The University of Texas at Austin
Aerospace Engineering
and Engineering Mechanics
Cockrell School of Engineering

CAELUS LAB

TEXAS SPACECRAFT LABORATORY
THE UNIVERSITY OF TEXAS AT AUSTIN

April 2024

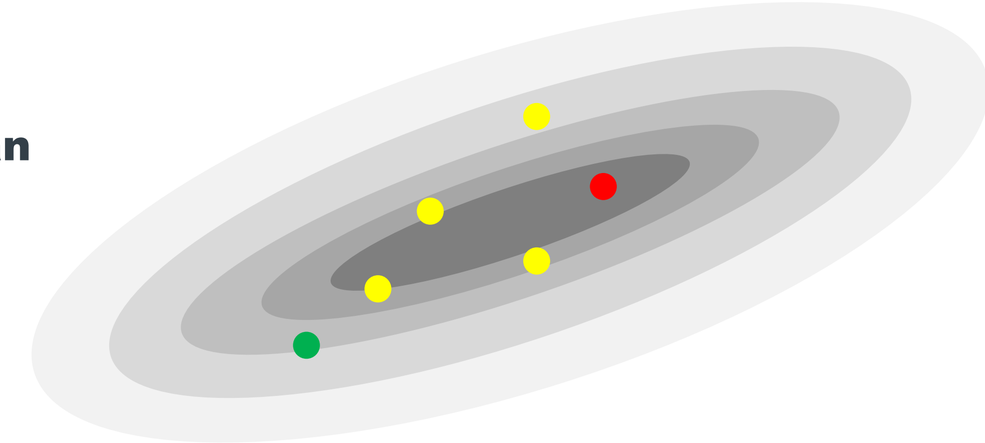
DEEP Q-LEARNING APPROACH FOR RAPID INITIAL ORBIT DETERMINATION

Brandon Lui
The University of Texas at Austin

Design Problem

Multidimensional search space characterized by a gaussian mixture model

Sensor-object pair



Sensor-green, object-yellow, desired object-red

How can we most effectively guide the sensor to the object?

- **Sensor Accuracy**
- **Mixture Model Propagation**
- **Search Space Noise**
- **Object Location Propagation**
- **Already Searched Space**

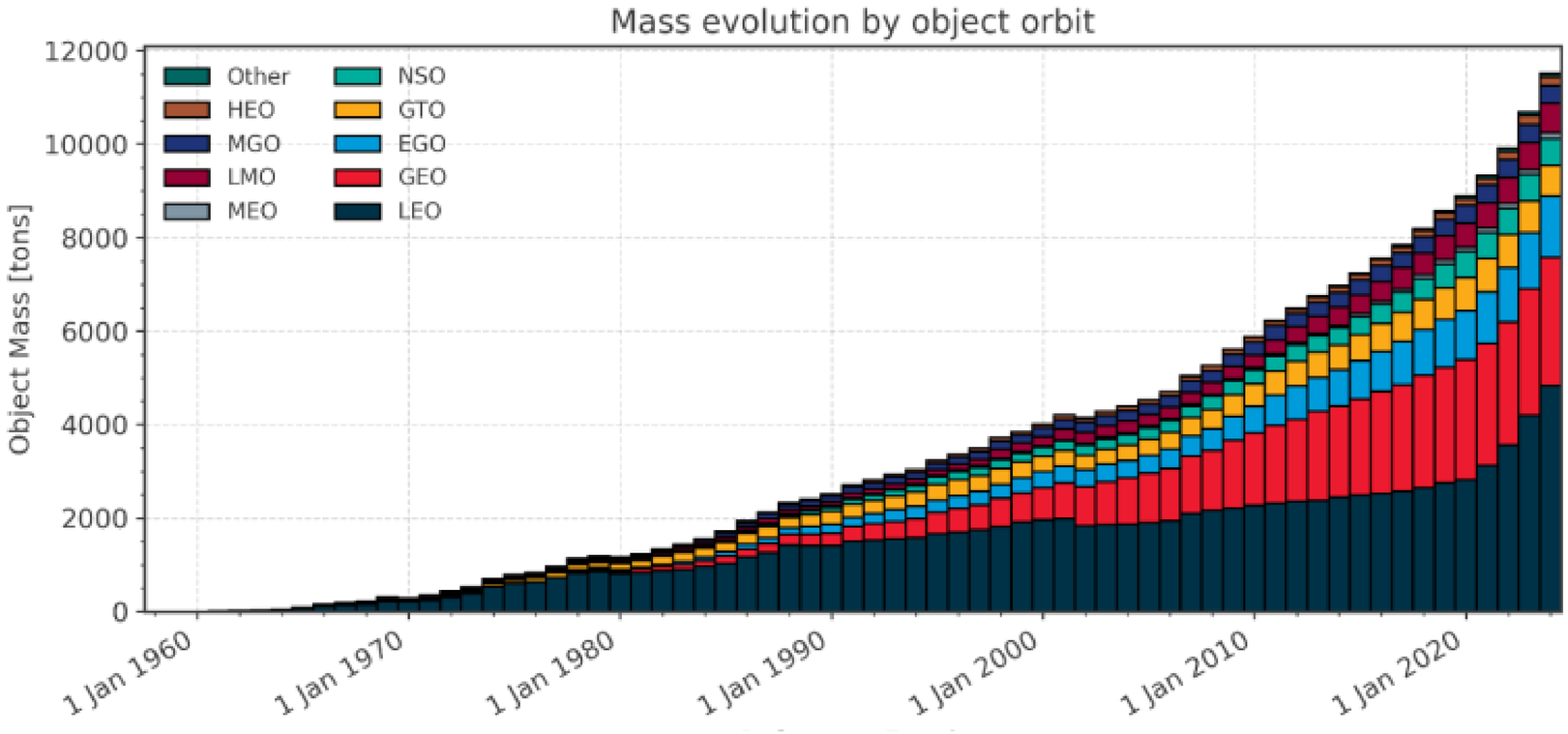
Background

Space is getting crowded with no signs of slowing down

Number of sensor resources can’t keep up

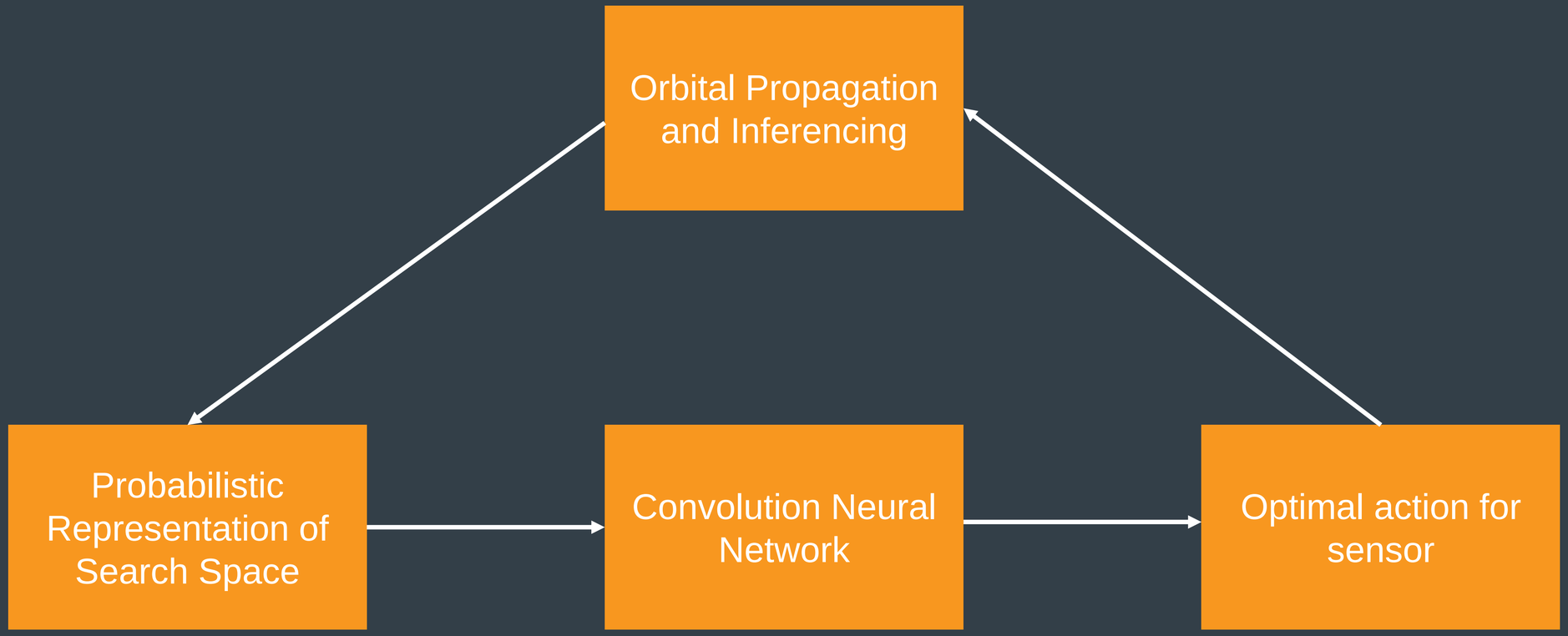
Cannot track every object simultaneously

Need to be smarter with our current resources



The University of Texas at Austin
Cockrell School of Engineering

Design Solution



Simplifications

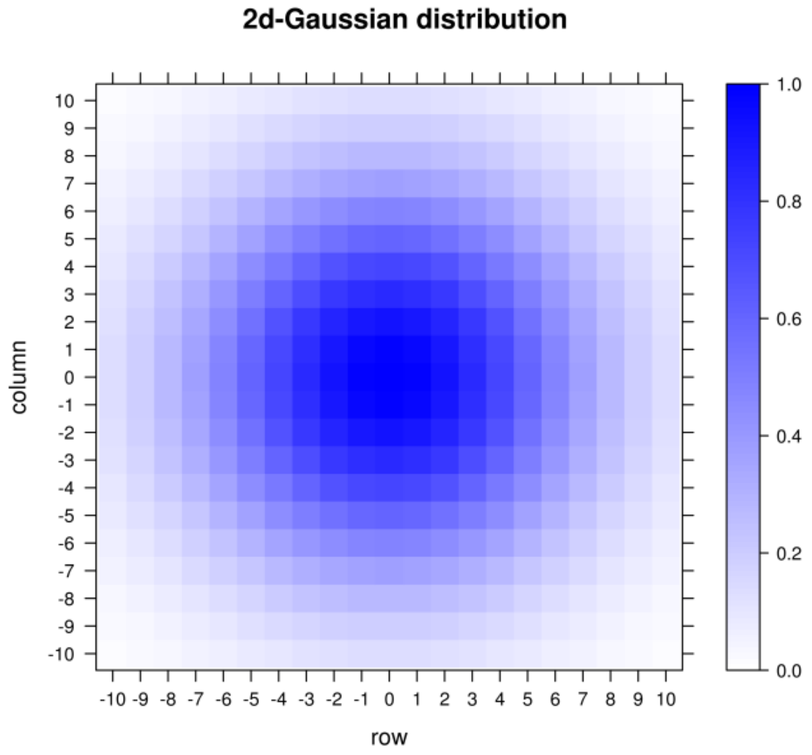
2-dimensional search space (84x84), represented by a 2-d gaussian pdf integrated over each cell

- **Characterization does not change in response to action taken or time**

Sensor never fails to detect object

There are no other detectable objects in the search space

$$f(x,y) = \frac{1}{2\pi\sigma_X\sigma_Y} e^{-\frac{1}{2}\left(\frac{x-\mu_X}{\sigma_X}\right)^2 + \left(\frac{y-\mu_Y}{\sigma_Y}\right)^2}$$



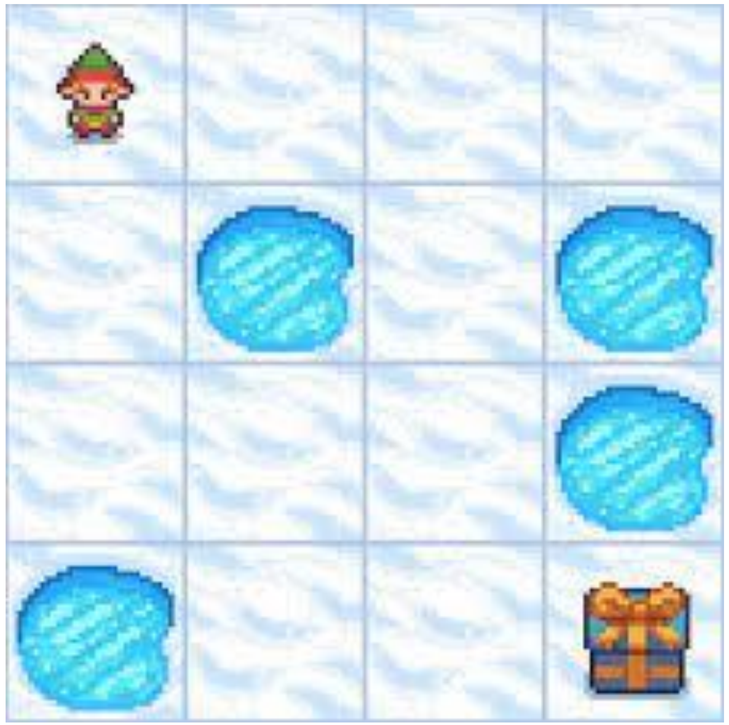
Reinforcement Learning Environment

Open AI's frozen lake environment is analogous to our problem

- **Discrete 2-d space**

Four directions of movement: up, left, down, right

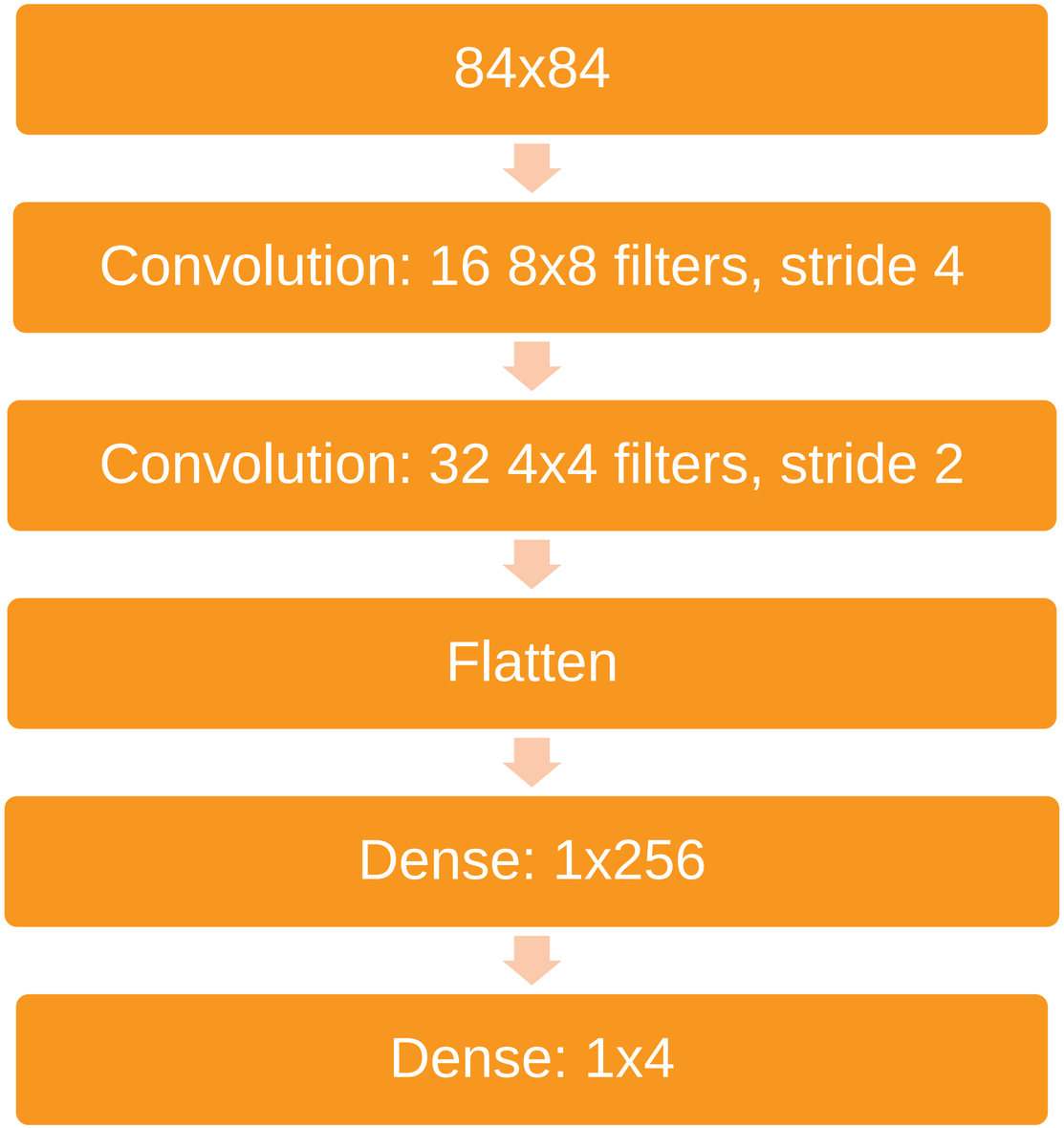
Saves work from building an environment ground-up



Network Architecture

Accepts an 84x84 pdf image with designated sensor-object cells

Returns an array of Q-values for each direction of motion possible

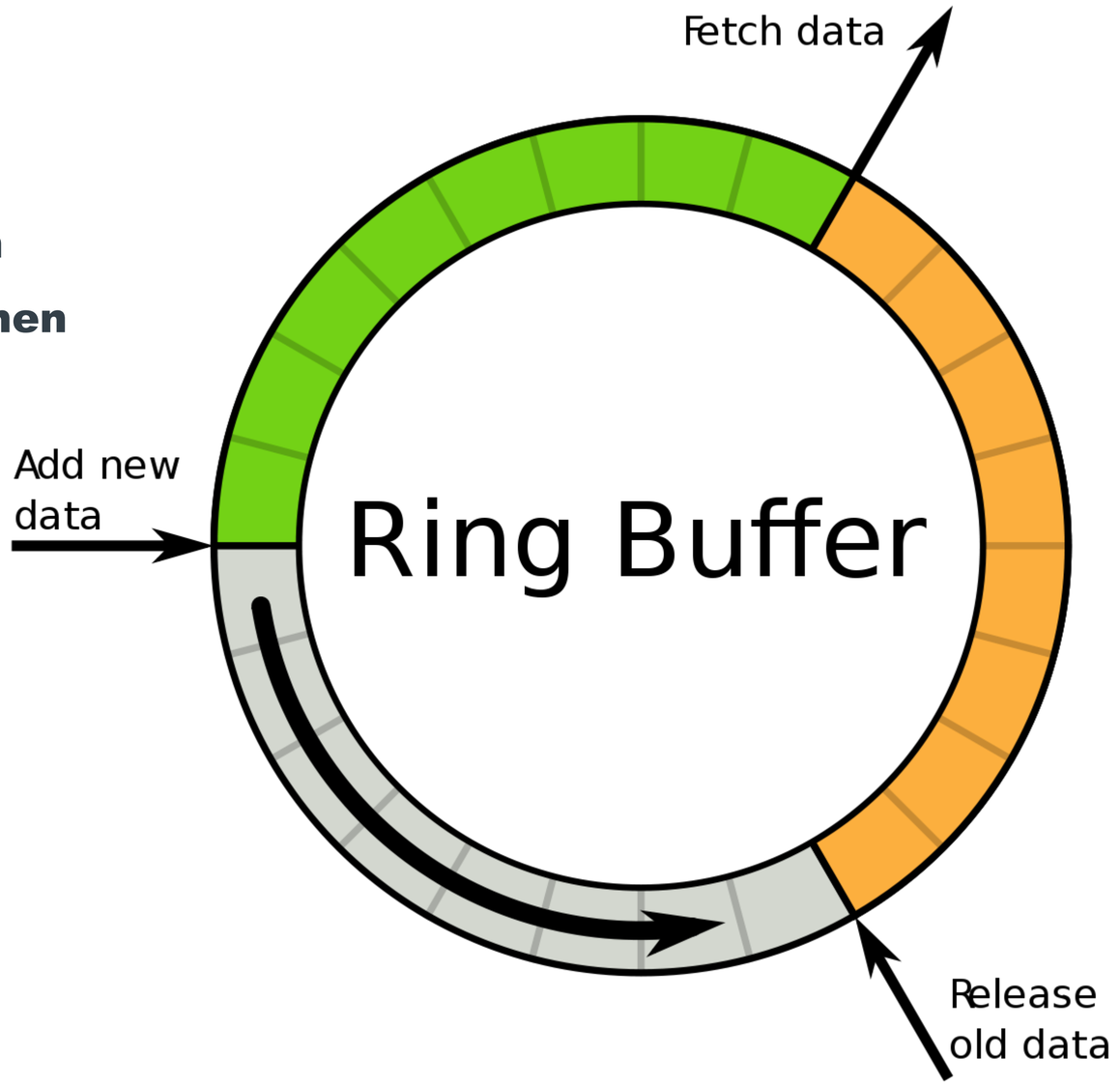


Experience Replay System

Motivation: Exposing the model to past states can prevent potential biases towards recent states when fitting the model

State:

- **PDF image before action**
- **PDF image after action**
- **Reward**
- **Action taken**



Training Regimen

Take a random action or action predicted by model

Compute reward based on probability model

Update input image to account for action taken

Store state of the environment in experience replay system

Sample experience replay system and fit to model

Exposed to a different pdf-sensor-object combination after successful detection or 200 steps

Probability of taking a random action decreases at a linear rate over 100000 actions beginning at 1

Reward is the pdf integrated over the cell the sensor is “stepping into”

Future Work

Theoretical:

- **Expanding dimensionality/scale**
- **Multiple detectable objects**
- **Non-perfect sensor**
- **More complex pdf image updates**

Practical:

- **Implementation on a real system**
- **Optimization for a real-time-sensitive system**

Look for other applications for this model

Design Problem

Uniform PDF: Input image is a uniform “color” arbitrarily set to non-zero value (.25) for each cell to see the updates to the image and pathing of the agent. Used on a model only trained on one step to prevent the possibility of getting a reward other than .25.

Using this uniform PDF on the newly initialized model results in seemingly random results. Model is initialized with random weights, maybe we aren’t affecting these weights enough? Initial output of model with uniform pdf:

```
[array([ 0.03611743, -0.03234098,  0.05994831,  0.00649316], dtype=float32)]
```

To be sure, shouldn’t we initialize model weights s.t. these 4 values are equal?

Output of model with unform pdf after stepping right (one of the right moves to do) we get:

```
[array([ 0.035591  , -0.03027848,  0.04781304,  0.00564717], dtype=float32)]
```

% difference: -1.457551105, 6.377357767, -20.24288925, -13.02894122

Decreasing the value of initially moving left, right, and up while increasing the value of moving down after observing a negative reward doesn’t make sense

Let's use the softmax activation function in our last layer which converts a vector of values to a probability distribution, which is what we want our last layer to be interpreted as.

Also, initialize the weights of all layers to some value, 0 in our case. In this way before any training, our model considers any action to be equally “good” and then any changes fitting the model can be ensured to affect the results proportionally

```
output = keras.layers.Dense(n_actions)(hidden)
```

Becomes:

```
output = keras.layers.Dense(n_actions, activation= 'softmax', kernel_initializer=init)(hidden)
```

Now before any training:

```
[0.25, 0.25, 0.25, 0.25]
```

And after taking one step up and receiving a negative reward for it:

```
[0.33333334, 0.33333334, 0.33333334, 0.0]
```

This makes more sense than before, but immediately setting the up action to 0 after one step is a bit drastic, just because the object is not in that cell doesn’t mean the optimum path doesn’t include that cell.

Updating the training for 1 million frames-ish

Before, our training was set to small number 1,2,3,4 of “runs”, which might not train the model on enough frames (10 million in the deepmind paper).

“runs” is how many times the model is trained to completion, finds the object or has taken 100 unsuccessful steps

Decided to train on about 1 million “runs” which is more than 1-million frames.

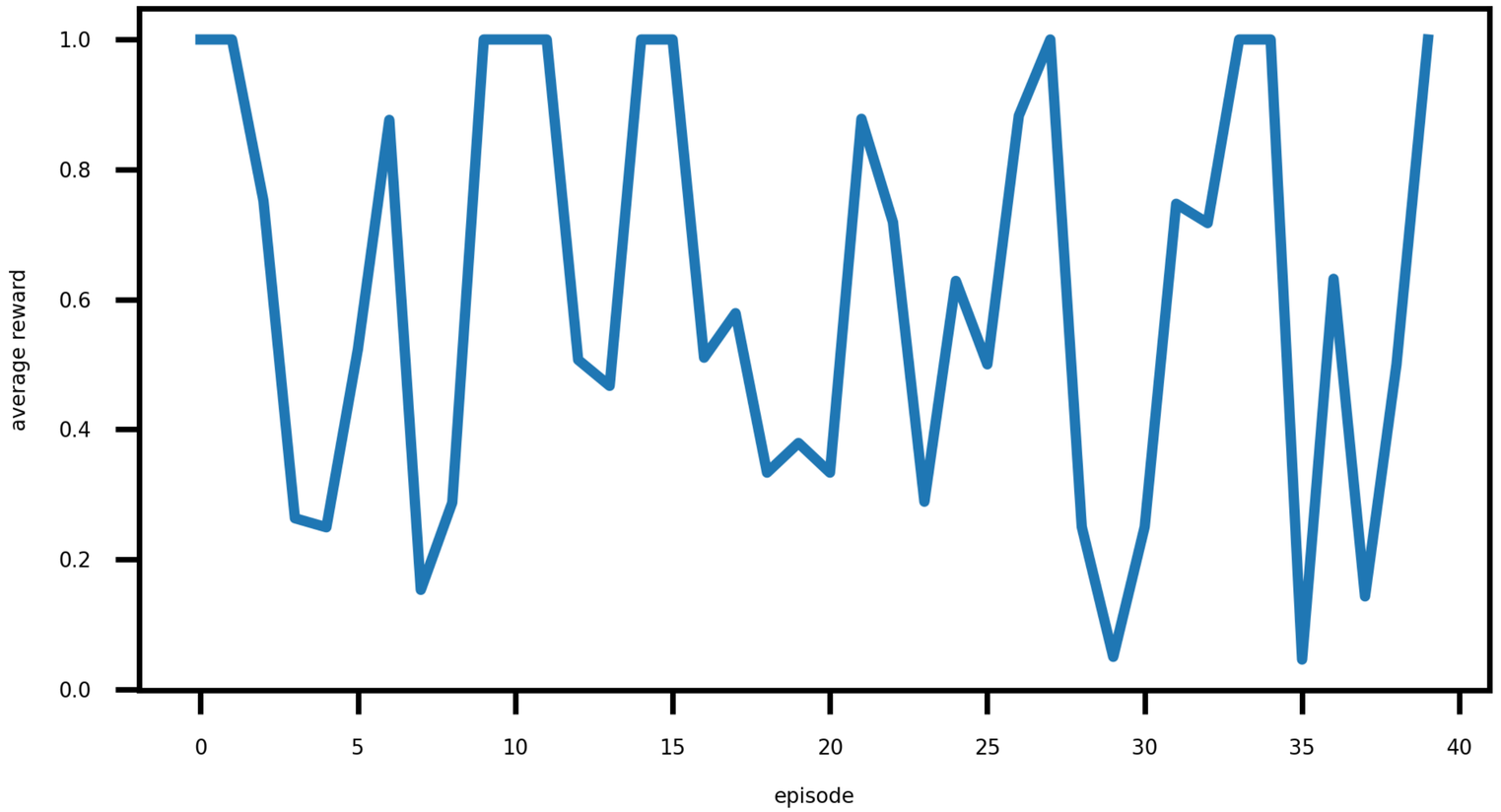
Took too long limited it to 300 “runs”, but the result still has the same problem of not responding to different inputs.

```
iteration = 0;
limited = 1;
while (limited != 1000000):
    state = np.copy(savestart)
    done = 0
    graph = plt.imshow(state, cmap='gray',norm=LogNorm(vmin=0.0000000000000001, vmax=.01, clip= True))
    while (1 != done and limited// 100 != 0):
        prevloc, nextstate, done, state = q_iteration(env,model,state,iteration,remember,prevloc)
        graph.remove()
        graph = plt.imshow(state, cmap='gray',norm=LogNorm(vmin=0.0000000000000001, vmax=.01, clip= True))
        iteration = iteration + 1
        limited = limited + 1
    limited = limited + 1
    env.reset()
    state = np.copy(savestart)
    plt.clf()
model.save("GYM\model.keras")
```

Evaluating Model Performance

Two metrics graphed in the DeepMind paper and mentioned in the community

- Average reward per episode
- Average Q value per episode



Decreasing the Epsilon Decay Rate and Altering the Training Regimen

Before:

- Epsilon decay rate was linearly decayed over 100 frames, from 1 to 0, reset at every new episode
- Training length depended on number of episodes completed and/or maximum number of frames allowed
- Training scenarios altered only three times throughout entire course of training

After:

- Epsilon decay rate is linearly decayed over 100000 frames, then kept fixed
- Training length now depends on number of frames played, each episode is still limited by number of frames (200)
- Training scenarios now alternate between episodes