Introduction

Research question: Can deep Q-networks(DQN) be trained to play Atari 2600 games and can differential neural computers(DNC) outperform DQN in learn rate with previous training in similar games?

Neural networks are a common way to train an artificial agent to learn to predict categories given a set of related features. They are useful in a variety of supervised learning problems including but not limited to classification and linear regression. They essentially assign weights to several nodes whose outputs are passed through more layers of nodes until a final layer which outputs a prediction of what category or value a target has. This is however a very time consuming process and not easily generalizable but it does allow us to interpret a large search space of data. Neural networks are typically trained using simple gradient descent to

Minh et al.(2015) found a particularly innovative application of this framework in the Deep Q-Network (DQN) algorithm. Typically, games are solved using a method called reinforcement learning. The outcome, will be a policy which dictates what an agent should do for every possible state of an environment. However, some games have too large of action spaces or are stochastic, limiting the accuracy and efficiency of reinforcement learning algorithms alone in solving them. The DQN network use convolutional neural nets to implement an evolving strategy for predicting rewards for next game state based on a given action at a state. It stores these expected rewards and uses them as the baseline for training an agent in successive episodes of play. In

their paper they show that agents trained using DQN's are successful in playing 28 different classic Atari 2600 games better than expert human agents.

Environment and Methods

Environment:

We trained our models on the classic Atari game, Breakout, which were accessed via the OpenAI Atari simulator, which is made available free for research purposes with gym. The OpenAI gym environment provides an open source framework for developing and testing reinforcement learning algorithms. We selected games that are well suited for training algorithms to be generalizable across other games with similar action spaces

The gym environment and the Atari simulator can be installed following instructions on the OpenAI website (openai.com) where documentation for the gym environment can also be found.

Methods:

The flow of each algorithm followed very similarly from the original DQN paper, and involved several steps, as follows.

Preprocessing:

This was done per the method of Minh et al.(2015) in their original DQN paper. Because the raw Atari 2600 frames would add too much computation and memory requirements, the following steps were taken to minimize the complexity of input:

1. To minimize flickering which occurs as result of limitations in the original Atari 2600 display, first the frames of the video game are encoded by taking the max pixel color over value of frame for each frame and the one before it.
2. Extract Y channel (luminance) from RGB frame and rescale to 84 x84. Essentially this converts our video game input to black and white for improved efficiency in computing. This processing step is applied to next *'m'* frames so the Q-function has a trailing input.

Model:

The output of the preprocessing will form the input layer of the convolutional neural net. Following this there are three hidden layers that successively convolve the input image and then apply a rectifier non-linearity in the following order:

1. 32 8x8 filters
2. 64 4x4 filters
3. A fully connected layer with 512 rectifier units

The final layer is a fully connected linear layer that produces one output for each action.While previously other research has used history action pairs as input to the neural network, the DQN researchers take state representation as input instead because it allows the algorithm to obtain outputs for each possible action without doing repeating the forward space search for each additional action.

Training:

The details of the training algorithm can be found in the original DQN paper by Minh, et al. but we shall a general overview to provide a frame of reference. Each episode begins by preprocessing the initial state to provide input to the convolutional neural net. Then for T time steps, the algorithm progresses through the following actions:

1. Select an action either randomly at some probability ε or in a way that maximizes the reward function at the current time step.
2. Execute the action and observe the actual reward and new state.
3. Store the transition state and then sample a minibatch of transitions.
4. Use the sample to estimate an output using a value function if at penultimate gamestep, or using gradient descent otherwise.
5. Periodically update Q function

Originally we had intended to test implement and test DNC as well however upon implementation found that the training time to find the correct hyper parameters alone would be enormous and not feasible in the given time frame for this project and we have allocated it to future research. The untrained code will be attached for review.

Results and Summary:

Unfortunately DQN was not successful. Although we trained for well over 2,000,000 episodes, it's performance did not improve. We have chosen not to show graphs of the training history as they essentially just show a flat line of low scores. We

did attempt to bug check and compare our algorithm with the present research but are still unable to obtain similar results. We intend to continue this work beyond this course and hope to publish an update in the near future.

Future work -- DNC

One of the most interesting models to come out recently is DeepMind's Differentiable Neural Computer, or the DNC. This builds off of their previous work with Neural Turing Machines, NTMs, and falls into a general class of Memory Augmented Neural Networks, or MANNs.

A Memory Augmented Neural Net (MANN) is very much what it sounds like. The idea is to connect a neural network to an external memory that it can read and write from. Recurrent neural networks do something similar in that they are capable of propagating information from previous inputs forward in time, but this memory is very much internal to the network, making it very vulnerable to accidental erasure or modification. MANNs on the other hand have a slightly more complicated relationship with memory -- usually with strong limitations on how much of the memory they can change. This makes the memory much more durable, enabling linkage of long term dependencies. And of course, the interaction with this memory must be end-to-end differentiable.

Neural Turing Machines (NTMs) were DeepMind's first attempt at building a general purpose MANN architecture [1]. The architecture involves a controller network -- which is usually either a feed-forward or recurrent neural network, a block of memory, and a number of read and write heads to interface between the two. As with any other

ML model, the controller has an input and an output given by the programmer, but appended to these are the interactions with the memory. The information retrieved from the memory by the read heads is appended to the controller's input, and a portion of the controller's output is used to control what is written to the memory and where in memory the reads and writes happen.

The most important feature of the NTM, as well as the key difference between the it and the Differentiable Neural Computer (DNC), which has largely replaced it, is the way they choose where reads and writes happen -- i.e. how memory addressing works. The NTM does this in two different ways -- based on content and based on location in the memory. Content based addressing is performed by emitting a key vector, which is compared against the vectors stored in the different memory locations. Memory locations with a greater similarity to the key vector participate more in the associated read or write. Location based addressing starts with the values of the previous addressing step and with the values generated via content based addressing, then uses information emitted from the controller to determine how to rotate the selection through the block of memory.

By using both of these addressing methods in conjunction, the NTM is able to both jump to a location in memory and move spatially through the memory. For example, it can jump to a specific location a, then move to a location 5 steps to the right -- just like indexing an array "a[5]", or it can jump to a location, iterate through each location to the left of it for 12 steps, then jump back to the first location -- just like a for loop. These analogies seem to indicate an ability to learn things very similar to the basic

structures I learned in intro to programming. In fact, it is not very far of a leap to say that it can be trained to write algorithms for itself; that is essentially the purpose of the architecture.

The DNC is very similar to the NTM, but they differ in how they handle location addressing and in the scope of their addressing [2]. Where the NTM uses spatial locations -- where memory locations are in the block relative to each other, the DNC uses temporal associations -- when memory locations were written to relative to each other. Not only is this a much more relevant and intuitive metric to address based on (easier to train), but also enables a more complex relationship between addresses. Where the spatial relationship is static and predetermined, this temporal location is dynamic and develops specifically as a result of interactions.

The scope of addressing is perhaps an even more important distinction between the two models. The NTM writes to the entire memory all at once at every timestep, only to widely varying degrees based on the addressing processes described above. The DNC on the other hand first decides if it wants to write at all, then uses its addressing to select a location. The DNC also prioritizes writing to unused locations over used ones, and has a mechanism for freeing unused / unimportant memory locations when memory starts to fill up. This increased specificity and restriction on writing is a major advantage in retaining long term memory, as it protects important pieces of memory from accidentally being overwritten, or from gradually fading away due to the noisy, global writes of the NTM.

So, the NTM is capable of writing algorithms for itself and much more easily maintaining long term memory, and the DNC is a general improvement on this (to the point that it seems the field doesn't talk so much about NTMs anymore), but what are the applications? There are A LOT of potential applications, but the one that we were interested in for our project (but unfortunately didn't reach implementation of) is *transfer learning*. The idea of transfer learning is to use experience gained from one task to get a head start in learning another.

Transfer learning is often difficult and intensive for normal, feed-forward or recurrent neural networks because for these simple models, learning a new behaviour necessarily involves comprehensive modification of their structure via backpropagation. Essentially a feedforward network is just a direct mapping from an input to an output. A recurrent net is a mapping from an input or sequence of inputs to an output or sequence of outputs. Thus to adapt one of these nets from one task to another, since the entire model is a direct mapping, the entire model must be modified.

The same structures that protect long term memory in Memory Augmented Neural Networks like the NTM and DNC may make them more suitable for transfer learning than the simpler models described above. Rather than map directly from input to output like the simpler models, a MANN can map inputs to locations in its memory and then from those or related locations in memory to an output. This means that instead of having to relearn the entire mapping structure from input to output upon changing tasks, a MANN can instead alter its interfaced memory to quickly rebind inputs to outputs.

This isn't speculation on our part -- a paper released in May of last year demonstrated this kind of learning with an NTM [3]. Each episode they presented images to be classified, and trained the model by backpropagating its error as normal. Between episodes, the specific images presented were changed, but not just the images -- the classes AND the mapping from images to classes were also shuffled, so that the task was new each time. To do well in each episode, the model had to store the image and class mapping in its memory, then on seeing another image it compared against all the images stored in its memory to find which it fit best. Thus it learned to learn significant information about what constitutes a class from even a single image -- this is what the authors called "one-shot learning."

An extremely interesting study (to be honest it was our *over-reaching* goal) would be applying this kind of learning to reinforcement learning and the Atari suite. The model could be set up to learn to play a new game each episode, just as the model in the one-shot learning paper learned to classify a new set of images each episode. Each episode could consist of a number of trials on a specific game, and the reward for the episode would be the maximum score it achieved. Every episode would be a different game, or possibly the same game with slightly different controls. This way, the model wouldn't be able to learn to play just by adjusting the weights of its controller, like a simple feed-forward or recurrent net would. Rather, the model would be forced to write memories describing what its actions do and the rules of the game into its interfaced memory. It would have to use algorithms written in memory and adjusted over time to fit the specific game it's playing.

It would have been incredible if we actually successfully ran this, and so much more so if it worked, but we realized, that at least in the time we had available in the semester, it would be infeasible. The experiment assumes of course that the model would be capable of learning to play any single game well at all, which we found out is a far from trivial assumption. It would also quite likely take an extremely long time to train, even on a very powerful computer. The DQN model that we created to replicate DeepMind's Atari performance never really learned to play, and even if it did eventually, it seems it would've taken a very long time. The model I described for the transfer learning experiment is far more complex, both in terms of writing the code and in terms of computation cost, and the learning task is far, far more difficult, so we realized that actually implementing it and getting results in the few days we had left would be basically impossible, especially on the hardware we have available to us.

However, I do intend to research this further and attempt an implementation at a later date. I will most definitely let you know if anything comes of it.

Bibliography

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., ... & Badia, A. P. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, *538*(7626), 471-476.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529-533.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D., & Lillicrap, T. (2016). One-shot learning with memory-augmented neural networks. *arXiv preprint arXiv:1605.06065*.