CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 17

1 Reading

End-to-End Learning for Self-Driving Cars

In this paper, the authors train a CNN to map pixels to steering commands (as turning radii). This is done in a supervised fashion, such that training data consists of (image, steering angle) pairs. Since the system must be able to recover from mistakes, they also apply random shifts and rotations to some images and pair them with commands that would take the vehicle back to the desired pose in a small period of time ($\sim$ 2s).

*Note: these transformed images are in **addition** to all of the original training data.*

The network is trained to minimize the MSE between the predicted steering command and the expert steering command (or the "recovery" steering command if the image is part of the augmentation set).

DAgger

One problem with sequential prediction problems, where future observations depend on previous predictions (i.e. actions), is that they violate the i.i.d. assumptions which are often made in statistical learning. Most of the training data is heavily dependent on itself, and only involves observations that an expert would see. *If we make even small deviations from an expert's actions, we can quickly end up in a space of observations that is completely uncovered by the training data, and our error will compound!*

In order for methods like imitation learning to work well, the distribution of observations seen during training must be as close as possible to the distribution of observations seen in practice. [2] introduces an iterative method for making this so. Their method, **DAgger** ("dataset aggregation"), repeatedly

- samples trajectories according to the current policy (this can be the expert policy on the first iteration), *adding observed states and their corresponding expert actions to the dataset*
- trains the policy on the total dataset

in order to produce a deterministic policy that is able to achieve good performance guarantees under its induced distribution of states.

Learning Transferable Policies

In [3], the authors propose a generic framework for training motion policies using data from a different (but related) domain. *Motivation*: for many robot tasks, it is difficult, impractical, or dangerous to obtain real training data.

More specifically, they provide the ability to train a domain-adaptive policy using labeled information from the source domain and unlabeled information from the target domain. The goal is to learn a set of features x which reduce the cross-domain discrepancy, alongside a policy $y = \pi_\theta(x)$. Qualitatively, their "transferable policy" enjoys significantly boosted performance in the target domain when compared to a source domain policy that is simply plopped into the target domain.

Domain transferability can be enhanced by minimizing the maximum mean discrepancy (MK-MMD) between representations of the source domain $\mathcal{D}_s$ and target domain $\mathcal{D}_t$ in the task-specific layers of the neural network.

From Virtual Demonstration to Real-World Manipulation

This paper describes an approach for transferring behavior from virtual demonstrations to a physical robot. The virtual demonstrations are used to train an LSTM controller for generating trajectories. The training process also uses an MDN to calculate error signals appropriate for the multimodal distribution of demonstrations. (There are often multiple ways to perform a task: sometimes we might demonstrate this way, sometimes we might demonstrate this way... yet both ways will solve the task in an efficient manner. Additionally, different people might have different demonstration tendencies.)

The authors claim (1) that the controller learned from virtual demonstrations successfully transfers to a robot in the real world, and (2) that their LSTM with an MDN-based error signal outperforms architectures such as feedforward networks with MSE-based error signals.

2 Lecture

Recap

The GAN framework tries to generate high-quality data that is indistinguishable from real data. The discriminator in the GAN framework is just a classifier, so it can be trained with L2 or cross-entropy loss.

Typically, the discriminator and generator are trained in an alternating fashion. On the discriminator's turn, inputs are fed in from either the generator or the real distribution. On the generator's turn, its output is fed through the discriminator, and the gradients propagate all the way back from the discriminator's output to the generator (but the discriminator is not updated during this time).

As it happens, this alternating optimization scheme minimizes the Jensen-Shannon divergence between the artificial data distribution and the real data distribution.

Deep Control

Deep networks are good at fitting functions to labeled input and output data, so one thing we can try is **behavioral cloning**, i.e. imitating expert control behavior. Note: in order to do so, we'll need to copy both the input (the image or whatever the human is using) and the output. This is because there's a distinction between imitating *control* and imitating *action*, and we want a closed-loop system (with feedback and adaptation) rather than simple open-loop actions.

So we'll try to learn the mapping from sensor input to actuator output (*sensorimotor learning*). This will involve a sensorimotor loop: we'll repeatedly sense something, then take an action, which changes what we sense, and so on.

Terminology and Notation

- $\mathbf{x}_t$ - state at time t
- $\mathbf{o}_t$ - observation at time t
- $\mathbf{u}_t$ - action at time t
- $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$ - policy, which is a conditional distribution over the actions $\mathbf{u}_t$ given the observation $\mathbf{o}_t$

The next state $\mathbf{x}_{t+1}$ is conditioned on the current state $\mathbf{x}_t$ and action $\mathbf{u}_t$. The action $\mathbf{u}_t$ is conditioned by the policy π_θ on the observation $\mathbf{o}_t$. The observation $\mathbf{o}_t$ (which is all the policy can see) is conditioned on the state $\mathbf{x}_t$.

One of the biggest challenges with imitation learning is that it tends to accumulate error. If the controller diverges even a little from the known space of trajectories, it will continue to diverge even further until everything is hopeless. In fact errors in the learned model's trajectory can grow quadratically with time.

NVIDIA's approach in [1] to dealing with this error was to enrich the data with simulated inputs (i.e. images with a viewpoint change, to simulate drifting away from the human trajectory) and corresponding artificial corrections.

DAgger

More Terminology

The **behavior policy** $\pi_0(\mathbf{u}|\mathbf{o})$ is the policy that the agent uses to act in the world. The **target policy** $\pi^t_{\theta^t}(\mathbf{u}|\mathbf{o})$ is the policy the agent is trying to learn.

Also, there are both on-policy and off-policy methods. An **on-policy** method uses the agent's own experience, so the target and the behavior policy are the same (e.g. RL; there's only one actor and we have to re-run the policy after making an update). When the target and behavior policies are different, it's **off-policy**. This is more general. *Imitation learning is an off-policy method, since it uses another agent's experience.*

Error accumulates because the policy ends up in states that the expert was never in. As a result, there's no direct data that the agent can use to figure out what to do in such situations. *Generally, off-policy data is hard to learn from, because it doesn't tell us what we should do in the states that we **ourselves** are likely to get to.*

DAgger is a well-known method that addresses this problem in a simple and effective way. It generates training data in an on-policy fashion by running $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$ and then getting a human to label the observations that were seen during these rollout(s). Overall, the algorithm looks like this:

1. Train $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$ on human data $\mathcal{D} = \{\mathbf{o}_1, \mathbf{u}_1, ..., \mathbf{o}_N, \mathbf{u}_N\}$.
2. Run $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$ to get dataset $\mathcal{D}_\pi = \{\mathbf{o}_1, ..., \mathbf{o}_M\}$.
3. Ask human to label $\mathcal{D}_\pi$ with actions $\mathbf{u}_t$.
4. Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$.

Typically, the most difficult and expensive part is getting a human to label the observations.

Domain Adaptation

We can train a policy in simulation (or at least in a different environment) that's easier, and then explicitly transfer the policy to our target environment s.t. the behavior is the same. This is similar to deep neural style transfer, which we talked about previously in the class.

MAV application: imagine we had labeled data for the training domain but not for the target domain. We would want to train a policy (using DAgger, etc.) on the training domain and apply it to the target domain. To do so, we would use a network with a common convolutional frontend splitting off into separate fully connected layers.

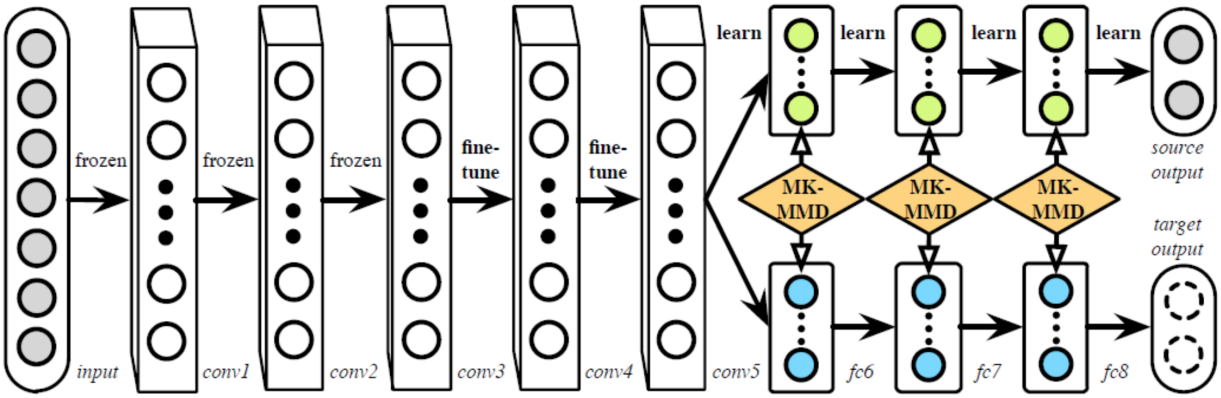
The entire network would be trained end-to-end to minimize this composite loss:

$$\min_\theta \frac{1}{n_s} \sum_{i=1}^{n_s} J(\theta(x_i^s), y_i^s) + \lambda \sum_{l=l_1}^{l_2} d_k^2(\mathcal{D}_s^l, \mathcal{D}_t^l)$$

$$\min_\theta \text{ (label loss)} + \text{(distribution loss)}$$

The first component quantifies the difference between the predicted value $\theta(x_i^s)$ and the label y. The second component is a statistical difference measure (a Jensen-Shannon distance) between the density maps of the two outputs.

We output a density map of trajectories generated by the human-trained controller, and a similar density map of trajectories generated by the unlabeled target domain images. Then we try to minimize the difference



source: slides from John Canny's CS 194-129 lecture / original paper [3]

between them as probabilistic densities. In a way, we're styling trajectories from novel inputs s.t. the overall distribution of resulting trajectories is similar to the distribution of trajectories from the labeled data.

Motivation for all of this: it's often easier to get human training data in an environment different from the target environment (e.g. in simulation).

Error Recovery: Learning ADL Tasks with an LSTM

Up to this point, our controllers have had no memory and have relied on the environment providing them with all of the information necessary for making a decision. For more complicated tasks, e.g. pick-and-place, it could be important to remember that we haven't picked the object up, that when it's in our grasp we shouldn't let go, etc. Essentially, for nontrivial tasks we might want a stateful controller.

[4] studies the control of stateful LSTM controllers for some "activities of daily living" (ADL) tasks. The idea is to collect trajectories in simulation and train the network on this data, then run the trained network on a physical robot. The authors use a three-level LSTM which outputs an action e at each time step, which is fed back in as the next input alongside an observation q.

Their output, instead of a simple softmax, is actually a mixture density (meaning it can have multiple modes). This allows it to respond and react to different executions of the same task; it can do things in different ways.

GAIL: Generative Adversarial Imitation Learning

GAIL goes further than other techniques, using for example inverse reinforcement learning (where instead of trying to blindly mimic the human, it tries to figure out what the user is trying to do; "what is the criteria for performing the task?"). In other words, it tries to learn the user's cost function by observation. Once it has a cost function, it can generate control using optimal trajectories or RL.

This is the state of the art for imitation learning.

Inverse reinforcement learning (IRL) is heavily under-constrained; the cost function, as a function of the states and actions along an entire trajectory, is potentially very high-dimensional. Also, just given a set of observations, it's possible to construct many cost functions such those trajectories would be optimal. So it's important to regularize.

One of the popular regularizers is entropy:

$$H(\pi) = \mathbb{E}_\pi[-\log \pi(\mathbf{u}|\mathbf{o})]$$

which involves taking the log probability of a particular trajectory across an entire set of actions. Again, entropy is a measure of randomness in a probability distribution. A uniformly random policy has maximum

entropy. By contrast, a deterministic policy has an entropy of 0. *In our context, entropy captures how much uncertainty or randomness there is in the policy.*

The "maximum entropy" version of imitation learning tries to find policies are as conservative as possible in their action choices.

$$\mathrm{RL}(c) = \arg\min_{\pi \in \Pi} \left(-H(\pi) + \mathbb{E}_{\pi}[c(\mathbf{u}, \mathbf{o})]\right)$$

If a particular action doesn't affect the outcome, the policy will uniformly sample that action. Essentially, it will explore as much as possible (i.e. as long as an action doesn't have an outcome we believe to be bad). Note: the second component just minimizes the cost of the trajectory.

The corresponding "maximum entropy" IRL formulation, which gives us our cost function c, is

$$\max_{c \in \mathcal{C}} \left(\min_{\pi \in \Pi} \left(-H(\pi) + \mathbb{E}_{\pi}[c(\mathbf{u}, \mathbf{o})]\right) - \mathbb{E}_{\pi_E}[c(\mathbf{u}, \mathbf{o})] \right)$$

where π_E is the expert trajectory and $\mathbb{E}_{\pi_E}[c(\mathbf{u}, \mathbf{o})]$ is the **expert trajectory cost**. $-H(\pi) + \mathbb{E}_{\pi}[c(\mathbf{u}, \mathbf{o})]$ is the **learned policy cost**; we're trying to choose a cost function which maximizes the difference between the learned policy cost and the expert policy cost. (A function is a good function if it helps differentiate the policy we're learning from the expert policy, because then we'll minimize it and it will push us closer to the expert policy.)

More concretely, GAIL generates trajectories according to its own policy. There is a density map ρ_π for these trajectories, along with a density map ρ_{π^E} for the expert trajectories. Instead of using the Jensen-Shannon loss between these two distributions, it trains a discriminator between the distribution of target (learned) trajectories and the distribution of expert trajectories. Then it does the usual adversarial backpropagation through the discriminator to the target policy, which makes the trajectories generated by GAIL look more like the expert trajectories in state occupancy.

Basically, it's a GAN framework, where the generator is our imitation learning policy and the discriminator is trying to discriminate the expert and learned policies by their state occupancy functions ρ_π and ρ_{π^E}. The goal is to make the traces of trajectories from the target policy look as similar as possible to the traces of trajectories from an expert.

Imitation Learning: Summary

Imitation learning (getting the most out of copying human behavior) is more complicated than just using labeled data for images or translation, since action with robots or vehicles isn't just a feedforward task. It's a feedforward task with a control loop, where we change the sensor data whenever we take an action.

Usually we can't just use a few demonstrations to train (due to the distribution mismatch problem). Domain adaptation or IRL can help with this. Tricks have also been used (e.g. adding on-policy data as per DAgger).

References

[1] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, Karol Zieba. End to End Learning for Self-Driving Cars. *arXiv preprint arXiv:1604.07316* (2016).

[2] Stephane Ross, Geoffrey J. Gordon, J. Andrew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. *Journal of Machine Learning Research*, 15:627?635, 2011.

[3] Shreyansh Daftry, J. Andrew Bagnell, Martial Hebert. Learning Transferable Policies for Monocular Reactive MAV Control. ISER 2016.

[4] Rouhollah Rahmatizadeh, Pooya Abolghasemi, Aman Behal, Ladislau Blni. From Virtual Demonstration to Real-World Manipulation Using LSTM and MDN. *arXiv preprint arXiv:1603.03833* (2016).