Reinforcement Learning Special Topic:
Imitation Learning

Aaron Dharna

April 2019

Contents:

1 Introduction

Imitation Learning has proved to be one of the key components for modern RL systems. In AlphaStar, DeepMind began the training of their StarCraft playing agent with learning from the game-traces of the top players. They used this as the starting point of their agent who then, through self-play, improved its policy to the point where AlphaStar took on the top StarCraft players in the world in the Fall of 2018. Furthermore, AlphaGO is built upon: Supervised learning + policy gradient methods + value function methods + Monte Carlo tree search!

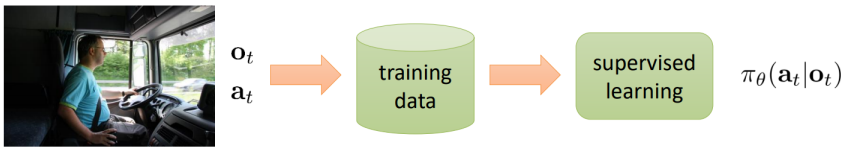
Our ultimate goal is to have an agent who can learn from watching an expert. Thus we will begin with the simplest formulation of this via **Behaviour Cloning** – we want to map states to an expert's actions.

We improve upon BC with **Inverse Reinforcement Learning**, but that comes at a computational and paradigm-shifting cost.

Finally, we will, next time, take a look at two **modern works** of Imitation Learning and take a look at what they bring to the table.

2 Behaviour Cloning

Figure 1: Behaviour Cloning



In a nutshell, Behaviour Cloning is supervised learning of an expert's policy. In the parlance of RL, this is control via Supervised Learning. We know our policy to be an arbitrary function from state-space to action-space, therefore, if we have state-action pairs, we can perform standard supervised learning techniques to learn that mapping. In this formulation, we are simply trying to mimic the expert and NOT surpass it.

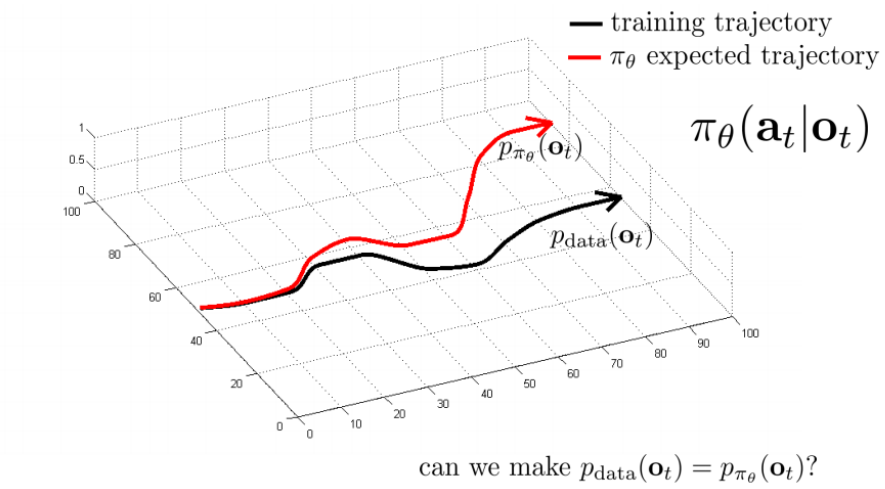
Before getting to surpassing our expert, we first need to address a big flaw with this technique – **dimensional drift**.

We know that in the supervised learning scenario, our standard methods achieve generalization – when given a new input, we produce the correct output (i.e.

when given a new picture of an airplane, we can in fact, realize that this is an airplane and not a gazelle).

We are not learning class labels here. We are attempting to learn what actions our agent should take when presented with a situation. We would hope that our agent could infer that when the car is drifting to the left that it should correct back to the right. However, that will not happen in BC. After we train on the expert's data, the policy that we generate will fail us if we **ever** leave the happy path.

Figure 2: Dimensional Drift



As soon as our agent sees a state that it has not been trained on, it will use the learned policy to make an action – but a non-optimal action (most likely). This will result in a slight error between what the agent does and what the expert did. Because of this error, the new state that the agent arrived in is different from what the expert gave us instructions for. Therefore, the next action we choose will continually pull us further away from the happy path the expert staked out. This forms a vicious cycle where we will eventually diverge completely from the trajectory the expert generated data for.

2.1 Fighting Dimensional Drift

The reason we drift away from the happy path is that our model is imperfect and that the small errors it makes compound into larger errors. Therefore, how might we fight that process?

What if we were to train on a dataset that had some "noise" where the expert's actions in the tangential state-space push the agent back towards the happy path?

Instead of trying to switch our policy to be more like the expert's we can instead augment the data distribution that we learn, $p_{data}(s_t)$.

DAGGER: Dataset Aggregation
given $\mathcal{D}$: dataset of expert data – $\{(s_1, a_1), ..., (s_n, a_n)\}$
while *True* **do**
  Train $\pi_\theta(a_t|s_t)$ from $\mathcal{D}$
  Run $\pi_\theta(a_t|s_t)$ to generate $\mathcal{D}_\pi = \{s'_1, ..., s'_m\}$
  Have a human label $\mathcal{D}_\pi$ with the optimal actions for $\{s'_1, ..., s'_m\}$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$
end

So, why is this potentially problematic:

1. What if we have non-markovian behaviour?
2. What if we have a multi-modal behaviour?
3. What if we don't have the workforce to expertly label the generated data in a corrective (and consistent) way?

These concerns are all side effects of trying to mimic the expert without understanding the environment at a deeper level. We already have a framework in place to do optimal control if we can receive states, actions, and rewards – namely Reinforcement Learning. In this case of behaviour cloning we are skating around the problem of the missing reward function by simply using the expert's actions as the absolute truth (1 there and 0 otherwise).

Therefore, what if we were to try to **recover an unknown reward function** when given expert data? Furthermore, what if we want our agent to **surpass our expert** or be able to achieve the same results in ways that we do not explicitly teach it?

3 Inverse Reinforcement Learning (eventually)

3.1 Latent Variable Models

Let $p(x)$ be an arbitrary distribution
Let $p(z)$ be a simple distribution (say, Gaussian)
Let $p(x|z)$ be a simple conditional distribution (say, conditional Gaussian)

$$p(x) = \int p(x|z)p(z)dz$$

You can represent any distribution, p(x), as a non-linear transformation upon a known simple distribution. As an example, if you want to sample from a gaussian distribution, you first generate a random number from 0 to 1, and then transform that generated number by the CDF of your gaussian (This is how computers do it).

Please, go see the 10/5/18 lecture from Berkeley for a full treatment of this material and as an intro to Variational Inference. It's really damn cool.

3.2 Reframing control as Probabalistic Inference

We have learned a lot about computational frameworks for optimal control algorithms in this course. Furthermore, when people act, we can assume they're doing so rationally, at least sometimes, so how can those frameworks be used to make predictions about near-optimal agents.

We assume that an ***optimal*** agent, when choosing what actions to take, will solve the following system:

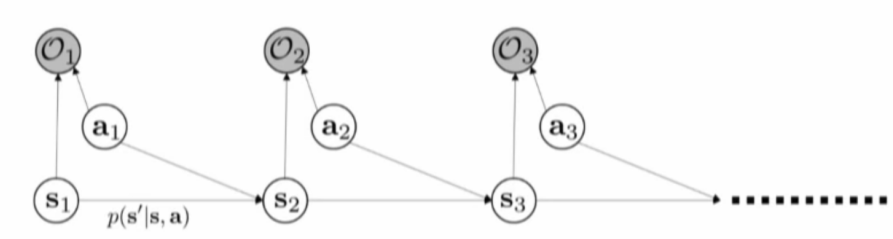
$$a_1, ..., a_T = argmax_{a_i} \sum_i r(s_i, a_i)$$

$$s_{t+1} = f(s_t, a_t)$$

IF we have seen an agent who acts according to this framework, then can we optimize $r(s, a)$ to explain the data? This would tell us what their goal might be. However, this would be for an optimal agent.

To generalize this into a near-optimal agent, we need to introduce a new variable into our framework – $\mathcal{O}_t$ an optimality variable. This is to model why the agent might have taken the (near optimal)-action it took. Introducing $\mathcal{O}_t$ into our MDP system results in:

Figure 3: Augmented MDP



Let,

$$(s_{1:T}, a_{1:T}) = \tau$$

$p(\tau)$ tells us what is the probability of the trajectory we observed happening given the dynamics of our system. We encode the new optimality information by ***picking*** a distribution for $p(\mathcal{O}_{1:T}|\tau)$

So, our goal is now to figure out $p(\tau|\mathcal{O}_{1:T})$ Let us first choose:

$$p(\mathcal{O}_t|s_t, a_t) \propto \exp(r(s_t, a_t))$$

Then,

$$p(\tau|\mathcal{O}_{1:T}) = \frac{p(\tau,\mathcal{O}_{1:T})}{p(\mathcal{O}_{1:T})} \propto p(\tau)\Pi_t \exp(r(s_t, a_t)) = p(\tau)\exp(\textstyle\sum_t(r(s_t, a_t)))$$

This here is saying that the probability of the trajectory occurring is proportional to the chance of the trajectory randomly occurring within the dynamics times the exponential of the sum of the total reward of the trajectory. A consequence of this is that sub-optimal trajectories can still occur but become exponentially less likely as the reward decays.[1]

If we know τ then we can invert this paradigm to figure out the reward function! Furthermore, this framework allows us to use inference problems to solve control and planning problems and this model of agents who act sub-optimally is a useful one if we want to prime exploration.

If we train with mistakes then at test time when the environment gets perturbed, the agent's actions will be robust against these new changes. **This helps us fight against the dimensional drift problem seen in behaviour cloning!**

Planning/decision making amounts to **solving an inference problem** – given evidence, find the posterior probability of the other variables in the system.

How do we compute the probability that all our remaining actions are optimal from some given time step:

$$\beta(s_t, a_t) \coloneqq p(\mathcal{O}_{t:T}|s_t, a_t)$$

These backward messages will allow us to calculate a policy very effectively where a policy is defined as an action, based on the current state, and that the optimality variables are true.

$$\pi(a_t|s_t, \mathcal{O}_{1:T}) \coloneqq p(a_t|s_t, \mathcal{O}_{1:T})^{2}$$

Finally, we need to compute forward messages which will be very useful when we try to infer reward functions in IRL:

$$\alpha(s_t) \coloneqq p(s_t|\mathcal{O}_{1:t-1})^{3}$$

[1] The agent will usually take the correct action, however, it will make mistakes, but those mistakes will be the ones which are the least costly.

[2] This can be understood as, what is the probability that you take a certain action, given a state and the fact that you're trying to reach a certain goal.

[3] which say, what is the probability that you will land in a certain state if you have acted optimally up until now.

To calculate $\beta_t(s_t, a_t)$ we do the following:

$$\beta_t(\mathbf{s}_t, \mathbf{a}_t) = p(\mathcal{O}_{t:T}|\mathbf{s}_t, \mathbf{a}_t) = \int_{\mathcal{S}} \beta_{t+1}(\mathbf{s}_{t+1}) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) p(\mathcal{O}_t|\mathbf{s}_t, \mathbf{a}_t) d\mathbf{s}_{t+1}.$$

where $\beta_t(s_t)$ is:

$$\beta_t(\mathbf{s}_t) = p(\mathcal{O}_{t:T}|\mathbf{s}_t) = \int_{\mathcal{A}} p(\mathcal{O}_{t:T}|\mathbf{s}_t, \mathbf{a}_t) p(\mathbf{a}_t|\mathbf{s}_t) d\mathbf{a}_t = \int_{\mathcal{A}} \beta_t(\mathbf{s}_t, \mathbf{a}_t) p(\mathbf{a}_t|\mathbf{s}_t) d\mathbf{a}_t.$$

This results in a recursive definition for calculating the backwards message of $\beta_t(s_t, a_t)$.

Let

$$V(s_t) \coloneqq \log(\beta(s_t))$$

and similarly, let

$$Q(s_t, a_t) \coloneqq \log(\beta(s_t, a_t))$$

then, when you plug those definitions into our equations above, you'll find this:

$$V(s_t) = \log(\int (\exp Q(s_t, a_t)))$$

$$Q(s_t, a_t) = r(s_t, a_t) + \log(\mathrm{E}(\exp(V_{t+1}(s_{t+1}))))$$

Similarly, we can calculate $\pi_t(a_t|s_t, \mathcal{O}_{1:T})$:

$$p(\mathbf{a}_t|\mathbf{s}_t, \mathcal{O}_{t:T}) = \frac{p(\mathbf{s}_t, \mathbf{a}_t|\mathcal{O}_{t:T})}{p(\mathbf{s}_t|\mathcal{O}_{t:T})} = \frac{p(\mathcal{O}_{t:T}|\mathbf{s}_t, \mathbf{a}_t) p(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_t)}{p(\mathcal{O}_{t:T}|\mathbf{s}_t) p(\mathbf{s}_t)} \propto \frac{p(\mathcal{O}_{t:T}|\mathbf{s}_t, \mathbf{a}_t)}{p(\mathcal{O}_{t:T}|\mathbf{s}_t)} = \frac{\beta_t(\mathbf{s}_t, \mathbf{a}_t)}{\beta_t(\mathbf{s}_t)}$$

Furthermore, if we add in the Q and V transformations, we find that our policy is:

$$\pi_t(a_t|s_t, \mathcal{O}_{1:T}) = \exp(Q(s_t, a_t) - V(s_t)) = \exp(A(s_t, a_t)))$$

Finally, what is $\alpha_t(s_t)$ (the probability that we landed in a given state if we were optimal up until this point):

$$p(s_t|\mathcal{O}_{1:t}) = \int \int p(s_t|s_{t-1}, a_{t-1}) p(a_{t-1}|s_{t-1}, \mathcal{O}_{t-1}) \alpha_{t-1}(s_{t-1}) ds_{t-1} da_{t-1}$$

where:

$$p(a_{t-1}|s_{t-1}, \mathcal{O}_{t-1}) = \frac{p(\mathcal{O}_{t-1}|a_{t-1}, s_{t-1}) p(a_{t-1}|s_{t-1})}{p(\mathcal{O}_{t-1}|s_{t-1})} \propto \exp(r(s_t, a_t))$$

However, that is not quite what we want: We want, $p(s_t|\mathcal{O}_{1:T})$ or what is the probability of landing in a given state along a near-optimal state/action trajectory.

$$p(s_t|\mathcal{O}_{1:T}) = \frac{p(s_t, \mathcal{O}_{1:T})}{p(\mathcal{O}_{1:T})} = \frac{p(\mathcal{O}_{t:T}|s_t)p(s_t, \mathcal{O}_{1:t-1})}{p(\mathcal{O}_{1:T})} \propto \beta(s_t)p(s_t|\mathcal{O}_{1:t-1}) = \beta(s_t)\alpha(s_t)$$

If I want to know the probability of being in a given state, in a distribution over near-optimal trajectories, all I need to do is multiply together the forward and backward state messages and then normalize!

This framework allows us to analyze near-optimal behavior of other agents and recover their rewards fns! Furthermore, These softened versions of our RL algorithms improve exploration by introducing **stochasticity and maximizing entropy** of the distribution over the optimality variables.

3.3 Entropy

Differential Entropy is defined as:

$$\mathcal{H}(p) = -\mathrm{E}_{x\sim p(x)}[log(p(x))] = -\int_x p(x)log(p(x))dx$$

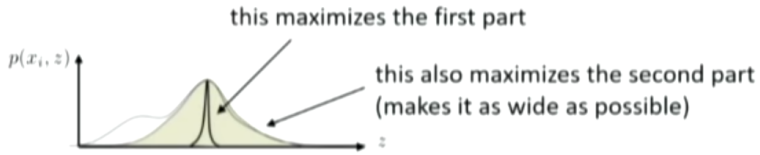
Entropy can be thought of in two ways:

1. How random is a random variable? Deterministic random variables have the least entropy and stochastic random variables have the most entropy.
2. How large is the log probability of this distribution in expectation under itself. If the distribution has low log probability then it assigns low probabilities to the things it is likely to pick – therefore the variable is quite random since the distribution must sum up to one.

In layman terms, this can be thought of as 'how wide is the distribution, for continuous random variables?'

How will this be useful though: If we are trying to maximize a distribution that we're learning (like, say, over the optimality variables), if there is an entropy term, the entropy term will force us to make the distribution we're learning as wide as possible.

$$E_{z\sim q_i(z)}[\log p(x_i|z) + \log p(z)] + \mathcal{H}(q_i)$$



3.4 Inverse Reinforcement Learning (IRL)

3.4.1 Motivation

IRL[4] focuses on recovering an unknown reward function. According to Ng and Abbeel, "for many problems, the difficulty of manually specifying a reward function represents a significant barrier to the broader applicability of reinforcement learning and optimal control algorithms."

Where do reward functions come from?

Typically, they're highly problem dependent and need to be engineered specifically for the problem. For example, when NCSOFT were creating 'Arena Battle AI for Blade and Soul' using RL algorithms, their final reward function was crafted to reward winning **and** maximizing the difference between health-bars.

Figure 4: Crafting Reward Fns

When learning Atari, DeepMind used the built-in score of the game as their reward function. However, what if we cannot think of a reward function which would maximize **exactly** what we want? When Ubisoft was building some self-driving car tech inside of Watch-Dogs 2, they found that their initial reward

[4]This is also called Inverse Optimal Control: there is a long history of this research in Optimization and Control theory.

function was also maximized by the car going backwards and driving at the edge of the road – in direct contrast with what they wanted: driving forward and in the middle of your lane. These methods are very vulnerable to reward misspecification.

So, if we cannot think of exactly the function we want to maximize, what if we were to instead try to learn that function from an experts example demonstrations? Ideally, we want our agent to recover/infer the goal from the expert's demonstrations so that the agent can optimize the path to the goal itself – resulting in behaviours that the expert did not use, but still reaches the same goal.

Can we reason about ***what*** the expert is trying to achieve?

Given Info:

1. State and Actions trajectories of an expert
2. Samples from π^* (a near-optimal policy)
3. Dynamics Model (sometimes)

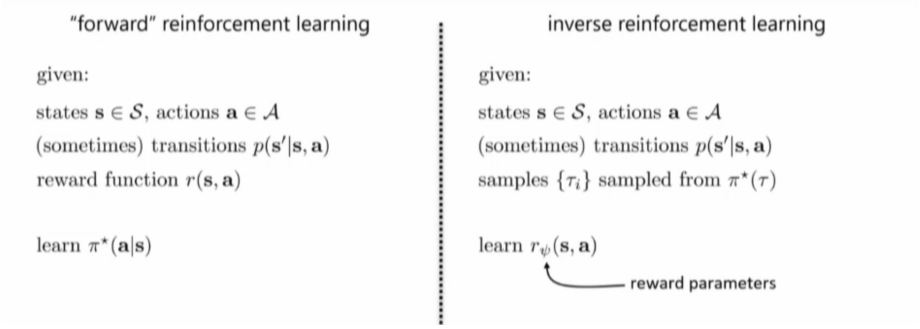
Goal:

1. Recover expert's reward Fn $\leftarrow$ IRL
2. Use the recovered reward Fn to generate a policy $\leftarrow$ RL

However, there are many challenges to this: The problem can be under-defined, it can be difficult to measure your learned reward function, or the demonstrations you're trying to learn from may not actually be optimal. As such, we need a way to account for these things. Furthermore, we now find our selves extending our learning into two tasks where one of them is dependent on the first!

You can use concepts from RL and control to analyze human behaviour. If you assume that person is acting in a near-optimal way you can observe their behaviour, model it as though they were solving a control problem, and then perhaps you can figure out what their objective was. Furthermore, you can use generative models (including latent variable models) for exploration! In exploration these models can be very useful in determining if this is a state you have never seen before or have seen many times. This can then be used to guide exploration and have your policy visit new states.

Figure 5: RL vs IRL



3.4.2 Math

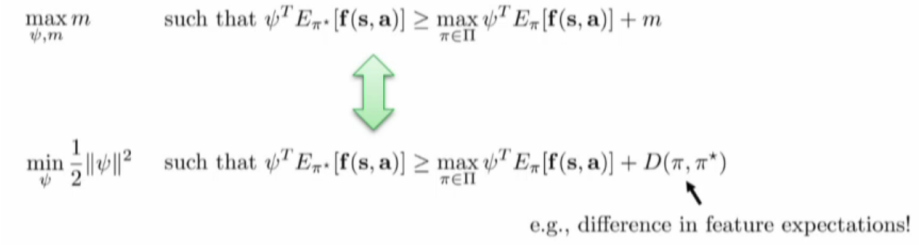
There are two main frameworks for this task:

1) **Linear function approximation** [5]

$$r_\psi(s, a) = \psi^T f(s, a)$$

This format used, as a proxy for measuring differences between policies, expectations of features and attempted to **maximize the margins** between the infinitely potentially many policies. Given that you have an expert's trajectory, you can figure out the expected value of all of the features the expert saw.[6] Given that you assume those feature values represent good behavior, you try to ensure that the learned reward fn matches the expert's distribution and base your margin on the difference between the expectations of the features seen in π_θ and π^*.

Figure 6: Margin-based IRL



2) Probabilistic and Maximum Entropy methods

[5] see Ng and Abbeel '04 and Ng and Russell '00

[6] For example, if you're building a self-driving car a good policy will never crash the car. Therefore the expected value of that feature will be 0. You want the reward fn (and subsequent policy) that you learn to have the same distribution of features.

The augmented MDP methods we talked about earlier allow us to arrive at soft-optimal solutions that favor the optimal behavior, but still are able to execute sub-optimal actions. This allows the agent to have a goal-driven behavior but still make small mistakes that are not impactful to the overall goal.

Remember: learning the reward function of our probabilistic model amounts to learning the distribution over the optimality variables.

$$p(\mathcal{O}_t|s_t, a_t, \psi_t) \propto \exp(r_\psi(s_t, a_t))$$

As above,

$$p(\tau|\mathcal{O}_{1:T}, \psi) \propto p(\tau) \exp(\sum_t r_\psi(s_t, a_t))$$

We want to optimize this probability for some example trajectory, then we can do MLE optimization with respect to the variables encapsulated in ψ and maximize the log probabilities of the trajectory. If we take the log of the exponential of the total rewards which then our objective becomes maximize the reward of the demonstrations minus a normalizing constant, $\mathcal{Z}$.

$$\begin{aligned}
\mathcal{L} &= p(\tau) \exp(\sum_t r_\psi(s_t, a_t)) \\
&= \max_\psi(\frac{1}{N} \sum_i (\log p(\tau|\mathcal{O}_{1:T}, \psi)) - \log \mathcal{Z}) \\
&= \max_\psi(\frac{1}{N} \sum_i (r_\psi(\tau_i) - \log \mathcal{Z}))
\end{aligned}$$

where Z is a normalizing partition function over all possible trajectories. [7]

$$\mathcal{Z} \coloneqq \int_\tau p(\tau) \exp(r_\psi(\tau)) d\tau$$

Therefore, our derivative of $\mathcal{L}$ with respect to ψ is:

$$\nabla_\psi \cdot \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \nabla_\psi r_\psi(\tau_i) - \frac{1}{\mathcal{Z}} \int p(\tau) \exp(r_\psi(\tau)) \nabla_\psi r_\psi(\tau_i) d\tau$$

Note that $\frac{1}{\mathcal{Z}} p(\tau) \exp(r_\psi(\tau))$ is simply $p(\tau|\mathcal{O}_{1:T}, \psi)$ – the normalized probability that this trajectory occurred given our current reward function. Similarly, the first half of the expression is the probability that this trajectory occurred under the expert's policy. Therefore, if we switch to a sample based method, we can switch out this $\sum$ and $\int$ for Expectations.

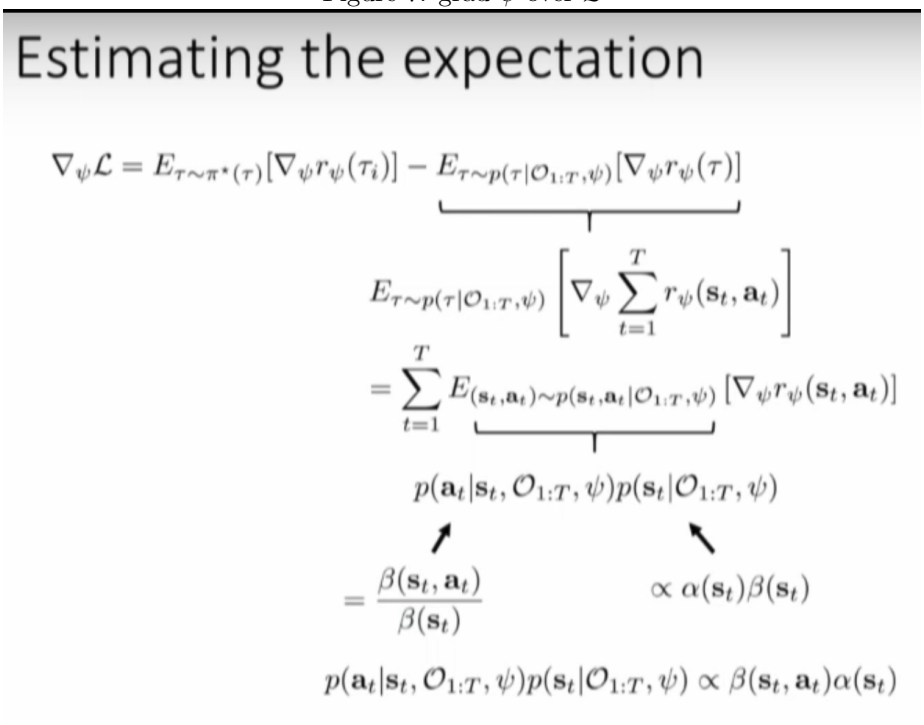
$$\nabla_\psi \cdot \mathcal{L} = \mathrm{E}_{\tau \sim \pi^*}[\nabla_\psi r_\psi(\tau_i)] - \mathrm{E}_{\tau \sim p(\tau|\mathcal{O}_{1:T}, \psi)}[\nabla_\psi r_\psi(\tau_i)]$$

[7] You don't want to just maximize the reward at the states/actions the expert shows you, otherwise you'd just make everything large at the start. You also want to minimize the reward everywhere else (a more sophisticated version of what we blindly did in Behavior Cloning!)

The first term here we can calculate from the expert data, while the second will be estimated by calculating the Max-Entropy optimal policy for the current reward using probabilistic inference.

Note: this is where we're going to pull back in the forward/backward message set up from probabilistic inference.

Figure 7: grad ψ over $\mathcal{L}$



Going one step further, we can let:

$$\mu_t(s_t, a_t) = \beta(s_t, a_t)\alpha(s_t)^{8}$$

then:

$$\mathrm{E}_{\tau\sim p(\tau|\mathcal{O}_{1:T},\psi)}[\nabla_\psi r_\psi(\tau_i)] = \sum_{1}^{T} \int\int \mu_t(s_t, a_t)\nabla_\psi r_\psi(\tau_i) da_t ds_t$$

This double integral is simply taking, for all time steps, take every combination of the states and actions.

[8] Essentially, given some state and action, what is the probability that you're in that state-action pair.

This actually gives us everything we need for the:

Maximum Entropy IRL algorithm:

1. given ψ, compute $\beta(s_t, a_t)$
2. given ψ, compute $\alpha(s_t)$
3. compute $\mu_t(s_t, a_t)$
4. Let

$$\nabla_\psi \cdot \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\psi r_\psi(s_{i,t}, a_{i,t}) - \sum_{i=1}^{T} \int \int \mu_t(s_t, a_t) \nabla_\psi r_\psi(s_t, a_t) ds_t da_t$$

5. $\psi \leftarrow \psi + \eta \nabla_\psi \cdot \mathcal{L}$
6. repeat until convergence

   see Ziebart et al - Maximum Entropy Inverse Reinforcement Learning

This amounts to a Maximum Likelihood Optimization. It is trying to find the ψ that maximizes the likelihood of the demonstrations that you saw under this model of an approximately optimal policy!

In the case where $r_\psi(s_t, a_t) = \psi \cdot f(s_t, a_t)$, we can show that:

$$\max_\psi \mathcal{H}(\pi^{r_\psi}) \text{ such that } \mathrm{E}_{\pi^{r_\psi}}(f) = \mathrm{E}_{\pi^*}(f)$$

where we are trying to maximize entropy of our policy subject to the constraint where our feature expectations are equal to the experts feature expectations. Furthermore the best policy is now the one which matches feature expectations **and** assumes as little as possible about the rest of the policy!

With non-linear function approximators, the same interpretation as far as maximizing reward plus maximizing entropy will still hold.

4 Modern Work

4.1 Connections between GANs, IRL, and Energy Based Models

Key inference here is that: We can look at the interactions between IRL and RL as a game!

Maximum Entropy IRL has provided us with a probabilistic framework for learning reward function, however, computing the necessary gradients require

enumerating state-action visitation probabilities, $\mu_t(s_t, a_t) \forall s, a$. This format is only feasible for small discrete state and action spaces, and amounts to dynamic programming of the forward-backward inference algorithm we described above. Ideally, we would be able to handle large state/action spaces, and be able to learn under unknown system dynamics via sampling methods.

Recall our form of the gradient of $\mathcal{L}$ above:

$$\nabla_\psi \cdot \mathcal{L} = \mathrm{E}_{\tau \sim \pi^*}[\nabla_\psi r_\psi(\tau_i)] - \mathrm{E}_{\tau \sim p(\tau|\mathcal{O}_{1:T}, \psi)}[\nabla_\psi r_\psi(\tau_i)]$$

where the first term is estimated via samples from the expert and the second term is estimated via samples drawn from our soft-optimal policy under our current reward function by running regular RL using any of our known model-free methods.

idea: learn $p(\mathbf{a}_t|\mathbf{s}_t, \mathcal{O}_{1:T}, \psi)$ using any max-ent RL algorithm
then run this policy to sample $\{\tau_j\}$

$$J(\theta) = \sum_t E_{\pi(\mathbf{s}_t, \mathbf{a}_t)}[r_\psi(\mathbf{s}_t, \mathbf{a}_t)] + E_{\pi(\mathbf{s}_t)}[\mathcal{H}(\pi(\mathbf{a}|\mathbf{s}_t))]$$

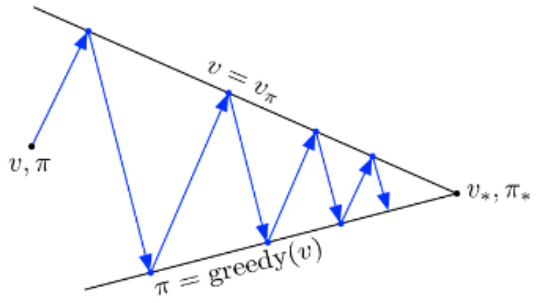
$$\nabla_\psi \mathcal{L} \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\psi r_\psi(\tau_i) - \frac{1}{M} \sum_{j=1}^{M} \nabla_\psi r_\psi(\tau_j)$$

sum over expert samples — sum over policy samples

This formulation is highly expensive to run – any time we make a change to our reward parameters, ψ, we need to recompute our gradient which means running a full sweep of the RL algorithm we used to generate our policy. So, if we needed N steps to learn our reward function, we would need to run Q-learning N times.

Perhaps, what we might do here is rather than outright learning the best policy at each time/reward increase, we might simply improve our policy a little bit. We might only take a small step in the direction of improvement a la:

Figure 8: Generalized Policy Improvement



However, if we take this route, then our estimator has become biased!

Therefore, we need to, using samples from a given distribution, estimate an expectation under a different distribution. Welcome back Importance Sampling since calculating importance weights is easier than calculating optimal policies.

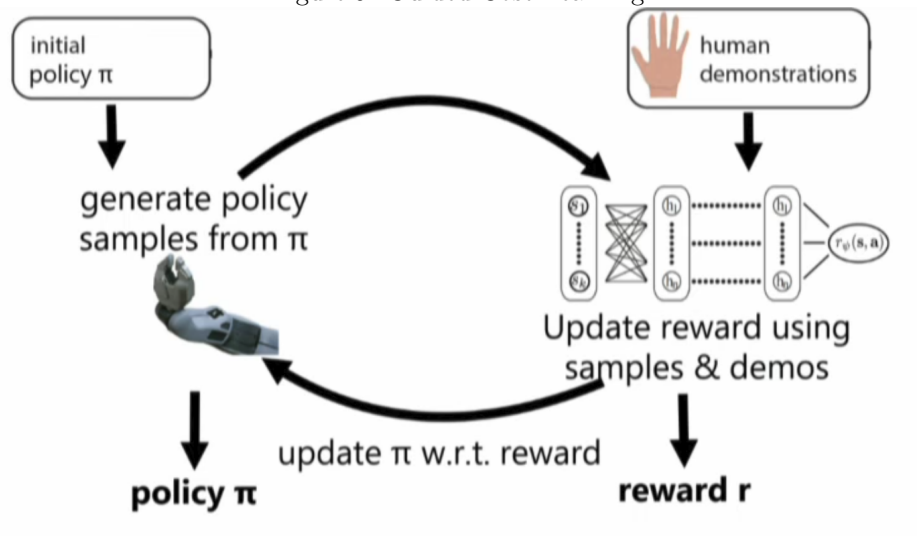
Our gradient will look about the same, but we will need to weight our biased term by some error correction term.

$$\nabla_\psi \cdot \mathcal{L} \simeq \frac{1}{N} \sum_{i=1}^{N} \nabla_\psi r_\psi(\tau_i) - \frac{1}{\sum_j w_j} \sum_{j=1}^{M} w_j \nabla_\psi r_\psi(\tau_j)$$

where

$$w_j = \frac{\exp(r_\psi(\tau_i))}{\pi_{r^\psi}(\tau_j)}$$

Figure 9: Guided Cost Learning



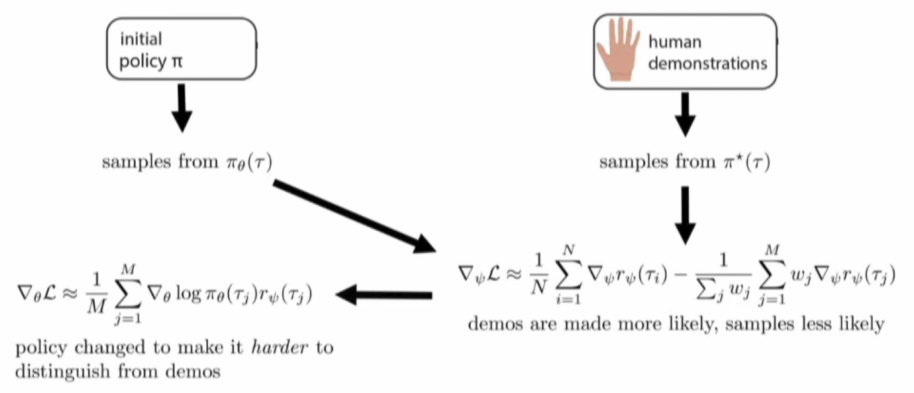
If you look closely to this, you might notice that it looks a bit like a game[9]!

The intuition here comes from $\nabla_\psi \cdot \mathcal{L}$. When we maximize this sum, the first term is us trying to maximize the trajectories that the expert shows us, and the second term is minimizing everywhere else in the search space. We then use the extracted reward function to train our policy which will produce a new trajectory.

[9]To go further, please follow the link above to Dr. Levine's lecture. This is timestamped to pick up directly at this point.

In the ideal world, this gradient will go to zero as there is no difference between the expert's trajectory and our policy-extracted-from r_ψ. This makes the gradient signal act like the discriminator in a GAN, while the policy we're trying to learn, if optimal, will "fool"/be-equal-to the expert's data.

Figure 10: IRL → GAN



Once the policy is the optimal policy and r_ψ equals r_{π^*}, (the distribution over our optimality variables) then in expectation, the difference is zero.

4.2 Imitation Learning with Concurrent Actions in 3D games

According to Harmer et al: "When training agents to interact in complex environments with large action spaces, the behaviour associated with having a single action per time step (SAPS) policy, as is almost always the case in RL, is often undesirable." This could be for a multitude of reasons – complex behaviours often require multiple actions to be taking place simultaneously – i.e. moving forward and shooting in a video game at the same time.

As we saw with Mnih et al in Atari, deepRL networks rely on modeling all possible combinations of moves, and effectively discritizing the action space. However, as the action space gets larger, the ability to enumerate all possible combinations is quickly reduced to zero.

> Modeling all possible combinations [of a video game controller] would require a policy that has 2^{20} output neurons. Joint action representations in such a large action space make it much harder for the agent to learn the value of each of the true actions, and do not take advantage of the underlying relationships between different sets of individual actions. ... Instead of applying imitation learning as a

> pre-training step [as AlphaGo did], we apply it at the same time as TD RL as a way of regularizing the TD learning. Each batch update is comprised of both expert and live agent data. At every update step, the network predicts the action of the expert, from a sample of the expert data, whilst learning a policy that maximises the discounted future reward of the live agent stream. Training the network in this way allows the network to maintain a valid TD learning compatible state, throughout training (Harmer et al).

The Algorithm that Harmer et al come up with is Multi-Action per time step Imitation Learning, or MAIL.

Figure 11: MAIL

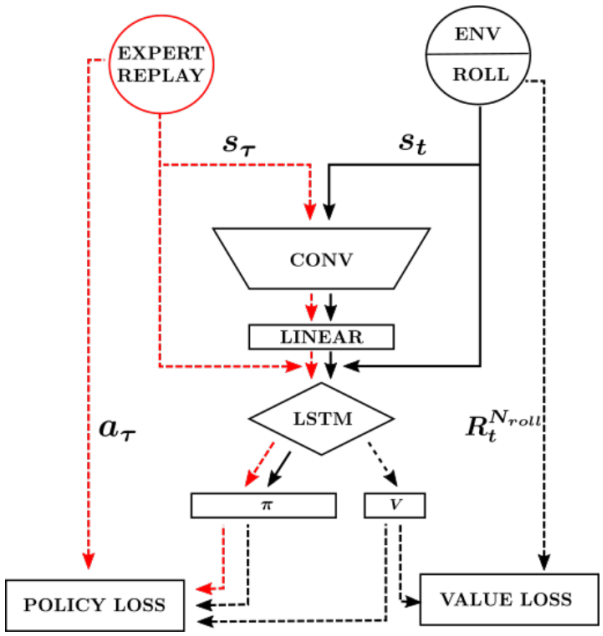


Fig. 3: MAIL Neural network architecture. Solid black lines represent the flow of data during inference. Dashed lines represent the flow of data during training. Red lines represent the flow of expert data.

As noted in the analysis portion of the paper:

This speed-up [granted by combining IL with TD RL] is most pronounced in the early stages of training when reward sparsity severely limits the effectiveness of TD learning updates; imitation learning provides useful feedback at every training step from the very start of training. Supervised learning allows the vision system to be trained much more rapidly than TD RL. Further, mimicking the behaviour of the expert significantly improves the exploration of state-space in comparison to the unguided random actions in the early stages of TD RL.

5 Further Resources

-Berkeley DeepRL Course 2018-2019 ← Highly Recommend.
Figures 1, 2, 3, 5, 6, 7, 9, 10, and several integrals are from the above lecture series along with:
-Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review
-Maximum Entropy Inverse Reinforcement Learning
-Modeling Interactions via Maximum Causal Entropy
-ICML Imitation Learning tutorial
-Apprenticeship learning via Inverse Reinforcement Learning
-A Connection between Generative Adversarial Networks, Inverse Reinforcement Learning, and Energy-Based Models
-Generative Adversarial Imitation Learning
-Imitation Learning with concurrent actions in 3d games
Figure 11 comes from the directly-above paper.