COMP3330 HWA1 Report

Evan Gresham c3196094

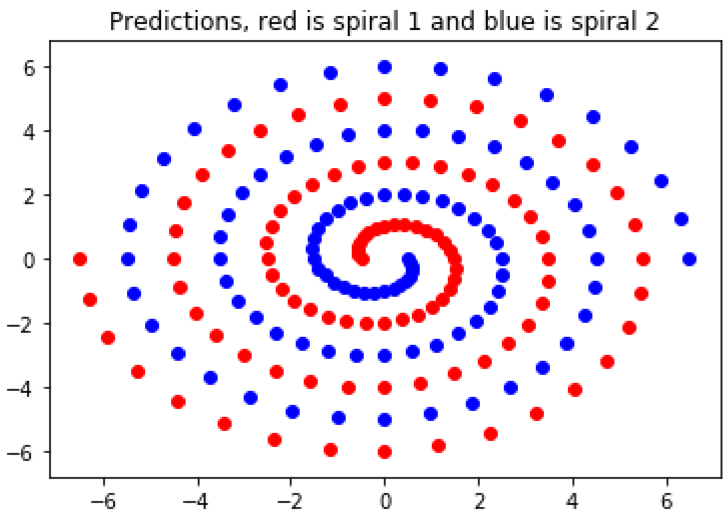
Jordan Haigh c3256730

Sebastian Wallman c3162759

Question 1 – Variations of the Two Spiral Task

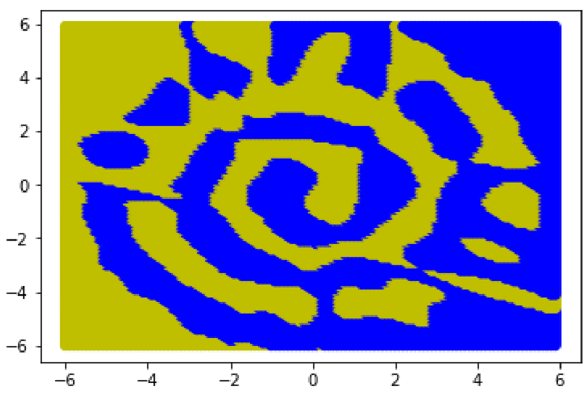
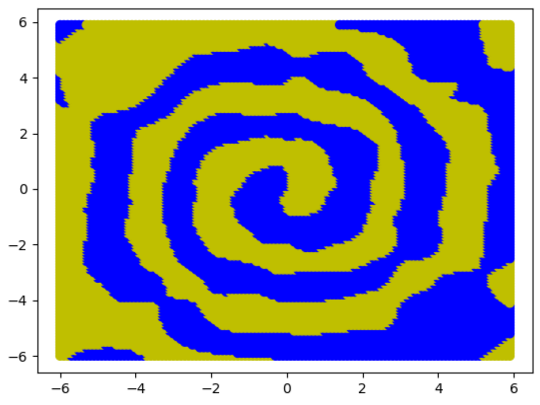
a) Start with original dataset with 194 training points. How fast and well can you solve this task using a feed forward NN? The (x,y) coordinates of the points in data is supplied on blackboard.

To start this problem, the code examples from the TensorFlow lab sessions were reviewed for reference. This code gave us a starting point for using the libraries and how to apply them to our problem (analysing and distinguishing the spirals). To read the two-spiral dataset, it involved appending coordinates in an alternating fashion for each spiral and allocating a label for each respective spiral.



With the data read and labels applied, the number of nodes and number of hidden layers could start to be tweaked for the best layout. It was determined that adding two additional inputs, sin(x0) and sin(x1) would help in classifying the two spirals and complete the task in a low number of epochs. The final setup involved two hidden layers, each containing 40 neurons.

With this setup, the session containing 4 inputs was completed in less than 5000 epochs. Without using these two additional inputs, a session could take up to 50000 epochs, generating a worse result.

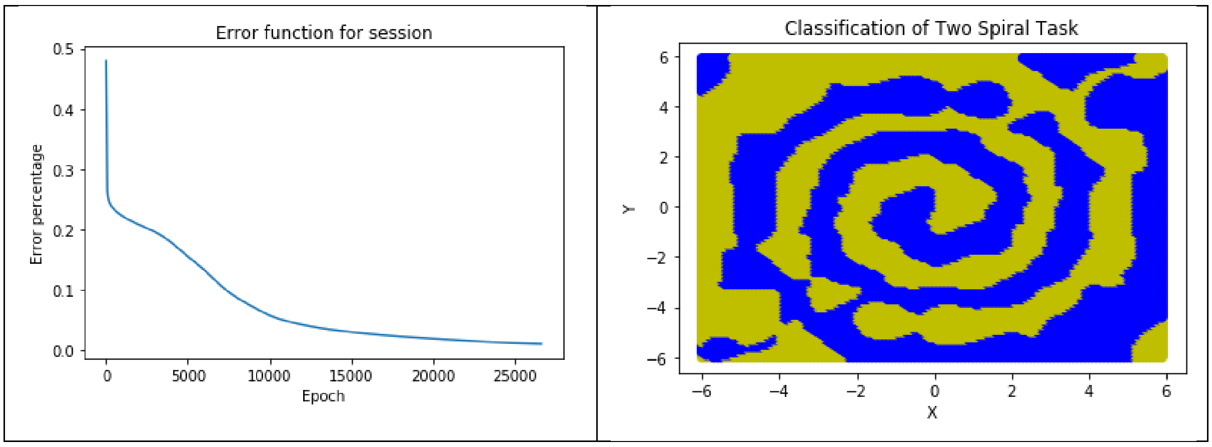
	
50000 epochs without using sine inputs	5000 epochs using sine inputs

Adding sine inputs generated a much smoother classification of the two spirals.

To improve on the functioning two-spiral classifier, learning rates, hidden layers and neurons were changed to find better session runtimes.

Each session was allowed a maximum of 50000 epochs, though if it could converge below 0.01 before the epoch limit, the session would finish at the current epoch.

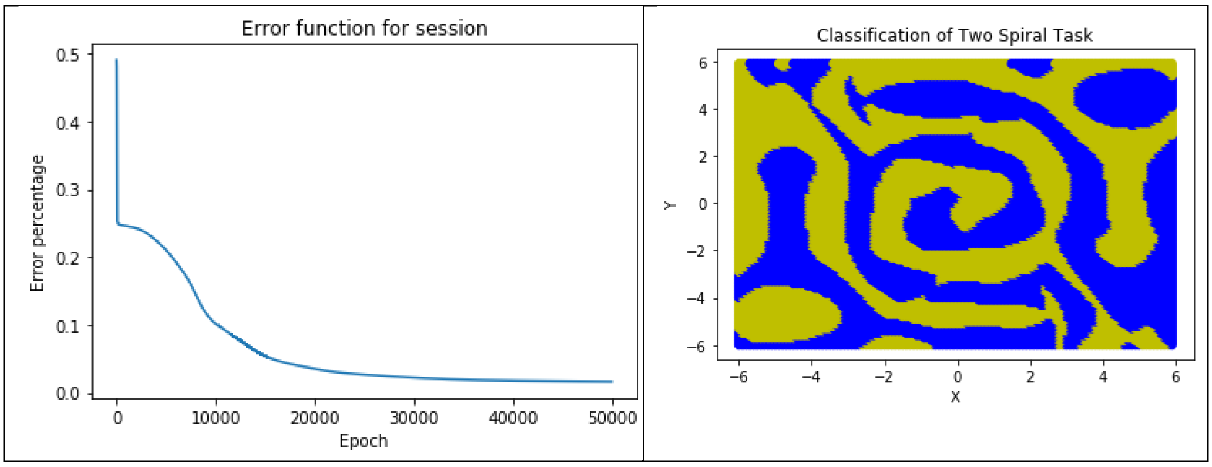
Session 1: Reduced number of Hidden Layers (1 Hidden Layer – 40 Neurons)



Session converged below 0.01 at Epoch #26608. The error count was 0.009999903.

Though the session converged, it took longer than anticipated. The spiral figure was not as smooth as hoped for, as well as incorrect classification in areas.

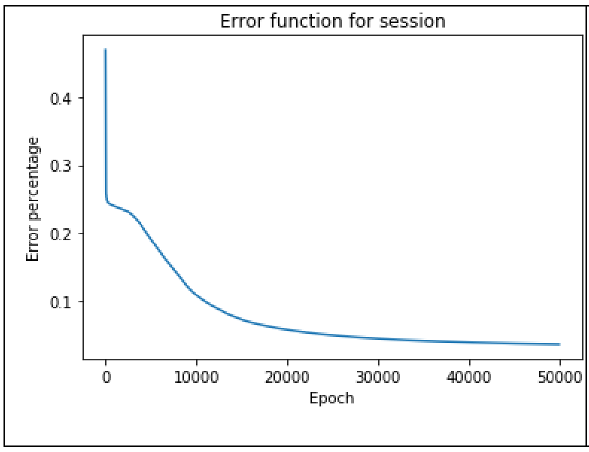
Session 2: Reduced number of Neurons (2 Hidden Layers - 10 Neurons each)

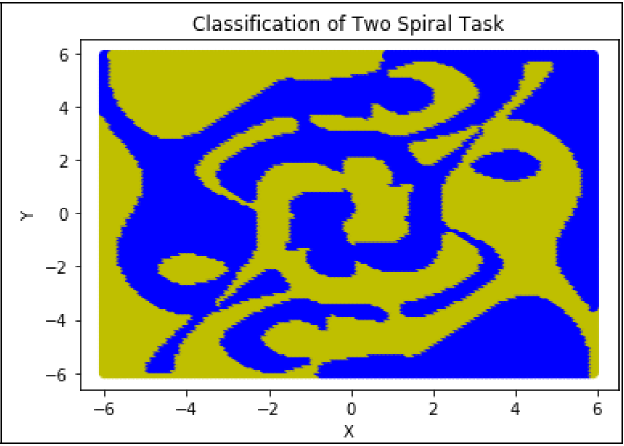


Session did not converge within 50000 epochs. Error count at Epoch #50000 was 0.01596638.

It was hoped that this setup would work. After using the "playground.tensorflow.com" website, an example session converged within 1000 epochs using a similar setup. This could be due to using different parameters. With the session that was ran, it was still able to generate a mostly smooth distinguished spiral, though errors still exist.

Session 3: Reduced number of Hidden Layers and Neurons (1 Hidden Layer - 10 Neurons)

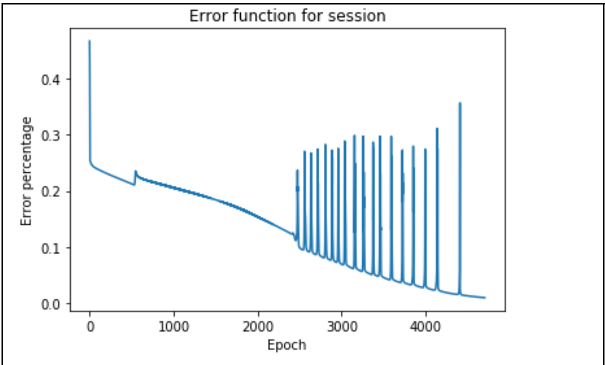


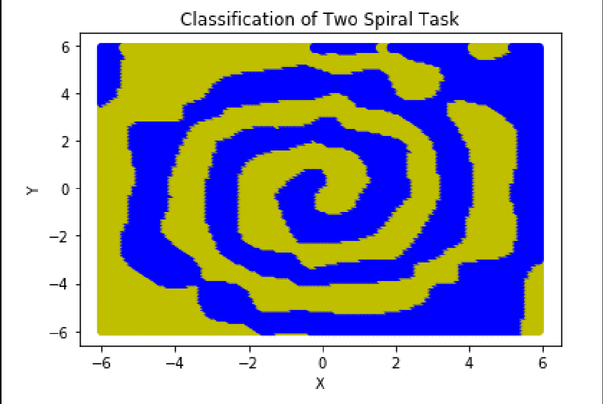


Session did not converge within 50000 epochs. Error count at Epoch #50000 was 0.036558725.

After the results of Session 2, it was not expected that this session would do well. If Session 2 was not able to converge within 50000 epochs with 2 hidden layers of 10 neurons, it would be a similar fate for this session. Regardless, the session attempted to distinguish the spirals, though the error count was still too high.

Session 4: Increased number of Hidden Layers (4 Hidden Layers - 40 Neurons)

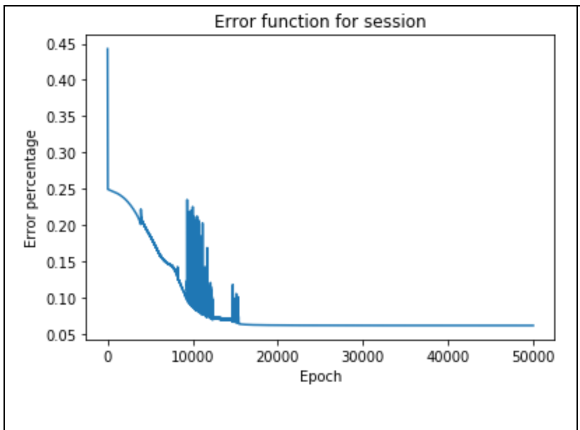


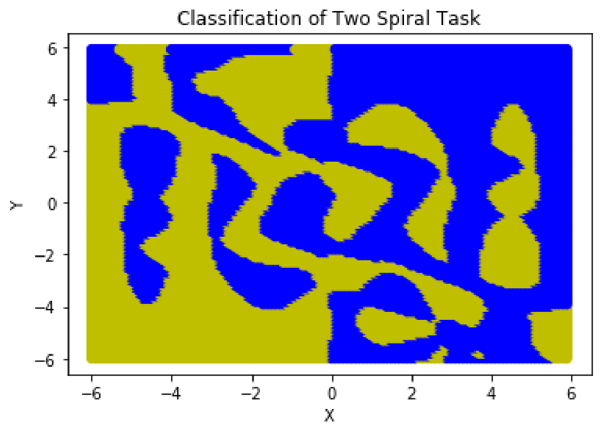


Session converged below 0.01 at Epoch #4706. The error count was 0.009989062.

This session proved to work quite well when using 4 hidden layers. Though it was much slower than all but one other sessions, it generated an acceptable distinguishing spiral. The error function showed the most volatility out of all sessions.

Session 4: Increased number of Hidden Layers, reduced number of Neurons (4 Hidden Layers - 10 Neurons)

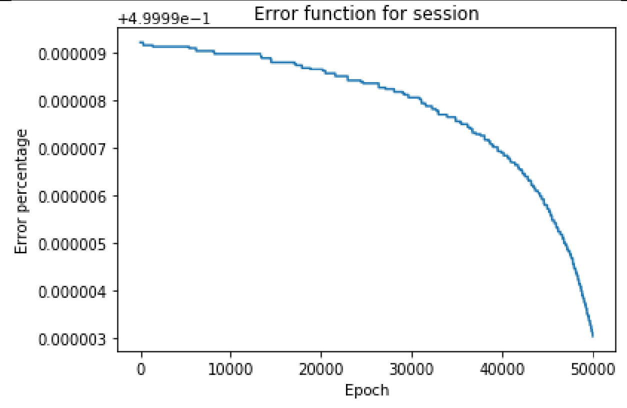


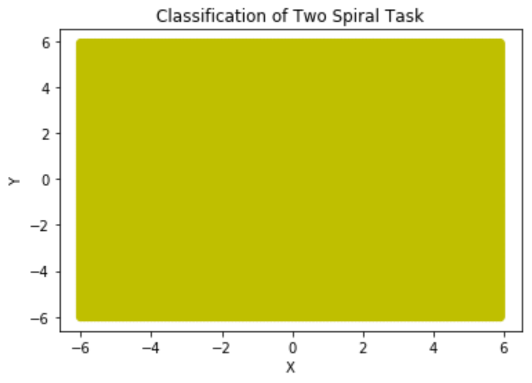


Session did not converge within 50000 epochs. Error count at Epoch #50000 was 0.06190094.

This session started strong with a steep learning curve in the initial 5000 epochs, though had a lot of issues from Epoch #10000 -20000. It eventually slowed to a crawl and was not able to converge within the allocated number of epochs.

Session 5: Increased number of Neurons (2 Hidden Layers - 60 Neurons)



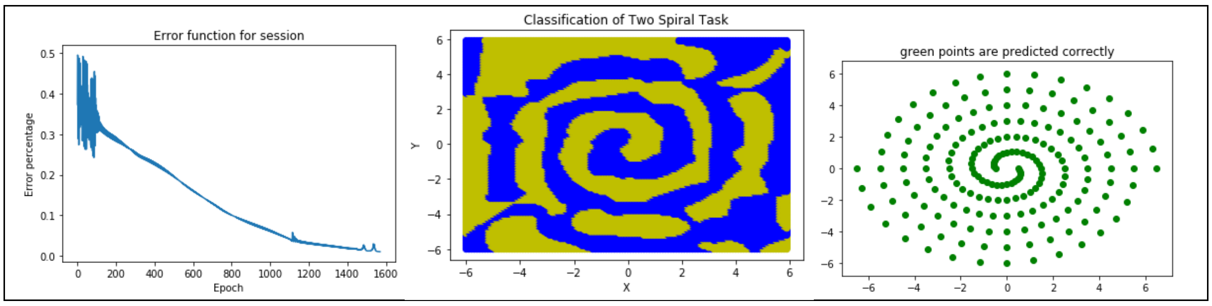


Session did not converge within 50000 epochs. Error count at Epoch #50000 was 0.49999383.

This session showed hope under the belief that increasing the number of neurons would result in a higher accuracy generated in less time. This setup was the slowest session and provided no improvement at all. This session was a waste of resources.

After exploring alternative layouts, the original layout of 2 Hidden Layers of 40 neurons each was superior for accuracy and speed. Learning rates can now be altered to see if changing the value would provide a faster/accurate result.

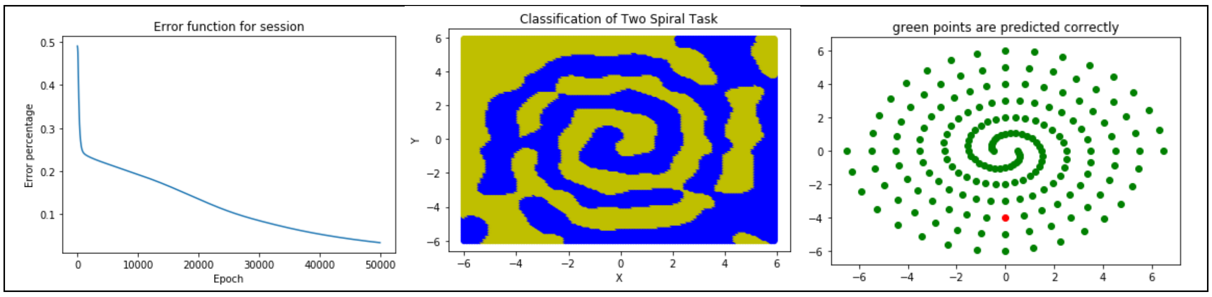
Session 6: Higher Learning Rate (Learning Rate = 5)



Session converged below 0.01 at Epoch #1569. The error count was 0.009996394.

It was surprising to see this finish so quickly, though after reviewing the classification, it was not as distinguishable as hoped. Note the large fluctuations within the first few hundred epochs.

Session 7: Lower Learning Rate (Learning Rate = 0.05)



Session did not converge within 50000 epochs. Error count at Epoch #50000 was 0.03549712.

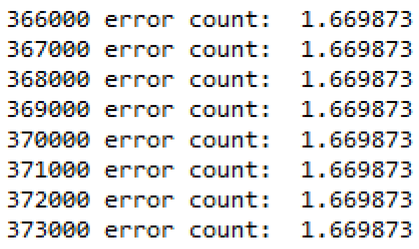
With a lower learning rate, it was hoped that this session would create a smoother spiral. Though it achieved this, there are sections were the two spirals have not yet been distinguished.

b) Generate your own variation of the 2 Spiral Task. Solve associated classification using ANNs and discuss your approach in comparison to Question 1A

A dataset containing four spirals was created, each spiral containing 50 coordinates. The reason behind using a lower number of points is so that the ANN did not take too long classifying all the points correctly. If there were a larger number of coordinates, it may have taken too long to converge. A new python script was generated, this time including the '*pandas'* data science library. Rather than using our own CSV reader, the '*pandas'* library comes with a method that is able to read an entire file and store it into a *DataFrame*. This allowed a simpler approach to analysing the data.

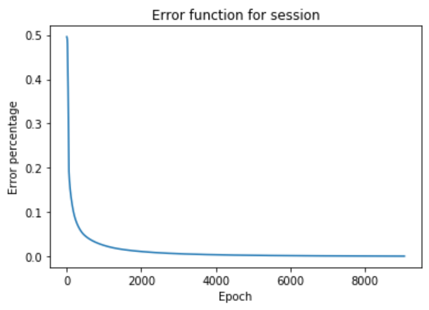
Using the same setup for TensorFlow as Question 1A, the session commenced. The error count started from roughly 4.0, but could not converge below 3.66. It was decided to move on from this dataset and find simpler variations of the 2 -spiral task.

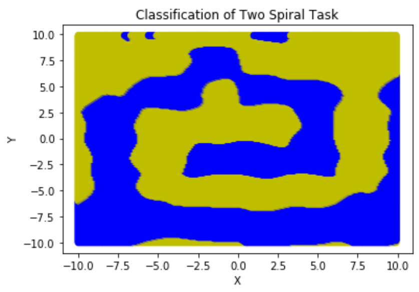
A dataset containing three spirals was created, this time each spiral contained 100 coordinates. When inputting this file into the current neural network setup, it was not able to converge within 50000 epochs. To try and brute force the session to converge, total number of epochs was removed and could run infinitely. This was unsuccessful after leaving it for 15 minutes (370000 epochs and stuck at an error count of 1.669873). There was an attempt to tweak variables (number of neurons, hidden layers, learning rate) but to no avail. Whilst it was not able to distinguish these three spirals, a Support vector machine could certainly solve the problem.

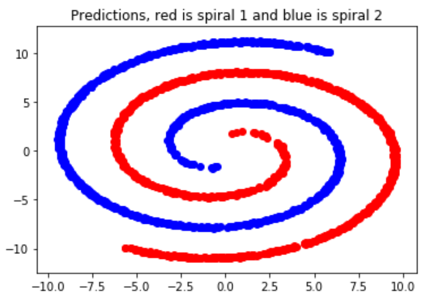

```
366000 error count:  1.669873
367000 error count:  1.669873
368000 error count:  1.669873
369000 error count:  1.669873
370000 error count:  1.669873
371000 error count:  1.669873
372000 error count:  1.669873
373000 error count:  1.669873
```

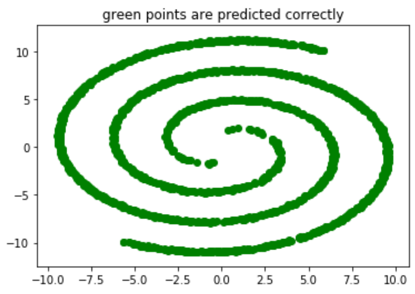
No sign of convergence with an artificial neural network on a three-spiral dataset

Other variations included creating another 2-spiral dataset that was larger and rotated by 90 degrees. Each spiral contained 1000 points each. Because of this, the epochs took longer to run, though the ANN was still able to classify the two spirals correctly. As there were more data points to work with, we lowered the error percentage to 0.001% for a more accurate result. After reviewing question 1A, confidence was high that the Neural Network would be able to solve this 2-spiral dataset.


Error function for session
Error percentage
Epoch

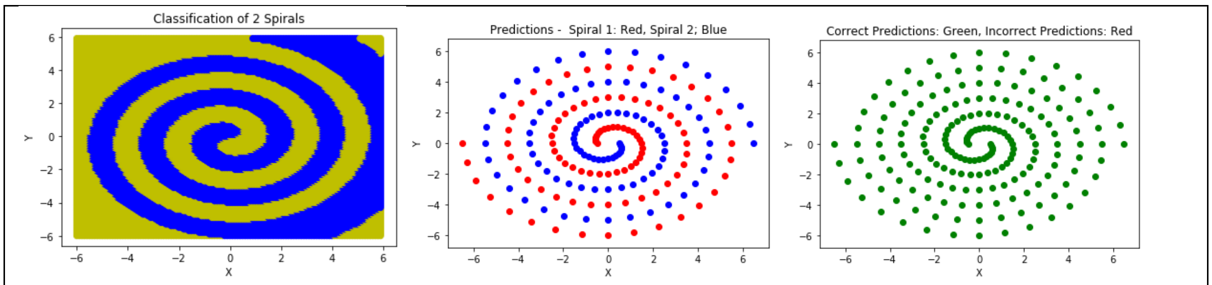

Classification of Two Spiral Task
Y
X


Predictions, red is spiral 1 and blue is spiral 2


green points are predicted correctly

c) Compare ANN and SVM

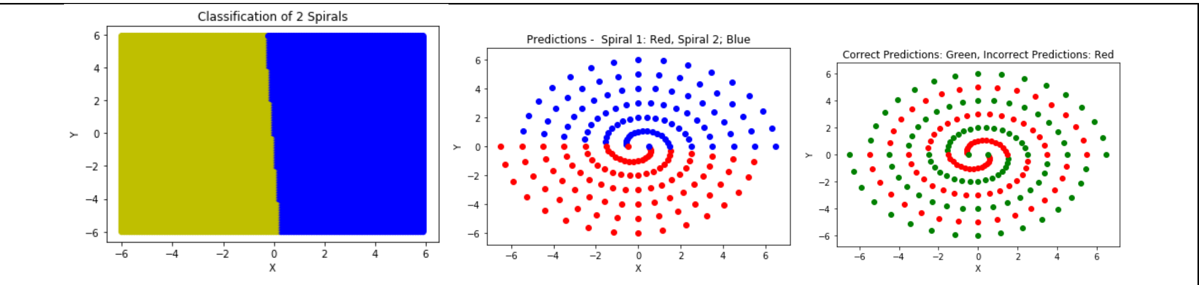
We referred to the SVM examples from the workshops as a starting point. This involved us having to implement our file reader to read the dataset (as there were alternating lines).



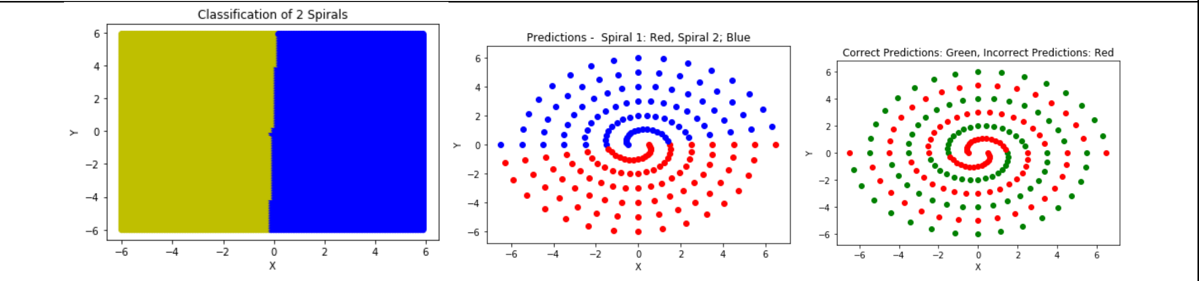
Original 2 Spiral Dataset: SVM Activation with Cost = 0.5, Gamma = 5, Kernel = RBF

The time required to successfully train the SVM compared to an ANN was significantly lower, which was to be expected.

Exploring different kernels, we applied the 'linear' and 'poly' kernels to the task, though both proved to be of no usefulness.

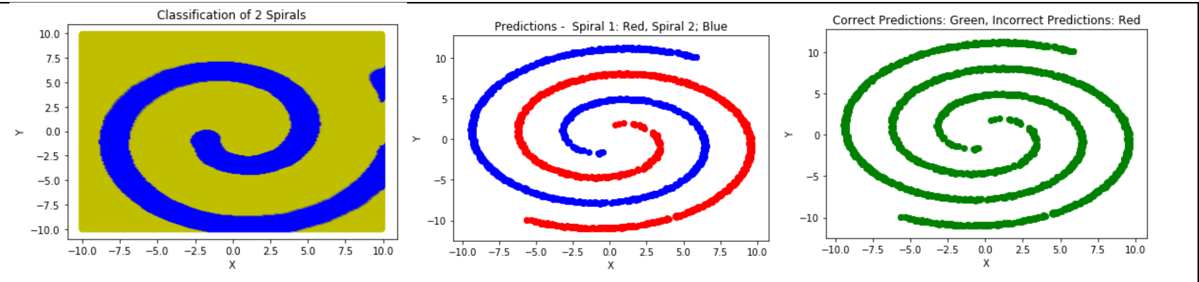


SVM Activation with cost = 0.5, Gamma = 5, Kernel = Linear



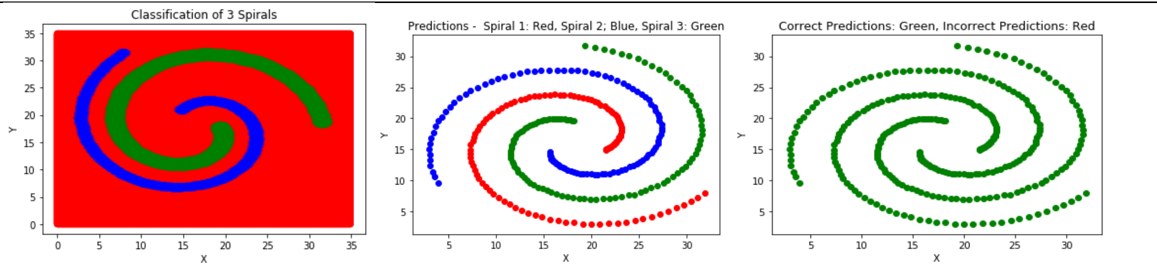
SVM Activation with cost = 0.5, Gamma = 5, Kernel = Poly

Using our own generated 2 Spiral dataset (Rotated by 90 degrees with more points), the SVM was also able to classify the spirals much faster and accurately than using an ANN.



Modified 2 Spiral Dataset: SVM Activation with Cost = 0.5, Gamma = 5, Kernel = RBF

We also ran the 3 Spiral dataset to see how the SVM would run using more than two spirals. The results were impressive to classify the three spirals correctly and within a short period of time. Again, this was achieved in less time using an SVM rather than an Artificial Neural Network.



Three Spiral Dataset. SVM Activation with Cost = 0.5, Gamma – 5, Kernel = RBF.

Question 2 - Mushroom Classification Task

Submit the most successful classifier you can create, and document the process of researching and creating this classier. For solving this you can train a SVM or a Neural Network, or some combination. Discuss how well your classifier performs e.g. by using some suitable form of cross-validation and considering false positives and false negatives.

To begin this task, the 'pandas' library was utilised again to read the data file to a DataFrame so that it is much easier to work with. With this DataFrame, the next task would be to replace the question marks that can be found in some rows of the set. One method would be to remove the rows entirely, but that would compromise the integrity of the file. Instead, the most appropriate method is to use an *Imputer*, a transformer for missing values. This can be found within the '*sklearn.preprocessing'* library.

This *Imputer* method proved to use the wrong format for this task. The *sklearn.preprocessing.Imputer* requires real values, whereas this dataset contains categorical data.

A user on the Stack Overflow forums created a categorical data imputer (https://stackoverflow.com/a/25562948), taking in a missing value, and replacing it with the mean of the current column. This was implemented into the python file to fix the issue.

Now that the '?' values were catered for, categorical data could now be encoded into corresponding labels. There was a choice to use *'sklearn.preprocessing LabelEncoder()'*, or *pandas.get_dummies()* to categorise the data. The difference between the two functions are defined in the following paragraphs.

The LabelEncoder() method converts the values in each column to corresponding integers, for example{Red, Green, Blue}→{0,1,2}. The problem in this example is that Blue has a higher value than Green, and Green has a higher value than Red. This is creating some order over the data, which is not what we want in categorical data.

The pandas.get_dummies() method converts categorical data into dummy/indicator variables. For each new variable found in the column, it will create a new column specifically for that variable, controlling the output through a 0 or 1 output. For example, for the string "*abca*":

	str
0	a
1	b
2	c
3	a

$\longrightarrow$

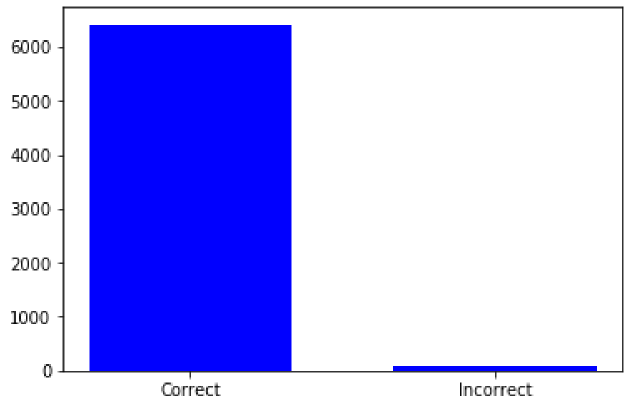
	a	b	c
0	1	0	0
1	0	1	0
2	0	0	1
3	1	0	0

This is exactly what is required, such that there is no order of preference of one value over another. Each value is equal. This expands our 22 input columns out to 116 input columns.

Now the categorical data is in a readable format for the neural network. Before running the entire network, it was decided to split the entire dataset into three parts, 80% being Training Data, 10% being Test Data and 10% being Validation Data.

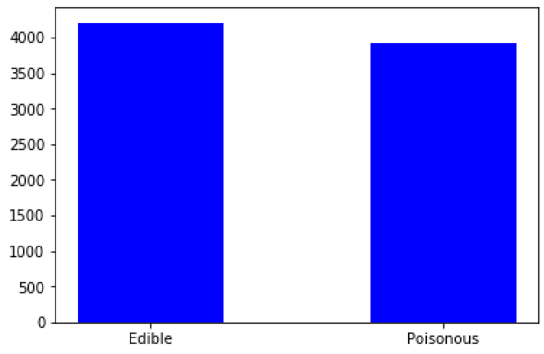
A Neural Network was created composed of three hidden layers in the setup {10,20,10}. The neural network classified the training data with an error percentage of 0.01%. at Epoch #933.

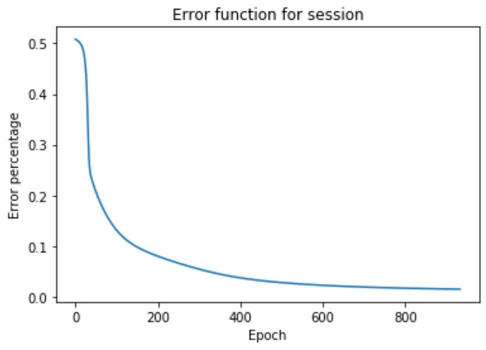
The Neural Network was checked to determine how many cases it got wrong. Out of 6409 rows in the training data, it classified 54 incorrectly (which was to be expected). The neural network could have continued to 50000, though this would have taken a very long time to reach (each epoch took a lot of time to compute).



Further research was collected to find a modified TensorFlow layout that could generate a much faster and more efficient classification to expand on our attempt at the Neural Network (http://vprusso.github.io/blog/2017/tensor-flow-categorical-data). This new network calculated an error percentage lower than 0.001% and was much faster than the original design.

To confirm this great accuracy result, the validation set was run through the classifier to determine if it could correctly classify data not seen before. It was successfully able to categorise all data correctly.





Question 3 – Select your own Data

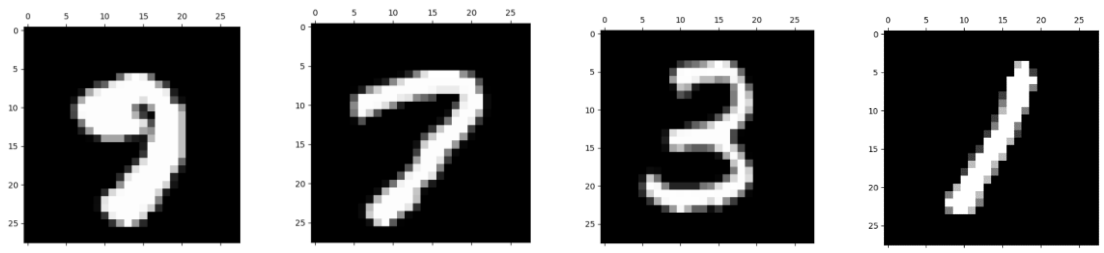
For this question please perform a comparison study of SVMs and ANNs on a data set of your choice.

a) Submit your full study with all specifications so that the marker can verify it.

b) Describe and discuss your approach in a concise report that is detailed enough to allow your solution to be replicated. Include a detailed analysis of your classifier.

Introduction

This study will compare the abilities of ANNs and SVMs to classify the *mnist* dataset. The *mnist* dataset is a set of 55000 28x28 images of hand written digits (0-9) and their labels.



Sample data from the *mnist* dataset

ANN

The initial attempts of creating a neural network to classify this dataset wasn't a huge success, the accuracy of the neural network on the testing data only reached about 15% which is marginally better than just blindly guessing. It was evident that we needed to change some things up.

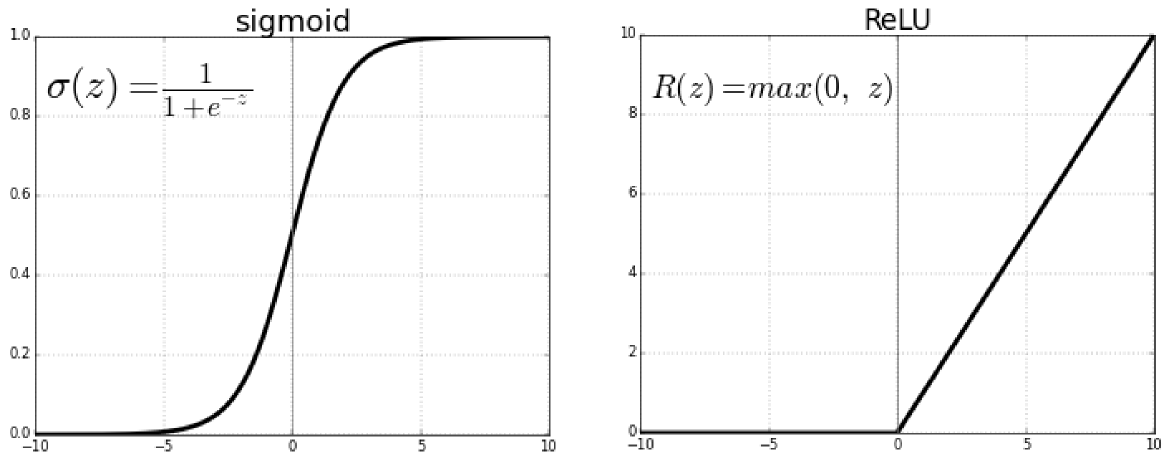
The first thing to change was the activation function. Previously the sigmoid function was used, which isn't used very often in computer vision (essentially what is being used for this dataset).

The reason the sigmoid function is scarcely used is because of the phenomena known as *Vanishing gradients*. The *Vanishing Gradient* problem is a difficulty found when each of the neural network's weights receive an update proportional to the gradient of the error function (with respect to the current weight). The problem in some cases is that the gradient will be 'vanishingly' small, preventing weights from changing the value. In the worst scenario, it could stop the neural network from future training.

Since the sigmoid function 'squashes' the input value to between 0 and 1, even if a neuron receives incredibly large inputs the most the activation function can output is 1. The 'extreme activation' of a neuron can be lost after a few layers.

So, what does this mean in respect to computer vision? Computer vision requires very large and deep Neural networks to perform well, because it's a very complicated task. This means that with sigmoid functions the information from the earlier layers can be lost. If a neuron that identifies a feature of a digit receives a large input, it's likely that the image fed into the ANN is of that digit. However, if this neuron appears in the earlier layers of the ANN that information can be lost, rendering the use of Deep NNs pointless.

To find a solution, a new activation function can be introduced: Rectified Linear Unit (ReLU). This activation function is used heavily in computer visions and other fields involving deep learning.



Comparison of the Sigmoid function and the Rectified Linear Unit (Useful for computer vision tasks)

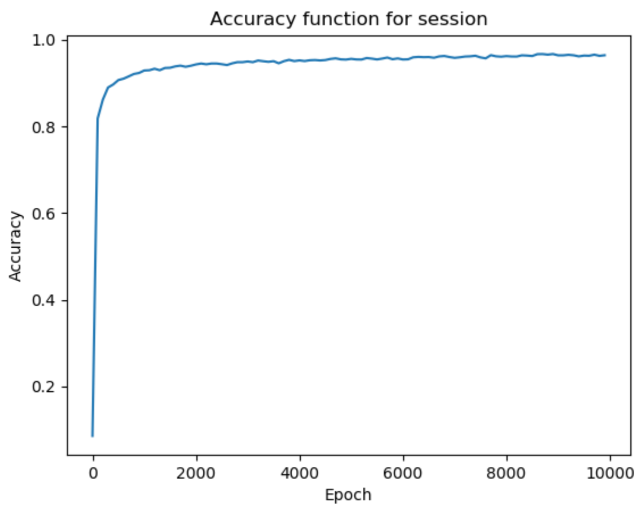
ReLU prevents the output from being a negative value. If the input is negative, it will always output 0. If the input is positive, it outputs the input. This means that if a neuron receives a large amount of inputs, the output will simply be the sum of those inputs (the activation function in this case does nothing). A neuron can output very large numbers and the information won't be lost in a few layers.

Now that the ReLU activation function is in place, more layers can be added to our neural network. The best NN that was tested consisted of 4 hidden layers with 500 neurons in each layer.

One more thing to change was splitting up the epochs into batches, the mnist dataset contains 55000 images which means that each epoch takes a very long time. Picking the batch size took a bit of guess and check, a batch size too big would take too long but a batch size too small overfits to the data in the batch. The final batch size was agreed upon was 200.

The error function used was *Softmax Cross-Entropy*, which is commonly used when '*one hot'* labels are used. It is a natural choice for this.

The results of this neural network were much more impressive then the initial attempt. After 10000 Epochs the ANN reached an accuracy of 96.44% on the testing data.



SVM

Next up was to test out a SVM for the *mnist* dataset. Before we do, let's make a hypothesis. The reason that the *mnist* dataset was chosen was to highlight the advantages of using ANNs, since earlier we demonstrated that SVMs were superior to ANNs for the Spiral problems. Machine vision is supposed to be a very difficult task for SVMs which is why Deep learning is traditionally used. Our hypothesis is that SVMs will not be close to the accuracy of the ANN.

It was initially tested using a SVM with a RBF kernel and the following values C = 5, gamma = 0.05.

This was not successful as the SVM could not fit the data, it was taking way too long. After some online research we discovered that using this method would take about 2 days to compute the classifications with a fast computer. The ANN took about 5 minutes to train on a slow computer so a training time of 2 days would be unacceptable. Another solution was necessary.

A SVM was needed to be used which could fit the data within a reasonable amount of time. The linear kernel was decided to be the preferred kernel in this situation.

Linear kernels are useful when training on big datasets, like *mnist*. This is because they use algorithms which terminate in O(n) time whereas the RBF kernel uses many algorithms which terminate in O(n^2) time.

However, there are trade-offs; by using faster algorithms the resulting SVM is less accurate, but for this purpose, it is necessary to reduce the training time.

This linear SVM took 913 seconds (about 15 mins) to train and produced an accuracy of 93.63% which was better than we expected.

We used parameter values C = 1 and gamma = 0.8, we couldn't experiment too much with these values because the training takes so long but these were the best values that we tested.

While it still took a while the training time was 0.005% of the estimated training time of a RBF SVM. In the end, the use of a linear SVM was worth it.

Conclusion:

This study tested the ability of ANNs and SVMs to classify the *mnist* dataset. The ANN was initially not successful but by tweaking some parameters the ANN was able to achieve an accuracy of 96.447% on the testing data within a reasonably short amount of time (5 minutes).

Like the ANN, the initial test of the SVM was a failure, threatening to take more than 2 days to train. However, after changing the type of kernel to a linear kernel the SVM could train in about 15 minutes and achieved an accuracy of 93.63% on the testing data.

This study demonstrates the strengths of ANNs and some of the flaws of SVMs. To properly classify hand written digits both systems needed a large amount of training data. The ANN could handle the large dataset considerably better than the SVM, training faster and achieving a higher accuracy. The SVM with the RBF kernel trained in O(n^2) time, with n being the number of data entries, which meant that when a large dataset like this is used the time required is massive. To combat this the linear kernel was used which was successful but by doing so the accuracy of the SVM was sacrificed.

While the SVM was a clear victor in the spiral problem the opposite was true for the *MNist* dataset. Meaning there are merits to both systems and choosing which to implement completely depends on the problem you wish to solve.

Side Note:

There were issues with using pickle to save the Neural Network configurations. Instead, it was decided to use Tensorflow's in-built functionality to save the configurations to a folder. Though we could not use pickle for saving Tensorflow layouts, SVM layouts were able to be saved using pickle.

For more information please see the following issue on Github:

https://github.com/keras-team/keras/issues/8343

Our solution was found through this Stack Overflow post:

https://stackoverflow.com/questions/38000180/save-tensorflow-model-to-file?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa