LECTURE #1

LECTURER: PERCY LIANG (STANFORD) SCRIBE: KIRAN VODRAHALLI

Contents

1 Motivation

There are lots of prediction tasks: what animal is this, when was the last time Greece held the Summer Olympics, what is the correct move to play in a game of Go?

Machine learning has been quite successful in easier tasks: I am more interested in tasks like question answering, game playing, and theorem proving.

For instance, consider a question-answering dataset: WikiTableQuestions (2108 tables, 22033 questions). More difficult than freebase, and the tables in the test data are not seen during training. Liang's group created this dataset by crowd-sourcing. We are pretty happy with this set as a challenge problem in NLP. Of course this number is very small compared to ImageNet (1.2 million). If you can actually do this well after training on a small amount of data.

As a baseline, you define some features and then try to rank the cells (information retrival) which only does 12.7%. Our method is better (37.1%) but it's not 99%.

2 Computational Process

So how do we solve this problem?

We will answer three questions:

1. How do we model this prediction process?
2. What is the right training signal?
3. How do we identify the right process?

3 How do we model the prediction process?

This is an area in NLP where we want to map natural language to logical forms. The main idea is factorization: First we map natural language into some sort of structure which corresponds to what the question is asking, and then we try to figure out the answer to the question. Given the question 'When was the last time Greece held the Summer Olympics?' and the input $R[\text{Date}], R[\text{Year}], \text{argmax}(Country, Greece, Index)$, we get the answer.

So suppose someone told you the correct logical form. How would you predict that from the question?

3.1 Semantic Parsing

Suppose you want to label parts of speech. You can have a graphical model: CRFs capture many different tasks, including tagging, chunking, segmentation, parsing.

As for how to learn: log likelihood of exponential family, and then use a gradient learning method. In order to learn, you have to solve an inference problem. The learning/inference interface: On one hand, learning says you will update using gradients, and on the other hand, inference is going to compute marginals or an argmax or whatever. Most models are not going to look like this though.

Most models we want to use in NLP have long range dependencies, and more structure. In that case, you can't really solve inference problem - instead you will solve approximate inference. But this is problematic, since if you use approximate inference and learning does not really learn about it, then you do not get anything. The issue is there are lots of rich features we want to put in our model, but they actually make life more difficult and make things worse. So this seems morally wrong.

Another way to think about what is going on: Our traditional view: the complexity of inference in models is inadequate. Tree-width to some extent captures this. Ideally, we want to think of models which give more bang for your computation buck. How you improve this accuracy tradeoff is pretty non-trivial.

There is also some empirical evidence that you can do as well as CRF with logistic regression. Constituency parsing: Dynamic programming gets the same as beam search. Similarly, dependency parsing: Using a neural net beats an MST parser. Back in 2005, structured prediction was the rage, and structure was important. But now all these results are coming out saying you don't need that kind of structure. A lot of fast methods can work well. That really made us think, perhaps we should re-think the way we approach model families.

The idea is that instead of trying to model only the output, we will try to model the steps of inference as latent variables (work with Jacob Steinhardt). We parametrize th emodel: $p_\theta(y) \sim \exp(\phi(y) \cdot \theta)$. If $\phi(y)$ is arbitrary, no control over computation. Instead we will parametrize the sampling algorithm. The idea is that each step $y_1 \to y_2 \to \cdots$ would be cheap to run.

The natural definition for complexity of a sampler is the mixing time of a sampler. It is hard to analyze mixing time though. Instead, we will create Markov chains which are fast mixing by construction. For instance, the sampler for an intractable model may be $B_\theta\left(y_1|y_{t-1}\right)$, and you also have a tractable model, $u_\theta(y_t)$. Then the idea is you take a mixture over the two: $A_\theta(y_t|y_{t-1}) = (1-\epsilon)B_\theta + \epsilon u_\theta$. Here y_{t-1} is any sequence - it is the entire state of the object you are trying to predict. Then this give syou fast mixing.

Theorem 3.1. *Mixing time.*
The spectral gap of $A_\theta = \mathcal{O}(\frac{1}{\epsilon})$

This is not necessarily close to the true stationary distribution; however, we have defined a framework whereby we can explicitly tradeoff computation and accuracy.

The objective in learning is basically with the objective $\max_\theta \log A_\theta^\infty(y)$. We are maximizing log-likelihood under the stationary distribution. Under this construction, you can draw an exact sample under the chain. So you are basically sampling from u, and running the Markov chain on average $\frac{1}{\epsilon}$ steps, which is the mixing time. Then you can use this unbiased gradient to optimize; however, this optimization is non-convex. It is interesting to note that we might have started with a convex problem, and to do inference, gotten to non-convex problem. However, at testing time, we can run this very quickly.

4 Experiments

4.1 Inferring Words from Keyboard Gestures

(Doeblin is his method).

The simple model u is character-level bigram, and the more complex model includes long-distance dependencies etc. Doeblin learns fast-mixing chains and we improve the results. As you increase the number of samples, Doeblin improves, whereas for Gibbs this is not the case.

4.2 Agenda-based parsing

We can construct logical forms given some utterance. Typically, parsing goes like this: If you have heard of CKY parser (dynamic programming). You basically construct a lot of hypotheses which are completely useless. There is a fairly old idea of Agenda-based parsing: control the order of derivation generation. The action is derivation on agenda: Then you put it in the chart, combine with the neighboring things, and add it to the agenda. The question is, which agenda items do you actually choose. Here we are basically trying to learn the cost-function.

We can think of this problem as a Markov decision process, where a state s is a (chart, agenda) tuple. The history is this entire sequence, and you have a reward if you get the right answer. The idea is that to find this $q_{theta}(a|s) \sim p_\theta(a|s)\exp(\cdot)$. The update is sample history from q, and then do a gradient update (though this is not on a particular loss that we can figure out).

Agenda-based parsing was able to sparsify the amount of derivations generated. So it got 6 times faster with no loss in accuracy. So we tried to figure out how to get to y over a sequence of steps, and how to model that sequence.

The main idea here: The design space of algorithms is much larger than the design space of models. The real question is how to take advantage of this richness.

So looking forward, recurrent neural nets have been very popular: The idea is you eat up all these words into some vector, and we pop them out. In some sense this state is capturing something about the history that allows you to make decisions in the future. We also want to have a more structured approach: Maybe we want to learn the structure of computations in a coarse-to-fine way (take multiple passes), adaptive (parse easy things first), and manage uncertainty explicitly, which is important for adaptivity, since you have to be aware of where you need to explore and search. These are some important open questions.

5 What is the right training signal?

Summary: Supervision is an interesting problem when we combine it with a model.

There is an important difference between training and testing. We can train really fast inference models, they are fast at test-time and high-predicting, so they work at test-time. But you still have to search in order to get the good parameters at training. What we want to do is **Learning from relaxed supervision** (joint work with Jacob Steinhardt). How can we give partial credit in a consistent way for learning? Moving to a simpler task than semantic parsing, our input is a sequence and our output is an unknown substitution cipher: We get a multiset otuput. There is also a latent sequence associated with input. The model $p_\theta(z|x)$ gives us soft substitutions between xs and zs, and we want to know if we can still output y. What is hard is the supervision aspect of this training, which is why it is training only part of the problem. Models you can control for: You can modify the model you use. But the supervision is something you do not get to control, you have to figure out how to reason with what you are given. Here the super vision is testing where the multiset of z equals y.

The key idea is to relax this supervision to the conjunction of a bunch of properties matching. Thus we can write $\cap_{j=1}^{V} \mathbf{1}[\text{count}(z,j) = \text{count}(y,j)]$. We can relax by saying $\text{match}_\beta(y,z) = \exp(\sum_{j=1}^{V} \beta_j \psi_j(z,y))$. Our objective is $\sum_z p_\theta(z|x) \cdot \text{match}_\beta(z,y)$. As $\beta \to \infty$, we recover the original supervision. It would be nice if we could optimize $\beta \geq 0$, but then the optimum is $\beta = 0$. Can we get natural pressure to increase β. The key idea is that we will normalize into a probability distribution. Then you can treate it like a joint model.

If you jointly train, there is a natural pressure for $\beta \to \infty$. Then you can introduce more supervision.

If β is small, supervision is weak, and doesn't change the model that much. If θ is large, then the model is strong and supervision is redundant.

5.1 Data?

Data is hard to get, even input is hard to get. The traditional approach is starting with zero examples. The traditional approach is to write stuff down. The traditional approach is ad-hoc: you require annotations, which are expensive.

In this approach, we flipped things around and decided to generate candidate logical forms (this is easy) and we can have a way of rendering logical form in an understandable form. Then we go to Mechanical Turk and ask people to parse this into actual English. Now we have examples paired with actual logical forms. This is kind of a neat trick, and leverages the decomposition (factorizing understanding and answering the question). Paraphrasing has nothing to do with knowing the answer, but still uses a lot of human power.

If you want to do something really fast and cheap, this method is pretty useable.

5.2 Reflections on training signal

As a summary: $x \to z \to y$. z is big and missing. We could try to relax the intractable superivision y, but how to bootstrap and anneal in general? Is there an abstract interpretation? The second path is to generate z and ask for x. How to generate novel z (active learning)? How do you manage the distribution of x - traditional thinking that data comes from a distribution no longer applies, you can get screwed if you are not careful.

Here x is the sentence, z is the logical form generated. y here is the answer to a question. If we have z, we do not need y since there exists $f(z) = y$, where f is a database query (this is the notion of factorization). Thus we are proxying z for y as a means of training better (highlighting training on the **process** to get to the answer rather than training a loss function on the answer itself).

So why do we care about z? But getting the correct z is important because of interpretability (where the answer comes from!) and robustness: Guaranteeing the correct process is much stronger than giving a simple raw output. If you got z correct, then whatever database table you have, you are getting the right answer. It is a powerful formalized guarantee to cover a much larger space. You have no guarantees for some large neural net that you will do any better. In some sense, do neural nets already do this sort of thing by learning 'functions' from each layer to the other?

Another difficulty is moving from supervision loss functions over y to supervision loss over z via some form of relaxation. This is particularly interesting.

6 Conclusion

We focus on prediction tasks requiring computation, and model that computational process directly. We also want to be able to latch onto bits and pieces of your supervision. In some sense, x and y are far away. Information theoretically there may be a very strong constraint on x, y. But computationally, we want to do the pieces of difficult pieces of supervision and let the model manage this. We maybe want to think about the true process, a true language understanding function and actively seek out inputs and outputs.

Another idea we have been playing around with is semantic parsing with 100% supervision. The assumption is that