Learning Turing Machines from Examples

Brian Gu, MIT '20

February 2017

1 Introduction

We are interested in learning an algorithm, in the form of a discrete Turing Machine, from a series of input/output examples.

2 Model

The simplest model consists of three parts: an input tape, an output tape, and a Probabilistic Turing Machine (PTM). The PTM operates on a collection of symbols which form an *alphabet* Σ–in the simplest case containing only two symbols, 0 and 1. In addition to the standard alphabet symbols, our model includes four special symbols for the purpose of I/O mechanics.

Input tape: This is a 1D tape. The start of the tape is indicated by a special start symbol s, and the end of the tape is indicated by a special terminal symbol t. The remaining symbols are standard alphabet symbols. The tape head is initially located at the first cell. For more complex tasks, we may choose to make the input tape two dimensional and simply give the PTM the option to move its tape head in any of the four directions.

Output tape: This is a 1D tape that is initially entirely blank. The tape head is initially located at the leftmost cell. At any time step, the control can either write a symbol at the current cell and move one cell to the right, or do nothing.

Controller: Drawn from the idea of a Probabilistic Turing Machine, the controller is a state machine with probabilistic transitions and write operations. A deterministic Turing Machine in this model can be fully specified with a transition function $\delta : Q \times (\Sigma \cup \{s, t\}) \to Q \times (\Sigma \cup \{\epsilon, \varnothing\}) \times \{L, R\}$. Here, ϵ refers to the empty string (a "no-op write"), and $\varnothing$ refers to the action of terminating the procedure. A PTM, however, assigns probabilities to transitions, and is specified by a function $f : Q \times (\Sigma \cup \{s, t\}) \times Q \times (\Sigma \cup \{\epsilon, \varnothing\}) \times \{L, R\} \to [0, 1]$. Here, $f(q_1, s_1, q_2, s_2, D) = \Pr(\delta(q_1, s_1) = (q_2, s_2, D))$, where q_1 and q_2 are states, s_1 and s_2 are symbols, and $D \in \{L, R\}$. At every step, the controller selects a new state and a symbol to write to the output tape (or no-op, or terminate), and then moves the input tape head either left or right.

One state of the PTM is arbitrarily designated the start state.

Internally, rather than storing probabilities, we store unnormalized likelihoods which can be converted to probabilities with softmax. Thus, instead of directly storing f, we store a function w such that

$$f(q_1, s_1, q_2, s_2, D) = \frac{e^{w(q_1, s_1, q_2, s_2, D)}}{\sum_{(q', s', d) \in Q \times (\Sigma \cup \{\epsilon, \varnothing\}) \times \{L, R\}} e^{w(q_1, s_1, q', s', d)}}$$

This normalization also enforces the constraint that

$$\sum_{(q', s', D) \in Q \times (\Sigma \cup \{\epsilon, \varnothing\}) \times \{L, R\}} f(q_1, s_1, q', s', D) = 1$$

The PTM thus operates as follows:

1. The PTM begins with both tape heads on the leftmost cells of the input and output tapes; it begins at the start state.

2. The PTM probabilistically selects a new state, a symbol to write, and a direction to move the input tape head based on the probabilistic transition function.

3. Repeat step 2 until the PTM outputs the termination symbol.

3 Strongly Supervised Training

3.1 Extracting a Turing Machine

Strongly supervised experiments are interesting for two reason: first, they are equivalent to the sub-problem of extracting a Turing Machine/human-readable procedure from a "black box" (for example, the control of a Neural Turing Machine). Second, they provide an upper bound on the performance of this approach in the weakly supervised setting, allowing us to assess viability of this approach.

3.2 Initial Program

Define an "action" to be a pair, consisting of the symbol written (or no-op, or terminate) and the direction of input tape head movement, in a single timestep during the operation of a PTM. For a given "correct" sequence of actions, define a "succeeding branch of computation" to be a (possibly partially completed) branch of computation whose actions match those of the given sequence. Finally, define a "successful branch of computation" to be a completed branch of computation whose actions perfectly match the actions in a given sequence.

In general, the correct sequence of actions is given by a *target* deterministic Turing Machine, programmed to carry out the desired algorithm.

To train the PTM, we run all succeeding branches of computation simultaneously, pruning out non-succeeding branches and updating probabilities to favor succeeding branches. This is accomplished with a simple dynamic programming algorithm: we represent the PTM at any timestep as a vector of real numbers adding to one, with each coordinate representing the probability that the PTM is at a specific state. This is the only information that must be tracked, since all succeeding branches will have the same tape head location and output tape state at the same timestep.

Let τ be the learning schedule. At time t during training, suppose that the PTM is at state q_1 with probability p, and that it has just read symbol s_1. The update rule for $w_{t+1}(q_1, s_1, q_2, s_2, D)$ is as follows:

The likelihood that the PTM is at a state q at timestamp $t+1$ is the sum of the probabilities that succeeding branches from the previous timestamp transitioned to state q. To find the probability that the PTM is at state q in timestamp $t+1$, the likelihoods must be renormalized to sum to 1 (to account for non-succeeding branches).

3.3 Possible additions to initial program

3.3.1 Update Rule 1: SGD/backprop-like

To perform stochastic gradient descent, we must define a loss function and a gradient descent procedure over probabilistic Turing Machines.

We define the loss function over a training example as the negative log likelihood of successfully producing the output and the correct sequence of steps. To calculate gradients over time, we "unroll" the computation of the PTM, much like the the backpropagation procedure of an RNN.

Suppose that the probability of performing a successful transition at timestep t (conditional on success up through timestep $t-1$) is p_t. Our loss can then be written as

$$L = -\sum_i \log p_i$$

Given the definition of w in the above section, we derive the update rule. It turns out that, with the correct formulation, partial derivatives can be computed directly without the need for backpropagation of gradients. For a correct transition at time step t, the partial derivative is as follows:

$$\frac{\partial L}{\partial w(q,s,q',s',d)} = \sum_t \frac{Pr(\text{At state } q \text{ at time } t-1)Pr(\text{Transition to } q', \text{ output } s', \text{ move } d)}{p_t}$$

For incorrect transitions, the partial derivative is:

$$\frac{\partial L}{\partial w(q,s,q',s',d)} = -\sum_t \frac{Pr(\text{Correct action at time } t \;\&\&\; \text{previous state } q)Pr(\text{Transition to } q', \text{ output } s', \text{ move } d)}{p_t}$$

We find that the likelihood converges when using this update rules.

3.3.2 Forcing specialization with the loss function

When the PTM is trained with the above update rule, states generally specialize in the actions they output. For example, in the reverse task, the first state upon reading any input symbol outputs the correct action– move right, no output. The second state, upon reading any input symbol, also outputs the correct action– move left, output the same symbol as the input. However, state transitions are not always correct; in particular, the PTM ends up learning to simultaneously be at both states with equal probability at all times. To correct for this, we may wish to alter the loss function which we perform SGD with respect to:

$$L' = -\sum_i \log p_i + \lambda \sum_i (p_{i,0}^2 + p_{i,1}^2)$$

where $p_{i,q}$ is the probability that the PTM is at state q in timestep i. Including a function that is concave up with respect to state probabilities would encourage the PTM to stick to specific states, rather than trying to probabilistically be at every state. We are in the process of implementing this.

3.3.3 Update Rule 2

$$w_{t+1} \leftarrow \begin{cases} w_t(q_1,s_1,q_2,s_2,D) + \tau_t p f_t(q_1,s_1,q_2,s_2,D) & (s_2,D) \text{ is the correct action} \\ w_t(q_1,s_1,q_2,s_2,D) - \tau_t p f_t(q_1,s_1,q_2,s_2,D) & (s_2,D) \text{ is not the correct action} \end{cases}$$

The update rule given above rewards correct actions in proportion to the probability of their execution, conditional on the PTM succeeding at that timestep. Correct actions at any timestep are rewarded with a fixed reward; incorrect actions are punished with a fixed loss. Note that this does not exactly correspond to gradient descent on any natural loss functions, including negative log likelihood of successful computation; in fact, negative log likelihood of successful computation diverges. However, we found that the following modifications caused different and possibly viable results in training:

- Multiply the update expression by $p_{succeeding}$, which can also be tracked with a simple dynamic programming procedure alongside the process of running succeeding branches simultaneously. This forces the PTM to learn the steps of the procedure "in sequence", rather than all in parallel. In trials so far, this causes the PTM to learn early procedures in the algorithm at the expense of later procedures.
- Remove p (the probability of being in a given state), or replace it with $\sqrt{p}$ or something similar. One problem with the current training program is that the PTM can become stuck at local optima. It is unable to navigate away from these local optima or learn better transitions, since probabilities assigned to possibly better transitions are too small to allow for significant updates. In trials so far, this causes the PTM to be less likely to be stuck in local optima, though it also makes the PTM more uncertain on what transitions to actually reinforce.

When training with this update rule, or with the modifications, the PTM approaches a deterministic Turing Machine: transition probabilities approach 0 or 1, and the resulting machine often resembles the true/correct machine, with a few erroneous transitions.

3.3.4 Gradual discretization/transition locking

With this addition, transitions are gradually "locked" one at a time. Once the probability of the most likely transition from any particular (state, input symbol) combination exceeds some threshold in training, that transition is locked, and all non-locked transitions have their probabilities random reinitailized. This causes the PTM to learn transitions one-by-one; reinitialization in theory ought to ensure that every new learned transition is consistent with existing learned transitions, and that no conflicting transitions are learned simultaneously (which would push the PTM to a local optima). However, an error-correction mechanism is needed, so that the PTM may be able to go back and correct locked but bad transitions. In trials, we note that the PTM is "doomed" once it learns its first bad transition.

3.3.5 State-by-state learning

In this strategy, the PTM learns the procedure one state at a time. Initially, only one state is left "unlocked." The PTM trains until convergence with only this one state. Once transitions have converged, a second state is then "unlocked," and transitions involving the second state are now allowed. Once again, the PTM trains until convergence; the process repeats with remaining states. Coupled with the $p_{succeeding}$ addition to the update rule, this has the effect of forcing the PTM to learn algorithms one phase at a time. The motivation for this strategy comes from observations around the *reverse* task: with transition locking and the $p_{succeeding}$ update rule, the PTM learns the move-right phase of the algorithm with all of its states. By the time it has perfectly learned to move right until the end of the input, it has no states left to handle the move-left-and-copy phase of the algorithm. Too many states by this point have been allocated to the first phase. One important hyperparameter in this strategy is the amount of probability we initially assign transitions from learned states to newly-unlocked states. For instance, after training only state 0, we need to allow some initial probability for transitions to state 1 as soon as state 1 is unlocked; otherwise, state 1 will be ignored. However, probabilities that are too high will cause us to lose the structure learned and progress made with only state 0.

3.3.6 Locked transition perturbations

In this strategy, locked/high-confidence transitions in a PTM that has converged/reached a local optima are occasionally randomly reinitailized and retrained. In trials, this has sometimes allowed PTMs to reach less incorrect local optimas. The frequency of perturbation should change based on how successful the PTM is.

If a PTM is initialized with many more states than is necessary, with transition locking it should be able to learn arbitrary procedures with any of the above update rules. States with identical transition rules can then be "collapsed" into the same state once a correct PTM has been learned. However, from trials, it appears that the number of states required for this approach to work (without any of the other strategies) is far more than would be computationally feasible.

3.3.7 High-variance initialization

Currently, parameters are initialized by drawing from a normal distribution. Increasing the variance of the distribution allows us to endow the PTM with vague structure to begin with. It is unclear if this is actually useful.

3.3.8 Summary

It is likely thta a combination of methods will need to be used. In particular, coupling state-by-state learning with the $p_{succeeding}$ addition to the update rule seems particularly promising, as it may cause the PTM to learn algorithms in phases in a supervised fashion.

4 Weakly Supervised Training

A number of similar strategies can be adopted for unsupervised training, where the PTM is trained off of only input and output pairs.

- Replace the above update rule with something similar to the following:

$$w_{t+1} \leftarrow \begin{cases} w_t(q_1, s_1, q_2, s_2, D) + \tau_t f_t(q_1, s_1, q_2, s_2, D) & s_2 \text{ is the correct next symbol} \\ w_t(q_1, s_1, q_2, s_2, D) & s_2 = \epsilon \\ w_t(q_1, s_1, q_2, s_2, D) - \tau_t f_t(q_1, s_1, q_2, s_2, D) & s_2 \text{ is not the next symbol} \end{cases}$$

  Note the omission of p; since we cannot run all branches in parallel (the analog of "succeeding" branches may have different output tape states or different tape head locations at different times), we must choose single branches of computation to run probabilistically.

- Use a reinforcement learning method based off of Q-learning, perhaps similar to that of `http://jmlr.org/proceedings/papers/v48/zaremba16.pdf`

- Assign a constant reward for each correct output symbol, loss for incorrect outputs/early termination, and use a backpropagation-like procedures to update probabilities.

5 Tasks

To determine viability, we would like to test our model on the following simple tasks.

Copy: Copy a string. (Perfectly learned in the supervised case, with only 1 state)

Reverse: Print the reverse of a string. This is qualitatively harder than copy, walk, and unary addition, in that ir requires two distinct phases–move right, and move left and copy.

Walk: Follow a trail of direction-indicating symbols on a 2D input tape. (Perfectly learned in the supervised case, with only 1 state)

Unary addition: Perform the addition of two unary numbers.

Unary subtraction: Perform the subtraction of two unary numbers.

Addition: Perform the addition of two binary numbers.

Find-min: Find the minimum of a list of digits, and later possibly arbitrary numbers.

6 Discretization

From any PTM, we extract a corresponding deterministic Turing Machine by extracting a transition function as follows:

$$\delta(q_1, s_1) = \arg\max_{q_2, s_2, D} f(q_1, s_1, q_2, s_2, D)$$

7 Miscellaneous Notes

- Because the model is given only a read-only input tape, and a strictly constrained output tape (with no working memory tape), it is actually quite limited in the algorithms that it can learn. However, the supervised training experiments can still be performed to allow for a working memory tape, though unsupervised learning will be more difficult.