Natural Computing Coursework 2

University of Edinburgh, s1759855@ed.ac.uk, s1732459@ed.ac.uk

Abstract

We explore optimisation of Neural Networks with differing amounts of hidden neurons and layers using gradient-free algorithms. We see that in general, classical optimisation methods reminiscent of Stochastic Gradient Descent tend to perform better than gradient-free methods such as Particle Swarm Optimisation, Genetic Algorithms, and Genetic Programming. We see that although optima reached by gradient-free methods are not as 'good' as classical gradient-based methods, they are nevertheless good enough. That is, we reach very small errors on test sets generalising to unseen data. The main difference being time to converge.

Introduction

We will explore gradient-free optimisation methods for optimising Neural Networks (NN). Including Particle Swarm Optimisation (PSO), Genetic Algorithms (GA), and Genetic Programming (GP).

Gradient-free algorithms will then be compared to baseline NNs optimised using standard gradient-based methods (view Appendix 1 for a description of baseline). Including the Stochastic gradient descent (SGD) classical way to optimise NNs, Adam an optimiser similar to SGD, and Limited memory Broyden Fletcher Goldfarb Shanno (LBFGS) quasi-Newton method for optimisation.

Task 1

View Appendix 2 for a description of PSO.

Task 1.1

The fitness function that will be employed for PSO is Mean Squared Error (MSE). This metric calculates the average value of squared error. We can think of this as the average of differences in predictions and true values, squared. Looking closely at the formula, we see that it outlines a measure similar to euclidean distance.
Consider $\hat{y}_i$ as predicted value for i_{-th} datapoint and y_i as corresponding true value of i_{-th} datapoint, then the MSE over $n_{samples}$ is defined as:

$$\text{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2$$

This metric makes it easy to compare the quality of optima found by PSO and respective baselines models, which all use MSE as their fitness function (allowing for axes that are aligned in plots).

A downside to MSE is it is related to variance and mean, which are extremely sensitive to outliers, any difference in the true and predicted value is squared meaning that errors will increase quadratically.

To fix this we could use Mean Absolute Error, whereby instead of square, we take the absolute difference of true and predicted values. However, in some circumstances, it makes sense to give more weight to data points further from the mean. That is, being off by 10 is more than twice as bad as being off by 5. In the case of the two spiral dataset, we conclude that MSE is a more appropriate measure of error.

In practice, this is calculated and used at each step of PSO for every particle. We consider a randomly generated batch of 10 data points the two spiral distributions (5 class-1, 5 class-0). This batch is fed into (forward propagation) the respective NN, where MSE is applied as the metric to evaluate the current position (weight parameter assignment).

We only consider populations initialised with a constant number of hidden neurons, so we know the shape of the NN.

Task 1.2

The search space will be based on a NN with a constant number of neurons for each population. Since we are only considering NNs with a single hidden layer for PSO, we can easily infer the dimensionality (rank) of the search space.

E.g. if we initialise a population of NNs with 4 hidden neurons and linear and quadratic input features, (X, Y, X^2, Y^2) i.e. 4 input features. We will have 2 weight matrices of shape 4x4 and 4x2. Thus the particle's positions would be (4x4 + 4x2) 24 dimensions in (trainable-)parameter-space.

Consider the following toy example which illustrates the process, with a NN with 2 input features, a single hidden layer with 2 hidden neurons, and 2 output neurons.

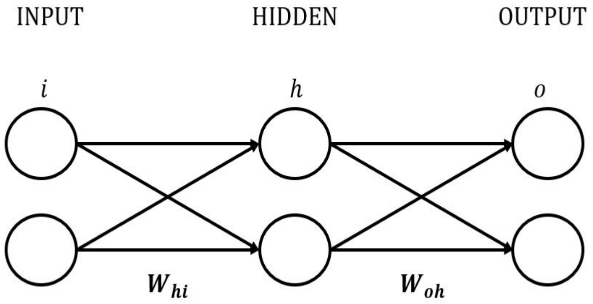

Figure 1.2.1 NN with 2 input features (i), a single hidden layer (h) with 2 hidden neurons and 2 output neurons (o) and corresponding weight matrices W_{hi} (connecting the input layer to the hidden layer) and W_{oh} (connecting the hidden layer to the output layer)

This network will have 2 sets of weights, connecting the input to the hidden layer and hidden to the output layer. These weight matrices are converted to vectors and concatenated together to obtain a 'position' in parameter-space.

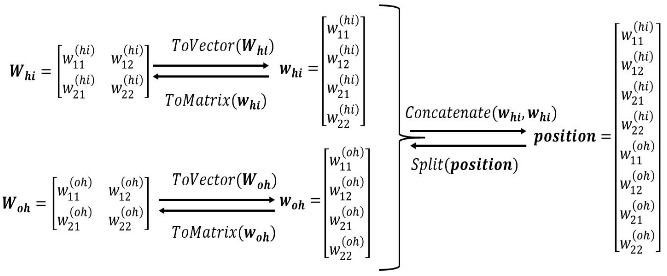
$W_{hi} = \begin{bmatrix} w_{11}^{(hi)} & w_{12}^{(hi)} \\ w_{21}^{(hi)} & w_{22}^{(hi)} \end{bmatrix}$
$ToVector(W_{hi})$
$ToMatrix(w_{hi})$
$w_{hi} = \begin{bmatrix} w_{11}^{(hi)} \\ w_{12}^{(hi)} \\ w_{21}^{(hi)} \\ w_{22}^{(hi)} \end{bmatrix}$
$W_{oh} = \begin{bmatrix} w_{11}^{(oh)} & w_{12}^{(oh)} \\ w_{21}^{(oh)} & w_{22}^{(oh)} \end{bmatrix}$
$ToVector(W_{oh})$
$ToMatrix(w_{oh})$
$w_{oh} = \begin{bmatrix} w_{11}^{(oh)} \\ w_{12}^{(oh)} \\ w_{21}^{(oh)} \\ w_{22}^{(oh)} \end{bmatrix}$
$Concatenate(w_{hi}, w_{hi})$
$Split(position)$
$position = \begin{bmatrix} w_{11}^{(hi)} \\ w_{12}^{(hi)} \\ w_{21}^{(hi)} \\ w_{22}^{(hi)} \\ w_{11}^{(oh)} \\ w_{12}^{(oh)} \\ w_{21}^{(oh)} \\ w_{22}^{(oh)} \end{bmatrix}$

Figure 1.2.2 Converting to position from weight matrices and vise versa, vectors w_{hi} and w_{oh} (weight matrices W_{hi} and W_{oh} reshaped into weight vectors) are concatenated/split to create position and original weight vectors, respectively

PSO then iteratively updated these positions (i.e. trainable parameters of the NN). At the end of each iteration, we convert the position back into 2 weight matrices, load them into a NN with the correct amount of neurons and evaluate the fitness function (MSE) with a batch of data points taken from the two spiral dataset.

Task 1.3

For testing purposes we used the PSO parameters provided to us in the tutorial: *ω=0.7, α1=α2=2.02, and population=30*, as our starting point.

Using these parameters we ran our algorithm to find the best combination of input features, Activation functions {Tanh, Sigmoid, ReLU, Identity}, and the number of neurons {4,..,8} - same as baselines.

Input Feature	Activatio n	Neu rons	Error (MSE)	Accurac y
(X, Y)	Sigm oid	8	0.3568	0.6667
(X, Y, X^2, Y^2)	Tanh	6	0.2197	0.7538
(X, Y, sinX, sinY)	Sigm oid	8	0.0985	0.8901
(X, Y, sinX, sinY, X^2, Y^2)	Sigm oid	8	0.0682	0.9356

Table 1.3.2 Best Activation function and number of neurons for PSO NN found by a grid search for each input feature.

We can see from Table 0.2 that the combination of input features of Linear+Quadratic+Sines with Sigmoid activation and 8 hidden neurons provided the least MSE of 0.0682. We used these combinations and ran various tests to find the *optimal PSO parameters* for our algorithm, which turned out to be ω=0.7298, α1=α2=2.055, and population=40 (Discussed further in Task 1.5).

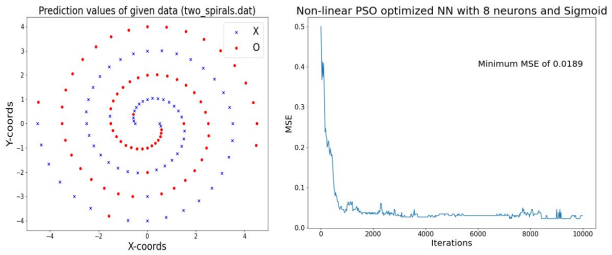

Figure 1.3.1 (Left) Plot of the test set for the optimal PSO parameters. Figure 1.3.2 (Right) Plot of Training Error (MSE) over 10,000 iterations. For optimal PSO parameters {ω=0.7298, α1=α2=2.055, n=40}.

We can see from Figure 1.3.2, training our NN using the optimal PSO parameters gave us a minimum error of 0.0189. Using this model to predict the data points provided as a test set (from NAT Learn page) we obtain Figure 1.3.1, with an accuracy of 96.59%.

Comparing this to the Linear+Quadratic+Sines baseline (Appendix 1 Table 0.1) mentioned earlier, We can see that there is only a difference of 0.0189, as the baseline value was 0 (estimated global optimum). Therefore, it is safe to say that it is possible to solve this problem for non-linear input features with only a single hidden layer by using the PSO parameters mentioned prior.

Task 1.4 - 1.5

After achieving a low test error for the input features Linear+Quadratic+Sines, we now try to solve the problem by using only linear input features. As we have found from our previous tests, Linear input features performed best with activation function sigmoid and 8 hidden neurons. So, we will start by using these and the optimal PSO parameters found above for non-linear.

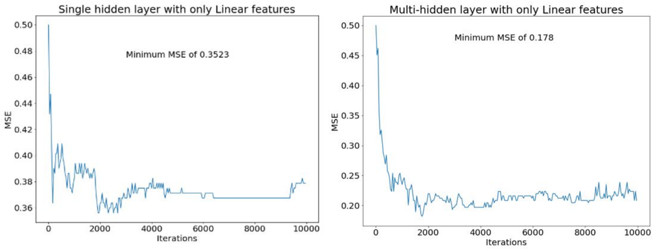

Figure 1.4.1 (Left) MSE plot for a single hidden layer with 8 neurons NN using linear input features with optimal PSO parameters (found for non-linear features). Figure 1.4.2 (Right) MSE plot for 4 hidden layer NN with shape (2,8,8,4,4,2) using linear input features with optimal PSO parameters (found for non-linear features)

From Figure 1.4.1, a single hidden layer with a shape (2,8,2) with only linear input features gives a minimum error of 0.3523. After extensive tests with PSO parameters, we are still not able to reduce this error. However, compared to the minimum error of the baseline, 0.3905, it is a slight improvement. Therefore, we believe this is the best we can do with a single hidden layer for linear input features only.

We then considered different architectures. In this case, a varying number of hidden layers. We started by adding a hidden layer with 4 neurons (i.e. shape is now {i,{2,8,4,2}o}) and that decreased our error by 0.05, from 0.3523 to 0.2853 (19% drop in error).

This shows that making the NN deeper yields performance boosts. We gradually increased the number of hidden layers and found an optimum NN; Figure 1.4.2 shows our results, we achieved a 0.1780 test error using shape {2,{8,8,4,4},2}. A significant improvement from 0.3523 (50.1% drop in error compared to the optimal single hidden layer NN). Regardless, we found that making the NN increasingly deep does not further reduce the error, even for very high iterations.

We conclude, that with only linear input features, a multilayer NN significantly outperforms a single hidden layer NN on the two spiral classification problem. Conversely, although multilayer NNs (through extensive testing) beat single layer NNs. When we introduce non-linear input features (quadratics/sines), single layer NNs perform better (Figure 1.3.2).

Our PSO algorithm has 4 parameters that we can control: Inertia (ω), attraction to personal best (α1), attraction to global best (α2), and population size.

We vary values of ω from {0.1, ..0.9}. We found that for 0< ω<0.4, we get an accuracy of less than 10% on average and plotting the swarm showed us that the swarm was more exploitative, locally oriented search, and caused premature convergence to a local optimum. For ω>0.8 we found that the swarm is more exploratory with an accuracy score of ~55%, which is similar to randomly guessing. However, the swarm is now less exploitative and more unstable. We found our algorithm produces the best results for 0.65<ω<0.75 with an average error of less than 0.2 with an accuracy of over 75%. Through a grid-search, we were able to find that ω=0.7298 works best with an accuracy of 95% and an error of less than 0.07 on average.

The parameters α1 and α2, cognitive and social components of the particle respectively, were initially tested independently. For α1=0, the swarm was able to converge to a solution much quicker however, in many of our tests the swarm prematurely converged to a local optimum. When

$\alpha2=0$, our swarm was not able to converge as the particles all acted independently and were spread out, due to not having any social component. We did not look into the negative values of $\alpha1$ and $\alpha2$ as we know that particles in the swarm will repel each other instead of attracting.

When only $\alpha2=0$, particles had no social ability and only had their cognitive component, and this caused those particles to act independently with no information being passed between each other. After plotting the swarm, we saw that the particles were all spread out around their own personal best and were not able to find the optima. We then followed the claims from Kennedy et al. that $\alpha1+\alpha2=4$ and $\alpha1=\alpha2$, give good results [1], and tested for $\alpha1=\{0.5,0.55,..,2.3\}$ and found out that the range $2<\alpha<2.1$ gives us the best results with an accuracy of 80% and error of less than 0.2 on average. We carried out further tests to find that $\alpha1=\alpha2=2.055$ gives us the most consistent solution with an error of less than 0.10 and an accuracy of ~88%.

As for population, we tested out the range 20-200 given in the lectures for an interval of 5. We found out that the higher the swarm size, the search performed by the swarm was more scattered. However, this also increased the computation time but had very little effect on the result. So, we took the best performing population in 5 runs, which turned out to be 40. We noticed that a low population causes the particles to converge quicker than a high population.

Task 2

GA is a gradient-free optimisation method that mimics natural selection, imitating the biological processes of selection, inheritance, crossover, and mutation. We propose a way to finetune the network structure of NNs (number of hidden layers and neurons).

Task 2.1 - 2.2

We find the optimal NN structure is (6, [4, 6, 6, 5], 2) which reached a test accuracy of 96.2%.

We will evolve the structure of the NN but we leave the assignments of the individuals' weights down to LBFGS (optimisation based on gradient-descent, see Appendix 1 for further details). Also, we opt to pre-specify hyperparameters (activation, batch size, learning rate, etc.) as our baseline provides an estimate of optimum values to use.

Similar to baselines, we consider all features. DNA encoding of the structure of each NN in our implementation of GA will be as follows; Encoding is a set of numbers specifying how many neurons per hidden layer, e.g. [2] corresponds to a NN in Figure 2.1.1. We are evolving the intermediate layers of our NN.

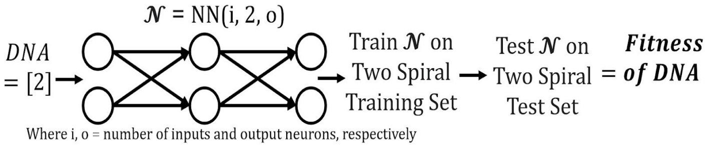
$\mathcal{N}$ = NN(i, 2, o)
DNA = [2]
Train $\mathcal{N}$ on Two Spiral Training Set
Test $\mathcal{N}$ on Two Spiral Test Set
= Fitness of DNA
Where i, o = number of inputs and output neurons, respectively

Figure 2.1.1 Converting DNA encoding of NN structure and evaluating fitness given DNA (we assume DNA has gone through selection, inheritance, crossover, and mutation before this process being applied)

We train the NNs via classical gradient-based optimisation (LBFGS). We opt for this over PSO since we tend to reach better optima using the prior, this can be seen in our baseline (Table 0.1) and PSO (Table 1.3.1) tables.

We use a random seed to generate a weight assignment for each layer. The seed determines the random numbers generated, this is done so that we can obtain reproducible results across multiple function calls. We use a random seed of 0.

Task 2.3

GA has parameters population size, mutation rate, replacement from selection, and crossover points. Similar to PSO, we used parameters given in tutorials as a starting point. We ran GA for 100 generations with population size 10, crossover probability 0.7, and mutation probability 0.01. This combination of parameters gave us the best results; Similar to PSO we performed a grid search to find optimal parameters and compared them with parameters mentioned earlier. We found the higher the population size, the more accurate the result.

However, the time to converge significantly increased.

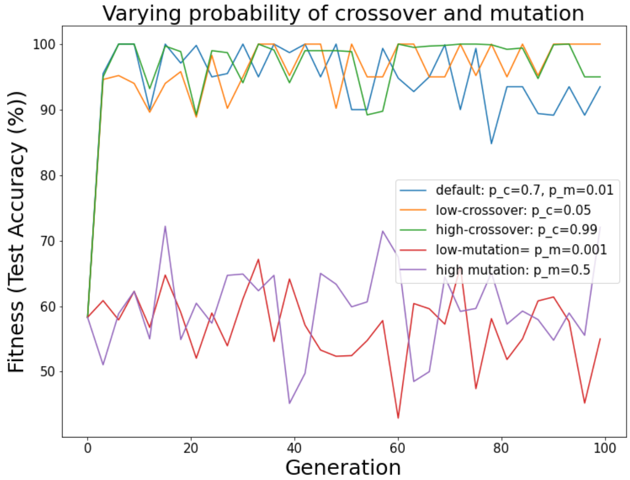

Figure 2.3.1 Varying crossover probability vs average population fitness using the number of layers found by best individual (4 HLs)

When mutation-probability is low, there is less genetic diversity, thus the population does not change, this causes the population to be trapped at a bad solution. Whereas, when mutation probability is high, we have high genetic diversity however, at that point GA behaves more like a random search. As for crossover, we found out that it had very little effect on the solution however, the generations needed to find a solution was the only factor affected. The low crossover probability needs more generations than the high-crossover probability to reach the solution.

Finally, for selection, we used the roulette wheel selection. We realised this had very little effect on our algorithm as the population size we used was small. However, testing for a higher population, we were able to see that roulette wheel selection has both negative and positive effects; It can cause a stagnant solution to improve or cause a good solution to perform worse.

Task 2.4

We control the complexity of the evolving network, by not using elitism. Meaning the complexity/runtime of the network is slow due to the fact our implementation does not use elitism. This means that when we do the selection, we are using the 'roulette wheel' to randomly select members of the population to reproduce. We have chosen to do this as it ensures high genetic diversity (useful in high dimensional search spaces [1]). GA will converge slowly and not all members of the population will have max fitness. However, the max fitness individual will propagate and

Adding elitism to GA would not change runtime although it would increase time to converge (given sufficient genetic diversity i.e. mutation rate). We opted not to use elitism as we want the largest amount of genetic diversity possible for this problem and compute power was not an issue.

The best individual found using GA was {6,{4,6,6,5},2}. This means we used a maximum of 6 input features and 2 output neurons, with the intermediate layers consisting of 4, 6, 6, 5 neurons.

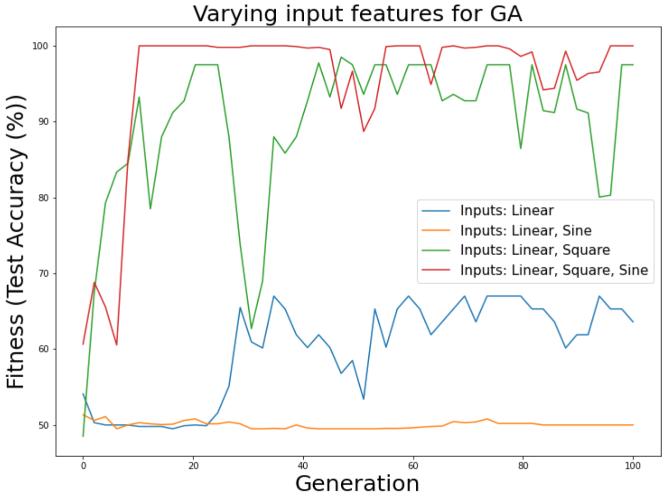

Figure 2.4.1 Varying input features vs average population fitness using the number of layers found by best individual (4 HLs)

Task 3

Unlike GA, which is encoded as binary/integer strings, GP is encoded as m-ary trees (where m is the number of children for each node). Each node in this tree represents a hidden layer from GA which can contain any number of neurons and have its activation function. Our algorithm considers a choice of activation for the nodes from the list: {Tanh, Sigmoid, ReLu}. Our GP took in an extra parameter this type, learning rate, which can vary from {0,1}. Unfortunately, our code for GP was not able to perform how we wanted it to and we are only able to get a maximum accuracy of 60% with an MSE of 0.24, which is far below any of the results we achieved for the previous two tasks.

Each node in the tree can be considered as a NN with its structure and weight assignment, with a node branching into two nodes (to create its children). When the node creates new children, mutations are applied and we can do crossover.

The depth of the tree indicates the level of genetic diversity in the population so far. If the average depth of the tree is shallow, it either indicates a good solution has been found (and thus the population is stagnant due to the fact we've reached a good optimum) or it means there was not enough genetic diversity and the population is stuck.

As mentioned prior GP is a special case for GA where each node is allowed to have two children. A Tree indicates a lineage for a given ancestor within the population.

GP operators that we considered were crossover and mutation [2]. For GP the subtree of a node is dependent on the node. An example would be if we decide to crossover one node, we will need to crossover the entire subtree of that node as well. The following is an example of crossover with subtrees.

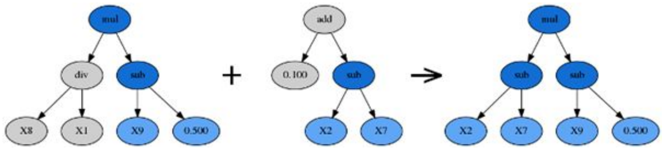

Figure 3.2.1 Example of how crossover takes place for GP

A type of mutation that can be considered is Hoist mutation. It selects a node, through a selection process be it roulette wheel or tournament, and selects a subtree of that node randomly and is 'hoisted' (taken). This means that only that subtree is used for that node.

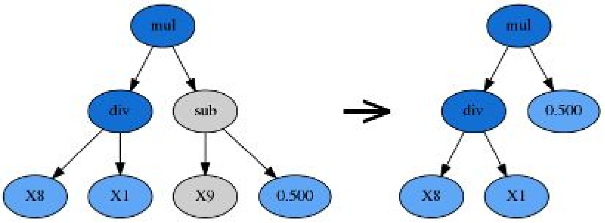

Figure 3.2.2 Example of how during a hoist mutation only a subtree of a node is used.

Another type of mutation that we can use is a point mutation, this is when a node is swapped with another node but the subtrees of those nodes are not swapped.

We speculate that GP would do slightly worse than GA (but better than PSO), as we are restricting the number of children each member of the population can have. As well as, trying to keep lineages intact which could be counterproductive when new shallower (but better) are found.

Future work

For PSO one thing we could have investigated would be to consider a constriction factor to accelerate the convergence of the swarm and reduce the computational time. As for GA, we could have implemented 'elitism' to our code for a faster convergence as we choose not to as it reduced the genetic diversity. Another thing we could have tested would be the robustness of the algorithms by adding noise to our data. Finally, for GP we could have tested out different mutations such as, Create, Remove and shrink and check the effects such as time taken to converge or robustness when combined with the existing ones.

We see that these gradient-free methods of optimisation are good but the time for convergence is extremely slow compared to standard SGD. In the future we would like to gain insights into how these gradient-free methods can be used to optimise aspects of a neural network which do not have gradients/derivatives associated with them (i.e. don't optimise the error function), but rather looking at more difficult problems within NNs for example, where to put special layers (e.g. Batch Normalisation layers) or how to set optimal hyperparameters using these methods.

References

[1] J. Kennedy et al. (2007) Particle swarm optimisation

[2] Trevor Stephens et al. (2016) Genetic Programming in Python with SKLearn inspired API(https://gplearn.readthedocs.io/en/stable/intro.htm)

[3] Scikit-Learn (2020) Machine Learning in Python (https://scikit-learn.org/)

[4] R. Tagore (2017) Solving the Two Spirals problem with Keras (https://glowingpython.blogspot.com/2017/04/solving-two-spirals-problem-with-keras.html)

[5] Q. Suire (2019) PSO-ANN (https://github.com/kuhess/pso-ann)

[6] W. Ahmed (2020) Cartesian genetic programming neural networks (https://github.com/wiqaaas/youtube/tree/master/Machine_Learning_from_Scratch/Cartesian_Genetic_Programming_Neural_Networks)

Appendix

Appendix Note 1: Baselines

Using the classical gradient-based optimisation methods mentioned prior. All of which are some sort of variant on SGD (i.e. hill descending algorithm). We will create baseline models for each task. The reason we have chosen to do this is to compare the optima found by each algorithm to its respective baseline NN with the corresponding number of neurons and layers.

This is done because, although we can (almost) never know what the true global optima for our fitness/error functions are, the classical NNs provide an estimate of the global optima. Thus, it makes it easy for us to evaluate the quality of the optima found. Baseline models find their hyperparameters via grid search; an exhaustive search over a set number of neurons (4, 5, 6, 7, 8), optimisers *(LBFGS, SGD, Adam),* and activation functions (Identity, Sigmoid, Tanh, ReLU).

We only train the baseline models for 2 iterations, that is 2 steps in the parameter-space to minimise error, if we train for more iterations we tend to reach 'perfection'. That is accuracy on the test set of 100%. Also, we created a two spiral generator function (provided by the TensorFlow playground) that creates random points from the two spiral distributions. This Generator function [4] was slightly modified to allow for the creation of extra input features (quadratics and sines).

All baseline models are trained and tested on 2000 data points with a 50-50 split. We use Sklearn's Multilayer-perceptron (MLP) [3] with default settings, *(Note: assume SKlearn's default settings unless stated otherwise)* a batch size of 200, and a learning rate of 0.001. We do not consider bias (linearly translate/shift decision boundaries) terms within any of our NNs as all data is centered at the origin and thus is not needed.

The following table showcases the best NN obtained via grid-search and its metrics. These will be compared to different gradient-free algorithms to assess the quality of the optima found (by comparing MSE).

Input Feature	Activation	Optimiser	Neurons	Test Accuracy	Error (MSE)
Linear (X, Y)	ReLU	LBFGS	6	0.6095	0.3905
Linear+Quadratics (X, Y, X^2, Y^2)	Identity	LBFGS	4	0.9460	0.0540
Linear+Sines (X, Y, sinX, sinY)	Sigmoid	LBFGS	8	0.5750	0.4250
Linear+Quadratics+Sines ($X, Y, \sin X, \sin Y, X^2, Y^2$)	Identity	LBFGS	4	1.0000	0.0000

Table 0.1 Best baseline NNs optimised using best performing optimiser (SGD, LBFGS, Adam) found by a grid-search for each input feature

We can see from Table 0.1, that linear features (X, Y) only let us achieve just above 60% accuracy on the two spiral datasets. This is ~10% better than classifying at random (since we adjust sample sizes for each class).

Looking on, we see that adding quadratic terms in conjunction with linear features (X, Y, X^2, Y^2) seems to increase the performance of the model to near perfection. With an increase of ~34% compared to just linear features. This suggests that quadratic terms contain very useful information for classification. The NN only requires 4 hidden neurons to achieve this result, further suggesting that quadratic terms are very useful. Furthermore, the best performing activation function is the Identity i.e. Linear f(x)=x. Meaning that we just have a linear combination of our weights and input. That is, in 4-dimensions (X, Y, X^2, Y^2) a hyperplane can be found to separate one spiral from another - In simple terms, the model can separate these inputs with a line.

Contrast this with adding sine features (X, Y, sinX, sinY) that do not seem to add any performance benefit. Instead, slightly hindering the performance of the model, this is odd. If we think of this input feature as a position in input-space, the decrease in performance could be due to the values of the sine terms only ranging from [-1, 1] thus not changing the position.

Finally, when we add quadratic and sine terms together (X, Y, sinX, sinY, X^2, Y^2) we manage to reach a perfect NN in just 2 iterations and 4 hidden neurons.

Appendix 2: PSO Description Task 1

PSO is an algorithm inspired by how flocks of birds travel together or how shoals of fish avoid predators. It is a stochastic population-based optimisation method.

In this case, PSO creates a population/swarm of particles each initialised with random positions (which correspond with different weight assignments in parameter-space, for a NN with a constant number of neurons). The potential solutions which have an associated position, called particles, 'fly' through the problem space (weight parameter-space) by following the current optimum particles.

Each particle keeps track of its position in the weight parameter-space and a series of 'best' solutions. That is, each particle keeps track of where it achieved (its own) best solution, called the personal best, $B_{Personal}$. The local best solution found in its topographical neighborhood (i.e. other particles that are 'close' in the search space), known as the local best, B_{Local}. The best global solution found by all particles in the population, called B_{Global}.

In PSO at each time step, changing the velocity of (accelerating) each particle toward its $B_{Personal}$ and B_{Local} locations. Acceleration is weighted by a constant, with separate random numbers being generated for acceleration toward $B_{Personal}$ and B_{Local} locations.

However, our implementation of PSO does not consider attraction to the local best solution, B_{Local}. PSO's behavior depends on the parameters α1 (attraction to $B_{Personal}$), α2 (attraction to B_{Global}), (and α3 if we *were* considering B_{Local}), ω (weight term for the portion of the previous velocity at iteration t, to be considered in t+1) and population size.

The particles are updated in such a way to enforce separation (avoid collision with neighboring positions), alignment (match the velocity of neighboring position), and cohesion (stay near neighboring position).

Appendix 3: Our Code

We created a GitHub containing all of our code which will be made public on the day of submission (03/12/2020 4 pm)

This GitHub can be found at:https://github.com/Sakib56/NAT-CW2