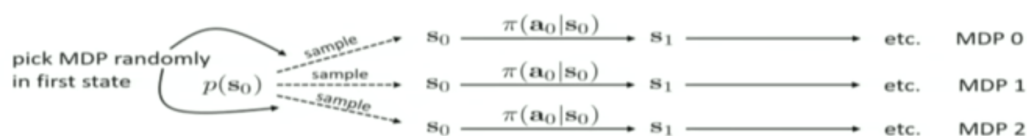


transfer learning & meta-learning

- the big caveat in all the below discussion is that these are recent research ideas and not proven out methods. its more like a recent literature survey.
- what are the things we can have shared representations for that can be useful across tasks?
 - q-function estimating neural nets that share parameters
 - policy function neural nets
 - dynamic model neural nets
 - features/hidden states: provides us with a good representation for states, like mapping images to hidden states that are more suitable to train on/more suitable for decision making.
- we can get benefits of:
 - faster learning → humans get this all the time as they perform a range of tasks in their day to day lives and encounter new tasks
 - better final performance when started with a good learned representation from a related domain
- source domain is the domain we will train on and target domain is the domain that we want to do well on.
- terminology:
 - zero-shot learning: model trained on source just works when applied to target
 - one-shot learning: model trained on source needs to try the task once, adjust some weights etc and then its ready to go
 - few shot learning: model trained on source needs to try target task a few times.
- strategies for fine tuned transfer learning:
 - try and hope for the best: we just learn a model/policy on a source domain and hope it works on target domain. its meh, but a good start.
 - fine-tuning: train a bunch on source and retrain a bit on target.
 - in supervised learning this is the equivalent of training a deep CNN on ImageNet and then throwing away the last few fully connected layers, adding new ones and training on the target domain.
 - the challenges are that:
 - features are less general/transferable in RL than in supervised learning.
 - optimal policies in deterministic MDPs are deterministic, they don't do much exploring in the new domain → loss of exploration. these low entropy policies learn very slowly.
 - one way to get around these challenges is to reward entropy of policy while training so that it learns to solve source task in different ways. when applied to target domain this tends to learn faster and explore more.
 - the thing is fine tuning tends to work really well with really deep networks trained on a ton of initial data. problem is when we move to the target domain we can't retrain from scratch because we usually have lesser data in the target domain. one strategy to overcome this is to **train a really deep net** on huge amounts of source domain data and then **freeze it, add a small number of layers** and train just those layers on target domain. this approach is called **progressive neural networks**.
 - pro tip: when someone says i'm using fine tuning/transfer learning and it does better, a good question to ask is how do you know its not just doing better because you added a few extra layers

while freezing the rest of the pretrained network

- the trick to making transfer learning work well is to **get a diversity of pre-trained experience**. that is why large datasets with a ton of diversity helps. one way to achieve this in RL is to **introduce diversity/randomization into the source domain**.
- another idea is to train the policy with a parameter sweep version of environments/simulators, like varying the mass of the robot arm, along with explicit system identification → in the form of a parameter (eg arm mass) thats fed in as input. then, on the target domain, we can use another network that is trained to identify this parameter and feed that in as input to the policy network.
- another idea is to train a GAN to take in an image produced by a simulator, generate a more realistic version of that image and train the policy on that realistic version, feeding its action back into the simulator. then the images it sees in the target domain are hopefully not too different from the simulated images it has seen during training. this is an example of a **domain adaptation** technique in deep RL, one in which we try to prepare better for our target domain.
- multi task learning:
 - this can involve either training on multiple tasks just to improve diversity and quantity of training data to perform better across all of those tasks OR it could involve training on a set of source tasks before transferring to a target task(s).
 - as a motivating factor, this is what people do when for examples we create specific practice exercises for sports players!
 - but of course, transfer is not always obvious, some experience might be useful in target task, some not.
 - one idea is to learn specific aspects of a RL algorithm from another domain. for example, learning the dynamics model from another domain where we believe the dynamics is similar and we have a lot of data in the source domain and not so much in the target domain.
 - can we solve multiple tasks at once?
 - one idea is to construct a joint MDP of all the tasks we are interested in. sampling from this would look something like sampling one of the tasks (uniformly, for example) and then sampling a set of state, action pairs from the thus chosen task. then this problem turns into a single RL problem that we can solve with any of our known RL algorithms.



sometimes the above can just make the task much harder and we'll do worse in all the tasks.

- to address the above, we can use something called policy distillation.
 - first, some background on **model distillation**. ensembles usually do better than a single model, but of course they are computationally expensive. can we somehow capture the knowledge in ensembles and have a single model do almost as well as an ensemble? Hinton wrote a paper called Distilling the Knowledge in a Neural Net where he shows that if you train an ensemble and use the predictions of the ensemble as the training labels for a single model, we can get a model that's better than just training one model. this is because we are forcing it to fit to more knowledge (the soft targets) than just the hard labels.
 - the RL version of the above is to train separate policies on different tasks/games and then train a single policy based on the above with soft labels from them. this is called **policy**

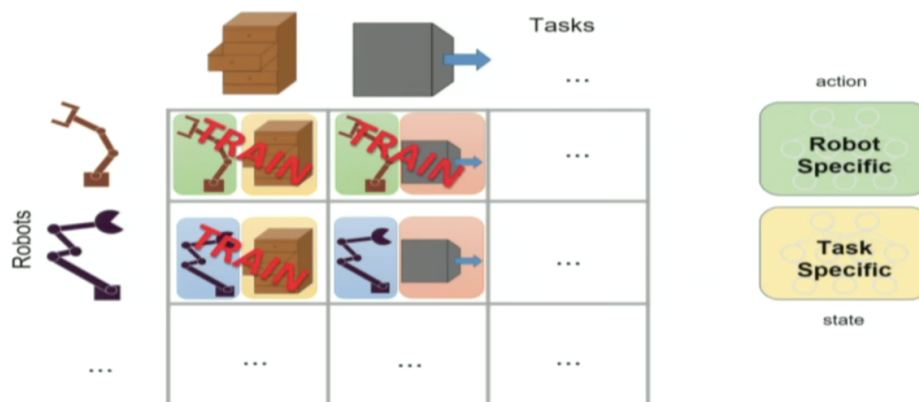
distillation.

- this happens to work to different extents on different games/tasks. on some tasks, this joint learning helps do much better than training on just that task. on other tasks, we do worse. when learning on another domain helps, it is called “positive transfer”. when it hurts, its called “negative transfer”.
- now, in these games, once we train a master policy how does it know what to do? implicitly, it has learnt what to do based on the observed state → since the image is input, it implicitly figures out what game it is and what action should be taken based on that image.
- now, what if we want to do multi task learning like above where the states don't clearly distinguish the tasks → humans do this all the time for example, unloading clean dishes from a dishwasher vs loading clean dishes into it. a lot of images in these two tasks are the same but there is some implicit context on what the aim is. or with autonomous driving, a lot of maneuvers can look similar but overall driving is driven by destination.

- What if the policy can do *multiple* things in the *same* environment?



- enter **contextual policies**: standard policy is $\pi_{\theta}(a|s)$. contextual policy is $\pi_{\theta}(a|s, \omega)$ where ω is the state but we won't treat this any differently than state. new state space is (s, ω) . That's it! train using these new state spaces and it will often work pretty well. very simple idea, but tends to work pretty great.
- so far we have looked at multi task learning as training a single neural net/model by either framing a joint MDP or using the policy distillation trick. can we do a more modular approach to parameter/knowledge sharing? we can! by using modular networks, we can share knowledge across tasks and robots with slightly different dynamics. a high level idea is below:



Note we have two different robots and two different tasks and we can make them share representations. Ideally we'd like these shared networks to learn shared knowledge across tasks. if we train on a small number of tasks then it can turn out that half the model is used for one task and other for another task which is not what we want. in order to get true shared knowledge, we need to train on a large number of tasks (like we do large number of label categories in ImageNet). how to do this is an active area of research.

- So based on the above image and description, in summary, **more tasks = more diversity = better transfer.**
- now we will shift to talking about meta learning.
- meta learning is concerned with: **if we have learned to perform 100 tasks, can we learn the 101st a lot more efficiently?** note that this isn't just saying can we perform better on 101st task using the previous tasks' knowledge. it is saying can we learn the 101st one more efficiently.
- Meta learning can be:
 - learning an optimizer
 - learning an RNN that "ingests" experience
 - learning a good representation that makes future tasks easier
- what can a meta-learned learner do:
 - explore more intelligently/avoid trying new actions that are known to be useless
 - acquire the right features more quickly
- supervised learning formulation of meta-learning is to view a training set as a set of datasets with train/test breakdown. we train on the overall training set and test the meta learner on the overall test set as below:

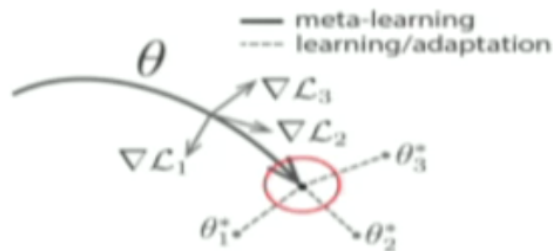
Meta-learning with supervised learning



supervised learning: $f(x) \rightarrow y$
 input (e.g., image) output (e.g., label)

supervised meta-learning: $f(\mathcal{D}_{\text{train}}, x) \rightarrow y$
 training set

- so in the above supervised learning approach formulation of a meta learner, instead of being $f(x) \rightarrow y$ as in the usual supervised learning context, we are trying to get good at $f(\mathcal{D}_{train}, x) \rightarrow y$. Similarly, in RL, it is $f(\mathcal{D}_{train}, s) \rightarrow a$ where \mathcal{D}_{train} is $(s_1, a_1, r_1), (s_2, a_2, r_2) \dots (s_n, a_n, r_n)$
- A common research approach that is being tried for the above tasks is to use RNNs for meta learning. Here, we have trained the RNN on some meta-training set (see image above to see what this means). Then, we take that RNN and run thru an example in a meta-testing set where we run thru some sample (state, action, reward) tuples and then test on the test set.
- another idea is to setup the gradient update function in such a way as to reward getting to a point where from its quite easy to get to a set of different points that each correspond to doing well on a particular task.

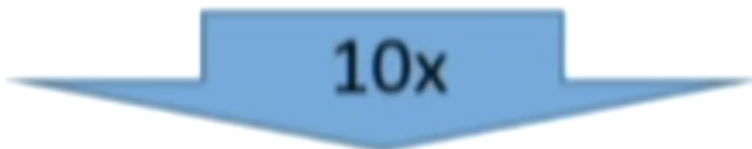


- put another way, take a gradient step so that when you take a couple additional gradient steps on the target domain (task i), you will do well to that task i , for any i that we are interested in. this involves computing the gradient of [reward + eventual gradient step] which means its a second order gradient.
- note that after the meta-training phase, the points don't necessarily do well on any of the tasks (because they are just at a point where they can get close to good points but are not actually at good points). you need the training on the meta-testing set to get close to good points for the given task. this is called **model-agnostic meta learning**.
- One of the big challenges with RNN policies is that they are extremely hard to train, and likely not scalable. Like you can't hope to train an RNN on 100,000 data points and just use back prop thru time to get gradients from all the way back to the front.
- The downsides of model agnostic learning described here is that it is difficult to compute second order gradients.

A few concluding notes:

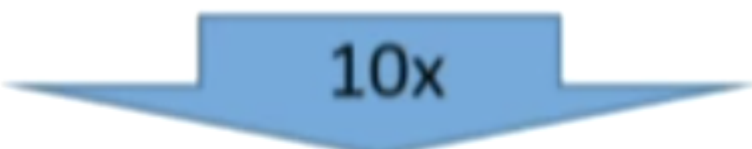
- Quote/true say:
 - "Good, stable RL algorithms are not great for beating benchmarks because you probably won't beat 'em initially but is absolutely critical for making RL algorithms widely adopted and popular for real world problems!" - Sergey
- Sample efficiency comparison. Note that this does NOT necessarily correspond to wall clock time comparisons.

gradient-free methods
(e.g. NES, CMA, etc.)



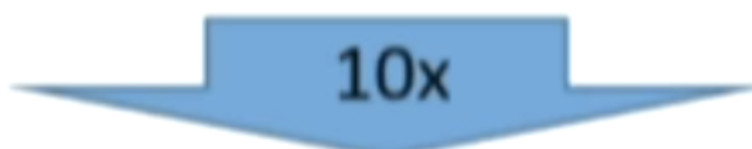
10x

fully online methods
(e.g. A3C)



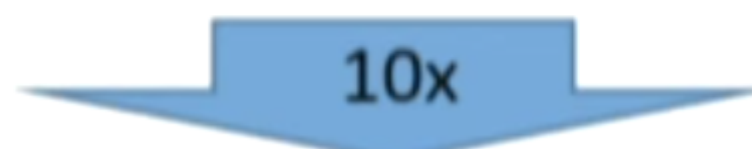
10x

policy gradient method
(e.g. TRPO)



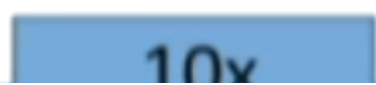
10x

replay buffer value estimation
(Q-learning, DDPG, NAF,




10x

model-based deep RL
(e.g. guided policy search)



10x



model-based “shallow” (e.g. PILCO)

- speculation: in supervised learning, we get quite a few bits of sample at each learning step. in RL, we just get a reward, which if the state isn't greatly informative, we can't learn a lot from (classic credit assignment problem, again). can we vastly increase this amount of feedback and guide the parameters to good places?
- one answer to the above might be density estimation models/unsupervised learning methods with dense representation of information? this is kinda also hinted at by the fact that humans already know so much before they learn a new task and are hence able to learn it well.
- Sergey's 3 promising directions:
 - unsupervised learning/density estimation/generative models/yann lecunn's cake
 - imitation learning → these tasks are really quite complex and humans are just good because many humans have figured it out and we are good imitation creatures.
 - maybe everything is good, yes credit assignment problem but we just need to keep doing backups and figure out great value functions → least likely in my opinion, but of course I have no way to back this up.
- “Pick problems to work on that have elements of these and ingredients so that if you solve them you can build useful systems!”. don't pick games where getting good at it might mean nothing!