EECS 127 Project: Optimisation Algorithms

Ishaan Srivastava

May (the) 4th (be with you), 2020

1 Overview

In this section, I'm going to explain the general structure of my project. In the implementation section, I'll talk about the process of implementing my algorithms. Since all the code I've written is already available in the accompanying Jupyter notebook, I'll discuss only certain features of my implementation in this report. The exploration and challenge section has a subsection devoted to each of the 4 different objective functions. For the sake of concision, I'm just going to report some of the different settings I tried for each function, and comment on what behaviour I observed and errors I recorded. There will also be accompanying 2-D contour plots with path tracing and 3-D plots (or videos) of each objective function for the final choice of method and hyperparameters used. The plots will be provided in the report itself, while for some functions, videos with 3-D path tracing will be uploaded to Google Drive and linked in the report. Videos are provided for only some functions because they would take too long to generate for all the different functions, especially when the number of iterations is large. In the extra credit section, I discuss how and why I went beyond the project requirements, and wrap up the project with a short summary.

2 Implementation

The parts of my implementation that are worth mentioning are as follows: for my exit condition using tolerance, rather than check whether the norm of the difference between two consecutive iterates is $\leq$ the tolerance, I instead checked if the 2-norm of the Jacobian of the iterate was $\leq$ the tolerance. For the challenge section of the project, I set the tolerance to 0 in most cases so this wouldn't matter, but given my results for x^2 using all my different optimisation methods, this stopping condition clearly led to good convergence. That being said, I think this is why my algorithm sometimes ran for an iteration more than the provided staff values when optimising x^2. Refer to notebook for further details.

To check if my method exited successfully, I initialised a variable success_bool as False and set its value to True if the Jacobian of the iterate was $\leq$ the tolerance or if the total number of iterations was fewer than the max number of possible iterations. Hence success_bool is set to True if and only if the optimiser exits successfully. Note that as per the project spec, moving averages (wherever applicable) are initialised as 0.

While working on the momentum function, I noticed that my iterates for x^2 were different from the provided staff values, even though I was fairly confident in my implementation. I noticed that the discrepancy was caused by a different sign convention being used to update the momentum term and perform the overall update. Refer to the notebook for my implementation compared to the code which yields the same iterate values as the staff implementation. Note that at first glance they appear to give equivalent results, but after the first iteration the iterates in the respective implementations start diverging from each other. As far as I can tell, it's because of the alternating addition and subtraction of the gradient from the momentum term in the staff implementation. In the specific case of the hyperparameters being used for the provided staff implementation of momentum to optimise x^2, the iterate actually diverged to ∞ after enough iterations, while in comparison my implementation converged in less than 100 iterations with a tolerance of 10^{-3}. I also found this sign convention causing similar (but less severe) issues for the staff implementation of nag, and brought it to their notice at `https://piazza.com/class/k5j2k0fnzj91vh?cid=650`

3 Exploration and Challenge

Note that in this section (especially in the challenge part), the tolerance was set to 0 in some cases when I was trying to get as close to the optimal value as possible, even if each update was almost negligible. This section frequently mentions what ans.get_min_errs() outputs to evaluate how well the function was optimised. As provided for us in the notebook, ans.get_min_errs() "Returns a tuple containing

1. float representing the closest (in L2 norm) the optimization procedure got to the global minimizer.
2. float representing the closest the optimization procedure got to the global minimum function value."

Also note that whenever path plots and videos are provided, the path taken is traced in black.

3.1 Booth Function

1. ans.set_settings(fn_name='booth', method='nag', x0=np.array([8, 9]), lr=lambda x: 0.05, num_iters=30, tol=1e-7)
   - From the beginning, nag seems like a promising bet based on the plot, with low error values, and quick convergence in just a few iterations. Notably, it proceeded directly in the direction of the optimal solution without overshooting it
   - ans.get_min_errs() yielded (0.0037880896875566008, 1.4349623480970287e-05)
2. ans.set_settings(fn_name='booth', method='momentum', x0=np.array([8, 9]), lr=lambda x: 0.02, num_iters=200, tol=1e-7)
   - The momentum function overshoots the optimal point multiple times, but ultimately converges well. It has lower error values than the previous nag implementation, but requires far more iterations, and I can't increase the learning rate to lower the number of iterations because then momentum will overshoot the optimal point and not converge properly
   - ans.get_min_errs() yielded (7.2664080476633422e-05, 4.7465993532352449e-08)
3. ans.set_settings(fn_name='booth', method='gd', x0=np.array([8, 9]), lr=lambda x: 0.01, num_iters=1000, tol=1e-7)
   - Vanilla gradient descent converges really well, steadily moving towards the optimal value with smaller steps as the gradients get smaller (as expected). At this point I've realised that my initial number of iterations for nag was too low, and since the code runs quickly enough, I can have more iterations and if needed, smaller learning rate for better convergence

- ans.get_min_errs() yielded (4.9973145668290556e-08, 2.4973153036766789e-15)

4. ans.set_settings(fn_name='booth', method='momentum', x0=np.array([8, 9]), lr=lambda x: 0.00003, num_iters=1000, tol=1e-7)
    - At this point, it becomes clear that it's quite difficult to find the right balance between learning rate and maximum number of iterations, especially if I have to worry about changing the value of α. As such, it's probably best to stick to one of the other optimisation algorithms for this function
    - ans.get_min_errs() yielded (0.39018634429405052, 0.16055149562402674)
5. ans.set_settings(fn_name='booth', method='adagrad', x0=np.array([8, 9]), lr=lambda x: 5, num_iters=200, tol=1e-7)
    - Adagrad does better than I thought it would, with its adaptive learning rate helping it take big initial steps towards the optimal value and then taking smaller more precise steps to end up very close to the optimal value in relatively few iterations
    - ans.get_min_errs() yielded (4.628544582148059e-08, 2.1423721448155546e-15)
6. ans.set_settings(fn_name='booth', method='nag', x0=np.array([8, 9]), lr=lambda x: 0.02, num_iters=400, tol=0)
    - This turned out to be the crown jewel. Using the knowledge I gained from messing around with the different optimisers and algorithms, I decided to lower the learning rate and increase the number of iterations relative to my initial nag implementation. I also set the tolerance to 0. Unlike my earlier implementation we first overshoot the optimal point, but given how well this converges I don't think that's a problem. I submitted these parameters for the challenge to optimise this function
    - ans.get_min_errs() yielded (3.1401849173675502e-15, 6.3108872417680944e-30)
    - The corresponding contour plot is provided below.

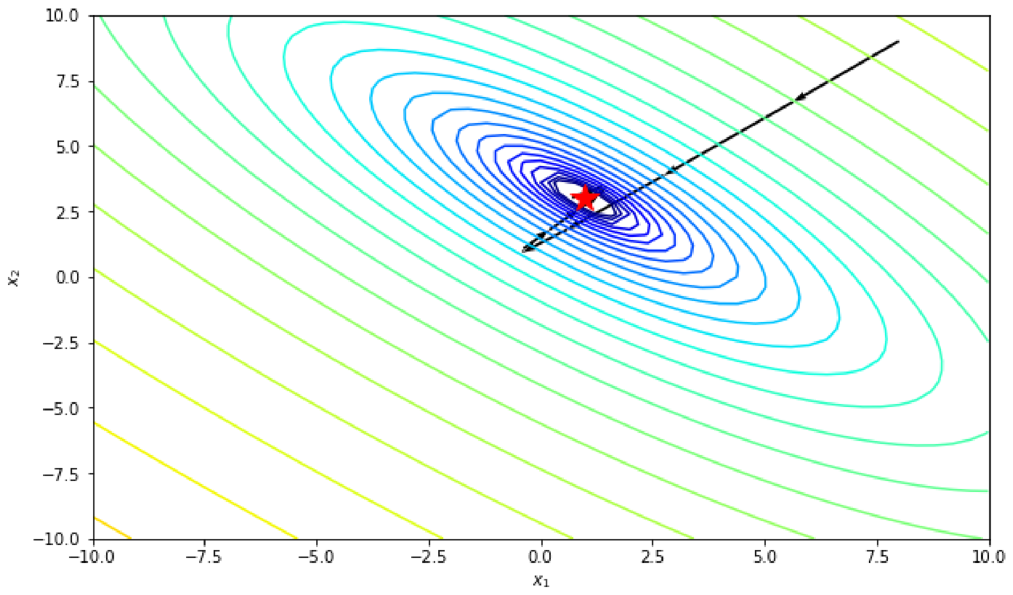

Figure 1: 2-D Contour Plot for Best Booth Function Optimisation

The video for the 3-D path is available at
`https://drive.google.com/open?id=1efg6DChjVKvrlqp2U7nmSxlnVp2MK9as`

3.2 Beale Function

1. ans.set_settings(fn_name='beale', method='gd', x0=np.array([3, 4]), lr=lambda x: 0.001, num_iters=30, tol=1e-7)

   - Turns out that I need to keep learning rate really low here. I received the following warning when I ran this code "Runtime Warning: overflow encountered in double_scalars". From a visual perspective, gradient descent shoots straight past the desired optimal point, and seems to diverge to ∞
   - ans.get_min_errs() yielded (nan, nan)

2. ans.set_settings(fn_name='beale', method='gd', x0=np.array([3, 4]), lr= lambda x: 0.0001, num_iters=100000, tol=1e-7)

   - Lowering the learning rate and increasing the maximum number of iterations helps a fair bit. However, vanilla gradient descent still initially diverges away from the optimal point and then starts doubling back in its direction, which may suggest that it isn't the best algorithm to use here
   - ans.get_min_errs() yielded (0.021365145976007426, 7.0339511751617203e-05)

3. ans.set_settings(fn_name='beale', method='gd', x0=np.array([3, 4]), lr=lambda x: 0.0001, num_iters=1000000, tol=0)

   - Turns out I was wrong. Increasing the number of iterations by a factor of 10 leads to far better convergence, so I don't think it's necessary for gradient descent to exclusively move in the exact direction of desired optimal point. This time I also set tolerance to 0, which resulted in a far better solution.

- ans.get_min_errors() yielded (7.3678272478466685e-12, 8.1829282365009847e-24)

4. ans.set_settings(fn_name='beale', method='momentum', x0=np.array([3, 4]), lr=lambda x: 0.000001, num_iters=1000000, tol=0)

    - Momentum also initially strays away from the optimal point but turns back towards it before gradient descent does. However it has far worse convergence than gradient descent, so I now strongly think that always moving in the "right" direction" isn't necessary, and may not even be sufficient if you terminate too early or take too long to converge. I also set tolerance to 0 here to make sure low gradients weren't causing the optimiser to exit early
    - ans.get_min_errs() yielded (0.016999146809209838, 4.43280720884354e-05)

5. ans.set_settings(fn_name='beale', method='nag', x0=np.array([3, 4]), lr=lambda x: 0.000001, num_iters=100000, tol=1e-7)

    - I first tried using nag with a learning rate of 0.0001 (which led to decent convergence for gradient descent), but I got the same Runtime Warning and nans as before. As such, I lowered by learning rate by a factor of 100 and tried using nag. With these hyperparameters, nag follows a similar path to the optimal point as momentum, but terminates before coming particularly close to it. Even an learning rate of 0.000002 results in the aforementioned warning. Higher number of max iterations might result in convergence?
    - ans.get_min_errs() yielded (0.4432557827965013, 0.047858691826097866)

6. ans.set_settings(fn_name='beale', method='nag', x0=np.array([3, 4]), lr=lambda x: 0.000001, num_iters=1000000, tol=0)

    - Increased number of max iterations by factor of 10 and also set tolerance to 0. Better convergence, but at this point it's clear nag is not the way to go about optimising this function
    - ans.get_min_errs() yielded (0.016719937969624538, 4.2871478819078913e-05)

7. ans.set_settings(fn_name='beale', method='adagrad', x0=np.array([3, 4]), lr=lambda x: 5, num_iters=1000000, tol=0)

    - Adagrad is the way to go here. Despite a very high initial learning rate (I say initial because it adapts), there are no numerical instability issues and it converges very well. Despite having the same number of max iterations as nag, it also converges more quickly than it. It has lowest the errors of all the different settings I've tried including my third attempt. It also converges faster, making it the best choice here. I submitted these parameters for the challenge to optimise this function.
    - ans.get_min_errs() yielded (3.9181953113304346e-12, 2.3139575704826605e-24)
    - The corresponding contour plot and 3D plot are provided below.

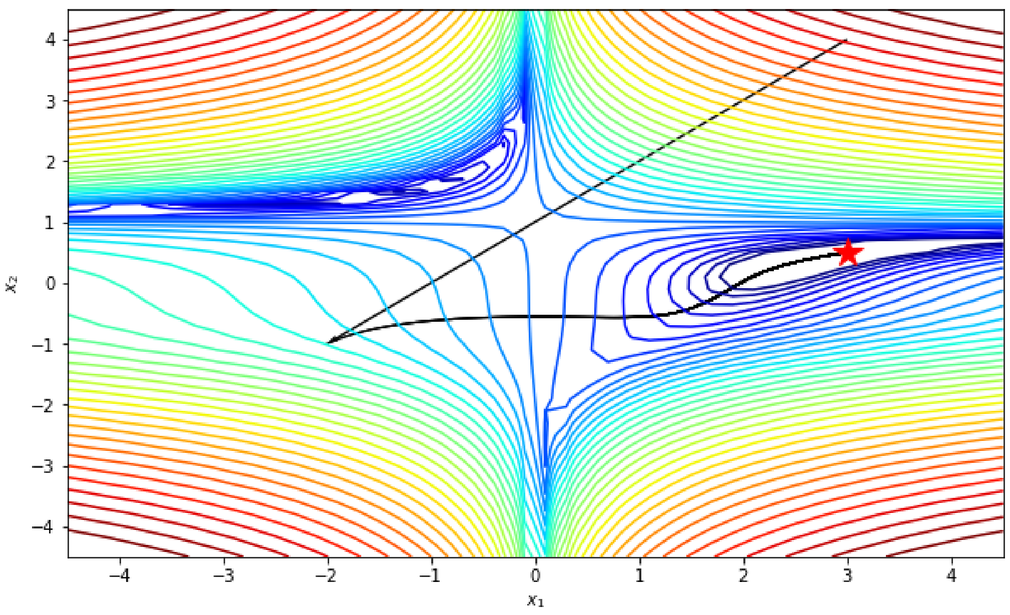

Figure 2: 2-D Contour Plot for Best Beale Function Optimisation

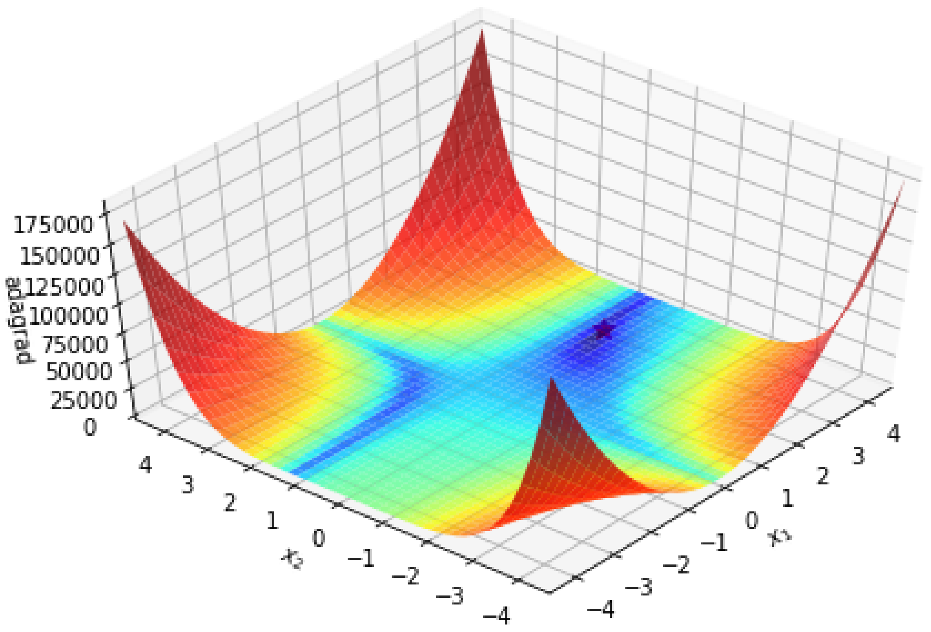

Figure 3: 3-D Plot for Best Beale Function Optimisation

3.3 Rosenbrock Function

1. ans.set_settings(fn_name='rosen2d', method='gd', x0=np.array([8, 9]), lr=lambda x: 0.00005, num_iters=1000000, tol=0)

   - Even with a very low learning rate, gradient descent skips past the first depression of the function. However, it does start approaching the optimal point after entering the depression from the other side (refer to the plots to understand what the function looks like). Overall, pretty decent convergence

- ans.get_min_errs() yielded (1.8122373827368268e-08, 6.5579118837466712e-17)

2. ans.set_settings(fn_name='rosen2d', method='adagrad', x0=np.array([8, 9]), lr=lambda x: 15, num_iters=1000000, tol=0)

   - After gradient descent, I decided to try adagrad to see if the adaptive learning rate would lead to better convergence. Interestingly enough, adagrad initially shoots off in the opposite x_2 direction before eventually coming quite close to the optimal point. It approaches from the second depression and essentially doubles back to the optimal point so it doesn't take the most efficient path, but I've learned that isn't really problem as long as we have good convergence in a reasonable number of iterations, especially for a function like this. I also fiddled around with different learning rates to see if there was an easy way to get better convergence. Reducing the learning rate to 10 didn't stop the issue of overshooting and then doubling back via the second depression, but it did result in higher errors, probably because a lower learning rate made it more susceptible to saturation early on. Increasing the learning rate to 20 actually caused the iterate to move along the first depression as I initially desired, but once again resulted in higher errors than a learning rate of 15.
   - ans.get_min_errs() yielded (8.8944234196865982e-13, 1.5815581396317265e-25)

3. ans.set_settings(fn_name='rosen2d', method='nag', x0=np.array([8, 9]), lr=lambda x: 0.00005, num_iters=1000000, tol=0)

   - Nag overshoots the first depression, then doubles back and starts proceeding along it in order to find the optimal point. It has pretty good convergence (slightly better than adagrad), but I think using the accelerated gradient instead of the momentum is suboptimal since the update seems to approach 0 sooner in this case
   - ans.get_min_errs() yielded (7.3780931740658838e-13, 1.0870286337196607e-25)

4. ans.set_settings(fn_name='rosen2d', method='momentum', x0=np.array([8, 9]), lr=lambda x: 0.00005, num_iters=500000, tol=0)

   - Looking at the shape of the function, I thought momentum might be a good idea, with the thinking that the added momentum term will prevent the update term from being too small which might be what's stopping gradient descent from achieving better convergence. It converges pretty well so I was considering increasing my α value to boost the role of the momentum term in the update, but I realised that the algorithm overshoots both the first and second depression before doubling back and proceeding down the second depression, so increasing the α value would cause it to overshoot even further initially, possibly resulting in lower overall convergence. Of all the different methods and hyperparameters I've tried so far, this works the best for the Rosenbrock function.
   - ans.get_min_errs() yielded (3.661738717576632e-13, 2.6774789613864582e-26)
   - The corresponding contour plot and 3D plot are provided below. Note that the first depression I've mentioned refers to the blue contours on the right, while the second so-called depression refers to the blue contours on the left.

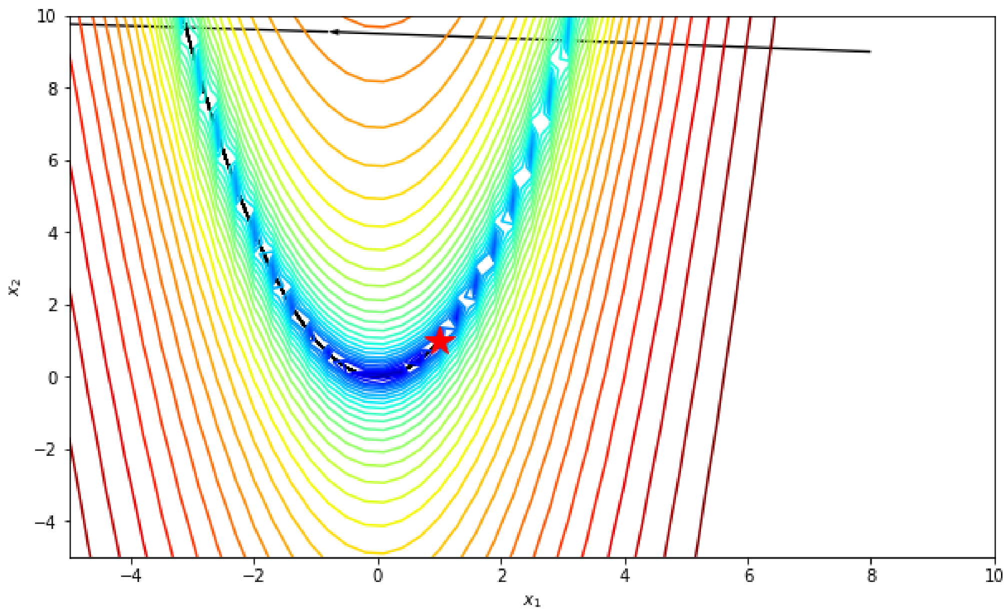

Figure 4: 2-D Contour Plot for Best Rosenbrock Function Optimisation

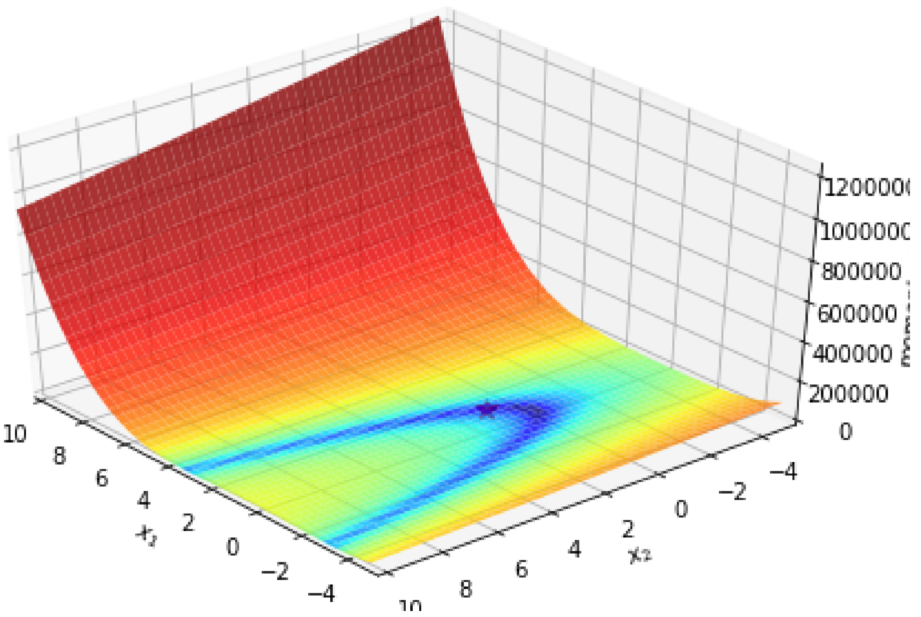

Figure 5: 3-D Plot for Best Rosenbrock Function Optimisation

3.4 Ackley Function

Note that the most important part in optimising this function is the learning rate scheduler used. Please refer to my Jupyter notebook for the relevant function definitions.

1. ans.set_settings(fn_name='ackley2d', method='nag', x0=np.array([25, 20]), lr=nag_boi, num_iters=7, tol=1e-7)

- I didn't even come close with nag, it kept shooting all over the place. Even the slightest change in the learning rate caused the iterate to move in a completely different direction. Increasing number of iterations only made matters worse. Including this only to say that optimising this function with nag seems exceptionally difficult.
- ans.get_min_errors() yielded (11.435733788280084, 17.754846762137987)

2. ans.set_settings(fn_name='ackley2d', method='nag', x0=np.array([25, 20]), lr=nag_boi, num_iters=7, tol=1e-7)

   - Even after some involved hyperparameter tuning, it was difficult to get decent convergence. The issue is that setting a low learning rate after a specific point doesn't work as intended because even if I get the iterate almost into the well (refer to plot to see what I mean), the momentum will push it out. Trying to adjust the α value to fix this while keeping the other hyperparameters constant would result in an algorithm that doesn't approach the well in the first place, regardless of how it would behave once it was there. As is the case with nag, this function does not easily lend itself to being optimised with momentum
   - ans.get_min_errors() yielded (0.55875252171140555, 3.4366291582559647)

3. ans.set_settings(fn_name='ackley2d', method='adagrad', x0=np.array([25, 20]), lr = other_boi, num_iters=1000000, tol=1e-7)

   - Finally made some real progress here. The algorithm approached the optimal solution, then started bouncing into and out of the well. I extensively modified the learning rate scheduler, but to little avail. I don't think adagrad is adaptive enough for this function.
   - ans.get_min_errors() yielded (0.00014102497905554007, 0.00039940846211905523)

4. ans.set_settings(fn_name='ackley2d', method='gd', x0=np.array([25, 20]), lr=lr_boi, num_iters=75000, tol=0)

   - Vanilla gradient descent actually works quite well here, because there aren't too many moving parts to keep track of, making it easier to tune hyperparameters. I tweaked my learning rate scheduler and maximum number of iterations until I ended up with what I thought was pretty good convergence. While the path taken is not perfectly smooth, it's easier to interpret and tune then other algorithms for this function.
   - ans.get_min_errors() yielded (4.3797345574749381e-08, 1.2387764769528076e-07)
   - The corresponding contour plot and 3D plot are provided below. Note that the well I've mentioned refers to the conical part of the function surrounding the optimal value.

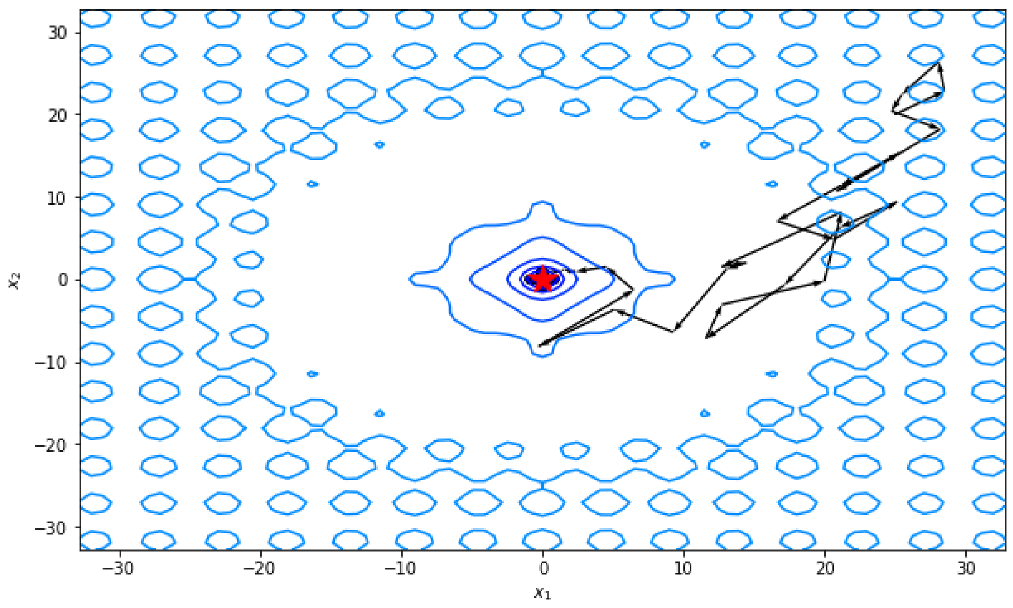

Figure 6: 2-D Contour Plot for Best Ackley Function Optimisation

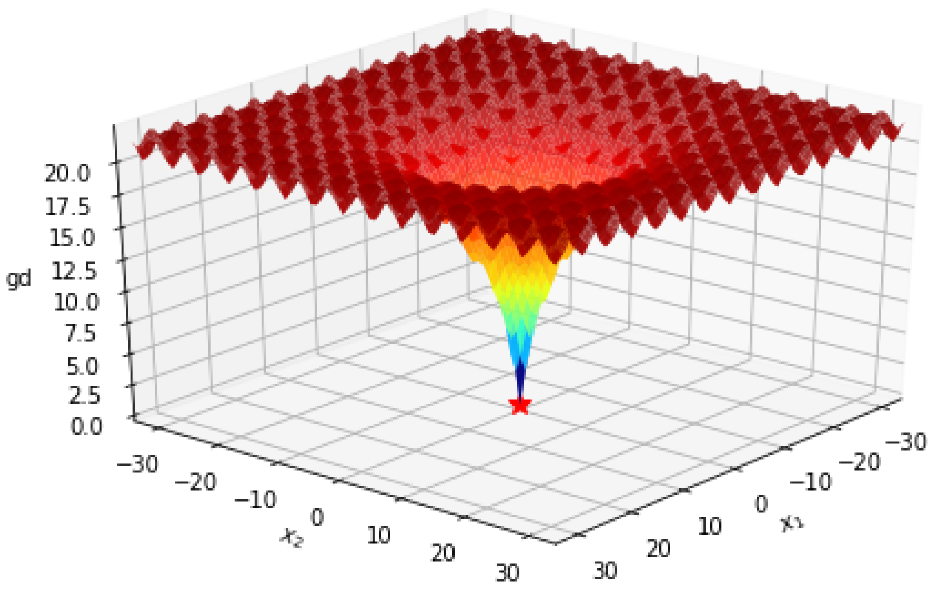

Figure 7: 3-D Plot for Best Ackley Function Optimisation

4 Extra Credit

Here I mention the things I've done that go beyond the project spec; some of these are already mentioned elsewhere in the report, so I only touch on them here. In terms of implementation, it made more sense to me to look at the 2-norm of the Jacobian of the gradient instead of the 2-norm of the vector difference of two consecutive iterates. Given my convergence values (even with nonzero tolerance), I think my implementation was as effective, if not more, than the provided suggestion. I also investigated and caught a bug in the staff implementation of nag and momentum, as mentioned before.

When reflecting on the different optimisation algorithms used, I noticed that adagrad was always one of the better ones to use, but only in the case of the Beale function did it turn out to be the best of all 4. This led me to believe that adaptive methods were probably the best choice to use here, but specifically adagrad wasn't excelling because the accumulation of square of the gradients at each time step was resulting in premature learning rate saturation, preventing better convergence. As such, I read up on different adaptive methods and came across adam, which I thought might fix my issues because it had decaying moving averages of the first and second moments of the gradients, which seemed less prone to saturation than adagrad. Additionally once these estimates were bias-corrected, the overall algorithm seemed pretty robust, integrating the better parts of adagrad and momentum to converge to the optimal value. My observations using adam to optimise the four objective functions are documented below:

1. Booth Function

   - ans.set_settings(fn_name='booth', method='adam', x0=np.array([8, 9]), lr=lambda x: 0.1, num_iters=10000, tol=0)
   - ans.get_min_errors() yielded (3.1401849173675502e-15, 6.3108872417680944e-30)
   - Note that these are the same values returned by my best prior optimisation of the objective function, which was achieved by using nag with a constant learning rate of 0.02 with 0 tolerance and 400 iterations.
   - The contour plot (using adam) and training loss graph (compared to the prior best method) are provided below. The hyperparameters of nag are the same as in the case of the best implementation except for the number of iterations. Notice that nag converges to the same value in fewer iterations and then terminates, so it's still the best method to use here, but adam's performance is encouraging.

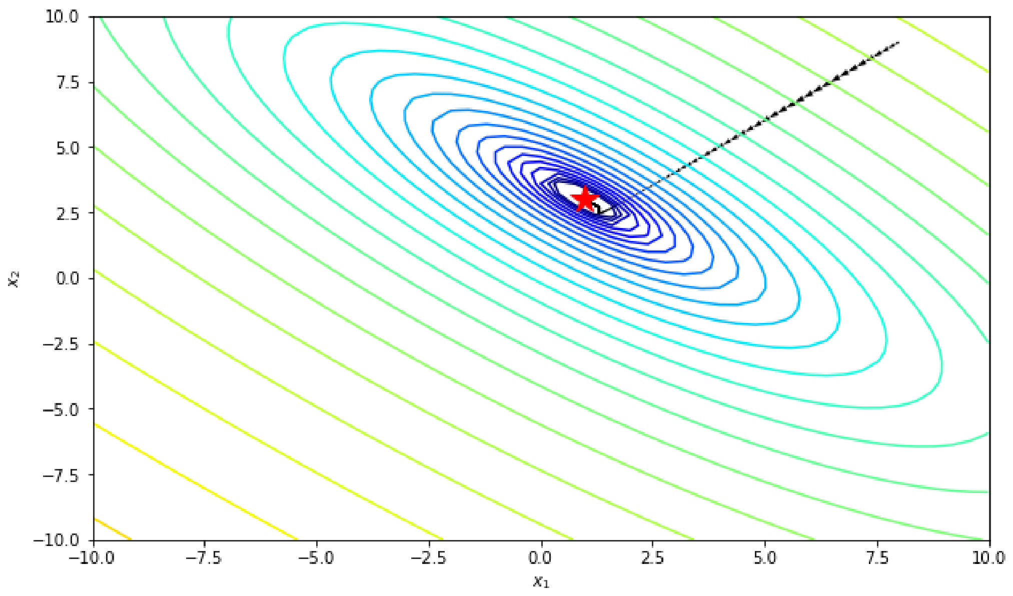

Figure 8: 2-D Contour Plot for Booth Function Optimisation with adam

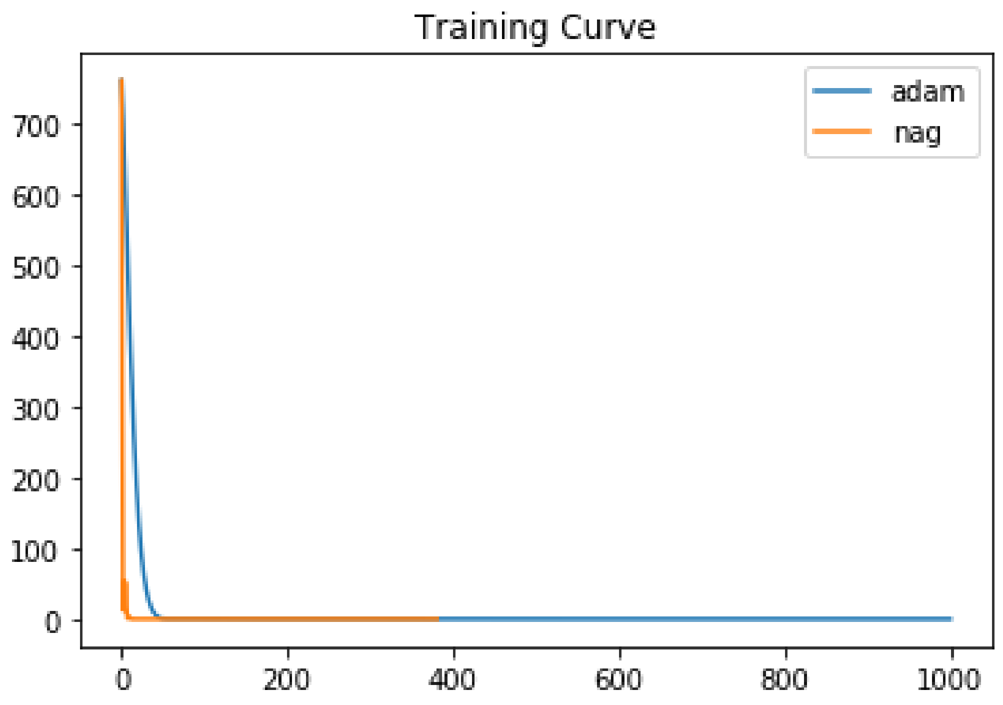

Figure 9: Comparing Training Loss for adam vs nag for Booth Function

2. Beale Function

   - ans.set_settings(fn_name='beale', method='adam', x0=np.array([3, 4]), lr=lambda x: 0.0005, num_iters=50000, tol=0)
   - ans.get_min_errors() yielded (2.5081379732971261, 0.92542122053154952)
   - Unfortunately adam fails here. It appears to get stuck changing an undesireable local minimum instead of our preferred optimal point. Lowering the learning rate and increasing the maximum number of iterations was not effective. Adagrad with 0 tolerance, learning rate 5 and 10^6 iterations is the best method I found here.
   - The contour plot (using adam) and training loss graph (compared to the prior best method) are provided below. The hyperparameters of adagrad are the same as in the case of the best implementation except for the number of iterations. This is not adam's greatest moment, but I have faith.

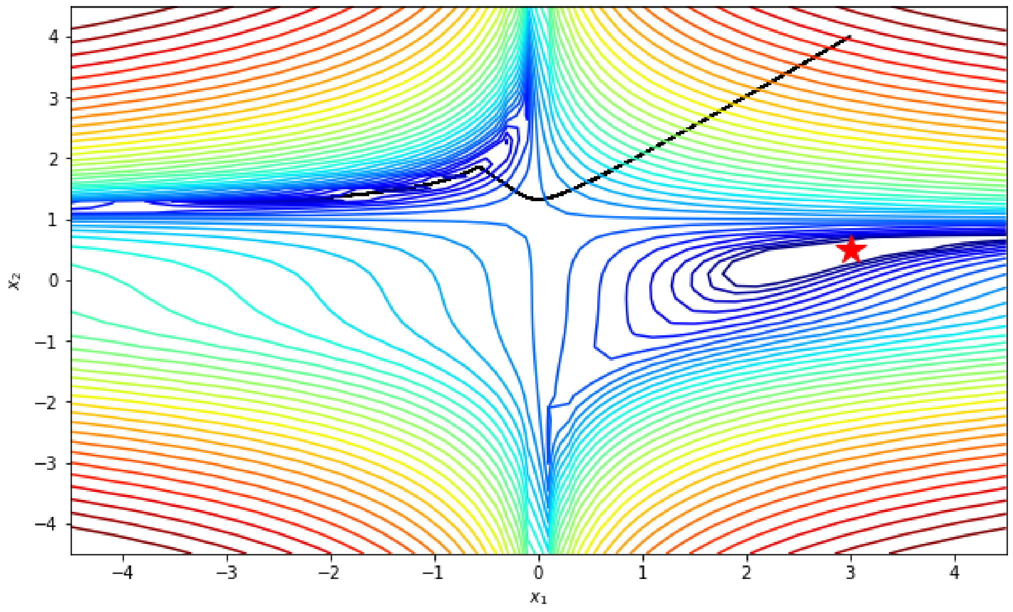

Figure 10: 2-D Contour Plot for Beale Function Optimisation with adam

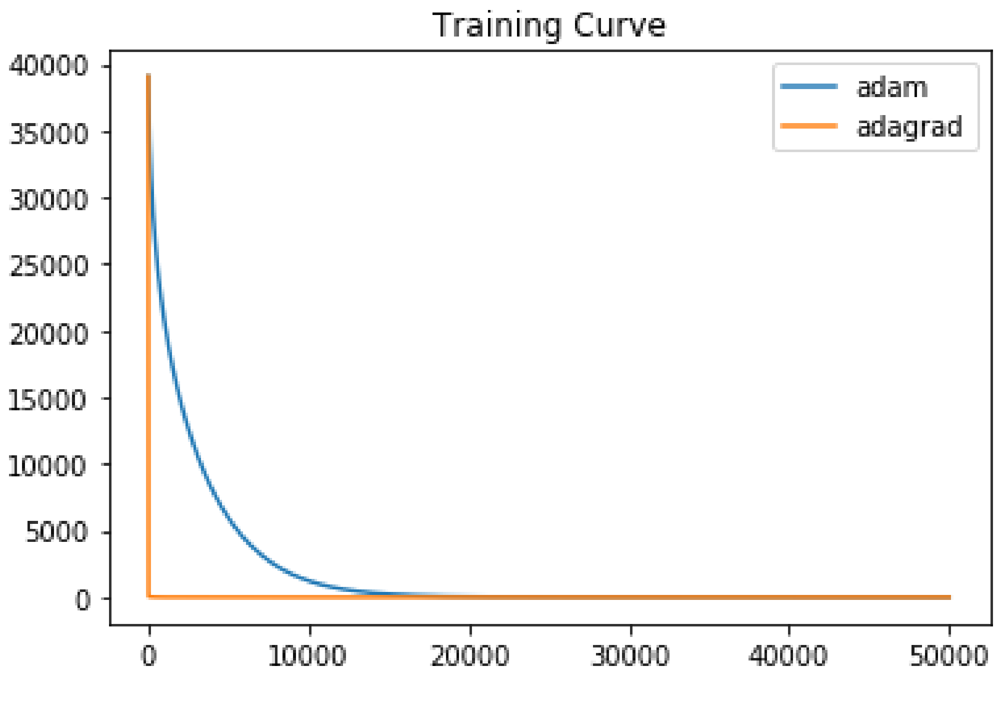

Figure 11: Comparing Training Loss for adam vs adagrad for Beale Function

3. Rosenbrock Function

- ans.set_settings(fn_name='rosen2d', method='adam', x0=np.array([8, 9]), lr=lambda x: 0.5, num_iters=500000, tol=0)

- ans.get_min_errors() yielded (2.2204460492503131e-16, 1.1093356479670479e-31)
- Finally, adam starts to shine. Looking at the contour plot, we see that the iterate initially proceeds in the opposite x_2 direction, but then it starts approaching the optimal value along the second depression and converges very well. The errors from adam are orders of magnitude lower than my previous best implementation, which was momentum with 0 tolerance, a learning rate of 0.00005 and 500000 iterations.
- The contour plot (using adam) and training loss graph (compared to the prior best method) are provided below. The hyperparameters of momentum are the same as in the case of the best implementation. The training loss plot is not particularly informative here, but is included for completeness' sake. I submitted these parameters (ie the adam implementation) for the challenge to optimise this function.

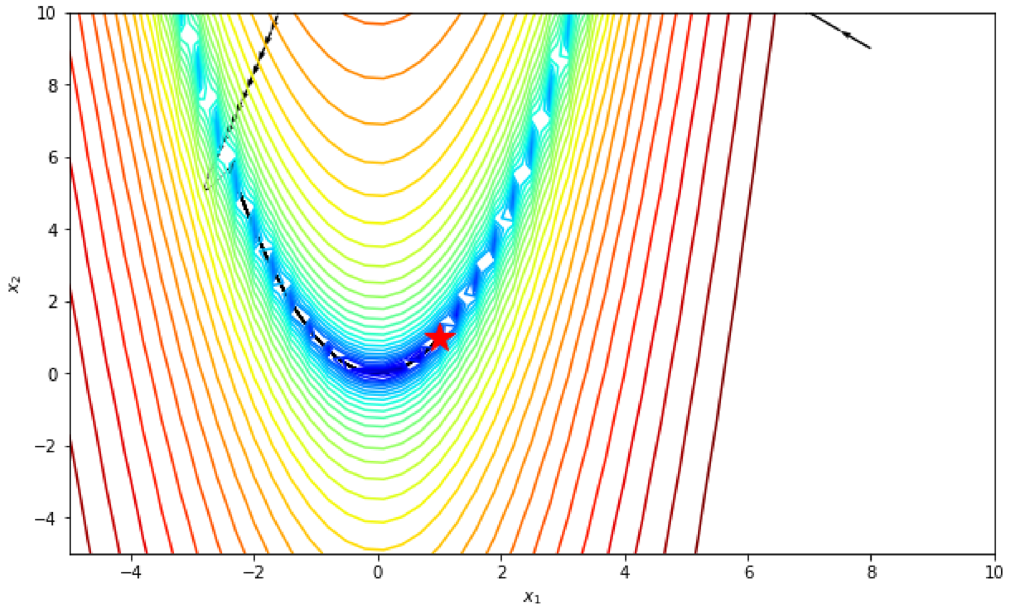

Figure 12: 2-D Contour Plot for Rosenbrock Function Optimisation with adam

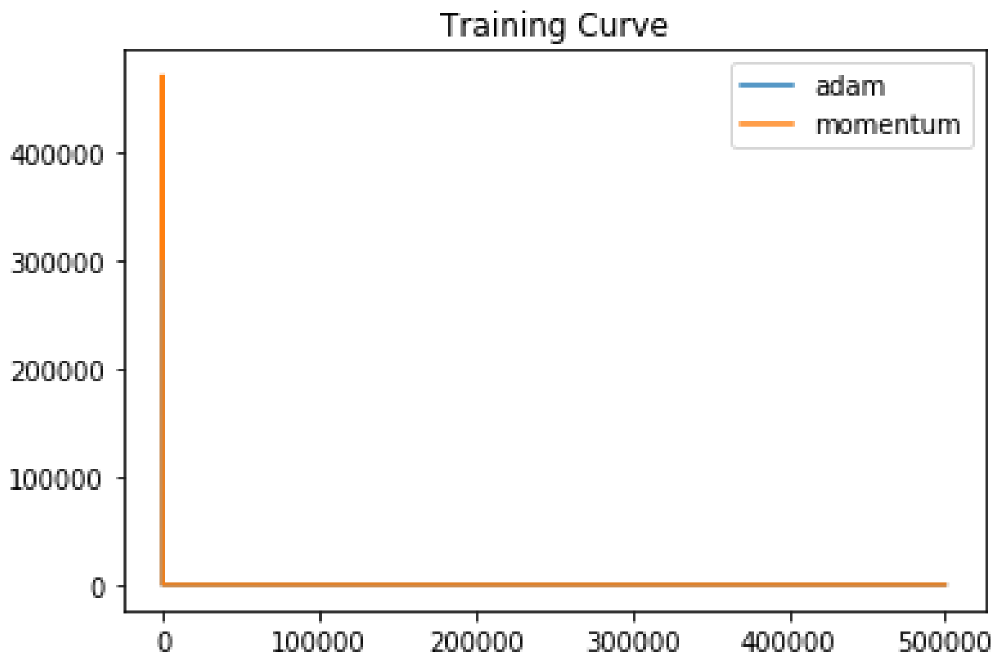

Figure 13: Comparing Training Loss for adam vs Momentum for Rosenbrock Function

4. Ackley Function

   - ans.set_settings(fn_name='ackley2d', method='adam', x0=np.array([25, 20]), lr=final_boi, num_iters=300000, tol=0). As before, please refer to notebook for learning rate scheduler (in this case final_boi) definition.
   - ans.get_min_errors() yielded (5.0878457240188813e-11, 1.4390622027349309e-10)
   - This is where the benefits of adam really come through. Looking at the contour plot, we see that the iterate tends towards the well with a stability we haven't really seen with the other optimisation algorithms. As is the case with Rosenbrock function, the errors from adam are orders of magnitude lower than my previous best implementation, which was gradient descent with 0 tolerance, learning rate scheduler lr_boi and 75000 iterations.
   - The contour plot (using adam) and training loss graph (compared to the prior best method) are provided below. The hyperparameters of gradient descent are the same as in the case of the best implementation except for the number of iterations. Observing the training loss graph, it's clear that adam is preferable due to the aforementioned lower errors and its stable behaviour. In comparison, the training loss values for gradient descent oscillate and are highly unstable in the first few thousand iterations At no point does gradient descent have lower training loss than adam, reiterating the latter's superiority. I submitted these parameters (ie the adam implementation) for the challenge to optimise this function. All I have to say is: adam? More like a-damn.

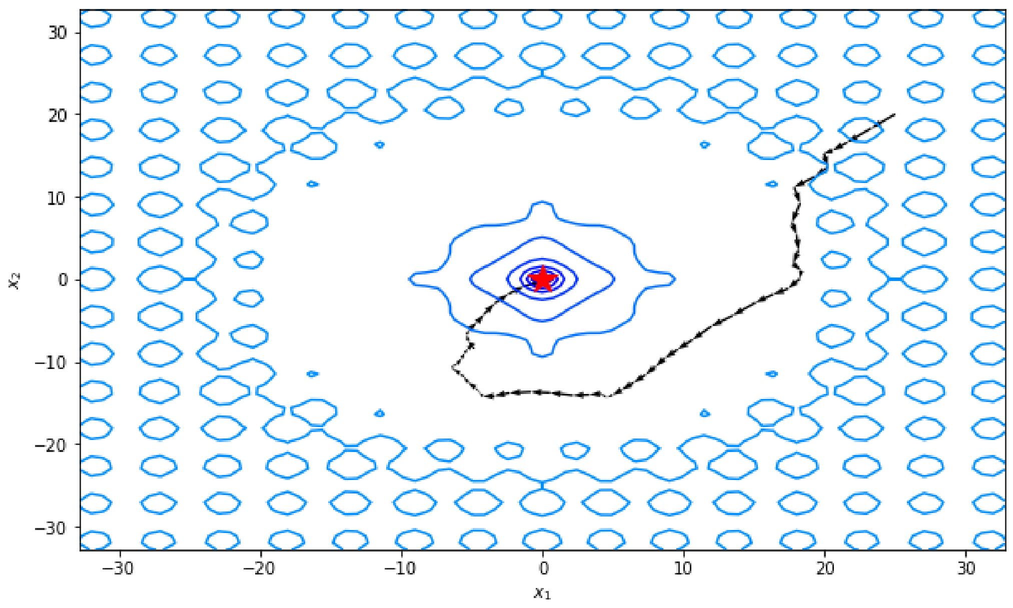

Figure 14: 2-D Contour Plot for Ackley Function Optimisation with adam

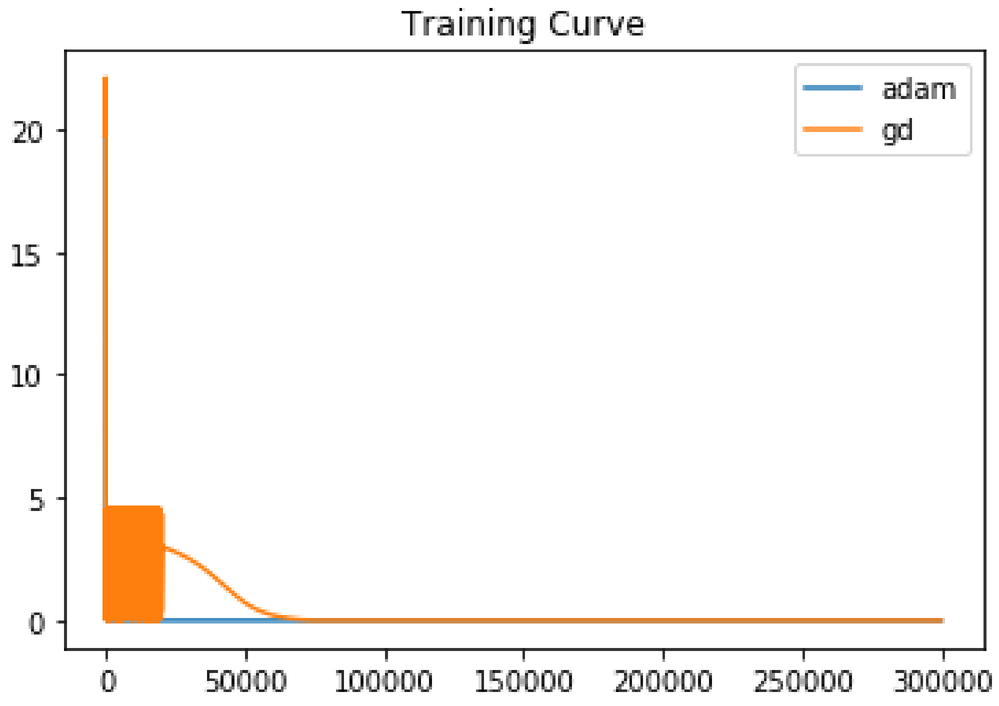

Figure 15: Comparing Training Loss for adam vs Gradient Descent for Ackley Function

5 Summary

I've already spent far too many pages talking about coming up with the best optimisation algorithms, so to conclude I'll just mention one thing about their application. Throughout this class, we've focused on optimising some function or the other, and doing our best to find the parameters that correspond to the optimal value of the objective function. This was altogether unsurprising (this is an optimisation class after all), but an interesting perspective I came across while writing this report is that in machine learning, while we may not always be able to perfectly optimise our objective function, who says we want to in the first place? After all, exactly optimising our function is just another term for overfitting. I know we handled that issue in this class by discussing regularisation and other such methods, but this was a subtle connection between theory and practice that initially slipped by me. I wasn't sure where to share this, but I couldn't submit my report without mentioning it because I found it eye-opening.

I'd like to end this report by apologising to anyone who was expecting to grade a few pages at most and then came across this monstrosity. I put a lot of effort into writing this, I can only hope it's what you were looking for. Regardless, this class was amazing, and I thank you for the incredible experience.