2.7 Impala: Importance weighted actor-learning architecture

- NOT actor-critic, it is just a critic (DQN augmented)
- off-policy in the sense that the actors are distributed and experience is used in a buffer by a central learner
- discrete actions

Can scale to thousands of machines. Many actors are distributed and generate trajectories which are fed to a global (actually several "global") learner (GPU). They also use something similar to Retrace(λ) with a new *V-trace* off-policy actor-critic algorithm to correct for difference in the log caused by actors using "old" parameters (not using the most recent parameters of the learner). They use a CNN + LSTM structure. When the global learner receives a batch of trajectories, the time domain (trajectories) are simply rolled into the batch dimension and all into the CNN in parallel. As well, it outputs all the output layers for all time steps in parallel once all the LSTM states have been computed. This increases the effective batch-size to thousands. They also introduce other speed ups available in TensorFlow, etc. Essentially, over A3C, they are able to increase the frames per second significantly.

2.8 Unicorn

- NOT actor-critic, it is just a critic (DQN augmented)
- off-policy in the sense that the actors are distributed and experience is used in a buffer by a central learner
- discrete actions

Unicorn is acontinual learning agent with universal off-policy updates: UVFA. The overall architecture is a distributed setup of M actors acting in parallel with the current set of parameters. Each actor is acting with respect to a different goal or task (authors define these differently). The rollouts of experience $\{s_0, a_0, ...\}$ for each of the M actors is transfered the the learner UVFA via a queue. Loss functions are computed, gradient updates, then after each training step the M actors are synchronized with the latest global UVFA parameters. A task defines a different reward perspective (e.g. r_{key}, r_{door}), which the agent receives at every time step, no matter what its current goal is. A goal encodes a certain behaviour.

The $M = 200$ actors are each trained on distinct tasks separately. The architecture is a CNN followed by an LSTM. The goal is injected at a later layer, as is a "inventory stack" which captures the most recent items the agent has collected in the environment. n-step learning is performed, however due

to the off-policy nature of the data collection, the n-step return is truncated at time t where a_t in the replay buffer does not match the current policy. The learner (UVFA) batches trajectories of experienced pulled from the global queue. Batching happens across all K training tasks, and across batch size $B \times H$, H being the trajectory length. No target network is used and no replay buffer. The domain is a deepmind domain that is a gian room with tv's, balls, balloons and cake to represent keys, doors, locks and chests.

The experiments were done quite well, and tested transfer and zero-shot transfer as well. I won't go into it too much, but the results were decent. They also showed that tasks with dependencies (e.g. get the key before going to the lock) were learned under this structure.

3 Successor Features and Successor Representations

3.1 Successor Features for Transfer in RL

This paper has started quite a bit of research. The idea is that the agent is found in non-changing environment, and the only thing that changes is the reward function. The model assumes that the reward can be decomposed, and has the following structure:

$$r(s,a) = \phi(s,a)^T w$$

where $\phi(s,a) \in \mathbb{R}^d$ are features of (s,a), and w are weights. Hence, the reward function is completed determined by w. The resulting representation of Q can be seen as:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + ... | S_t = s, A_t = a] \\ &= \mathbb{E}[\phi_{t+1}^T w + \gamma \phi_{t+2}^T w + ... | S_t = s, A_t = a] \\ &= \mathbb{E}[\sum_{i=t}^{\infty} \gamma^{i-t} \phi_{i+1} | S_t = s, A_t = a]^T w \\ &= \Psi^\pi(s,a)^T w. \end{aligned}$$

Here, $\Psi^\pi(s,a)$ are the Successor Features of (s,a) under π. In words, they are the expected discounted sum of features to be experienced under the current policy. Due to the assumption of the reward function decomposition, if a new task is provided and encoded as w', then the value of the prior policy π can immediately be computed as $\Psi^\pi(s,a)^T w'$.

Note, if $\phi(s,a)$ is a one-hot encoding of (s,a), then this is simply the successor representation. Another note, if $r(s,a) = -1$, $\forall (s,a)$, then w should be a constant function, and hence not provide any form of transfer. Hence, goal

based settings such as this are not conducive to this formulation. Since in the RL setting neither Ψ^π nor w are given, they must be learned. Learning w is simply supervised learning, and nicely,

$$\Psi^\pi(s,a) = \phi_{t+1} + \gamma \mathbb{E}[\Psi^\pi(S_{t+1}, \pi(S_{t+1}))|S_t = s, A_t = a],$$

and hence follow the same recursive structure as any TD learning algorithm.

Transfer: MDP framework The authors assume that $\phi \in \mathbb{R}^d$ is fixed, and this defines a class of MDPs,

$$\mathcal{M}^\phi(\mathcal{S}, \mathcal{A}, p, \gamma) = \{\mathcal{M}(\mathcal{S}, \mathcal{A}, p, r, \gamma) | r(s,a) = \phi(s,a)^T w,\ w \in \mathbb{R}^d\}.$$

The idea for transfer is that, given that n tasks have been solved by now, the agent can use "generalized policy improvement" to find a new policy, based purely on the old ones, that are "no worse" than the older polices.

Theorem 1. (Generalized Policy Improvement) Let $\pi_1, ..., \pi_n$ be n decision policies and let $\tilde{Q}^{\pi_1}, ..., \tilde{Q}^{\pi_n}$ be their respective approximations such that $|Q^{\pi_i}(s,a) - \tilde{Q}^{\pi_i}(s,a)| \leq \epsilon$, $\forall (s,a)$ and $\forall i \in \{1, 2, ..., n\}$. Define $\pi(s) \in argmax_a\ max_i\ \tilde{Q}^{\pi_i}(s,a)$. Then,

$$Q^\pi(s,a) \geq max_i\ Q^{\pi_i}(s,a) - \frac{2\epsilon}{1-\gamma},\ \forall (s,a).$$

. The authors claim this as a positive result for transfer. However, without lower bounds on Q^{π_i} **with respect to the current task**, these bounds don't say anything. The authors then also consider when $\epsilon = 0$, then this new transfer policy does no worse than the old policies. However, I proved that under this condition, the result is equivalent to saying that under every π_i, all the associated Q vale functions are exactly equal the same and exactly equal to Q_{max}. So it is a trivial result. Moreover, Lemma 1 is also completely trivial as it states: *Let* $\delta_{ij} = max_{s,a}|r_i(s,a) - r_j(s,a)|$. Then,

$$Q_i^{\pi_i^*}(s,a) - Q_i^{\pi_j^*}(s,a) \leq \frac{2\delta_{ij}}{1-\gamma}.$$

This Lemma states that, given two separate tasks, give me the max reward in one, and the smallest reward in the other, then take the optimal policies from both tasks and evaluate them in one of those tasks, and the difference in the rewards is no worse than the bounds provided. However, we can improve this (trivially) by a factor of 2. Since in either task, any value function is bounded by $V_{min} = \frac{r_{min}}{1-\gamma} \leq V \leq V_{max} = \frac{r_{max}}{1-\gamma}$, where the bounds come from a policy always either getting the min or the max rewards. We see that for the same task $\delta_{ii} = r_{max} - r_{min}$ and so for any two policies, the difference is bounded above by $\frac{\delta_{ii}}{1-\gamma}$. Now, their result is true, it might be that $\delta_{ij} > \delta_{ii}$, uless the scale of the rewards wildly change from task to task, this bound is useless. Mostly, people assume all rewards can be rescaled to r_{min} and r_{max}, and do so in practice, to make training more stable.

Finally, their last theorem states that using this greedy policy, it is not worse than the optimal policy for this new task by the upper bound of $\frac{2}{1-\gamma}(\phi_{max}\, min_j||w_i - w_j|| + \epsilon)$. However, it seems that it is up to the researcher to determine ϕ, and hence we do not know what bounds are on this value. Moreover, the same can be said with respect to w. Without these sorts of bounds, these results have no real meaning.

3.2 Deep Successor RL

This paper uses a DQN (CNN) to do Deep Successor Representations. They use a CNN with a fixed *feature* layer as the first (512 dimension) layer after the CNN. This then goes to a deconv layer that aims to reconstruct the input image. This feature layer also maps to $\mathcal{A}$ deep neural net heads which attempt to estimate the successor representation for each action. It attempts to estimate the expected future "feature occupancy" conditioned on this state and the actions. Finally, from the aforementioned feature layer, a linear weight layer w is used to attempt to predict the immediate reward. Given the successor representation:

$$M(s, s', a)\mathbb{E}\Big[\sum_{t\geq 0}\gamma^t \mathbb{1}[s_t = s']|s_0 = s, a_0 = a\Big].$$

We note that the SR has a recursive Bellman structure, and moreover $Q^\pi(s, a) = \sum_{s'\in\mathcal{S}} M(s, s', a)R(s')$. Hence, once the SR is learned, it is sufficient to simply learn the immediate reward.

The results are ok, but what is interesting is they maintain two separate replay buffers: one of non-zero rewards, and one with all rewards.

3.3 Universal Successor Representations for Transfer RL

This paper of Bengio's uses a similar definition as DeepMind's Sucessor Features. They assume that $\phi(s, a, s')^T w = r(s, a, s')$, though in their implemention $\phi(s)$ is a function of the current state only. Their architecture is parameterized by goals, like a UVFA. Hence, two inputs, s, g, and one network is simply an autoencoder, using some bottleneck layer as $\phi(s)$. g is input to a regression layer to predict $w(g) = w_g$, e.g. the weight vector that encodes g. The other architecture takes both s, g as inputs, and with a early shared layer, then split with separate parameters, attempts to learn the policy (the actor) $\pi(s; g)$, and the successor represention $\Psi(s; g)$. The loss functions are L_w for squared loss of $r_t - \phi(s_{t+1})^T w(g; \theta_w)$. Hence, trying to learn a good w representation. The next one is the SR loss, the MSE loss of $\phi(s_t) + \gamma\Psi(s_{t+1}; g) - \Psi(s_t; g)$. Then the advantage is computed: $[\phi(s_t) + \gamma\Psi(s_{t+1}; g) - \Psi(s_t; g)]^T w(g)$. Then gradient descent for the actor $J_\pi = \log \pi(s_t; g)A_t$.

3.4 Transfer in Deep RL Using Successor Features and Generalised Policy Improvement

- This is a very important paper to compare to LOVR and $V^{\mathbb{L}}$ state representation.
- Extends the prior work on SFs, and uses a state represntation of $\{Q^{\pi_i}\}_{i=1}^{B}$, for some *basis* set of tasks, B.
- The authors recognize that under some conditions, this method will completely fail.

Continuing with the notion that SFs decouple the dynamics of MDP M_i from its reward, the authors look to extend the results to a less restrictive class of environments. Previously, with $\phi(s,a,s') \in \mathbb{R}^d$ a feature vector (representation),

$$\mathcal{M}^{\phi}(\mathcal{S},\mathcal{A},p,\gamma) = \{\mathcal{M}(\mathcal{S},\mathcal{A},p,r,\gamma) | r(s,a) = \phi(s,a)^T w,\ w \in \mathbb{R}^d\}.$$

Now, the authors assume an arbitrary reward function $r(s,a,s')$, and examine transfer within

$$\mathcal{M}(\mathcal{S},\mathcal{A},p,\gamma) = \{\mathcal{M}(\mathcal{S},\mathcal{A},p,.,\gamma)\}.$$

Hence $\mathcal{M}^{\phi}$ is a subset of this new set of environments. The authors provide theoretical upperboaunds on their GPI transfer framework with respect to the optimal Q-value function in this new class of environments, but again, they are essentially trivial.

Again, ϕ is wanted such that it *spans* the set of all tasks. However, given the new class of MDPs, this may not be feasible. Suppose, though, that the agent has been trained on D prior tasks,

$$\hat{\mathcal{M}} = \{M_1, ..., M_D\} \subset \mathcal{M}^{\phi},$$

and learned approximations

$$\tilde{\phi}(s,a,s')^T \tilde{w}_i \approx r_i(s,a,s'),\ \text{for } i = 1,2,...,D,$$

then the authors argue that the state representation $\tilde{\phi}$ can simply be the learned rewards. This is done since if $\mathbf{r}(s,a,s') = \mathbf{W}\phi(s,a,s')$, $\mathbf{W} \in \mathbb{R}^{Dxd}$ are the D vectors w_i stacked. If the we have D linearly independent tasks, then $\phi(s,a,s') = (\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{r}(s,a,s') = \mathbf{W}^{\dagger}\mathbf{r}(s,a,s')$. Hence, ϕ and $\mathbf{r}$ are related by linear maps, and one can be constructed from the other. Hence, the authors state that the rewards themselves $\mathbf{r}$ can be used as features. The authors state this is intuitively understand when we think of the base tasks as a set of basis for all tasks. However, I note that one must also ensure that $\mathbf{r}$ is injective.

The authors then note that the resulting SFs, $\Psi(s,a)$ are simply the Q-value functions. That is, the j^{th} component of $\tilde{\Psi}^{\pi_i}(s,a)$ is simply $\tilde{Q}_j^{\pi_i}(s,a)$. The authors propose two algorithms, and discuss issues surrounding this approach. The first algorithm assumes that a set of basis tasks have been solved, with $\tilde{\phi}$ and $\{\tilde{\Psi}_i\}_{i=1}^D$ already learned. The algorithm allows the user to either simply use GPI on the new task, or to also extend the basis by learning $\tilde{\Psi}_{n+1}$ for this $n+1$ task. The authors note that there is some difficulty in how to start this approach. If one uses Algorithm 1 from scratch (including to build the basis tasks), then one must learn everything in parallel, on-line, and may be unstable, non-stationary and problematic. Another possibility is to first learn the basis tasks using some base learning algorithm (Impala), then once $\tilde{phi}, \tilde{\Psi}_{i\in[D]}$ are learned, use this algorithm. Similar to $Q^{\mathbb{L}}$, the authors define:

$$\tilde{\Psi}^{\pi_i} = \tilde{\mathbf{Q}}^{\pi_i} = [\tilde{Q}_1^{\pi_i}, ..., \tilde{Q}_D^{\pi_i}],$$

which is to say, it is an element in $\mathbb{R}^{\mathcal{SA}\times\mathcal{D}}$, which measures the Q-values of (s,a) with respect to the i^{th} policy when evaluated in each of the $1, 2, ..., D$ base environments, whereas $Q^{\mathbb{L}} = [Q^{\ell_1}(s,a), ..., Q^{\ell_{\mathbb{L}}}(s,a)]$. Hence, the two representations are slightly different and carry with them different semantics.

Algorithm 2 defines how the SFs can be built for the basis set of tasks. The agents are in a DeepMind lab environment (room with objects to collect), and the architecture uses a CNN + LSTM whose output is then mapped to $D+1$ heads, to predict $\tilde{Q}^{\pi_i}$, $i \in [D]$, and $\tilde{r}$. This prediction uses $\tilde{w}$. The authors show pretty good transfer results, it is a good setup. They do note the following limitations and defects:

"Although we want to be able to represent the reward functions of all tasks of interest, this does not guarantee that the resulting GPI policy will perform well. To see this, suppose we replace the positive rewards in the basis tasks with negative ones. Clearly, in this case wewould still have a basis spanning the same space of rewards; however, since now a policy that stands still is optimal in all tasks, we should not expect GPI to give rise to good policies in tasks with positive rewards. One can ask how to define a "behavioural basis" that leads to good policies across the set of environments through GPI. We leave this as an interesting open question."

4 Goal Based and Multi-Task Studies

4.1 Unsupervised control through non-parametric discriminative rewards

This paper looks at a method of learning how to learn to go to goals, where the reward is learned and meant to replace $r(s) = \mathbb{1}[s = s_g]$. Since in continuous domains the likelihood of ever reaching the same state twice is vanishingly small,

we want *some* reward function, most likely to be learned, that can measure "Is the current state the goal state?".

Much like LOVR, this study is motivated that first we should learn to control the environment, then, given a goal task, we can use that to solve the task. The authors introduce DISCERN: Discriminative embedding reward networks. They learn a *goal achievement reward function* $r(s; s_g)$, by measuring the mutual information between the current state s and the goal state s_g. The framework assumes that a goal is sampled from some distribution, and the agent can act for T time steps in the environment, and at s_T we want to assess if it is the goal state or not. Hence, $\forall t < T,\ r(s_t) := 0$, then $r(s_T; s_g) \in [0, 1]$. The authors note that the goal achievement reward function can maximize the mutual information between the goal and state s_T:

$$I(s_g, s_T) = H(s_g) + \mathbb{E}[\log\, p(s_g|s_T)]$$

The authors lower bound this instead by replacing $p(s_g|s_T)$ by a variational distribution $q_\phi(s_g|s_T)$. Since the entropy term $H(s_g)$ plays no role in optimizing any policy π_θ it isalso removed. Hence the objective is to maximize $\mathbb{E}[\log\, q_\phi(s_g|s_T)]$. They view this as a cooperative game, where the variational distribution q_ϕ is trying to give good rewards, and the policy is trying to learn what the intended goal was, given the last state. Goals are sampled from previously reached states from some state buffer of fixed size. As newer states are reached, older states in this buffer are replaced.

The architecture is a CNN (RGB image). Then this final layer is taken and passed into an embedding layer. Let $h(s)$ denote the representation output from the CNN. Then $\ell_g = e_\phi(h(s_T))^T e_\phi(h(g))$, and the log likelihood is maximized by sampling K "decoy" goals, and computing:

$$\log\, \hat{q}(s_g = g|s_T, \pi_\theta) = \log\, \frac{exp(\beta \ell_g)}{exp(\beta \ell_g) + \sum_{k=1}^{K} exp(\beta\ ell_{d_k})}.$$

They fix $\beta = K + 1$. Essentially, they are trying to learn a similarity embedding (like word embeddings). Moreover, rather than using $\log\, q_\phi$ as the reward, they simply use $\max(0, \ell_g)$ as reward (the positive portion of cosine similarity). Moreover, they also select any state within H time steps of T as having reached the reward s_g.

They don't measure performance, per se, rather the proportion of "goals achieved" which they measure if the avatar in the game (Montezuma's Revenge and Seaquest) is within 10 percent of the play area as that prescribed by the goal sampled. This has given me good ideas about using word embedding style training for the "out ball" for landmark balls.

4.2 The Intentional Unintentional Agent: Learning to solve many continuous control tasks simultaneously

The authors us DDPG in a multi-task setting (mujoco play area). They use this actor-critic setup with 18 different heads to their architectures, for 18 different tasks (sometimes up to 43 tasks). While training, the agent follows only a single policy (task) on-policy: the hardest task, and from the experience replay buffer samples transitions and uses those transitions to update the critic for all the tasks. This is possible as they define their objective function $J(\theta) = \mathbb{E}[\sum_i Q^i_\pi(s, \pi^i_\theta(s))]$, for all the i tasks. The DPG policy theorem is applied here, except of course, it is summed over all of the tasks (heads of the neural net). The following updates are performed:

$$\delta^i_j = r^i_j + \gamma Q^i_{w'}(s_{j+1}, \pi^i_{\theta'}(s_{j+1})) - Q^i_w(s_j, a_j)$$
$$w = w + \alpha_{critic} \sum_j \sum_i \delta^i_j \nabla_w Q^i_w(s_j, a_j)$$
$$\theta = \alpha_{actor} \sum_j \sum_i \nabla_\theta \pi^i_\theta(s_j) \nabla_{a^i} Q^i_w(s_j, a^i)|_{a^i = \pi^i_\theta(s_j)}$$

Where j represents the set of indices for the mini-batch. Note that for the updates of w, the same action a_j is used to update the Q values for all the critics Q^i. Target networks are used for actors and critics.

4.3 Learning by Playing - Soving Sparse Reward Tasks from Scratch (Scheduled Auxiliary Control)

Like landmarks, at each transition, each state is associated to a list of rewards from a list of auxiliary tasks. Each reward is associated to a distinct policy, and a higher level controllers decides when to schedule the tasks/policies. All learning is off-policy and asynchronously. This is very similar to LOVR without reflection, and covers how the first phase should be done.

For each task, a reward is determined by some metric (e.g. positive reward if $d(s, s_g) < \epsilon$). The auxiliary tasks are trained while the main task is also trying to be solved. The entire loss function is thus the sum of loss functions over all the auxiliary tasks and the main task:

$$\mathcal{L}(\theta) = \mathcal{L}(\theta; \mathcal{M}) + \sum_{i=1}^{|\mathcal{T}|} \mathcal{L}(\theta; \mathcal{T}_i)$$

Where $\mathcal{T}$ is a task, and its loss is nothing more than the loss of a Q-network. The architectures for robotics control have the following structure: 200 ELU units with LayerNorm for activations, followed by 200 ELU units. This is shared for all tasks. Then, each of the tasks (n of them) has their own channel stream, which consists of 100 ELU then 18 ELU followed by a final tanh activation.

The output of this network is a policy. For the critic architecture, the action (one hot encoded) is concatenated to the observation. The shared layers are as similar to the actor network, except 400 units. The n auxiliary streams are 200 then 1 ELU unit, and outputs $Q(s, a)$. I belive for both actor and critic, one of those n streams is the "main task".

THIS IS SOMETHING I SHOULD COMPARE LOVR TO

4.4 Multi-task Deep RL with PopArt

The authors use a single architecture to learn acorss *ALL* Atari games, as well across ALL DeepMind Suite robotics tasks. The issues highlighted are how to schedule training of tasks (shared and limited resources, e.g. parameters), and how to scale rewards across different tasks. The authors use the IMPALA algorithm, where actors each interact with their environment for 100 steps and send all transitions to a global queue. An LSTM is used.V-Trace is used. To scale rewards, some exponential weighted averages and scaling technique is used. Their results are strong, with PopArt-IMPALA outperforming a multi-headed IMPALA, and regular IMPALA (one network trained on everything).

5 Model based Deep RL

5.1 Learning model-based planning from scratch (IBP: Imagination Based Planner)

The authors propose an "imagination-based planner" that ca learn to construct, evaluate and execute plans. Before any action, it can perform a variable number of imagination steps, which involve proposing an imagined action and evaluating it with its model-based imagination. All imagined actions and outcomes are aggregated, iteratively, into a "plan context" which conditions future real and imagined actions. Learning how much to imagine is also possible, by jointly optimizing rewards and computational costs associated to with using its imagination.

There is a high level *manager* that decides at each time step to either imagine, or to act. The higher level manager uses an LSTM, and so is acting on the entire history, which may include imagined states/transitions or real ones. In either case, the agent will either act in the real world, or perform a an imagination act (simulation). Note: the agent can imagine an action from any previously taken state up until now, from real states, or previously imagined states. The higher level manager is optimized using the external task reward and a cost associated to the time used for imagination. The imagination module is trained to predict the next state (and reward) in a supervised fashion. The authors used REINFORCE with entropy regularization, and looked at 1-step, n-step and "tree"-based imagination.

5.2 Imagination-Augmented Agents for Deep RL (I2A)

This was a good paper, and simplified and improved upon the IBP work. The authors note that when an imperfect model is used, the errors are compounded when planning, and can lead to bad results. They propose that the model based agent not only learn a good model, but also how/when to use or *not* use it. This is done y encoding the imagined future through a rollout encoder (context vector), which is used by an aggregator in comjunction with a model-free architecture path, and the two are concatenated to train an actor-critic. Hence, the aggregator can learn when to trust or disregard the rollout encodings, and not use the model.

The architecture has two separate streams that are ultimately concatenated to output actor and critic heads. The model-free path is a regular model-free architecture. The model-based path takes the current action and selects an action based on a rollout policy network, which is a small neural net trained using policy distillation on the true agent policy (combined streams). This action is represented as a one hot vector, then is broadcasted and concatenated to the size of the observation (image of size (h,w)), concatenated to the original image and fed into the environment model to predict the next state observation and reward. This occurs for a fixed number of steps, and in total constitutes a rollout. n-rollouts per original real observation are performed. The entire rollout is a sequence of features that is fed into a rollout embedding, mapping it to an embedding. Then, the n embeddings resulting from n rollouts is fed into an aggregator which maps this to a single context. This context is concatenated to the other network stream (model-free) and used to produce the actor and critic heads.

The encoder is an LSTM with convolutional encoder feeding the predictions in in reverse order. For experiments (Sokoban), the environment model was pre-trained and then kept frozen for experiments. They found that 1-step rollout improved over no model, 3-steps improved over 1-step, 5-steps improved over 3 steps, and 15-steps improved over 5. To show that I2A was reslient to imperfect models, they used a separate pre-trained model that was smaller with poorer predictions than the prior model, and showed that it performed just as well as the "good" model. They compare this to using a Monte Carlo predctions without the rollout encoder and show I2A drastically outperforms MC predictions. The reasoning is that the rollout encoder is important for summarizing useful information from the model, and for discarding the information when it is not helpful. Great paper.

5.3 Mastering the game of Go with deep neural networks and tree search

This is to outline the framework and rough ideas around this approach. Note they trained with almost 200 GPU's and over a 1000 CPU's.

- Beginning with a pool of data from the KGS Go server, they train a 13-layer policy network π_σ, using SL
- They also trained a smaller rollout policy network (for faster predictions) called π_π
- They trained another large policy network from RL, π_ρ. It has the same architecture as the large SL policy network and is initialized with the same weights. It plays games with itself (self play) and with other prior policy networks (sampled randomly) and trained using REINFORCE policy gradients.
- A value network v^ρ associated to the RL policy is built. This is trained using regression against values in $\{-1, 1\}$ to denote a loss or win. The dataset used to train this was the KGS Go server, as well as 30million states, each from a unique game played by the RL policy against itself.
- For the MCTS: each node (state) stores $Q(s, a) =$ the average value of n rollouts from s where action a was taken from s, and the average is with respect to the value of the arrived at leaf node.
- A leaf node is evaluated first by the value network v^ρ, and then a random rollout from the leaf state is performed and the outcome of that game is used, resulting in $V(s_L) = (1 - \lambda)v^\rho(s_L) + \lambda z_L$, where $z_L \in \{-1, 1\}$ is the result of the game rollout.

There are a couple of other details with respect to how the UCB approach of MCTS is done, however this almost the whole story.

6 Human in the Loop RL

6.1 Deep RL from Human Preferences

The authors train agents using less than 1 percent of data. The goal is to learn a reward function from humans, then learn a policy that optimizes on that reward function. They state that between 15min and 5hours of labelled videos is all that is required for the agents. Since the reward function is learned as the agent interacts with the environment (through SL), we can view the reward as non-stationary, and hence they use policy gradient methods (A2C for Atari and TRPO for robotics). The predicted reward function was normalized to have mean zero and a "constant" standard deviation.

Humans are shown two clips, each 1-2 seconds long, and asked which do they prefer. They are given a context (what should be happening). Given two trajectories, either all the mass is put onto one (if prefered), equally shared

(no preference) or the data is discarded (don't know). The reward function is trained using supervised learning, using the following loss:

$$loss(\hat{r}) = -\sum_{\tau_1,\tau_2 \in \mathcal{D}} \mu(1)\log\, P(\tau_1 > \tau_2) + \mu(2)\log\, P(\tau_2 > \tau_1)$$

The authors are not explicit about how the reward is learned, and say they use modifications of this approach but don't explicitly state. They train an ensemble of predictors of r For experiments, as control they use an oracle, which gives the "preference" as given by the true reward for the two trajectories. They also compare to agent with actual true reward. They experimented with increasing number of human queries. The ensemble was used in a sort of active learning approach, to decide which trajectories to query by selecting those with the most disagreement (Variance).

Ablation studies used random sampling of trajectories; no ensemble of predictors;

6.2 Deep COACH: Deep RL from policy-dependent human feedback

This was a well done and **very** crafty paper. The authors build on COACH, for deeper neural nets, which is an actor-critic approach to RL where humans provide the rewards. They use a very small (10x10) Minecraft world where the goal is a golden block in the center of the room. Human rewards are given as $f_t(s_t) \in \{-1, 0, 1\}$. The main idea of their work is that human feedback is an unbiased estimator of the advantage function. Very interesting. And so, the policy gradient can be written as $\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s) f_t]$. Hence, updates only need to take place when the human provides negative or positive feedback (e.g. $|f_t| = 1$).

Their setup involves "real-time" human rewards. Since the game engine can process many steps per second, which doesn't give the human enough reactive time to view the sequence, judge, and provide feedback, they slow down the game engine from 50ms to 250ms, and use a delay window size L, where upon human (non-zero) feedback, the L most recent transitions are stored in an experience replay buffer as a single entry (e.g. the experience replay buffer atoms are length L trajectory windows), and this L-window of experiences is associated to a single f_t human reward. Eligibility traces with importance sampling are applied to each of these length L-window trajectories.

The true genius of their experiments is the following: Due to the simplistic nature of the environment, they use a quite small NN. They pre-train an embedding layer (first couple of layers of their policy network) using a CNN autoencoder (reconstruction loss). This is then fixed for all subsequent learning

(RL), and the output of which leads to 2 hidden layers with "at most" 30 units each. 3 actions (turn left, turn right, move forward) are used. Moreover, they use entropy regularization which was optimized to make the probability of each action as close to uniform as possible. This is crucial. The policy doesn't sample from the action probabilities (as all actor-critics do), rather, it takes the arg max. The reason this is genius is that, they use an incredibly small network, hence is much more susceptible to immediate changes from a loss function (gradients). By forcing the actions to be almost uniform, once the human gives a preference (positive or negative), it will immediately cause a single action to be preferred or deterred, and due to their arg max action selection, can quickly learn what the right action is. Hence, the human can just watch the agent, make a few suggestions/judgements online, and the agent can quickly adapt and solve the very simple problem.

This works really well for the domain used here, but will be quite difficult to scale to larger networks, as this setup is quite hand crafted and specific. Great nonetheless!

6.3 TAMER: Interactively shaping agents via human reinforcement

This paper uses a human feedback in the loop. The agent tries to learn the human reward function. Moreover, since it happens online, to perform credit assignment of which (s, a) is to receive the human reward, the authors cook up a credit assignment heuristic which is an integral over some prior interval of time that preceeds the human reward. They do this rather than eligibility traces. Essentially, it assigns this human reward over an interval of (s_t, a_t) for some time t. The results are decent.

6.4 Learning Non-myopically from human genereated reward (VI-TAMER)

This paper deals with episodic RL problems with human provided rewards, but notice that in their experiments that if a human provides a few positive rewards which can induce a closed loop, the agent will never reach the goal state and do the task set for itself. They notice that the discount factor also plays a contingent role in this (larger discound leads to more perverse behaviour). The authors try and solve this by turning an episodic MDP into a continuing MDP by transitioning from the goal state back to the start state. They look at the role of gamma in this. Nothing special, but worth noting that *for goal based MDP's, if humans provide positive rewards, the agent may never reach the goal state. Hence, this motivates the use for action penalty reward and instead learning what the goal state is (supervised learning).*

6.5 Deep-TAMER

This paper just does a deep CNN version of TAMER to play Atari bowling.