An Actor-Critic implementation for solving platform games on SpiNNaker

Stefan Grigore

Supervisor: Prof. Steve Furber

April 2019

Third Year Project Report
MEng Computer Science
School of Computer Science - University of Manchester

Abstract — **Translating high level environmental inputs to meaningful actions and strategies is an important problem for fields such as mobile robotics or robotic prostethics. Some of the greatest obstacles faced are the number of iterations that conventional learning models require to produce effective strategies, and the unpredictable results that actions in the real world have. Furthermore, encoding these strategies and retrieving resulting actions require complex data structures that pose limitations regarding scalability, power consumption, memory use and computational time for elaborate tasks performed by embedded systems. This report presents an implementation on the SpiNNaker neuromorphic computing platform of an Actor-Critic inspired model, applied to solve a platform game that outlines the problems a real world task might present. The project aims to use the SpiNNaker platform to tackle the obstacles of such applications, and to produce a biologically plausible model that might give an insight into how learning and memory work in the human brain.**

Key Words: actor-critic methods, reinforcement learning, neuromorphic computing

Contents

1 Introduction

The main limitations of neural network implementations is scalability, power consumption, memory use and computation time. The chip area required by synapses increases quadratically with the number of neurons (Benjamin et al., 2014). Furthermore, the computational complexity, memory and power consumption of the model grows with the precision of the simulation (the firing rate of neurons) as each signal needs to be propagated across the network to alter the synaptic weights (Brette et al., 2007). All these factors pose problems for models that aim to solve complex, faster than real time tasks as part of embedded systems. The motivation behind pursuing a biologically plausible model and using a platform such as SpiNNaker is also inspired by the great discrepancy of several orders of magnitude (Sarpeshkar, 1998) in computational efficiency between neurobiology and electronics, which has been first pointed out by Mead (Mead, 1990).

1.1 Project goals

The scope of this project is to produce a high performance, high scalability, customisable and biologically plausible spiking neural network reinforcement learning model that solves pathfinding and platform games in a limited number of learning episodes, and that could be adapted for other tasks such as controlling hardware with the purpose of reaching similar learning and progress-based goals.

One important aim of the project is to gain practical experience and an understanding of neuromorphic computing and spiking neural networks that would give an inshight into how the human brain works and how this knowledge can help us build more efficient software systems.

The report will detail the different approaches for managing, representing and extracting information from spiking neural networks, the ability of the model to learn and adapt, the performance of the model, limitations of the platform, advantages compared to other frameworks and future possible developments.

2 Background

2.1 The SpiNNaker system

SpiNNaker (Furber et al., 2014) is a massivelly parallel, event-based neuromorphic computing platform composed of low-power ARM processors which enables the real time simulation of spiking neural networks. The advantage of a spiking neuron over models such as the fully digital, first generation perceptron model or the second generation sigmoidal neural net is that it describes the output of a biological neuron more accurately by incorporating the concept of time in order to encode information (Maass, 1996). The application presented in this report has been implemented by using a 4 node SpiNN-3 board. Each node is composed of 18 processing cores, one of them elected at start-up as a Monitor Core, leaving 16 cores to support the application and one left for fault tolerance and manufacturing yield enhancing purposes (Furber et al., 2013). Each core can model around 1000 neurons, such as the leaky integrate and fire model, which simulates the loss of potential that occurs at the membrane level of the biological neuron.

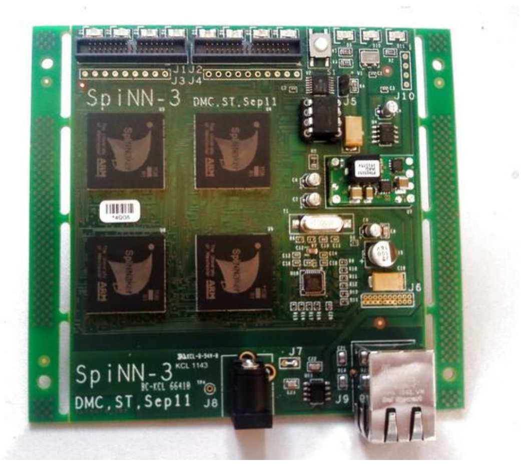
Figure 1: A 4-node SpiNN-3 board

The supporting software provided is the sPyNNaker Python library which enables the specification of neural models using the PyNN description language, and the run-time user interaction with the simulation.

2.2 The Game

The game chosen to showcase the learning model is called Super Meat Boy, a commercially available video game developed by independent studio Team Meat. The game belongs to the genre of platform games, where a player controlled character has to jump and climb between suspended platforms while avoiding obstacles in order to reach a goal.

The game was chosen because it showcases some of the properties of the gridworld task (Sutton et al., 1998), used as a common test case for reinforcement learning. The character can move in the 4 directions of the 2D plane, with the addition of properties such as gravity, intertia or friction. These additional elements mean that the movements are not precise, and the actions of the agent do not always go as planned, which highlights problems that real world application have to tackle.

2.3 Reinforcement Learning

Supervised learning models learn from labeled examples and with the use of a ”knowledgeable teacher”. They aim to gradually decrease the deviation (error) between the target output and actual output of the model. Compared to this method, reinforcement learning models use a ”critic” which does not need to have knowledge of the target output. The critic only evaluates the behaviour of the system, and the model aims to maximise the amount of reward and decrease the amount of penalty. This mimics the behaviour described in the theory which states that neurons are individually ”hedonists” that work to maximise a local analog of pleasure, while minimizing a local analog of pain (Klopf, 1982). Furthermore, experiments found

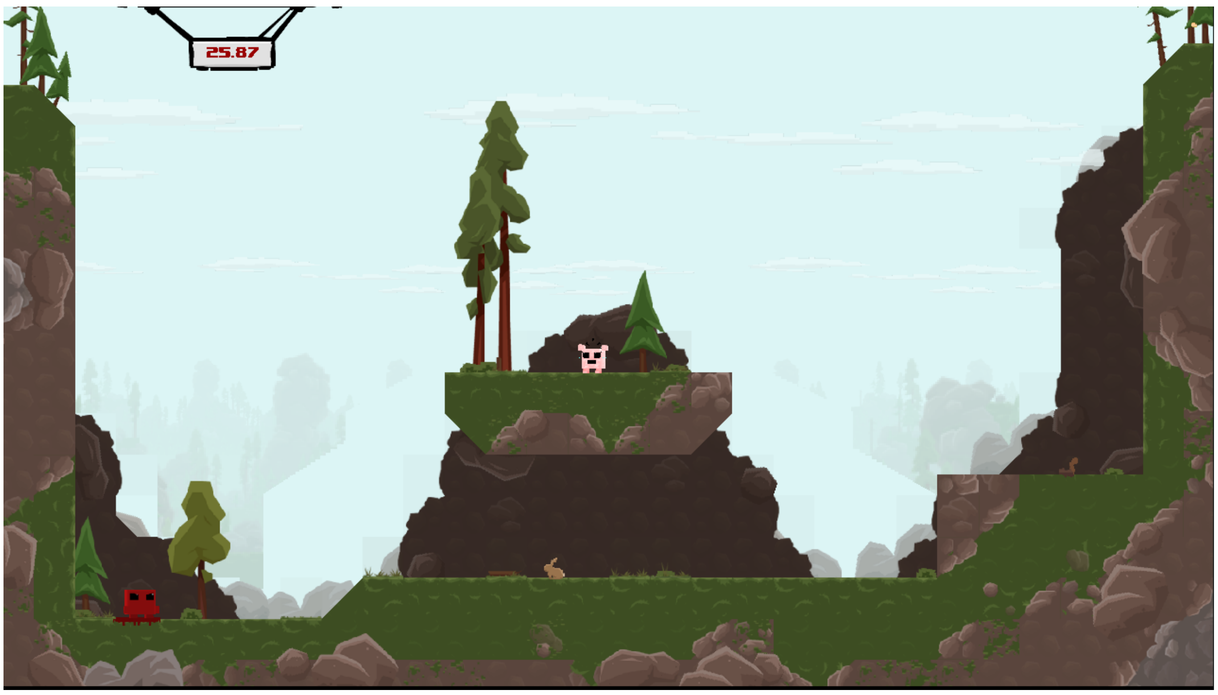

Figure 2: Frist level of Super Meat Boy, with the red, square shaped main character on the bottom left and the pink goal on the middle platform

that animals learn through an ”associative process”, where by trial and error over repeated iterations they can find the optimal solution to puzzles (Thorndike, 1898). In these experiments it was found that actions closely followed by satisfaction become more likely to occur, while those which are accompanied by discomfort become less likely (the law of effect).

Reinforcement learning exploits actions that worked best for each situation while searching new actions through trial and evaluation.

3 Development

3.1 The sPyNNaker Python library

The project has been implemented by using the sPyNNaker Python library which enables the modelling of spiking neurons and neural networks, the simulation of those models and the run-time interaction with them (Rhodes et al., 2018).

3.1.1 Spiking neural networks

Biological neurons have been observed to produce sharp increases in electrical potential across their membrane, lasting roughly 1 millisecond, commonly known as spikes. As charges are transferred across synapses from presynaptic neurons to a postsynaptic neuron, the potential of the postsynaptic neuron builds up until it releases the charge itself in the form of a spike that is sent forward in the network, where the process repeats.

The library uses the PyNN neural network description language to build the models of these neurons and the interactions between them. Neural populations are created, representing groups of neurons with similar properties. A number of standard neuron models are provided by PyNN, with one of the most basic of them being the Leaky Integrate and Fire (LIF) model. The LIF neuron is modeled as a resistor and capacitor connected in parallel. Charge is built in the capacitor as spikes are received, but leaks out through the resistor over time. If the voltage exceeds a threshold, the charge is released in the form of a spike which is transmitted to the connected postsynaptic neurons. Once a refractory period where the neuron is not allowed to spike again passes, the neuron resumes operation as before (Rhodes et al., 2018).

Projections between these populations of neurons can be created, describing the connections between them and their type. The simulation is then loaded and ran on the SpiNNaker board.

3.1.2 Spike receivers and transmitters

PyNN does not support live interaction with the simulation. The sPyNNaker library, however, provides a module that enables the live injection and retrieval of spikes during a simulation, while maintaining its real-time operation.

The SpiNNaker machine can send or receive multicast packets describing spiking events (i.e. the ID of the neuron that fired). The transmitters and receivers communicate at UDP ports, and hardware such as sensors or actuators could potentially be connected to these ports for implementing real world applications. In the case of our game, spike receivers have been configured to execute button presses that move the agent across the level, while spike transmitters associated with the critic and environment networks alter the behaviour of the agent and trigger new moves.

When live I/O is used, a database of the network is created. Spikes could be injected during the simulation by either loading spiking callbacks onto the database before the start of the simulation, or by running the spike transmitters and receivers in a separate execution thread which would communicate with the simulation thread.

3.1.3 Spike Timing Dependent Plasticity

According to the Hebbian theory describing the learning mechanisms of biological systems, the strength of synaptic connections is suggested to be influenced by the causal

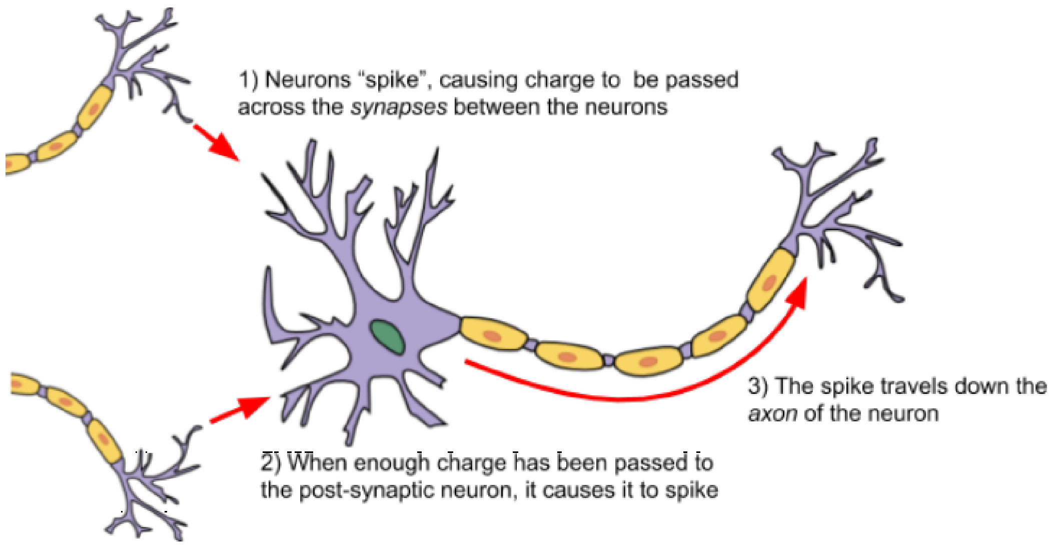
1) Neurons "spike", causing charge to be passed across the *synapses* between the neurons
2) When enough charge has been passed to the post-synaptic neuron, it causes it to spike
3) The spike travels down the *axon* of the neuron

Figure 3: Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals (University of Manchester, 2016)

relation between pre and postsynaptic neurons (Hebb, 1949). Spike Timing Dependent Plasticity (STDP) is a biological process where the strengths of the synapses are modified depending on the timing between the spikes of two connected neurons. If a spike of a presynaptic neuron is followed closely by a postsynaptic spike, then it is considered that the presynaptic neuron directly contributed to the firing of the postsynaptic neuron, and the weight of the synapse is increased (potentiation). Conversely, if a postsynaptic spike comes before a presynaptic one, then it is considered that it was caused by an input from somewhere else in the network, and the synaptic weight between these neurons is decreased (depression) (Mikaitis et al., 2018).

This mechanism is considered to be the basis of learning and memory in the human brain. A model of this process can be instantiated for projections between neuron populations by using the sPyNNaker library.

3.2 The Actor-Critic learning model

The actor-critic model can be viewed through the framework of control theory. *'The behavior of the controlled system is influenced by disturbances, and feedback from the controlled system to the controller provides information on which the control signals can depend'* (Barto, 1995).

Fig. 4 shows a diagram adapted from Sutton & Barto, 1998, of an actor-critic architecture implemented by Potjans et al., 2009. The actor supplies an action to the environment based on its policy. As a result of the action, a new state is entered and the environment transmits the new state information to the actor and critic. The critic evaluates whether the new state is better or worse and emits an error signal used to update both the value function and the policy.

In the case of our game, the agent executes actions and the environment sends both the agent and the critic the information regarding the resulting state. The critic then evaluates the results and modifies both the policy of the agent and its own internal information, and then the game is restarted. This counts as one learning episode, and the agent in the next episode will now use the policy learned and the signal given by the environment to execute new actions, change its behaviour if previous ones have been modified, exploit rewarded actions and explore new ones if mandated by the critic.

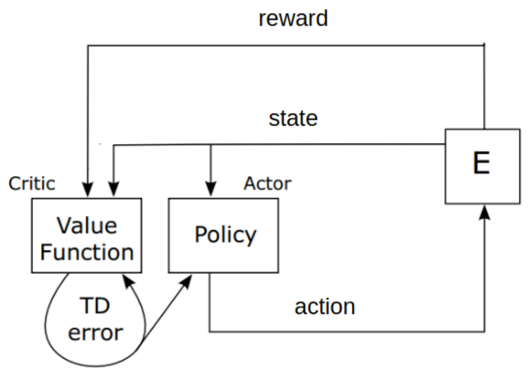

Figure 4: Actor critic architecture (Potjans et al., 2009)

This cycle is repeated, the agent aquires a policy and the critic optimises it until the game is solved.

3.2.1 The agent

'A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state.' (Sutton et al., 1998). There are 4 meaningful actions that the agent could take in the case of our game: walking right, left, jumping right or jumping left. Furthermore, we need to keep track of the action that should be taken at each step in time (i.e. for each state). These actions across steps are represented by two populations of LIF neurons, a presynaptic state population and a

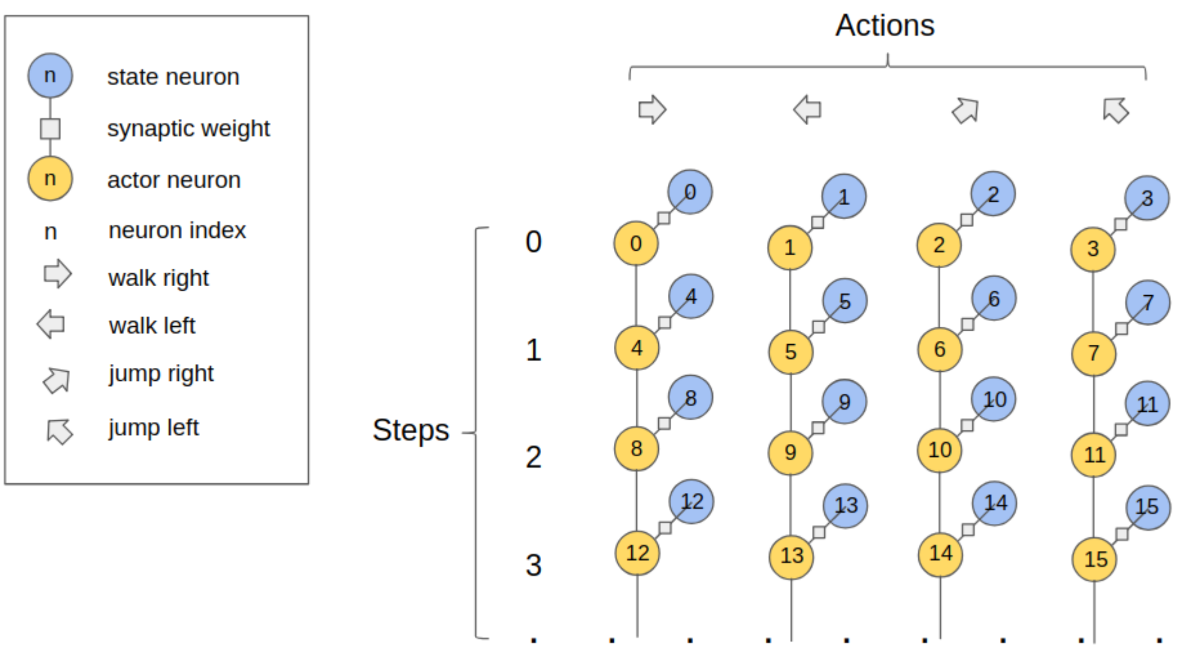

Figure 5: Diagram showing how the synaptic weights between the state neurons and the actor neurons encode the policy of the agent.

postsynaptic actor population, illustrated in Fig. 5 as blue and yellow respectively.

The neurons in each population are connected one-to-one by synapses implementing the STDP model. The populations are split so that the index of each neuron describes the action and step that they correspond to. For example, for *n* possible actions, the modulo and quotinent of the division by *n* of the index of the neuron that fired would give the action index and step index respectively. So, in our example, if we have 4 possible actions and the agent gives out a sequence of neurons that spiked such as [0, 6, 8, 12], this would translate into [walk right, jump right, walk right, walk right] which would be executed in that order. Furthermore, neurons representing actions at each step are connected to their analogues for future steps (the connections are seen as the vertical lines in Fig. 5), mimicking the behaviour where actions executed in the present will increase the probability that the same actions will be executed in the future. In the case of the first level of the game, these additional connections improved the probability of the agent solving the level in under 10 learning iterations from X% to Y%.

'The synaptic weights between the state neurons and the actor neurons encode the policy of the agent' (Potjans et al., 2009). Each action at each step has a weight associated with it, implemented by the STDP synapses between the state and actor populations. For any particular step, the action with the greatest weight is retrieved from the network by using first-spike coding, where the neuron with the greatest synaptic weight is the most likely to fire first (VanRullen, 2005). *The stronger the synaptic weights between the activated state and a given actor neuron, the greater the probability that it will fire first. Whichever actor neuron fires first in response to the activation of a state is interpreted by the environment as the chosen action.'* (Potjans et al., 2009). To achieve this, a population of spike injector neurons is connected to the state population, sending spikes to all neurons for each step (as seen in Fig. 6), while the spike receivers wait for the first spike generated by the actor neurons that will translate into an action to be executed on the environment.

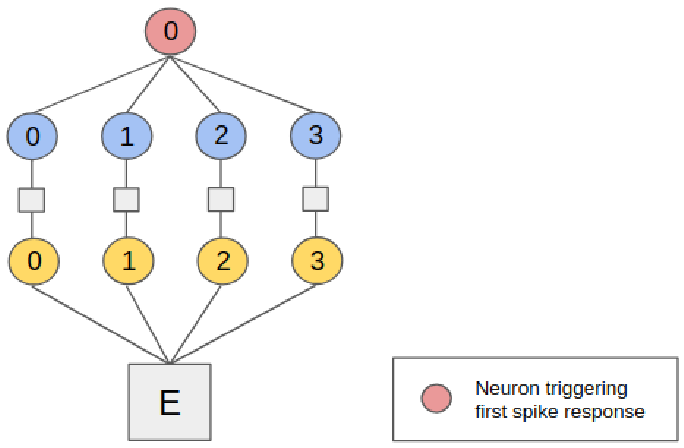

Figure 6: Spike injector neuron triggering first spike response for first step (step 0)

With these techniques, no other data structure is used in order to store the information about the policy of the agent, which gives a biologically accurate model for how memory might work in the brain by using synaptic plasticity.

Fig. 7 shows a diagram of the neuronal implementation of the actor-critic architecture given by Potjans et al., 2009, by which this implementation was inspired.

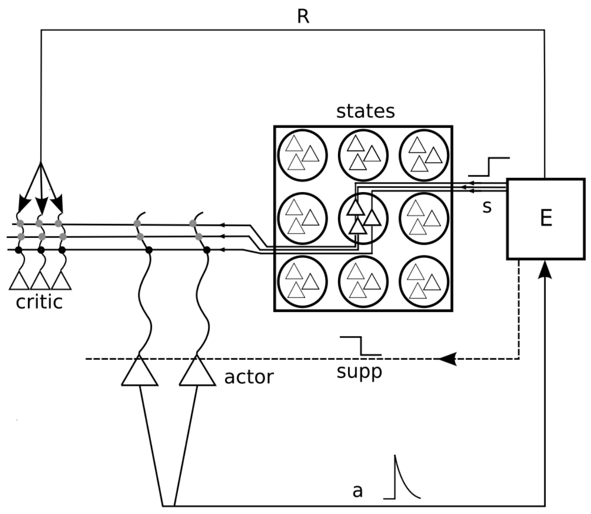

Figure 7: Neuronal implementation of the actor critic architecture (Potjans et al., 2009)

3.2.2 The environment

To model the state, as in the *'signal conveying to the agent some sense of "how the environment is" at a particular time'* (Sutton et al., 1998), a Computer Vision system is used. An external Python library which uses template matching takes a template of how the agent and goal look like and identifies their coordinates in a screenshot of the running game. There is a population of spike injectors corresponding to the environment which is connected to the state neurons described in the previous section. Depending on where the agent is at the end of a particular episode the environment sends to the agent a signal corresponding to the next action that will get the it directly to the goal, which will then be the action executed in the next learning episode. At the same time, the environment sends the information to the critic which compares the results of the previous learning episode to the results of the current one.

3.2.3 The critic

As the critic gets the information from the environment regarding the progress of the agent towards the goal, it compares it to the progress made in the last state of the previous learning episode and alters the weights of the corresponding actions via STDP to reward or punish those actions. This is propagated to all the actions that happened in that learning episode, with the effect decreasing linearly and the most recent action being influenced the most.

If the agent continually does not make progress (i.e. it does not get closer to the goal), the critic enables an exploration phase in which the agent will try a random action rather than the action suggested by the environment until progress is made.

Actions are grouped into classes depending on whether they produce a similar result (i.e. they take the character in the same direction), so that in the exploration phase the agent tries a random action from a class different from the one that was unsuccesfully executed previously. For the first level of the game, this method proved to raise the probability that the agent will solve it in under 10 learning iterations from X% to Y%.

Fig. 8 shows the synaptic weights of 2 possible actions (walking right versus jumping right) that the agent could take as its second move, recorded during a simulation on the first level of the game. In this case, the agent chose to walk right at the second step in the first learning episode. At the end of it, the critic observed that the agent got closer to the goal then it previously was, so it rewarded this move and the synaptic weight increased. For the second learning episode, however, this choice turned out to lead to the agent not making progress, so it has been punished, the weight becoming lower than the option of jumping right at the second step. Over time, this option stabilised as the optimal choice, and we can see that the weights of those two actions begin to diverge.

The actual values of the weights are not as important as the relative relationship between them. For the actor-critic model, we are only interested in the action with the greatest weight, regardless of its value.

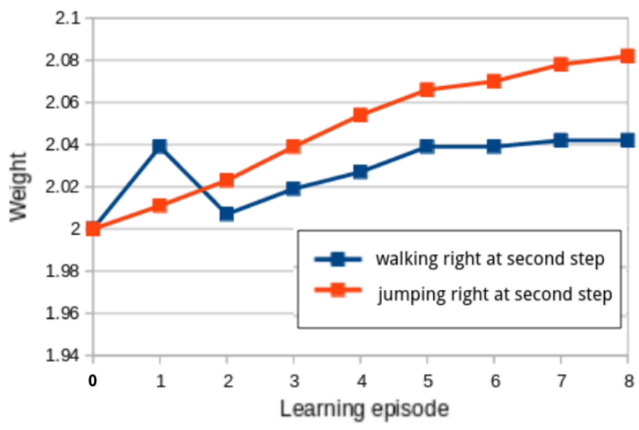

Figure 8: Weight of 2 possible actions for a given step across learning episodes

3.3 Limitations of the SpiNNaker platform

The *run()* method of the SpiNNaker simulation is a blocking operation, and methods such as *getWeights()* cannot be executed until the simulation has been completed. This is because the synaptic weights represent a large amount of data compared to the sparse spiking events, and accessing it could potentially flood the network and likely destroy the simulation (Stokes, 2016).

It is possible however to call *run()* a number of times, extracting data between each run and passing it to an external simulation (University of Manchester, 2016). An initial version of this project would have each learning episode in a separate call of the *run()* method, with simulations being started for each of them. The matrix of synaptic weights would thus be extracted between runs, and the actions with the greatest weights could be selected. The resulting spikes given by the critic and the environment would then be put on the event database as callbacks which would be executed during the run of the next learning episode.

With the current implementation of sPyNNaker, if the memory requirements for a simulation cannot be met, it is split furthermore into steps where it is executed for a shorter amount of time, paused, processed and then resumed repeatedly until it completes. This method, however, repeatedly instantiates the callbacks which were intended to be only ran once during the simulation, executing them at each step, a hidden behavior that might be an error in the current implementation.

Starting a simulation is a relatively lengthy process, taking around 10 seconds. A later version of the project would run the model in a separate execution thread to the thread running the simulation. The critic and environment would now inject spikes into the network during the simulation with the use of spike transmitters instead of callbacks added before the simulation starts. For the weights, we are not interested in the actual values as much as we are interested in which synapse has the greatest value relative to the others. First spike coding (VanRullen, 2005), suggests that the stronger the synaptic weight for a given postsynaptic neuron, the greater the probability that it will fire first. By injecting spikes to groups of neurons corresponding to actions over steps and recording the indices of the first neurons that spike, we can determine the synapses with the greatest weights without interrupting the run, removing the great overhead of repeatedly initialising simulations.

3.3.1 The SpiNNaker Live Event Database

Explanation of how spike events are stored in the database which allows external recievers and senders to figure out which neurons have spiked. Explanation of the implementation of callbacks to store events at the beginning of the simulation. Explanation of how the simulation is set up, issues such as the inability of retreiving data such as the synaptic weights while the simulation is running, and how long simulations need to be split up into multiple runs which might affect the callbacks set before the simulation.

3.3.2 Multithreading

Explanation of the use of separate threads for the simulation and external receivers/transmitters, and how it is preferred over setting all actions through callbacks before the simulation. Explanation of how it enables concurrent actions by setting separate threads for each action.

3.4 Learning

3.4.1 The Leaky Integrate and Fire model

Explanation of the concept of the LIF neuron, the properties modeled by the SpiNNaker platform and what effect they have on the behaviour of the model. Parallels to the biological model, some illustrations.

3.4.2 Spike Timing Dependent Plasticity

Add some graphs of how the weights are altered over the simulation of this application.

4 Performance

4.1 Speed

For the task of the first level illustrated in Figure 1, the model converges to a solution in an average of 11 trials (σ = 1), taking 2 minutes and 30 seconds (σ = 3s) to complete the level. The game was not simulated or sped up, so these are real time benchmarks.

The speed at which the spikes propagate across the network from the moment they are emitted by the spike transmitter to the moment they are received by the spike receiver is 5 milliseconds.

4.2 Scalability

The model can solve increasingly more difficult levels such as the one illustrated in Figure 3 in 25 trials and a total of around 7 minutes.

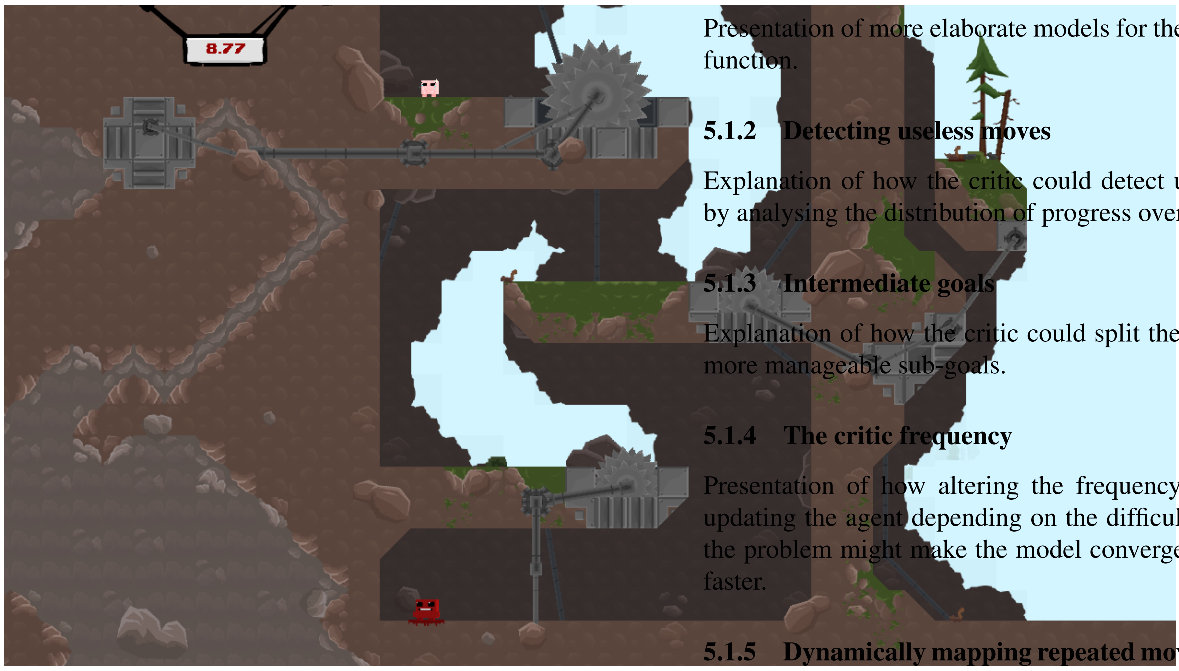

Figure 9: More complex level, with an increased number of platfroms and where sawblades placed on each platform *'kill'* the player and bring him back to the starting point

The number of neurons required to solve the levels is a function of the number of learning episodes, times the number of possible actions. We require this number of neurons for the pre-synaptic and post-synaptic populations of the agent, the populations corresponding to the neruons triggering the first spike action, and the population corresponding to the network of the critic.

$$Number\ of\ neurons = 4 \times number\ of\ trials \times number\ of\ possible\ moves$$

As the agent only has 4 possible moves, so to complete the first level for example it would require $4 \times 11 \times 4 = 176$ neurons. For level in Figure 3, the required number of trials rises to 25, resulting in $4 \times 25 \times 4 = 400$ neurons.

4.3 Sensitivity

The most important variable that the model could be sensitive to is the configuration of the actions. Each action corresponds to a button press, however holding the button for a different amount of time would result in widely different actions. Currently, each action (button press) is held for 0.5 seconds. The model can successfully solve the level in under 12 moves 70% of the time with this configuration. Increasing the duration that the moves are held by up to 30%, however, does not affect the performance of the model, demonstrating that it is resilient to imprecisions or arbitrary configurations in the actions.

5 Future development

5.1 Improving the critic

5.1.1 The value function

Presentation of more elaborate models for the critic's value function.

5.1.2 Detecting useless moves

Explanation of how the critic could detect useless moves by analysing the distribution of progress over time.

5.1.3 Intermediate goals

Explanation of how the critic could split the problem into more manageable sub-goals.

5.1.4 The critic frequency

Presentation of how altering the frequency of the critic updating the agent depending on the difficulty of areas of the problem might make the model converge to a solution faster.

5.1.5 Dynamically mapping repeated moves to a single neuron

Explanation of how the critic could detect sets of repeated moves and map them to a single neuron in the actor in order to improve memory space.

5.2 Connecting hardware

Explanation of how hardware could listen to the UDP ports used by the platform to transmit and receive information and execute actions.

6 Project management and planning

Presentation of how the project was organised by using iterative and incremental development.

7 Conclusion

Presentation of the outcomes of the project.

Acknowledgements

References

Barto, A. G. (1995) *Adaptive Critics and the Basal Ganglia*

Benjamin, B. V., Gao, P., McQuinn, E., Choudhary, S., Chandrasekaran, A. R., Bussat, J.-M. K., et al. (2014) *Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations*

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007) *Simulation of networks of spiking neurons: a review of tools and strategies*

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014) *The SpiNNaker project*

Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., Brown, A. D. (2013) *Overview of the SpiNNaker system architecture*

Hebb, D. O. (1949) *The Organization of Behavior*

Klopf, A. H. (1982) *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*

Maass, W. (1996) *Networks of Spiking Neurons: The Third Generation of Neural Network Models*

Mead, C. A. (1990) *Neuromorphic electronic systems*

Mikaitis, M., Pineda García, G., Knight, J. C., Furber, S. B. (2018). *Neuromodulated Synaptic Plasticity on the SpiNNaker Neuromorphic System*

Potjans, W., Morrison, A., Diesmann, M. (2009) *A spiking neural network model of an actor-critic learning agent*

Rhodes, O., Bogdan, P. A., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., Lester, D. R., Mikaitis, M., Plana, L. A., Rowley, A. G., Stokes, A. B., Furber, S. B. (2018) *sPyNNaker: a Software Package for Running PyNN Simulations on SpiNNaker*

Sarpeshkar, R. (1998) *Analog Versus Digital: Extrapolating from Electronics to Neurobiology*

Stokes, A. B. (2016) *Weight updates during simulation - SpiNNaker Users Group*

Sutton, R. S., Barto, A. G. (1998) *Reinforcement learning: An Introduction*

Thorndike, E. L. (1898) *Animal intelligence: An experimental study of the associative processes in animals*

University of Manchester (2016) *Sixth SpiNNaker workshop lab manuals*

VanRullen, R., Guyonneau, R, Thorpe, S. J. (2005) *Spike times make sense*