Nathan Tibbetts
CS 501R
David Wingate
19 December 2017

Crittersim

Introduction:

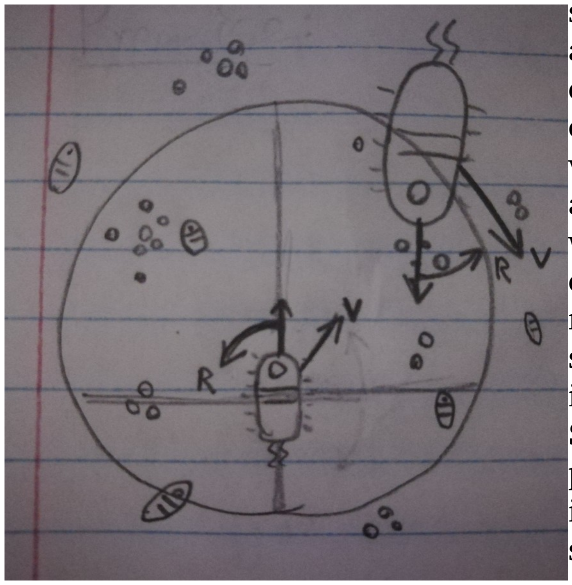
One of the most fascinating things to me is simulation. Whether it's planetary motion, plant growth, or a microbial environment, I love the idea of interacting with a recreated, detailed version of something real, with an ecosystem of interacting variables that you simply can't calculate, except by measurement of results. I wanted to recreate such an environment, something you might see under a microscope, only this time with creatures capable of learned behaviors. Little AI life-forms which run away, chase, forage, or hide.

This environment took me longer to create than expected, and it is a simplified version of what I would eventually like to produce, but I had the very interesting experience throughout of learning to plan a programmed environment in such a way as to simply be able to plug any desired machine learning algorithm into it, and even to test several side-by-side competitively with measurable results. I succeeded in creating this environment in a beginning form, and in planning out what algorithms to use. As I began to run out of time, I realized I would not be able to actually test out any of these algorithms in time, though I was ready to, and so I have done what I could to describe in detail what I planned. I do not, however, consider the exercise a waste, even with respect to deep learning, because I learned a lot about how to connect machine learning algorithms into a real-time environment which is constantly advancing by time-steps, which creates a few complications in how to code it up in a way that is not rediculously inefficient.

I will begin by explaining my efforts with the simulator, and then proceed to my plans on how I would implement the algorithms I intended.

Premise:

I wanted an environment which was to be richly interactive for its denizens. Hence, we have small creatures which can accelerate in the direction they are facing, slow down, and rotate (without acceleration); and which are affected by friction or drag, can eat creatures sufficiently smaller than themselves, and must flee anything much larger than themselves. Note that direction of travel is different from the direction the creature is facing, and so momentum plays a role. The variables available to each machine learning algorithm to access are, generally, their own states as well as those of nearby creatures. This can include distances or directions to other creatures, as well as raw coordinates if need-be, though only abosolute sizes, not relative, nor any kind of hard-coded interpretation of which directions might be advisable. Speeds are limited. When a creature eats, it gets a positive reward relative to the size of the food. When it grows to a certain size it is forced to split into five smaller creatures, and receives a larger reward. When



a creature is killed, it receives a negative reward. In order to make the system trainable, each species will probably have a single shared neural network, instead of each creature having a different set of weights. Hence any one instance of the class must update class-wide information as it is trained. This was part of the challenge.

Minor Goals:

- Learn to run from enemies.
- Learn to find food.

Major Goals/phenomena to accomplish:

- Learn to go after clusters of food
- Learn to attack just after a cell divides, perhaps following a larger creature from behind, out of sight.
- Learn to use bigger nearby enemies as bait to get away from even larger ones chasing you.

Simulator and Critter design:

It took a lot of time to set up the entire environment, planning all along the way how it would have to interact with algorithms which were shared by many instances. I succeeded in designing a base class for the Critter, on which is called a step function at every time-step. It keeps track of all the instances in order to execute each of their actions and check for collisions. The game design principles involved were not new to me, so it took much less time to implement than it would have otherwise. It did, however, take time to figure out how to render it, which I decided I had to do in order to actually prove that what I had coded for the critter was working. The step function calls class-specific _act() functions which would be the equivalent of sampling from a trained network. It gets states as input, and it gives actions as output. A policy. When needed, the step function also calls the _train() function, which back-propagates the reward, given the current (and probably several previous) states.

Some things learned:

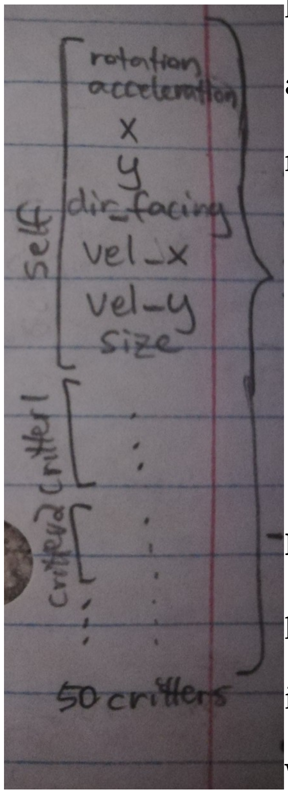
- How to use @staticmember functions, as well as class-wide variables, and some experimentation with @classmember functions.
- Inheritance in python - and some dealing with old-style classes (that was a few hours of debugging).
- How to use pygame, and some bits of pyglet, to render things. Still a few bugs to work out that weren't worth the time, here.
- How not to render things, for useful compuation times in training an algorithm.
- How to manage a dynamic group of instances.
- How to receive keyboard input with pygame, and run a time-step based event loop.
- How manage python imports/packages
- How to separate sampling and training in an object-oriented way.
- Conceptualized why a feed-forward network won't work in a dynamic compuation graph; ie, why we need RNN's.
- Conceptualized how to use convolutions for serialized, spacially represented state data, instead of on massive images.
- Conceptualized how to use an RNN for dynamicly-sized state data instead of dynamic time-steps

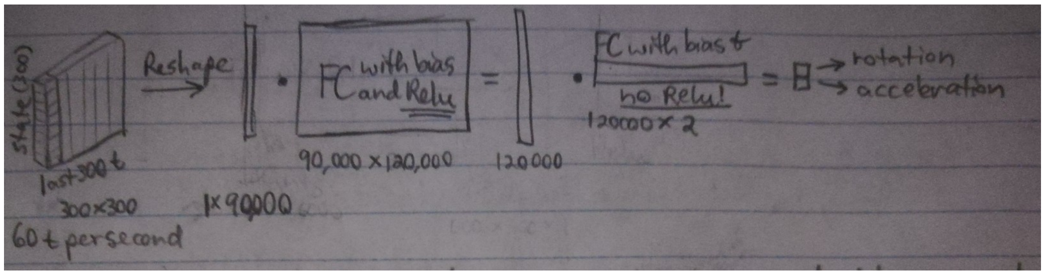
Thought Processes on What Neural Networks to use:

Feed Forward:

Problems:

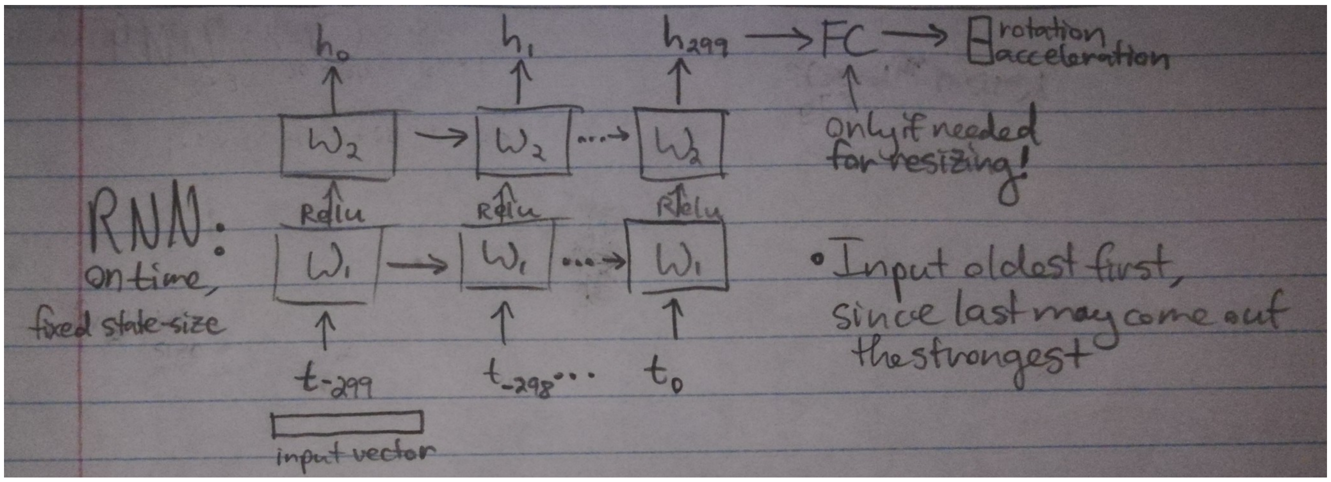
- Probably couldn't learn unless given multiple time-steps, since reward/cost is scarce - only happens upon death, eating, or reproducing.
- But then, if we use multiple time steps, we either need a set number (which would require truncating data and throwing out any live-times that were too short), or an RNN (LSTM or GRU).
- Assume a set number of time steps; but the number of enemies we can see is still variable, so the state data STILL isn't staticly quantifiable, unless we also truncate our knowledge of the state, or focus on the most important ones nearby.
    - Potential solution: Make the total number of creatures static, and all visible to every other one all at once.
    - But this defeats the purpose of having an ambiguously dynamic, realistic environment! Not life-like.





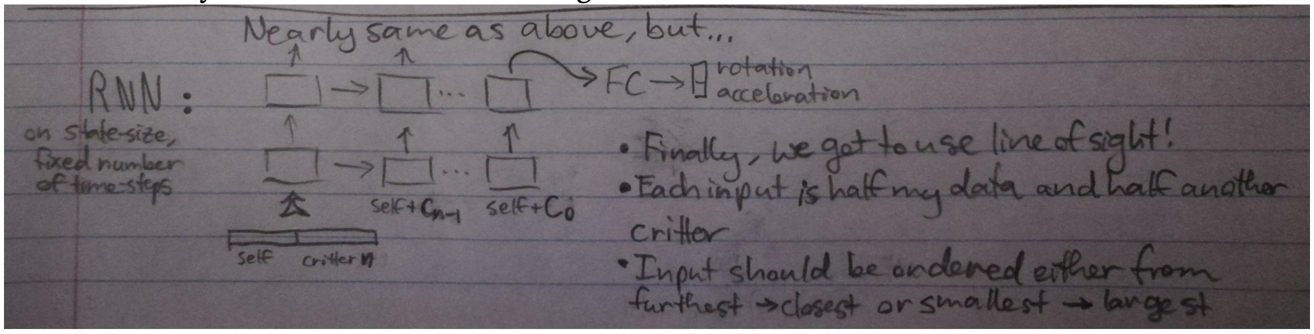
RNN on time-steps:

- Could have each time step be one of the recursive inputs. The problem of static state-size remains.
- Input the oldest information first, because the last-input information will probably dominate the system.
- Feed all in with the same loss; the loss the creature ended with. We can collect all information as a batch since the last trainable event.



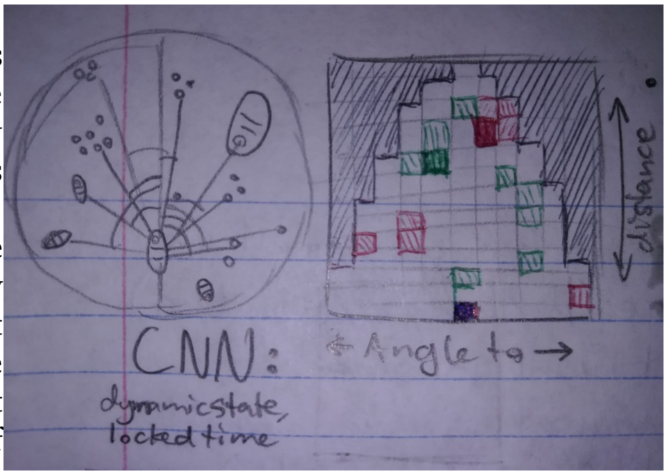
RNN on state-size:

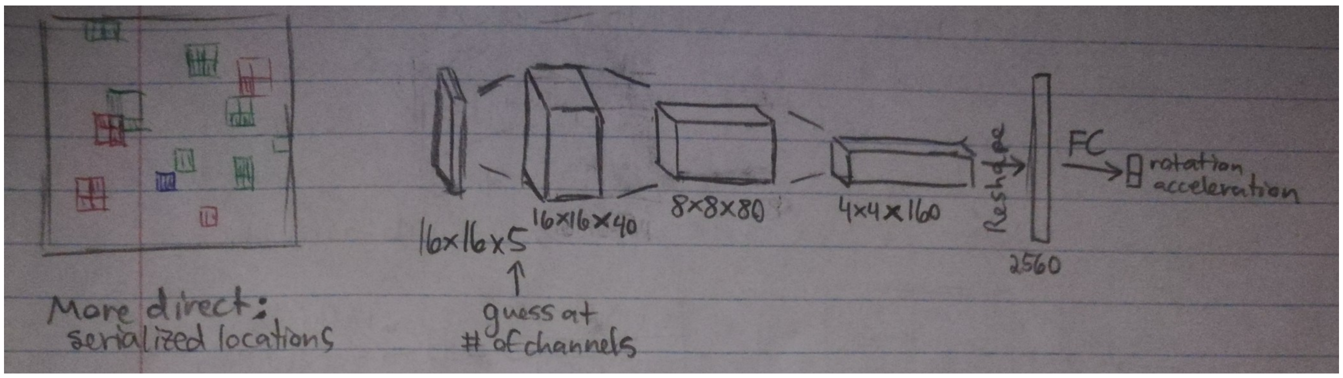
- If we concede a fixed number of time steps, we can recursively provide inputs that are sets of state information (per-critter basis).
- With each input, left half of vector should be own information, the right half the information of the other creature, else we wouldn't be able to tell which one is self.
- Or we could add an extra bit that is on when self, off otherwise, but the first method would probably work better.
- Finally we can limit the line of sight of the creature!


Nearly same as above, but...
RNN:
on state-size, fixed number of time-steps
FC
rotation
acceleration
self+Cn-1
self+C0
self
critter N
• Finally, we got to use line of sight!
• Each input is half my data and half another critter
• Input should be ordered either from furthest → closest or smallest → largest

CNN:

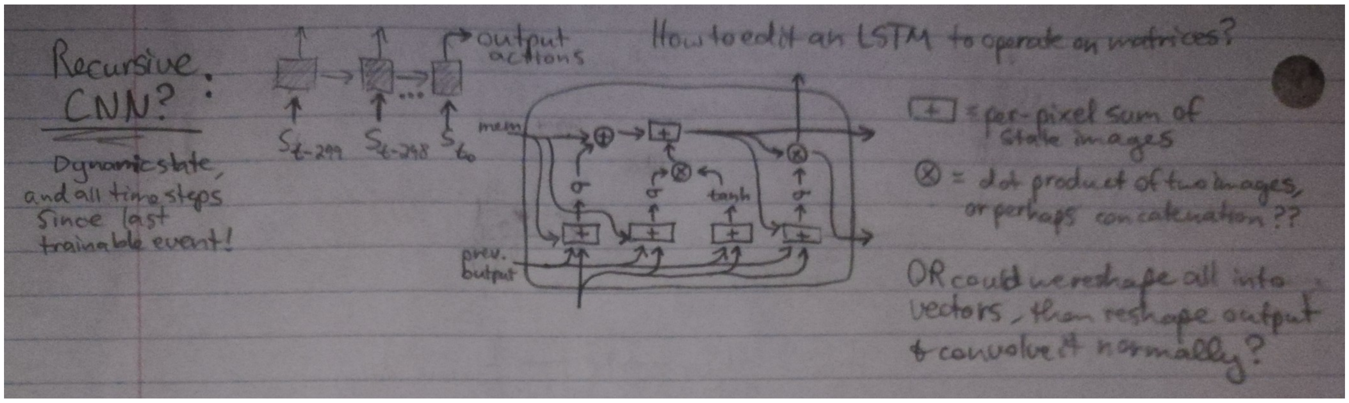
- If we store information about what is around us in a serialized way, in a matrix, we can pretend it is an image, even if it has many channels. We can then perform convolutions on it.
- Note: some state information would be difficult to encode this way, such as velocity and direction, though size can easily represent intensity. We could potentially use more channels for this, and or have each object occupying four representative pixels instead of one.
- We could encode either relative or absolute information this way.
- Finish with a fully connected layer.





Recursive CNN?

- My final thought is that if we could potentially create an LSTM whose internal operations are convolutions instead of vector functions, we might be able to produce a very effective algorithm which combines both dynamic state information and a dynamic number of time steps.
- Brief research online showed that similar things appear to have been done, but futher research would be necessary to confirm its feasability here.


Recursive CNN?:
Dynamic state, and all time steps since last trainable event!
S_{t-299}
S_{t-298}
S_{t_0}
output actions
How to edit an LSTM to operate on matrices?
mem
prev. output
σ
σ
tanh
σ
$\boxed{+}$ = per-pixel sum of state images
⊗ = dot product of two images, or perhaps concatenation??
OR could we reshape all into vectors, then reshape output & convolve it normally?

Conclusion:

I honestly don't think I have enjoyed any school project this much in years, and I am interested to see what I can do with it in the future. Thank you for everything, Dr. Wingate. I hope my efforts here are sufficient, despite my foretold lack of time to accomplish all of what I wanted to, especially in the area of actually implementing the machine learning algorithms. I do feel, though, that the things I learned through this project will be applicable and useful in the near future. It also helped me get a better grasp of which algorithms might be applied in what situations, and perhaps when it might be time to make a new one.

Time Log:

Total: 31 hrs.

Research on pygame/pyglet:
  30 min
  1 hr
  30 min
  30 min

Sprite Design:

30 min
20 min
30 min

Planning/Design for NN archtectures:
40 min
30 min
1 hr

Simulator Programming:
5 hrs 30 min
1 hr 30 min

Simulator Debugging:
2 hrs 15 min
2 hrs

Environment balancing:
30 min

Getting Visuals & Controls working:
1 hr 30 min
3 hrs

Randomized critter Programming:
1 hr 30 min
45 min

Player-controlled critter Programming:
45 min
45 min

Trying to connect parts for NN to use/debugging:
2 hrs

Report diagrams and specific NN plans:
2 hrs

Report Writing:
1 hr 30 min