Office of Strategic National Alien Planning

Kirby Artifical Intelligence Defense

(K-AID)

Project Team:
Ryan Leonard

June 12, 2017

Executive Summary

The Office of Strategic National Alien Planning (OSNAP) has made a substantial investment in a mech defense system titled Kirby. As part of the acquisition, a simulation system Super Smash Bros. (SSB) was provided for training our Kirby effectively. K-AID is an artifical intelligence engine we created to control Kirby in SSB.

K-AID gathers state information directly from the simulation's memory space about Kirby and the opponnent. Included in this information is location, velocity, acceleration, enemy action, damage dealt, and finally life loss. We pack all of this information into a reinforcement learning model consisting of state, action, and reward. Using this model, K-AID determines a correct action to take in order to maximize future reward (i.e. destroying the enemy without loss of life). The technique used is reinforcement learning implemented using a parallelized deep neural network.

Project Description

We built an artificial intelligence (AI) actor that can play Super Smash Bros. (SSB) as the character Kirby. We set out to be able to defeat an average human player; our project timeline did not allow for such performance.

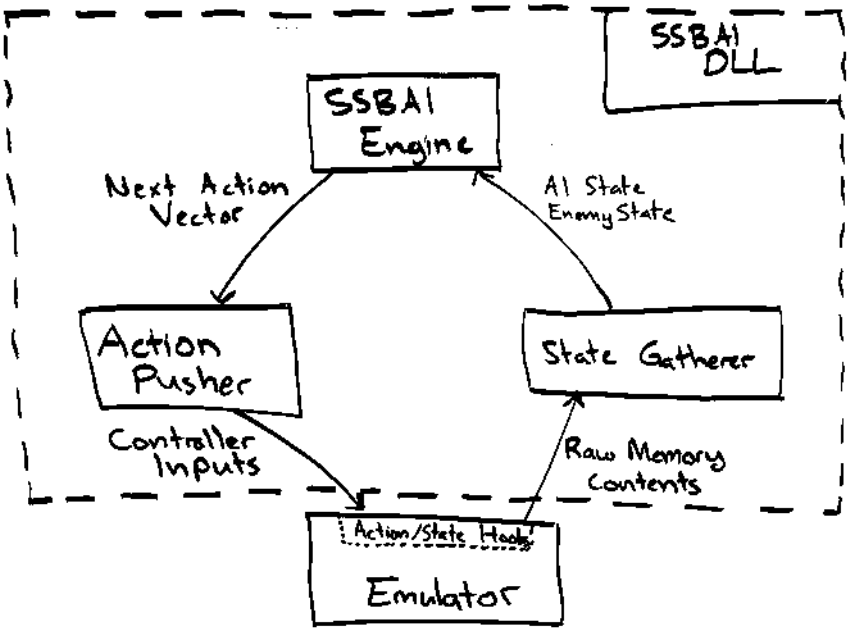
Although our original goal was not met, we were able to accomplish many large subgoals. First, we constructed a framework for any generic AI agent to play SSB using the open source Project64 Emulator. Second, we implemented three neural network layer prediction algorithms including: Perceptron, ReLU, and Linear. Third, we implemented a parameterized neural network construction that allows building networks of various sizes. Finally, we integrated all of these pieces and did performance analysis with various levels of parallelization throughout the implemented network.

Our AI engine performs a major computation, forward propagation through a deep neural network, to determine the best next action for Kirby given the current state and action. Our target performance is to be able to calculate the next action of the AI engine every k^{th} frame. This means we want to perform this AI work approximately 15 times per second, or at a rate of $66ms$ per engine output. We easily achieve this performance through parallelization of the AI engine.

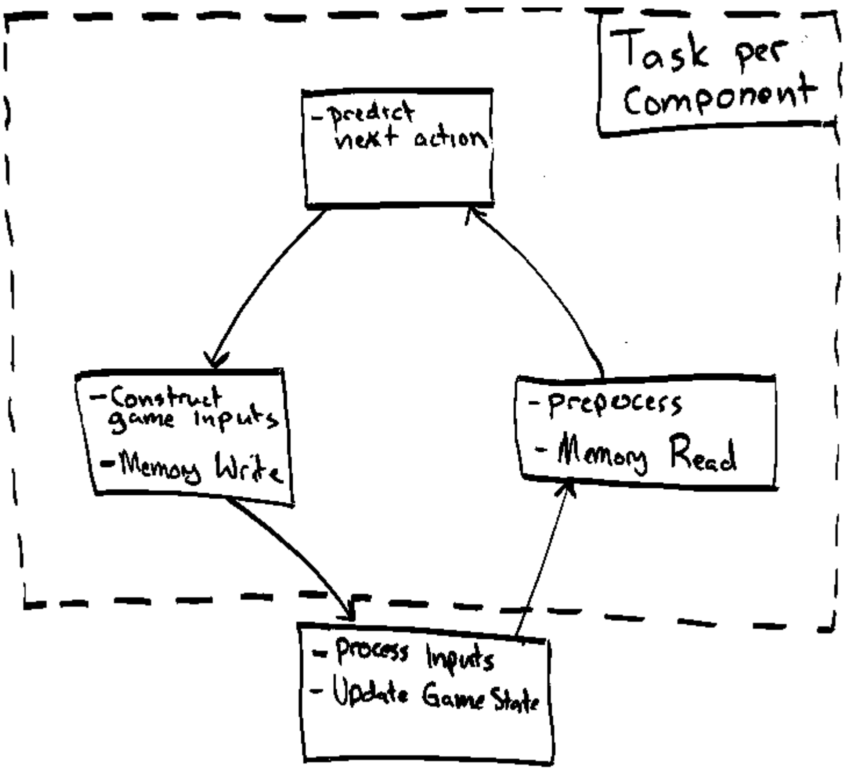
Highlevel Architecture

All development was be performed on the Windows platform using Visual Studio 2015. We have based our project off of the open source Project64 emulator.

Below is the overlying software architecture which indicates the data flow.



We explode this data flow to reveal critical tasks that are completed per frame.



The Hook: Emulator Modification

First, we inserted one simple hook was into the emulator's per-frame-update function. We used this hook to get a pointer to the N64 memory space and the N64 memory-mapped controller data structure. This provided freedom to access any in-game data structures in real time, at each time step.

Finding N64 Data Structures

Determining the memory offset of interesting data structures within the emulators address space was performed using Cheat Engine. Usage of this tool can be summarized as

1. reviewing some range of the process' memory,
2. running the process for a step performing some distinct action, then
3. narrowing the investigated memory range using a byte-wise diff.

More precisely, this tool acts as a debugger with a rich interface and a set of features that easily enable finding particular emulated data structures. The tool is performing a sequence of diffs over the virtual memory space of the debugged process as the process is running. Each diff reduces the amount of virtual memory space being considered until eventually there are only a handful of addresses that are correlated to the distinct action being performed.

As an example, to find the memory offset associated to the location of a character:

1. Run an initial scan over the N64 virtual address space
2. Move the character to the left in the emulator
3. Run a diff in Cheat Engine
4. Repeat step 2 and 3 until there are only a few relevant/correlated addresses.

At this point, we have determined the memory addresses that are correlated to the action of the character moving. Doing some investigation of and around these memory addresses, we easily find the address corresponding to the floating point values of the characters x and y position! Similar investigations were performed to find each characters' damage and life loss state.

Extracting State Values

State extraction was performed using very crude memory manipulation. A set of macros was defined for reading arbitrary memory into our AI data structures. Issues arose with correctly determining the memory offset: original calculations of the N64 RAM to N64 data structure memory offset were incorrect which was only discovered between successive runs after restarting Visual Studio. The ultimate solution was to find the relevant data structure and get the offset of that from the emulated N64 memory address all from within Visual Studio. The result is that state gathering works across all development machines successfully.

Normalization of the state was also performed. This was simply moving all data members to the range of $(-1.0, 1.0)$. We used domain specific information (e.g. SSB level limitations, SSB max damage) to get the normalization range. We also clipped the values in case somehow the limits are exceeded during game play to ensure clean processed data.

Enacting Actions

Pushing an action into the emulator was trivial using a unioned bit-field. The controller data structure in the emulated N64 memory space Is provided by the N64 emulator, so there was no need to hunt down the data structure via cheat engine.

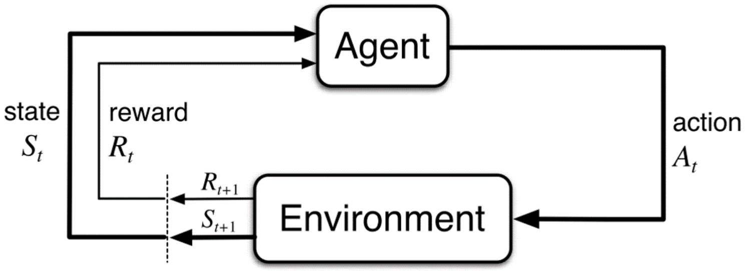
The possible action space was originally 2^{30}, we limited our scope to 2^7. Only combinations of the following actions are permitted:

- Attack
- Special
- Shield
- Jump
- Sprint/Walk
- Left/Right
- Up/Down

AI Engine

Reinforcement Learning

Reinforcement learning (RL) is a model for learning that learns in an active environment with rewards. This is an alternative to both of the major learning paradigms: supervised and unsupervised learning. The RL model can be thought of as containing two components, the environment (SSB) which stochastically determines state and reward and the agent (AI) which intelligently decides the next action based on the current environment.



In a nutshell, the “label” of supervised learning is replaced with “reward” in RL. Additionally, the nature of systems modeled as RL are not viewed as distinct independent events, but rather a *stream* of events. Based on this stream nature and the per-moment reward, the Markov Decision Process is used as the underlying mathematical model for RL problems. From this, the Bellman Equation (Q function) is the core of reinforcement learning. This function tells us the *maximum future return of each state-action pair*:

$$Q(s, a) = r + \gamma \texttt{max}_{a'} Q(s', a') \tag{1}$$

Where s is the current state, a is the current action, r is the reward earned in the next step, γ is the learning rate, s' is the resultant state, and a' is the resultant action.

This is a huge insight for our AI agent! The substructure nature of the Q function allows us to solve this problem via Dynamic Programming if we have a small enough state and action complexity. Once we have the solution for the Q function, we decide our next action by running the Q function over every action for the current state to greedily decide what the best action is for our AI agent's future.

$$\texttt{best}(a) = \texttt{argmax}_a Q(s, a) \tag{2}$$

Unfortunately, determining the Q function absolutely is not tractable for our problem; our state space is very large. We must instead approximate the Q function. We do this using a deep neural network.

Deep Neural Network

A deep neural network (NN) is commonly defined as a neural network with more than one hidden layers. A neural network is a typically a collection of neurons (compute nodes) arranged in a network (regular directed acyclic graph). In the remainder of this document, we consider NNs with > 1 hidden layers that have a fully connected forward feed network.

Our NN was implemented using C++11 with the "O3" optimization flag. We used the standard library vectors for our primary container for inputs. To make our implementation simpler, we leverage standard shared pointers when doing several tasks. This has not demonstrated any performance issues, but the unpredictable nature of garbage collection may require being more careful with memory usage. Finally, all randomization was performed using the standard uniform random number generation distribution using the standard Mersene Twister 19937 random number engine.

Our organization is focused at completing the computation of a network layer as expediently as possible, which we will discuss below. The structure indicated below is in a structure-of-arrays (SoA) style to represent a Network Layer.

```
struct NetworkLayer {
  vector<vector<float>*> weights;
  vector<float> biases;
};
```

Our implementation is able to perform forward propagation on any network layer using three simple variety of computation including.

1. Lossy Rectifier
2. Linear
3. Threshold (Perception)

Currently, the NN has been constructed to allow all hidden layers to use one variety of computation while the output layer can use a different variety. The neural network used in testing was height 512 and depth 7 .

Parallelization and Evaluation

In regards to parallelization, there are many restrictions disallowing us from parallelizing major portions of our application. However, the NN implementation is nearly embarrassingly parallel except for some very regular reduction and stencil dependencies.

All tests were performed on the same machine with no other major processes running except for the development environment and the emulator. Each version of the agent was compiled then executed for 30 seconds (approximately 500 AI predictions) in the same emulator state. The average prediction time was recorded by inserting instrumentation code to capture elapsed time then outputting the total elapsed time divided by the number of frames measured. This is a simple average that might not account for any initialization bias in our code.

Sequential Restrictions

There is a serial bottleneck in that our game AI works in a world where each scene depends on the prior. This is kin to a serial physics simulation, and is an inherit limitation to building a video game AI. To get more parallelization at this level, we could instead perform multiple simulations in parallel. Although this is out of the scope of this project, it is actually a technique commonly used in recent years to paralyze training efforts. This is a variety of weak scaling that we do not target in this work.

Parallelization of Nodes

Each node performs a dot product between two vectors of 512 floats (weights and inputs). This is a reduction of $O(height)$ or 512 elements in our system. The granularity of this effort is very vectorizable, but much smaller than the recommended grain size proposed by TBB and Cilk for generic reduction. We use OpenMP in our implementation. We reason that there is a high likelihood that the thread spawn overhead outweighs the potential parallelization benefits of doing a sum reduction on 512 elements.

```
// Node Compute Code
float acc = 0.0;
#pragma omp parallel for reduction(+:acc)
for (int i = 0; i < 512; ++i)
  acc += input_layer[i] * weights[i];
output_layer[p] = acc > 0 ? acc : 0.01 * acc;
```

Enabling parallelization at this level results in a significant slowdown with increased CPU utilization. This can be attributed to being a bad granularity to enable the heavyweight parallelization that OpenMP offers. A set of threads being allocated for performing a reduction over 512 elements is nonsensical. Additionally, the reduction nature of this computation enforces a strictly sub linear speedup. Finally, this code is a great candidate for vectorization.

Threads	Speedup	CPU Utilization
1	0.34	5
2	0.35	18
3	0.39	31
4	0.40	42

Parallelization of Network Layers

Each layer has to perform $O(height)$ or 512 node computations. This step is embarrassingly parallel!

```
forward_propogation(NetworkLayer &layer, vector<float> &↩
    input_layer, vector<float> &output_layer)
{
  // Layer Compute Code
  #pragma omp parallel for
  for (int p = 0; p < 512; ++p)
  {
    vector<float>  weights = layer.getWeights(p);
    float          bias = layer.getBias(p);
    // Node Compute Code
    ...
  }
}
```

We see parallelization at this level resulting in a positive speedup! As we increase the number of threads available we see approximately a 3.1× speedup for 4 available processors!

Threads	Speedup	CPU Utilization
1	1.00	5
2	1.91	18
3	2.66	31
4	3.09	43

Parallelization between Network Layers

The network as a whole has to perform $O(depth)$ or 7 network layer computations. This can be reasoned about as a stencil where each node depends on the entire prior layer. So we we reason that we can use the recurrence pattern to ripple the computation through the network to achieve some parallelization. We notice that this recurrence is actually fully realized by the "parallelization of network layers" technique from the section above.

Alternately, we notice that we could use a staged pipelining approach to be able to actually compute layers in parallel. This would enable us to achieve $O(depth)$ speedup, but would also result in an added latency to get a network output for its provided inputs. We can emulate the potential speedup achievable by not-modifying the node weights and biases and using the map pattern between network layers.

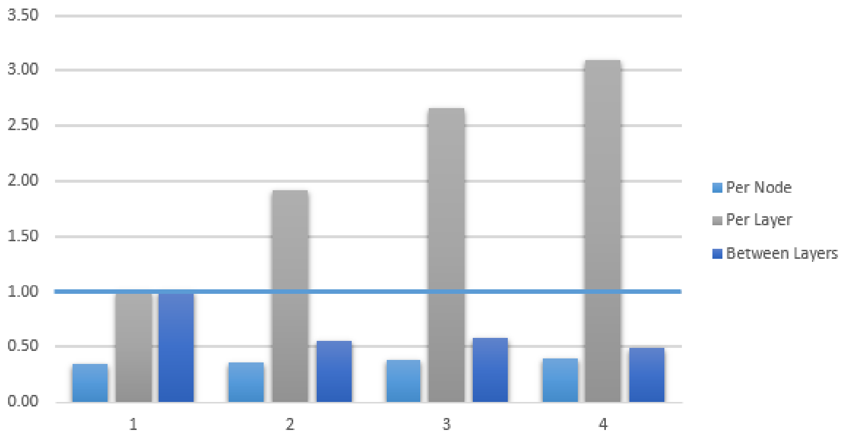
```
#pragma omp parallel for
for (int i = 0; i < 7; ++i)
{
  forward_propogation(layers[i], hli, layer_outputs[i]);
  hli = layer_outputs[i];
}
```

We see parallelization at this level resulting in a slowdown. We attribute this as following: either (1) there is a synchronization happening between loops or (2) the overhead of initializing scheduling and initializing so few threads is dominating the parallelization. The latter seems unlikely based on our investigation of Layer Parallelism seen in the prior section. Further investigation is required to determine what is causing this slowdown.

Threads	Speedup	CPU Utilization
1	1.00	5
2	0.56	18
3	0.58	31
4	0.49	43

Performance Discussion

Obviously, all of these parallelization schemes to achieve real time performance from our AI engine are orthogonal. We can apply any one, two, or all three of them as valid parallelization schemes. Unfortunately, after our preliminary investigation, we found that the node parallelization and network parallelization actually result in more overheads than parallelization and we achieve a slowdown from using these techniques! As such, we decide empirically that for our network of size on a consumer machine with four cores, only Network Layer Parallelization should be enabled to maximize performance. This is portrayed as the "Per Layer" series in the following graphic.



The y-axis is the speedup achieved, the x-axis is the number of processors made available for parallelization. "Per Network" is the speedup achieved parallelizing between network layers. "Per Node" is the speedup achieved parallelizing at the node level. "Per Layer" is the speedup achieved parallelizing at the network layer.