EmotiLines:
Learning Motion Paths with a "Style" that Mimics the Demonstrator

Joel Iventosch

***Abstract*— In this research project I apply a previously developed Apprenticeship Learning via Inverse Reinforcement Learning (IRL) framework [2] to a new and interesting domain - learning motion paths with *style*. In this work I attempt to implement this framework, with the goal of training an autonomous agent to learn from demonstrations to navigate from a generalized start to goal in a 2D grid world while preserving the *style of motion* of the demonstrator. The results were largely unsuccessful, but highlighted many interesting aspects of this challenging domain and also suggest very promising avenues for future work.**

I. Introduction

A. Motivation

The ultimate goal of my personal research endeavors is to learn how to model the human emotional state. My hope is that this eventually enables me to create a learning agent that observes a particular human over an extended period of time and learns to simulate that person's personality and emotions - eventually via a full text, video, and graphical interactive interface. Although this goal is a long ways off (and perhaps AI complete), I view this class project as a first stepping stone along my path to modelling human personality and emotion.

The relation between this ultimate endeavor and my class project is the idea that we attribute personalities and emotion to agents as simple as geometric shapes moving around in a 2D grid world, just based on their style of movement [Scott Niekum]. With this in mind, the goal of this project is to capture the emotion associated with different trajectories, based on their style of movement as the motion path goes from a start location to a goal in the 2D world.

B. Problem Definition

From a technical standpoint, the formal problem being addressed is: **to learn the unknown reward function of the human demonstrator for each set of motion path demonstrations provided for each of several fundamental emotions chosen by the demonstrator (e.g. happy, sad, angry, surprised, afraid, disgusted).** In other words, the user is asked to give several motion demonstrations (for each emotion) that show the learning agent how to get from a start to a goal location in a certain style that corresponds to each specific emotion for that user. It is assumed that the demonstrator is acting based on an optimal policy that subconsciously attempts to maximize the sum of future discounted rewards based on some inherent, but unspecified, reward function. It is also assumed that the same unknown reward function is used for every demonstration for a particular emotion, but that the human is utilizing different reward functions for the demonstrations corresponding to different emotions. Thus the formal hypothesis is: **given several demonstration trajectories for each emotion from a specific human user, can the autonomous agent learn a reward function that induces an optimal policy for each emotion that performs as close as possible to the demonstrator on the task of getting from start to goal using the style of the particular emotion?**

C. Experimental Setup

The hypothesis can be broken down into two components:

1) a *quantitative metric*: did the autonomous agent learn to navigate from a start location to a goal for generalized start and goal positions?
2) a *qualitative metric*: was the agent able to learn the "style of motion" for each emotion, based on the specific user's demonstrations?

Evaluation of the first metric is very straight-forward. The qualitative metric, on the other hand, is more difficult to assess due to its subjective nature. To obtain quantitative metrics for this qualitative goal, I used the following experimental setup: a user was asked to provide a small number of motion demonstrations (five to ten) for each of the three emotions. I then trained the autonomous agent using my implementation of the algorithm. Upon completion of training, the agent was tasked with navigating from a randomly selected start to goal using an approximately optimal policy induced from one of the three learned reward functions, selected at random. The autonomous agent's motion path was then displayed on the GUI while the user watched, and the user was asked to select which of his three emotional styles of movement the agents trajectory most closely resembled. This was repeated for 50 different trajectories created by the autonomous agent, where in each case the agent randomly selected which of the learned reward functions to utilize. Based on the ratio of correctly identified emotional styles, a quantitative metric was computed. This was repeated for three different demonstrators (one of which was myself).

II. Related Work

Although there exists a fair amount of previous work on the topic of Inverse Reinforcement Learning (IRL), I chose to narrow the scope of my research project to only a handful of select related works in the IRL subfield. I made this decision based on the notion that my primary goal was to get a deep dive into one particular framework

and gain hands on experience applying this framework to a relatively new and interesting domain, rather than attempting to come up with a new framework of my own. As such, the primary related work that I relied very heavily on was Andrew Ng and Peter Abbeel's 2004 work on Apprenticeship Learning via IRL [2]. Other previous works that I reviewed and drew ideas from for my project were Ziebart et. al's 2008 work on Maxium Entropy (MaxEnt) IRL [4] and Wulfmeier and Andruskas 2015 work on Deep IRL [8]. Both of these works involve IRL and suggested frameworks, although neither apply specifically to my project's domain. I referenced several other sources extensively throughout my project for ideas (see bibliography section), although those were not related works per se, as much as they were algorithmic references [5] [6] [1] [7] [3].

III. Technical Approach

A. Framework

The technical approach that I took is very similar to that used in Pieter Abbeel and Andrew Ngs 2004 paper Apprenticeship Learning via IRL [2]. I used their same basic apprenticeship learning framework, in which each state gets mapped into a features vectors. It is then assumed that the reward for any such state is a linear combination of the components of this feature vector. The task of the IRL portion of the algorithm is to iteratively compute a better estimation of the weight vector used in this linear combination, by updating it to have weights that maximize the margin between the expected summation of future discounted feature vectors of the expert policy (i.e. the demonstrator) and that of the next best candidate policy thus far (at each iteration of the algorithm). Again following the outline of Abbeel and Ng, I estimated these candidate policies in an inner RL step of the algorithm by using the best reward function (defined by its learned weight vector) found thus far in the outer IRL portion of the algorithm. Initally I used a full-fledged inner RL step to compute the approximately optimal policy, first with *Fitted Value Iteration* and then later with *SARSA Gradient Descent*. However, due to poor performance issues for both, I eventually switched to using a simple, strictly greedy policy that selects an action at each time step by choosing the action that maximizes the possible reward attainable from the current state (more details regarding the inner RL step in my implementation in section VI).

Several differences between my approach and the work presented by Abbeel and Ng are:

1) They computed their features vector as a mapping from states to features, whereas I map state-action pairs to features in my implementation. Based on my action space (see next sub-section below), this is a critical aspect of my project, since the style associated with a motion trajectory is largely dependent on the velocity and acceleration at each timestep - e.g. angry motion paths might be very fast, while sad motion paths might move very slowly.
2) They had a discrete and fairly small state-space, so they used an exact RL algorithm (SARSA) in their inner RL step, whereas I was unable to use an exact RL algorithm, due to the very large state-action-space of my problem domain.

B. State-Action Space

The state-space for my project was a four-dimensional vector consisting of the agent's current x, y coordinates within the 2D grid world, as well as the agent's current $velocity$ in both the x and y directions at a given timestep. For notational clarity, I will refer to a given state, $s \in S$ as the four-dimensional vector: $s = [x, y, \dot{x}, \dot{y}]$. Although the state-space was designed to be continuous on the range of $(0, 500)$ for both the x and y position coordinates (and hypothetically on the range of all reals for x and y velocities), it ended up being discretized into integer-only values for x, y, $\dot{x}$, and $\dot{y}$ within their respective ranges. My code infrastructure was designed to handle a continuous state-space, but due to the way I implemented my Graphical User Interface (GUI) for collecting demonstration trajectories (using a python wrapper library for Tkinter), the state-space was naturally discritized into integer-values. The 2D grid world within my GUI for collecting demonstrations was a $500x500$ pixel canvas, thus the agent's x and y coordinates were always collected as integer values, since they corresponded to the x, y location of a particular pixel within the GUI. As a result $\dot{x}$, and $\dot{y}$ were always computed as integer values as well.

Formally, the state-space was defined as:
$S = x \times y \times \dot{x} \times \dot{y}$,
where $x, y \in \mathbb{Z} \mid 0 \leq x, y \leq 500$ and $\dot{x}, \dot{y} \in \mathbb{Z}$.

The action-space consisted of increasing or decreasing $\dot{x}$ and/or $\dot{y}$ at each time step (i.e. setting $\ddot{x}$ and/or $\ddot{y}$). Formally, the action-space was defined as:
$A = \ddot{x} \times \ddot{y}$, where $\ddot{x}, \ddot{y} \in \mathbb{Z} \mid -2 \leq \ddot{x}, \ddot{y} \leq 2$.
In other words, the action-space consisted of setting the acceleration vector, $[\ddot{x}, \ddot{y}]$, to one of the elements in the set $\{-2, -1, 0, 1, 2\} \times \{-2, -1, 0, 1, 2\}$, hence there were 25 possible actions in total.

Due to the nature of the domain it was important to reward *specific actions* taken from *specific states* to produce the desired style of motion. As a result, reward had to be modelled as a function of both state and action. Having the action-space as a part of the reward structure necesitates the need to argmax over all possible actions when chosing the optimal action to take at a given timestep from a policy implicitly defined by the learned value function (if using value function approximation or a strictly greedy policy, both of which were case at different stages throughout my project). As such, the action-space was kept discrete and fairly limited in order to maintain computational tractability, particularly in light of the very large state-space.

Due to the very high sample rate for my motion paths (ranging between 10 Hz and 100 Hz throughout the development process, settling on 50 Hz for use in experiments), $\dot{x}$ and $\dot{y}$ were estimated by simply taking the difference in the agent's position from one time step to the next. In a similiar fashion, $\dot{x}$ and $\dot{y}$ were updated based on selected actions by setting $\dot{x}_{i+1} = \dot{x}_i + \ddot{x}_i$ and $\dot{y}_{i+1} = \dot{y}_i + \ddot{y}_i$. Although carrying

the agent's velocity as part of the state-action-space is not ideal, it was necessary in order to compute the desired feature mapping for every state-action pair, as described in the next subsection.

C. Feature Mapping

Feature engineering was a key process throughout this research project, and consumed a significant amount of development time. Although my initial feature set was poorly designed, after researching different feature mapping strategies and tweaking my feature set, I ulitimatley arrived at what I consider to be a fairly quality feature mapping. I think my feature engineering was one of the more successful aspects of this research and that the subpar results (see section IV) can largely be explained by other factors (see section VI). Although this hypothesis cannot be confirmed without additional experimentation, ultimatley I view my feature engineering efforts (and the amount learned along the way) as a key takeaway from this project.

The feature mapping that I eventually settled on, consisted of 20 binary featueres, 17 of which were unweighted (i.e. value of 0 was assigned if feature was not present, and value of 1 assigned if feature was present), and three of which were weighted (i.e. value of 0 assigned if feature not present, value of $c \in \mathbb{Z}$ assigned if feature present). Thus, formally, my feature mapping was a function that mapped $S \times A \longrightarrow Feats$, where $Feats$ was 20-dimensional vector of the following features:

- **Norm of the velocity vector** discretized into five bins. Bin ranges were automatically detected by the algorithm, defined uniquely for each set of demonstrations corresponding to a specific motion style. (Note, these bin ranges were also provided to agent after the learning algorithm had run, to enable the agent to appropriately set the bin ranges again when actually performing feature mapping during the execution of new trajectories on generalized starts and goals.)
- **Angle of motion relative to previous timestep**, discretized into 6 bins:
  $\{[0, 30), [30, 60), [60, 90), [90, 120), [120, 150), [150, 180)\}$
  measured in degrees. This feature was computed as:
  $$\arccos \frac{\langle [\dot{x}_i, \dot{y}_i] \cdot [\dot{x}_{i+1}, \dot{y}_{i+1}] \rangle}{\|[\dot{x}_i, \dot{y}_i]\|_2 * \|[\dot{x}_{i+1}, \dot{y}_{i+1}]\|_2}$$
- **Angle of motion relative to goal**, discretized into 6 bins (using same discretization as above). This feature was computed as:
  $$\arccos \frac{\langle v_{goal} \cdot v_{goal} \rangle}{\|v_{goal}\|_2 * \|v_{goal}\|_2}$$
  where:
  $v_{goal} = [(x_{goal} - x_{i+1}), (y_{goal} - y_{i+1})]$ and
  $v_{action} = [(x_{i+2} - x_{i+1}), (y_{i+2} - y_{i+1})]$
- **Closer to Goal?** Specifically, does the selected action move me closer to the goal, measured as the difference between the Euclidean distance from the goal in my next state and the Euclidean distance from the goal two states into the future. (Note that the comnparison is between the x, y positions for $state_{i+1}$ and $state_{i+2}$ since the x, y poisiton for $state_{i+1}$ is not affected at all by the current action, a_i, since the action-space is two derivatives away from position-space.)
- **In Goal?** Specifically, does the selected action move my state two time-steps into the future into the goal region? (goal region was defined as a $14x14$ rectangular box surrounding the goal coordinates).
- **Out of Bounds?** Specifically, does the selected action move my state two time-steps into the future out of bounds? i.e. off the $500x500$ pixel grid. (Note that a terminal state was considered either the agent being in the goal region or more than 50 pixels off the grid in any direction.)

Although weighting certain binary features is not a typical practice, given the way my algorithm computed the aggregated feature vector over the course of a trajectory (referred to as *feature expectations* or *FE* for notational convenience), giving more weight to certain features that I wanted to emphasize had a noticeable impact on the algorithm's tendancy to learn appropriate reward weights for those features. For instance, intuition tells us that the algorithm should learn to assign a negative weight of substantial magnitude for being out of bounds. Yet, since the agent was only ever allowed to travel up to 50 pixels off the grid in any given direction (based on the way a terminal state was defined), FEs for early candidate trajectories that hadn't learned appropriate reward weights yet, would not accumulate significant sums in their ***out of bounds*** feature if it was unweighted. And since demonstration trajectories always have a value of zero in their FE for this feature, the algorithm would have difficulty determining that being out of bounds is a very bad thing, since the FE for the candidate trajectory would only differ by a small amount from the demonstrator for the ***out of bounds*** feature. Similarly, assigning a heavier weight for moving closer to the goal and a very heavy weight for being in the goal region seemed to help significantly as well.

The above commentary is supported based on the data observed throughout the development and experimentation process. However, it should be considered with some caution in light of the fact that my algorithm normalized FE vectors throughout, which as it turns out, was likely a key factor to the relatively poor performance of the algorithm overall (see section VI for further discussion on this topic).

D. Algorithm

My implementation of the algorithm is based on the framework presented by [2]. Accordingly, it is very similar and relies heavily on the pseudocode that they provide in their paper. All of the code, however, for my implementation was written on my own (in python 3.4), with the exception of library calls to:

- ***SciPy*** (I use their ***NumPy*** module extensively throughout my code)

- ***Sci-Kit Learn*** (I use their SVM library for my subroutine called in line 25 of my pseudocode below)
- ***Matplotlib***
- ***Tkinter*** for my GUI
- Various standard python modules and libraries, including: statistics, math, os, time, etc.

All of my acutal code is available at: https://github.com/joeliven/lfd. Note, that the function calls in the psuedocode below do not necessarily reflect the names of the functions in my actual code. Some liberty in renaming has been taken to improve the readability of the pseudocode.

IV. Experimental Results

Experiments were performed (as described in section I) for three different demonstrators, each of whom selected three different motion styles. Demonstrator 1 (D1) was asked to select his motion styles with (relatively) no restrictions on the level of complexity. Results were then computed for D1, and after observing the algorithm's relative inability to learn the fairly complex motion styles demonstrated by D1, D2 was asked to provide a simpler set of motion styles. Results for D2 were then computed. Although these results were slightly improved, the algorithm was still unable to pick up some of the more nuanced aspects of D2's motion styles. Accordingly, D3 (myself) chose to employ very simple set of motion styles, in order to assess the algorithm's ability to learn from these much simpler demonstrations.

Although influencing the demonstrator's choice of motion paths largely nullifies the statistical validity of the overall results, it allowed me to assess the algorithm's relative ability to learn motion styles of varying levels of complexity. This is very valuable information in that it provides a "complexity threshhold" of sorts that gives a soft bound on the level of complexity that the algorithm can handle, while still being able to successfully learn how to get to the goal and mimic the style. One final disclaimer: using myself as one of the demonstrators (D3) introduced significant added bias since my intimate knowledge of how the algorithm works (its reward structure, feature mapping, how it segments distance partitions, etc.) undoubtedly enabled me to correctly identify a larger percentage of motion styles than I otherwise would have. Also, D2 had to leave part way through replaying/identifying the learned motion paths on generalized start and goal locations - as such, I identified the second half of his set of 50. Although, I hadn't seen the truth values either, it likely introduced additional bias for the same reasons as mentioned above.

Table I shows the numerical results from the three experiments and table II provides additional details about the motion styles selected by each demonstrator. Figures 1, 2, 3, 4, 5, 6, 7, 8, 9, show the demonstrations paths, highlights, lowlights, and interesting paths collected during the experiments for each of the three demonstrators. Additional commentary on these results is provided in the next section.

LearnRewardWeights($rawDemos, dparts$)**:**

1: $k \leftarrow numberOfFeatures$ {e.g. 20}
2: $numBins \leftarrow numberOfBins$ {e.g. 5}
3: $dparts \leftarrow numberOfDistancePartitions$ {e.g 4}
4: $policies \leftarrow emptyList()$
5: $mixedPolicies \leftarrow emptyList()$
6: $FEs \leftarrow emptyList()$
7: $rweights \leftarrow emptyList()$
8: $margins \leftarrow Array(dparts)$ {initilize each entry to ∞}
9: $demos = processDemos(rawDemos)$
10: $bins \leftarrow Array(dparts, numbins)$ {initialize array}
11: $bins = detectVnormBins(rawDemos)$
12: {FE_{demos} is a **dparts**x**k** array with a different FE for each distance partition}
13: $FE_{demos} = computeFE(demos, bins)$
14: $EFs.append(FE_{demos})$
15: {each policy has $dparts$ different subpolicies}
16: $policy_0 = generateRandomPolicy(demos, bins)$
17: $policies.append(policy_0)$
18: $EF_0 = computeFE(policy_0, bins)$
19: $EFs.append(EF_0)$
20: $i \leftarrow 0$
21: **while** $\|margins\|_2 \leq \epsilon$ and $i \leq maxIters$ **do**
22: $reweight_i \leftarrow Array(dparts, k)$ {initialize array}
23: **for** dp in $dparts$ **do**
24: {use SVM to learn reward weights for each dp}
25: $margins[dp], rweights_i[dp] = optimizeRewardWeights(EFs, EF_{demos}, bins)$
26: **end for**
27: $rweights.append(rweights_i)$
28: {RL step $\rightarrow$ SARSA gradient descent or nothing for this step if using strictly greedy policy}
29: $policy_i = computeOptimalPolicy(rweights_i, bins)$
30: $policies.append(policy_i)$
31: $EF_i = computeFE(policy_i, bins)$
32: $EFs.append(EF_i)$
33: {determine the best mixed policy by finding the best policy computed so far for each dpart}
34: $mixedPolicy_i = findBestP(policies, EFs, EF_{demos})$
35: $mixedPolicies.append(mixedPolicy_i)$
36: $i++$
37: **end while**
38: return $rweights, mixedPolicies$

TABLE I
Results from 3 Experiments

	Hit Goal	%	Emo IDd	%	Both	%
D1	13/50	26%	22/50	44%	11/50	22%
D2	16/50	32%	25/50	50%	7/50	14%
D3	25/50	50%	29/50	58%	14/50	28%
Tot	**54/150**	**36%**	**76/150**	**50.67%**	**32/150**	**21.33%**

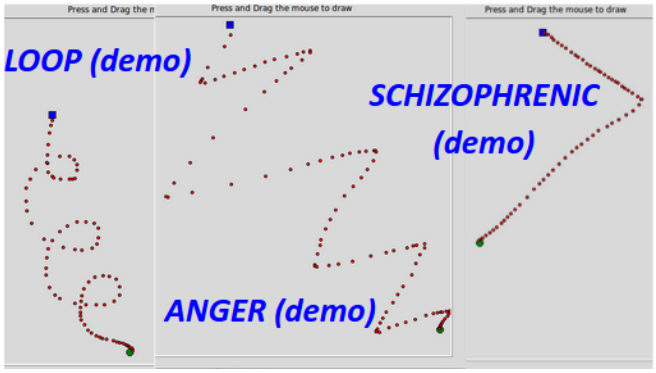

Fig. 1. The three different *styles of motion* demonstrated by D1

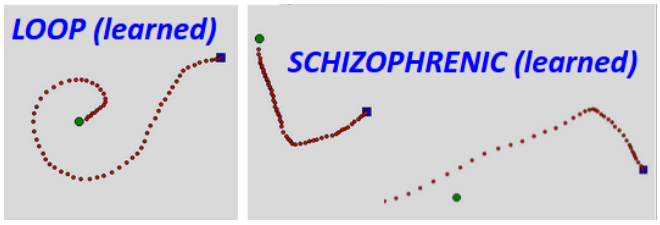

Fig. 2. Highlights from Experiment 1 (D1)

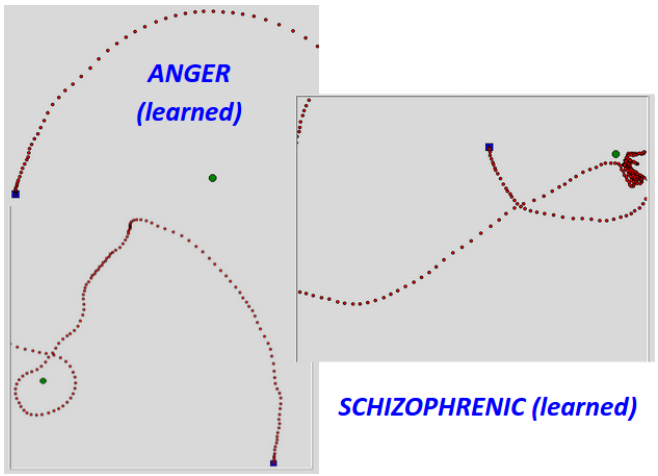

Fig. 3. Lowlights and Interesting paths from Experiment 1 (D1)

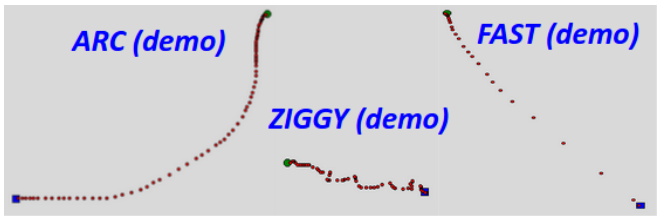

Fig. 4. The three different *styles of motion* demonstrated by D2

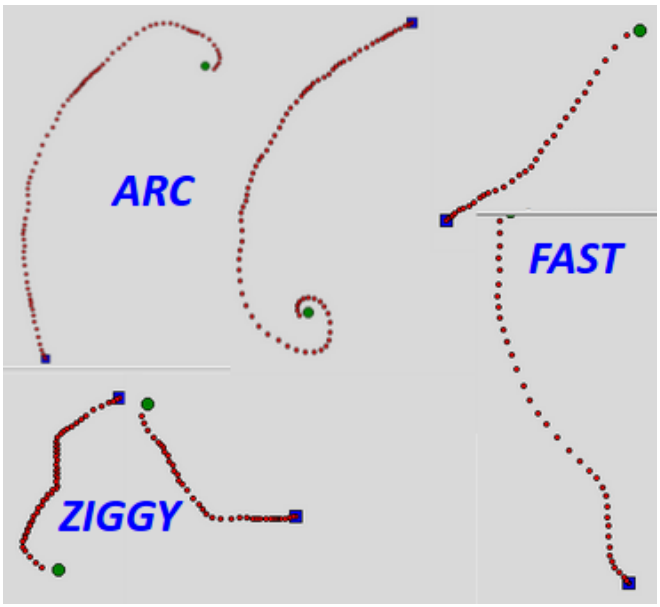

Fig. 5. Highlights from Experiment 2 (D2)

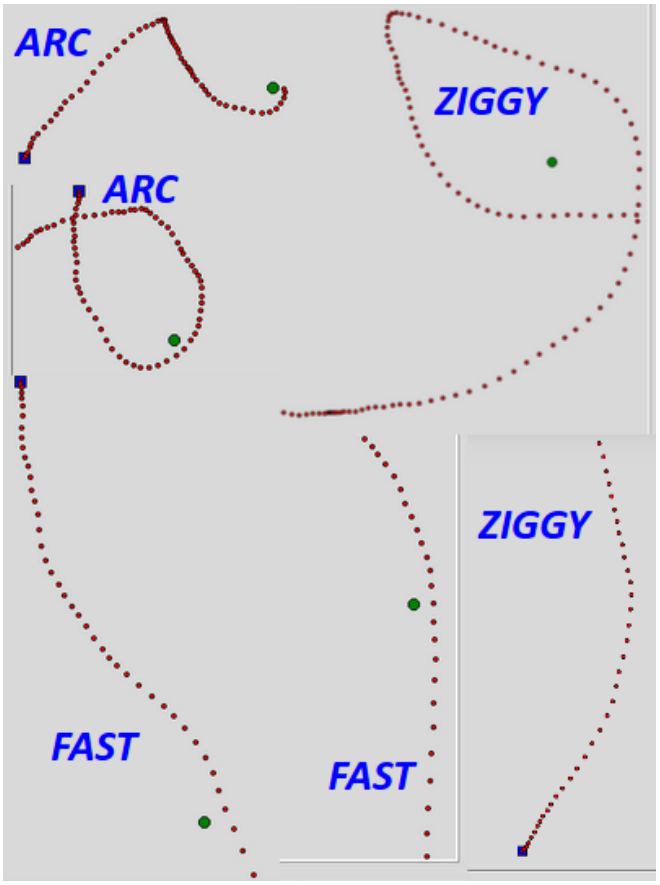

Fig. 6. Lowlights and Interesting paths from Experiment 2 (D2)

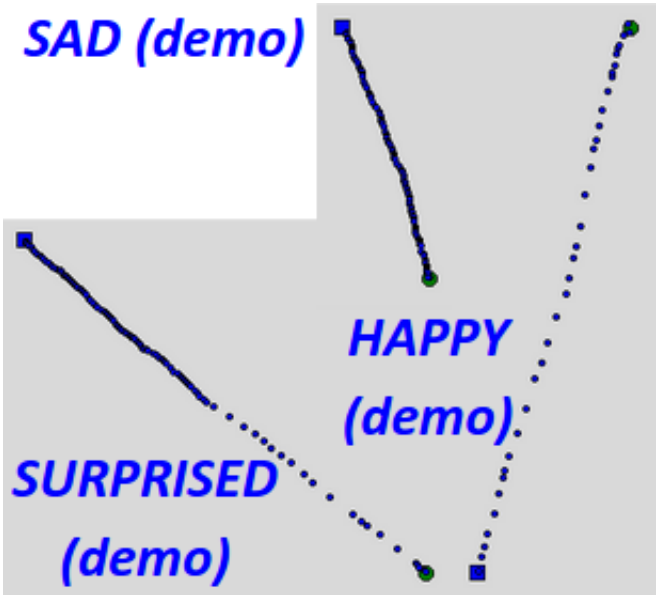

Fig. 7. The three different *styles of motion* demonstrated by D3

Fig. 8. Highlights from Experiment 3 (D3)

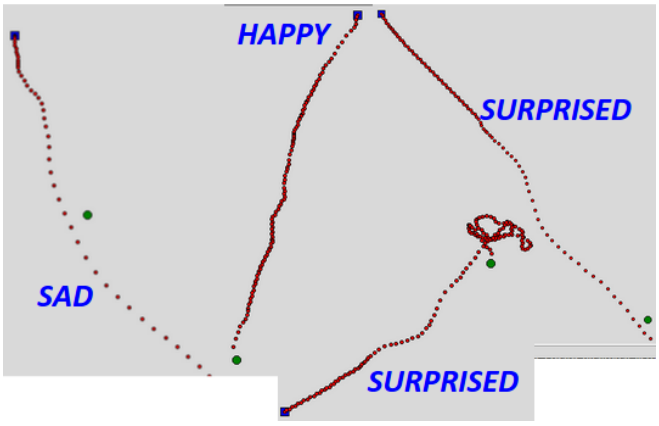

Fig. 9. Lowlights and Interesting paths from Experiment 3 (D3)

TABLE II
DESCRIPTIONS OF MOTION STYLES SELECTED BY DEMONSTRATORS

D#	# Emos	Emos	Description	Complexity
D1	3 + ctrl	*Loop*	loopy, circuitous	difficult+
		Anger	long zig-zags, sharp angles	difficult+
		Schizo	heads nearly perpendicularly away from goal for a while, then straight to goal	difficult
		Ctrl	slow, direct path	simple
D2	3	*Arc*	consistent arching path	moderate
		Ziggy	small, jagged, path towards goal	difficult
		Fast	fast, direct path	simple
D3 (me)	3	*Sad*	slow, direct path	simple
		Happy	fast, direct path	simple
		Surprised	slow direct path towards goal for first part of trajectory, then fast, direct path to goal	moderate

V. DISCUSSION OF RESULTS

In this section I will highlight the key findings in in my experimental results, and discuss what these results mean in terms of evaluating my original hypothesis. I will then turn to a discussion of what these results imply in the context of evaluating the performance of my implementation of this framework.

A. Evaluation of Hypothesis

The experimental results suggests that neither the quantitative nor qualitative parts of my hypothesis were true and that my algorithm failed to train the autonomous agent to learn to navigate from a genearlized start to goal, while still maintaining the *style of motion* learned from the demonstrator. Analysis of table I reveals that over the course of all three experiments, the agent was able to successfully navigate to the goal only 36% of the time. Performance for complex and moderate difficulty motion styles was particularly poor at 26% and 32% respectively. Although performance for simple motion styles was significantly better at 50%, that is still far worse than anticipated. On the whole, the data collected from these three experiments indicates that the agent was not able to consistently navigate to the goal, suggesting that the quantitative aspect of the hypothses was a failure.

Interestingly, performance with regards to the qualitative part of the hypothesis, namely can the agent learn to mimic the *style of motion* of the demonstrator, was significantly better, averaging just over 50% accross all experiments, and achieving a maximum performance of 58% for experiment 3. Also of interest is that the performance spread between the difficult, moderate, and simple motion styles was much lower with regard to the qualitative aspect of the hypothesis than the quantitative. This could be explained by one of several things: either the algorithm was biased in some way to encourage the agent to place a higher priority on learning

the style of motion versus getting to the goal; or the task of identifying a style of motion is much simpler by nature given that the demonstrator had a 33% chance of guessing the correct style at random; or the fact that I was the one identifying the style of motion over half the time introduced a "designer's bias" as discussed above. I think a combination of all three factors is most likely. Whatever the case, the data - both the numerical results in table I and certain anecdotal evidence as can be seen in the figures - suggests that the algorithm performed significantly better with regard to the second part of the hypothesis, but still not near well enough to suggest success.

B. Evaluation of Performance

In spite of the above discussion, the results of these experiments don't necessarily suggest that the hypothesis is **not** true - in the sense that it isn't attainable. I think the data much more so suggests that my implementation was flawed and could use improvements. In particular, some of the anecdotal evidence from the experiments shows very promising potential for an improved algorithim. Section VI discusses some of these key implementation flaws and potential fixes and future work in more detail. An improved algorithm that addresses some of these flaws certainly might be capable of solving the challenges posed by my hypothesis. As such, I conclude that the results of these experiments suggest that both parts of the hypothesis were a failure, but they don't conclusively indicate that the hypothesis is unattainable.

VI. Lessons Learned and Future Work

In this section I will discuss in a topic-by-topic fashion what worked well in my research project, as well as what "interesting failures" led to important lessons learned and suggested directions for future work.

A. Multi-Controller Model using Distance Partitions

I think one of the really successful aspects of this project that appeared to work fairly well (at least based on the anecdotal evidence observed throughout the experiments) was my implementation of a multi-controller model. In its simplest form, this was achieved by segmenting the demonstration trajectories into several partitions based on the agents Euclidean distance from the goal at each time step, and then running the same underlying IRL apprenticeship learning algorithm separately to learn a different set of optimal reward weights (and thus a different optimal controller) for each of the several distance partitions. This attempts to incorporate the notion of a time-varying reward function into the framework. While the apprenticeship learning framework is very effective in many cases, it relies on the underlying assumption that the unknown reward function of the demonstrator is the same at every timestep, since FEs are computed across all timesteps in a demonstration, and the same feature mapping is used without regard to time.

Although my multi-controller model doesn't directly encode the temporal aspects of a motion path (since not all trajectories travel in a montonically increasing or decreasing fashion), it at least incorporates some sense the demonstrator might not always be acting with regard to the same unknown reward function. Although I lack statistical data to support this, looking at certain paths that the agent learned (for instance the D3 highlghts in figure 8 for the *surprised* motion style) suggests that algorithm was clearly able to learn (and then switch between) different controllers for the different distance partitions. This resulted in motion paths that learned the style of the demonstrator fairly well in some cases. My implementation arbitrarily chose the number of distance partitions and hard-coded this into the algorithm, but future work could possibly use a more sophisticated method, such as changepoint detection, to automatically determine the appropriate number of distance partitions.

B. Normalizing Feature Expectations

In hindsight it is apparent that one key flaw in my implementation of the Apprenticeship Learning framework was normalizing the feature expectations vector at all points throughout the algorithm (both for the demonstration trajectories and for the trajectories induced by the learned reward weights). Although I did not realize it until after giving my presentation, this flaw likely had a very negative impact on my algorithm because it can cause perceptual aliasing [Scott Niekum] - in other words, by normalizing the feature expecations vector it causes two proportional FE vectors to appear the exact same. While, we might expect two trajectories with proportional FEs to share some similarities with regard to certain features, a comparison of unormalized FE vectors vs their normalized counterparts, reveals that very different trajectories could actually have the same FE vector once normalized. For instance, consider a path that takes 100 timesteps to get from start to goal, accumulating a certain (unormalized FE) vector, call it $f_1 = [c_1, c_2, ..., c_k]$ for the various k features. Now consider a proportional (still unormalized) FE vector, f_2, that accumulates feature expectations of $[ac_1, ac_2, ..., ac_k]$ in only 20 timesteps. Once normalized we have:
$f_{1normalized} = \frac{f_1}{\sqrt{c_1^2+c_2^2+...+c_k^2}} = \frac{[c_1,c_2,...,c_k]}{\sqrt{c_1^2+c_2^2+...+c_k^2}}$
and
$f_{2normalized} = \frac{[ac_1,ac_2,...,ac_k]}{\sqrt{(ac_1)^2+(ac_2)^2+...+(ac_k)^2}} = \frac{a[c_1,c_2,...,c_k]}{\sqrt{a^2(c_1^2+c_2^2+...+c_k^2)}} = \frac{a[c_1,c_2,...,c_k]}{a\sqrt{c_1^2+c_2^2+...+c_k^2}} = \frac{[c_1,c_2,...,c_k]}{\sqrt{c_1^2+c_2^2+...+c_k^2}} = f_{1normalized}$.
Yet, assuming that the trajectories that led to f_1 and f_2 had the same start and goal locations, we know that at a minimum, the two trajectories have very different average velocities since the $Vavg_2 = 5Vavg_1$ just based on the number of timesteps that each trajectory took. Unfortunately I did not realize this until after all experiments had been performed and results tallied. However, it is likely that this was a key factor contributing to the suboptimal results achieved by the algorithm.

C. SVM

The above flaw impacts the algorithm at a deeper level than just perceptual aliasing. By normalizing the FE vectors it significantly changes the maximum margin surrounding the hyperplane that separates the demonstrator's FE vector and the FE vectors of all other trajectories from learned policies. As a result, the margin returned by the SVM in the reward-weights optimization step of the algorithm was likely not an accurate representation of the actual margin that was desired.

This helps explain a peculiar phenomenon that I noticed in my implementation of the algorithm, but was unsure how to debug (or if it was indeed a bug). Namely, the max margin returned by the SVM classifier was always a very low value initially after the first iteration of the algorithm (˜0.05) and then very quickly jumped up to the $0.9 - 1.1$ range in almost all cases, where it typically remained for the duration of learning process. (Note that the algorithm was set to terminate upon the margin reaching a sufficiently low value *or* a maximum number of iterations being reached.) Although this seemed like an issue, given my lack of understanding how to fix it (prior to the realization that FE vectors should not be normalized) and noticeable improvements that I was able to achieve by optimizing the feature set, I ended up focusing more of my attention on feature engineering. Looking back on it now, the fact that the maximum margin rarely improved should have been much more of a red flag.

The (rather deep-seeded) flaw induced by normalizing the FE vectors explains another strange trend that I noticed throughout my experiments: often times the best trajectories produced by the learned policies were generated from the reward weights learned in the first several iterations of the algorithm. Given the lack of improvement in the maximum margin, this makes reasonable sense. The SVM achieved significant optimazation of the reward weights in early iterations while the demonstrator's FE vector was still significantly different from the first FE vector induced by the randomly initialized policy (in spite of the erroneous normalization). Thus the SVM was able to point the reward weights vector in generally the "correct direction" in early iterations, leading to significant improvements. However, after several iterations, if a number of "false positive" FE vectors were accumulated from policies that induced trajectories *artifically similar* to the demonstrators FE vector (meaning the FE vectors were really not very similar to the demonstrator's, but their normalized versions were), then the SVM would not be able to optimize the reward weights towards the direction of actual improvement.

One very positive takeaway, however, is that in spite of the flaw described above and its impact on the reward-weights optimization step of the algorithm, the SVM still managed to do a rather impressive job of getting the algorithm to learn a close approximation to optimal reward weights of the demonstrator. Although the optimal reward function of the demonstrator is unknown in this framework, thus a true measurement of similarity is not possible, by careful analysis of the reward weights learned by the SVM, the similarity can be qualitatively evaluated by seeing if the learned reward weights "make sense" given the motion style of the trajectory. In other words, if a given motion style is characterized by an arc-like movement (as in Demonstrator 2's trajectories), then it makes sense that the learned weight of the reward corresponding to the feature for ***angle of motion relative to goal = 30-60 degrees*** would be fairly high in the controller for the initial distance parition for that motion style (see figure 5). Careful qualitative analysis such as this across multiple experiments leads me to believe that the SVM works impressively well to learn the rewards weights, even despite the flaws discussed above. This suggests that this framework might be able to achieve quality results in this domain if the above flaws were addressed. Although it is always dissappointing when experimental results are subpar - and when flaws are recognized after experiments have already been performed - I view this as a positive development overall, since it helps explain some of the suboptimality, and suggests the possibility of much better results in future work.

D. RL Step

One of the major short comings of my algorithm was the lack of implementing a successful RL step using function approximation. I initially used Fitted Value Iteration, without much success. I then tried implementing SARSA gradient descent, using a linear combination of binary features (the same set of binary features as reward-learning was using plus a in an extra feature always set to 1 to capture the intercept). This did not perform very well either, so I ultimately ended up using a simple greedy policy that chose the action that achieved the maximum current reward at each timestep.

Although this seemed to perform better and allowed for signicant (and more direct) feature engineering, it ultimatley failed to learn high quality policies because future discounted rewards were not taken into account. I think this had a very negative impact on the agent's ability to learn to get to the goal.

In hindsight, I am unsure if the issues with my RL steps were implementation based, or were really rooted in the issues surrounding normalizing feature vectors - in which case, the poor performance of the RL algorithms might have been just another manifestation of this flaw.

Potential future work could include changing the RL step to use more traditional state-action features as input to the function approximator, such as x, y position, $\dot{x}, \dot{y}$, and $action_x, action_y$ rather than the same features as reward learning was using. Additionally, the RL step could perform function approximation using either a linear combination of fourier basis functions or a neural network. One item in particular that I am very intersted in exploring, but unsure how to implement, is how to backpropogate the error in a neural network when used in an RL function approximation setting. What should be used as the target values for training?

E. Other Frameworks

Additional future work could also involve switching to a different IRL framework. For example, the MaxEnt framework [4] and Deep IRL framework [8] are both very promising and arguably more powerful frameworks that this domain could be applied to.

References

[1] P. Abbeel. Markov decision processes and exact solution methods.
[2] P. A. Andrew Ng. Apprenticeship learning via inverse reinforcement learning. 2004.
[3] A. M. David Poole. Sarsa with linear function approximation, 2010.
[4] B. D. Z. et. al. Maximum entropy inverse reinforcement learning. 2008.
[5] A. Ng. Reinforcement learning and control.
[6] M. L. Richard Sutton. Control with function approximation, 2005.
[7] M. L. Richard Sutton. Sarsa: On-policy td control, 2005.
[8] M. Wulfmeier. Deep inverse reinforcement learning. 2015.