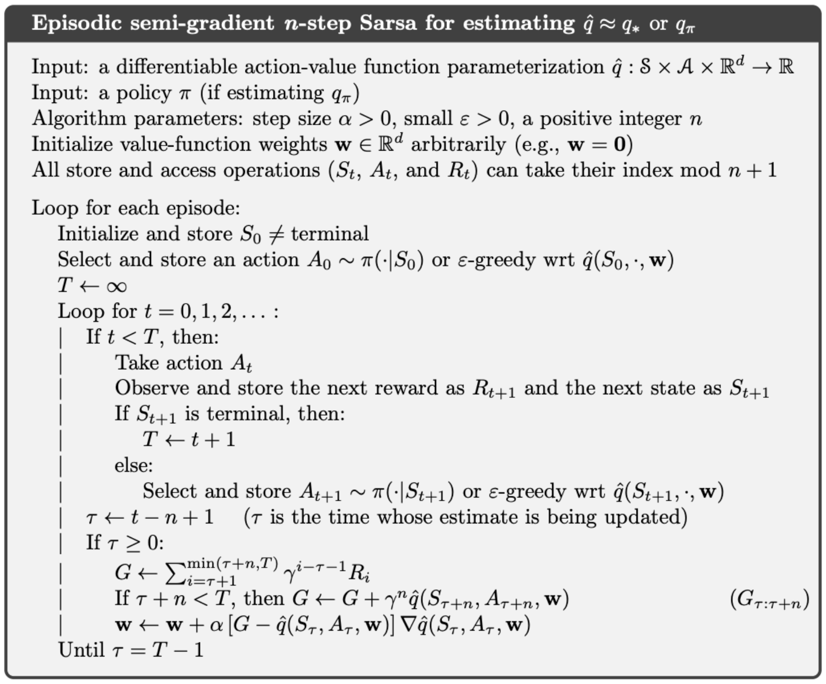
Luke Davidson
CS 5180
Ex7

1.) The pseudocode for a MC control would look something like:
    a.) Inputs: policy pi, q_hat, alpha, **w**_init
    b.) Loop:
        i.) Generate episode S0, A0, R1, S1, A1, R2 … S(T-1), A(T-1), R(T)
        ii.) For step t in range(T):
            (1) Calculate Gt
            (2) **w** = **w** + a*(Gt - q_hat(St, At))*grad{q_hat(St, At)}

    This pseudo code is likely not provided since it is a much simpler weight update function than say TD or SARSA. MC methods are more accurate predictors of target value Ut than TD or SARSA, so I would anticipate the control to learn an optimal policy faster and/or reach a more optimal policy than TD/SARSA methods.

2.) The semi-gradient one-step expected SARSA pseudocode would be almost identical to below, except the **w** update step (and all n=1).
    a.) The gamma^n * q_hat(S(t+n), A(t+n), **w**) term in the return (G) calculation of the weight update would change to gamma * (the sum over all actions of pi(a | S')*q_hat(S, a, **w**)), and all other instances of n would be 1.



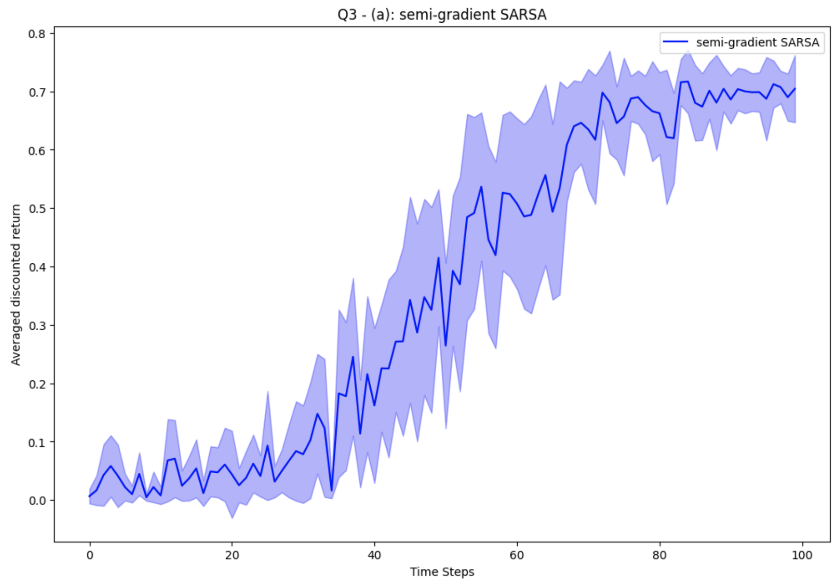
Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

```
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy $\pi$ (if estimating $q_\pi$)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer $n$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations ($S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or $\varepsilon$-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$ :
  |  If $t < T$, then:
  |      Take action $A_t$
  |      Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
  |      If $S_{t+1}$ is terminal, then:
  |          $T \leftarrow t + 1$
  |      else:
  |          Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or $\varepsilon$-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  |  $\tau \leftarrow t - n + 1$    ($\tau$ is the time whose estimate is being updated)
  |  If $\tau \geq 0$:
  |      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  |      If $\tau + n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$          $(G_{\tau:\tau+n})$
  |      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{q}(S_\tau, A_\tau, \mathbf{w})\right] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
  Until $\tau = T - 1$
```

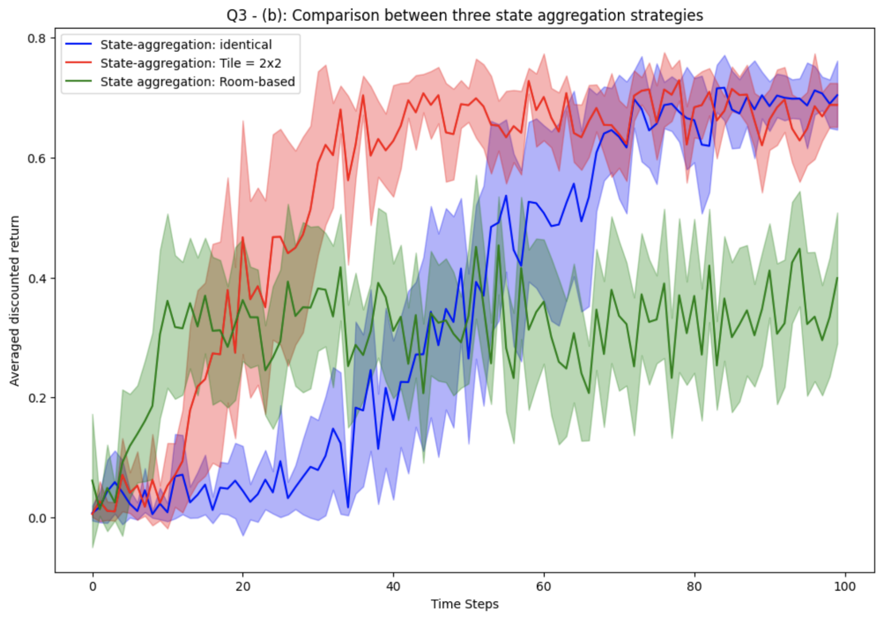
    b.) For semi-gradient Q-learning, it would again be similar up until the return update step (3 rows from bottom). After this IF statement (if t + n < T:), Q values would be predicted using Q(St+1, At+1, **w**) = w.T • x(s), and the target would be updated directly based on these Q values: G += gamma*Q. Following this would then be the same **w** update rule seen in the pseudocode above.

3.) Semi-gradient one step SARSA

a.) Plot for 10 trials of 100 episodes:



b.) Plot for single state aggregation (part a), 2x2 tiles, and room aggregation for 10 trials of 100 episodes each:



As seen in the above plot, aggregating states can help boost learning speed to a certain degree, but too much aggregation can lower overall results. Single state aggregation (essentially no aggregation) is shown in the blue curve, while the 2x2 tile aggregation is shown in red, and room aggregation in green. The 2x2 tile aggregation clearly learns a policy much faster than single state aggregation, and reaches the same level of success in terms of return. However, aggregating as much as possible will lead to worse results, shown in the green curve. Learning seems to increase quickly, but the method does not converge to a return nearly as high as the other two methods. This shows that aggregating to a certain degree can be very helpful in speeding up learning while still reaching an optimal policy, however too much aggregation can greatly affect the potential for a good policy to be found due to too much generalization.

c.) To create feature vectors that are unique for each state-action pair, I appended the one-hot action feature to the end of the unique state feature vector. The actions were appended as follows:

i.)	"Up" →	[1, 0, 0, 0]
ii.)	"Down" →	[0, 1, 0, 0]
iii.)	"Left" →	[0, 0, 1, 0]
iv.)	"Right" →	[0, 0, 0, 1]

So the final feature vectors were in the form →

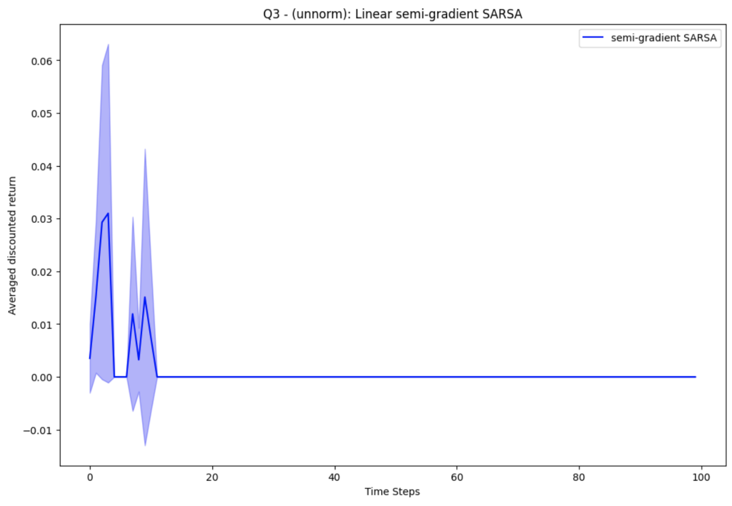
[(features), ***1*** * "up", ***1*** * "down", ***1*** * "left", ***1*** * "right"] where ***1*** = 1 if the action is present, else = 0.

Example implementation:

```
if action == "up":
    sa_feat_vectors.update({name: np.array([x, y, 1, 1, 0, 0, 0]).reshape(self.reshape_num,1)})
elif action == "down":
    sa_feat_vectors.update({name: np.array([x, y, 1, 0, 1, 0, 0]).reshape(self.reshape_num,1)})
elif action == "left":
    sa_feat_vectors.update({name: np.array([x, y, 1, 0, 0, 1, 0]).reshape(self.reshape_num,1)})
elif action == "right":
    sa_feat_vectors.update({name: np.array([x, y, 1, 0, 0, 0, 1]).reshape(self.reshape_num,1)})
```

The extra "1" as a feature is extremely important because it essentially allows for translation. Since Q = w^T•x(s), our Q calculation will come out to Q = w0*x + w1*y + w2. This w2 term is equivalent to the b term in y = mx + b. It can shift the results of the features without directly relying on them. Without the extra "1", there is a loss of a degree of freedom and results will not be as accurate.

Plot for the features of (x, y, 1):

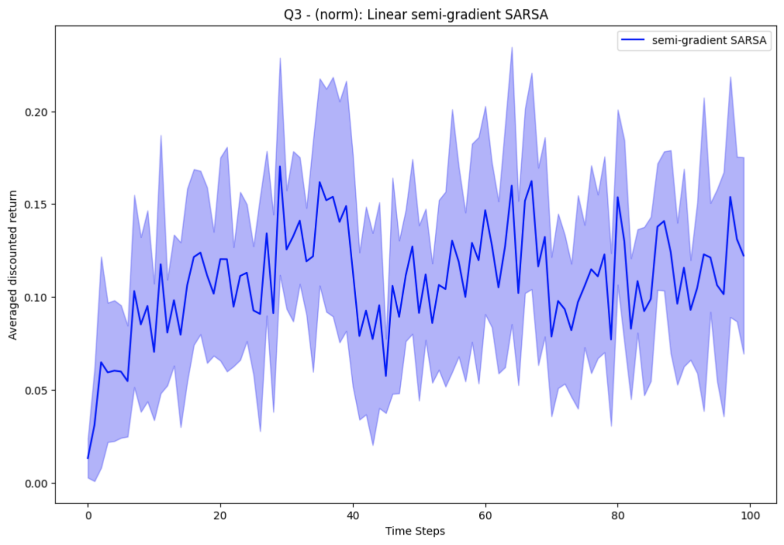


As one can see from the above plot, no learning occurred for this set of features. This is due to the magnitude of the features and the effect the magnitude has on the weight vector updates. A simple print out of the final weight vector of a random trial shows this effect:

```
Run #8/10
100%|████████████████████
Weights:
[[-4.52057961e-22]
 [-1.12911288e-21]
 [-1.02383932e-16]
 [ 1.02407423e-16]
 [ 1.02407413e-16]
 [ 1.02407379e-16]
 [ 1.02407377e-16]]
```

All trials resulted in weights of this magnitude, which are essentially 0. Because the features are not normalized between [0, 1], the weights are being driven to -inf. With a small amount of non-descriptive features and inaccurate weights, learning cannot be completed nearly as well as state aggregation above.

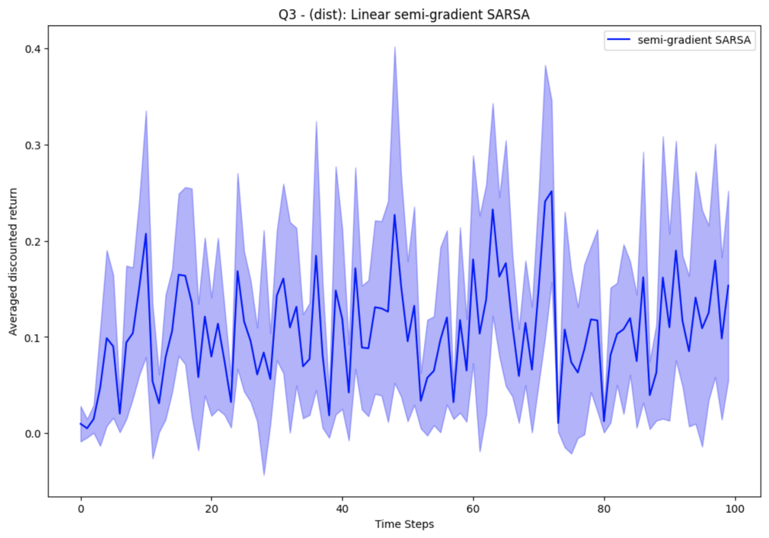
e.) A simple resolution to the above problem is making sure all features are less than or equal to 1. To do this, we simply divide by the max value x and y can be, so the new features are (x/10, y/10, 1). A plot for these normalized features is shown below:



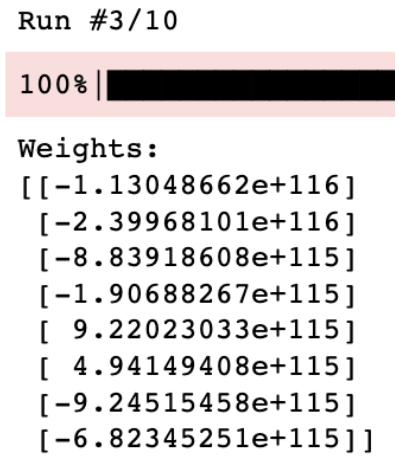
As one can see, learning has occurred and a policy was able to be created that converged to an average discounted return of roughly 0.11. The weights are also all reasonable in magnitude and make much more sense than the above unnormalized features:

```
Run #9/10
100%|████████████████
Weights:
[[ 0.7368187 ]
 [ 0.71282498]
 [ 0.13603084]
 [ 0.14848896]
 [-0.03172149]
 [-0.02267767]
 [ 0.04194104]]
```

Adding more features to the feature vector, such as distance away from the goal state, often allows for learning to happen quicker and reach a higher level of return. When creating a feature vector of (d, x, y, 1), we get the following results:

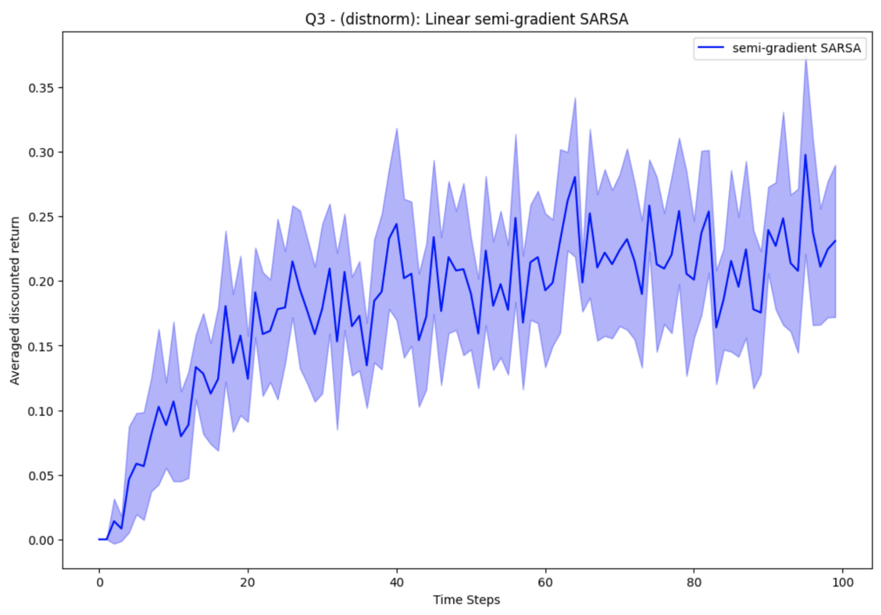


With weights around:

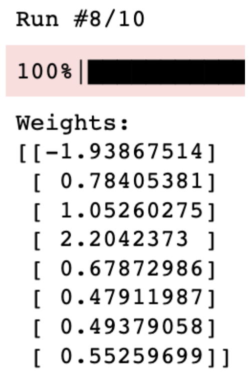


```
Run #3/10

100%|██████████

Weights:
[[-1.13048662e+116]
 [-2.39968101e+116]
 [-8.83918608e+115]
 [-1.90688267e+115]
 [ 9.22023033e+115]
 [ 4.94149408e+115]
 [-9.24515458e+115]
 [-6.82345251e+115]]
```

The weights are of the order of 115. This is because of a similar reason to part d, the features are large in value and amplify the weights. Some learning was able to happen due to the added feature, although there is a lot of uncertainty and oscillation around the converged return due to these weights. To show the effect of normalizing features, I ran another test where I normalized the distance, x and y coordinates, so the features were now (d/20, x/20, y/20, 1). The results for the normalized distance are shown below:



With weights around:



```
Run #8/10

100%|██████████

Weights:
[[-1.93867514]
 [ 0.78405381]
 [ 1.05260275]
 [ 2.2042373 ]
 [ 0.67872986]
 [ 0.47911987]
 [ 0.49379058]
 [ 0.55259699]]
```

The effect of normalizing the features is clear as learning was able to occur quickly with far less variability and a much higher return convergence of around 0.22 compared to 0.11. Again, the weights are also reasonable in magnitude.