Strategy Prediction in StarCraft: Brood War using Multilayer Perceptrons

Henrik Sørensen (hsoere01@student.aau.dk)
Johannes Garm Nielsen, (jniels06@student.aau.dk)
Aalborg University, Denmark

October 28, 2011

Aalborg University
Department of Computer Science

TITLE:
Strategy Prediction in StarCraft: Brood War using Multilayer Perceptrons

PROJECT PERIOD:
February 1st, 2011
October 28th, 2011

PROJECT GROUP:
f11d621a

GROUP MEMBERS:
Henrik Sørensen
Johannes Nielsen

SUPERVISORS:
Yifeng Zeng

NUMBER OF COPIES: 4

TOTAL PAGES: 98

SYNOPSIS:

In this project we first formulate a theory of strategy prediction in real-time strategy games based the notion that a players strategical options are dependent on the in-game assets available. We then test a number of common training algorithms to train multi-layer perceptrons to predict the in-game assets of a player at time n given information about information about the same players in-game assets at time m, $m < n$.

A novel feature selection technique based on real-time strategy game design principles is introduced and tested, to address the number of possible input features that can be extracted from in-game assets.

A series of experiments to evaluate the strategy prediction performance are performed on MLPs trained with the Backpropagation, RProp, Genetic Algorithm and two Genetic Algorithm hybrids using Backpropagation and RProp.

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Introduction

In the real-time strategy genre of video games, the role of the AI is both to provide challenge and the illusion of intelligence. While human players have the ability to predict actions of their opponents, AI opponents current lack this ability and are as such limited in the intelligence they can exhibit. In this paper we present a method for predicting the strategy of a player derived from common design features in the real-time strategy genre of video games which we argue can be applied to any game with the aforementioned design features. The predictions are done by multilayer perceptrons which are trained to use data about the in-game assets of a player to predict the same players in-game assets at a later point in the game. We implement the solution to predict on captured data from the game StarCraft: Brood War. Using the captured data multilayer perceptrons are trained to use a players assets at one point in a game to predict the assets of that player at a later point in that game.

Opponent modelling for real-time strategy games has previously been the subject of research [2, 7, 14]. In [2], Schadd et al. use a hierarchical approach to strategy modelling that attempts to classify a player strategy as one of a predetermined set of strategies. In [14] Herik at al makes a competent general AI for 2 player competitive games using game trees a analysis of the goals of AI in commercial computer games is presented. An approach to AI training using data mining of expert gameplay is documented in [7] by Weber et al.

1.2 Real-Time Strategy Games

As the name implies, the real-time strategy (RTS) genre of video games has two hallmark traits: strategic army management and real-time (as opposed to turn based) gameplay. In RTS games, the player typically takes the role of a military commander and is tasked with controlling and expanding an army.

In multiplayer games, the win condition is typically eliminating all assets belonging to an opponent though it varies from game to game. Since every RTS revolves in part around construction of an army, every RTS also has an economy system and a production system. Typically, the economy features two resources - one general resource and one advanced resource. In the RTS StarCraft: Brood War, these resources are called minerals and gas and units called workers must be assigned to gather them, while in the Dawn of War series of RTS the resources are called requisition and electricity and they are gathered by claiming points on the playing field.

Part of the challenge in RTS games is to manage resource collection and spending better than your opponent, thereby getting more in-game assets faster. Thus much of the activity in multiplayer games revolves around securing resources or denying resources, in an effort to limit how many in-game assets the opposition is capable of producing. This effort to control resources comes down to the army of the players and how well the players control the army. The units in RTS typically perform significantly better if the player manually controls them, as opposed to issuing a single command and waiting for the outcome. Ideally players would multitask well enough to

Figure 1.1: A screen shot of a base in the RTS "Command & Conquer: Red Alert 2".

both manage their resource collection and spending while also manually controlling their army since that allows for both the maximal amount of units at any given time as well as the best unit efficiency assuming the player has perfect control.

Much like strategic board games such as chess, high level play in RTS usually revolves around opening plays. By using familiar opening plays, the players increase the chance of the state the game enters after the opening being one they have been in before. This is important since the game is played in real-time - it is usually too much for a player to figure out what the best option is in an unfamiliar situation while managing the economy and controlling the army efficiently.

1.2.1 Technology Trees

One of the core design elements in RTS is the technology tree. Essentially, the technology tree is a visual representation of the relationship between buildings and units as it is designed in the game. In figure 1.2 we see the technology tree for the Protoss faction in StarCraft: Brood War. Although the concept of the technology tree is the same in most RTS the nature of the relationships it describes differs significantly from game to game. In order to provide variance, players in StarCraft: Brood War can choose from three different factions, each with a technology tree that is different from the other two. As an example, we see in figure 1.2 that the node labelled Cybernetics Core is a child of the node labelled Gateway, meaning that a Gateway is required to be present for a Cybernetics Core to be built; If a Gateway is not currently under the control of a player trying to build a Cybernetics Core, the game will not permit it being built even if enough resources are available.

Because of the relationships described in the technology tree, a players choice of buildings and units is highly significant strategically. As an example, in StarCraft: Brood War a Protoss player could opt to build a Gateway first and then a Cybernetics Core, or the player could opt for two Gateways. In the first case, having the Cybernetics Core made means that the player will be able to build a Stargate or a Robotics Facility as soon as resources become available, whereas in the second case a Cybernetics Core would have to be built first. However in the second case twice the number of Zealots could be produced at the time compared to the first case.

1.2.2 Predicting strategies in StarCraft: Brood War

StarCraft: Brood War is an economy focused RTS made by Blizzard Entertainment. Despite the games age, there is still an active competitive multiplayer community around the game, in particular in South Korea where it is has been and is a national sport for years at the time of writing. Because of this community, StarCraft: Brood War is a common subject for AI research

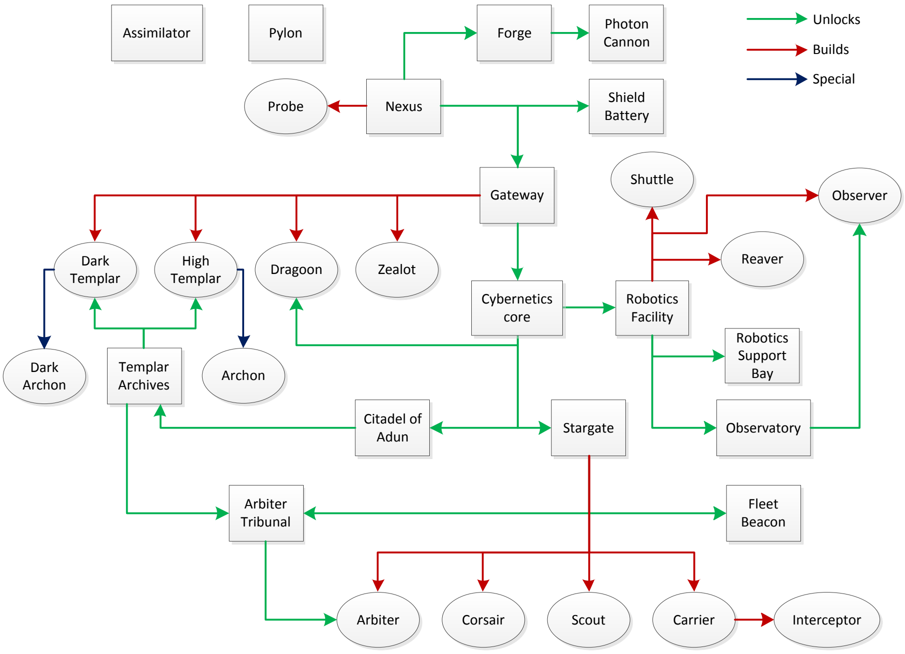


Figure 1.2: Part of the Protoss technology tree from StarCraft

and competitions since the game is very well understood at this point and an API for accessing the StarCraft: Brood War engine named BWAPI is available.

In multiplayer games of StarCraft: Brood War, the players choose one of three factions to control. These factions are Terran, Zerg and Protoss. The factions are completely unique, meaning that there are no shared units or buildings between them - each faction has a separate technology tree with a separate set of nodes. In standard competitive multiplayer matches the players start out with their factions main building and four worker units (e.g. a Nexus and four Probes for a Protoss player) and the game ends when either side has lost all buildings or surrenders. Much like chess however, most competitive games end when a player surrenders having realized that a loss is close to guaranteed.

In competitive multiplayer, players use several information sources to predict the strategy of an opponent. If they know their opponent from before the game, they might know which opening players the opponent usually favours. The playing fields used, called maps, often favour certain strategies due to their layouts - easily defended resources encourages players taking them relatively early while starting locations in close proximity to each other favour aggressive opening plays. Perhaps the most important information a player can get is from see what buildings and units an opponent has opted to invest in. Due to the technology tree restrictions, knowing what an opponent has allows a player to rule out a number of potential strategies. As an example, if a player sees an opponent has a large number of production buildings for a certain time in the game, it would be reasonable to assume the opponent is going to be producing units at a higher rate than normal and will therefore have a larger army than normal.

In StarCraft, as well as most commercial RTS, there is a large number of different in-game assets that factor into a players capabilities. For a Protoss player in StarCraft: Brood War, there are 31 unit and building types in the technology tree.

1.3 Possible applications

The application possibilities of a strategy prediction system for real-time strategy games have the potential to enable AI players to closer resemble human players and offer more interesting gameplay to the players. With the capability to predict the strategy of an opposing human player, an AI player could can counteract the strategy of the human player intelligently. As an example, consider an AI player that is capable of strategy prediction and which also has a set of strategies it can execute along with data about how effective they are against commonly observed strategies. This AI would be capable of picking a counter strategy to that of a human player opponent based on the level of challenge desired - a poor counter strategy can be chosen if a low level challenge is the goal or a good counter strategy can be chosen if a high level challenge is the goal.

By extending the prediction system described in this paper to work with partial information, a system governing what an AI using the prediction system observes can be implemented. This is an interesting prospect since this would allow a human player to hide in-game assets from an AI player and thereby influence the information available to the AI. As an example a human player could feint a specific strategy by allowing the AI player to observe information indicating this strategy and then deviate from the strategy. Another way for human players to take advantage of such a system would be to limit the observations of the AI player as much as possible, hindering its ability to make an accurate predicting.

As such we believe that the aforementioned difficulty scaling and gameplay improvements that could be achieved by means of the prediction system presented in this paper, the experience of playing against an AI opponent in RTS could be significantly improved.

1.3.1 Problem Statement

The goal of this project is to provide a means for AI players real-time strategy games to predict the strategy of an opponent, facilitating AI that more closely mimic human behaviour.

In this project we first formulate a theory of strategy prediction in real-time strategy games based the notion that a players strategical options are dependent on the in-game assets available. We then test a number of common training algorithms to train multi-layer perceptrons to predict the in-game assets of a player at time n given information about information about the same players in-game assets at time m, $m < n$.

A novel feature selection technique based on real-time strategy game design principles is introduced and tested, to address the number of possible input features that can be extracted from in-game assets.

CHAPTER

TWO

THEORY

2.1 Perceptrons

Perceptrons are a class of Artificial Neural Networks based on a unit called a perceptron unit. Perceptron units use the output function $o : \mathbb{R}^n \rightarrow \mathbb{R}$ defined as

$$o(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n > 0 \\ -1 & \text{otherwise.} \end{cases} \tag{2.1}$$

where $w_0, w_1, \ldots, w_n \in \mathbb{R}$ are called the node weights and determine the significance of each input. The threshold value $-w_0$ ensures that the linear combination of the inputs must exceed its value in other for the perceptron unit to give a positive output value.

The perceptron unit can be seen as representing a hyperplane in $\mathbb{R}^n$, in that it returns 1 if the given input $\vec{x} \in \mathbb{R}^n$ lies on one side of the hyperplane and -1 if it lies on the other side. This hyperplane is given by the equation $\vec{w}\vec{x} = 0$ where $\vec{w} = (w_0, \ldots, w_n)$.

Learning a perceptron constitutes finding the right weights and in this section we cover multiple techniques for learning the weights of a single perceptron. The task at hand is to, given a training set of inputs and outputs $D \subseteq \mathbb{R}^n \times \{-1, 1\}$, find weights such that for each $(\vec{x}, t) \in D$, $o(\vec{x}) = 1$ if $t = 1$ and $o(\vec{x}) = 0$ if $t = 0$. Thus D contains pairs of inputs and target outputs. Note that not all such sets can be divided by a hyperplane such that all positive target values are on one side and all negative target values on the other and as such we cannot expect to learn successful weights for any given training set. Training sets that can be separated by a hyperplane are called linearly separable.

The basic idea in perceptron learning is to begin with random weights and then try each input from the training set and adjust the weights if the output from the perceptron does not match that in the training set. For each input $\vec{x} = (x_1, \ldots, x_n)$ with target output t, each weight is updated using the formula

$$w_i \leftarrow w_i + \Delta w_i(\vec{x}, t). \tag{2.2}$$

We call the function Δw_i the weight update rule. In the following, we describe two candidate weight update rules.

2.1.1 The perceptron training rule

The perceptron training rule specifies a weight update rule using the formula

$$\Delta w_i(\vec{x}, t) = \eta(t - o(\vec{x}))x_i \tag{2.3}$$

where $\eta \in \mathbb{R}_+$ is the learning rate. η is usually a constant but it is sometimes set to decrease with each iteration.

The perceptron training rule has been shown to converge towards successful weights, given that a sufficiently small learning rate is chosen and the input data in the training set is linearly separable [9]. The gradient descent rule, which we describe next, does not have the latter requirement.

2.1.2 Gradient descent

The idea behind the gradient descent method is to minimize an error function over the space of potential weights, or more precisely to find $\vec{w} = \text{argmin}_{\vec{v} \in \mathbb{R}^{n+1}} E(\vec{v})$ where E is some error function. This minimization can be done by following the gradient of the error function downwards along the steepest descent until a local minimum is found.

A common error function is

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o(\vec{x}_d))^2 \tag{2.4}$$

where $\vec{x}_d$ and t_d denotes the input and associated target output of training sample d. Recall that the steepest increase of $E(\vec{w})$ is given by the gradient of $E(\vec{w})$

$$\nabla E(\vec{w}) = \left(\frac{\partial E}{\partial w_o}, \ldots, \frac{\partial E}{\partial w_n} \right). \tag{2.5}$$

Thus we can follow the steepest descent as intended by using the technique described in section 2.1 with the weight update rule

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{2.6}$$

where $\eta \in \mathbb{R}_x$ is the learning rate.

Notice that the weight update rule shown above requires that we calculate the gradient in each iteration of the learning process. Each gradient component is given by the formula

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \tfrac{\partial}{\partial w_i} \tfrac{1}{2} \textstyle\sum_{d \in D} (t_d - o(\vec{x}_d))^2 && (2.7) \\ &= \textstyle\sum_{d \in D} (t_d - o(\vec{x}_d))(-x_{id}) && (2.8) \end{aligned}$$

where x_{id} is the ith component of $\vec{x}_d$. This gives us the weight update rule

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o(\vec{x}_d)) x_{id} \tag{2.9}$$

2.2 Multilayer Perceptrons

The perceptrons described in section 2.1 can only express linear decision surfaces or hyperplanes, and even though compositions of perceptrons can be used with the same expressive power as most boolean operations, they can still only represent linear decision surfaces. We often require a model that can represent non-linear decision surfaces in order to learn arbitrary groupings.

Multilayered Perceptrons consists of multiple layers of units, such that the first layer works as the input units and the last layer is the perceptrons output units. The units in the intermediate layers are called the hidden units. Each unit in each layer takes as it's input the vector consisting of the outputs from all units in the proceeding layer. This structure is called a feed-forward network.

We need a non-linear unit to replace the perceptron unit from before. This new units output must be a continuous function of its inputs such that it can be differentiated for gradient descent learning of linearly inseparable training sets. One such output function is

$$o(\vec{x}) = \sigma(\vec{w} \cdot \vec{x}) \tag{2.10}$$

where

$$\sigma(r) = \frac{1}{1 + e^{-r}} \tag{2.11}$$

where σ is called the sigmoid function. This output function will give us outputs ranged between 0 and 1.

Like the techniques we discussed in section 2.1 for learning the weights in single-unit perceptrons, the Backpropagation method for learning the weights in a MultiLayered Perceptron (MLP) involves minimizing an error function. The error function we use is similar to 2.4 but modified to sum the errors of all the output nodes.

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{k_d} - o_{k_d})^2 \tag{2.12}$$

where outputs is the set of output units, o_{k_d} is the output for unit k when given input from training sample d and t_{k_d} is the target output of node k associated with training sample d.

Where the error function in 2.4 only has one minimum, the one in 2.12 can have multiple local minima. A consequence of this is that a gradient descent technique might only find a local minimum rather than the global minimum. However the Backpropagation method, which is a gradient descent technique, has been shown to get good results in many applications [9]. One method for compensating for the presence of local minima is momentum, which will be discussed later.

The Backpropagation algorithm works as follows:

Algorithm 1: Backpropagation algorithm [9] p.98

```
   Input: $Network$, $T$, $\eta$
   $Network$ is a Feed forward Neural Network using sigmoid units with $n_{in}$ input nodes, $n_{out}$
   output nodes, $n_{hiddennodes}$ hidden nodes, and $n_{hiddenlayers}$ hidden layers.
   $T$ is a set of training samples, where each sample $\langle \vec{x}, \vec{t} \rangle \in T$ consist of an input vector $\vec{x}$ and
   the corresponding target output vector $\vec{t}$.
   $\eta$ is a learning rate parameter
1  Initialise every weight $w_{ji}$ in $Network$ to small random numbers, where $w_{ji}$ is the weight
   between neuron $i$ and $j$
2  while termination condition is not met do
3      foreach $\langle \vec{x}, \vec{t} \rangle \in T$ do
4          Input $\vec{x}$ to the input units and compute the output of every unit in the network.
5          foreach output unit $k$ in output layer do
6              $\delta_k = o_k(1 - o_k)(t_k - o_k)$
7              where $o_k$ is the output of $k$ and $t_k$ is the target output of $k$ as given in $\vec{t}$.
8          end
9          foreach hidden unit $h$ in hidden layers do
10             $\delta_h = o_h(1 - o_h) \sum_{k \in \text{next layer}} w_{kh} \delta_k$
11             where $o_h$ is the output of $h$, $w_{kh}$ is the weight of the input from $k$ in $h$ and $\delta_k$ is
               the error term of $k$.
12         end
13         foreach $w_{ji}$ in $Network$ do
14             $\Delta w_{ji} = \eta \delta_j x_{ji}$ , where $x_{ji}$ is the input value from neuron $i$ to neuron $j$
15             $w_{ji} = w_{ji} + \Delta w_{ji}$
16         end
17     end
18 end
```

To help overcome local minima, we can make the update of the weights in one iteration depend partially on the updates that occurred at the previous iteration. This is referred to as adding momentum to the Backpropagation To do this, we replace the update rule on line 14 of the algorithm with

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \tag{2.13}$$

where $\Delta w_{ji}(n)$ is the weight update of iteration n and $0 \leq \alpha < 1$ is the momentum constant.

Adding momentum to the Backpropagation has an effect analogous to the effect of momentum on a ball rolling down a slope. It will let the gradient descent overcome comparably shallow local minima, however there is no guarantee that it will not overcompensate and thus render the algorithm incapable of finding the global minimum.

2.3 Resilient Propagation

As discussed in section 2.2 the back propagation algorithm does require one to chose suitable learning rate and momentum parameters. If one chooses too small parameters convergence towards a minimum will take an undesirable amount of weight updates, and too large parameters will lead to oscillation around the minimum. Various modifications of the basic back propagation algorithm have been proposed to adapt the parameters to the specific problem during optimisation However, as Riedmiller and Braun [11] points out, most of these proposal disregard the effect of varying size of the partial derivative in the weight update rule, seen below:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E}{\partial w_{ji}} + \alpha \Delta w_{ji}(n-1)$$

A carefully adapted learning rate η and momentum α can suddenly be less than optimal depending on the behaviour of the partial derivative. Therefore in Riedmiller and Braun's RProp algorithm, also referred to as Resilient Propagation, only considers the sign of the partial derivative, i.e. whether the error related to weight w_{ji} is decreasing or increasing.

RProp uses a individual weight update value w_{ji} to vary the size of the weight change, which is determined as:

$$\Delta_{ji}^{(t)} = \begin{cases} \eta^{+} \cdot \Delta_{ji}^{t-1}, \text{ if } \frac{\partial E}{\partial w_{ji}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ji}}^{(t)} > 0 \\ \eta^{-} \cdot \Delta_{ji}^{t-1}, \text{ if } \frac{\partial E}{\partial w_{ji}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ji}}^{(t)} < 0 \\ \Delta_{ji}^{t-1} \end{cases} \tag{2.14}$$

where $0 < \eta^{-} < 1 < \eta^{+}$

The intuition for the step size rule is if the partial derivative changes its sign, the step taken last iteration was too large, and thus a smaller one needs to taken next time, if the sign stays the same a larger step is taken to speed up convergence towards the minimum.

Once the step size has been determined, the weight delta for this iterations is given by:

$$\Delta w_{ji}^{(t)} = \begin{cases} \text{sign}(\Delta_{ji}^{(t)}), \text{ if } \frac{\partial E}{\partial w_{ji}}^{(t)} \neq 0 \\ 0, \text{else} \end{cases} \tag{2.15}$$

$$w_{ji}^{(t+1)} = w_{ij}^{(t)} + \Delta w_{ij}^{(t)} \tag{2.16}$$

However, if partial derivative changed sign, it means that weight has been pushed past the minimum and the previous weight change was too large, therefore the change is undone with the following rule:

$$\Delta w_{ji}^{(t)} = -\Delta w_{ji}^{(t-1)}, \text{ if } \frac{\partial E}{\partial w_{ji}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ji}}^{(t)} < 0 \tag{2.17}$$

The sign function used in the weight update rule 2.15 is defined as:

$$sign(n) = \begin{cases} 1, \text{ if } n > 0 \\ -1, \text{ if } n < 0 \\ 0, \text{ if } n = 0 \end{cases} \tag{2.18}$$

Putting it all together one gets the algorithm in 2. The parameters Δ_{max} and Δ_{min} are introduced to cap the minimum and maximum step size taken, values suggested by the authors are 0.000001 and 50, and a initial Δ_{ji} value of 0.1.

Research [11], [5] shows that Resilient Propagation out performs other propagation methods such as Back Propagation, SuperSAB, Quickprop and the conjugate gradient method (CG), both on constructed test cases and in some real world data tests. However, as the weight rules focus entirely on the sign of the gradient, the algorithm will zero in on the nearest local minimum with no mechanism such as Back Propagation's momentum to nudge itself out of a small local minimum.

Algorithm 2: RProp algorithm with weight backtracking [11]

```
Input : $Network$
$Network$ is a Feed forward Neural Network using sigmoid units with $n_{in}$ input nodes, $n_{out}$ output nodes, $n_{hiddennodes}$ hidden nodes, and $n_{hiddenlayers}$ hidden layers.
foreach $w_{ji}$ do
    if $\frac{\partial E}{\partial w_{ji}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ji}}^{(t)} > 0$ then
        $\Delta_{ji}^{(t)} = min(\Delta_{ji}^{(t-1)} \cdot \eta^{+}, \Delta_{max})$
        $\Delta w_{ji}^{(t)} = -sign(\frac{\partial E}{\partial w_{ji}}^{(t)}) \cdot \Delta_{ji}^{(t)}$
        $w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$
    end
    else if $\frac{\partial E}{\partial w_{ji}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ji}}^{(t)} < 0$ then
        $\Delta_{ji}^{(t)} = max(\Delta_{ji}^{(t-1)} \cdot \eta^{-}, \Delta_{min})$
        $w_{ji}^{(t+1)} = w_{ji}^{(t)} - \Delta w_{ji}^{(t-1)}$
        $\frac{\partial E}{\partial w_{ji}}^{(t)} = 0$
    end
    else if $\frac{\partial E}{\partial w_{ji}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ji}}^{(t)} = 0$ then
        $\Delta w_{ji}^{(t)} = -sign(\frac{\partial E}{\partial w_{ji}}^{(t)}) \cdot \Delta_{ji}^{(t)}$
        $w_{ji}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ji}^{(t)}$
    end
end
```

2.4 Using Genetic Algorithms to Train Artificial Neural Networks

While the Backpropagation algorithm has been used successfully to train ANNs for decades, it does have a drawback. The gradient descent method as the name suggests locates a minimum by taking steps proportional to the negative of the gradient However, there is no guarantee that the located minimum is the global minimum of the function being optimised, as the algorithm essentially gets stuck in whatever minimum it has descented into. [9]

Just as gradient descent, genetic algorithms are a general approach to searching for a good candidate in a hypothesis space. The name, genetic algorithms, stems from the inspirations that led to their creation - the natural selection and mutation occurring in nature. A basic example of a genetic algorithm can be described in the following steps, where a predefined fitness function determines the appropriateness of a selected hypothesis:

2.4.1 Representation of hypothesis and operators

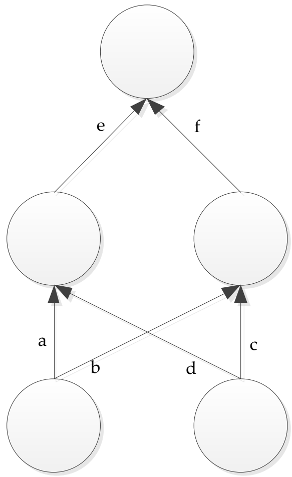
The original genetic algorithm expressed hypothesises as strings of bits, with non-boolean variables encoded as a section of the bit string. According to Yao [16] there is research that suggests the binary representation might not be the optimal encoding. In the context of using Genetic Algorithms for weight training of Artifical Neural Networks (ANN), real valued hypothesises have been used with success in many experiments [10] [16]. Each hypothesis is expressed as a vector of real values, where each real value is a connection weight.

Selection Operator

The selection of the hypothesises that are allowed to reproduce in line 4 is generally carried out probabilistically in a way such that the chance of a hypothesis being selected is related to fitness value of the hypothesis under consideration. The number of hypothesises selected in line 4 and the number of offspring produced in line 5 is balanced such that a set fraction of the population is

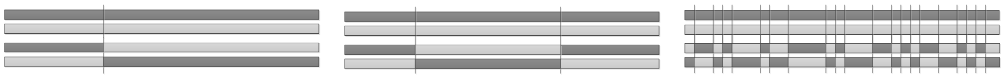
Algorithm 3: A Basic Genetic Algorithm [9] p.251

```
Input  : $p$, $cr$, $m$
$p$ is the size of the genetic population.
$cr$ is the fraction of $P$ to be replaced with new hypothesises created by crossover.
$m$ is the fraction of $P$ to be mutated.
Output: The hypothesis with the best fitness value found.
Start with a random population $P$ with $p$ hypothesises.
Evaluate: foreach Hypothesis $h$ in $P$ do find $fitness_h$
while termination condition not met do
    Selection: Probabilistically select $(1 - cr)p$ hypothesises from $P$ and add them to $P_S$.
    Crossover: Probabilistically select $\frac{cr\cdot r}{2}$ pairs of hypothesises from $P$ and produce
      offspring from these pairs by applying a crossover operator. Add the offspring to $P_S$.
    Mutation: Select $m \cdot p$ hypothesises from $P_S$ and apply a mutation operator to them.
    New Generation: $P = P_S$
    Evaluate: foreach Hypothesis $h$ in $P$ do find $fitness_h$
end
return hypothesis from $P$ with the best fitness.
```



	Binary	Real
a	00101110 01100110	0.1
b	00111000 00000000	0.5
c	10111010 00000000	-0.75
d	00111101 01011001	1.337
e	10111011 11010111	0.98
f	01000100 01010100	4.33

Figure 2.1: Two possible encodings of the weights in the illustrated ANN. Binary encoding using half precious floating point, and real valued weights.



(a) One Point Crossover (b) Two Point Crossover (c) Uniform Crossover

Figure 2.2: Crossover Examples

replaced by new offspring each iteration. A constant r controls the fraction of the population that is replaced by offspring such that $(1-r)|P|$ members of P is selected in line 4. In line 5, $\frac{r|P|}{2}$ pairs of hypothesises is chosen and two offspring is produced for each pair.

One commonly used selection method is the fitness proportional selection method, often also refereed to as roulette wheel selection. The chance of a hypothesis being selected is given by the probability $Pr(h) = \frac{h.fitness}{\sum_{h_i \in P} h_i.Fitness}$. The whole selection method can be seen in algorithm 10.

```
Algorithm 4: Roulette Wheel Selection Operator
Input  : $P$, $n$
$P$ is the population to select from
$n$ is number of hypothesisses to select
Output: $S$ set of selected hypothesises
$P_{sorted}$ = $P$ sorted in descending order of fitness
while $\|S\| < n$ do
    select random value $roll$ in the interval $[0, 1]$
    repeat
        $h$ = next member of $P_{sorted}$
        $accumulated = accumulated + \frac{h.fitness}{\sum_{h_i \in P_{sorted}} h_i.fitness}$
    until $accumulated < roll$
    $S = S \cup h$
    $P_{sorted} = P_{sorted} \setminus h$
end
```

Crossover Operator

The reproductive step of line 5 in the general genetic algorithm described earlier requires a crossover operation. This biologically inspired process takes two parent hypothesises and combines the genome of each parent to produce a new offspring hypothesis. Some common crossover operations for bit strings are shown in figure 2.2.

Mutation Operator

To improve the search efficiency, new genes that necessarily a part of the populations' genome are introduced with an mutation operator With a binary representation the mutation is relatively simple, flip a given number of bits in each hypothesis chosen to be mutated. Montana and Davis [10] suggest some mutation operators that work on real value representations, by selecting a set of weights based on a criterion and then either replacing the weight with a random value or adding a random value to the current weight. They show that adding a value to the weight instead of replacing the weight entirely performs better. They posit this is because the current weight has been chosen through generations of well performing weights. The current weight will therefore perform better than the average newly randomly chosen weight, therefore it is better to centre the mutation around the current weight than 0. The weight selection criteria used in their tests are either selecting a number of random weights, selecting weights belonging to a number of neurons, or targeting the weights of weak neurons for mutation. The strength of a neuron is defined as the difference between the performance of the network with neuron enabled and disabled by setting its input weights to zero.

2.4.2 Hybrid Training

Yao [16] points out in his overview article that are a lot of conflicting results on whether global search using evolution algorithms (EA), such as a Genetic Algorithm, or local searches, e.g. back propagation, perform better at finding ANN connection weights. He attributes it to the difference in the particular details of the global and local search methods used and the problem used to compare the search methods. One approach that has been used with success is to combine a global and local search method when training ANN weights. The aim is to avoid the downsides each method, EA have trouble fine tuning weights quickly while local searchs tend to get stuck in a local minimum. The use of a Genetic Algorithm and a Back Propagation algorithm is a common combination [8, 16] that has encouraging results in a lot of experiments. However, Kitano [6] found in his testing that as the network complexity rose even the GA-BP combination was out performed by the Quickprop algorithm.

CHAPTER
THREE

PREDICTION

This chapter begins with an explanation of our prediction methodology and limitations in section 3.1. In section 3.2 we go over the theory behind the feature selection method used and in section 3.3 the handling of data through out the data extraction, MLP training and prediction processes is outlined.

3.1 Prediction Overview

The central concept in our strategy prediction method is that the potential strategies a player can execute is dictated by the units and buildings available, which means that if we know what units and buildings a player has available at a certain time the strategies that player can execute can be inferred. To allow for brevity we refer to the units and buildings belonging to a player at time t during a game of StarCraft: Brood War as the *game state* of that player and the values of buildings and units in the *game state* as *game state variables*.

Definition 1 (Game State) *We define a game state as containing the following:*

- *The time,* ***t****, at which the game state was captured as an integer.*
- *The number of each unit and building type observed as a set of integers, the* ***game state variables****.*

During the course of a competitive game of StarCraft: Brood War, several factors are introduced which makes prediction more complex. As the game progresses players will in general expand their economies and advance through the technology tree. A consequence of this is that prediction becomes more complex, as the player has more resources to allocate, more ways to spend resources and a higher chance of player error due to the more complex situation. Additionally the chance of a player losing buildings and units increases as the game progresses, making it harder to predict future game states.

The presence of losses in the data presents an interesting choice: Should losses of units and buildings be reflected in Game states? As stated in 1.2, players of RTS will commonly stick to a small set of strategies because of the advantage of familiarity. Because the strategic options of a player is dictated by the Game State of the player, the same strategy will usually work towards a certain goal game state. If the purpose of our predictions is to identify the goal state a player is working towards, any losses the player sustains has the potential to act as noise. Alternatively, if the purpose of our predictions is to identify the exact game state a player will be in at a later point, knowing the exact game state at present could be beneficial.

Experiments comparing the performance of both game state variants are documented in this paper.

Definition 2 (Prediction) *We define a prediction to be a real number corresponding to a real number in the target game state.*

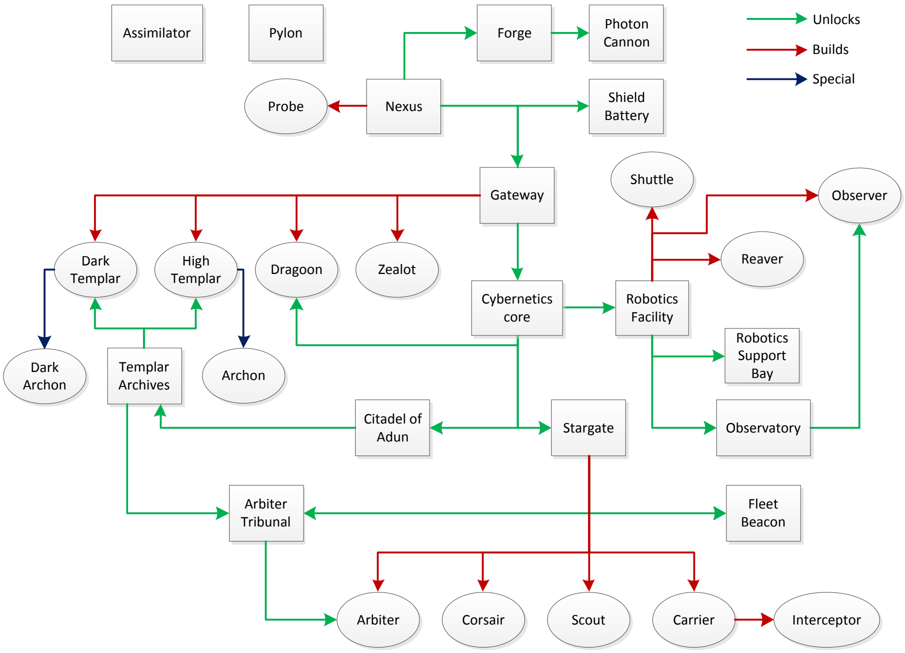


Figure 3.1: The Technology Tree for the Protoss faction.

3.2 Feature selection using Technology Trees

The feature selection employed in our experiments relies on knowledge of the technology tree of the three factions in StarCraft: Brood War.

As can be seen from the tree structure in figure 3.1, a parent node either unlocks or produces its child nodes. Thus for a unit or building of a certain type to be made, buildings of all its parent types must be present. While parents in *unlocks* connections only need to be present for the child to be built, parents in *builds* connections are occupied for a certain amount of time in order to produce a single unit or building of the child type. For these reasons, the parents in the technology tree of a given node are good candidates for input features.

Less obvious candidates for feature selection include the variables related to resource gathering and variables indicating alternate technologies. Each playable faction has 2 variables related to resource gathering - a worker and a resource dump. For the Protoss faction, the variables related to resource gathering are the *Probe* and *Nexus* variables. Since these variables indicate the income of a player, they also indicate how many resources a player can spend in the timespan between the data point and time we predict at. Since technology requires an investment of resources by the player, effective strategies often rely on few technology buildings in the early game. If an investment in technology other than the parents of the variable predicted on is observed it indicates the player intends to allocate resources to something unrelated to the variable predicted on.

As an example, consider the feature selection in figure 3.2. This feature selection is meant for predicting the number of Dragoons present at a certain time during a game and contains the parents of the Dragoon node (Gateway and Cybernetics Core), the Protoss faction economy nodes (Probe and Nexus) and four Alternate Technology nodes (Zealot, Citadel of Adun, Robotics Facility and Stargate). The importance of the Gateway and Cybernetics Core inputs derives from the fact that they are requirements for the construction of Dragoons. If no Cybernetics Core is present at the time of the input game state, then it will have to have been built and subsequently Dragoons would have to have been built for there to be an increase in Dragoons by the time of the target game state.

The economic nodes are similar, in that all units and building require resources to be spent. It may be that the player has enough Gateways to produce 6 Dragoons between the input time and the target time, but only has enough income to make 5.

The Alternate Technology nodes are meant to provide information about investments in tech-

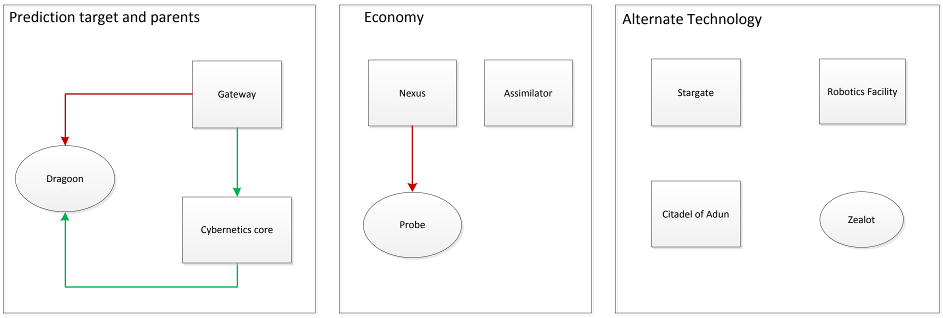


Figure 3.2: Example feature selection for predicting on the Protoss Dragoon Unit.

nology that is not related to the variables predicted on. In this case, the Robotics Facility and Stargate buildings are unit producing structures and could limit the amount of Dragoons made since they require resources to make and to produce units. Since the Zealot is also a child of the Gateway node via a *builds* connection, it could be the players strategy to make Zealots with his Gateways instead of Dragoons and thus knowing how many zealots a player already has is useful. Lastly the Citadel of Adun has a *unlocks* connection with the Templar Archives building, which in turn has *unlocks* connections with the High Templars and Dark Templar units. These two units are also children of the Gateway node through a *builds* connection and could therefore indicate that the Gateways are not going to be making Dragoons.

3.3 Prediction Pipeline

This section describes how data is extracted from StarCraft and used throughout the MLP training and prediction processes. As discussed in section 3.1, the training data we use is in the form of *game states* which we have to extract from StarCraft replays. In video game terminology, a replay is a file containing most information about the events of a single match or level. In Real-Time Strategy games replays normally contain all the necessary information to recreate a multiplayer match. As such, the data need to train our prediction MLPs can be generated from replays.
Replays in StarCraft: Brood War are essentially records of player actions rather than actual game states. Because the game engine executes deterministically, in order to play a replay the game simply interprets the recorded actions in the replay as player actions, leading to the exact same course of events as transpired in the original game. In order to get information about the game state throughout a replay, we made replay dumper plug in using BWAPI, an API for interfacing with the StarCraft: Brood War game engine, that can monitor unit and building numbers from the StarCraft: Brood War engine as it is playing a replay. The game states extracted from a replay is stored on disc separately from the remainder.

3.4 Creating Data from StarCraft Replays

StarCraft replays only contain information about the players' actions, so to get the state of a game at time t one need to play it back using the StarCraft engine. BWAPI [12] is a library that provides an API to interface with the StarCraft: Brood war client memory. It allows a developer to access the state of the game and issue orders to units to play the game as a player. In the previous semester [1] the authors developed a plug in using BWAPI to dump the state of the game every time it changes. By a game state change it meant every time a unit or building is created or destroyed in the game. Upon the end of the game replay, all this state information is then dumped into a file using a custom file format. This plug in was improved by adding support for non-protoss players and players that observe a game by joining it as a player.

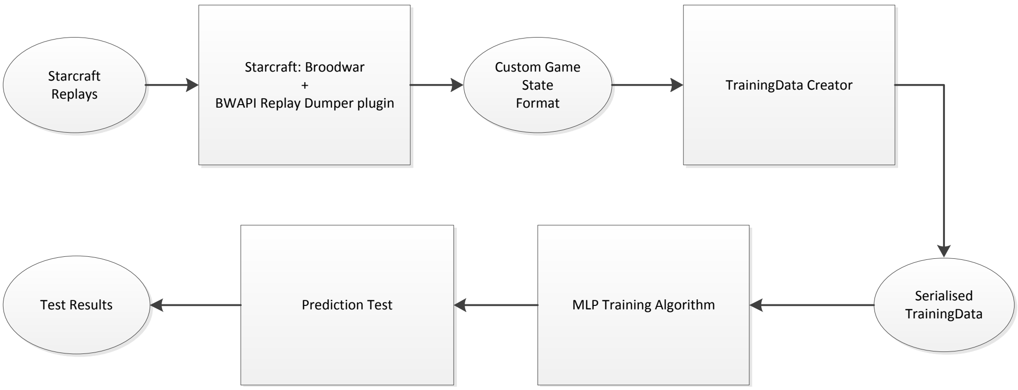


Figure 3.3: The data flow from data extraction through prediction testing.

3.5 Creating Training Data

The actual parsing of the generated game state data is done by a slightly improved version of the parsing library written in C# from the authors' last semester [1]. Our training data creator program turns this game state information into a .NET serialised instance of the TrainingData class, with using the classes seen in figure 3.4. The game state data is loaded into a series of instances of the TrainingInstance class, with only the information about the state of the game for every *interval* seconds. Each TrainingInstance instance represents a strategy taken by a player against an opponent, expressed as a series of time and game state pairs as explained in section 3.1. Each replay of a two player game will produce two TrainingInstances. The TrainingInstanceCollection acts as a container class for TrainingInstances that is mainly responsible for loading and storing the game state data.

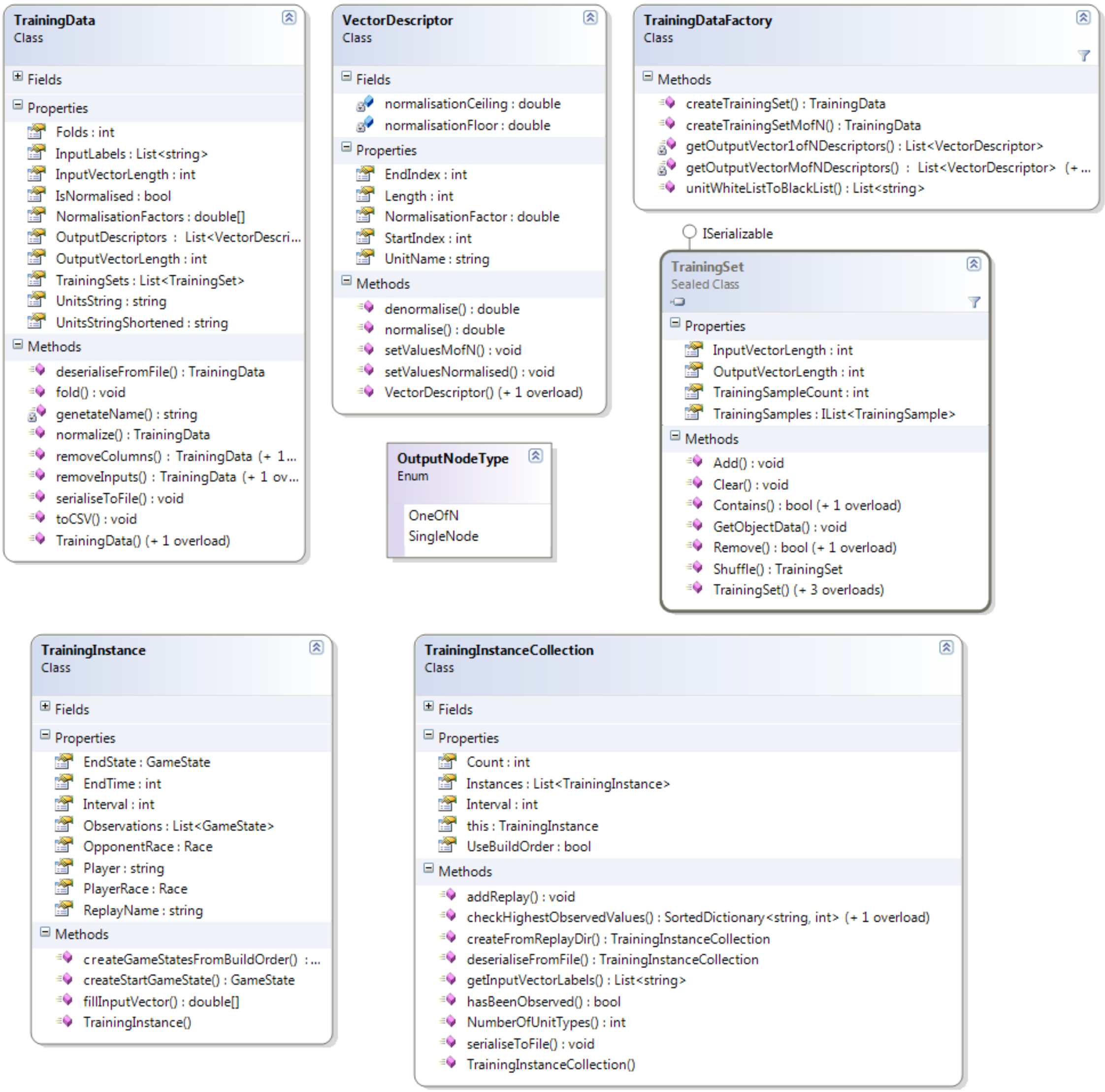
Once a TrainingInstanceCollection has been created, the TrainingDataFactory's methods can be used to create a TrainingData object. The arguments given to the factory method determine which data is put into created TrainingData object. The actual TrainingData object consists metadata properties about the games included, which inputs and outputs are included and how they are encoded, and the actual data. The training data is stored as input and output arrays of double precision floating point numbers encapsulated in a TrainingSet object. If cross validation is desired, factory method will create split the data into multiple TrainingSet object. The TrainingSet class is taken is a part of the NeuronDoNet [15] framework. Finally the constructed TrainingData object is serialised and compressed using the gzip algorithm [3, 13]. The compression step reduces the on disk size of the serialised TrainingData object by 10 to 20 times.

3.5.1 Input and Output Encoding

Each value in the input vector is either an amount of how many units of a certain type existed in the game state, or the game time in seconds for that game state. There are two possible output encodings, either using a single output per predicted unit, or using a 1 of n representation for each unit. Given a game state with an amount m of a particular unit, each output value o_i where $i \neq m$ is set to 0, and o_m is set to 1. When using the 1 of n encoding the difference in the highest scoring output value and the next highest scoring output can be seen a confidence measure in the prediction.

3.5.2 Input and Output Normalisation

As the neurons with a sigmoid activation function used in the experiments are only able to output values in the range of $]0, 1[$, the training output values have to be in that range. However, because the sigmoid function has two asymptotes at $y = 0$ and $y = 1$, as illustrated in figure 3.5, it is hard for the training algorithms to optimise the input weights for the entire output range. Therefore



TrainingData
Class

⊞ Fields

⊟ Properties
- Folds : int
- InputLabels : List<string>
- InputVectorLength : int
- IsNormalised : bool
- NormalisationFactors : double[]
- OutputDescriptors : List<VectorDescri...
- OutputVectorLength : int
- TrainingSets : List<TrainingSet>
- UnitsString : string
- UnitsStringShortened : string

⊟ Methods
- deserialiseFromFile() : TrainingData
- fold() : void
- genetateName() : string
- normalize() : TrainingData
- removeColumns() : TrainingData (+ 1...
- removeInputs() : TrainingData (+ 1 ov...
- serialiseToFile() : void
- toCSV() : void
- TrainingData() (+ 1 overload)

VectorDescriptor
Class

⊟ Fields
- normalisationCeiling : double
- normalisationFloor : double

⊟ Properties
- EndIndex : int
- Length : int
- NormalisationFactor : double
- StartIndex : int
- UnitName : string

⊟ Methods
- denormalise() : double
- normalise() : double
- setValuesMofN() : void
- setValuesNormalised() : void
- VectorDescriptor() (+ 1 overload)

TrainingDataFactory
Class

⊟ Methods
- createTrainingSet() : TrainingData
- createTrainingSetMofN() : TrainingData
- getOutputVector1ofNDescriptors() : List<VectorDescriptor>
- getOutputVectorMofNDescriptors() : List<VectorDescriptor> (+ ...
- unitWhiteListToBlackList() : List<string>

OutputNodeType
Enum

- OneOfN
- SingleNode

ISerializable

TrainingSet
Sealed Class

⊟ Properties
- InputVectorLength : int
- OutputVectorLength : int
- TrainingSampleCount : int
- TrainingSamples : IList<TrainingSample>

⊟ Methods
- Add() : void
- Clear() : void
- Contains() : bool (+ 1 overload)
- GetObjectData() : void
- Remove() : bool (+ 1 overload)
- Shuffle() : TrainingSet
- TrainingSet() (+ 3 overloads)

TrainingInstance
Class

⊞ Fields

⊟ Properties
- EndState : GameState
- EndTime : int
- Interval : int
- Observations : List<GameState>
- OpponentRace : Race
- Player : string
- PlayerRace : Race
- ReplayName : string

⊟ Methods
- createGameStatesFromBuildOrder() : ...
- createStartGameState() : GameState
- fillInputVector() : double[]
- TrainingInstance()

TrainingInstanceCollection
Class

⊞ Fields

⊟ Properties
- Count : int
- Instances : List<TrainingInstance>
- Interval : int
- this : TrainingInstance
- UseBuildOrder : bool

⊟ Methods
- addReplay() : void
- checkHighestObservedValues() : SortedDictionary<string, int> (+ 1 overload)
- createFromReplayDir() : TrainingInstanceCollection
- deserialiseFromFile() : TrainingInstanceCollection
- getInputVectorLabels() : List<string>
- hasBeenObserved() : bool
- NumberOfUnitTypes() : int
- serialiseToFile() : void
- TrainingInstanceCollection()

Figure 3.4: Classes involved in generating training data for the prediction pipeline

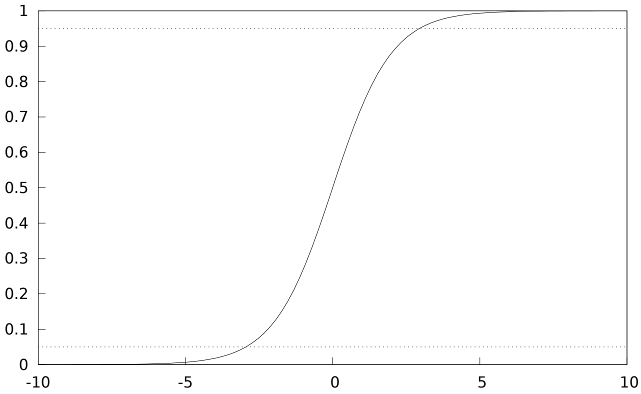


Figure 3.5: The sigmoid activation function: $\frac{1}{1+e^{-x}}$

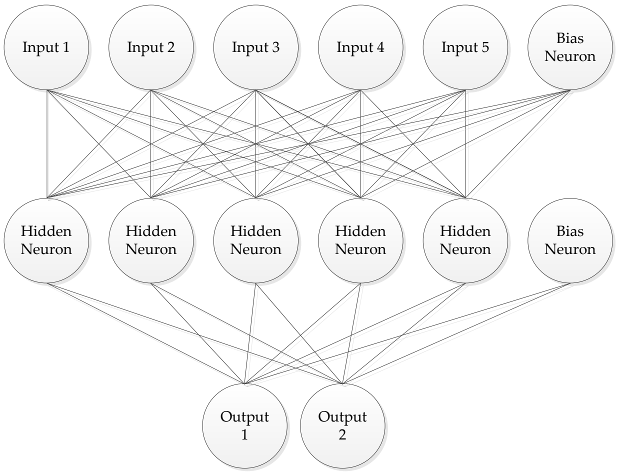


Figure 3.6: Example MLP used in Experiments

the output values are normalised in the range of $[0.05, 0.95]$. The input values are normalised into the range of $[0, 1]$. For both input and output normalisations, the factors are stored in the created TrainingData object, so values can be denormalised again.

3.6 MLP Training

The experiments are performed with an MLP that uses an input layer using linear activation functions, a hidden layer and an output layer, both using sigmoid activation functions. The input and hidden layer each contain a bias neuron. This is a neuron that always outputs the value 1 and is unconnected to any neurons in the previous layer. As the input from bias node is always 1, the input weight applied to it essentially shifts the output value's centre, i.e. to output a value close to 0 when the input is near 1. This allows the neuron to adapt to a constant component in its input. An example of such an MLP is shown in figure 3.6.

The experiment framework initially used NeuronDotNET 3.0 [15] as its MLP implementation which is an open sourced (GPLv3), easy to understand MLP library with a Back Propagation training algorithm. However, the way the training algorithm is designed means it's hard to add another training algorithm to it cleanly, and the design of the Feed Forward Network classes means causes performance penalties. The library was refactored by the authors. The Back Propagation algorithm was put into its own class to ease the use of alternative training algorithms. And the bias threshold value on each neuron was changed into a bias neuron for each non-output layer, this means bias values become just another a input weight and training algorithms only need a single weight update rule, rather than a separate rule for weights and bias thresholds.

The Encog DotNet Neural Network Framework [4] contains a highly optimised Feed Forward Network and Back/Resilient Propagation implementation, a test experiment using our framework shows that Back Propagation is roughly $8\frac{1}{2}$ times as fast using Encog compared to NeuronDotNet.

To hide the complexities of multiple MLP types and training algorithms two interfaces were defined ITrainer for training algorithms and IFeedForwardNetwork for the two MLP types. The definition of both interfaces and the classes that implement them can be seen in figure 3.7. MLPs and Training Algorithms are not completely interchangeable Breaking the tight coupling between Encog's array based FlatNetwork MLP and its training algorithms would introduce performance reducing indirection and not to mention a lot of refactoring work. Therefore training algorithms deriving from the BaseEncogTrainer class require a EncogMLP network, likewise classes deriving from the BaseNeuronDotNetTrainer require a NeuronDotNetMLP network.

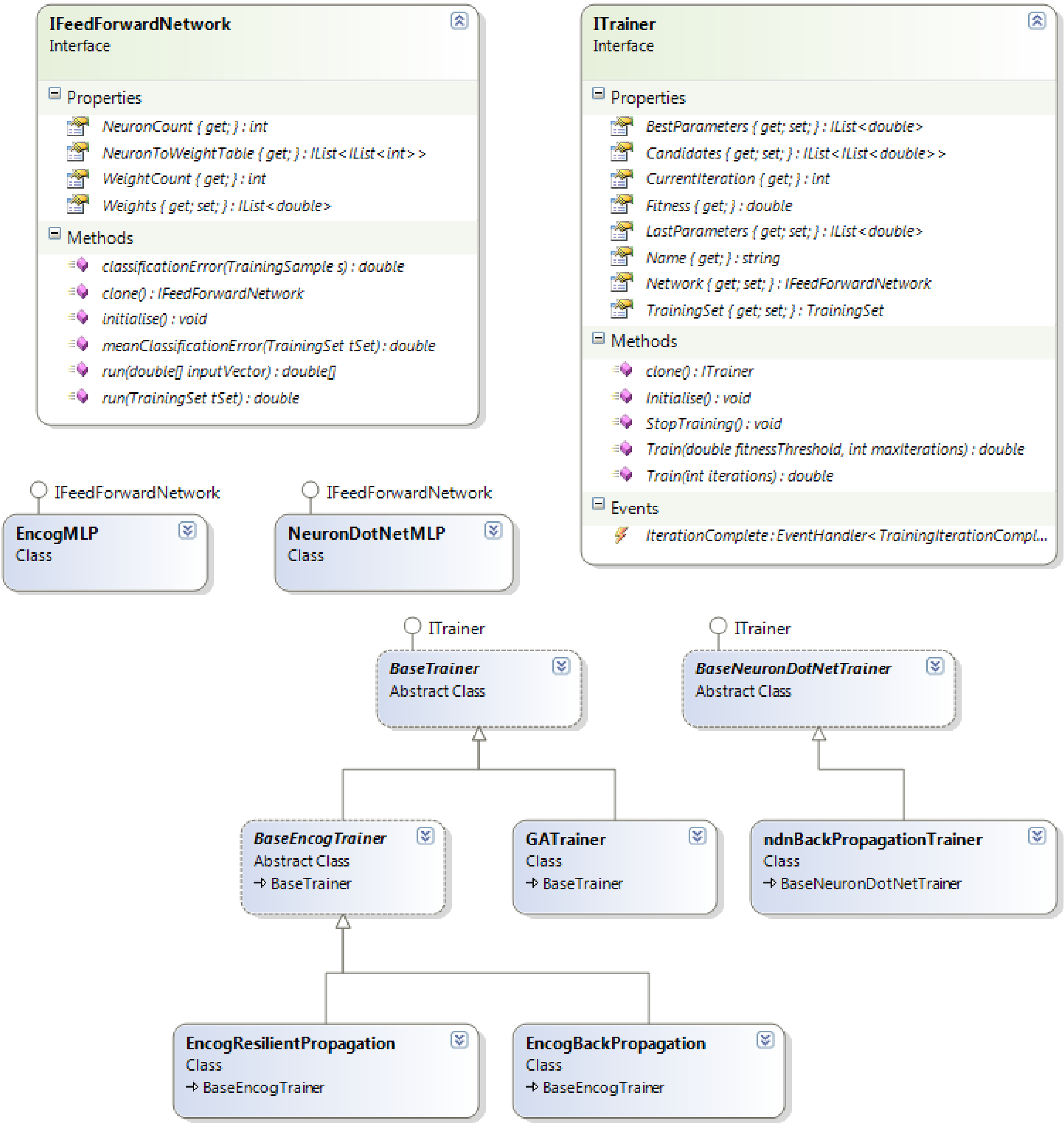

IFeedForwardNetwork
Interface
Properties
NeuronCount { get; } : int
NeuronToWeightTable { get; } : IList<IList<int>>
WeightCount { get; } : int
Weights { get; set; } : IList<double>
Methods
classificationError(TrainingSample s) : double
clone() : IFeedForwardNetwork
initialise() : void
meanClassificationError(TrainingSet tSet) : double
run(double[] inputVector) : double[]
run(TrainingSet tSet) : double
ITrainer
Interface
Properties
BestParameters { get; set; } : IList<double>
Candidates { get; set; } : IList<IList<double>>
CurrentIteration { get; } : int
Fitness { get; } : double
LastParameters { get; set; } : IList<double>
Name { get; } : string
Network { get; set; } : IFeedForwardNetwork
TrainingSet { get; set; } : TrainingSet
Methods
clone() : ITrainer
Initialise() : void
StopTraining() : void
Train(double fitnessThreshold, int maxIterations) : double
Train(int iterations) : double
Events
IterationComplete : EventHandler<TrainingIterationCompl...
IFeedForwardNetwork
EncogMLP
Class
IFeedForwardNetwork
NeuronDotNetMLP
Class
ITrainer
BaseTrainer
Abstract Class
ITrainer
BaseNeuronDotNetTrainer
Abstract Class
BaseEncogTrainer
Abstract Class
→ BaseTrainer
GATrainer
Class
→ BaseTrainer
ndnBackPropagationTrainer
Class
→ BaseNeuronDotNetTrainer
EncogResilientPropagation
Class
→ BaseEncogTrainer
EncogBackPropagation
Class
→ BaseEncogTrainer

Figure 3.7: MLP and Training Algorithm Classes.

3.7 Genetic Algorithm implementation

The genetic algorithm used in the experiments is presented in pseudo code in algorithm 5 and its class structure is shown in figure 3.8. It is made reusable by operating on a generic IHypothesis type. If the Hypothesis class is thread safe the fitness of each population member is evaluated in parallel, greatly increasing the execution speed on modern multiprocessor machines. To train ANN connection weights a MLPHypothesis class was created, encoding weights as double precision floating point values in an array, as discussed in section 2.4. Initialisation values of hypothesises and mutation values are generated with the Student-T probability distribution supplied by the Math.NET numerics library. This is inspired by Montana and Davis [10], who posit that connection weights are a combination of many relatively small values and a few large values, therefore a starting genome with a some large values will lead to faster convergence. The initialisation source is replaceable by supplying another object that implements the IRandomSource interface during the construction of the MLPWeightHypothesis prototype.

In addition to the standard one point, two point, uniform crossover operators, a per neuron crossover operator is implemented in the MLPWeightHypothesis class.

The mutation operation is split into two stages, a target selection to chose which weights will be mutated, and a mutation type which determines how the chosen weights will be changed. Either a number of randomly chosen weights or a number of randomly chosen neuron's weight can be targeted. The mutation types implemented are replacing the weight with a new random value from the initialisation source, adding a random value from the initialisation source to the current weight, or adjusting the weight with random value from a uniform random distribution.

The actual fitness evaluation of a MLPWeightHypothesis is performed by a IFitnessMethod implementer Two different fitness metrics are supported, the mean squared error (MSE) of the output on a trainingset and mean classification error (MCE). As both MCE and MSE become better with smaller values, and the genetic algorithm is designed to expect that higher fitness values are better, $\frac{1}{MCE}$ and $\frac{1}{MSE}$ are used as the fitness values. Either of the two fitness metrics can be used directly by the MLPWeightHypothesis, or combined with TrainerFitnessMethod to first doing a set number of training iterations with a supplied ITrainer method before calculating the MSE or MCE. This is how hybrid GA-BP or GA-RP training is performed in the framework.

Algorithm 5: Genetic Algorithm used in Experiments

```
Input  : $prototype$, $p$, $cr$, $m$, $crossover$, $mutator$, $fitness$, $keepBest$, $deviationThresh$
$p$ is the size of the genetic population.
$cr$ is the fraction of $P$ to be replaced with new hypothesises created by crossover.
$m$ is the fraction of $P$ to be mutated.
$crossover$ is the crossover method used when creating a new generation.
$mutator$ is the mutation method used when mutating $m \cdot p$ hypothesisses.
$fitness$ is fitness method used to determine the fitness of a hypothesis.
$keepBest$ is number of highest fitness hypothesisses copied directly over to the new
generation, before selection and crossover.
$deviationThresh$ is the threshold value the standard deviation of the population's fitness
values must fall below, before every hypothesis but the best has its genome randomised.
Output: The best fitness value found.
$P = \text{clone}(prototype, p)$
randomise($P$)
while termination condition not met do
    foreach Hypothesis $h \in P$ do
        $h.Fitness = \text{fitness}(h)$
        if $fitness$ can change genome then  $h.genome = updatedGenome$
    end
    if $standardFitnessDeviation < deviationThreshold$ then
        randomise($P \setminus \{h_{fittest}\}$)
    else
        $P_S = keepBest$ fittest hypothesises in $P$
        $P_S = P_S \cup \text{rouletteWheelSelection}(P, (1 - cr)p))$
        foreach Pair $h, i \in rouletteWheelSelection(P, cr \cdot p)$ do
            $P_S = P_S \cup \text{crossover}(h, i) \cup \text{crossover}(i, h)$
        end
        foreach $h \in mutationSelection(P_S, m)$ do  $h = \text{mutation}(h)$
        $P = P_S$
    end
end
foreach Hypothesis $h \in P$ do
    $h.Fitness = \text{fitness}(h)$
    if fitness can change genome then  $h.genome = updatedGenome$
end
return hypothesis from $P$ with the best fitness.
```

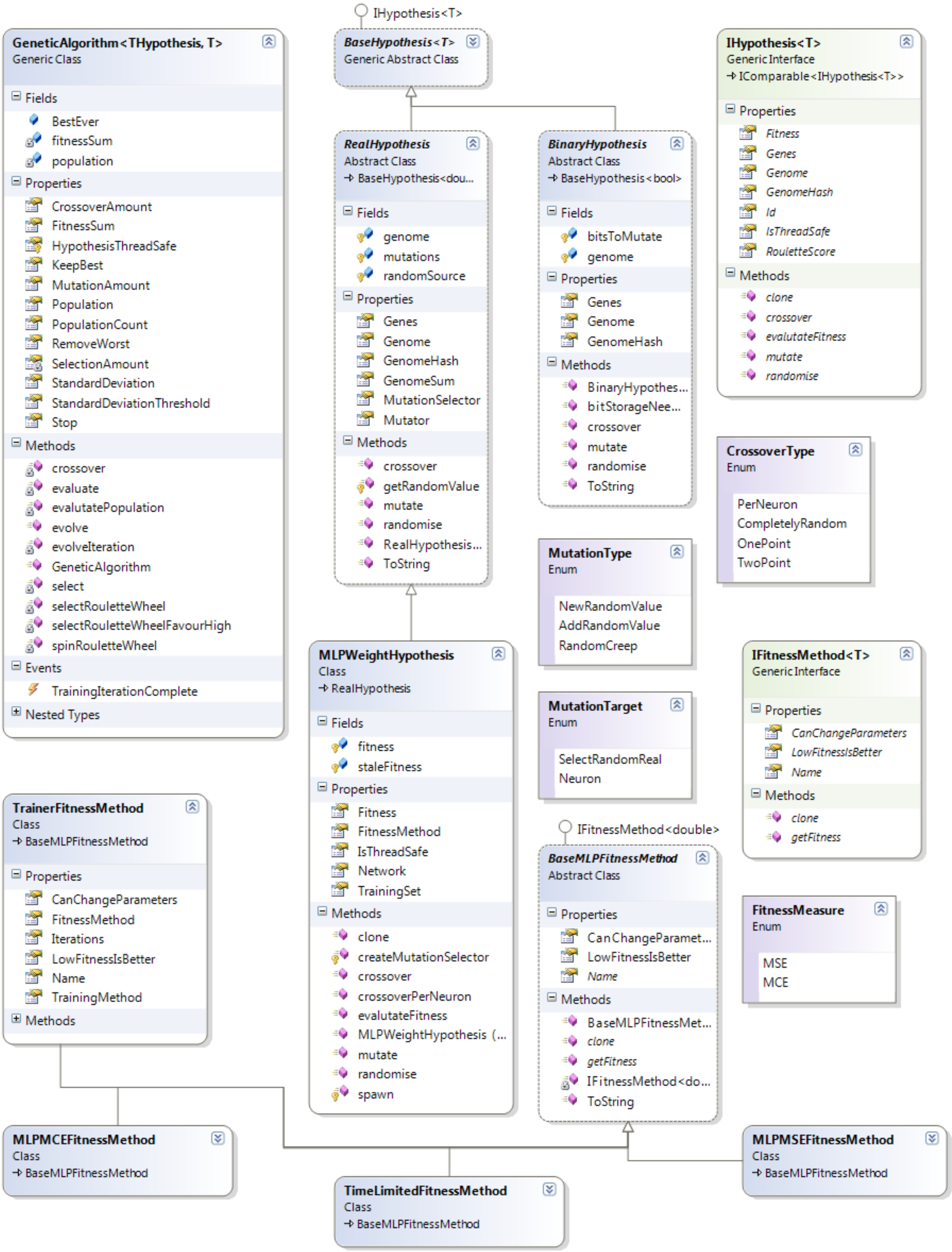


Figure 3.8: Classes used in the implementation of Genetic Algorithm for ANN weight training.

CHAPTER

FOUR

TESTING AND RESULTS

4.1 Experiments

This section describes the testing process and results. First we examine a set of tests listed as preliminary experiments, the results of which impact the setup of the final tests. In the last part of this section, the setup, result and interpretation of the result of the main test is documented.

4.1.1 Preliminary Experiments setup and mean classification error

Both the preliminary experiments and the main experiment use an error measure called the mean classification error.

Definition 3 (Mean Classification Error) *The mean classification error (MCE) of a prediction system is defined as* $\frac{\sum_{i=0}^{n} \|v_p^i - v_a^i\|}{n}$*, where:*

- *n is the number of training samples in the test set.*
- *v_p^i is the predicted value of training sample i.*
- *v_a^i is the actual value of training sample i.*

This measure along with information extracted from the dataset about the distributions of values predicted on will be used to evaluate the efficiency of our predictions in the main experiment. In the preliminary experiments, the focus is the relative efficiency of the tested multilayer perceptrons and the distribution data is therefore not used.

Unless otherwise specified in the subsection of a preliminary experiment, the preliminary experiments are conducted with the following configuration:

- The training data used is extracted from 135 Protoss versus Protoss replays, yielding a total of 3240 training samples.
- All tests use four fold cross-validation.
- The following 20 input features are used: Time, Nexus, Pylon, Assimilator, Gateway, Forge, Photon Cannon, Cybernetics Core, Robotics Facility, Stargate, Citadel Of Adun, Templar Archives, Robotics Support Bay, Probe, Zealot, Dragoon, Dark Templar, High Templar, Reaver and Shuttle.
- The networks use 20 hidden nodes, including a bias node.
- The networks use the same features as the input minus the Time feature.
- When a Genetic Algorithm is used, including hybrid GA/BP and GA/RP algorithms, a population of 48, a crossover rate of 0.8 and a mutation rate of 0.2 is used.

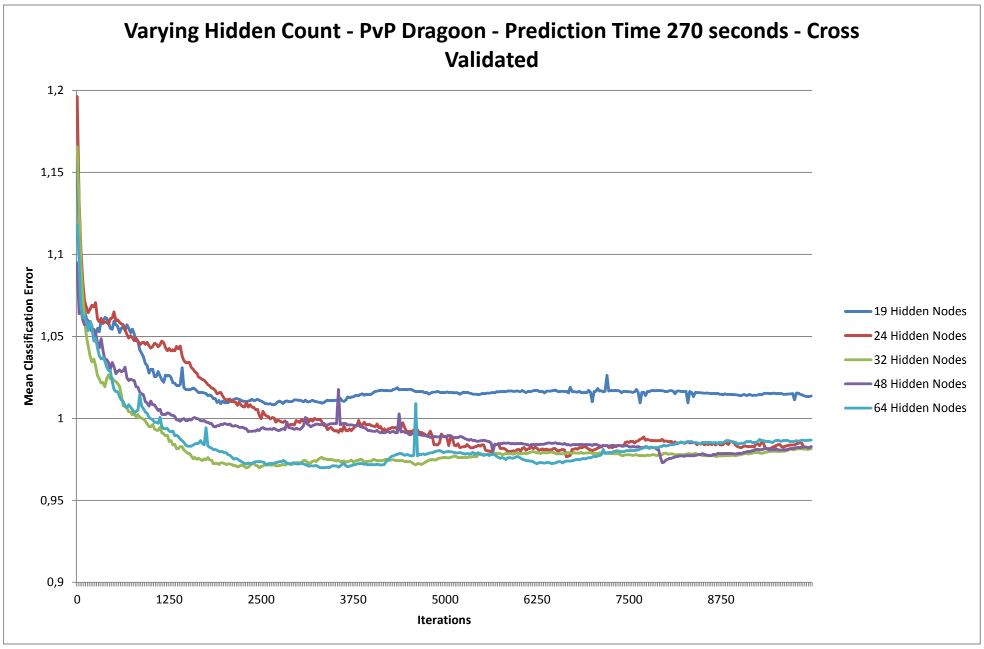


Figure 4.1: Test results for different hidden node counts predicting on the Dragoon game state variable at time 270.

- When a backpropagation training algorithm is used, including in the hybrid GA/BP tests, a learning rate of 0.00125 and a momentum of 0.0075 used.

Note that the unusual backpropagation learning rate and momentum is due to the implementation of the backpropagation algorithm in Encog.

4.1.2 Hidden node experiment

The number of hidden nodes in a network is important for the ability of the network to predict, however if too many hidden nodes are used it may prolong the training time. To this end, we trained a number of networks with different hidden nodes counts attempting to predict different game state variables. All the networks used 20 inputs, 19 outputs and were trained using the resilient propagation algorithm. In figure 4.1, the mean classification error of networks with different hidden node counts is displayed.

In all experiments the performance did not vary between the different hidden node numbers significantly. In a small number of cases a network with a large hidden node count performed the best, such as the network with 40 hidden nodes shown in figure 4.1, however in the majority of the test cases there was only marginal difference. Thus a number of hidden nodes equal to the input layer node count is used, in order to minimize training time.

4.1.3 Genetic algorithm crossover experiments

The genetic algorithm used is detailed in section 3.7. In this section, experiments involving variations of crossover rate and crossover type used for our genetic algorithm experiments is documented. These experiments were conducted to help narrow down optimal settings.

On figure 4.2 we see the mean classification error of MLPs trained with our genetic algorithm using several different crossover rates. As seen on the figure predictions using game states from time 300 all have a mean classification error around 1, with the GA using a crossover rate of 0.8

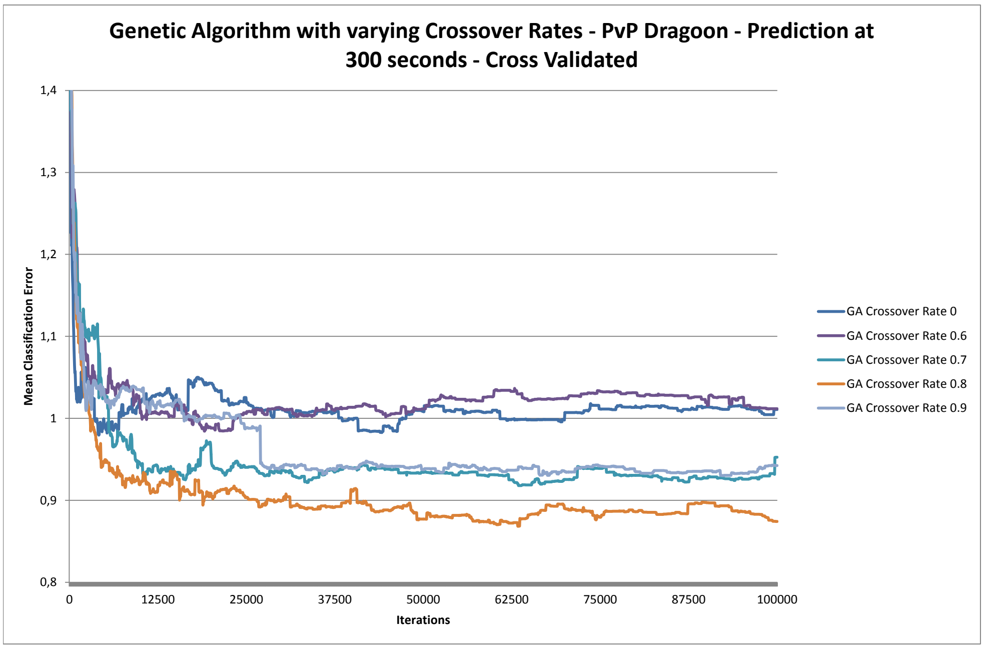


Figure 4.2: Test results for different crossover rates predicting on the game state variable Dragoon at time 300

marginally lower than the remainder. This is consistent with the rest of our results leading us to the conclusion that a crossover rate of 0.8 is optimal.

Similarly, an experiment was run to examine the performances of the different crossover types

Figure 4.3 plots the mean classification error of four different crossover types at times 300 and 360 respectively. As can be observed, Randomized Crossover appears to reach lower mean classification errors and as such will be used for further testing.

4.1.4 Genetic algorithm mutation experiment

The last adjustable feature of our genetic algorithms is the mutation rate, which we also ran an experiment on in an attempt to find n optimal value.

In the next figure, figure 4.4, we see the results of MLPs trained by GAs using different mutation rates. It is immediately apparent that the network trained by the GA that did not mutation the weights performed significantly worse than the rest. A mutation rate of 0.2 was chosen as it performed the best on average.

4.1.5 Backpropagation experiments

For the backpropagation training algorithm, we tested several different learning rate and momentum combinations. The results are shown in figure 4.5.

The graphs in figure 4.5 indicate that a learning rate of 0.01 is too high, since the mean squared error of the test oscillates with iterations - the weight adjustment with each iteration is too large to narrow down the minimum. For learning rate 0.0025 and 0.00125 the graphs appear thinner, indicating less oscillation is taking place. The lowest mean squared error is achieved with a learning rate of 0.0025 and a momentum of 0.0075. The unusually low values we attribute to the implementation of the backpropagation algorithm in Encog and seems to correspond to 1/10 of conventional recommended values.

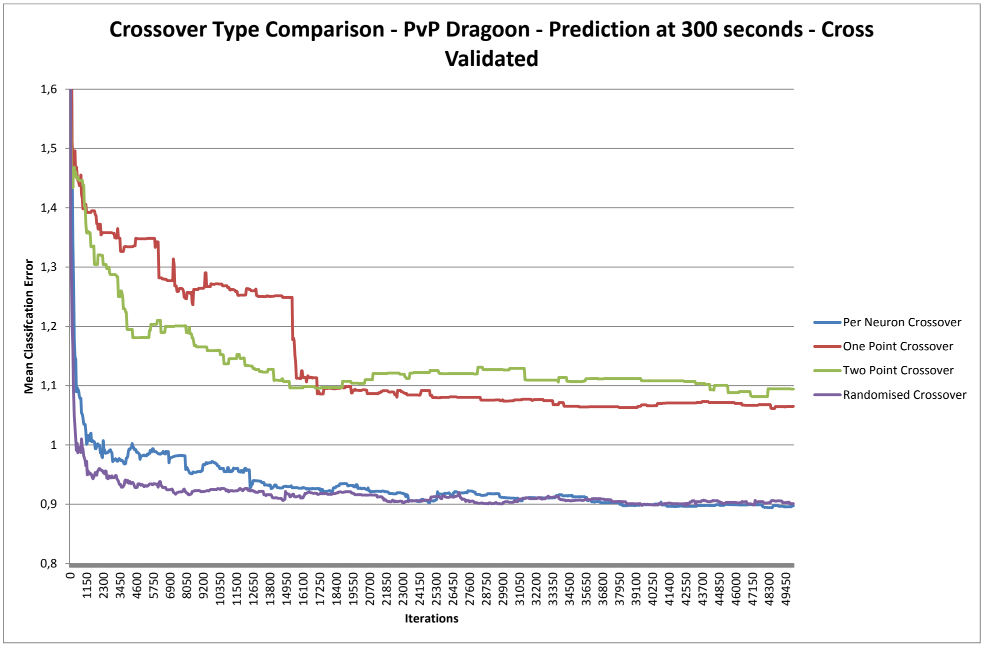


Figure 4.3: Test results for different crossover types predicting on the game state variable Dragoon at time 300.

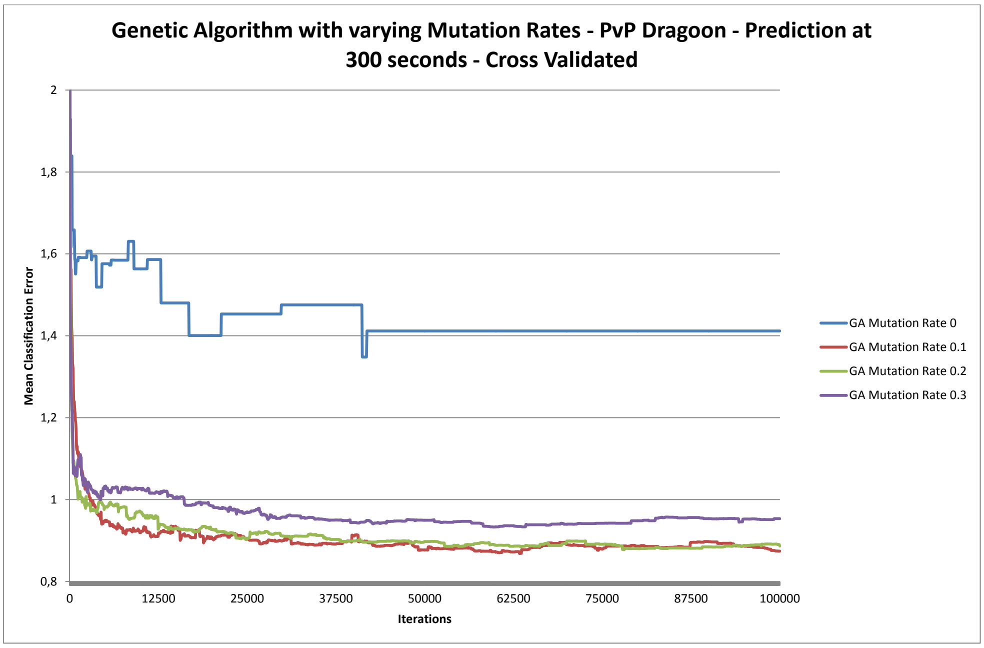


Figure 4.4: Test results for different mutation rates predicting on the game state variable Dragoon at time 300.

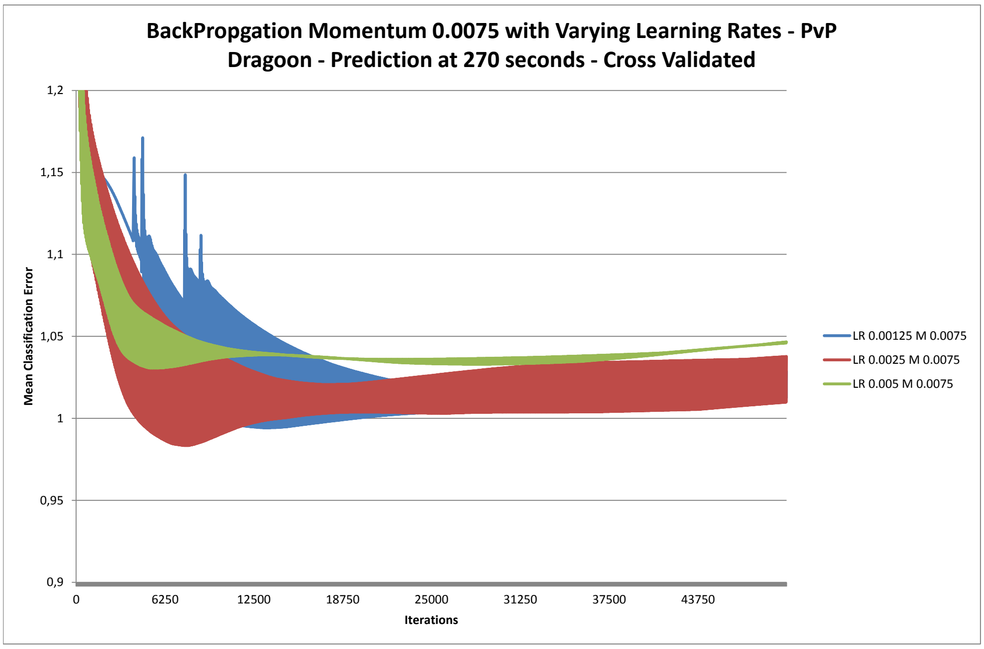


Figure 4.5: Test results for MLPs trained with a learning rate of 0.01 predicting on the game state variable Dragoon at time 270

4.2 Main test

In this section we first walk through the main test setup and data used for the test and then present and interpret the main test results.

4.2.1 Setup and dataset used for the main test

For the main test, we use the parameters we arrived to in the preliminary experiments. To summarize the following test setup is used, unless otherwise is specified:

- The networks have a single hidden node layer with 20 hidden nodes, one of which is a bias node.

- When Backpropagation is used for training, it uses a learning rate of 0.0025 and a momentum of 0.0075.

- When the genetic algorithm is used,a mutation rate of 0.2, a crossover rate of 0.8 and the crossover type randomized is used.

- A total of 135 Protoss vs Protoss replays were used to generate 3240 game states which were used as training samples.

4.2.2 Prediction Accuracy Test Results

The final results for the Protoss versus Protoss prediction test is presented in figures 4.7, 4.8 and 4.6. The mean classification error of the network trained with the genetic algorithm is the highest in the tests. In figure 4.7 the backpropagation algorithm achieves the lowest mean classification error, while in figures 4.6 and 4.8 we see that the resilient propagation algorithm outperforms

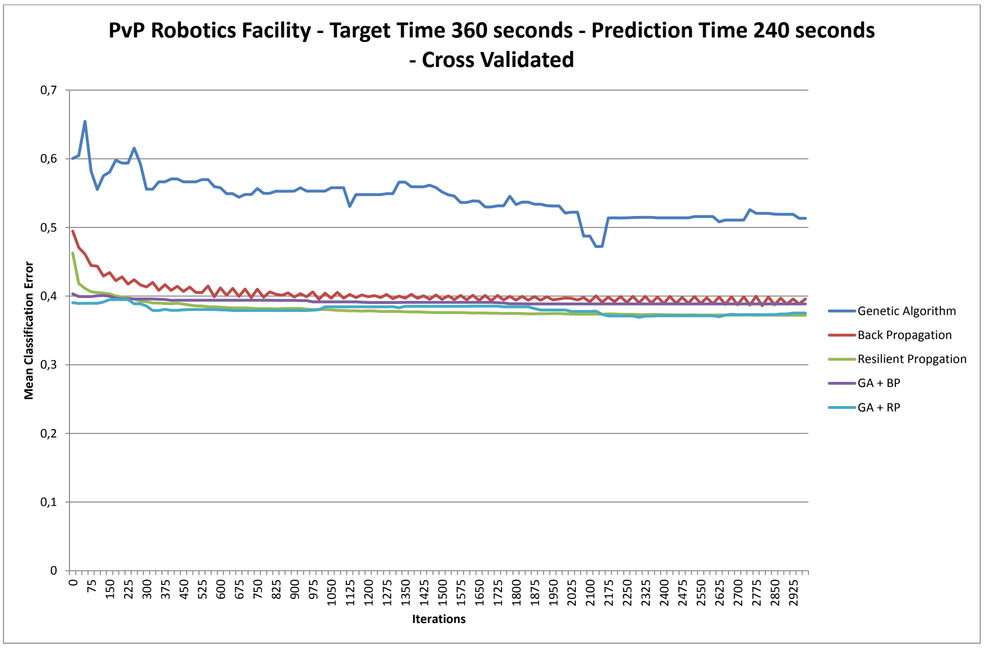


Figure 4.6: Prediction accuracy test results for Protoss versus Protoss - Robotics Facility

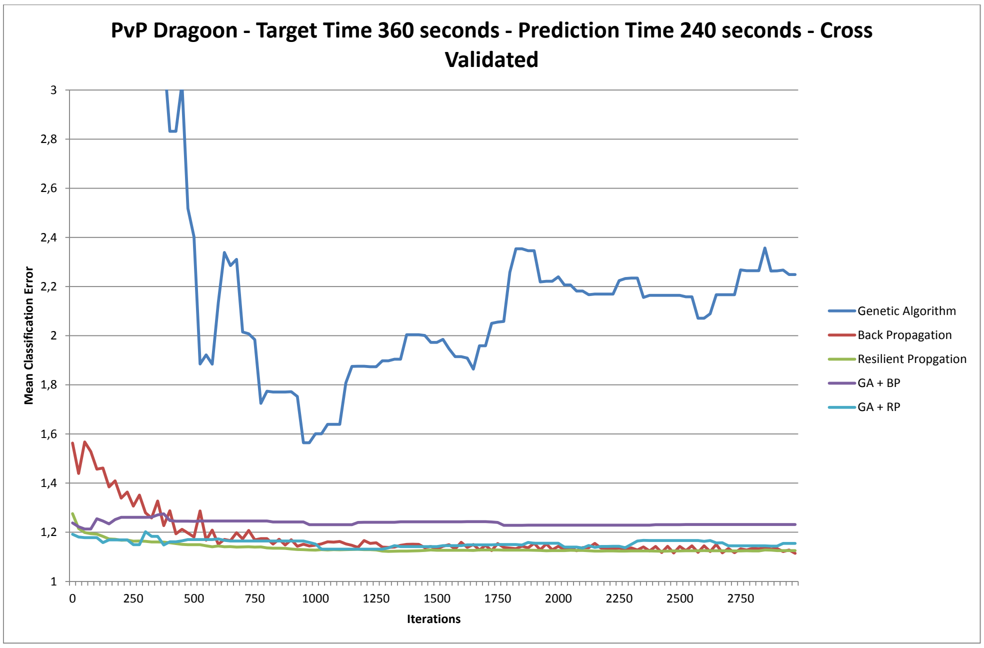


Figure 4.7: Prediction accuracy test results for Protoss versus Protoss - Dragoon

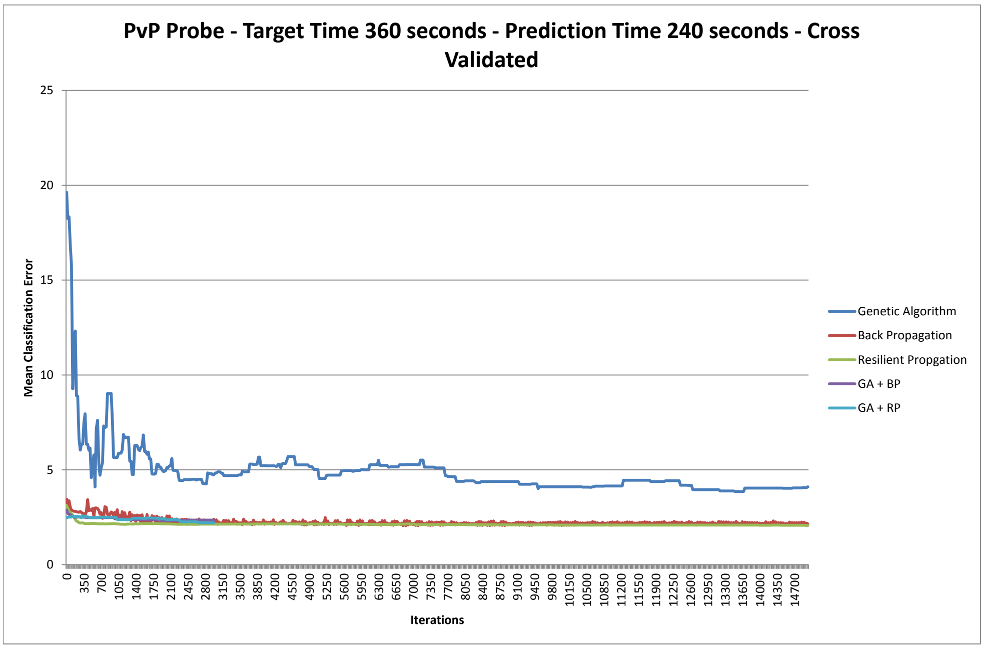


Figure 4.8: Prediction accuracy test results for Protoss versus Protoss - Probe

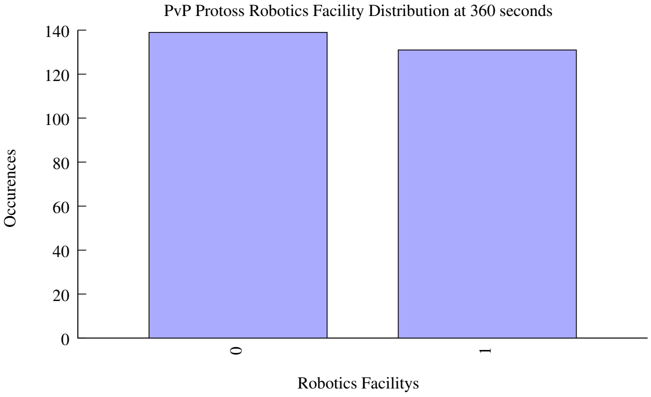


Figure 4.9: Game state variable distribution - Robotics Facility

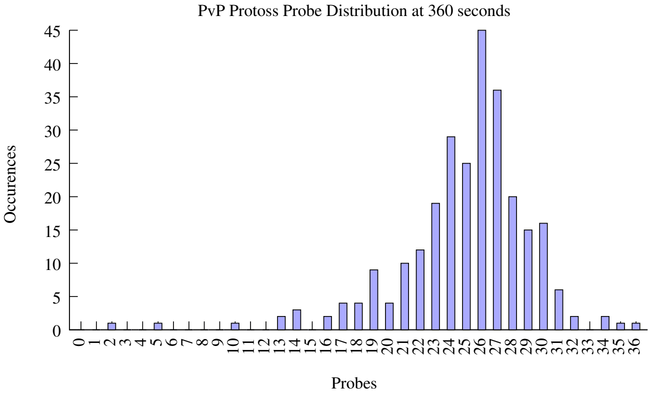


Figure 4.10: Game state variable distribution - Probe

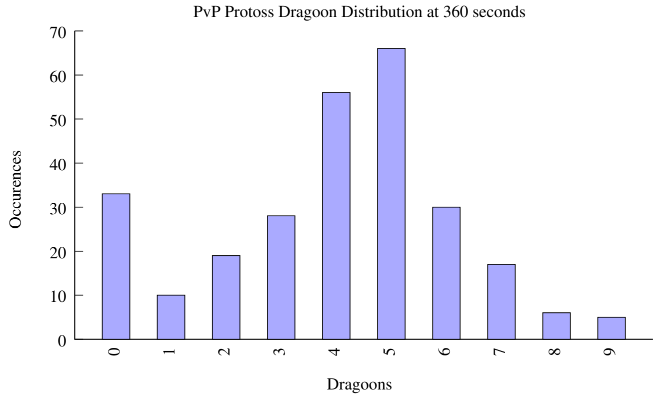


Figure 4.11: Game state variable distribution - Dragoon

it slightly. In the majority of tests, the hybrid algorithms were both capable of matching the propagation algorithms in prediction accuracy but with a far greater training time.

The distributions of the game state variables predicted on can be seen in figures 4.11, 4.10 and 4.9. With these distributions we can compare our prediction MCE to that of the statistically best guess:

- Nexus - Prediction MCE: ca. 0.3, Best Guess MCE: 0.19.
- Robotics Facility - Prediction MCE: ca. 0.36, Best Guess MCE: 0.49.
- Probe - Prediction MCE: ca. 2.4, Best Guess MCE: 2.96.
- Zealot - Prediction MCE: ca. 0.8, Best Guess MCE: 1.1.
- Dragoon - Prediction MCE: ca. 1.1, Best Guess MCE: 1.76.

These results are representative of the rest of our test data. The majority of game state variables are predicted more accurately than the statistically best guess, with the rest predicted significantly worse.

4.2.3 Input type test results

Two sample results of the input type experiments are presented in figures 4.12 and 4.13. We differentiate between two types of input: input with losses and input without losses. The difference between the two is whether the game states used for input are corrected for units that have been lost or not. When input without losses is used, the target game state is also generated without losses.

In figure 4.12 we see that the MLPs using data accounting for losses are more accurate, but in figure 4.13 the data without losses appears to produce the best results by a significant margin. The lacking performance of the MLPs trained with input without losses in predicting on Dragoons could be caused by Dragoons lost earlier in the game, as the player may choose to remake lost units in order to be able to defend. Input with losses may indicate unusually low numbers due to losses which input without losses cannot, resulting in MLPs trained with input with losses being able to predict if a player will remake the lost units. This theory is supposed by the tests predicting on Zealots and Probes in which using input with losses also produced the best results. Robotics Facilities are less likely to be lost since they can take more damage than Dragoons and are often better protected since they represent a greater investment on the part of the player. As discussed in the real-time strategy section 1.2, opening plays dictate the order in which a player builds units and buildings and since the Robotics Facility represents a significant investment it is likely to be built at the same time every time a certain opening play is used. By using the input without losses, we are essentially using what the player has chosen to build as the input and since the players choice of what to build is essential to opening plays when it comes to technology progressing buildings especially.

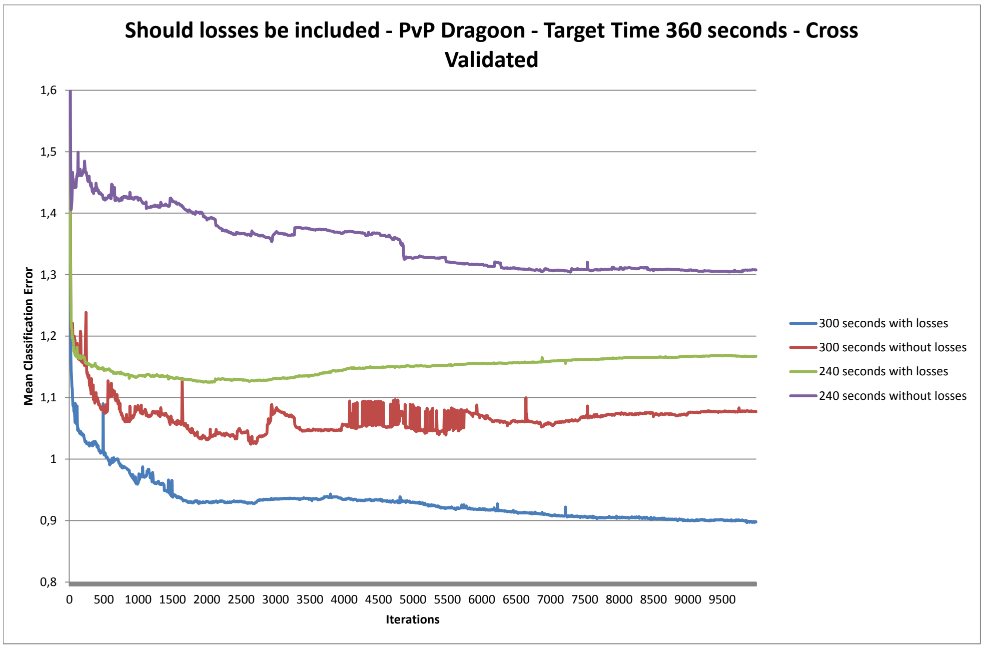


Figure 4.12: Test results for input with and without loss reflection predicting on the game state variable Dragoon

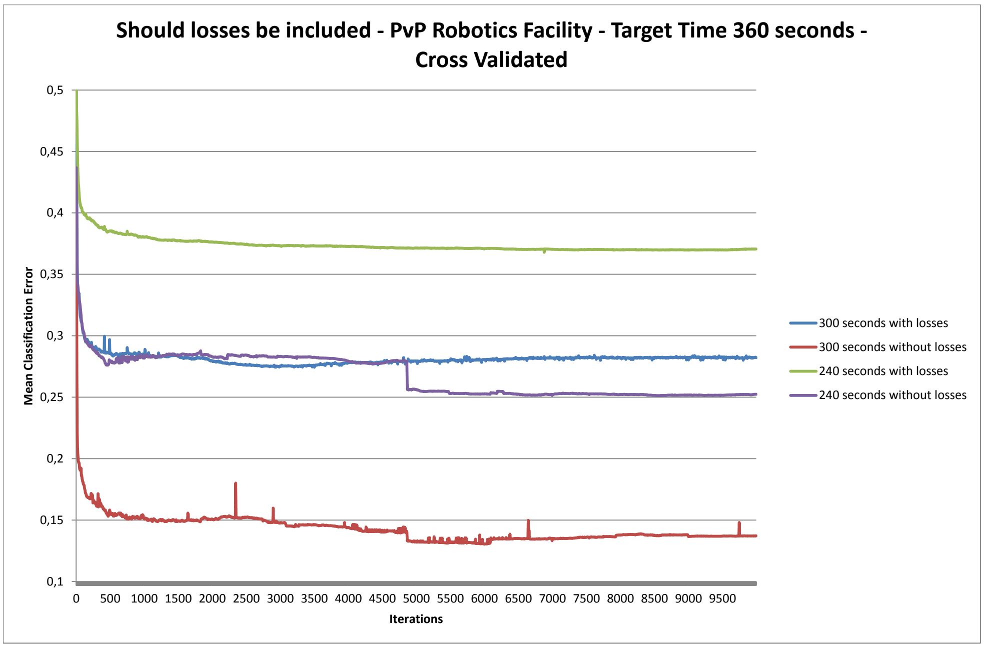


Figure 4.13: Test results for input with and without loss reflection predicting on the game state variable Robotics Facility

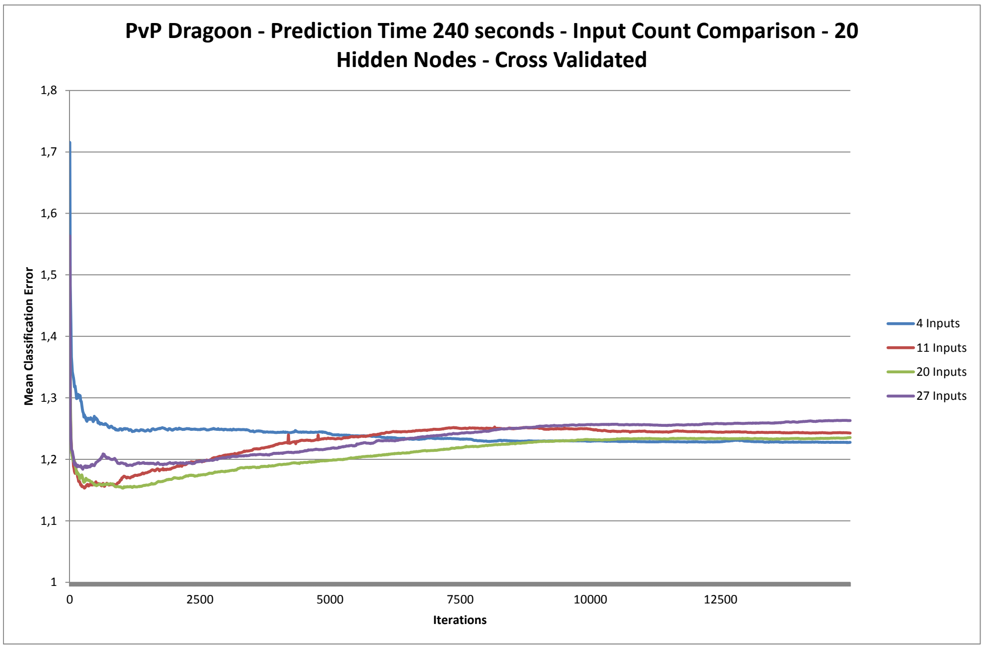


Figure 4.14: Feature selection test results for predicting on the game state variable Dragoon at time 240

4.2.4 Feature Selection

In this experiment, several feature selections derived from the technology tree as described in section 3.2 were tested in order to evaluate the effect of the feature select. The input nodes of three MLPs were configured as per the assigned feature selection and trained using resilient propagation and a fourth MLP using the entire feature set as input was trained for comparison. The MLPs used a single output node and a hidden node count equal to their input node count, including one bias node.

As can be seen of figure 4.14 and figure 4.15, the performance of the networks using feature selection consistently exceeded that of the network using the entirety of the feature set.

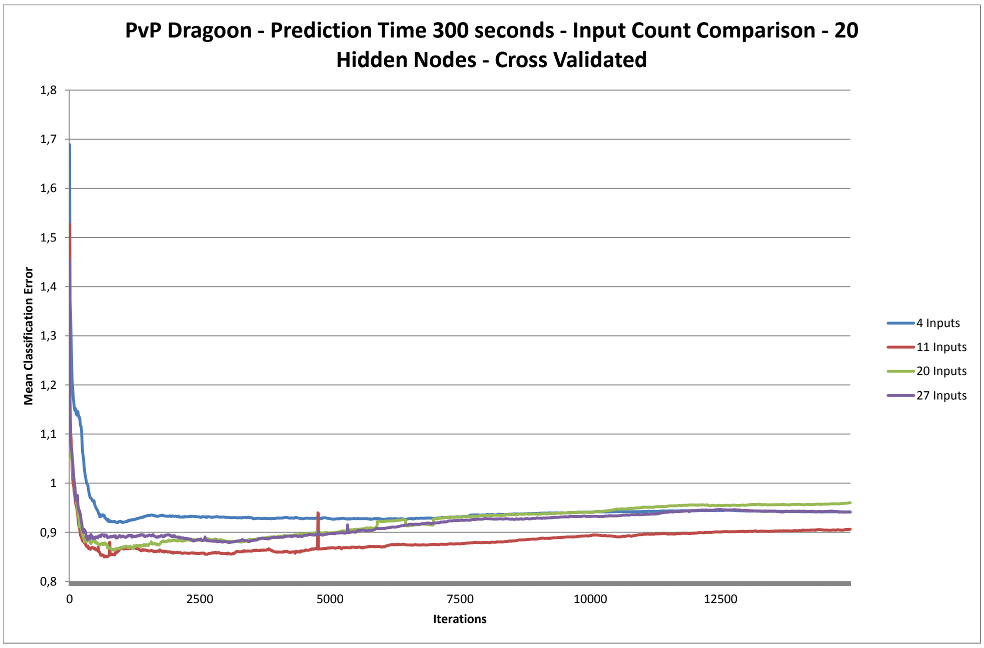


Figure 4.15: Feature Selection test results for predicting on the game state variable Robotics Facility

CHAPTER
FIVE

CONCLUSION

5.1 Conclusion

We have proposed, implemented and tested a multilayer perceptron based strategy prediction system for the real-time strategy game StarCraft: Brood War. Also proposed is a novel feature selection technique relying on real-time strategy game design, which was also tested. The resilient propagation algorithm was found to be the most effective at training strategy predicting MLPs and the resulting predictions were found to be an improvement to the statistically best guess. Networks configured using the novel feature selection were found to be more effective than networks not employing feature selection.

Two methods for using game states as training samples were examined and were found to perform better than each other in differing scenarios.

5.2 Future Work

The predictions presented here are limited in the sense that only the game state at time 360 is predicted and only the game StarCraft: Brood War was used as a test case. If a system predicting the state a set amount of time in the future was implemented instead, it would be more useful in making a functional AI as the predictions could then be applied throughout the entirety of a game. As described in the real-time strategy section, 1.2, of this report, the theory behind the predictions made in this report should be applicable to any real-time strategy game using a technology tree which could be corroborated if the prediction system was implemented to work with different real-time strategy games.

In terms of improving the accuracy of the predictions, a few logical steps can be taken. In this report only the previous game state of the player whose strategy we predict is used as input meaning that the opposing players game state is not factored in. As such examining the effect of including variables from the opposing players game state as input on the prediction accuracy is promising. Extracting additional data from replays concerning unit movements, map layouts and building placements which could also be used as input to improve prediction accuracy is also a possibility.

In our preliminary experiments we found that unexpectedly, not accounting for losses improved the prediction accuracy significantly in some cases and theorized that this is due to the losses making it harder to identify the opening play being executed. It may be possible to represent losses sustained by the player in the input data in some form or analyse in which cases data without losses provide better results. A modification of the prediction system presented here which predicts the set of most likely candidate game states, instead of predicting the most likely value of each game state variable at the target time. While an exact prediction would be preferable, it is possible that sufficiently small set of possible strategies could be identified more easily.

Lastly as the presented system is meant to be incorporated in an AI player, creating such an AI player and documenting its performance is of obvious interest.

BIBLIOGRAPHY

[1] Frederik Frandsen, Mikkel Hansen, Henrik Sørensen, Peder Sørensen, Johannes Garm Nielsen, and Jakob Svane Knudsen. Predicting player strategies in real time strategy games. 2010.

[2] Sander Bakkes Frederik Schadd and Pieter Spronck. Opponent modeling in realtime-strategy games. 2007.

[3] Jean-Loup Gailly and Mark Adler. GNU zip compression utility. `http://www.gzip.org/`.

[4] Jeff Heaton. Encog java and dotnet neural network framework. `http://www.heatonresearch.com/encog`.

[5] Christian Igel and Michael Hüsken. Empirical evaluation of the improved rprop learning algorithm. *Neurocomputing*, 50:2003, 2003.

[6] Hiroaki Kitano. Empirical studies on the speed of convergence of neural network training using genetic algorithms. In *Proceedings of the eighth National conference on Artificial intelligence - Volume 2*, AAAI'90, pages 789–795. AAAI Press, 1990.

[7] Ben G. Weber & Micheal Mateas. A data mining approach to strategy prediction. 2009.

[8] M. McInerney and A.P. Dhawan. Use of genetic algorithms with backpropagation in training of feedforward neural networks. In *Neural Networks, 1993., IEEE International Conference on*, pages 203 –208 vol.1, 1993.

[9] Tom M. Mitchell. *Machine Learning, International Edition*. McGraw-Hill, 1997.

[10] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, pages 762–767, 1989.

[11] M Riedmiller and H Braun. A direct adaptive method for faster backpropagation learning: the rprop algorithm. *IEEE International Conference on Neural Networks*, 1993(3):586–591, 1993.

[12] BWAPI Project Team. Bwapi: An API for interacting with Starcraft: Broodwar 1.16.1. `http://code.google.com/p/bwapi/`.

[13] IC#Code team. The Zip, GZip, BZip2 and Tar implementation for .NET. `http://www.icsharpcode.net/opensource/sharpziplib/`.

[14] H. Jaap van den Herik, Hieronymus Hendrikus Lambertus Maria Donkers, and Pieter H. M. Spronck. Opponent modelling and commercial games. *IEEE Symposium on Computational Intelligence and Games*, 2005.

[15] D. Vijeth. Neurondotnet. `http://neurondotnet.freehostia.com/index.html`, August 2008.

[16] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423 –1447, sep 1999.

APPENDIX

A

EXPERIMENT GRAPHS

A.1 Genetic Algorithm Parameters

A.1.1 Crossover Fraction

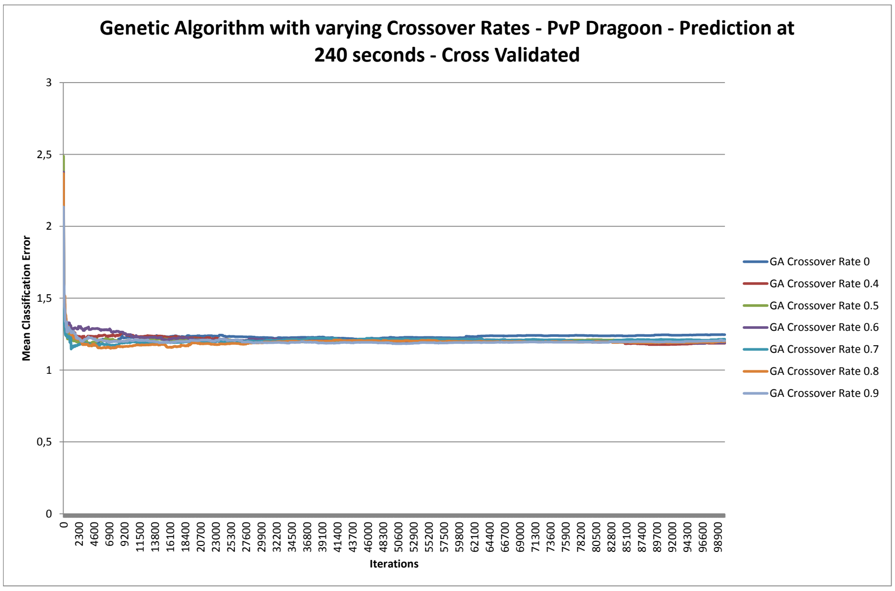


Figure A.1: Mean Classification Error for predictions made at 240 seconds

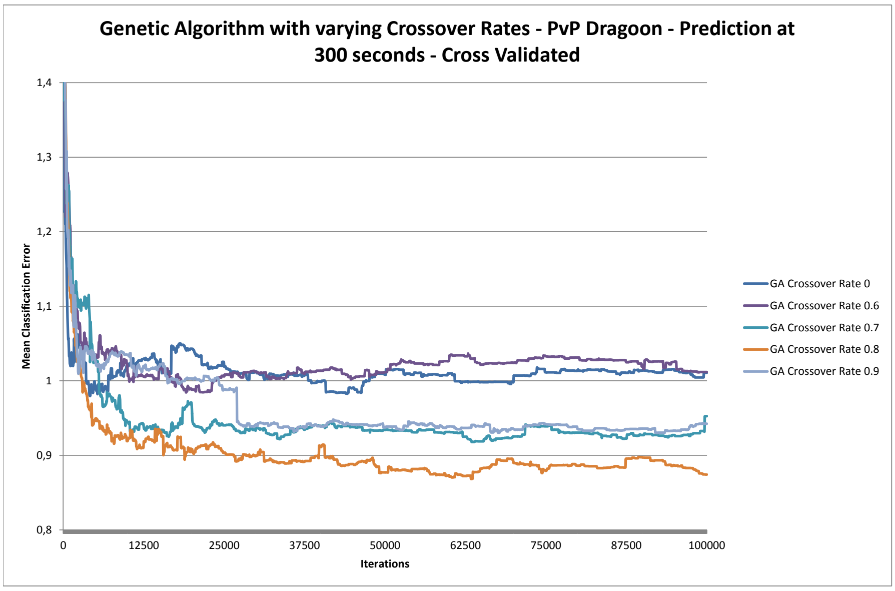


Figure A.2: Mean Classification Error for predictions made at 300 seconds

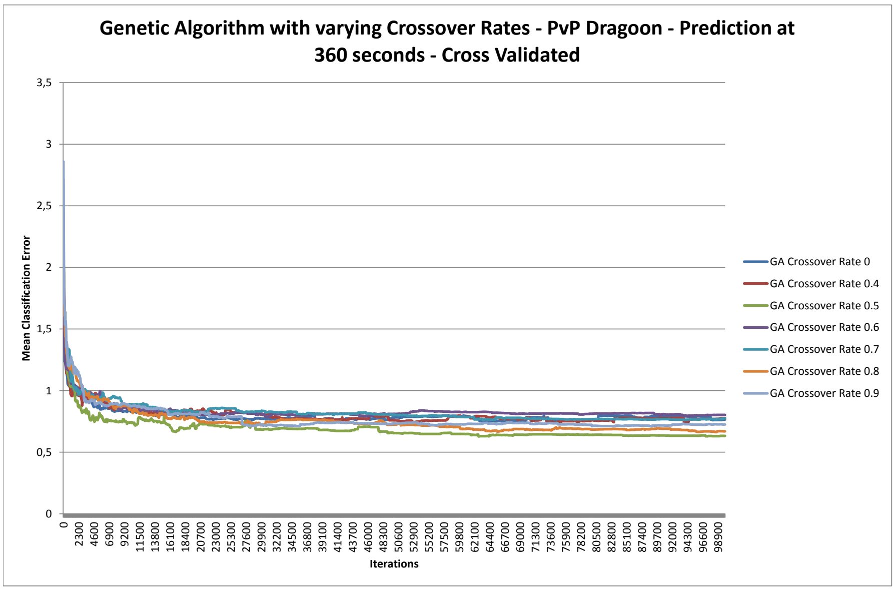


Figure A.3: Mean Classification Error for predictions made at 360 seconds

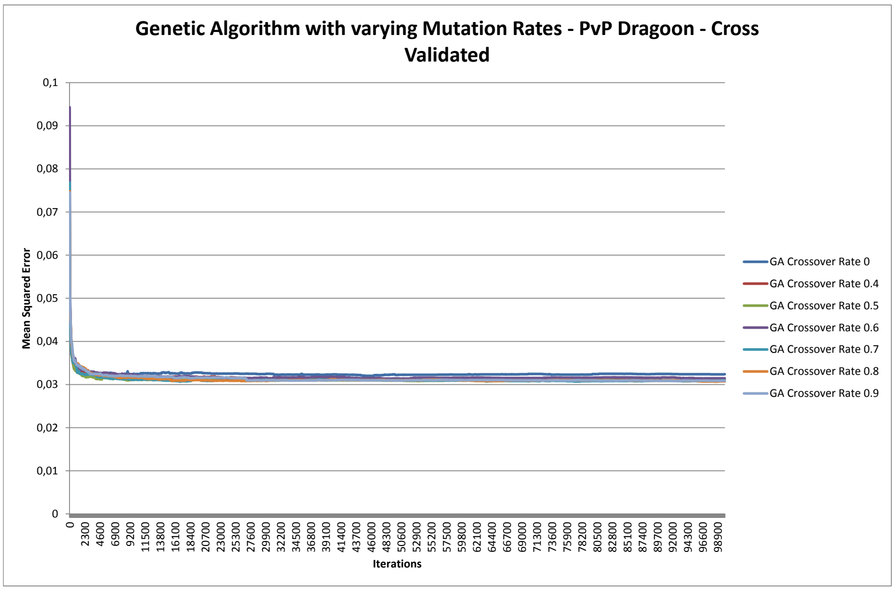


Figure A.4: Mean Square Error

A.1.2 Crossover Type

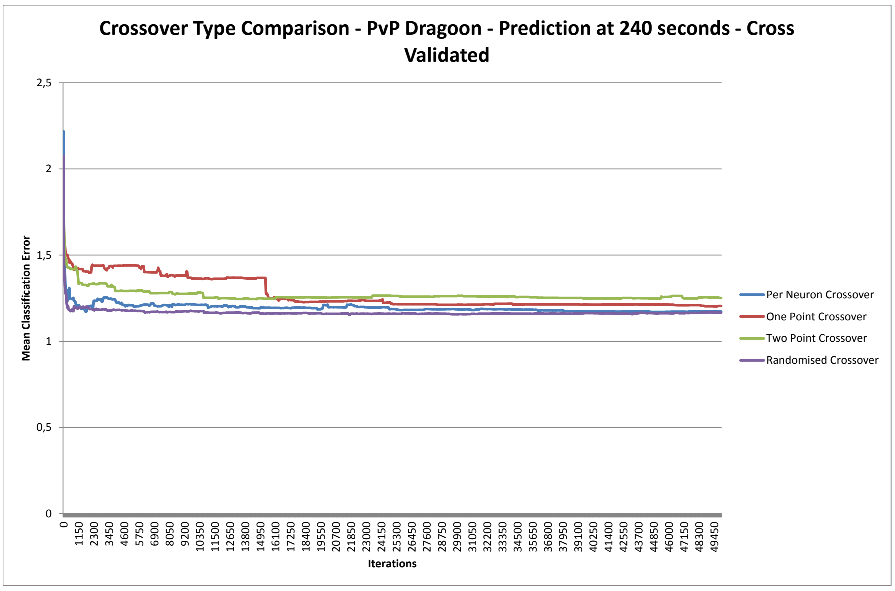


Figure A.5: Mean Classification Error for predictions made at 240 seconds

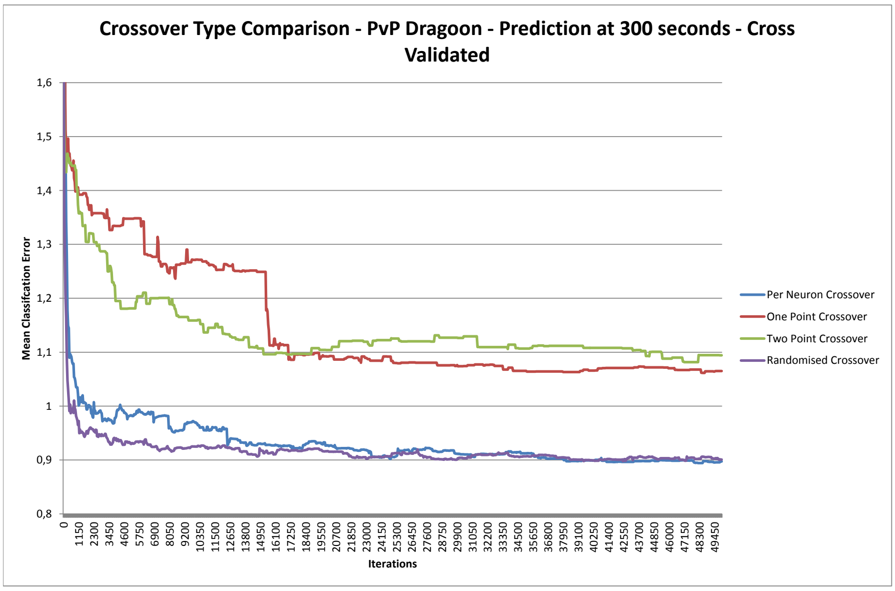


Figure A.6: Mean Classification Error for predictions made at 300 seconds

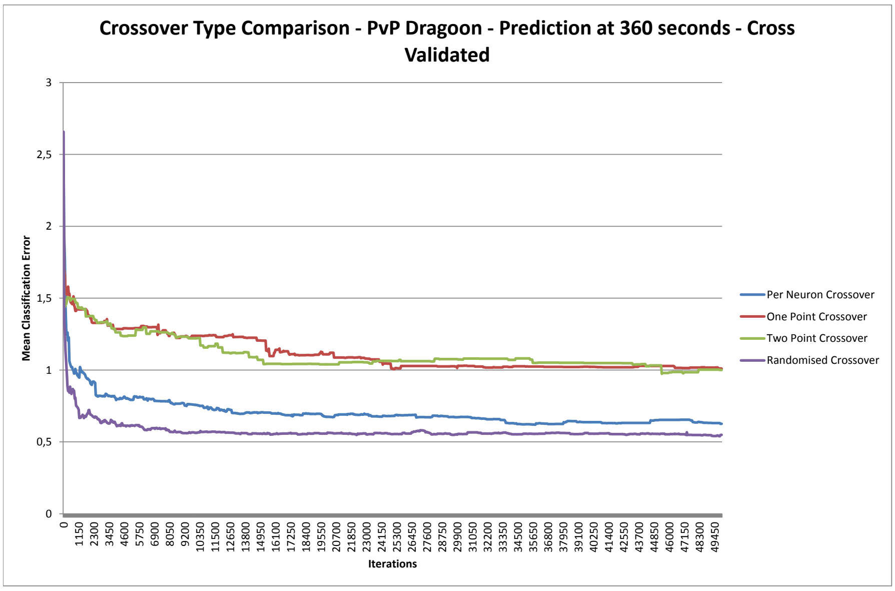


Figure A.7: Mean Classification Error for predictions made at 360 seconds

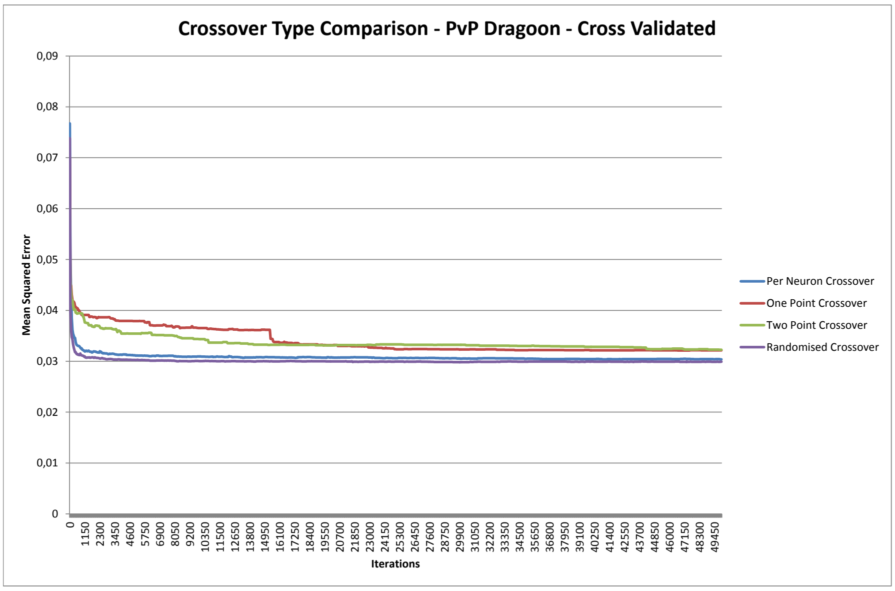


Figure A.8: Mean Square Error

A.1.3 Mutation Fraction

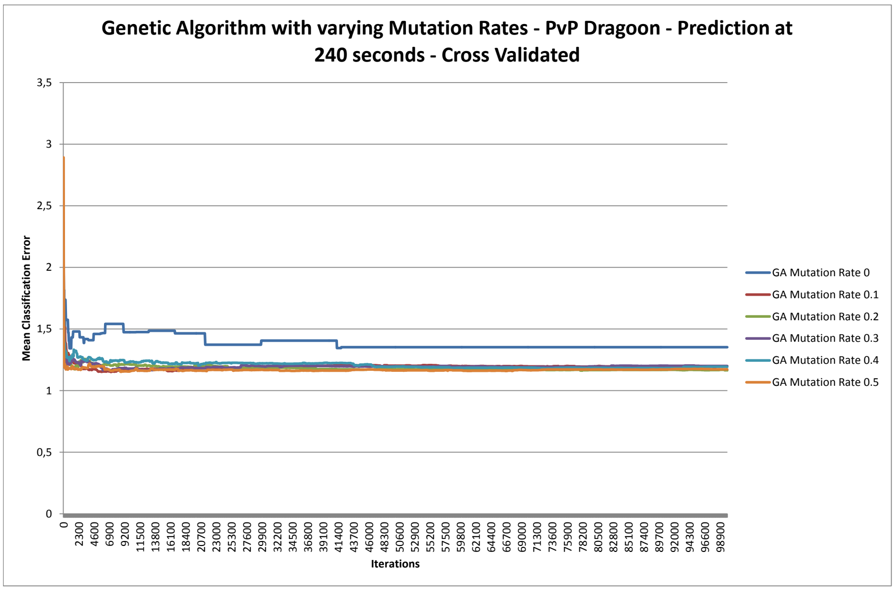


Figure A.9: Mean Classification Error for predictions made at 240 seconds

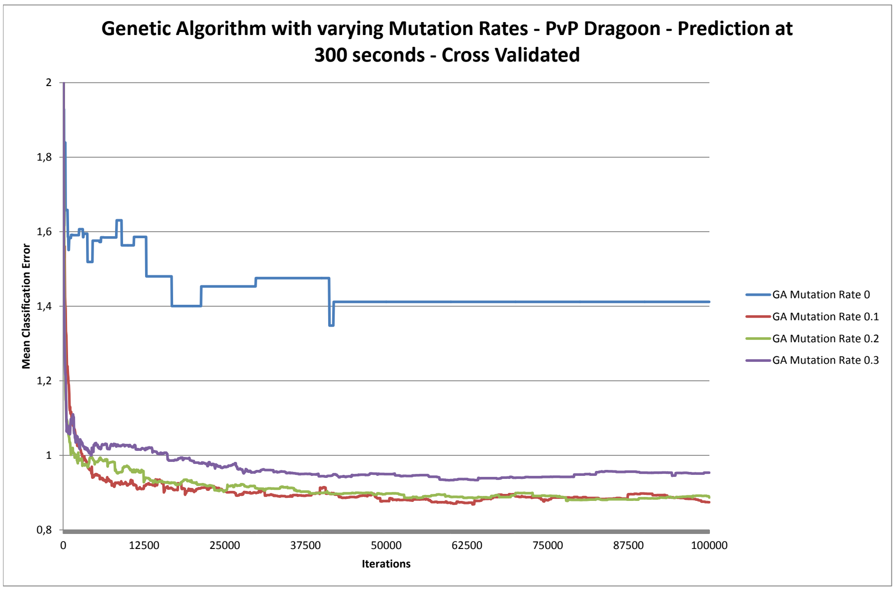


Figure A.10: Mean Classification Error for predictions made at 300 seconds

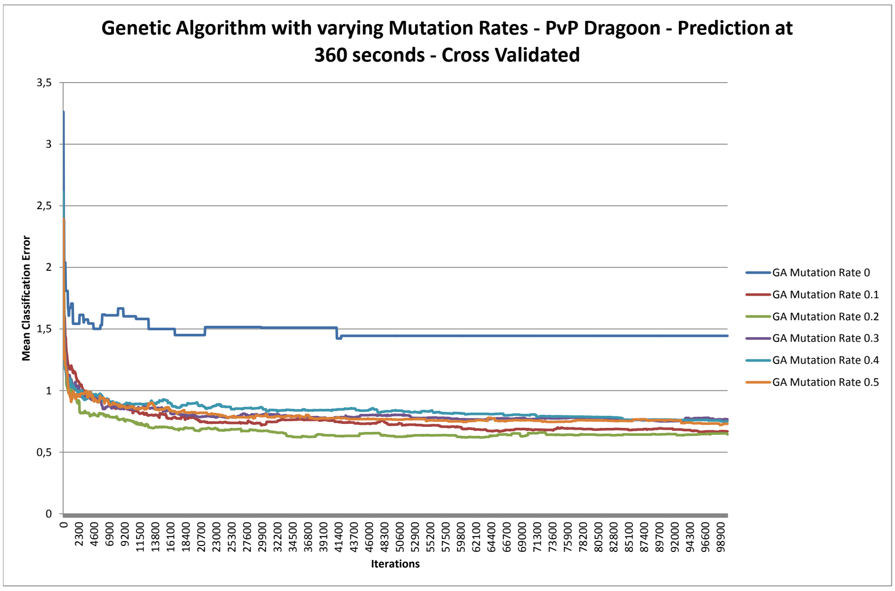


Figure A.11: Mean Classification Error for predictions made at 360 seconds

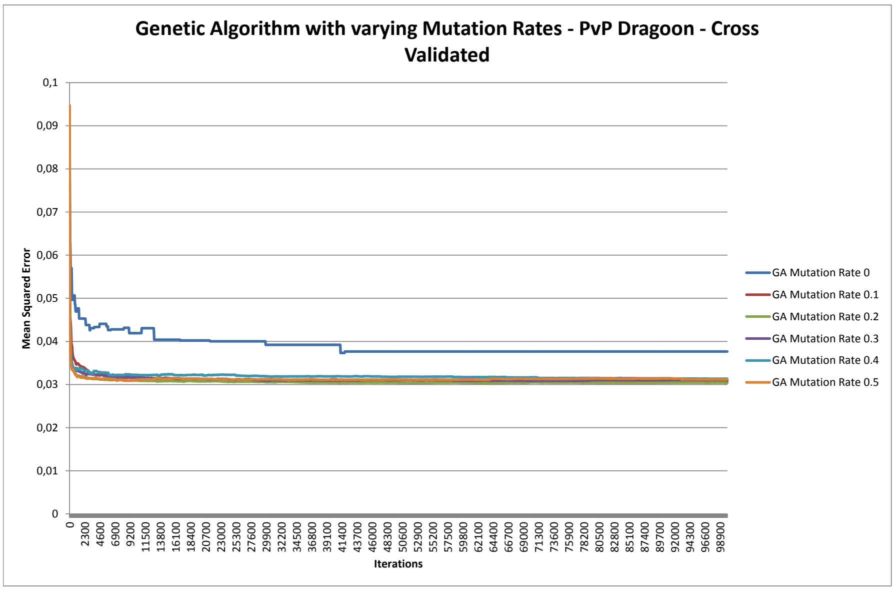


Figure A.12: Mean Square Error

A.1.4 MutationCount

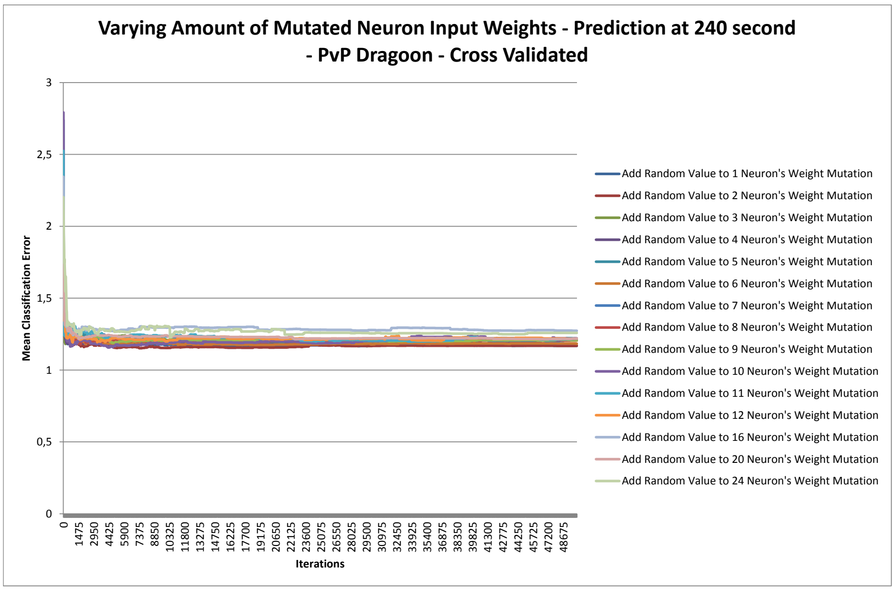


Figure A.13: Mean Classification Error for predictions made at 240 seconds

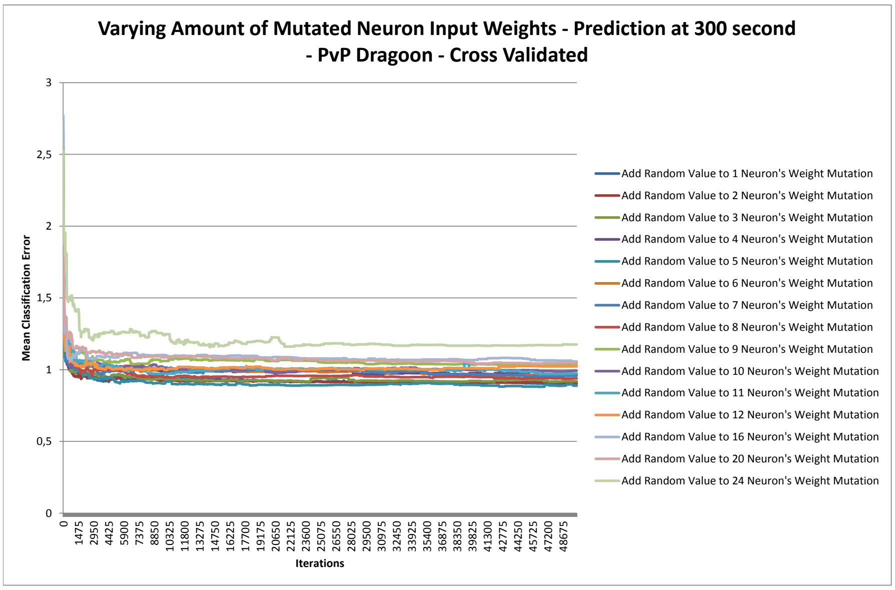


Figure A.14: Mean Classification Error for predictions made at 300 seconds

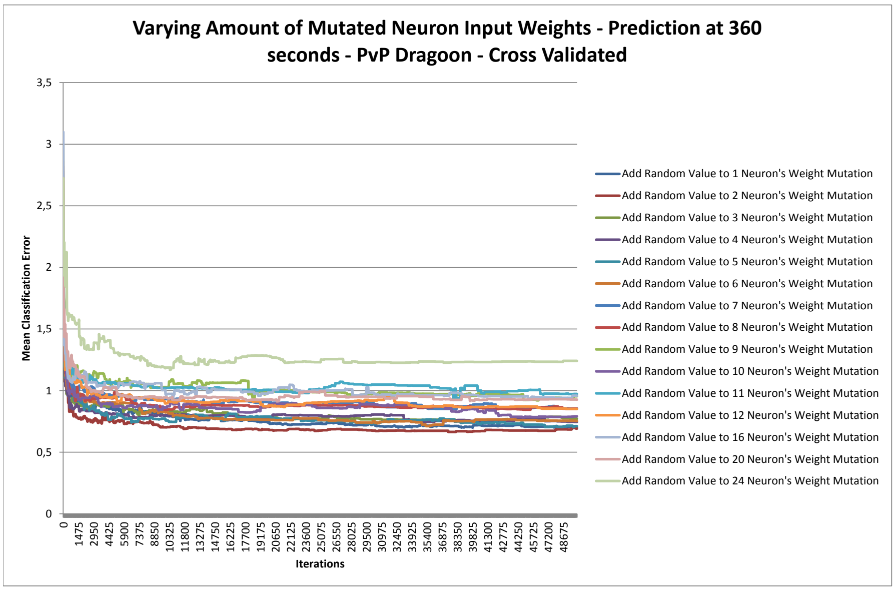


Figure A.15: Mean Classification Error for predictions made at 360 seconds

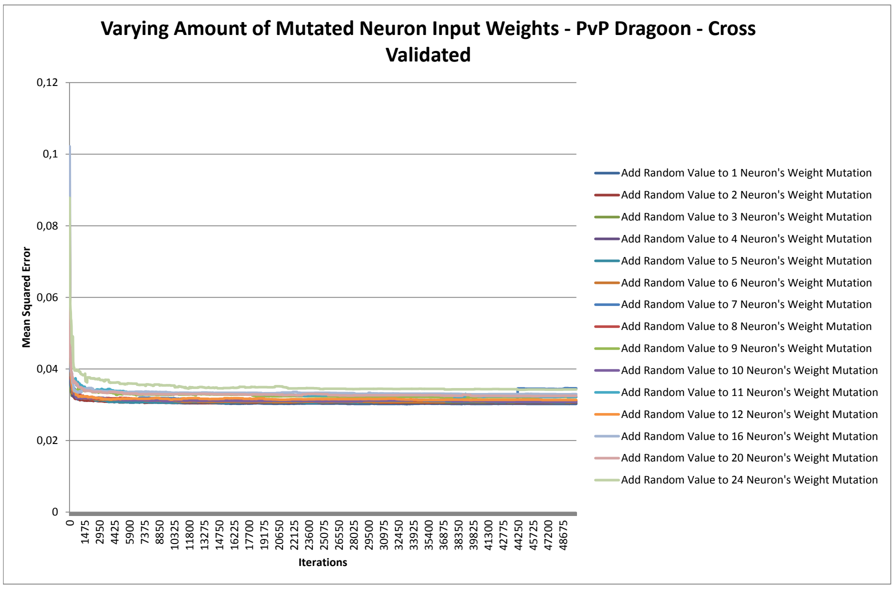


Figure A.16: Mean Square Error

A.1.5 MutationType

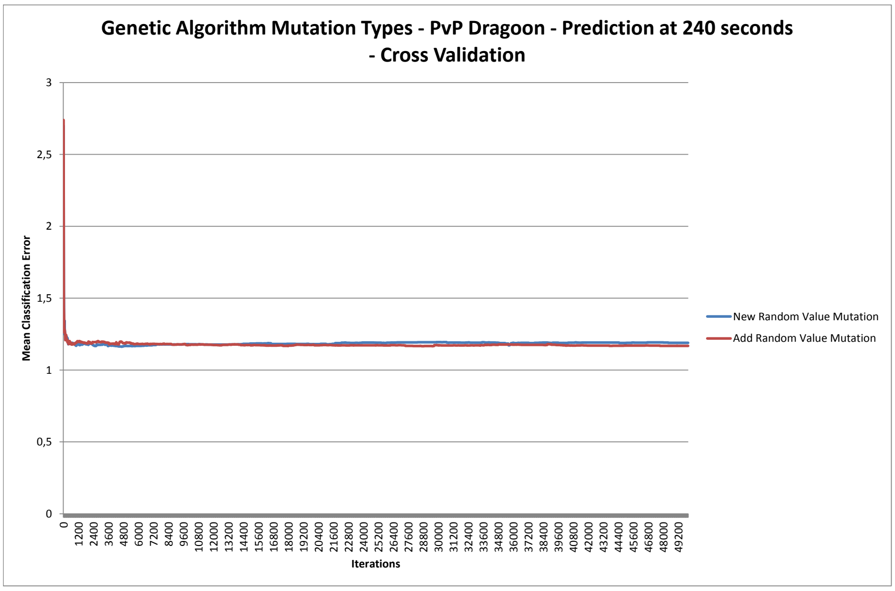


Figure A.17: Mean Classification Error for predictions made at 240 seconds - Cross Validation

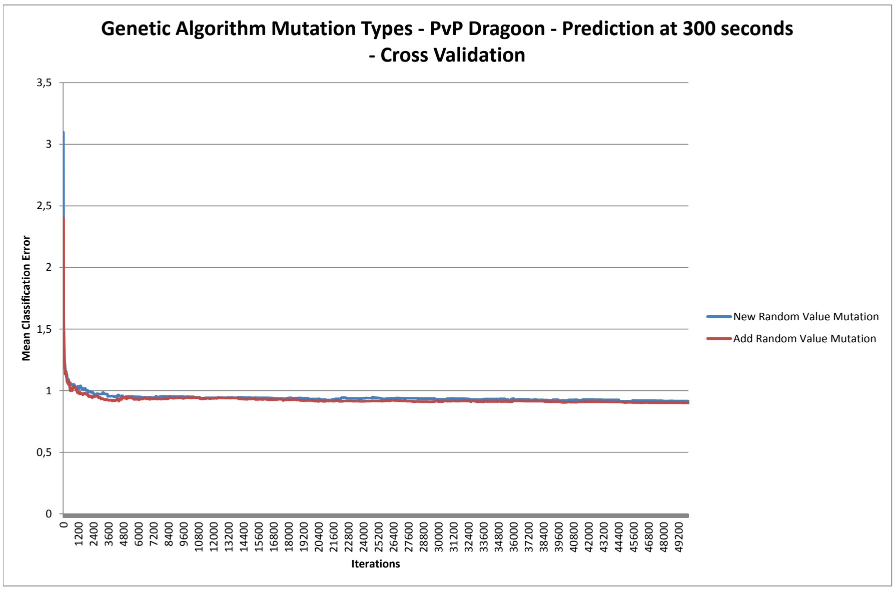


Figure A.18: Mean Classification Error for predictions made at 300 seconds

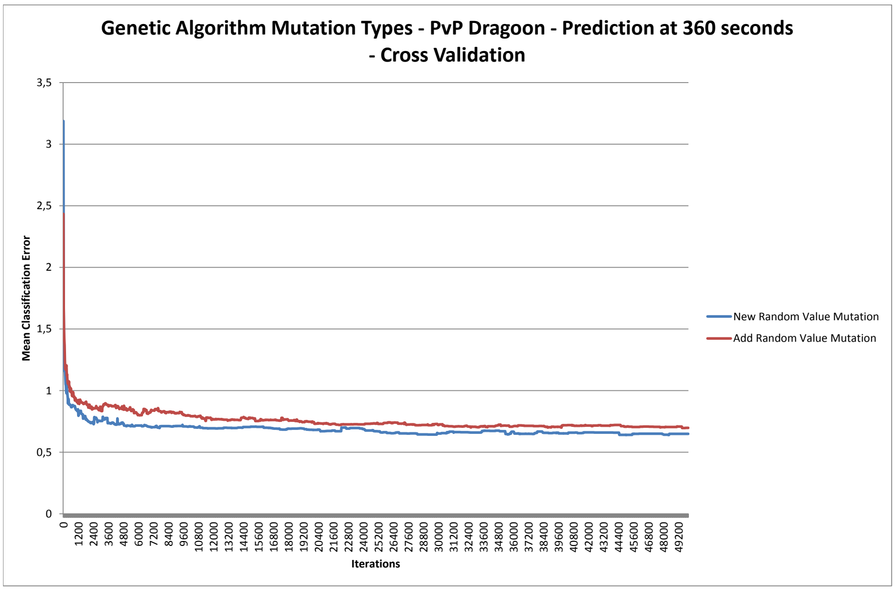


Figure A.19: Mean Classification Error for predictions made at 360 seconds

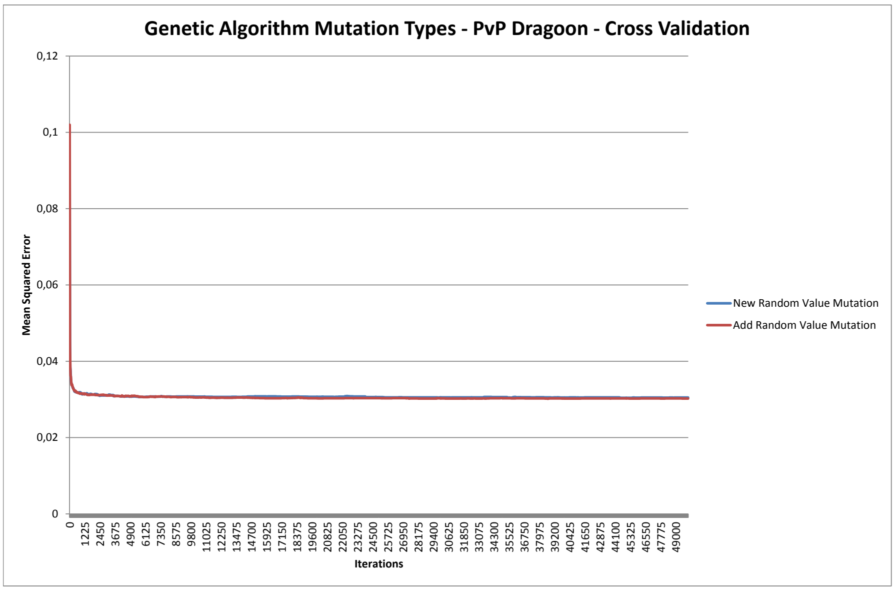


Figure A.20: Mean Square Error

A.1.6 Hybrid Iteration Count

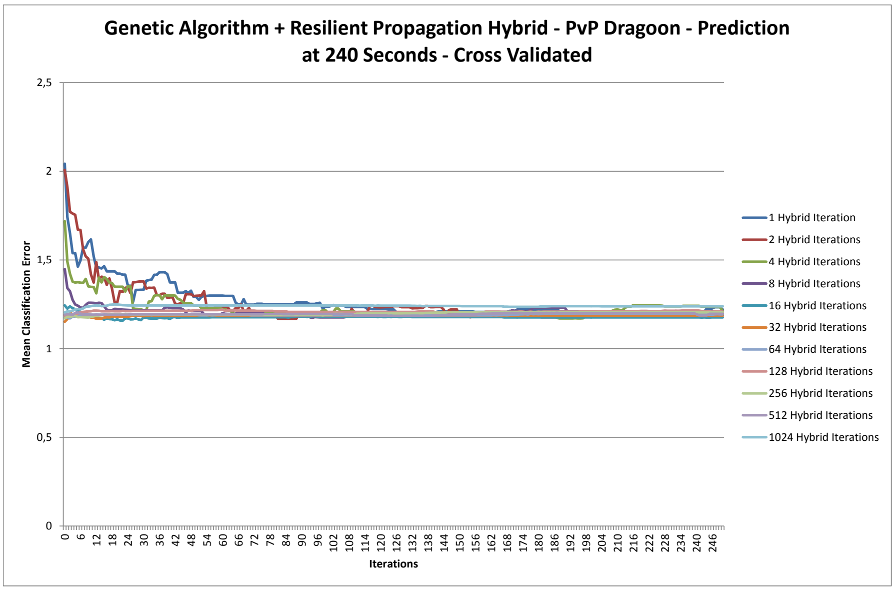


Figure A.21: Mean Classification Error for predictions made at 240 seconds

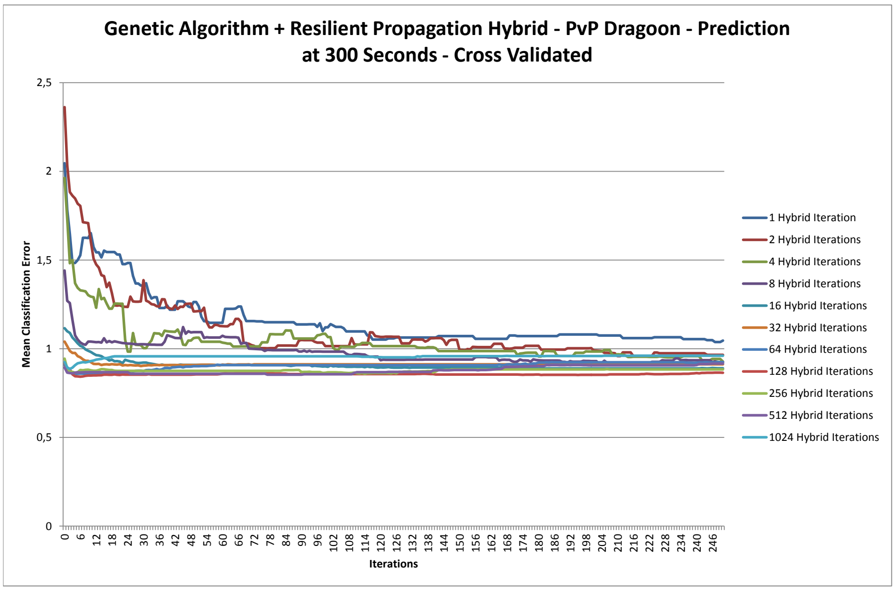


Figure A.22: Mean Classification Error for predictions made at 300 seconds

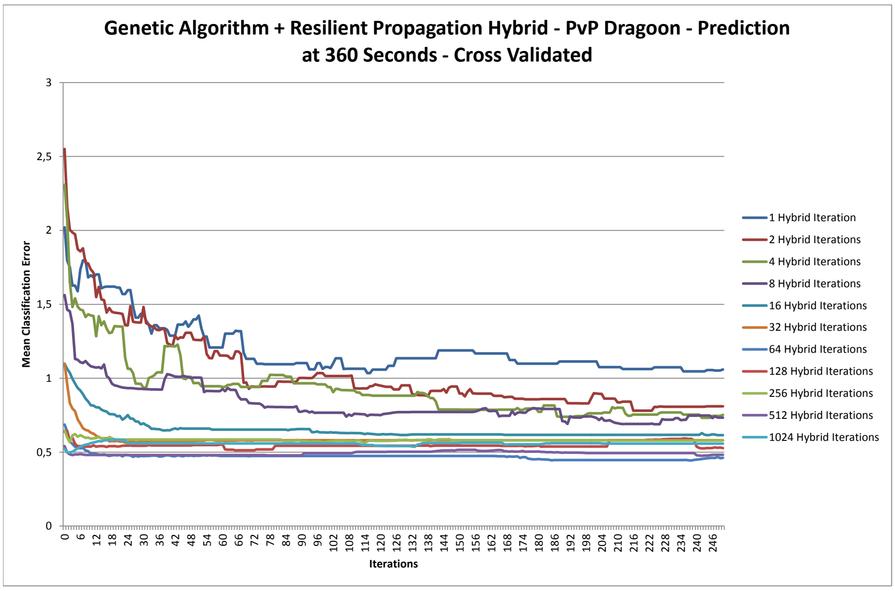


Figure A.23: Mean Classification Error for predictions made at 360 seconds

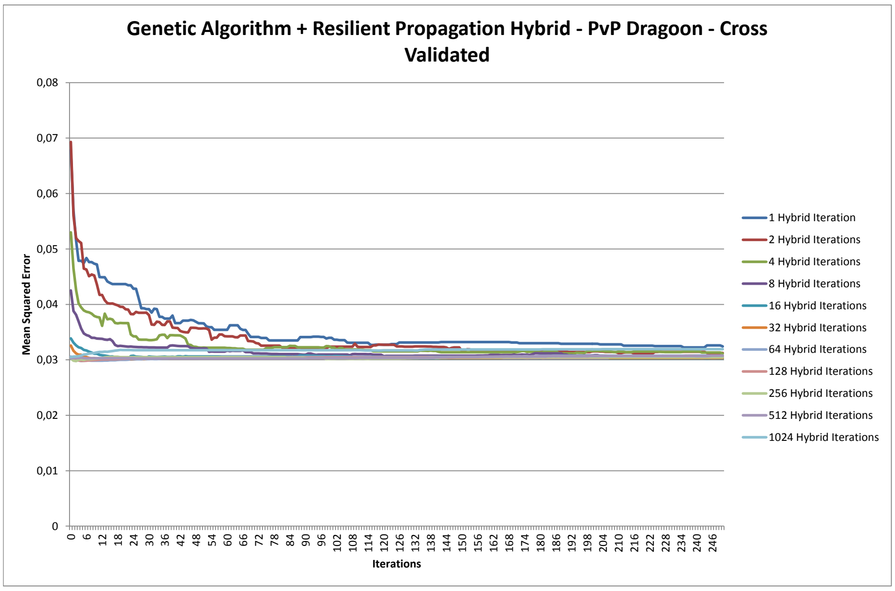


Figure A.24: Mean Square Error

A.1.7 Keepbest

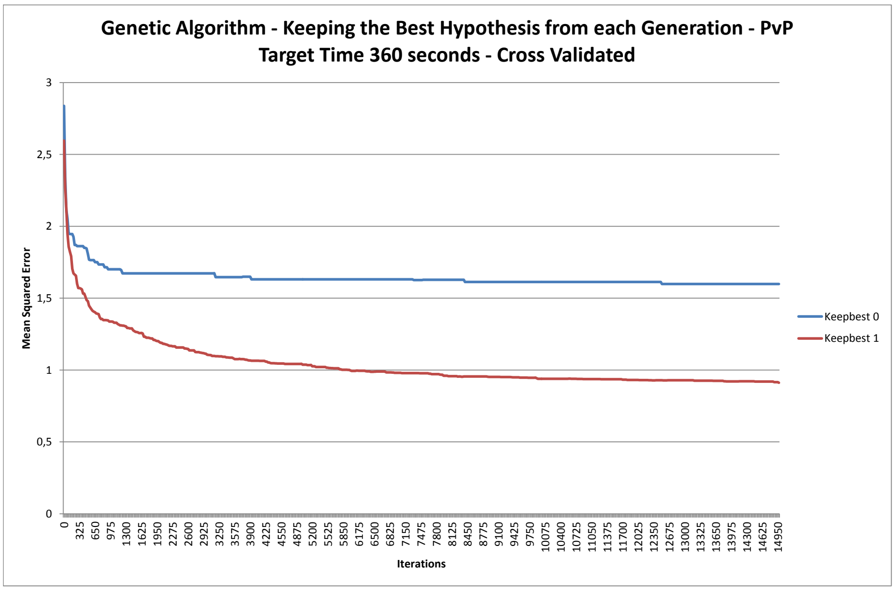


Figure A.25: Mean Square Error

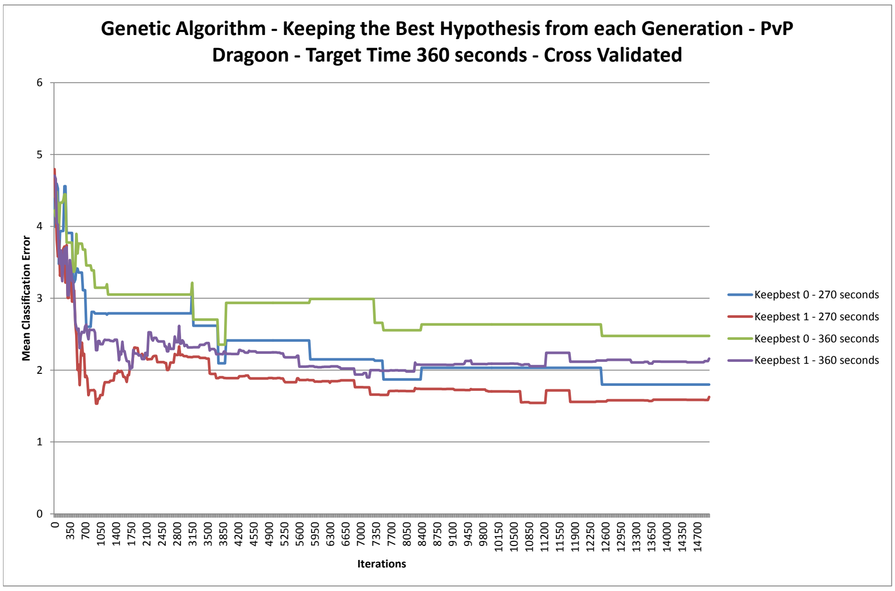


Figure A.26: Mean Classification Error for predictions made at 270 and 360 seconds

A.2 Feature Selection

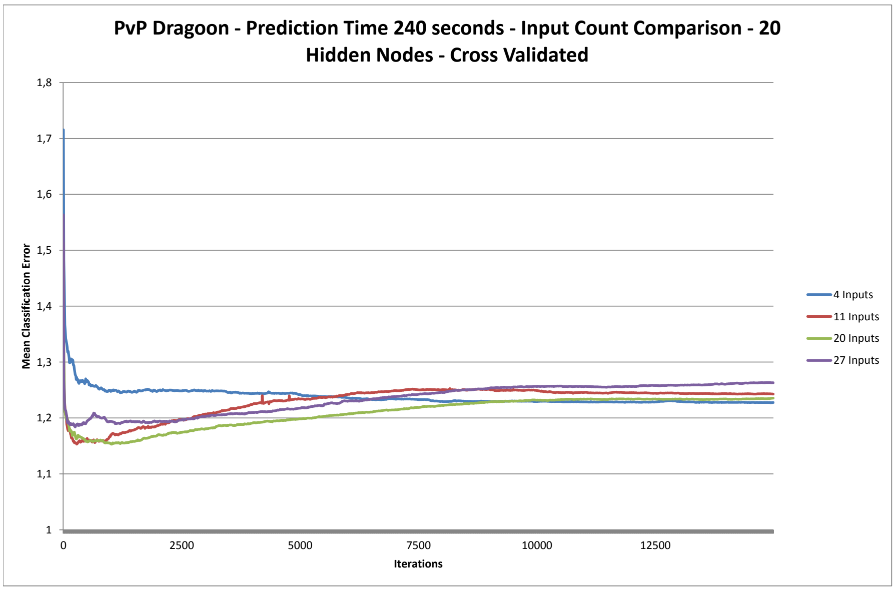


Figure A.27: Mean Classification Error for predictions made at 240 seconds

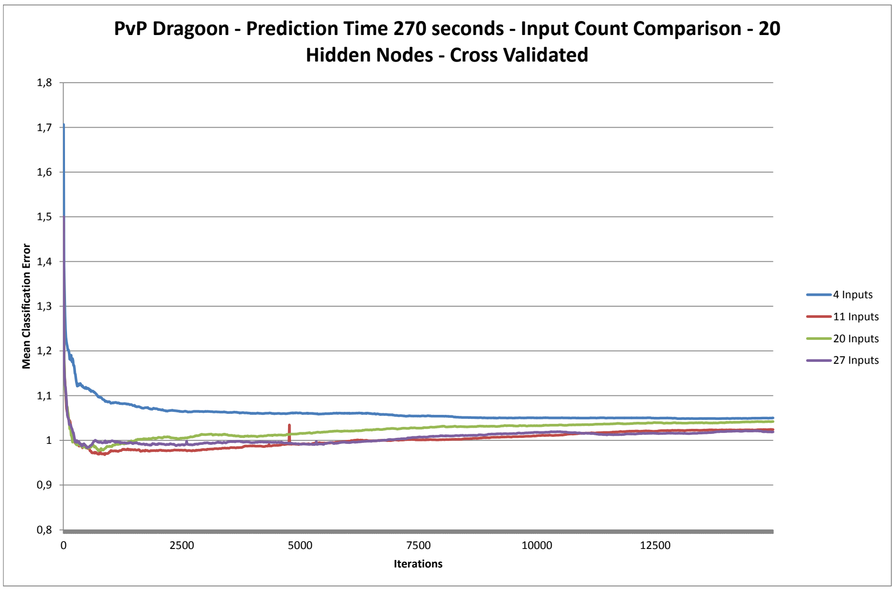


Figure A.28: Mean Classification Error for predictions made at 270 seconds

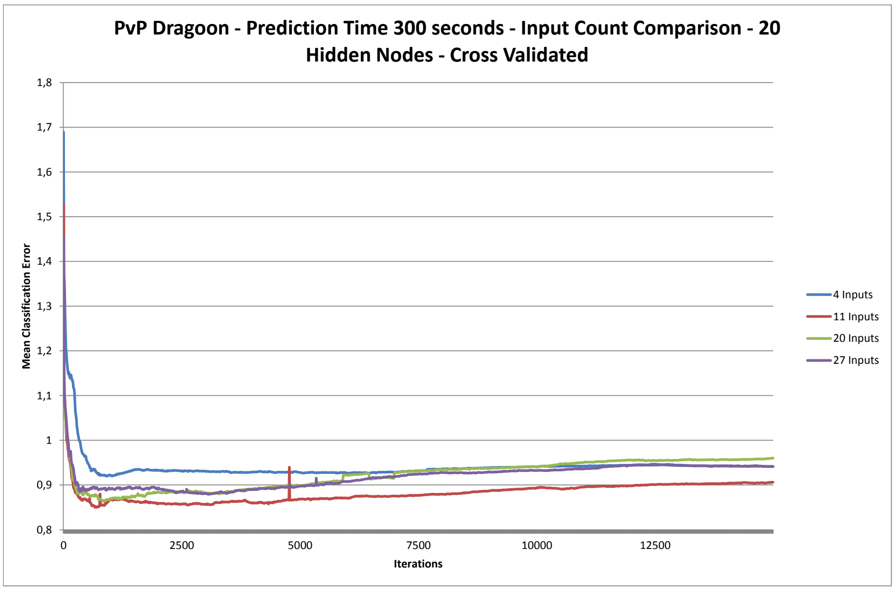


Figure A.29: Mean Classification Error for predictions made at 300 seconds

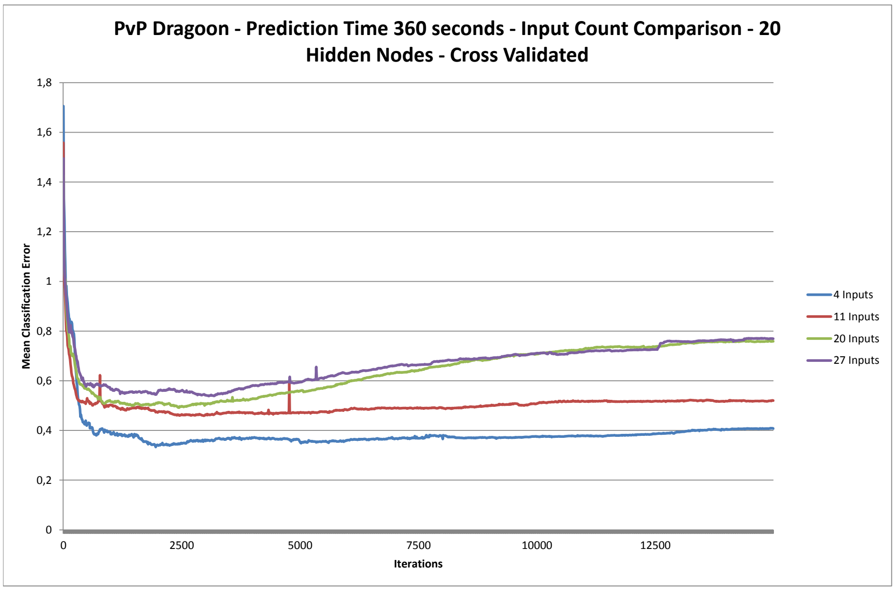
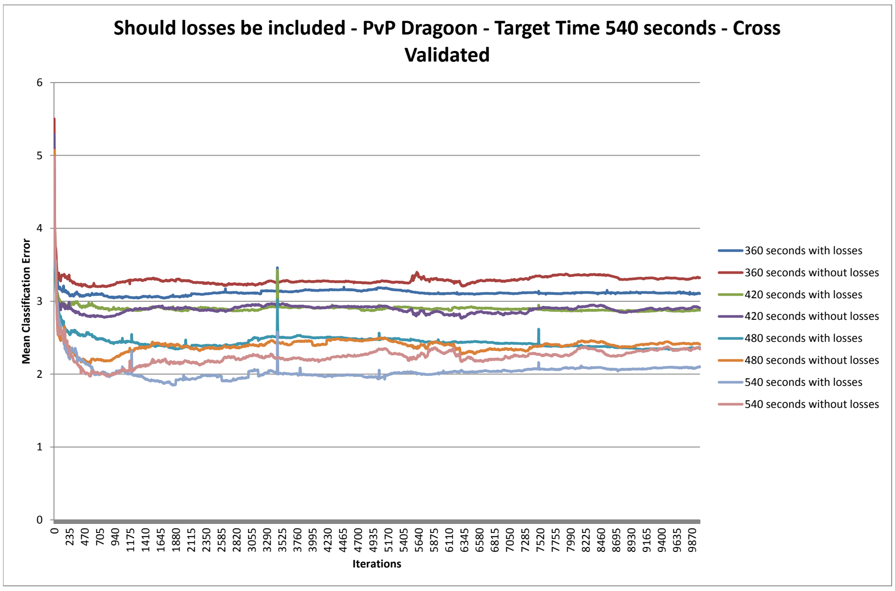
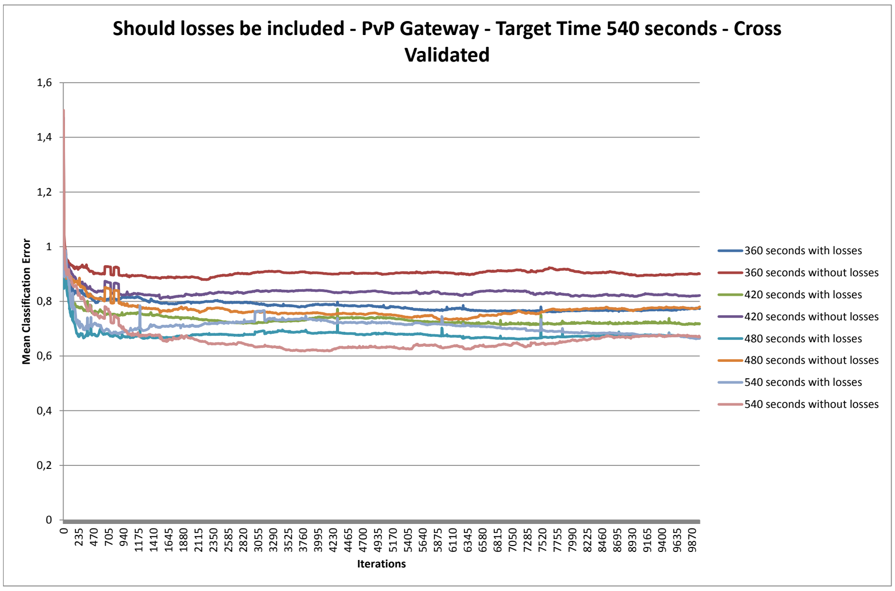


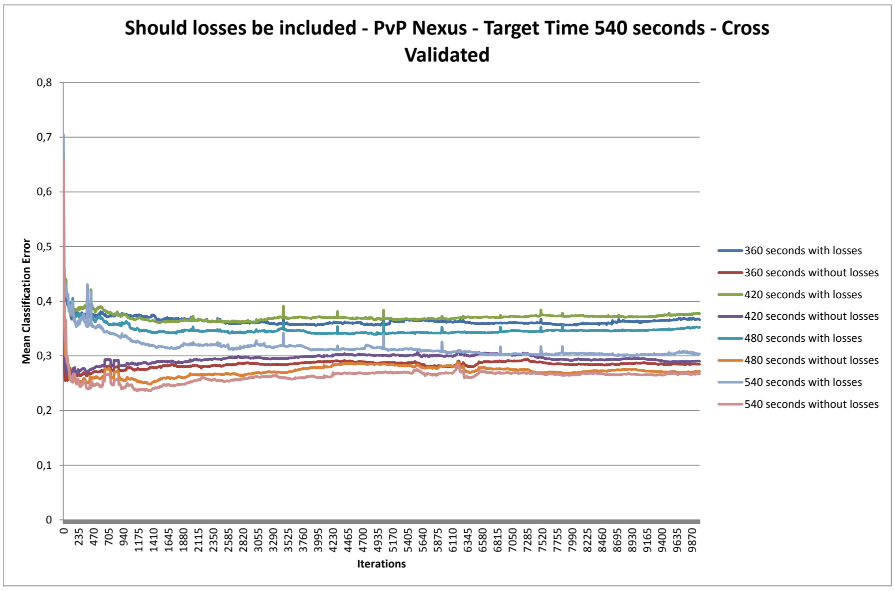
Figure A.30: Mean Classification Error for predictions made at 360 seconds

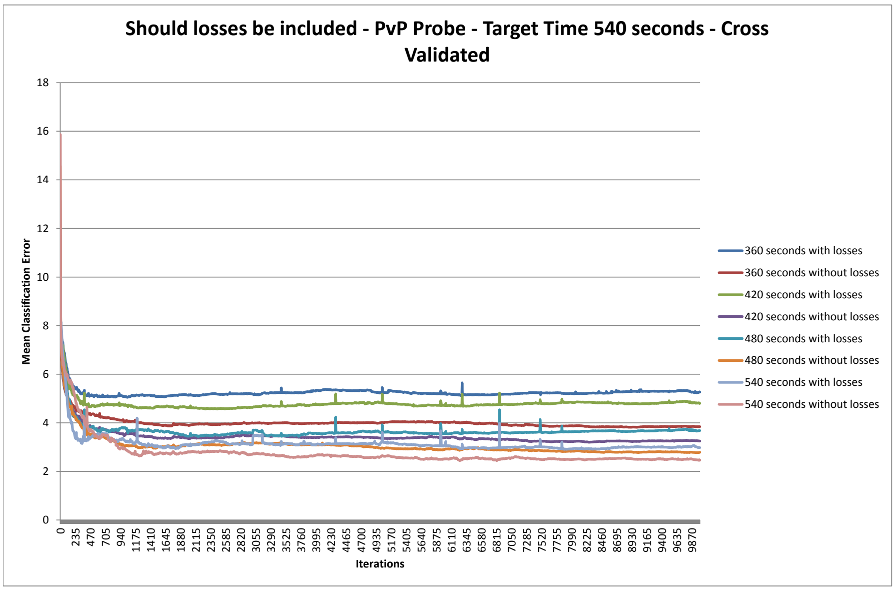
A.3 Game State Loss Reflection

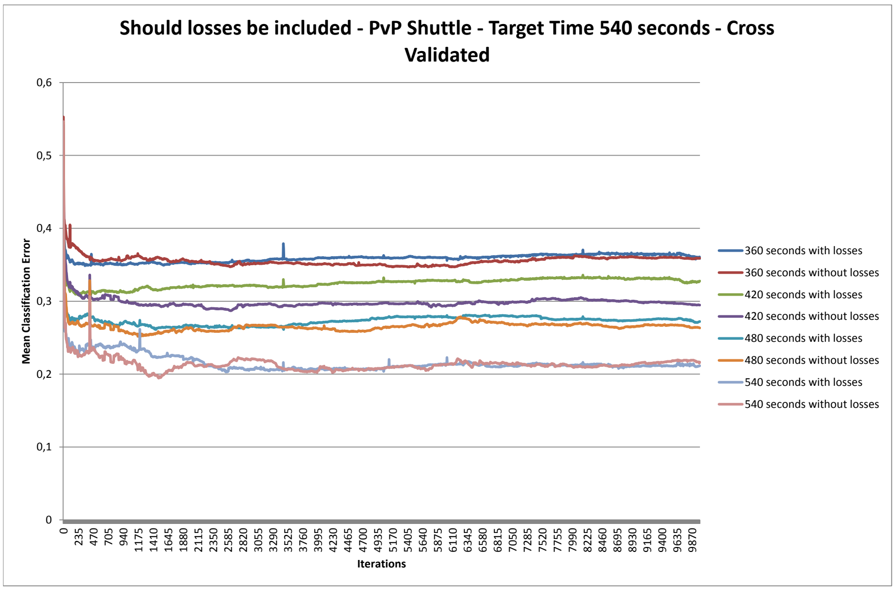
A.3.1 Target time 540 seconds

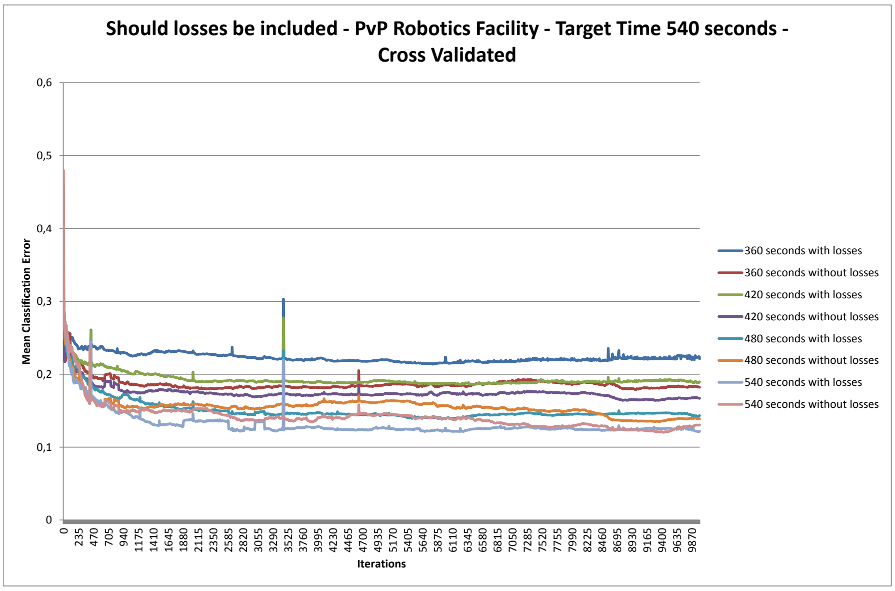


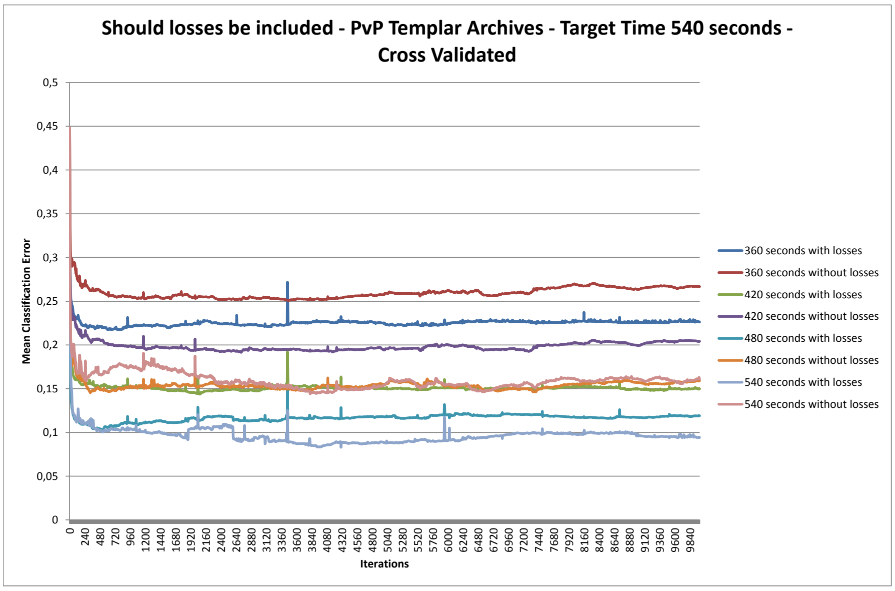

Should losses be included - PvP Gateway - Target Time 540 seconds - Cross Validated
Mean Classification Error
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
Iterations
360 seconds with losses
360 seconds without losses
420 seconds with losses
420 seconds without losses
480 seconds with losses
480 seconds without losses
540 seconds with losses
540 seconds without losses

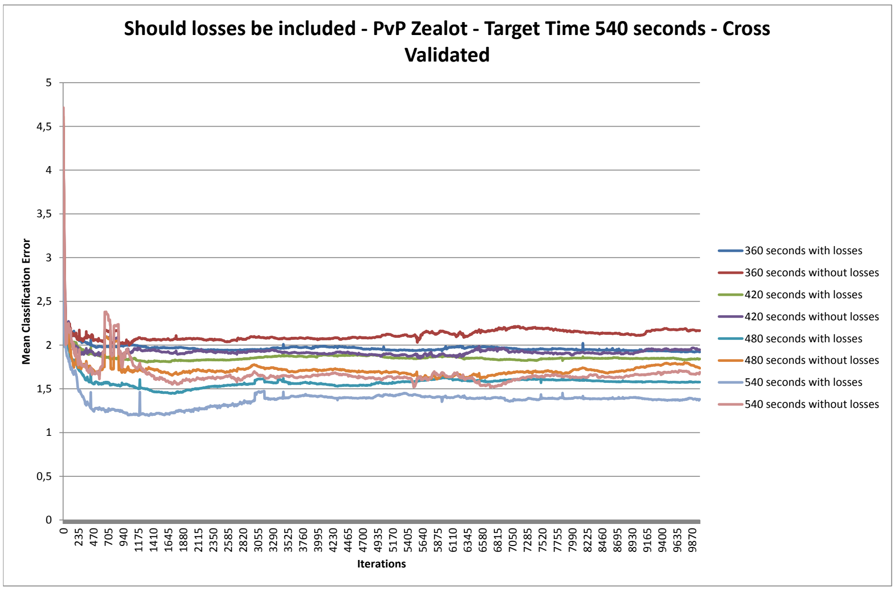

Should losses be included - PvP Nexus - Target Time 540 seconds - Cross Validated
Mean Classification Error
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
Iterations
360 seconds with losses
360 seconds without losses
420 seconds with losses
420 seconds without losses
480 seconds with losses
480 seconds without losses
540 seconds with losses
540 seconds without losses


Should losses be included - PvP Probe - Target Time 540 seconds - Cross Validated
Mean Classification Error
0
2
4
6
8
10
12
14
16
18
Iterations
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
360 seconds with losses
360 seconds without losses
420 seconds with losses
420 seconds without losses
480 seconds with losses
480 seconds without losses
540 seconds with losses
540 seconds without losses

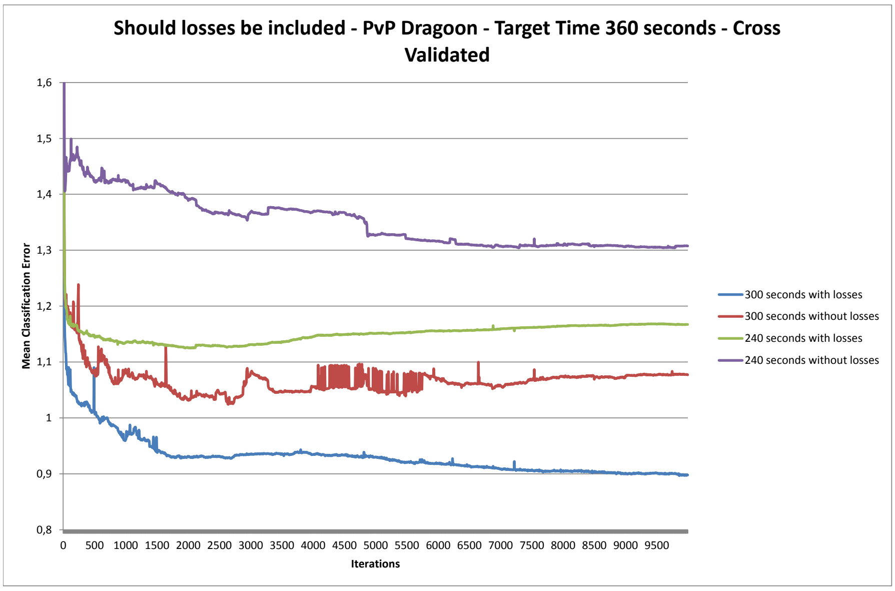

Should losses be included - PvP Shuttle - Target Time 540 seconds - Cross Validated
Mean Classification Error
0
0,1
0,2
0,3
0,4
0,5
0,6
Iterations
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
360 seconds with losses
360 seconds without losses
420 seconds with losses
420 seconds without losses
480 seconds with losses
480 seconds without losses
540 seconds with losses
540 seconds without losses

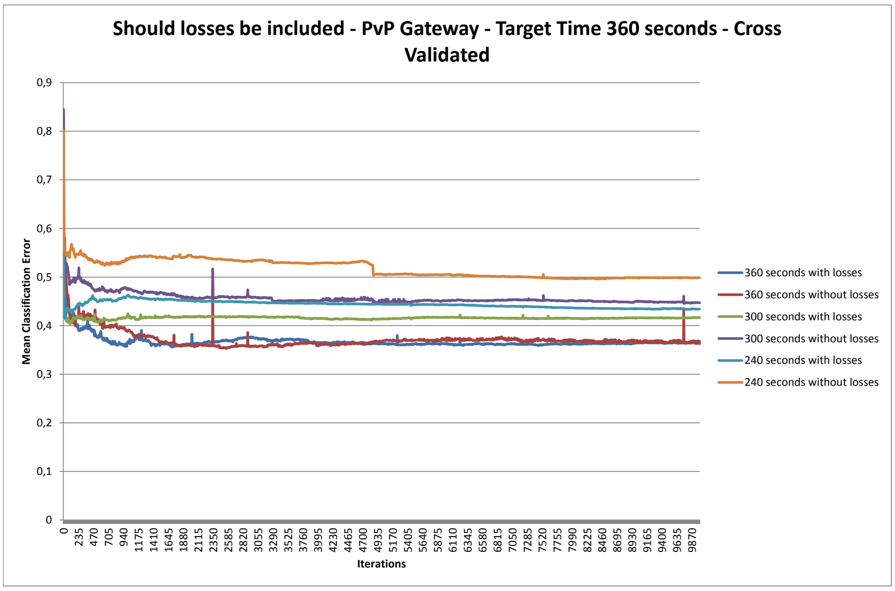

Should losses be included - PvP Robotics Facility - Target Time 540 seconds - Cross Validated
Mean Classification Error
0,6
0,5
0,4
0,3
0,2
0,1
0
Iterations
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
360 seconds with losses
360 seconds without losses
420 seconds with losses
420 seconds without losses
480 seconds with losses
480 seconds without losses
540 seconds with losses
540 seconds without losses


Should losses be included - PvP Templar Archives - Target Time 540 seconds - Cross Validated
Mean Classification Error
0,5
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
240
480
720
960
1200
1440
1680
1920
2160
2400
2640
2880
3120
3360
3600
3840
4080
4320
4560
4800
5040
5280
5520
5760
6000
6240
6480
6720
6960
7200
7440
7680
7920
8160
8400
8640
8880
9120
9360
9600
9840
360 seconds with losses
360 seconds without losses
420 seconds with losses
420 seconds without losses
480 seconds with losses
480 seconds without losses
540 seconds with losses
540 seconds without losses

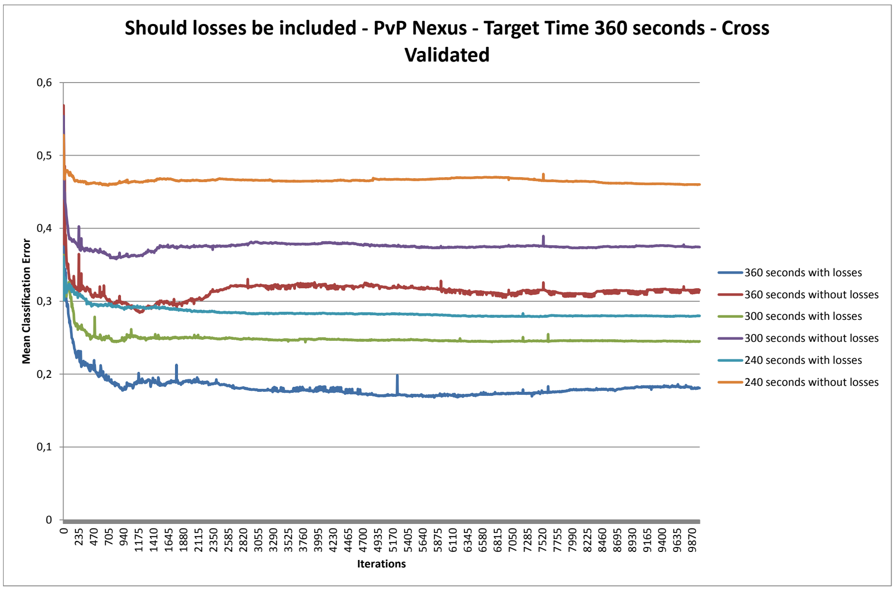


A.3.2 Target time 360 seconds

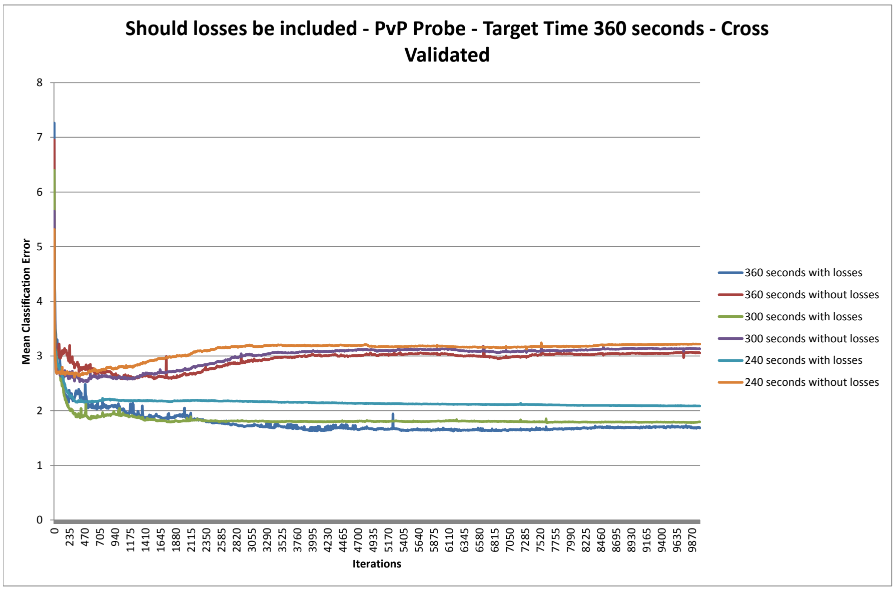


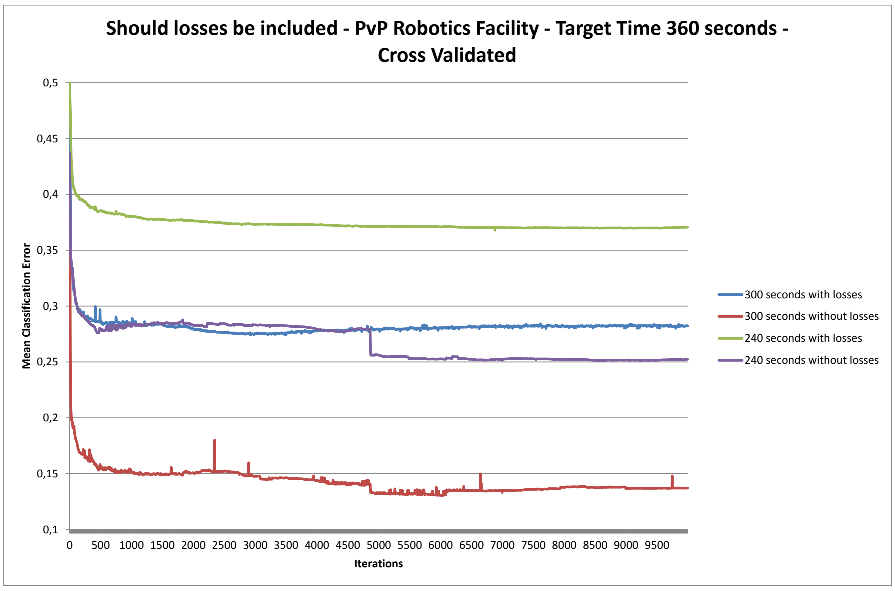


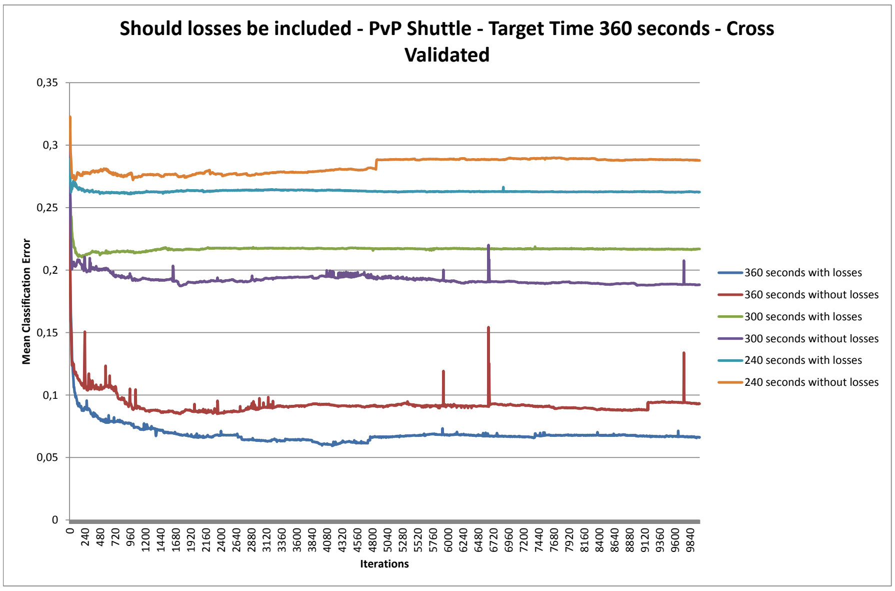
Should losses be included - PvP Gateway - Target Time 360 seconds - Cross Validated
Mean Classification Error
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Iterations
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
360 seconds with losses
360 seconds without losses
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses

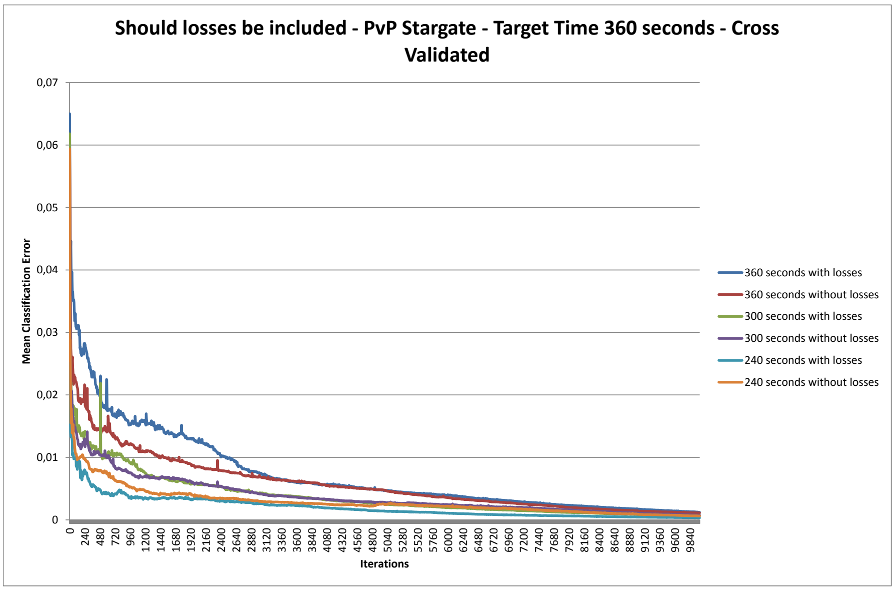


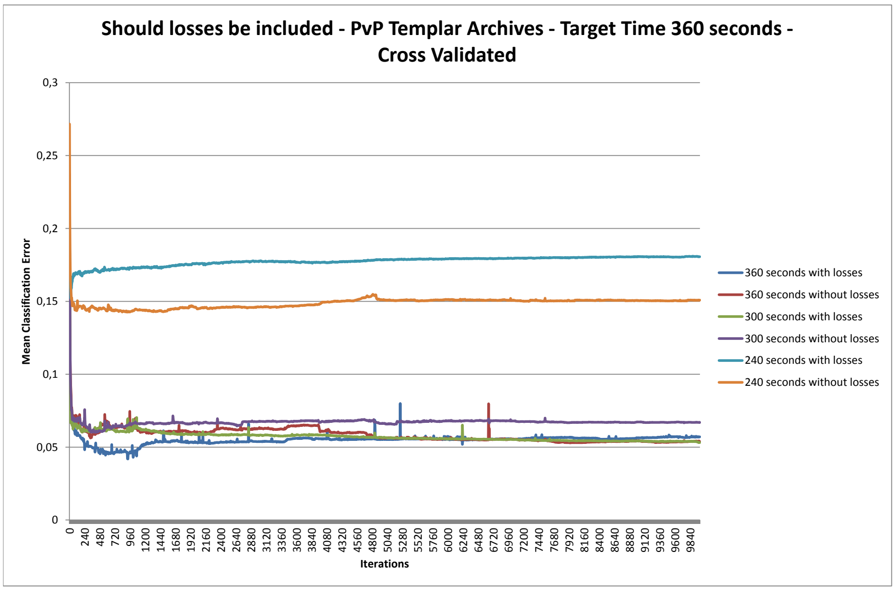
Should losses be included - PvP Nexus - Target Time 360 seconds - Cross Validated
Mean Classification Error
0,6
0,5
0,4
0,3
0,2
0,1
0
Iterations
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
360 seconds with losses
360 seconds without losses
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses

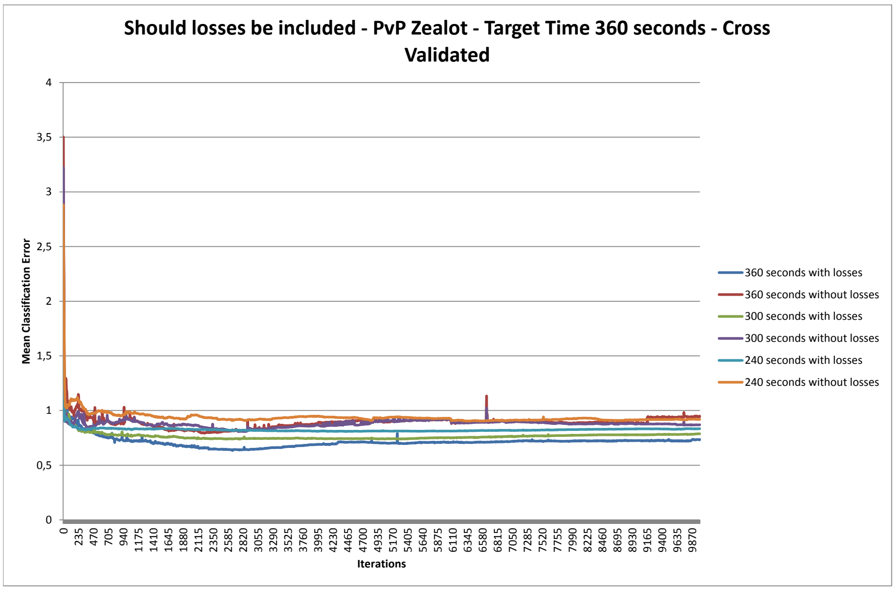

Should losses be included - PvP Probe - Target Time 360 seconds - Cross Validated
Mean Classification Error
0
1
2
3
4
5
6
7
8
Iterations
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
360 seconds with losses
360 seconds without losses
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses

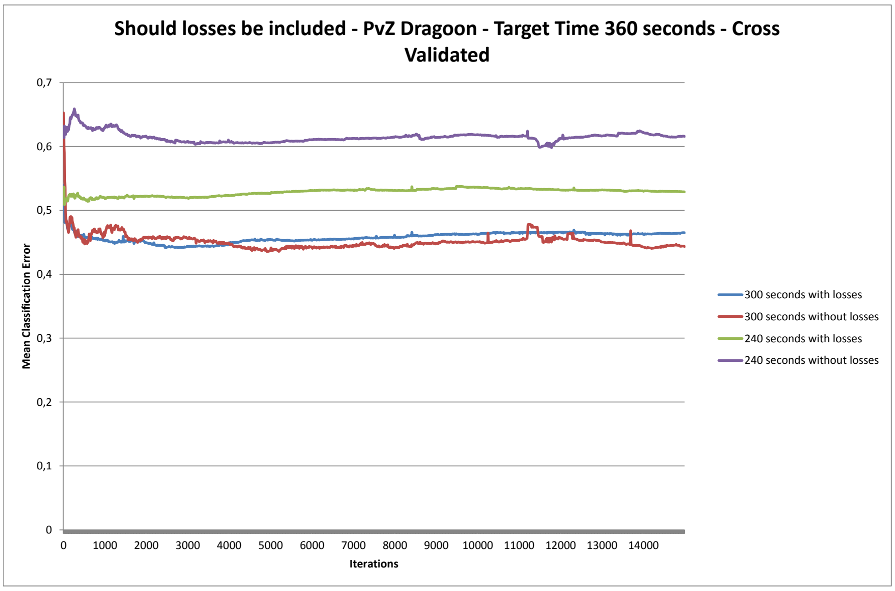

Should losses be included - PvP Robotics Facility - Target Time 360 seconds - Cross Validated
Mean Classification Error
0,1
0,15
0,2
0,25
0,3
0,35
0,4
0,45
0,5
Iterations
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
8500
9000
9500
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses

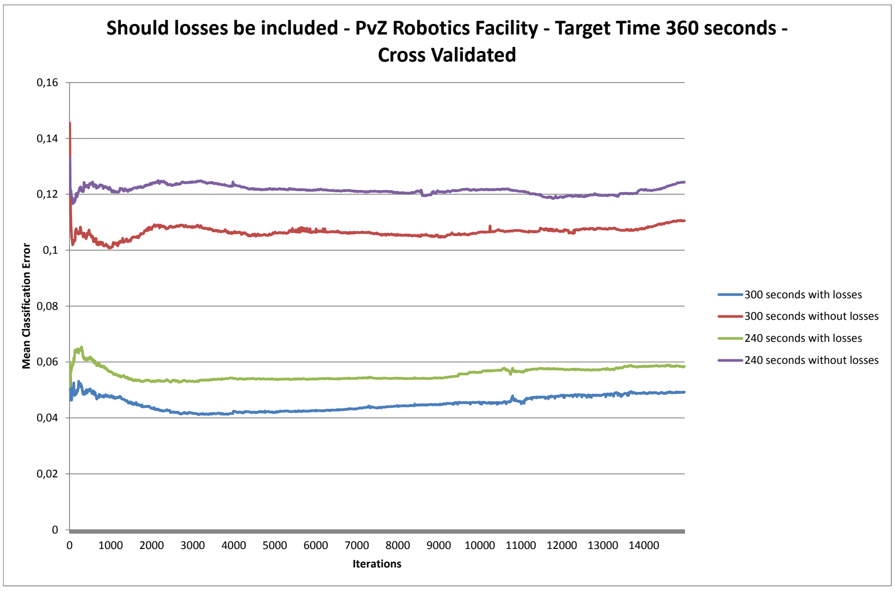

Should losses be included - PvP Shuttle - Target Time 360 seconds - Cross Validated
Mean Classification Error
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
240
480
720
960
1200
1440
1680
1920
2160
2400
2640
2880
3120
3360
3600
3840
4080
4320
4560
4800
5040
5280
5520
5760
6000
6240
6480
6720
6960
7200
7440
7680
7920
8160
8400
8640
8880
9120
9360
9600
9840
360 seconds with losses
360 seconds without losses
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses

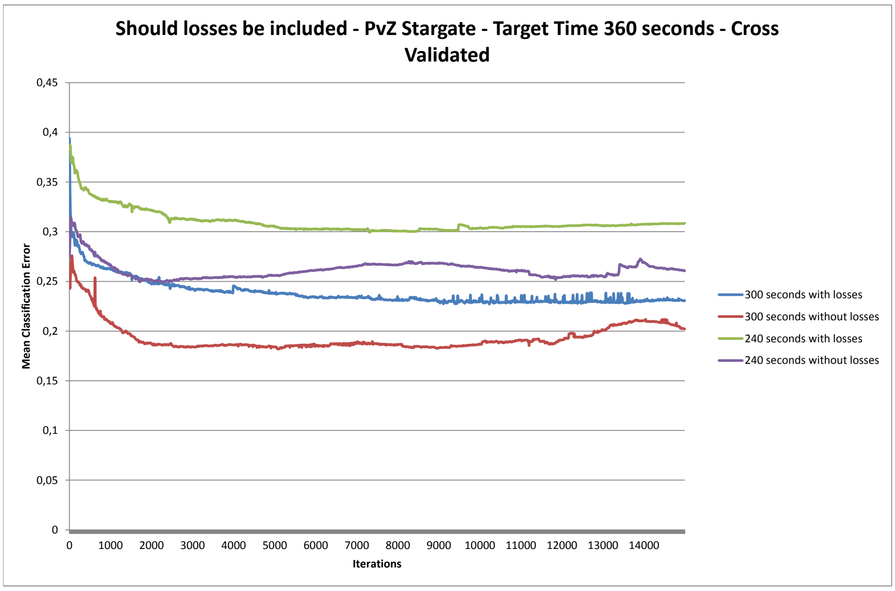

Should losses be included - PvP Stargate - Target Time 360 seconds - Cross Validated
Mean Classification Error
0,07
0,06
0,05
0,04
0,03
0,02
0,01
0
Iterations
0
240
480
720
960
1200
1440
1680
1920
2160
2400
2640
2880
3120
3360
3600
3840
4080
4320
4560
4800
5040
5280
5520
5760
6000
6240
6480
6720
6960
7200
7440
7680
7920
8160
8400
8640
8880
9120
9360
9600
9840
360 seconds with losses
360 seconds without losses
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses


Should losses be included - PvP Templar Archives - Target Time 360 seconds - Cross Validated
0,3
0,25
0,2
0,15
0,1
0,05
0
Mean Classification Error
0
240
480
720
960
1200
1440
1680
1920
2160
2400
2640
2880
3120
3360
3600
3840
4080
4320
4560
4800
5040
5280
5520
5760
6000
6240
6480
6720
6960
7200
7440
7680
7920
8160
8400
8640
8880
9120
9360
9600
9840
Iterations
360 seconds with losses
360 seconds without losses
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses


Should losses be included - PvP Zealot - Target Time 360 seconds - Cross Validated
4
3,5
3
2,5
2
1,5
1
0,5
0
Mean Classification Error
0
235
470
705
940
1175
1410
1645
1880
2115
2350
2585
2820
3055
3290
3525
3760
3995
4230
4465
4700
4935
5170
5405
5640
5875
6110
6345
6580
6815
7050
7285
7520
7755
7990
8225
8460
8695
8930
9165
9400
9635
9870
Iterations
360 seconds with losses
360 seconds without losses
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses

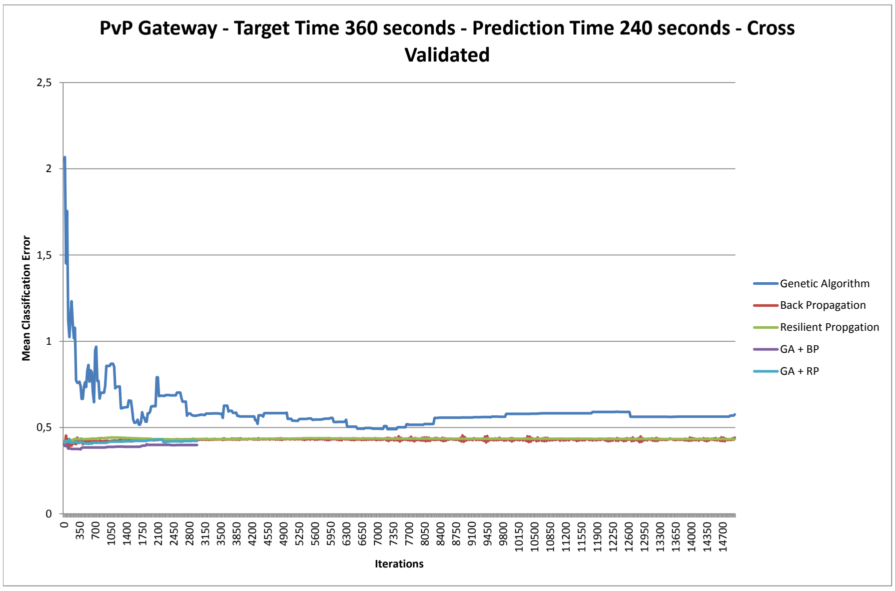

Should losses be included - PvZ Dragoon - Target Time 360 seconds - Cross Validated
Mean Classification Error
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
Iterations
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses

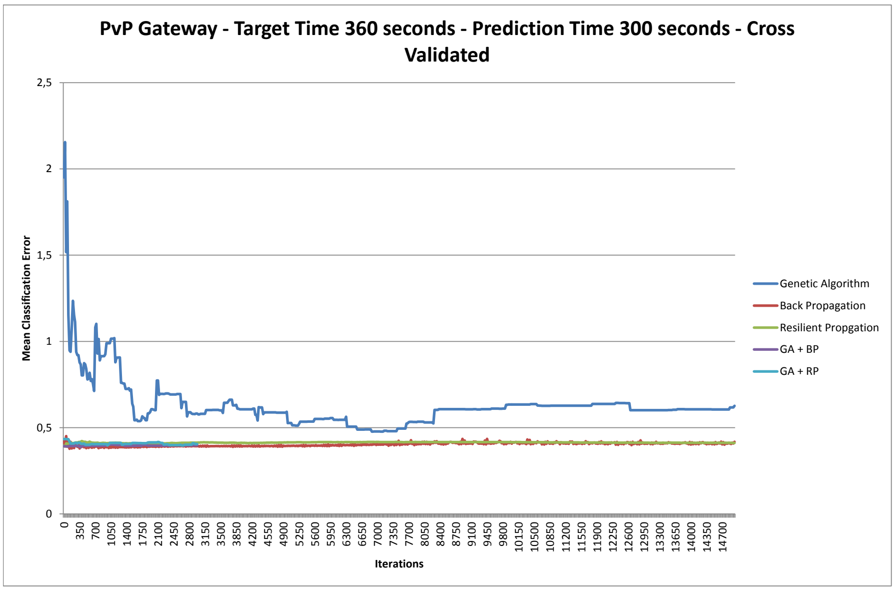

Should losses be included - PvZ Robotics Facility - Target Time 360 seconds - Cross Validated
Mean Classification Error
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
Iterations
300 seconds with losses
300 seconds without losses
240 seconds with losses
240 seconds without losses

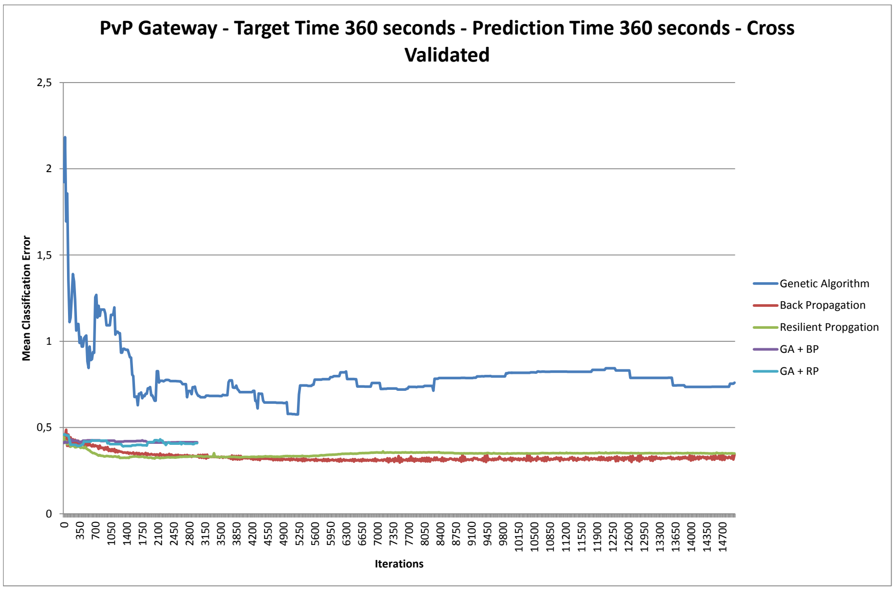


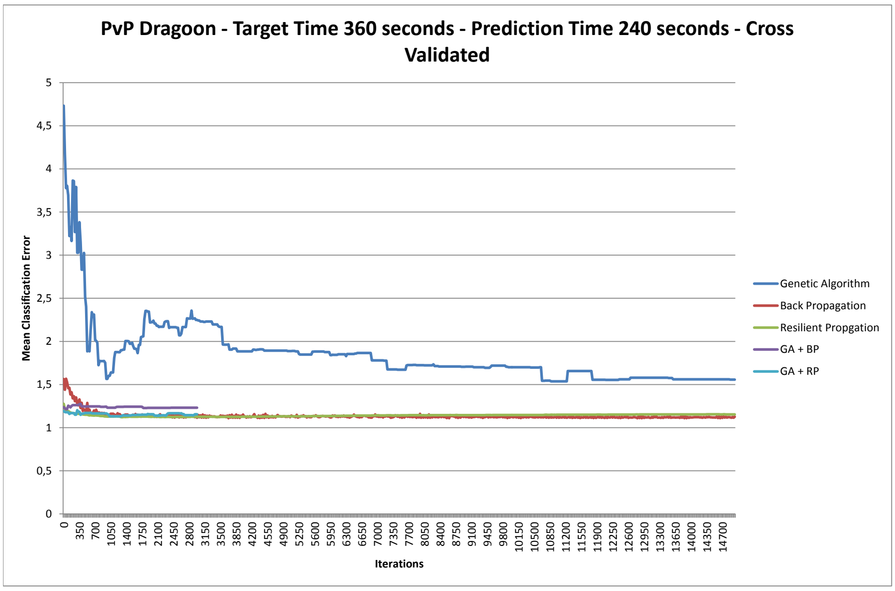
A.4 Training Algorithms Comparison

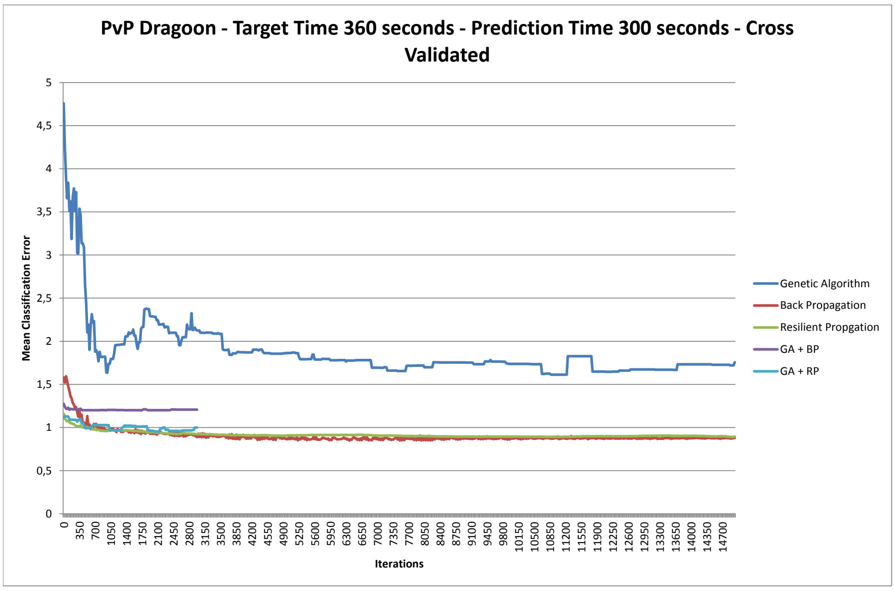
A.4.1 Protoss vs Protoss

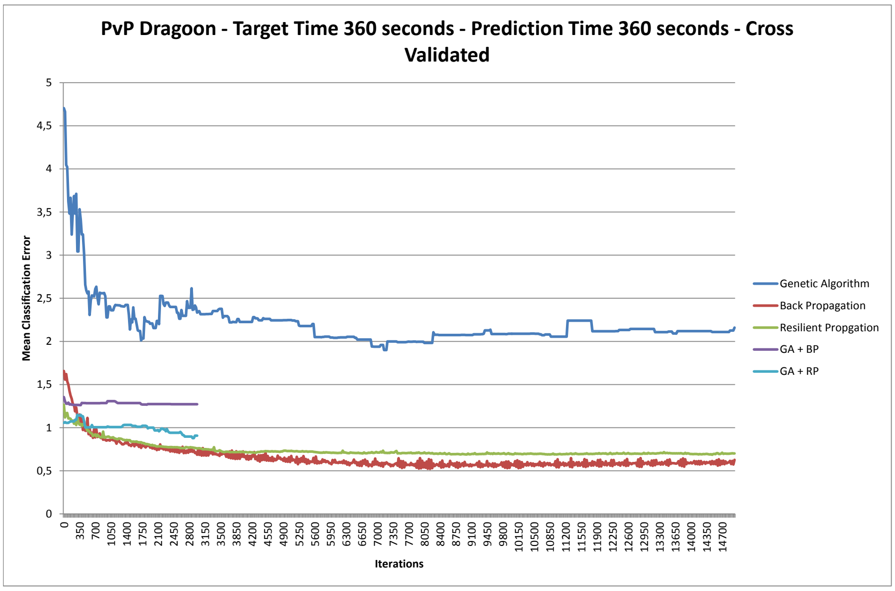


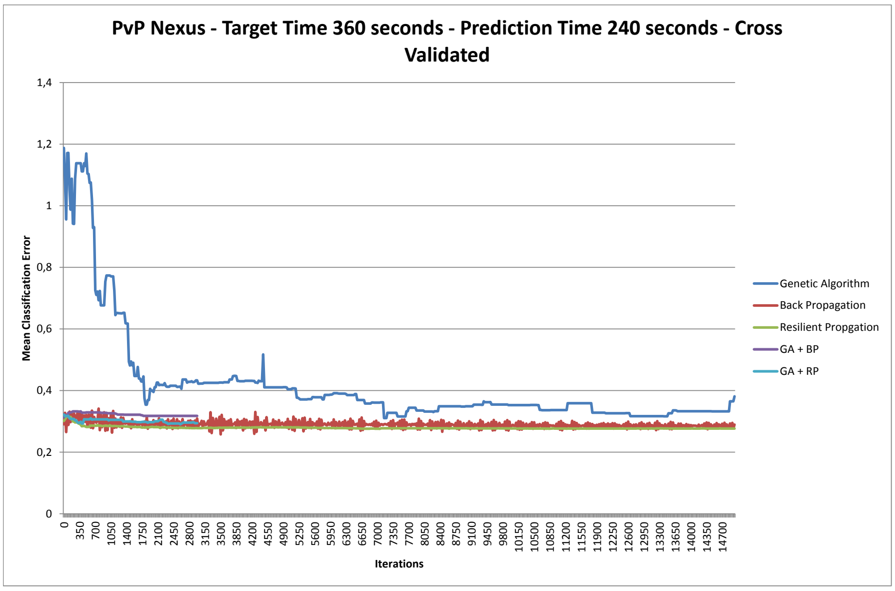

PvP Gateway - Target Time 360 seconds - Prediction Time 300 seconds - Cross Validated
Mean Classification Error
2,5
2
1,5
1
0,5
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

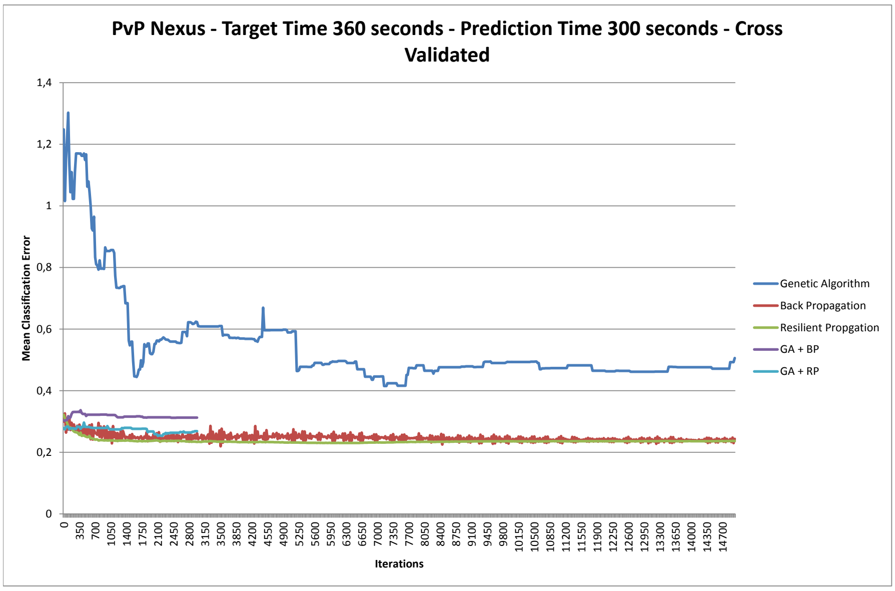

PvP Gateway - Target Time 360 seconds - Prediction Time 360 seconds - Cross Validated
Mean Classification Error
2,5
2
1,5
1
0,5
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

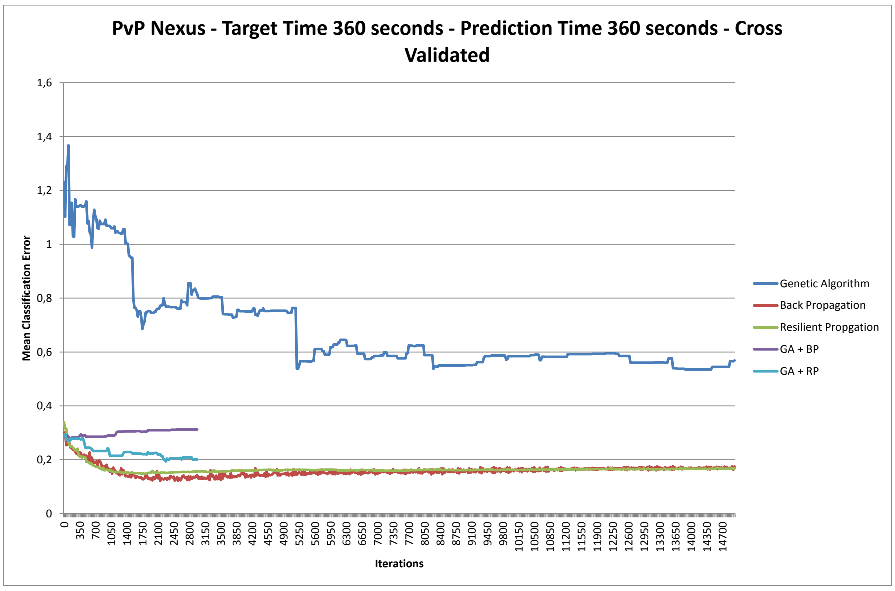

PvP Dragoon - Target Time 360 seconds - Prediction Time 240 seconds - Cross Validated
Mean Classification Error
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Iterations
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

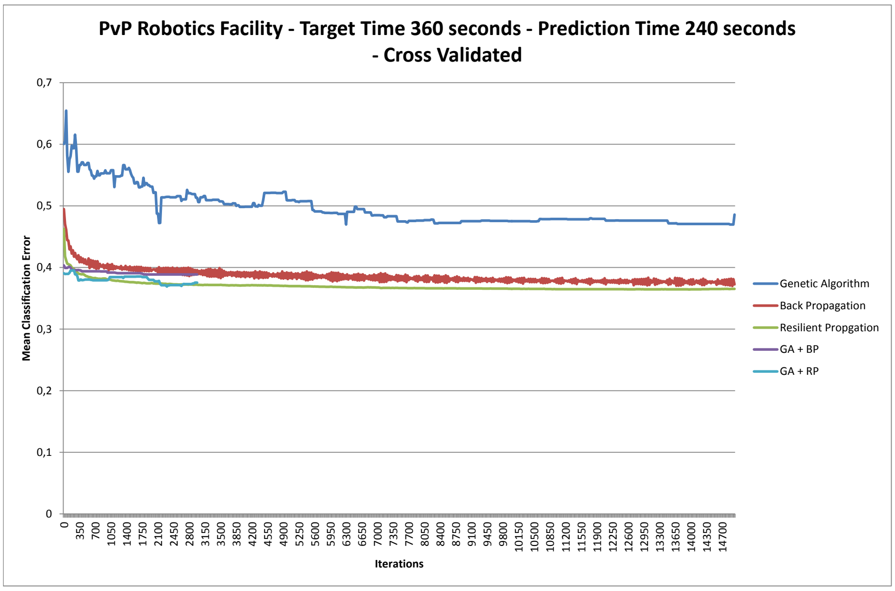

PvP Dragoon - Target Time 360 seconds - Prediction Time 300 seconds - Cross Validated
Mean Classification Error
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Iterations
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

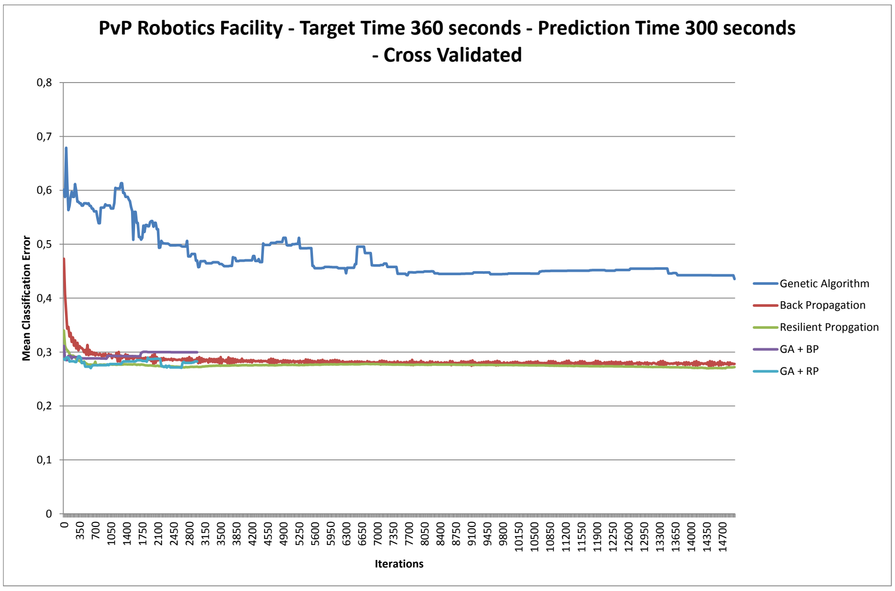

PvP Dragoon - Target Time 360 seconds - Prediction Time 360 seconds - Cross Validated
Mean Classification Error
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

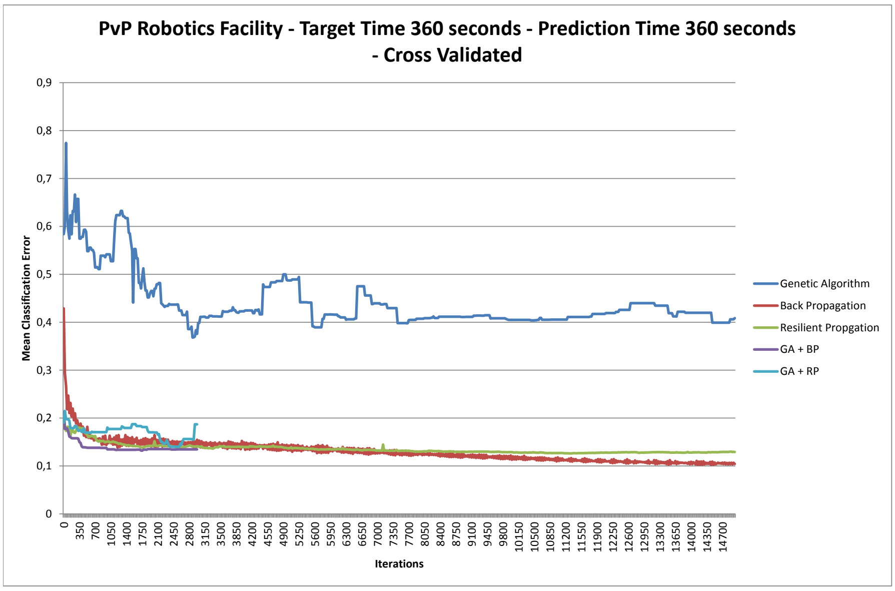

PvP Nexus - Target Time 360 seconds - Prediction Time 240 seconds - Cross Validated
Mean Classification Error
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

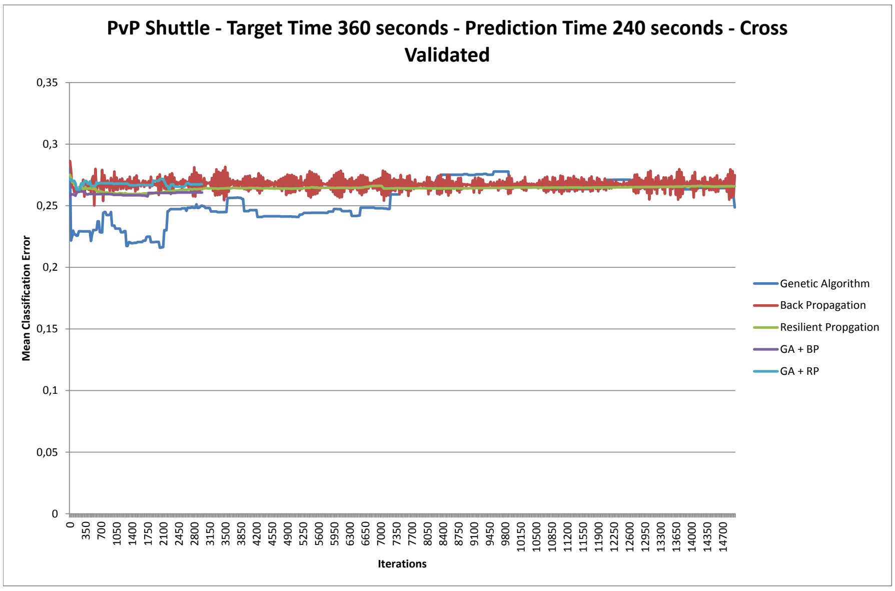

PvP Nexus - Target Time 360 seconds - Prediction Time 300 seconds - Cross Validated
Mean Classification Error
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

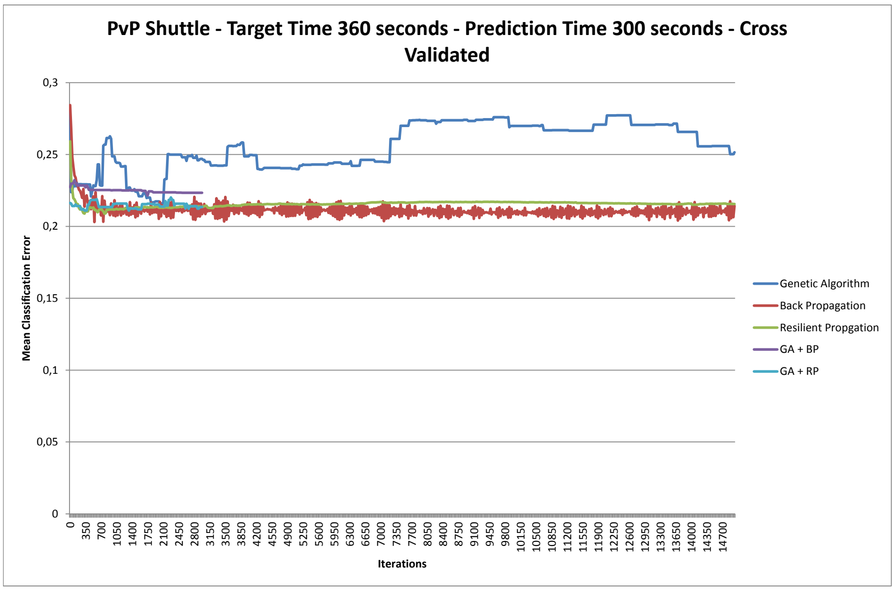

PvP Nexus - Target Time 360 seconds - Prediction Time 360 seconds - Cross Validated
Mean Classification Error
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP


PvP Robotics Facility - Target Time 360 seconds - Prediction Time 240 seconds - Cross Validated
Mean Classification Error
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

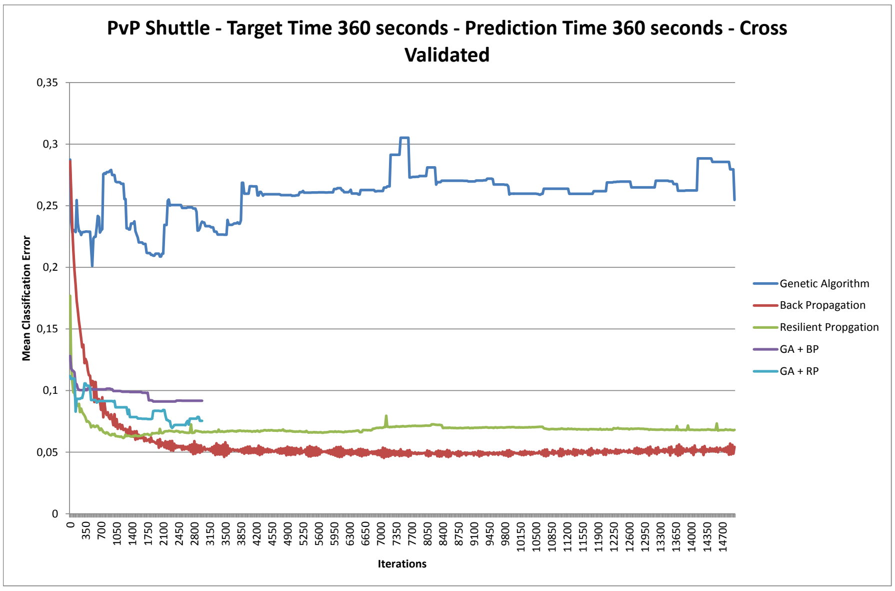

PvP Robotics Facility - Target Time 360 seconds - Prediction Time 300 seconds - Cross Validated
Mean Classification Error
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

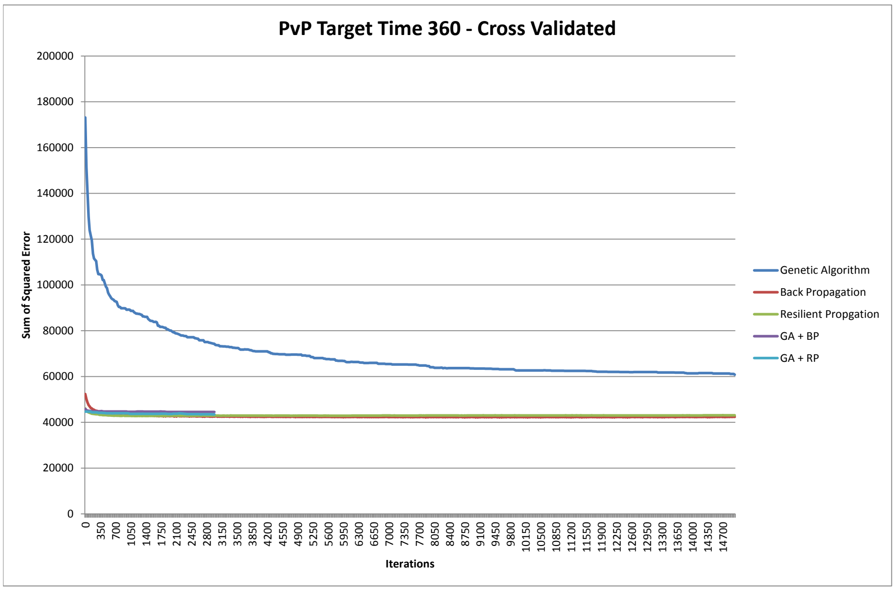

PvP Robotics Facility - Target Time 360 seconds - Prediction Time 360 seconds - Cross Validated
Mean Classification Error
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0 350 700 1050 1400 1750 2100 2450 2800 3150 3500 3850 4200 4550 4900 5250 5600 5950 6300 6650 7000 7350 7700 8050 8400 8750 9100 9450 9800 10150 10500 10850 11200 11550 11900 12250 12600 12950 13300 13650 14000 14350 14700
Iterations
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

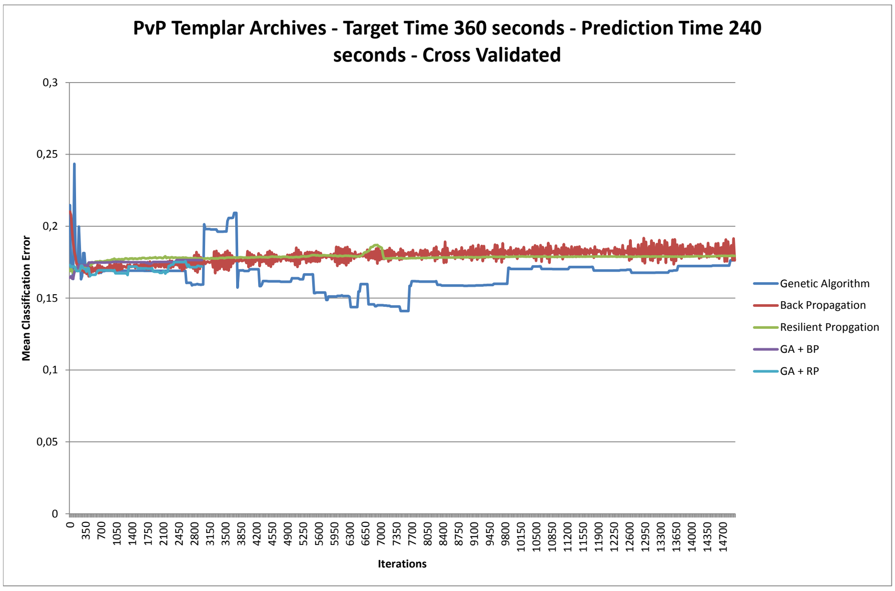

PvP Shuttle - Target Time 360 seconds - Prediction Time 240 seconds - Cross Validated
Mean Classification Error
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
0 350 700 1050 1400 1750 2100 2450 2800 3150 3500 3850 4200 4550 4900 5250 5600 5950 6300 6650 7000 7350 7700 8050 8400 8750 9100 9450 9800 10150 10500 10850 11200 11550 11900 12250 12600 12950 13300 13650 14000 14350 14700
Iterations
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

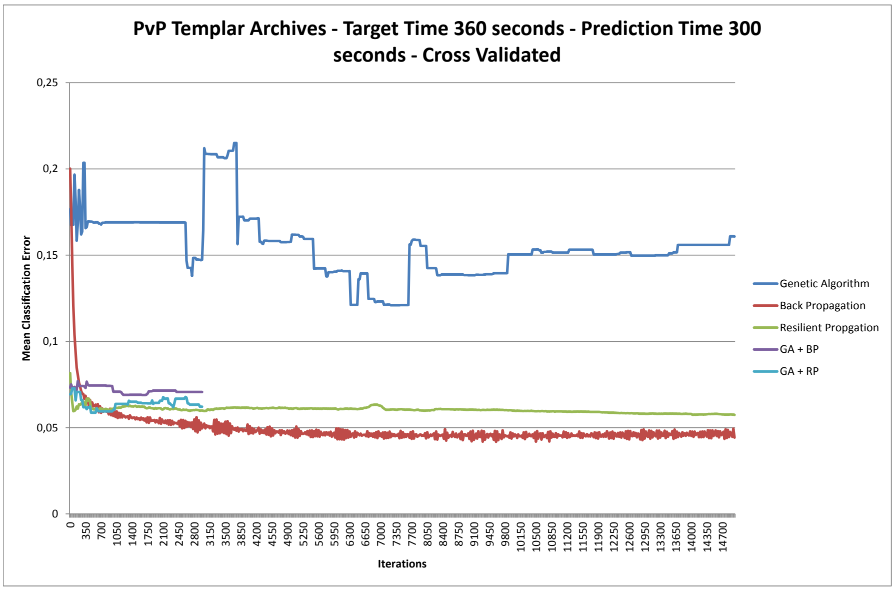

PvP Shuttle - Target Time 360 seconds - Prediction Time 300 seconds - Cross Validated
Mean Classification Error
0,3
0,25
0,2
0,15
0,1
0,05
0
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Iterations
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

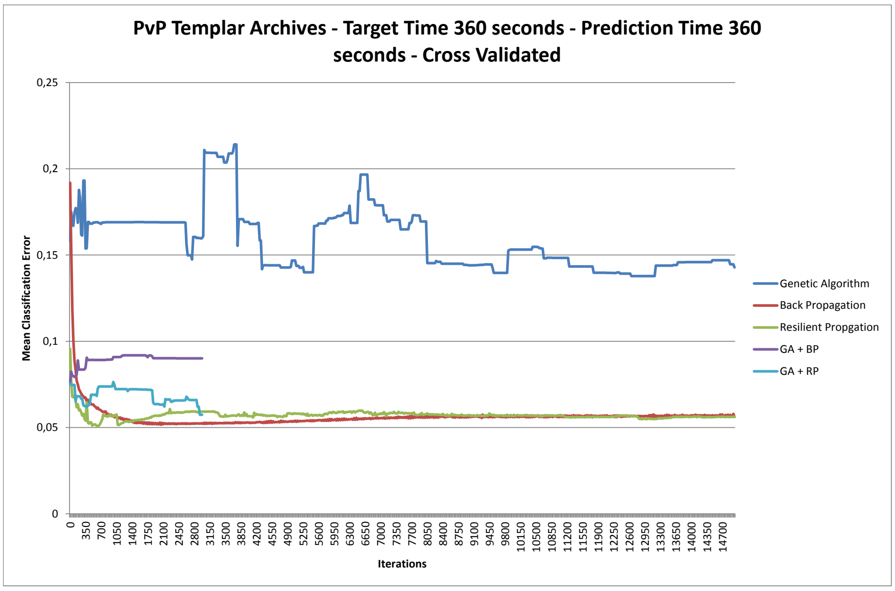
$\frac{1}{2}$

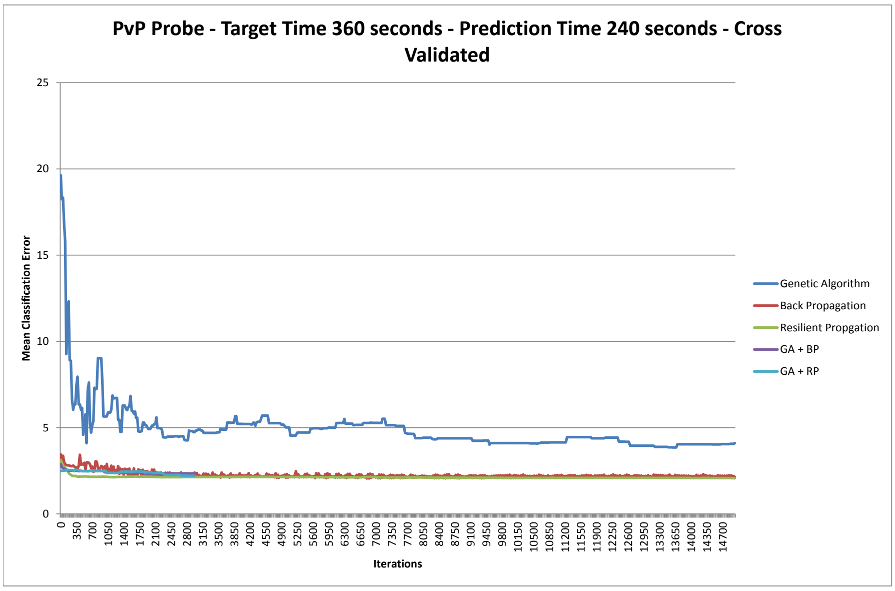

PvP Shuttle - Target Time 360 seconds - Prediction Time 360 seconds - Cross Validated
Mean Classification Error
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Iterations
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

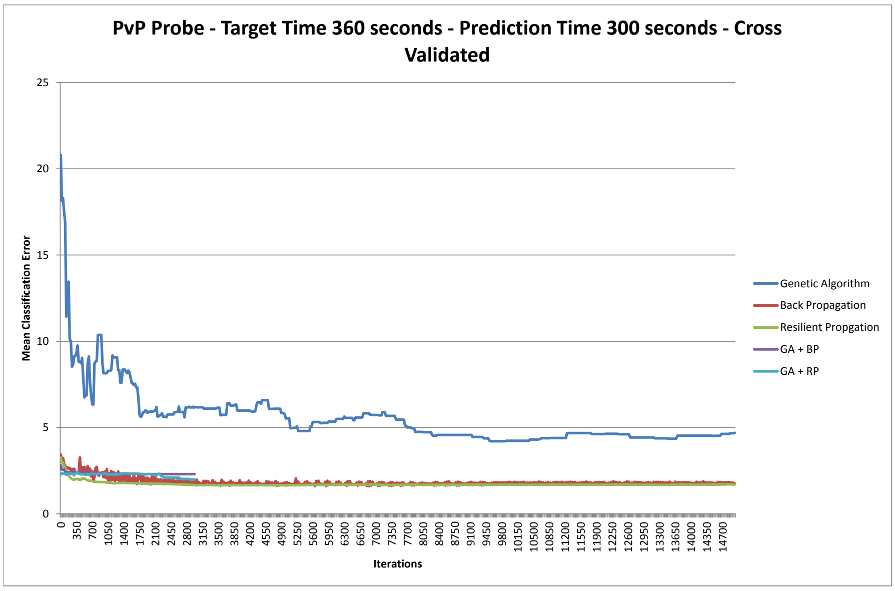

PvP Target Time 360 - Cross Validated
Sum of Squared Error
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
Iterations
0 350 700 1050 1400 1750 2100 2450 2800 3150 3500 3850 4200 4550 4900 5250 5600 5950 6300 6650 7000 7350 7700 8050 8400 8750 9100 9450 9800 10150 10500 10850 11200 11550 11900 12250 12600 12950 13300 13650 14000 14350 14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

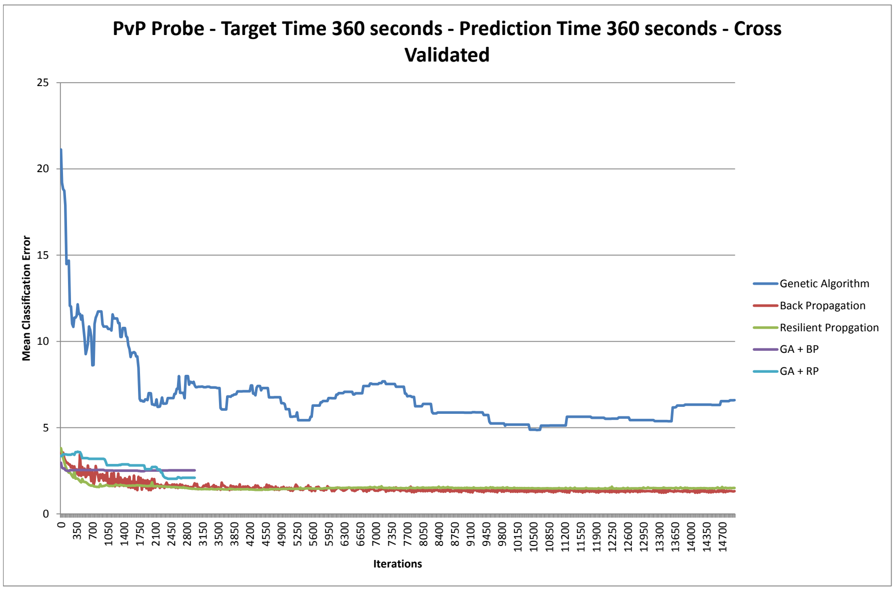

PvP Templar Archives - Target Time 360 seconds - Prediction Time 240 seconds - Cross Validated
Mean Classification Error
0,3
0,25
0,2
0,15
0,1
0,05
0
Iterations
0 350 700 1050 1400 1750 2100 2450 2800 3150 3500 3850 4200 4550 4900 5250 5600 5950 6300 6650 7000 7350 7700 8050 8400 8750 9100 9450 9800 10150 10500 10850 11200 11550 11900 12250 12600 12950 13300 13650 14000 14350 14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

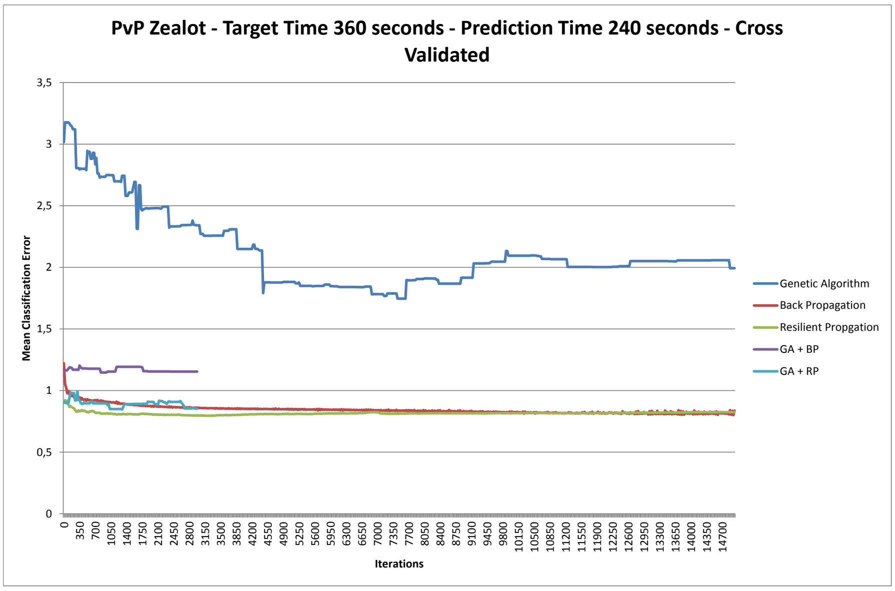

PvP Templar Archives - Target Time 360 seconds - Prediction Time 300 seconds - Cross Validated
Mean Classification Error
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

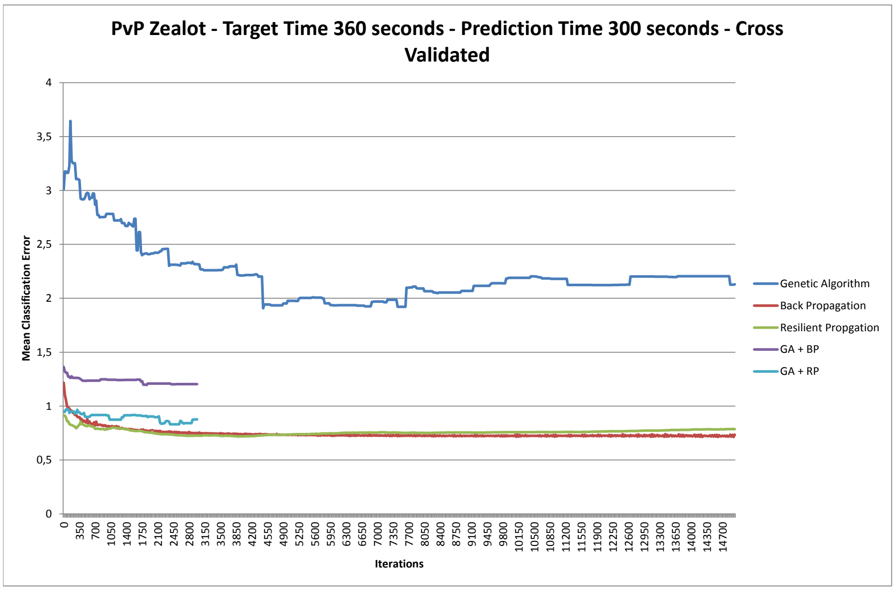

PvP Templar Archives - Target Time 360 seconds - Prediction Time 360 seconds - Cross Validated
Mean Classification Error
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

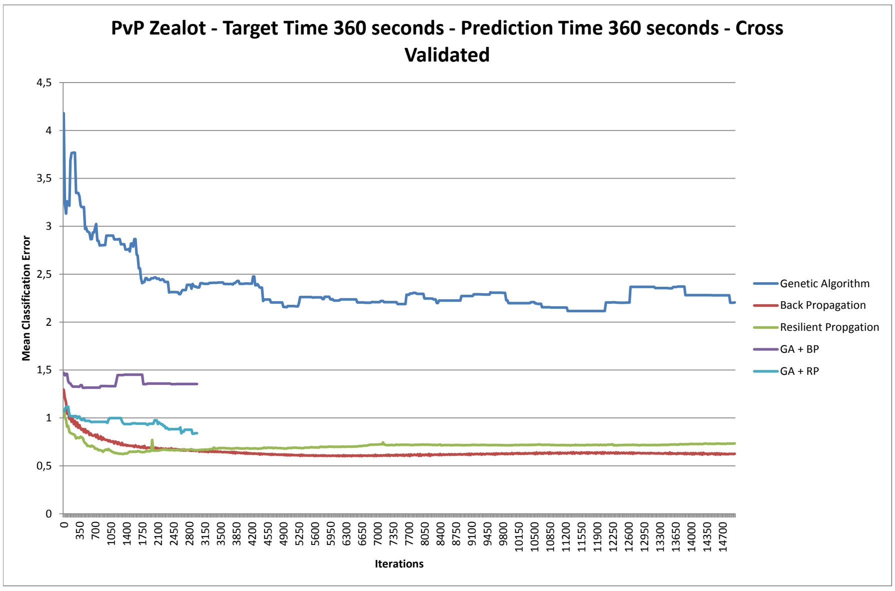

PvP Probe - Target Time 360 seconds - Prediction Time 240 seconds - Cross Validated
Mean Classification Error
25
20
15
10
5
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP


PvP Probe - Target Time 360 seconds - Prediction Time 300 seconds - Cross Validated
Mean Classification Error
25
20
15
10
5
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

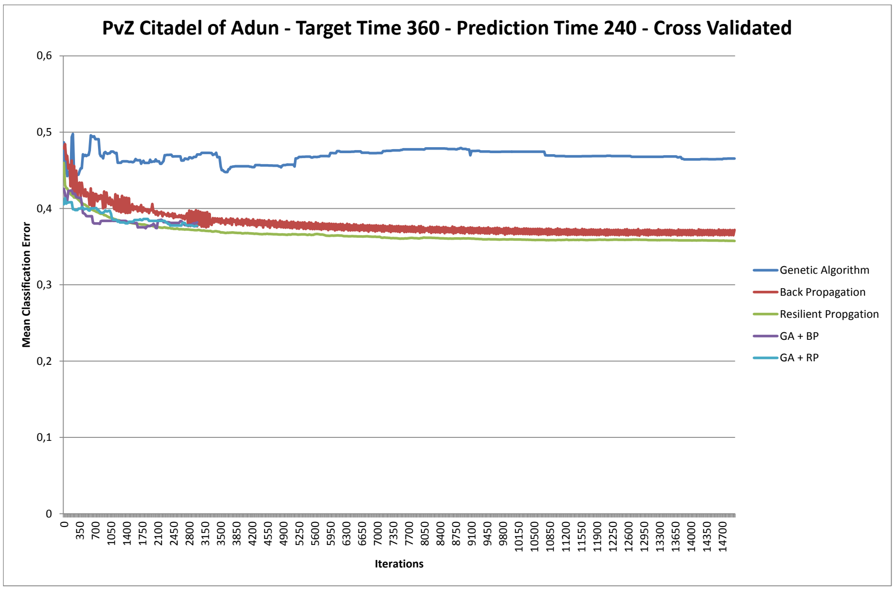

PvP Probe - Target Time 360 seconds - Prediction Time 360 seconds - Cross Validated

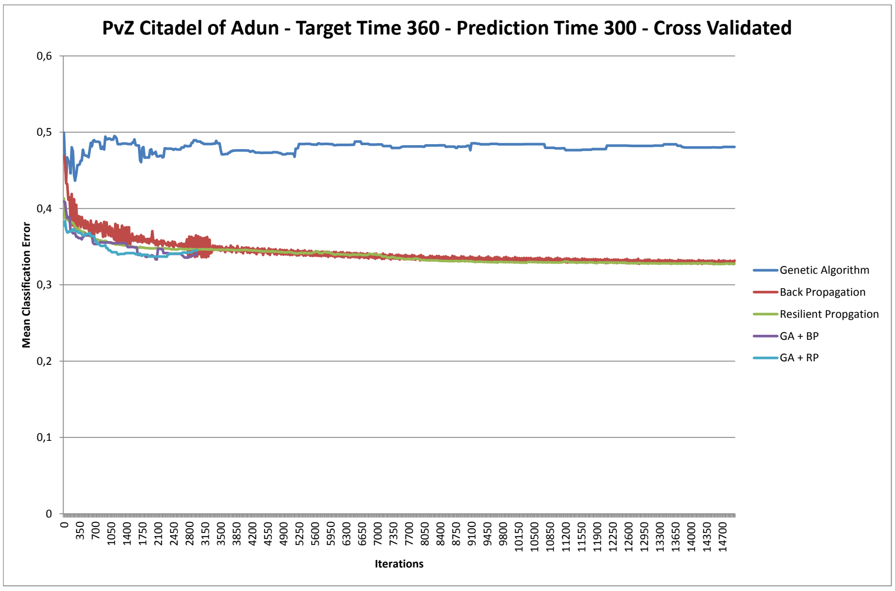

PvP Zealot - Target Time 360 seconds - Prediction Time 240 seconds - Cross Validated

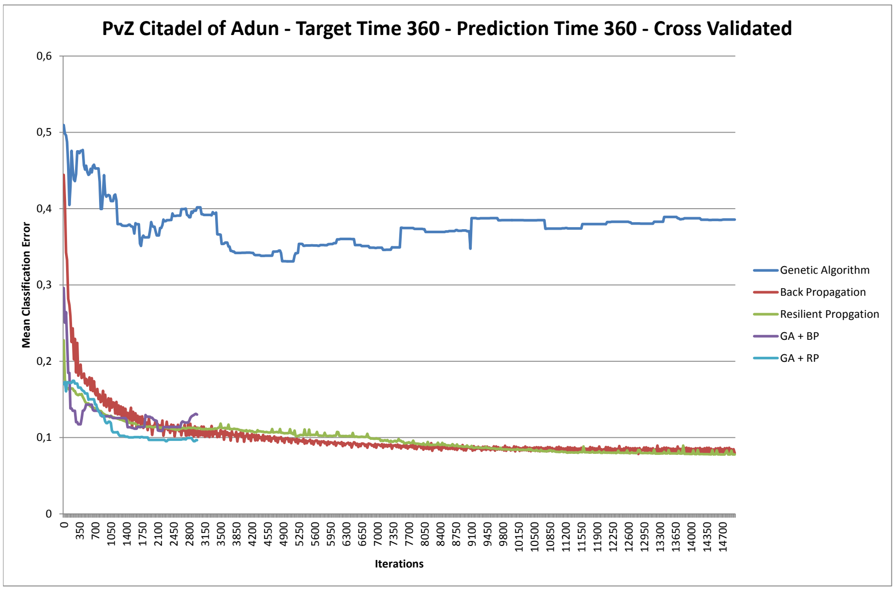

PvP Zealot - Target Time 360 seconds - Prediction Time 300 seconds - Cross Validated
Mean Classification Error
4
3,5
3
2,5
2
1,5
1
0,5
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

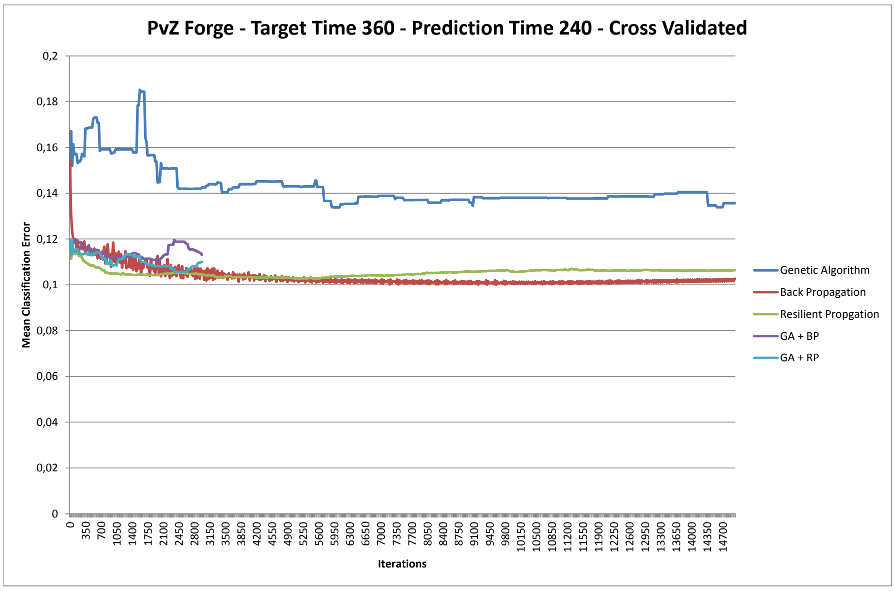

PvP Zealot - Target Time 360 seconds - Prediction Time 360 seconds - Cross Validated
Mean Classification Error
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

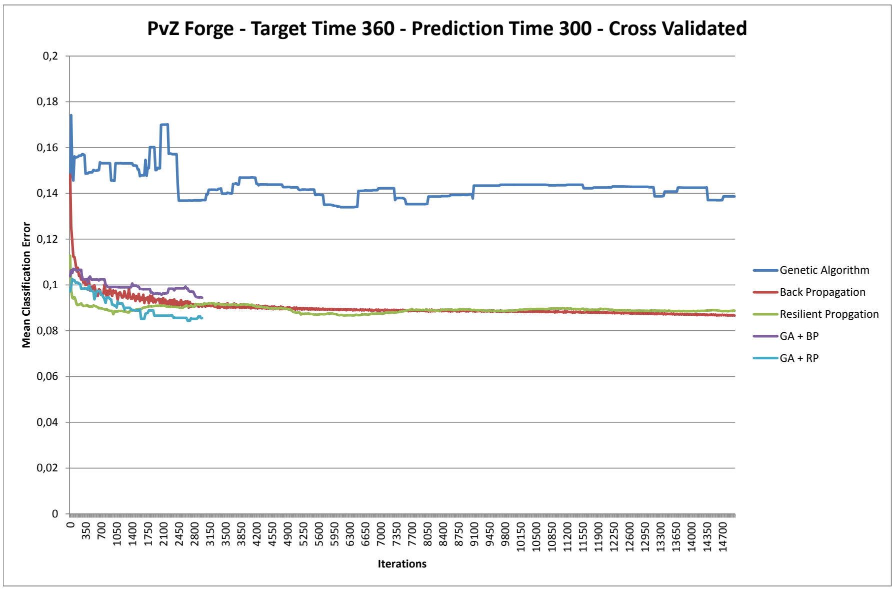
A.4.2 Protoss vs Zerg

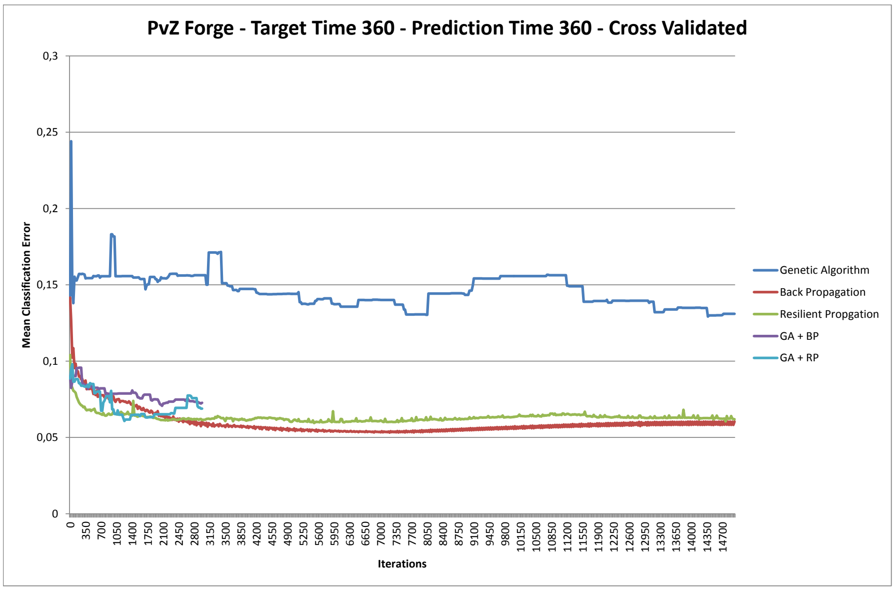


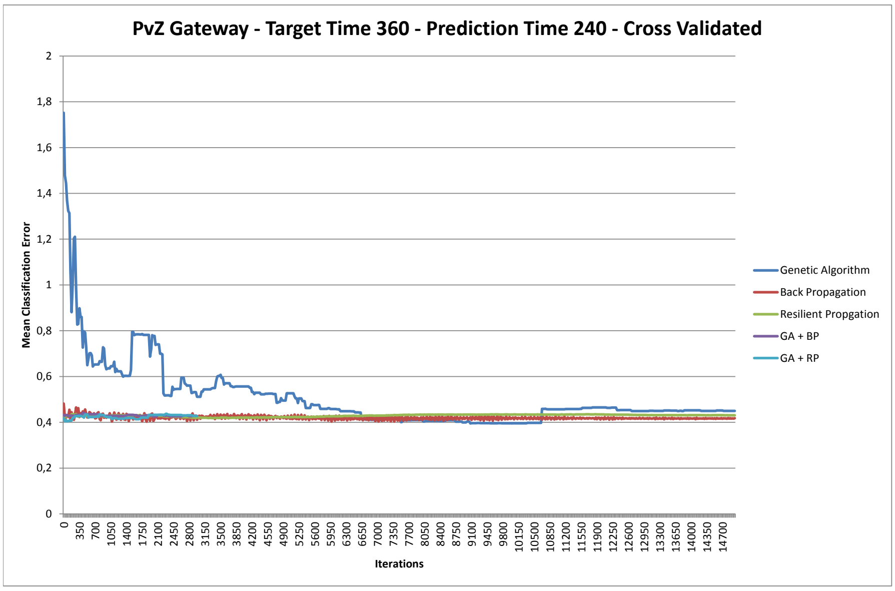


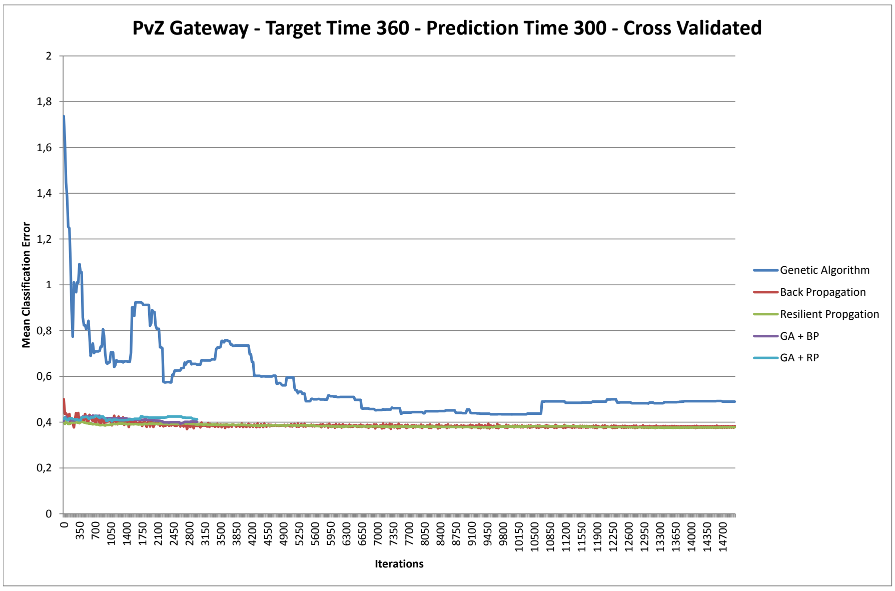

PvZ Citadel of Adun - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
0,6
0,5
0,4
0,3
0,2
0,1
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

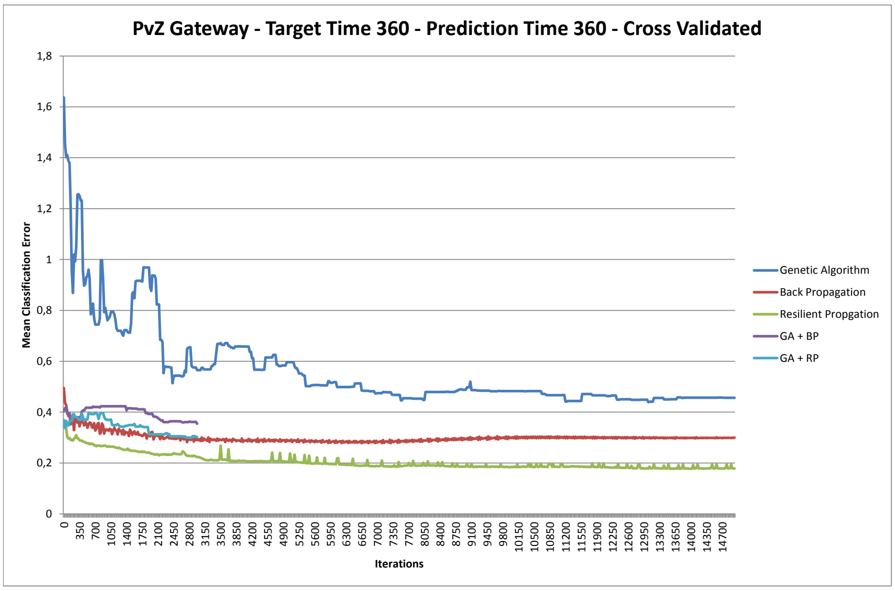

PvZ Forge - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
0,2
0,18
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

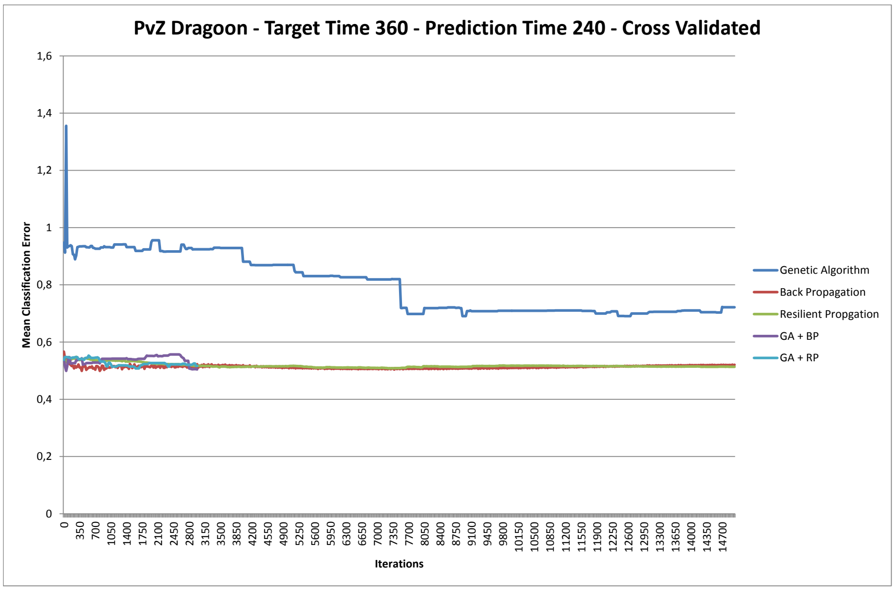

PvZ Forge - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
0
0,02
0,04
0,06
0,08
0,1
0,12
0,14
0,16
0,18
0,2
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

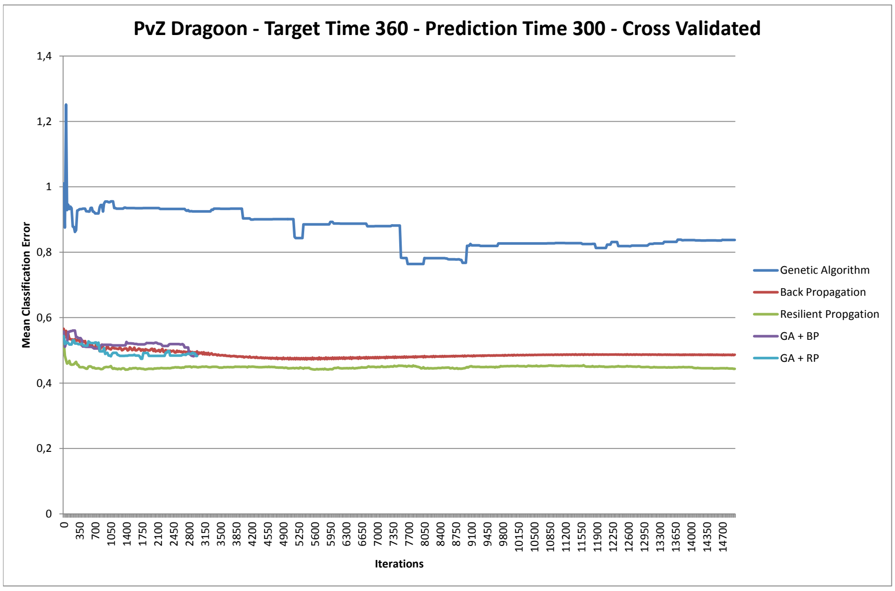

PvZ Forge - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
0
0,05
0,1
0,15
0,2
0,25
0,3
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

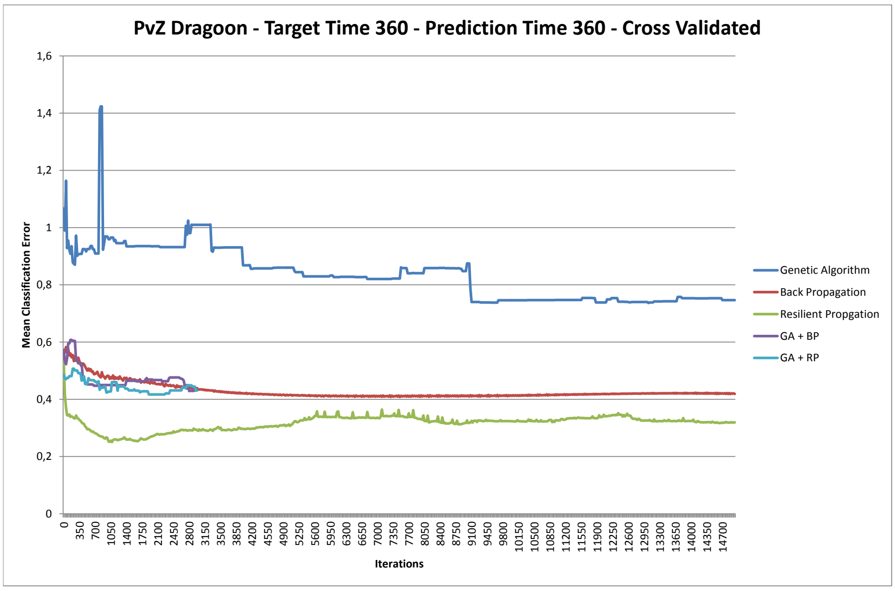

PvZ Gateway - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
0
0,2
0,4
0,6
0,8
1
1,2
1,4
1,6
1,8
2
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

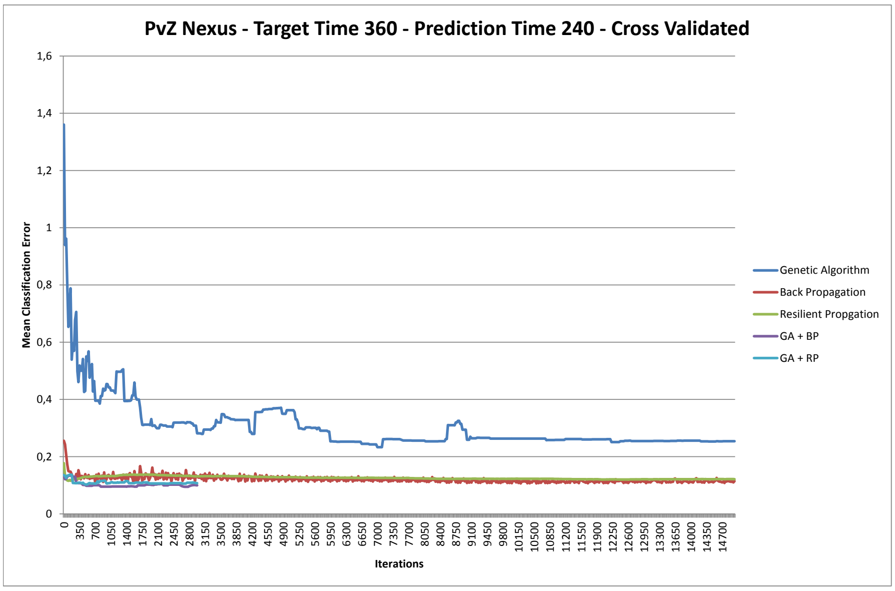

PvZ Gateway - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
0
0,2
0,4
0,6
0,8
1
1,2
1,4
1,6
1,8
2
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

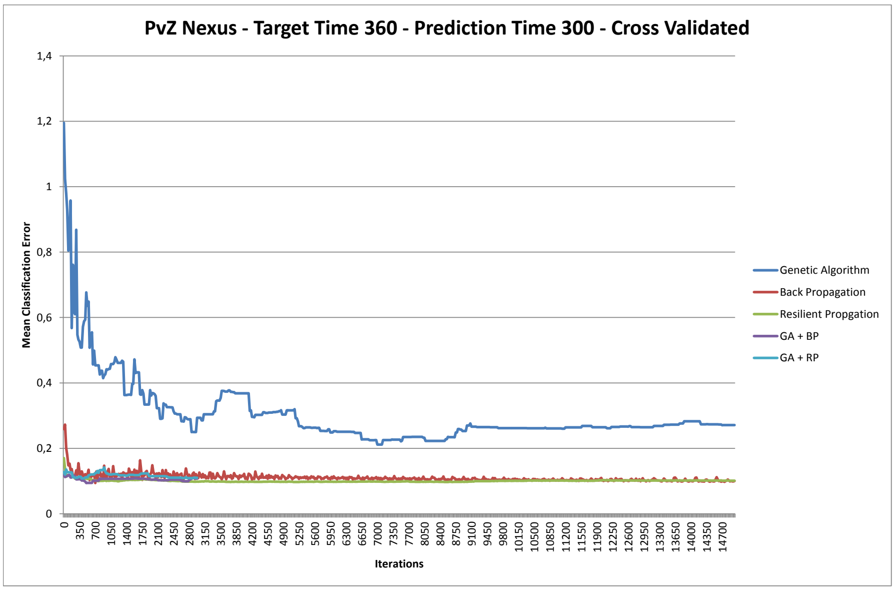

PvZ Gateway - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

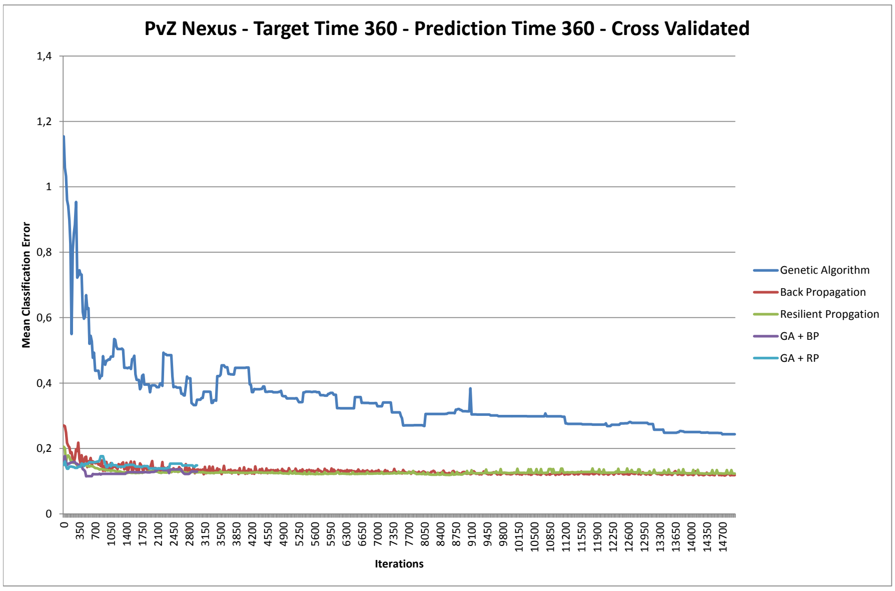

PvZ Dragoon - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

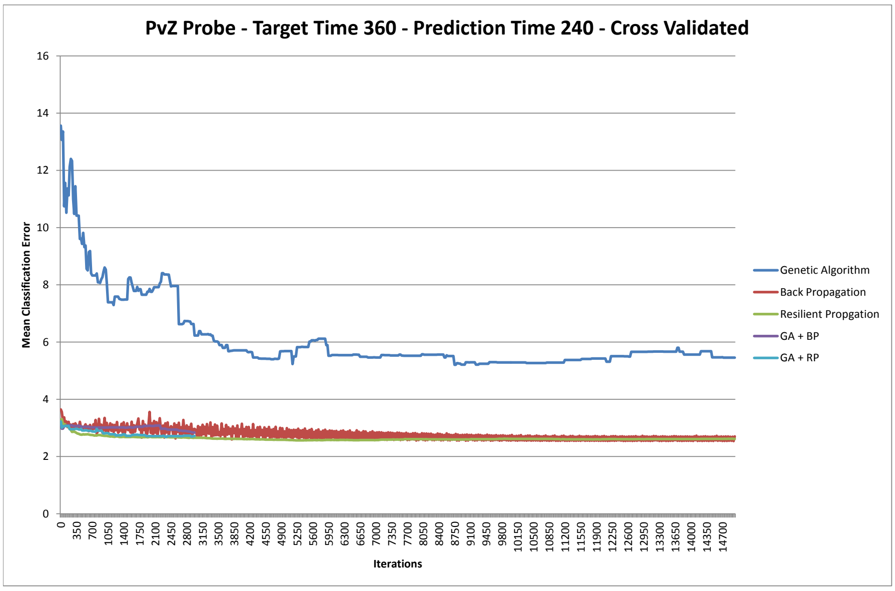

PvZ Dragoon - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Iterations
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

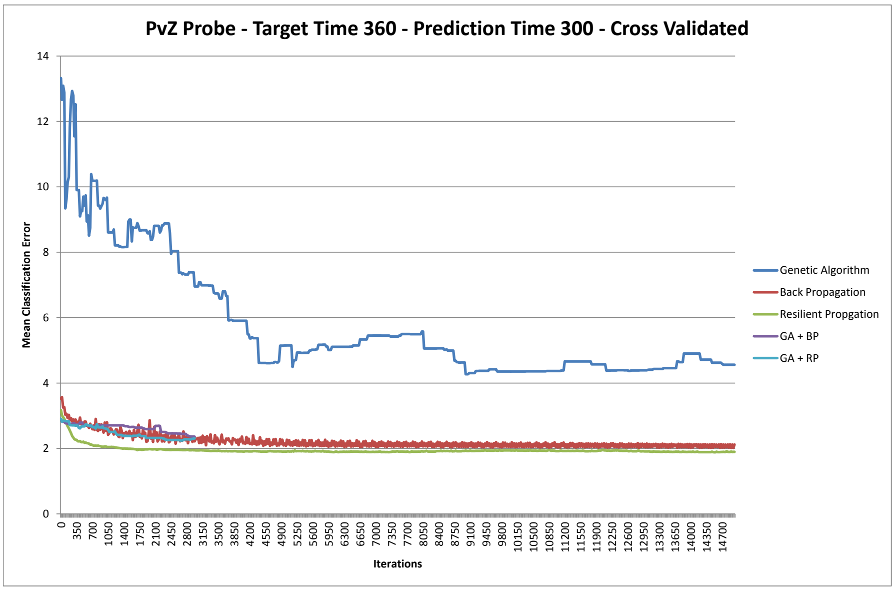

PvZ Dragoon - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Iterations
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

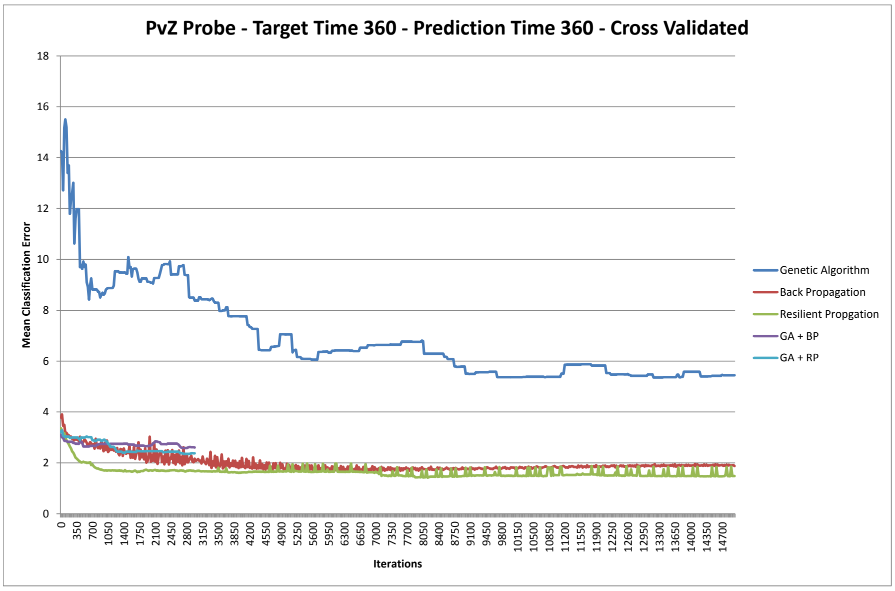

PvZ Nexus - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

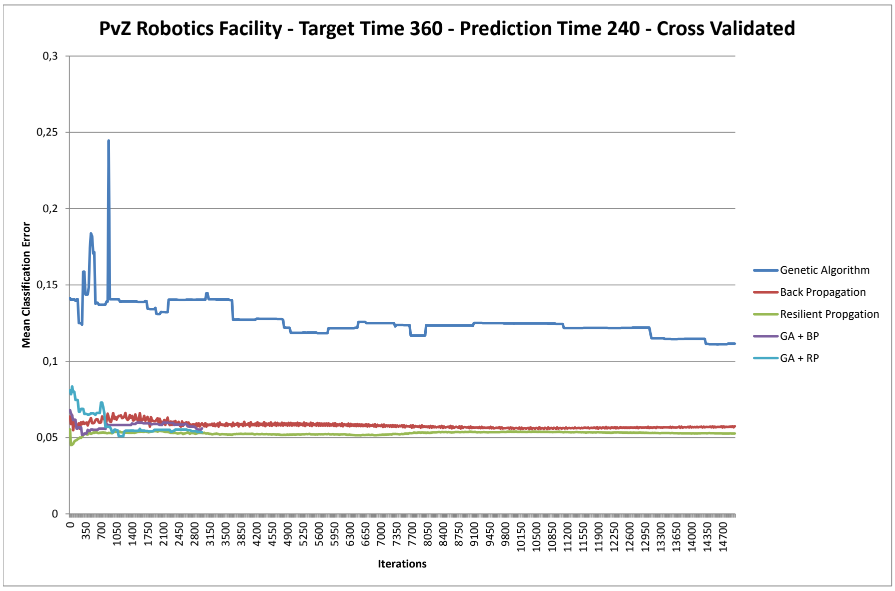

PvZ Nexus - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

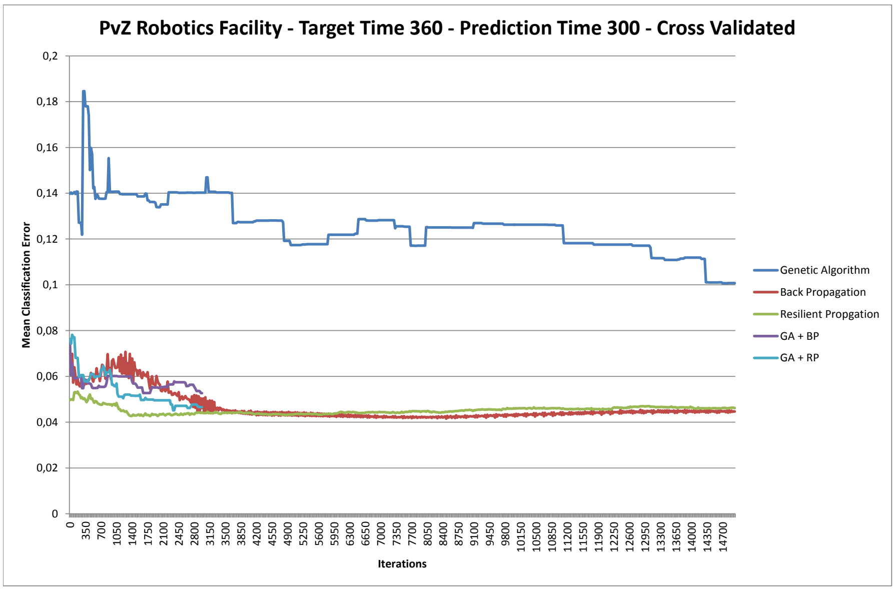

PvZ Nexus - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

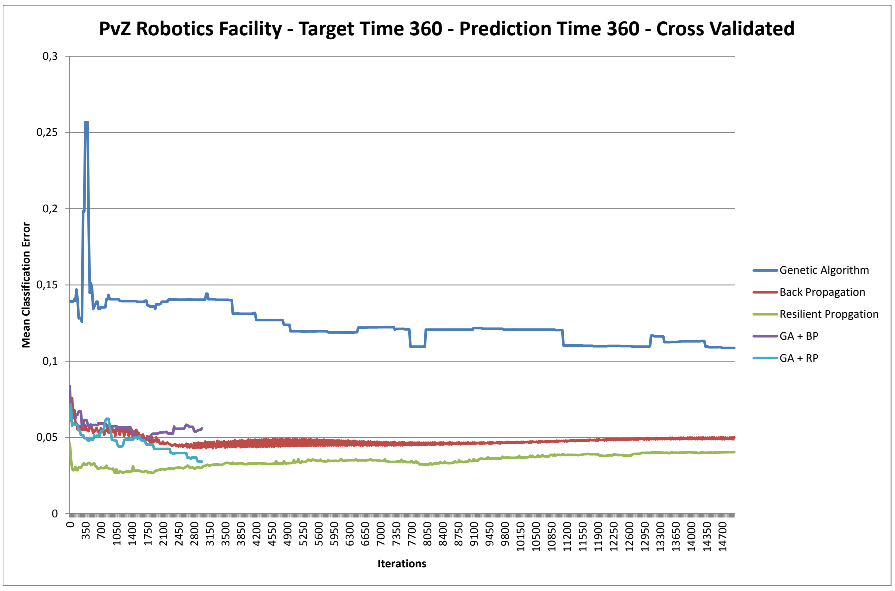

PvZ Probe - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
16
14
12
10
8
6
4
2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

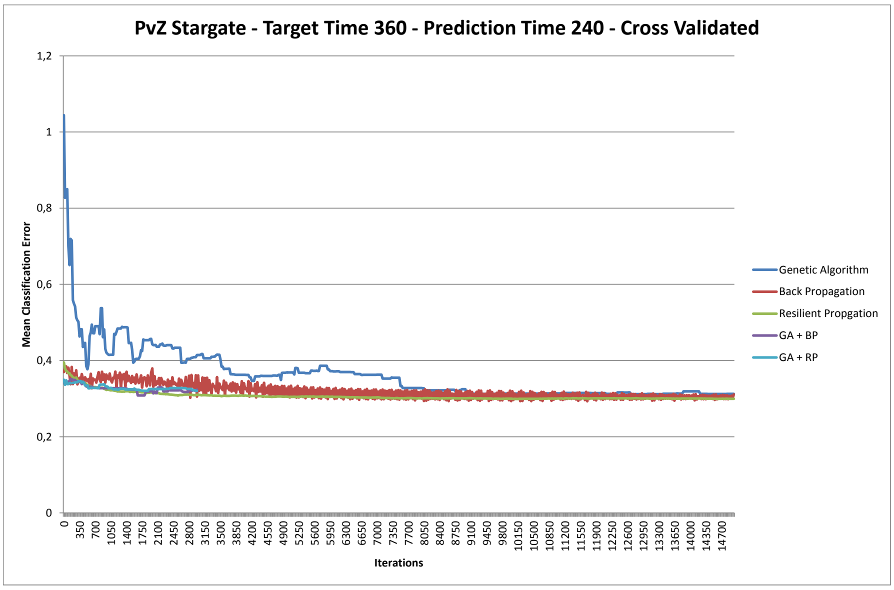

PvZ Probe - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
Iterations
0
2
4
6
8
10
12
14
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

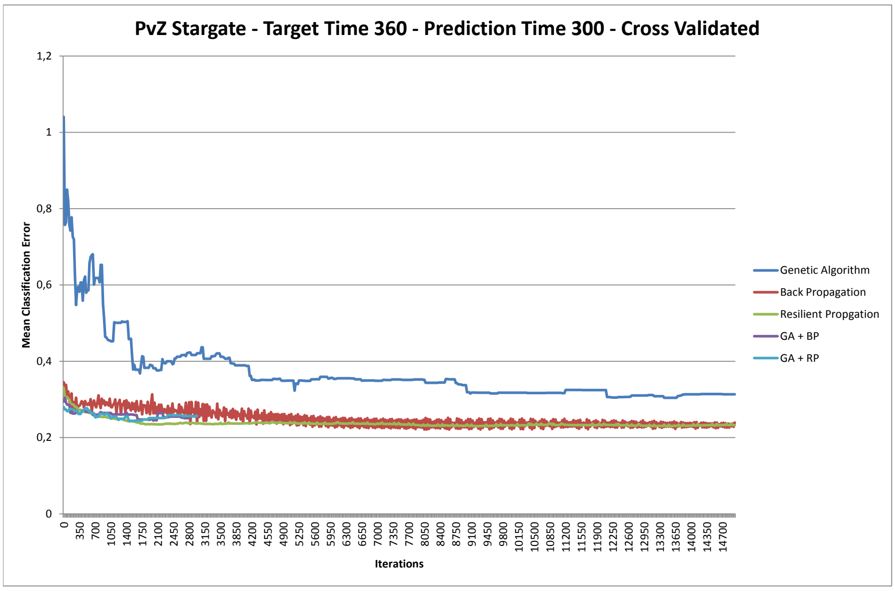

PvZ Probe - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
Iterations
0
2
4
6
8
10
12
14
16
18
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

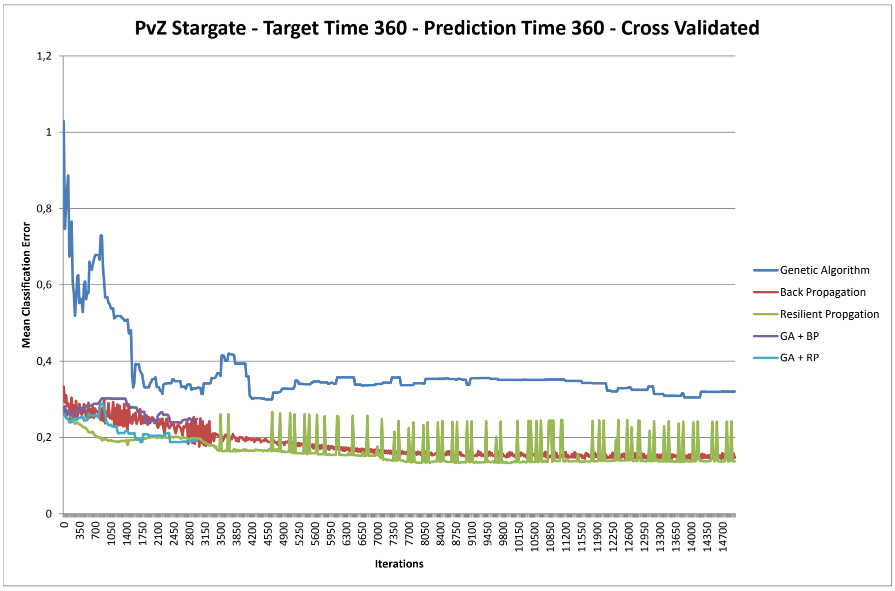

PvZ Robotics Facility - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
0,3
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

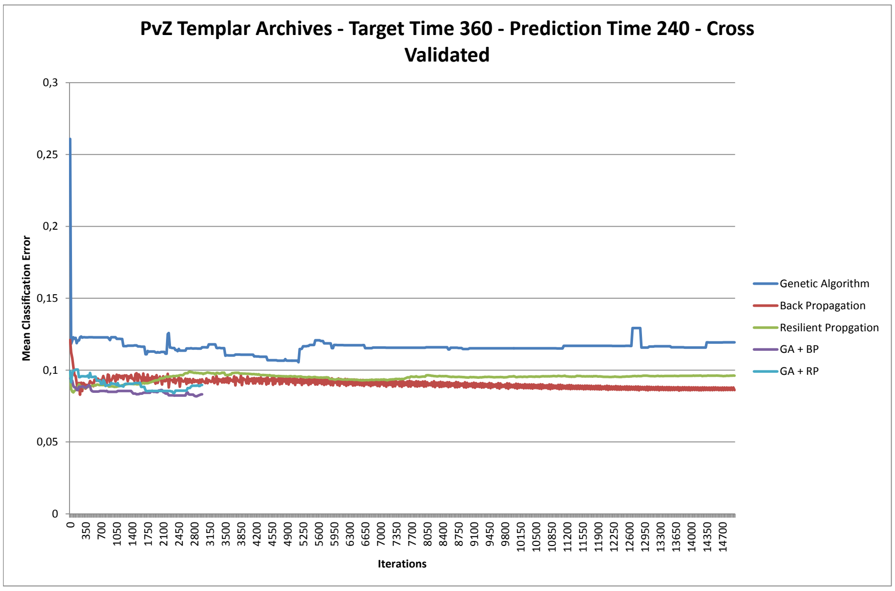

PvZ Robotics Facility - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
0,2
0,18
0,16
0,14
0,12
0,1
0,08
0,06
0,04
0,02
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

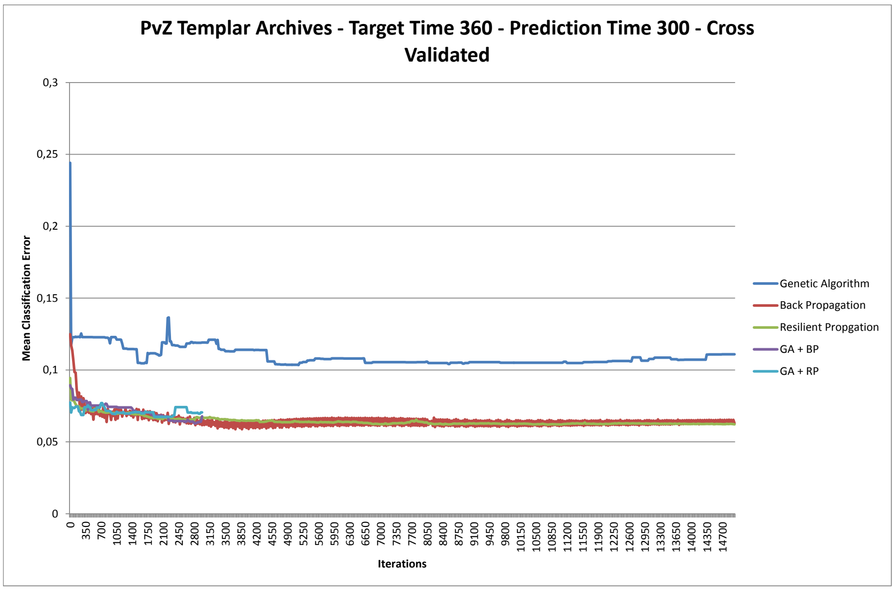

PvZ Robotics Facility - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
0,3
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

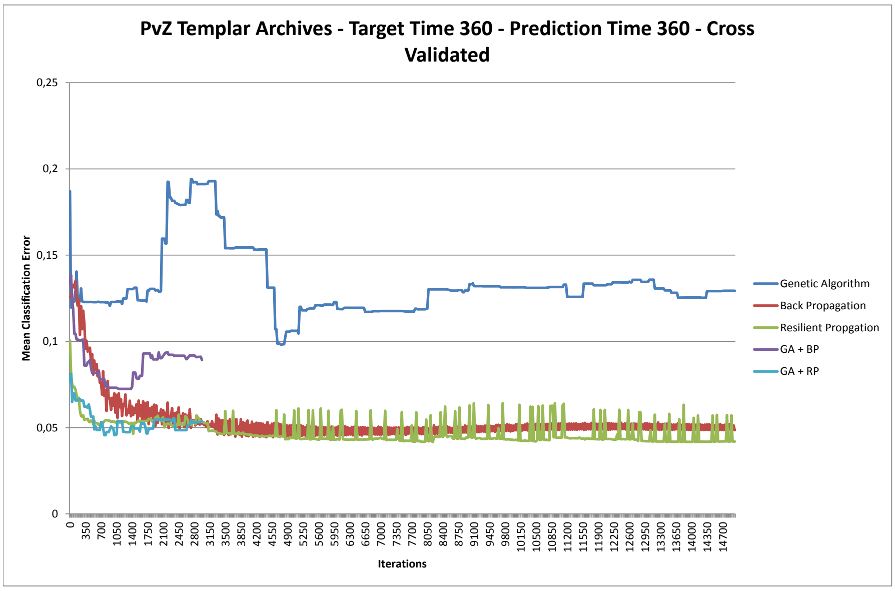

PvZ Stargate - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

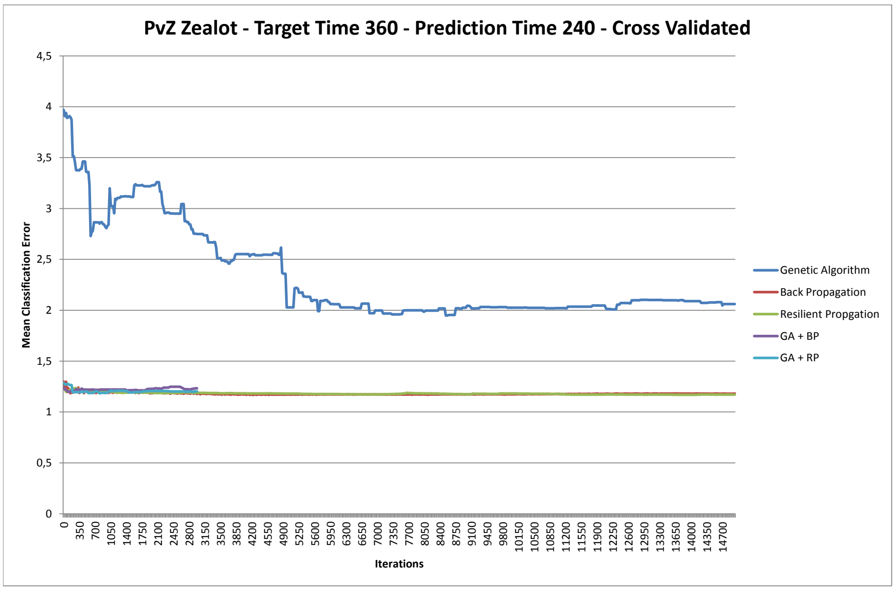

PvZ Stargate - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

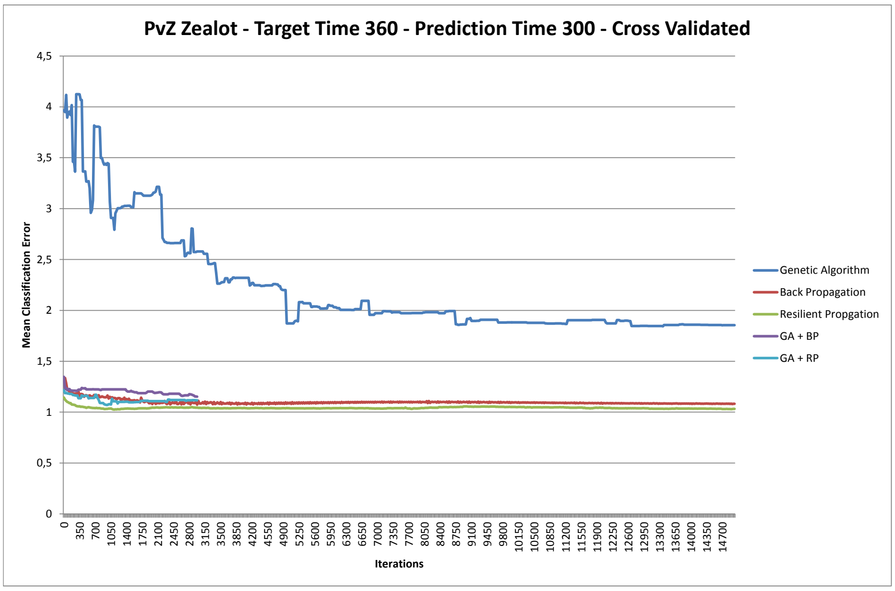

PvZ Stargate - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
1,2
1
0,8
0,6
0,4
0,2
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

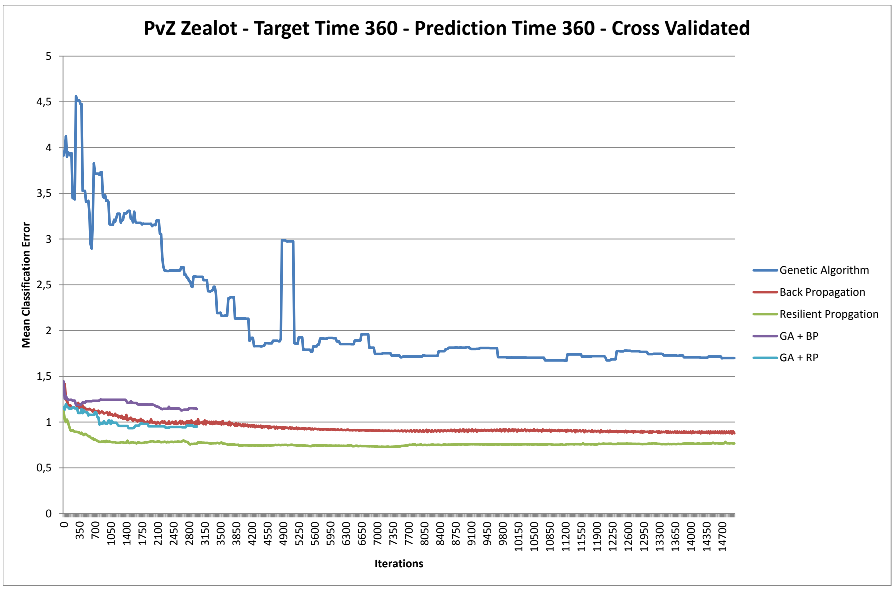

PvZ Templar Archives - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
0,3
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP

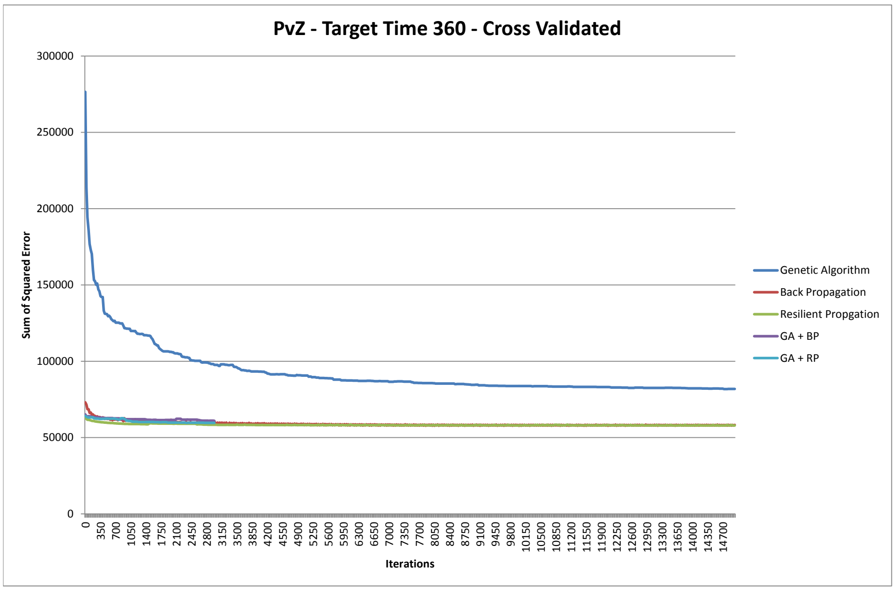

PvZ Templar Archives - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
0,3
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP


PvZ Templar Archives - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
0,25
0,2
0,15
0,1
0,05
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP


PvZ Zealot - Target Time 360 - Prediction Time 240 - Cross Validated
Mean Classification Error
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP


PvZ Zealot - Target Time 360 - Prediction Time 300 - Cross Validated
Mean Classification Error
0
0,5
1
1,5
2
2,5
3
3,5
4
4,5
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP


PvZ Zealot - Target Time 360 - Prediction Time 360 - Cross Validated
Mean Classification Error
0
0,5
1
1,5
2
2,5
3
3,5
4
4,5
5
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP


PvZ - Target Time 360 - Cross Validated
Sum of Squared Error
300000
250000
200000
150000
100000
50000
0
Iterations
0
350
700
1050
1400
1750
2100
2450
2800
3150
3500
3850
4200
4550
4900
5250
5600
5950
6300
6650
7000
7350
7700
8050
8400
8750
9100
9450
9800
10150
10500
10850
11200
11550
11900
12250
12600
12950
13300
13650
14000
14350
14700
Genetic Algorithm
Back Propagation
Resilient Propgation
GA + BP
GA + RP