Part 1: Particle Swarm Optimisation

Results/Formulas	***f*₁(x), *D* = 20**	***f*₂(x), *D* = 20**	***f*₂(x), *D* = 50**
Mean	30.562	0.03279	0.055005
Standard deviation	31.001	0.01931	0.020685

Rosenbrock Results Discussion

The mean result of my PSO applied on Rosenbrock's function (f_1) is 31. This is notably higher than the minimum value that my PSO can achieve, as indicated by the large standard deviation. I reported the results of each run (can be seen in script.txt) to more closely observe the performance of my EC. 22 out of the 30 runs had results below 20, so the other results with values in the 70s and 80s would have affected the mean. A larger outlier also exists with a result of 130. This also explains the standard deviation value of 31, representing the large spread of results. After my PSO ran and I collected the data, I calculated the median of the 30 runs. This metric is better for representing the performance of my PSO as it is not as prone to outliers, which are present in my results. The median of my results from Rosenbrock's function is 15.

These larger results are likely because of the PSO getting stuck in a local minima that it couldn't get out of. This is an example of the stochastic nature of PSO and how each run performs slightly differently. In this case, each run has the potential to get stuck in a local minima that is considerably far off the much lower, global minima. Despite some runs getting much better results, they still did not converge directly to the exact global minimum. If I were to improve my PSO implementation to better exploit minimas, I would implement a dynamic weight inertia value. This would make my PSO gradually change from focusing on exploring the search space, to exploiting the region that it is currently in. I would expect to see faster and better convergence on the global minimum using this method.

Griewank Results Discussion

The mean result of Griewank's function (f_2) with 20 variables (D = 20) is 0.033 which is quite close to the global minimum of 0. The standard deviation is very small, indicating a low spread in results. Observing the individual results from each run (available in script.txt) doesn't reveal any outliers or more interesting patterns. Each result is fairly close to the mean, with the lowest result being 0.007 and the highest being 0.06. While testing, I was observing the results of each generation as the PSO was running. I noticed that it quickly reached a value close to the minimum, but took considerably longer to exploit the minima and converge.

The mean result of the same Griewank function with more variables (D = 50) is slightly higher than the previous run with fewer variables. I expect that the mean has a slightly worse result when D = 50 because it adds more dimensions to the equation and increases the search space. The PSO essentially has to do more work to reach the same mean result of the previous run because it needs to optimise many more values. However, the standard deviations of both runs are very similar, with the function where D = 50 only being ever so slightly higher than the other run. For the same reasoning of a higher mean on the function with more variables, I expected the disparity of the standard deviations to be greater. However, I was surprised to see the outcome of the result spread be so similar.

Similar to Rosenbrock's function, my PSO would likely converge faster on the global minimum of the Griewank function by having the inertia weight linearly change during evolution. Better exploitation of the global minima region would help it converge faster and provide a better result.

Part 2: Non-dominated Sorting Genetic Algorithm-II

Library Choice

I chose to use the Pymoo library for my solution. This was because it was popular for multi-objective optimisation, had extensive documentation and support, and a handful of guides for different applications. I also chose it because of its capability to calculate the hyper-volume of a solution set which I required. I familiarised myself with the library using a guide on finding a solution set in a multi-objective search space.[1] I then read another guide to learn how to use a unique individual representation within the library.[2] Fortunately, these guides were very useful and were all I needed to use to build my solution.

Individual Representation

I chose to use bit strings to represent individuals for the same reasons as in the feature selection part of the first project. They're most appropriate for this use case as they are very simple in communicating which features are selected. The ratio of selected features is also easy to calculate, along with sampling, crossover, and mutation being easy to perform. Creating the functionality for these tasks was straightforward with the bit string individual representation and I didn't run into any problems.

Fitness Function

The fitness function included assessing the error rate, and ratio of selected features to total features. The error rate for an individual was found by subtracting the classification accuracy from 1. The selected feature ratio was simply the quotient of the number of selected features and the number of total features available. Classification accuracy was found with my wrapper-based approach of using a KNN classifier from scikit-learn with the default number of neighbours as 5. Testing my NSGA-II program showed this classifier to be appropriate for determining classification accuracies for the two datasets. The error rate and selected feature ratio are given to a Python dictionary that Pymoo manages.

Crossover and Mutation

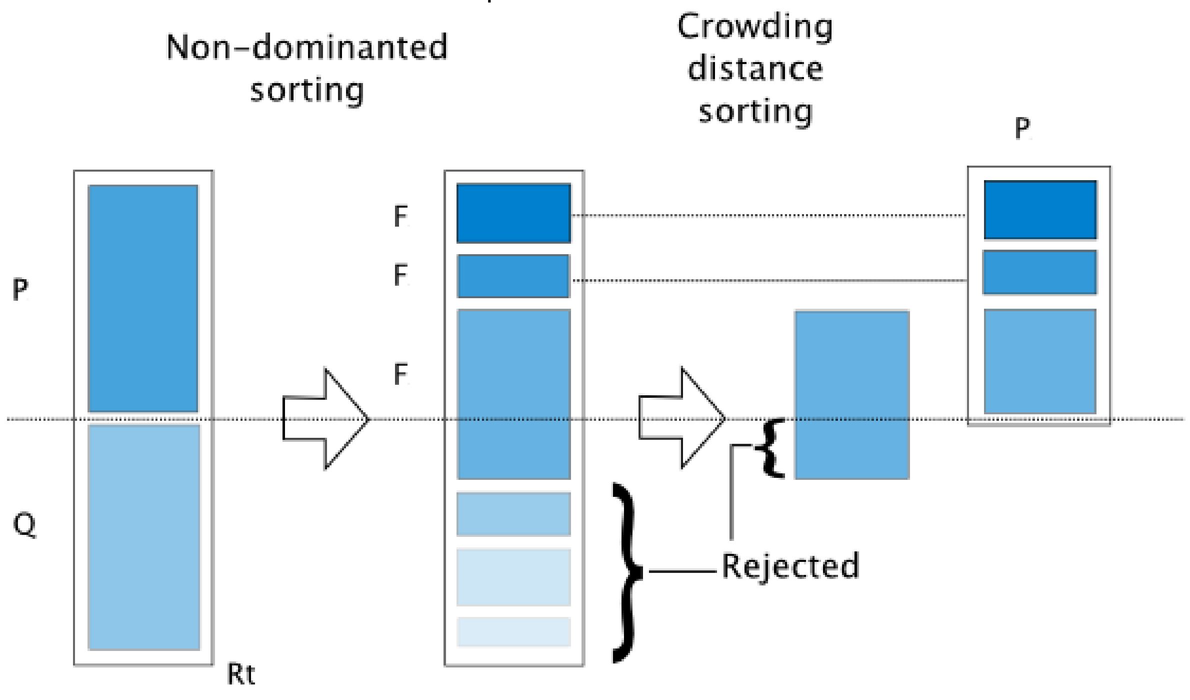
Individuals are initially created by randomly choosing whether or not each of the features are selected. This is done by creating a bit string with a length the same as the number of features and random '0' or '1' values for each character. Crossover is done using the 1-point

[1] https://pymoo.org/getting_started/part_2.html
[2] https://pymoo.org/customization/custom.html
https://pymoo.org/algorithms/moo/nsga2.html

crossover method. It creates two substrings from two individuals at a random point and swaps their corresponding parts. Two offspring individuals are created with this crossover process and they will share traits of their parents while not being identical. Mutation is attempted on every individual, with a 20% chance of flipping a random bit in the bit string.

Individual Selection

Selecting individuals in this NSGA-II solution is firstly done by including the best fronts and then the best individuals determined by crowding distance. The best individuals exist on fronts closest to the pareto curve, so they are the first individuals to be included in the next generation. When only part of a front can be selected because there isn't enough space left in the new population, individuals are ranked based on crowding distance and the best ones are then chosen. Pymoo specifically uses the Manhattan Distance for calculating crowding distance. Choosing the remaining individuals based on crowding score helps ensure diversity among the population from that front. It avoids selecting individuals that are too similar, providing the GA with useful variation. The diagram below is from Pymoo's page on NSGA-II and visualises this selection process.



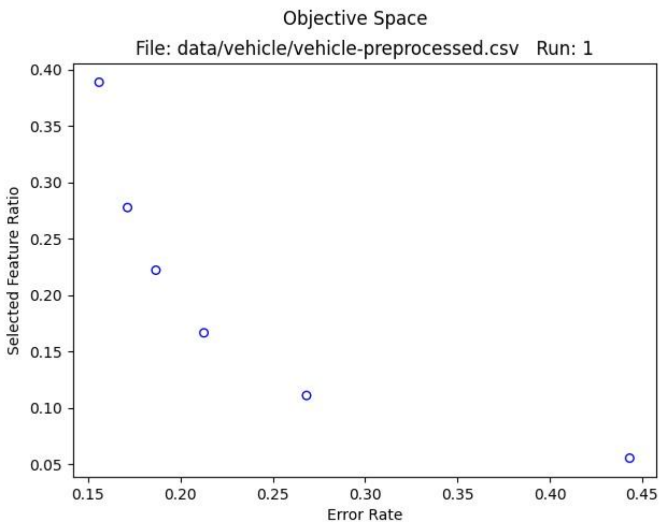
Parameters

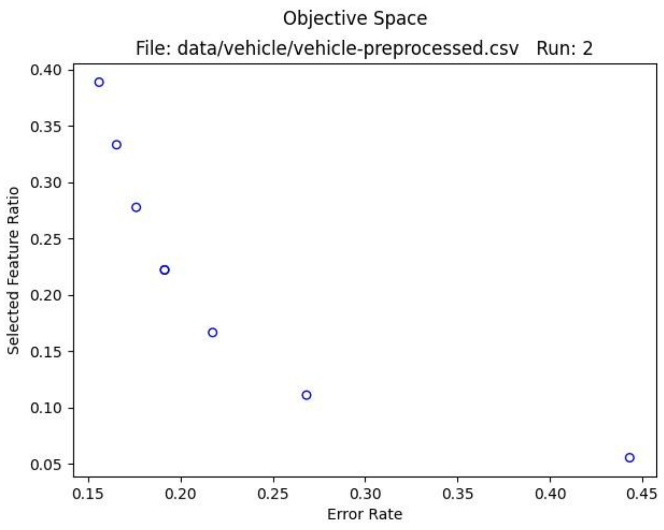
A population size of 100 with 30 generations was able to consistently provide a strong set of solutions that closely resemble the pareto front. Those parameters gave my NSGA-II enough individuals to provide sufficient variation, and enough opportunity to evolve the population to contain a valuable solution set. Experimenting with other parameter values did not improve performance.

Results

I obtained the following results after running my NSGA-II 3 times with different random seeds on each data set.

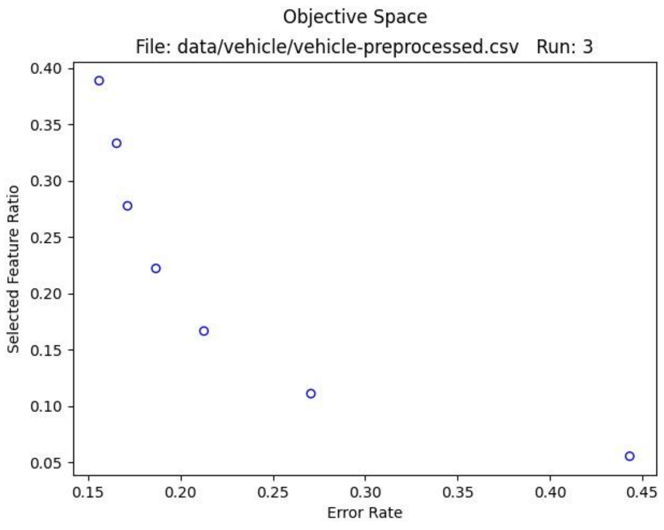
Vehicle Dataset (Error rate with all features = 0.21)						
	Run 1		*Run 2*		*Run 3*	
Solution	**Error Rate**	**Features Selected**	**Error Rate**	**Features Selected**	**Error Rate**	**Features Selected**
1	0.16	7	0.15	7	0.16	7
2	0.17	5	0.17	6	0.17	6
3	0.19	4	0.18	5	0.17	5
4	0.21	3	0.19	4	0.19	4
5	0.27	2	0.19	4	0.21	3
6	0.44	1	0.22	3	0.27	2
7			0.27	2	0.44	1
8			0.44	1		
Hyper-volume	0.9572		0.9567		0.9574	





These graphs clearly show that my NSGA-II algorithm was able to find a solution set which appropriately resembles the expected Pareto front shape. Furthermore, the results effectively visualise the relationship between the number of features selected and the error rate.

If more features are selected, the individual would have more information for the classifier to learn from and could then use that to make better decisions. For example, solution #1 in Run 1 from the table above has an error rate of 0.16 with 7 features selected. Solution #6 from the same run contrasts this with an error rate 0.44 with only 1 feature selected. In-between these two solutions are others that demonstrate the inverse relationship between the selected feature ratio, and error rate.



Each run performed similarly well, with a smooth Pareto front being created from the solution set.

An interesting result that I found is that each run was able to find the same solution of 1 feature with an error rate of 44%. While this is quite high, it is notably better than the 75% error rate of a random guess was used to classify an instance out of the four classes. This is with the significant reduction of 94% in features being used.

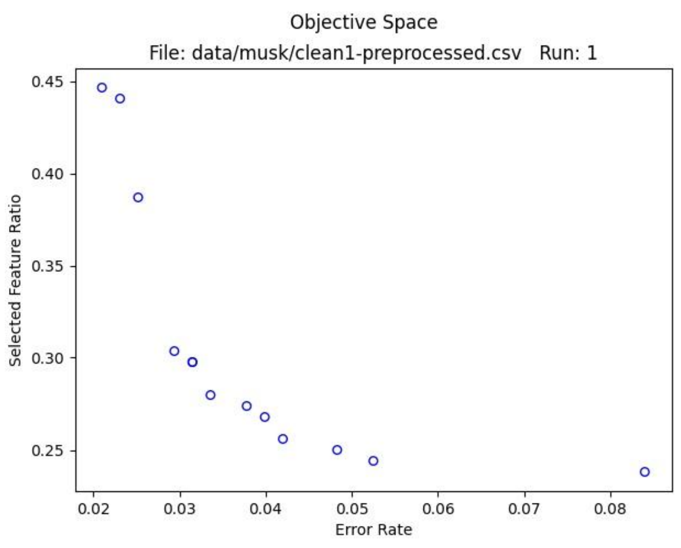
Another interesting observation I made from my results is how each solution compares to the error rate of when all features are used for classification. As expected, those with significantly less features/information do not perform as well. However, solutions with 20% - 40% of features selected actually perform better than when using all of the features. Perhaps this is due to some features containing a lot of noise and making classification more difficult. The learner must recognise this and avoid those features, reducing the selected feature ratio and also the error rate.

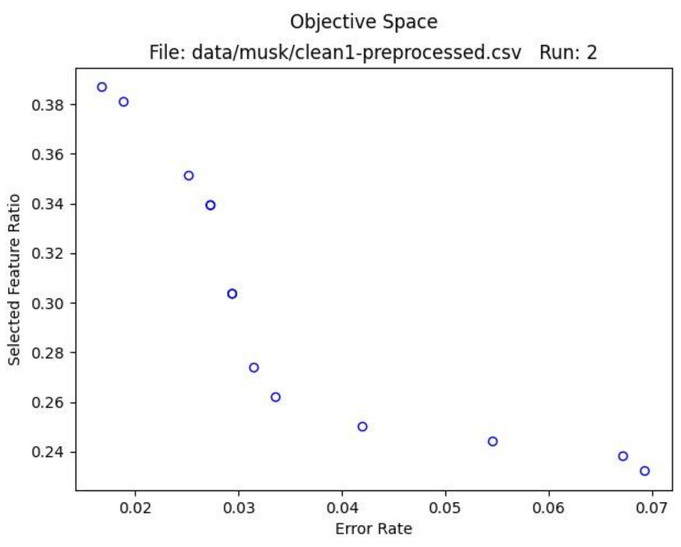
The hyper-volumes of all of the runs were very high, indicating good performance. The difference between them was very small, indicating that the quality of results did not change much. This is visualised in the graphs to back up these hyper-volume results.

An improved Pareto front would contain more solutions for a better distribution, but I wasn't able to achieve this using Pymoo or increasing the population size.

Clean1 Dataset (Error rate with all features = 0.046)						
	Run 1		*Run 2*		*Run 3*	
Solution	**Error Rate**	**Features Selected**	**Error Rate**	**Features Selected**	**Error Rate**	**Features Selected**
1	0.021	75	0.017	65	0.017	54
2	0.023	74	0.019	64	0.019	48
3	0.025	65	0.025	59	0.025	47
4	0.029	51	0.027	57	0.027	46
5	0.032	50	0.027	57	0.032	43
6	0.032	50	0.029	51	0.042	40

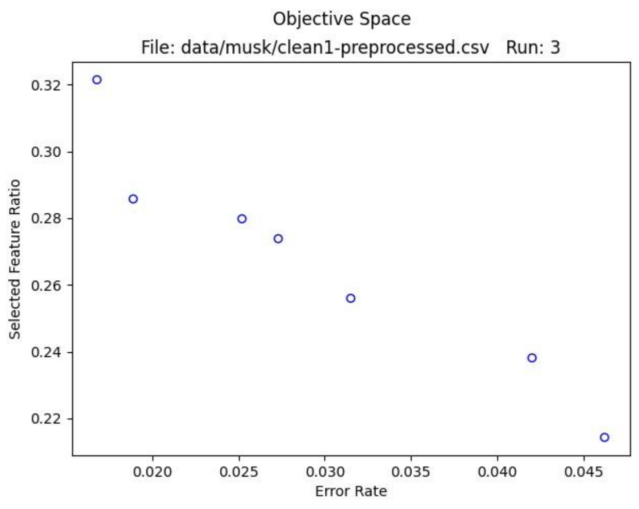
7	0.034	47	0.032	51	0.046	36
8	0.038	46	0.034	46		
9	0.040	45	0.037	44		
10	0.042	43	0.042	42		
11	0.048	42	0.055	41		
12	0.053	41	0.067	40		
13	0.084	40	0.069	39		
Hyper-volume	0.9276		0.9374		0.9578	





The solution set and resultant Pareto front from my NSGA-II on this dataset were not as "smooth" or consistent as on the Vehicles dataset. This may be because of the drastically higher number of features, and therefore larger solution space that the GA needs to search. Compared to the Vehicles dataset, two of the three runs provided a much larger solution set.

These graphs visualise the inconsistent results of my NSGA-II on this dataset. The first run provided a solution set that closely resembles the Pareto curve, but with relatively poor distribution. The second run had quite good distribution, but it contained a concave section in its Pareto front which is unlike any other run from either dataset. The last run's solution set was quite small and didn't closely follow the Pareto front. This could be an example of the stochastic nature of this GA, where some runs do not perform as well as expected.



Just like the Vehicles dataset, multiple solutions from each run, using far fewer features, are able to achieve notably better performance than when all of the features are used. This pattern is more prevalent with the Clean dataset as almost all solutions have a lower error rate than when all features are used, while having a selected feature ratio of <0.4.

The hyper-volumes of the solution sets on this dataset were also very high, reflecting the good quality of solutions with very small error rates and less features. They varied more than the Vehicles dataset which can be expected given the greater variation in results. This is also backed up by the visualised patterns in the graphs.

Part 3: Cooperative Co-evolution Genetic Programming

Function/Terminal Set Discussion

As we don't know anything about the target model except for it being piecewise, I have included more operators in the function set and more constants in the terminal set compared to the GP component of Project 1. Originally, my function set included addition, subtraction, division (specially implemented to avoid 0 error), multiplication, squaring, and sin. My terminal set from the last project was quite small, consisting of only integers 1 through 7. The extra trigonometric functions I've included are cos and tan, in addition to the original sin function. I've also included a larger range of constants in the terminal set, ranging from 1 to 21. As we also don't know how the different parts of the piecewise target model differ, the function and terminal sets are the same between the two sub-populations.

The same fitness cases and fitness function have been used from my GP component in the previous project. The fitness cases range from -100 to 98 which is a wide enough range for individuals to be properly assessed. The fitness function takes in all of these cases and calculates the mean squared error of the expected outputs and the outputs given by the individual.

Parameter Discussion

A total population size of 1000 was used and each sub-population had a size of 500. These parameter values gave the GP enough variation to help explore the solution space. A maximum tree depth of 17 was used, just like in my code for the previous project. I chose this value based on my research into GP. For termination criteria, I had the program stop after 100 generations. I found this to be acceptable as the GP would always find excellent individuals within this scope. However, my termination criteria could be improved upon viewing the results. My GP was able to find expressions which gave the perfect results, so the criteria could have instead been when the error was 0. As for the crossover and mutation rates, I had them set to 0.9 and 0.1, respectively. This allowed for sufficient exploration with enough randomness, while making sure that 100% of the population had been covered as the two values add to 1.

Results

Running my GP 3 times with different random seeds gave me the following results. The outputs of the individuals are a sample of the total outputs, available in my script file along with the tree structure. It would be impractical including them in the report due to their length.

Run 1
```
-8 ) Expected: 51.0     GP Result: 51.0
-6 ) Expected: 27.0     GP Result: 27.0
-4 ) Expected: 11.0     GP Result: 11.0
-2 ) Expected: 3.0     GP Result: 3.0
0 ) Expected: 3.0     GP Result: 3.0
2 ) Expected: 1.4092974268256817     GP Result: 1.416587541675153
4 ) Expected: -0.5068024953079282     GP Result: -0.7674422603384949
6 ) Expected: -0.1127488315322592     GP Result: -0.2798164675245637
8 ) Expected: 1.114358246623382     GP Result: 1.034287964861129
```

Run 2
```
-8 ) Expected: 51.0     GP Result: 51.0
-6 ) Expected: 27.0     GP Result: 27.0
-4 ) Expected: 11.0     GP Result: 11.0
-2 ) Expected: 3.0     GP Result: 3.0
0 ) Expected: 3.0     GP Result: 3.0
2 ) Expected: 1.4092974268256817     GP Result: 1.4092974268256817
4 ) Expected: -0.5068024953079282     GP Result: -0.5068024953079282
6 ) Expected: -0.1127488315322592     GP Result: -0.1127488315322592
8 ) Expected: 1.114358246623382     GP Result: 1.114358246623382
```

Run 3
```
-8 ) Expected: 51.0     GP Result: 51.0
-6 ) Expected: 27.0     GP Result: 27.0
-4 ) Expected: 11.0     GP Result: 11.0
-2 ) Expected: 3.0     GP Result: 3.0
0 ) Expected: 3.0     GP Result: 3.0
2 ) Expected: 1.4092974268256817     GP Result: 1.3876147120763627
4 ) Expected: -0.5068024953079282     GP Result: -0.757213110688501
6 ) Expected: -0.1127488315322592     GP Result: -0.2794977401401723
8 ) Expected: 1.114358246623382     GP Result: 0.9990480692401078
```

Results Discussion

The results of my cooperative co-evolution GP are very promising. Overall, the trees created by the GP were able to give almost perfect results. In all three runs, it managed to find expressions that give exactly the expected values when X > 0. The results for when X < 0 are still very good, particularly in the second run. These samples of the fitness cases are representative of each individual's fitness, but the full results can be seen in my script file.

I noticed that the results of my GP were consistently better when the cooperative co-evolution component was implemented compared to Project 1. This is understandable as the sub-populations allowed the GP to start searching for an expression for the respective piecewise part immediately. This is unlike the GP in the previous project, which had to learn to split the tree into two expressions. Therefore, faster convergence and better results can be expected with this cooperative co-evolution approach.

Overall, these results show that my implementation of this type of GP and my parameter choices have been appropriate in finding excellent results.

Part 4: Reinforcement learning problems and algorithms

CartPole-v1

CartPole-v1 is a problem that involves a pole balancing on top of a cart that moves left and right. The pole pivots around the centre point of the cart and naturally falls left and right due to gravity. The solution to the problem is having the pole continually balance upright without falling beyond a certain vertical angle. The cart is also not allowed to move more than 2.4 units from the centre.

The states in the CartPole-v1 problem are as follows, as defined :

- Position of the cart on the track
- Angle of the pole with the vertical
- Cart velocity
- Rate of change of the angle[3]

This is all the information that the policy will need to know which action to take in order to keep balancing the pole. Velocities include direction as they're a vector which is information that the policy needs. Simply recording speed would not be useful.

The discrete action space contains two actions - left and right, in regards to moving the cart. It is possible to end the episode when performing one of these actions as it could exceed the cart's distance limit from the centre. These actions also directly influence the angle of the pole, which affects how good the episode is.

The reward function operates by giving a reward for every time unit that the pole remains in the allowed angle range. In other words, the episode is rewarded while the pole remains upright, regardless of the exact angle.

[3] G. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem", IEEE Transactions on Systems, Man, and Cybernetics, 1983, page 838.

As the pole can theoretically be balanced forever in the CartPole-v1 problem, it could have an infinite episode length. However, episodes should terminate at some point so that the policy can be assessed. In the journal article “Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem”, episodes were terminated after 500,000 steps (2.8 hours).[3] Therefore, the maximum length of any episode is determined by the termination point.

Criteria for successful learning includes having the policy be able to balance the pole for an extended period of time. This would be achieved by having the policy receive a large reward, effectively making it learn to keep the pole upright. The policy also needs to be able to move the cart in a manner which keeps the pole balanced, otherwise learning would be unsuccessful because the learner wouldn’t be useful.

High-Level PPO Summary

Proximal Policy Optimisation (PPO) is an on-policy, first order algorithm which features some benefits of Trust Region Policy Optimisation (TRPO), while being a better optimiser for a few reasons. It is easier to implement, can be applied more generally, and has better sample complexity.[4] Its first order attribute means that it only uses the first derivative within the method. As PPO is on-policy, it is aiming to improve the same policy that is being used to generate results. This is unlike off-policy, which would be improving a different policy.

In PPO, two policies are maintained during optimisation. One is the current policy, and the other is the policy that we want to improve based on the current one. The optimiser uses samples (a collection of an initial state, action taken, new state, and received reward) from the current policy and uses them to evaluate the new policy.

PPO can be implemented with a clipping function, which prevents the new policy from deviating too far from the current policy. It chooses the minimum of the unclipped and clipped objective function values. Policies are prone to drastic performance degradation from taking steps too far, so clipping improves reliability by keeping the next refinement of the policy close to the current one. The article “Proximal Policy Optimization Algorithms” shows the clipping method to be valuable for the optimiser as results were often worse than when clipping wasn’t applied.[4]

Pseudo code for the clipping implemented PPO is as follows, obtained from Spinning AI.[5]

[4] https://arxiv.org/pdf/1707.06347.pdf
[5] https://spinningup.openai.com/en/latest/algorithms/ppo.html#background

Algorithm 1 PPO-Clip

```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, ...$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
       $$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
       $$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$
       typically via some gradient descent algorithm.
8: end for
```

As indicated in the pseudo code, the policy is updated by using the function in step 6. The new policy is estimated with samples from the current policy. The ratio of the probabilities show how similar/different the two are. Clipping also occurs, which prevents the new policy from allowing state updates that are too far from the current policy after performing an action. It chooses the policy with the minimum objective function value. This is one of the main functions which allows PPO to learn. It explores the solution space around the current policy. The extremity of the exploration is dependent on the "initial condition and the training procedure."[6] To help the policy reach an optima, the policy can progressively become less random. However, this may cause it to get stuck in a local optima.

Experimental Evaluation

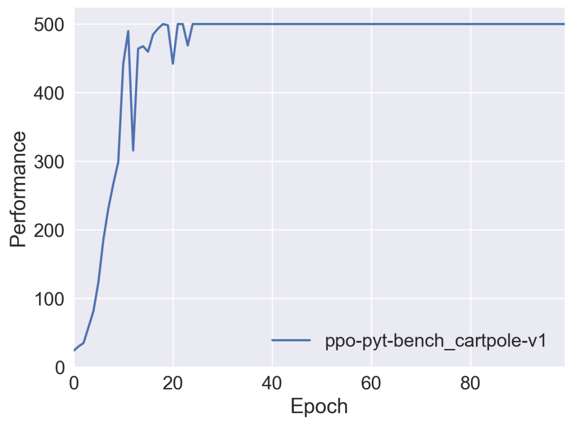
Following the implementation of PPO in "Proximal Policy Optimization Algorithms"[7], my policy neural network used a fully connected MLP with two hidden layers of 64 units. It also used tanh nonlinearities like in the journal article. The discount factor value was initially set to 0.99 which is the default in Spinning Up, but this was later changed for the last experiment. A maximum episode length of 500 was used.

Performance Curves

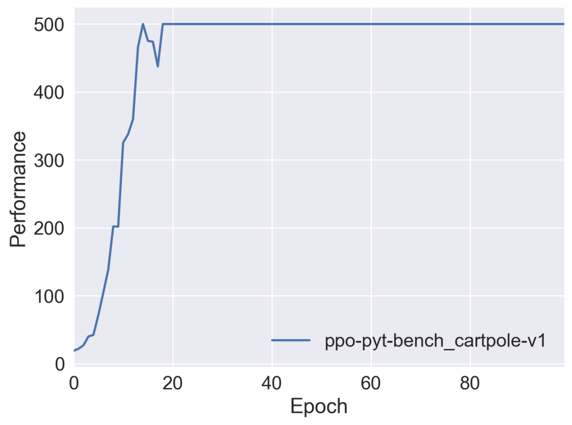
Running my PPO 5 times gave me the following graphs and average learning performance averages.

[6] https://spinningup.openai.com/en/latest/algorithms/ppo.html#exploration-vs-exploitation
[7] https://arxiv.org/pdf/1707.06347.pdf

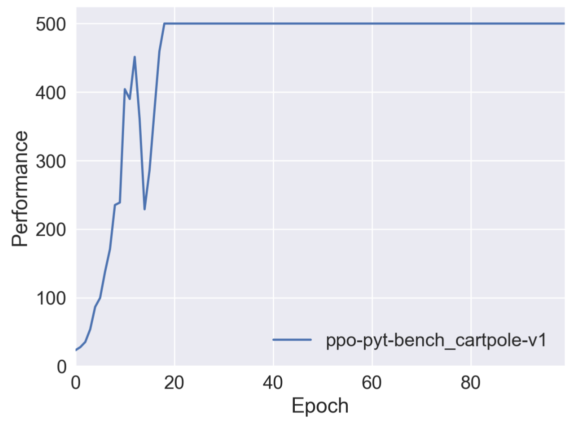
Run 1
Average Learning Performance: 458.613



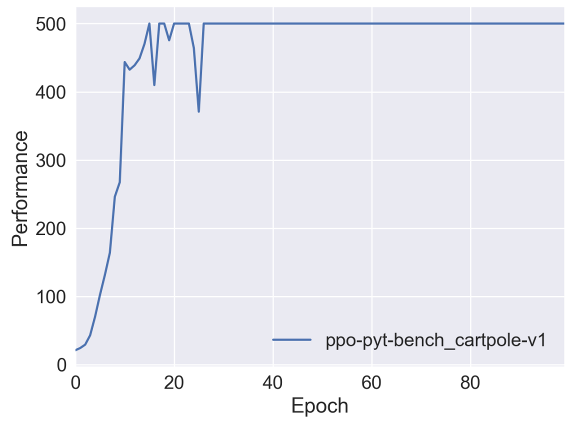
Run 2
Average Learning Performance: 452.455



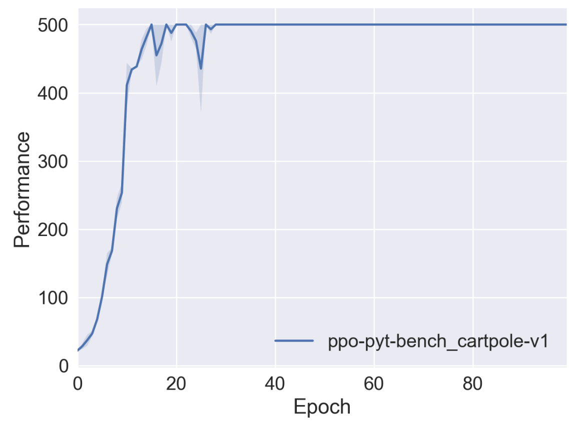
Run 3
Average Learning Performance: 450.618



Run 4
Average Learning Performance: 455.525



Run 5
Average Learning Performance: 457.333



Discount Factor Results and Discussion

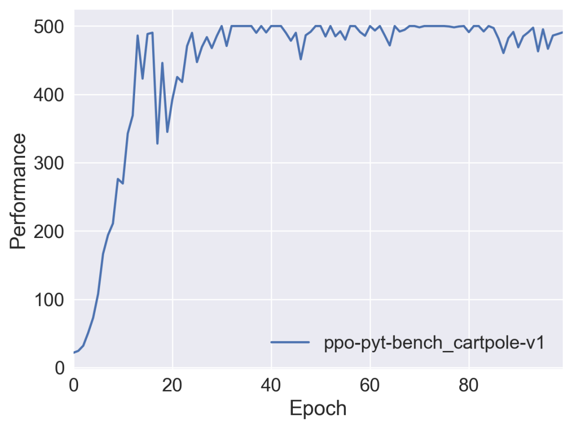
Gamma value: 0.99

Refer to results above in ‘Performance Curves’

Gamma value: 0.9

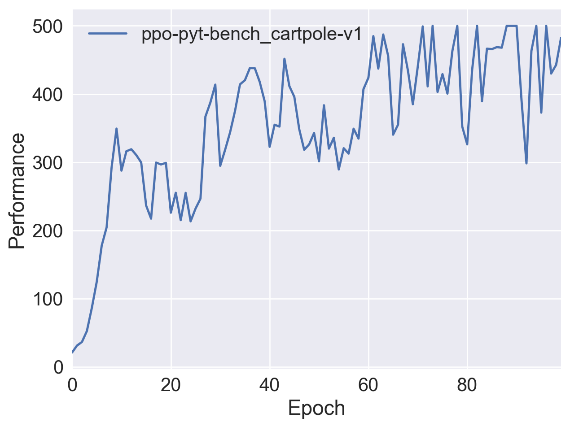
Run 1

Average Learning Performance: 441.140



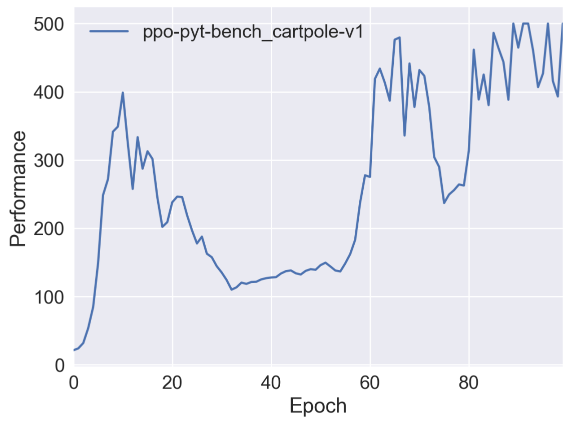
Run 2

Average Learning Performance: 354.701



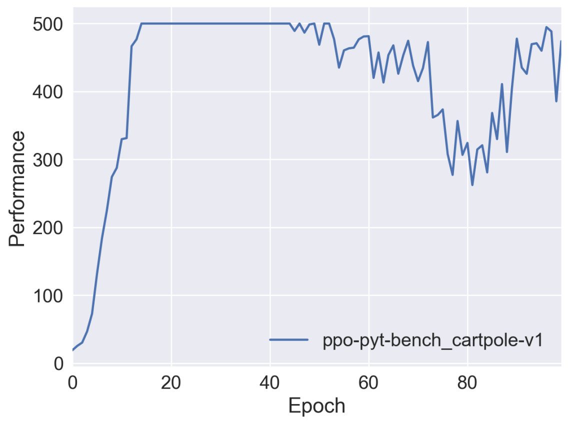
Run 3

Average Learning Performance: 266.688



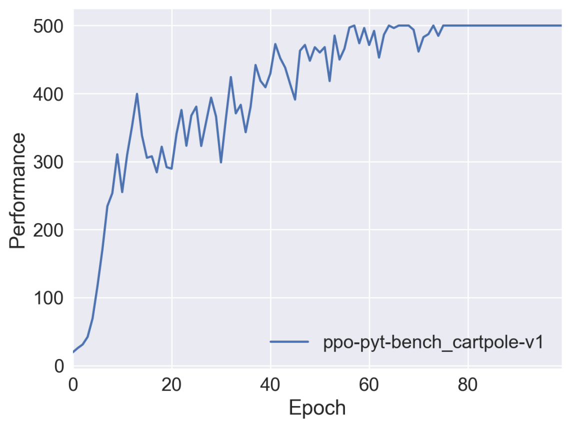
Run 4

Average Learning Performance: 416.677



Run 5

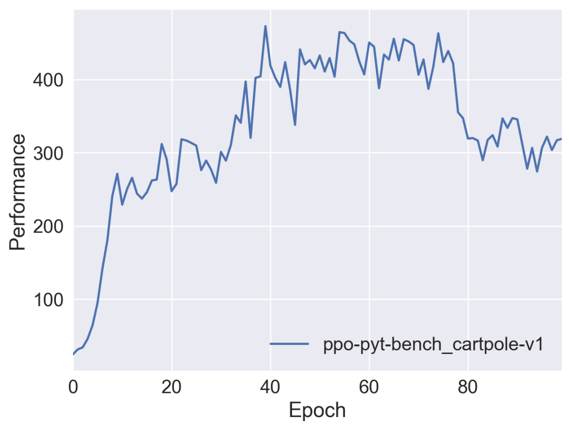
Average Learning Performance: 407.683



Gamma value: 0.7

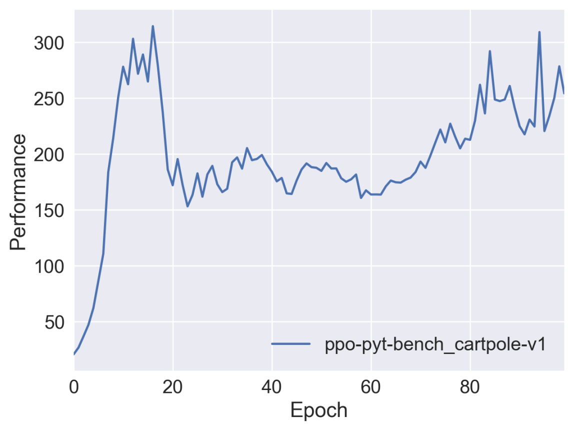
Run 1

Average Learning Performance: 333.592



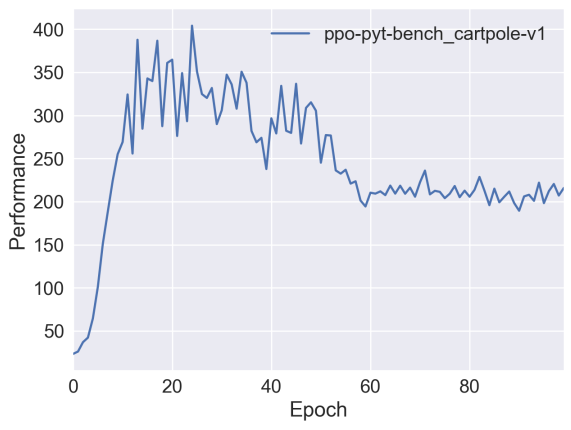
Run 2

Average Learning Performance: 196.860



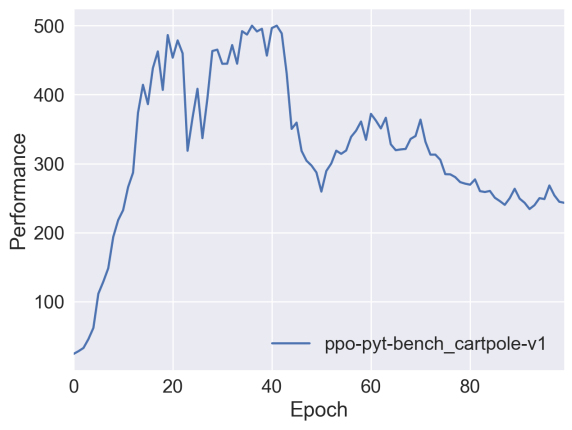
Run 3

Average Learning Performance: 244.764



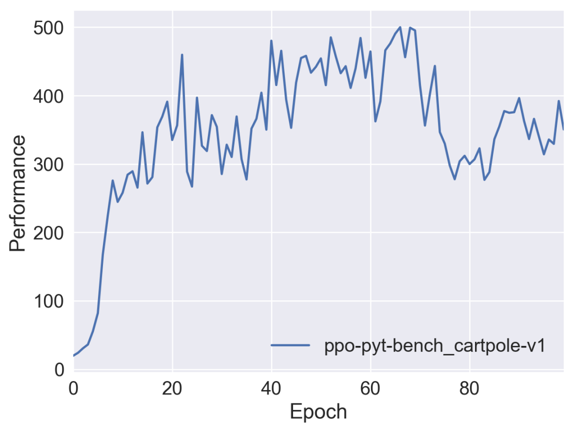
Run 4

Average Learning Performance: 321.227



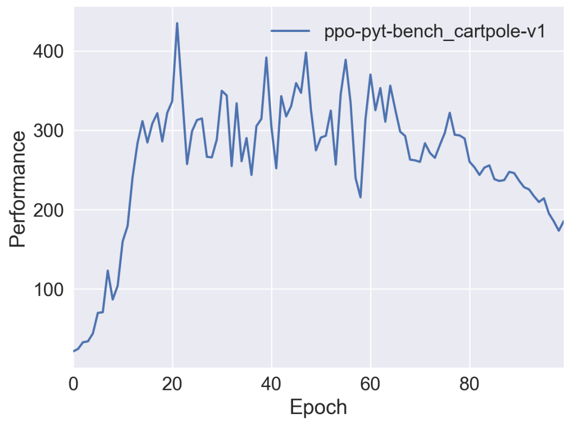
Run 5

Average Learning Performance: 347.852

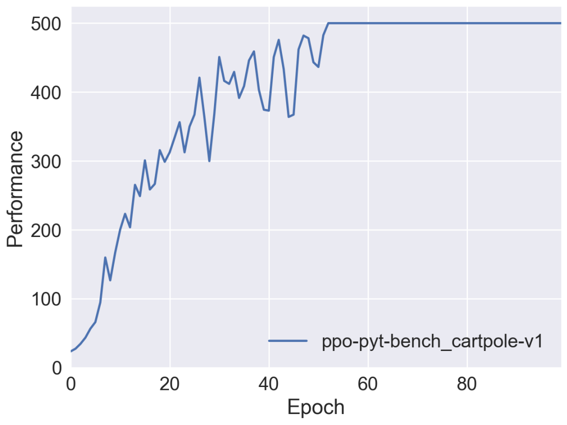


Gamma value: 0.5

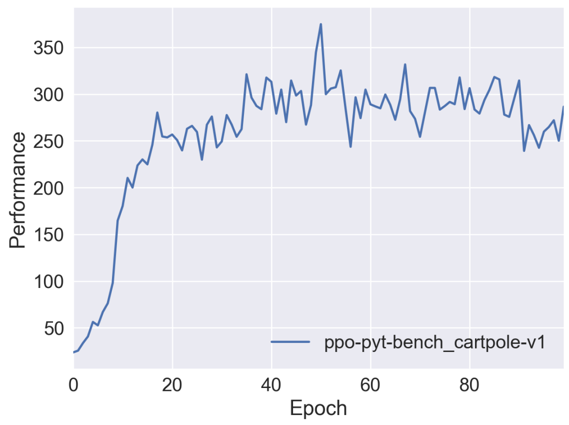
Run 1
Average Learning Performance: 263.250



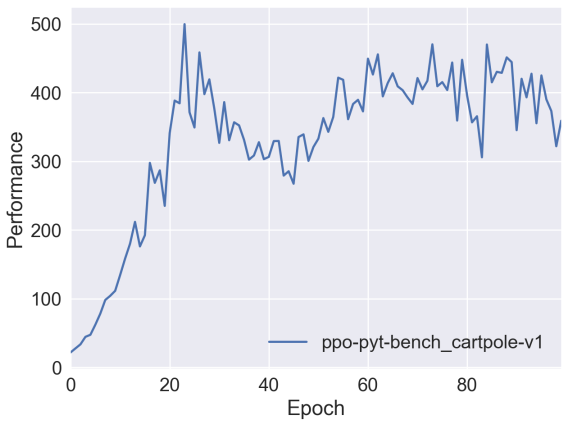
Run 2
Average Learning Performance: 403.033



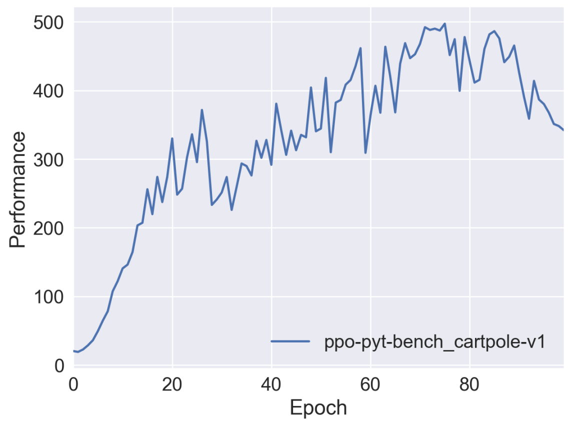
Run 3
Average Learning Performance: 257.192



Run 2
Average Learning Performance: 331.483



Run 5
Average Learning Performance: 329.228



These graphs visualise the performance of the PPO with the different discount factor values. While general trends can be seen across the different values, the results are fairly inconsistent within each group. This demonstrates the stochastic nature of this PPO and how each run will provide different results.

However, different patterns can be seen from using varying gamma values. The higher gamma values prioritise high, early performance while lower values prioritise reaching high performance levels in the long run. This theory is best visualised at either end of my gamma level results.

All of the runs with the gamma level set to 0.99 remained at the maximum level of performance for the large part of their duration. The average learning rates of all 0.99 gamma runs were also very high, consistently scoring above 450. A slightly lower gamma level of 0.9 also resulted in strong final performance of 480-500, but greater fluctuation during training. The learning performance averages were most commonly around 400. However, run #3 may be a possible outlier as its relatively poor performance can be seen on the graph and it has a notably lower average learning performance value. Independent runs with gamma values 0.7 and 0.5 continue these trends, with greater fluctuation and lower overall performance as gamma decreases.

Some interesting patterns can be seen in the runs with the gamma value set to 0.99. Despite the high gamma level making the PPO favour long term performance, it consistently and quickly reaches the maximum reward of 500. This contrasts lower discount factor values which are not as consistent and do not learn as fast. Some low discount factor runs do learn very quick for early performance, but they’re not consistent and sometimes aren’t as fast.

I also didn’t expect the results to have this much variation within each discount factor group. Particularly in the lower gamma levels, each run can be very different from the next. This showed me how important multiple runs of the PPO were to help investigate the effect of the discount factor parameters. Perhaps a low discount factor value isn’t as appropriate for this problem, but the runs with 0.99 gamma value showed to be very valuable.