Daniele Bacarella 870116-8356 (Daniele.Bacarella.3279@student.uu.se)

Kiril Goguev 881225-3618 (Kiril.Goguev.3417@student.uu.se)

Win-Loss Classification of Starcraft II Replays

Abstract— **This project aims to build a model of classification based on Starcraft 2 replay data to predict a win or loss. Not all the information contained in replays are used towards the classification but a small portion of them, in particular the first few actions involving constructions building and trained units.**

I. Introduction

Machine learning is a relatively young discipline with many applications. One of which has been quite useful in analyzing game mechanics. Classification-A task of pattern recognition along with feature extraction can be used to analyze and predict things like win/loss ratios, and artificial intelligence decisions. However, many researchers in machine learning seem to be quite interested in the applications of classification in RTS (Real Time Strategy) video games. Where game mechanics allow for forward prediction analysis of moves in order to train players.

Players in RTS games make decisions and actions in real time, meaning their choices immediately affect the game world. A popular RTS game today is Starcraft II created by Blizzard Entertainment in which players choose to play amongst one of three races namely Protoss, Zerg and Terran. Each race consists of unique and individual units, buildings, and upgrades. While it is considered a well balanced game, players constantly strive to create the best “Build-orders” in order to increase their chances of winning against an opponent. Build-orders are a list of actions which the players perform during the game.

The purpose of this paper is to see whether we can create a classifier which will be able to recognize build-orders and predict which ones are more likely to lead to a win and conversely which ones were more likely to lead to a failure. To achieve this we started collecting Starcraft II replay data from various websites. Classification problems normally require vast amounts of data in order to train on and this project was no exception.

The use of Starcraft as a machine learning concept has been explored in two previous papers from Stanford University where the authors of the first paper Spann et. Al used Starcraft I , select maps and a very small subset of build orders from the races in order to predict the map imbalance. Where as the authors of the second paper Cox et. Al used Starcraft II replay data in addition to page ranking and statistical information from the replay to predict the win-loss ratios of two given players. This project was more of a social networking experiment using statistical analysis.

Starcraft II is also considered to be a professional tournament game, which means that it qualifies to be played as an E-sport and has a very large fan base. This allows us to collect the top ranking players and their replays very easily as they are organized in season and tournament packs with each pack having up to 200 replays inside.

Near 1,200 replay data have been collected , pre-processed and used as the data set input for the classifier. As with most classification problems the problem is essentially solved in the pre-processing phase. This required careful thinking on our part since having the wrong encoding could make your classifier yield unpredictable results or worse, results which would not correlate to the real world problem.

To reach our goal of having a proper classifier for the Starcraft II game we require specific steps to be taken. Primarily Data Collection, Data-Parsing and pre-processing, Building and tuning of the classification network, and finally testing the network. Lastly we will show some future applications involving this project's idea.

We estimate that with enough replay data, proper encoding of the data and a network that is properly tuned our classifier should give us near real world projection data.

II. DATA COLLECTION

In order to build a successful and reliable classifier, all classifiers require vast amounts of data to be trained on. Starcraft II conveniently provides a source of data for training players in the format of replay files. These replay files contain only player actions in terms of mouse clicks and keyboard input, however no video or statistical information is contained inside.

Since StarCraft II is a very popular game, the creator keeps it up to date with patches which may introduce severe changes in the players play styles. For example a Unit which took some amount of time to be created may have been a very good choice before a specific patch, but now takes way too long.

StarCraft II's status as E-sport game makes it a great candidate for having multiple fan-owned and professional websites which maintain and list tournament/event replays and regular game replays.

Since our goal is to create a classifier that is most consistent with the real world(in this case current state of the video game) we considered only Top Ranked players and those that participated in high level events.

StarCraft II makes it easy to distinguish which players are considered top ranking by using its own ranking system within the game, namely Bronze, Silver, Gold, Diamond and Masters. While all players may record their replays and upload them to the various sites mentioned below, we have elected to use replays from the tournaments which only allow Masters, Diamond and Gold players.

We chose the players and tournament level because we can safely assume that these players have the highest level of reflexes since the game requires split second actions and input in order to execute a perfect strategy.

The replays were taken from the following websites:

http://www.sc2win.com/category/events/

http://drop.sc/packs

http://www.gamereplays.org/starcraft2/replays.php?s=d034bf8a5bf019dbd3cfd1b72afd4c33&game=33&show=events&st=0

It is important to note that these replays were from the current patch and about 5 levels below. We did not consider any lower version since the changes in them would potentially train our classifier wrongly.

We got near 1,200 targeted replays which were 1v1, or one player versus another because it is much simpler to solve the problem for a replay which contains 2 players information and races than a bigger replay which contains up to 8 different players(4v4) the problem then becomes combinatorial when they are all of different races and all choose different build orders.

III. DATA PRE-PROCCESSING

As mentioned in the introduction, the classification problem can usually be solved in the data-preprocessing phase. This means that we have to be very careful in this section in feature extraction.

In an RTS game such as StarCraft II, the strategies are not so well defined, rather a win is not strictly correlated to a sequence of actions. Players are free to make the choices and play styles up to a certain degree(e.g. A Protoss player can not build an advanced unit until the lower level requirements are met). The Build-orders are essentially an individual players choice of actions in the game which we follow and consider up to a certain point in the game. Usually the first few minutes.

Normally for a Player of a low skill, the build order time frame is a lot longer, quite a few minutes (10 or so for a low bronze). However, for a player of Masters or Diamond, we expect their time to be much shorter (2 or 3 minutes or so). Regarding this, we have actually opted to not include the time frame since it essentially already encapsulated in the top players strategies (e.g. Perfect execution of actions in respect to time)

As stated before, the game is played by players who control three(3) different races, each with their own unique buildings, units and upgrades. This means that while we extract these actions we have to devise some way to encode them as input into our classifier. The problem here is how exactly we encode them in the input file, The games replay does not care so much about the properties of the unit or building, it is up to us to distinguish one unit or building from another.

As with [2], we also have to analyze the replay files and based on the format already published in various websites , we intended to use an existing parser called PHP Sc2 Replay[3].

The replay parser allowed the parsing of one replay file at a time, however we required a better method to parse all of our replays at once. The parser is freely available under the GNU GPL v3 license scheme, this means we had full access to the code to modify it however we liked.

Our idea was to use the parsing features and to extract only the following: Race, Build-orders string and win/loss.

Since the Build-orders will have to be used as input to train the classifier, each building and unit of all the three races have been assigned a unique numeric value. Hence, during the parsing phase, each building/unit's name obtained from the replay files had a mapping function applied to it to get its numeric value.

Here, our goal was to encode the data in such a way that the build orders of the individual players(records) would not be counted as too similar right away. Rather we wanted each record to be considered uniquely.

The intermediate input format was the following:

RACE-NAME, Id0 , .. ,Id14, {0,1}

where 0,1 indicates loss or win.

This format was then written to a CSV file, in which further pre-processing was done.

Having collected about 2,359 records, the next pre-processing step was about getting values of the Target Attribute (at this stage {0,1} is dependent on all the actions taken in the entire game) which is dependent only on the build-order and its distribution on the dataset.

Firstly, for each build-order we took the number of duplicates out of the entire dataset along with the number of duplicates that resulted in a win.
Secondly we got the winning ratio out as a floating point value and last we set a threshold of 0,55 and turned all the values <0,55 into 0 and 1 otherwise.

We tried to leave the values in the more interesting fractions however, the network was unable to give us good results using the fractions when testing using the test set. The values that came out were always good for true positive but always 0 for true negative. When we switched to having the targets as {0,1} we started getting real results.

As last step of the pre-processing , we ordered all the records based on the race's name and then we split the entire dataset into three, each one containing records of the same race.

The final input format is:

Id0 , .. ,Id14, {0,1}

We also elected to have one data set containing all the build orders from all three races just to compare how well the networks generalized as well as the encoding of the networks

IV. TRAINING

The data set was split into three different sets because we wanted to test out the encoding of our problem and the relationship of the build orders to the win/loss ratio.

Encoding is a huge problem for us and we can either train 2 different styles of networks, for example : 3 unique networks one for each race, or we can try training one big one with all 3 races inside. – The 3 unique race networks was suggested to us during a meeting for the development of this project in order to see how well the network could differentiate by the input data.

For speed and ease of use, we opted to test out our ideas using the built in neural network toolbox which came with MatLab. The toolbox contains many pre-built networks and features which are fully customizable.[4]

Since we already had inputs and targets, we decided the best way to evaluate our classifier would be to use supervised learning which is a form of learning that allows the use of external information in the training environment (targets). In our case, our training environment consisted of the 3 different data sets namely the inputs (Protoss, Terran and Zerg) Build-orders in the above format. As well as the corresponding target values (0,1). More precisely , the size of the 3 datasets are :

- Protoss - 803 records
- Terran – 738 records
- Zerg – 818 records

This problem may be considered an example of the infamous "No free lunch theorem" which suggests that because a problem does poorly on one set of training functions and parameters it may be much more suited on a different training function/parameters, and therefore we attempted to finely tune the network on a set of possible training functions and parameters in order to find the best fit for our problem.

Testing consisted of the following functions and values,

Activation Functions

Trainrp-Resiliant backpropagation
Trainscg-Scaled conjugate gradient backpropagation
Traingd-Standard Gradient Descent backpropagation
Traingdm- Gradient Descent with momentum
Traingdx-Gradient Descent with momentum and adaptive learning rate

Hidden Nodes

5
10
15
20

Training Functions

Purelin
Logsig

Learning rate

1.05
0.01
0.06
0.0006

Maximum Number of Epochs

1000
1500
2000

Maximum Validation Checks

Inf(9999999999)

Minimum Gradient

0

Detailed tests with the above parameters

Our dataset was unfortunately relatively small due to our constraints of having a stable build order set (if we chose to include games with lower patch level, we would not be able to guarantee that the build orders would work for the next games)

We start testing by using Resilient Back propagation, where we let the accumulated partial derivative $E' = \partial E / \partial w_{ji}$ decide the direction only. The step length 'eta' is then determined by the formula:

$\Delta wji=\{-\Delta ji, E' > 0 ; +\Delta ji, E' < 0 ; 0$ otherwise

The step length, increases with a factor ή+ as long as E' does not change, however it will be decreased by a factor of ή- when E' does change. Here we see the following effect: Acceleration down the slopes and a deceleration when we would have overshot a minimum. [5]

Scaled Conjugate back propagation

While in the standard back propagation algorithm weights are adjusted in the steepest descent direction, which is the direction where the performance function is decreased most.

In normal conjugate methods, a linear search is done at each iteration of the algorithm. However with scaled conjugate gradient method this search is avoided resulting in a time saving search.

It has also been shown that this algorithm is significantly faster than the standard back propagation algorithms [6]

Standard Gradient descent

The derivative of the error is calculated by the following formula:

$\Delta wji= - ή*\partial E/\partial wji$.

The gradient is now used to adjust the weight and bias' of the network so that the next batch of input is supposed to be closer to the output.

However the issue with this algorithm is that if the learning rate: ή is not set properly, the network will not be stable(zig zag/bouncing effect) or the network will take too long of a time to converge. This version of the algorithm does not change the learning rate and does not carry a momentum for the ball rolling down the error landscape. This is a problem because we may potentially get stuck in local minima, instead of the target global minima. [7]- We shall see the above effect in a graph in a few sections.

Gradient descent with adaptive learning

This learning method incorporates the changes in the learning rate. A normal gradient descent uses a stationary (non-changing) learning rate, which means that the learning rate parameter will have to be guessed and tuned quite a lot before getting good /accurate enough results. The adaptive learning rate now becomes sensitive to the changes in the local error environment [8] . – However this learning algorithm was not used explicitly in tests but we as used to guess the best learning rate.

Gradient Descent with momentum

Same as the Standard gradient descent, however the update formula, now comes with a momentum term- that is a constant speed applied to the ball as it rolls down the landscape in order to not get stuck in local minima.

The following formula is used:

$\Delta wji(t+1)=ή*\partial j*xi+[\alpha*\Delta wji(t)]$ –momentum constant

Gradient Descent with momentum and adaptive learning rate

Same as the above Standard Gradient Descent but with the added momentum constant and adaptive learning rate as well as adaptive momentum rate instead of a constant.

Hidden nodes

Hidden nodes were tested between 5, 10, 15 and 20 in order to provide enough structure to analyze information but not enough to overwhelm the network. It was determined in most cases that 20 hidden nodes would be sufficient enough to get good results.

Transfer functions

The following transfer functions were tested with: Logsig in the Output and Purelin in the input. The reasoning behind this was that the input, was not bounded between 0 and 1 and therefore we thought that a linear transfer function should suffice, however it was later found that Logsig was a better choice in both Hidden layer and Output layer nodes. The Logsig makes sense in the Output layer since the targets are scaled to be between 0 and 1. While the Purelin function did not scale the inputs properly down towards the output function of 0..1, we did not get any good results with Purelin and Logsig so we switched to both Logsig and Logsig. Tansig, was not considered because the data was not scaled from -1 to 1.

Learning Rate

The learning rate was tested on each learning algorithm from a range of 1.05 down to 0.0006. When the learning rate was set to 1.05, it caused a lot of problems with training the network, as the ball in the error landscape would wildly bounce in and out of valleys. However we overcame this problem of setting the learning rate by using an Adaptive learning algorithm and checking the training state at various points during training.

We found that the minimum jumping of the ball with a good speed down the slope was either 0.0005 or 0.0006, which became the basis for all of the learning algorithms that used the learning rate parameter. The following figure below this section shows the bouncing effect. This also was a predominant effect in Adaptive learning rate algorithms when they were not tuned enough.

A Note on Adaptive Learning Rates

Adaptive learning rates are supposed to alleviate the problem of setting a learning rate, however one must be careful because you can run into an issue where the learning rate pulsates between a set range of numbers, of course this is dependent on the training data as well as the error landscape but you get a gradient that dips consistently, almost synonymous with setting a learning rate too high on normal back propagation functions.

This too required careful tuning but with some luck the bouncing issue has been minimized.

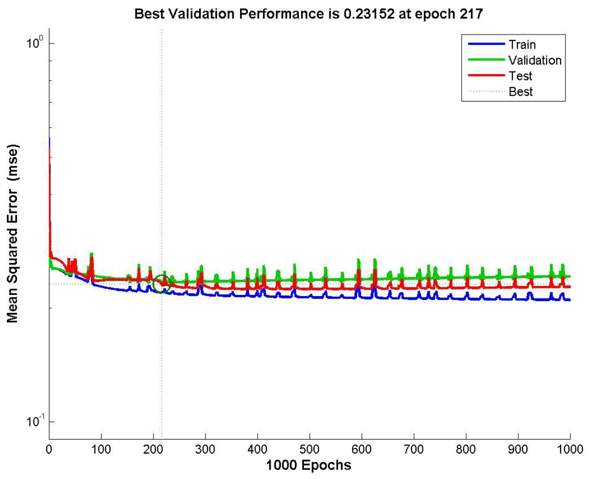


Figure 1. A performance graph showing the effects of adaptive learning rates

Maximum Epochs

While testing consisted of a range of epoch values from 1000,1500, 2000, 3000. It was found for smaller datasets such as the ones that were split by the races, it was better to use a smaller number of epochs while training. Therefore the tests showed that using a 1000 epochs for each separate race and 2000 epochs for the dataset which contained all in one. The problem with having a high number of epochs for a small dataset is that the network will become over trained due to the time (epochs) being high. As can be seen in figure 2, with the training line vastly below the testing and validation.

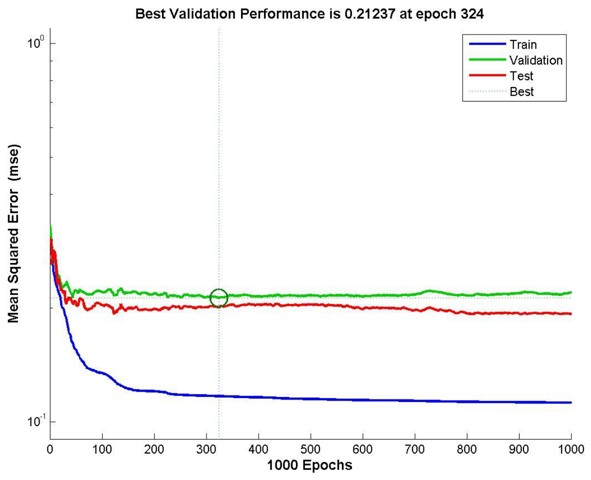


Figure 2. A performance plot where there was a possibility of overtraining the network

Maximum Validation Checks and Minimum Gradient

Since our learning functions success is measured by a error rate, we simply wish to test how much we can minimize this error rate by letting the network strive towards 0. We can accomplish this is MatLab by setting the 'max_fail' to Inf. (9999999) and setting the 'min_grad' to 0. The effect this has on training is the following:

Training will not stop when the Validation Performance has not increased more than 'max_fail' times, and the minimum gradient will always be 0 so the network will keep training until it hits it(usually will hit the max number of epochs first)

The following parameters worked reasonably well for our data:

Hidden nodes 20
Transfer function Hidden layer: Logsig
Transfer function Output layer: Logsig

Learning Algorithm: TrainRp
net.TrainParam.min_grad=0;
net.TrainParam.epochs=1000;
net.TrainParam.max_fail=999999999999999;
net.TrainParam.lr=0.0005;
net.TrainParam.delta0=0.03;

Learning Algorithm: Trainscg
net.TrainParam.min_grad=0;
net.TrainParam.epochs=1000;
net.TrainParam.max_fail=999999999999999;

Learning Algorithm: Traingd
net.TrainParam.min_grad=0;
net.TrainParam.epochs=1000;
net.TrainParam.max_fail=999999999999999;
net.TrainParam.lr=0.0006;

```
Learning Algorithm: Traingdm
net.TrainParam.min_grad=0;
net.TrainParam.epochs=1000;
net.TrainParam.max_fail=999999999999999;
net.TrainParam.lr=0.0006;
net.TrainParam.mc=0.4;
```

```
Learning Algorithm: Traingdx
net.TrainParam.min_grad=0;
net.TrainParam.epochs=1000;
net.TrainParam.max_fail=999999999999999;
net.TrainParam.lr=0.0005;
net.TrainParam.mc=0.8;
net.TrainParam.lr_inc=1.02;
net.TrainParam.lr_dec=0.4;
```

Finally to test the confusion matrix on the test set the following was used(Where X is the name of the dataset, Protoss, Terran, Zerg, AIO):

```
i3=stats.testInd;
Outputs=sim(trained_net,X_Input);
t3=X_Target(:,i3);
y3=Outputs(:,i3);
plotconfusion(t3,y3);
```

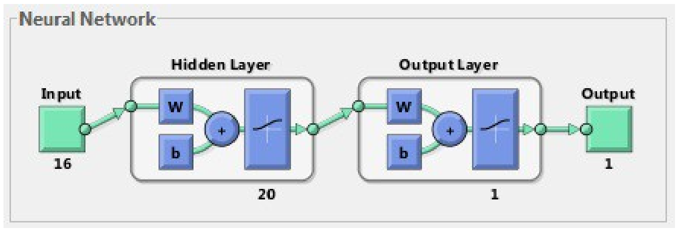


Figure 3 shows network infrastructure

Mean squared error (Mse) - Error function to be minimized

The mean squared error is usually the error function that is used in gradient descent problems. and finally, dividerand as the function which splits up the input data as training, validation and test in the following order, 70% training, 15% validation and a final 15% for testing.

Finally to further understand our accuracy rate, we used the built in plot-confusion matrix for the test set only, to show us the right numbers after training.

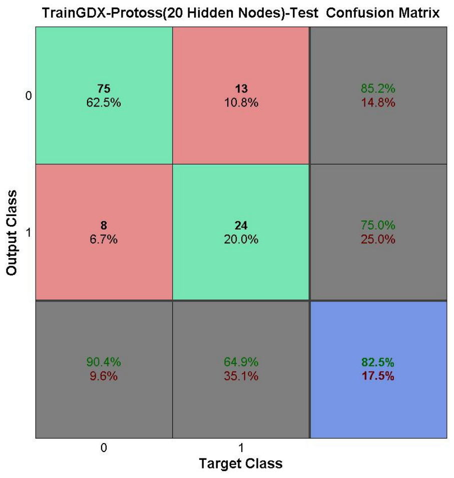
V. WIN PREDICTION RESULTS

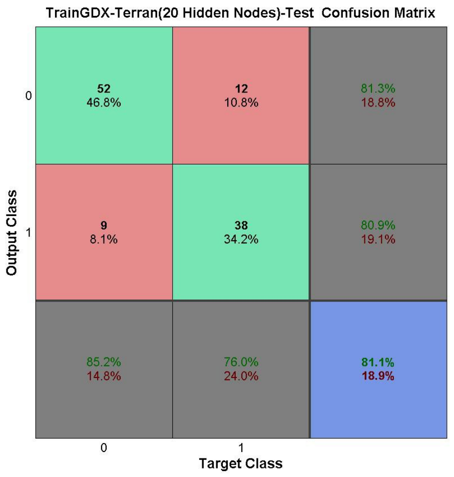
The expected results out of the training phase were a fair 50% true positive and 50% true negative for the split data sets. However for the dataset which contained all 3 races in one we expected a much lower true positive and true negative due to the problem of encoding, where the neural network seems to group similar or like values and not be able to distinguish the difference.

Although the building orders we took in consideration have not been taken entirely as well as the information regarding the two opponent races in a game, we got satisfactory results in the classification, here some statistics:

We found the best algorithm to use on the Protoss, Terran and Zerg data set (Separately) was the following: TrainGDX-Gradient Descent with momentum and adaptive learning rate.

This algorithm consistently gave us high true negative and true positive rates in the confusion matrices. Approximately 78-80% for Protoss and Terran and 65-69% for Zerg, respectively for true positive and 18-30% for true negative.





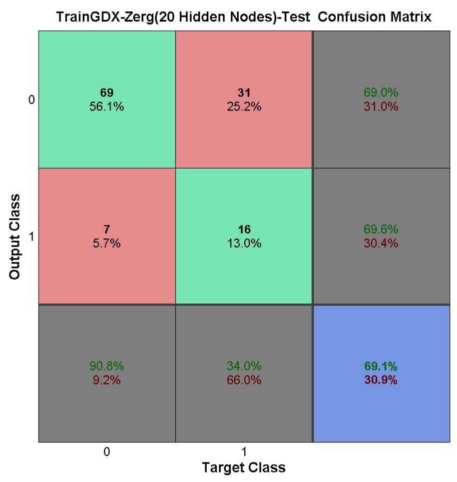
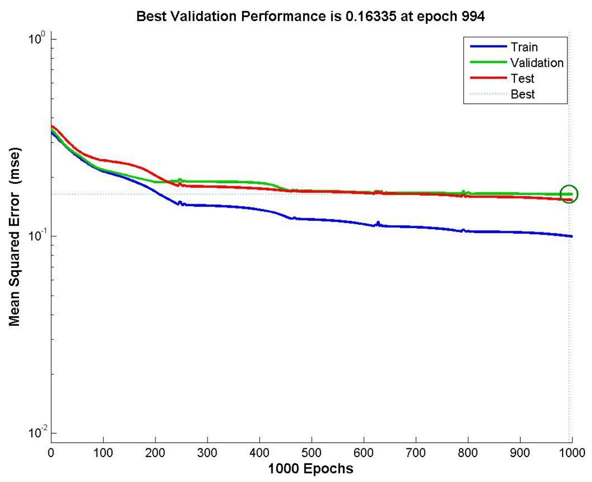
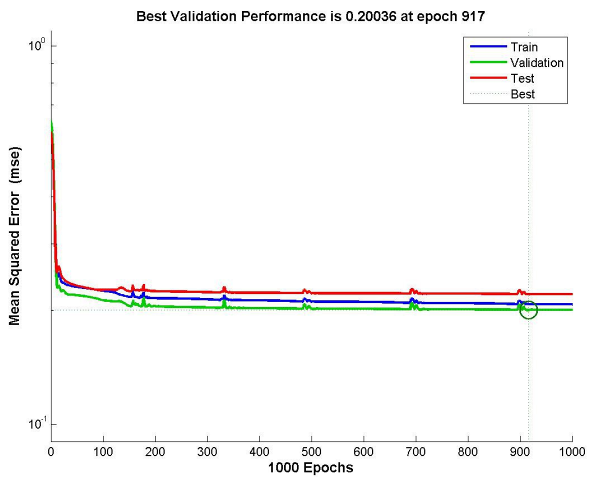


Figure 4- plot confusion matrix for Protoss, Terran Zerg(TrainGDX)

We can also see a good decline in the error measure with almost no oscillations for the traingdx algorithm shown in the figures below in order (Protoss, Terran and Zerg).





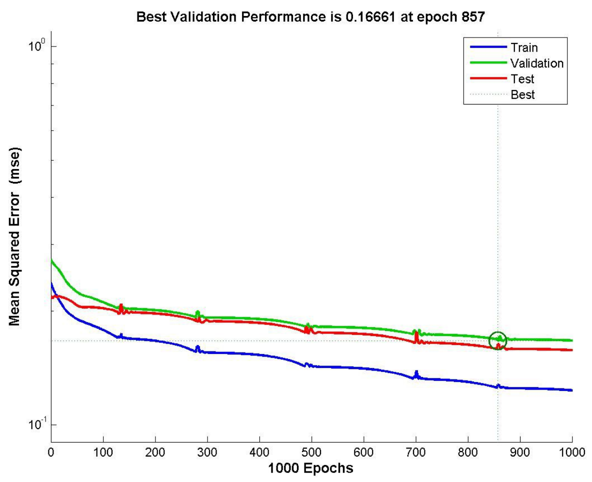
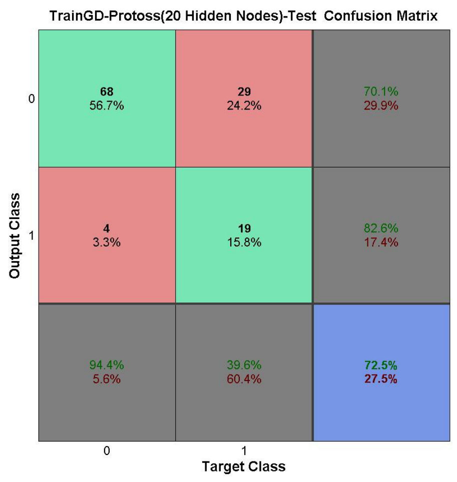
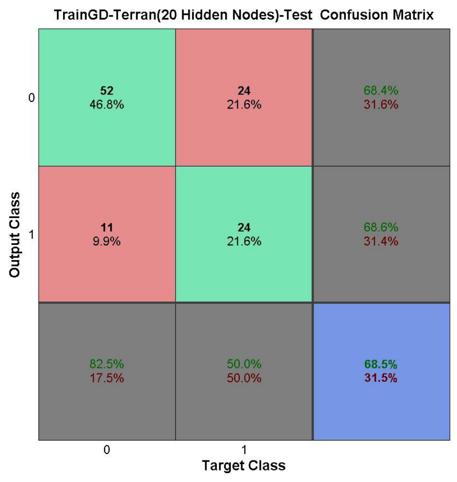
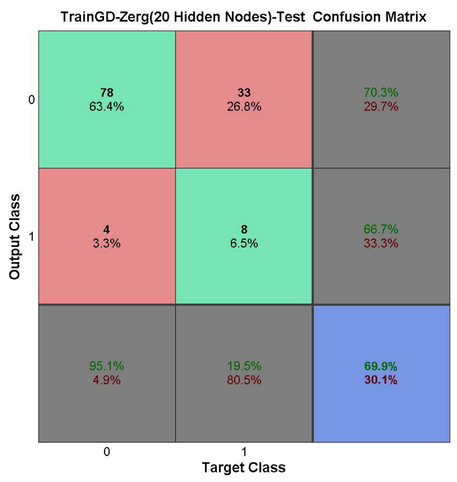


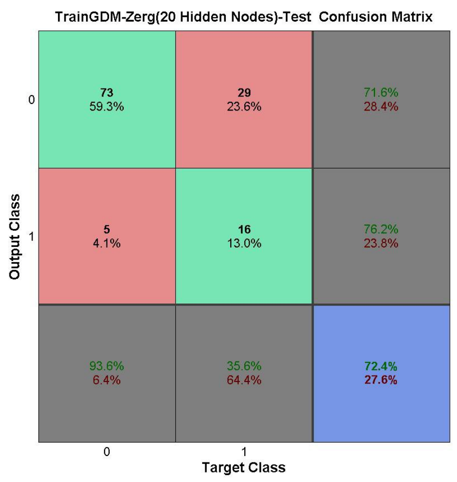
Figure 5 – Performance graphs-TrainGDX, Protoss, Terran, and Zerg(respectively)

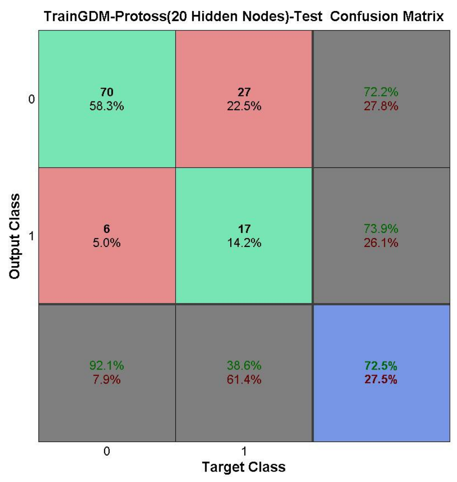
The other algorithms gave true positive values around 70-75 % and true negative values around 27-35% the following figures are samples of some of the tests:

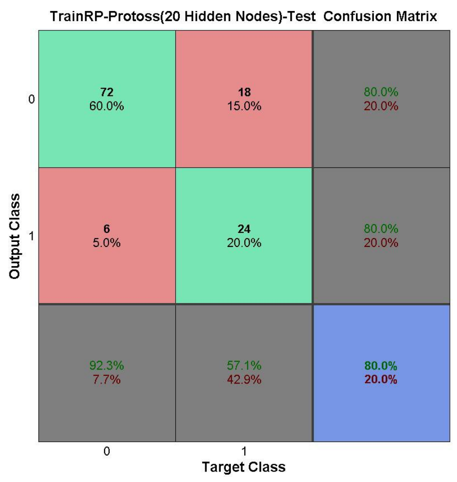


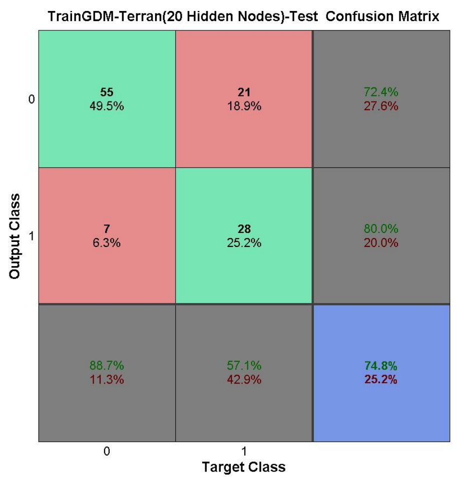


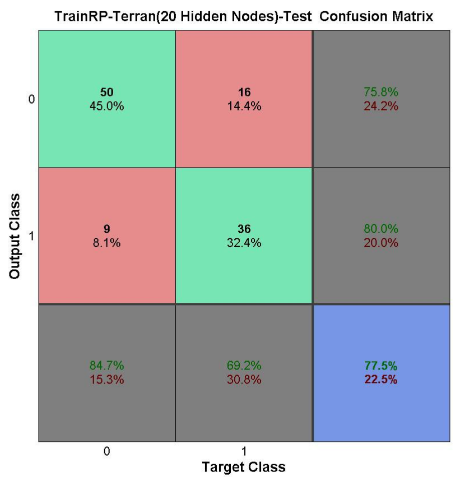

TrainGD-Zerg(20 Hidden Nodes)-Test Confusion Matrix
Output Class
Target Class
0
1
78
63.4%
33
26.8%
70.3%
29.7%
4
3.3%
8
6.5%
66.7%
33.3%
95.1%
4.9%
19.5%
80.5%
69.9%
30.1%

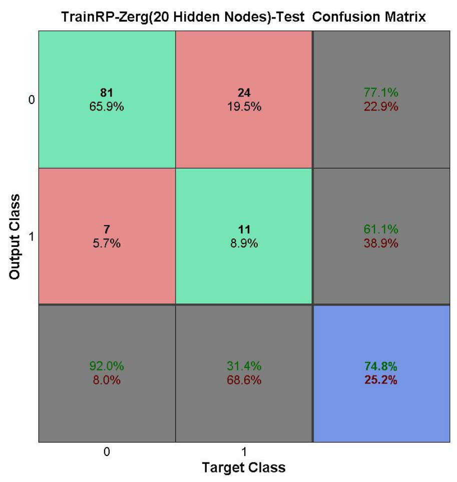

TrainGDM-Zerg(20 Hidden Nodes)-Test Confusion Matrix
Output Class
Target Class
0
1
73
59.3%
29
23.6%
71.6%
28.4%
5
4.1%
16
13.0%
76.2%
23.8%
93.6%
6.4%
35.6%
64.4%
72.4%
27.6%

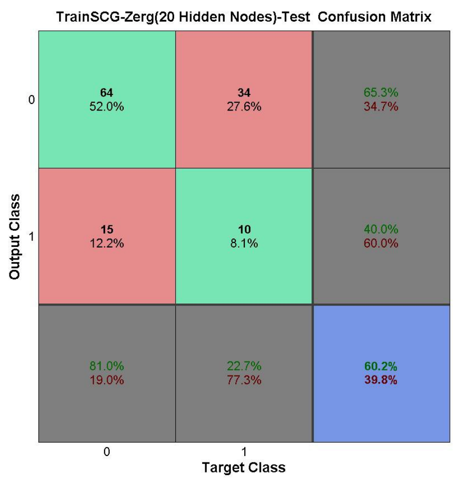

TrainGDM-Protoss(20 Hidden Nodes)-Test Confusion Matrix
Output Class
Target Class
0
1
70
58.3%
27
22.5%
72.2%
27.8%
6
5.0%
17
14.2%
73.9%
26.1%
92.1%
7.9%
38.6%
61.4%
72.5%
27.5%

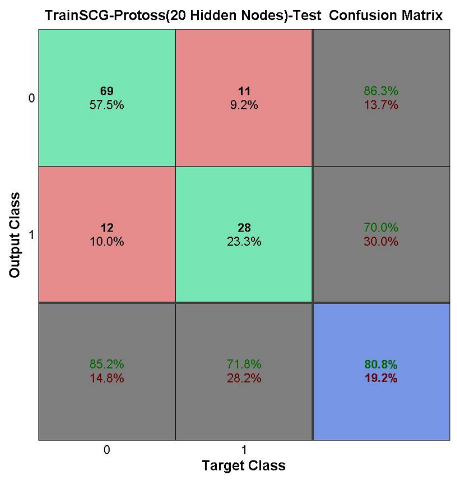

TrainRP-Protoss(20 Hidden Nodes)-Test Confusion Matrix
Output Class
Target Class
0
1
72
60.0%
18
15.0%
80.0%
20.0%
6
5.0%
24
20.0%
80.0%
20.0%
92.3%
7.7%
57.1%
42.9%
80.0%
20.0%


TrainGDM-Terran(20 Hidden Nodes)-Test Confusion Matrix
Output Class
Target Class
0
1
55
49.5%
21
18.9%
72.4%
27.6%
7
6.3%
28
25.2%
80.0%
20.0%
88.7%
11.3%
57.1%
42.9%
74.8%
25.2%


TrainRP-Terran(20 Hidden Nodes)-Test Confusion Matrix
Output Class
Target Class
0
1
50
45.0%
16
14.4%
75.8%
24.2%
9
8.1%
36
32.4%
80.0%
20.0%
84.7%
15.3%
69.2%
30.8%
77.5%
22.5%







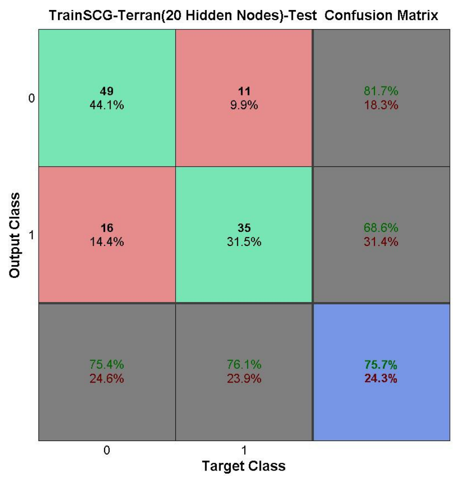
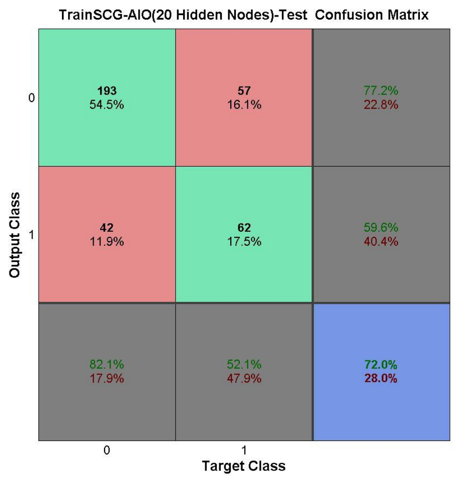
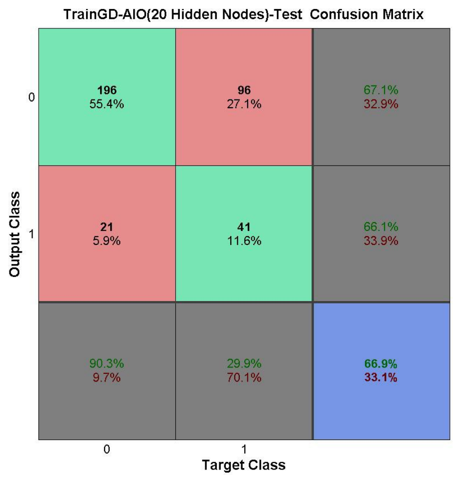
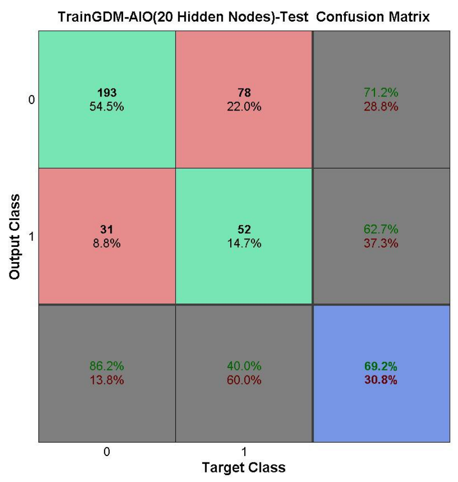


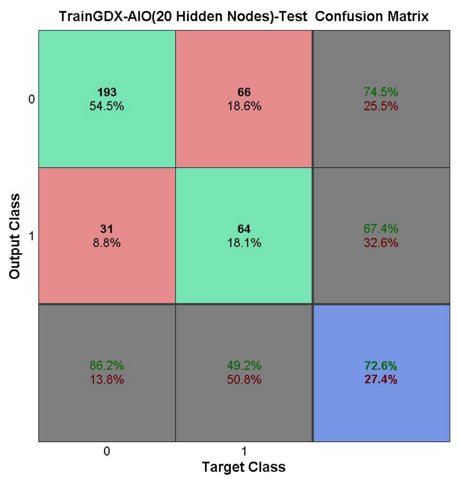
Figure 6 – plot confusion matrix groups for remaining Algorithms

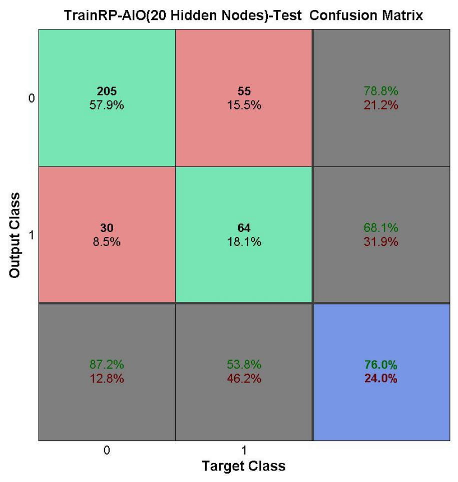
For the dataset which consisted of all the build orders from all the races, the following tests were produced:











As can be seen above the all in one data set has a consistent classification rate of true positive 67-76% and true negative of 24-33%. This dataset despite all the tuning never got past 76% where as when we broke down the network into 3 parts, we were able to get consistently higher percentages on some algorithms, in particular the GDX algorithm before gave us a high percentage of around 81-82% where as here it gave us only 72%.

We can also see the error performance graphs , which show little progress when we have a data set with all the build orders in one,. The Error rate is always at a constant dip down towards 0 but then curving off and usually(except for SCG) staying way above 10^{-1} .

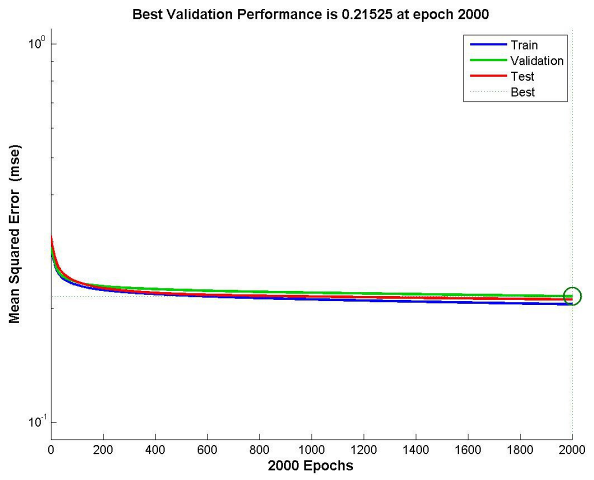


Figure –TrainGD- All-in-one-20 hidden nodes error graph

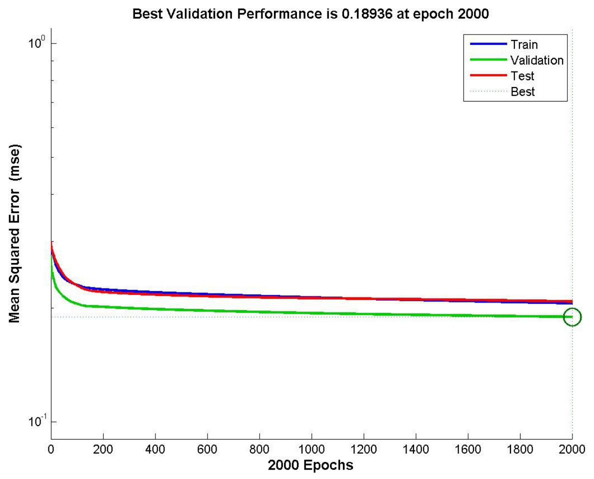


Figure –TrainGDM- All-in-one-20 hidden nodes error graph

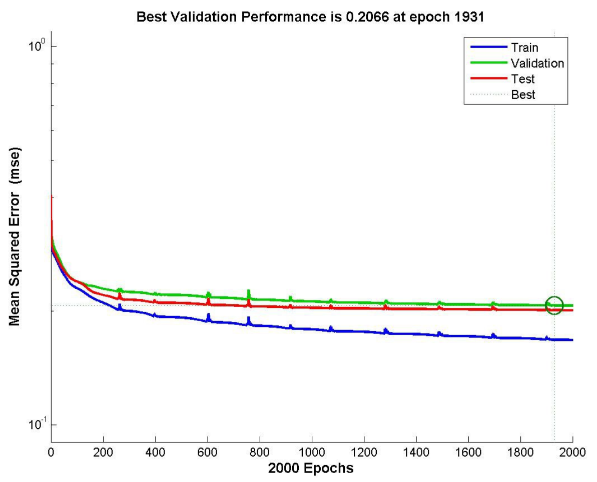


Figure –TrainGDX- All-in-one-20 hidden nodes error graph

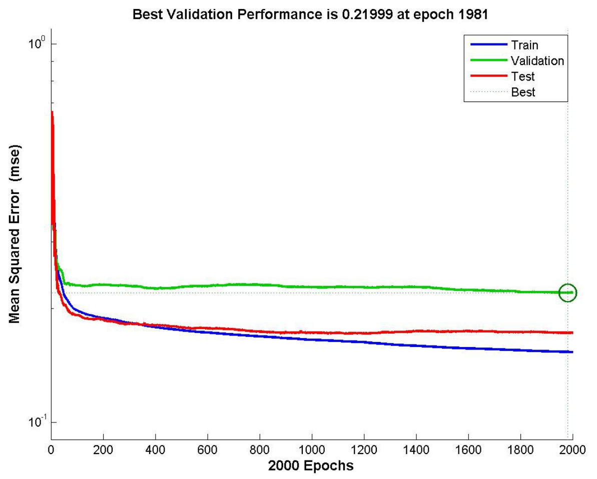


Figure –TrainRP- All-in-one-20 hidden nodes error graph

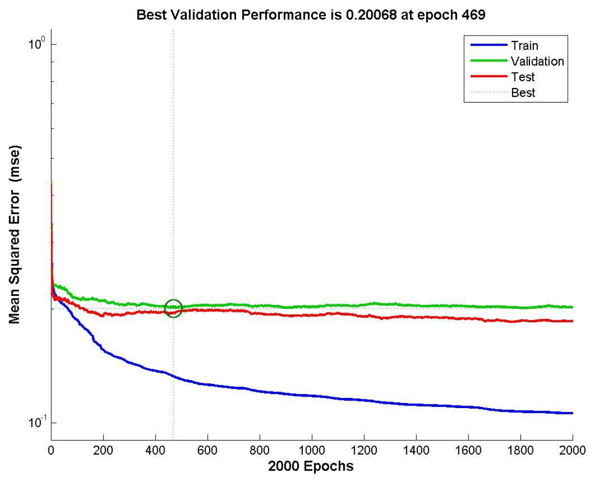


Figure –TrainSCG- All-in-one-20 hidden nodes error graph

VI. Conclusion

It seems that our goal of building a classifier which could possibly be used in the real world (e.g. For training on the game) is not yet equipped to handle all the possibilities of the game. The Build-order format does not take into account all the information such as the race matchups, for example Zerg is quite fast at expanding early the game. This type of game play strategy would factor into the build order but it does not solely determine whether you would win or not. It should also be noted that StarCraft II features combat bonuses to units which are on different types of terrain, for example high ground vs low ground gives you a much better chance of surviving.

Tuning of the parameters turned out to be quite difficult, with the majority of learning functions giving us terrible results such as severe oscillations in the error graph.

Narrowing down the choices to gradient descent algorithm with adaptive learning rate with momentum seemed to give us at least some viable results (above 65%). We believe this is because the error landscape is quite complex when it comes to build orders, although we assumed that the build orders we collected were done with perfect execution, if we were to actually factor in the time, this would drastically change the error landscape requiring even more tuning. For this reason, using an adaptive learning rate is best as it can immediately change based on the error landscape and keep the best tuned parameter after several epochs- within reason, and as long as a momentum factor is added and tuned properly this will drive the algorithm down towards a global minima.

Additionally, the game seems to be quite balanced as the data sets seems to suggest. However just because you get a really good build order right from the start if you fail to play properly during the middle and quite possibly end game, what was classified as a win in the beginning for you may turn out to be a loss.

The biggest problem we had during this project was the encoding problem, it seems as though a neural network will take the data and "group" points which are similar, for example an encoding of unique values 25 and 50 would look pretty much the same in the network. Perhaps encoding the data with unique hex values could get around the scaling issue. But in general it is really difficult to come up with a great way to force the network to treat the input values differently.

VII. Future Work

As future work we imagine to extend the size of the building orders as well as to involve in the training additional features like the map (though not always applicable since customized maps are widely used) and/or considering the race matchups. Beside the features, a deeper research on the best training algorithm to adopt as well as a more accurate parameters tuning would help in getting better results.

Appendix

References

[1] A. Spann, E. Lin, H. Kang, “ Starcraft map Imbalance Prediction Based on Chosen Build Order” December 11 2009. Last access May 28th 2012 (http://cs229.stanford.edu/proj2009/SpannLinKang.pdf)

[2] E. Cox, S. Kodesh, D. Preston, Final Project , Team 31 ” Predicting Win/Loss Records using Starcraft 2 Replay Data “ , 2010. Last access May 28th 2012 (http://snap.stanford.edu/class/cs224w-2010/proj2010/31_final_project.pdf)

[3] Php Sc2 Replay Last Access: May 28th 2012 (https://code.google.com/p/phpsc2replay/downloads/list)

[4] MatLab Neural Network Toolkit Last access May 28th, 2012(http://www.mathworks.se/products/neural-network/)

[5] MatLab Neural Network Toolkit Last access June 10th,2012 (http://www.mathworks.se/help/toolbox/nnet/ref/trainrp.html)

[6] MatLab Neural Network Toolkit Last access June 10th, 2012 (http://www.mathworks.se/help/toolbox/nnet/ref/traingd.html)

[7] Scaled Conjugate BackPropagation Algorithm http://www.ra.cs.uni-tuebingen.de/SNNS/UserManual/node243.html#SECTION0010172000000000000000

[8] MatLab Neural Network Toolkit Last access May 29th 2012, (http://www.mathworks.se/help/toolbox/nnet/ref/traingda.html)