CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 8

This discussion covers unsupervised pretraining methods in NLP and imitation learning methods.

1 Unsupervised Pretraining in Natural Language Processing

We will review several techniques for unsupervised pretraining in NLP. The general idea is to use unlabelled text, which is often easily accessible (for example on the internet, in books, other publications, etc...) in order to learn representations of language that can be useful for downstream tasks.

To illustrate why we might expect this to be helpful, we can imagine we want to translate English sentences to French, and are given a labelled dataset of English/French sentence pairs. You can imagine this task would be really difficult if you had no prior knowledge of English, while being much more manageable if you came in with a general understanding of the English language already, which can be learned using unsupervised data (for example, all the English text we see on the internet).

1.1 Word Embeddings

Before in lectures, we didn't worry much about how individual words were represented, often abstracting them away as one-hot vectors for simplicity. Our goal with word embeddings is to map words to real vectors such that distances in the representation are in some sense meaningful, which can make learning much easier for the downstream task. This would imply that "similar" words should be mapped to representations that are close to one another.

1.1.1 Naive Word2Vec

One way to learn embeddings is to optimize the representation to predict nearby words. More precisely, we can consider a *center word* c and try to predict its neighbors in the sentence (*context words* o) via logistic regression. We will associate each two vectors u_w and v_w, and optimize these word embeddings to optimize the likelihood

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w\in V}\exp(u_w^T v_c)}$$

averaged over all selections of center word c and context word o. Intuitively, this objective means that words that occur together (and so would show up as center/context word pairs o, c) would have higher inner product $u_0^T v_c$. It would also mean that if words a, b were similar in meaning and so were interchangeable, we would expect their embeddings u_a and u_b to be similar (as well as v_a, v_b), since they would appear in similar contexts. The embedding for each word was split into two components u, v to make optimization more tractable, which are simply averaged together to produce the final embedding after training.

1.1.2 Making Word2Vec Tractable

As mentioned in lecture, the normalizing factor $\sum_{w\in V}\exp(u_w^T v_c)$ involves summing the logits for *every* word in the entire vocabulary, which is very slow given how many words there are in total.

One way to address the issue is to simply train using *binary classification*. We can instead choose to optimize

$$p(o \text{ is the right word}|c) = \frac{1}{1 + \exp\left(-u_o^T v_c\right)}.$$

The issue with this approach is that if we only sample positive examples (pairs of words o, c that are actually neighbors of one another), then a trivial solution would be to make u, v to be the same, very large vector for all words, which would cause our binary classifier to always confidently predict that any two words are valid center and context pairs.

To avoid this degenerate solution, we also include *negative samples* in every batch. Instead of just maximizing the likelihood on valid center/context pairs, we'll also reduce the likelihood of words randomly sampled from the dictionary to get the objective

$$\max \sum_{c,o} \left(\log p(o \text{ is the right word}|c) + \sum_{w} \log p(w \text{ is wrong}|c) \right),$$

where the negative examples w are randomly sampled. Now, optimizing this objective will try to push v_c and u_o closer together for valid context/center pairs, while pushing v_c away from u_w for all other words in the vocabulary.

1.2 Pretrained Language Models

One weakness of the previous Word2Vec approach is that we train by simply predicting neighbors from an individual word, and so the representations for a word do not depend on its context after training is complete. This is somewhat limiting, since words can have very different meanings depending in the context. At a high level, one simple way we can embed words in a context-dependent manner is to take a language model (for example an LSTM) trained on some task, and to run a sentence through it, taking the hidden state of the model as the embedding for each word. Since these language models presumably had to use the context in order to solve the task they were trained on, using the hidden state as an embedding should provide context-dependent representations of words. We will briefly discuss two examples of this approach.

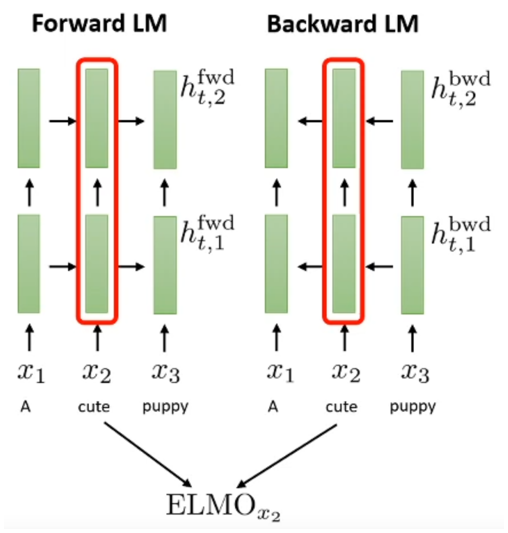

Figure 1: ELMo takes the hidden states in a bi-directional LSTM to generate word embeddings. The LSTMs are both trained via sequence prediction.

ELMo: We note that if we simply ran an LSTM forward through a sentence to generate the embeddings of words, the embedding of each word would only depend on those that came before it, rather than the full context of the word. ELMo addresses this issue by simply training a bi-directional LSTM (both trained to predict the next/previous word), and concatenating hidden states of both directions together to form an embedding. This approach is very similar to the previous Word2Vec approach in that the embeddings are trained based on predicting nearby words, with the main difference being that the we aren't exactly embedding single words, but rather words in their context.

BERT: Instead of using a bi-directional LSTM like ELMo, BERT relies on a *single* transformer to generate embeddings. While the previous transformers we saw for sequence modeling relied on masked self-attention to avoid peeking into the future, our goal here is to digest the entire context of a word to produce an embedding, which eliminates the need for the mask. However, this presents a complication if we were to try train embeddings to predict the next word like ELMO did. The issue here is that if we did unmasked self-attention, we can already directly see the next word in the input, making prediction completely trivial and preventing useful representations from being learned.

The solution is to simply change the unsupervised task. Instead of predicting the next word in sentence, we instead randomly mask out certain words in the input, and then train the embedding to predict the masked out words. In this way, our model is forced to learn context dependent word-level representations to predict the missing words.

In addition to learning word-level representations by predicting masked out words, BERT also tries to learn *sentence-level* representations. To train this, BERT takes in pairs of sentences, randomly permutes their order, and trains a binary classifier to predict which of the two sentences came first originally. Depending on the downstream task, we can either use the sentence level representation outputted by BERT or the word-level representations in the downstream task. We can use BERT for downstream tasks both by simply finetuning the entire model on the downstream tasks, or taking combinations of the hidden states as fixed representations.

2 Imitation Learning

So far in the course, we have focused on supervised learning for prediction tasks. In imitation learning and reinforcement learning, our goal will now be to solve *control* problems. Control problems will generally involve making sequences of decisions, with each decision affecting the future, in order to accomplish some goal. To formalize this problem, we will introduce **Markov Decision Processes** (MDPs). In imitation learning, we assume we have access to an expert policy that already solves the task we care about, and our goal will be to learn a policy to solve a task by copying these expert demonstrations.

2.1 Markov Decision Processes

A Markov decision process (MDP) is specified by a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $P(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ defining the probability of the next state given the current state and action, and a reward function $r(\mathbf{s}, \mathbf{a})$. A key property in MDPs is the *Markov property*, which similar to Markov chains, states that conditioned on the current state and action, the next state is conditionally independent of anything in the past.

The goal is to learn to take actions in order to maximize the sum of rewards over trajectories in the MDP. Due to the Markov property, all such optimal policies are Markovian (also called stationary processes) and described as conditional distributions $\pi(\mathbf{a}|\mathbf{s})$ over only the current state. This essentially states that the optimal action depends only on the current state, rather than any states or actions further in the past.

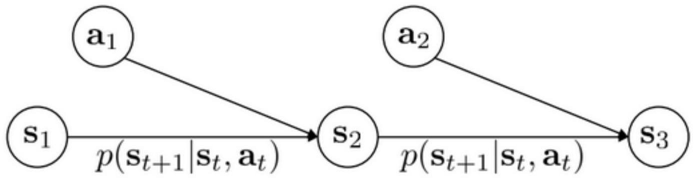
$\mathbf{a}_1$
$\mathbf{a}_2$
$\mathbf{s}_1$
$\mathbf{s}_2$
$\mathbf{s}_3$
$p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
$p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$

Figure 2: Markov property in MDPs

We can imagine the state $\mathbf{s}$ as capturing the current state of the world, actions $\mathbf{a}$ to be the decisions we make, the transition function $P(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ describing how our actions affect the world around us, and the reward $r(\mathbf{s}, \mathbf{a})$ capturing some notion of success at what we want to accomplish.

While solving imitation learning problems will not explicitly require access to the reward, we should keep in mind that success in imitation learning is not necessarily measured directly in how well we match the expert (as measured in perhaps negative-log-likelihood on the expert dataset like we would consider in supervised learning), but in how well our learned policy actually executes the task we care about. The task we care about is often specified (loosely) as a reward function.

Problem 1: Probability of a trajectory under a Markovian policy

Given a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$, compute the log probability of a trajectory $\tau = ((\mathbf{s}_0, \mathbf{a}_0), (\mathbf{s}_1, \mathbf{a}_1), \ldots, (\mathbf{s}_T, \mathbf{a}_T))$ using the Markov property.

Solution 1: Probability of a trajectory under a Markovian policy

Using the chain rule of probability, we can decompose the probability as

$$p(\tau) = p(s_0, a_0)\prod_{t=1}^{T} p(\mathbf{s}_t, \mathbf{a}_t | \mathbf{s}_0, \mathbf{a}_0, \ldots, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}) \tag{1}$$

$$= p(\mathbf{s}_0)\pi(\mathbf{a}_0|\mathbf{s}_0)\prod_{t=1}^{T} p(\mathbf{s}_i | \mathbf{s}_0, \mathbf{a}_0, \ldots, \mathbf{s}_{t-1}, \mathbf{a}_{t-1})\pi(\mathbf{a}_t|\mathbf{s}_t) \qquad \text{Markovian policy} \tag{2}$$

$$= p(\mathbf{s}_0)\pi(\mathbf{a}_0|\mathbf{s}_0)\prod_{t=1}^{T} p(\mathbf{s}_t | \mathbf{s}_{t-1}, \mathbf{a}_{t-1})\pi(\mathbf{a}_t|\mathbf{s}_t) \qquad \text{Markov property of transitions} \tag{3}$$

Taking logarithms, we have

$$\log p(\tau) = \log p(\mathbf{s}_0) + \log \pi(\mathbf{a}_0|\mathbf{s}_0) + \sum_{t=1}^{T} \log p(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1}) + \log \pi(\mathbf{a}_t|\mathbf{s}_t).$$

2.1.1 Partially Observed Markov Decision Processes

In practice, we may not be able to access the full state $\mathbf{s}$ that satisfies the Markov property, but instead rely on some limited observations $\mathbf{o}$. For example, when driving a car, we cannot see everything around us, even though things we cannot see may impact how the world behaves around us.

Here, optimal policies may no longer be stationary, but may need to depend on all past observations ($\pi(\mathbf{a}_t|\mathbf{o}_1 \ldots, \mathbf{o}_t)$). Unless otherwise specified, we're going to assume we're in a fully observed MDP with access to the true state.

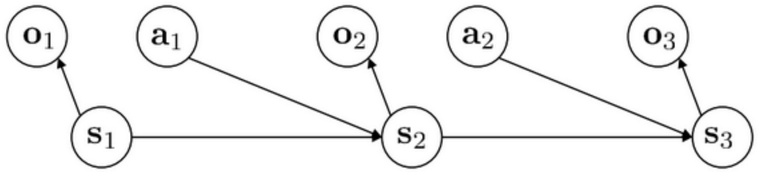

Figure 3: The Markov property still holds for the underlying state $\mathbf{s}$ in POMDPs, but our policies cannot depend on $\mathbf{s}$ directly. Instead, we are forced to take actions based only on the observations $\mathbf{o}$.

2.2 Behavior Cloning

We now describe a very simple approach to imitation learning, which we refer to as **behavior cloning**. We assume that we have a collection of N expert trajectories, where each trajectory τ is a sequence of state-action pairs $((\mathbf{s}_0, \mathbf{a}_0), (\mathbf{s}_1, \mathbf{a}_1), \ldots, (\mathbf{s}_T, \mathbf{a}_T))$. We add every state action pair into our dataset, forming a dataset $\mathcal{D} = \{(\mathbf{s}_0, \mathbf{a}_0), \ldots, (\mathbf{s}_{NT}, \mathbf{a}_{NT})\}$.

Behavior cloning simply trains our policy $\pi(\mathbf{a}|\mathbf{s})$ via supervised learning on this dataset of expert experiences. Each state is considered an input, and the corresponding expert action is considered the label, and if we train with maximum likelihood, our objective becomes

$$\max_{\theta} \sum_{\mathbf{s},\mathbf{a}\sim\mathcal{D}} \log \pi_\theta(\mathbf{a}|\mathbf{s}).$$

Problem 2: Behavior cloning maximizes likelihood of expert trajectories

Using the decomposition of the probability of a trajectory from the previous problem, show that the behavior cloning objective maximizes the probability of the expert trajectories (assumed to be generated by some expert β that produces a distribution over trajectories) under the learned policy

$$\max_\theta \mathbb{E}_{\tau\sim\beta} \log \pi_\theta(\tau).$$

Solution 2: Behavior cloning maximizes likelihood of expert trajectories

Recall that

$$\log p(\tau) = \log p(\mathbf{s}_0) + \log \pi(\mathbf{a}_0|\mathbf{s}_0) + \sum_{t=1}^{T} \log p(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1}) + \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t).$$

Ignoring terms that don't depend on the policy parameters, maximizing the log-likelihood of a trajectory τ is then equivalent to maximizing

$$\sum_{t=0}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t).$$

Averaging over all the timesteps in each trajectory and over all the trajectories, we see this is identical to the supervised objective for behavior cloning (up to having only an empirical distribution over expert trajectories in the dataset instead of the true distribution).

2.3 DAgger

However, having a policy that assigns high likelihoods to expert trajectories is often not enough to ensure the learned policy attains good performance.

The primary issue is a distribution shift between training and test time. During training, the policy is only being trained on states that were visited by the expert policy. During test time, we actually execute the learned policy, and small mismatches between the learned policy and the expert can potentially take us to new states not seen during training. On these new states that the imitation policy hasn't been trained on, the imitation policy would likely not match the expert's behavior well. Thus, even if the expert policy were capable of recovering from the earlier mistakes and still solve the task, the imitation learning policy would instead continue to make mistakes.

One way to remedy this is to simply do a better job at matching the expert policy to avoid deviating far from the expert trajectories. However, this can require extremely accurate models, and can be tricky to accomplish (even with large amounts of expert data) when the expert policy is non-Markovian (so cannot be matched exactly by a Markovian policy) or if the expert policy is multimodel (and choice our probability distribution for our policy is not expressive enough to match it).

Another approach alter our data distribution we train on to better cover the trajectories we'll encounter during test time. This is the approach taken in the **dataset aggregation (Dagger**) algorithm, which iteratively collects new trajectories from the current policy, labels those trajectories using the expert policy, and adds the relabeled trajectories to our dataset and retraining. This way, we are constantly updating our state distribution to include our current policy, and we can stop when our imitation policy's distributions stabilize and we obtain good performance in our desired task.

While Dagger can be very effective at mitigating the distribution mismatch issues, it does require the agent to interact with the environment during learning, as well querying the expert to figure out what actions it would take at the new states. Both of these can potentially be costly.