen. If either of these conditions are met, the bird should be removed logically and graphically from the game.

Sidebar

One important component of the sidebar is the ability to run the game in multiple speeds, so we can run the game faster for observation or debugging. You’ll also need, as usual, the quit button to exit the program.
For now, you may also want to display other statistics about the game on the sidebar like bird average fitness, all time best fitness, number of generations, etc. Feel free to be creative with this part!

**Coding Part I Incrementally**

**Clone**
1. Get the frame to show up.
2. Get a bird (can be just a circle) to display.
3. Get your bird working—think DoodleJump!
   a. Set up a Timeline and physics simulation so the bird can fall.
   b. Set up a KeyHandler to allow the bird to jump when the user presses Spacebar.
4. Set up your initial pipe. The pipes can just be rectangles with a gap in between them, and you should have a set gap height.
5. Create pipes that are equal x-distance apart from each other and fill up the frame with pipes. Each pipe should have a semi-randomly generated gap location.
6. Make the pipes scroll. No need to worry about where the bird is for this step! The bird will always stay in the same x-location and the pipes should scroll at a constant rate.
7. Implement collision. If the bird intersects either the upper pipe or the lower pipe, the game should be over both graphically and logically.
   a. Make sure to properly handle the bird hitting the ceiling or the floor as well!
8. You have successfully coded FlappyBird! Now onto the Sidebar...

**Sidebar**
1. Implement different game speeds, with buttons that activate each speed. You should only need to use one TimeHandler to do this!
2. You can add the other labels from the demo or your own personal labels if you want at this point, but they will be static until Part II is implemented.
Part II: Thinking and Learning

Congrats! You’ve coded FlappyBird. Now, onto the fun part of the project: Machine Learning! You may conceptually divide this portion of the project into two main concepts: thinking and learning. Thinking is the action of assessing given information about the agent’s environment and deciding what action to perform. Learning is the evaluation of the outcomes of these decisions in order to pass on good traits, so that we can create an agent that performs as optimally as possible.

Thinking

Neural Networks

A neural network is a set of algorithms designed to recognize patterns. It receives any amount of inputs, performs some computation using these inputs, and returns an output value. This output is used when deciding which action to perform. A standard neural network diagram is pictured below.

![Neural Network Diagram]

Each input is a number that represents a quality of the environment, while each output designates an action to be taken. The hidden layer is just an intermediate step in computation; the nodes in the hidden layer have no particular meaning. Each edge (arrow) is a weight: a number to multiply a node’s value by.

For example, imagine a reinforcement agent playing DoodleJump. Inputs could be the x-distance to the nearest 3 platforms, and outputs could be to move left or right. Given the
positions of the three nearest platforms, this neural network would output one node, and the value of the node determines if the doodle should move left or right. Typically there is an output node for each action. In this project, FlappyBird, there is only one action, and thus only one output node. Based on the information we provide, our neural network needs to think about what to do: should we jump or not jump?

**Input Nodes**

Input nodes are descriptors of an agent’s environment. More inputs allow agents to use more information when making a decision, and may eventually lead to faster convergence. Think about different factors that may affect the bird’s jumping behavior (the location of the bird, the location of the nearest pipe, etc.), and be creative! Keep in mind that when implementing Neural Networks, simplicity often leads to better results.

The bird will learn with any reasonable configuration of input nodes which adequately describe the bird’s environment. Some combinations are more effective than others, so use trial and error.

**Forward Propagation**

To calculate the output of your NeuralNetwork based on input values, you will need to create a forward propagation method. You should implement the following steps:

1. Multiply the value of each input node by the initial weights to compute the hidden layer
2. Activate the hidden layer (more on the Activation Function later)
3. Multiply the value of each node in the hidden layer with the secondary weights to compute the output layer
4. Activate the output layer
5. Return the value of the output node

Consider using static helper methods to represent any mathematical formulas. (Be sure to thoroughly test them!)

**Weights**

A Neural Network requires weights matrices to be used when computing the hidden and output layers. You can choose to contain these matrices in a wrapper class or within the neural network. Weights should be able to be copied and mutated easily for the next part, learning, so keep this in mind when designing containment.

You should have two weights matrices that transform (input → hidden) and (hidden → output). If you want, call them syn0 and syn1. (for the curious: Why?) For a brand new neural network, you should initialize weights with random values between -1 and 1. (for the curious: Why?)
**Activation Function**

In forward propagation, we multiply inputs by a set of weights and sum them to get the hidden layer, and multiply and sum again to get the output layer. That’s a lot of multiplication and summation! This will cause each successive layer to get larger and larger, and our output will be unbounded and hard to interpret. This is a problem.

An activation function “squishes” each value of a layer into a range, so we know after computing forward propagation of each layer that the result will live in the range of our activation function. Sounds complicated, but all this means is after multiplying by weights and summing, we can just apply a function to each node in the layer that “cleans” our data. There are many different activation functions (and you can even make your own), but for now we recommend using Sigmoid Activation. It’s one of the simplest and most common activation functions used in most ML tasks. This will take any value and convert it to be between 0 and 1, so that you know your result is based on this range. You can revisit your choice of activation function in part 3.

**Learning**

**Neuroevolution**

With each successive generation, we train the weights in the neural networks of the Birds with evolution, much like biological evolution. The driving factors of evolution both in nature and in ML are randomness and selection. Mutations randomly occur, and if these mutations help survival, they will be selected to be passed on to the next generation. Thus, we’ll just make a population of many birds, randomly initialize their weights, and let them play the game with random actions dictated by their random Neural Networks. The best-performing birds will pass on their weights to the next generation, with very small random changes, or mutations, and the worst performing birds will be left behind.

How do we learn what behaviors are good and what are bad? We should define “good” and “bad” behaviors with a reward function that returns the fitness of a bird, or how well it performed. In this case, the fitness simply indicates how far the bird has travelled. We can rank each bird with its fitness score and pass the weights from the candidate(s) with the highest score(s) from each generation to the next.

**Fitness Tracking**

The fitness of an agent is a measure of how well it can complete the designated task. There are many ways to do this but a simple and sufficient approach is a measure of how long a bird is alive. This can be implemented by incrementing a counter in the bird’s update method until death. Feel free to experiment later with more complex fitness schemes to see if you can accelerate learning!
Population Class

Your population class should use **selection** and **mutation** to allow birds to learn and converge towards an optimal weight configuration. Your population should be **no larger than 50**.

Selection

Selection is an evolutionary concept, in which favorable traits that improve fitness become more common over time. In this project, you will mimic natural selection by selecting the fittest bird(s) after every generation, and passing their weights on to the birds in the next generation. The number of birds selected from each generation should be calculated using a selection rate defined in the constants class. You may want to set some conditions that must be satisfied for selection to occur. Think about a case where you may not want selection to occur, and what should happen in that situation.

Sometimes, we want to create birds with completely random weights, and other times we want to initialize birds with the neural network of a parent. An elegant solution to this design constraint is constructor overloading. It’s recommended to overload the constructor of the `Bird` and the `NeuralNetwork` classes, depending on your implementation. Each definition of the constructor should change the way weights are initialized.

Mutation

Mutation is the driving factor of evolution. Without random mutations, all birds will look and act exactly the same. You should randomly change a small percentage of weights of each bird’s `NeuralNetwork` (use a mutation rate) to add diversity to each generation. You will optimize this process in Part III.

Coding Part II Incrementally

Thinking

1. Create Weights
   a. To start, initialize weights with random values between -1 and 1.
   b. You can call them `syn0` and `syn1` if you want
2. Create Neural Network
   a. Forward Propagation
      i. Mathematical functions you should implement: sigmoid function, activation function (apply the sigmoid function to the whole layer after multiplying and summing nodes and weights)
3. Add Neural Network functionality to Bird class
a. Make sure each Bird has a Neural Network.
b. Create a fitness variable that measures the bird's success to determine whether its traits should be passed down to future generations
c. Create a method for killing the bird and setting it as dead
d. SEE: calculate input node values—these can be any data point that you think will influence the bird's success (e.g. bird's height, gap height, etc.)
e. THINK: calculate the output node value using the input nodes and forward propagation (via your Neural Network)
f. DO: perform action (jump or not jump) based on the output node value

Learning
1. Create a Population of no more than 50 birds per generation
   a. Birds that collide with pipes should no longer update (fitness should stop increasing)
   b. Calculate fitness stats and update the Sidebar with your Timeline (best all time fitness, number of generations, the number of alive birds, etc.)
2. Natural selection: With each generation, keep track of a list of “elite birds” that had the highest fitness of the last population.
3. For the first generation, create a population of birds with random weights.
4. For each subsequent generation, implementation of selection is up to you
   a. One option is to add a random selection of the elite birds weighted by normalized fitness values
5. Mutation: Use a mutation rate and Math.random() to mutate the weights matrices, which can then mutate neural networks and then birds.
Part III: Optimizations

Wow! Birds jump through pipes by themselves! Cool right? Maybe you have the time to sit and wait for hundreds of generations to go by (or use your handy speed buttons), but this is computationally expensive and we usually don’t have the time or money to wait that long for agents to converge to an optimal solution. Try some of these optimization tips to get your birds to a high average fitness in as few generations as possible:

- Change the generation size (we generally want to keep this relatively small)
- Change the number of input nodes
- Change the number of hidden nodes
- Use a different activation function
- Increase/decrease the selection rate
- Populate the new generation with elite birds from the previous generation proportionally to their normalized fitness values
- Set conditions for when selection should or should not occur
- Increase/decrease mutation rate
- Add/subtract values from individual weights instead of changing them to a random value
- Change the fitness function to better describe the fitness of birds, other than just time alive
- Implement crossover (swap some weights between elite birds). This needs to be implemented carefully.
- Add more hidden layers (Not supported by visualizer)
- Anything else you can think of!! Be creative!

Note: It’s possible that your algorithm will produce “optimal” birds after very few generations. These seemingly optimal birds are a product of randomness and will not always be an indicator of a highly optimal training algorithm. If you find that your birds are learning “too fast” to visualize their results, you need to make your game harder to play (decreasing pipe gap size, increase gravity, etc.) to prove that your algorithm is capable of convergence.

Graphical User Interface (GUI) Enhancements

As you optimize your neuroevolution process, we also ask you to expand/enhance your graphical interface. You can choose any two of the following features to implement, and additional well-done ones will be considered for bells & whistles.

- Allow the user to change your game settings graphically, such as pipe distance, pipe gap height, gravity, and so on.
- **Additional feature:** support these settings changes in real-time. Make sure your program design still upholds proper encapsulation (i.e. there should probably be some intermediary Settings class).
- Allow the user to input custom weights to give to the first generation (instead of starting entirely randomly)
- Allow users to choose which optimization algorithm to use (through a menu)
- Be able to display multiple instances of Evolution, using different neuroevolution algorithms, in the same window simultaneously. (Check out the sorting demo from lecture!)
- Provide the option to play regular Flappy Bird (as a human player)
- Write your own network visualizer (see below) to best represent your neuroevolution algorithms—this should be significantly more expansive than the skeleton one we provide you. For instance, allow more than one hidden layer. Or, perhaps allow the user to edit weights/inputs and see the output result live on the visualizer.
- Use sprites for everything—birds, pipes, and also a scrolling background. (The background should be able to scroll infinitely, so think about how you can use tiling instead of an ultra-long image.)
  - **Additional feature:** add in animations for the bird sprite! Each animation should have a few frames, and the bird should have at least two different states: flapping and falling.
  - **Additional feature:** support at least 3 different themes/skins. Watch out for repetitive code!
- **Counts as two features:** in addition to Flappy Bird, also use a neural net for Doodle Jump. The platforms can scroll based on the highest Doodle.
- Anything else that you can think of! Be creative—just get it approved by an Evolution HTA (Noah or Julie).

**Write-up**

The most essential parts of Machine Learning are optimization and a thorough explanation of your process. Through intercommunication, teams of ML engineers are able to work together towards the best possible result. Thus, the ability to fully describe the different choices you made, experiments you tested, and paths you pursued is just as important as finding a good solution.

A detailed write-up will be a significant portion of the Functionality section of Part III's grade. The writeup should include the following:

1. The optimal configuration you found for your algorithm’s training parameters.
2. A summary of your design for parts 1 and 2. How did you build the clone to include functionality required for later parts of the project? How did you model the population of birds? How did you design your algorithm to make the training process feasible and efficient?
3. Your experience optimizing the design and training parameters of your algorithm. What did you change during the optimization process and what were the effects of these changes? How does the modification of each individual parameter affect the convergence of your algorithm? What are we not seeing in your handin that you tried? Be specific.

4. Visualizations of your algorithm’s convergence. You should select some metrics to measure the performance of your population over time, gather data during training, and represent them using some kind of data visualization. You should comment on patterns you see in the data and on any noticeable features. How does changing your training parameters affect these visualizations? You should include multiple visualizations corresponding to different configurations of the training parameters.

5. Feedback you have about the assignment. This is the first year we have offered this assignment, so your feedback is invaluable. Feel free to include things you liked about this project, and things that didn’t work so well that may have hindered your learning.

**Network Visualizer**

Provided as support code is a class `CS15NetworkVisualizer.java` that can be used to visualize the performance of your program. More information about the visualizer can be found [here](#). You should use this for part III to fully optimize your project. Note how much visualization of your data can help your decision making process, and improve the end result! Think about other ways you can visually represent your data.
Minimum Functionality Requirements

Your Evolution project will have to do the following things in order for you to meet minimum functionality (REMINDER: you have to achieve minimum functionality on all projects to pass the course, and since this is the final project, you cannot hand in again!)

MF requirements are not the same as the requirements for full credit on the project. You should attempt the full requirements on every project to keep pace with the course material. An ‘A’ project would meet all of the requirements enumerated in the assignment specification section of the handout and have good design and code style.

To meet minimum functionality for Evolution, you must complete Part I and Part II. That is:

- **Part I: Flappy Bird Game**
  - A **Bird** that can jump with the press of the spacebar.
  - A series of **Pipes** that continues to generate and move until the bird has collided into one. The horizontal gap between each pipe should be constant, while the vertical gap’s location should be semi-random.
  - If the bird collides with a pipe, the game should automatically restart.
  - The bird should die if it hits the “floor” or “ceiling”

- **Part II: Thinking and Learning**
  - A **Population** of up to 50 birds is generated at the start of the game, all of which are agents who make decisions.
  - A **NeuralNetwork** which is able to return an output, or action, based on the values of the input nodes and the hidden layer.
  - A new generation is created when every bird in the current generation is dead
  - A population of birds that learns over time and generations
  - A sidebar that displays:
    - the current generation
    - previous generation’s average fitness

- **Part II: Optimizations**
  - 1 additional GUI enhancement
Full Functionality Checklist

Your Evolution project will have to do the following things in order for you to meet full functionality. Remember that an ‘A’ project would meet all of the below requirements as well as good design and code style.

- Part I: Flappy Bird Game
  - A Bird that can jump with the press of the spacebar.
  - A series of Pipes that continues to generate and move until the bird has collided into one. The horizontal gap between each pipe should be constant, while the vertical gap’s location should be semi-random.
  - If the bird collides with a pipe, the game should automatically restart.
  - The bird should die if it hits the “floor” or “ceiling”

- Part II: Thinking and Learning
  - A Population of up to 50 birds is generated at the start of the game, all of which are agents who make decisions.
  - A NeuralNetwork which is able to return an output, or action, based on the values of the input nodes and the hidden layer.
  - A new generation is created when every bird in the current generation is dead
  - A population of birds that learns over time and generations
  - A sidebar that displays:
    - the current generation
    - previous generation’s average fitness

- Part II: Optimizations
  - 2 additional GUI enhancements
  - A fully fleshed-out, detailed write-up that documents your extensive optimization process

Lastly:

**Start Early... Start Today... Start Yesterday!!!**

- Prof. Flapp E. Bird