Levenshtein Transformer

Jiatao Gu[†]**, Changhan Wang**[†]**, and Jake Zhao (Junbo)**[‡◇]
[†]Facebook AI Research
[‡]New York University [◇]Tigerobo Inc.
[†]{jgu, changhan}@fb.com [‡]jakezhao@cs.nyu.edu

Abstract

Modern neural sequence generation models are built to either generate tokens step-by-step from scratch or (iteratively) modify a sequence of tokens bounded by a fixed length. In this work, we develop Levenshtein Transformer, a new partially autoregressive model devised for more flexible and amenable sequence generation. Unlike previous approaches, the atomic operations of our model are *insertion* and *deletion*. The combination of them facilitates not only generation but also sequence refinement allowing dynamic length changes. We also propose a set of new training techniques dedicated at them, effectively exploiting one as the other's learning signal thanks to their complementary nature. Experiments applying the proposed model achieve comparable performance but much-improved efficiency on both generation (e.g. machine translation, text summarization) and refinement tasks (e.g. automatic post-editing). We further confirm the flexibility of our model by showing a Levenshtein Transformer trained by machine translation can straightforwardly be used for automatic post-editing.

1 Introduction

Neural sequence generation models are widely developed and deployed in tasks such as machine translation (Bahdanau et al., 2015; Vaswani et al., 2017). As we examine the current frameworks, the most popular autoregressive models generate tokens step-by-step. If not better, recent non-autoregressive approaches (Gu et al., 2018; Kaiser et al., 2018; Lee et al., 2018) have proved it possible to perform generation within a much smaller number of decoding iterations.

In this paper, we propose Levenshtein Transformer (LevT), aiming to address the lack of flexibility of the current decoding models. Notably, in the existing frameworks, the length of generated sequences is either fixed or monotonically increased as the decoding proceeds. This remains incompatible with human-level intelligence where humans can revise, replace, revoke or delete any part of their generated text. Hence, LevT is proposed to bridge this gap by breaking the in-so-far standardized decoding mechanism and replacing it with two atomic operations — *insertion* and *deletion*.

We train the LevT using imitation learning. The resulted model contains two policies and they are executed in an alternate manner. Empirically, we show that LevT achieves comparable or better results than a standard Transformer model on machine translation and summarization, while maintaining the efficiency advantages benefited from parallel decoding similarly to (Lee et al., 2018). With this model, we argue that the decoding becomes more flexible. For example, when the decoder is given an empty token, it falls back to a normal sequence generation model. On the other hand, the decoder acts as a refinement model when the initial state is a low-quality generated sequence. Indeed, we show that a LevT trained from machine translation is directly applicable to translation post-editing without any change. This would not be possible with any framework in the literature because generation and refinement are treated as two different tasks due to the model's inductive bias.

Preprint. Under review.

One crucial component in LevT framework is the learning algorithm. We leverage the characteristics of *insertion* and *deletion* — they are complementary but also adversarial. The algorithm we propose is called "dual policy learning". The idea is that when training one policy (insertion or deletion), we use the output from its adversary at the previous iteration as input. An *expert* policy, on the other hand, is drawn to provide a correction signal. Despite that, in theory, this learning algorithm is applicable to other imitation learning scenarios where a dual adversarial policy exists, in this work we primarily focus on a proof-of-concept of this algorithm landing at training the proposed LevT model.

To this end, we summarize the contributions as follows:

- We propose Levenshtein Transformer (LevT), a new sequence generation model composed of the insertion and deletion operations. This model achieves comparable or better results than a strong Transformer baseline in both machine translation and text summarization, but with much better efficiency (up to $\times 5$ speed-up);
- We propose a corresponding learning algorithm under the theoretical framework of imitation learning, tackling the complementary and adversarial nature of the dual policies;
- We recognize our model as a pioneer attempt to unify sequence generation and refinement, thanks to its built-in flexibility. With this unification, we empirically validate the feasibility of applying a LevT model trained by machine translation directly to translation post-editing, without any change.

2 Problem Formulation

2.1 Sequence Generation and Refinement

We unify the general problems of sequence generation and refinement by casting them to a Markov Decision Process (MDP) defined by a tuple $(\mathcal{Y}, \mathcal{A}, \mathcal{E}, \mathcal{R}, \boldsymbol{y_0})$. We consider the setup consisting an agent interacting with an environment $\mathcal{E}$ which receives the agent's editing actions and returns the modified sequence. We define $\mathcal{Y} = \mathcal{V}^{N_{\max}}$ as a set of discrete sequences up to length $N_{\max}$ where $\mathcal{V}$ is a vocabulary of symbols. At every decoding iteration, the agent receives an input $\boldsymbol{y}$ drawn from scratch or uncompleted generation, chooses an action $\boldsymbol{a}$ and gets a reward r. We use $\mathcal{A}$ to denote the set of actions and $\mathcal{R}$ for the reward function. Generally the reward function $\mathcal{R}$ measures the distance between the generation and the ground-truth sequence, $\mathcal{R}(\boldsymbol{y}) = -\mathcal{D}(\boldsymbol{y}, \boldsymbol{y}^*)$ which can be any distance measurement such as the Levenshtein distance (Levenshtein, 1965). It is crucial to incorporate $\boldsymbol{y_0} \in \mathcal{Y}$ into the our formulation. As the initial sequence, the agent receives—when $\boldsymbol{y_0}$ is an already generated sequence from another system, the agent essentially learns to do refinement while it falls back to generation if $\boldsymbol{y_0}$ is an empty sequence. The agent is modeled by a policy, π, that maps the current generation over a probability distribution over $\mathcal{A}$. That is, $\pi : \mathcal{Y} \rightarrow P(\mathcal{A})$.

2.2 Actions: Deletion & Insertion

Following the above MDP formulation, with a subsequence $\boldsymbol{y}^k = (y_1, y_2, ..., y_n)$, the two atomic actions – *deletion* and *insertion* – are called to generate $\boldsymbol{y}^{k+1} = \mathcal{E}(\boldsymbol{y}^k, \boldsymbol{a}^{k+1})$. Here we let y_1 and y_n be special symbols `<s>` and `</s>`, respectively. Since we mainly focus on the policy of a single round generation, the superscripts are normally omitted in this section. For conditional generation like MT, our policy also includes an input of source information $\boldsymbol{x}$ which is also omitted here.

Deletion The deletion policy reads the input sequence $\boldsymbol{y}$, and for every token $y_i \in \boldsymbol{y}$, the deletion policy $\pi^{\text{del}}(d|i, \boldsymbol{y})$ makes a binary decision which is 1 (delete this token) or 0 (keep it). We additionally constrain $\pi^{\text{del}}(0|1, \boldsymbol{y}) = \pi^{\text{del}}(0|n, \boldsymbol{y}) = 1$ to avoid sequence boundary being broken.

Insertion It is slightly more complex to build the insertion atomic because it involves two phases: *placeholder* prediction and *token* prediction so that it is able to insert multiple tokens at the same slot. First, among all the possible inserted slots (y_i, y_{i+1}) in $\boldsymbol{y}$, $\pi^{\text{plh}}(p|i, \boldsymbol{y})$ predicts the possibility of adding one or several placeholders. In what follows, for every placeholder predicted as above, a token prediction policy $\pi^{\text{tok}}(t|i, \boldsymbol{y})$ replaces the placeholders with actual tokens in the vocabulary.

Policy combination Recall that our two atomic operations are complementary. Hence we combine them in an alternate fashion. For example in sequence generation from the empty, insertion policy is first called and it is followed by deletion, and then repeat till the certain stopping condition is fulfilled. Indeed, it is possible to leverage the parallelism in this combination. We essentially decompose

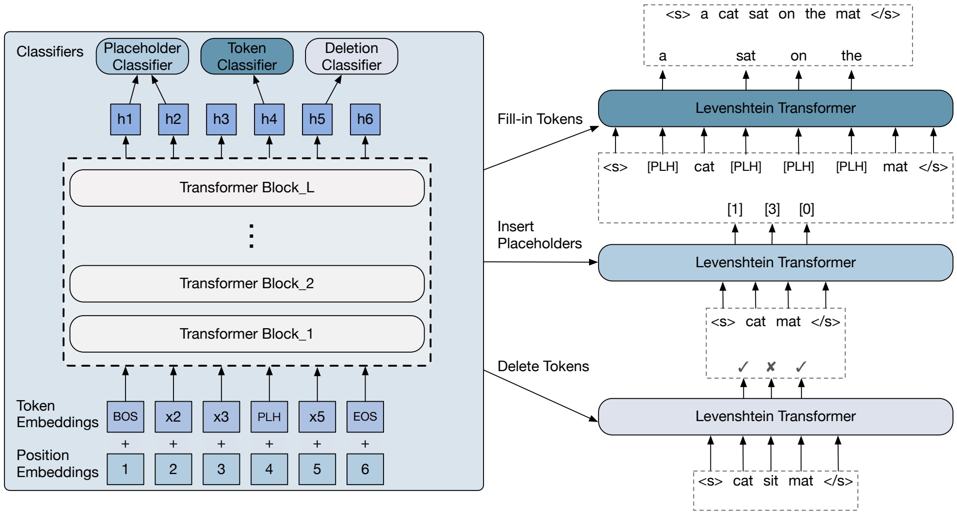


Figure 1: The overall framework of the decoder of the proposed Levenshtein Transformer. We show how the same architecture can be applied for three different tasks with specific classifiers. For simplicity, the attention between the encoder outputs is omitted within each Transformer-Block.

one iteration of our sequence generator into three phases: "delete tokens – insert placeholders – replace placeholders with new tokens". Within each stage, all operations are performed in parallel. More precisely, given the current sequence $\boldsymbol{y} = (y_0, \ldots, y_n)$, and suppose the action to predict is $\boldsymbol{a} = \{\underbrace{d_0, \ldots d_n}_{\boldsymbol{d}}; \underbrace{p_0, \ldots, p_{n-1}}_{\boldsymbol{p}}; \underbrace{t_0^1, \ldots t_0^{p_0}, \ldots, t_{n-1}^{p_{n-1}}}_{\boldsymbol{t}}\}$, we our policy for one iteration is:

$$\pi(\boldsymbol{a}|\boldsymbol{y}) = \prod_{d_i \in \boldsymbol{d}} \pi^{\text{del}}(d_i|i, \boldsymbol{y}) \cdot \prod_{p_i \in \boldsymbol{p}} \pi^{\text{plh}}(p_i|i, \boldsymbol{y}') \cdot \prod_{t_i \in \boldsymbol{t}} \pi^{\text{tok}}(t_i|i, \boldsymbol{y}''), \tag{1}$$

where $\boldsymbol{y}' = \mathcal{E}(\boldsymbol{y}, \boldsymbol{d})$ and $\boldsymbol{y}'' = \mathcal{E}(\boldsymbol{y}', \boldsymbol{p})$. We parallelize the computation within each sub-tasks.

3 Levenshtein Transformer

In this section, we cover the specs of Levenshtein Transformer and the dual-policy learning algorithm. Overall our model takes a sequence of tokens (or none) as the input then iteratively *modify* it by alternating between insertion and deletion, until the two policies combined converge. We describe the detailed learning and inference algorithms in the Appendix.

3.1 Model

We use Transformer (Vaswani et al., 2017) as the basic building block. For conditional generation, the source $\boldsymbol{x}$ is included in each TransformerBlock. The states from the l-th block are:

$$\boldsymbol{h}_0^{(l+1)}, \boldsymbol{h}_1^{(l+1)}, ..., \boldsymbol{h}_n^{(l+1)} = \begin{cases} E_{y_0} + P_0, E_{y_1} + P_1, ..., E_{y_n} + P_n, & l = 0 \\ \text{TransformerBlock}_l(\boldsymbol{h}_0^{(l)}, \boldsymbol{h}_1^{(l)}, ..., \boldsymbol{h}_n^{(l)}), & l > 0 \end{cases} \tag{2}$$

where $E \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{model}}}$ and $P \in \mathbb{R}^{N_{\max} \times d_{\text{model}}}$ are the token and position embeddings, respectively.

Policy Classifiers The decoder outputs $(\boldsymbol{h}_0, \boldsymbol{h}_2, ..., \boldsymbol{h}_n)$ are passed to three policy classifiers:

1. *Deletion Classifier*: LevT scans over the input tokens (except for the boundaries) and predict "deleted" (0) or "kept" (1) for each token position,

$$\pi_\theta^{\text{del}}(d|i, \boldsymbol{y}) = \text{softmax}\left(\boldsymbol{h}_i \cdot A^\top\right), \quad i = 1, \ldots n-1, \tag{3}$$

where $A \in \mathbb{R}^{2 \times d_{\text{model}}}$, and we always keep the boundary tokens.

2. *Placeholder Classifier*: LevT predicts the number of tokens to be inserted at every consecutive position pairs, by casting the representation to a categorical distribution:

$$\pi_\theta^{\text{plh}}(p|i, \boldsymbol{y}) = \text{softmax}\left(\texttt{concat}(\boldsymbol{h}_i, \boldsymbol{h}_{i+1}) \cdot B^\top\right), \;\; i = 0, \dots n-1, \tag{4}$$

   where $B \in \mathbb{R}^{(K_{\max}+1)\times(2d_{\text{model}})}$. Based on the number ($0 \sim K_{\max}$) of tokens it predicts, we insert the considered number of placeholders at the current position. In our implementation, placehoder is represented by a special token <PLH> which was reserved in the vocabulary.

3. *Token Classifier*: following the placeholder prediction, LevT needs to fill in tokens replacing all the placeholders. This is achieved by training a token predictor as follow:

$$\pi_\theta^{\text{tok}}(t|i, \boldsymbol{y}) = \text{softmax}\left(\boldsymbol{h}_i \cdot C^\top\right), \;\; \forall y_i = \texttt{<PLH>}, \tag{5}$$

   where $C \in \mathbb{R}^{|\mathcal{V}|\times d_{\text{model}}}$ with parameters being shared with the embedding matrix.

Early Exit Although it is parameter-efficient to share the same Transformer architecture across the above three heads, there is room for improvement as one decoding iteration requires three full passes of the network. To make trade-off between performance and computational cost, we propose to perform *early exit* (attaching the head to an intermediate block instead of the last one) for π^{del} and π^{plh} while keeping π^{tok} always based on the last block, considering that token prediction is usually more challenging than the other two tasks.

3.2 Dual-policy Learning

Imitation Learning We use imitation learning to train the Levenshtein Transformer. Essentially we let the agent imitate the behaviors that we draw from some expert policy π^*. The expert policy is derived from direct usage of ground-truth targets or less noisy version filtered by sequence distillation (Kim and Rush, 2016). The objective is to maximize the following expectation:

$$\underbrace{\mathbb{E}_{\substack{\boldsymbol{y}_{\text{del}}\sim d_{\tilde{\pi}_{\text{del}}} \\ \boldsymbol{d}^*\sim\pi^*}} \sum_{d_i^*\in\boldsymbol{d}^*} \log \pi_\theta^{\text{del}}(d_i^*|i, \boldsymbol{y}_{\text{del}})}_{\textit{Deletion Objective}} + \underbrace{\mathbb{E}_{\substack{\boldsymbol{y}_{\text{ins}}\sim d_{\tilde{\pi}_{\text{ins}}} \\ \boldsymbol{p}^*,\boldsymbol{t}^*\sim\pi^*}} \left[\sum_{p_i^*\in\boldsymbol{p}^*} \log \pi_\theta^{\text{plh}}(p_i^*|i, \boldsymbol{y}_{\text{ins}}) + \sum_{t_i^*\in\boldsymbol{t}^*} \log \pi_\theta^{\text{tok}}(t_i^*|i, \boldsymbol{y}'_{\text{ins}})\right]}_{\textit{Insertion Objective}},$$

where $\boldsymbol{y}'_{\text{ins}}$ is the output after inserting palceholders $\boldsymbol{p}^*$ upon $\boldsymbol{y}_{\text{ins}}$. $\tilde{\pi}_{\text{del}}$, $\tilde{\pi}_{\text{ins}}$ are the *roll-in* polices and we repeatedly draw states (sequences) from their induced state distribution $d_{\tilde{\pi}_{\text{del}}}, d_{\tilde{\pi}_{\text{ins}}}$. These states are first executed by the *expert* policy returning the suggested actions by the expert, and then we maximize the conditional log-likelihood over them.

Roll-in Policy By definition, the *roll-in* policy determines the state distribution fed to π_θ during training. In this work, we have two strategies to construct the roll-in policy — adding noise to the ground-truth or using the output from the adversary policy. Figure 2 shows a diagram of this learning paradigm. We formally write down the roll-in policies as follows.

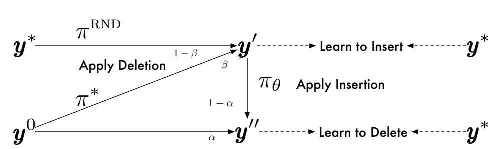


Figure 2: The data-flow of learning.

1. *Learning to Delete*: we design the $\tilde{\pi}_{\text{del}}$ as a stochastic mixture between the initial input $\boldsymbol{y}^0$ or the output by applying insertion from the model with some mixture factor $\alpha \in [0, 1]$:

$$d_{\tilde{\pi}_{\text{del}}} = \{\boldsymbol{y}^0 \;\; \text{if} \;\; u < \alpha \;\; \text{else} \;\; \mathcal{E}\left(\mathcal{E}\left(\boldsymbol{y}', \boldsymbol{p}^*\right), \tilde{\boldsymbol{t}}\right), \;\; \boldsymbol{p}^* \sim \pi^*, \tilde{\boldsymbol{t}} \sim \pi_\theta\} \tag{6}$$

   where $u \sim \text{Uniform}[0, 1]$ and $\boldsymbol{y}'$ is any sequence ready to insert tokens. $\tilde{\boldsymbol{t}}$ is obtained by sampling instead of doing argmax from Eq. (5).

2. *Learning to Insert*: similar to the deletion step, we apply a mixture of the deletion output and a random word dropping sequence of the round-truth, inspired by recent advances of training masked language model (Devlin et al., 2018). We use random dropping as a form of noise injection to encourage more exploration. Let $\beta \in [0, 1]$ and $u \sim \text{Uniform}[0, 1]$,

$$d_{\tilde{\pi}_{\text{ins}}} = \{\mathcal{E}\left(\boldsymbol{y}^0, \boldsymbol{d}^*\right), \;\; \boldsymbol{d}^* \sim \pi^* \;\; \text{if} \;\; u < \beta \;\; \text{else} \;\; \mathcal{E}\left(\boldsymbol{y}^*, \tilde{\boldsymbol{d}}\right), \;\; \tilde{\boldsymbol{d}} \sim \pi^{\text{RND}}\} \tag{7}$$

Expert Policy It is crucial to construct an expert policy in imitation learning which cannot be too hard or too weak to learn from. Specifically, we considered two types of experts:

1. *Oracle*: One way is to build an oracle which accesses to the ground-truth sequence. It returns the optimal actions $\boldsymbol{a}^*$ (either oracle insertion $\boldsymbol{p}^*, \boldsymbol{t}^*$ or oracle deletion $\boldsymbol{d}^*$) by:
$$\boldsymbol{a}^* = \arg\min_{\boldsymbol{a}} \mathcal{D}(\boldsymbol{y}^*, \mathcal{E}(\boldsymbol{y}, \boldsymbol{a})) \tag{8}$$
Here, we use the Levenshtein distance (Levenshtein, 1965)[1] as $\mathcal{D}$ considering it is possible to obtain the action suggestions efficiently by dynamic programming.
2. *Teacher Model*: We also explore to use another teacher model to provide expert policy, which is known as knowledge distillation (Kim and Rush, 2016). This technique has been widely used in previous approaches for nonauoregressive generation (Gu et al., 2018). More precisely, we first train an autoregressive teacher model using the same datasets and then replace the ground-truth sequence $\boldsymbol{y}^*$ by the beam-search result of this teacher-model, $\boldsymbol{y}^{\text{AR}}$. We use the same mechanism to find the suggested option as using the ground-truth oracle.

3.3 Inference

Greedy Decoding At inference time, we apply the trained model over the initial sequence $\boldsymbol{y}^0$ for several iterations. We greedily pick up the actions associated with high probabilities in Eq. (3)(4)(5). Moreover, we find that using search (instead of greedy decoding) does not yield much gain in LevT. This observation is quite opposite to what has been widely discovered in autoregressive decoding. We hypothesize this is because the local optimal point brought by greedy decoding in autoregressive models is often far from the optimality point globally. Search techniques resolve this issue with tabularization. In our case, however, because LevT inserts or deletes tokens dynamically, it could easily revoke the tokens that are found sub-optimal and re-insert better ones.

Termination Condition The decoding stops when one of the following conditions is fulfilled:

1. *Nothing to delete, Nothing to insert*: The policy chooses to keep all the current tokens, and predicts "empty" placeholders at everywhere.
2. *Direct-Loop*: Unfortunately, our MDP assumption cannot avoid the situations where the agent gets stuck in an infinite loop; i.e. the insertion and deletion counter each other and keep looping. Although this is not common, we terminate the decoding once this is spotted.
3. *Timeout*: We further set a maximum number of iterations (timeout) to guarantee a constant-time complexity in the worst case (Lee et al., 2018; Ghazvininejad et al., 2019).

Penalty for Empty Placeholders Similar to Stern et al. (2019), we add a penalty to insert "empty" placeholder in decoding. Overly inserting "empty" placeholders may result in shorter output. A penalty term $\gamma \in (0, 7]$ is subtracted from the logits of 0 in Eq. (4).

4 Experiments

We validate the efficiency, effectiveness, and flexibility of Levenshtein Transformer extensively across three different tasks — machine translation (MT), text summarization (TS) and automatic post-editing (APE) for machine translation, from both generation (§4.1) and refinement (§4.2) perspectives.

4.1 Sequence Generation

For the sequence generation perspective, we evaluate LevT model on MT and TS. As a special case, sequence generation assumes empty $\boldsymbol{y}^0 =$ <s></s> as input and no initial deletion is applied.

Data & Evaluation We use three diversified language pairs for MT experiments: WMT'16 Romanian-English (Ro-En)[2], WMT'14 English-German (En-De)[3] and WAT2017 Small-NMT English-Japanese (En-Ja)[4]. The TS experiments use preprocessed data from the Annotated En-

[1] We only consider the variant which only computes insertion and deletion. No substitution is considered.

[2] `http://www.statmt.org/wmt16/translation-task.html`

[3] `http://www.statmt.org/wmt14/translation-task.html`

[4] `http://lotus.kuee.kyoto-u.ac.jp/WAT/WAT2017/snmt/index.html`

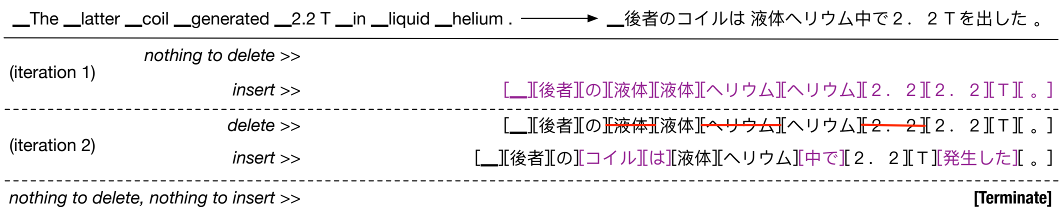


Figure 3: An example of WAT'17 En-Ja translation with two decoder iterations by LevT. We present the inserted tokens in purple and deleted tokens with ~~red strikethrough~~

.

	Dataset	Metric	Transformer greedy	Transformer beam4	Levenshtein Transformer oracle	Levenshtein Transformer teacher
Quality ↑	Ro-En	BLEU	31.67	32.30	**33.02**	–
	En-De	BLEU	26.02	26.56	24.43	**26.67**
	En-Ja	BLEU	42.86	**43.68**	42.36	43.17
	Gigaword	ROUGE-1	34.91	35.19	35.57	**36.08**
		ROUGE-2	17.05	17.58	17.11	**18.33**
		ROUGE-L	32.66	32.98	33.55	**33.81**
Speed ↓	Ro-En	Latency (ms) /I_{DEC}	326 / 27.1	349 / 27.1	**97 / 2.19**	–
	En-De	Latency (ms) /I_{DEC}	343 / 28.1	369 / 28.1	126 / 2.88	**92 / 2.05**
	En-Ja	Latency (ms) /I_{DEC}	261 / 22.6	306 / 22.6	112 / 2.61	**106 / 1.97**
	Gigaword	Latency (ms) /I_{DEC}	116 / 10.1	149 / 10.1	98 / 2.32	**84 / 1.73**

Table 1: Generation quality (BLEU ↑, ROUGE-1/2/L ↑) and latency (ms ↓) as well as the average number of decoder iterations (I_{DEC}) on the standard test sets for LevT and the autoregressive baseline (with both greedy and beam-search outputs). We show the results of LevT trained from both oracle and the autoregressive teacher model.

glish Gigaword (Rush et al., 2015)[5]. We learn byte-pair encoding (BPE, Sennrich et al., 2016) vocabulary on tokenized data. Detailed dataset statistics can be found in the Appendix. For evaluation metrics, we use BLEU (Papineni et al., 2002) for MT and ROUGE-1,2,L (Lin, 2004) for TS. Before computing the BLEU scores for Japanese output, we always segment Japanese words using KyTea [6].

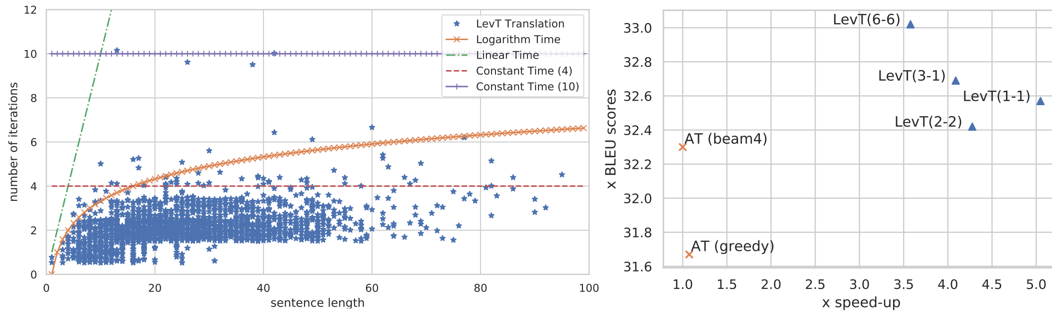
Models & Training We adopt the model architecture of Transformer base (Vaswani et al., 2017) for the proposed LevT model and the autoregressive baseline. All the Transformer-based models are trained on 8 Nvidia Volta GPUs with maximum $300K$ steps and a total batch-size of around $65,536$ tokens per step (We leave more details to the Appendix).

Overall results We present our main results on the generation quality and decoding speed in Table 1. We measure the speed by the averaged generation latency of generating one sequence at a time on single Nvidia V100 GPU. To remove the implementation bias, we also present the number of decoder iterations as a reference. It can be concluded that for both MT and summarization tasks, our proposed LevT achieves comparable and sometimes better generation quality compared to the strong autoregressive baseline, while LevT is much more efficient at decoding. A translation example is shown in Figure 3 and we leave more in Appendix.

Oracle v.s. Teacher Model As shown in Table 1, training the LevT with the teacher model achieves better results than the oracle counterpart, in most of the cases. Another interesting observation is the model has smaller latency. We conjecture that this is due to that the output of the teacher model possesses fewer modes and much less noisy than the real data. Consequently, LevT needs less number of iterations to converge to this expert policy.

Analysis of Efficiency As shown in Figure 4 (a), our model learns to properly terminate the decoding and adjust it based on the length of input. We also explore the variants of "early exit" where we denote LevT(m-n) as a model with m and n blocks for deletion (Eq. (3)) and placeholder prediction (Eq. (4)) respectively. Figure 4 (b) shows that althought it compromises the generation

[5] `https://github.com/harvardnlp/sent-summary`
[6] `http://www.phontron.com/kytea/`



(a) Number of decoding iterations[7] v.s. output length measured on the test set of Ro-En. For most of the time, LevT decodes with much smaller number (generally, 1∼3) of iterations.

(b) BLEU scores v.s. speed-up of latency across for LevT across variant halting layers and the autoregressive baselines on the test set of Ro-En.

Figure 4: Plots showing the decoding efficiency of the proposed Levenshtein Transformer.

quality a bit, our model with early exit achieves up to $\times 5$ speed-up with on-par performance comparing against a strong autoregressive Transformer using beam-search.

Importance of mixture roll-in policy We perform an ablation study on the learning algorithm. Specifically, we train a model with no mixing of the π_θ in Equation (6). We name this experiment by π^{DAE} due to its resemblance to a denoising autoencoder. We follow closely a standard pipeline established by Lee et al. (2018). Table 2 shows this comparison. As we can see that the deletion loss from π^{DAE} is much smaller while the generation BLEU score is inferior. We conjecture that this is caused by the mismatch between the states from the model and the roll-in policy in training the π^{DAE}.

$\tilde{\pi}_{\text{del}}$	BLEU	NLL_{del}
π_θ	**33.02**	≈ 0.202
π^{DAE}	31.78	≈ 0.037

Table 2: Test BLEU and training loss for deletion (NLL_{del}) using variant roll-in polices on WMT Ro-En dataset.

4.2 Sequence Refinement

We evaluate LevT's capability of refining sequence outputs on the APE task. In this setting, inputs are pairs of the source sequence and a black-box MT system generation. The ground-truth outputs are from real human edits with expansion using synthetic data.

Dataset We follow a normal protocol in the synthetic APE experiments (Grangier and Auli, 2017): we first train the input MT system on half of the dataset. Then we will train a refinement model on the other half based on the output produced by the MT model trained in the previous phase. For the real APE tasks, we use the data from WMT17 Automatic Post-Editing Shared Task[8] on En-De. It contains both real PE triples and a large-scale synthetic corpus.

Models & Evaluation The baseline model is a standard Transformer encoding the concatenation of the source and the MT system's output. For the MT system here, we want some imperfect systems that need to be refined. We consider a statistical phrase-based MT system (PBMT, Koehn et al., 2003) and an RNN-based NMT system (Bahdanau et al., 2015). Apart from BLEU scores, we additionally apply translation error rate (TER, Snover et al., 2006) as it is widely used in the APE literature.

Overall results We show the major comparison in Table 3. When training from scratch, LevT consistently improves the performance of the input MT system (either PBMT or NMT). It also achieves better performance than the autoregressive Transformer in most of the cases.

Pre-training on MT Thanks to the generality of the LevT model, we show it is feasible to directly apply the LevT model trained by generation onto refinement tasks — in this case — MT and APE. We name this a "zero-shot post-editing" setting. According to Table 3, the pre-trained MT models are always capable of improving the initial MT input in the synthetic tasks.

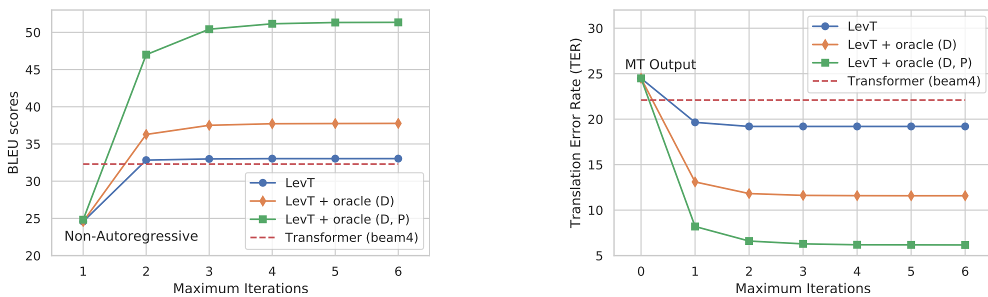
The real APE task, however, differs quite a bit from the synthetic tasks because human translators normally only fix a few spotted errors. This ends up with very high BLEU scores even for the

[7] We plot the iterations $\pm u$ where $u \sim \text{Uniform}[0, 0.5)$ for better visualization.

[8] `http://www.statmt.org/wmt17/ape-task.html`

Dataset		MT system	Do-Nothing	Transformer	Levenshtein Transformer Scratch	Levenshtein Transformer Zero-shot	Levenshtein Transformer Fine-tune
Synthetic	Ro-En	PBMT	27.5 / 52.6	28.9 / 52.8	29.1 / **50.4**	**30.1** / 51.7	–
		NMT	26.2 / 56.5	26.9 / 55.6	**28.3** / **53.6**	28.0 / 55.8	–
	En-De	PBMT	15.4 / 69.4	22.8 / 61.0	**25.8** / **56.6**	16.5 / 69.6	–
	En-Ja	NMT	37.7 / 48.0	41.0 / 44.9	**42.2** / **44.3**	39.4 / 47.5	–
Real	En-De	PBMT	62.5 / 24.5	67.2 / 22.1	66.9 / 21.9	59.6 / 28.7	**70.1 / 19.2**

Table 3: Performance (BLEU ↑ / case-sensitive TER ↓) comparison on APE. "do nothing" represents the results of the original MT system output; the autoregressive model uses beam-size 4. For the proposed LevT, we use "scratch" to denote training from scratch on the APE triple data, and use "zero-shot" to denote applying an MT pre-trained LevT model directly for post-editing tasks. The same model can be further fine-tuned. All scores with underlines are from the model trained with an autoregressive teacher model as the expert policy.



(a) Test set BLEU scores for WMT Ro-En

(b) Test set TER scores for Real APE En-De

Figure 5: MT & PE Performance v.s. Timeout iterations w/o oracle instructions.

"Do-nothing" column. However, the pre-trained MT model achieves the best results by fine-tuning on the PE data indicating that LevT is able to leverage the knowledge for generation and refinement.

Collaborate with Oracle Thanks to the saperation of *insertion* and *deletion* operations, LevT has better interpretability and controllability. For example, we test the ability that LevT adapts oracle (e.g. human translators) instructions. As shown in Figure 5, both MT and PE tasks have huge improvement if every step the oracle *deletion* is given. This goes even further if the oracle provides both the correct *deletion* and the number of *placehoders* to insert. It also sheds some light upon computer-assisted text editing for human translators.

5 Related Work

Non-Autoregressive or Non-Monotonic Decoding Breaking the autoregressive constraints and monotonic (left-to-right) decoding order in classic neural sequence generation systems has recently attracted much interest. Stern et al. (2018); Wang et al. (2018) designed partially parallel decoding schemes to output multiple tokens at each step. Gu et al. (2018) proposed a non-autoregressive framework using discrete latent variables, which was later adopted in Lee et al. (2018) as iterative refinement process. Ghazvininejad et al. (2019) introduced the masked language modeling objective from BERT (Devlin et al., 2018) to non-autoregressively predict and refine translations. Welleck et al. (2019); Stern et al. (2019); Gu et al. (2019) generate translations non-monotonically by adding words to the left or right of previous ones or by inserting words in arbitrary order to form a sequence.

Editing-Based Models Novak et al. (2016) predict and apply token substitutions iteratively on phase-based MT system outputs using convolutional neural network. QuickEdit (Grangier and Auli, 2017) and deliberation network (Xia et al., 2017) both consist of two autoregressive decoders where the second decoder refines the translation generated by the first decoder. Guu et al. (2018) propose a neural editor which learned language modeling by first retrieving a prototype and then editing over that. Freitag et al. (2019) correct patterned errors in MT system outputs using transformer models trained on monolingual data.

6 Conclusion

We propose Levenshtein Transformer, a neural sequence generation model based on insertion and deletion. The resulted model achieves performance and decoding efficiency, and embraces sequence generation to refinement in one model. The insertion and deletion operations are arguably more similar to how human writes or edits text. For future work, it is potential to extend this model to human-in-the-loop generation.

Acknowledgement

We would like to thank Kyunghyun Cho, Marc'Aurelio Ranzato, Douwe Kiela, Qi Liu and our colleagues at Facebook AI Research for valuable feedback, discussions and technical assistance.

References

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*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Markus Freitag, Isaac Caswell, and Scott Roy. 2019. Text repair model for neural machine translation. *arXiv preprint arXiv:1904.04790*.

Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. 2019. Constant-time machine translation with conditional masked language models. *CoRR*, abs/1904.09324.

David Grangier and Michael Auli. 2017. Quickedit: Editing text & translations by crossing words out. *arXiv preprint arXiv:1711.04805*.

Jiatao Gu, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. 2018. Non-autoregressive neural machine translation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, Canada, April 30-May 3, 2018, Conference Track Proceedings*.

Jiatao Gu, Qi Liu, and Kyunghyun Cho. 2019. Insertion-based decoding with automatically inferred generation order. *arXiv preprint arXiv:1902.01370*.

Kelvin Guu, Tatsunori B Hashimoto, Yonatan Oren, and Percy Liang. 2018. Generating sentences by editing prototypes. *Transactions of the Association of Computational Linguistics*, 6:437–450.

Lukasz Kaiser, Samy Bengio, Aurko Roy, Ashish Vaswani, Niki Parmar, Jakob Uszkoreit, and Noam Shazeer. 2018. Fast decoding in sequence models using discrete latent variables. In *International Conference on Machine Learning*, pages 2395–2404.

Yoon Kim and Alexander Rush. 2016. Sequence-level knowledge distillation. In *EMNLP*.

Philipp Koehn, Franz Josef Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54. Association for Computational Linguistics.

Jason Lee, Elman Mansimov, and Kyunghyun Cho. 2018. Deterministic non-autoregressive neural sequence modeling by iterative refinement. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 1173–1182.

Vladimir Iosifovich Levenshtein. 1965. Binary codes capable of correcting deletions, insertions, and reversals. In *Doklady Akademii Nauk*, volume 163, pages 845–848. Russian Academy of Sciences.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Roman Novak, Michael Auli, and David Grangier. 2016. Iterative refinement for machine translation. *arXiv preprint arXiv:1610.06602*.

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.

Alexander M. Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 379–389, Lisbon, Portugal. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Matthew Snover, Bonnie Dorr, Richard Schwartz, Linnea Micciulla, and John Makhoul. 2006. A study of translation edit rate with targeted human annotation. In *In Proceedings of Association for Machine Translation in the Americas*, pages 223–231.

Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. 2019. Insertion transformer: Flexible sequence generation via insertion operations. *arXiv preprint arXiv:1902.03249*.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. In *Advances in Neural Information Processing Systems*, pages 10107–10116.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*.

Chunqi Wang, Ji Zhang, and Haiqing Chen. 2018. Semi-autoregressive neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 479–488, Brussels, Belgium. Association for Computational Linguistics.

Sean Welleck, Kianté Brantley, Hal Daumé III, and Kyunghyun Cho. 2019. Non-monotonic sequential text generation. *arXiv preprint arXiv:1902.02192*.

Yingce Xia, Fei Tian, Lijun Wu, Jianxin Lin, Tao Qin, Nenghai Yu, and Tie-Yan Liu. 2017. Deliberation networks: Sequence generation beyond one-pass decoding. In *Advances in Neural Information Processing Systems*, pages 1784–1794.

A Learning & Inference Algorithm

We present the detailed algorithms for learning and decoding from Levenshtein Transformer as follows. For simplicity, we always omit the source information $\boldsymbol{x}$ in conditional sequence generation tasks such as machine translation which is handled by the cross-attention with an encoder on $\boldsymbol{x}$.

The learning algorithm is shown in Algorithm 1. $\mathcal{E}$ is the environment and $\mathcal{D}$ is denoted as the Levenshtein distance, and we can easily back-track the optimal insertion and deletion operations through dynamic programming. We only show the the case with single batch-size for convenience. We also present the inference algorithm in Algorithm 2. If the initial sequence $\boldsymbol{y}^0$ is empty (`<s></s>`), the proposed model will skip the first deletion and do sequence generation. Otherwise, the model starts with deletion operations and refine the input sequence.

Algorithm 1 Learning for Levenshtein Transformer

Initialize: Training set $\mathcal{T}$, expert policy π^*, model policy π_θ, random deletion policy π^{RND}, α, β
repeat
  Sample a training pair $(\boldsymbol{y}^0, \boldsymbol{y}^*) \sim \mathcal{Y}$
  if expert π^* is a teacher model **then**
    Set the teacher's output as the target $\boldsymbol{y}^* = \boldsymbol{y}^{\text{AR}}$
  end if
  Sample $u, v \sim$ Uniform$[0, 1]$
  if $u < \beta$ **then**
    $\boldsymbol{y}_{\text{ins}} = \mathcal{E}(\boldsymbol{y}^0, \tilde{\boldsymbol{d}})$, where $\tilde{\boldsymbol{d}} = \arg\min_{\boldsymbol{d}} \mathcal{D}(\boldsymbol{y}^*, \mathcal{E}\left(\boldsymbol{y}^0, \boldsymbol{d}\right))$
  else
    $\boldsymbol{y}_{\text{ins}} = \mathcal{E}(\boldsymbol{y}^*, \tilde{\boldsymbol{d}})$, where $\tilde{\boldsymbol{d}} \sim \pi^{\text{RND}}(\cdot|\boldsymbol{y}^*)$
  end if
  $\boldsymbol{y}'_{\text{ins}} = \mathcal{E}(\boldsymbol{y}_{\text{ins}}, \boldsymbol{p}^*)$, where $\boldsymbol{p}^*, \boldsymbol{t}^* = \arg\min_{\boldsymbol{p},\boldsymbol{t}} \mathcal{D}(\boldsymbol{y}^*, \mathcal{E}\left(\boldsymbol{y}_{\text{ins}}, \{\boldsymbol{p}, \boldsymbol{t}\}\right))$
  if $v < \alpha$ **then**
    $\boldsymbol{y}_{\text{del}} = \boldsymbol{y}^0$
  else
    $\boldsymbol{y}_{\text{del}} = \mathcal{E}(\boldsymbol{y}'_{\text{ins}}, \hat{\boldsymbol{t}})$, where $\hat{\boldsymbol{t}} = \arg\max_{\boldsymbol{t}} \sum_{y_i \in \boldsymbol{y}'_{\text{ins}}, y_i = \texttt{<PLH>}} \log \pi_\theta^{\text{tok}}(t_i|i, \boldsymbol{y}'_{\text{ins}})$
  end if
  $\mathcal{L}_\theta^{\text{ins}} = -\left[\sum_{y_i \in \boldsymbol{y}_{\text{ins}}, p_i^* \in \boldsymbol{p}^*} \log \pi_\theta^{\text{plh}}(p_i^*|i, \boldsymbol{y}_{\text{ins}}) + \sum_{y_i \in \boldsymbol{y}'_{\text{ins}}, y_i = \texttt{<PLH>}, t_i^* \in \boldsymbol{t}^*} \log \pi_\theta^{\text{tok}}(t_i^*|i, \boldsymbol{y}'_{\text{ins}})\right]$
  $\mathcal{L}_\theta^{\text{del}} = -\sum_{y_i \in \boldsymbol{y}_{\text{del}}, d_i^* \in \boldsymbol{d}^*} \log \pi_\theta^{\text{del}}(d_i^*|i, \boldsymbol{y}_{\text{del}})$, where $\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \mathcal{D}(\boldsymbol{y}^*, \mathcal{E}\left(\boldsymbol{y}_{\text{del}}, \boldsymbol{d}\right))$
  $\theta = \theta - \lambda \cdot \bigtriangledown_\theta \left[\mathcal{L}_\theta^{\text{ins}} + \mathcal{L}_\theta^{\text{del}}\right]$
until Maximum training steps reached

B Dataset and Preprocessing Details

Table 4 and 5 list the statistics (# of sentences, vocabulary) for all the datasets used in this work. We learn BPE vocabulary with $32,000$ joint operations for WMT En-De and Gigaword and $40,000$ joint operations for WMT Ro-En. For WAT En-Ja, we adopt the official $16,384$ BPE vocabularies learned separately on source and target side.

Dataset		Train	Valid	Test	Vocabulary
Translation	WMT'16 Ro-En	608,319	1999	1999	34,983
	WMT'14 En-De	4,500,966	3000	3003	37,009
	WAT'17 En-Ja	2,000,000	1790	1812	17,952 / 17,801
Summarization	English Gigaword	3,803,957	189,651	1951	30,004

Table 4: Dataset statistics for sequence generation tasks (MT and TS).

Algorithm 2 Decoding for Levenshtein Transformer

Initialize: Input $\boldsymbol{y} = \boldsymbol{y}^0$, step $t = 0$, maximum step $T_{\max}$, model policy π_θ.
repeat
  if $\boldsymbol{y} = \texttt{<s></s>}$ **then**
    Empty sequence, skip deletion: $\boldsymbol{y}' = \boldsymbol{y}$
  else
    Delete tokens: $\boldsymbol{y}' = \mathcal{E}(\boldsymbol{y}, \hat{\boldsymbol{d}})$, where $\hat{\boldsymbol{d}} = \arg\max_{\boldsymbol{d}} \sum_{y_i \in \boldsymbol{y}} \log \pi_\theta^{\text{del}}(d_i | i, \boldsymbol{y})$
  end if
  if (t > 0) & ($\boldsymbol{y}' = \tilde{\boldsymbol{y}}$) **then**
    Termination condition satisfied: direct loop
    break
  end if
  Assign deleted output for back-up $\tilde{\boldsymbol{y}} = \boldsymbol{y}'$
  Insert placeholders: $\boldsymbol{y}'' = \mathcal{E}(\boldsymbol{y}', \hat{\boldsymbol{p}})$, where $\hat{\boldsymbol{p}} = \arg\max_{\boldsymbol{p}} \sum_{y_i y_{i+1} \in \boldsymbol{y}'} \log \pi_\theta^{\text{plh}}(p_i | i, \boldsymbol{y}')$
  if $\boldsymbol{y}'' = \boldsymbol{y}' = \boldsymbol{y}$ **then**
    Termination condition satisfied: nothing to delete, nothing to insert.
    break
  end if
  if $\boldsymbol{y}'' = \boldsymbol{y}'$ **then**
    Nothing to insert, skip insertion: $\boldsymbol{y} = \boldsymbol{y}''$
  else
    Replace placeholders: $\boldsymbol{y} = \mathcal{E}(\boldsymbol{y}'', \hat{\boldsymbol{t}})$, where $\hat{\boldsymbol{t}} = \arg\max_{\boldsymbol{t}} \sum_{y_i \in \boldsymbol{y}'', y_i = \texttt{<PLH>}} \log \pi_\theta^{\text{tok}}(t_i | i, \boldsymbol{y}'')$
  end if
  Update steps: t = t + 1
until Reach the maximum length $t = T_{\max}$
return $\boldsymbol{y}$

Dataset		MT-Train	APE-Train	Valid	Test	Vocabulary
Synthetic	WMT'16 Ro-En	300,000	308,319	1999	1999	34,983
	WMT'14 En-De	2,250,000	2,250,967	3000	3003	37,009
	WAT'17 En-Ja	1,000,000	1,000,000	1790	1812	17,952 / 17,801
Real	WMT'17 APE En-De	4,391,180	526,368 (fake) + 24,000 (real)	2000	2000	40,349

Table 5: Dataset statistics for sequence refinement tasks (APE).

C Model and Training Details

C.1 Sequence Generation Tasks

Transformer models are used for autoregressive baselines as well as teacher models (for the expert policy). By default, we set $d_{\text{model}} = 512$, $d_{\text{hidden}} = 2048$, $n_{\text{heads}} = 8$, $n_{\text{layers}} = 6$, $\text{lr}_{\max} = 0.0005$, label-smooth $= 0.1$, warmup $= 10000$ and dropout $= 0.3$. Source and target side share embeddings in all the training pairs except for WAT En-Ja where BPE vocabularies of both side are learned separately and are almost non-overlapping.

Since the training objectives for Levenshtein Transformer contains randomness terms (Eq. (6) (7)), we instead use BLEU (for MT) or ROUGE-2 (for TS) to select the best checkpoint by validation scores. We do not average checkpoints in this work.

C.2 Sequence Refinement Tasks

For synthetic APE tasks, we keep the same training conditions for LevT as those for MT tasks (§C.1). As described earlier in §4.2, we build the baseline Transformer by concatenating the source and MT system's output as the input sequence for the encoder. Specially, we restart the positional embeddings

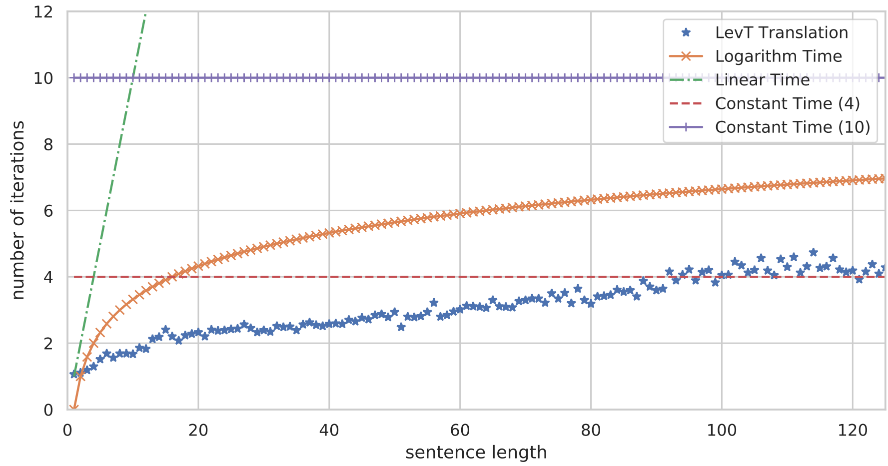


Figure 6: Averaged number of decoding iterations v.s. length of the source sentences on Romanian (Ro) monolingual corpus.

for the MT output, add an additional language embedding for each token of the input sequence to show its language type. The detailed hyperpameters are the same as the standard Transformer.

As described in §4.2, we consider the following two different imperfect MT systems to provide the refinement inputs. Firstly, we consider the traditional statistical phrase-based machine translation system (PBMT). We follow the instruction to build the basic baseline model via moses[9]. As for the NMT-based model, we use a single layer attention-based model composed by LSTM. We build this model on fairseq-py[10] with the default configuration.

For the real APE task, we follow the procedures introduced in Junczys-Dowmunt and Grundkiewicz (2016). Synthetic corpus has two subsets: a 500K one and a 4M one. We over-sample real data by 10 times and merge it with the 500K synthetic data to train APE models. Besides, we also train a LevT MT model on the bigger (4M) synthetic corpus where we only use the source and target pairs.

C.3 Implementation

Both the proposed Levenshtein Transformer and the baseline Transformer are implemented using PyTorch[11]. The code will be released based on the acceptance.

D Balanced Speed Test on All Lengths

We see from figure 4 (a) that most of the translations are gotten within 3 iterations. Long sentences (e.g. with over 100 tokens) however are relatively underrepresented in the validation set and have sparser data points than that for shorter ones.

To mitigate this bias, we try adding extra data points for Ro-En by selecting and translating long sentences from a monolingual corpus based on New Crawl. Specifically, we group Ro sentences based on their lengths from $1 \sim 128$ and for each group, we randomly sample 64 sentences to decode for each group. We show the averaged number of decoding iterations of each length group in Figure 6.

We see that the time-complexity of the proposed LevT is approximately *linear* (not *constant*) to the input length ($1 \sim 128$), but with a much smaller ratio (≈ 1 iteration $/40$ tokens) compared to the standard auto-regressive modes (1 iteration $/1$ token).

[9] `http://www.statmt.org/moses/?n=Moses.Baseline`
[10] `https://github.com/pytorch/fairseq/blob/master/fairseq/models/lstm.py`
[11] `https://pytorch.org/`

E More Decoding Examples

We present more examples from the proposed Levenshtein Transformer as follows.

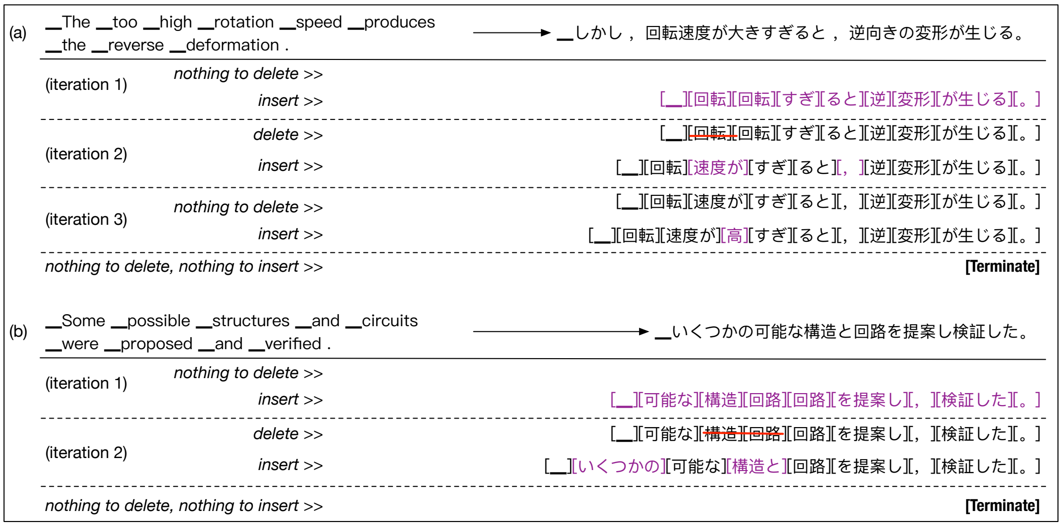


Figure 7: Translation examples for WAT'17 Small-NMT En-Ja with the Levenshtein Transformer.

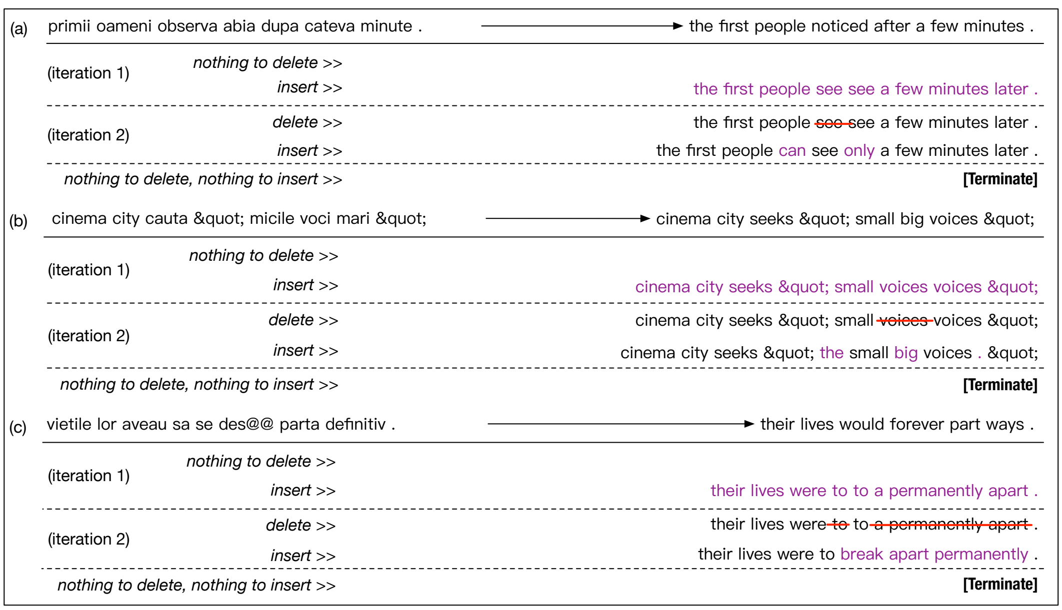


Figure 8: Translation examples for WMT'16 Ro-En with the Levenshtein Transformer.

(a)	Or search for plan@@ ets similar to the Earth and thus perhaps discover ex@@ tr@@ ater@@ restri@@ al life ? →	Oder suchen Sie nach Planeten ähnlich der Erde und entdecken Sie damit vielleicht das ex@@ tr@@ ater@@ res@@ tr@@ ische Leben ?
(iteration 1)	*nothing to delete >>*	
	insert >>	Oder suchen Sie Planeten Planeten Planeten Erde Erde Erde Erde und vielleicht Ex@@ tr@@ tr@@ tr@@ tr@@ ische entdecken ?
(iteration 2)	*delete >>*	Oder suchen Sie ~~Planeten Planeten~~ Planeten ~~Erde Erde Erde~~ Erde und ~~vielleicht Ex@@ tr@@ tr@@ tr@@ tr@@ ische entdecken~~ ?
	insert >>	Oder suchen Sie nach Planeten ähnlich der Erde und entdecken so vielleicht außer@@ tr@@ dische Leben ?
(iteration 3)	*delete >>*	Oder suchen Sie nach Planeten ähnlich der Erde und entdecken so vielleicht ~~außer@@ tr@@ dische~~ Leben ?
	insert >>	Oder suchen Sie nach Planeten ähnlich der Erde und entdecken so vielleicht das Leben ?
(iteration 4)	*nothing to delete >>*	Oder suchen Sie nach Planeten ähnlich der Erde und entdecken so vielleicht das Leben ?
	insert >>	Oder suchen Sie nach Planeten ähnlich der Erde und entdecken so vielleicht das eigene Leben ?
	nothing to delete, nothing to insert >>	**[Terminate]**
(b)	Local public transport will also become more expensive . →	Der öffentliche Nah@@ verkehr werde auch te@@ urer .
(iteration 1)	*nothing to delete >>*	
	insert >>	Auch der öffentliche Nah@@ verkehr werden te@@ urer .
(iteration 2)	*delete >>*	Auch der öffentliche Nah@@ verkehr ~~werden~~ te@@ urer .
	insert >>	Auch der öffentliche Nah@@ verkehr wird te@@ urer .
	nothing to delete, nothing to insert >>	**[Terminate]**

Figure 9: Translation examples for WMT'14 En-De with the Levenshtein Transformer.

(a)	strengthened border patrol has led to a ##–percent drop in arrests of undocumented migrants this year at the u.s.–mexico frontier , it was reported on wednesday . →	arrests of bor@@ der–@@ cross@@ ers drop
(iteration 1)	*nothing to delete >>*	
	insert >>	border patrol leads arrests ## migrants at u.s.–mexico border
(iteration 2)	*delete >>*	border patrol ~~lead~~s arrests ~~##~~ migrants at u.s.–mexico border
	insert >>	border patrol reduces arrests of migrants at u.s.–mexico border
	nothing to delete, nothing to insert >>	**[Terminate]**
(b)	us lawyer ed f@@ agan said wednesday he will bring a multi–million dollar lawsuit in the united states against the polish government unless it takes concrete steps to repay a huge debt to holders of bonds issued before world war ii . →	us star lawyer ed f@@ agan to sue poland for unpaid bonds
(iteration 1)	*nothing to delete >>*	
	insert >>	f@@ agan to multi–million sue poland poland over debt debt
(iteration 2)	*delete >>*	f@@ agan to ~~multi–million~~ sue ~~poland~~ poland over ~~debt~~ debt
	insert >>	f@@ agan threatens to sue poland over debt repayment
	nothing to delete, nothing to insert >>	**[Terminate]**

Figure 10: Translation examples for English Gigaword with the Levenshtein Transformer.

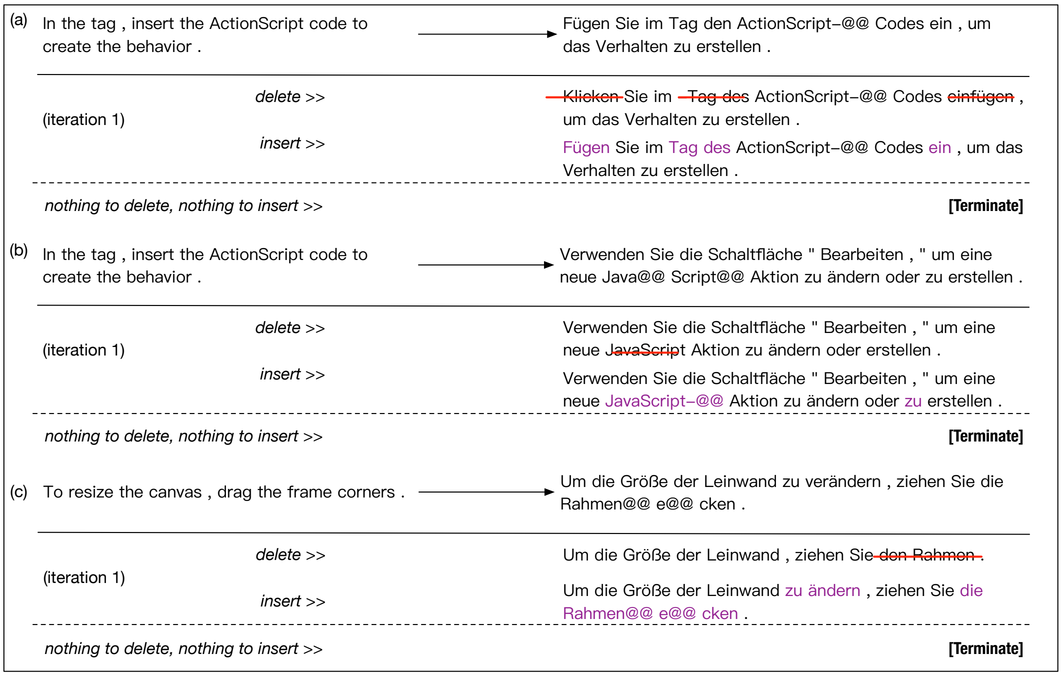


Figure 11: Post-editing examples for WMT'17-APE En-De with the Levenshtein Transformer.

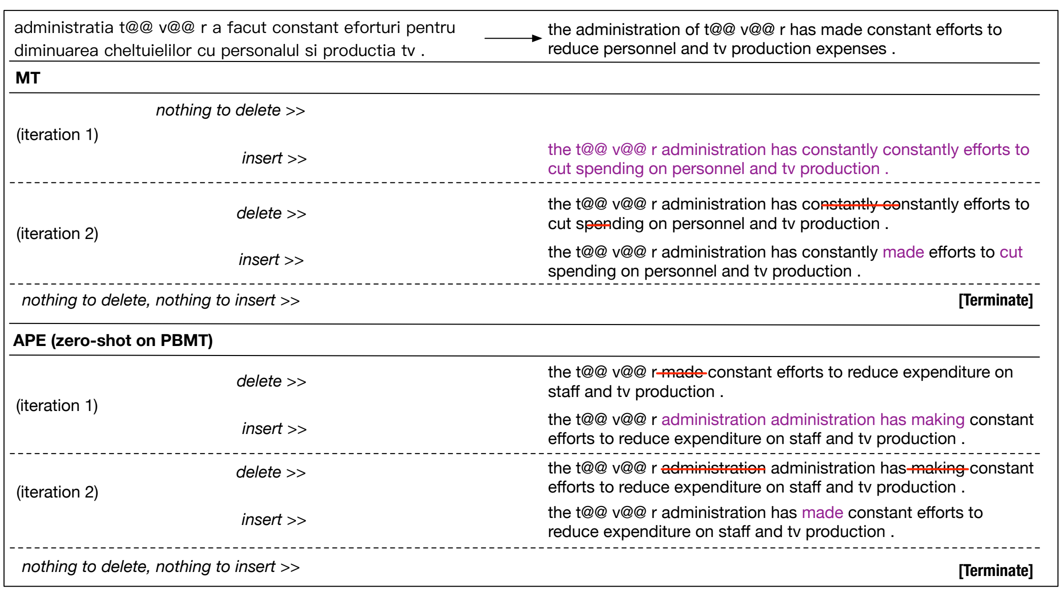


Figure 12: An example for machine translation and zero-shot post-editing over a PBMT system's output on WMT'16 Ro-En with the Levenshtein Transformer (LevT) trained for MT. It is clear to find that, the pre-trained LevT can directly adapt to the PBMT's output and have a different refinement results compared to translate from scratch.