Noise reduction and targeted exploration in imitation learning for Abstract Meaning Representation parsing

James Goodman* Andreas Vlachos† Jason Naradowsky*
* Computer Science Department, University College London
james@janigo.co.uk, jason.narad@gmail.com
† Department of Computer Science, University of Sheffield
a.vlachos@sheffield.ac.uk

Abstract

Semantic parsers map natural language statements into meaning representations, and must abstract over syntactic phenomena, resolve anaphora, and identify word senses to eliminate ambiguous interpretations. Abstract meaning representation (AMR) is a recent example of one such semantic formalism which, similar to a dependency parse, utilizes a graph to represent relationships between concepts (Banarescu et al., 2013). As with dependency parsing, transition-based approaches are a common approach to this problem. However, when trained in the traditional manner these systems are susceptible to the accumulation of errors when they find undesirable states during greedy decoding. Imitation learning algorithms have been shown to help these systems recover from such errors. To effectively use these methods for AMR parsing we find it highly beneficial to introduce two novel extensions: noise reduction and targeted exploration. The former mitigates the noise in the feature representation, a result of the complexity of the task. The latter targets the exploration steps of imitation learning towards areas which are likely to provide the most information in the context of a large action-space. We achieve state-of-the-art results, and improve upon standard transition-based parsing by 4.7 $F_1$ points.

1 Introduction

Meaning representation languages and systems have been devised for specific domains, such as ATIS for air-travel bookings (Dahl et al., 1994) and database queries (Zelle and Mooney, 1996; Liang et al., 2013). Such machine-interpretable representations enable many applications relying on natural language understanding. The ambition of Abstract Meaning Representation (AMR) is that it is domain-independent and useful in a variety of applications (Banarescu et al., 2013).

The first AMR parser by Flanigan et al. (2014) used graph-based inference to find a highest-scoring maximum spanning connected acyclic graph. Later work by Wang et al. (2015b) was inspired by the similarity between the dependency parse of a sentence and its semantic AMR graph (Figure 1). Wang et al. (2015b) start from the dependency parse and learn a transition-based parser that converts it incrementally into an AMR graph using greedy decoding. An advantage of this approach is that the initial stage of dependency parsing is well-studied and trained using larger corpora than that for which AMR annotations exist.

Greedy decoding, where the parser builds the parse while maintaining only the best hypothesis at each step, has a well-documented disadvantage: error propagation (McDonald and Nivre, 2007). When the parser encounters states during parsing that are unlike those found during training, it is more likely to make mistakes, leading to states which are increasingly more foreign and causing errors to accumulate.
One way to ameliorate this problem is to employ imitation learning algorithms for structured prediction. Algorithms such as SEARN (Daumé III et al., 2009), DAGGER (Ross et al., 2011), and LOLS (Chang et al., 2015) address the problem of error propagation by iteratively adjusting the training data to increasingly expose the model to training instances it is likely to encounter during test. Such algorithms have been shown to improve performance in a variety of tasks including information extraction (Vlachos and Craven, 2011), dependency parsing (Goldberg and Nivre, 2013), and feature selection (He et al., 2013). In this work we build on the transition-based parsing approach of Wang et al. (2015b) and explore the applicability of different imitation algorithms to AMR parsing, which has a more complex output space than those considered previously.

The complexity of AMR parsing affects transition-based methods that rely on features to represent structure, since these often cannot capture the information necessary to predict the correct transition according to the gold standard. In other words, the features defined are not sufficient to “explain” why different actions should be preferred by the model. Such instances become noise during training, resulting in lower accuracy. To address this issue, we show that the α-bound Khardon and Wachman (2007), which drops consistently misclassified training instances, provides a simple and effective way of reducing noise and raising performance in perceptron-style classification training, and does so reliably across a range of parameter settings. This noise reduction is essential for imitation learning to gain traction in this task, and we gain 1.8 points of F1-Score using the DAGGER imitation learning algorithm.

DAGGER relies on an externally specified expert (oracle) to define the correct action in each state; this defines a simple 0-1 loss function for each action. Other imitation learning algorithms (such as LOLS, SEARN) and the variant of DAGGER proposed by Vlachos and Clark (2014) (henceforth V-DAGGER) can leverage a task level loss function that does not decompose over the actions taken to construct the AMR graph. However, these require extra computations to roll-out to an end-state AMR graph for each possible action not taken. The large action-space of our transition system makes these algorithms computationally infeasible, and roll-outs to an end-state for many of the possible actions will provide little additional information. Hence we modify the algorithms to target this exploration to actions where the classifier being trained is uncertain of the correct response, or disagrees with the expert. This provides a further gain of 2.7 F1 points.

This paper extends imitation learning to structured prediction tasks more complex than previously attempted. In the process, we review and compare recently proposed algorithms and show how their components can be recombined and adjusted to construct a variant appropriate to the task in hand. Hence we invest some effort reviewing these algorithms and their common elements.

Overall, we obtain a final F-Score of 0.70 on the newswire corpus of LDC2013E117 (Knight et al., 2014). This is identical to the score obtained by Wang et al. (2015a), the highest so far published. Our gain of 4.5 F1 points from imitation learning over standard transition-based parsing is orthogonal to that of Wang et al. (2015a) from additional trained classifiers, including co-reference and semantic role labelers, incorporated in the feature set. We further test on five other corpora of AMR graphs, including weblog domains, and show a consistent improvement in all cases with the application of imitation learning using DAGGER and the targeted V-DAGGER we propose here.

2 Transition-based AMR parsing

AMR parsing is an example of the wider family of structured prediction problems, in which we seek a mapping from an input x ∈ X to a structured output y ∈ Y. Here x is the dependency tree, and y the AMR graph; both are graphs and we notationally replace x with s1 and y with sT, with s1:T ∈ S. si are the intermediate graph configurations (states) that the system transitions through.

A transition-based parser starts with an input s1, and selects an action a1 ∈ A, using a classifier. ai converts si into si+1, i.e. si+1 = a_i(si). We term the set of states and actions ⟨s1, a1,...,aT−1, sT⟩ a trajectory of length T. The classifier π is trained to predict a_i from si, with \( \hat{\pi}(s) = \arg\max_{a_i \in A} w_{a_i} \cdot \Phi(s) \), assuming a linear classifier and a feature function \( \Phi(s) \).

We require an expert, \( \pi^* \), that can indicate what actions should be taken on each si to reach the target (gold) end state. In problems like POS-tagging these are directly inferable from gold, as the number of actions (T) equals the number of
Param. \( \beta \)

Outcome of action

Set label of edge \((\sigma_0, \beta_0)\) to \(l_e\). Pop \(\beta_0\).

Set concept of node \(\sigma_0\) to \(l_e\). Pop \(\sigma_0\), and initialise \(\beta\).

Make \(\beta_0\) parent of \(\sigma_0\) (reverse edge) and its sub-graph. Pop \(\beta_0\) and insert \(\beta_0\) as \(\sigma_1\).

Pop \(\sigma_0\) and delete it from the graph. Parents of \(\sigma_0\) become parents of \(\beta_0\). Other children of \(\sigma_0\) become children of \(\beta_0\). Insert \(\beta_0\) at the head of \(\sigma\) and re-initialise \(\beta\).

Pop \(\beta_0\) and delete edge \((\sigma_0, \beta_0)\). Attach \(\beta_0\) as a child of \(\kappa\). If \(\kappa\) has already been popped from \(\sigma\) then re-insert it as \(\sigma_1\).

Pop \(\sigma_0\) and delete it from the graph.

Insert a new node \(\delta\) with AMR concept \(l_e\) as the parent of \(\sigma_0\), and insert \(\delta\) into \(\sigma\).

Insert a new node \(\delta\) with AMR concept \(l_e\) as a child of \(\sigma_0\).

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Param.</th>
<th>Pre-conditions</th>
<th>Outcome of action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextEdge</td>
<td>(l_e)</td>
<td>(\beta) non-empty</td>
<td>Set label of edge ((\sigma_0, \beta_0)) to (l_e). Pop (\beta_0).</td>
</tr>
<tr>
<td>NextNode</td>
<td>(l_e)</td>
<td>(\beta) empty</td>
<td>Set concept of node (\sigma_0) to (l_e). Pop (\sigma_0), and initialise (\beta).</td>
</tr>
<tr>
<td>Swap</td>
<td>(\beta) non-empty</td>
<td>Make (\beta_0) parent of (\sigma_0) (reverse edge) and its sub-graph. Pop (\beta_0) and insert (\beta_0) as (\sigma_1).</td>
<td></td>
</tr>
<tr>
<td>ReplaceHead</td>
<td>(\beta) non-empty</td>
<td>Pop (\sigma_0) and delete it from the graph. Parents of (\sigma_0) become parents of (\beta_0). Other children of (\sigma_0) become children of (\beta_0). Insert (\beta_0) at the head of (\sigma) and re-initialise (\beta).</td>
<td></td>
</tr>
<tr>
<td>Reattach</td>
<td>(\kappa)</td>
<td>(\beta) non-empty</td>
<td>Pop (\beta_0) and delete edge ((\sigma_0, \beta_0)). Attach (\beta_0) as a child of (\kappa). If (\kappa) has already been popped from (\sigma) then re-insert it as (\sigma_1).</td>
</tr>
<tr>
<td>DeleteNode</td>
<td>(\beta) empty; leaf (\sigma_0)</td>
<td>Pop (\sigma_0) and delete it from the graph.</td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td>(l_e)</td>
<td></td>
<td>Insert a new node (\delta) with AMR concept (l_e) as the parent of (\sigma_0), and insert (\delta) into (\sigma).</td>
</tr>
<tr>
<td>InsertBelow</td>
<td></td>
<td></td>
<td>Insert a new node (\delta) with AMR concept (l_e) as a child of (\sigma_0).</td>
</tr>
</tbody>
</table>

Table 1: Action Space for the transition-based graph parsing algorithm

 Algorithm 1: Greedy transition-based parsing

Data: policy \(\pi\), start state \(s_1\)

Result: terminal state \(s_T\)

1 \(s_{\text{current}} \leftarrow s_1\); 
2 while \(s_{\text{current}}\) not terminal do 
3 \(a_{\text{next}} \leftarrow \pi(s_{\text{current}})\); 
4 \(s_{\text{current}} \leftarrow a_{\text{next}}(s_{\text{current}})\); 
5 \(s_T \leftarrow s_{\text{current}}\);

tokens with a 1:1 correspondence between them. In dependency parsing and AMR parsing this is not straightforward and dedicated transition systems are devised.

Given a labeled training dataset \(D\), algorithm 1 is first used to generate a trajectory for each of the inputs \((d \in D)\) with \(\pi = \pi^*\), the expert from which we wish to generalise. The data produced from all expert trajectories (i.e. \((s_i, d; a_i, d)\) for all \(i \in 1 \ldots T\) and all \(d \in 1 \ldots D\)), are used to train the classifier \(\hat{\pi}\), the learned classifier, using standard supervised learning techniques. Algorithm 1 is used to apply \(\hat{\pi}\) to unseen data. Our transition system (defining \(A, S\), and feature sets are based on Wang et al. (2015b), and are not the main focus of this paper. We introduce the key concepts here, with more details in the supplemental material.

We initialise the state with the stack of the nodes in the dependency tree, root node at the bottom. This stack is termed \(\sigma\). A second stack, \(\beta\) is initialised with all children of the top node in \(\sigma\). The state at any time is described by \(\sigma, \beta\), and the current graph (which starts as the dependency tree with one node per token). At any stage before termination some of the nodes will be labelled with words from the sentence, and others with AMR concepts. Each action manipulates the top nodes in each stack, \(\sigma_0\) and \(\beta_0\). We reach a terminal state when \(\sigma\) is empty. The objective function to maximise is the Smatch score (Cai and Knight, 2013), which calculates an F1-Score between the predicted and gold-target AMR graphs.

Table 1 summarises the actions in \(A\). NextNode and NextEdge form the core action set, labelling nodes and edges respectively without changing the graph structure. Swap, Reattach and ReplaceHead change graph structure, keeping it a tree. We permit a Reattach action to use parameter \(\kappa\) equal to any node within six edges from \(\sigma_0\), excluding any that would disconnect the graph or create a cycle.

The Insert/InsertBelow actions insert a new node as a parent/child of \(\sigma_0\). These actions are not used in Wang et al. (2015b), but Insert is very similar to the Infer action of Wang et al. (2015a). We do not use the Reentrance action of Wang et al. (2015b), as we found it not to add any benefit. This means that the output AMR is always a tree.

Our transition system has two characteristics which provide a particular challenge: given a sentence, the trajectory length \(T\) is theoretically unbounded; and \(|A|\) can be of the order \(10^4\) to \(10^4\). Commonly used transition-based systems have a fixed trajectory length \(T\), which often arises naturally from the nature of the problem. In PoS-tagging each token requires a single action, and in syntactic parsing the total size of the graph is limited to the number of tokens in the input. The lack of a bound in \(T\) here is due to Insert actions that can grow the graph, potentially \(ad infinitum\), and actions like Reattach, which can move a sub-graph repeatedly back-and-forth. The action space size is due to the size of the AMR vocabulary, which for relations (edge-labels) is restricted to about 100 possible values, but for concepts (node-labels) is almost as broad as an En-
The large action space and unbounded T also make beam search difficult to apply since it relies on a fixed length T with commensurability of actions at the same index on different search trajectories.

3 Imitation Learning for Structured Prediction

Imitation learning originated in robotics, training a robot to follow the actions of a human expert (Schaal, 1999; Silver et al., 2008). The robot moves from state to state via actions, generating a trajectory in the same manner as the transition-based parser of Algorithm 1.

In the imitation learning literature, the learning of a policy \( \hat{\pi} \) from just the expert generated trajectories is termed “exact imitation”. As discussed, it is prone to error propagation, which arises because the implicit assumption of i.i.d. inputs \((s_t)\) during training does not hold. The states in any trajectory are dependent on previous states, and on the policy used. A number of imitation learning algorithms have been proposed to mitigate error propagation, and share a common structure shown in Algorithm 2. Table 2 highlights some key differences between them.

The general algorithm firstly applies a policy \( \pi_{RollIn} \) (usually the expert, \( \pi^* \), to start) to the data instances to generate a set of ‘RollIn’ trajectories in line 6 (we adopt the terminology of ‘RollIn’ and ‘RollOut’ trajectories from Chang et al. (2015)). Secondly a number of ‘what if’ scenarios are considered, in which a different action \( a^*_t \) is taken from a given \( s_t \) instead of the actual \( a_t \) in the RollIn trajectory (line 8). Each of these exploratory actions generates a RollOut trajectory (line 10) to a terminal state, for which a loss \((L)\) is calculated using a loss function, \( F(s'_T) \), defined on the terminal states. For a number of different exploratory actions taken from a state \( s_t \) on a RollIn trajectory, the action cost (or relative loss) of each is calculated (line 13). Finally the generated \( \langle s_t, a^*_t, ActionCost_t^* \rangle \) data are used to train a classifier, using any cost-sensitive classification (CSC) method (line 15). New \( \pi_{RollIn} \) and \( \pi_{RollOut} \) are generated, and the process repeated over a number of iterations. In general the starting expert policy is progressively removed in each iteration, so that the training data moves closer and closer to the distribution encountered by just the trained classifier. This is required to reduce error propagation. For a general imitation learning algorithm we need to specify:

- the policy to generate the RollIn trajectory (the RollInPolicy);
- the policy to generate RollOut trajectories, including rules for interpolation of learned and expert policies (the RollOutPolicy);
- which one-step deviations to explore with a RollOut (the Explore function);
- how RollOut data are used in the classification learning algorithm to generate \( \hat{\pi}_t \) (within the Train function).

Exact Imitation can be considered a single iteration of this algorithm, with \( \pi_{RollIn} \) equal to the expert policy, and a 0-1 binary loss for \( F(0 \text{ loss for } \pi^*(s_t), \text{ the expert action, and a loss of } 1 \text{ for any other action}) \); all one-step deviations from the expert trajectory are considered without explicit RollOut to a terminal state.

In SEARN (Daume III et al., 2009), one of the first imitation learning algorithms in this framework, the \( \pi_{RollIn} \) and \( \pi_{RollOut} \) policies are identical within each iteration, and are a stochastic blend of the expert and all classifiers trained in previous iterations. The Explore function considers every possible one-step deviation from the RollIn trajectories, with a full RollOut to a terminal state.

---

**Algorithm 2: Generic Imitation Learning**

**Data:** data \( D \), expert \( \pi^* \), Loss function \( F(s) \)

**Result:** learned classifier \( C \), trained policy \( \hat{\pi} \)

1. Initialise \( C_0 \); for \( n = 1 \) to \( N \) do
2. Initialise \( E_n \) = \( \emptyset \);
3. \( \pi_{RollIn} = RollInPolicy(\pi^*, C_{0...n-1}, n) \);
4. \( \pi_{RollOut} = RollOutPolicy(\pi^*, C_{0...n-1}, n) \);
5. for \( d \in D \) do
6. Predict trajectory \( \hat{s}_{1:T} \) with \( \pi_{RollIn} \);
7. for \( s_t \in \hat{s}_{1:T} \) do
8. foreach \( a^*_t \in Explore(s_t, \pi^*, \pi_{RollIn}) \) do
9. \( \Phi_t^* = \Phi(d, a^*_t, \hat{s}_{1:t}) \);
10. Predict \( \hat{s}'_{t+1:T} \) with \( \pi_{RollOut} \);
11. \( L_t^* = F(s'_T) \);
12. foreach \( j \) do
13. \( ActionCost_t^* = L_t^* - \min_k L_t^k \)
14. Add \((\Phi_t, ActionCost_t^*)\) to \( E_n \);
15. \( \hat{\pi}_n, C_n = Train(C_{1...n-1}, E_1 \ldots E_n) \);
**Learned only**

All 1-step E

Explore

Deterministic

Learned only

Mixture E

Expert only

Stochastic

None. 0/1 expert loss

Mixture

Stochastic

None. 0/1 expert loss

Random

Mixture, step-level stochastic

Deterministic

Mixture, trajectory-level stoch.

Train

Deterministic

RollOut

Learned only

Mixture, trajectory-level stoch.

All 1-step

Deterministic

π

Each step is not possible. SCB-LOLS

vironment), so exhaustive RollOut exploration at

ditions (SCB) as a partial information case, the SCB

alternative actions as an

π

previously gathered data

π

ministic learned policy (Chang et al., 2015), with

π

chosen for all actions in the trajectory; either

π

∗ with probability β, or π

−1 otherwise. Both

LOLS and SEARN use an exhaustive search of

π

function. Chang et al. (2015) consider Structured Contextual Bandits (SCB) as a partial information case, the SCB modification of LOLS permits only one cost function call per RollIn (received from the external environment), so exhaustive RollOut exploration at each step is not possible. SCB-LOLS Explore picks a single step \( t \in \{1 \ldots T\} \) at random at which to make a random single-step deviation.

Another strand of work uses only the expert policy when calculating the action cost. Ross and Bagnell (2010) introduce SMILE, and later DAGGER (Ross et al., 2011). These do not RollOut as such, but as in exact imitation consider all one-step deviations from the RollIn trajectory and obtain a 0/1 action cost for each by asking the expert what it would do in that state. At the \( n \)th iteration the training trajectories are generated from an interpolation of \( \pi^* \) and \( \hat{\pi}_{n-1} \), with the latter progressively increasing in importance; \( \pi^* \) is used with probability \((1-\delta)^{n-1}\) for some decay rate \( \delta \). \( \hat{\pi}_n \) is trained using all \( E_{1 \ldots n} \). Ross et al. (2011) discuss and reject calculating an action cost by completing a RollOut from each one-step deviation to a terminal state. Three reasons given are:

1. Lack of real-world applicability, for example in robotic control.
2. Lack of knowledge of the final loss function, if we just have the expert’s actions.
3. Time spent calculating RollOuts and calling the expert.

Ross and Bagnell (2014) do incorporate RollOuts to calculate an action cost in their AGGREGATE algorithm. These RollOuts use the expert policy only, and allow a cost-sensitive classifier to be trained that can learn that some mistakes are more serious than others. As with DAGGER, the trained policy cannot become better than the expert.

V-DAGGER is the variant proposed by Vlachos and Clark (2014) in a semantic parsing task. It is the same as DAGGER, but with RollOuts using the same policy as RollIn. For both V-DAGGER and SEARN, the stochasticity of the RollOut means that a number of independent samples are taken for each one-step deviation to reduce the variance of the action cost, and noise in the training data. This noise reduction comes at the expense of the time needed to compute additional RollOuts.

### 4 Adapting imitation learning to AMR

Algorithms with full RollOuts have particular value in the absence of an optimal (or near-optimal) expert able to pick the best action from any state. If we have a suitable loss function, then the benefit of RollOuts may become worth the computation expended on them. For AMR parsing we have both a loss function in Smatch, and the ability to generate arbitrary RollOuts.

We therefore use a heuristic expert. This reduces the computational cost at the expense of not always predicting the best action. An expert needs an alignment between gold AMR nodes and tokens in the parse-tree or sentence to determine the actions to convert to one from the other. These alignments are not provided in the gold AMR, and our expert uses the AMR node to token alignments of JAMR (Flanigan et al., 2014). These alignments are not trained, but generated using regex and string matching rules. However, trajectories are in the range 50-200 actions for most training sentences, which combined with the size of \(|A|\) makes an exhaustive search of all one-step deviations expensive. Compare this to unlabeled shift-
reduce parsers with 4 actions, or POS tagging with |A| ∼ 30.

4.1 Targeted exploration

To reduce this cost we note that exploring RollOuts for all possible alternative actions can be uninformative when the learned and expert policies agree on an action and none of the other actions score highly with the learned policy. Extending this insight we modify the Explore function in Algorithm 2 to only consider the expert action, plus all actions scored by the current learned policy that are within a threshold τ of the score for the best rated action. In the first iteration, when there is no current learned policy, we pick a number of actions (usually 10) at random for exploration. Both SCB-LOLS and AGGREVAITE use partial exploration, but select the step t ∈ 1...T, and the action a_t at random. Here we optimise computational resources by directing the search to areas for which the trained policy is least sure of the optimal action, or disagrees with the expert.

Using imitation learning to address error propagation of transition-based parsing provides theoretical benefit from ensuring the distribution of s_t, a_t in the training data is consistent with the distribution on unseen test data. Using RollOuts that mix expert and learned policies additionally permits the learned policy to exceed the performance of a poor expert. Incorporating targeted exploration strategies in the Explore function makes this computationally feasible.

4.2 Noise Reduction

Different samples for a RollOut trajectory using v-DAGGER or SEARN can give very different terminal states s_T (the final AMR graph) from the same starting s_t and a_t due to the step-level stochasticity. The resultant high variance in the reward signal hinders effective learning. Daumé III et al. (2009) have a similar problem, and note that an approximate cost function outperforms single Monte Carlo sampling, “likely due to the noise induced following a single sample”.

To control noise we use the α-bound discussed by Khardon and Wachman (2007). This excludes a training example (i.e. an individual tuple s_i, a_i) from future training once it has been misclassified α times in training. We find that this simple idea avoids the need for multiple RollOut samples.

An attraction of LOLS is that it randomly selects either expert or learned policy for each RollOut, and then applies this consistently to the whole trajectory. Using LOLS should reduce noise without increasing the sample size. Unfortunately the unbounded T of our transition system leads to problems if we drop the expert from the RollIn or RollOut policy mix too quickly, with many trajectories never terminating. Ultimately π̂ learns to stop doing this, but even with targeted exploration training time is prohibitive and our LOLS experiments failed to provide results. We find that v-DAGGER with an α-bound works as a good compromise, keeping the expert involved in RollIn, and speeding up learning overall.

Another approach we try is a form of focused costing (Vlachos and Craven, 2011). Instead of using the learned policy for β% of steps in the RollOut, we use it for the first b steps, and then revert to the expert. This has several potential advantages: the heuristic expert is faster than scoring all possible actions; it focuses the impact of the exploratory step on immediate actions/effects so that mistakes π̂ makes on a distant part of the graph do not affect the action cost; it reduces noise for the same reason. We increase b in each iteration so that the expert is asymptotically removed from RollOuts, a function otherwise supported by the decay parameter, δ.

4.3 Transition System adaptations

Applying imitation learning to a transition system with unbounded T can and does cause problems in early iterations, with RollIn or RollOut trajectories failing to complete while the learned policy, π̂, is still relatively poor. To ensure every trajectory completes we add action constraints to the system. These avoid the most pathological scenarios, such as disallowing a Reattach of a previously Reattached sub-graph. These constraints are only needed in the first few iterations until π̂ learns, via the action costs, to avoid these scenarios. They are listed in the Supplemental Material. As a final fail-safe we insert a hard-stop on any trajectory once T > 300.

To address the size of |A|, we only consider a subset of AMR concepts when labelling a node. Wang et al. (2015b) use all concepts that occur in the training data in the same sentence as the lemma of the node, leading to hundreds or thousands of possible actions from some states. We use the smaller set of concepts that were assigned by the expert to the lemma of the current node any-
where in the training data. We obtain these assignments from an initial application of the expert to the full training data.

We add actions to use the actual word or lemma of the current node to increase generalisation, plus an action to append ‘-01’ to ‘verbify’ an unseen word. This is similar to the work of Werling et al. (2015) in word to AMR concept mapping, and is useful since 38% of the test AMR concepts do not exist in the training data (Flanigan et al., 2014).

Full details of the heuristics of the expert policy, features used and pre-processing are in Supplemental Material. All code is available at https://github.com/hopshackle/dagger-AMR.

### 4.4 Na"ıve Smatch as Loss Function

Smatch (Cai and Knight, 2013) uses heuristics to control the combinatorial explosion of possible mappings between the input and output graphs, but is still too computationally expensive to be calculated for every RollOut during training. We retain Smatch for reporting all final results, but use ‘Na"ıve Smatch’ as an approximation during training. This skips the combinatorial mapping of nodes between predicted and target AMR graphs. Instead, for each graph we compile a list of:

- Node labels, e.g. name
- Node-Edge-Node label concatenations, e.g.
  
  leave-01:ARG0:room
- Node-Edge label concatenations, e.g.
  
  leave-01:ARG0

The loss is the number of entries that appear in only one of the lists. We do not convert to an $F_1$ score, as retaining the absolute number of mistakes is proportional to the size of the graph.

The flexibility of the transition system means multiple different actions from a given state $s_i$ can lead, via different RollOut trajectories, to the same target $s_T$. This can result in many actions having the best action cost, reducing the signal in the training data and giving poor learning. To encourage short trajectories we break these ties with a penalty of $T/5$ to Na"ıve Smatch. Multiple routes of the same length still exist, and are preferred equally. Note that the ordering of the stack of dependency tree nodes in the transition system means we start at leaf nodes and move up the tree. This prevents sub-components of the output AMR graph being produced in an arbitrary order.

### 5 Experiments

The main dataset used is the newswire (proxy) section of LDC2014T12 (Knight et al., 2014). The data from years 1995-2006 form the training data, with 2007 as the validation set and 2008 as the test set. The data split is the same as that used by Flanigan et al. (2014) and Wang et al. (2015b).

We first assess the impact of noise reduction using the alpha bound, and report these experiments without Rollouts (i.e. using DAGGER) to isolate the effect of noise reduction. Table 3 summarises results using exact imitation and DAGGER with the $\alpha$-bound set to discard a training instance after one misclassification. This is the most extreme setting, and the one that gave best results. We try AROW (Crammer et al., 2013), Passive-Aggressive (PA) (Crammer et al., 2006), and perceptron (Collins, 2002) classifiers, with averaging in all cases. We see a benefit from the $\alpha$-bound for exact imitation only with AROW, which is more noise-sensitive than PA or the simple perceptron. With DAGGER there is a benefit for all classifiers. In all cases the $\alpha$-bound and DAGGER are synergistic; without the $\alpha$-bound imitation learning works less well, if at all. $\alpha=1$ was the optimal setting, with lesser benefit observed for larger values.

We now turn our attention to targeted exploration and focused costing, for which we use V-DAGGER as explained in section 4. For all V-

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Exact Imitation</th>
<th>Imitation Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\alpha=0$</td>
<td>$\alpha=1$</td>
</tr>
<tr>
<td>AROW, C=10</td>
<td>65.5</td>
<td>66.8</td>
</tr>
<tr>
<td>AROW, C=100</td>
<td>66.4</td>
<td>66.6</td>
</tr>
<tr>
<td>AROW, C=1000</td>
<td>66.4</td>
<td>67.0</td>
</tr>
<tr>
<td>PA, C=100</td>
<td>66.7</td>
<td>66.5</td>
</tr>
<tr>
<td>Perceptron</td>
<td>65.5</td>
<td>65.3</td>
</tr>
</tbody>
</table>

Table 3: DAGGER with $\alpha$-bound. All figures are F-Scores on the validation set. 5 iterations of classifier training take place after each DAgger iteration. A decay rate ($\delta$) for $\pi^*$ of 0.3 was used.

1Formally Flanigan et al. (2014; Wang et al. (2015b) use the pre-release version of this dataset (LDC2013E117). Werling et al. (2015) conducted comparative tests on the two versions, and found only a very minor changes of 0.1 to 0.2 points of F-score when using the final release.
DAGGER experiments we use AROW with regularisation parameter C=1000, and δ=0.3.

Figure 2 shows results by iteration of reducing the number of RollOuts explored. Only the expert action, plus actions that score close to the best-scoring action (defined by the threshold) are used for RollOuts. Using the action cost information from RollOuts does surpass simple DAGGER, and unsurprisingly more exploration is better.

Figure 3 shows the same data, but by total computational time spent\(^2\). This adjusts the picture, as small amounts of exploration give a faster benefit, albeit not always reaching the same peak performance. As a baseline, three iterations of V-DAGGER without targeted exploration (threshold \(=\infty\)) takes 9600 minutes on the same hardware to give an F-Score of 0.652 on the validation set.

Figure 4 shows the improvement using focused costing. The ‘n/m’ setting sets \(b\), the number of initial actions taken by \(\hat{\pi}\) in a RollOut to \(n\), and then increases this by \(m\) at each iteration. We gain an increase of 2.9 points from 0.682 to 0.711. In all the settings tried, focused costing improves the results, and requires progressive removal of the expert to achieve the best score.

We use the classifier from the Focused Costing 5/5 run to achieve an F-Score on the held-out test set of 0.70, equal to the best published result so far (Wang et al., 2015a). Our gain of 4.7 points from imitation learning over standard transition-based parsing is orthogonal to that of Wang et al. (2015a) using exact imitation with additional trained analysers; they experience a gain of 2 points from using a Charniak parser (Charniak and Johnson, 2005) trained on the full OntoNotes corpus instead of the Stanford parser used here and in Wang et al. (2015b), and a further gain of 2 points from a semantic role labeller. Table 4 lists previous AMR work on the same dataset.

\(^2\)experiments were run on 8-core Google Cloud n1-highmem-8 machines.

### Table 4: Comparison of previous work on the AMR task. R, P and F are Recall, Precision and F-Score.

<table>
<thead>
<tr>
<th>Authors</th>
<th>Algorithmic Approach</th>
<th>R</th>
<th>P</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flanigan et al. (2014)</td>
<td>Concept identification with semi-markov model followed by optimisation of constrained graph that contains all of these.</td>
<td>0.52</td>
<td>0.66</td>
<td>0.58</td>
</tr>
<tr>
<td>Werling et al. (2015)</td>
<td>As Flanigan et al. (2014), with enhanced concept identification</td>
<td>0.59</td>
<td>0.66</td>
<td>0.62</td>
</tr>
<tr>
<td>Wang et al. (2015b)</td>
<td>Single stage using transition-based parsing algorithm</td>
<td>0.62</td>
<td>0.64</td>
<td>0.63</td>
</tr>
<tr>
<td>Pust et al. (2015)</td>
<td>Single stage System-Based Machine Translation</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Peng et al. (2015)</td>
<td>Hyperedge replacement grammar</td>
<td>0.57</td>
<td>0.59</td>
<td>0.58</td>
</tr>
<tr>
<td>Artzi et al. (2015)</td>
<td>Combinatory Categorial Grammar induction</td>
<td>0.66</td>
<td>0.67</td>
<td>0.66</td>
</tr>
<tr>
<td>Wang et al. (2015a)</td>
<td>Extensions to action space and features in Wang et al. (2015b)</td>
<td>0.69</td>
<td>0.71</td>
<td>0.70</td>
</tr>
<tr>
<td>This work</td>
<td>Imitation Learning with transition-based parsing</td>
<td>0.68</td>
<td>0.73</td>
<td>0.70</td>
</tr>
</tbody>
</table>

### Table 5: Comparison of Exact Imitation (EI), DAGGER (D), V-DAGGER (V-D) on all components of the LDC2014T12 corpus.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Validation F-Score</th>
<th>Test F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EI</td>
<td>D</td>
</tr>
<tr>
<td>proxy</td>
<td>0.670</td>
<td>0.686</td>
</tr>
<tr>
<td>dfa</td>
<td>0.495</td>
<td>0.532</td>
</tr>
<tr>
<td>bolt</td>
<td>0.456</td>
<td>0.468</td>
</tr>
<tr>
<td>xinhua</td>
<td>0.398</td>
<td>0.623</td>
</tr>
<tr>
<td>lpp</td>
<td>0.340</td>
<td>0.346</td>
</tr>
</tbody>
</table>

Table 5: Comparison of Exact Imitation (EI), DAGGER (D), V-DAGGER (V-D) on all components of the LDC2014T12 corpus.

Using DAGGER with this system we obtained an F-Score of 0.60 in the Semeval 2016 task on AMR parsing, one standard deviation above the mean of all entries, (Goodman et al., 2016)

Finally we test on all components of the LDC2014T12 corpus as shown in Table 5, which include both newswire and weblog data, as well as the freely available AMRs for *The Little Prince*, (lpp)\(^3\). For each we use exact imitation, DAGGER, and V-DAGGER on the train/validation/splits specified in the corpus. In all cases, imitation learning without RollOuts (DAGGER) improves on exact imitation, and incorporating RollOuts (V-DAGGER) provides an additional benefit. Rao et al. (2015) use SEARN on the same datasets, but with a very different transition system. We show their results for comparison.

Our expert achieves a Smatch F-Score of 0.94 on the training data. This explains why DAGGER, which assumes a good expert, is effective. Introducing RollOuts provides additional theoretical benefits from a non-decomposable loss function that can take into account longer-term impacts of an action. This provides much more information than the 0/1 binary action cost in DAGGER, and we can use Naïve Smatch as an approximation to our actual objective function during training. This informational benefit comes at the cost of increased noise and computational expense, which we control with targeted exploration and focused

---

\(^3\)http://amr.isi.edu/download.html
costing. We gain 2.7 points in F-Score, at the cost of 80-100x more computation. In problems with a less good expert, the gain from exploration could be much greater. Similarly, if designing an expert for a task is time-consuming, then it may be a better investment to rely on exploration with a poor expert to achieve the same result.

6 Related Work

Other strategies have been used to mitigate the error propagation problem in transition-based parsing. A common approach is to use beam search through state-space for each action choice to find a better approximation of the long-term score of the action, e.g. Zhang and Clark (2008). Goldberg and Elhadad (2010) remove the determinism of the sequence of actions to create easy-first parsers, which postpone uncertain, error-prone decisions until more information is available. This contrasts with working inflexibly left-to-right along a sentence, or bottom-to-top up a tree.

Goldberg and Nivre (2012) introduce dynamic experts that are complete in that they will respond from any state, not just those on the perfect trajectory assuming no earlier mistakes; any expert used with an imitation learning algorithm needs to be complete in this sense. Their algorithm takes exploratory steps off the expert trajectory to augment the training data collected in a fashion very similar to DAGGER.

Honnibal et al. (2013) use a non-monotonic parser that allows actions that are inconsistent with previous actions. When such an action is taken it amends the results of previous actions to ensure post-hoc consistency. Our parser is non-monotonic, and we have the same problem encountered by Honnibal et al. (2013) with many different actions from a state $s_i$ able to reach the target $s_T$, following different “paths up the mountain”. This leads to poor learning. To resolve this with fixed $T$ they break ties with a monotonic parser, so that actions that do not require later correction are scored higher in the training data. In our variable $T$ environment, adding a penalty to the size of $T$ is sufficient (section 4.4).

Vlachos and Clark (2014) use v-DAGGER to give a benefit of 4.8 points of F-Score in a domain-specific semantic parsing problem similar to AMR. Their expert is sub-optimal, with no information on alignment between words in the input sentence, and nodes in the target graph. The parser learns to link words in the input to one of the 35 node types, with the ‘expert’ policy aligning completely at random. This is infeasible with AMR parsing due to the much larger vocabulary.

7 Conclusions

Imitation learning provides a total benefit of 4.5 points with our AMR transition-based parser over exact imitation. This is a more complex task than many previous applications of imitation learning, and we found that noise reduction was an essential pre-requisite. Using a simple 0/1 binary action cost using a heuristic expert provided a benefit of 1.8, with the remaining 2.7 points coming from RollOuts with targeted exploration, focused costing and a non-decomposable loss function that was a better approximation to our objective.

We have considered imitation learning algorithms as a toolbox that can be tailored to fit the characteristics of the task. An unbounded $T$ meant that the LOLS RollIn was not ideal, but this could be modified to slow the loss of influence of the expert policy. We anticipate the approaches that we have found useful in the case of AMR to reduce the impact of noise, efficiently support large action spaces with targeted exploration, and cope with unbounded trajectories in the transition system will be of relevance to other structured prediction tasks.
Acknowledgments

Andreas Vlachos is supported by the EPSRC grant Diligent (EP/M005429/1) and Jason Naradowsky by a Google Focused Research award. We would also like to thank our anonymous reviewers for many comments that helped improve this paper.

References


