Deep Reinforcement Learning

SoftServe Data Science Group

Ihor Kostiuk, Roman Grubnyk, Iurii Milovanov, Ievgenii Baliuk, Pavlo Kramarenko, Chris Poulin

DSG@softserveinc.com

v.02

Contents

1 Introduction

Although games are usually undertaken for enjoyment, they are also used as an educational tool or as a tool for simulating complex life situations for training in critical thinking and decision making. Now there are a great variety of games ranging from competitive sports, to board and video games, or even mind training focused games. Each game has its individual type, form, and structure.

While for much of their history, other humans were the logical opponent for games. But in early 1950s, when the Era of the Personal Computer came, this restriction was removed. The computer has become number-one opponent. Since then there was an enormous amount of competitions which tried to estimate a human brain's abilities in playing games in comparison to the state of the art computer algorithms. [1] [2] Although there was a great progress in developing such 'opponent agents' during the last years, the main goal of our project lies in the artificial intelligence of the 'human player simulating' game playing agent.

The DeepMind [3] team had recently proposed to use Convolution Neural Networks to extract features from raw data (without any hand written features). This new 'unsupervised learning' approach provides a huge flexibility and algorithm abilities increase in order to work on different data and solve different problems, with noteworthy performance.

With credit to the prior projects by Deepmind [4] , our Data Science Group has also decided to develop a solution which would be able to achieve at or better performance in playing video games with computer than an expert human player. Our main goal is to completely generalize this idea and make it able to be applied to any video game. This particular use case could become a breakthrough in the study of Artificial Intelligence.

2 Theory

At this time, we are not about to introduce any new mathematical ideas or concepts in our project. Rather we want to put together currently existing state-of-the-art approaches and techniques to achieve a higher goal by combining them. In this chapter we will briefly explain the main underlying concepts of our solution such as Reinforcement Learning and Neural Networks, to let you understand 'how it actually works'. [Note: There will be a great deal of math and other process descriptions, so feel free to skip this section and jump directly to Implementation (or even Getting Started with DRL) if you're already familiar with these concepts.]

2.1 Reinforcement Learning

Reinforcement Learning (RL) [5] is the set of algorithms (or approaches) inspired by behaviorist psychology. The idea behind it is that a computer algorithm (hereafter, agent) interacts with some unknown or semi-known environment (may be stochastic) using predefined and limited set of actions in a way that maximize some cumulative reward. The environment is typically formulated in terms of Markov Decision Process (MDP) [6]. It defines an environment ε as a two finite sets:

- $S = \{s_1, \ldots, s_n\}$ – set of environment internal states;

- $A = \{a_1, \ldots, a_k\}$ – set of environment legal actions;

---

[1] http://en.wikipedia.org/wiki/Watson_(computer)#Jeopardy.21

[2] http://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov

[3] http://deepmind.com/

[4] http://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[5] http://en.wikipedia.org/wiki/Reinforcement_learning

[6] http://en.wikipedia.org/wiki/Markov_decision_process

- $\delta : S \times A \to S$ – transition function (usually unknown);

An agent interacts with environment ε in discrete time steps. On each step t in a state s_t it chooses an action $a_t \in A$ which is subsequently sent as a command to the environment. The environment changes agent's state from s_t to s_{t+1}. As a result of transition $[s_t, a_t, s_{t+1}]$ (means: from state s_t to state s_{t+1} by taking action a_t) agent immediately receives a reward r_{t+1}. The reward given by environment often is discounted by some so-called discount-factor $0 \le \gamma \le 1$. Thus we can define an expected discounted return after game termination by:

$$R = \sum_{t=0}^{N} \gamma^t r_{t+1} \tag{1}$$

here, N is the game termination time.

The agent makes its decisions on action to be taken in order to maximize an expected reward by following some action-value function which calculates quality of a state-action pair with respect to the past experience:

$$Q : S \times A \to R \tag{2}$$

Based on Bellman equation of optimality, we can define the optimal action-value function $Q^*(s, a)$ as the maximum expected reward for the state-action pair (s, a) achievable by taking action a in state s and thereafter following an optimal strategy by maximizing the expected value of $r + \gamma Q^*(s', a')$:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \tag{3}$$

The problem of optimal strategy determination, based on formulated optimization criterion, is referred to the dynamic programming problems. There are two general approaches:

- Iteration by strategy

- Iteration by values

Both approaches are computer-intensive, especially when power of possible states and action sets is significant enough. It explains inappropriateness of using traditional algorithms, when Markov Decision Process is reduced to hidden Markov model. In this case we can apply some approximate approaches, and one of the most successful solutions is applying Q-learning approximation by using apparatus of artificial neural networks.

The agent follows a greedy strategy on action selection $a = \max_a Q(s, a)$. However it can randomize its selection to explore states that have not been visited yet. Thus we define the ϵ-greedy strategy that follows the greedy strategy with probability $1 - -\epsilon$ and selects a random action with probability ϵ.

2.2 Neural Networks

Artificial Neural Networks (ANNs) are mathematical models inspired by biological neural networks and are limited imitation of human's central nervous system (in particular the brain). In Machine Learning [7] they are used as an approach to approximate functions $f : X \to Y$ (usually non-linear) by "learning" it from large amount of input data.

2.2.1 Architecture

There is a variety of approaches to build Neural Network topology, but in our project we consider only feedforward approach [8]. It contains following basic elements:

- Nodes (known as neurons) connected together by directed links to form a network;

- Connections between neurons associated with weights;

- Layers composed of neurons;

---

[7] http://en.wikipedia.org/wiki/Machine_learning
[8] http://en.wikipedia.org/wiki/Feedforward_neural_network

A connection from neuron i to neuron j propagates the activation a from i to j. Each connection also has a numeric weight $w_{i,j}$ associated with it, which determines the strength of the link. Each neuron j, in order to calculate its activation value a_j, first computes a weighted sum of all of its inputs, and then applies an activation function g to it:

$$a_j = g(\sum_{i=0}^{n} w_{i,j} a_i) \tag{4}$$

The activation function could be:

- sigmoid function [9].

- linear function [10].

- other advanced techniques (such as SoftMax, MaxOut etc)

Neural Networks usually are arranged in multiple layers of nodes in a directed graph, with each layer fully connected to the next one. The first one usually is called as Input layer and the final layer the Output layer. Intermediate layers of nodes between the Input and Output layers called the Hidden layer. The activation values on input layer represent the data input and the output layer activations represent the result value. The hidden layers are responsible for intermediate calculations that fit to the underlying data structure. The number of neurons in each layer depends on the amount of input data and the type (or complexity) of problem being solved.
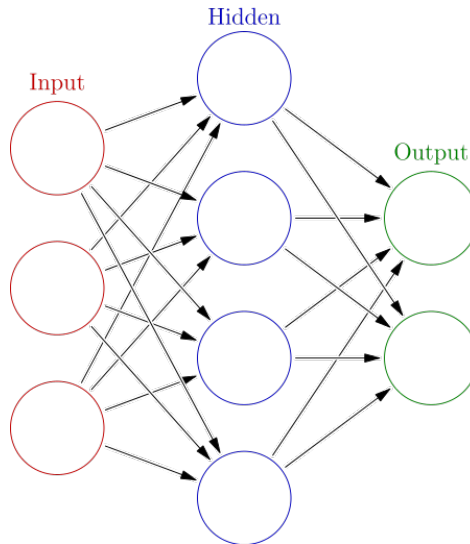

Figure 1: Neural network

2.2.2 Training

In Neural Networks learning occurs in changing connection weights by minimizing the Loss function. This is an example of supervised learning, and is carried out through backpropagation technique [11] in conjunction with an optimization method such as gradient descent.

2.3 Deep Learning

Deep learning is an area of machine learning which is composed of a set of algorithms and techniques that attempt to define the underlying dependencies in a data and to model its high-level abstractions. The main goal of this approach is to avoid manual description of a data structure (like hand-written features) by automatic learning it from the data. Its name refers to the fact that typically any Neural Network with two or more hidden layers is called deep neural network. Deep Learning techniques usually

---

[9] http://en.wikipedia.org/wiki/Sigmoid_function
[10] http://en.wikipedia.org/wiki/Linear_function
[11] http://en.wikipedia.org/wiki/Backpropagation

are applied to the tasks on a data which is not easy to represent as a valuable set of features because of its complexity (such as image or audio signal). Thus its main field of application lies in computer vision and speech recognition.

2.3.1 Convolutional Neural Networks

Convolutional neural network is a deep learning technique which is now widely used in image recognition. Architecturally, it is a type of feed-forward artificial neural network where the individual neurons are tiled in such a way that they respond to overlapping regions in the visual field. [12]
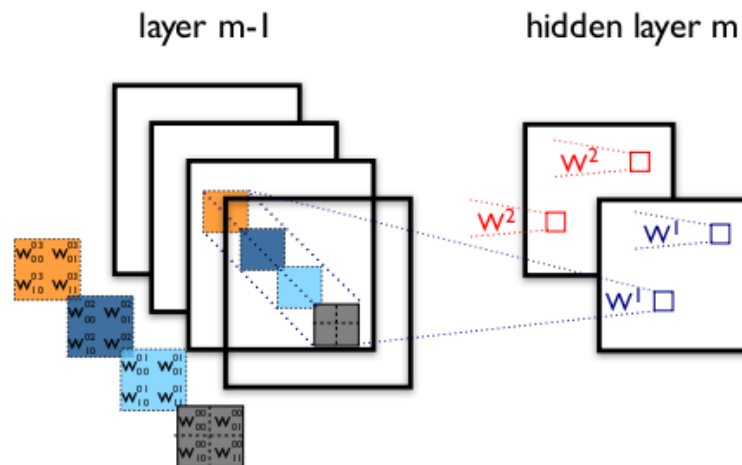

Figure 2: Convolutional neural network

Convolutional neural networks consists of a combination of multiple hidden layers of small cells of neurons which look specifically at small frame of the input image. The results of these cells are then tiled so that they overlap to obtain a better representation of the original image; this is repeated for every such layer. Because of this, they are able to tolerate translation of the input image. [13] These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images.

Convolutional neural networks also implement a few important concepts (such as Shared Weights [14] and MaxPooling [15]) that bring a big advantage in a field of Deep Learning and Computer Vision.
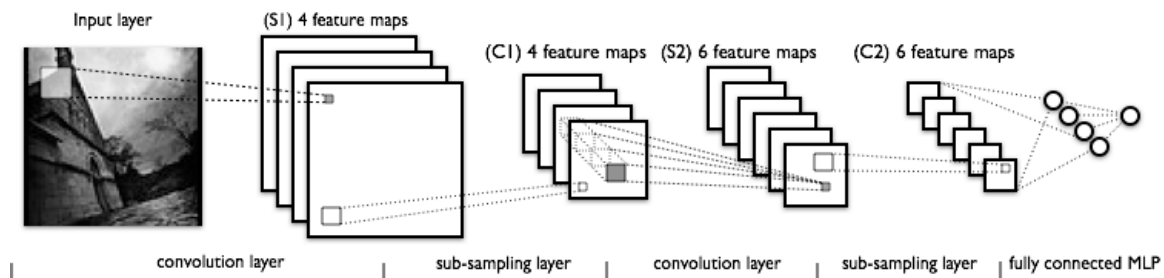

Figure 3: Full-stack model example

---

[12] http://en.wikipedia.org/wiki/Convolutional_neural_network#cite_note-deeplearning-1

[13] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.125.3812

[14] http://deeplearning.net/tutorial/lenet.html#shared-weights

[15] http://deeplearning.net/tutorial/lenet.html#maxpooling

3 Implementation

3.1 Process description

3.1.1 Explanation

The Diagram below (Figure 4) illustrates the flow of 'train and predict' functionality of the neural network. The blocks, marked by blue color and numbering are suggestions for future improvement. Meanwhile, there are also explanatory notes with benefits of each, with links for corresponding papers in chapter 3.8.
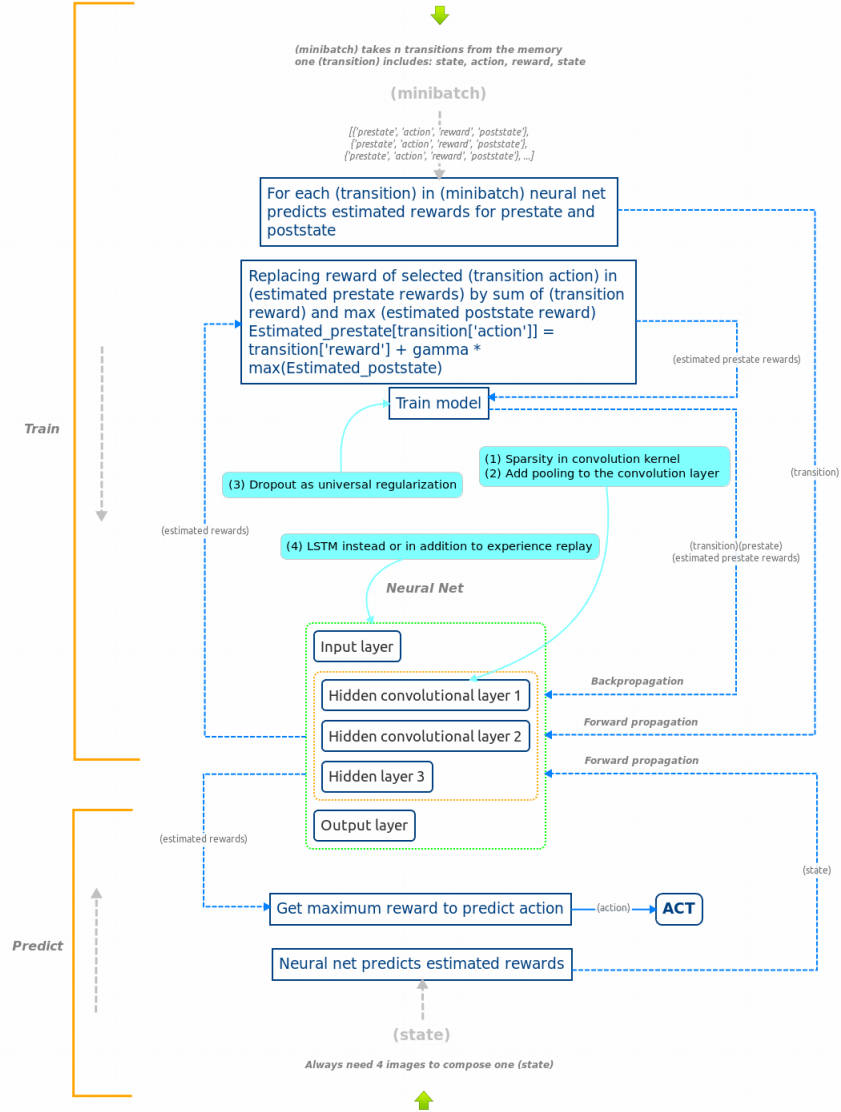

Figure 4: Learning diagram

3.1.2 Prediction

See Figure 4 where the network is designed to predict future rewards for all possible actions. There is a number of neurons that correspond to number of possible action in a game on the output layer (For Breakout there are four possible actions: no action, up (it's odd but possible), right, left).

- To make prediction we forward propagate the net with data describing current state at input and get activations of output layer.

- Then we use a greedy policy: define the neuron with maximum activation and perform corresponding actions.

For example, if the third neuron on output layer has a maximum activation we perform the third action. In diagram (Figure 4) this part of functionality starts from bottom to top. First block after bottom - forward propagation. And that one above this – illustrates a greedy policy algorithm.

3.1.3 Algorithm

Following algorithm was chosen for implementation in code (Figure 5).
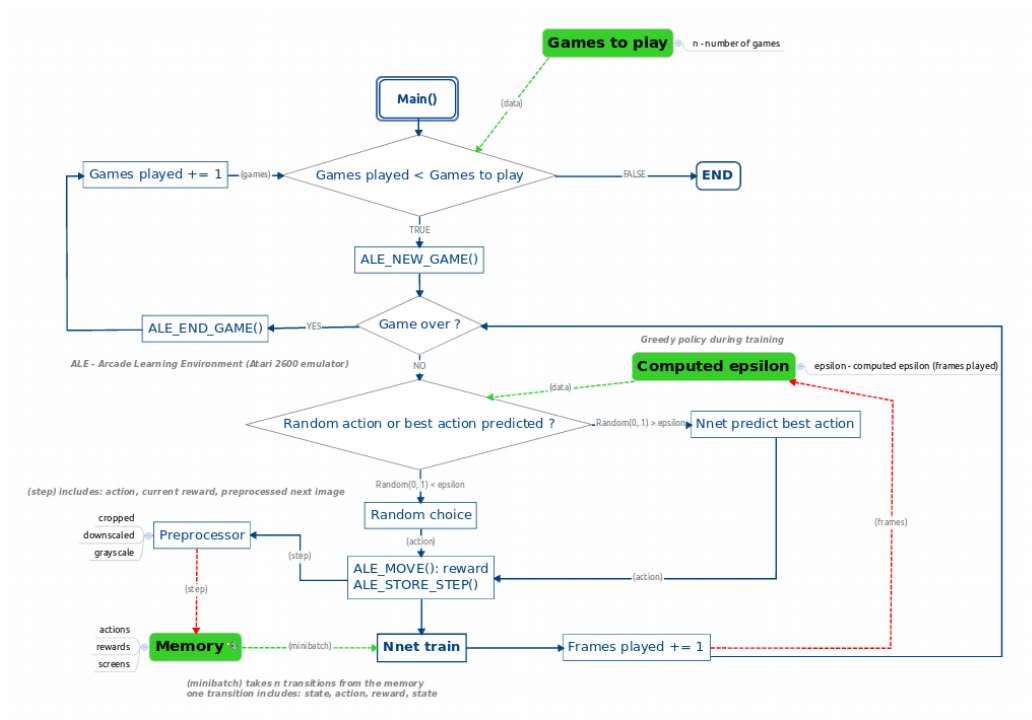

Figure 5: Implementation algorithm

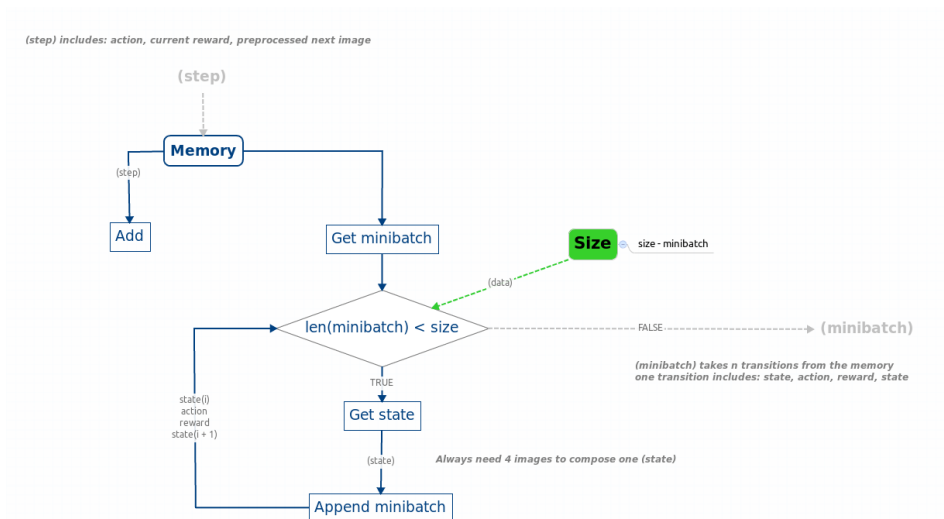Implementation of the memory model is relatively simple (Figure 6).

Figure 6: Memory model

3.2 Wrappers

Our library (DRL) provides a set of wrappers, each of these connects to appropriate game environment (Atari, Nintendo and others) using available software interfaces. The wrapper interacts with environment by sending it a commands to perform an actions or to obtain the necessary information about environment internal state (such as game reward, termination state etc). It also receives a video output from environment and processes it sequentially as an image stream. The wrapper is the only game-specific and non-generalized component in our solution. General diagram of DSG technology stack is provided on Figure 7.

3.2.1 List of the supported wrappers

Name	Type	Interface	Python class	Environments	Description
atari	Emulator	FIFO (text file)	ale.ALE	• Breakout • Space Invaders	The Arcade Learning Environment (ALE) is a simple object-oriented framework that allows to develop AI agents for Atari 2600 games [16].

3.3 TNNF Framework

Our Deep Learning implementation is based on TNNF framework which is also developed by our group. Please refer to the project documentation page [17] to get further information.
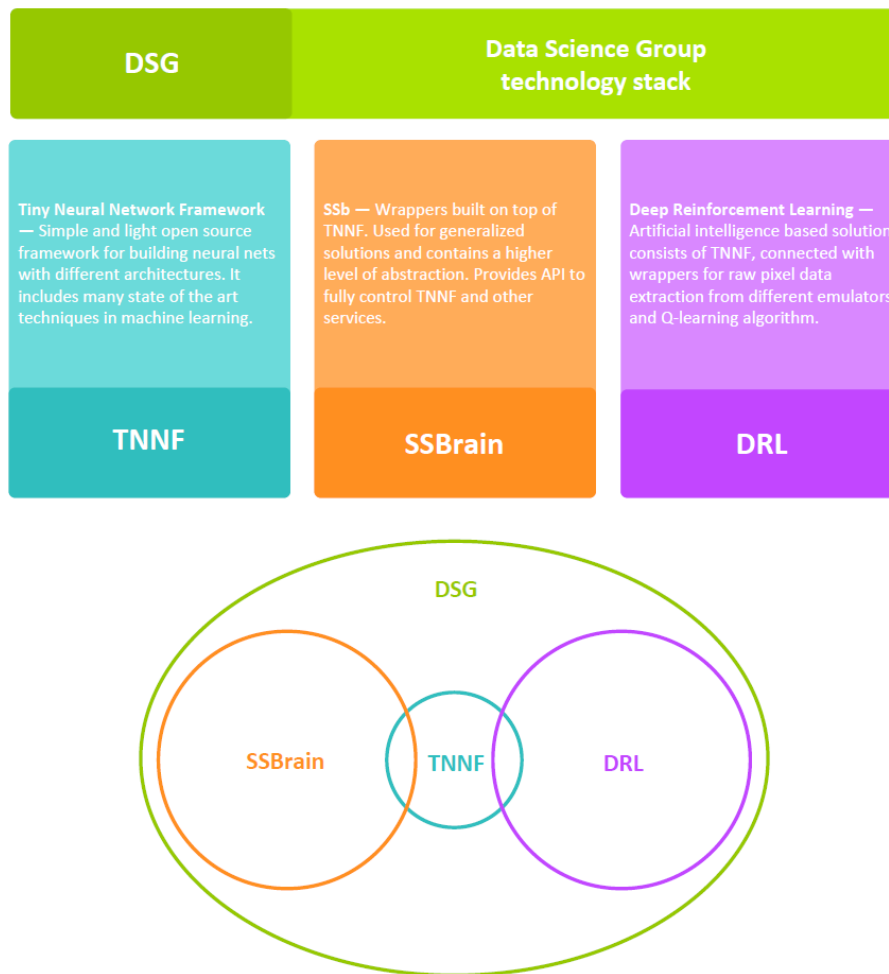

Figure 7: DSG technology stack

---

[16] http://www.arcadelearningenvironment.org/
[17] http://tnnf.readthedocs.org/en/latest

3.4 GPU Computing

Great progress achieved in last 10 years in the area of general-purpose computing on GPUs [18] [19] also brought a great benefit to the field of Artificial Intelligence (and to the Deep Learning in particular [20]) since its incredible ability to massively parallel computation provides an exponential increase in performance (up to 50 times faster) on linear algebra operations that are so resource intensive with traditional CPU-based approach, especially on a large datasets.

There are two main frameworks that implements this computational model:

- NVIDIA CUDA (proprietary) [21]

- OpenCL (open source) [22]

The neural network framework (TNNF) we use in this project, is built on the top of the Theano library [23] which in turn uses CUDA to perform GPU-based computations. Please refer to the project documentation [24] for more information on how to install and configure CUDA-backend in case if your GPU device is supported by vendor [25].

Our solution uses all advantages of this approach, however it still could be used in a traditional way of CPU computing (see Configuration)

3.5 MDP Environment

In our project, we consider the following MDP environment ε:

- Raw screen images from the game video output represent $s \in S$;

- Available game actions obviously represent set $A = \{a_1, \ldots, a_k\}$;

- Game reward obtained through API represents r_t;

3.6 Preprocessing

We perform a basic preprocessing step on each of raw 210×160 RGB images from input video stream. It consists:

- cropping a square 160×160 region of original image;

- downscaling to 84×84 image;

- converting RGB to gray-scale or B/W representation;

3.7 Neural Network

We use a Neural Network in conjunction with advanced Deep Learning techniques as a action-value function approximator. We refer to it as a DRL-network with weights θ which estimates $Q(s, a; \theta)$.

3.7.1 Network topology

DRL-network architecture consists of the following layers:

---

[18] http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
[19] http://www.nvidia.com/object/what-is-gpu-computing.html
[20] http://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/
[21] http://en.wikipedia.org/wiki/CUDA
[22] http://en.wikipedia.org/wiki/OpenCL
[23] http://deeplearning.net/software/theano/
[24] http://deeplearning.net/software/theano/tutorial/using_gpu.html
[25] http://en.wikipedia.org/wiki/CUDA#Supported_GPUs

1. Input layer (L1)

   - Layer type: Convolutional
   - Activation function: Leaky Rectified Linear Unit [26]
   - Number of inputs: 28224
   - Number of outputs: 6400
   - Convolutional filter shape: $16 \times 4 \times 8 \times 8$

2. Hidden layer (L2)

   - Layer type: Convolutional
   - Activation function: Leaky Rectified Linear Unit
   - Number of inputs: 6400
   - Number of outputs: 2592
   - Convolutional filter shape: $32 \times 16 \times 4 \times 4$

3. Hidden layer (L3)

   - Layer type: Fully connected
   - Activation function: Rectified Linear Unit [27]
   - Number of inputs: 2592
   - Number of outputs: 256

4. Output layer (L4)

   - Layer type: Fully connected
   - Activation function: Linear function
   - Number of inputs: 256
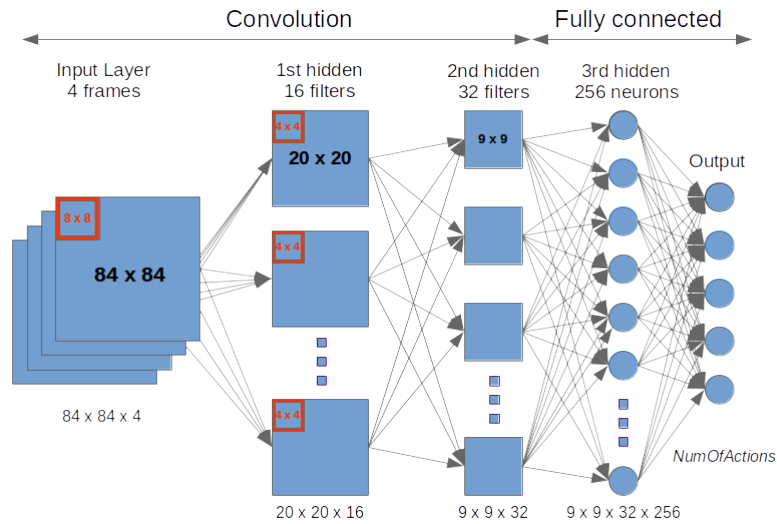   - Number of outputs: *depends on a number of actions in particular game environment*

Figure 8: DRL Neural Network

---

[26]http://en.wikipedia.org/wiki/Rectifier_(neural_networks)#Leaky_ReLUs
[27]http://en.wikipedia.org/wiki/Rectifier_(neural_networks)

3.7.2 Network model

The following functions are used:

$$C'_p(i,j) = ln(1 + exp(\sum_{z=-3}^{4} \sum_{v=-3}^{4} (h'_{pzv} \times S_u(2(i-3)+1+z, 2(j-3)+1+z)))) \tag{5}$$

$$C''_t(i,j) = ln(1 + max(\sum_{z=-1}^{2} \sum_{v=-1}^{2} (h''_{tzv} \times C'_p(2(i-1)+1+z, 2(j-1)+1+z)), 0)) \tag{6}$$

$$F_k = ln(1 + max(\sum_{t=1}^{32} \sum_{i=1}^{9} \sum_{j=1}^{9} (w^C_{tijk} \times C''_t(i,j)), 0)) \tag{7}$$

$$Q_r = \sum_{k=1}^{256} (w^F_{kr} \times F_k) \tag{8}$$

where

$$t = (p-1) \times 2 + q_2; \quad p = (u-1) \times 4 + q_1; \quad q_1 = \overline{1,4}; \quad q_2 = \overline{1,2}; \quad u = \overline{1,4} \tag{9}$$

S — input layer images
C' — first convolutional layer images
C'' — second convolutional layer images
F — hidden fully connected layer
Q — output layer

3.7.3 Training

A DRL-network is trained by minimizing a loss functions $L_i(\theta_i)$ on each iteration i, using RMSProp algorithm with mini-batches of size 128 samples:

$$L_i(\theta_i) = (y_i - Q(s,a,\theta_i))^2 \tag{10}$$

Here, $y = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a]$ is the target return for iteration i.

Differentiating the loss function with respect to the weights we get the following gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a, \theta_i)\right] \nabla_{\theta_i} Q(s, a, \theta_i) \tag{11}$$

In terms of MDP, on each time-step t in states s_t, s'_{t+1} we divide environment timeline (Figure 9) into following parts:

- Pre-state - set of the last 4 image before state s_t;

- Post-state - set of the last 4 image before state s'_{t+1}, when an action a_t has already been applied;
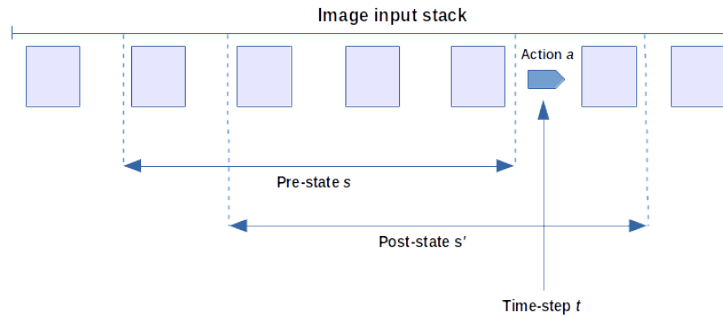

Figure 9: Timeline illustration

On each game iteration i, agent receives from environment wrapper a pre-state sequence s. Then it selects an action to be performed by following an ϵ-greedy strategy and using DRL-network to estimate $\max_a Q(s, a; \theta_i)$. Subsequently agent moves into post-state s' and gets a reward r. Reward takes value from set $\{-1, 0, 1\}$, where

- 1 - success (ball broke the brick)
- 0 - no action is required
- -1 - failure (ball was lost)

The sequence (s, a, r, s') forms a transition unit and is used to train DRL-network by estimating $y = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a]$. Complete training process workflow is reflected on Figure 10.
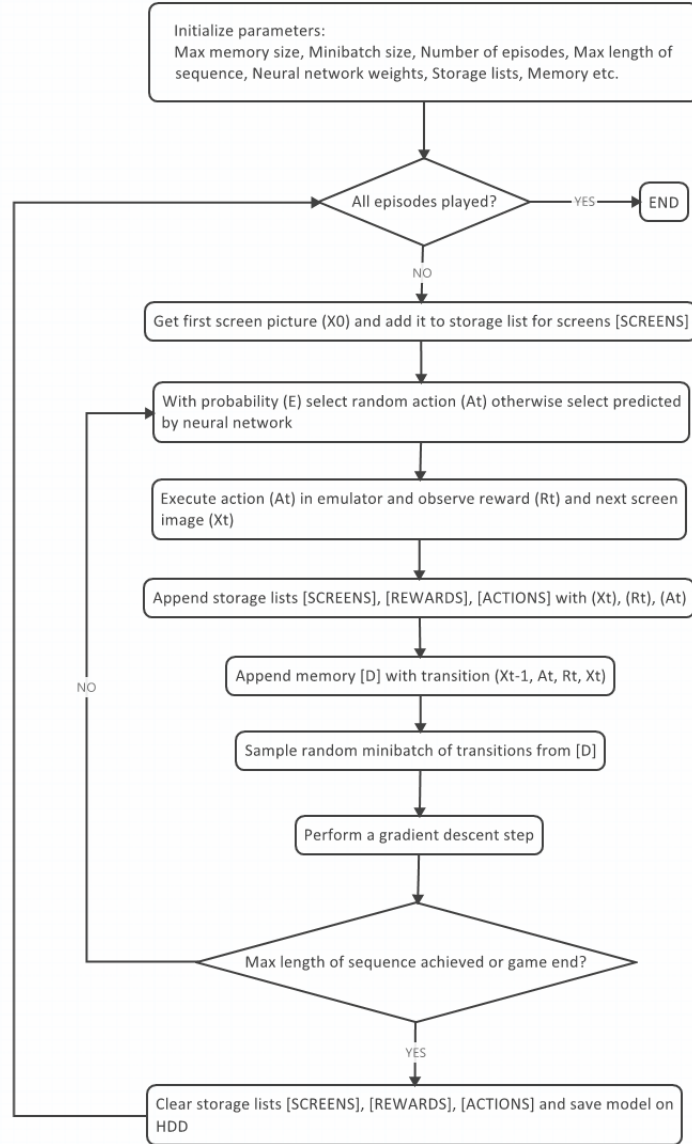
Figure 10: Complete workflow

3.8 Further improvements

There still exist, many possible options for our algorithm's improvement:

1. Initialization of connection weights to C-layers, based on preliminary analysis of the controlled object. We can assume that the convulsion kernels, which were got after neural network training finishing, reduce to the set of "lines" and "spots" filters, so if the initialization was performed based on pre-set filters, it could significantly decrease required training time.

2. Network architecture can be extended by recurrent layer, for providing the possibility to get clustering of video series with unlimited number of clusters. This network can be trained to recognize the new information without retraining. This approach allows to generalize input information and use the optimal solution in the case, when we get new image (video series), but which is referred to the known class.

3. Sparsity helps extract better features from raw data, more complex and generalized. [28]

4. After obtaining features using convolution, we would next like to use them for classification. In theory, one could use all the extracted features with a classifier such as a softmax classifier, but this can be computationally challenging. To address this we could use aggregation operation called pooling. [29]

5. Dropout as universal regularization works better than weight decay and fix the overfitting problem. [30]

6. Long short term memory (LSTM), which is a recurrent neural network (RNN) architecture, can be used instead of experience replay technique. [31]

4 Getting Started with DRL

Questions, that are covered in this section:

- Installing all needed packages for project to run;

- Getting DRL source code and preparing for launching the training process;

- Structure of the project;

- Advance fitting.

All actions should be performed using the command line. Please, perform the next steps to launch terminal emulator in Ubuntu 14.04 or CentOS 7 (supported systems in this guide):

- Ubuntu 14.04
  Press the **Alt-Ctrl-T** shortcut or use the graphic interface: press **Win**, then type "term" and choose your favorite:
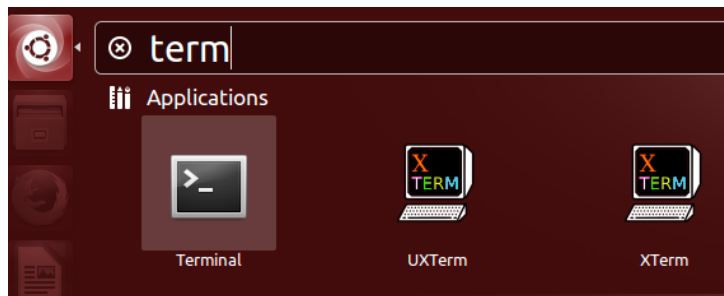
Starting terminal using GUI in Ubuntu

- CentOS 7
  In CentOS there is no hotkey to start the terminal. But there is two ways to start it using GUI:

      Using "Application" menu:

---

[28] http://ufldl.stanford.edu/wiki/index.php/Autoencoders_and_Sparsity

[29] http://ufldl.stanford.edu/wiki/index.php/Pooling

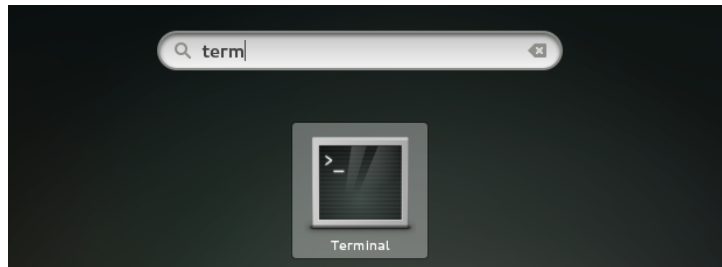[30] http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

[31] http://en.wikipedia.org/wiki/Long_short_term_memory
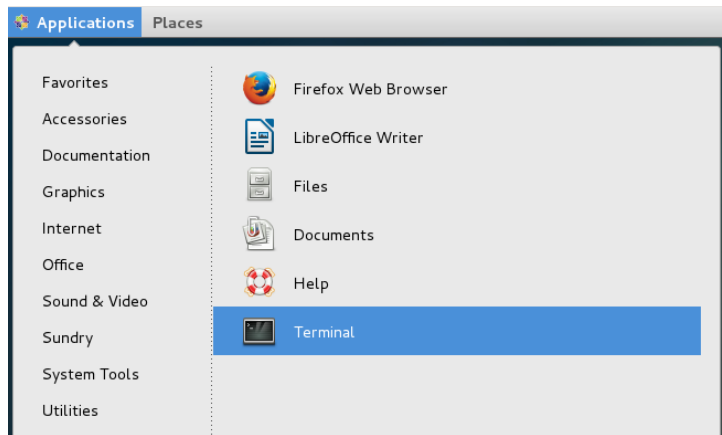
Starting terminal using GUI in CentOS with Application menu

Using **Win** button: press **Win**, then type "term" and choose your favourite

Starting terminal using GUI in CentOS with Win button

4.1 Requirements

> **Note**
>
> Please check the following requirements before you start an installation.

Supported operating systems (desktop versions):

- CentOS 7 [32]
- Ubuntu 14.04 [33]

Hardware requirements:

- **CPU**: 64-bit Intel or AMD multi-core processor (2+ Cores)
- **RAM**: 8GB of RAM (16GB recommended)
- **GPU**: CUDA supported video card [34]
- **Disk Space**: 8GB of free disk space

Make sure that your user is able to sudo. Run in terminal this command to do it:

```
sudo -l
```

The output should be like following:

---

[32]http://www.centos.org/download
[33]http://www.ubuntu.com/desktop
[34]https://developer.nvidia.com/cuda-gpus

> **Output**
>
> User <user> may run the following commands on <host>:
> (ALL : ALL) ALL

Check sudo access

If there is an error in the output, please ask your system administrator to provide you appropriate sudo rights. [35]

4.2 Virtual Machine

For your convenience, we prepared a virtual machine image (in OVA format) containing fully operative demonstration setup of the DRL. With this image you would be able to test the pre-trained model, but without training, so you don't need CUDA supported video card and 4GB of RAM is enough.

> **Note**
>
> Demonstration build can be used only with a traditional CPU-based approach, since it is currently impossible to emulate a NVIDIA environment.

There are two distributions available in a Download section on a project web page:

- CentOS based

- Ubuntu Linux based

After extracting an image from archive, follow the instructions below:

1. Import an extracted VM image into your virtual environment (VMware Player, VirtualBox etc.);

2. Configure an instance according to the hardware requirements;

3. Start an instance;

4. Access to the virtual machine console and log into a desktop environment using following credentials:

   - Login: drl
   - Password: admin

5. Open the README file on the Desktop;

6. Follow the instructions in README.

Now you can test the pre-trained model. If you want to train a model by yourself, follow the instructions in the following sections.

---

[35]http://www.pendrivelinux.com/how-to-add-a-user-to-the-sudoers-list/

4.3 Installation

Installation process was tested on supported systems (CentOS 7 and Ubuntu 14.04) installed on virtual machines from the official sites. Before installation there wasn't any additional software except for software, that was provided with this distributives.

In this section will be covered questions about what packages are needed for DRL to work properly and how to install them.

4.3.1 Needed packages

See "How to install" section (4.3.2) to perform installation of this libraries.

Names of packages are provided in format:

Conventions

<Ubuntu_package>/<CentOS_package> - names of packages in both supported systems delimited by slash (/)

OR

Conventions

<Name_of_package> (System) - if package is needed to be installed only

---

General purpose packages:

- *g++/gcc-c++* - the C++ front end for GCC compiler.

- *libopenblas-dev/blas-devel* - The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. [36]

- *libyaml-dev/libyaml-dev* - YAML is a human friendly data serialization standard for all programming languages. Is needed for pylearn2. [37]

- SDL packages:

  > *libsdl1.2-dev/SDL-devel.x86_64*
  >
  > *libsdl-image1.2-dev/SDL_image-devel.x86_64*
  >
  > *libsdl-gfx1.2-dev/SDL_gfx-devel.x86_64*

  SDL - Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. It is used by video playback software, emulators, and popular games. [38]

- *gcc (CentOS)* - compiler system produced by the GNU Project supporting various programming languages. [39]

- *gcc-gfortran (CentOS)* - free Fortran 95/2003/2008 compiler for GCC. [40]

- *lapack-devel (CentOS)* - provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. [41]

- *atlas-devel (CentOS)* - provides C and Fortran77 interfaces to a portably efficient BLAS implementation. [42]

---

[36] http://www.netlib.org/blas/
[37] http://www.yaml.org
[38] https://www.libsdl.org/
[39] https://gcc.gnu.org/
[40] https://gcc.gnu.org/wiki/GFortran
[41] http://www.netlib.org/lapack/
[42] http://math-atlas.sourceforge.net/

- *zlib-devel.x86_64 (CentOS)* - general purpose compression library. [43]

Python packages:

This section is about what packages should be installed before DRL installation. The common commands for installation of this packages you can find in the next section. This information is provided for you to be informed what libraries do DRL use. Some of this packages need to be installed using **pip**. To find out which tool to use - see the next section.

List of packages:

- *python-dev/python-devel* - contains the header files and libraries needed for python to be extended with dynamically loaded extensions. [44]

- *python-pip/pip* - recommended tool for installing Python packages. [45]

- *python-numpy/numpy==1.6.1* - the fundamental package for scientific computing with Python. [46]

- *python-scipy/scipy==0.10.1* - Python-based ecosystem of open-source software for mathematics, science, and engineering. [47]

- *python-matplotlib/python-matplotlib* - Python 2D plotting library. [48]

- *Theano/Theano* - a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. [49]

- *pylearn2/pylearn2* - machine learning library. Needed in DRL for Convolution Networks. [50]

- *python-pil/python-pillow* - Python imaging library. [51]

- *python-nose/python-nose* - testing library for Python. [52]

4.3.2 How to install

DRL is written in Python, so make sure that you have at Python 2.7 or later installed on your system. It may be easily checked from the command line:

```
python --version
```


Check Python version

If the output looks like *"Python 2.7.x"* (Python 3.x.x is not ok) - you have the correct version of Python and you can skip Python installation section. In case of error or another version, please, install the correct version of Python on your system and, if needed, make a symlink *"python"* in one of the $PATH directories to the interpreter of this version.

---

[43]http://www.zlib.net/
[44]https://www.python.org/
[45]https://pypi.python.org/pypi/pip/
[46]http://www.numpy.org/
[47]http://www.scipy.org/
[48]http://matplotlib.org/
[49]http://deeplearning.net/software/theano/
[50]https://github.com/lisa-lab/pylearn2
[51]http://www.pythonware.com/products/pil/
[52]https://nose.readthedocs.org/en/latest/

Python installation:

- Ubuntu 14.04:

```
sudo apt-get install build-essential
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev \
libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
cd ~/Downloads/
wget http://python.org/ftp/python/2.7.5/Python-2.7.5.tgz
tar -xvf Python-2.7.5.tgz
cd Python-2.7.5
./configure
make
sudo make install
```

- CentOS 7:

```
sudo yum groupinstall "Development tools"
sudo yum install zlib-devel bzip2-devel openssl-devel ncurses-devel \
sqlite-devel readline-devel tk-devel gdbm-devel db4-devel libpcap-devel \
xz-devel
cd ~/Downloads/
wget --no-check-certificate \
https://www.python.org/ftp/python/2.7.5/Python-2.7.5.tar.xz
tar -xvf Python-2.7.5.tar.xz
cd Python-2.7.5
./configure --prefix=/usr/local
make
sudo make altinstall
```

Pre-installation:
If you are using CentOS, you should add *EPEL* and *RPMForge* repositories:

```
sudo yum install \
http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-5.noarch.rpm
sudo yum install \
http://pkgs.repoforge.org/rpmforge-release/rpmforge-release-0.5.3-1.el7.rf.x86_64.rpm
```

Paste following lines line by line to your terminal to install all needed packages:

- Ubuntu 14.04:

```
sudo apt-get install python-pil python-numpy python-scipy python-dev \
python-pip python-nose g++ libopenblas-dev git libsdl1.2-dev \
libsdl-image1.2-dev libsdl-gfx1.2-dev python-matplotlib libyaml-dev
sudo pip install -U numpy
sudo pip install -U pillow==2.7.0
sudo pip install Theano
```

- CentOS 7:

```
    sudo yum install python-devel python-nose python-pillow python-setuptools \
    gcc gcc-gfortran gcc-c++ blas-devel lapack-devel atlas-devel \
    SDL-devel.x86_64 SDL_image-devel.x86_64 SDL_gfx-devel.x86_64 \
    zlib-devel.x86_64 git python-matplotlib libyaml-dev
    sudo easy_install pip
    sudo pip install -U numpy
    sudo pip install -U pillow==2.7.0
    sudo pip install scipy
    sudo pip install Theano
```

Also, you should install *pylearn2* package. Steps are the same for the both systems:

```
cd ~/ # Or any other directory where you want the pylearn2 project to be stored
git clone https://github.com/lisa-lab/pylearn2.git
cd ./pylearn2
sudo python setup.py develop
```

DRL project is developed and tested with CUDA (parallel computing platform and programming model created by NVIDIA[53]). CUDA is developed for fast GPU computing. CPU mode is also available, but deep reinforcement learning in this mode is too slow, so CPU mode is not recommended for learning. However, if you have a trained model, it is possible to test it on CPU without any speed issues.

Follow the instructions on NVIDIA page to install CUDA SDK[54].

Also, if you want to use CUDA not only in this terminal session, you have to change some global variables in your *~/.bashrc*:

```
cat >> ~/.bashrc <<EOF
export PATH=/usr/local/cuda-6.5/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda-6.5/lib64:$LD_LIBRARY_PATH
EOF
```

Great! You system is ready for DRL to be installed and tested.

DRL installation:

To install DRL you need to:

1. Make sure that correct Python version is installed in your system and Python interpreter can by started with "python" command.

2. Make all the pre-installation steps. Make sure that everything is installed properly without errors.

3. Clone source code from GitHub[55]:

   ```
   git clone https://github.com/DSG-SoftServe/DRL
   ```

4. Navigate to the project directory

   ```
   cd DRL
   ```

5. Run *install.sh* script.

   ```
   bash ./install.sh
   sudo chown -R $USER ~/.theano # to run the demo without sudo
   ```

---

[53]https://developer.nvidia.com/cuda-zone
[54]http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/index.html#pre-installation-actions
[55]https://github.com/DSG-SoftServe/DRL

If installation is succeed, the environment is ready to use.

4.3.3 CPU mode

It is not recommended to run the demo in CPU mode. But, if you don't have NVIDIA GPU and want to check out if it's working, apply this fix:

```
cd src
drl_optimize=False # False for CPU, True for GPU
drl_regex="(\s*optimized=)[a-zA-Z]+(\))"
sed -ri "s:$drl_regex:\1$drl_optimize\2:g" ai/neuralnet.py
```

> **WARNING**
>
> Use only "False" and "True" values

To launch the demo in CPU mode, use *"cpuRun.sh"* instead of *"gpuRun.sh"* in future paragraphs.

4.4 Launching and using

4.4.1 First launch

You have DRL installed and it's time for the first launch.
Navigate to the DRL project directory and perform following steps to launch the learning process:

```
cd ./src/
bash gpuRun.sh main_exp.py
```

Congratulations, you have launched the deep reinforcement learning process.
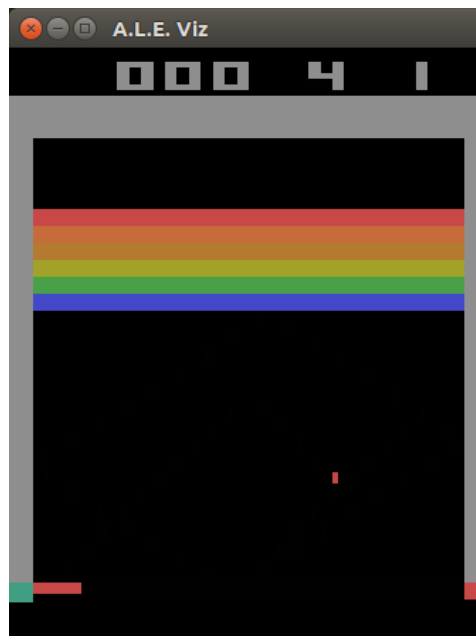Now you should see the window like this:

Figure 11: Breakout

Also, there should be an output in the command line like this:

> **Output**
>
> action RAND: 2
> S: 50 A: 49 R: 49 D: 171
> _____ Reward 0
> Step: 177
> 0.04417

Let's find out what does this indicators means:

- *action RAND: 2* - there is 4 available actions ([0..3]). The action can be chosen from neural network (NN) experience with probability ϵ, where $0<\epsilon<1$. Than you'll see "NEURALNET" instead of "RAND". Otherwise, NN will perform a random action;

- *S: 50* - current state. States are changing with ascending order starting from 0 since last negative reward;

- *A: 49* - performed action. Logic of numbering is similar to state;

- *R: 49* - reward for action. Logic of numbering is similar to state;

- *D: 171* - memory;

- *Reward 0* - current value of reward. Could be -1, 0 or 1;

- *Step: 177* - number of steps performed from the learning start;

- *0.04417* - time, spent for the step (in minutes).

"S", "A" and "R" are from the Markov decision processes (MDPs) model[56].
So, we can see such correlations between command line and window:

- when the "action" is changing, the actor (bottom block) changes it's behavior;

- when the shell hits one of the upper blocks, the reward equals 1;

- when the actor looses the last life, the reward equals -1.

4.4.2 Project structure

Let's look closer at some of the important parts of the project. Below, you can see the output of Unix "tree -d" command, except for unnecessary for understanding folders:

---

[56]http://en.wikipedia.org/wiki/Markov_decision_process

```
DRL
├── libraries
│   └── ale
│       ├── doc
│       ├── roms
│       ├── src
│       └── state
└── src
    ├── ai
    ├── ale
    ├── data
    ├── graphs
    ├── memory
    ├── models
    └── tnnf
```
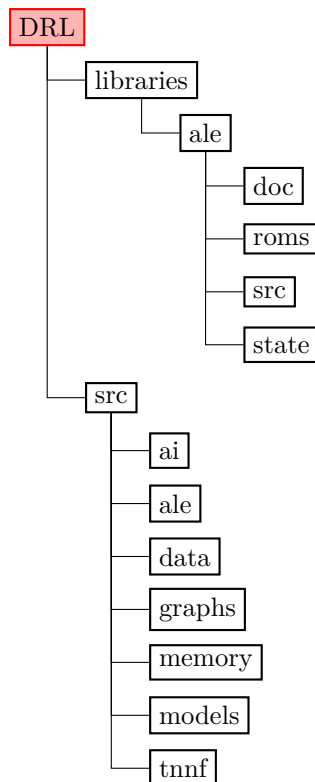
There is two folders in the root of the project structure that need to be mentioned:

- **libraries** - contains *"ale"* folder. ALE is a tool, which gave the ability to learn models on different Atari games. From the official site:
  The Arcade Learning Environment (ALE) is a simple object-oriented framework that allows researchers and hobbyists to develop AI agents for Atari 2600 games. It is built on top of the Atari 2600 emulator Stella and separates the details of emulation from agent design. [57] The most interesting folder in libraries/ale directory is roms. Here you can find games, for which you can train NN. (see the additional fitting chapter)

- **src** - the main folder obviously. All source code of the DRL project is located in this folder. Five directories should be mentioned here:

  ai - source code of neural network, which is used within the project;

  ale - wrapper code for interaction with the Arcade Learning Environment;

  data - scripts to work with data;

  graphs - graphs with experiment results;

  memory - implementation of neural network memory;

  models - directory, where all trained models will be located;

  tnnf - source code for Tiny Neural Net Framework. [58] used to create and train the neural network, which is used in this project.

The **graphs** directory contains following graphs:

- *rewargs.png* - sum of rewards for multiple iterations (1000 by default), e.g. Left plots on Figures 12 and 13;

- *rgstat.png* - average rewards for 1 finished game from multiple iterations (1000 by default), e.g. Right plots on Figures 12 and 13;

- *wsum.png* - sum of the weights on both convolutional layers of neural network.

---

[57]http://www.arcadelearningenvironment.org/
[58]http://tnnf.readthedocs.org/en/latest/index.html

4.4.3 Using existing models

The result of the training part is a file, saved in *DRL/src/models/* directory. You can use this file to find out how good your NN was trained. You can use the *play.py* script in the *src* directory to play a game with your trained NN. Launching game with a trained NN is similar to training launching:

```
cd DRL/src
bash gpuRun.sh play.py
```

You should see the similar to the training output: window with the game and output in command line. But there shouldn't be any training time in the output.

In the first version of DRL there is no ways to configure which model to use, except for editing *play.py* script. By default, the pre-trained model "~~p-AS.model.weights.NN.NMem.906508" is hardcoded.

You can change the script to use your own model in two ways: to edit the file or to run the *sed* tool.

> **Note**
>
> path to the model is relative to the play.py script

Edit the file using text editor

1. In command line navigate to the DRL project directory, then to *src* folder. Using your favorite text editor, open the *play.py* script;

2. Perform search to navigate to the line, which starts with "self.nnet.loadModel";

3. Change the path in quotes to the path of your model;

4. Save the file.

String to change

Edit the file using sed

Also, you can use the sed command to change the file. Navigate to the DRL project directory, then to src folder and run this command to do it:

```
drl_model=./models/model.weights.NN.old.pillow # change this value
sed -ri "s:(\s*self\.nnet\.loadModel\(').*('\)):\1$drl_model\2:" play.py
```

> **WARNING**
>
> Don't change the sed command except for the path to your model, unless you know what you are doing.

Additional fitting

In the first version of the DRL project you can't change training configuration easily using config files. But, if you really want to set up your own training parameters, you can do it by editing source files. Below you can see explanation of the most important training parameters and then, instruction how to change them.

Training parameters:

- *Memory size* - how many samples should we store in memory to take mini-batch from at one step;

- *Mini-batch size* - how many samples should we take from memory to perform the next learning step;

- *Decrease of epsilon step* - decreasing rate of epsilon which is responsible for decision to take a random step or a step, predicted by NN;

- *Number of allowed actions* - number of actions which NN can perform;

- *Skip frames* - there is too many frames which the emulator produces. Some of them should be skipped;

- *Maximum sequence length* - maximum number of steps that NN can perform for one game;

- *Maximum number of episodes* - maximum number of games during training;

- *Saving counter threshold* - when should the program save the game. Counts relatively to steps performed by NN;

- *Base model name* - the first part of the name of model which will be generated;

How to change parameters:
All this parameters are changing in file *DRL/src/main.py*, so navigate to the *src* directory:

```
cd DRL/src
```

In examples below will be used default values for you to be able to restore the initial state. Just change the value of the variable in the first command to change the certain training parameter.

> **WARNING**
>
> It is supposed that you are working with the unedited version of *"main.py"* of the first version of DRL. If you are not sure, please, don't use this commands. They may take no effect or even corrupt the code. Also, you may change this parameters in your favorite text editor, but it's not recommended.

Memory size:

```
drl_mem_size=100000 # change this value
drl_regex='(\s*self\.size\s?=\s?)[0-9]+(\s*#\s?(M|m)emory\ssize)'
sed -ri "s:$drl_regex:\1$drl_mem_size\2:" main_exp.py
```

Minibatch size:

```
drl_minibatch=128 # change this value
drl_regex='(\s+self\.minibatch_size\s?=\s?)[0-9]+(\s*#\s?(M|m)inibatch\ssize)'
sed -ri "s:$drl_regex:\1$drl_minibatch\2:" main_exp.py
```

Decrease of epsilon step:

```
drl_eps_step=700000 # change this value
drl_regex='(\s*self\.epsilon_step_dec\s*=\s*)[0-9]+(.*)'
sed -ri "s|$drl_regex|\1$drl_eps_step\2|" main_exp.py
```

Number of allowed actions:

```
drl_actions=4 # change this value
drl_regex='(\s+self\.number_of_actions\s?=\s?)[0-9]+(\s*#\s?(L|l)eft,?\sright.*)'
sed -ri "s:$drl_regex:\1$drl_actions\2:" main_exp.py
```

Skip frames:

```
drl_skipframes=6 # change this value
drl_regex='(\s*self\.ale\s*=\s*ALE\(self\.memory,\sskip_frames=)[0-9]+(\).*)'
sed -ri "s|$drl_regex|\1$drl_skipframes\2|" main_exp.py
```

Maximum sequence length:

```
drl_max_seqlength=50000 # change this value
drl_regex='(\s+while\sself\.T\s<\s)[0-9]+(\sand\snot\sself\.ale\.game_over:)'
sed -ri "s|$drl_regex|\1$drl_max_seqlength\2|" main_exp.py
```

Maximum number of episodes:

```
drl_max_episodes=50000000 # change this value
drl_regex='(\s+while\sself\.M\s<\s)[0-9]+(:)'
sed -ri "s|$drl_regex|\1$drl_max_episodes\2|" main_exp.py
```

Saving counter threshold:

```
drl_count_threshold=20000 # change this value
drl_regex='(\s+if\sself\.count\s>=\s)[0-9]+(:)'
sed -ri "s|$drl_regex|\1$drl_count_threshold\2|" main_exp.py
```

Base model name:

```
drl_model_name=AS.model.weights.NN.NMem # change this value
drl_regex='(self\.nnet\.saveModel\(\'\.\/models\/).*(\.'\s+\sstr\(self\.steps\)\)'
sed -ri "s|$drl_regex|\1$drl_model_name\2|" main_exp.py
```

To revert all changes:

```
drl_regex='(\s*self\.size\s?=\s?)[0-9]+(\s*#\s?(M|m)emory\ssize)'
sed -ri "s:$drl_regex:\1100000\2:" main_exp.py
drl_regex='(\s+self\.minibatch_size\s?=\s?)[0-9]+(\s*#\s?(M|m)inibatch\ssize)'
sed -ri "s:$drl_regex:\1128\2:" main_exp.py
drl_regex='(\s*self\.epsilon_step_dec\s*=\s*)[0-9]+(.*)'
sed -ri "s|$drl_regex|\1700000\2|" main_exp.py
drl_regex='(\s+self\.number_of_actions\s?=\s?)[0-9]+(\s*#\s?(L|l)eft,?\sright.*)'
sed -ri "s:$drl_regex:\14\2:" main_exp.py
drl_regex='(\s*self\.ale\s*=\s*ALE\(self\.memory,\sskip_frames=)[0-9]+(\).*)'
sed -ri "s|$drl_regex|\16\2|" main_exp.py
drl_regex='(\s+while\sself\.M\s<\s)[0-9]+(:)'
sed -ri "s|$drl_regex|\150000000\2|" main_exp.py
drl_regex='(\s+while\sself\.T\s<\s)[0-9]+(\sand\snot\sself\.ale\.game_over:)'
sed -ri "s|$drl_regex|\150000\2|" main_exp.py
drl_regex='(\s+if\sself\.count\s>=\s)[0-9]+(:)'
sed -ri "s|$drl_regex|\120000\2|" main_exp.py
drl_regex="(self\.nnet\.saveModel\(\'\.\/models\/).*(\.\'\s+\sstr\(self\.steps\)\)"
sed -ri "s|$drl_regex|\1AS.model.weights.NN.NMem\2|" main_exp.py
```

5 Experiments

The model was trained with 900000 steps on NVIDIA GeForce GTX 880M. The training time was two days. This model was compared to random action model. The trained model, as expected, showed better results. The analysis consists of two comparisons: of total rewards for 1000 steps and of average games rewards for 1000 steps.

The experiment was evaluated on 100000 steps. For one hit, the agent gets reward equals 1. On graphs, one point - is result for 1000 steps. On the "Y" axes there is a reward and on the "X" axes there is a number on batch.

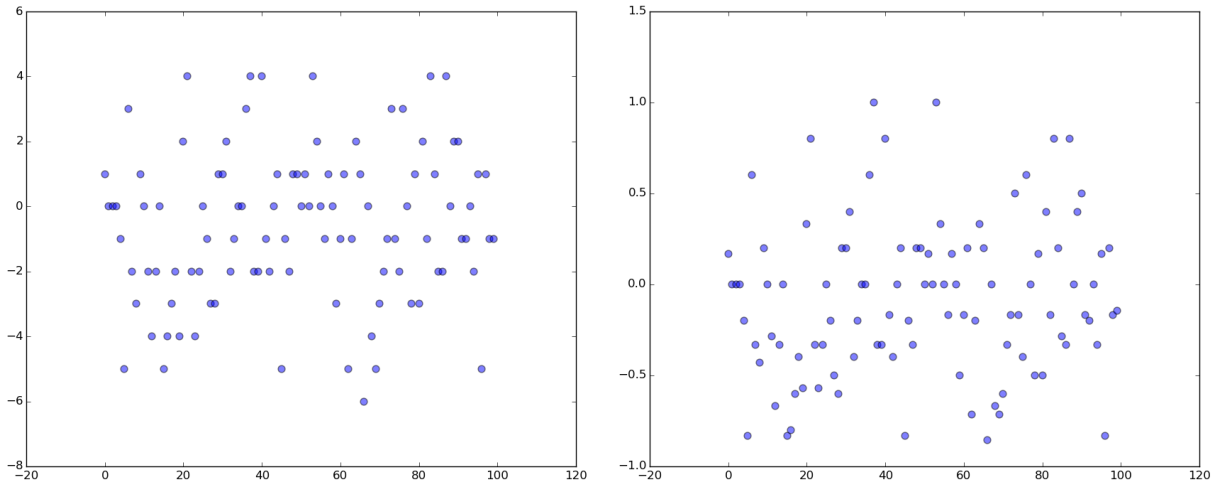Below you can see raw results for the random action model (Figures 12):

Figure 12: Random action model. Left - rewards for 1000 iterations; Right - average rewards for 1 finished game.

The graphs shows expected behavior. Mean rewards are around 0.
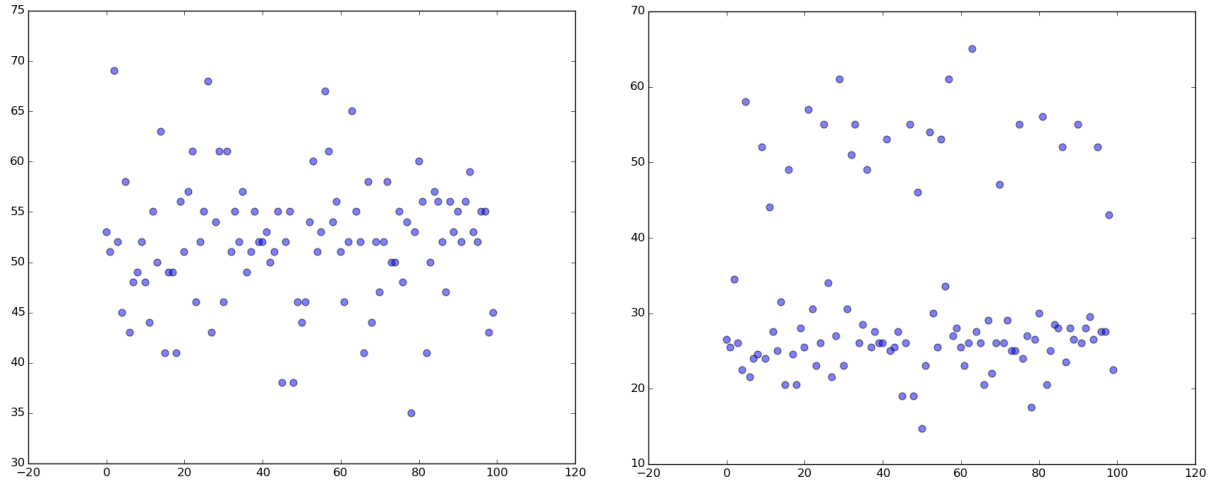But for the trained model - rewards are higher (Figures 13):

Figure 13: Trained model. Left - rewards for 1000 iterations; Right - average rewards for 1 finished game.

This graphs shows different behavior. Mean rewards are around 32 which is higher than the random action model results.

In the the Table 1 below statistical information about the models is presented:

	Random rewards	Random averages	Trained rewards	Trained averages
Min	-6	-0.85	35	14.67
Max	4	1.0	69	65.0
Mean	-0.62	-0.08	52.0	32.32
Median	-1.0	-0.15	52.0	27.0
STD	2.35	0.42	6.32	12.42
Variance	5.53	0.18	40.0	154.19

Table 1: Comparing

Figures 14 and 15 below [59] shows relation between results for both models on the same scale. The top figure within each plot is for trained model, and the bottom one is for random action model. The median is marked with a bold dotted line on each figure.

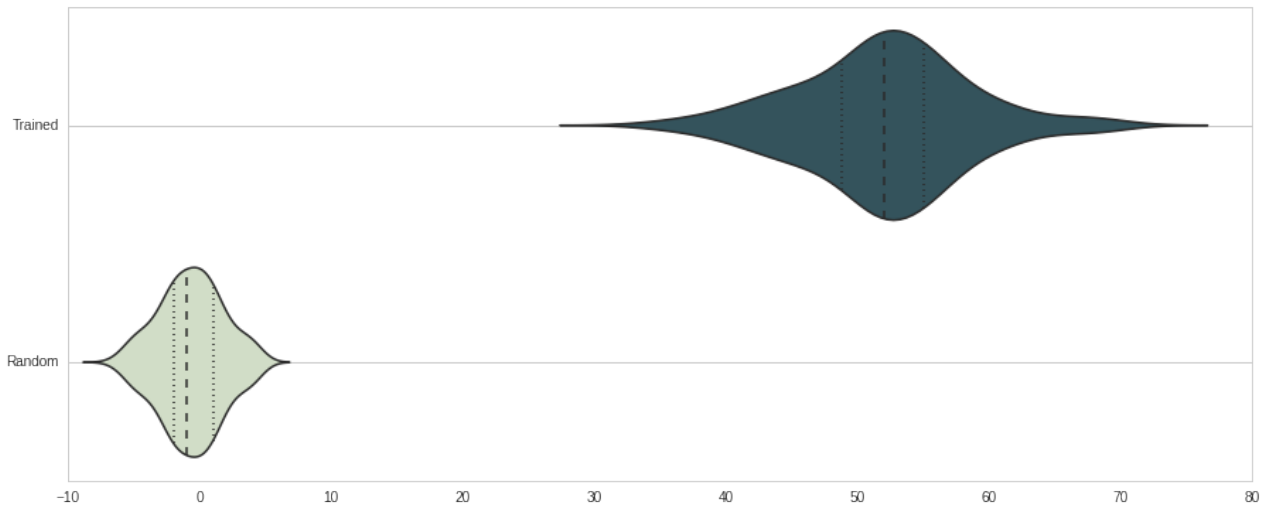This diagrams shows that the results of the trained model are higher then results of the random action model.
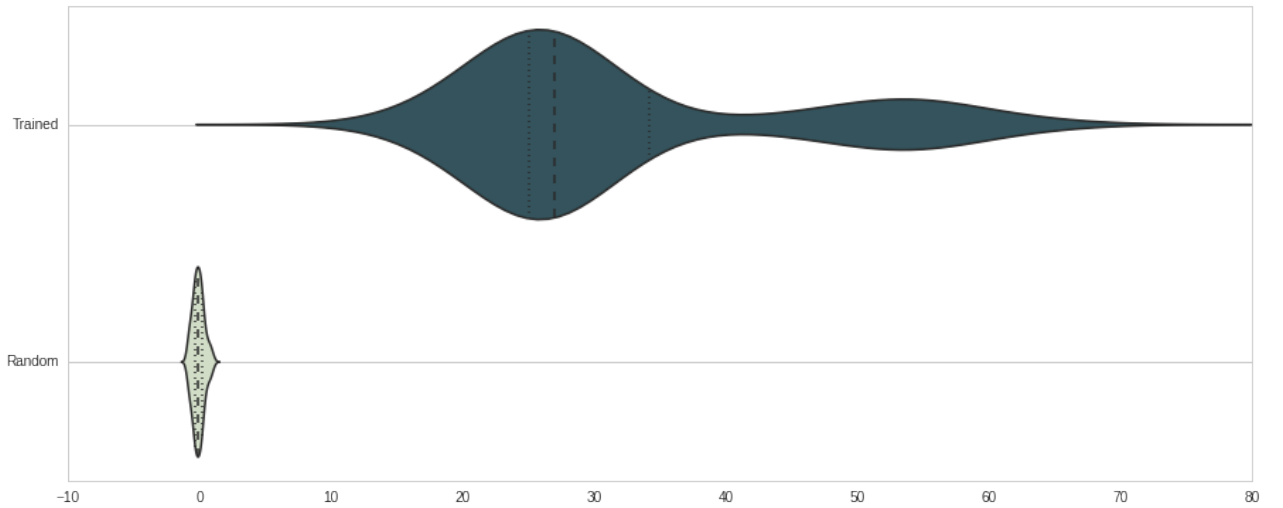

Figure 14: Compared rewards

Figure 15: Compared averages

---

[59] http://en.wikipedia.org/wiki/Violin_plot