We propose INSNET, an expressive insertion-based text generator with efficient training and flexible decoding (parallel or sequential). Unlike most existing insertion-based text generation works that require re-encoding of the context after each insertion operation and thus are inefficient to train, INSNET only requires one pass of context encoding for the entire sequence during training by introducing a novel insertion-oriented position encoding and a light-weighted slot representation strategy to enable computation sharing. Furthermore, we propose an algorithm INSNET-Dinic to better determine the parallelization of insertion operations that provides a controllable switch between parallel and sequential decoding, making it flexible to handle more parallelizable tasks such as machine translation with efficient decoding, or less parallelizable tasks such as open-domain text generation to guarantee high-quality outputs. Experiments on two lexically constrained text generation datasets and three machine translation datasets demonstrate INSNET’s advantages over previous insertion-based methods in terms of training speed, inference efficiency, and generation quality.
Figure 1: The absolute position information of each token is volatile for insertion-based models. Thus, naive models with absolute position encoding have to re-encode the sequence after each insertion operation during training.

Table 1: Comparisons between INSNET and existing insertion-based models regarding training-time re-encoding steps (#Re-Enc. columns) and how the models encode positional information (the PosInfo column).

2 Related Works and Background

Auto-regressive Language Models minimize the negative-log-likelihood of a sequence of n tokens \( s_{1:n} = [x_0, x_1, ..., x_{n-1}] \) with a left-to-right factorization. With the transformer architecture (Vaswani et al., 2017), each step of likelihood estimation can be calculated in parallel while sharing the prefix context encoding calculations. This makes it possible to build powerful and efficient text generation models, like the GPT family (Radford et al., 2018, 2019; Brown et al., 2020). A lot of successful applications are based on this paradigm of models, such as automatic story generation (Yao et al., 2019; Tan et al., 2020), image captioning (Vinyals et al., 2015; Xu et al., 2015), machine translation (Bahdanau et al., 2015; Liu et al., 2020), and dialogue system (Li et al., 2017b,a).

Insertion-based Models with Parallel Decoding Insertion transformer (Stern et al., 2019) (InsT) proposes a design for insertion-based text generation. In each step, a bi-directional encoder transformer is performed on the partial sequence to compute the representation for each candidate slot between every two consecutive positions. Then, a model for the joint distribution of position-token is built to insert one or more token(s). Variants of InsT share the common atomic objective that models the step log-likelihood. On step \( t \) where a token \( x_{i:t+1} \) is inserted in slot \( i_{i:t+1} \) between position \( i \) and \( i + 1 \) of context \( s_t \), the step log-likelihood can be written as:

\[
\log p(x_{i:t+1}, i_{i:t+1} | s_t) = \log p_{\text{position}}(i + 1 | s_t) + \log p_{\text{token}}(x_{i:t+1} | e(s_t)_i \oplus e(s_t)_{i+1}),
\]

where \( e(\cdot)_i \) stands for the i-th position of bi-directional encoding of the sequence and \( \oplus \) stands for vector concatenation. InsT adopts the original absolute positional encoding of transformers, the
representation of the generated sequence has to be completely re-encoded after each step of context expansion to match the position changes of tokens. The expectation of the negative log step likelihood over all permitted context-insertion pairs at each step is computed as the step loss. The step losses from the first step to the last one are summed up as the sequence loss. Benefiting from the partially parallelized prediction of tokens, InsT can reduce the number of re-encoding steps to $\Theta(\log n)$.

Levenshtein Transformer (Gu et al., 2019b) (LevT) contains two phases during generation: 1) Insertion phase: it first uses a similar strategy as InsT, but only inserts placeholders; an MLM is applied to fulfill the placeholders. 2) Deletion phase: the model is trained as a token-wise discriminator to determine where to delete, under the evaluation of the Levenshtein distance by dynamic programming. In practice, this would result in a slower process compared to InsT, but better generation quality as it alleviates the incoherence caused by parallel insertions.

**Sequential Insertion-based Model** Previous exploration in insertion-based models mostly assumed the usage of parallel decoding in each decoding step, resulting in a partially auto-regressive procedure. For those are trained to only generate one token per decoding step, a vanilla implementation would result in an $O(n)$ factor in training time complexity. NMSTG (Non-monotonic Sequential Text Generation, Welleck et al. (2019)) is one of the first attempts at modeling a non-monotonic sequential insertion-based generation process. It constrains the dependencies of each inserted token to pseudo-Markovian on an expansion tree. InDIGO (Gu et al., 2019a) is proposed as a sequentially-decoded insertion-based model with the transformer architecture. It supports the efficient likelihood estimation by working around the aforementioned volatility problem (see Figure 1) at a cost of omitting the distance information in its position encoding, and is thus able to adopt the conventional computation sharing trick to boost the multiplicative factor in training time complexity for each sequence to as fast as $O(1)$. However, InDIGO uses a encoding of all previously inserted tokens to predict the next token regardless of their tentative position. It is only after the next token is predicted, the inserted position for it is predicted. Thus, there’s no trivial solution to use InDIGO as a (partially) non-autoregressive insertion-based generator.

**Absolute vs. Relative Position Embedding** The original transformer (Vaswani et al., 2017) uses sinusoidal, absolute position embeddings. Relative position embedding in transformers (Shaw et al., 2018; Dai et al., 2019) was originally proposed to make the modeling of spatial relation invariant to position translation, and to improve the long-term dependency performance of the model. In replacement of absolute positions, which are tied to each token in the sequence, relative positions try to encode the spatial layout of the tokens with a matrix that records a directed distance from the column token to the row token. We further develop the idea of relative position embedding as the key component to overcome the issue of volatile position information of absolute positions in an insertion-based generation process. A noticeable fact is that InDIGO’s implementation of position encoding can also be regarded as a direction-only relative position encoding system.

### 3 The InsNet Model

There are three major components of InsNet: 1) An context encoder based on the transformer architecture (Vaswani et al., 2017) that uses a novel way to compute insertion-oriented relative position encoding to better suit the insertion-based nature of the generation process and enable computation sharing (Section 3.1); 2) a module to compose expressive slot representation for predicting tokens to insert in different slots simultaneously (Section 3.2); and 3) an algorithm that adaptively determines the parallelization of the insertion operations to minimize conflicts (Section 3.3). Figure 2 illustrate the components of the full model.

#### 3.1 Context Encoder: A Transformer with Insertion-Oriented Relative Position

In the introduction, we discussed the challenges in (efficient) computation sharing caused by the volatile position information of the context tokens as shown in Figure 1. We address this problem by empowering the transformer-based context encoder with a distance-aware, insertion-oriented pairwise relative position encoding. The proposed insertion-oriented relative position encoding shares important designs with previous relative position embeddings (Dai et al., 2019; Yang et al., 2019; Shih et al., 2019), but differs in how it accurately depicts the spatial layout (i.e., the actual sequential order) of the inserted tokens in an insertion-based generation process.
Figure 2: Illustration of the full InsNet model. Given an input sequence with a specified insertion order as illustrated in the [Inputs] panel (the upper right panel in blue), a context encoder (Section 3.1) illustrated in the [Encoder] panel (the upper left corner in purple) with insertion-oriented position encoding (Section 3.1.1) is applied to the sequence of insertion operations to obtain step-contextualized token embeddings as the transformer outputs. Then, the slot representation module (Section 3.2) illustrated in the [Slot Representation] panel (the bottom panel in yellow) compose slot representations for each step from InsNet transformer outputs using its left, right token representation and a global, step-wise token representation. The slot representations are then used to compute the token/position likelihood and also to determine the auto-parallelization in InsNet-Dinic (Section 3.3) illustrated in the [InsNet-Dinic] panel (the middle right panel in green).

3.1.1 Offset: Insertion- Oriented Relative Position Encoding

To illustrate our relative position encoding, considering the partial context (to be completed by further insertions) of “I have pen” with an insertion order of “have pen I”. To insert an “a” in between “have” and “pen”, the distance vector for token “a” against the rest of the context inserted so far should be [(“have”, -1), (“pen”, +1), (“I”, -2)], simplified as [-1, 1, -2]. This relative position encoding clearly defines where the insertion happens by only describing the pairwise spatial relationship between the incoming token and the existing context tokens. We can pack the relative position vectors for each insertion step to get a matrix that reflects the relative spatial relation along the trajectory of insertions, with each row corresponding to an insertion step. We name it the offset matrix.

Figure 3: Comparison of different position encodings for an insertion-based generation process. From left to right, we illustrate (a) the volatile absolute positions, (b) the traditional non-insertion-oriented relative positions, (c) the proposed offset matrix presented in the insertion order, and (d) the same offset matrix permuted to show how it looks like if we restore the actual (partial-)sequence.
We hereby describe how to incorporate the offset matrix into each transformer layer in INSNet for efficient context encoding. In general, we inherit most designs from previous ones with relative position encoding (Dai et al. 2019; Yang et al. 2019). The encoder panel in Figure 2 (the upper left panel in purple) illustrate the process. Specifically, for layer number $i = 1 ... N$, given the output embedding $E^{i-1}$ from the last layer and the offset matrix $R$, the formulation of a transformer layer in an $N$-layer INSNet can be written as:
We apply a binary classification with the context efficiently encoded, the subsequent step of an insertion-based generator is to predict where. A naive design of the slot representation is to simply concatenate the representation vectors from where

\[
Q^i, K^i, V^i, P^i = W_{q,E}^i, W_{k,E}^i, E^{i-1}, W_{k,H}^i, R
\]

\[
A^i = Q^i K^i + Q^i P^i + u^i K^i + v^i P^i
\]

\[
V^i_{\text{reduced}} = \text{Masked-Softmax}(A^i) V^i
\]

\[
V^i_{\text{skipconn}} = V^i_{\text{reduced}} + E^{i-1}
\]

\[
E^i = \text{Feed-Forward}(V^i_{\text{skipconn}}; \theta^i)
\]

where \( Q^i, K^i, V^i, P^i \) are query, key, value, position matrices, respectively, for layer \( i \). \( u^i, v^i, W_q^i, W_{k,E}^i, W_v^i, W_{k,H}^i \) and \( \theta^i \) are learnable model parameters of each INSNET layer. We denote the input word embeddings as \( E_0 \) for notation consistency. The last layer of transformer in INSNET outputs the step-wise context-aware token embeddings. They will be used to compute slot representations for the insertion steps.

### 3.2 Slot Representation and Insertion Prediction

With the context efficiently encoded, the subsequent step of an insertion-based generator is to predict the next position and token to insert. Thus, a representation for each potential insertion slot should be computed. We hereby show how slot representations can be aggregated from INSNET outputs and the potential challenges during this aggregation process.

A naive design of the slot representation is simply to concatenate the representation vectors from the left-neighbor (in the natural observation order) of the slot as is done in prior works (Stern et al. 2019). This slot representation can efficiently compute the slot representation for all possible slots in parallel for each time step. However, the context encoding we obtained is insertion-order sensitive and unaware of the tokens inserted later in INSNET. This naive slot representation thus falls short of capturing the global sequence information. As a remedy, we propose to also include the last insertion token’s representation to compose the slot representation \( e_{ij}^{(t)} \).

Specifically, given the representations of the left-side neighbor \( e_{i-1} \), the right-side neighbor \( e_{i+1} \), and the last inserted token \( e_t \), the slot representation can be computed as:

\[
e_{ij}^{(t)} = \text{LayerNorm}(f_l(e_{i-1}) \oplus f_r(e_{i+1}) + e_t)
\]

where \( f_l \) and \( f_r \) are linear projections for position vectors and \( \oplus \) stands for vector concatenation. The slot representation panel in Figure 2 (the bottom panel in yellow) illustrates the process. Note that comparing step 3 and step 4, both compute the slot representation for the potential edges during this aggregation process.

The slot representation can then be converted into a probabilistic distribution over the vocabulary to predict the log-likelihood of inserted tokens using a log-linear transformation i.e.

\[
\log p(x_{ij}^{(t)} | e_{ij}^{(t)}) = \text{log-softmax}(W_p e_{ij}^{(t)} + b)_{x_{ij}^{(t)}}.
\]

To decide which slot to insert next, a position logit \( \alpha_k^{(t)} = w_o \top e_{ik}^{(t)} \) is computed for each slot \( k \) with a linear layer. Then a (log-)softmax operation is applied on top of the position logits to obtain the log-probability to insert to each candidate slot. i.e.

\[
\log o(x_{ik}^{(t)} | \{e_i^{(t)}\}) = \text{log-softmax}(\{\alpha_0^{(t)}, \alpha_1^{(t)}, ..., \alpha_k^{(t)}\})
\]

We apply a binary classification \( q(0/1|\cdot) \) on the last inserted token’s representation \( e_t \) to decide the termination of the generation at each step. During training, only the final step \( q(0/1|e_{n-1}) \) is trained to predict 1; all intermediate steps are trained to predict 0. In the following section, we will continue to discuss how to formulate the final objective function of INSNET using these step-wise distributions.

### 3.3 Adaptive Parallelization of Insertions

In addition to our efforts for better training efficiency, we also propose an algorithm to adaptively parallelize the generation process of INSNET to speed up the decoding while preserving the generation quality. The idea mimics the graph layerization process in the Dinic’s algorithm (Dinitz, 2006). Specifically, that we partition the tokens into different layers, such that the tokens within the
same layer are safe to insert in parallel without affecting the coherence of context. The algorithm aims to determine the partition such that the log-likelihood estimation of the resulting parallel insertion-based model is as close to the sequential insertion-based models as possible. The intuition is that the sequential insertion process maximally captures the inter-dependencies between the output tokens. By staying close to the sequential insertion-based model regarding likelihood prediction ability, we speed up the inference without severely sacrificing the likelihood estimation quality.

We initialize the layerization with a sequential insertion order, which is equivalent to single-node layerization, as is illustrated in Figure 5(a). Starting from this initialization, we perform IINSNET-Dinic to gradually evolve this fully-sequential layerization into a non-autoregressive, parallelized layerization, as shown in Figure 5(b). We promote a token $x_{wi}$ from its respective slot $(i', j')$ in layer $l$ to $(i^{l-1}, j^{l-1})$ in the previous layer $l - 1$ if two conditions are satisfied: 1) In layer $l - 1$, slot $(i^{l-1}, j^{l-1})$ does not yet have an assigned insertion; 2) $\log p(x_{wi} | e_{i_{l-1}j_{l-1}}^{l-1}) - \log p(x_{wi} | e_{i_{l-1}j_{l-1}}^{l-1}) \leq \tau$.

Here $\tau$ is a hyper-parameter that controls how much to parallel. Intuitively, larger $\tau$ usually indicates more tolerance for token likelihood loss caused by parallel decoding, i.e. more parallelization with tradeoff on likelihood estimation accuracy. The choice of $\tau$ is highly task/data-dependent as it is a value associated with the likelihood. Note that the promotion of tokens only affects the slot representation computation and the insertion likelihood calculation of the respective insertion. When encoding the context (as is discussed in Section 3.1), we always treat it as if these insertion operations are performed sequentially.

For a target sequence $s_{n} = [x_0, x_1, ..., x_{n-1}]$, after the layerization algorithm, if token $x_{ij}$ is layerized in the $l$-th layer and to be inserted in the slot $(i, j)$, we denote it as $x_{ij}^l$. Suppose we have $m$ effective layers in total. Denote the corresponding representation for slot $(i, j)$ given available context in layer $l$ as $e_{ij}^l$. The general objective of IINSNET parameterized by $\phi$ can be formulated as:

$$
\mathcal{L}(s, \phi) = - \sum_{l=0}^{m-1} \sum_{x_{ij}^l} \log p(x_{ij}^l | e_{ij}^l) + \log o(x_{ij}^l | \{e_{ij}^{l'}\}) - \sum_{l=0}^{m} \log q_{\phi}(1(l = m) | e_{-1}^{l})
$$

4 Experiments

We demonstrate the efficiency, flexibility, and model capability of IINSNET with two sets of experiments. 1) To show IINSNET's performance and the flexibility to switch between sequential and parallel decoding on datasets with high inter-token dependency (i.e., less suitable for parallel decoding). We follow the setup in [Zhang et al. (2020)] and address the unsupervised lexically constrained text generation problem on two datasets Yelp Review and News. 2) To further verify the effectiveness of IINSNET-Dinic, we evaluate IINSNET as a (partially) non-autoregressive machine translation model.
Table 2: Performance comparison on Yelp Review and News datasets. For Levenshtein Transformer, insertion and deletion stages are both counted in # of decoding-time iterations. It’s non-trivial to do unsupervised lexically constrained text generation with auto-regressive models. To work around this, we implemented a Plan-And-Write (Yao et al., 2019) style auto-regressive transformer-based model for better reference. Models with a star mark * are re-implemented by us. Other baseline results are directly taken from the original papers.

<table>
<thead>
<tr>
<th>Model</th>
<th>Yelp Review</th>
<th>News</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-regressive Transformer (Plan-And-Write-static, Yao et al., 2019)</td>
<td>16.68/5.46</td>
<td>2.79/2.86</td>
</tr>
<tr>
<td>NMTG (Welleck et al., 2019)</td>
<td>10.06/1.92</td>
<td>1.11/1.12</td>
</tr>
<tr>
<td>InDIGO* (Gu et al., 2019a) (w/ Searched Adaptive Order)</td>
<td>16.14/4.63</td>
<td>3.08/3.10</td>
</tr>
<tr>
<td>Levenshtein Transformer (Parallel Decoding, Gu et al., 2019b)</td>
<td>14.84/3.96</td>
<td>2.84/2.89</td>
</tr>
<tr>
<td>InsT-POINTER-Base (BERT init)</td>
<td>15.63/3.32</td>
<td>3.27/3.30</td>
</tr>
<tr>
<td>InsT-POINTER-Base (BERT init+Wiki)</td>
<td>16.78/3.79</td>
<td>3.49/3.53</td>
</tr>
<tr>
<td>InsT-POINTER-Large (BERT init+Wiki)</td>
<td>19.36/5.78</td>
<td>3.51/3.54</td>
</tr>
<tr>
<td>InsNet (Ours, Fully-Sequential)</td>
<td>13.31/2.30</td>
<td>3.10/3.13</td>
</tr>
<tr>
<td>InsNet-Dinic (Ours, ( \tau = 10.0 ))</td>
<td>16.73/4.35</td>
<td>3.19/3.20</td>
</tr>
</tbody>
</table>

4.1 Lexically Constrained Generation

Experimental Setup

Yelp Review dataset consists of 160K training sequences, 10K sequences for validation and 1K test sequences. News dataset consists of 268586 sentences in total, of which 10K are randomly selected as validation set, 1K for testing. YAKE1 is performed to the test split of each dataset to extract lexical constraints.

We vary the hyperparameter \( \tau \) in the range of \{10.0, 3.0, 1.0, 0.3, \(-\infty\) (fully-sequential)\} and collect the results on both datasets. For position prediction, we are inserting into slots with positions lying in the top 70% of the position distribution mass. For token prediction, we are doing top-\{1, 1, 3, 3, 5\} sampling over the vocabulary distribution. The results are shown in Table 2, Figure 6(a) and 6(b).

Discussion

1) InsNet is able to significantly outperform most previous works on the quality-latency trade-off spectrum. With more sequential flavor (i.e. smaller \( \tau \)), it is generally to achieve the new state-of-the-art in generation quality. Compared to a Plan-And-Write style autoregressive baseline, we are inserting into slots with positions lying in the top 70% of the position distribution mass. For token prediction, we are doing top-\{1, 1, 3, 3, 5\} sampling over the vocabulary distribution. The results are shown in Table 2, Figure 6(a) and 6(b).

1https://github.com/LIAAD/yake
Table 3: Generation examples with insertion steps. Every row shows 3 consecutive steps with orange, green, blue representing the first, second, and third insertion, respectively.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 3: the day decided to started focus on the local group hurt rule out</td>
<td>the day decided to started focus on the local group hurt rule out of the local group hurt rule out of the of the year.</td>
</tr>
<tr>
<td>Step 6: the _day , he decided to get started focus on the local group hurt the rule out of the of the</td>
<td>the _day , he decided to get started focus on the local group hurt the rule out of the of the of the of the of the</td>
</tr>
<tr>
<td>Step 9: _on the day , he decided to get started focus on the court the local group hurt the government rule out of the of the year.</td>
<td>_on the day , he decided to get started focus on the court the local group hurt the government rule out of the of the</td>
</tr>
<tr>
<td>Step 12: but on the next day , he decided to get started to focus on the court for the</td>
<td>but on the next day , he decided to get started to focus on the court for the</td>
</tr>
<tr>
<td>Step 15: but on the next day , he decided to get started to focus on the court for the</td>
<td>but on the next day , he decided to get started to focus on the court for the</td>
</tr>
<tr>
<td>Step 17: but on the next day , he decided to get started to focus on the court for the</td>
<td>but on the next day , he decided to get started to focus on the court for the</td>
</tr>
</tbody>
</table>

**Generation sample**

We also show two samples of the generation process in Table 3. It is interesting to observe that the model tends to insert punctuation and function words (e.g., Articles and Prepositions) first, and then more concrete content words.

### 4.2 Non-autoregressive Machine Translation

We train INSNET-Dinic models with \( \tau = 2.00 \) for extended investigation of the performance on machine learning problems. The results in Table 4 show that, INSNET-Dinic is able to achieve comparable or even better performance compared to previous state-of-the-art non-autoregressive (insertion-based) machine translation models in terms of generation quality and latency.

### 4.3 Training Time Analysis

To show the training efficiency of the proposed model, we hereby do an empirical and theoretical analysis on the candidate models' training procedure. See Table 5 for empirical results, the statistics are collected on the Yelp Review unsupervised lexically constrained text generation problem. All results are collected on a single NVIDIA RTX3090 GPU. The transformer Seq2seq baseline is trained.

Table 4: Machine translation evaluation. Each generation iteration of Levenshtein Transformer requires at least two full executions of the transformer model. Results with a star mark * are collected from our re-implementation. Other baseline results are directly taken from the original papers. The results for a vanilla transformer is taken from the LevT paper (Gu et al., 2019b).

<table>
<thead>
<tr>
<th>Model</th>
<th>Ro-En</th>
<th>En-De</th>
<th>En-Ja</th>
<th>#Dec Step.</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>InDiGO-SAO (w/o KD)</td>
<td>32.47</td>
<td>26.14*</td>
<td>40.87*</td>
<td>n</td>
<td>516ms</td>
</tr>
<tr>
<td>InDiGO-random (w/o KD)</td>
<td>20.20</td>
<td>17.48*</td>
<td>23.91*</td>
<td>n</td>
<td>502ms</td>
</tr>
<tr>
<td>InsT-uniform (+KD)</td>
<td>27.85</td>
<td>26.72</td>
<td>41.89*</td>
<td>( \Theta(\log n) \leq 10 )</td>
<td>107ms</td>
</tr>
<tr>
<td>InsT-binary (( \tau = 0.5 ), +KD)</td>
<td>30.66</td>
<td>27.41</td>
<td>42.17*</td>
<td>( \Theta(\log n) \leq 10 )</td>
<td>92ms</td>
</tr>
<tr>
<td>LevT (+KD) (Gu et al., 2019b)</td>
<td>33.26</td>
<td>27.27</td>
<td>42.36</td>
<td>( \Theta(\log n) \leq 2 \times 10 )</td>
<td>116ms</td>
</tr>
<tr>
<td>Vanilla Transformer (w/o KD)</td>
<td>32.30</td>
<td>27.17</td>
<td>43.68</td>
<td>n</td>
<td>389ms</td>
</tr>
<tr>
<td>INSNET-uniform</td>
<td>29.13</td>
<td>26.45</td>
<td>41.67</td>
<td>( \Theta(\log n) \leq 10 )</td>
<td>92ms</td>
</tr>
<tr>
<td>INSNET-Dinic (Ours, ( \tau = 2.00 ))</td>
<td>33.41</td>
<td>27.36</td>
<td>43.71</td>
<td>( \Theta(\log n) \approx 15.8 )</td>
<td>105ms</td>
</tr>
<tr>
<td>+ KD</td>
<td><strong>33.91</strong></td>
<td><strong>28.05</strong></td>
<td><strong>44.10</strong></td>
<td>( \Theta(\log n) \approx 16.1 )</td>
<td><strong>103ms</strong></td>
</tr>
</tbody>
</table>
Table 5: Empirical & theoretical comparative study of different algorithms’ training efficiency. # of Steps/Seq means during training how many time the transformer needs to be executed per sequence.

<table>
<thead>
<tr>
<th>Model</th>
<th>$T_{\text{train}}$/Epoch</th>
<th># of Steps/Seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-regressive Decoder Transformer</td>
<td>35min48s</td>
<td>1</td>
</tr>
<tr>
<td>InDIGO-SAO</td>
<td>58min36s + 2h12min(SAO)</td>
<td>$1 + O(n)$</td>
</tr>
<tr>
<td>Insertion Transformer-Sequential</td>
<td>26h31min</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Levenshtein Transformer-Sequential</td>
<td>37h24min</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insertion Transformer</td>
<td>8h24min</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Levenshtein Transformer</td>
<td>16h33min</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>InsT-POINTER</td>
<td>9h36min</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>INSNET/INSNET-Dinic</td>
<td>1h12min</td>
<td>1</td>
</tr>
</tbody>
</table>

as a Plan-And-Write model. POINTER’s training time per epoch is calculated with the official implementation with mixed precision supported by the NVIDIA APEX library.

Discussion The results in Table 5 confirm the significant improvement in training efficiency of INSNET over most previous baselines. In addition, this concretely shows that it is not practically affordable to obtain a sequential insertion-based generator directly with InsT/LevT. Note that SAO of InDIGO is an offline, inference-only algorithm that does not require any computation of gradients. Thus its constant factor is significantly smaller than other $O(n)$ gradient-propagating procedures. According to its original paper and our previous experiments, InDIGO practically needs this process to achieve reasonable performance.

4.4 Ablation Study on the Effectiveness of Components

Since the model components of InsNet are closely entangled, it is nontrivial to do ablation study. We design experiments to show that 1) the proposed offset matrix is a powerful insertion-oriented position encoding with significantly better capacity. We show that this particularly helps the model learn to terminate the sequence in correct timings. To study this, we truncate the offset matrix to range $[-1.0, 1.0]$ (InsNet-truncated) so that it is similar to InDIGO’s position encoding, and 2) the global representation is necessary for computing the slot representation in INSNET. It in general helps to improve the sequence termination performance, and leads to convergence to better local optimum. We report the results of an ablated version of InsNet without global representation (InsNet-noglobal).

The termination NLL (see Sec 3.3 for its definition) and BLEU scores for each model can be found in appendix Table 7. We also show some random samples from the ablated variants and the failure cases can be find in appendix Table 8.

5 Conclusion & Future Work

We propose INSNET, an efficient and performant insertion-based generator that supports sequential and parallel decoding. Experiments on two unsupervised lexically constrained text generation datasets and three machine translation datasets show the advantages of INSNET over previous methods.

Future work can explore obtaining a large-scale pre-trained version of INSNET for further fine-tuning under different downstream scenarios. We anticipate such insertion-based models to have better compositional generalizability and controllability.

Acknowledgement

We thank I-Hung Hsu, Dr. Rujun "RJ" Han, Te-Lin Wu, Kai-Wei Chang, Sarik Ghazarian, Alexander Spangher, Yining Hong, Mingyu Derek Ma and all other members from PlusLabNLP/UCLANLP group for their participation in initial discussions and comments on paper writing. We would like to thank Huggingface for their great work of the Transformers project. The work is partially supported by a Meta SRA and an Amazon research gift.

https://github.com/dreasysnail/POINTER
References


Rujun Han, Hong Chen, Yufei Tian, and Nanyun Peng. 2022. Go back in time: Generating flashbacks in stories with event temporal prompts. In 2022 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL).


Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [No]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
(b) Did you include complete proofs of all theoretical results? [Yes] Most theoretical results are self-explanatory.

3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] They will be released upon camera ready.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you’re using/curating? [Yes]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]