Learning to Search (Part II)
Reminders

• Homework 1: Neural Networks for Sequence Tagging
  – Out: Wed, Sep 7 (later today!)
  – Due: Fri, Sep 16 at 11:59pm

• Homework 2: Learning to Search for RNNs
  – Out: Fri, Sep 16
  – Due: Wed, Sep 28 at 11:59pm
IMITATION LEARNING
Imitation Learning

state (sensors)

agent (car)

policy (neural network)

action (left / right)

Figures from Pan et al. (2018)
Imitation Learning

Whiteboard:

– Fully supervised imitation learning
– The pitfall of fully supervised imitation learning
– DAgger for imitation learning
Imitation Learning

• Policies: \( \pi : S \rightarrow A \)
  – Def: a **policy** is a function that maps from a state to an action
  – Def: a **model policy** is one that is parameterized such that we can learn its parameters
  – Def: an **expert policy** is the one we want to learn to mimic

• Trajectories: \( \tau = [(s_1, a_1), (s_2, a_2), \ldots, (s_T, a_T)] \)
  – Def: a **trajectory** is sequence of state/action pairs
  – Def: the **time horizon** (e.g. \( T \)) is the length of the trajectory
  – Def: a **training trajectory** is a trajectory where the actions were those taken by an expert policy
  – (in imitation learning, the training dataset is a collection of training trajectories \( \mathcal{D} = \{ \tau^{(i)} \}_{i=1}^{N} \))
Imitation Learning

Here we consider two algorithms:

– **Algorithm 1**: Supervised Imitation Learning
– **Algorithm 2**: DAgger Imitation Learning

We will describe both for the setting where training is done by Stochastic Gradient Descent (SGD)

However, both are general enough that they could be employed with any optimization technique (e.g. Gradient Descent)
**Imitation Learning**

**Algorithm 1: Supervised Imitation Learning**

- **Key idea:**
  - follow the expert policy to collect the sequence of states that it visits and the actions it takes
  - then train a multi-class classifier to take similar actions to the expert

**Diagram:**
- blue: expert policy trajectory
- green arrows: training examples \((s_t, a_t)\) of state \(s_t\) visited by expert, and action \(a_t\) taken by expert
Imitation Learning

Algorithm 2: DAgger Imitation Learning

• blue: expert policy trajectory
• red: model policy trajectory
• purple arrows: training examples \((s_t, a_t)\) of state \(s_t\) visited by model, and action \(a_t\) taken by expert
• grey box: the purple states inside would never be visited if we were only following the expert policy

• Key idea:
  – follow the model policy to collect the sequence of states, but use the expert policy to record what action should have been taken
  – then train a multi-class classifier to take similar actions to the expert
  – in this way, we learn how to correct for the model’s mistakes!
def trainSupervised(\(\pi^*, E\)):
    initialize policy \(\pi_\theta\)
    for \(i\) in \(1...N\):
        for \(t\) in \(1...T\):
            observe state \(s_t\)
            take action \(a_t = \pi^*(s_t)\)
            \(\tau^{(i)} = \tau^{(i)} + [(s_t, a_t)]\)
            \(D_{\text{train}} = D_{\text{train}} \cup \{\tau^{(i)}\}\)

    for \(\tau^{(i)}\) in \(D_{\text{train}}\):
        for \((s_t, a_t)\) in \(\tau^{(i)}\):
            update policy \(\pi_\theta\) with one step of SGD on example \((s_t, a_t)\)

repeat for \(E\) epochs
return \(\pi_\theta\)

def predict(\(\pi_\theta\)):
    for \(t\) in \(1...T\):
        observe state \(s_t\)
        take action \(a_t = h_\theta(s_t)\)
        incur loss \(\ell_t\)

Algorithm 1: Supervised Imitation Learning (Version 1)
def trainSupervised(\(\pi^*, E\)):
    initialize policy \(\pi_\theta\)
    for \(i\) in 1…N:
        for \(t\) in 1…T:
            observe state \(s_t\)
            take action \(a_t = \pi^*(s_t)\)
            incur loss \(\ell_t\)
        update policy \(\pi_\theta\) with
        one step of SGD on example \((s_t, a_t)\)
    repeat for \(E\) epochs
    return \(\pi_\theta\)

def predict(\(\pi_\theta\)):
    for \(t\) in 1…T:
        observe state \(s_t\)
        take action \(a_t = \pi_\theta(s_t)\)
    create training dataset and
    train model policy (i.e. classifier)

This returns exactly the
same model policy as the
previous version.

The only change is that
we’ve combined the
collection of training data
and the SGD update.
def trainDAgger(\(\pi^*\), E):
    \(\pi_\theta = \text{trainSupervised}(\pi^*, 1)\)
    for i in 1... N:
        for t in 1... T:
            observe state \(s_t\)
            take action \(\hat{a}_t = \pi_\theta(s_t)\)
            store action \(a_t = \pi^*(s_t)\)
            update policy \(\pi_\theta\) with one step of SGD on example \((s_t, a_t)\)

repeat for E-1 epochs
return \(\pi_\theta\)

def predict(\(\pi_\theta\)):
    for t in 1... T:
        observe state \(s_t\)
        take action \(a_t = \pi_\theta(s_t)\)
        incur loss \(\ell_t\)

Algorithm 2: DAgger for Imitation Learning (Version 1)

We initialize by running 1 epoch of supervised imitation learning
observe state \(s_t\)
take action \(a_t = \pi_\theta(s_t)\)
incur loss \(\ell_t\)

Now the action we take is given by the model policy

We still train by updating on the expert policy’s action
Imitation Learning

**Algorithm 2: DAgger for Imitation Learning (Version 2)**

```python
def trainDAgger(\(\pi^*, E, \beta = [\beta_1, \ldots, \beta_N]\)):
    initialize policy \(\pi_\theta\)
    for \(i\) in \(1 \ldots N\):
        \(\pi_i = \beta_i \pi^* + (1 - \beta_i) \pi_\theta\)
    for \(t\) in \(1 \ldots T\):
        observe state \(s_t\)
        sample action \(\hat{a}_t \sim \pi_i(s_t)\)
        store action \(a_t = \pi^*(s_t)\)
        update policy \(\pi_\theta\) with one step of SGD on example \((s_t, a_t)\)

repeat for \(E\) epochs
return \(\pi_\theta\)
```

We’ve dropped the call the supervised imitation learning. Instead, we compute a policy at each iteration that is a probabilistic mixture of the expert policy and our (current) model policy.

Since the mixture policy is stochastic, we sample a policy.

\(\beta = [\beta_1, \ldots, \beta_N]\) is our schedule for how much weight to put on the expert/model policies in the mixture.
Imitation Learning

Algorithm 2: DAgger for Imitation Learning (Version 3)

def trainDAgger(\(\pi^*, E, \beta = [\beta_1, \ldots, \beta_N]\)):
    initialize policy \(\pi_\theta\)
    for \(i \in 1 \ldots N\):
        \(\pi_i = \beta_i \pi^* + (1 - \beta_i) \pi_\theta\)
    for \(t \in 1 \ldots T\):
        observe state \(s_t\)
        sample action \(\hat{a}_t \sim \pi_i(s_t)\)
        store action \(a_t = \pi^*(s_t)\)
        \(\tau^{(i)} = \tau^{(i)} + [(s_t, a_t)]\)
    for \((s_t, a_t)\) in \(\tau^{(i)}\):
        update policy \(\pi_\theta\) with one step of SGD on example \((s_t, a_t)\)
    repeat for \(E\) epochs
    return \(\pi_\theta\)

def predict(\(\pi_\theta\)):
    for \(t \in 1 \ldots T\):
        observe state \(s_t\)
        take action \(a_t = \pi_\theta(s_t)\)
        incur loss \(\ell_t\)

Build the i’th trajectory
Take T steps of SGD to train on the the i’th trajectory
Mixing Policies

**Question:**
How would you implement the mixture policy if the model and expert policies were **deterministic**?

\[ \pi_i = \beta_i \pi^* + (1 - \beta_i) \pi_\theta \]

**Answer:**

**Question:**
How would you implement the mixture policy if the model and expert policies were **stochastic**?

\[ \pi_i = \beta_i \pi^* + (1 - \beta_i) \pi_\theta \]
STRUCTURED PREDICTION AS SEARCH
Structured Prediction as Search

- **Key idea:** convert your structured prediction problem to a search problem!
- **Example:** for POS tagging, each node in the search space corresponds to a partial tag sequence.

![Diagram of search space for POS tagging]
Basic Neural Network

• Suppose we wish to predict the tags **greedily left to right**
• Simple neural network looks at the previous word, the previous tag **prediction**, the current word, and the next word
• From these it builds a **probability distribution over output tags**
• Then it **selects the argmax**
Basic Neural Network

• Suppose we wish to predict the tags **greedily left to right**
• Simple neural network looks at the previous word, the previous tag prediction, the current word, and the next word
• From these it builds a **probability distribution over output tags**
• Then it **selects the argmax**
Basic Neural Network

- Suppose we wish to predict the tags **greedily left to right**
- Simple neural network looks at the previous word, the previous tag **prediction**, the current word, and the next word
- From these it builds a **probability distribution over output tags**
- Then it **selects the argmax**

Suppose we wish to predict the tags **greedily left to right**

Simple neural network looks at the previous word, the previous tag **prediction**, the current word, and the next word

From these it builds a **probability distribution over output tags**

Then it **selects the argmax**
Learning to Search

Whiteboard:
- Problem Setting
- Ex: POS Tagging
- Other Solutions:
  - Completely Independent Predictions
  - Sharing Parameters / Multi-task Learning
  - Graphical Models
- Today’s Solution: Structured Prediction to Search
  - Search spaces
  - Cost functions
  - Policies