Applying NEAT to Slime Volleyball | Builders Blog | December 2024

While exploring existing resources for NEAT implementation in JAX, I stumbled upon the highly relevant tensorNEAT library:

https://github.com/EMI-Group/tensorneat

I decided to base my slimevolley implementation on tensorNEAT, and I attempt to create a wrapper around tensorNEAT that can replace CMA as the solver in the `train_slimevolley.py` file. Since the tensorNEAT repo correctly evolves both the weights and the topology, I also created a `NeatPolicy` Class that inherits from `PolicyNetwork` that is compatible with my `NeatWrapper` Class.

My modified files are placed as follows within the EvoJAX library

```
evojax/algo/neat_wrapper.py
evojax/policy/tensorneat.py
examples/train_slimevolleyneat.py
```

I install both tensorNEAT and EvoJAX as dev dependencies in order to more conveniently use tensorNEAT for the slimevolley task, as tensorNEAT is not currently available as a module on PyPI. For more context on this setup, see README.md in the GitHub repository I am working out of (currently private, let me know if you would like me to share access):

https://github.com/Shivamkak19/evojax_tensorneat

Before beginning, I had to fix a minor bug in the `evojax/task/slimevolley.py` file. In the `display()` method of Particle and Agent Classes, I had to caste jax.numpy.float32 values to floats by calling `arr.item()`. See docs and the sample code diff below.

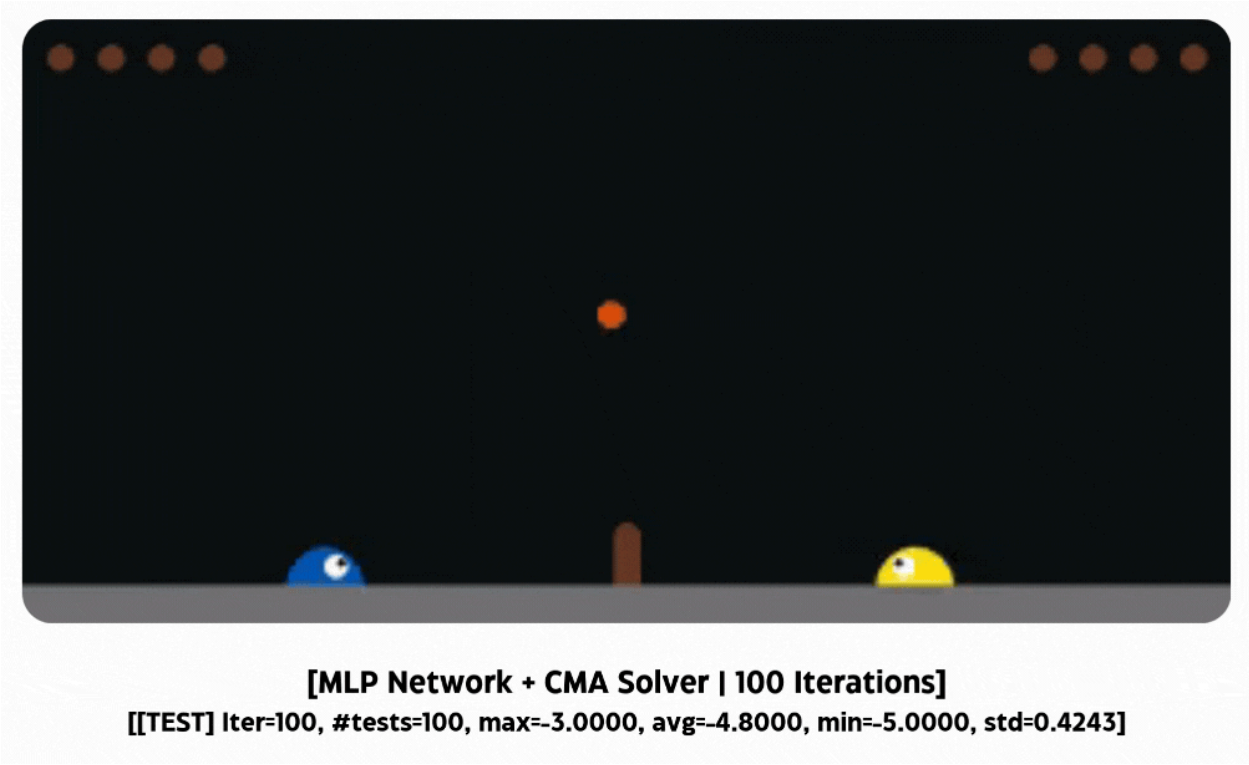
https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.float32.html

```
# BEFORE
def display(self, canvas, ball_x, ball_y):
    bx = float(ball_x)
    by = float(ball_y)
```

```
# AFTER
def display(self, canvas, ball_x, ball_y):
    print("DIAGNOSTIC DISPLAY: ball_x:", ball_x, "ball_y:", ball_y)
    bx = float(ball_x.item())
    by = float(ball_y.item())
```

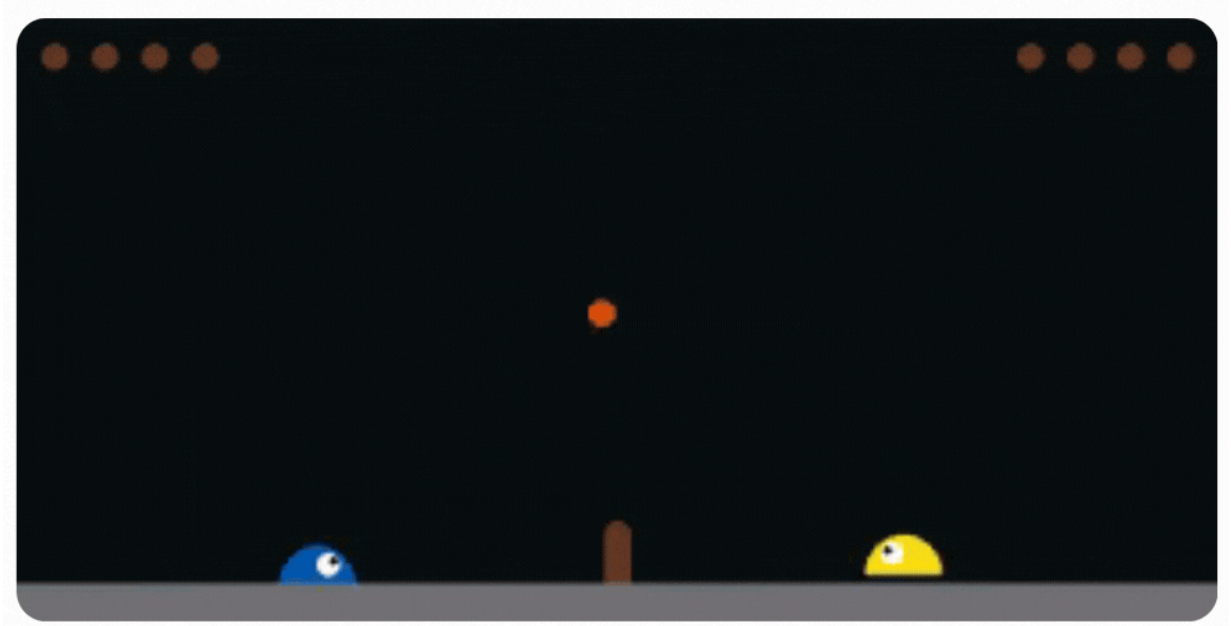
RESULTS & ANALYSIS

To start, below I have attached the baseline training with CMA solver and MLP after 100 and 500 iterations:



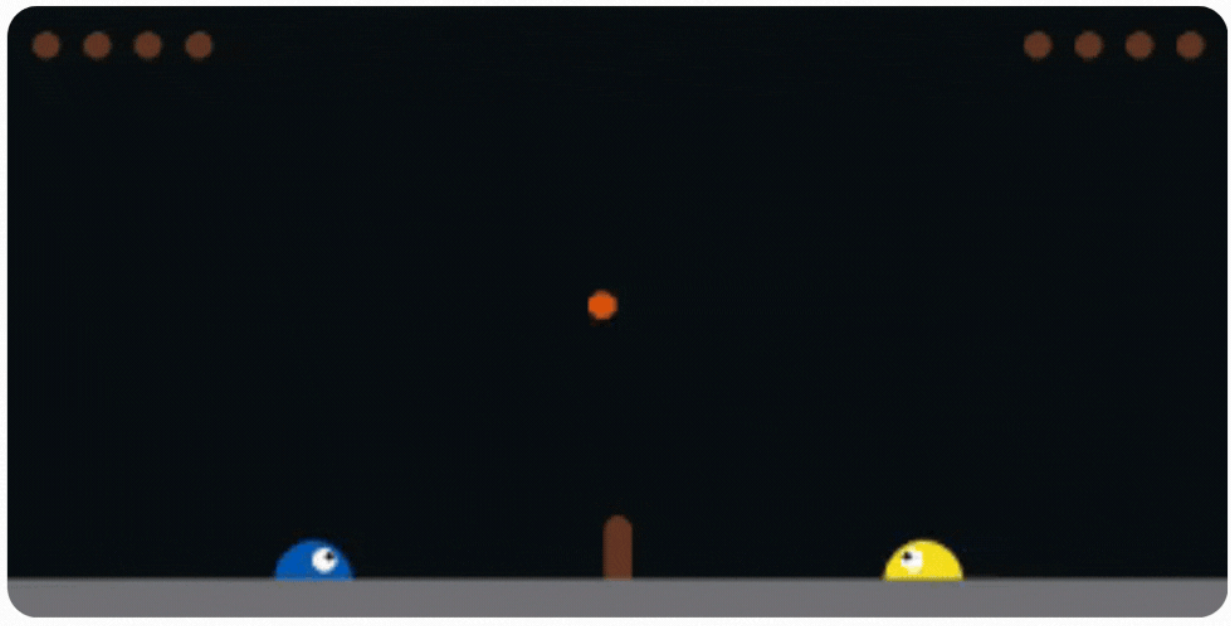
[MLP Network + CMA Solver | 100 Iterations]

[[TEST] Iter=100, #tests=100, max=-3.0000, avg=-4.8000, min=-5.0000, std=0.4243]

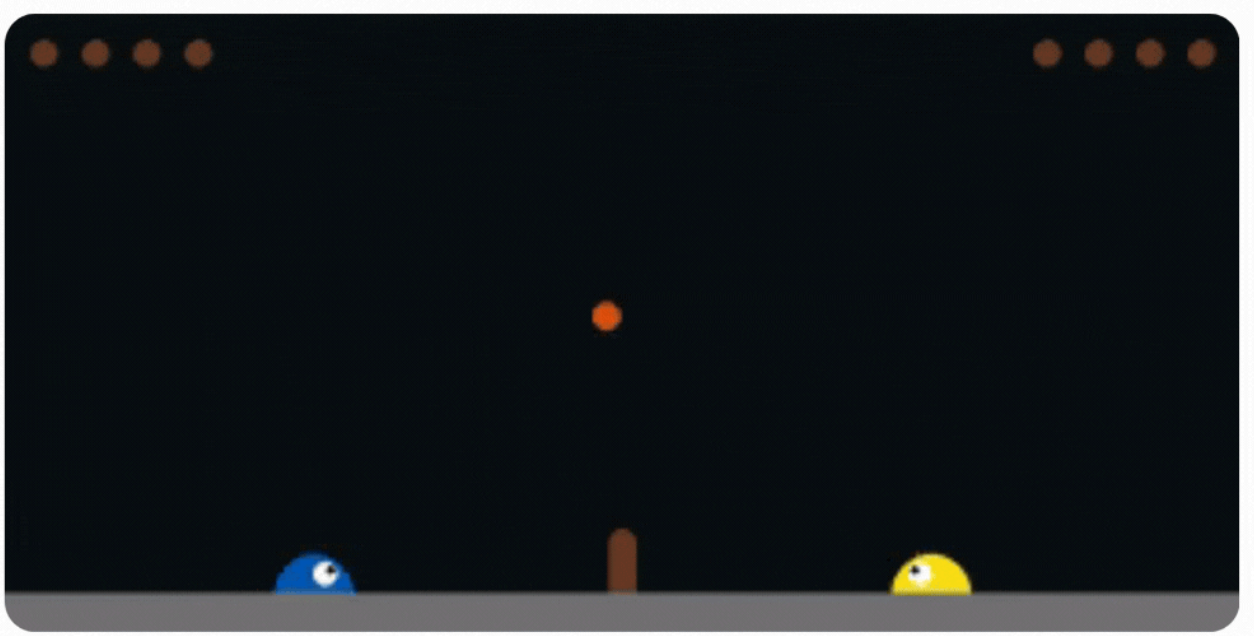


[MLP Network + CMA Solver | 500 Iterations]
[[TEST] Iter=500, #tests=100, max=2.0000, avg=0.2400, min=-1.0000, std=0.7499]

Next, I have included a visualization of the tensorNEAT implementation after 100 iterations of training and after 500 iterations.

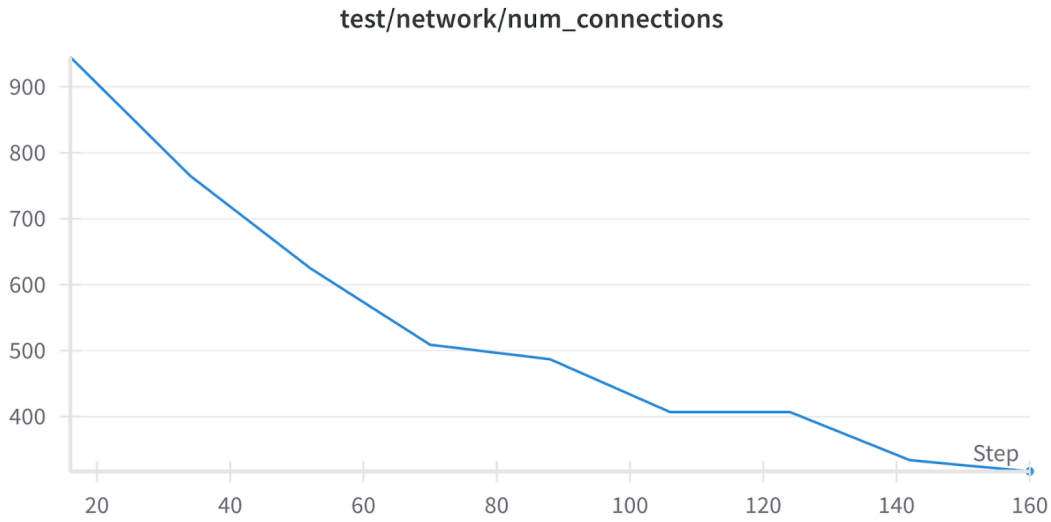


[NEATPolicy Network + NeatWrapper Solver | 100 Iterations]
[[TEST] Iter=100, #tests=100, max=-3.0000, avg=-4.8600, min=-5.0000, std=0.4005]

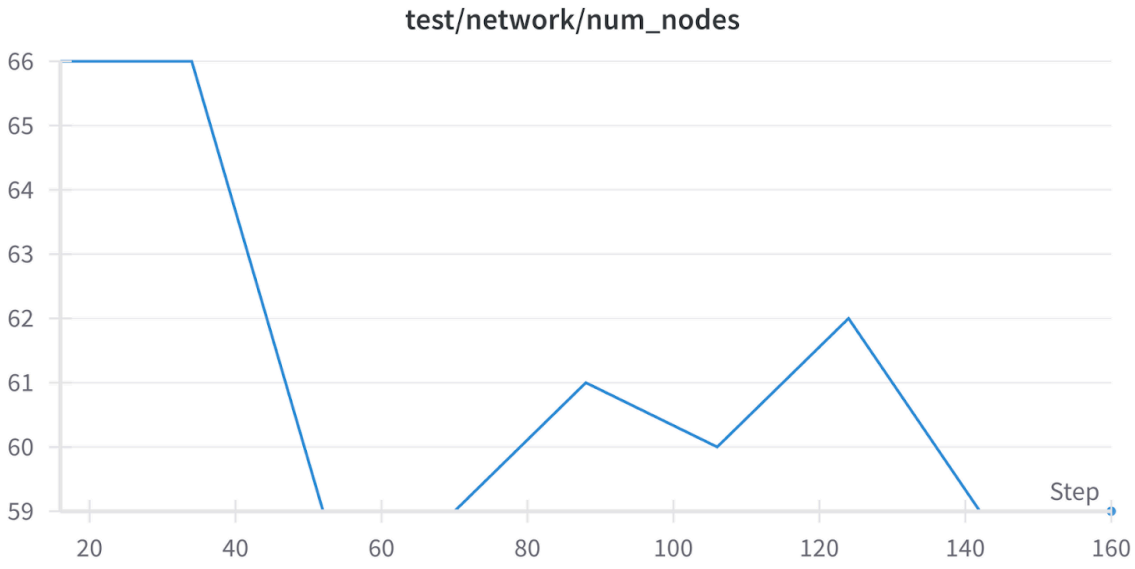


[NEATPolicy Network + NeatWrapper Solver | 500 Iterations]
[[TEST] Iter=500, #tests=100, max=-3.0000, avg=-4.8500, min=-5.0000, std=0.4093]

Heuristically, one can notice that the tensorNEAT slimevolley agent has learned basic movement in the initial phase of the match, but fails to replicate this movement past the initial game phase. This may be due to an overly dense initialization of the network architecture. Specifically, there was a clear tendency in my experiments for simpler network architectures over time with fewer nodes and fewer connections. This is observed in the two plots attached on the following page.

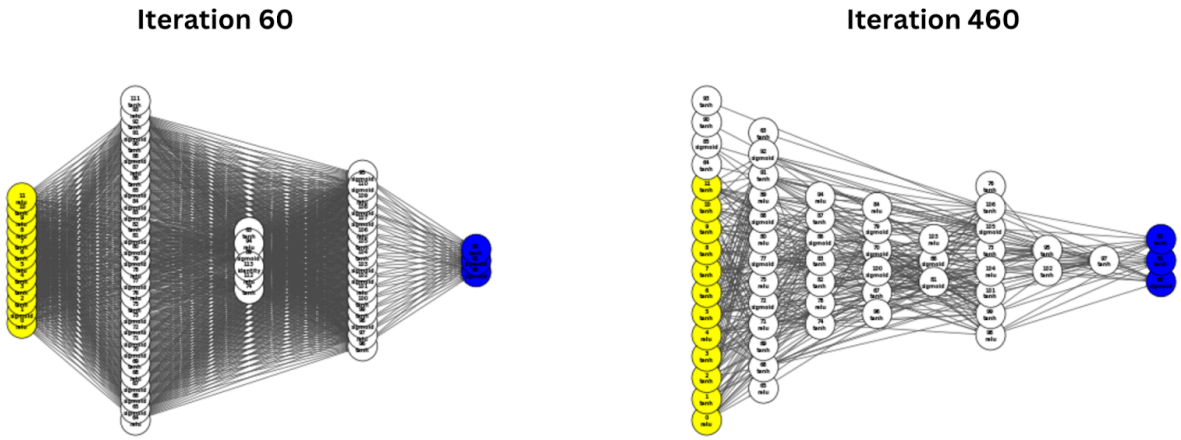


[tensorNEAT Network Connection Count over Iterations]



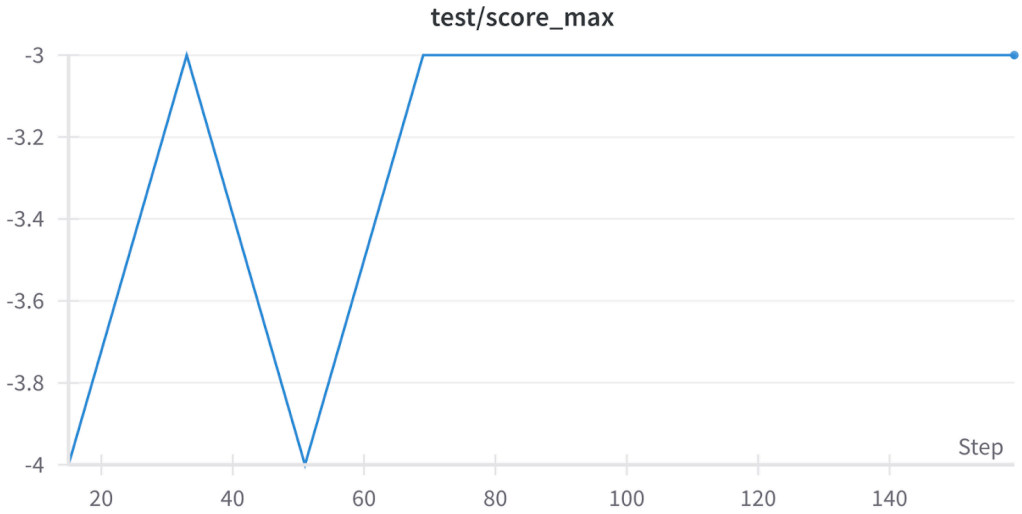
[tensorNEAT Network Node Count over Iterations]

This can be seen with a visual representation of the network architecture as well. Below, I have sampled the network graphs from two iterations in the training process. (DPI was set low in order to safely upload the images to wandb)

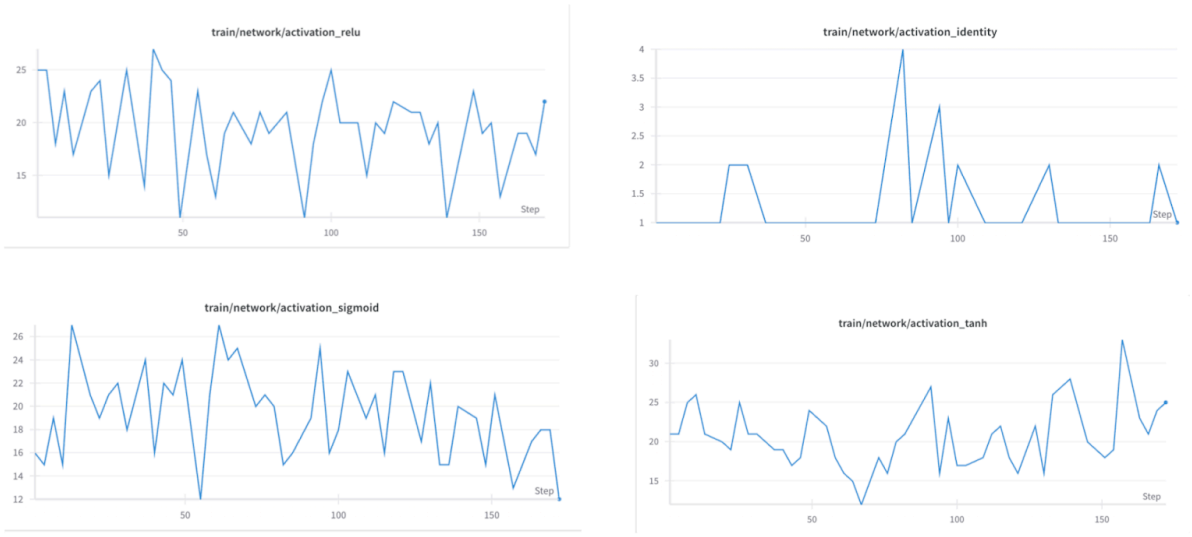


[Evolution of tensorNEAT Network Architecture over Iterations]

My results converge to a best maximum test score of -3. However, if you look closely this is not at all impressive because the lives lost by the internal agent were due to self-error on the part of the internal agent.

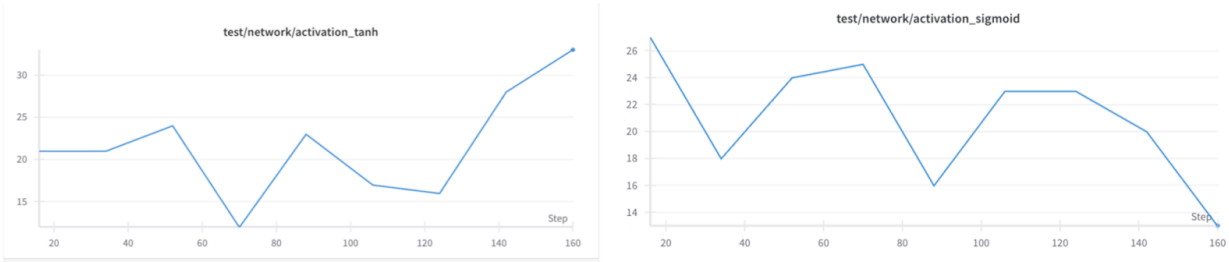


[tensorNEAT Network Max Score over Iterations]



[tensorNEAT Train Network Activations]

An interesting peculiarity of the network evolution is that a general downward trend in sigmoid activations is accompanied with a generally upward trend in tanh activations. These trends are far more pronounced in test networks (below) than in the train networks (above). This intuitively makes sense for the slimevolley task, as its action space of 3 is more suitable for a tanh activation.



[tensorNEAT Test Network Activations]

BACKPROP NEAT

I first used the given source code to first download the dataset, zipped below:

https://drive.google.com/file/d/1LIsxgInC0Lw83U1L-Z9bEdbKIsfS1YhK/view?usp=sharing

TensorNEAT already limits its activation functions to differentiable and mostly differentiable functions (RELU, ABS). See their function definitions in

`src/tensorneat/common/functions/act_jnp.py`

Specifically, the activation functions I am using include:

```
scaled_sigmoid
sigmoid
scaled_tanh
tanh
sin
relu
lelu
identity
inv
log
exp
abs
```

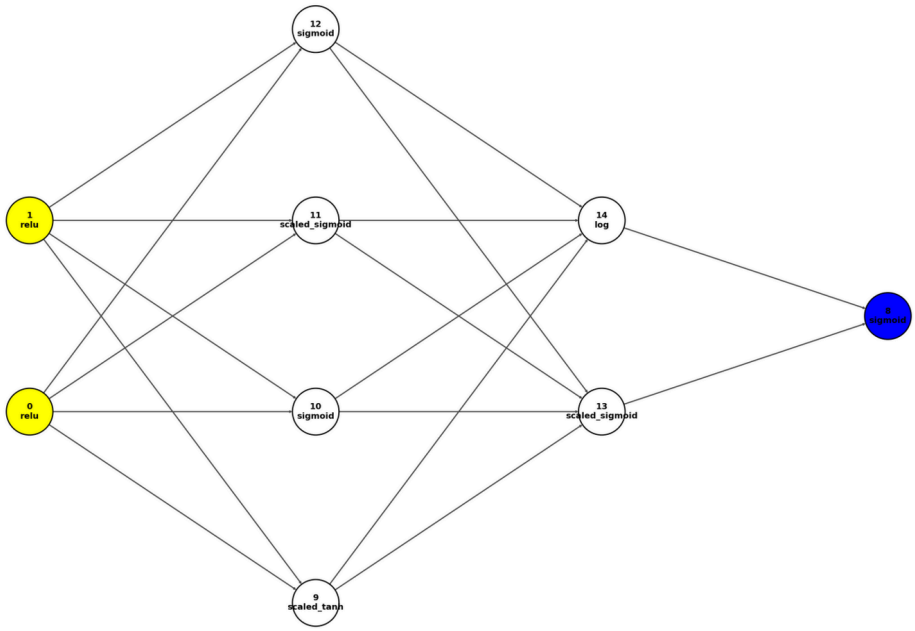
Once again, my implementation relies around tensorNEAT, except this time I no longer use the provided `DefaultGenome` in the source code. I define a `BackpropGenome` Class that inherits the `DefaultGenome` class but also supports loss computation to evolve weights. Specifically, `BackpropGenome` modifies the default implementation to compute gradients and update the weights via binary cross-entropy loss and an Adam optimizer. It retains `DefaultGenome`'s mutation and crossover for topology changes but replaces random weight updates with gradient descent in minibatches. The forward pass uses `ReLU`, or optionally `leaky RELU`, activation and includes safeguards for numerical stability (primarily clipping of gradients and prediction values).

In each generation the algorithm first obtains the current population and trains on each individual in the population, updating both nodes and connections. After training, each network is given a fitness score, which is fed back into NEAT's process of speciation, mutation, and crossover. The

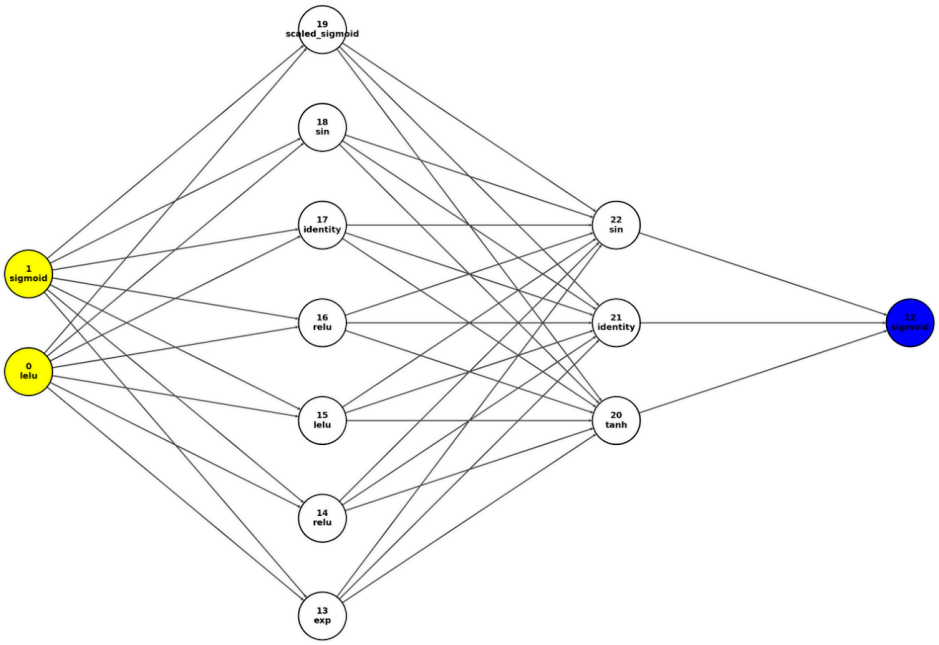
best performing network is tracked across generations with training being terminated upon reaching max generations, hitting the target fitness, or encountering 5 consecutive generations below a set threshold for fitness improvement (patience mechanism).

RESULTS

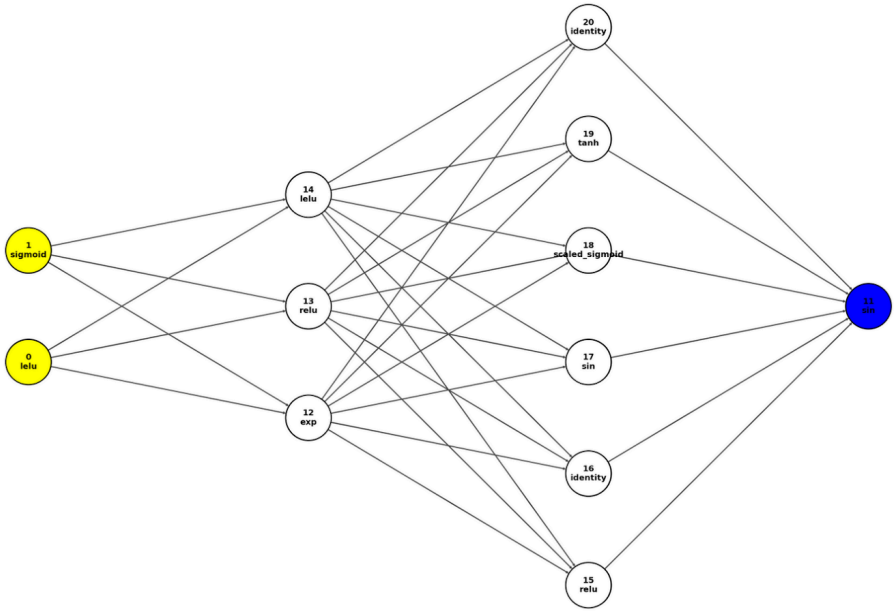
Below, I have attached several network visualizations of architectures that I personally found to be notable or interesting as well as the generation number in which they occurred. Note, I only ran the evolution process for a maximum of five generations.



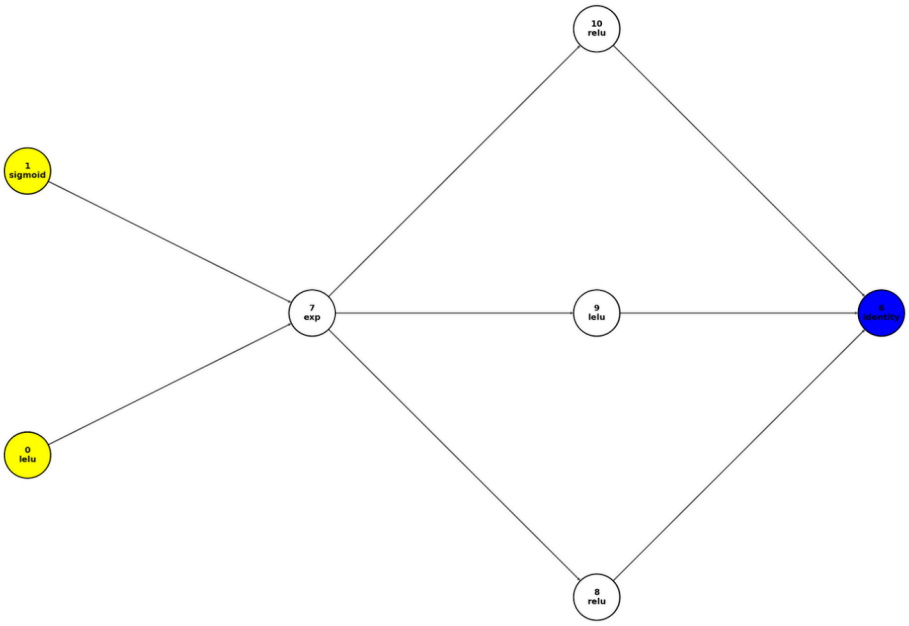
[Backprop NEAT Network Architecture | Generation 5]



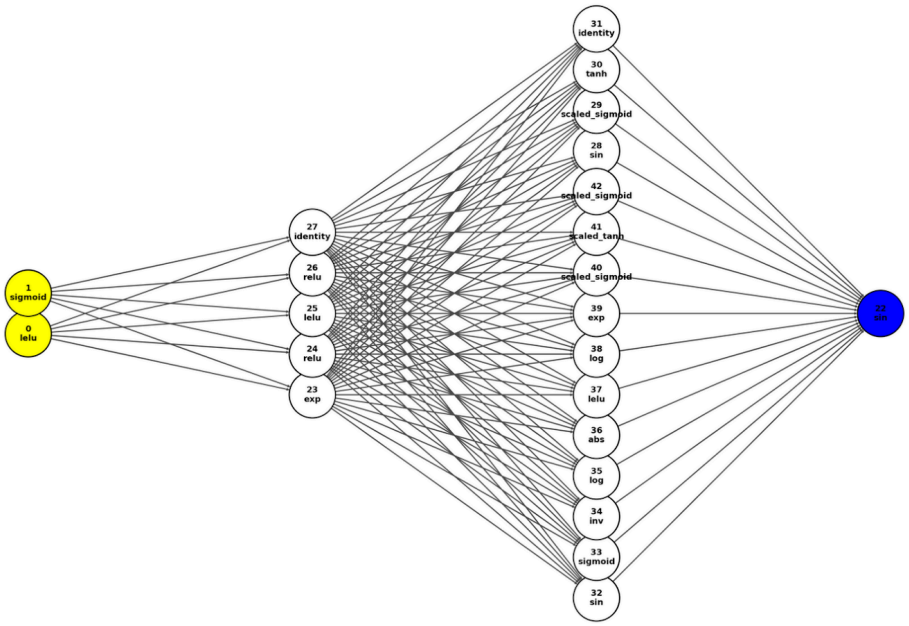
[Backprop NEAT Network Architecture | Generation 3]



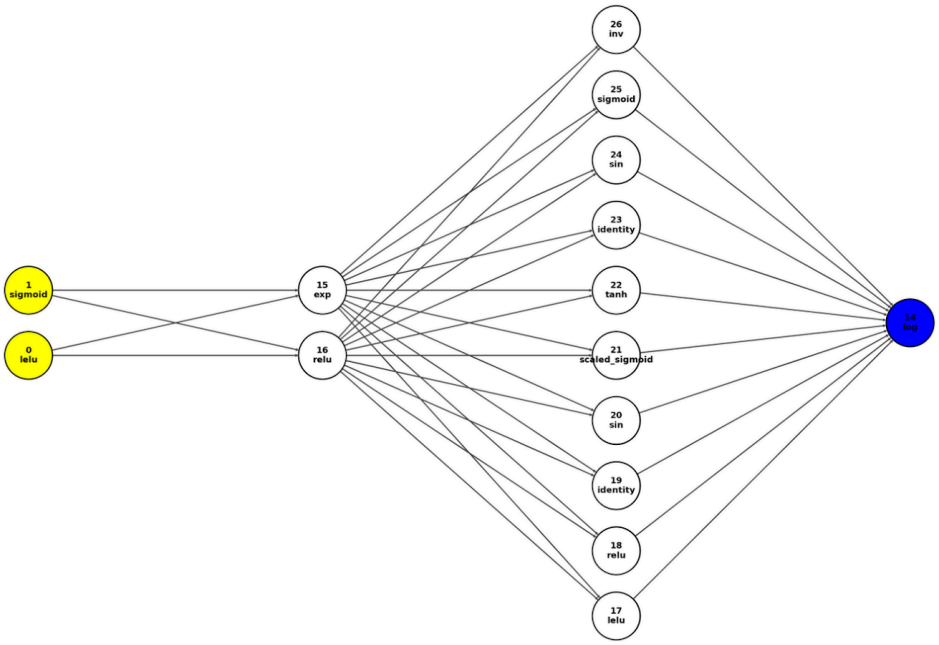
[Backprop NEAT Network Architecture | Generation 4]



[Backprop NEAT Network Architecture | Generation 4]



[Backprop NEAT Network Architecture | Generation 5]



[Backprop NEAT Network Architecture | Generation 3]

ANALYSIS & NOTES

The most consistent pattern across generations appeared to be a preference for wide layers as opposed to deep networks. This makes sense intuitively because binary classification types typically do not require many hidden layers at all. Especially for a simple task such as circle classification, even a single hidden layer will suffice if designing the neural net by hand. In this sense, the neural networks generated by my tensorNEAT implementation are generally speaking excessive and redundant. The choice in activation functions also varies greatly across generations and different experimental runs on the same dataset. This may largely be due to an early termination of the algorithm, as genomes may not have lived through enough generations to have converged on a universally good classifier.

I enjoyed this task, yet regarding the first half it is frustrating to be submitting a NEAT agent that does not actually defeat the internal agent. For this reason, I will be seeing the slime volley gumdrop creature in my dreams (nightmares) … until I have time to go back to this project and defeat it.
