Pacing Optimization for Cycling Performance Through Neural Evolution

Physiology, Physics, Neural Evolution

Jaden Rand

Final Report

New Mexico Supercomputing Challenge 2025-2026
Santa Fe Preparatory School
Santa Fe, NM, USA
April 1, 2026

Contents

1 Introduction

1.1 Project Overview

Within cycling, the discipline called the Individual Time Trial, or ITT, has contestants complete a course one at a time, with no external support. Pacing plays a tremendous role in one's performance in an ITT, such that a weaker athlete with a superior pacing strategy, in many cases, can beat a stronger athlete.[1] This project aims to create a model that, when given a course as an input, generates an optimized pacing strategy for that course, benefiting riders of any skill or strength level.

This project utilizes models simulating rider fatigue and velocity to simulate a rider along a course, where, given inputs of power outputs[1] at times along the course, it will complete the course similarly to how a human would. With this simulation, an AI model can be trained to complete the course, generating an optimized pacing strategy.

1.2 Problem Statement

Due to how much impact pacing has on one's performance in an ITT, riders with a superior pacing plan have a considerable advantage over those with weaker ones. This especially becomes an issue because riders may hire private coaches, who generally have extensive experience creating pacing plans and can therefore give those riders an advantage,[1] creating an environment in which athletes with greater monetary resources can outperform others, creating a less equal playing field. This model aims to balance this environment by allowing riders to have optimized pacing strategies regardless of their financial situations.

1.3 Existing Solutions

Currently, other than private coaching, very few methods exist to find a pacing strategy for a given course. Some online options exist, the most notable of which being BestBikeSplit[2], though this option may produce oversimplified strategies and often relies on information about the rider, equipment, weather, wind, and many other variables that may not always be available, and while they produce more specific pacing plans, are not necessary and sometimes lead to inefficiency. The program also does not use an AI-based approach like this project does, meaning it may have less ability to adapt to corner cases and complicated situations. The AI approach of this project also allows for further possibilities, especially in terms of real-time adjustment, which is much less achievable with the physics-based approach used by BestBikeSplit. Finally, the program only offers three free plans before the user must sign up for a paid membership. While this lowers the financial barrier of an ITT as compared to private coaching, it does not remove it. This project, by creating a model that

[1] Elaborated in 1.4

outputs more optimized pacing strategies while requiring less information that the rider may not have available and by offering it freely, could greatly lower the financial barrier to entry of ITTs, removing the advantage some riders hold.

1.4 Power Measurement for Cycling

Due to the effects of wind, road gradation, equipment, and other factors in the speed at which a rider travels, the most accurate way to quantify cycling effort, and therefore the best metric to use for a pacing plan, is power, measured in watts. Riders can measure power via a power meter, a device within the rider's pedals, shoes, or crankset, which measures how many watts the rider outputs and transmits it to be displayed on the rider's smartphone or dedicated cycling GPS device.[4] This project, in order to be applicable to the real world, uses power for the generated pacing plans. Although this does mean that a cyclist would be required to purchase a power meter to benefit from this, almost any cyclist serious enough to be racing an ITT already owns one, and the cost of one of these devices[9] is equivalent to roughly one month of paying for a private coach[3] or about a year of membership to BestBikeSplit[8], and the power meter can last almost indefinitely, meaning it still considerably reduces the cost of creating and using a pacing plan.

2 Methodology

2.1 Conceptual Approach

The goal of this project was to be able to create an optimized pacing plan for any given course. At first, the problem was approached by attempting to perform arithmetic on the slope of the course at each point, trying to create a strategy that pushed harder on uphills and went easier to recover on descents. This turned out to be oversimplified, as it did not take into account how riders fatigue or where on the course a rider was. The next approach, and the one used in the project, was to train a neural network to complete the course. This is possible through two methods, gradient descent and neural evolution. In gradient descent, a large data set consisting of inputs and the correct outputs for those inputs is used to create a model that can predict the correct output for a slightly different set of inputs. This approach was ruled out due to the lack of large sets of publicly available optimized pacing strategies. This left neural evolution, in which many neural networks[2] complete a task and are assigned a fitness score based on how well the task is performed. The worst-performing networks are then removed, and the population is refilled with slightly modified copies of the better-performing networks. This process then repeats many times, resulting in a network that performs the task near-optimally.[7]

[2]Elaborated in 3.5

With neural evolution having been decided as the method to train the AI, a simulation of a rider (AI Agent) on the course was necessary to be able to give genomes a fitness score.[3] This need for a simulation meant that some sort of model for physics was required, which, for simplicity's sake, only needed to model rider velocity. Furthermore, to avoid allowing agents to output high power for longer than humanly possible, a model for rider fatigue was necessary. Lastly, the program needed to be able to intake a course. Courses of this nature are usually represented in a GPX (GPS Exchange Format) file, so a program to standardize these files to be inputted to the networks was necessary. Ultimately, this meant the three components needed for the simulation were models for velocity and fatigue and a course processing program.

2.2 Considerations and Simplifications

The complexity of the problem required a thoughtful approach to which aspects of the simulation needed to be realistic and which could be omitted for simplicity's sake without altering results. Within the velocity model, air resistance had to be considered, as otherwise, high speeds would be unrealistically easy to maintain, which could skew results. Other detriments, however, namely equipment inefficiencies and rolling resistance, could be ignored, as they affect high and low speeds equally. Other factors could be assumed to be constant, such as rider and equipment weight and surface area. The fatigue model, due to the complexity and non-regularity of human physiology, does not fully represent how humans fatigue. The model uses a rider's distance from steady-state power to represent how energy stores are used, which are then used to show fatigue levels. This does not take into account how a rider's respiratory and cardiovascular systems may be affected, as many variables impact these, leading to a model being much more complex, which gives only marginally more insight into the fatigue level of a rider.[4] Rider steady-state power and maximum energy stores, two variables used in the fatigue model, are kept constant, as the final pacing plan can simply be scaled to fit a specific rider's values for these factors. The processing of courses slightly simplifies them, though solely due to how GPX files store a course. This means that very small changes in slope may not be shown, nor will factors such as weather, road condition, or sharp turns. Overall, none of these simplifications should have drastically altered the results of the project, and all could be accounted for using a more complex model that takes more input from users.[5]

2.3 System Overview

- A population of random neural networks is created
- Each network is simulated on the course via the following process

[3] Elaborated in 3.5
[4] Possible solution proposed in 5.1 and 6.2
[5] More information in 5.1 and 6.2

  - The network takes inputs of its location on the course, the slope of the road, its fatigue level, and the average remaining slope and gives an output of power output at that time.
  - Velocity is calculated based on the network's power output and slope, and its position is updated accordingly.
  - The network's energy stores are adjusted based on its power output
  - This process repeats until the network finishes the course or a cutoff time is reached.

- The networks are given a fitness score based on how far into the course they reached and how long it took them.
- The worst networks are removed from the population, which is restored with slightly mutated copies of the better networks.
- The networks are simulated on the course again. This process repeats an inputted number of times, usually 500 to 3000, each corresponding to a generation.
- Once all generations finish, the best network is saved, where it can then be analyzed.

3 Implementation

3.1 Course Processing

For the neural networks to gain information from the course, the GPX files needed to be processed and expressed in a form more easily readable by a neural network. The structure of a GPX file includes many points, each with a latitude, longitude, and elevation value. These points are processed by iterating through them, calculating the distance and slope between each pair, which are then stored in a list, essentially giving a list of segments, each defined by their length and slope.

```
# Helper - get slope between two points
def get_slope(lat1, long1, alt1, lat2, long2, alt2):
    point1 = (lat1, long1)
    point2 = (lat2, long2)
    distance = geodesic(point1, point2).meters
    d_alt = (alt1-alt2)
    if distance < 0.5:
        return(0)
    else:
        return(round((d_alt/distance)*100, 2))

# Helper - get distance in meters between two points
```

```
13 def get_distance(lat1, long1, lat2, long2):
14     point1 = (lat1, long1)
15     point2 = (lat2, long2)
16     distance = geodesic(point1, point2).meters
17     return distance
18 
19 # Create course segments from GPX file
20 def make_course_segments(gpx_file_path):
21     gpx_file = open(gpx_file_path, 'r')
22     gpx = gpxpy.parse(gpx_file) # Give GPX file to gpxpy library
23 
24     course_segments = []
25 
26     # Iterate through every point and save segments as distance
       ↪  and slope tuples
27     for track in gpx.tracks:
28         for segment in track.segments:
29             for i, point in enumerate(segment.points):
30                 if i == 0:
31                     prev_point = point
32                     continue
33                 distance = get_distance(prev_point.latitude,
                   ↪  prev_point.longitude, point.latitude,
                   ↪  point.longitude)
34                 slope = get_slope(prev_point.latitude,
                   ↪  prev_point.longitude, prev_point.elevation,
35                                   point.latitude,
                                     ↪  point.longitude,
                                     ↪  point.elevation)
36                 course_segments.append((distance, slope))
37                 prev_point = point
38 
39     return course_segments
```

3.2 Velocity

The core of the simulation is the velocity of the AI agent, specifically how it changes at different power outputs and slopes. The majority of how velocity is calculated for this project is based on an article written by Steve Gribble of The Computational Cyclist.[6] This article shows that the power needed to maintain a certain velocity can be expressed as a cubic function of velocity, with coefficients determined by factors such as slope, rider weight, and rolling resistance, all of which are constant, except for slope. This means that for a given slope and power, by finding the zeroes of the cubic function, the velocity at that slope and power can be determined.

Initially, the implementation of solving the cubic function was approached via the use of Cardano's formula, a method to find the real zeroes of a cubic function. While this worked for most cases, in situations where the slope was considerably downhill (lower than about -5 percent), the cubic function would have three real roots, and Cardano's formula would be unable to solve it. The problem was then approached via the use of a Python library, NumPy, which has a function that takes inputs of a polynomial's coefficients and outputs its roots. These roots could then be filtered to use the largest real root, which corresponds to velocity. While this method worked correctly, once neural evolution was simulated, it became clear that this method of finding roots, which, due to its ability to find imaginary roots, was quite slow, would not be fast enough to complete training in a reasonable amount of time. Finally, Cardano's method was revisited, with another method introduced to handle cases where the cubic function has three roots. This ultimately led to a function which uses the standard Cardano's method when the cubic has only one root and a trigonometric solver when it has three.

```
# Helper - cube root function that handles negative numbers
↪  correctly
def cbrt(x):
    return math.copysign(abs(x)**(1/3), x)

# Constants for velocity calculations
W = 73
C_dA = 0.3
Rho = 1.225

def get_velocity(P, G):

# Coefficients for cubic func.
    a = 0.5 * C_dA * Rho
    b = 0.0
    c = 9.8067 * W * math.sin(math.atan(G/100))
    d = -P

    # Coefficients for Cardano's
    p = (3*a*c - b*b) / (3*a*a)
    q = (2*b**3 - 9*a*b*c + 27*a*a*d) / (27*a*a*a)

    # Discriminant - determines how many roots the cubic has
    delta = (q*q)/4 + (p*p*p)/27

    if delta >= 0:
        # one real root
        S = cbrt(-q/2 + math.sqrt(delta))
        T = cbrt(-q/2 - math.sqrt(delta))
        x = S + T - b/(3*a)
```

```
30          return x
31      else:
32          # three real roots - use trig method
33          r = math.sqrt(-(p**3)/27)
34          phi = math.acos(-q/(2*r))
35          r = (-p/3)**0.5
36
37          # Find the three roots and return largest (all others are
            ↪  negative)
38          x1 = 2*r*math.cos(phi/3) - b/(3*a)
39          x2 = 2*r*math.cos((phi + 2*math.pi)/3) - b/(3*a)
40          x3 = 2*r*math.cos((phi + 4*math.pi)/3) - b/(3*a)
41
42          return max(x1, x2, x3)
```

3.3 Fatigue

Though it was obvious early in the project that fatigue needed to be simulated to accurately simulate a rider on a course, it was unclear how to actually represent fatigue. Many methods of quantifying fatigue were researched, including those based on heart rate, power, and a mixture of the two. For this project, a model based on power ended up being the most practical, as heart rate can be impacted by many factors, such as heat, elevation, and stress, whereas power is a much more accurate representation of how hard a rider is pushing. This led to a model using critical power (a rider's steady-state power output in watts) and W' balance (a rider's energy stores, measured in joules), in which W' balance represents how fatigued a rider is, and the difference between their outputted power and critical power dictates their W' balance expenditure and recovery. A resource that ended up being tremendously useful in this section of the project was a blog post written by Aert Goossens detailing different ways to model these relationships and how they differ.[5] This blog post outlines three different methods to calculate W' balance and compares their results. The first method discussed, and the original method developed, is the Integral Algorithm, in which an integral function of the rider's distance from critical power must be computed at each timestep. The second method, the Waterworth Optimized Algorithm, is very similar to the Integral Algorithm, except instead of computing an integral, a running sum is stored, and the W' balance is found by an arithmetic function of that sum. This method was found in the blog post to give almost identical results to the original algorithm and run considerably faster when implemented in code. The final method shown is the Differential Algorithm, which expresses W' balance as a differential function of the rider's distance from critical power. This method, while faster than the original, was found to be considerably slower than the Waterworth Optimized Algorithm and gave vastly different results than the other two methods. While little research exists on the accuracy of this method, the Integral Algorithm and Waterworth Optimized Algorithm are the industry standard and have considerable amounts

of research verifying their accuracy. This project ended up using the Waterworth Optimized Algorithm to optimize both realism and efficiency. This model uses the following functions, which were implemented in Python with relative ease.

$$W'_{bal}(t) = S(t) \times e^{t/\tau_{W'}} \tag{1}$$

$$S(t) = \sum_{0}^{t} W'_{exp}(t) \times e^{t/\tau_{W'}} \tag{2}$$

$$\tau_{W'} = 546 \times e^{-0.01 \times D_{CP}} + 316 \tag{3}$$

Where W'bal(t) is the W' balance at timestep t, S(t) is the running sum kept, W'exp is power minus critical power when positive, and zero otherwise, and Dcp is the average power when below critical power

```python
# Constants for fatigue model
cp = 250.0
Wprime = 20000.0
t = 0.0
S = 0.0
Wbal = Wprime

rec_time = 0.0
rec_total_power = 0.0

def get_Wbal(P, t, S, rec_time, rec_total_power):  # Can't use global variables with multiprocessing
    if P >= cp:
        Wexp = (P-cp)
    else:
        rec_total_power += P

        Wexp = 0
    rec_time += 1
    Dcp = cp - (rec_total_power / rec_time)
    tau = 546 * math.exp(-0.01 * Dcp) + 316

    S += Wexp * math.exp(t / tau) # Running sum is kept - used to calculate Wbal at each time step
    Wbal = Wprime - 1/2 * S * math.exp(-t / tau)

    return Wbal, S, rec_time, rec_total_power # Variables passed back to genome to be stored
```

3.4 Rider Simulation

With programs completed to process courses and calculate W' balance and velocity, assembling these pieces together to create a simulation of a rider on

a course went very quickly. In the simulation, a course is processed, and the segments are stored. The program then iterates through a number of steps until either the course is completed or a cutoff time is reached, 2500 seconds for most of the training. Through each of these iterations, the rider's W' balance, current segment, position within that segment, and total distance traveled are stored. Within an iteration, first, a power value is found for the current timestep, either via manual input or later, from the neural network.[6] This power output is capped so as not to exceed what would be possible by a human with the same critical power and W' balance. Next, the velocity of the rider is found based on the slope of the current segment and the power output. This velocity is used to advance the rider's position within the current segment. If the end of the segment is reached, the rider moves on to the next one in the list. The rider's W' balance is updated as well, according to the power output. Finally, the total distance traveled and power output are saved into a list to be graphed later on.

3.5 Neural Evolution

To implement neural evolution in this project, a strategy called NEAT, or Neural Evolution of Augmenting Topologies, was utilized. In this process, the population is stored as a list of genomes, each corresponding to a neural network. These networks are made up of a number of nodes, which intake a set of numbers and perform a function on them, and connections, which pass the output of a node into another's inputs. Each generation, every genome in the population performs a task and is given a fitness score based on its execution of that task. The worst genomes in the population are then removed, and their places are filled in with slightly mutated copies of better-performing genomes, in which the function performed by a node may be slightly different. The advantage of NEAT over other neural evolution strategies is that occasionally the structure of the network is altered in a mutation, either adding or removing nodes or connections. This leads to the possibility of many different strategies of various complexity emerging, perfect for a project such as this one, which may have a very complex solution.

To implement NEAT in this project, the Python library NEATpy was utilized. This library makes storing and mutating genomes considerably easier, but a simulation and a fitness function were still necessary. Once the simulation of a rider on the course was completed, the implementation of NEAT using this library was relatively fast. Within the simulation, where a power output is found, the neural network is activated, with inputs of the distance remaining, the current W' balance, the current grade, and the average remaining slope. This gives an output between zero and one, which is then multiplied by 3 times the critical power. This ultimately means that the network gives an output between zero and, in the simulations used, 750 watts, capped such that W' balance cannot go below zero. The most difficult part of NEAT implementation was

[6] Elaborated in 3.5

determining a fitness function that gave optimal results. Originally, the fitness was determined by the total distance traveled divided by the time spent. This gave unideal results, so a number of "punishments" were implemented to try and steer evolution towards more optimal results. Some examples included slightly decreasing a genome's fitness when it changed power too quickly to try and promote smoother pacing and subtracting a genome's leftover W' balance to try and force genomes to push harder throughout the course. These punishments, ultimately, overcomplicated the model and made performance worse, so along with some optimizations in the speed of the model, allowing longer training periods, the fitness function was changed to the distance traveled minus the time spent, and optimal results were finally achieved. One of the most significant of those speed optimizations was the removal of NumPy-based velocity calculation and its replacement with a numerical solver[7], as velocity is calculated every timestep, for every genome, over every generation, meaning it must be calculated nearly one billion times over one run of the program. This led to about a 50 percent increase in efficiency of the program. Even more impactful on the speed of training, however, was the implementation of multiprocessing. Originally, each genome would be evaluated one at a time, which was incredibly inefficient. Multiprocessing allows different processes to run on different CPUs (Central Processing Units) of a computer in parallel, meaning multiple genomes in the same generation could be evaluated at once. This was implemented via the use of a Python library, Multiprocessing, which allows functions to be assigned to CPUs or evenly spread across them. For this project, implementation included assigning the function that simulated a genome on the course to be evenly spread across CPUs each generation, as well as defining a function to evaluate all genomes within a generation, which then called a function to individually evaluate genomes using multiprocessing. This ended up accelerating training by about 40 times, making it tremendously useful in the efficiency of the project, bringing training times down from nearly five hours to just over ten minutes. Due to these increases in speed, it became practical to train models on multiple courses, where each genome would be evaluated on each course, and the fitness scores from each course would be averaged to give the genome an overall fitness score.

3.6 Results Storage

Two main strategies were used to track the results and progress of the genomes through evolution. The first of which was to save the best-performing genome at the end of evolution, where it could then be analyzed. This analysis occurred via a separate program, which ran the genome through the course simulation, storing the power output and distance traveled at each timestep. This graph was then overlaid on an elevation profile of the course, allowing its actions in different sections of the course to be analyzed.

[7] Elaborated in 3.2

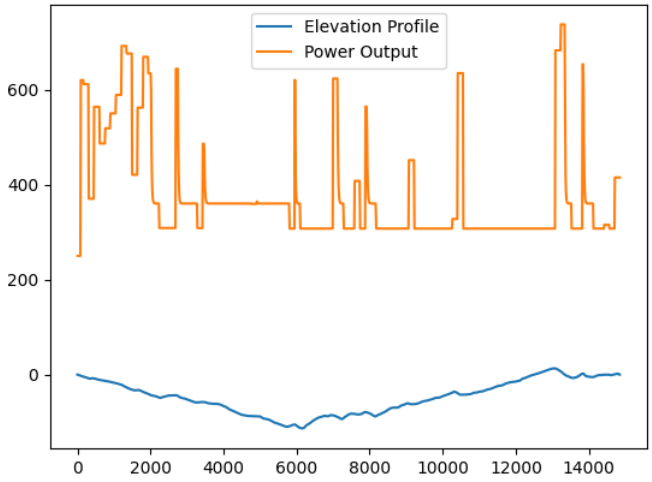


Figure 1: Graph of power output vs elevation profile of a winning genome.

The other strategy used was to create a similar graph, but during the training. At the beginning of training, one of the training courses is chosen, and for each generation, every genome's power to distance graph on that course is saved in a list. Every ten generations, all the graphs from that generation are graphed on a single plane, allowing the overall performance of the population to be seen. While this approach works reasonably well, a significant number of genomes, due to mutations, end up maintaining one specific power value until their W' balance no longer allows them to, after which they ride at or near critical power. The graphs of these genomes end up adding a large block to the graph of the generation, making it difficult to see the genomes that are truly pacing. To avoid this, these genomes are filtered out prior to graphing.

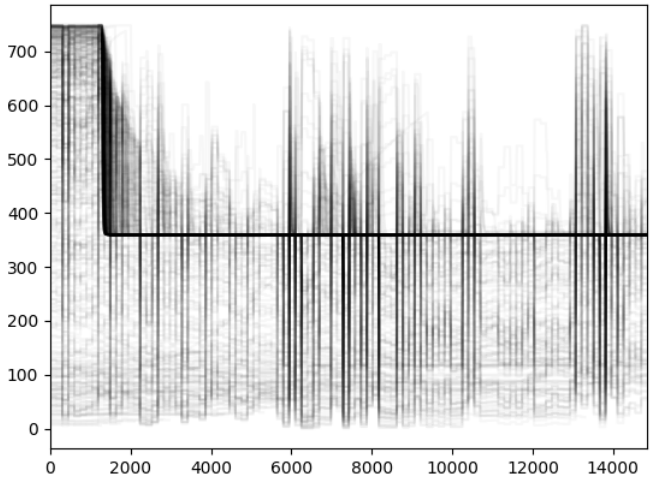


Figure 2: Graph of power outputs of filtered genomes in a generation.

4 Result Analysis

The first valid training session run contained only one course, a roughly 10-mile loop with 600 feet of elevation gain, the same course that is used for all the graphs shown, so as to avoid performance being affected by the course. The 2000-member population trained for 3000 generations, finishing in about 3 hours, with the following winning genome's graph.

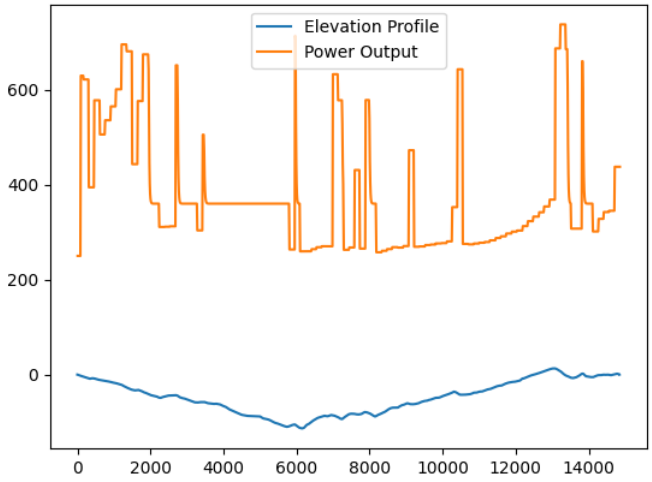


Figure 3: Graph of initial winning genome's power output over a course

While this graph looks somewhat like random noise, it shows clear pacing when analyzed. The genome starts off hard, outputting relatively high power for the first half of the descent. It then recovers some before spiking up a hill, recovering more, and pushing up another small hill. The genome then slowly recovers, riding near steady state for the second half of the descent. Next, through the slightly uphill yet somewhat rolling section along the bottom, it pushes up steeper sections and recovers on flats and descents, before riding relatively steadily up a long stretch of hill, only spiking once at a steep section and once at the top before a short descent. The genome finally steadily builds through the final prolonged climb, pushing hard at the top before a short recovery, pushing up a hill, and building into the finish. This follows many of the thought processes usually used in pacing, especially those of pushing harder up hills to recover on descents and starting and ending hard, with the middle more steady.[1] The genome also continued to perform very well. Another 500 generations were run, in which rather than starting from random genomes, the simulation began with the whole population made up of copies of this winning genome, and the chances and severity of mutations were decreased, creating an environment that prioritized optimization near this initial genome. At the end of this training period, the exact same genome won, showing that it truly was near-optimal on this course.

Though this initial training seemed to produce a near-optimal genome, another training period was run, with a smaller population of 1000 and the same generation number of 3000, to verify that a similar strategy would emerge. The

winning genome of this run ultimately performed slightly worse than that of the previous, though all the spikes were at the exact same places and very similar wattages. The only difference between the two pacing strategies was that the one created in the second training session lacked some of the nuance of the first, especially in terms of building over long periods of time, likely due to the smaller population allowing less room for experimentation. This genome ultimately performed only slightly worse than the initial genome, completing the course in 1198 seconds, as opposed to 1190.

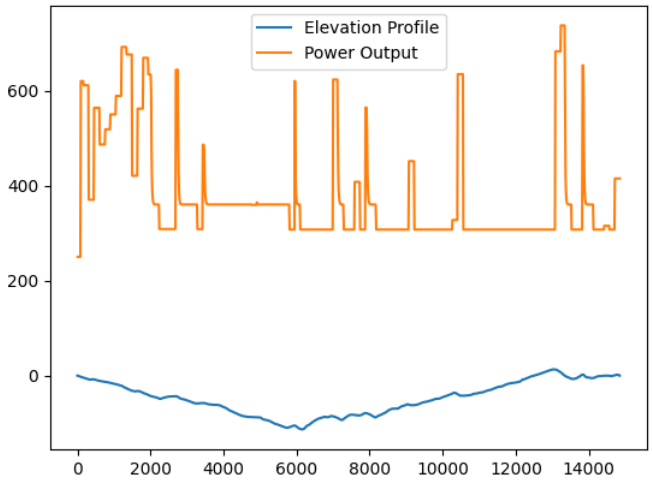


Figure 4: Graph of later trial's winning genome's power output over a course

Finally, a training session was run using five different courses, with lengths varying from around 4 miles to around 12 and profiles ranging from essentially flat to hilly to steep climbs and descents, in which each genome would be simulated on every course, and their individual fitness scores from each course would be averaged to achieve an overall fitness score. Due to time limitations, this session only had 100 members in each generation, with just 500 generations. Nevertheless, when the winning genome was graphed on one of the courses, the same one as shown above, it reached a relatively similar pacing plan as the other two sessions, with a very similar overall structure and spikes in essentially the same places.

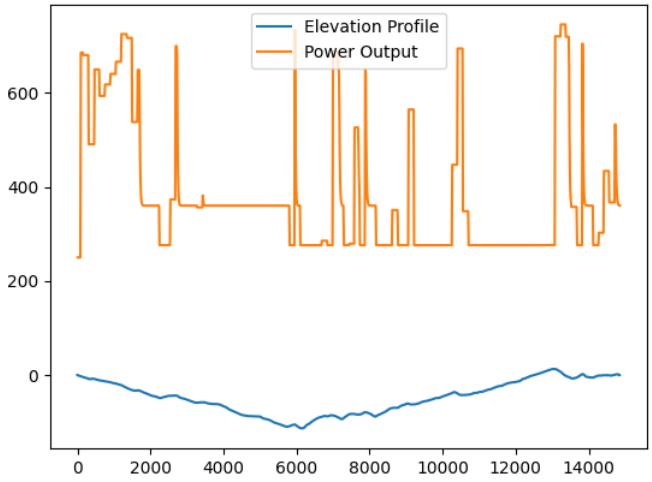
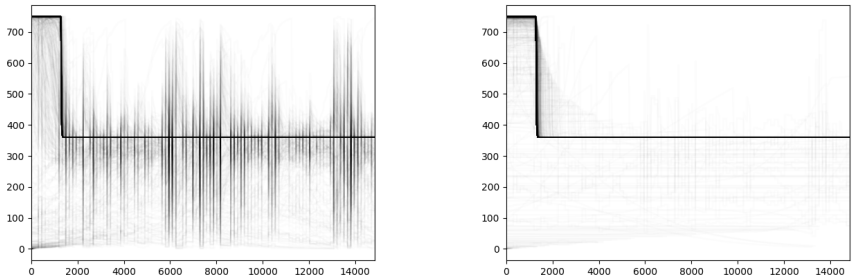


Figure 5: Graph of winning genome from training with 5 courses' power output over a course

Surprisingly, this genome also performed very similarly to the other two, finishing the course in 1196 seconds. Even with just 10 percent of the training time and focus spread across five courses, this model managed to beat the one created in the second training session, which was essentially purpose-built for this course. This is likely due to the final genome created needing to learn how to pace on any course, leading to it creating a truer pacing plan, rather than falling into a local optimum, which may have happened for the single-course models. With more training time, this model likely would have been able to match, and even surpass, the first genome's performance.

While the graphs of winning genomes showed a clear strategy, which remained consistent across different sessions, the graphs of all genomes within each generation did not show nearly as much correlation. It was originally expected that once one genome "found" a near-optimal solution, as evolution went on, the strategy would spread through the population, slightly differing, and the whole population would slowly approach the optimal solution. Instead, the graphs are essentially random, with only slightly clearer spikes near prominent course features.



(a) Generation 1

(b) Generation 10

(c) Generation 100

(d) Generation 500

Figure 6: Graphs of filtered genomes in four different generations' power outputs. From a training session with only one course

This seeming randomness is likely due to the randomness of mutations within NEAT. Because the networks that pace correctly are so complicated, with many rounds of multiplication occurring within them, most mutations to them will drastically change their behavior. This causes the patterns of similar pacing, just vertically shifted, that appear on the graph. When a genome mutates such that a variable is scaled by a different factor, it causes it to behave similarly, just at a higher or lower power. Many genomes also mutate such that they produce a very high power at the beginning, then deplete their W' balance and ride at steady state, causing the very visible step-downs and horizontal lines that appear on the graphs. The genomes that employ this strategy, as it is inefficient and clutters the graphs, are sorted out using a function that removes any pacing strategies with no considerable increase in power at any time. Some networks, however, initially follow this pattern, then act unpredictably at some point later in the course, and therefore still appear on the graph.

While the fact that the genomes do not converge towards the optimal pacing plan may seem unideal, it may actually lead to stronger results. The fact that

the genomes in a generation produce relatively dissimilar pacing plans means that they are much more likely to find the true optimum. If the population quickly settled near the best genome, it could lead to situations in which one genome finds a solution that outperforms the rest of the population but is far from the true optimum. If the rest of the population then only utilized strategies similar to this one for the rest of the generations, the better solution could never be found.

5 Possible Real-World Application

Currently, the models produced are applicable to very few people, due to the specificity of variables used in training, such as the critical power, W' balance, and weight of the rider. With little modification, however, a model could be trained such that these variables change, and it must adapt and be able to generate pacing plans for any values. A model of that type would then have many applications in the real world. By creating an application for a smartphone or cycling GPS device, riders could have access to generated pacing plans while on the course. These pacing plans could even be customized to specifically fit a rider's needs, as once a model that can handle any values for rider weight, critical power, etc. exists, further training could occur, starting with that model, on the rider's specific course, with their values for these variables fixed. With an application for an on-bike device, personalization and adjustment could even occur while the rider is on the course. By creating a mapping between power and heart rate, the rider's heart rate at any given time during the pacing plan could be estimated. This could then be compared to the rider's actual heart rate, if they have a device to measure it, and the remainder of the pacing plan could be scaled accordingly, ensuring that a pacing plan would always closely match the rider's ability level.

With even more training time and available data, it could even be possible to create a model that can handle situations other than an ITT. For example, with inputs about the size of a group one is riding in and how other riders affect aerodynamics, a model could be built that gives pacing strategies for group road racing, adjusting in real time based on the actions of other riders. Another possible application with more data is usage in mountain bike racing. Generally, these races are impacted much less by group aerodynamics, so with data about how rough terrain and turns impact power and speed, pacing plans could be generated for off-road racing.

6 Conclusion

6.1 Progress and Reflections

While there still are possibilities and ways to take this project further, some of which are detailed in the previous section as well as the Project Proposal

and Interim Report, the fundamental aspects were all implemented, and little more was left to discover or learn in the uncompleted sections of the project. Specifically, the understanding gained of velocity and fatigue showed real-world applications of topics such as physics and calculus, and building these models gave much more insight into the inner workings of the simulation used later in the project. Furthermore, the process of implementing NEAT gave tremendous insight into how AI models function, and the need to build a simulation of a rider gave experience bringing smaller pieces of code and modeling together to create a more finalized program. The results from the completed sections of the project also have almost all of the insights that the full project, as detailed in the Proposal would have. The structure and possibility of an optimized pacing plan for any course were the primary focuses of the project, and while further application and implementation could have given insight into how applicable these models are to the real world, the simulation constructed gives a more regulated testing environment, removing any possible effects from random chance.

6.2 Next Steps

Likely the most useful possible next step would be to run a much longer training session on many courses of various lengths, difficulties, and profiles, with varying rider weights, critical powers, and other variables, hopefully resulting in a model that near-optimally handles any situation. This model could then be tested in the real world against self-pacing and other strategies, showing whether it can outperform human-made pacing plans. Lastly, an application could be built as detailed in section 5, to allow riders to create and utilize pacing plans.

6.3 Learning Highlights

Though almost every aspect of this project resulted in significant learning, certain parts especially stood out. Building a model for W' balance showed how physiology and technology can intersect, with the ability to model one's fatigue initially seeming impossible. The implementation of NEAT also stood out, as it showed just how unpredictable neural networks and evolution can be, yet still find extremely complicated solutions to problems. The concept of NEAT as a whole continued to show how nature and technology can combine, simulating evolution to create an AI model. Finally, introducing multiprocessing to accelerate training showed just how powerful computers can be when used efficiently, accelerating a program from taking nearly 5 hours to run to just over 10 minutes, an increase in speed that would have seemed impossible early in training.

6.4 Generative AI Use

In the interest of accelerating progress and utilizing available tools, the generative artificial intelligence ChatGPT was used occasionally to streamline progress. These cases generally consisted of its use to accelerate debugging code and to

research libraries used within the code. Examples include determining why the velocity model would return zero for any slope below -8 degrees and finding information about how to populate a pre-made genome into NEATpy, as the official documentation did not include any. Generative AI also aided in the creation of this report, giving recommendations regarding if any important information was left out. None of this document was written by generative AI.

7 Works Cited

References

[1] Allison, Zack. "The Art and Science behind Time Trial Pacing." TrainingPeaks. Accessed April 1, 2026. https://www.trainingpeaks.com/blog/the-art-and-science-behind-time-trial-pacing/.

[2] Best Bike Split. Last modified 2026. Accessed March 31, 2026. https://www.bestbikesplit.com/home.

[3] "Coach Match Packages." TrainingPeaks. Last modified 2026. Accessed March 31, 2026. https://trainingpeaks.com/coach-match/.

[4] Friel, Joe. "What Is Power for Cycling?" TrainingPeaks. Last modified December 25, 2020. Accessed April 1, 2026. https://velo.outsideonline.com/road/road-training/what-is-power-for-cycling/?scope=initial.

[5] Goossens, Aert. "Comparison of W' Balance Algorithms." GSSNS. Last modified 2025. Accessed April 1, 2026. https://gssns.io/posts/w-prime-balance-algorithms/.

[6] Gribble, Steve. "Cycling Power and Speed." The Computational Cyclist. Accessed April 1, 2026. https://www.gribble.org/cycling/power_v_speed.html.

[7] Hulstaert, Lars. "Gradient Descent vs Neuroevolution." Medium. Last modified December 20, 2017. Accessed April 1, 2026. https://medium.com/data-science/gradient-descent-vs-neuroevolution-f907dace010f.

[8] "Premium Subscriptions." Best Bike Split. Last modified 2026. Accessed March 31, 2026. https://www.bestbikesplit.com/member-premium.

[9] "Rival AXS Power Meter Upgrade." SRAM. Last modified 2026. Accessed March 31, 2026. https://www.sram.com/en/sram/models/pm-riv-assy-e1.