DeepNash

…as explained by Dayne Howard

**Context**

For this explanation, I will refer to two papers:

1. Hennes, Daniel, et al. "Neural replicator dynamics: Multiagent learning via hedging policy gradients." *Proceedings of the 19th international conference on autonomous agents and multiagent systems*. 2020.
2. Perolat, Julien, et al. "Mastering the game of stratego with model-free multiagent reinforcement learning." *Science* 378.6623 (2022): 990-996.

Pull them up on the side so you can follow along. I have been using the classic game, Connect-4, as my environment for figuring this out, so I will explain each portion of DeepNash in that context.

[1] introduces “Neural Replicator Dynamics” (NeuRD), which is used as a key component in DeepNash. The abstract of [1] describes NeuRD as an actor-critic algorithm with a 1-line change in the policy update step on the actor. For Connect-4 Actor and Critic, what I have right now are two neural networks with identical structures, like this: A diagram of a computer

Description automatically generated

I realize this may cause some controversy, as the Critic network traditionally takes in a state **and** an action and then outputs a single Q-value, however my Critic network is taking in just the state and then outputting the Q-value for each action. Conceptually, I saw this as an equivalent option to the traditional first option, but it was easier to implement. So far this seems to work fine.

Notation I will use:

Here is the basic flow of the training process:

A diagram of a process

Description automatically generated with medium confidence

Now that the stage is set, I’ll go into the details of what DeepNash does in each of these three steps.

**Step 1**

Consider the tuple we are collecting: (State, action, reward, next\_state, done).

In the context of a two-player, turn based game, your opponent and the moves they make are just part of the transition function from one state to the next. So when you collect “State” and “next\_state”, these are both what a single player (ex: player 1) sees when it is their turn, from one turn to their next. “action” is just whatever move was selected by the player from their “State”. For Connect-4, this is the index of one of the 7 possible moves (columns). “done” is a Boolean indicating whether or not this was the last move made by the player. Debatable whether or not this Boolean is actually necessary.

The reward is where I will introduce a DeepNash innovation. Starting with your basic reward scheme (I am using -1 for losing, 0 for draw, +1 for win, applied only at the terminal state), each action will receive a reward transformation. Jump into paper [2], page 4, you will see in the caption of Figure 1 the “**Reward Transformation**”:



The way they present it here, it is set up for a game in which both players take an action simultaneously. Each player gets a reward based on their action (the term with the negative in front of it) and a reward based on their opponent’s action (the term with the positive in front of it).

For a turn-based game, page 10 of [2] shows how the reward is transformed in the paragraph starting with “**The R-NaD loop:**”. It looks like only one term (either the positive or negative) is applied per action. However, we want to maintain this as a zero-sum game and we are going to be collecting (state, action, reward, next\_state, done) tuples from both players’ turns. Therefore, the player making the move gets the term with the negative in front of it, and the other player gets the negative of that. The reason I phrased the underlined portion that way is that if player 1 and player 2 are not being controlled by the same agent, then . Let’s draw this out in a concrete example, where

= action by player 1 on turn #[odd]

= action by player 2 on turn #[even]

= player 1’s policy

= player 2’s policy

|  |  |  |
| --- | --- | --- |
|  | Player 1’s reward transformation | Player 2’s reward transformation |
| Turn 1 |  |  |
| Turn 2 |  |  |
| Turn 3 |  |  |
| Turn 4 |  |  |
| … | … | … |
| Turn 25 |  |  |
| Turn 26 (last move) |  |  |

Three notes I will make about this: First, you will notice that this reward transformation maintains the game as a “zero-sum” game, as each term has a negative and a corresponding positive. The exception to this is player 1’s first move, because that move does not exist in any of player 2’s transitions from any “state” to “next\_state”.

Second note is that reference [2] page 8 states that this reward transformation gives the system a convergence property.

Third note is that is not the same as the learning rate used in backpropagation. I used 0.2, same as in [2] page 32.

This reward transformation is added to the basic reward that is given to each player when they take “action” from “state”, as determined by the reward function. My reward function only applied a +1 for the last move made if the player won, and -1 for the last move the player made if they lost. If a player made an illegal move, I gave that move a -1 reward and ended the game.

**Step 2**

Now we will define the update rules for the Actor and the Critic. In each case, the goal is to calculate the *target* logit for each sample in the batch, use that to get the mean squared error (MSE) between the target and the actual output, and then use backpropagation (I used Adam optimizer) to drive the actual output towards the target.

Actor:

In [1], page 4, Equation 9 defines what the target should be for the Actor *if* you use MSE as your loss function. The notation between papers jumps around a bit, so refer to the notation I declared above and I will explain additional terms.

is the “advantage”, which is a dot product between the Actor policy, , and Critic Q-value, Q over all possible actions from state, s.

here is different than the used in the reward transformation. I believe that this is the learning rate traditionally used in backpropagation. However, when coding it, it is included in the Adam optimizer, so I just left it out of the equation (or let it be =1 in this equation).

Note: for all possible actions **not** taken will be just equal to the original logit, y. So the MSE will be zero for all actions that were **not** selected from state, s. I tried one experiment earlier on where I applied the equation to all actions, and it didn’t turn out well. I’ve made a few modifications and advances since then, so it might be worth revisiting.

Critic:

My critic updates were similar to traditional Q-learning, Bellman equation methods. I have a fixed-critic network which gets updated every so often. I calculate the *target* Q value for the action selected as follows:

Here, is the reward for taking action, a, from state, s. This includes the reward transformation. is calculated as before (dot product) but for the state after state, s.

I mentioned the fixed-critic network, which is commonly used for stability purposes. That gets used in the calculation of for . However, I tried also using the fixed-critic in the calculation of , but this yielded bad results. It was an earlier experiment, so may need revisiting, but I suspect that since I am only updating the Actor network once for every ~10 updates of the Critic network, then using the most current Critic network is appropriate.

Some coding notes: When you calculate the targets (either or ), you have to do those calculations within a “with torch.no\_grad():” block. Otherwise, your calculated gradients will try driving the targets towards the output values, at least in part, which you don’t want.

**Step 3**:

When you very first start training, you have an arbitrary , used for the reward transformation. However, this regularization policy has to be updated to be equal to the policy produced by your most up-to-date Actor every so often. I believe that the frequency of this update is expressed as on page 32 of [2]. I am using it as a set 5k, but I haven’t experimented much with it.

**V-trace:**

Page 29 of [2] gives equations for what is called the v-trace estimate. [2] doesn’t explain it or lay it out well, in my opinion, but it originates from other papers. Basically, this is a way of adjusting the gradients we calculated in step 2 above to better enable offline learning (learning from samples that were collected using a policy different than our current policy).

The real power of this comes when you use the **Infrastructure and Setup** section of [2], starting on page 41. Since I have not separated my Actors and Learners onto different resources (CPU/GPU) yet, nor have I employed multiple machines, I have not yet attempted to implement V-trace.

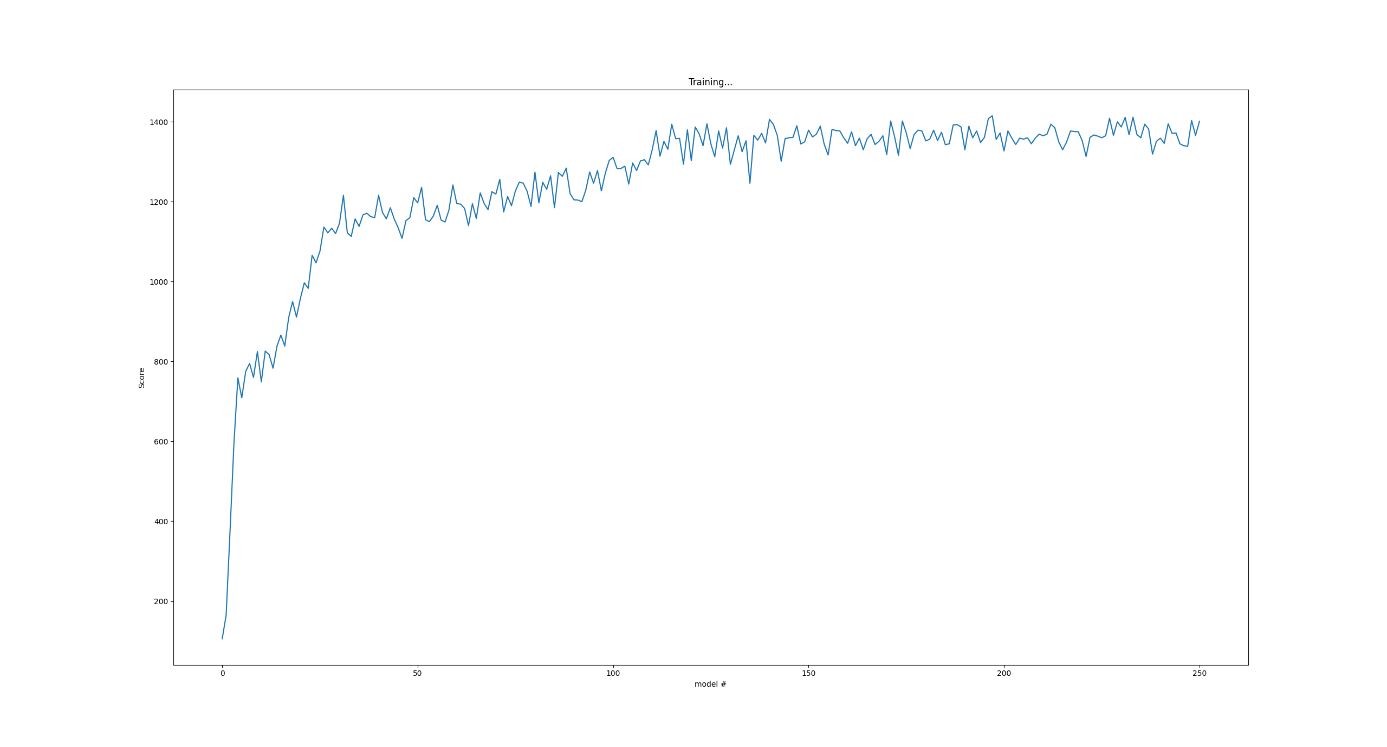
**Results and Conclusions**

I started on this project with just Q-learning. I played around with that for a bit before extending to NeuRD and then finally introducing the reward transformation. Lots of learning along the way, but I kept a log of what I changed with each experiment and how the performance changed.

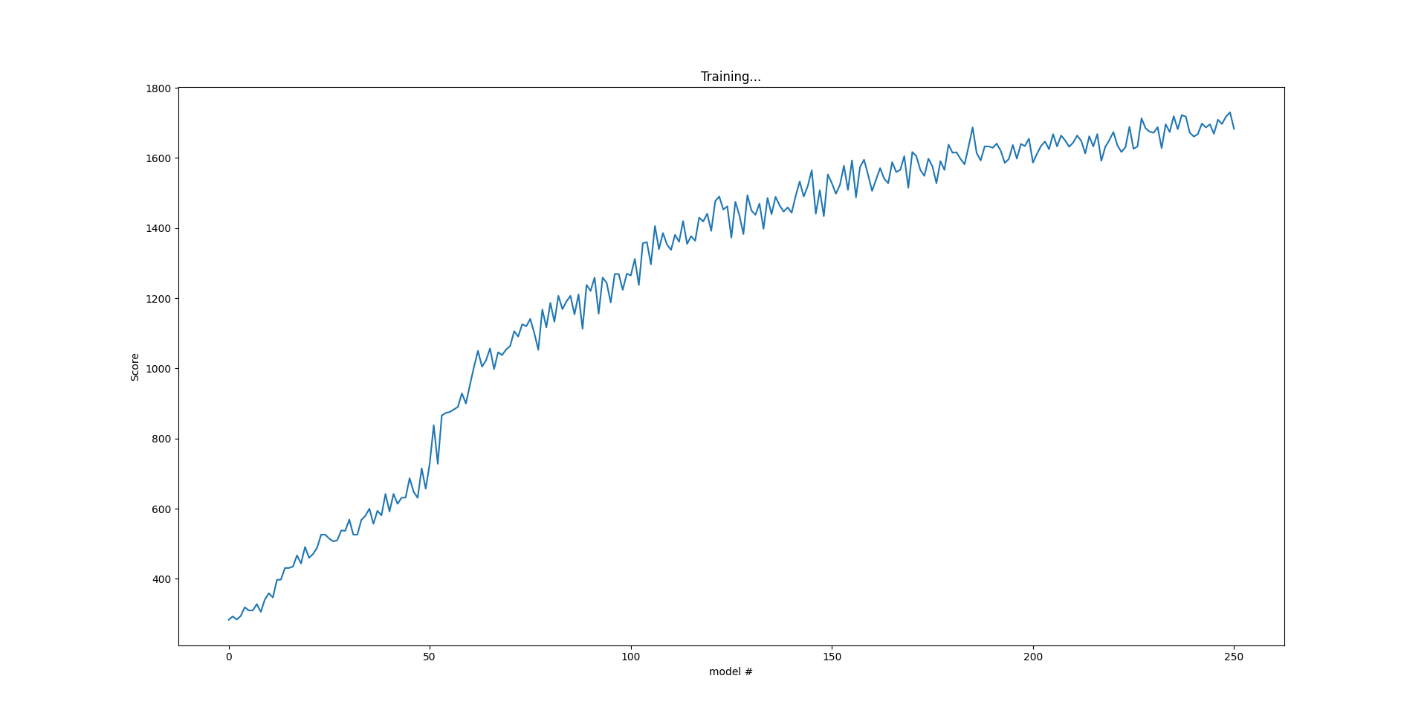
What I have found so far is this:

* Basic Q-learning (no separate Actor/Critic) gave pretty good rapid results, but the ability of the network plateaued quickly.
* NeuRD by itself was good, but not great until I introduced the reward transformation. (Calling this combo DeepNash, even though I didn’t include V-trace).
* DeepNash produced steady, stable improvements with an unconfirmed plateau better than the performance plateau of Q-learning.
* My best DeepNash network won ~67% of the time against my best Q-learning network.
* I have done almost nothing regarding hyperparameter optimization, and I have been using a shallow network (2 convolutional layers, followed by 2 linear layers).

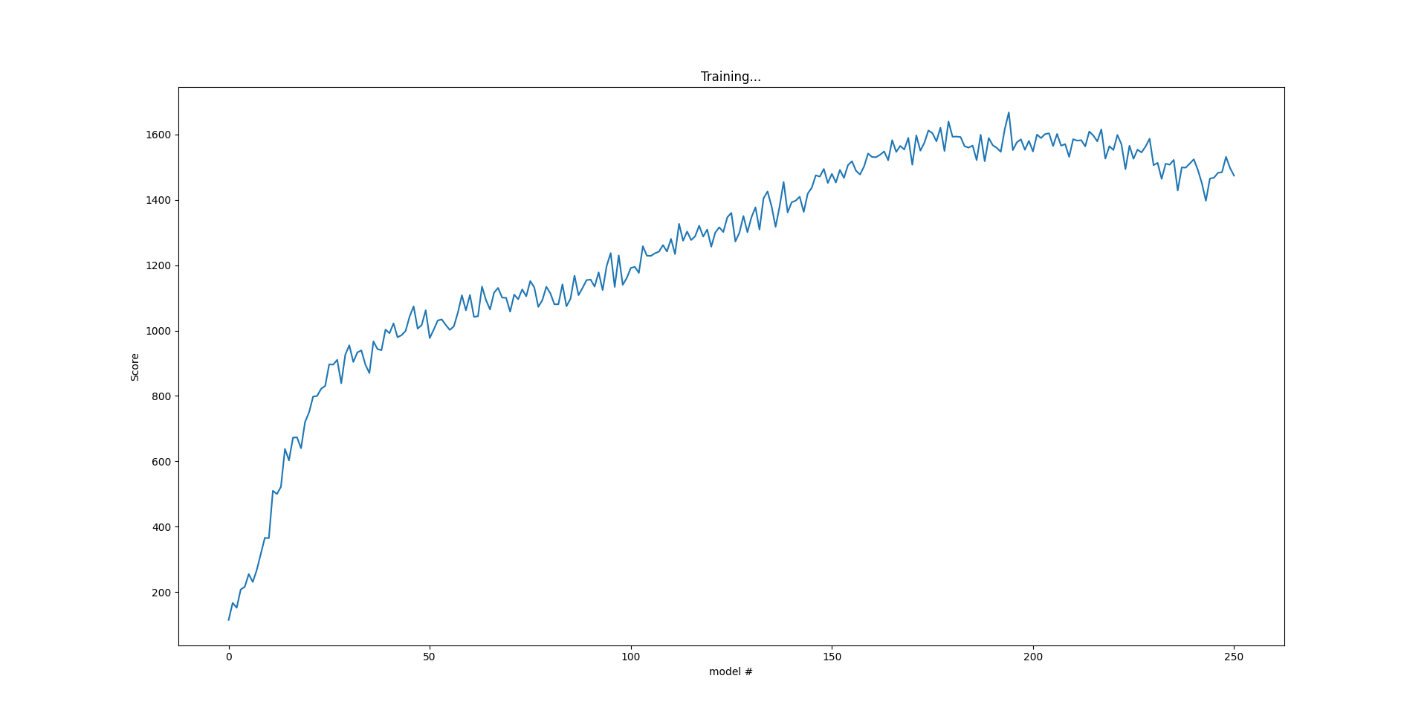
Here are some examples of the training curves. The x-axis shows 250 different save points of the network, and the y-axis is the “score” (number of wins) when playing round robin (10x) against itself at each different save point. Note the scale of the y-axis on each different plot, and how DeepNash improves much more on itself.

Q-Learning

DeepNash



DeepNash with 5x more training epochs (and 5x less frequent saving)



**Future Work**

* Explore methods of being “pickier” about which samples are trained on. Meeting soon with Dr. Themis Sapsis (my old thesis advisor at MIT) to see if a concept he has worked on called “Information FOMO” might apply.
* Explore methods of progressively adjusting the size and structure of the neural networks during training. I’m thinking something like Neural Evolution of Augmenting Topologies (NEAT), but at a larger scale.
* Explore methods of pseudo-online hyperparameter optimization.
* Implement it for other simple games, particularly imperfect information games, for verification purposes.
* Figure out and implement V-trace, so we can ramp up the compute power.