Visual Vector Agent Environment

Manual

November 21, 2012

Quickstart

The ViVAE is stored in free git repository at http://github.com/HKou/vivae/. The easiest way of getting it is to make git clone of the sources (assuming that you have a git tool installed in your system):

```
git clone git://github.com/HKou/vivae.git
```

Then, change to vivae directory and type:

```
ant
```

to compile the sources. An example of usage is included. It can be executed using:

```
ant run
```

Normally, a java window appears (see Figure 1) and a simulation is loaded. The agents in the simulation are equipped with randomly generated fully connected recurrent neural networks that controll wheel movement based on sensory information. There are two types of sensors included in this simulation. The first type is the distance sensor. the second one is the surface sensor. Five instances of each sensor arecircularly distributed around the agents. 1000 iterations is performed and the program ends typing out the final fitness, which is the average speed of all agents in the simulation. The program outputs the following (fitness values may be different):

```
ant run
Buildfile: build.xml

run:
     [java] average speed fitness = 0.08090585578183623
     [java] average ontop fitness = 0.031265608469645184
```

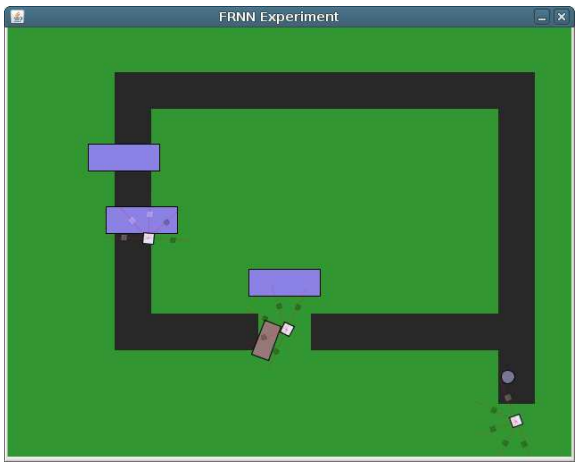

Figure 1: First vivae execution. FRNNExperiment class is started, Java window appears and simulation with three agents in the arena begins. Each agent is equipped with 5 distance sensors (red lines) and 5 surface sensors (black/white) boxes. There is also two obstacles in the simulation.

The experiment consists of FRNN* classes in vivae.example package. FRNNExperiment class contains the main() method. FRNNController contains the recurrent neural network and controlls

the agent movement. FRNNControlledRobot class is the controlled robot itself containing the sensors. The program outputs two fitness values. One is the average speed of the robots, the second one reflects how much have the obstacles been moved closer to tue upper edge of the arena.

1 Experiment Design

This section describes how to design a new experiment scenario and how to load it into the ViVAE. The experimental scenario is saved in an SVG [**?**] file, that is loaded by arena.loadScenario(String svgFilename) function. The loadScenario(String svgFilename) function opens the SVG file and tries to construct all aobject according to the SVG label specified ofr the particular graphics element. The SVG file can be created by e.g. Inkscape [**?**]

1.1 Objects in Simulation

1.2 Known Issues

2 Measurements

This section presents methods used for measurement of the simulation and obtaing the results. Most common way of doing it is a fitness function that returns a given quality of the simulation. the fitness function is used afterwards. The fitness classes are placed in vivae.fitness package.

2.1 Fitness Functions

The concept of fitness functions contains a set of classes, where each class has getFitness() method that returns the actual value of the fitness measure specified in the class.
The basic usage calls the particular fitness class conructor before the experiment starts to initialize it. The getFitness() class is called after the experiment ends to get the fitness value.
The fitness functions not neccessarily have to be called after the experiment ends.

2.1.1 Average Speed

Average speed fitness getFitness() method returns the average speed of all agents in the simulation. Each agent represented by an offspring of Robot class has an odometer o, which accumulates the distance it travels each simulation step. The overall fitness is divided by a number of agents A in the simulation and number of simulation steps:

$$f_v = \frac{\sum_{\forall a}^{A} o_a}{A \cdot steps} \tag{1}$$

2.1.2 Movables on Top

MovablesOnTop fitness class saves sum of distances d of all Movable offspring classes M in simulation first (in the constructor). At the end of simulation (t_{end}) a sum vertical positions is subtracted from the saved value divided by a number of the movable obstacles and normalized to the arena height according to the following equation:

$$d = \sum_{m}^{M} y_m, \qquad f_m = \frac{d^{t_0} - d^{t_{end}}}{M \cdot Y_m} \tag{2}$$

where y_m is the vertical position of the movable object center and Y_m is the arena height.

3 examples

This section describes how to execute a simple experiment with ViVAEfrom the programmer's point of view.

3.1 FRNN Experiment

There are classes for the simple experiment that demonstrates usage of the ViVAE. The main FRNNExperiment class, FRNNControlledRobot class that contains an implementation of the agent and FRNNController that contains the fully connected recurrent neural network [**?**].

3.1.1 FRNNExperiment class

First, and instance of FRNNExperiment is created in the main() method. Then an instnce of vivae.arena.Arena is created with two parameters. First parameter is a string with a path to the SVG file containing the scenario. The second is a boolean parameter that defines, whether the simulation runs with or without the visualization. Offline simultion runs much faster (20-30x).

```
FRNNExperiment exp = new FRNNExperiment();
exp.createArena("data/scenarios/arena3.svg",true);
```

The createArena(String svgFilename, boolean visible) method creates instances of the Arena and a frame to which the visualization is performed. If the frame is not to be visualized, the sleep time of the computation thread is setup to 0. Otherwise, the thread is being periodically syspended to allow updating of the visualization.
Then, number of neurons and sensors is determined. In this example, number of sensors is 5 and number of neurons is 2. The agent is equipped with distance and surface sensors. In tis case, the agent will have 5 distance and 5 surface sensors.
Afterwards a random matrix is constructed and implanted to the controllers. the matrix is three-dimensional. It is a list of neural network weight matrices. Each weight matrix is then n by $i+n+1$ matrix, where n is a number of neurons and i is a number of neural network inputs. In our case, $n = 2$ and $i = 10$. Thus the inner matrices have dimension of 2 by 13. It is up to a user how many matrices he specifies. The setupExperiment() method takes this matrices (matrix) and distributes them among the agents starting with the first matrix. If there is less weight matrices than agents, the method reads the matrix array from the beginning. For example, if there are 3 agents in the simulation and the highest dimension of the three-dimensional array of matrices is 3, the each agent gets a distinct weight matrix. If the highest dimension (slowest changing) is 1, then all agents receive the same weight matrix. Having the highest dimension two times smaller than number of agents, two groups of agents are formed. The method takes the agents in the order they are stored in the SVG file.
In the example, the random three-dimensional array of matrices is created as follows:

```
double[][][] wm = Util.randomArray3D(3,neurons,2*sensors+neurons+1,-5,5);
exp.setupExperiment(wm,50,25);
```

The highest (slowest changing) dimension is 3, thus each agent receives it's own randomly generated weight matrix of size 2×13. Each matrix xontains randomly generated numbers in interval $(-5, 5)$. The matrix is used in the setupExperiment(wm, 50, 25), where 50 is the length of distance sensors (the maximum distance the sensor examines) and 25 is the distance, to which the surface sensors are placed. The number of sensors as well as number of neurons is determined from the weight matrix.

Then, fitness function classes are initialized, the experiment is performed and fitness values obtained are printed out using the following code:

```
FitnessFunction mot = new MovablesOnTop(exp.arena);//initialize fitness
FitnessFunction avg = new AverageSpeed(exp.arena);
exp.startExperiment();
System.out.println("average speed fitness = "+ avg.getFitness());
System.out.println("average ontop fitness = "+ mot.getFitness());
```

Each FitnessFunction offspring class has an access to the whole Arena. The classes are usually initialized before the experiment starts to collect the data. The, the getFitness() method is called after the experiment ends. The method collects the results and returns the fitness as a double value.

3.1.2 FRNNController class

The FRNNController class is assigned to the agent and it's moveControlledObject() is called each simulation step. The method purpose is to adjust agent's rotation and velocity through angle change and acceleration using Robot.rotate(float) and Robot.accelerate(float) methods.
The FRNNController contains initFRNN(double[][], double[][],double[]) methot that initializes the FRNN:

```
public void initFRNN(double[][] wIn, double[][] wRec, double[] wThr) {
    frnn.init(wIn, wRec, wThr);
}
```

The first parameter is the input weight matrix of size $n \times i$, the second is the recurrent weights matrix $n \times n$ and the last is the threshold vector of length n.
The moveControlledObject() looks like this:

```
public void moveControlledObject() {
    if (robot instanceof FRNNControlledRobot) {
        double[] input = Util.flatten(((FRNNControlledRobot) robot).getSensoryData());
        double[] eval = frnn.evalNetwork(input);
        double lWheel = eval[0];
        double rWheel = eval[eval.length - 1];
        double angle;
        double acceleration = 5.0 * (lWheel + rWheel);
        if(acceleration<0)acceleration=0;
        double speed = Math.abs(robot.getSpeed() / robot.getMaxSpeed());
        speed = Math.min(Math.max(speed,-1),1);
        if (rWheel > lWheel) { angle = 10 * (1.0 - speed); }
            else {angle = -10 * (1.0 - speed);}
        robot.rotate((float) angle);
        robot.accelerate((float) acceleration);
    }
}
```

First, the controller checks whether it controlls the correct instance. Then, it gets the sensory data throught FRNNControlledRobot.getSensoryData() method, which returns two-dimensional array of size $2 \times i/2$, of sensor outputs. First row is the output of distance sensors, second is the output of surface sensors. The array is flattened to a single dimensional array of size i (distance sensors preceed the surface sensors). The recurrent network is evaluated using the sensory data as the

input vector. The result vector of size n is used to controll the robot. First neuron output is used to controll the simulated left wheel, last neuron output is used to controll simulated right wheel. The wheel acceleration is transformed to robot rotation an acceleration as can be seen in the code. The maximum angular change is inversely proportional to the agent velocity.

3.1.3 FRNNControlledRobot

The most important methods in FRNNControlledRobot class are double[][] getSensoryData() that returns the sensory data array and setSensors(int, double, double, double, double) that setups the sensors. The first method returns the sensory data in a form of two-dimensional array, where the first row contains the distances to the objects and the second row contains the surface frictions. Both arrays are normalized in interval $(0, 1)$, where distance of 0 says that there is no object in the range of the sensors and distance of 1 means that the robot is as close as possible to the object. Friction of 0 is the road friction and friction of 1 is the grass friction.
The second method sets the sensors for the agent. The first parameter is the number of sensors (of each type), the second is the starting angle, the third is the angle increment, the fourth is the maximum distance for the distance sensor and the fifth is the distance of surface sensors.
Each sensor is added using the following code (in case of the DistanceSensor):

```
Sensor s = new DistanceSensor(this, angle, sensorNumber, maxDistance);
sensors.add(s);
sensorsMap.put(sensorNumber, s);
sensorNumber++;
```

3.2 Random Search

FRNNRandomSearch class demonstrates usage of the agents in a simple experiment that sets up a baseline for the on-road driving by a random search among the recurrent neural network controllers.
There are two useful methods in this class. First one double[][] search(String scenario, int sensors, int neurons, int evals) performs search among specific number of evaluations of the randomly generated weight matrices for a particular number of sensors and neurons.
The second void play(String scenario, double[][] wm) plays the best weight matrix found with visualization turned on.

3.3 Mathematica Example

The random search is implemented as well as a notebook for Wolfram mathematica. the notebook uses the ViVAEas a fitness function evaluated through JLink interface. Mathematica does not allow to call System.exit() method to be called from the mathematica kernel. It looks like that the method is searched using static class analysis thus we cannot directly use the FRNNExperiment class but a wrapper JLinkExperiment class is used instead. The class overrides createArena() method and adds the following evaluate() method:

```
public double evaluate(){
    avg = new AverageSpeed(arena);
    startExperiment();
    return avg.getFitness();
}
```

that returns the average velocity fitness.
The class can be used within the Mathematica Notebook in the following way (the code is implemented in RandomSearchExample.nb Mathematica Notebook).
The first part of the Mathematica code initializes the JLink, sets the directory, installs Java and adds all jars distributed with ViVAEto the classpath.

```
JLink Setup
Needs["JLink`"]
nbdir=NotebookDirectory[];
SetDirectory[nbdir];
jars=Map[StringJoin[nbdir,#]&,FileNames["*.jar",{"*"},Infinity]];
ReinstallJava[CommandLine="/usr/lib/jvm/java-6-sun/bin/java"];
AddToClassPath[Sequence@@Append[jars,nbdir<>"build/classes"]];
```

The second part defines evalonce[] function that first creates an instance of JLinkExperiment class. Than calls the createArena() method with scenario file name and visualization (True or False). The it sets up the experiment, evaluates it and releases the class. The class release is necessary. It calls the garbage collector. Not calling the ReleaseJavaObject[] function causes Java heap overflow after some executions of the evalOnce[] function.

```
evalOnce[scenario_,wm_,visualization_]:=Module[{exp,res},
    exp=JavaNew["vivae.example.JLinkExperiment"];
    exp@createArena[nbdir<>scenario,visualization];
    exp@setupExperiment[wm,50,25];
    res=exp@evaluate[];
    ReleaseJavaObject[exp];
    res
]
```

The next cell sets up number of sensors and neurons that we want to use in the experiment and generates three weight matrices as a test example. The matrices are implanted to controllers of the three agents found in the arena1.svg scenario and the experiment is executed without the visualization using the following code:

```
sensors=5;neurons=2;
wm=RandomReal[{-15,15},{3,neurons,2 sensors+neurons+1}];
evalOnce["data/scenarios/arena1.svg",wm,False]
```

The last block implements the random search. search[] function searches performs specified number of evaluations of random weight matrices (note the second parameter in RandomReal function call, all agents have the same weight matrix). The function is called and the result is stored in the res symbol. The function sorts the results and returns the best wight matrix together with it's fitness.

```
search[scenario_,sensors_,neurons_,evals_]:=
    Last@Sort@Map[{evalOnce[scenario,#,False],#}&,
        RandomReal[{-15,15},{evals,1,neurons,2 sensors+neurons+1}]
                  ]
res=search["data/scenarios/arena1.svg",sensors,neurons,10];
```

The best network can be visualized using the evalOnce[] function call changing the visualization parameter to True.

```
evalOnce["data/scenarios/arena1.svg",res[[2]],True]
```