10-418/10-618 Machine Learning for Structured Data

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Learning to Search (Part II)

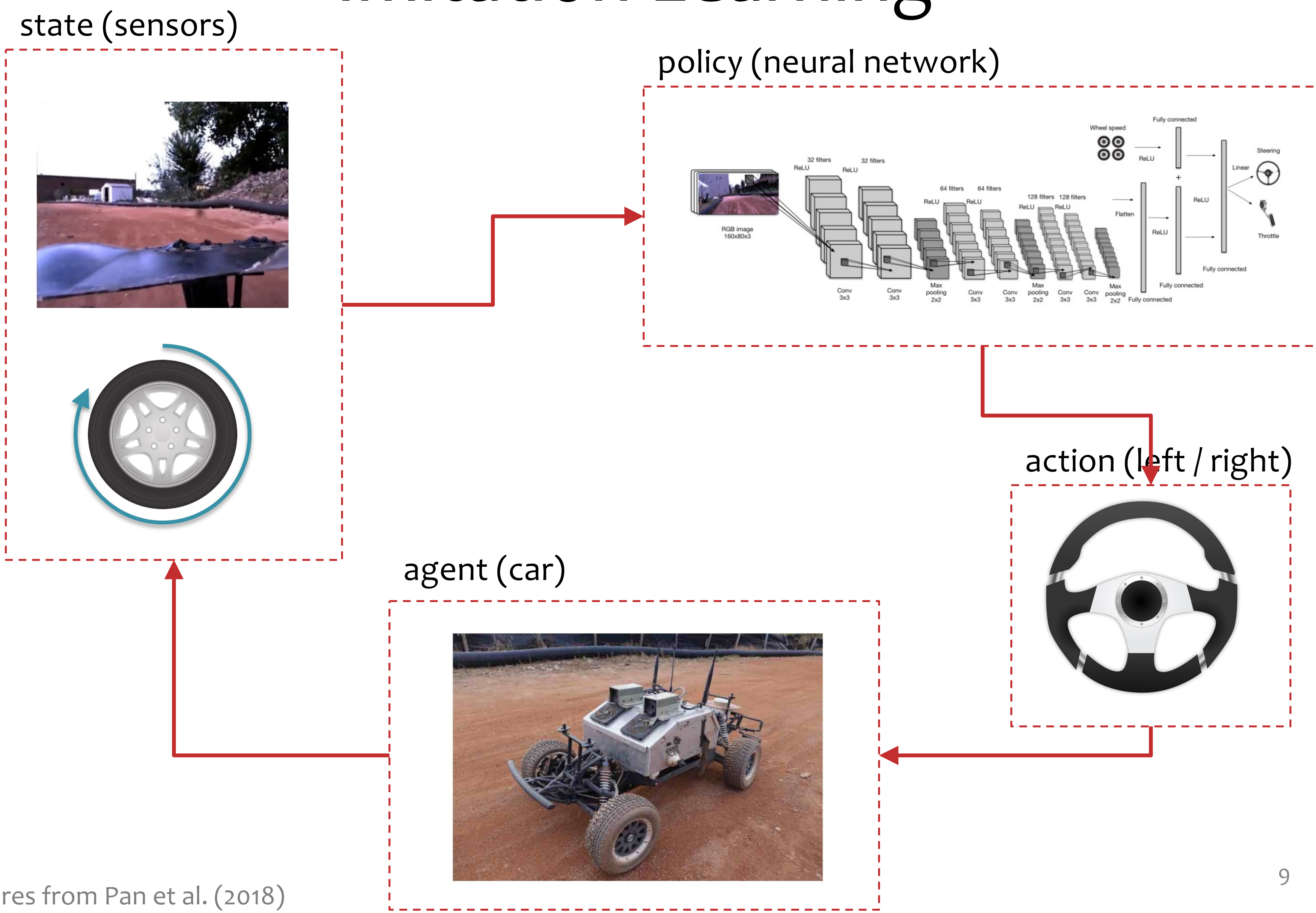
Matt Gormley
Lecture 5
Sep. 14, 2022

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



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: $\pi : \mathcal{S} \rightarrow \mathcal{A}$
  - *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: $\boldsymbol{\tau} = [(s_1, a_1), (s_2, a_2), \ldots, (s_T, a_T)]$
  - *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 $\mathcal{D} = \{\boldsymbol{\tau}^{(i)}\}_{i=1}^{N}$)

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)

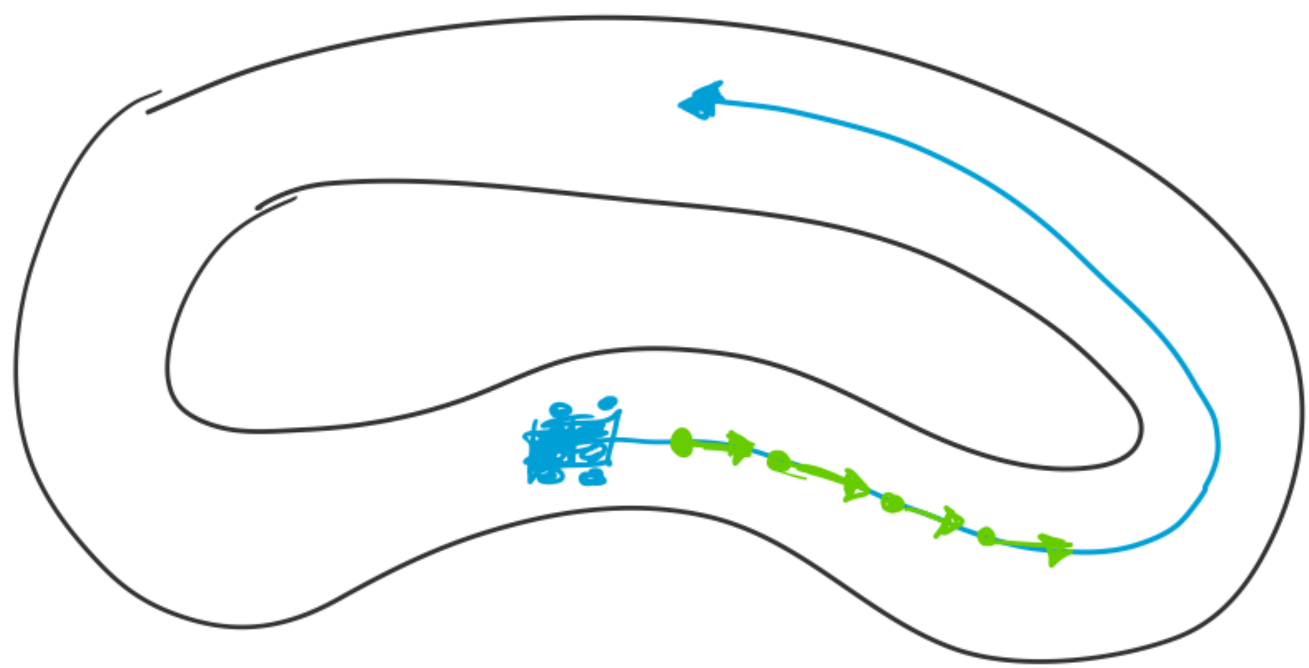
However, both are general enough that they could be employed with any optimization technique (e.g. Gradient Descent)

Imitation Learning

Algorithm 1: Supervised Imitation Learning

- 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

- **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

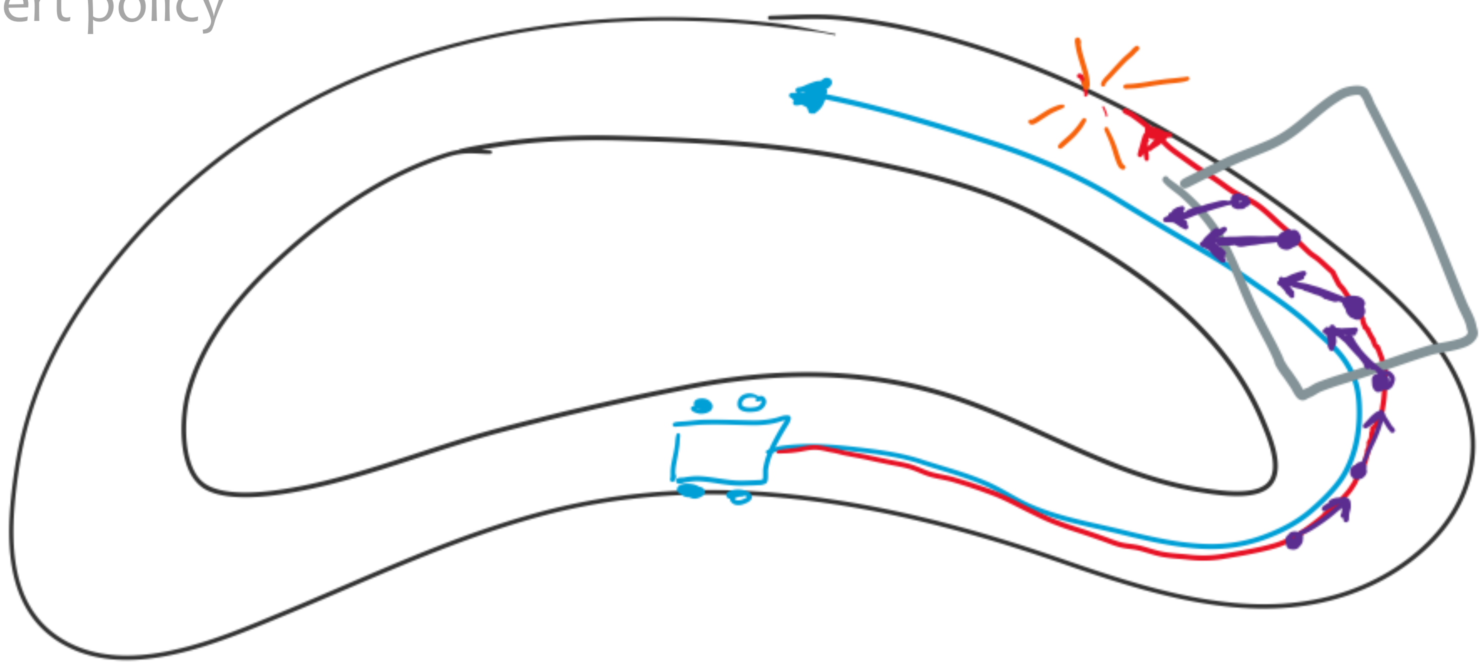


Imitation Learning

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**!



Imitation Learning

Algorithm 1: Supervised Imitation Learning (Version 1)

def trainSupervised(π^*, E):
    initialize policy π_θ
    for i **in** 1… N:
        for t **in** 1… T:
            observe state s_t
            take action $a_t = \pi^*(s_t)$
            $\boldsymbol{\tau}^{(i)} = \boldsymbol{\tau}^{(i)} + [(s_t, a_t)]$
        $D_{train} = D_{train} \cup \{\boldsymbol{\tau}^{(i)}\}$

create training dataset

    for $\boldsymbol{\tau}^{(i)}$ **in** D_{train}:
        for (s_t, a_t) **in** $\boldsymbol{\tau}^{(i)}$:
            update policy π_θ *with one step of SGD on example* (s_t, a_t)
    repeat *for* E *epochs*
    return π_θ

h_θ

$(\vec{x}, y)$

train model policy (i.e. classifer)

def predict(π_θ):
    for t **in** 1… T:
        observe state s_t
        take action $a_t = \pi_\theta(s_t)$
        incur loss ℓ_t

total loss $= \sum_t \ell_t$

$\pi_\theta'(s_t) = \operatorname{argmax}_a p_\theta(a|s_t)$

$\pi_\theta(s_t) \sim p_\theta(a|s_t)$

Imitation Learning

Algorithm 1: Supervised Imitation Learning (Version ~~1~~ 2)

*online

def trainSupervised(π^*, E):
    initialize policy π_θ
    for i **in** 1… N:
        for t **in** 1… T:
            observe state s_t

            take action $a_t = \pi^*(s_t)$

            update policy π_θ *with one step of SGD on example* (s_t, a_t)

    repeat *for* E *epochs*
    return π_θ

def predict(π_θ):
    for t **in** 1… T:
        observe state s_t
        take action $a_t = \pi_\theta(s_t)$
        incur loss ℓ_t

create training dataset and train model policy (i.e. classifer)

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.

Imitation Learning

Algorithm 2: DAgger for Imitation Learning (Version 1)

def trainDAgger(π^*, E):
  π_θ = trainSupervised(π^*, 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 π_θ *with one step of SGD on example* (s_t, a_t)

  repeat *for* E-1 *epochs*
  return π_θ

def pre
  fo
    observe state s_t
    take action $a_t = \pi_\theta(s_t)$
    incur loss ℓ_t

We initialize by running 1 epoch of supervised imitation learning

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)

def trainDAgger(π^*, E, $\boldsymbol{\beta} = [\beta_1, \ldots, \beta_N]$):
  initialize policy π_θ
  for i **in** 1… N:
    $\pi_i = \beta_i \, \pi^* + (1 - \beta_i) \, \pi_\theta$
    for t **in** 1… T:
      observe state s_t
      sample and take *action* $\hat{a}_t \sim \pi_i(s_t)$
      store action $a_t = \pi^*(s_t)$

      update policy π_θ *with one step of SGD on example* (s_t, a_t)

  repeat *for* E *epochs*
  return π_θ

def pr…
  for t **in** 1… T:

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~~ action

$N>1: \beta = [\overbrace{1, 1, 1, \ldots, 1}^{N}, 0, 0, 0, \ldots, 0]$

$N=1: \beta = [1, 0, \ldots, 0]$

$\boldsymbol{\beta} = [\beta_1, \ldots, \beta_N]$ is our schedule for how much weight to put on the expert/model policies in the mnixture

Imitation Learning

Algorithm 2: DAgger for Imitation Learning (Version 3)

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

    repeat *for* E *epochs*
    return π_θ

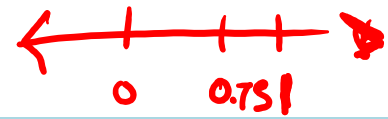
def predict(π_θ):
    for t **in** 1… T:
        observe state s_t
        take action $a_t = \pi_\theta(s_t)$
        incur loss ℓ_t

Build the i'th trajectory

Take T steps of SGD to train on the the i'th trajectory

Mixing Policies

$\beta_i = 0.75$



Question: Q1

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:

L, R, S

$\pi^* = [1 \quad 0 \quad 0]$

$\pi_\theta = [0 \quad 1 \quad 0]$

$\pi_i = [0.75, 0.25, 0]$

Question: Q2

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

$\pi_i = \beta_i \, \pi^* + (1 - \beta_i) \, \pi_\theta$

Answer:

$\pi^* = [0.9, 0.1, 0]$

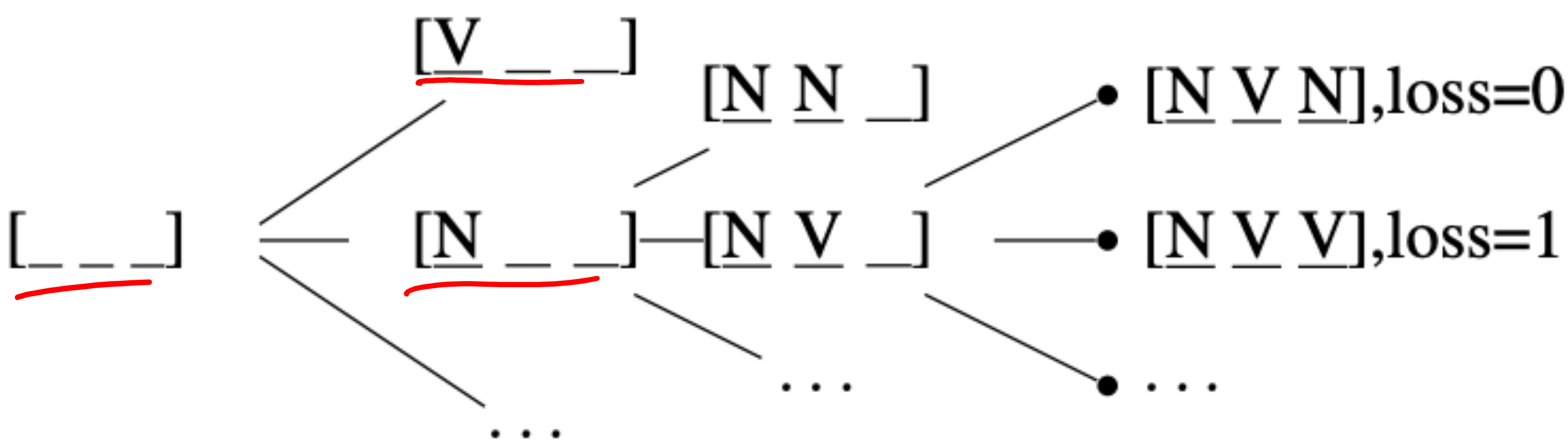
$\pi_\theta = [0.2, 0.7, 0.1]$

$\pi_i = \frac{[0.9+0.2,\ 0.1+0.7,\ 0+0.1]}{2}$

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



$\vec{x}$ = cat sat mat

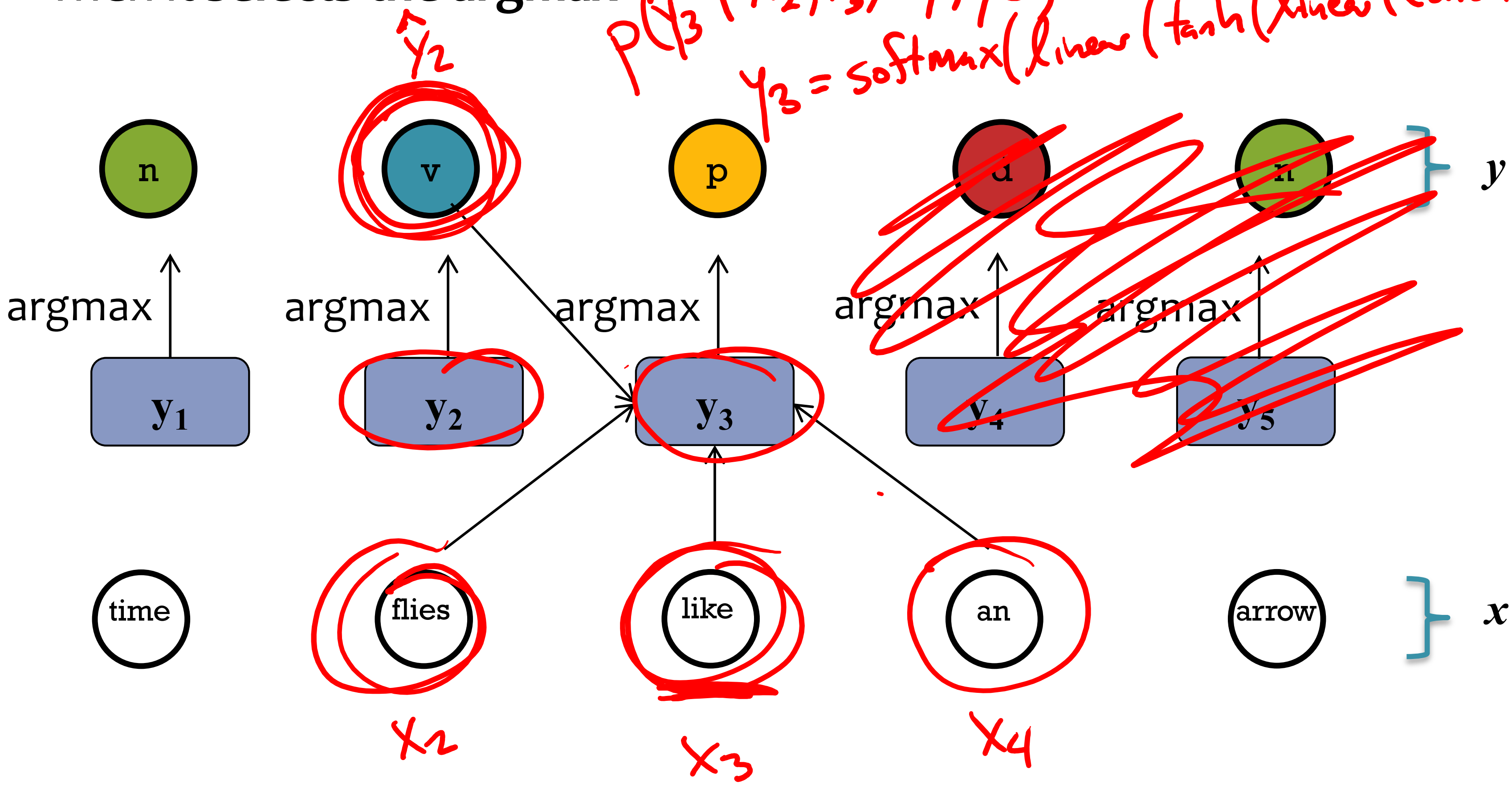
$\vec{y}^*$ = N V N

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**

$p(\hat{y}_3 \mid x_2, x_3, x_4, \hat{y}_2)$

$y_3 = \mathrm{softmax}(\mathrm{linear}(\tanh(\mathrm{linear}(\mathrm{concat}(x_2, x_3, x_4, \hat{y}_2$

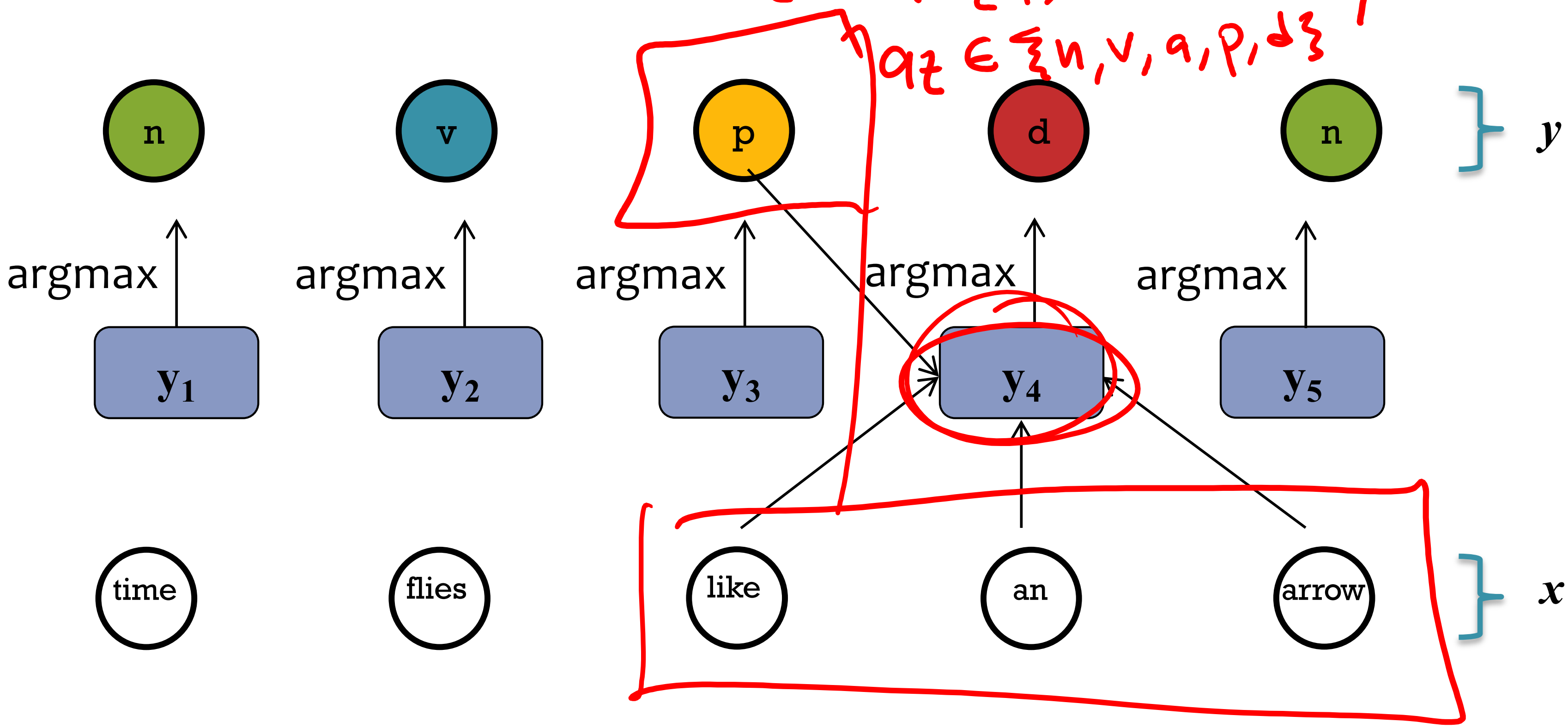


Basic Neural Network

- 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**

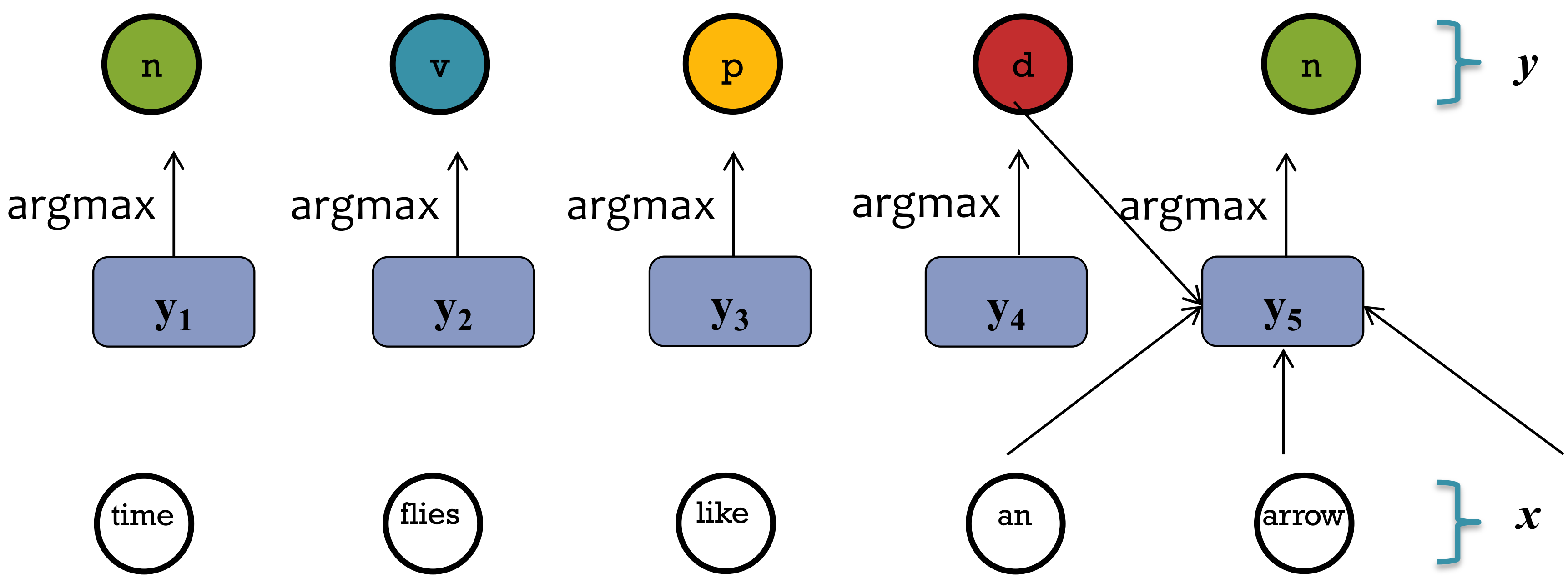
$s_t = \langle x_{t-1}, x_t, x_{t+1}, \hat{y}_{t-1} \rangle$

$a_t \in \{n, v, a, p, d\}$



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**



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