# Text Editing as Imitation Game

[*]Ning Shi[♠♡] Bin Tang[♡] Bo Yuan[♡] Longtao Huang[♡]
Yewen Pu[♣] Jie Fu[♢] [†]Zhouhan Lin[★]
[♠]Alberta Machine Intelligence Institute, Dept. of Computing Science, University of Alberta
[♡]Alibaba Group [★]Shanghai Jiao Tong University
[♣]Autodesk Research [♢]Beijing Academy of Artificial Intelligence
ning.shi@ualberta.ca, {tangbin.tang,qiufu.yb,kaiyang.hlt}@alibaba-inc.com
yewen.pu@autodesk.com, fujie@baai.ac.cn, lin.zhouhan@gmail.com

## Abstract

Text editing, such as grammatical error correction, arises naturally from imperfect textual data. Recent works frame text editing as a multi-round sequence tagging task, where operations – such as insertion and substitution – are represented as a sequence of tags. While achieving good results, this encoding is limited in flexibility as all actions are bound to token-level tags. In this work, we reformulate text editing as an imitation game using behavioral cloning. Specifically, we convert conventional sequence-to-sequence data into state-to-action demonstrations, where the action space can be as flexible as needed. Instead of generating the actions one at a time, we introduce a dual decoders structure to parallel the decoding while retaining the dependencies between action tokens, coupled with trajectory augmentation to alleviate the distribution shift that imitation learning often suffers. In experiments on a suite of Arithmetic Equation benchmarks, our model consistently outperforms the autoregressive baselines in terms of performance, efficiency, and robustness. We hope our findings will shed light on future studies in reinforcement learning applying sequence-level action generation to natural language processing.

## 1 Introduction

Text editing (Malmi et al., 2022) is an important domain of processing tasks to edit the text in a localized fashion, applying to text simplification (Agrawal et al., 2021), grammatical error correction (Li et al., 2022), punctuation restoration (Shi et al., 2021), to name a few. Neural sequence-to-sequence (seq2seq) framework (Sutskever et al., 2014) establishes itself as the primary approach to text editing tasks, by framing the problem as machine translation (Wu et al., 2016). Applying a seq2seq modeling has the advantage of simplicity, where the system can simply be built by giving input-output pairs consisting of pathological sequences to be edited, and the desired sequence output, without much manual processing efforts (Junczys-Dowmunt et al., 2018).

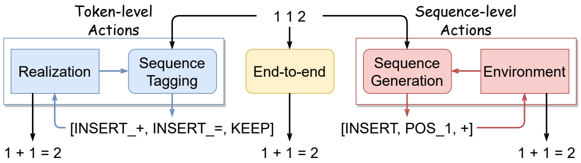

Figure 1: Three approaches – sequence tagging (left), end-to-end (middle), sequence generation (right) – to turn an invalid arithmetic expression "1 1 2" into a valid one "1 + 1 = 2". In end-to-end, the entire string "1 1 2" is encoded into a latent state, which the string "1 + 1 = 2" is generated directly. In sequence tagging, a localized action (such as "INSERT_+", meaning insert a "+" symbol after this token) is applied/tagged to each token; these token-level actions are then executed, modifying the input string. In contrast, sequence generation output an entire action sequence, generating the location (rather than tagging it), and the action sequence is executed, modifying the input string. Both token-level actions and sequence-level actions can be applied multiple times to polish the text further (up to a fixed point).

However, even with a copy mechanism (See et al., 2017; Zhao et al., 2019; Panthaplackel et al., 2021), an end-to-end model can struggle in carrying out localized, specific fixes while keeping the rest of the sequence intact. Thus, sequence tagging is often found more appropriate when outputs highly overlap with inputs (Dong et al., 2019; Mallinson et al., 2020; Stahlberg and Kumar, 2020). In such cases, a neural model predicts a tag sequence – representing localized fixes such as insertion and substitution – and a programmatic interpreter implements these edit operations through. Here, each tag represents a *token-level* action and determines the operation on its attached token (Kohita et al., 2020). A model can avoid modifying the overlap by assigning no-op (e.g., KEEP), while the action space is limited to token-level modifications,

[*] Work was done at Alibaba Group.

[†] Zhouhan Lin is the corresponding author.

*Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 1583–1594
December 7-11, 2022 ©2022 Association for Computational Linguistics

such as deletion or insertion after a token (Awasthi et al., 2019; Malmi et al., 2019).

In contrast, alternative approaches (Gupta et al., 2019) train the agent to explicitly generate free-form edit actions and iteratively reconstructs the text during the interaction with an environment capable of altering the text based on these actions. This *sequence-level* action generation (Branavan et al., 2009; Guu et al., 2017; Elgohary et al., 2021) allows higher flexibility of action design not limited to token-level actions, and is more advantageous given the narrowed problem space and dynamic context in the edit (Shi et al., 2020).

The mechanisms of sequence tagging and sequence generation against end-to-end are exemplified in Figure 1. Both methods allow multiple rounds of sequence refinement (Ge et al., 2018; Liu et al., 2021) and imitation learning (IL) (Pomerleau, 1991). Essentially an agent learns from the demonstrations of an expert policy and later imitates the memorized behavior to act independently (Schaal, 1996). On the one hand, IL in sequence tagging functions as a standard supervised learning in its nature and thus has attracted significant interest and been widely used recently (Agrawal et al., 2021; Yao et al., 2021; Agrawal and Carpuat, 2022), achieving good results in the token-level action generation setting (Gu et al., 2019; Reid and Zhong, 2021). On the other hand, IL in sequence-level action generation is less well defined even though its principle has been followed in text editing (Shi et al., 2020) and many others (Chen et al., 2021). As a major obstacle, the training is on state-action demonstrations, where the encoding of the states and actions can be very different (Gu et al., 2018). For instance, the mismatch of the lengths dimension between the state and action makes it tricky to implement for an auto-regressive modeling that benefits from a single, uniform representation.

To tackle the issues above, we reformulate text editing as an *imitation game* controlled by a Markov Decision Process (MDP). To begin with, we define the input sequence as the initial state, the required operations as action sequences, and the output target sequence as the goal state. A learning agent needs to imitate an expert policy, respond to seen states with actions, and interact with the environment until the success of the eventual editing. To convert existing input-output data into state-action pairs, we utilize *trajectory generation* (TG), a skill to leverage dynamic programming (DP) for an efficient search of the minimum operations given a predefined edit metric. We backtrace explored editing paths and automatically express operations as action sequences. Regarding the length misalignment, we first take advantage of the flexibility at the sequence-level to fix actions to be of the same length. Secondly, we employ a linear layer after the encoder to transform the length dimension of the context matrix into the action length. By that, we introduce a *dual decoders* (D2) structure that not only parallels the decoding but also retains capturing interdependencies among action tokens. Taking a further step, we propose *trajectory augmentation* (TA) as a solution to the distribution shift problem most IL suffers (Ross et al., 2011). Through a suite of three Arithmetic Equation (AE) benchmarks (Shi et al., 2020), namely Arithmetic Operators Restoration (AOR), Arithmetic Equation Simplification (AES), and Arithmetic Equation Correction (AEC), we confirm the superiority of our learning paradigm. In particular, D2 consistently exceeds standard autoregressive models from performance, efficiency, and robustness perspectives.

In theory, our methods also apply to other imitation learning scenarios where a reward function exists to further promote the agent. In this work, we primarily focus on a proof-of-concept of our learning paradigm landing at supervised behavior cloning (BC) in the context of text editing. To this end, our contributions[1] are as follows:

1. We frame text editing into an imitation game formally defined as an MDP, allowing the highest degrees of flexibility to design actions at the sequence-level.
2. We involve TG to translate input-output data to state-action demonstrations for IL.
3. We introduce D2, a novel non-autoregressive decoder, boosting the learning in terms of accuracy, efficiency, and robustness.
4. We propose a corresponding TA technique to mitigate distribution shift IL often suffers.

## 2 Imitation Game

We aim to cast text editing into an imitation game by defining the task as a recurrent sequence generation, as presented in Figure 2 (a). In this section, we describe the major components of our proposal, including (1) the problem definition, (2) the data translation, (3) the model structure, and (4) a solution to the distribution shift.

[1] Code and data are publicly available at GitHub.

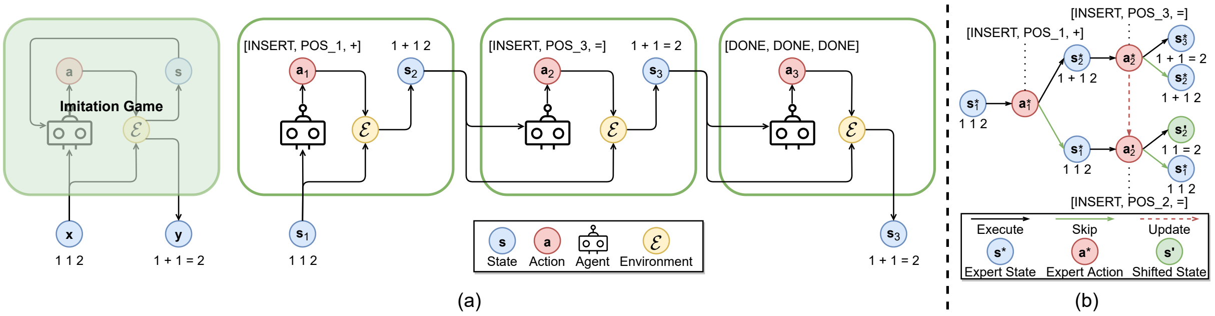

Figure 2: (a) shows the imitation game of AOR. Considering input text $\mathbf{x}$ as initial state $\mathbf{s}_1$, the agent interacts with the environment to edit "1 1 2" into "$1+1=2$" via action $\mathbf{a}_1$ to insert "+" at the first position and $\mathbf{a}_2$ to insert "=" at the thrid position. After $\mathbf{a}_3$, the agent stops editing and calls the environment to return $\mathbf{s}_3$ as the output text $\mathbf{y}$. Using the same example, (b) explains how to achieve shifted state $\mathbf{s}'_2$ by skipping action $\mathbf{a}^*_1$ and doing $\mathbf{a}'_2$. Here we update $\mathbf{a}^*_2$ to $\mathbf{a}'_2$ accordingly due to the previous skipping. The new state $\mathbf{s}'_2$ was not in the expert demonstrations.

### 2.1 Behavior cloning

We tear a text editing task $\mathcal{X} \mapsto \mathcal{Y}$ into recurrent subtasks of sequence generation $\mathcal{S} \mapsto \mathcal{A}$ defined by an MDP tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{E}, \mathcal{R})$.

**State** $\mathcal{S}$ is a set of text sequences $\mathbf{s} = s_{j \le m}$, where $s \in \mathcal{V}_{\mathcal{S}}$. We think of a source sequence $\mathbf{x} \in \mathcal{X}$ as the initial state $\mathbf{s}_1$, its target sequence $\mathbf{y} \in \mathcal{Y}$ as the goal state $\mathbf{s}_T$, and every edited sequence in between as an intermediate state $\mathbf{s}_t$. The path $\mathbf{x} \mapsto \mathbf{y}$ can be represented as a set of sequential states $\mathbf{s}_{t \le T}$.

**Action** $\mathcal{A}$ is a set of action sequences $\mathbf{a} = a_{i \le n}$, where $a \in \mathcal{V}_{\mathcal{A}}$. In Figure 3, "INSERT", "POS_3", and "=" are three action tokens belonging to the vocabulary space of action $\mathcal{V}_{\mathcal{A}}$. In contrast to token-level actions in sequence tagging, sentence-level ones set free the editing by varying edit metrics $\mathbf{E}$ (e.g., Levenshtein distance) as long as $\mathcal{X} \xmapsto{\mathcal{A}_{\mathbf{E}}} \mathcal{Y}$. It serves as an expert policy $\pi^*$ to demonstrate the path to the goal state. A better expert usually means better demonstrations and imitation results. Hence, depending on the task, a suitable $\mathbf{E}$ is essential.

**Transition matrix** $\mathcal{P}$ models the probability $p$ that an action $\mathbf{a}_t$ leads a state $\mathbf{s}_t$ to the state $\mathbf{s}_{t+1}$. We know $\forall \mathbf{s}, \mathbf{a}.\ p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) = 1$ due to the nature of text editing. So we can omit $\mathcal{P}$.

**Environment** $\mathcal{E}$ responds to an action and updates the game state accordingly by $\mathbf{s}_{t+1} = \mathcal{E}(\mathbf{s}_t, \mathbf{a}_t)$ with process control. For example, the environment can refuse to execute actions that fail to pass the verification and terminate the game if a maximum number of iterations has been consumed.

**Reward function** $\mathcal{R}$ calculates a reward for each action. It is a major factor contributing to the success of reinforcement learning. In the scope of this paper, we focus on BC, the simplest form of IL. So we can also omit $\mathcal{R}$ and leave it for future work.

**Algorithm 1** Trajectory Generation (TG)

```
Input: Initial state $\mathbf{x}$, goal state $\mathbf{y}$, environment $\mathcal{E}$, and edit metric $\mathbf{E}$.
Output: Trajectories $\tau$.
$\tau \leftarrow \emptyset$
$\mathbf{s} \leftarrow \mathbf{x}$
$ops \leftarrow \mathrm{DP}(\mathbf{x}, \mathbf{y}, E)$
for $op \in ops$ do
    $\mathbf{a} \leftarrow \mathrm{Action}(op)$        ▷ Translate operation to action
    $\tau \leftarrow \tau \cup [(\mathbf{s}, \mathbf{a})]$
    $\mathbf{s} \leftarrow \mathcal{E}(\mathbf{s}, \mathbf{a})$
end for
$\tau \leftarrow \tau \cup [(\mathbf{s}, \mathbf{a}_T)]$   ▷ Append goal state and output action
return $\tau$
```

The formulation turns out to be a simplified $\mathcal{M}_{BC} = (\mathcal{S}, \mathcal{A}, \mathcal{E})$. Interacting with the environment $\mathcal{E}$, we hope a trained agent is able to follow its learned policy $\pi : \mathcal{S} \mapsto \mathcal{A}$, and iteratively edit the initial state $\mathbf{s}_0 = \mathbf{x}$ into the goal state $\mathbf{s}_T = \mathbf{y}$.

### 2.2 Trajectory generation

A data set to learn $\mathcal{X} \mapsto \mathcal{Y}$ consists of input-output pairs. It is necessary to convert it into state-action ones so that an agent can mimic the expert policy $\pi^* : \mathcal{S} \mapsto \mathcal{A}$ via supervised learning. A detailed TG is described in Algorithm 1.

Treating a pre-defined edit metric $\mathbf{E}$ as the expert policy $\pi^*$, we can leverage DP to efficiently find the minimum operations required to convert $\mathbf{x}$ into $\mathbf{y}$ in a left-to-right manner and backtrace this path to get specific operations.

Operations are later expressed as a set of sequential actions $\mathbf{a}^*_{t \le T}$. Here we utilize a special symbol DONE to mark the last action $\mathbf{a}^*_T$ where $\forall a \in \mathbf{a}^*_T.\ a = \mathtt{DONE}$. Once an agent performs $\mathbf{a}^*_T$, the current state is returned by the environment as the final output.

Given $\mathbf{s}^*_1 = \mathbf{x}$, we attain the next state $\mathbf{s}^*_2 =$

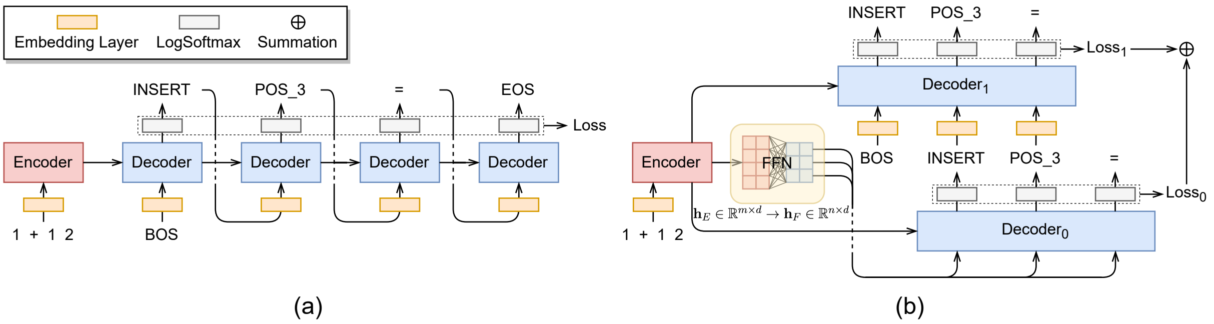

Figure 3: The conventional autoregressive decoder (a) compared with the proposed non-autoregressive D2 (b) in which the linear layer aligns the sequence length dimension for the subsequent parallel decoding.

$\mathcal{E}(\mathbf{s}_1^*, \mathbf{a}_1^*)$ and continue the rest until achieving $\mathbf{s}_T^* = \mathbf{y}$, resulting in a set of sequential states $\mathbf{s}_{t\le T}^*$.

After one-to-one correspondence between states and actions, we collect a set of sequential expert's demonstrations $\tau^* = [(\mathbf{s}_{t\le T}^*, \mathbf{a}_{t\le T}^*)]$. Repeating the same process, we eventually convert $\mathcal{X} \mapsto \mathcal{Y}$ into trajectories $\mathcal{T}^* : \mathcal{S} \mapsto \mathcal{A}$.

### 2.3 Model architecture

We form $\mathcal{S} \mapsto \mathcal{A}$ as sequence generation. More precisely, a neural model (i.e., the agent) takes states as input and outputs actions. Training an imitation policy with BC corresponds to fitting a parametric model $\pi_\theta$ that minimizes the negative log-likelihood loss $l(\mathbf{a}^*, \pi_\theta(\mathbf{s}))$. Most seq2seq models have an encoder-decoder structure.

**Encoder** takes an embedded state $\mathrm{E}(\mathbf{s}) \in \mathbb{R}^{m\times d}$ and generates an encoded hidden state $\mathbf{h}_E \in \mathbb{R}^{m\times d}$ with $d$ being the hidden dimension.

**Autoregressive decoder** in Figure 3 (a) conditions the current step on the encoded context and previously predictions to overcome the mismatch of sequence length. It calculates step by step

$$h_D^i = \mathrm{AR}(\mathrm{E}(a_{<i}), \mathbf{h}_E) \in \mathbb{R}^d, i = 0, \cdots, n+1,$$

$$\hat{a}_i = \mathrm{LogSoftmax}(h_D^i) \in \mathbb{R}^{|\mathcal{V}_\mathcal{A}|}, i = 0, \cdots, n+1,$$

and in the end, returns $\hat{\mathbf{a}} \in \mathbb{R}^{n\times|\mathcal{V}_\mathcal{A}|}$. The training is conducted as back-propgating $l(\mathbf{a}^*, \hat{\mathbf{a}})$. Note that $a_0^* = \texttt{BOS}$ and $a_{n+1}^* = \texttt{EOS}$ encourage the decoder to learn to begin and end the autoregression.

**Non-autoregressive decoder** instead provides hidden states in one time. It is feasible to apply techniques of non-autoregressive machine translation. However, one of the primary issues solved by that is the uncertainty of the target sequence length. When it comes to state-action prediction, thanks to the flexibility at the sequence-level, we are allowed to design actions on purpose to eliminate such uncertainty. Specifically, we enforce action sequences to be of fixed length. On this basis, we propose D2 as shown in Figure 3 (b). To address the misalignment of sequence length between state and action, we insert a fully connected feed-forward network between the encoder and decoder$_0$.

$$\mathrm{FFN}(\mathbf{h}_E) = (\mathbf{h}_E^{\mathbf{T}} W + b)^{\mathbf{T}} \in \mathbb{R}^{n\times d}$$

where $W \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^{d\times n}$ transform the length dimension from $m$ to $n$ so as to project $\mathbf{h}_E$ into $\mathbf{h}_F \in \mathbb{R}^{n\times d}$. The alignment of the sequence length allows us to trivially pass $\mathbf{h}_F$ to decoder$_0$.

$$\mathbf{h}_{D_0} = \mathrm{NAR}_0(\mathbf{h}_F, \mathbf{h}_E) \in \mathbb{R}^{n\times d}$$

$$\hat{\mathbf{a}}^0 = \mathrm{LogSoftmax}(\mathbf{h}_{\mathrm{D}_0}) \in \mathbb{R}^{n\times|\mathcal{V}_\mathcal{A}|}$$

For a clear comparison with the autoregressive decoder, we make minimal changes to the structure and keep modeling the dependence between two contiguous steps through decoder$_1$. To elaborate, we shift $\mathbf{a}^0$ one position to the right as $\acute{\mathbf{a}}^0$ by appending $a_0^*$ at the beginning and remove $a_n^0$ to maintain the sequence length. After that, we continue to feed $\acute{\mathbf{a}}^0$ to decoder$_1$.

$$\mathbf{h}_{D_1} = \mathrm{NAR}_1(\mathrm{E}(\acute{\mathbf{a}}^0), \mathbf{h}_E) \in \mathbb{R}^{n\times d}$$

$$\hat{\mathbf{a}}^1 = \mathrm{LogSoftmax}(\mathbf{h}_{\mathrm{D}_1}) \in \mathbb{R}^{n\times|\mathcal{V}_\mathcal{A}|}$$

At last, we conduct backpropagation with respect to the loss summation $l(\mathbf{a}^*, \hat{\mathbf{a}}^0) \oplus l(\mathbf{a}^*, \hat{\mathbf{a}}^1)$. Conventional seq2seq architectures are often equipped with intermediate modules such as a full attention distribution over the encoded context (Bahdanau et al., 2015), which is omitted in the above formulation for simplicity. In the implementation, we always assume to train decoder$_0$ and decoder$_1$ separately to increase the model capacity, yet weight sharing is possible.

### 2.4 Trajectory augmentation

IL suffers from distribution shift and error accumulation (Ross et al., 2011). An agent's mistakes can easily put it into a state that the expert demonstrations do not involve and the agent has never seen during training. This also means errors can add up, so the agent drifts farther and farther away from the demonstrations. To tackle this issue, we propose TA that expands the expert demonstrations and actively exposes shifted states to the agent. We accomplish this by diverting intermediate states and consider them as initial states for TG. An example is offered in Figure 2 (b).

Given expert states $\mathbf{s}^*_{t\leq T}$ and corresponding actions $\mathbf{a}^*_{t\leq T}$, we utilize the divide-and-conquer technique to (1) break down the chain of state generation $\mathbf{s}^*_t \xmapsto{\mathbf{a}^*_t} \mathbf{s}^*_{t+1}$ into two by either executing $\mathbf{a}^*_t$ to stay on the current path or skipping $\mathbf{a}^*_t$ to branch the current path; (2) recursively calling this process until reaching the goal state $\mathbf{s}^*_T$; (3) merge intermediate states from branches and return from bottom to top in the end. As illustrated in Algorithm 2, we collect a set of shifted states

$$\mathbf{S}' = \mathrm{TA}(\emptyset, \mathbf{s}^*_1, \mathbf{s}^*_{t\leq T}, \mathbf{a}^*_{t\leq T}, \mathcal{E}),$$

regard them as initial states paired with the same goal state to produce extra trajectories

$$\tau' = \bigcup_{\mathbf{s}^*\in\mathbf{S}^*} \mathrm{TG}(\mathbf{s}^*, \mathbf{s}^*_T, \mathcal{E}, \mathbf{E}),$$

and finally yield the augmented expert demonstrations $\mathcal{T}^* \cup \mathcal{T}'$ after looping through $\mathcal{X}$.

TA is advantageous because it (i) only exploits existing expert demonstrations to preserve the i.i.d assumption; (ii) is universally applicable to our proposed paradigm without a dependency on the downstream task; (iii) does not need domain knowledge, labeling work, and further evaluation.

## 3 Experiments

We adapt recurrent inference to our paradigm and evaluate them across AE benchmarks.

### 3.1 Setup

**Data.** Arithmetic Operators Restoration (AOR) is a short-to-long editing to complete an array into a true equation. It is also a one-to-many task as an array can be completed as multiple true equations differently. Arithmetic Equation Simplification (AES) aims to calculate the parenthesized parts and keep the equation hold, resulting in a long-to-short and many-to-one editing. Arithmetic Equation Correction (AEC) targets to correct potential mistakes in an equation. Diverse errors perturb the equation, making AEC a mixed many-to-many editing. To align with the previous work, we follow the same data settings $N$, $L$, and $D$ for data generation, as well as the same action design for trajectory generation. The edit metric $\mathbf{E}$ for AOR and AEC is Levenshtein, while $\mathbf{E}$ for AES is a self-designed one (SELF) that instructs to replace tokens between two parentheses with the target token. Examples are presented in Table 2. We refer readers to Shi et al. (2020) for an exhaustive explanation. As shown in Table 1, the data splits are 7K/1.5K/1.5K for training, validation, and testing respectively.

**Algorithm 2** Trajectory Augmentation (TA)

```
Input: States $\mathbf{S}$, state $\mathbf{s}_t$, expert states $\mathbf{S}^*$, actions $\mathbf{A}$, and
    environment $\mathcal{E}$.
Output: Augmented states $\mathbf{S}$.
if $|\mathbf{A}| > 1$ then
    $\mathbf{a}_t \leftarrow \mathbf{A}.\mathrm{pop}(0)$
    $\mathbf{s}_{t+1} \leftarrow \mathcal{E}(\mathbf{s}_t, \mathbf{a}_t)$
    $\mathbf{S} \leftarrow \mathbf{S} \cup \mathrm{TA}(\mathbf{S}, \mathbf{s}_{t+1}, \mathbf{S}^*, \mathbf{A}, \mathcal{E})$    ▷ Execute action
    $\mathbf{A} \leftarrow \mathrm{Update}(\mathbf{A}, \mathbf{s}_t, \mathbf{s}_{t+1})$
    $\mathbf{S} \leftarrow \mathbf{S} \cup \mathrm{TA}(\mathbf{S}, \mathbf{s}_t, \mathbf{S}^*, \mathbf{A}, \mathcal{E})$    ▷ Skip action
else if $\mathbf{s}_t \notin \mathbf{S}^*$ then
    $\mathbf{S} \leftarrow \mathbf{S} \cup [\mathbf{s}_t]$    ▷ Merge shifted state
end if
return S
```

**Evaluation.** Sequence accuracy and equation accuracy are two primary metrics with token accuracy for a more fine-grained reference. In contrast to sequence accuracy for measuring whether an equation exactly matches the given label, equation accuracy emphasizes whether an equation holds, which is the actual goal of AE tasks. It is noted that there is no hard constraint to guarantee that all the predicted actions are valid. However, when the agent makes an inference mistake, the environment can refuse to execute invalid actions and keep the current state. This is also one of the beauties of reformulating text editing as a controllable MDP.

**Baselines.** Recurrent inference (Recurrence) exhibits advantages over conventional end-to-end (End2end) and sequence tagging (Tagging) (Shi et al., 2020) . However, for AES and AEC, it[2] allows feeding training samples to a data generator and exposing more variants to models. These variants, as source samples paired with corresponding target samples, are used as the augmented dataset. This is impractical due to the strong dependency on domain knowledge. Given an input "1 + **(2 + 2)** =

[2] github.com/ShiningLab/Recurrent-Text-Editing

| **AOR** ($N = 10$, $L = 5$, $D = 10$K) | | | **AES** ($N = 100$, $L = 5$, $D = 10$K) | | | **AEC** ($N = 10$, $L = 5$, $D = 10$K) | | |
|---|---|---|---|---|---|---|---|---|
| **Train/Valid/Test** | **Train TA** | **Traj. Len.** | **Train/Valid/Test** | **Train TA** | **Traj. Len.** | **Train/Valid/Test** | **Train TA** | **Traj. Len.** |
| 7,000/1,500/1,500 | 145,176 | 6 | 7,000/1,500/1,500 | 65,948 | 6 | 7,000/1,500/1,500 | 19,764 | 4 |

Table 1: Data statistics of AE benchmarks.

| **Term** | **AOR** ($N = 10$, $L = 5$, $D = 10$K) | **AES** ($N = 100$, $L = 5$, $D = 10$K) | **AEC** ($N = 10$, $L = 5$, $D = 10$K) |
|---|---|---|---|
| Source $\mathbf{x}$ | 3 6 2 9 3 | 65 + ( 25 - 20 ) - ( 64 + 32 ) + ( 83 - 24 ) = ( - 25 + 58 ) | - 2 * + 4 10 + 8 / 8 = 8 |
| Target $\mathbf{y}$ | - 3 - 6 / 2 + 9 = 3 | 65 + 5 - 96 + 59 = 33 | - 2 + 10 * 8 / 8 = 8 |
| State $\mathbf{s}_t^*$ | - 3 - 6 / 2 9 3 | 65 + 5 - ( 64 + 32 ) + ( 83 - 24 ) = ( - 25 + 58 ) | - 2 + **4** 10 + 8 / 8 = 8 |
| Action $\mathbf{a}_t^*$ | [POS_6, +] | [POS_4, POS_8, 96] | [DELETE, POS_3, POS_3] |
| Next State $\mathbf{s}_{t+1}^*$ | - 3 - 6 / 2 **+** 9 3 | 65 + 5 - **96** + ( 83 - 24 ) = ( - 25 + 58 ) | - 2 + 10 + 8 / 8 = 8 |
| Shifted State $\mathbf{s}_t'$ | - 3 - 6 / 2 9 **=** 3 | 65 + 5 - ( 64 + 32 ) + **59** = ( - 25 + 58 ) | - 2 + 4 10 * 8 / 8 = 8 |

Table 2: Examples from AE with specific $N$ for integer size, $L$ for the number of integers, and $D$ for data size.

5" and output "1 + **4** = 5" in AES, a variant "1 + **(1 + 3)** = 5" can be generated based on the knowledge **1 + 3 = 4**. Nevertheless, if this knowledge is not provided in the other training samples, the model should only know **2 + 2 = 4**.

**Models.** As discussed, since the previously reported experiments are not practical, we re-run Recurrence source code for a more reasonable baseline (Recurrence*) that only has access to the fixed training set. Meanwhile, in our development environment, we reproduce Recurrence* within the proposed paradigm according to the compatibility in between. The encoder-decoder architecture inherits the same recurrent network as the backbone with long short-term memory units (Hochreiter and Schmidhuber, 1997) and an attention mechanism (Luong et al., 2015). The dimension of the bidirectional encoder is 256 in each direction and 512 for both the embedding layer and decoder. We apply a dropout of 0.5 to the output of each layer (Srivastava et al., 2014). This provides us a standard autoregressive baseline AR, as well as a more powerful AR* after increasing the number of encoder layers from 1 to 4. On the one hand, to construct a non-autoregressive baseline NAR, we replace the decoder of AR* with a linear layer that directly maps the context to a probability distribution over the action vocabulary. In addition, we add two more encoder layers to maintain a similar amount of trainable parameters. On the other hand, replacing the decoder of AR* with D2 leads to our model NAR*. We strictly unify the encoder for a fair comparison regarding the decoder. Model configurations are shared across AE tasks for a comprehensive assessment avoiding particular tuning against any of them.

**Training.** We train on a single NVIDIA Titan RTX with a batch size of 256. We use the Adam optimizer (Kingma and Ba, 2015) with a learning rate of $10^{-3}$ and an $\ell_2$ gradient clipping of 5.0 (Pascanu et al., 2013). A cosine annealing scheduler helps manage the training process and restarts the learning every 32 epochs to get it out of a potential local optimum. We adopt early stopping to wait for a lower validation loss until there are no updates for 512 epochs (Prechelt, 1998). Teacher forcing with a rate of 0.5 spurs up the training process (Williams and Zipser, 1989). In AES and AEC, the adaptive loss weighting guides the model to adaptively focus on particular action tokens in accordance with the training results. Reported metrics attached with standard deviation are the results of five runs using random seeds from [0, 1, 2, 3, 4].

### 3.2 Results

**Baselines.** As summarized in Table 3, prohibiting the access of Recurrence to domain knowledge outcomes a fair baseline and significantly weakens Recurrence* in AES and AEC. We also would like to point out that, even in the same impractical setting, our NAR* can achieve around 99.33% and 67.49% for AES and AEC with respect to equation accuracy, which is still much higher than that (87.73% and 58.27% for AES and AEC) reported in the previous work. In AOR, a one-to-many editing, no augmented source sequence is retrieved from the target side. We confirm that the slight accuracy drop of Recurrence* in AOR results from bias through multiple tests. Although AR is our reproduction of Recurrence*, the overall advancement of AR over Recurrence* proves the goodness of our framework and implementation. Participation of added three encoder layers in AR* improves model capacity and thus contributes to higher accuracy. A simple linear header already enables NAR to parallel the decoding; nevertheless, it dramatically reduces performance, especially in AES.

| **Method** | **AOR** ($N = 10$, $L = 5$, $D = 10$K) | | | **AES** ($N = 100$, $L = 5$, $D = 10$K) | | **AEC** ($N = 10$, $L = 5$, $D = 10$K) | | |
|---|---|---|---|---|---|---|---|---|
| | **Tok. Acc. %** | **Seq. Acc. %** | **Eq. Acc. %** | **Tok. Acc. %** | **Eq. Acc. %** | **Tok. Acc. %** | **Seq. Acc. %** | **Eq. Acc. %** |
| End2end | – | – | 29.33 | 84.60 | 25.20 | 88.08 | 57.27 | 57.73 |
| Tagging | – | – | 51.40 | 87.00 | 36.67 | 84.46 | 46.93 | 47.33 |
| Recurrence | – | – | 58.53 | 98.63 | 87.73 | 83.64 | 57.47 | 58.27 |
| Recurrence* | 60.30 ± 1.30 | 27.31 ± 1.33 | 56.73 ± 1.33 | 79.82 ± 0.37 | 22.28 ± 0.52 | 82.32 ± 0.56 | 41.72 ± 0.74 | 42.13 ± 0.75 |
| AR | 61.85 ± 0.51 | 28.83 ± 1.14 | 59.09 ± 0.95 | 88.12 ± 2.37 | 37.05 ± 6.57 | **82.61 ± 0.53** | 45.81 ± 0.36 | 46.31 ± 0.31 |
| AR* | 62.51 ± 0.62 | **30.85 ± 0.41** | 61.35 ± 0.33 | 99.27 ± 0.32 | 93.57 ± 2.91 | 82.29 ± 0.39 | 45.99 ± 0.49 | 46.35 ± 0.52 |
| NAR | 59.72 ± 0.70 | 24.16 ± 1.16 | 51.64 ± 1.97 | 83.87 ± 1.60 | 29.49 ± 2.51 | 80.28 ± 0.76 | 44.91 ± 1.71 | 45.40 ± 1.78 |
| NAR* | **62.81 ± 0.89** | 30.13 ± 1.31 | **61.45 ± 1.61** | **99.51 ± 0.13** | **95.67 ± 0.93** | 81.82 ± 0.68 | **45.97 ± 1.07** | **46.43 ± 1.10** |
| AR +TA | 62.35 ± 0.61 | 32.28 ± 0.67 | 63.56 ± 1.06 | 88.05 ± 1.20 | 38.39 ± 3.45 | **83.94 ± 0.42**∗ | 49.36 ± 1.23 | 49.83 ± 1.21 |
| AR* +TA | 62.58 ± 0.63 | 33.01 ± 1.31 | 65.73 ± 1.38 | 99.44 ± 0.27 | 95.24 ± 2.38 | 83.39 ± 0.74 | 48.95 ± 0.65 | 49.47 ± 0.73 |
| NAR +TA | 61.30 ± 0.86 | 32.04 ± 1.99 | 63.75 ± 2.08 | 90.38 ± 2.21 | 47.91 ± 8.18 | 81.36 ± 0.40 | 48.01 ± 1.07 | 48.47 ± 1.15 |
| NAR* +TA | **63.48 ± 0.38**$^*$ | **34.23 ± 0.92**$^*$ | **67.13 ± 0.99**$^*$ | **99.58 ± 0.15**$^*$ | **96.44 ± 1.29**$^*$ | 82.70 ± 0.42 | **49.64 ± 0.59**$^*$ | **50.15 ± 0.55**$^*$ |

Table 3: Evaluation results on AOR, AES, and AEC with specific $N$, $L$, and $D$. The token and sequence accuracy for AOR were not reported, thus we leave these positions blank here. With or without TA, our proposed NAR* achieves the best performance in terms of equation accuracy across the board.

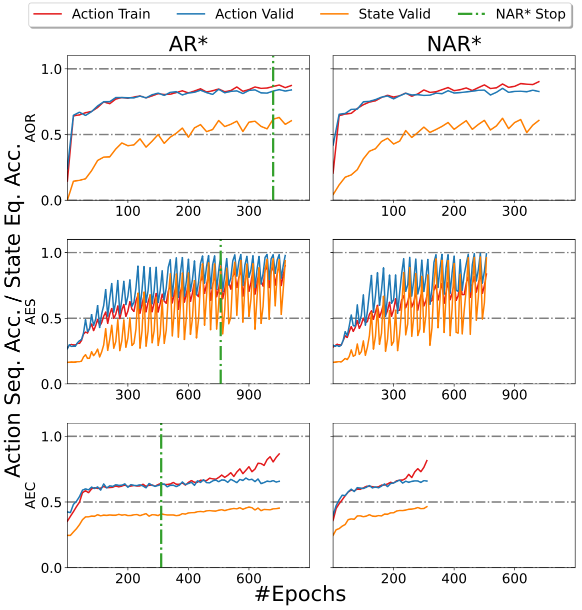

Figure 4: The learning curve of AR* (left column) and NAR* (right column) across AE tasks (rows). The red and blue lines represent the training on actions w.r.t sequence accuracy. The orange line stands for the validation on returned states w.r.t equation accuracy. The dashed line in green marks the earlier stop epoch of NAR* than that of AR* during training.

**Non-autoregressive.** What stands out is the dominance of NAR*, achieving 61.45%, 95.67%, and 46.43% in terms of equation accuracy for AOR, AES, and AEC, separately. Particularly in AES, its better performance over AR* by more than 2.1% equation accuracy underlines the success of NAR* in capturing the interdependencies among target tokens. Its superiority with respect to equation accuracy boosting by around 66.18% over NAR highlights the contributions of D2 again.

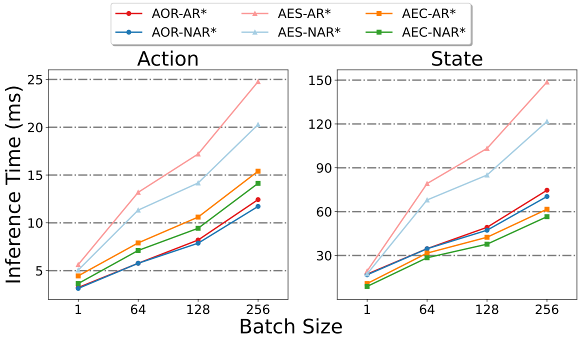

Figure 5: Inference time of AR* and NAR* to predict action (left) and return state (right) across AE tasks.

**Trajectory augmentation.** As expected, the incorporation of TA consistently promotes the accuracy of all models in our learning regime throughout AE tasks. Taking NAR as an example, training with TA brings it a substantial equation accuracy gain, remarkably up to 18.42% in AES. Even more, it pushes the gap between NAR* and the other baselines. The most notable advance comes from AOR, where NAR* outperforms AR* by a substantial margin of 5.68% equation accuracy. It appears that TA is more effective for non-autoregressive models than autoregressive ones.

## 4 Analysis

We conduct extensive sensitivity analyses to better illustrate and understand our methods.

### 4.1 Efficiency

From the learning curve (Figure 4) and inference time (Figure 5) of AR* and NAR* in AE, in addition to a higher accuracy, we find NAR* needs less number of training epochs to converge and trigger the early stopping. The periodic fluctuation of the learning curve is the consequence of using a scheduler. When it comes to inference, NAR* saves much time for every step of action determi-

| Design | Action Sequence | Method | Tok. Acc. % | Eq. Acc. % |
|---|---|---|---|---|
| #1 | [Pos.$_{\text{L}}$, Pos.$_{\text{R}}$, Tok.] | AR* | $99.27 \pm 0.32$ | $93.57 \pm 2.91$ |
| | | NAR* | $\mathbf{99.51 \pm 0.13}$ | $\mathbf{95.67 \pm 0.93}$ |
| | | AR* +TA | $99.44 \pm 0.27$ | $95.24 \pm 2.38$ |
| | | NAR* +TA | $\mathbf{99.58 \pm 0.15}^*$ | $\mathbf{96.44 \pm 1.29}^*$ |
| #2 | [Pos.$_{\text{L}}$, Tok., Pos.$_{\text{R}}$] | AR* | $99.08 \pm 0.93$ | $92.35 \pm 7.21$ |
| | | NAR* | $\mathbf{99.50 \pm 0.27}$ | $\mathbf{95.55 \pm 2.28}$ |
| | | AR* +TA | $99.52 \pm 0.29$ | $95.68 \pm 2.49$ |
| | | NAR* +TA | $\mathbf{99.54 \pm 0.20}^*$ | $\mathbf{95.97 \pm 1.64}^*$ |
| #3 | [Tok., Pos.$_{\text{L}}$, Pos.$_{\text{R}}$] | AR* | $98.06 \pm 0.79$ | $83.79 \pm 6.25$ |
| | | NAR* | $\mathbf{99.53 \pm 0.14}$ | $\mathbf{95.99 \pm 0.81}$ |
| | | AR* +TA | $98.43 \pm 0.49$ | $87.29 \pm 3.70$ |
| | | NAR* +TA | $\mathbf{99.61 \pm 0.06}^*$ | $\mathbf{96.55 \pm 0.46}^*$ |

Table 4: Evaluation of AR* and NAR* in AES across three action designs that vary from each other by token order. They directs to the same operation with Pos.$_{\text{L}}$/Pos.$_{\text{R}}$/Tok. denoting left parenthesis/right parenthesis/target token.

nation and ends up returning the edited state faster. As AR* and NAR* share exactly the same encoder structure, we conclude that D2 contributes to the advanced efficiency.

### 4.2 Action design

Due to the liberty of sequence generation, the same operation can be represented as different action sequences. In AES, the operation, instructing to substitute tokens between left and right parentheses with the required token, can fit the three action designs in Table 4, where Pos.$_{\text{L}}$, Pos.$_{\text{R}}$, and Tok. denote the positions of two parentheses and the target token. Design #1 is the default one. A simple swap of action tokens offers designs #2 and #3.

AR* severely suffers such perturbation, causing an equation accuracy decline by $9.78\%$ in #3. Contrastly, NAR* holds around its results and even slightly improves to $95.99\%$ in #3. Despite the joining of TA, AR* still goes down from $95.24\%$ in #1 to $87.29\%$ in #3, while NAR* stays nearly consistent across three designs. It is reasonable that AR* is sensitive to the order of action tokens because the position information helps the inference of the target token. This also reflects that NAR* can catch the position information but with little dependence on token order. Such robustness allows greater freedom of action design.

### 4.3 Trajectory optimization

A better edit metric $\mathbf{E}$ often means a smaller action vocabulary space $|\mathcal{V}_{\mathcal{A}}|$, shorter trajectory length $T_{\text{max}}$, and, therefore, an easier IL. Taking AES as an instance, a SELF-action, replacing tokens enclosed in parentheses with the target one, actually is the compression of several Levenshtein-actions including multiple deletions and one substitution. Although either can serve as an expert policy, SELF causes a much shorter $T_{\text{max}}$ as indicated in Table 5. The change from SELF to Levenshtein brings on a longer $T_{\text{max}}$ and consequently a significant performance gap of $75.2\%$ and $77.74\%$ for AR* and NAR* in terms of equation accuracy. Doing one edit in 31 steps rather than 6 undoubtedly raises the difficulty of the imitation game.

As one more exploration, we introduce Longest Common Subsequence (LCS) as an alternative $\mathbf{E}$ to AEC. Token replacement is not allowed in LCS but in Levenshtein. A replacement action has to be decomposed as one deletion and one insertion in LCS. From this, LCS has a small $|\mathcal{V}_{\mathcal{A}}|$, while Levenshtein has a shorter $T_{\text{max}}$. We train NAR* with these two and report in Figure 6. For a clear comparison, the test set is divided into two groups. In w/o REPLACE, both yield the same $T_{\text{max}}$, but, in w/ REPLACE, Levenshtein takes a shorter $T_{\text{max}}$.

| Edit Metric E | $T_{\text{max}}$ | Method | Tok. Acc. % | Eq. Acc. % |
|---|---|---|---|---|
| SELF | 6 | AR* | $99.27 \pm 0.32$ | $93.57 \pm 2.91$ |
| | | NAR* | $\mathbf{99.51 \pm 0.13}$ | $\mathbf{95.67 \pm 0.93}$ |
| Levenshtein | 31 | AR* | $\mathbf{69.53 \pm 2.29}$ | $\mathbf{18.37 \pm 0.70}$ |
| | | NAR* | $67.58 \pm 0.87$ | $17.93 \pm 0.07$ |

Table 5: Evaluation of AR* and NAR* trained with edit metrics SELF and Levenshtein in AES. $\boldsymbol{T}_{\mathbf{max}}$ refers to the maximum length of expert trajectories.

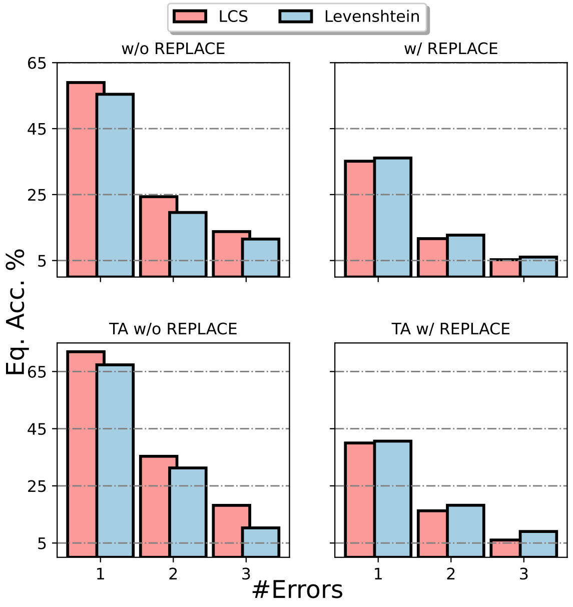

Figure 6: Evaluation of NAR* trained with edit metrics LCS and Levenshtein in AEC. Results are grouped by two trajectory lengths caused by whether the policy involves REPLACE.

| Decoder | AOR ($N = 10$, $L = 5$, $D = 10$K) | | | AES ($N = 100$, $L = 5$, $D = 10$K) | | AEC ($N = 10$, $L = 5$, $D = 10$K) | | |
|---|---|---|---|---|---|---|---|---|
| | **Tok. Acc. %** | **Seq. Acc. %** | **Eq. Acc. %** | **Tok. Acc. %** | **Eq. Acc. %** | **Tok. Acc. %** | **Seq. Acc. %** | **Eq. Acc. %** |
| Linear | $61.84 \pm 0.94$ | $28.55 \pm 1.57$ | $57.72 \pm 1.55$ | $99.41 \pm 0.26$ | $95.01 \pm 2.01$ | $81.35 \pm 0.92$ | $42.47 \pm 1.85$ | $42.81 \pm 1.87$ |
| Decoder$_0$ | $61.78 \pm 0.83$ | $28.20 \pm 1.57$ | $58.36 \pm 1.58$ | $99.24 \pm 0.23$ | $93.49 \pm 2.03$ | $80.84 \pm 0.66$ | $43.97 \pm 1.82$ | $44.32 \pm 1.82$ |
| Shared D2 | $61.74 \pm 0.71$ | $28.68 \pm 0.94$ | $58.05 \pm 1.01$ | $99.28 \pm 0.24$ | $93.85 \pm 2.14$ | $81.38 \pm 1.04$ | $43.64 \pm 2.03$ | $44.09 \pm 2.02$ |
| D2 (NAR*) | $\mathbf{62.81 \pm 0.89}$ | $\mathbf{30.13 \pm 1.31}$ | $\mathbf{61.45 \pm 1.61}$ | $\mathbf{99.51 \pm 0.13}$ | $\mathbf{95.67 \pm 0.93}$ | $\mathbf{81.82 \pm 0.68}$ | $\mathbf{45.97 \pm 1.07}$ | $\mathbf{46.43 \pm 1.10}$ |
| Linear +TA | $61.41 \pm 0.28$ | $31.75 \pm 0.93$ | $63.15 \pm 0.96$ | $99.42 \pm 0.17$ | $95.08 \pm 1.47$ | $81.54 \pm 0.66$ | $46.79 \pm 2.26$ | $47.33 \pm 2.30$ |
| Decoder$_0$ +TA | $62.50 \pm 1.24$ | $32.48 \pm 1.87$ | $64.47 \pm 1.88$ | $99.47 \pm 0.13$ | $95.33 \pm 1.13$ | $82.02 \pm 0.40$ | $46.80 \pm 2.04$ | $47.32 \pm 1.91$ |
| Shared D2 +TA | $61.64 \pm 0.87$ | $31.21 \pm 0.34$ | $62.77 \pm 0.85$ | $99.53 \pm 0.12$ | $95.91 \pm 1.25$ | $81.80 \pm 0.47$ | $47.23 \pm 1.07$ | $47.61 \pm 1.14$ |
| D2 (NAR*) +TA | $\mathbf{63.48 \pm 0.38}^*$ | $\mathbf{34.23 \pm 0.92}^*$ | $\mathbf{67.13 \pm 0.99}^*$ | $\mathbf{99.58 \pm 0.15}^*$ | $\mathbf{96.44 \pm 1.29}^*$ | $\mathbf{82.70 \pm 0.42}^*$ | $\mathbf{49.64 \pm 0.59}^*$ | $\mathbf{50.15 \pm 0.55}^*$ |

Table 6: Evaluation of agents equipped with same encoders but different decoders on AE benchmarks.

In the former, LCS exceeds Levenshtein with or without TA. In the latter, the opposite is true, where Levenshtein outperforms LCS under the same condition. This support our assumption at the beginning that an appropriate $\mathbf{E}$, leading to a small $|\mathcal{V}_{\mathcal{A}}|$ and a short $T_{\max}$, is conducive to IL, suggesting trajectory optimization an interesting future work.

### 4.4 Dual decoders

As an ablation study, we freeze the encoder of NAR* and vary its decoder to reveal the contributions of each component in D2. As listed in Table 6, replacing the decoder with a linear layer leads to Linear and removing the second decoder from NAR* results in Decoder$_0$. Moreover, sharing the parameters between two decoders of NAR* gives the Shared D2. All of them can parallel the decoding process. We then borrow the setup of Section 3 and test them on AE.

Among four decoders, NAR* dominates three imitation games. The performance decrease caused by shared parameters is more significant than expected. Besides the reason that saved parameters limit the model capacity, another potential one is the input mismatch of two decoders. The input of decoder$_0$ is the projected context from the linear layer after the encoder, yet that of decoder$_1$ is the embedded prediction from the embedding layer. When incorporating TA, we find the same trend persists. The gap between NAR* and the others is even more apparent. Since they share the same encoder, such a gap clarifies the benefits of D2.

## 5 Conclusion

We reformulate text editing as an imitation game defined by an MDP to allow action design at the sequence-level. We propose D2, a non-autoregressive decoder for state-action learning, coupled with TG for data translation and TA for distribution shift alleviation. Achievements on AE benchmarks evidence the advantages of our methods in performance, efficiency, and robustness. Sequence-level actions are arguably more controllable, interpretable, and similar to human behavior. Turning tasks into games that agents feel more comfortable with sheds light on future studies in the direction of reinforcement learning in the application of text editing. The involvement of a reward function, the optimization of the trajectories, the design of sequence-level actions, and their applications in more practical tasks, to name a few, are interesting for future work. Suggesting text editing as a new testbed, we hope our findings will shed light on future studies in reinforcement learning applying to natural language processing.

## Limitations

Each time the state is updated, the agent can get immediate feedback on the previous action and thus a dynamic context representation during the editing. This also means that the encoder (e.g., a heavy pretrained language model) will be called multiple times to refresh the context matrix. Consequently, as the trajectory grows, the whole task becomes slow even though we have paralleled the decoding process. Meanwhile, applying our methods in more realistic editing tasks (e.g., grammatical error correction) remains a concern and needs to be explored in the near future.

## Acknowledgements

We gratefully appreciate Che Wang (Watcher), Yichen Gong, and Hui Xue for sharing their pearls of wisdom. We also would like to express our special thanks of graitude to Yingying Huo for the support, as well as EMNLP anonymous reviewers for their constructive feedback. This work was supported by Shining Lab and Alibaba Group.

## References

Sweta Agrawal and Marine Carpuat. 2022. An imitation learning curriculum for text editing with non-autoregressive models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7550–7563, Dublin, Ireland. Association for Computational Linguistics.

Sweta Agrawal, Weijia Xu, and Marine Carpuat. 2021. A non-autoregressive edit-based approach to controllable text simplification. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 3757–3769, Online. Association for Computational Linguistics.

Abhijeet Awasthi, Sunita Sarawagi, Rasna Goyal, Sabyasachi Ghosh, and Vihari Piratla. 2019. Parallel iterative edit models for local sequence transduction. 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 4260–4270, Hong Kong, China. Association for Computational Linguistics.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

S.R.K. Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. 2009. Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 82–90, Suntec, Singapore. Association for Computational Linguistics.

Yangyi Chen, Jin Su, and Wei Wei. 2021. Multi-granularity textual adversarial attack with behavior cloning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4511–4526, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yue Dong, Zichao Li, Mehdi Rezagholizadeh, and Jackie Chi Kit Cheung. 2019. EditNTS: An neural programmer-interpreter model for sentence simplification through explicit editing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3393–3402, Florence, Italy. Association for Computational Linguistics.

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. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5599–5610, Online. Association for Computational Linguistics.

Tao Ge, Furu Wei, and Ming Zhou. 2018. Fluency boost learning and inference for neural grammatical error correction. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1055–1065, Melbourne, Australia. Association for Computational Linguistics.

Jiatao Gu, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. 2018. Non-autoregressive neural machine translation. In *International Conference on Learning Representations*.

Jiatao Gu, Changhan Wang, and Junbo Zhao. 2019. Levenshtein transformer. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 11181–11191. Curran Associates, Inc.

Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep reinforcement learning for syntactic error repair in student programs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):930–937.

Kelvin Guu, Panupong Pasupat, Evan Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1051–1062, Vancouver, Canada. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Shubha Guha, and Kenneth Heafield. 2018. Approaching neural grammatical error correction as a low-resource machine translation task. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 595–606, New Orleans, Louisiana. Association for Computational Linguistics.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Ryosuke Kohita, Akifumi Wachi, Yang Zhao, and Ryuki Tachibana. 2020. Q-learning with language model for edit-based unsupervised summarization. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 470–484, Online. Association for Computational Linguistics.

Jiquan Li, Junliang Guo, Yongxin Zhu, Xin Sheng, Deqiang Jiang, Bo Ren, and Linli Xu. 2022. Sequence-to-action: Grammatical error correction with action guided sequence generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(10):10974–10982.

Zhongkun Liu, Pengjie Ren, Zhumin Chen, Zhaochun Ren, Maarten de Rijke, and Ming Zhou. 2021. Learning to ask conversational questions by optimizing Levenshtein distance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5638–5650, Online. Association for Computational Linguistics.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

Jonathan Mallinson, Aliaksei Severyn, Eric Malmi, and Guillermo Garrido. 2020. FELIX: Flexible text editing through tagging and insertion. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1244–1255, Online. Association for Computational Linguistics.

Eric Malmi, Yue Dong, Jonathan Mallinson, Aleksandr Chuklin, Jakub Adamek, Daniil Mirylenka, Felix Stahlberg, Sebastian Krause, Shankar Kumar, and Aliaksei Severyn. 2022. Text generation with text-editing models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Tutorial Abstracts*, pages 1–7, Seattle, United States. Association for Computational Linguistics.

Eric Malmi, Sebastian Krause, Sascha Rothe, Daniil Mirylenka, and Aliaksei Severyn. 2019. Encode, tag, realize: High-precision text editing. 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 5054–5065, Hong Kong, China. Association for Computational Linguistics.

Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. 2021. Copy that! editing sequences by copying spans. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(15):13622–13630.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1310–III–1318. JMLR.org.

Dean A. Pomerleau. 1991. Efficient Training of Artificial Neural Networks for Autonomous Navigation. *Neural Computation*, 3(1):88–97.

Lutz Prechelt. 1998. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer.

Machel Reid and Victor Zhong. 2021. LEWIS: Levenshtein editing for unsupervised text style transfer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 3932–3944, Online. Association for Computational Linguistics.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings.

Stefan Schaal. 1996. Learning from demonstration. In *Advances in Neural Information Processing Systems*, volume 9. MIT Press.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada. Association for Computational Linguistics.

Ning Shi, Wei Wang, Boxin Wang, Jinfeng Li, Xiangyu Liu, and Zhouhan Lin. 2021. Incorporating External POS Tagger for Punctuation Restoration. In *Proc. Interspeech 2021*, pages 1987–1991.

Ning Shi, Ziheng Zeng, Haotian Zhang, and Yichen Gong. 2020. Recurrent inference in text editing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1758–1769, Online. Association for Computational Linguistics.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Felix Stahlberg and Shankar Kumar. 2020. Seq2Edits: Sequence transduction using span-level edit operations. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5147–5159, Online. Association for Computational Linguistics.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc.

Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning structural edits via incremental tree transformations. In *International Conference on Learning Representations*.

Wei Zhao, Liang Wang, Kewei Shen, Ruoyu Jia, and Jingming Liu. 2019. Improving grammatical error correction via pre-training a copy-augmented architecture with unlabeled data. 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 156–165, Minneapolis, Minnesota. Association for Computational Linguistics.