Learning to Repair: Repairing model output errors after deployment using a dynamic memory of feedback

Anonymous ACL submission

Abstract

Large language models (LMs), while powerful, are not immune to mistakes, but can be difficult to retrain. Our goal is for an LM to continue to improve after deployment, without retraining, using feedback from the user. Our approach pairs an LM with (i) a growing memory of cases where the user identified an output error and provided general feedback on how to correct it (ii) a *corrector model*, trained to translate this general feedback into specific edits to repair the model output. Given a new, unseen input, our model can then use feedback from similar, past cases to repair output errors that may occur. We instantiate our approach using an existing, fixed model for *script generation*, that takes a goal (e.g., "bake a cake") and generates a partially ordered sequence of actions to achieve that goal, sometimes containing errors. We show that our memory-enhanced system, FBNET, learns to apply user feedback effectively to repair such errors (up to 30 points improvement), while making a start at avoiding similar past mistakes on new, unseen examples (up to 7 points improvement in a controlled setting). This is a first step towards strengthening deployed models, potentially broadening their utility.[1]

1 Introduction

While language models (LMs) have achieved remarkable success for many tasks (Wang et al., 2018; Talmor et al., 2019), they are still prone to mistakes (Bender and Koller, 2020). Correcting such mistakes by retraining is not always easy due to the cost and/or unpredictability of how additional training data will change the model. Instead, our goal is to allow users to correct such errors directly through interaction, without retraining – by giving corrective feedback on the model's output. Our approach is to maintain a growing, dynamic memory of such feedback, and use a trained *corrector model* to apply such feedback to repair the model output. By doing so, the system can also potentially fix output errors for new unseen inputs using feedback from similar, past cases.

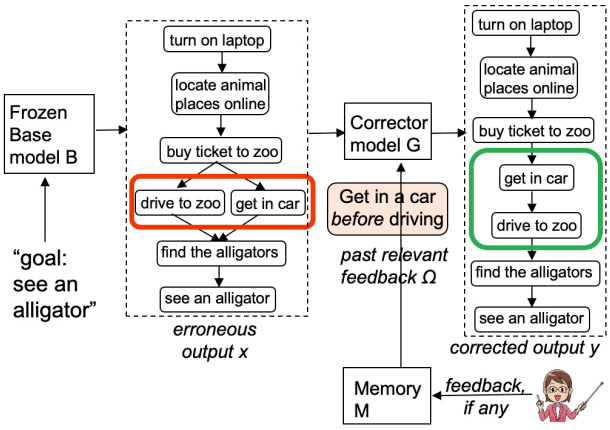

Figure 1: Given a frozen model **B**, we train a *corrector model* **G** to apply feedback from a user about errors made by the original model. In the example, **B** has generated a script with an error in, stating that "driving" and "getting in a car" can occur in any order (red box). The user provides general feedback ("Get in a car before driving"), and **G** operationalizes this to generate a corrected graph (by predicting and applying a graph edit operation) in which "get in car" happens first (green box). The feedback is stored in a memory $\mathcal{M}$ so it can also be retrieved to repair similar, future errors.

We consider the class of problems where the model's output is *repairable*, namely a structured output that is (typically) nearly correct, and fixable through a small number of edit operations. Specifically, we apply our approach to the task of *script generation*, using an existing, fixed model called proScript (Sakaguchi et al., 2021) that satisfies this constraint. proScript takes as input a goal to achieve (expressed in natural language), and outputs a partially ordered sequence of steps - a *script* - required to achieve that goal. Our interest here is not in proScript itself, but what to do when proScript's output contains an error.

[1] Our code and data is available at https://anonymous.4open.science/r/interscript

This instantiation of our approach is illustrated in Figure 1. Here, proScript has generated a script $\mathbf{x}$ to achieve the goal "see an alligator", but the script contains an error: it states that the steps of "driving to the zoo" and "get in car" can be applied in any order. To repair this, the user provides the general feedback "Get in a car before driving". The corrector model $\mathbf{G}$ then takes that feedback and the erroneous script, translates it into appropriate edit operations on the script, and applies those edits to generate a corrected script ($\mathbf{y}$ in Figure 1). The feedback is stored in memory $\mathcal{M}$ so it can also be retrieved in the future. Our system, FBNET, comprises of the corrector module $\mathbf{G}$, the memory $\mathcal{M}$, and searching and writing operations. To train our system, we collect examples of bad outputs, general feedback, and specific edits that the feedback should translate to (Section 4.2). This allows $\mathbf{G}$ to learn how to translate general feedback into specific edits to apply. Further, pairing $\mathbf{G}$ with the memory $\mathcal{M}$ allows FBNET to repair new, unseen scripts containing similar errors to the one the user corrected.

Our approach loosely follows some early AI systems that maintained a memory of the output problems and how to fix them (Sussman, 1973; Hammond, 1986; Riesbeck, 1981), but here, we use neural methods and interact with a user to provide corrective feedback. It also builds on the idea of allowing users to specify edits in natural language, e.g., NLEdit (Elgohary et al., 2021), except we use *general* user feedback (then translated to example-specific edits by $\mathbf{G}$) and add a memory so that feedback can also be automatically reused.

We evaluate FBNET along two dimensions: (a) How well does FBNET interpret NL feedback? (b) How well can FBNET learn from prior mistakes? We find that (a) FBNET can use NL feedback effectively to repair script errors, with +30% (absolute) improvement over a baseline that does not use feedback, and that (b) the system makes a start at avoiding past mistakes (+7% (absolute) improvement in a controlled setting). Although these results are only for a single deployment of our general approach, they suggest that memory-based architectures can help deployed models continue to improve with time, without retraining, potentially broadening their utility.

2 Related work

There have been numerous approaches to using user feedback to improve model performance, including:
(1) **Providing additional training examples:** Dasgupta et al. (2019) show how a user can correct bad model behavior by carefully selecting new training examples for the system to learn from, a style of interactive active learning (Settles, 2012).
(2) **Marking/scoring the system's answer(s):** In SHRDLURN, the user provides feedback by identifying which of the system's alternative interpretations of a user command is correct (Wang et al., 2016).
(3) **Providing hints:** (Mehta and Goldwasser, 2019) show how a system can learns to understand regional (e.g., "top left") and directional (e.g., "move down") hints from the user for a (simulated) robot.
(4) **Provide additional information:** In TeachYourAI (Talmor et al., 2020), given a wrong answer to a question, users can enter NL facts and rules to use as context when reasking the question, to (ideally) produce the correct answer.
(5) **Correcting bad answers:** In the semantic parsing task of NL-to-SQL, NLEdit learns to interpret and apply syntactic edit operations from the user expressed in NL, e.g., "replace course id with program id." (Elgohary et al., 2021).

These methods all augment/replace the standard use of automated answer feedback (if available), e.g., testing whether a semantic parse correctly executes to the correct answer, e.g., (Zettlemoyer and Collins, 2005), sometimes using unsupervised techniques to generate additional training data, e.g., BIFI (Yasunaga and Liang, 2021).

Our work expands on the above approaches in two important ways. First, users provide *general* feedback in NL, that can potentially be applied to *multiple* cases (rather than just correcting a specific instance). The corrector model $\mathbf{G}$ is trained to operationalize that advice in different ways for different examples appropriately, in contrast to (say) NLEdit where the user-provided specific corrective edits for a single example only.

Second, we use a feedback memory, allowing feedback to be reused. While adding external memory to neural systems is not new, e.g., RAG (Lewis et al., 2020), REALM (Guu et al., 2020), ours is the first to utilize a memory of prior user feedback

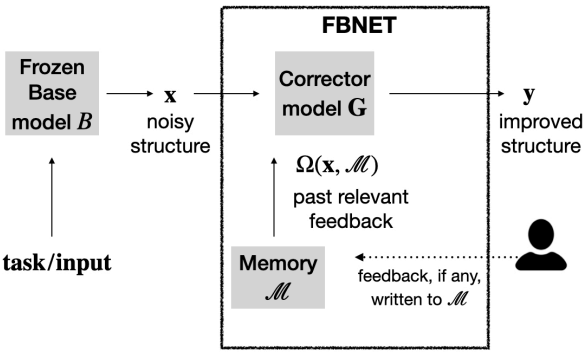

Figure 2: Proposed architecture: (left) $\mathbf{B}$ does not account for user feedback. (right) FBNET maintains a memory $\mathcal{M}$ of corrective feedback, and searches for feedback from prior queries with similar error intent as $\mathbf{x}$ using a retrieval function Ω. $\mathbf{x}$ is then concatenated to the retrieved feedback to form the input to the corrector model $\mathbf{G}$. Users can also give new feedback which is added to $\mathcal{M}$.

to improve future neural model performance. This can be viewed as a modern approach to failure-driven reminding, an essential theme in earlier AI and Cognitive Science research (Riesbeck, 1981; Schank and Leake, 1989; Ross, 1984).

3 FBNET

3.1 Overview of the Architecture

Fig. 2 gives an overview of FBNET. The input is a potentially noisy graph $\mathbf{x}$ generated by a base model $\mathbf{B}$ and the output $\mathbf{y}$ is a corrected graph. At inference time, i.e., after deployment, a user can critique $\mathbf{y}$ by providing natural language feedback $\mathbf{fb}$ on an error $\mathbf{e}$. As output, the model generates the corrected graph $\mathbf{y}$ that accounts for $\mathbf{fb}$.

The corrector model $\mathbf{G}$ is responsible for improving the potentially noisy output from $\mathbf{B}$. $\mathbf{G}$ achieves it using user feedback stored in a continuously updated memory $\mathcal{M}$.

The Memory $\mathcal{M}$ is a growing lookup table of key-value pairs: key ($\mathbf{x}_i$) - value ($\mathbf{fb}_i$), where $\mathbf{x}_i$ is a particular incorrect graph, and $\mathbf{fb}_i$ is the corresponding feedback. This memory supports lookup (read) and write operations. Given a new query $\mathbf{x}$, FBNET uses feedback $\mathbf{fb}$ from similar, prior queries in the memory to enrich $\mathbf{x}$. This feedback $\mathbf{fb}$ is retrieved using the lookup function $\Omega(\mathbf{x}, \mathcal{M})$. The corrector then combines $\mathbf{fb}$ with $\mathbf{x}$, and generates $\mathbf{y}$. The write operation is used whenever a user gives new feedback.

3.2 Assumptions

We make two assumptions on the characteristics of the feedback and the input.

A1. Base model $\mathbf{B}$'s output is **repairable**: $\mathbf{B}$ typically produces syntactically correct output graph but can have semantic errors that the user can recognize and describe using natural feedback.

A2. Feedback is **reusable**: If two examples i, j have similar errors e_i and e_j then the feedback fb for one should apply to the other, i.e., $(e_i \sim e_j \Leftrightarrow fb_i \sim fb_j)$

3.3 Memory $\mathcal{M}$ and Ω

As mentioned, the feedback is stored in a memory of key ($\mathbf{x}$), value ($\mathbf{fb}$) pairs. Ω is a retrieval function that matches a query key ($\mathbf{x}_j$) to a similar $\mathbf{x}_i$ in memory implicitly on the similarity of the errors e_i and e_j.

3.4 Corrector model G

The graph corrector model $\mathbf{G}$ generates an improved output $\mathbf{y}$ given a noisy graph $\mathbf{x}$ and $\mathbf{fb}$. This is done in a two-step process, (i) learning to predict a graph edit operation $\mathbf{y}^e$ given $\mathbf{x}$ and $\mathbf{fb}$ (ii) using simple graph operations to apply $\mathbf{y}^e$ to $\mathbf{x}$ to produce $\mathbf{y}$. Our approach of generating an edit instead of directly generating the corrected graph is beneficial for two reasons. First, generating edits is simpler for the model than generating entire graphs. Second, it simplifies evaluation metrics as it is much simpler to compare two smaller generated edits. Note that given an edit we can deterministically fix a script, and thus the two-step process helps us achieve the same end goal (corrected scripts from noisy scripts and feedback).

3.5 Training and Inference

As mentioned, the graph corrector $\mathbf{G}$ first generates an edit $\mathbf{y}^e$, which is applied to the incorrect graph $\mathbf{x}$ to generate the correct graph $\mathbf{y}$. We need a corpus of $(\mathbf{x}, \mathbf{fb}, \mathbf{y})$ to train this system. Specifically, we extract an edit from each such tuple, where edit y^e is the difference between the output $\mathbf{y}$ and the input $\mathbf{x}$. $\mathbf{x}$ and $\mathbf{y}$ can be expressed in a string representation using a graph description language such as DOT. We then train a language model to estimate $P_\theta(y^e \mid \mathbf{x}, \mathbf{fb})$, which allows us to generate an edit for a given $(\mathbf{x}, \mathbf{fb})$ using greedy sampling, where θ denotes the parameters of the language model.

4 Application: Script Generation

4.1 Task

We instantiate our framework for the task of *script generation*. Formally, the script generation task (Sakaguchi et al., 2021) takes as input a scenario and generates a script $G(V, E)$, where V is a set of essential events $\{v_1, ...v_i, ...v_{|V|}\}$ and E is a set of temporal ordering constraints between events $\{e_{ij}\}$ which means that the events v_i must precede the event v_j ($v_i \prec v_j$). Partial ordering of events is possible, e.g., you can wear a left sock and a right sock in any temporal order. To solve this task, script generation models are required to *generate* events (V) and predict the edges (E) jointly. See Figure 3 for an example.

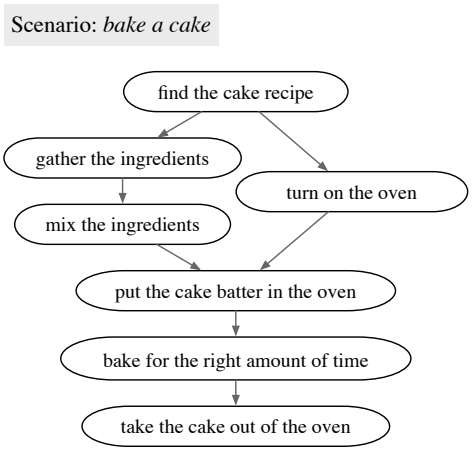

Figure 3: An example of a *script* in Sakaguchi et al. (2021). In a script generation task, models take the goal as the input and generate a (possibly) partial-order graph, which consists of essential steps and their ordering.

PROSCRIPT$_{gen}$ (Sakaguchi et al., 2021) is a recently released model that, given a goal, generates V and predicts the edge structure E jointly. It is based on the T5-XXL model (11B parameters) and generates the script as a graph in DOT format. The authors report that the DOT format is always valid at inference time and that V and the graph structure are generally of high quality. They characterize the graph edits required to correct a generated script (such as removing a node, adding a node, changing edge order, etc.). Mechanical Turk workers could repair most of the generated scripts within a few edits (typically an edit distance of 5) - we further validate this in Appendix §8.1. This makes for an attractive use-case for interactive learning because the generated content from the model is repairable through user feedback.

4.2 Feedback Data Collection

To train the corrector G, as well as evaluate our approach, we collected a set of (x, fb, y) tuples using crowdworkers, where x is a possibly erroneous script generated by PROSCRIPT$_{gen}$, fb is general feedback about the error (if any), and y is the corrected script. In practice, crowdworkers specified the edits to x to create y (using simple graph operations we can generate $\mathbf{y}$ from $\mathbf{y}^e$– see Table 7 for an example). We collected 1542 tuples of data, randomly splitting it into 843 train, 154 validation, and 545 test points. Examples of the resulting dataset are shown in Table 1.

4.3 Training the Corrector Model

We initialize θ with a checkpoint from the text-to-text pre-trained T5 transformer (Raffel et al., 2020) and fine-tune on our dataset. We use the default hyperparameters (including the Adafactor optimizer) in the T5 library.[2] We fine-tune a T5-XXL model for the main results, fine-tuned for 5,000 steps (batch size 8), selecting the checkpoint with the highest validation score (usually the final step). To implement the memory $\mathcal{M}$, we use a BERT-based Sentence Transformer to encode x (Reimers and Gurevych, 2019), and use cosine distance with a threshold of 0.9 to find a matching key $\mathbf{x}_m$. We leave the investigation of more complex retrieval functions (e.g., using attention mechanism to future work.)

5 Experiments

We empirically evaluate two questions:

RQ1. **How well does FBNET interpret NL feedback?** Specifically, we measure how well FBNET can translate general feedback **fb** from a user into the correct repair edit on an imperfect script x. The main focus of RQ1 is to test the performance of **G** in the pipeline (Fig. 2)

RQ2. **How well can FBNET learn from prior mistakes?** We make the same measurement, but using feedback **fb** recalled from similar, previous examples. The main focus of RQ2 is to test the performance of $\mathcal{M}$ and Ω.

Metrics To compare the gold edit $\mathbf{y}^{e^*}$ and the generated edit $\mathbf{y}^{\hat{e}}$, we use standard metrics used to evaluate generated text. We report the following metrics:

[2] https://github.com/google-research/text-to-text-transfer-transformer

Error type	Input script x	Feedback fb	Expected edit $\mathbf{y}^{e^*}$	Generated edit $\mathbf{y}^{\hat{e}}$	score		
					EM	EM_{type}	EM_{loc}
missing step	1. get out of car 2. stop in front of car 3. turn body toward back of car 4. walk to back of car 5. take blanket out of car 6. walk to desired location 7. throw blanket down	a person needs to open the door before they take an object out	insert node 'open the back door of the car' before 'take blanket out of car'	insert node 'open car door' before 'take blanket out of car'	0	1	1
missing step	1. buy a video game 2. talk to the cashier 3. make the transaction 4. get the receipt 5. load video game into the car 6. get into the car 7. take xbox home	after a person makes a transaction, they then head to their car	insert node 'walk to the car' after 'get the receipt'	insert node 'get into the car' after 'make the transaction'	0	1	0
wrong step	1. make a bunch of cards 2. grab a pen 3. grab some paper 4. pick up a pen 5. place the paper on the table 6. pick up the pen 7. write names on the cards	good plans shouldn't include redundant steps	remove node 'pick up the pen'	remove node 'pick up the pen'	1	1	1
wrong order	1. leave home and get in car 2. remem. destination address 3. look around for the car 4. walk towards the car 5. open the car door 6. sit down in the car 7. put on the seatbelt	you wouldn't look for something you're already with	reorder edge between '⟨ leave home and get in car , look around for the car ⟩'	remove node 'look around for the car'	0	0	0

Table 1: Some examples of the data points and model predictions. $\mathbf{y}^e$ takes the form: `<EDIT TYPE>` over `[<ARG>]` at `<LOCATION>` The dataset contains partial order points as well, but they are omitted here for simplicity.

- **Exact match**: EM gives a score of 1 if $\mathbf{y}^{e^*}$ is equal to $\mathbf{y}^{\hat{e}}$ and 0 otherwise.
- **Generation metrics**: We report standard generation metrics BLEU (Papineni et al., 2002) and ROUGE (Lin, 2004) to account for similar but not exact matches. We use the implementation released in the metrics package of the GEM-benchmark (Gehrmann et al., 2021).[3]

We report these metrics over the entire edit: EM, BLEU, ROUGE. The components of $\mathbf{y}^e$ broadly follow a template: `<EDIT TYPE> over [ARG] at <LOCATION>` (see Table 1). This allows comparison of the location or edit type in $\mathbf{y}^{e^*}$ and $\mathbf{y}^{\hat{e}}$: EM $_{loc}$, BLEU $_{loc}$, ROUGE $_{loc}$ and EM $_{type}$, BLEU $_{type}$, ROUGE $_{type}$

[3] https://github.com/GEM-benchmark/GEM-metrics/

Baseline As baseline, we train a model that does not use any feedback (we call this, NO-FB) and is trained only with *input* = *erroneous script* and *output* = *edit*. The language model used in this baseline and FBNET is the same (T5-XXL), allowing a meaningful comparison.

5.1 RQ1: How well does FBNET interpret NL feedback?

To measure how well the graph corrector **G** learns to interpret NL feedback, we provide oracle feedback to FBNET, and we call this FBNET$_o$. Table 2 shows that FBNET$_o$ learns to react to the feedback. Further, we note that the model is good at identifying the error type that the feedback indicates. Still, it is difficult for the model to localize the error in the graph, probably because the location is not explicitly mentioned in the feedback, and the model needs to infer it.

	EM	EM_{loc}	EM_{type}	BLEU	ROUGE
NO-FB	3.5	9.7	30.4	21.7	39.0
FBNET$_o$	**38.6**	**45.8**	**69.3**	**54.2**	**70.6**

Table 2: **Interpreting NL Feedback:** Correctness of Predicted Edits .... Given an erroneous script x, and general feedback fb from the user, FBNet perfectly predicts the specific repair edits 38% of the time (EM, exact match) - an order of magnitude better than a baseline NO-FB predicting the repair from x alone. EM_{loc} and EM_{type} compare just parts of the edit sequences (locations/types of the required edits, respectively), while BLEU and ROUGE are softer matching metrics.

How consistently does FBNET interpret similar feedback? In ∼15% of the data points, multiple **fb** can lead to the same $(\mathbf{x}, \mathbf{y})$ pair. FBNET is expected to behave consistently for such re-phrasings of **fb**. The model consistently produces exactly the same $\mathbf{y}$ for **fb** re-phrasings ∼ 60% of the time. Furthermore, we observe majority agreement as the number of **fb** re-phrasings for a $(\mathbf{x}, \mathbf{y})$ pair increases. In our analysis, a large proportion of the inconsistent edits occur because different **fb** phrasings prompt the model to generate slightly different, but semantically similar edits: see Table 3 for an example.

feedback	predicted edit
The feedback is if a person is going to open a book, they need to choose one first	insert node 'choose a book to read' before 'open the book'
The feedback is you can't open something you're not holding	insert node 'get the book out of the bag' before 'open the book'

Table 3: Multiple feedbacks for the same $(\mathbf{x}, \mathbf{y})$. Here, $\mathbf{x}$ is: You are given a plan to read to child. decide which books to read, open the book, read the book to the child, turn the pages ... . $\mathbf{y}^e$ is *insert node 'pick a book off the shelf' before 'open the book'*

How well can FBNET handle wrong feedback? While the ability to react to feedback is a desired trait of FBNET, we also want to ensure that the performance of FBNET is proportional to the quality of feedback. This will ensure that FBNET can act faithfully in settings where the feedback might be potentially misleading. We investigate this question by identifying lexically similar scripts but irrelevant feedback from the training set for each test example. Note that our setup easily allows us to test this hypothesis since the train/test/val splits were carefully designed to ensure no overlap between the examples. Thus the feedback from one example will typically not apply to another example. We find that with irrelevant feedback, the performance of FBNET drops to 3%. This shows that FBNET is sensitive to the quality of feedback, and no feedback is better than misleading and irrelevant feedback.

How well does FBNET perform across error types FBNET$_o$ gets the highest performance (EM 63.0%) on wrong-step error type where **fb** typically contains negative words that signal the error type, and the model learns to localize the error node. One of the most challenging error types is partial order removal or addition (EM 10.5%). This can be attributed to the challenging localization involving multiple nodes that participate in a partial order. The lowest-performing is the missing step error type (EM 2.73). The reason for this low EM score is that the edit is generates the missing node, and EM undercounts the correctness. Other metrics such as ROUGE are much higher validating that the model performs well on this error type. Section §9 Table 8 breaks down the performance of FBNET by error type.

5.1.1 Error analysis

We randomly sampled 50 instances from the test set where the model generates an incorrect edit (i.e., $EM = 0$). Our goal is to understand the typical errors made by the model and use the analysis to calibrate the findings in Table 2.

- **Lexical variation (36%)** Exact match underestimates the performance of our model (as the task is involves generation). We find that more than 35% of the predicted edits are semantically similar (typically lexical variation) to the reference gold edit. Some examples include: insert node *picking a book...* vs, insert node *choosing a book to read*. Another kind of example is model suggesting swapping the order of *edges A and B* while the reference edit swaps *edges B and A* - but both of these are equivalent.
- **Challenging feedback (24%)** This type of error occurs when the model fails to interpret a feedback because it is difficult to interpret e.g., the feedback is expressed abstractly. For example, for the goal "go to locker room," the generated script repeats the step "walk to the locker room.". The feedback is

'you can't go where you already are', and FBNET generates the edit "reorder edge between '⟨ walk towards the locker room , walk to the locker room ⟩' " , failing to interpret the feedback.

- **Error not localized (20%)** In about 20% of the failures, FBNET fails to localize the error given the feedback. For example, consider the erroneous input script about the goal *buy an xbox*: *1. go to the store 2. talk to the cashier 3. make the transaction 4. get the receipt 5. load the video game into the car 6. get into the car 7. take xbox home* The feedback is *after a person makes a transaction, they then head to their car*. The expected edit is: *insert node 'walk to the car' after 'get the receipt'*, but the predicted edit *insert node 'get into the car' after 'make the transaction'* does not correctly identify the erroneous node. The feedback is pointing to making a transaction, but it also involves getting the receipt.
- **Alternative answers (16%)** We also encounter cases where there are multiple ways to correct a script. For example, an edit can be expressed as *insert node 'X' before 'step 4'* or *insert node 'X' after 'step 3'*. This comprises ∼ 16% of the errors.

In ∼32% cases, the model generates a correct edit that differs from the gold. Extrapolating this performance under-counting to the entire test set, the accuracy of FBNET in Table 2 would increase to ∼70% (+32%).

5.2 RQ2: How well can FBNET learn from prior mistakes?

	EM	EM_{loc}	EM_{type}	BLEU	ROUGE
NO-FB	6.94	15.3	34.7	24.1	44.2
FBNET	16.72	20.9	56.9	32.5	48.5
FBNET$_o$	22.2	27.8	72.2	44.6	65.8

Table 4: **Learning from prior mistakes:** On the reuse dataset, given an erroneous script x, and feedback fb recalled from similar, prior examples, FBNet perfectly predicts the specific repair edits 16.7% of the time (or 20.9% the edit location and 56.9% the edit type), a promising start to learning from prior mistakes.

Section §5.1 shows that the corrector **G** can utilize user-supplied feedback to fix an incorrect structure. FBNET combines **G** with a memory $\mathcal{M}$ of feedback, allowing us to leverage past feedback on new examples. This section presents a setup where feedback on previously seen inputs is used to fix new, unseen examples.

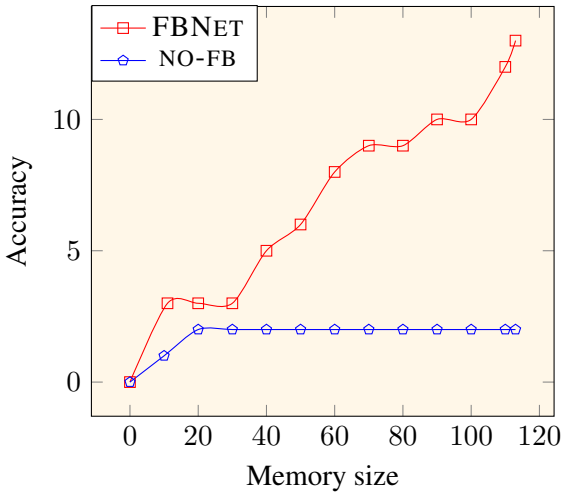

Figure 4: Performance on unseen examples (number of correct data points) improves as memory size grows. NO-FB baseline performance remains static.

To investigate this setting, we create a new test set, called the *interaction-reuse set* or ISET. To create it, we randomly sample 72 test points (referred as *interaction-reuse set*-genesis or ISET-SOURCE) and perturb them linguistically to generate *interaction-reuse set*(also referred as ISET). The perturbations are performed on the salient entities in the script, including (i) linguistic perturbation on ∼20% samples (e.g., box → carton, package) and (ii) the relatively harder analogical perturbation on the remaining ∼80% samples (e.g., bus → train, and how to lift blinds → how to open oven door because the event structure is analogical). The $\mathbf{y}^e$ to the original script also applies to the substituted script. We ensured that the perturbations did not introduce additional errors in the substituted script. This ensures that the *interaction-reuse set*now contains similar examples to the original test set, a condition that our original splits do not satisfy. There are a total of 72 data points in *interaction-reuse set*.

Continually learning using a memory of errors Examples in *interaction-reuse set* are randomly mixed with the original test set. This combined test set of queries Q is then evaluated using our setup as shown in Algorithm 1. Intuitively, *interaction-reuse set* allows us to simulate a setting where the system has been deployed in the wild, and end-users can query. Algorithm 1 runs the memory-based inference described in Section §3 (Figure 2). As the system is run through the stream of queries, we expect that i) the overall performance of the system will be better than no feedback, as some of the

Algorithm 1: FBNET inference on a stream of inputs with growing memory

Given: FBNET, $\mathcal{M}$, Ω
Given: Set $\{$ISET $\cup$ ISET-SOURCE$\}$ of N queries.
for $i \leftarrow 1, 2, \ldots, N$ **do**
  /* Check memory for feedback */
  $\bar{\mathbf{fb}}_i = \Omega(\mathbf{x}_i, \mathcal{M})$;
  /* Get corrected structure from FBNET. $\bar{\mathbf{fb}}_i$ can be empty. */
  $\mathbf{y}_i = \text{FBNET}\ (\mathbf{x}_i, \bar{\mathbf{fb}}_i)$;
  /* Get user feedback */
  $\mathbf{fb}_i =$ User feedback on $\mathbf{y}_i$;
  /* Grow memory with new feedback */
  Write $\mathbf{fb}_i$ to $\mathcal{M}$
end

examples in the interaction set will provide meaningful feedback, and ii) the *running* performance of the system will improve with growing memory: the probability of relevant feedback being present for an unseen example increases with time, thus boosting the performance.

Our experiments show that FBNET meets both these expectations. First, Table 4 shows that retrieved feedback improves over no feedback by 10 points (exact match) and similarly in terms of BLEU and ROUGE scores, respectively. Further, Figure 4 shows a graph confirming that FBNET can improve continuously as memory grows.

6 Scope

In principle, we could apply FBNET to any task that satisfies the assumptions (§3.2). However, our approach has some limitations in practice, several of which merit further detailed follow-up work.

- **On Assumption A1:** We assume that the output of $\mathbf{B}$ is repairable. Such an assumption is only possible for models that generate mostly correct outputs and have errors that are easy to highlight for humans. In practice, this implies that our approach will most efficiently work in conjunction with modern language models (Bommasani et al., 2021) that are shown to be syntactically correct in form, but can produce output that lacks commonsense (Bender and Koller, 2020), making their output repairable).
- **On Assumption A2:** Having reusable, general feedback is costly and requires careful instructions to collect from general users and crowdworkers (e.g., we asked the crowdworkers how they would explain the model error to a five-year-old). As the domain of the task becomes more specialized, such as database query generation (Elgohary et al., 2021) or code correction (Yasunaga and Liang, 2020), collecting data to train $\mathbf{G}$ becomes difficult. Systems that produce structured explanations are better suited to our model (see Wiegreffe and Marasović (2021) for an overview), rather than specialized domains that require expert users to provide feedback (e.g., in database query generation).
- **Consistent memory:** We show in Section §5.1 that FBNET is sensitive to the appropriateness of the feedback. However, adversarial or incorrect feedback could pollute the memory and possibly make it inconsistent. There has been some recent work to ensure consistency of beliefs of a model (Kassner et al., 2021), and more effort is required in this direction to apply to more complex settings like ours.
- **Using multiple feedbacks:** Ω can be enhanced with more complex attention mechanisms that aggregate from multiple relevant memory entries and possibly generalize them. We conducted an initial experiment using attention and found that we would need a larger dataset to train Ω effectively.

Advancements in these directions would further increase the applicability of FBNET. Still, there are several applications (Wiegreffe and Marasović, 2021) where our approach would currently apply in principle, or is easy to set up.

7 Summary

Our goal is to create a system that can continuously improve the structured output of a model. Our approach is to train an error correction model that uses natural language (NL) feedback to correct errors in that output. We have presented the first step towards this goal, showing that an error correction module can learn to interpret NL feedback successfully, resulting in 40% fewer errors in script generation. We have also described ongoing work on the next step, namely adding a memory layer where human feedback is stored and later retrieved efficiently. Together, these offer a possible path to systems that can continuously improve their output over time, with progressively less feedback and without retraining.

References

Emily M. Bender and Alexander Koller. 2020. Climbing towards nlu: On meaning, form, and understanding in the age of data. In *ACL*.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

S. Dasgupta, Daniel J. Hsu, Stefanos Poulis, and Xiaojin Zhu. 2019. Teaching a black-box learner. In *ICML*.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. 2021. Nl-edit: Correcting semantic parse errors through natural language interaction. *arXiv preprint arXiv:2103.14540*.

Sebastian Gehrmann, Tosin Adewumi, Karmanya Aggarwal, Pawan Sasanka Ammanamanchi, Aremu Anuoluwapo, Antoine Bosselut, Khyathi Raghavi Chandu, Miruna Clinciu, Dipanjan Das, Kaustubh D Dhole, et al. 2021. The gem benchmark: Natural language generation, its evaluation and metrics. *arXiv preprint arXiv:2102.01672*.

Kelvin Guu, Kenton Lee, Z. Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Realm: Retrieval-augmented language model pre-training. *ArXiv*, abs/2002.08909.

K. Hammond. 1986. Chef: A model of case-based planning. In *AAAI*.

Nora Kassner, Oyvind Tafjord, Hinrich Schutze, and Peter Clark. 2021. Beliefbank: Adding memory to a pre-trained language model for a systematic notion of belief. In *EMNLP*.

Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, V. Karpukhin, Naman Goyal, Heinrich Kuttler, M. Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *NeurIPS*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Nikhil Mehta and Dan Goldwasser. 2019. Improving natural language interaction with robots using advice. In *NAACL-HLT*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, M. Matena, Yanqi Zhou, W. Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

C. Riesbeck. 1981. Failure-driven reminding for incremental learning. In *IJCAI*.

B. Ross. 1984. Remindings and their effects in learning a cognitive skill. *Cognitive Psychology*, 16:371–416.

Keisuke Sakaguchi, Chandra Bhagavatula, Ronan Le Bras, Niket Tandon, Peter Clark, and Yejin Choi. 2021. proscript: Partially ordered scripts generation via pre-trained language models.

R. Schank and David B. Leake. 1989. Creativity and learning in a case-based explainer. *Artif. Intell.*, 40:353–385.

Burr Settles. 2012. Active learning. In *Active Learning*.

G. Sussman. 1973. A computational model of skill acquisition. Technical Report AITR-297, MIT.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. 2019. CommonsenseQA: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4149–4158, Minneapolis, Minnesota. Association for Computational Linguistics.

Alon Talmor, Oyvind Tafjord, P. Clark, Y. Goldberg, and Jonathan Berant. 2020. Teaching pre-trained models to systematically reason over implicit knowledge. *NeurIPS*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *ArXiv*, abs/1804.07461.

Sida I. Wang, Percy Liang, and Christopher D. Manning. 2016. Learning language games through interaction. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2368–2378, Berlin, Germany. Association for Computational Linguistics.

Sarah Wiegreffe and Ana Marasović. 2021. Teach me to explain: A review of datasets for explainable nlp. *ArXiv*, abs/2102.12060.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. *ArXiv*, abs/2005.10636.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. *arXiv preprint arXiv:2106.06600*.

Luke Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*.

8 Appendix

8.1 Initial study on the errors of B (PROSCRIPT)

On PROSCRIPT's test set, we performed inference using the released checkpoint (both GPT-2 and T5-XXL based model). We then randomly sampled 30 generated graphs and manually wrote feedback for them. Similar to Sakaguchi et al. (2021), we found that the model makes repairable mistakes (leading to assumption A1 being satisfied). Further, we found there instances where a general principle feedback applies across more than one instances (e.g., you have to be near something to use it). (see Table 5).

What was the error	General principle feedback
Script was missing the step of not turning off the alarm after waking up	People don't leave their alarms ringing all day.
Script mentioned coming to the doorway and passing through it	One cannot walk through the doorway without opening the door first.
Script tells that getting in car and drive in zoo can be done in any order	People must get into a vehicle, before driving to any place.
Script is looking for a butterfly after placing it	You don't need to look for a butterfly if it's already in a container.

Table 5: Sample error and the corresponding general principle feedback that could, in principle, repair the model output.

On an average, there were about two mistakes present in the graphs. Often, the error was that the script was using an entity before having it (e.g., write on the paper comes before the node find the paper or reach for the paper). Thus, there seems to be a possibility of applying similar feedback to more than one example. We also found some cases where the script might have to be changed to adapt to special cases. For example, for a script *visit Disneyland*, an event *obtain a visa* might be required for some users. We believe the original ProScript dataset aims to generate widely applicable scripts and grounded in commonsense; rather than cover all possible outcomes.

On the surface, the generated scripts were of good quality. However, a closer look at the mistakes revealed that most of them could be attributed to the model lacking basic commonsense. For example, Figure 1 shows a typical mistake the model makes. This underscores the gap between the syntax and semantic correctness of machine-generated output in the context of automatic script generation. This observation is in-line with other NLP tasks (Bender and Koller, 2020) that distinguish the success of recent models on the correctness of form rather than the far-from-over goal of understanding of meaning.

8.2 Data collection

An average user could point out mistakes in the generated scripts, as a majority of the errors in generated scripts are caused by a lack of basic commonsense (§8.1). Consequently, we designed a Mechanical Turk task to provide feedback on mistakes. A broad overview of the annotation process is shown in Figure 5.

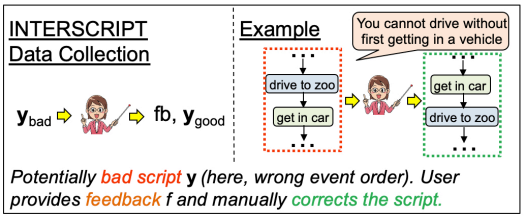

Figure 5: A broad overview of the annotation process. For actual annotation task (including the M-turk task template), see our code repository.

Annotation Now we discuss our crowdsourcing setup to collect the data. To maximize the opportunity to get more feedbacks for a predicted script, we filtered a subset of the test set in ProScript where the human evaluated graph edit distance was likely to be high (i.e., there were likely to be more errors). The ProScript authors released the graph edit value for the set of test set samples they evaluated. We performed inference using their released PROSCRIPT$_{gen}$ model on those data points with high graph edit distance value ($\geq$ 8). With this we collected about 400 (predicted graph, expected gold graph) tuples. The ProScript paper describes that their expected gold graph is also imperfect and might contain about 20% noise. Nevertheless, having the gold reference graph guides and constrains an annotator about the common script for a scenario rather than the wide open space of solving the task using multiple potentially correct scripts. (e.g., one could go to a zoo without driving the car by hiring a taxi and then they won't need to drive or park the car). As mentioned in §8.1 our annotation process must focus on scripts that are widely applicable and grounded in commonsense.

The annotators are shown the model-generated

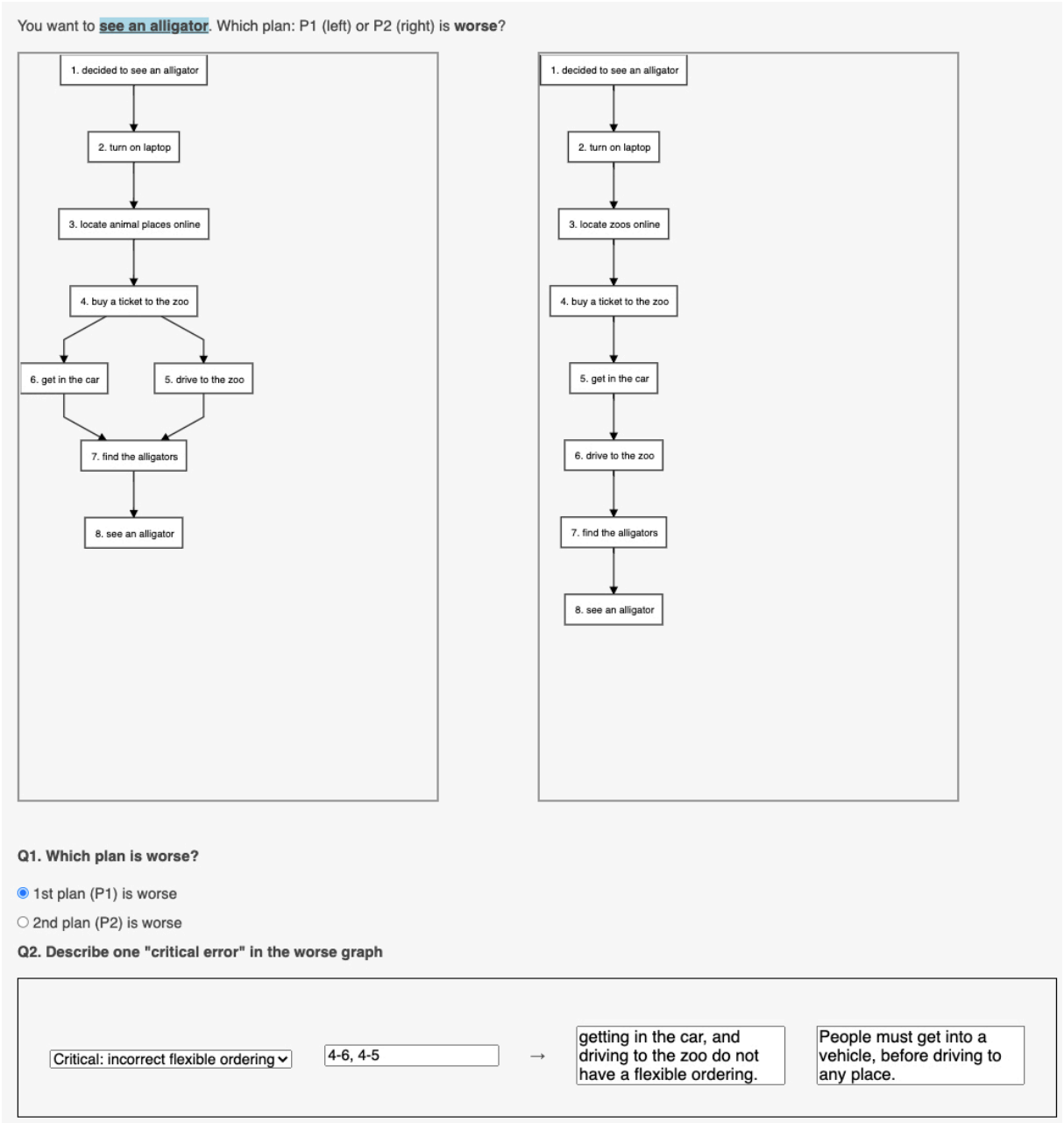

Figure 6: The mechanical turk page for annotation. We show the generated and the expected ProScript gold reference. The annotator must answer which script is worse and why. They must point out an egregious mistake (and not any trivial errors that have minor grammatical errors), and annotate: the error type (missing step, wrong step, wrong order, wrong partial order), localize the error by providing the node or edge id, and give feedback why it is wrong, and finally to gather the general principle behind the feedback they are asked to explain the feedback to a five-year-old.

and expected gold (reference) scripts, and are required to answer which script is worse and why. It is possible that the gold script is marked as worse. However, we later post-process and remove such cases, as our focus is to get errors on the generated scripts and not the manually created scripts. The annotators must point out an obvious mistake (e.g., an event or an edge that does not follow commonsense). They were asked to ignore grammatical and fluency errors, and focus on critical errors of four types:

- **Wrong ordering:** the order in the sequence of steps is not correct (e.g., wearing shoes is described before wearing socks).
- **Flexible ordering:** some steps can be done in a flexible order (e.g., you can wear left sock or right sock first). A good script captures such flexibility.
- **Missing critical steps:** a bad script might have missed critical steps (e.g., the script can say: "wait for a plane" followed by "get off the plane" – here an obvious step "get on the plane" is missing) . There is no strict definition for a critical step, so the annotators were instructed to use their commonsense judgment.
- **Wrong step:** a bad script might have irrelevant and wrong steps (e.g., the script describing "go to a party" might describe an irrelevant step such as read a book, open a book, etc.).

For every data point, the annotators were asked to answer the following:

- Explicit feedback type-1: the error type (missing step, wrong step, wrong order, wrong partial order)
- Explicit feedback type-2: localize the error by providing the erroneous node or edge id
- Implicit feedback type-1: give feedback in a few words, explaining the error
- Implicit feedback type-2: An explanation of the error that would potentially make sense to a five-year-old. Such an explanation of the feedback helped gather the general principle that is violated, and is targeted in the feedback.

Fig. 6 shows a sample of our Mechanical Turk task. Annotators were required to list only one critical error that they believe was most important. Each data point is annotated by three annotators, adding some diversity in the errors. The annotators were paid $15 an hour. The annotators were English speaking crowdworkers on Mechanical Turk from USA. The average time for completion of one script was 2 minutes.

fb type	count	example
explicit **fb** type-1	1,553	Remove node 'put the shirt on'
explicit **fb** type-2	1,553	The following step is not right: put the shirt on
implicit **fb** type-1	1,553	It tells you to iron your shirt while it's still on your body.
implicit **fb** type-2	1,553	If you hold a hot iron against the clothes you're currently wearing, you'll get terrible burns.
total	6,212	https://anonymous.4open.science/r/interscript/data.json

Table 6: Dataset statistics. In this paper, we use the hardest feedback (implicit-feedback-type-2). This example is from the input script: input script for the following table (goal: press the wrinkles out) = 1. put the shirt on, 2. find place to press, 3. grab iron from drawer, 4. place iron on shirt, 5. wait for iron to heat up, 6. use iron to smooth out wrinkles, 7. press the wrinkles out

We measured the agreement on labels (which graph is worse), and on explicit feedback type-1 and type-2. It was difficult to measure agreement on implicit feedback because it is not easy to perform binary comparison on the generated text without accounting for linguistic variations. On the labels, the Fleiss Kappa agreement was 0.90 (almost perfect agreement) and on explicit feedback the agreement was 0.75 Fleiss Kappa (substantial agreement). This also shows that there is some diversity in what the users perceive as a serious mistake in y_{bad}.

Eventually, we compiled these annotations into a dataset of 1,553 4-tuples where each entry consists of (explicit feedback type-1, explicit feedback type-2, implicit feedback type-1, implicit feedback type-2).

8.3 Model Output Examples

Table 7 provides some sample model outputs.

9 Performance of FBNET by error type

Table 8 breaks down the performance of FBNET by error type.

x	**fb**	$\mathbf{y}^{e^*}$	**y**
1. ... 2. set alarm for early morning 3. get out of bed 4. prepare for yoga 5. go to the bathroom 6. do yoga 7. do yoga in the morning	People don't leave their alarms ringing all day.	insert node 'wake up and turn off alarm' before 'get out of bed'	1. ... 2. set alarm for early morning 3. wake up and turn off alarm 4. get out of bed 5. ... 6. ... 7. ...
1. ... 3. put on shoes ... 5. open the door 6. drive to the train station 7. get into the car 8. reach the train station	You must get into a vehicle, before driving to any place.	reorder edge between '⟨ drive to the train station, get into the car ⟩'	1. ... 3. ... 5. open the door 6. get into the car. 7. drive to the train station 8. ...
1. ... 3. pick up the butterfly 4. put the butterfly in container 5. look for a butterfly ... 6. Take the butterfly home ...	You don't need to look for a butterfly if it's already in a container.	remove node 'look for a butterfly'	1. ... 3. pick up the butterfly 4. put the butterfly in container 5. Take the butterfly home 6. ...

Table 7: Task: Applying the graph edit to the bad script.

Edit type	EM%
Overall	38.6
Add partial order exactmatch	10.5
Add partial order type	44.7
Missing step exactmatch	2.8
Missing step type	65.5
Remove partial order exactmatch	0.0
Remove partial order type	0.0
Wrong ordering exactmatch	45.1
Wrong ordering type	72.8
Wrong step exactmatch	63.0
Wrong step type	78.6

Table 8: Performance of FBNET by error type