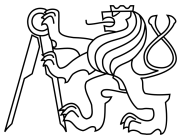
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Machine Learning in Game Playing using Visual Input
Student:	Bc. Martin Brázdil
Supervisor:	Ing. Pavel Kordík, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2016/17

Instructions

Survey machine learning and data processing techniques for game playing using raw visual input only. Implement and evaluate deep reinforcement learning or related algorithms on Pong simulator or Atari game collection. Describe your final approach in details, explain behavior of all components (preferably by visualization techniques) and report the performance of alternative methods and their settings.

References

Will be provided by the supervisor.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague October 11, 2014

Czech Technical University in Prague
Faculty of Information Technology
Department of theoretical computer science

Master's thesis

Machine Learning in Game Playing using Visual Input

Bc. Martin Brázdil

Supervisor: Ing. Pavel Kordík Ph.D.

10th May 2016

Acknowledgements

I would like to thank my supervisor Pavel Kordík for support even though I am very imperfect person.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 10th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Brázdil, Martin. *Machine Learning in Game Playing using Visual Input.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Hraní her z visuálního vstupu je komplexní problém, jehož úspěšné řešení vyžaduje kombinaci posilovaného učení a aproximačních metod. Tato práce detailně zkoumá obě oblasti a testuje vhodné metody na speciálně navržených minihrách. Finální funkční řešení je výsledkem inkrementálně nabalovaných dílčích výsledků.

Klíčová slova Posilované učení, neuronové sítě, evoluční programování.

Abstract

Game playing using visual input is complex problem which requires combination of reinforcement learning and function approximation for successful solution. This thesis investigates both fields and tests suitable methods on specially designed minigames. Final working solution is a result of incrementally aggregated partial solutions.

Keywords Reinforcemen learning, neural network, evolutionary programming.

Contents

List of Figures

List of Figures

Chapter 1

Introduction

Thesis is in the incremental order and each successive chapter build upon previous chapters.

First Chapter provides an introduction, some first insights and thoughts regarding problem domain and couple of loosely connected ideas in order to provoke relevant thoughts which need to be solved.

Second Chapter elaborates on metalearning, explains why it is important and interesting and how it is implemented and used in following experiments.

Third Chapter explores reinforcement learning and describes theory of learning from experience which is essential for a solution. Formalized description and learning methods for an agent acting in environment with goal to maximize reward is provided.

Fourth Chapter explores methods of function approximation using neural networks. Different architectures with varying neuron types, activation functions and connection patterns are tested. Fourth chapter starts with considered and used technologies. Then experimental framework architecture is justified from the point of view developed in third chapter.

1.1 What Is a Game

1.1.1 Possible description

Informally ordinary human-created game (such as Pong, Atari, Chess,..) is generally a structure (set of possible states) defined (generated) by game rules (possible actions in each state) in sequence of steps from predefined initial conditions and a subset of (terminal) states called (sub)goals with associated qualitative meaning (reward or punishment).

1.1.2 Complexity of state space

Game structure is completely determined by game rules. Interestingly rules (possible actions) are typically defined locally based on subset of current state or states visited earlier in sequence of steps taken. For example there is no separate definition of a chess piece moves specified for all different positions but only one local rule with boundary conditions. Although alternatively there is in principle a possibility to define chess piece moves for each unique state separately. The difference is that in first local approach all moves of given piece in all positions are encoded by a lossless compression compared to second approach. But there is a drawback associated with decompression (i.e. checking boundary conditions) which introduces sort of memory lookup where memory is a subset of current state.

Nevertheless such simple rules can be Turing complete (capable of any computation) as shown for example by Stephen Wolfram's automata which mean that tremendous state spaces could be described effectively by just a few rules and initial conditions. Such compressed representation of state space is called a game model.

1.1.3 Can it be learned?

Intuition of game model generating state space offers reversed possibility. Is it possible to learn game model just from information contained in a sequences of actions and resulting states? This assumption is essential for any learning and seems natural for humans. Infants are born with a model which prior information is insufficient even for their own survival. However they learn and build their internal model by exploration, i.e. waving limbs, tasting objects around.

One of properties which allow learning is assumption of deterministic model. It means if perfect observation of the whole state is made then resulting action is inevitable (defined by relations in such a model). Indeed this seems to be the case with nature - more and more deliberate observation seems to provide more and more definite answers with less and less uncertainty. [1]

1.1.4 Does free will exist?

All the states seemed to be determined at the moment when the rules are created. However which state sequences are actually going to be visited depends on strategy (which actions are chosen) of each involved player. What is the role of free will?

For example argument based on correlation of decision and previous neural activity is used as argument against free will. However alternative view could be that neural correlates do not determine our actions but instead restrict the range of actions that are available to freely choose from. From such perspective

no matter how much we correlate neural activities with our actions there will always be enough for free will to escape. To escape into a world of correlates.

Chapter 2

Metalearning of Hyperparameters

Metalearning is widely used cross-disciplinary term used not only in machine learning but also in neuroscience or psychology. Generally it is conscious awareness and control of learning process itself - in other words it is about learning how to learn. If the usual behavior patterns do not generate satisfactory result then metalearning allows self-reflection and offer a change of changing behavioral patterns which were fixed during previous trials. Often the behavioral patterns of interest are not directly associated with solution itself which means that they are usually sufficient to achieve results.

2.1 Metalearning in General

As soon as we consider that some pattern is usually sufficient we invoke some form of generalisation property which defines a class of problems. Unfortunately true generalisations exist only in conceptual world. At least some level of ignorance is required to fit real world into concepts which means that source of novel behavior patterns comes from world itself. It is true however that nature around is not perfect white noise and some patterns repeat and prevail in nature.

Interesting question is what would it mean to solve metalearning? Would the result be a kind of behavior (or perspective) where doing is immediate and never accompanied by second thoughts or regret any more yet still always results in (subjectively) good results? The dividing line between learning and metalearning from this point of view is subjective at best bacause it depends heavily on what is considered as common knowledge. Perhaps there is always possibility for another level of metalearning (or out of a box thinking).

2.2 Metalearning in Machine Learning

In machine learning community the term metalearning is defined in somewhat more definite meaning but still remains fuzzy. Different models achieve different performance on different data. There are multiple reasons why there is not yet one 'best' model. One of the well known results is 'no free lunch' theorem by D. Wolpert and D. Macready [2]. 'No free lunch' theorem explores connections between effective optimization algorithms and problems they are solving. For any algorithm any elevated performance over one class of problems is offset by performance in another class.

Metalearning then is about matching (or modifying) existing algorithms to a existing problems. In order to do so information another so called metadata information collection and processing is necessary. Connection between algorithm characteristics and problem characteristics are learned.

One particular metalearning task which can be further categorized as meta-optimization is optimization of hyperparameters, variables which are not set by learning process, in order to achieve best performance for a given problem.

2.3 Methods and Application

In this thesis I am using mathematical models which typically have a few real valued hyperparameters. Metaoptimization is used in two ways - first to search for best performing set of hyperparameters and second to explore the space of hyperparameters. Both can be done by iterative sampling of space of hyperparameters which are evaluated by computation of performance function called fitness function.

The simplest methods are grid sampling search and random sampling search which are empirically proven to work well but have a few drawbacks. Grid or random search might do unnecessary evaluations such as taking a small steps in 'obviously' negative gradient. But it is hard to tell which evaluations are unnecessary unless we have a prior expectation about fitness function, which is seldom true. The advantage of random search is convergence to globally best solution in limit because every point is visited given infinite time.

However despite lacking prior information it is sometimes useful to pick samples according to accumulated knowledge based on previous samples. Especially when fitness function for a given problem takes a long time to compute which is true in scope of this thesis. The goal is to maintain both exhaustive exploration and intelligent search. Evolutionary programming is a standard well researched method which offers both.

2.4 Evolutionary Programming

Evolutionary programming is a generic population-based metaheuristic optimization algorithm. Globally optimal solution is not guaranteed compared to random search. It is inspired by ideas from biological evolution. Population of individuals is evolved for a number of generations determined by stopping criterion. Mechanisms which take place in each generation are reproduction, mutation, recombination and selection. Various implementations of those abstract concepts exist.

2.5 Improved Fast Evolutionary Programming

Improved fast evolutionary programming(IFEP [3]) is a method of choice in this thesis (algorithm 1). IFEP uses strategy parameter η which reflect accumulated metadata during metalearning. IFEP is a combination of FEP (fast EP) and CEP (classical EP) because it utilizes two types of mutations - Gaussian mutation and Cauchy mutation. Strategy parameter is evolved along objective parameters and used as standard deviation for mutations. The main idea behind Cauchy mutation is to introduce bigger steps than Gaussian mutation. Bigger Cauchy steps have a chance to escape from local optima and potentially stumble upon global optimum while smaller Gaussian steps are more likely to intensify local search. This effect is achieved through (un)expected length of jumps:

$$E_{Gaussian}(x) = 2\int_0^{+\infty} x\frac{1}{\sqrt{2\pi}}\exp(-\frac{x^2}{2})dx = 0.8 \tag{2.1}$$

$$E_{Cauchy}(x) = 2\int_0^{+\infty} x\frac{1}{\pi(1+x^2)}dx = +\infty \tag{2.2}$$

Algorithm 1: Improved fast evolutionary programming algorithm

input : $fitness$ - arbitrary vector-valued fitness score function
input : $(\lambda_{min}, \lambda_{max}) \in (\mathbb{R}^N, \mathbb{R}^N)$ - objective vector boundaries
input : $generations \in \mathcal{N}$ - number of generations
input : $population \in \mathcal{N}$ - number of individuals

Initialize constants such that:
$\tau \leftarrow \sqrt{2\sqrt{N}}^{-1}$;
$\tau' \leftarrow \sqrt{2N}^{-1}$;

Initialize first generation $\forall i \in population$ such that:
$\rho \leftarrow (\lambda_{max} - \lambda_{min})/5$;
$\lambda_i \leftarrow \mathcal{U}_i(\lambda_{min}, \lambda_{max})$, $\lambda_i \in \mathbb{R}^N$;
$\eta_i \leftarrow \rho \cdot \mathcal{U}_i(0,1)$, $\eta_i \in \mathbb{R}^N$;
$score_i \leftarrow fitness(\lambda_i)$, $score_i \in \mathbb{R}$;
for *epoch* $\leftarrow$ *1 to generations* **do**
  for $i \leftarrow$ *1 to population* **do**
    /* $\mathcal{N}_n$ and $\mathcal{U}_n$ indicates that new random value is drawn for each variable in vector λ_i. Both offsprings are validated against (λ_{min}, λ_{max}) and if they fail new random numbers are drawn */
    $o1 \leftarrow population + i$;
    $\lambda_{o1} \leftarrow \lambda_i + \eta_i \cdot \mathcal{N}_n(0,1)$;
    $\eta_{o1} \leftarrow \eta_i \exp(\tau\mathcal{N}(0,1) + \tau'\mathcal{N}_n(0,1))$;
    $o2 \leftarrow 2 \cdot population + i$;
    // Q is Cauchy quantile function
    $\lambda_{o2} \leftarrow \lambda_i + \eta_i \cdot Q(\mathcal{U}_n(0,1))$;
    $\eta_{o1} \leftarrow \eta_i \exp(\tau\mathcal{N}(0,1) + \tau'\mathcal{N}_n(0,1))$;
  end
  foreach $i \leftarrow$ *1 to* $3 \cdot population$ **do**
    $score_i \leftarrow fitness(\lambda_i)$;
  end
  Sort $3 \cdot population$ of $(\lambda, \eta, score)$ by $score$
end

2.5.1 IFEP Experiment

IFEP is chosen because it should provide extensive yet intelligent search. Fitness function used in this experiment should be complicated enough to test those desirable properties. Particularly it should be multimodal with multiple local and global optima.

One of suitable fitness functions commonly used as a benchmark for evolutionary algorithms is Auckley's function. Generalized form of Auckley's function [4] to multiple variables is used:

$$f_{Auckely}(\lambda_1, \ldots, \lambda_N) = -20\exp(-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}) - \exp(\frac{1}{N}\sum_{i=1}^{N}\cos(2\pi x_i)) + 20 + e \tag{2.3}$$

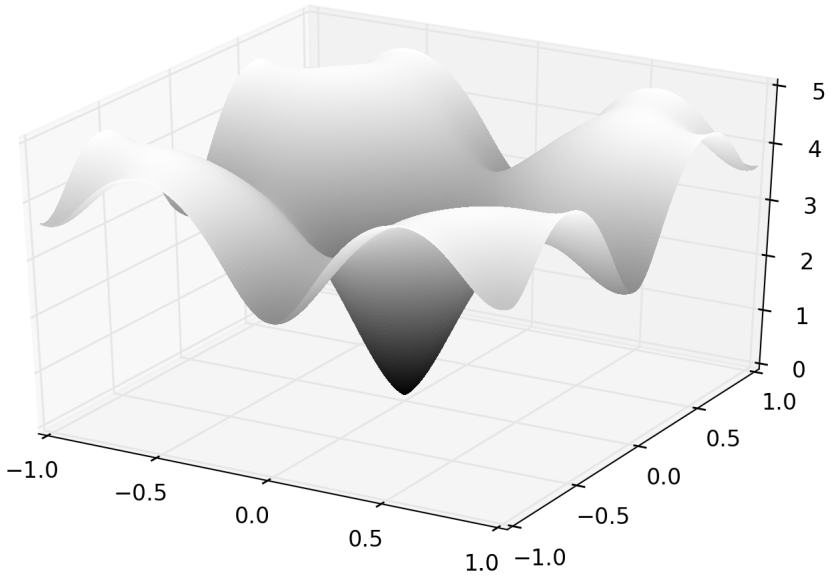


Figure 2.1: Interpolated surface of generalized Ackley's multimodal function in interval (-1, 1) for both variables produced by grid of 200x200 points (with constant step size).

Parameterization and Results

Evolution is parameterized by very small population size of 5 individuals per generation and modest number of 3 generations. Expected computation time of most complicated fitness functions in this thesis is a few minutes. Fitness function is evaluated 5 times for initial population and 10 times for each generation resulting in expected computation time of a few hours which is suitable.

From figure 2.3 it can be seen that reasonable enough exploration is maintained and multiple neighborhoods of global optima are explored. Of course the result differs each time and one example is not sufficient to draw any conclusions - purpose if purely demonstrative.

Quantitative comparison against random sampling is done by running each algorithm 10000 times. Result of this experiment is a very close match in favor of IFEP. Advantage is small but significant taking number of repetitions in

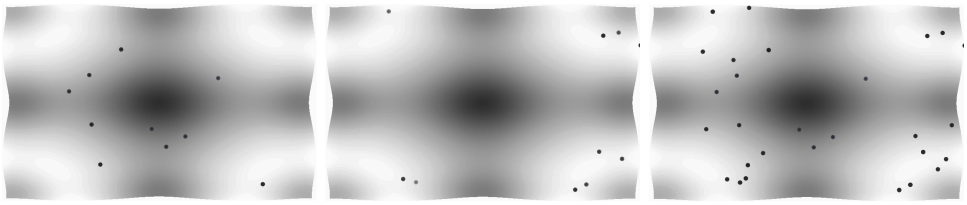


Figure 2.2: Visualisation of single run of IFEP with 5 generations and 10 individuals per generation. Left: Initial random generation. Middle: Last $5th$ generation. Right: All generations (without rejected offsprings).

consideration. Perhaps small difference in fitness can be attributed to quite a wide neighborhood around global optima. It is rather questionable if using IFEP instead of random sampling will bring any real difference. Nevertheless IFEP is a method of choice for any further metalearning tasks in this thesis.

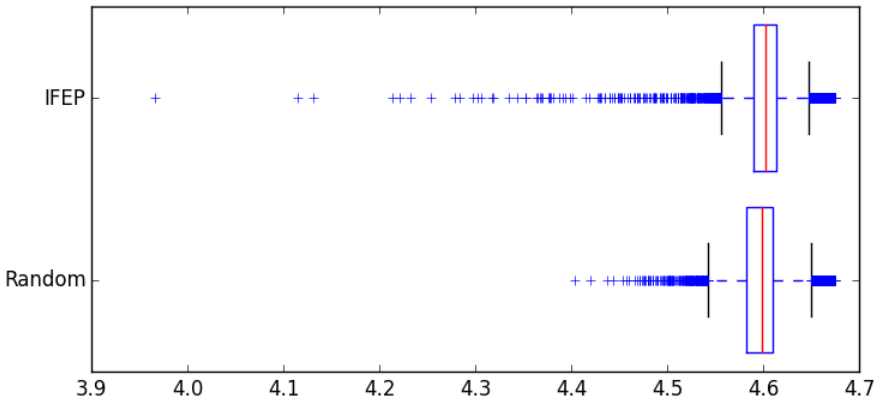


Figure 2.3: The mean difference in 10000 runs is approximately 0.003. IFEP has outliers with considerably lower fitness but it also has lower quartile closer to the mean which means that most of the points below mean are actually closer to the mean than in random sampling method.

Modification to IFEP

During exhaustive comparative experiments of IFEP with low population and few generations to grid and random search with equal number of fitness evaluations I have noticed that IFEP is the best method when boundaries of Ackley's function are larger than initially set (i.e. from (-1, 1) to (-35, 35)). The problem is in strategy parameter η which controls mutation variances because it is always initialized uniformly in interval (0, 1). Individuals from initial population are then initialized to absurdly large variances and it consumes unnecessary limited resources to adapt them.

I have made a modification which introduces constant ρ called dispersion which is computed from objective parameter boundary vector λ and then used to initialize η to a more sensitive value as follows:

$$\rho = \frac{\lambda_{max} - \lambda_{min}}{5} \tag{2.4}$$

$$\eta = \rho \cdot \mathcal{U}(0, 1) \tag{2.5}$$

Chapter 3

Reinforcement Learning

Reinforcement learning [5] is a computational approach to learning from interaction which means it does not incorporate detailed information based on how people learn but instead explores effectiveness of various models for algorithmic knowledge discovery. It applies to closed-loop systems where selected actions influence future states. Agent has no direct instructions what to do next in order to maximize reward or how particular decisions influence future actions.

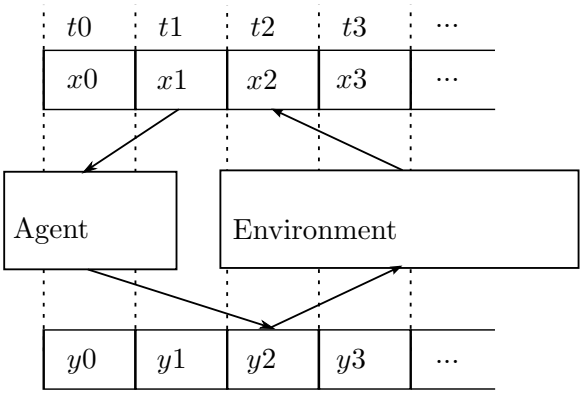


Figure 3.1: The most abstract view of reinforcement learning problem. Agent perceives state of environment x_0 then acts creating change y_2 which is received by environment which responds by closing the circle.

3.1 Comparison to Other Forms of Learning

Reinforcement learning is different from other forms of learning including supervised and unsupervised learning.

3.1.1 Supervised learning

Supervised learning focuses on learning labeled data where input signal comes in pairs - input and label. Learning from pairs is suitable for function approximation problems and it is very well established field. Supervised learning is not adequate for game playing because supervision signal (label) is usually not available. Nevertheless be utilized as approximators of various parts of reinforcement learning model. Supervised learning focuses on correct generalization (extrapolation) to inputs which were not present during training. Correct generalization is crucial since game state space is vast and only a fraction is ever experienced by agent. Exploration could potentially benefit from generalization if it is possible to discover rewarding states by extrapolation of other previously visited states.

3.1.2 Unsupervised learning

Unsupervised learning focuses on learning hidden structure in data. Traditionally unsupervised learning is understood as data clustering based on some similarity measure. Search for hidden structure is not itself adequate for learning because it's objective does not maximize total reward. Nevertheless a hidden structure found by unsupervised learning can be very useful for game model construction. Learning hidden structure first by unsupervised learning (technique called pretraining) has been shown to enhance performance (especially generalization) of supervised learning algorithms.

3.2 Curse of Dimensionality

In reinforcement learning from visual input we assume pixel representation. Pixels are obtained from visual input by discretization using two dimensional grid. The grid is two dimensional but each pixel is usually modeled as a single real dimension. Size of the grid (called resolution) defines how many pixel dimensions are used. We typically use trillions of pixels for image representation in daily life which is however totally infeasible in reinforcement learning. Even much lower number of pixels such as thousands or even hundreds is considered highly dimensional discrete space and therefore suffers from curse of dimensionality.

When we try to solve such problem we typically start developing techniques using model examples in low dimensional space which are intuitively easy to imagine. Then we scale the dimension of obtained technique to match practical problems which have typically much higher dimension.

From the technical point of view it is often easy to scale it. In most cases we plug in higher number representing number of dimensions and we are done. This is in contradiction with intuitive understanding which does not scale so

good. This can lead to underestimating of consequences if it is not taken into account.

3.2.1 Curse in Discrete Spaces

The problem is that if we discretize space it grow exponentially with number of dimensions. Exponential growth is underestimated by intuition. One example is folding a paper. It looks like an easy task if we fold it once, twice or thrice. But it is actually almost impossible to fold it 10 times. Folding it approximately 50 times will create stack that will take us to the moon. Folding it approximately 100 times would create mass of paper which will span the whole observable Universe. Each fold in this example represent one additional dimension in binary space. One of the problems of using pixels directly as state representation is that they cannot be stored in lookup tables which are a part of some reinforcement learning techniques.

3.2.2 Exploration of Cursed Discrete Space

Geometrically every point in high dimensional space with dimension N has N coordinates. Even in binary spaces number of points required in order to distribute them with small constant distance is exponential. Or from learning perspective we need to take exponential number of samples to explore such space.

3.2.3 Generalization in Cursed Discrete Space

Multiple geometrical examples exist showing how unintuitive and complicated highly dimensional spaces are. For example imagine two vectors $d = (a, a, \ldots, a) \in R^N$ and $p = (a, 0, ..., 0) \in R^N$. Then when N approaches infinity two interesting things happen. First d becomes orthogonal to p. And second d becomes infinitely larger. Both are important from learning perspective - for example when cosine similarity measure of orthogonality is affected.

$$\cos(\phi) = \frac{d^T p}{\sqrt{||d||^2 ||p||^2}} = \frac{a^2}{\sqrt{da^2 a^2}} = \frac{1}{\sqrt{d}} \to \infty \tag{3.1}$$

$$\frac{||d||^2}{||p||^2} = \frac{da^2}{a^2} = d \to \infty \tag{3.2}$$

3.3 Finite Markov Decision Process

3.3.1 Basic Notation

Finite Markov decision process is a useful method for describing very general reinforcement learning techniques. First we define *state* $S_t \in S$, where S is

the set of all possible states, *action*, $A \in A(s_t)$ where $A(s_t)$ is a set of actions available in S_t and *reward*, $R_t \in R \subset \mathbb{R}$ is a received reward from S_{t+1}.

At each time step agent queries a mapping from states to probabilities of selecting each possible action which is called agent's policy function $\pi_t(a|s)$ is a probability of selecting action $A_t = a$ when in state $S_t = s$.

3.3.2 Return Value

Reward hypothesis states: "That all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal." We formalize this statement by defining return G_t as some function of rewards following time t:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T = \sum_{t=0}^{T} R_{t+k+1}. \tag{3.3}$$

Then reward hypothesis can be though of as *return maximization*. T is a final step accompanied by S_T terminal state. If there is a final step then interaction with environment naturally breaks into subsequences which we call *episodes* which is implemented in a form of episodic learning algorithm 2 used for learning and greedy evaluation 3 used for evaluation. When there are no terminal states returns can be potentially infinite and we need *discounting* which leads to prioritization of closer rewards. Discounted return is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma R_T = \sum_{t=0}^{T} \gamma^k R_{t+k+1}, \tag{3.4}$$

where $0 \leq \gamma \leq 1$ is called *discount rate*. This formalism can be used for $T = \infty$ when $\gamma \neq 1$ or with a notion of absorbing states.

3.3.3 Markov Property

In general, causal environment can respond at time $t + 1$ by next state and reward conditioned on everything that has happened earlier. Such dynamics can be defined by complete joint probability distribution:

$$Pr\{S_{t+1} = s', R_{t+1} = r|S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}. \tag{3.5}$$

Markov property is particularly useful assumption which says that next state is conditioned only on current state:

Algorithm 2: Episodic agent learning algorithm

input: *world* ← object implementing IWorld interface
input: *agent* ← object implementing IAgent interface
input: *epizodes* ← number of epizodes to be generated
input: *limit* ← maximum number of steps before termination

```
for i ← 1 to epizodes do
    s_0 ← world.RandomState();
    for t ← 1 to limit do
        if world.Terminal(s_t) then
            break loop
        end
        p ← world.Observation(s_t);
        a ← agent.Choose(p);
        // s_t is altered to s_{t+1} by transition
        r ← world.Transition(s_t, a);
        agent.Feedback(p, a, r);
    end
    agent.LearnEpizode();
end
```

Algorithm 3: Greedy agent evaluation algorithm

input : *world* - object implementing IWorld interface
input : *agent* - object implementing IAgent interface
input : *episodes* - number of episodes to be generated, $episodes \in \mathbb{N}$
input : *limit* - maximum number of steps before termination
output: *reward*

```
for i ← 1 to episodes do
    s_0 ← world.RandomState();
    for t ← 1 to limit do
        if world.Terminal(s_t) then
            break loop
        end
        p ← world.Observation(s_t);
        a ← agent.GreedyChoose(p);
        // s_t is altered to s_{t+1} by transition
        r ← r + world.Transition(s_t, a);
    end
end
```

$$Pr(s', r|s, a) = PrS_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a. \tag{3.6}$$

If all histories have Markov property then a whole task has Markov property. It is particularly useful because this assumption enables to develop theory which can predict all future states and returns from current state based on current state only.

3.3.4 Value Functions

State value function defines value of a state s under policy π denoted $v_\pi(s)$ as expected return,

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]. \tag{3.7}$$

State value function is however not practical because algorithms based on it require perfect model of the environment because each possible next state needs to be evaluated. However if we have perfect model of the environment we can than compute state value function by value iteration algorithm 8.

Algorithm 4: State value iteration algorithm

input: $\gamma \in \mathbb{R} \leftarrow$ discount factor, $0 < \gamma \leq 1$
input: $c \in \mathbb{R} \leftarrow$ constant, $0 < c << 1$
input: $limit \in \mathbb{N} \leftarrow$ number of iterations before termination

Initialize V such that:
$\forall s \in S, V(s) = 0$

repeat
  $\Delta \leftarrow 0$;
  $limit \leftarrow limit - 1$;
  foreach $s \in S$ **do**
    $v \leftarrow V(s)$;
    $V(s) \leftarrow max_a \sum_{s',r} Pr(s', r|s, a)[r + \gamma V(s')]$;
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$;
  end
until $\Delta < c \wedge 0 < limit$;

Output a deterministic policy π, such that:
$\pi(s) = argmax_a \sum_{s',r} Pr(s', r|s, a)[r + \gamma V(s')]$

Similarly defined but much more practical is *state action value function* of a state s and action a under a policy $q_\pi(s, a)$ as expected return starting from s, taking action a and then following policy π:

$$\mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]. \tag{3.8}$$

If separate averages are kept for each action taken in each state then these averages converge to action values $q_\pi(s, a)$. Approximation can be obtained by *Monte Carlo methods* such as first visit Monte Carlo **??** and a variation of that algorithm called every visit Monte Carlo which computes averages of returns every time in backward order as in which s and a were visited.

Algorithm 5: First-visit Monte Carlo value estimation algorithm

```
Initialize:
π ← policy to be evaluated;
V ← an arbitrary state-value function;
Returns(s) ← an empty list, ∀s ∈ S
repeat
    Generate an episode S^E ⊆ S using π;
    foreach s ∈ S^E do
        G ← return following the first occurence of s;
        Append G to Returns(s);
        V(s) ← average(Returns(s));
    end
until terminated;
```

3.3.5 Bellman Equations

The fundamental property of state value function is this recursive relationship:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \sum_a \pi(a|s) \sum_{s',r} Pr(s', r|s, a)[r + \gamma v_\pi(s')] \tag{3.9}$$

and similarly for the state action value function:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \sum_a \pi(a|s) \sum_{s',r} Pr(s', r|s, a)[r + \gamma v_\pi(s')] \tag{3.10}$$

which are instances of the *Bellman equation*. The Bellman equation averages over possibilities weighting by probability of occurance. It states that the value of the start state must equal the (discounted) value of expected next

state plus the reward expected along the way. And it also does the same for state action pairs.

For finite Markov decision process the Bellman equation when iterated in limit converges to a unique optimal solution.

3.3.6 Q-learning

Another algorithm used for state action value function approximation is $Q-learning$. It is type of *temporal difference* method because it uses difference $max_a Q(S_{t+1}, a) - Q(S_t, A_t)$. Also it is type of bootstrapping method left term is action with maximum approximated value from after state.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (3.11)$$

Q-learning approximation method is part of deep Q-learning agent 6.

Algorithm 6: Off policy ϵ-greedy deep Q learning agent algorithm

```
Initialize:
B ← empty replay buffer list (s, a, r);
Q(s, a) ← randomly initialized neural network;
Returns(s, a) ← an empty list, ∀(s, a) ∈ S
repeat
    Generate an episode S^E ⊆ S using Q(s, a);
    foreach s ∈ S^E do
        G ← return following the first occurence of s;
        Append G to Returns(s);
        V(s) ← average(Returns(s));
    end
until terminated;
```

Chapter 4

Neural Network

Artificial neural network is a series of linear or nonlinear transformations of input vector to output vector conditioned on a vector of parameters and equipped with supervised learning procedure. Basic unit is an artificial neuron.

4.1 Neuron

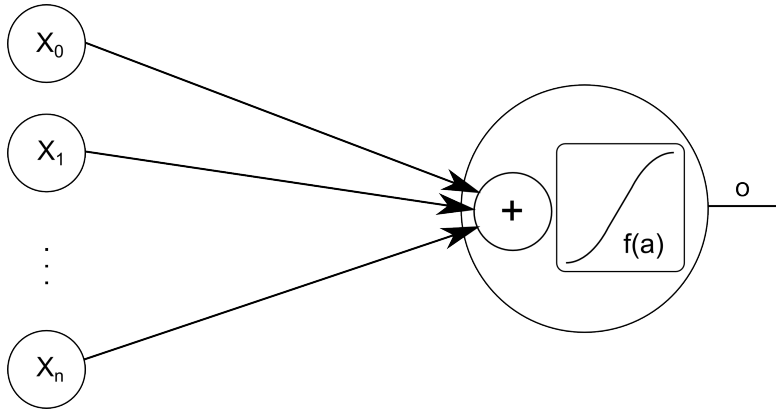


Figure 4.1: Neuron - basic unit of feedforward multilayer perceptron.

Basic unit of feedforward multilayer perceptron is a neuron 4.1. Activation is computed from any subset of input vector as follows,

$$a_j = \sum_{i=1}^{N} w_{ji} x_i + w_{j0}, \tag{4.1}$$

where $w_j i$ are adjustable parameters and w_{j0} is bias. Then 'squashing' function is computed taking activation as parameter producing an output o,

$$o = f(a) = \sigma(a). \tag{4.2}$$

4.2 Multiple Layers

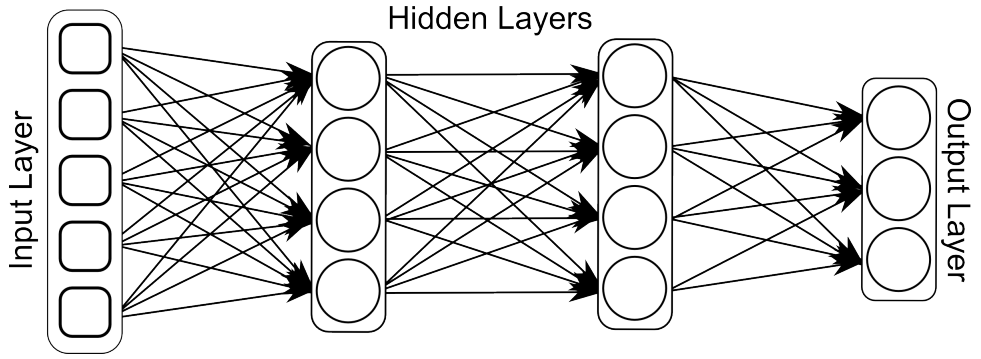


Figure 4.2: Three layered neural network. First input layer is not composed of neurons. Arrows symbolize feedforward direction (errors propagate back in exactly oposite way).

Neurons can be generally connected in any way up to a full graph but feedforward multilayer perceptron is restricted to layers. Generalization to multiple layers is straightforward. Layer is a group of neurons which is typically (in feedforward multilayer perceptron) connected to previous (inputs to the layer or flow-in) and next layer (they are inputs to next layer) and the neurons in the same layer are not interconnected(as opposed to recurrent neural networks). Flow-in inputs are always from some layer below - in other words network forms acyclic graph.

4.3 Convolutional Layer

Convolutional network is an architecture developed specifically for visual inputs. It is also inspired by biology of primate visual cortex. Typically it is used with pooling layer for image processing but not for example by Google Deepmind paper [6] for reinforcement learning. From one point of view convolutional architecture can be viewed just bunch of perceptrons which are sharing weights connected to the input in special way 4.3. From figure it can be seen that each perceptron has local square receptive field. Implementation in this thesis offers such convolutional layer with rectangular receptive fields which are not overlapping.

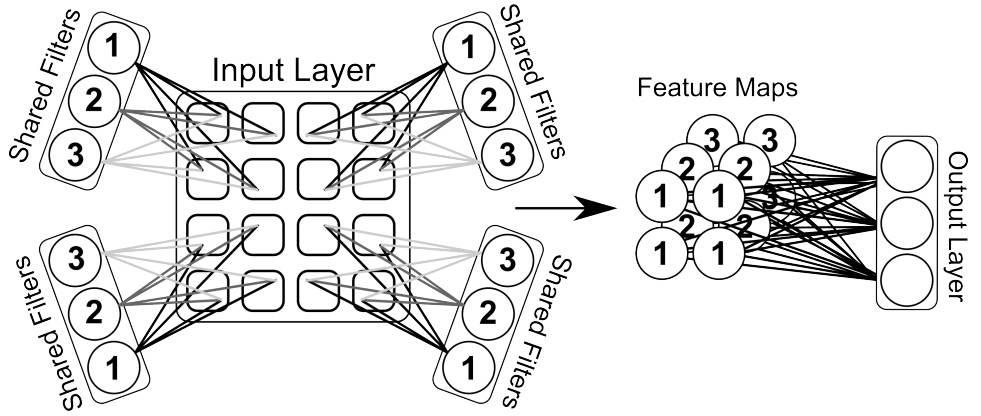


Figure 4.3: Connections represented by the same color are shared between special purpose perceptrons called filters. Filter outputs are stacked and called feature maps.

4.4 Identity Activation

The simplest squashing function of all is no squashing function at all - identity function. When identity function is used then even multilayer network is still linear approximator.

$$\sigma(a) = a. \tag{4.3}$$

4.5 Linear Rectifier

Another simple function similar to identity is linear rectifier. It has neat theoretical properties in solutions considering infinite error propagation with unitary matrices and other beyond the scope of this thesis.

$$\sigma(a) = max(0, a). \tag{4.4}$$

4.6 Sigmoid Activation

Sigmoid function is selected according to assumed probability distribution. For multiple binary classification problems logistic sigmoid function is used,

$$\sigma(a) = \frac{1}{1 + \mathrm{e}^{-a}}. \tag{4.5}$$

4.7 Hyperbolic Tangent Activation

Hyperbolic tangent is similar to to logistic function but with half of the range negative. I am using special parameterization of Y. Lecun.

$$\sigma(a) = 1.7159 \tanh(\frac{2x}{3}). \tag{4.6}$$

4.8 Universal Approximator

Approximation properties have been widely studied by for example K. Hornik et al., [7]. Network with one hidden layer can approximate any continuous function to arbitrary accuracy given enough hidden units and progressively decreasing learning rate. This is why neural networks are called universal approximators. However one hidden layer is not practical because number of neurons needed can for some problems be exponential in number of inputs. Those problems are typically intrinsically deep and can be solved by adding more layers into network with reasonable number of neurons in each layer.

4.9 Credit Assignment

Credit assignment is a basic problem which arises in network of interconnected simple units such as artificial neural network but also forms basis in neuroscience, cognitive psychology, philosophy of mind, etc. The question is how to penalize or reward single unit's contribution if required behavior is an emergent property of a whole network where one unit's contribution is conditioned on possibly many other units' contribution.

First approaches were inspired by D. Hebb's learning rule mentioned in The organization of behavior [8, 1949] which suggests to strenghten synapses between neurons which are repeatedly firing together (more specifically one after another). Hebb's learning rule formed the basis for development and formalization resulting in first learning algorithms such as perceptron learning rule pioneered by F. Rosenblatt [9] or later Hopfield networks [10] among others. Early networks after Rosenblat's paper about perceptron [9, 1958] but prior to late 80' were directly connecting input to output layer. Perceptron learning rule is also called delta rule because the synaptic strength change is proportional to the product of difference between actual output and desired output and input pattern. Perceptron learning rule is stated as:

$$\Delta w = \alpha(d - y)x \tag{4.7}$$

$$w_{t+1} = w_t + \Delta w \tag{4.8}$$

where $w_t \in \mathbb{R}$ is weight on synapse in time $t \in \mathbb{N}$, Δw is weight change, $\alpha \in \mathbb{R}$ is learning rate, $d \in \mathbb{R}$ is desired output, $y \in \mathbb{R}$ is actual output and $x \in \mathbb{R}$ is input.

Such simple architecture were able to learn useful output patterns from input patterns. However it was shown that learning capability is insufficient for many interesting problems. Minsky and Papert in their book Perceptrons [11] showed that it is indeed impossible for input to output network to learn xor function. Based on this argument they pointed out that multilayer perceptron is necessary to solve xor problem. The problem was there had not been any suitable algorithm available which would be capable to learn multilayer network where neurons in hidden layers (those between input and output) could be conditioned on outputs of other neurons in hidden layer.

Solution to credit assignment problem was proposed by P. J. Werbos in his thesis [12]. In following years the term backpropagation or simply backprop was coined.

4.10 Network Training

First two subsection show how to derive right error terms from chosen activation functions (which are chosen by required distribution properties).

4.10.1 Gaussian Distribution

We assume Gaussian distribution where network output is a mean dependent on input,

$$p(t|x, w) = \mathcal{N}(t|y(x, w), \beta^{-1}), \tag{4.9}$$

where $\beta \sim 1/\sigma^2$ is called precision, for independent identically distributed input variable maximum likelihood has a form,

$$p(t|x, w, \beta^{-1}) = \prod_{n=1}^{N} \mathcal{N}(t_n|x_n, w, \beta), \tag{4.10}$$

and decomposing it by taking negative logarithm we get,

$$-\ln p(t|x, w) = \frac{\beta}{2} \sum_{n=1}^{N} (y(x_n, w) - t_n)^2 - \frac{N}{2} \ln \beta + \frac{N}{2} \ln(2\pi) \tag{4.11}$$

$$\tag{4.12}$$

which means that minimizing finding maximum likelihood is the same as minimizing mean squared error,

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, w) - t_n)^2, \tag{4.13}$$

$$\tag{4.14}$$

4.10.2 Bernoulli Distribution

If we assume Bernoulli distribution (i.e. logistic function) dependent on the input,

$$p(t|x, w) = y(x, w)^t(1 - y(x, w))^{1-t} \tag{4.15}$$

then for independent class labels maximum likelihood is,

$$p(t|x, w) = \prod_{k=1}^{K} y_k(x, w)_k^t(1 - y_k(x, w))^{1-t_k} \tag{4.16}$$

and decomposing it taking negative logarithm we get,

$$-\ln p(t|x, w) = -\sum_{n=1}^{N}(t_n \ln y_n + (1 - t_n)\ln(1 - y_n)) \tag{4.17}$$

which means that finding maximum likelihood is the same as minimizing cross-entropy error,

$$E(w) = -\sum_{n=1}^{N}(t_n \ln y_n + (1 - t_n)\ln(1 - y_n)) \tag{4.18}$$

4.11 Backpropagation

Backpropagation is learning technique composed of two steps. First step is error evaluation using derivatives with respect to weights. Second part is weight update. In online learning mode, which is used, updates are done after each single feedforward and backpropagation.

4.12 Stochastic Gradient Descent

Gradient descent is very simple way how to evaluate derivatives. It is taking steps in the direction of negative gradient in order to minimize given error function and is stated as,

$$w^{t+1} = w^t - \eta\nabla E_n(w^t). \tag{4.19}$$

Identity Gradient

$$\frac{\partial E}{\partial w_{i,j}} = 1 \tag{4.20}$$

ReLU Gradient

$$\frac{\partial E}{\partial w_{i,j}} = x \quad if 0 < x \tag{4.21}$$

$$\frac{\partial E}{\partial w_{i,j}} = 0.01 * x \quad otherwise \tag{4.22}$$

Logistic Gradient

$$\frac{\partial E}{\partial w_{i,j}} = x(x-1) \tag{4.23}$$

Tanh Gradient

$$\frac{\partial E}{\partial w_{i,j}} = 1.14393 \frac{2}{\exp((2x)/3) + \exp(-(2x)/3)}^2 \tag{4.24}$$

4.12.1 Weight Stabilization

The magnitude of weights is unbound in context of Hebbian like updates which can cause trouble because floating point numbers in computer are very finite. This can be easily corrected by weight normalization called Oja's rule which is performed for each neuron separately.

$$w = \frac{w}{||w||^2} \tag{4.25}$$

4.13 Experiments on MNIST

Direct combination of neural networks as approximators for value functions given pixel data can be tricky and hard to debug. This is why I present separate experiments for evaluation performance and behavior network of different neural network architectures which result from combination of described techniques.

Algorithm 7: Cross Validation Algorithm

input : $network \leftarrow$ Network object with arbitrary layers
input : $mnist \leftarrow$ MNIST object with $(digit, label)$ pairs
input : $factor \in \mathbb{N} \leftarrow$ affects train/test batch size
output: $fitness \leftarrow$ sum of max of correctly classified in each batch

Initialize such that: $fitness \leftarrow 0$;

for $i \leftarrow 0$ *to* $factor - 1$ **do**
  $batch_{test} \leftarrow$ from $i\frac{mnist.count}{factor}$ to $(i+1)\frac{mnist.count}{factor}$;
  $correct_{max} \leftarrow 0$;
  $error_{old} \leftarrow mnist$.count;
  $error_{new} \leftarrow 0$;
  $correct \leftarrow 0$;
  $wrong \leftarrow 0$;
  for *ever* **do**
    foreach $(digit, label)$ *not in* $batch_{test}$ **do**
      $network$.Learn(digit, label);
    end
    foreach $(digit, label)$ *in* $batch_{test}$ **do**
      $class \leftarrow network$.Classify($digit$, $label$);
      if $class$ *equals* $label$ **then**
        $correct \leftarrow correct + 1$;
      else
        $wrong \leftarrow wrong + 1$;
      end
    end
    $correct_{max} \leftarrow max(correct_{max}, correct)$;
    $error_{new} \leftarrow error_{old} + 0.1 \cdot (wrong - error_{old})$;
    if $error_{old} < error_{new} + 0.1$ **then**
      break;
    end
    $error_{old} \leftarrow error_{new}$;
  end
  $fitness \leftarrow fitness + correct_{max}$;
  $network$.Reset();
end

4.13.1 MNIST dataset

MNIST dataset is composed of handwritten digits with labels which can be used as a supervised learning task. It is suitable in context of approximations with pixel input because digits are represented as pixels with resolution $28x28$ resulting in 784 dimensional input vectors which is more than enough for testing purposes.

4.13.2 Experiments Design

Whole dataset has 60000 training pairs and 10000 testing pairs. But for experimental purposes with respect to computation time considerations only a subset of a whole dataset is used. Experiments have two parts first is evolution of hyperparameters which provide insights into behavior of different architectures and then evolved parameters are used to find best performance.

Metalearning of Hyperparameters

Evolution is computation time consuming and only 500 pairs are used to evaluate fitness function. Fitness function is evaluated using corss validation algorithm 7. Cross validation is parameterized with $factor = 2$.

Perceptron with four different activation functions which are identity, linear rectifier, hyperbolic tangent and logistic. The only hyperparameter is learning rate α.

Two Layer Perceptron with four different activation functions which are identity, linear rectifier, hyperbolic tangent and logistic. Both layers use same activation function. Hyperparameters include separate learning rates for each layer α_1 and α_2 both bound into interval $(0.0001, 0.1)$ and also $ratio$ parameter bound into interval $(0.01, 0.5)$ which sets the size of second layer to ratio $\cdot$ first layer size.

ConvNet is a combination of convolutional layer followed by perceptron layer. Four different described activation functions are used which are identity, linear rectifier, hyperbolic tangent and logistic. Hyperparameters include separate learning rates for each layer α_1 and α_2 both bound into interval $(0.0001, 0.1)$ and also $filters$ parameter bound into interval $(4, 96)$ which sets the number of filters used in convolutional layer.

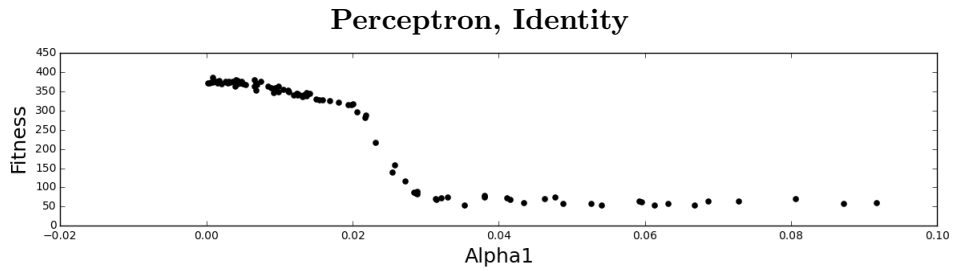


Figure 4.4: Perceptron with identity activation hyperparameter dependencies on MNIST.

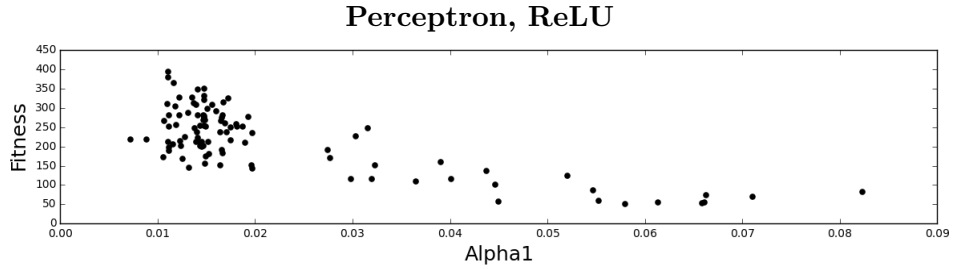


Figure 4.5: Perceptron with ReLU activation hyperparameter dependencies on MNIST.

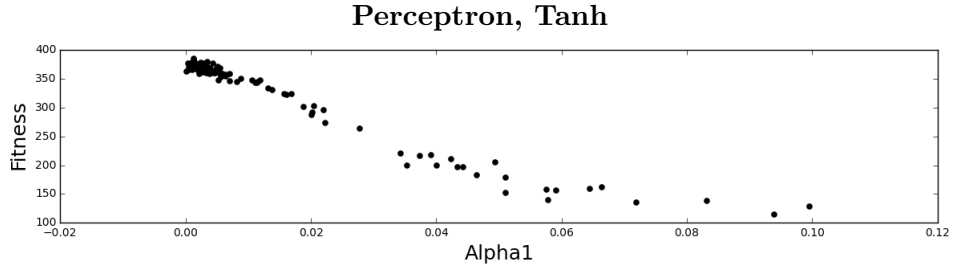


Figure 4.6: Perceptron with tanh activation hyperparameter dependencies on MNIST.

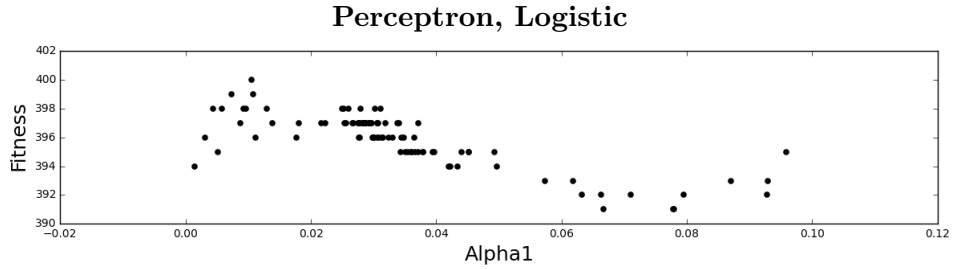


Figure 4.7: Perceptron with logistic activation hyperparameter dependencies on MNIST.

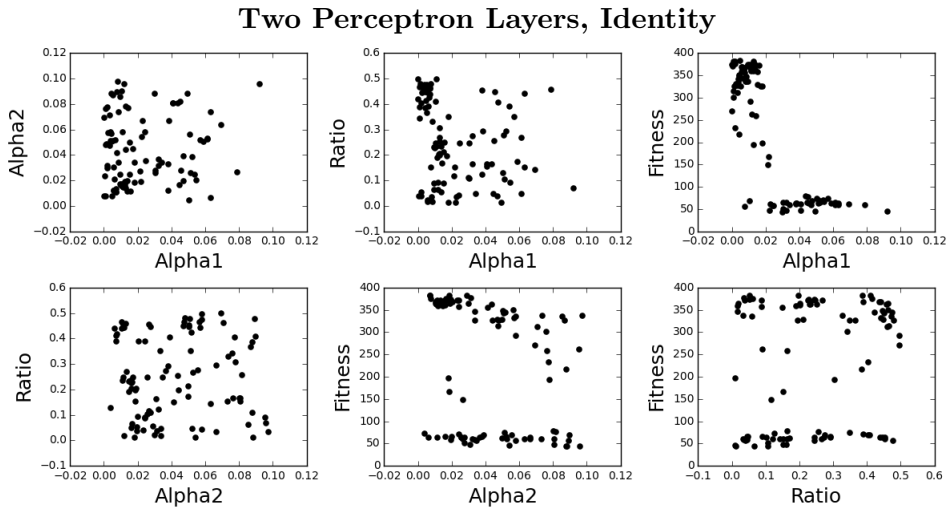


Figure 4.8: Two layered perceptron with identity activation hyperparameter dependencies.

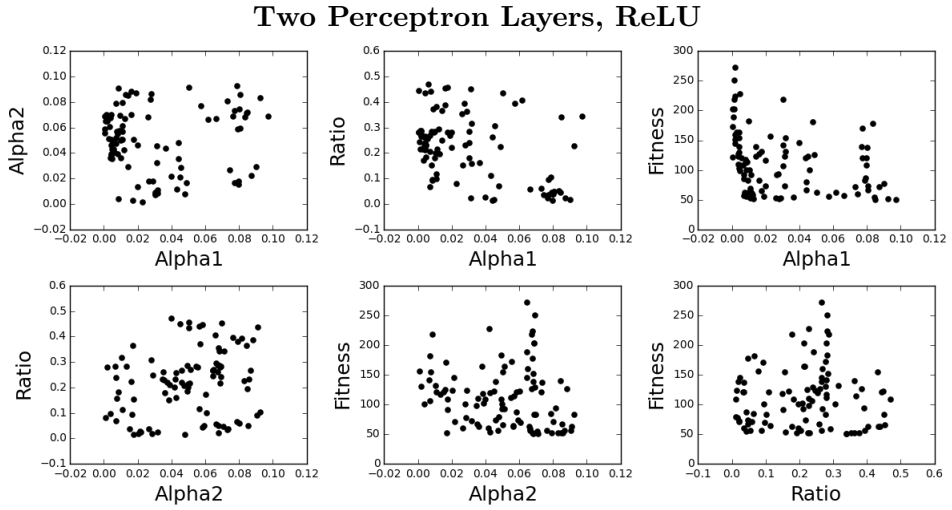


Figure 4.9: Two layered perceptron with ReLU activation hyperparameter dependencies.

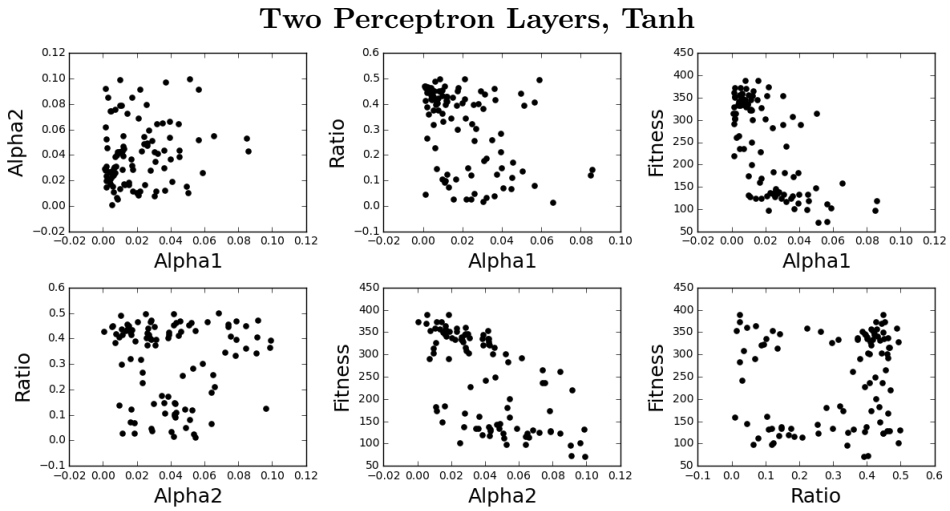


Figure 4.10: Perceptron with tanh activation hyperparameter dependencies on MNIST.

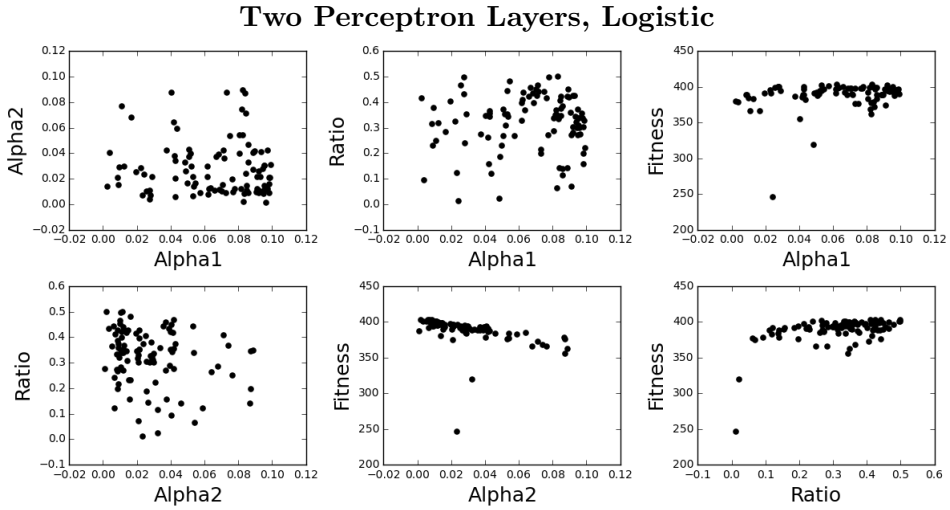


Figure 4.11: Two layered perceptron with logistic activation hyperparameter dependencies on MNIST.

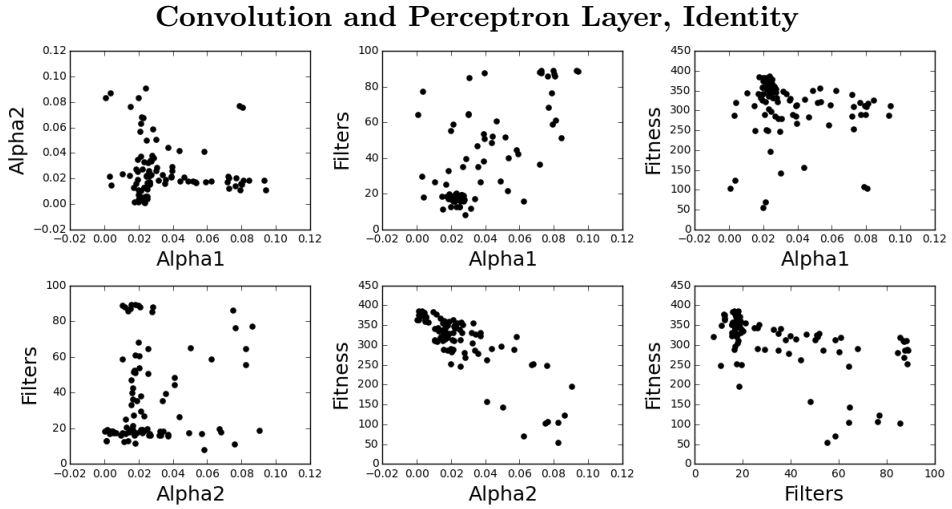


Figure 4.12: ConvNet with identity activation hyperparameter dependencies on MNIST.

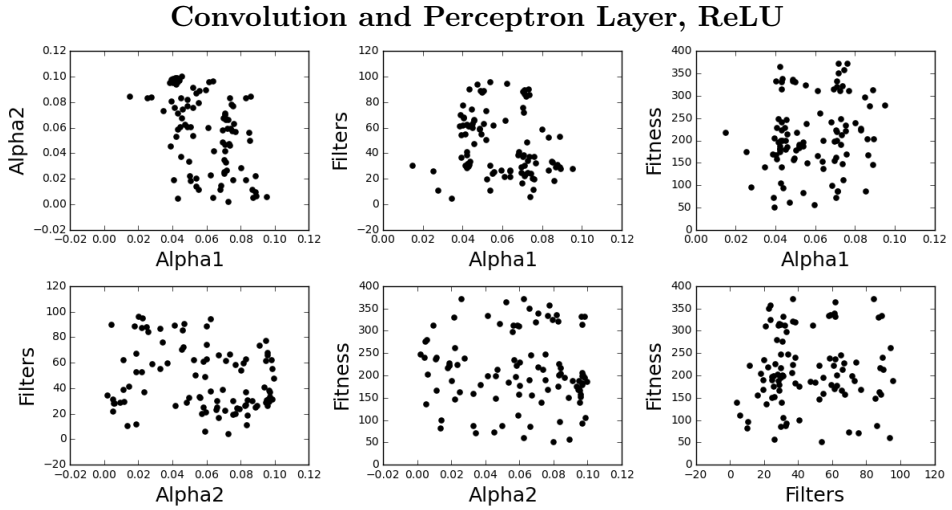


Figure 4.13: ConvNet with ReLU activation hyperparameter dependencies on MNIST.

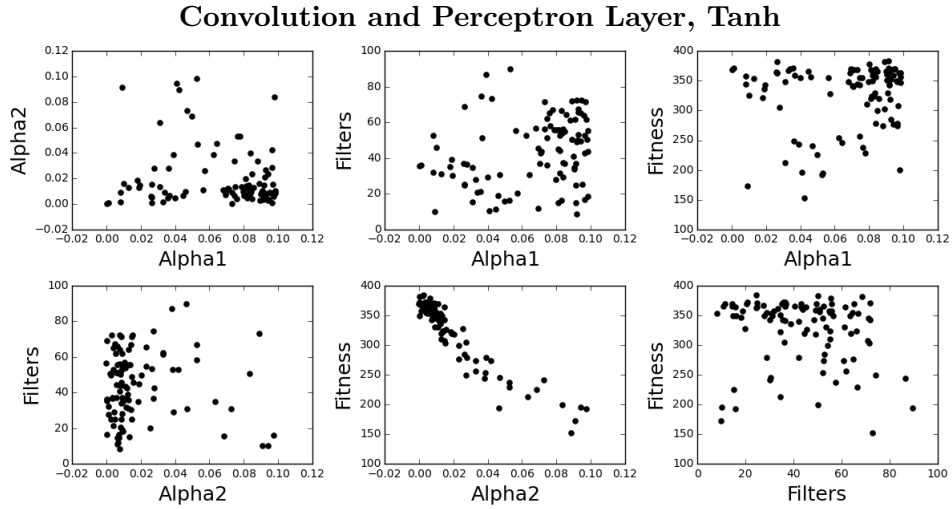


Figure 4.14: ConvNet with tanh activation hyperparameter dependencies on MNIST.

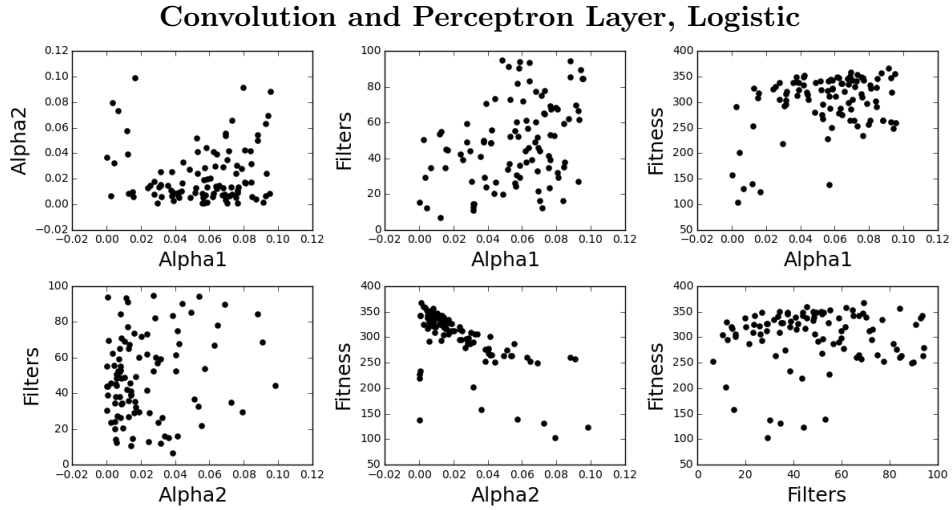


Figure 4.15: ConvNet with logistic activation hyperparameter dependencies on MNIST.

Chapter 5

Implementation

Solving non-trivial reinforcement learning problems like Pong or even more complex games requires decomposition to multiple simple parts (which are nevertheless a single tightly interconnected system). Learning from raw pixel data inherently evokes the curse of dimensionality problem which is not solvable by straightforward methods for any size of practical interest. Or artificial noise on pixel level makes problem even more non-stationary and state space so big it is impossible to solve without function approximation techniques. Of course those difficulties could be prevented by incorporation of prior domain knowledge but another requirement is that learning algorithm should be able to perform well in multiple different games with minimum (automatized) finetuning of minimum number of hyperparameters.

Because of complexity (considering all of the requirements mentioned) which leads to considerable amount of programming (which is error prone) the reasonable approach is to build and test system incrementally on problems which fulfill simplified assumptions. Simplified problems and simplified solutions then form a performance baseline testbed against which advanced techniques are compared and also serve as unit tests when generic and object oriented programming techniques are utilized in order to develop unified experimental architecture which avoids code copying. Then games with appropriate modifiable assumptions are used to experimentally verify expected performance of advanced techniques. The whole incremental approach represents continuous evolution of various techniques with a final best solution at the end.

5.1 Implementation

First thing to be chosen is a tool chain and/or framework suitable for implementation of all requirements in assignment. Requirements include: 1) ability to create or run multiple scalable and modifiable games in special ex-

perimental conditions, 2) (high performance) mathematical computations, 3) various visual and statistical graphs and plots.

5.1.0.1 Available Technologies

To this date there are many opensource or commercial solutions targeting each requirement available worldwide. Machine learning communities prefer combinations of scripting (Python, Lua...), high performance (C++...) and special purpose (Haskell-like, Wolfram...) programming languages. Among available machine learning frameworks are for example:

OpenAI non-profit artificial intelligence research company which released very recently (too late to be used) OpenAI Gym Beta standardized reinforcement learning testing environment.

TensorFlow open source software library for numerical computation using data flow graphs), Torch (a scientific computing framework with wide support for machine learning algorithms.

Torch a scientific computing framework with wide support for machine learning algorithms.

Caffe deep learning framework developed by Yangqing Jia while in the PhD program at University of California at Berkeley.

Theano a Python library that allows to efficiently define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays.

cuDNN list of primitives and standard routines useful for neural network implementation accelerated by NVIDIA GPUs.

Although using some of those very powerful frameworks would allow me to solve more complex problems faster it would also hide away too much detail. This is why I am using only C++ with standard libraries for all computation and Python with numerical and plotting library (Numpy and Matplotlib respectively) for visual presentation of results.

5.1.1 Required Technologies

Specifically I am using latest up to date C++11/14/17 standard compiler Visual-C++ provided with community edition of Visual Studio 2015 - which is needed to successfully load, compile and run project provided with this thesis.

C++11/14/17 standard is chosen for its much improved support of generic template metaprogramming which enables to achieve more while writing less without any runtime performance hit. Support of functional programming

especially lambda functions with accessible local scope (which proved very useful for implementation of fitness functions used in evolution). Whole new random library adopted (with a few other libraries) from Boost replacing an old C Rand with multiple powerful random number generators (such as Mersenne twister) and parameterizable distributions (such as normal, uniform, Cauchy, discrete to name a few used in this thesis) and many other features which increase productivity (tuples, initializer-lists...).

5.2 Architecture

During implementation of incremental approach many specific architectural requirements naturally arise. The core of incremental approach are experiments targeted on different aspects where all appropriate agents have to be tested in strictly equal both conditions and methodology. Different aspects can range from common basic principles like evaluation of states to some random particular aspects of a tiny subset of sophisticated methods. And not only single modules but all compatible combinations of modules have to be tested.

Naive approach such as copies of the same experiment manually adjusted for particular methods is infeasible. It suffers from unnecessary potentially exponential code repetition which is error prone and makes it hard to keep exactly the same conditions with different parameterizations. However there is one advantage of naive approach which is speed of single-purposed experiment which can be incorporated into incremental approach. Often it is easier to do trivial implementation for at most a few cases before abstracting to general case. Following subsections describe implementation and architectural details realized during multiple incremental steps which were done with respect to all given considerations. Each step is based on both theoretical analysis of required behavior and multiple naive implementations (a little bit of code jamming).

5.2.1 IWorld and IState

Multiple world introduced in this thesis share common abstract interface in order to achieve easy reusability by switching worlds used in experiments. All worlds are derived from IWorld which is inspired by Markov chain because it makes (forces) them to be 'memoryless'. Memorylessness here means that state of a world is strictly separated from implementation of world mechanics. Each state therefore uniquely represents single state in world state space.

However the distinction is made between the memoryless state space used as representation of true world state and perception state space as perceived by agent. They are not necessarily equal which also means that perceptions do not necessarily fulfill Markov 'memoryless' property. This also leads to implementation of modified value iteration algorithm called perception value

iteration algorithm 8. This algorithm is useful for insights about how different perception of the same environment affects convergence of value functions.

Algorithm 8: Perception value iteration algorithm

input: $world \leftarrow$ object implementing IWorld interface
input: $\gamma \in \mathbb{R} \leftarrow$ discount factor, $0 < \gamma \leq 1$
input: $c \in \mathbb{R} \leftarrow$ constant, $0 < c << 1$
input: $limit \in \mathbb{N} \leftarrow$ number of iterations before termination

Initialize S and V such that:
$S \leftarrow world$.AllStates();
$V(p) \leftarrow$ initialize to empty map, $\forall p \in P$, $V(p) = 0$;

repeat
  $\Delta \leftarrow 0$;
  $limit \leftarrow limit - 1$;
  foreach $s \in S$ **do**
    $p \leftarrow world$.Observation(s);
    $v \leftarrow V(p)$ $A \leftarrow world$.Actions(s);
    $V(p, a) \leftarrow$ initialize to empty list, $\forall a \in A, V(p, a) = 0$;
    forall $a \in A$ **do**
      $s_a \leftarrow s$;
      `// ` s_a `is altered from` s `to` s_a `by transition`
      $r \leftarrow world$.Transition(s_a, a);
      $p_a \leftarrow world$.Observation(s_a);
      $V(p, a) \leftarrow r + \gamma V(p_a)$;
    end
    $V(p) \leftarrow max_a V(p, a)$;
    $\Delta \leftarrow max(\Delta, |v - V(p)|)$;
  end
until $\Delta < c \wedge 0 < limit$;

Output a deterministic policy π, such that:
$\pi(p) = argmax_a \sum_{p',r} Pr(p', r|p, a)[r + \gamma V(p')]$

States in more complicated worlds usually contain so many variables that it is not convenient to use all of them. For example pong world state contains pong and ball positions but also discrete force vectors moving the ball and discrete time which parameterizes discrete ball movement function.

Agent generates perception from state using interface method 'Observation'. Agent also generates a set of possible actions for each state by using interface method 'Actions'. Selected action together with a state are used in interface method 'Transition' which changes given state to next state and returns immediate reward.

Clear separation of a state and mechanics has multiple advantages, e.g. it

allows to directly initialize world to any state or it allows to save and directly reuse state for the same or different agent by just taking a copy of a state. It is also possible to generate all possible states for any world using algorithm 9. Clear separation of world state from agent's perception allows to simulate state space with different properties such as partial observability which lead to violation of Markov decision process (because state is not identifiable which is one of the assumptions).

Algorithm 9: Breadth first search state space generating algorithm

input : *world* - object implementing IWorld interface
output: $S \leftarrow$ set of all generated (reachable) states

Initialize S, *queue* such that:
$S \leftarrow$ empty set;
queue $\leftarrow$ push *world*.DefaultState();

while *queue not empty* **do**
  $s \leftarrow$ *queue* front;
  foreach $a \in$ *world.Actions(s)* **do**
    $s_{next} \leftarrow s$;
    `//` s `is altered to` s_{next} `by transition`
    world.Transition(s_{next}, a);
    if $s_{next} \notin S \wedge$ *not world.Terminal*(s_{next}) **then**
      Add s to S;
      Push s_{next} to *queue*;
    end
  end
  Pop s from *queue*
end

5.3 Agents and Worlds

This chapter provides a detailed description of all worlds and all experiments

5.3.1 List of Agents

RU is an agent with random uniform policy. It serves as a baseline for other agents.

GFVMC is an agent with greedy policy which uses first visit Monte Carlo approximation of state action value function. It has only discount parameter gamma.

GFEVMC is an agent with greedy policy which uses every visit Monte Carlo approximation of state action value function. It has only discount parameter gamma.

EGFVMC is an agent with epsilon greedy policy which uses first visit Monte Carlo approximation of state action value function. It has discount parameter gamma and and policy parameter epsilon.

EGFEVMC is an agent with epsilon greedy policy which uses every visit Monte Carlo approximation of state action value function. It has discount parameter gamma and and policy parameter epsilon.

OSGFVMC is an agent with greedy policy which uses first visit Monte Carlo approximation of state action value function. If there are any unexplored states they are visited with probability 1. It has discount parameter gamma.

OSGFEVMC is an agent with greedy policy which uses every visit Monte Carlo approximation of state action value function. If there are any unexplored states they are visited with probability 1. It has discount parameter gamma.

OSEGFVMC is an agent with epsilon greedy policy which uses first visit Monte Carlo approximation of state action value function. If there are any unexplored states they are visited with probability 1. It has discount parameter gamma and policy parameter epsilon.

OSEGFEVMC is an agent with epsilon greedy policy which uses every visit Monte Carlo approximation of state action value function. If there are any unexplored states they are visited with probability 1. It has discount parameter gamma and policy parameter epsilon.

EGDQN is an agent with policy approximated by neural network which uses Q-learning approximation of state action value function. It is used only with pixel input and stacks pixel perceptions up to a certain number called $Markovorder$. It takes neural network as parameter and also discount parameter gamma, policy parameter epsilon and $Markovorder$. This agent is inspired by Google Deepmind paper [6].

5.3.2 K-Bandit World

K-Bandit world is inspired by an actual slot machine originally called a one-armed bandit for a special case with one lever. K-Bandit is generalization to a slot machine with K levers. Selecting (pulling) a lever immediately provides agent a reward. Agent does not know values associated with levers in advance and has to learn them in order to maximize total reward. It represents a simple

control problem which reflects core ideas of reinforcement learning. Those ideas are learning action selection model based on observation (as opposed to training signal) which necessarily introduces exploration versus exploitation dilemma.

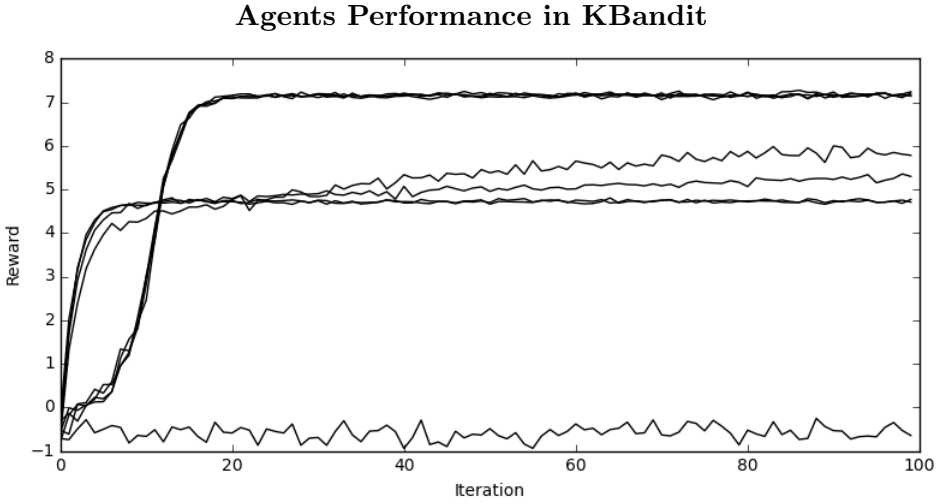


Figure 5.1: As sorted at last iteration: Lowest line at bottom is RU agent then GFVMC and EGFVMC agents, then OSGFVM, OSEGFVMC, GEVMC and EGEVMC with very similar performance.

5.3.2.1 Description

In this particular K-Bandit implementation reward R given after selecting discrete *action* $a \in A$ is a random variable R_a distributed according to Gaussian distribution with uniformly distributed mean μ and variance of one $\sigma = 1$:

$$\mu = \mathcal{U}(min, max) = \begin{cases} \frac{1}{max-min} & \text{for} \quad \mu \in [min, max] \\ 0 \quad \text{otherwise} \end{cases}, \tag{5.1}$$

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(x-\mu)^2}{2\sigma^2}), \tag{5.2}$$

$$R_a = \mathcal{N}(\mu, 1) = \frac{\exp(-(x-\mu)^2)}{\sqrt{2\pi}} \tag{5.3}$$

5.3.2.2 Action Value Estimation

Each of discrete actions is associated with single lever and therefore with single reward. There is always the best action $a \in A$ with greatest mean reward. Optimal strategy is to select only the best action all the time. But there are two complications. Firstly, it is not known to the agent which action is the best

apriori which means agent has to try all of them. Secondly reward signal is stochastic forcing the agent to try each action many times in order to discover best one. The definition of *value*, which is based on those two properties, is then *expected reward given action*, $q_*(a)$, which has to be estimated at each time step t as $Q_t(a)$ from agent's experience.

Algorithm 10: Bandit ϵ-greedy action selection algorithm

Initialize Q, N such that:
$\forall a \in A$, $Q(a) = 0$, $N(a) = 0$

repeat

$$A \leftarrow \begin{cases} argmax_a Q(a) & \text{with prob. } 1-\epsilon \text{ (breaking ties randomly)} \\ \text{a random action} & \text{with prob. } \epsilon \end{cases} \tag{5.4}$$

$R \leftarrow bandit(A)$;
$N(A) \leftarrow N(A) + 1$;
$Q(A) \leftarrow Q(A) + \frac{[R - Q(A)]}{N(A)}$;

until *terminated*;

One way to estimate $Q_t(a)$ from agent's experience is to compute average such as in algorithm 10. There is one average estimate for each discrete action. For example unbiased sample mean estimation methods like arithmetic average is trivially Fisher efficient in K-Bandit world where reward is distributed according to multivariate $\mathcal{N}(\mu, \Sigma)$.

5.3.3 Grid World

GridWorld is supposed to be easily scalable problem of higher difficulty with more properties than KBandit. KBandit is the simplest because it has only single state. GridWorld introduces multiple states and identifiable representation of each state becomes crucial for value function approximation. GridWorld can be easily scaled if necessary and is able to provide state space of a size in between KBandit and BreakOut which is useful for iterative development.

Definition

GridWorld is implemented as a simple two dimensional grid of variable width and height. Default testing scenario is a GridWorld of size 5x5 with two teleporting states located at (0, 1) and (0, 3) using zero indexing (see black squares in figure 5.4). First (0, 1) teleporting state provides agent with reward of 10 and second teleporting state provides agent with reward of 5.

5.3.3.1 Experiments and Evaluation

First value iteration was used to get insight about learning 5.4. Then all agents excluding EGDQN agent were learned in first by evolution and evolved parameters were used for evaluation by epizodic learning algorithm 2 until full convergence 5.2.

Various performance is achieved by various agents. First visit Monte Carlo agents seem to converge much more slowly.

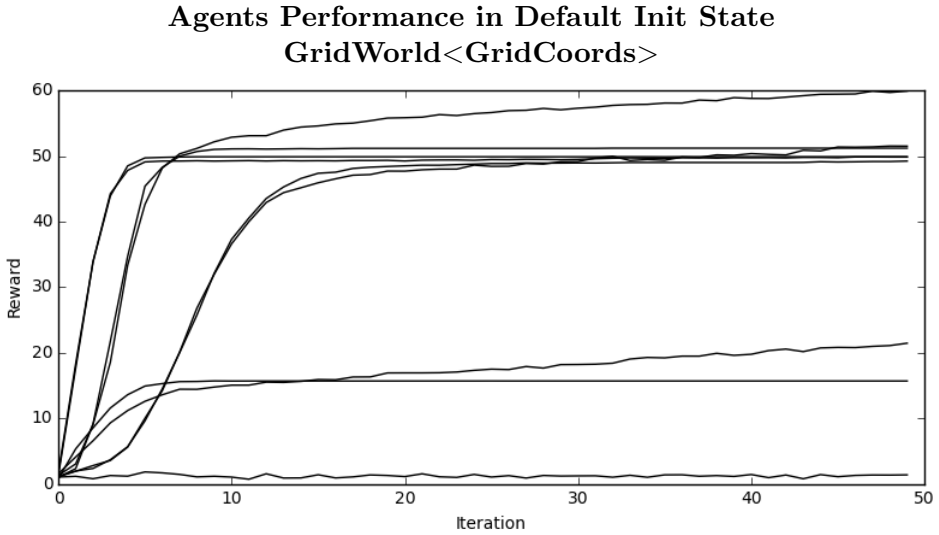


Figure 5.2: As sorted at last iteration: Lowest line at bottom is RU agent then GFVMC and EGFVMC agents. Then OSGFVM, OSEGFVMC, GEVMC and EGEVMC agents with very similar performance.

5.4 Breakout World

BreakOut world is a variation of Pong game simulator. Multiple variations exist each with different characteristics. First some standard implementations are described. Some of those characteristics are picked and finally implementation of BreakOut and its relation to theory is described in detail.

5.4.1 Original Pong

In original Pong version there are usually two players playing against each other. There are two paddles (one for each player) situated by the sides. Paddles are controlled by players and have three actions available: move up, move down, stay. There is also a ball moved around at constant or variable speed bouncing off top and bottom walls or paddles. Players score points

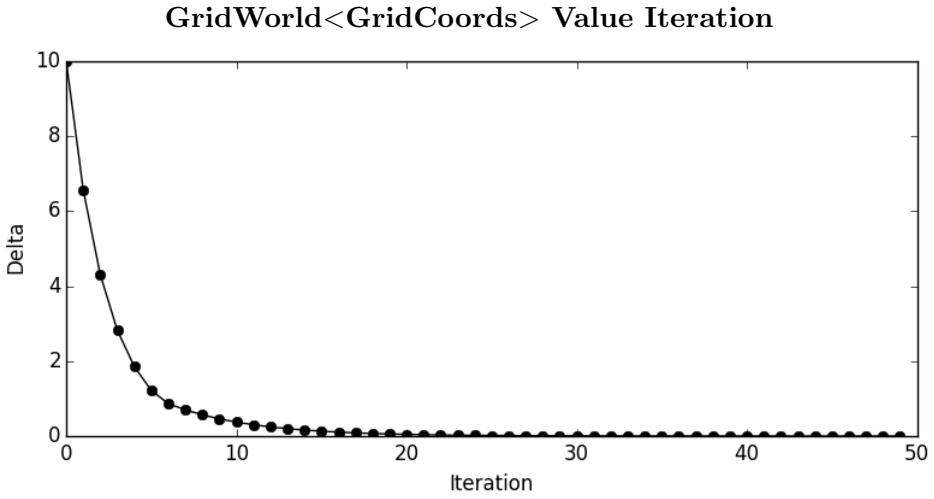


Figure 5.3: State value iteration 4 with parameters $\gamma = 0.9$, $\epsilon = 0.0001$, $limit = 50$ evaluated on GridWorld.

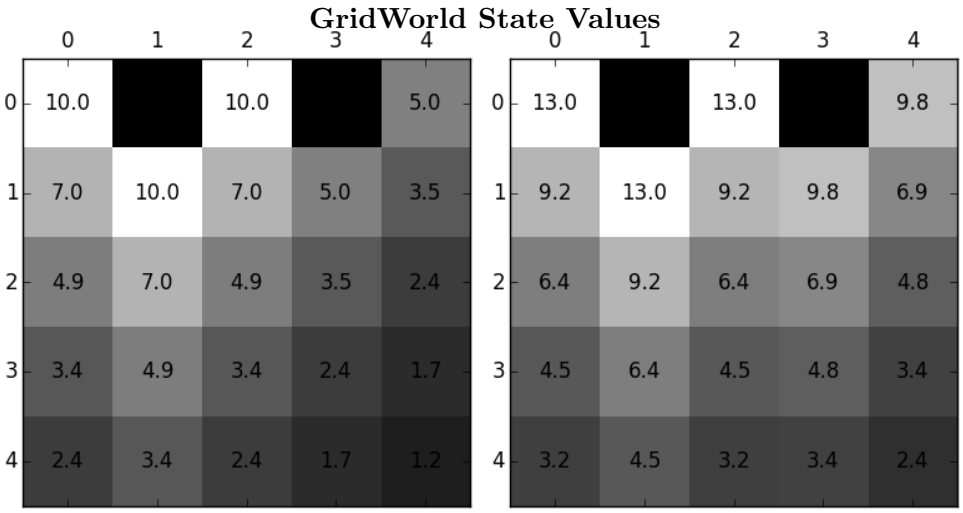


Figure 5.4: GridWorld state values generated by SVI 4 algorithm with parameters $\gamma = 0.7$, $\epsilon = 0.0001$, $limit = 50$ evaluated on GridWorld. Left: First iteration. Right: Last iteration. Greedy policy using first iteration is already optimal.

by when other player does not bounce incoming ball and lets it pass behind paddle. Game usually ends after some score is reached or until terminated.

5.4.2 Original BreakOut

Original BreakOut is usually single player version of Pong. Interesting game mechanics is usually added by concept of bricks. Bricks are small obstacles somehow scattered in space and can be broken for points. Some advanced

versions also include bricks with special characteristics like indestructibility or bonuses which occasionally appear when brick is destroyed. Game usually ends (or progresses to next level) when all bricks are destroyed.

5.4.3 BreakOut Implementation

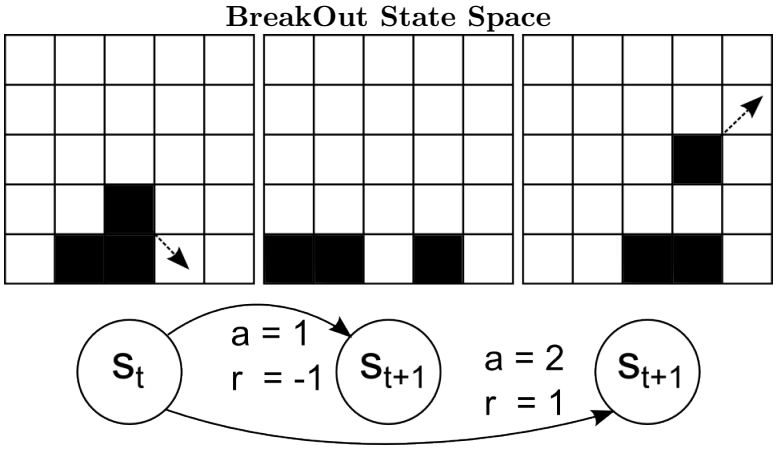


Figure 5.5

BreakOut implemented in this thesis is a very simple version. After some considerations and initial experiments only core concepts are implemented. There is only one paddle (single player) and a ball. This is actually quite sufficient because learning from pixels is non-trivial even when very little is happening on the screen.

State Space Discretization

Ultimately everything is discrete in digital computer. But conceptually we usually think about game space as continuous computed with floating digits numbers which is then discretized by a grid to create visual pixel representation. However this BreakOut implementation introduces a discretized game space.

It has several advantages over continuous representation. Especially when the whole game window is very small and when ball is defined by single pixel then continuous representation might give different results by various rounding of float values undermining Markov property. Ball movement when bouncing around may then appear almost as random fluctuation.

Ball movement is discretized by discrete force vectors obeying discrete force range 5.6. Such representation of ball movement is well defined even for very small instances of the BreakOut world. The same figure also shows all possible shortest paths from one position to another - with force range vector

movement there is only one path. In continuous representation ball will likely select those paths according to rounding errors.

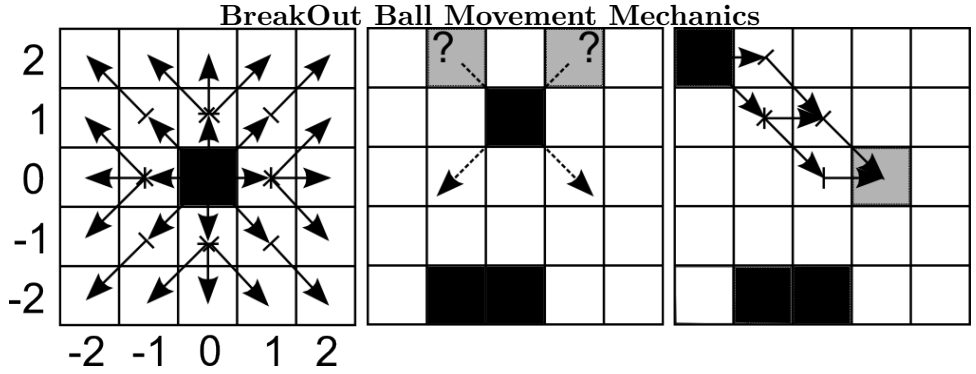


Figure 5.6: Left: Ball movements defined by force vector with force range parameter set to 3. Middle: Ball movement does not have Markov property, next state depends on previous. Right: All different shortest paths to given position.

Terminal States

Since there are neither bricks or opponent the obvious way of termination is time limit. Actually two kinds of termination are implemented. First is a termination after some time limit (which ensures process finiteness) combined with termination when ball is lost. One time tick corresponds to one state transition. Second is a termination on both successful paddle bounce and when ball is lost. It also introduces time limit but it is always ruled out by high value.

Reward Signal

Reward or punishment signal has to be connected to some desirable or undesirable states. With paddle and ball there are few events happening: bounce from paddle, bounce from wall, loosing the ball. Two kinds of reward signal are implemented and switchable by parameters. First is only positive reward at bounce from paddle event. Second adds also negative reward when ball is lost.

Perception Types

Three types of perception are provided. Each perception type modifies set of assumptions which are valid from agents perspective. They are:

PaddleBallAndForce perception which is actually enough to define state uniquely and has Markov property. This is the most easily learned representation which is feasible even for tabular estimation methods (all

agents up without RU and EGDQN). It should serve as a baseline for other perception types.

PaddleAndBall perception is missing ball force vector. Markov property is lost. This is still a nice representation and should represent intermediate level of complexity.

Pixels perception is a final type of perception. The point of previous perception types is to show how insanely much harder it is to learn from pixels even for a very simplified problem. Mastering this type of perception is main goal of this thesis.

5.4.4 BreakOut Experiments

5.4.4.1 Experiment Design Exluding EGDQN Agent

Experiments survey the ability of agents to learn the same instance of BreakOut with different perception types. Basic BreakOut parameters are: size is 8x8, paddle is 2 pixels large, force range is 3, termination on bounce and only positive reward. First the insights into expected learning difficulty are provided by perception value iteration algorithm 8 which is run for all perception types.

There are also two different state initialization regimes - random state and default state. Default state initializes to the middle of BreakOut screen and sets all state values to boundary values provided during world initialization. Epizode initialized by default state is always the same and differs only through agent actions and checks if agent is able to catch the ball which is always going the same way. Random state on the other hand randomizes ball position somewhere in the half of the screen opposite to paddle and also randomizes direction of initial ball force vector. Epizode initialized by random state is ability to catch the ball in general which can be thrown in many ways (even to oposite direction and then bounce off the wall multiple times).

All agents except EGDQN agent are then evaluated in two fold process. First evolution 1 of each agent's set of hyperparameters is used to explore hyperparameter space. Then the parameters of fittest individual are used for evaluation by episodic learning algorithm 2.

5.4.4.2 Experiment Design for EGDQN Agent

EGDQN agent experiments are designed and performed separately. Evolution is omitted because it would take too much time to evaluate peoperly fitness properly (to full convergence). Even evaluations of a single individual are not performed to full convergence with exception of small BreakOut with termination on paddle bounce. It would probably take many hours or days of computation time to converge on bigger BreakOut without termination.

There are three different versions of BreakOut used in experiment:

Simple this is first working simplified enough parameterization found. It is the same as for other agents in order to do direct comparison. BreakOut parameters are: size is 8x8, paddle is 2 pixels large, force range is 3, termination on bounce and only positive reward.

Big this is simply a little bit bigger BreakOut with size 12x12. To make things more interesting paddle is still only 2 pixels wide. Force range is 3, termination on bounce and only positive reward.

Hard this version is called hard because termination is only on miss (or time limit). Other parameters are the same as small: size is 8x8, paddle is 2 pixels large, force range is 3 and only positive reward.

For three different pong version there are three EGDQN agents with different neural networks. All three types of neural networks from experiments on MNIST dataset are used because their performance was comparable. Parameterization is also inspired by MNIST results.

Perceptron with hyperbolic tangent activation and learning rate $\alpha = 0.01$.

Two Layer Perceptron with 32 hidden neurons in first layer. Learning rates are set to $\alpha_1 = 0.005$ and $alpha_2 = 0.01$ for first and second layer respectively. Hyperbolic tangent activation is used in both layers.

ConvNet with receptive field 4x4 and 16 filters. Learning rates are set to $\alpha_1 = 0.005$ for convolutional layer and $alpha_2 = 0.01$ for perceptron layer.

5.4.5 Evaluation of Results

All results of evolution are because there is too much of them and only some of them are interesting. But for the sake of completeness they are included in appendix B.

5.4.5.1 Notes on Evolution

Now a few interesting observation from graphs from appendix B. First it needs to be mentioned that gamma parameters have no effect in KBandit world because it has effectively only one single state. Gamma parameters seem to have little to no effect overall parameterizations.

Looking on epsilon parameters of agents without one shot learning there is a triangluar shape of increasing variance as epsilon decreases. This is because of unlucky agents who explored bad actions first stick with them because of small epsilon value. However lucky agents get stuck with actually good states giving them advantage against more exploratory agents. This pattern is also

true for GridWorld with slight bends and twists under different parameterizations.

Slightly more epsilon curves can be observed for BreakOut world combined with paddle ball and force perception. Variance is increased with random initialization. For paddle and ball perception it is similar but with even more variance. With pixel perception epsilon parameters look randomly scattered.

5.4.5.2 All Agents Performance

All results are provided on figures below.

First perception type when ball and paddle positions and force vector are provided is easy to learn as expected. Value iteration algorithm shows fast convergence which indicates that task should be easy and it is proved by all agents who learn the task easily. Only when random initialization is used there is a problem with one shot agents probably because state space is too big and they get stuck in exploration.

Second perception type when only ball and paddle positions are provided is problematic as indicated by value iteration. However results are similar to first perception results and all agents with exception to one shot agents manage to learn it.

Third perception type is pixels where everything including value iteration and all agents fail miserably as expected. So even when it is possible to hold all pixels in memory learning itself is infeasible without preprocessing.

5.4.5.3 EGDQN Agent Performance

In simple BreakOut all agents seem to converge to optimal solution. The most prominent is two layered network which achieves around 90 out of 100 bounces.

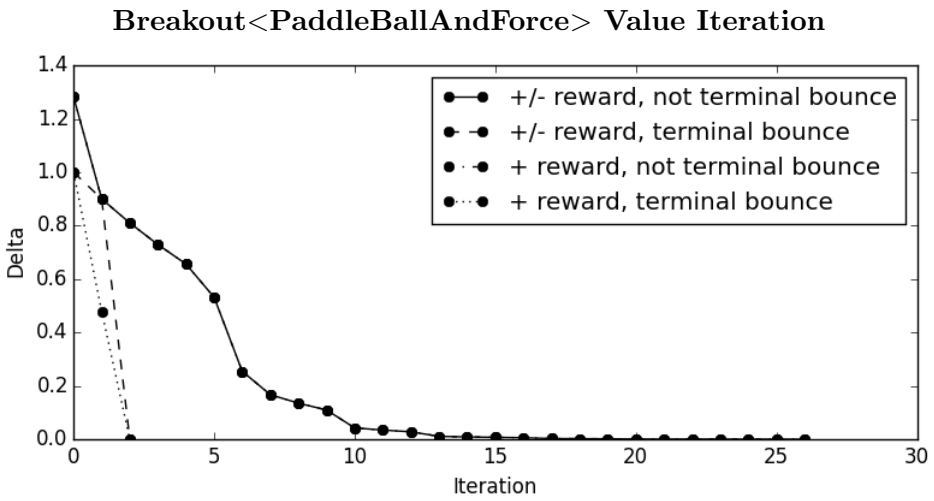


Figure 5.7: State value iteration 8 with parameters $\gamma = 0.9$, $\epsilon = 0.0001$, $limit = 50$ evaluated on 8x8 Breakout.

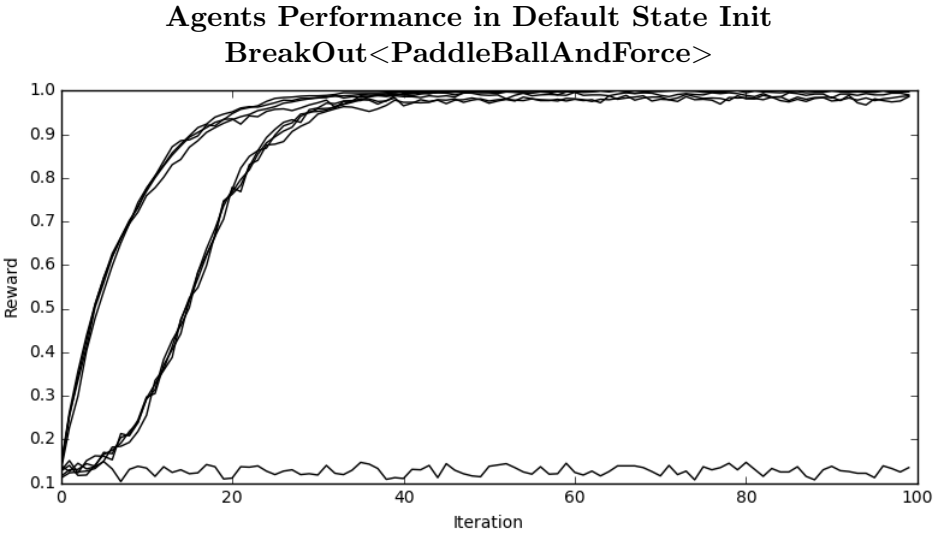


Figure 5.8: As ordered around iteration 10: At bottom there is RU agent. At bottom there are (without order) OSEGEVMC, OSEGFVMC, OSGFVMC and OSGEVMC agents. At top there are (without order) EGEVMC, EGFVMC, GFVMC and GEVMC.

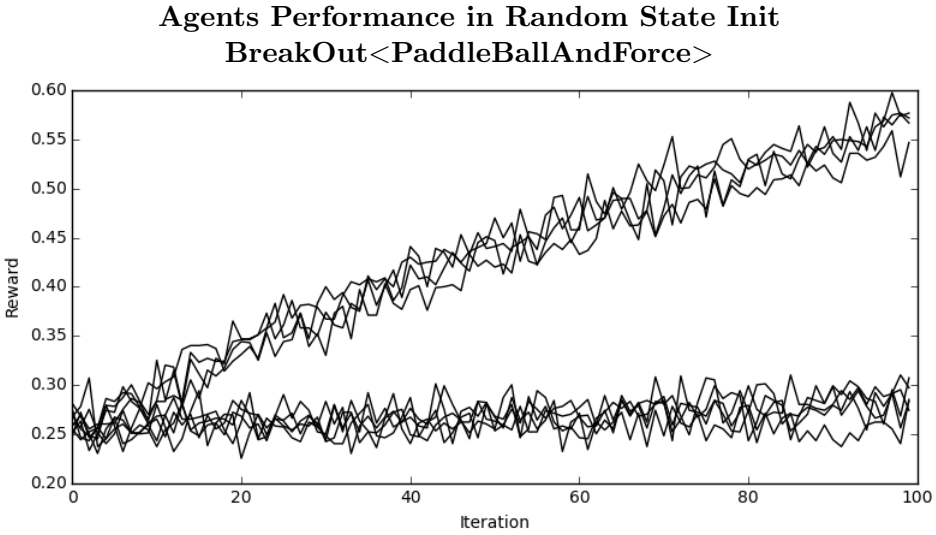


Figure 5.9: As ordered at last iteration: At bottom there are (without order) RU, OSEGEVMC, OSEGFVMC, OSGFVMC and OSGEVMC agents. At top there are (without order) EGEVMC, EGFVMC, GFVMC and GEVMC.

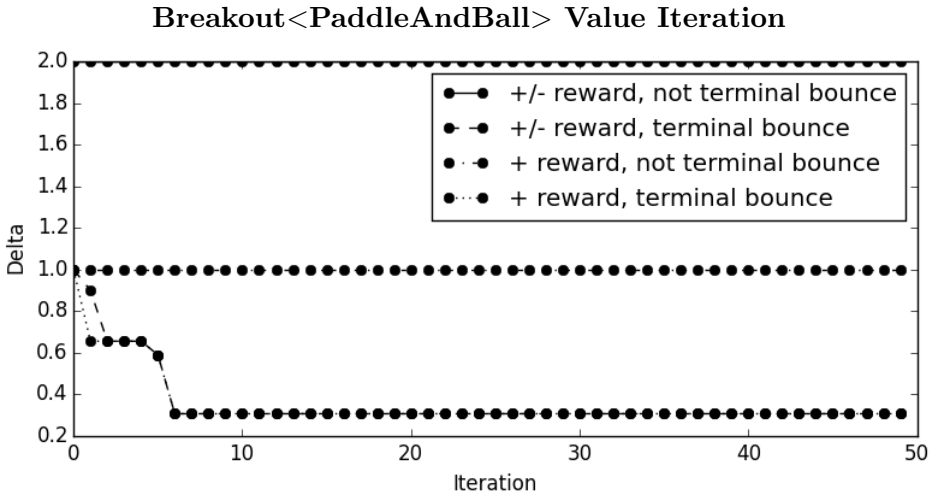


Figure 5.10: State value iteration 8 with parameters $\gamma = 0.9$, $\epsilon = 0.0001$, $limit = 50$ evaluated on 8x8 Breakout.

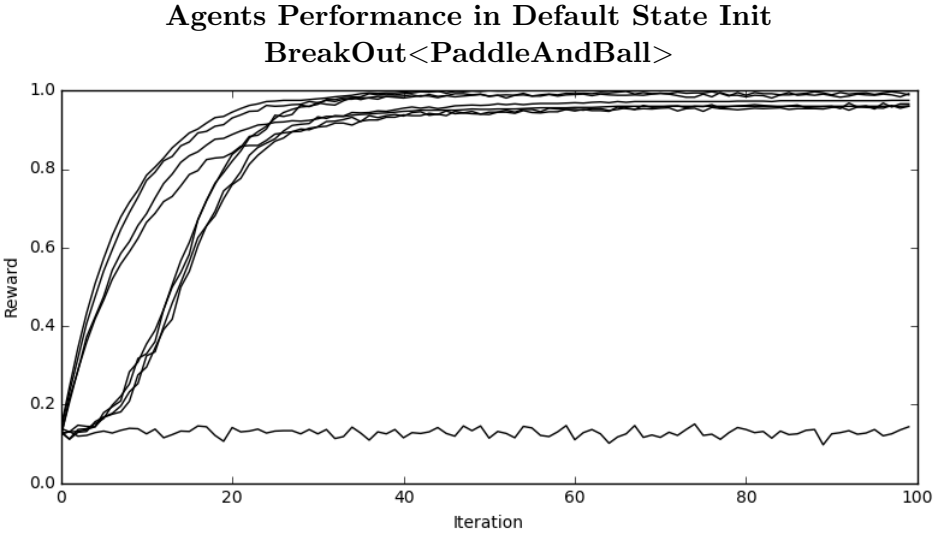


Figure 5.11: As ordered around iteration 10: Bottom line is RU agent. Four in the middle are OSEGFVMC, OSGFVMC (lower two) and OSEGEVMC, OSGEVMC (upper two). Top four are from bottom up EGFVMC, GFVMC, EGEVMC and GEVMC agents.

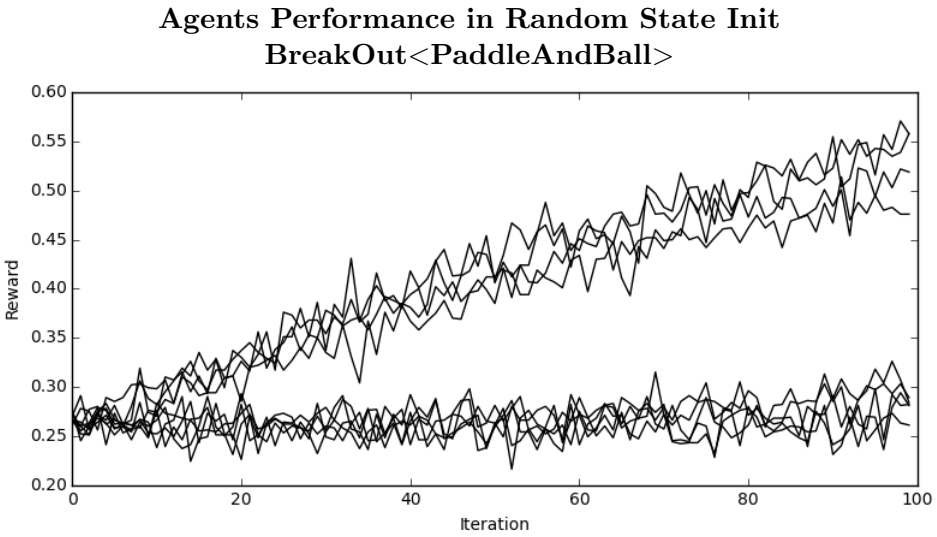


Figure 5.12: As ordered at last iteration: At bottom there are (without order) RU, OSEGEVMC, OSEGFVMC, OSGFVMC and OSGEVMC agents. At top there are (without order) EGEVMC, EGFVMC, GFVMC and GEVMC.

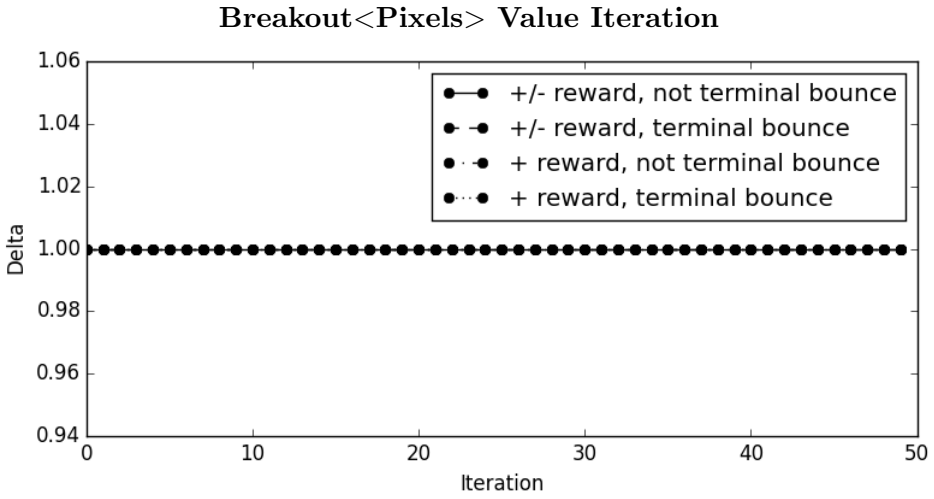


Figure 5.13: State value iteration 8 with parameters $\gamma = 0.9$, $\epsilon = 0.0001$, $limit = 50$ evaluated on 8x8 Breakout.

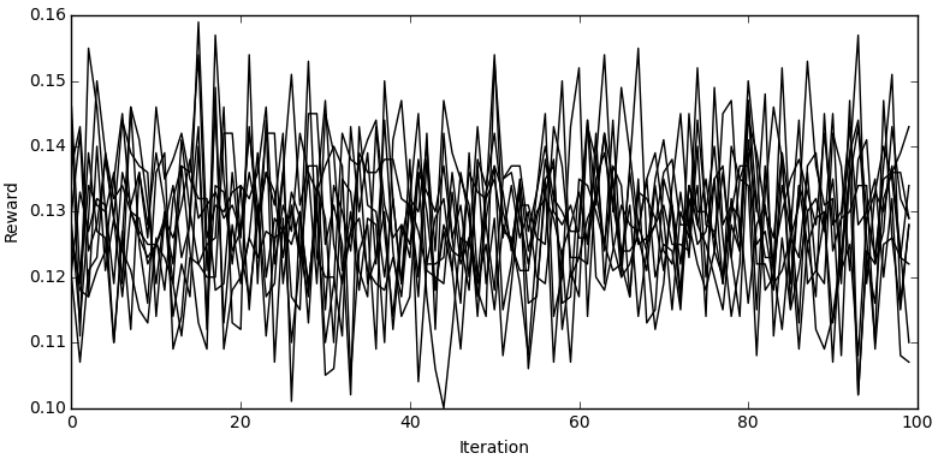


Figure 5.14: All agents totally fail. They are as useless as RU agent.

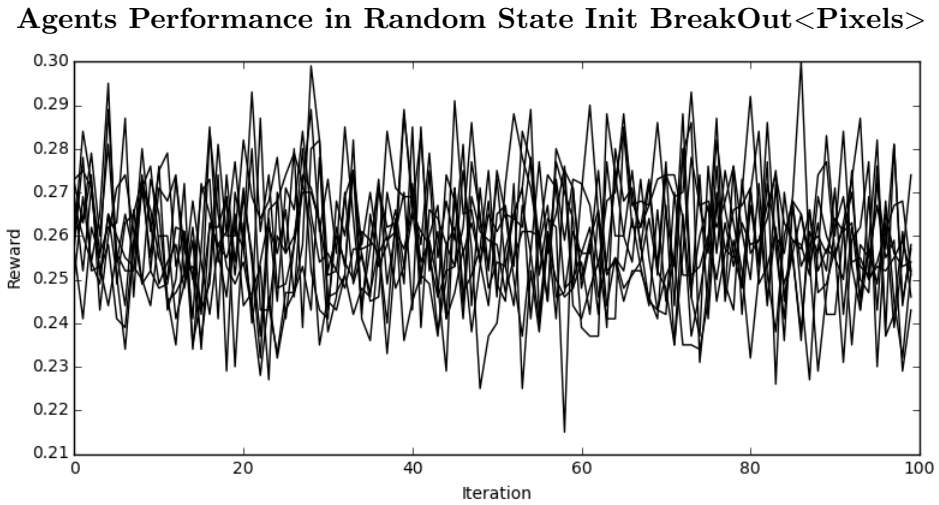


Figure 5.15: All agents totally fail. They are as useless as RU agent.

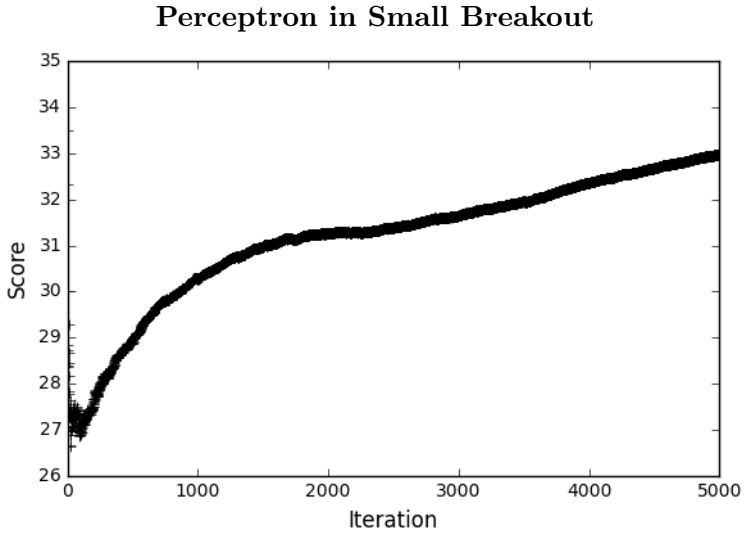


Figure 5.16: The most successful EGDQN agent in simple BreakOut which achieves around 90 out of 100 possible bounces.

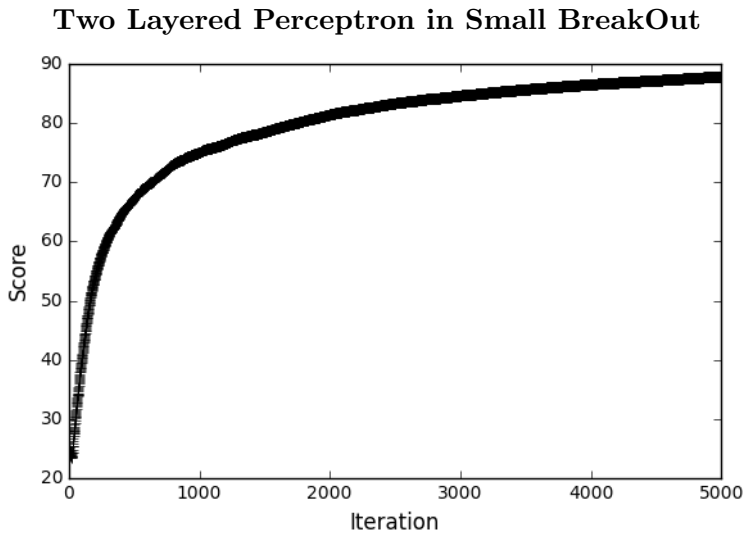


Figure 5.17: Seems to converge properly. Needs more training time.

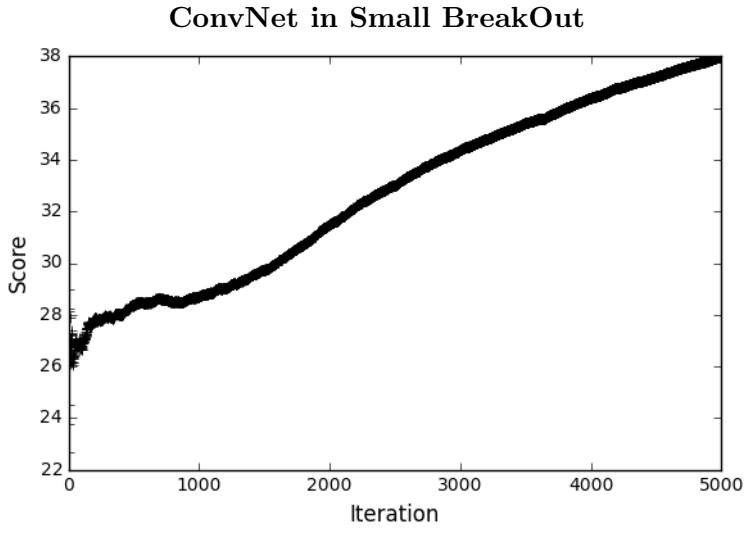


Figure 5.18: Seems to converge properly. Needs more training time.

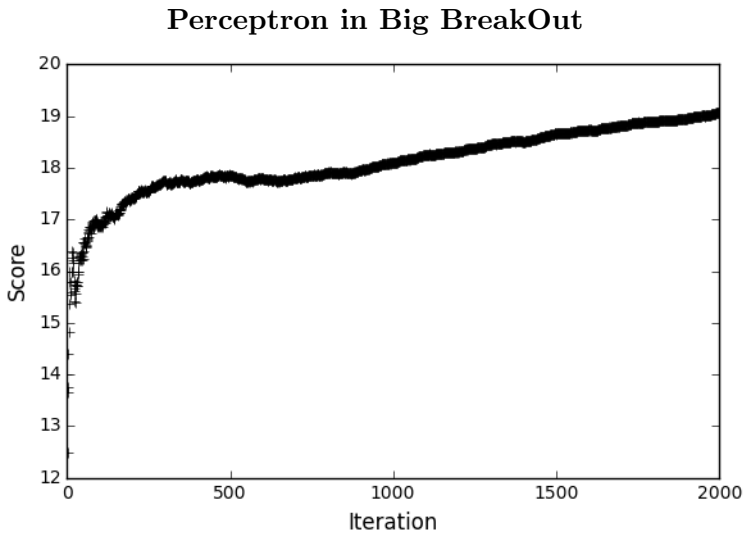


Figure 5.19: It looks like it is converging but this could be just result of some deverged state.

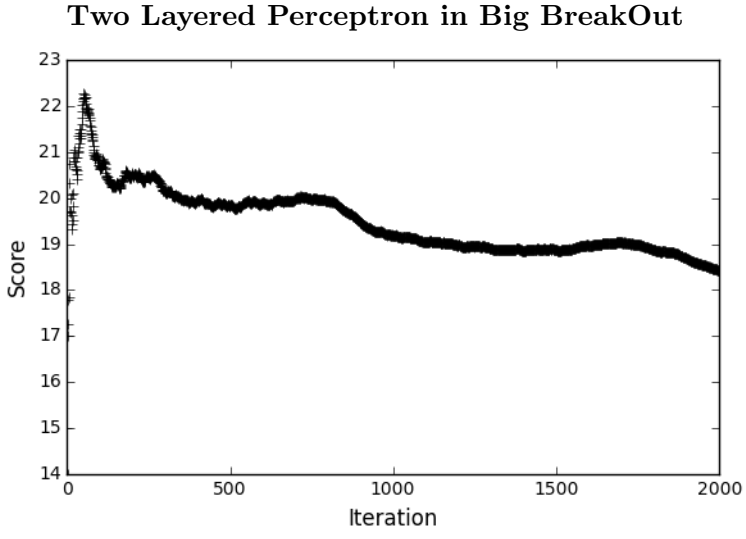


Figure 5.20: It seems to diverge immediately.

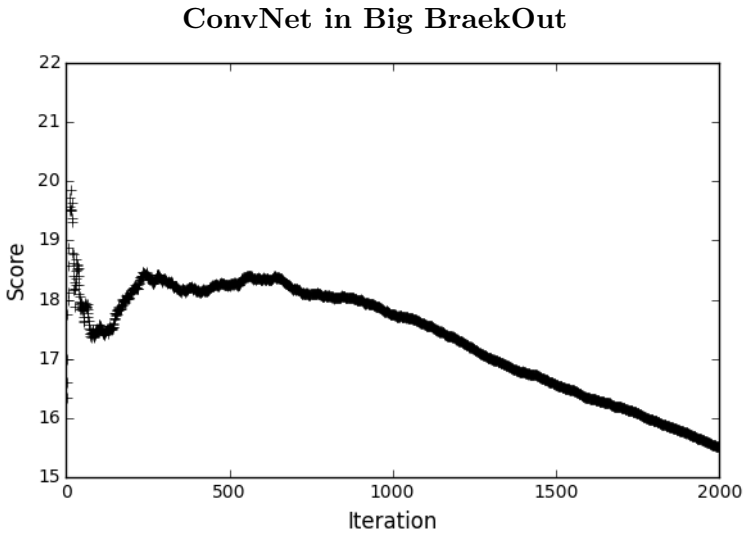


Figure 5.21: Seems to diverge.

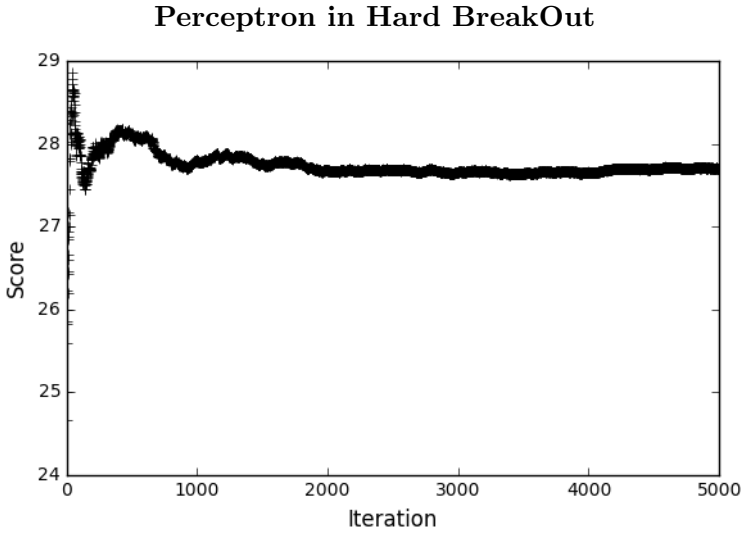


Figure 5.22: It seems to be stuck in oscilations.

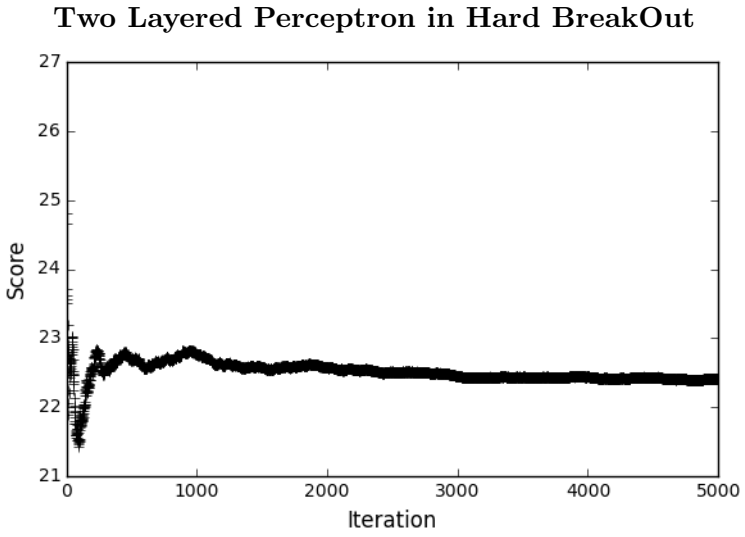


Figure 5.23: Seems to be stuck in oscilations.

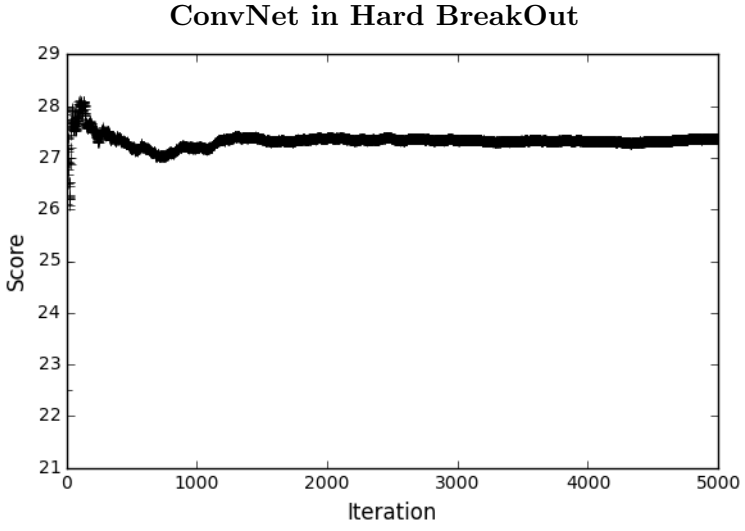


Figure 5.24: Seems to be stuck in oscilations.

Conclusion

Achieved Goals

Main Goal

The final goal of this thesis is to implement a learning algorithm with ability to learn how to play a simple game given only the visual input represented by pixels and number of available actions. There is no other prior or runtime information about game or game state respectively. Deep Q-learning network was implemented from ground up using only C++ programming language with standard libraries. DQN is able to, one can almost say, master simplified variation of Pong game. Although Pong game is simplified it is still non-trivial problem both theoretically and practically. Similar success with bigger DQN on more complicated Pong variation and other Atari games was achieved by Google Deepmind in 2013 was considered breakthrough in the field.

Incremental Approach

Deep Q-learning network is a combination of reinforcement learning which defines a target state action value function and function approximation which is used to approximate the target function based on high dimensional visual perception. Function approximation performance of various neural networks and their ability to represent visual state space with different parameterization explored by evolutionary programming is tested separately on MNIST dataset. Similarly state action value estimation is tested not only in Pong (with various state representation other than pixels) but also in two other games (KBandit and GridWorld) providing incrementally harder task. Sensitivity of value iteration algorithm for all worlds to various state representations provides insight on task complexity.

Developed Framework

Architecture

Final learning algorithm is actually implemented as a part of whole framework which models games as a worlds and learning algorithms as agents acting in those worlds. It has carefully designed theory based interfaces and leverages multiple programming paradigms such as object-oriented, functional programming and generic template metaprogramming. IWorld and IState interfaces enable to easily simulate worlds as finite Markov decision process. IState is perceived by IAgent as IPerception which could seamlessly change situation to finite partially observable Markov decision process.

Tensors and Neural Networks

Another part of a framework deals with necessary mathematical routines and neural networks. Complicated multidimensional operations such as inner and outer products, broadcasts, normalizations and other are implemented by tensors. Perceptron and convolutional layers implement ILayer interface and can be combined arbitrarily in resulting network. Parameterization such as learning rate or neuron types with different activation function is also done layer wise.

Metalearning by Evolution

Last part of the framework deals with metalearning of hyperparameters by improved fast evolutionary programming. Fitness function is supplied as lambda function with local scope capture making it easy to use in any situation. All IAgents can be evolved and resulting parameterization easily used.

Possible improvements

There is vast amount of improvements of various difficulty which can be applied on top of this thesis. The most obvious is to use the same learning algorithm as Google Deepmind [6] which is minibatch RMSProp learning. Use of minibatch opens possibility for minibatch normalization and other minibatch techniques. Other learning algorithms for example higher order approximators such as LBFGS or Conjugate Gradientss can be also utilized. Or other type of neural network can be used such as one of the recurrent neural networks (LSTM) which are naturally good for sequence learning. Even simple recurrent nets can be powerful with right techniques. More computation time should be definitely used. Parallel tensor operations on GPU are able to speed up whole network multiple times. Deeper hierarchy with pretrained with autoencoders would be interesting. Ensemble of multiple models also often increases performance. The list of improvements is almost endless.

Bibliography

[1] de Laplace, P.; Truscott, F.; Emory, F. *A Philosophical Essay on Probabilities*. A Philosophical Essay on Probabilities, Wiley, 1902. Available from: `https://books.google.cz/books?id=WxoPAAAAIAAJ`

[2] Wolpert, D. H.; Macready, W. G. No Free Lunch Theorems for Optimization. *Trans. Evol. Comp*, volume 1, no. 1, Apr. 1997: pp. 67–82, ISSN 1089-778X, doi:10.1109/4235.585893. Available from: `http://dx.doi.org/10.1109/4235.585893`

[3] Yao, X.; Liu, Y.; Lin, G. Evolutionary Programming Made Faster. *Trans. Evol. Comp*, volume 3, no. 2, July 1999: pp. 82–102, ISSN 1089-778X, doi:10.1109/4235.771163. Available from: `http://dx.doi.org/10.1109/4235.771163`

[4] Bäck, T. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, USA, 1996.

[5] Sutton, R. S.; Barto, A. G. *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, first edition, 1998, ISBN 0262193981.

[6] Mnih, V.; Kavukcuoglu, K.; Silver, D.; et al. Playing Atari with Deep Reinforcement Learning. Technical report arXiv:1312.5602 [cs.LG], Deepmind Technologies, Dec 2013.

[7] Hornik, K. Approximation Capabilities of Multilayer Feedforward Networks. *Neural Netw.*, volume 4, no. 2, Mar. 1991: pp. 251–257, ISSN 0893-6080, doi:10.1016/0893-6080(91)90009-T. Available from: `http://dx.doi.org/10.1016/0893-6080(91)90009-T`

[8] Hebb, D. O. *The Organization of Behavior*. Wiley, New York, 1949.

[9] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, volume 65, no. 6, Nov. 1958: pp. 386–408.

[10] Hopfield, J. J. Neurocomputing: Foundations of Research. chapter Neural Networks and Physical Systems with Emergent Collective Computational Abilities, Cambridge, MA, USA: MIT Press, 1988, ISBN 0-262-01097-6, pp. 457–464. Available from: `http://dl.acm.org/citation.cfm?id=65669.104422`

[11] Minsky, P., Marvin; Seymour. *Perceptrons*. Oxford, England: M.I.T. Press., 1969.

[12] Werbos, P. J. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Dissertation thesis, Harvard University, 1974.

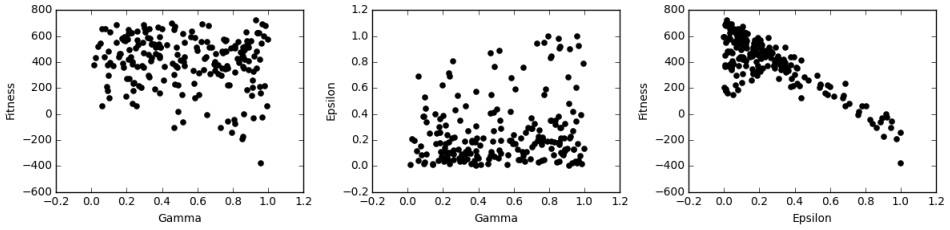
Appendix A

Excluded Graphs

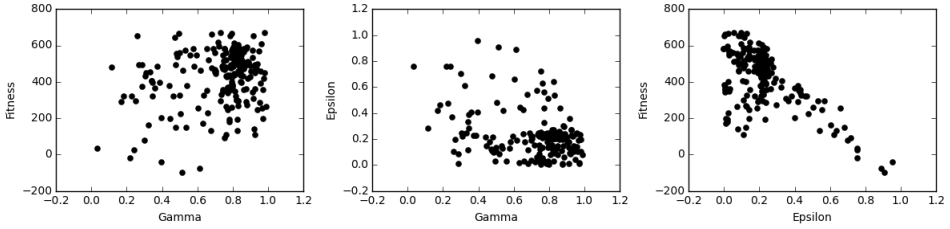
Appendix B

Excluded Graphs

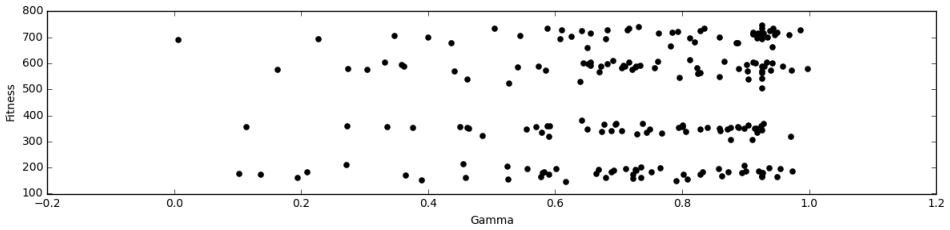
Metalearning of EGEVMC KBandit



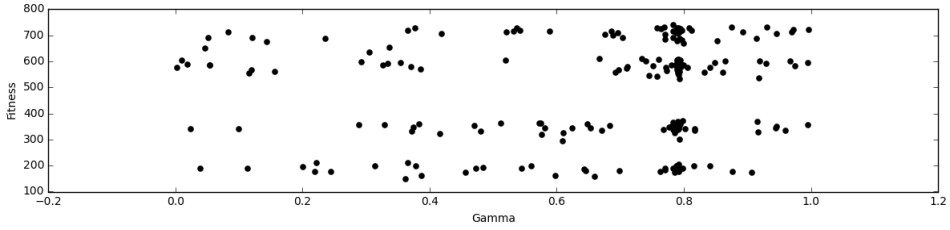
Metalearning of EGFVMC In KBandit



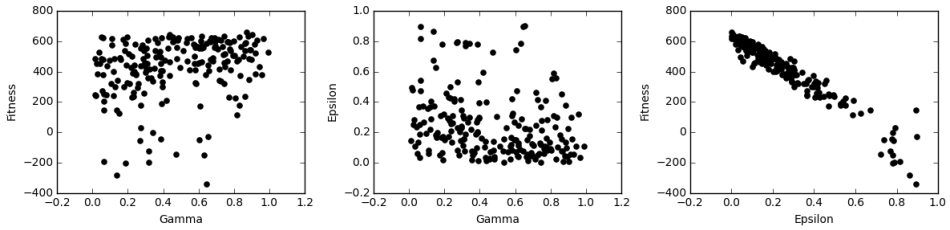
Metalearning of GEVMC In KBandit



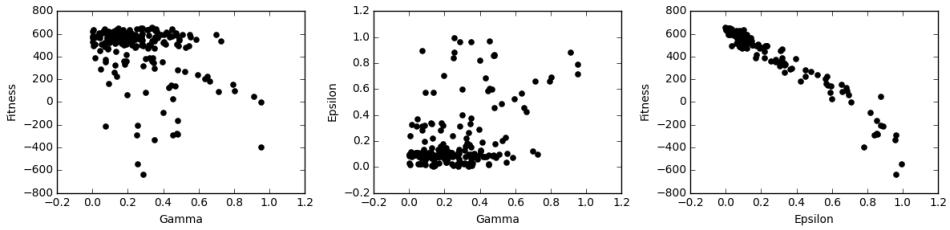
Metalearning of GFVMC In KBandit



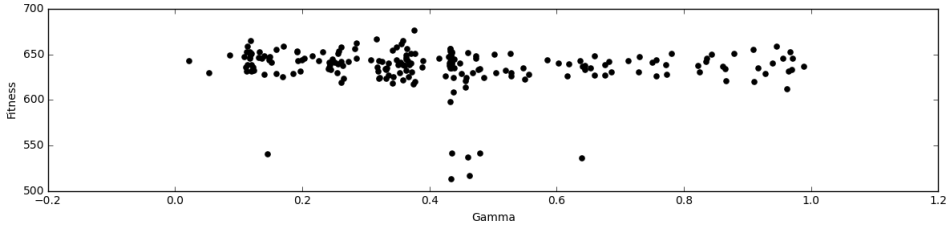
Metalearning of OSEGEVMC In KBandit



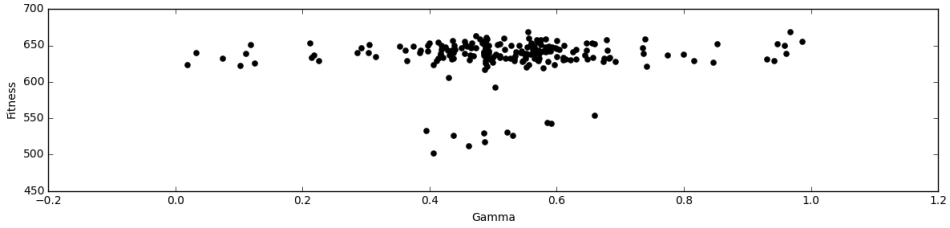
Metalearning of OSEGFVMC In KBandit



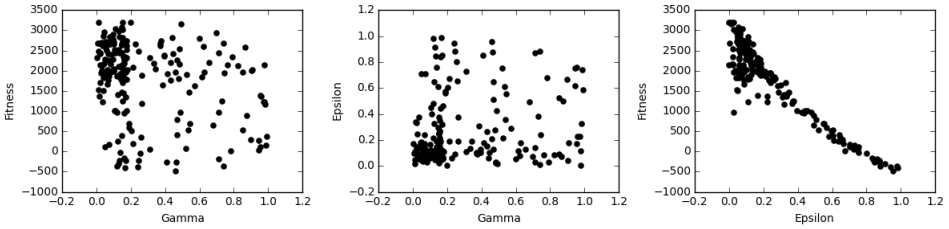
Metalearning of OSGEVMC In KBandit



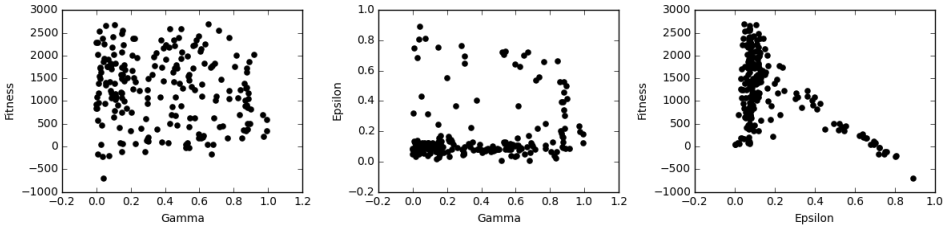
Metalearning of OSGFVMC In KBandit



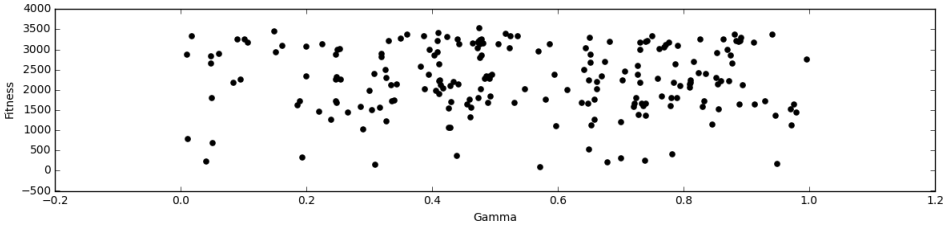
Metalearning of EGEVMC In Random State Init GridWorld<GridCoords>



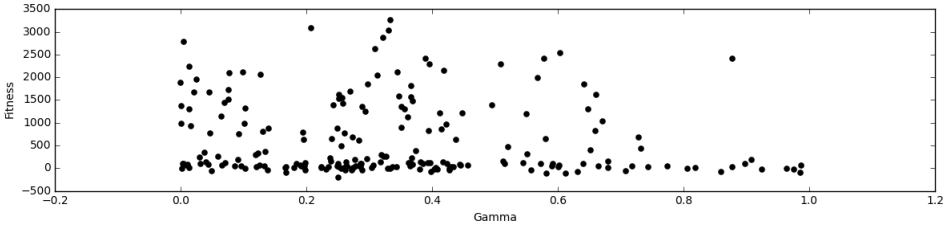
Metalearning of EGFVMC In Random State Init GridWorld<GridCoords>



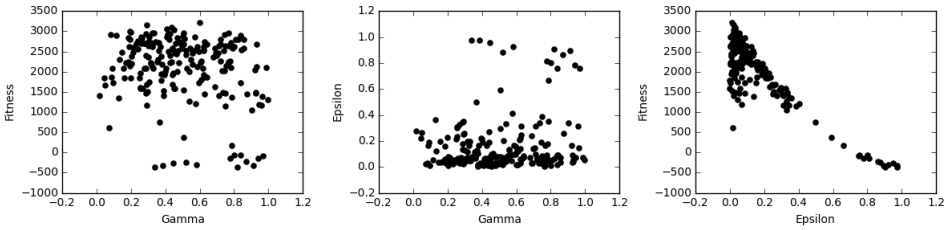
Metalearning of GEVMC In Random State Init GridWorld<GridCoords>



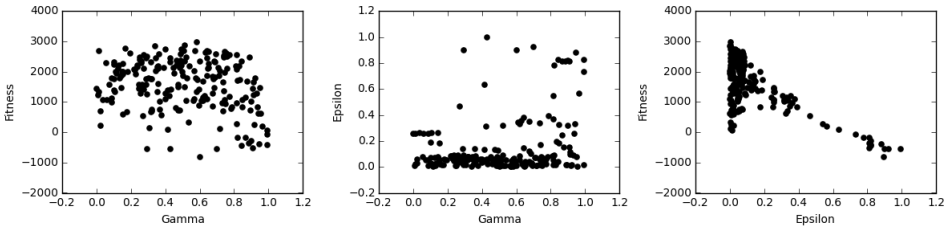
Metalearning of GFVMC In Random State Init GridWorld<GridCoords>



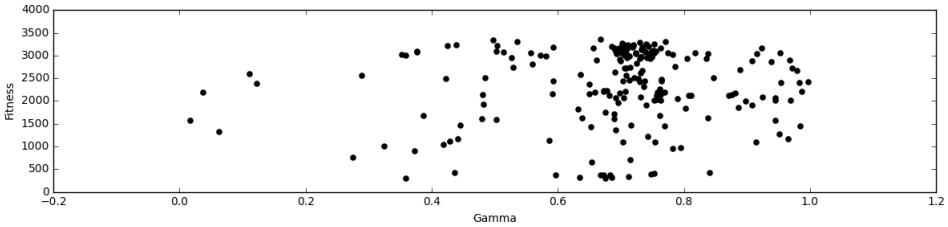
Metalearning of OSEGEVMC In Random State Init
GridWorld<GridCoords>



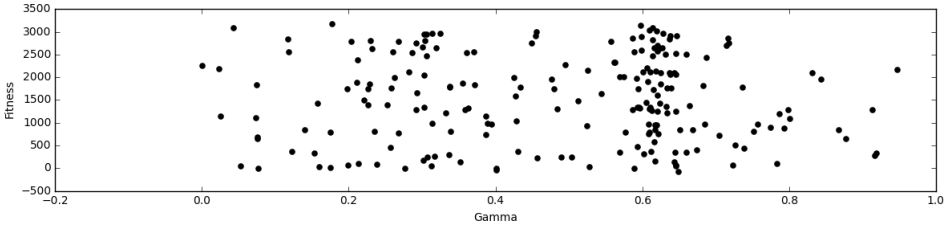
Metalearning of OSEGFVMC In Random State Init
GridWorld<GridCoords>



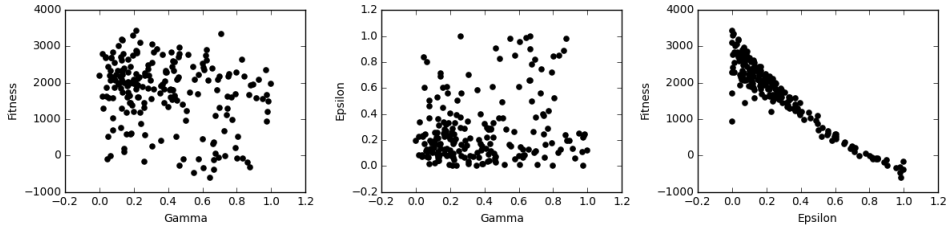
Metalearning of OSGEVMC In Random State Init
GridWorld<GridCoords>



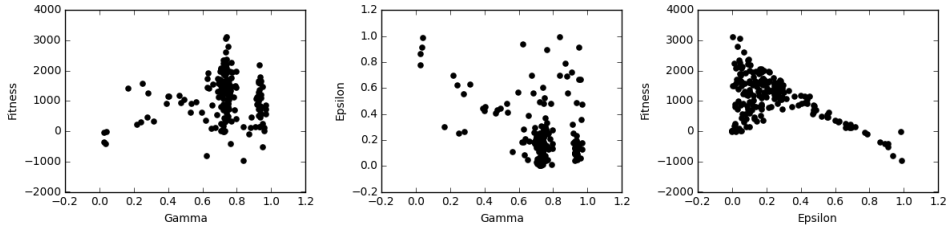
Metalearning of OSGFVMC In Random State Init
GridWorld<GridCoords>



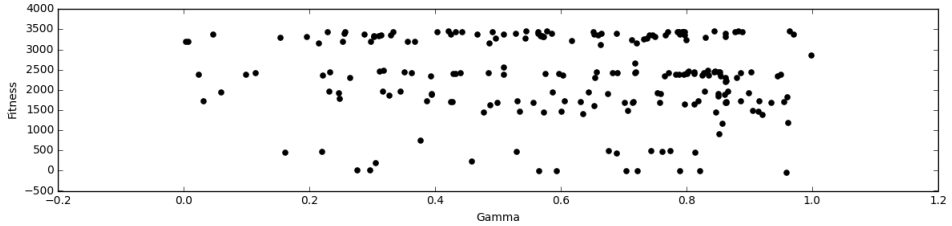
Metalearning of EGEVMC In Default State Init GridWorld<GridCoords>



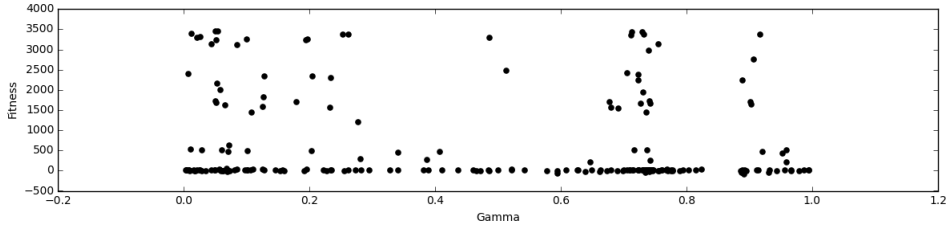
Metalearning of EGFVMC In Default State Init GridWorld<GridCoords>



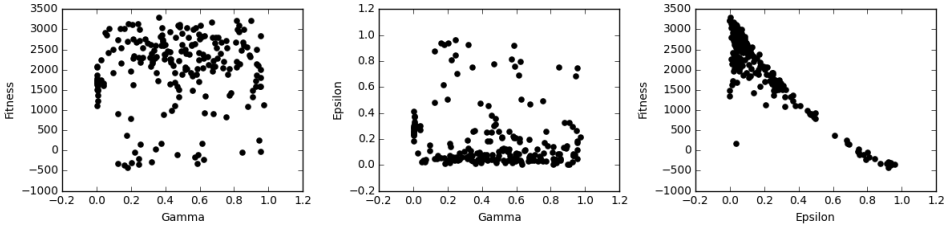
Metalearning of GEVMC In Default State Init GridWorld<GridCoords>



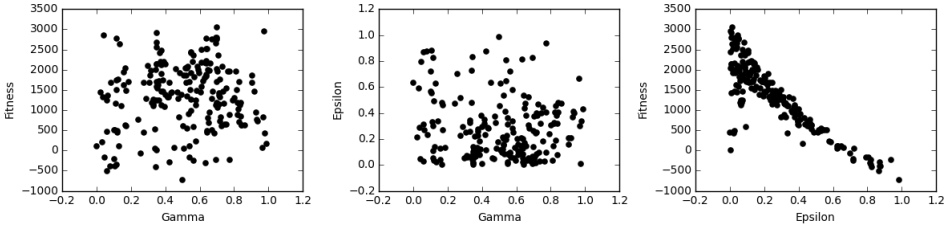
Metalearning of GFVMC In Default State Init GridWorld<GridCoords>



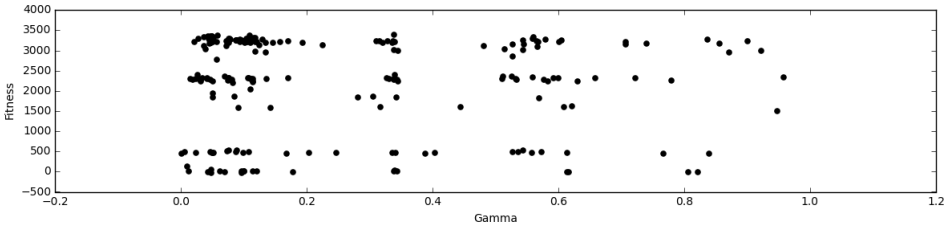
Metalearning of OSEGEVMC In Default State Init GridWorld<GridCoords>



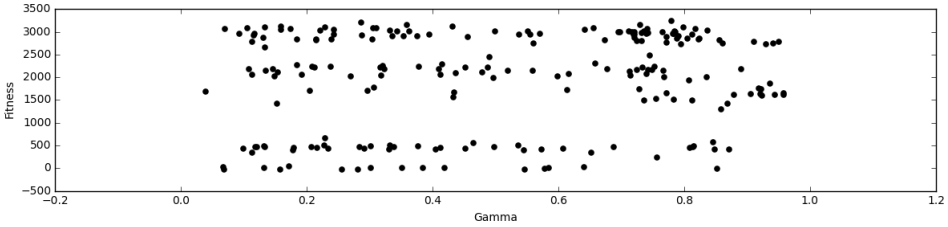
Metalearning of OSEGFVMC In Default State Init GridWorld<GridCoords>



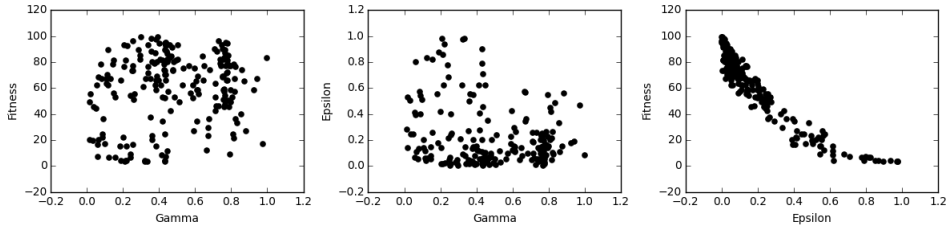
Metalearning of OSGEVMC In Default State Init GridWorld<GridCoords>



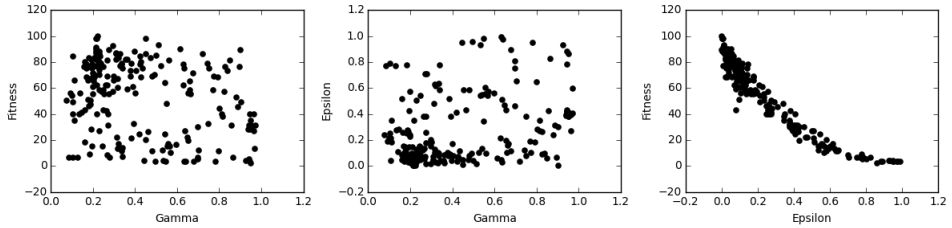
Metalearning of OSGFVMC In Default State Init GridWorld<GridCoords>



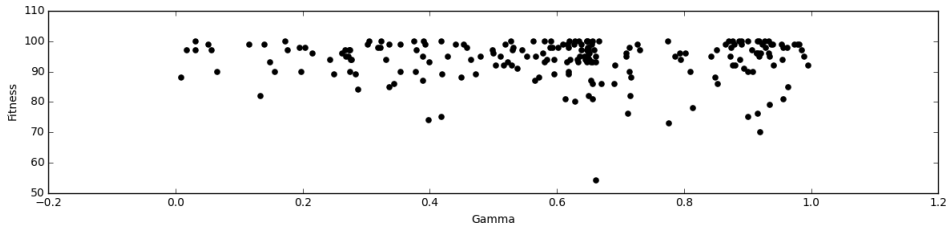
Metalearning of EGEVMC in Default State Init BreakOut<PaddleBallAndForce>



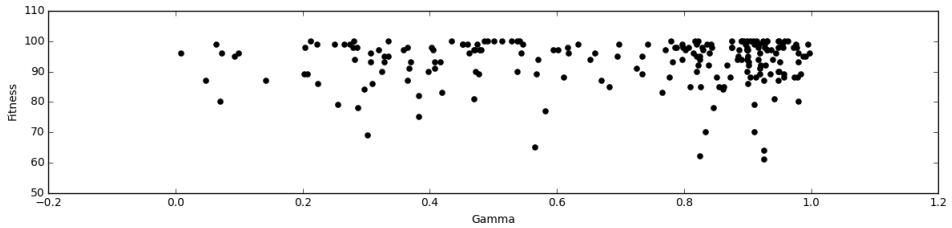
Metalearning of EGFVMC in Default State Init BreakOut<PaddleBallAndForce>



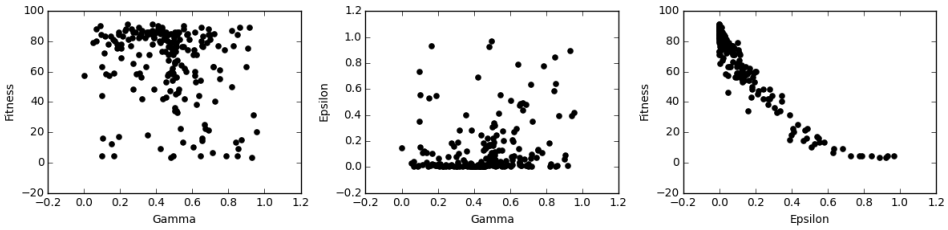
Metalearning of GEVMC in Default State Init BreakOut<PaddleBallAndForce>



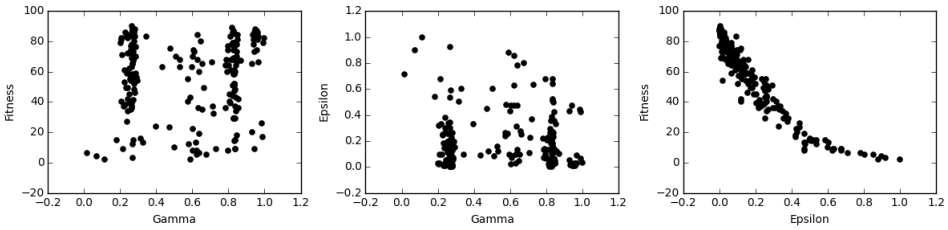
Metalearning of GFVMC in Default State Init BreakOut<PaddleBallAndForce>



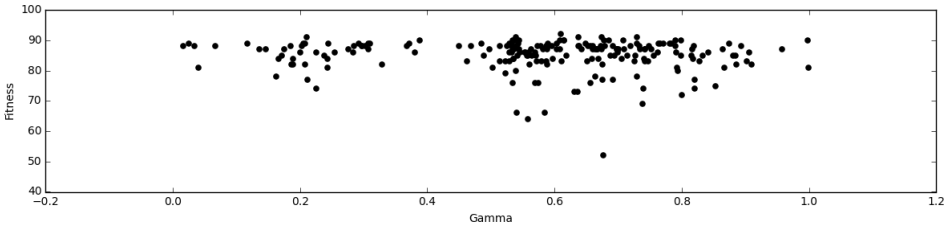
Metalearning of OSEGEVMC in Default State Init BreakOut<PaddleBallAndForce>



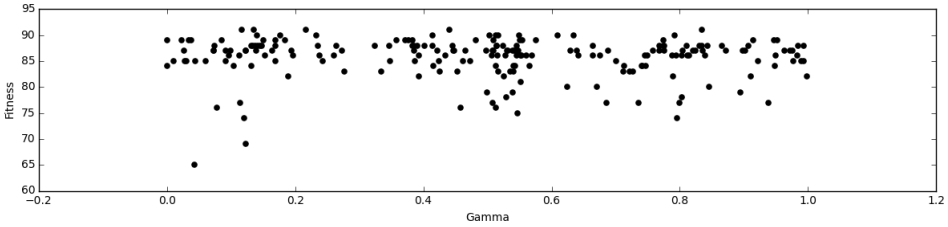
Metalearning of OSEGFVMC in Default State Init BreakOut<PaddleBallAndForce>



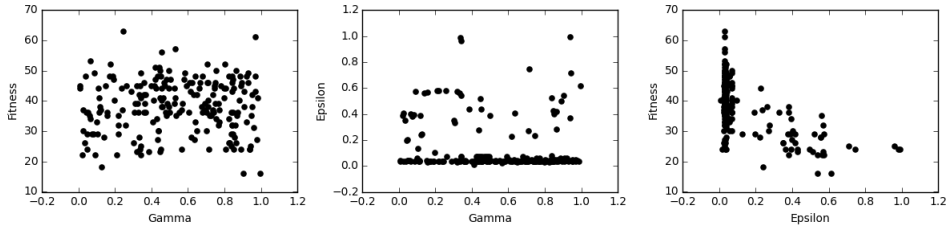
Metalearning of OSGEVMC in Default State Init BreakOut<PaddleBallAndForce>



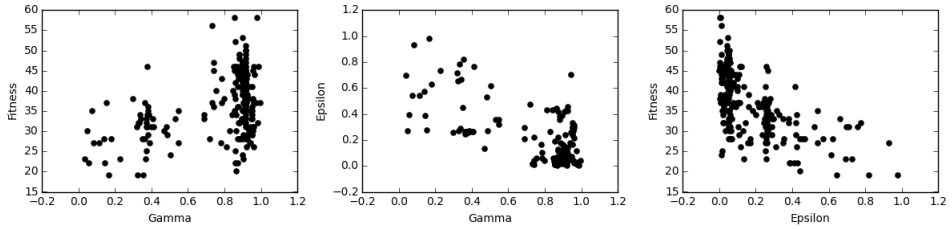
Metalearning of OSGFVMC in Default State Init BreakOut<PaddleBallAndForce>



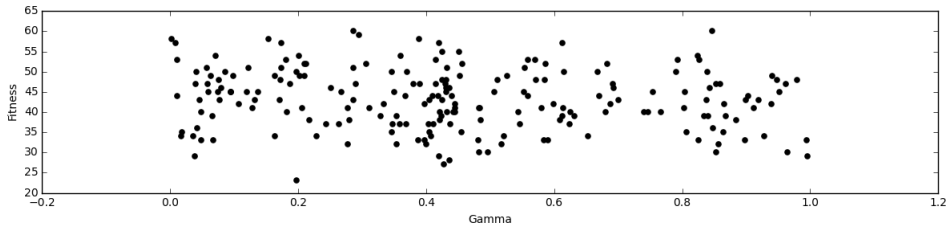
Metalearning of EGEVMC in Random State Init BreakOut<PaddleBallAndForce>



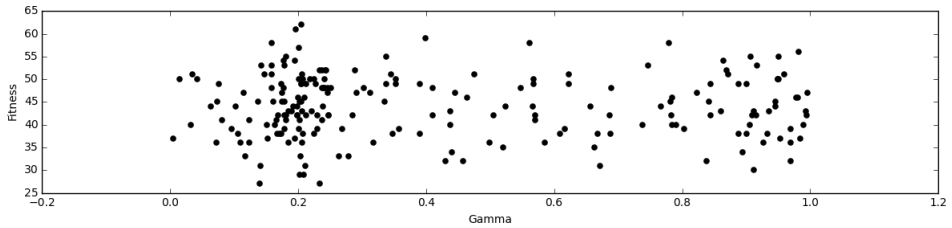
Metalearning of EGFVMC in Random State Init BreakOut<PaddleBallAndForce>



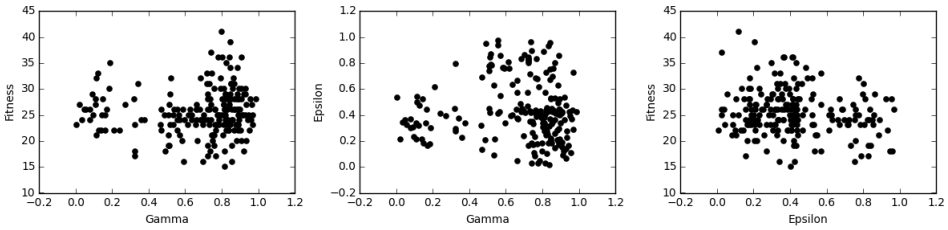
Metalearning of GEVMC in Random State Init BreakOut<PaddleBallAndForce>



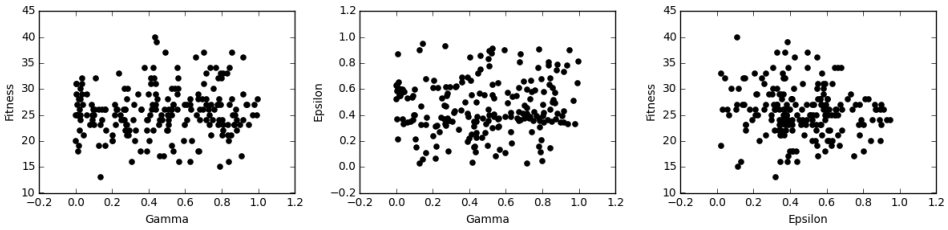
Metalearning of GFVMC in Random State Init BreakOut<PaddleBallAndForce>



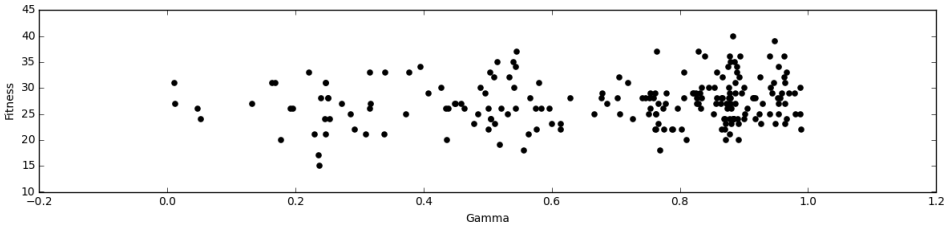
Metalearning of OSEGEVMC in Random State Init BreakOut<PaddleBallAndForce>



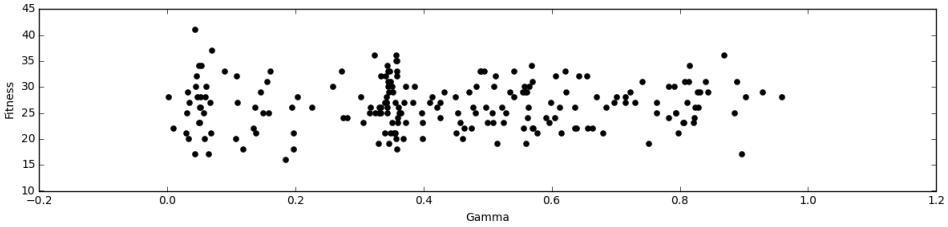
Metalearning of OSEGFVMC in Random State Init BreakOut<PaddleBallAndForce>



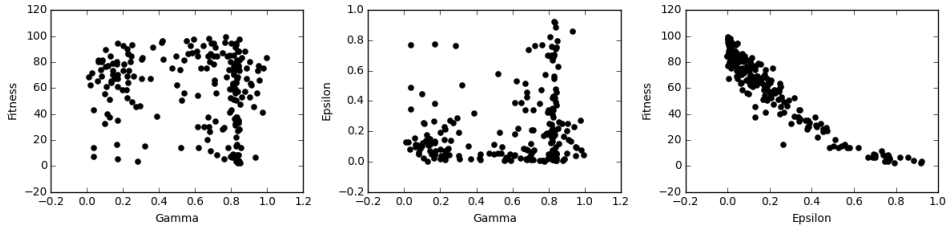
Metalearning of OSGEVMC in Random State Init BreakOut<PaddleBallAndForce>



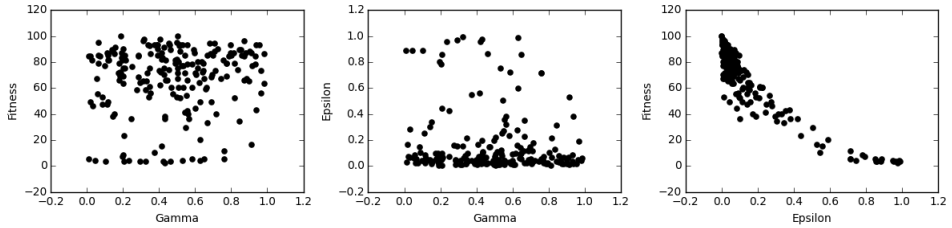
Metalearning of OSGFVMC in Random State Init BreakOut<PaddleBallAndForce>



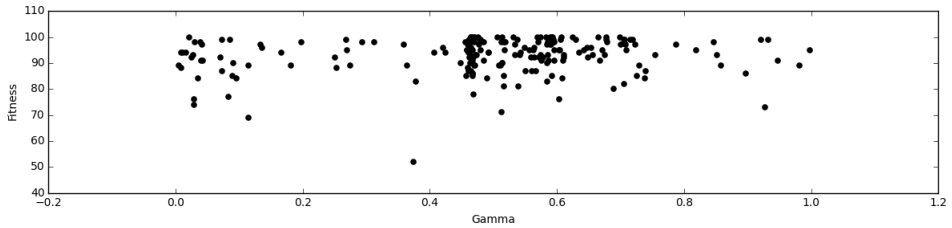
Metalearning of EGEVMC in Default State Init BreakOut<PaddleAndBall>



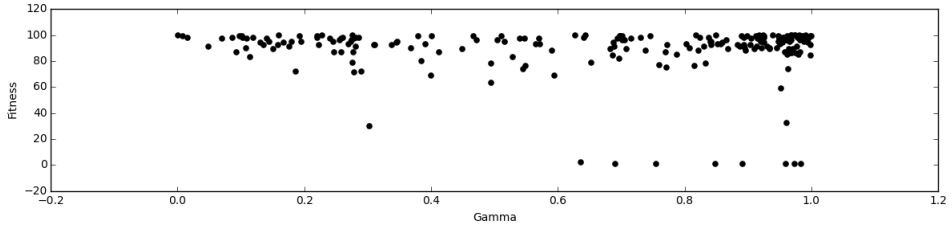
Metalearning of EGFVMC in Default State Init BreakOut<PaddleAndBall>



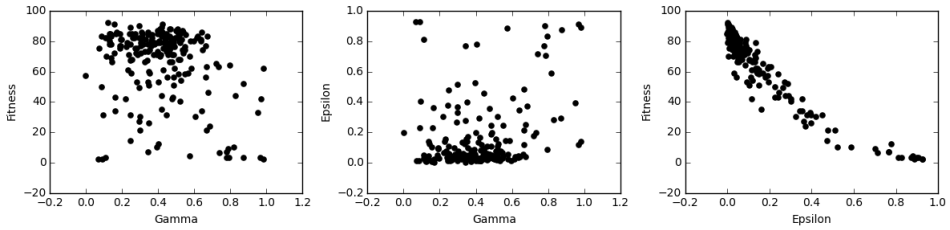
Metalearning of GEVMC in Default State Init BreakOut<PaddleAndBall>



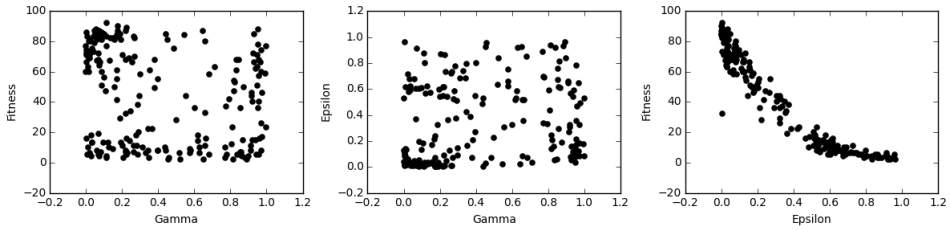
Metalearning of GFVMC in Default State Init BreakOut<PaddleAndBall>



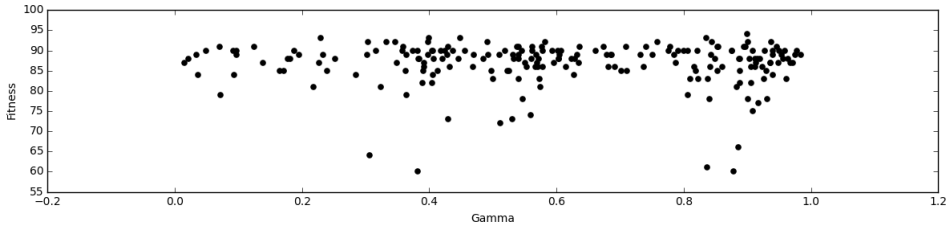
Metalearning of OSEGEVMC in Default State Init BreakOut<PaddleAndBall>



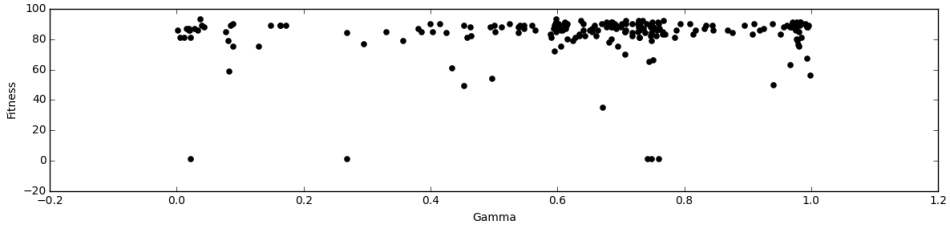
Metalearning of OSEGFVMC in Default State Init BreakOut<PaddleAndBall>



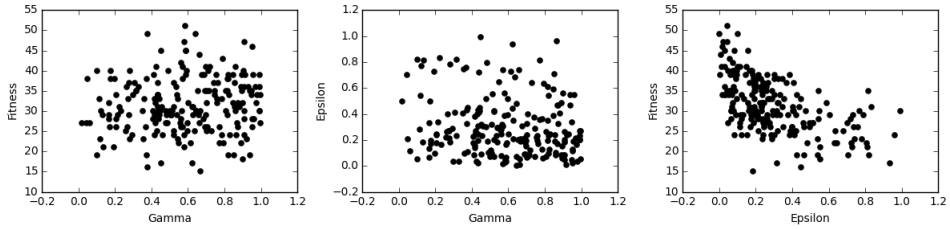
Metalearning of OSGEVMC in Default State Init BreakOut<PaddleAndBall>



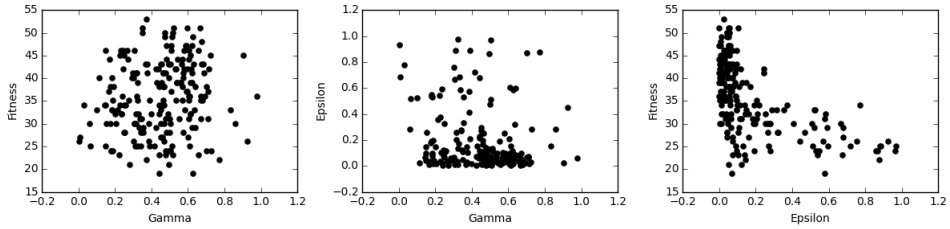
Metalearning of OSGFVMC in Default State Init BreakOut<PaddleAndBall>



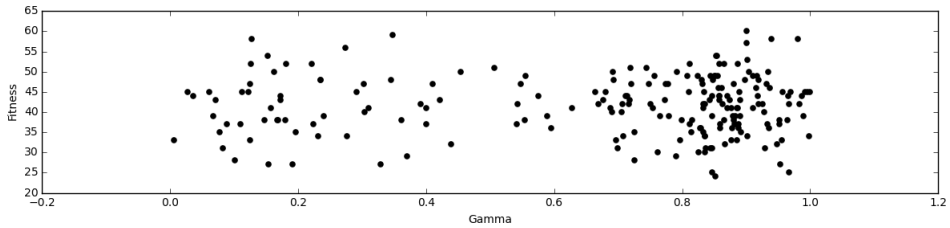
Metalearning of EGEVMC in Random State Init BreakOut<PaddleAndBall>



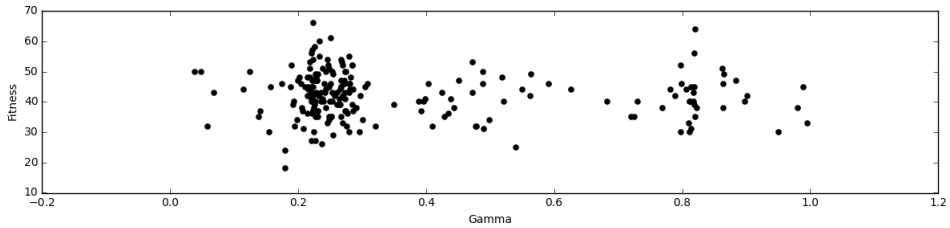
Metalearning of EGFVMC in Random State Init BreakOut<PaddleAndBall>



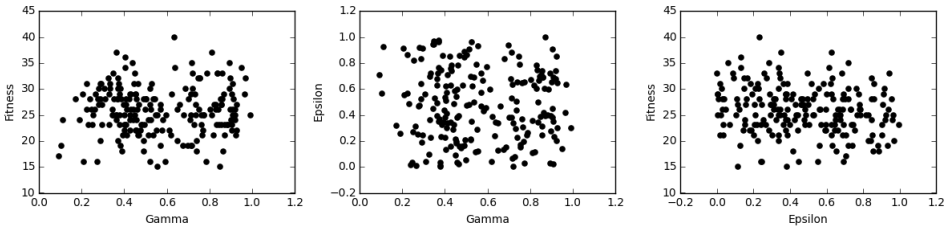
Metalearning of GEVMC in Random State Init BreakOut<PaddleAndBall>



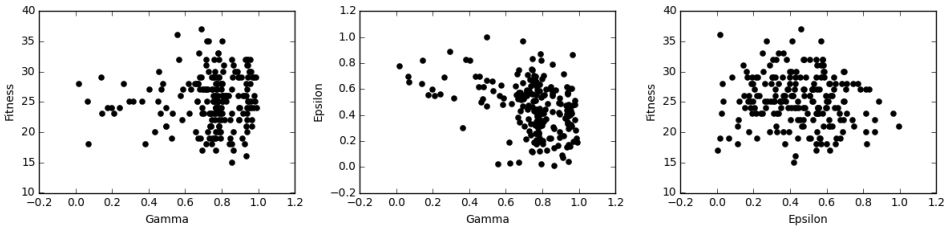
Metalearning of GFVMC in Random State Init BreakOut<PaddleAndBall>



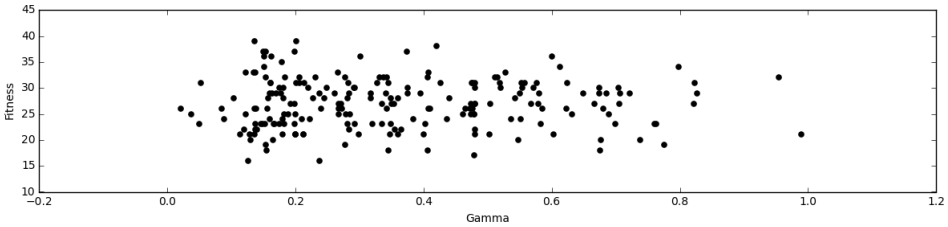
Metalearning of OSEGEVMC in Random State Init BreakOut<PaddleAndBall>



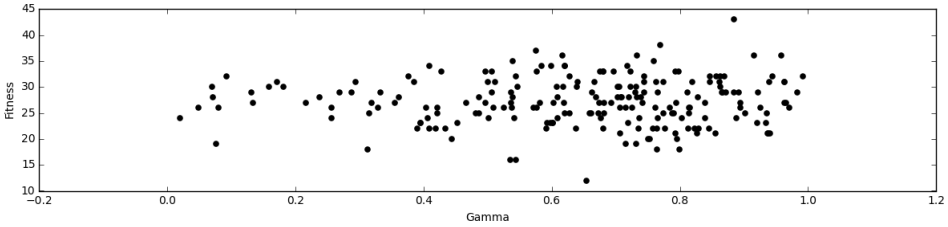
Metalearning of OSEGFVMC in Random State Init BreakOut<PaddleAndBall>



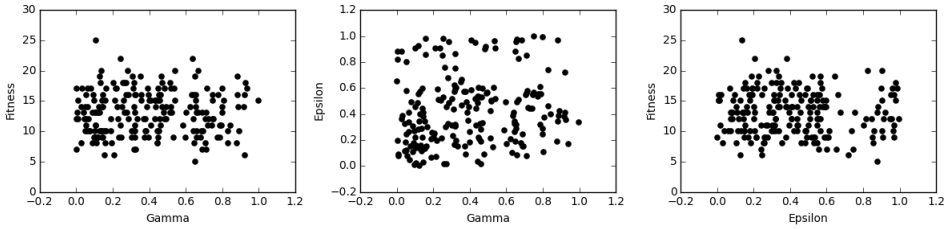
Metalearning of OSGEVMC in Random State Init BreakOut<PaddleAndBall>



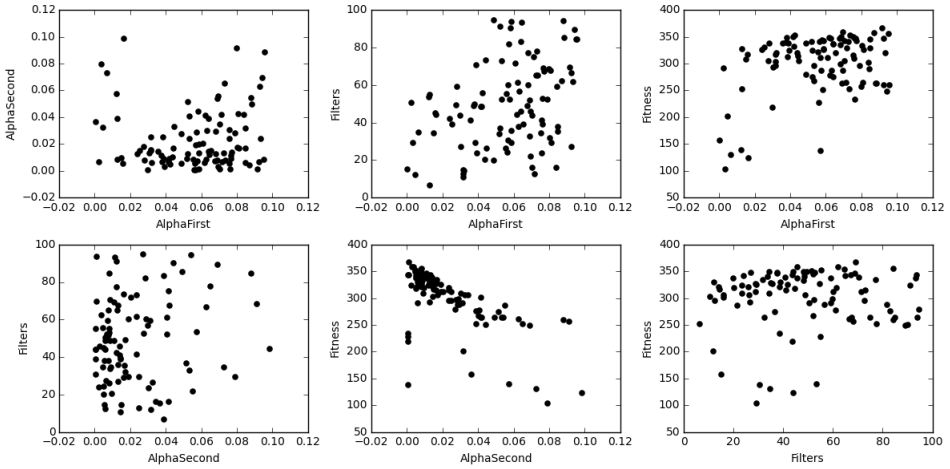
Metalearning of OSGFVMC in Random State Init BreakOut<PaddleAndBall>



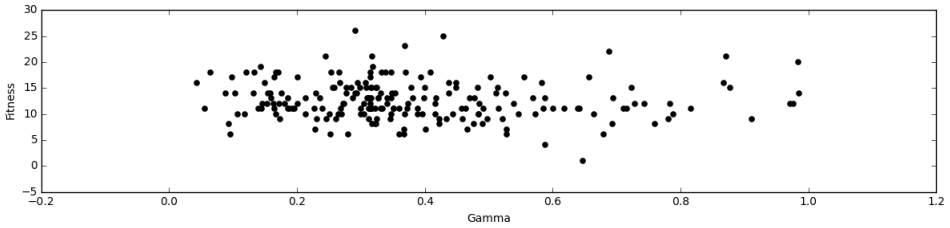
Metalearning of EGEVMC in Default State Init BreakOut<Pixels>



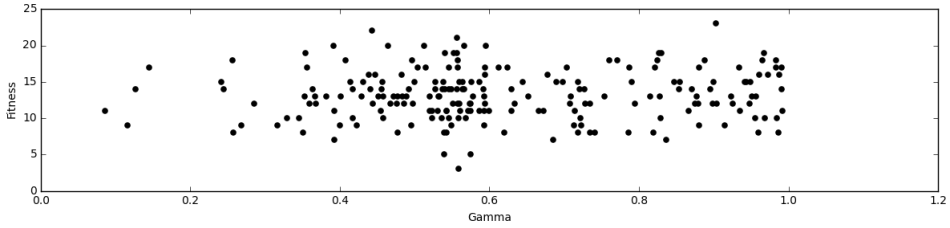
Metalearning of EGFVMC in Default State Init BreakOut<Pixels>



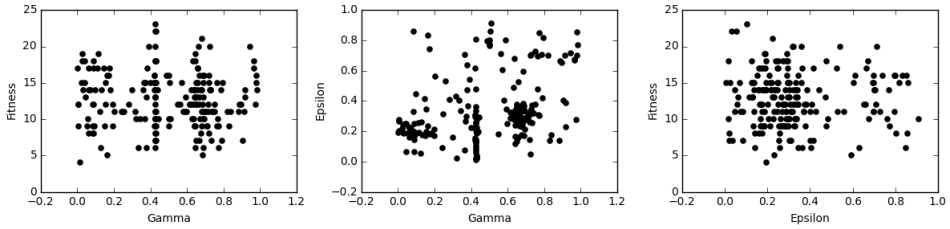
Metalearning of GEVMC in Default State Init BreakOut<Pixels>



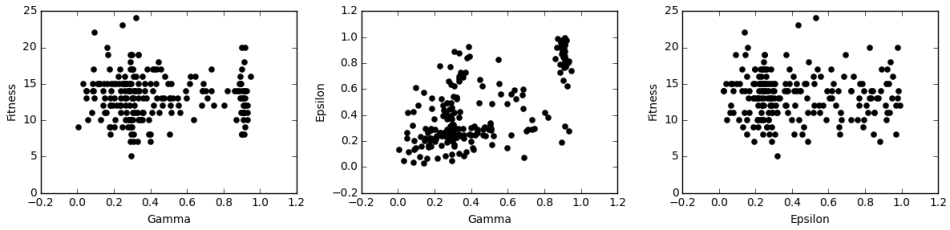
Metalearning of GFVMC in Default State Init BreakOut<Pixels>



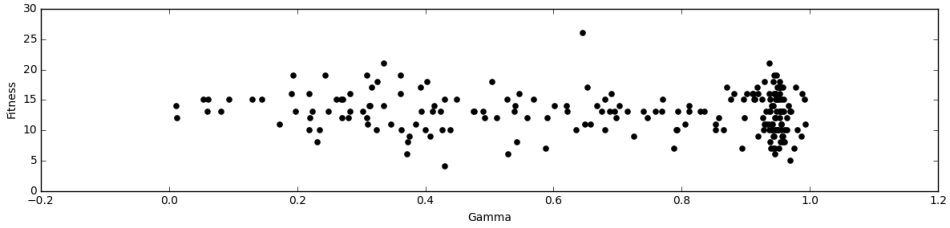
Metalearning of OSEGEVMC in Default State Init BreakOut<Pixels>



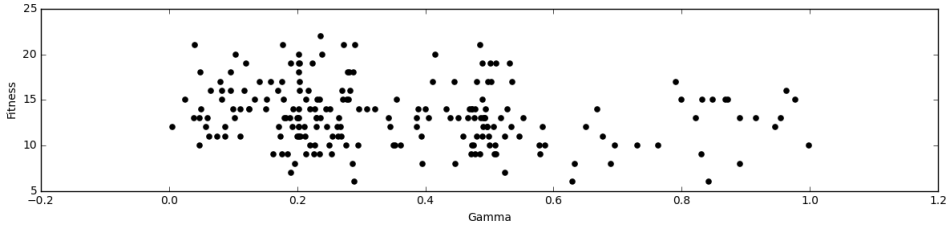
Metalearning of OSEGFVMC in Default State Init BreakOut<Pixels>



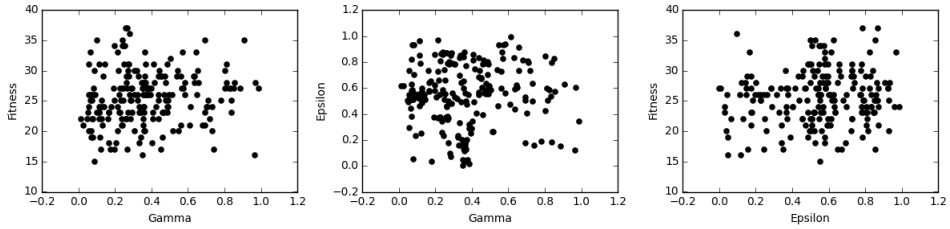
Metalearning of OSGEVMC in Default State Init BreakOut<Pixels>



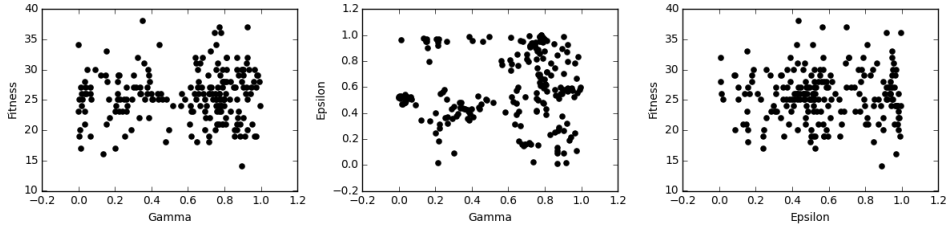
Metalearning of OSGFVMC in Default State Init BreakOut<Pixels>



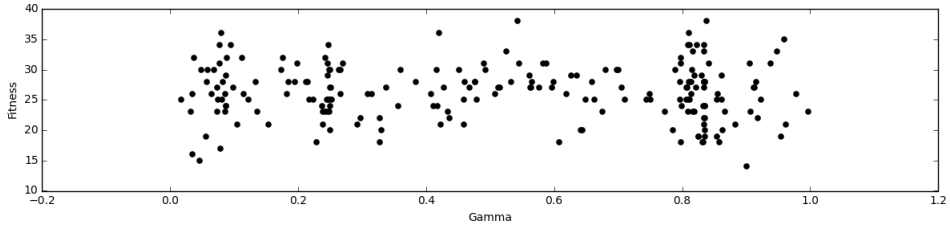
Metalearning of EGEVMC in Random State Init BreakOut<Pixels>



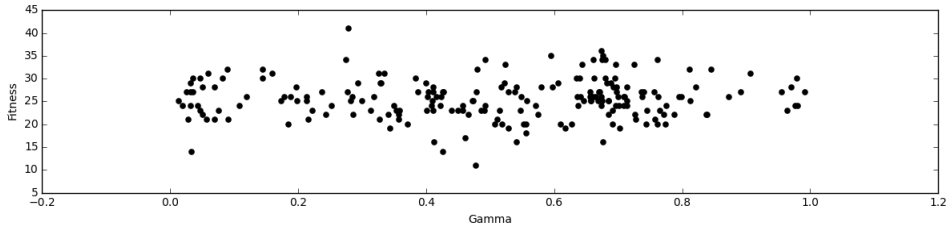
Metalearning of EGFVMC in Random State Init BreakOut<Pixels>



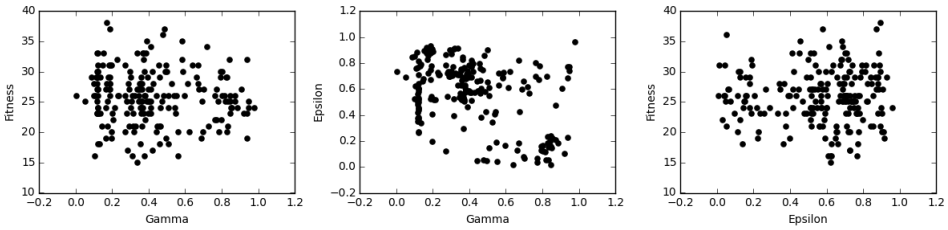
Metalearning of GEVMC in Random State Init BreakOut<Pixels>



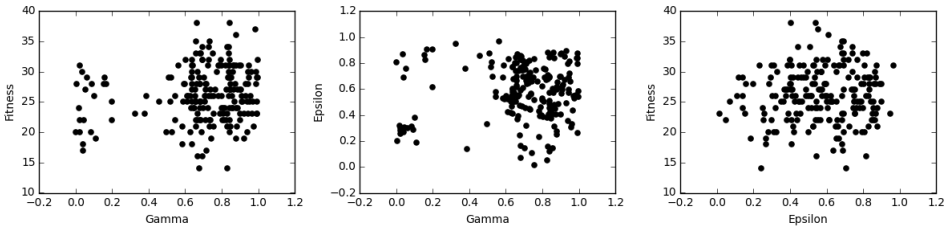
Metalearning of GFVMC in Random State Init BreakOut<Pixels>



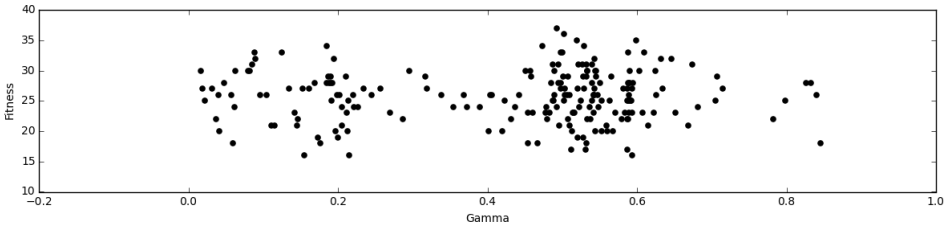
Metalearning of OSEGEVMC in Random State Init
BreakOut<Pixels>



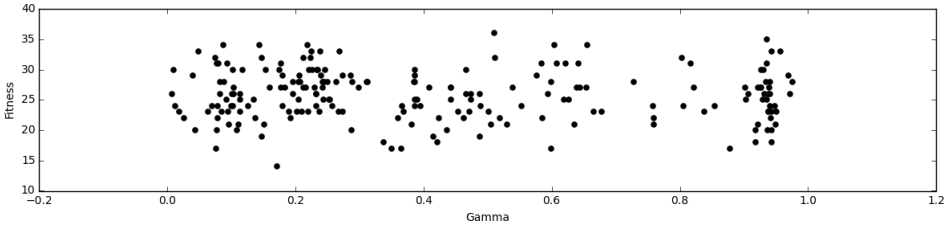
Metalearning of OSEGFVMC in Random State Init
BreakOut<Pixels>



Metalearning of OSGEVMC in Random State Init
BreakOut<Pixels>



Metalearning of OSGFVMC in Random State Init
BreakOut<Pixels>



Appendix C

Contents of enclosed CD

- `Thesis` ..... thesis directory with latex sources
  - `Graphics` ..... all graphics for thesis
- `RL` ..... visual studio 2015 project with sources
  - `RL` ..... source files and directories with experiments