Insertion Transformer: Flexible Sequence Generation via Insertion Operations

Mitchell Stern [1 2] William Chan [1] Jamie Kiros [1] Jakob Uszkoreit [1]

Abstract

We present the Insertion Transformer, an iterative, partially autoregressive model for sequence generation based on insertion operations. Unlike typical autoregressive models which rely on a fixed, often left-to-right ordering of the output, our approach accommodates arbitrary orderings by allowing for tokens to be inserted anywhere in the sequence during decoding. This flexibility confers a number of advantages: for instance, not only can our model be trained to follow specific orderings such as left-to-right generation or a binary tree traversal, but it can also be trained to maximize entropy over all valid insertions for robustness. In addition, our model seamlessly accommodates both fully autoregressive generation (one insertion at a time) and partially autoregressive generation (simultaneous insertions at multiple locations). We validate our approach by analyzing its performance on the WMT 2014 English-German machine translation task under various settings for training and decoding. We find that the Insertion Transformer outperforms many prior non-autoregressive approaches to translation at comparable or better levels of parallelism, and successfully recovers the performance of the original Transformer while requiring only logarithmically many iterations during decoding.

1. Introduction

Neural sequence models (Sutskever et al., 2014; Cho et al., 2014) have been successfully applied to many applications, including machine translation (Bahdanau et al., 2015; Luong et al., 2015), speech recognition (Bahdanau et al., 2016; Chan et al., 2016), speech synthesis (Oord et al., 2016a; Wang et al., 2017), image captioning (Vinyals et al., 2015b; Xu et al., 2015) and image generation (Oord et al., 2016b;c). These models have a common theme: they rely on the chain-rule factorization and have an autoregressive left-to-right structure. This formulation bestows many advantages in both training and inference. Log-likelihood computation is tractable, allowing for efficient maximum likelihood learning. Efficient approximate inference is also made possible through beam search decoding. However, the autoregressive framework does not easily accommodate for parallel token generation or more elaborate generation orderings (e.g., tree orders).

More recently, there has been work on non-autoregressive sequence models such as the Non-Autoregressive Transformer (NAT) (Gu et al., 2018) and the Iterative Refinement model (Lee et al., 2018). In both of these models, the decoder is seeded with an initial input derived from the source sequence, then produces the entire target sequence in parallel. Lee et al. (2018) adds an iterative refinement stage to the decoder in which a new hypothesis is produced conditioning on the input and the previous output.

While allowing for highly parallel generation, there are a few drawbacks to such approaches. The first is that the target sequence length needs to be chosen up front, preventing the output from growing dynamically as generation proceeds. This can be problematic if the chosen length is too short to accommodate the desired target, or can be wasteful if it is too long. In the case of Gu et al. (2018), there is also a strong conditional independence assumption between output tokens, limiting the model's expressive power. Lee et al. (2018) relaxes this assumption but in turn requires two separate decoders for the initial hypothesis generation and the iterative refinement stage.

In this work, we present a flexible sequence generation framework based on insertion operations. The Insertion Transformer is an iterative, partially autoregressive model which can be trained in a fully end-to-end fashion. Generation is accomplished by repeatedly making insertions into an initially-empty output sequence until a termination condition is met. Our approach bypasses the problem of needing to predict the target sequence length ahead of time

[1]Google Brain, Mountain View, Toronto, Berlin [2]University of California, Berkeley. Correspondence to: Mitchell Stern <mitchell@berkeley.edu>, William Chan <williamchan@google.com>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

Serial generation:

t	Canvas	Insertion
0	[]	$(\text{ate}, 0)$
1	[ate]	$(\text{together}, 1)$
2	[ate, together]	$(\text{friends}, 0)$
3	[friends, ate, together]	$(\text{three}, 0)$
4	[three, friends, ate, together]	$(\text{lunch}, 3)$
5	[three, friends, ate, lunch, together]	$(\langle\text{EOS}\rangle, 5)$

Parallel generation:

t	Canvas	Insertions
0	[]	$(\text{ate}, 0)$
1	[ate]	$(\text{friends}, 0)$, $(\text{together}, 1)$
2	[friends, ate, together]	$(\text{three}, 0)$, $(\text{lunch}, 2)$
3	[three, friends, ate, lunch, together]	$(\langle\text{EOS}\rangle, 5)$

Figure 1. Examples demonstrating how the clause "three friends ate lunch together" can be generated using our insertion framework. On the left, a serial generation process is used in which one insertion is performed at a time. On the right, a parallel generation process is used with multiple insertions being allowed per time step. Our model can either be trained to follow specific orderings or to maximize entropy over all valid actions. Some options permit highly efficient parallel decoding, as shown in our experiments.

by allowing the output to grow dynamically, and also permits deviation from classic left-to-right generation, allowing for more exotic orderings like balanced binary trees.

During inference, the Insertion Transformer can be used in an autoregressive manner for serial decoding, with one insertion operation being applied at a time, or in a partially autoregressive manner for parallel decoding, with insertions at multiple locations being applied simultaneously. This allows for the target sequence to grow exponentially in length. In the case of a balanced binary tree order, our model can use as few as $\lfloor \log_2 n \rfloor + 1$ operations to produce a sequence of length n, which we find achievable in practice using an appropriately chosen loss function during training.

2. Sequence Generation via Insertion Operations

In this section, we describe the abstract framework used by the Insertion Transformer for sequence generation. The next section then describes the concrete model architecture we use to implement this framework.

We begin with some notation. Let x be our source canvas and y be our target canvas. In the regime of sequence modeling, a canvas is a sequence and we use the terms interchangeably. While this paper focuses on sequence generation, we note that our framework can be generalized to higher-dimensional outputs (e.g., image generation).

Let $\hat{y}_t$ be the hypothesis canvas at time t. Because our framework only supports insertions and not reordering operations, it must be a subsequence of the final output hypothesis $\hat{y}$. For example, if the eventual output were $\hat{y} = [A, B, C, D, E]$, then $\hat{y}_t = [B, D]$ would be a valid intermediate canvas while $\hat{y}_t = [B, A]$ would not. We do not restrict ourselves to one insertion per step, meaning $\hat{y}_t$ could have more than t tokens.

Further, let $\mathcal{C}$ be our content vocabulary (i.e., token vocabulary for sequences). At each iteration t, the Insertion Transformer produces a joint distribution over the choice of content $c \in \mathcal{C}$ and all available insertion locations $l \in [0, |\hat{y}_t|]$ in the current hypothesis canvas $\hat{y}_t$. In other words, the Insertion Transformer models both what to insert and where to insert relative to the current canvas hypothesis $\hat{y}_t$:

$$p(c, l \mid x, \hat{y}_t) = \text{InsertionTransformer}(x, \hat{y}_t). \quad (1)$$

As an example, suppose our current hypothesis canvas is $\hat{y}_t = [B, D]$ and we select the insertion operation $(c = C, l = 1)$. This will result in the new hypothesis canvas $\hat{y}_{t+1} = [B, C, D]$. Also see Figure 1 for an example showing the full generation process for a typical English sentence.

The permitted insertion locations allow for insertions anywhere in the canvas from the leftmost slot ($l = 0$) to the rightmost slot ($l = |\hat{y}_t|$). Generation always begins with an empty canvas $\hat{y}_0 = []$ with just a single insertion location $l = 0$, and concludes when a special marker token is emitted. Exact details on termination handling can be found in Section 4.4, where we describe two variants.

3. Insertion Transformer Model

The concrete model we use for the Insertion Transformer is a modified version of the original Transformer (Vaswani et al., 2017), with the decoder having been altered to induce a distribution over insertions anywhere in the current output rather than just at the end. We outline the key changes below.

Full Decoder Self-Attention. We remove the causal self-attention mask from the decoder so that all positions can attend to all other positions, as opposed to just those to the left of the current position. This allows each decision to condition on the full context of the canvas hypothesis for the current iteration.

Slot Representations via Concatenated Outputs. The standard Transformer decoder produces n vectors for a sequence of length n, one per position, with the last one being used to pick the next word. Our model instead requires $n + 1$ vectors, one for each of the $n - 1$ slots between words plus 2 for the beginning and end slots. We achieve this by adding special marker tokens at the beginning and end of the decoder input to extend the sequence length by two. We then take the resulting $n + 2$ vectors in the final layer and concatenate each adjacent pair to obtain $n + 1$ slot representations. Hence each slot is summarized by the final representations of the positions to its left and right.

3.1. Model Variants

Beyond the required structural changes above, there are several variations of our model that we explore.

Content-Location Distribution. We need to model the joint content-location distribution for the insertion operations. We present two approaches: the first directly models the joint distribution, the second relies on a factorization.

Let $H \in \mathbb{R}^{(T+1)\times h}$ be the matrix of slot representations, where h is the size of the hidden state and T is the length of the current partial hypothesis. Let $W \in \mathbb{R}^{h\times|\mathcal{C}|}$ be the standard softmax projection matrix from the Transformer model. We can simply use this projection matrix to compute the content-location logits, then flatten this matrix into a vector and directly take the softmax over all the content-location logits to obtain a jointly normalized distribution:

$$p(c, l) = \text{softmax}(\text{flatten}(HW)). \tag{2}$$

Another approach is to model the joint distribution using a conditional factorization, $p(c, l) = p(c \mid l)p(l)$. We can model the conditional content distribution as is done in the normal Transformer:

$$p(c \mid l) = \text{softmax}(h_l W), \tag{3}$$

where $h_l \in \mathbb{R}^h$ is the l-th row of H. In other words, we apply the softmax per-row in the matrix HW. We separately model the location distribution by taking the softmax of the dot product of the hidden states and a learnable query vector $q \in \mathbb{R}^h$:

$$p(l) = \text{softmax}(Hq). \tag{4}$$

This approach requires a small number of additional parameters h compared to modeling the joint distribution directly.

Contextualized Vocabulary Bias. To increase information sharing across slots, we can perform a max pooling operation over the final decoder hidden vectors H to obtain a context vector $g \in \mathbb{R}^h$. We then project g into the vocabulary space using a learned projection matrix $V \in \mathbb{R}^{h\times|\mathcal{C}|}$ to produce a shared bias $b \in \mathbb{R}^{|\mathcal{C}|}$. We then add b to the result to the vocabulary logits at each position as an additional shared bias. We believe this may be useful in providing the model with coverage information, or in propagating count information about common words that should appear in multiple places in the output. Formally, we have

$$b = \text{maxpool}(H)V \tag{5}$$
$$B = \text{repmat}(b, [T + 1, 1]) \tag{6}$$
$$p(c, l) = \text{softmax}(HW + B) \tag{7}$$

Mixture-of-Softmaxes Output Layer. Unlike the output vectors of a typical autoregressive model which only need to capture distributional information about the next word, the slot vectors in our model are responsible for representing entire bags of words. Moreover, depending on the order of generation, they might correspond to any contiguous span of the final output, making this a highly nontrivial modeling problem. We posit that the language modeling softmax bottleneck identified by Yang et al. (2018) poses even greater challenges for our setup. We try including the mixture-of-softmaxes layer proposed in their work as one means of addressing the issue.

4. Training and Loss Functions

The Insertion Transformer framework is flexible enough to accommodate arbitrary generation orders, including those which are input- and context-dependent. We discuss several order loss functions that we can optimize for.

4.1. Left-to-Right

As a special case, the Insertion Transformer can be trained to produce its output in a left-to-right fashion, imitating the conventional setting where this ordering is enforced by construction. To do so, given a training example (x, y), we randomly sample a length $k \sim \text{Uniform}([0, |y|])$ and take the current hypothesis to be the left prefix $\hat{y} = (y_1, \ldots, y_k)$. We then aim to maximize the probability of the next content in the sequence $c = y_k$ in the rightmost slot location $l = k$, using the negative log-likelihood of this action as our loss to be minimized:

$$\text{loss}(x, \hat{y}) = -\log p(y_{k+1}, k \mid x, \hat{y}). \tag{8}$$

When the sequence is complete, i.e. $k = n$, we take y_{k+1} to be the end-of-sequence token $\langle\text{EOS}\rangle$. We note that there are several differences between our left-to-right order loss and a standard autoregressive Transformer log-probability loss. We describe them in detail in Section 4.5.

4.2. Balanced Binary Tree

A left-to-right strategy only allows for one token to be inserted at a time. On the other end of the spectrum, we can train for maximal parallelism by using a balanced binary tree ordering. The centermost token is produced first, then the center tokens of the spans on either side are produced next, and this process is recursively continued until the full sequence has been generated. As an example, for the target output $[A, B, C, D, E, F, G]$, the desired order of production would be $[] \rightarrow [D] \rightarrow [B, D, F] \rightarrow [A, B, C, D, E, F, G]$, where multiple insertions are executed in parallel. See Section 5 for more details on parallel decoding.

To achieve this goal, we use a soft binary tree loss encouraging the model to assign high probability to tokens near the middle of the span represented by a given slot. Partial canvas hypotheses are generated randomly so as to improve robustness and reduce exposure bias.

In more detail, given a training example (x, y), we first sample a subsequence $\hat{y}$ from the set of all subsequences of the target y. One option would be to sample uniformly from this set, which could be accomplished by iterating through each token and keeping or throwing it out with probability $1/2$. Though simple, this approach would overexpose the model to partial outputs with length close to $|y|/2$ and would underexpose it to hypotheses that are nearly empty or nearly complete.

To circumvent this issue, we instead use a biased sampling procedure that gives uniform treatment to all lengths. In particular, we first sample a random length $k \sim \text{Uniform}([0, |y|])$, then sample a random subsequence of y of length k. The latter step is carried out by constructing an index list $[1, \dots, |y|]$, shuffling it, and extracting the tokens corresponding to the first k indices in the order they appear in the target sequence y.

Once we have our randomly chosen hypothesis $\hat{y}$, it remains to compute the loss itself. For each of the $k + 1$ slots at locations $l = 0, \dots, k$, let $(y_{i_l}, y_{i_l+1}, \dots, y_{j_l})$ be the span of tokens from the target output yet to be produced at location l. We first define a function d_l giving the distance from the center of the span corresponding to location l:

$$d_l(i) = \left| \frac{i_l + j_l}{2} - i \right|. \tag{9}$$

We use the negative distance function $-d_l$ as the reward function for a softmax weighting policy w_l (Rusu et al., 2016; Norouzi et al., 2016) (see Figure 2 for an illustration):

$$w_l(i) = \frac{\exp(-d_l(i)/\tau)}{\sum_{i'=i_l}^{j_l} \exp(-d_l(i')/\tau)}. \tag{10}$$

Next we define the slot loss at location l as a weighted sum of the negative log-likelihoods of the tokens from its corresponding span:

$$\text{slot-loss}(x, \hat{y}, l) = \sum_{i=i_l}^{j_l} - \log p(y_i, l \mid x, \hat{y}) \cdot w_l(i). \tag{11}$$

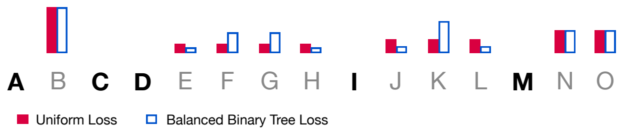

Figure 2. A visualization of the weighting of the per-token negative log-likelihoods in the balanced binary tree and uniform losses. The balanced binary tree loss strongly incentivizes the generation of the center word or center words within each slot.

In other words, the loss encourages the model to prioritize the tokens closest to the center based on d_l. The temperature hyperparameter τ allows us to control the sharpness of the weight distribution, with $\tau \rightarrow 0$ approaching a peaked distribution placing all the weight on the centermost token (or centermost two tokens in the case of an even-length span), and $\tau \rightarrow \infty$ approaching a uniform distribution over all the missing content for a slot.

Finally, we define the full loss as the average of slot losses across all locations:

$$\text{loss}(x, \hat{y}) = \frac{1}{k+1} \sum_{l=0}^{k} \text{slot-loss}(x, \hat{y}, l). \tag{12}$$

4.3. Uniform

In addition to encouraging the model to follow a particular generation order, we can also train it to learn an agnostic view of the world in which it assigns equal probability mass to each correct action with no special preference. This neutral approach is useful insofar as it forces the model to be aware of all valid actions during each step of decoding, providing a rich learning signal during training and maximizing robustness.

Such an approach also bears resemblance to the principle of maximum entropy, which has successfully been employed for maximum entropy modeling across a number of domains in machine learning.

To implement this loss, we simply take $\tau \rightarrow \infty$ in the binary tree loss of the previous section, yielding a slot loss of

$$\text{slot-loss}(x, \hat{y}, l) = \frac{1}{j_l - i_l + 1} \sum_{i=i_l}^{j_l} - \log p(y_i, l \mid x, \hat{y}). \tag{13}$$

This is the mean of the negative log-probabilities of the correct actions for the given slot, which we note is maximized by a uniform distribution. Then as before, we take the full loss to be the mean of the slot losses.

4.4. Termination

We experiment with two termination conditions for the binary tree and uniform losses, slot finalization and sequence finalization, and compare their empirical performance in our experiments.

For slot finalization, when computing the slot loss for a location corresponding to an empty span in the true output, we take the target to be a single end-of-slot token. Then, all slot losses are always well-defined, and at generation time we can cease decoding when all slots predict an end-of-slot. We note for clarity that this special token appears in the vocabulary of the model but is never actually produced; see Section 5 for more details.

Alternatively, for sequence finalization, we leave the slot losses undefined for empty spans and exclude them from the overall loss. Once the entire sequence has been produced and all locations correspond to empty spans, we take the slot loss at every location to be the negative log-likelihood of an end-of-sequence token. This is identical to the slot finalization approach at the very end, but differs while generation is ongoing as no signal is provided for empty slots.

4.5. Training Differences

In a typical neural autoregressive model, there is a unidirectional flow of information in the decoder. This allows hidden states to be propagated (and reused) across time steps during the generation process, since they will remain unaltered as the hypothesis is extended rightward. In contrast, because we allow for insertions anywhere in the sequence, our approach lacks this unidirectional property and we must recompute the decoder hidden states for each position after every insertion.

This has several consequences. First, there is no state (or gradient) propagation between generation steps. Next, instead of being able to efficiently compute the losses for all generation steps of an example in one fell swoop as is usually done, we can only compute the loss for one generation step at a time under the same memory constraints. Accordingly, our batch size is effectively reduced by a factor of the average sequence length, which has the potential to affect convergence speed and/or model quality. Finally, since we need to subsample generation steps during training, as opposed to a standard Transformer that can compute all the generation steps in a sequence for free, our gradient suffers from extra variance due to the sampling process. Under the right training conditions, however, we find these not to be major hindrances.

5. Inference

Recall that at each time step t, the Insertion Transformer yields a distribution $p(c, l \mid x, \hat{y}_t)$ over content c and location l given the input sequence x and current partial output sequence $\hat{y}_t$. This highly flexible model opens the door for both sequential and parallel inference techniques, which we describe in more detail below.

5.1. Greedy Decoding

First we have a standard greedy approach to decoding, in which the action with the highest probability across all choices of content c and location l is selected:

$$(\hat{c}_t, \hat{l}_t) = \operatorname*{argmax}_{c,l} p(c, l \mid x, \hat{y}_t). \tag{14}$$

Once the best decision has been identified, we insert token $\hat{c}_t$ at location $\hat{l}_t$ to obtain the next partial output $\hat{y}_{t+1}$.

For models trained towards sequence finalization, this process continues until an end-of-sequence token gets selected at any location, at which point the final output is returned.

For models trained towards slot finalization, we restrict the argmax to locations whose maximum-probability decision is not end-of-slot, and finish only when the model predicts an end-of-slot token for every location.

5.2. Parallel Decoding

If we train an Insertion Transformer towards slot finalization, we can also parallelize inference across slots within each time step to obtain a simple partially autoregressive decoding algorithm.

In more detail, for each location l we first compute the following maximum-probability actions:

$$\hat{c}_{l,t} = \operatorname*{argmax}_{c} p(c \mid l, x, \hat{y}_t). \tag{15}$$

For the version of the model whose joint distribution factors as $p(c, l) = p(l)p(c \mid l)$, the required conditional distribution $p(c \mid l)$ is already available. For the jointly normalized model, we can either obtain the conditional via renormalization as $p(c \mid l) = p(c, l)/p(l) = p(c, l)/\sum_{c'} p(c', l)$, or compute it directly by taking a softmax over the subset of logits at location l. In both cases, all the required conditional distributions can be computed in parallel.

Next, we filter out the locations for which the maximum-probability decision is an end-of-slot token, and for each location l that remains, insert the selected token $\hat{c}_{l,t}$ into that slot. The resulting sequence becomes the next partial output $\hat{y}_{t+1}$. This process continues until an end-of-slot token is predicted at every location.

Since the parallel decoding scheme described here allows for a token to be inserted in every slot at every time step, a sequence of length n could theoretically be generated in as few as $\lfloor \log_2 n \rfloor + 1$ steps. We find that this logarithmic complexity is attainable in practice in our experiments.

Loss	**Termination**	**BLEU (+EOS)**	**BLEU (+EOS)**	**BLEU (+EOS)**
			+Distillation	**+Distillation, +Parallel**
Left-to-Right	Sequence	20.92 (20.92)	23.29 (23.36)	-
Binary Tree ($\tau = 0.5$)	Slot	20.35 (21.39)	24.49 (25.55)	25.33 (25.70)
Binary Tree ($\tau = 1.0$)	Slot	21.02 (22.37)	24.36 (25.43)	25.43 (25.76)
Binary Tree ($\tau = 2.0$)	Slot	20.52 (21.95)	24.59 (25.80)	25.33 (25.80)
Uniform	Sequence	19.34 (22.64)	22.75 (25.45)	-
Uniform	Slot	18.26 (22.16)	22.39 (25.58)	24.31 (24.91)

Table 1. Development BLEU scores obtained via greedy decoding for our basic models trained with various loss functions and termination strategies. The +EOS numbers are the BLEU score obtained when an EOS penalty is applied during decoding to discourage premature stopping. The +Distillation numbers are for models trained with distilled data. The +Parallel numbers are obtained with parallel decoding, which is applicable to models trained with the slot finalization termination condition.

6. Experiments

In this section, we explore the efficacy of our approach on a machine translation task, analyzing its performance under different training conditions, architectural choices, and decoding procedures. We experiment on the WMT 2014 English-German translation dataset, using newstest2013 for development and newstest2014 for testing, respectively. All our experiments are implemented in TensorFlow (Abadi et al., 2015) using the Tensor2Tensor framework (Vaswani et al., 2018). We use the default `transformer_base` hyperparameter set reported by Vaswani et al. (2018) for all hyperparameters not specific to our model. We perform no additional hyperparameter tuning. All our models are trained for 1,000,000 steps on eight P100 GPUs.

6.1. Baseline Results

We first train the baseline version of our model with different choices of loss functions and termination strategies. Greedy decoding results on the development set are given for each setting in the third column of Table 1.

We observe that the binary tree loss performs the best when standard greedy decoding is used, attaining a development BLEU score of 21.02. We also find that our left-to-right models do poorly compared to other orderings. One explanation is that the gradients of the binary tree and uniform losses are much more informative, in that they capture information on all the missing tokens, whereas left-to-right only provides information about the next one. We note that in all cases, even after 1,000,000 steps the models are still improving and do not appear to overfit.

Upon inspecting the outputs of these models, we found that some of the most common and severe mistakes were due to the model assigning high probability to the terminal token (end-of-slot or end-of-sequence, both abbreviated as EOS) too early in the decoding process, resulting in artificially short outputs. To rectify this, we introduce an **EOS penalty** hyperparameter, which is a scalar subtracted from the log-probability assigned by the model to an EOS at each location during decoding. Using a penalty of β prevents the model from selecting an EOS unless there is a difference of at least β between the log-probability of EOS and the log-probability of the second-best choice. This approach is similar the length normalization techniques used in many sequence models (Graves, 2012). We perform a sweep over the range $[0, 7]$ and report the best result for each model in parentheses. A well-chosen EOS penalty can have a sizable effect, increasing the BLEU score by nearly 4 points in some cases, and its inclusion brings the highest development score to 22.64 for the uniform loss with sequence-level finalization.

6.2. Knowledge Distillation

Following prior work (Gu et al., 2018; Stern et al., 2018), we also train our models with knowledge distillation (Hinton et al., 2015; Kim & Rush, 2016) from a standard Transformer. We observe improvements of 3 to 4 BLEU points across the board, showing that distillation is remarkably effective for our setting. As before, the models trained with a binary tree loss are approximately 2 BLEU points better than those trained with a uniform loss when standard decoding is performed, but the differences largely vanish when using a properly-tuned EOS penalty for each model. The best model by a small margin is the one trained with a binary tree loss with temperature $\tau = 2.0$, which achieves a 25.80 BLEU score on the development set.

6.3. Architectural Variants

Next we explore different combinations of the architectural variants described in Section 3.1. Using the uniform loss, slot finalization, and distillation as a neutral baseline configuration, we train each variant and decode on the development set to obtain the results given in Table 2.

Many of the configurations help improve performance when decoding without an EOS penalty. In particular, using joint

Joint	Contextual	Mixture	BLEU (+EOS)
✗	✗	✗	22.39 (25.58)
✓	✗	✗	22.92 (25.14)
✗	✓	✗	23.00 (25.41)
✗	✗	✓	22.19 (25.58)
✓	✓	✗	23.22 (25.44)
✓	✗	✓	20.17 (24.19)
✗	✓	✓	23.29 (25.48)
✓	✓	✓	22.16 (25.44)

Table 2. Development BLEU scores obtained via greedy decoding when training models with the architectural variants discussed in Section 3.1. All models are trained with a uniform loss and slot finalization on distilled data.

Model	BLEU (+EOS)
Binary Tree ($\tau = 0.5$)	25.33 (25.70)
Binary Tree ($\tau = 1.0$)	25.43 (25.76)
Binary Tree ($\tau = 2.0$)	25.33 (25.80)
Uniform	24.31 (24.91)
Uniform + Contextual	24.54 (24.74)
Uniform + Mixture	24.33 (25.11)
Uniform + Contextual + Mixture	24.68 (25.02)

Table 3. Parallel decoding results on the development set for some of our stronger models. All numbers are comparable to or even slightly better than those obtained via greedy decoding, demonstrating that our model can perform insertions in parallel with little to no cost for end performance.

normalization, a contextualized vocabulary bias, or both leads to improvements of 0.5-0.8 BLEU over the baseline. Once we tune the EOS penalty for each setting, however, the improvements largely disappear. The best configurations, primarily those involving mixture-of-softmaxes, are within 0.1 BLEU of the baseline. This suggests that the core architecture is already sufficiently powerful when decoding is well-tuned, but that it may be useful to consider some variations when looking at other inference settings.

6.4. Parallel Decoding

Thus far, all our experiments have used greedy decoding. However, as described in Section 5, models trained towards slot finalization also permit a parallel decoding scheme in which tokens are simultaneously inserted into every unfinished slot at each time step until no such slots remain. We decode the development set using this strategy for some of our more promising models, giving results in Table 3. An example decode is provided in Figure 4 for reference.

First and foremost, we observe that all scores are on par with those obtained via greedy decoding, and in some cases are even better. This demonstrates that with a proper training objective, our model can seamlessly accommodate parallel insertions with little effect on end performance. The fact that some scores are improved suggests that greedy search may suffer from issues related to local search that are circumvented by making multiple updates to the hypothesis at once. We leave this as an interesting topic for future investigation.

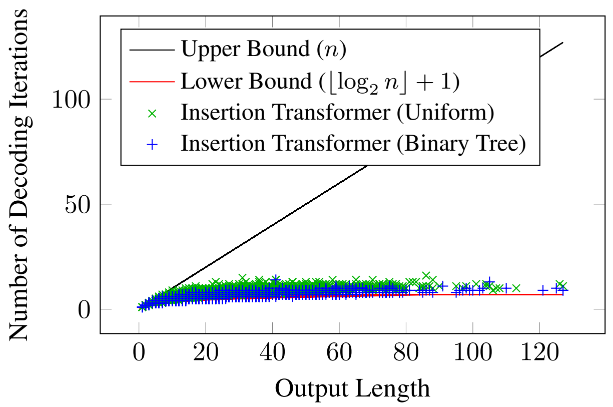

Figure 3. Plot showing number of decoding iterations versus output length as measured on the development set for our best models. To produce an output of length n, an insertion-based model requires at least $\lfloor \log_2 n \rfloor + 1$ iterations and at most n. While greedy decoding cannot do better than the upper bound, our parallel decoding scheme nearly achieves the lower bound in all cases.

In addition, we find that parallel decoding also helps close the gap between results obtained with and without an EOS penalty. We believe this may be due in part to the fact that the number of decoding iterations is reduced substantially, thereby giving fewer opportunities for the model to erroneously stop at an intermediate state.

We also perform a more careful analysis of the extent of the parallelism achieved by our highest-scoring models. In Figure 3, we plot the number of decoding iterations taken vs. the output length n for each development sentence. We also plot the theoretical lower bound of $\lfloor \log_2 n \rfloor + 1$ and the upper bound of n on the number of iterations. Note that greedy decoding takes n steps by definition. Our best model comes impressively close to the lower bound across the entire development set, rarely deviating by more than 1 or 2 iterations. This demonstrates that our framework is capable of producing high-quality output using a sub-linear (i.e. logarithmic) number of generation steps. In terms of wall-clock speedup, our best binary tree model exhibits a 4.19x latency improvement over a baseline Transformer for single-sentence decoding of the development set on GPU.

6.5. Test Results

Finally we report results in Table 4 on the newstest2014 test set using our best hyperparameters as measured on the development set. When compared with related approaches, we find that we match the high quality of models requiring a

Input: But on the other side of the state, that is not the impression many people have of their former governor.

Output: Aber auf der anderen Seite des Staates ist das nicht der Eindruck, den viele von ihrem ehemaligen Gouverneur haben.

Parallel decode (binary tree loss):

```
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ , _den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur _haben_ ._
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ , _den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur _haben_ ._
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ , _den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur _haben_ ._
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ , _den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur _haben_ ._
Aber_ auf_ der_ anderen_ Seite_ des_ Staates_ ist_ das_ nicht_ der_ Eindruck_ , _den_ viele_ von_ ihrem_ ehemaligen_ Gouverneur _haben_ ._
```

Figure 4. Example decode using a model trained with the binary tree loss. Within each row, the underlined blue words are those being inserted, and the gray words represent those from the final output that have not yet been generated. We observe that the model is able to achieve a high degree of parallelism, matching the logarithmic theoretical lower bound on the number of parallel decoding iterations thanks to its training objective.

Model	**BLEU**	**Iterations**
Autoregressive Left-to-Right		
Transformer (Vaswani et al., 2017)	27.3	n
Semi-Autoregressive Left-to-Right		
SAT (Wang et al., 2018)	24.83	$n/6$
Blockwise Parallel (Stern et al., 2018)	27.40	$\approx n/5$
Non-Autoregressive		
NAT (Gu et al., 2018)	17.69	1
Iterative Refinement (Lee et al., 2018)	21.61	10
Our Approach (Greedy)		
Insertion Transformer + Left-to-Right	23.94	n
Insertion Transformer + Binary Tree	27.29	n
Insertion Transformer + Uniform	27.12	n
Our Approach (Parallel)		
Insertion Transformer + Binary Tree	27.41	$\approx \log_2 n$
Insertion Transformer + Uniform	26.72	$\approx \log_2 n$

Table 4. BLEU scores on the newstest2014 test set for the WMT 2014 English-German translation task. Our parallel decoding strategy attains the same level of accuracy reached by linear-complexity models while using only a logarithmic number of decoding steps.

linear number of iterations while using a logarithmic number of generation steps. In practice, as shown in Figure 3, we rarely require more than 10 generation steps, meaning our empirical complexity even matches that of Lee et al. (2018) who use a constant 10 steps. When trained with the binary tree loss, we find that the Insertion Transformer is able to match the standard Transformer model while requiring substantially fewer generation iterations.

7. Related Work

There has been prior work on non-left-to-right autoregressive generation. Vinyals et al. (2015a) explores the modeling of sets, where generation order does not matter. Ford et al. (2018) explores language modeling where select words (i.e., functional words) are generated first, and the rest are filled in using a two-pass process. There has also been prior work in hierarchical autoregressive image generation (Reed et al., 2017), where $\log n$ steps are required to generate n tokens. This bears some similarity to our balanced binary tree order.

Shah et al. (2018) also recently proposed generating language with a dynamic canvas. Their work can be seen as a continuous relaxation version of our model, wherein their canvas is an embedding space, while our canvas contains discrete tokens. They applied their approach to language modeling tasks, whereas we apply ours to conditional language generation in machine translation.

In addition, there has been recent work on non-autoregressive machine translation (Gu et al., 2018; Lee et al., 2018) and semi-autoregressive translation (Stern et al., 2018; Wang et al., 2018). The key difference between our work and prior work is that the Insertion Transformer framework can accommodate for a dynamically growing canvas size while still achieving sub-linear generation complexity. Other models also tend to degrade with increasing parallelism, while our model trained with the balanced binary tree loss suffers no model degradation under parallel decoding.

8. Conclusion

In this paper, we presented the Insertion Transformer, a partially autoregressive model for sequence generation based on insertion operations. Our model can be trained to follow arbitrary generation orderings, such as a left-to-right order or a balanced binary tree order, or can be optimized to learn all possible orderings, making it also applicable to completion or infilling tasks. The model can be decoded serially, producing one token at a time, or it can be decoded in parallel with simultaneous insertions at multiple locations. When using the binary tree loss, we find empirically that we can generate sequences of length n using close to the asymptomatic limit of $\lfloor \log_2 n \rfloor + 1$ steps without any quality degradation. This allows us to match the performance of the standard Transformer on the WMT 2014 English-German translation task while using substantially fewer iterations during decoding.

Acknowledgements

We give thanks to Zhifeng Chen, Geoffrey Hinton, Mohammad Norouzi, Oscar Täckström and the Google Brain Team for useful comments, discussions, and technical assistance.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.

Bahdanau, D., Cho, K., and Bengio, Y. Neural Machine Translation by Jointly Learning to Align and Translate. In *ICLR*, 2015.

Bahdanau, D., Chorowski, J., Serdyuk, D., Brakel, P., and Bengio, Y. End-to-End Attention-based Large Vocabulary Speech Recognition. In *ICASSP*, 2016.

Chan, W., Jaitly, N., Le, Q., and Vinyals, O. Listen, Attend and Spell: A Neural Network for Large Vocabulary Conversational Speech Recognition. In *ICASSP*, 2016.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *EMNLP*, 2014.

Ford, N., Duckworth, D., Norouzi, M., and Dahl, G. E. The Importance of Generation Order in Language Modeling. In *EMNLP*, 2018.

Graves, A. Sequence Transduction with Recurrent Neural Networks. In *ICML Representation Learning Workshop*, 2012.

Gu, J., Bradbury, J., Xiong, C., Li, V. O., and Socher, R. Non-Autoregressive Neural Machine Translation. In *ICLR*, 2018.

Hinton, G., Vinyals, O., and Dean, J. Distilling the Knowledge in a Neural Network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

Kim, Y. and Rush, A. M. Sequence-Level Knowledge Distillation. In *EMNLP*, 2016.

Lee, J., Mansimov, E., and Cho, K. Deterministic Non-Autoregressive Neural Sequence Modeling by Iterative Refinement. In *EMNLP*, 2018.

Luong, M.-T., Pham, H., and Manning, C. D. Effective Approaches to Attention-based Neural Machine Translation. In *EMNLP*, 2015.

Norouzi, M., Bengio, S., Zhifeng Chen, N. J., Schuster, M., Wu, Y., and Schuurmans, D. Reward Augmented Maximum Likelihood for Neural Structured Prediction. In *NIPS*, 2016.

Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. WaveNet: A Generative Model for Raw Audio. In *arXiv*, 2016a.

Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. Pixel Recurrent Neural Networks. In *ICML*, 2016b.

Oord, A., Kalchbrenner, N., Vinyals, O., Espeholt, L., Graves, A., and Kavukcuoglu, K. Conditional Image Generation with PixelCNN Decoders. In *NIPS*, 2016c.

Reed, S., van den Oord, A., Kalchbrenner, N., Colmenarejo, S. G., Wang, Z., Belov, D., and de Freitas, N. Parallel Multiscale Autoregressive Density Estimation. In *ICML*, 2017.

Rusu, A. A., Colmenarejo, S. G., Gulcehre, C., Desjardins, G., Kirkpatrick, J., Pascanu, R., Mnih, V., Kavukcuoglu, K., and Hadsell, R. Policy Distillation. In *ICLR*, 2016.

Shah, H., Zheng, B., and Barber, D. Generating Sentences Using a Dynamic Canvas. In *AAAI*, 2018.

Stern, M., Shazeer, N., and Uszkoreit, J. Blockwise Parallel Decoding for Deep Autoregressive Models. In *NeurIPS*, 2018.

Sutskever, I., Vinyals, O., and Le, Q. Sequence to Sequence Learning with Neural Networks. In *NIPS*, 2014.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need. In *NIPS*, 2017.

Vaswani, A., Bengio, S., Brevdo, E., Chollet, F., Gomez, A. N., Gouws, S., Jones, L., Kaiser, L., Kalchbrenner, N., Parmar, N., Sepassi, R., Shazeer, N., and Uszkoreit, J. Tensor2Tensor for Neural Machine Translation. In *AMTA*, 2018.

Vinyals, O., Bengio, S., and Kudlur, M. Order Matters: Sequence to sequence for sets. In *ICLR*, 2015a.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. Show and Tell: A Neural Image Caption Generator. In *CVPR*, 2015b.

Wang, C., Zhang, J., and Chen, H. Semi-Autoregressive Neural Machine Translation. In *EMNLP*, 2018.

Wang, Y., Skerry-Ryan, R., Stanton, D., Wu, Y., Weiss, R. J., Jaitly, N., Yang, Z., Xiao, Y., Chen, Z., Bengio, S., Le, Q., Agiomyrgiannakis, Y., Clark, R., and Saurous, R. A. Tacotron: Towards End-to-End Speech Synthesis. In *INTERSPEECH*, 2017.

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R., and Bengio, Y. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *ICML*, 2015.

Yang, Z., Dai, Z., Salakhutdinov, R., and Cohen, W. W. Breaking the Softmax Bottleneck: A High-Rank RNN Language Model. In *ICLR*, 2018.