InsNeXt: Training Scalable Insertion-Based Language Models From Scratch

Anonymous Author(s)
Affiliation
Address
email

Abstract

Insertion-based language models like Insertion Transformer and InsNet have shown promises as strong alternatives to autoregressive models with better inference-time efficiency and controllablility. However, their training-time scalability has been limited by computational inefficiency and obsolete model designs. We aim to tackle this problem with **InsNeXt**, an insertion-based language model architecture integrating recent advancements of language model systems to achieve improved scalability. We scale InsNeXt from 154M up to as large as 0.6B parameters with context window of 4096 by combining sentence-level training and document-level training to better encode the context and bring out the benefits of insertion-based models to encode bi-directional contexts. In addition, we propose a novel context encoding mechanism specialized for insertion-based decoding. The inference-time mechanism sparsely introduces bidirectional re-encoding of context, thus effectively leverages the models' bidirectional context reception while preserving the same level of computational efficiency as conventional autoregressive decoding. We evaluate the pretrained InsNeXt models from the perspective of representation learning, commonsense reasoning and controllable generation. InsNeXt models achieve similar or better performance in comparison to the state-of-the-art similar-sized autoregressive models, making them a class of solid representation learners and powerful controllable insertion-based generators.

1 Introduction

Large-scale pretrained autoregressive language models have dominated the paradigm of natural language generation over the past few years. These models, including the GPTs (Radford et al., 2018, 2019; Brown et al., 2020; OpenAI, 2022; Achiam et al., 2023), LLaMAs (Touvron et al., 2023a,b; Dubey et al., 2024; Meta, 2025), Phis (Li et al., 2023a,b; Javaheripi et al., 2023; Abdin et al., 2024), Qwen LLMs (Bai et al., 2023; Yang et al., 2024a,b) and Deepseek LLMs (Bi et al., 2024; DeepSeek-AI et al., 2024; Liu et al., 2024; Guo et al., 2025), have demonstrated impressive training-time scalability and versatile performance across a wide range of tasks. However, their inference-time efficiency and lack of controllability motivate people to explore alternative methods for pretrained language models.

One potential alternative is the *insertion-based* language model (Stern et al., 2019; Gu et al., 2019; Lu et al., 2022). Unlike autoregressive models, which generate text strictly from left to right, insertion-based models allow tokens to be generated at arbitrary positions in arbitrary order, making them inherently more flexible and better aligned with the compositional nature of human language. Moreover, their potential for improved controllability and fine-grained editing offers unique advantages in tasks requiring structured or context-sensitive generation (Zhang et al., 2020). Despite these benefits, attempts to scale up insertion-based models remain limited, mainly due to their training-time inefficiency especially compared to modern *large language models* (LLMs).

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

The advent of efficient training techniques and architectural improvements in modern LLMs offers a pathway to overcoming these limitations. Practices such as FlashAttention/Memory Efficient Attention (Dao et al., 2022; Dao, 2023; Shah et al., 2024; Rabe and Staats, 2022; Dong et al., 2024) significantly reduce the memory overhead of attention mechanisms while accelerating computation. Advances in layer normalization (Xiong et al., 2020), optimization techniques (Loshchilov and Hutter, 2017; Rasley et al., 2020a), and data scaling strategies (Hoffmann et al., 2022) have revolutionized the training of large-scale models in other dimensions. These innovations have allowed models with billions of parameters to be trained efficiently (Achiam et al., 2023; Dubey et al., 2024), unlocking new levels of performance in NLP tasks. However, these advancements are mostly specialized for autoregressive LLMs, with little exploration of their applicability to insertion-based models.

In this work, we address these challenges by integrating state-of-the-art practices to forge the insertion-based language model InsNeXt, a modern architecture capable of training-time scaling on par with traditional autoregressive models. By incorporating techniques such as FlexAttention (Dong et al., 2024), improved positional encodings, and optimized training pipelines, we achieve substantial improvements in computational efficiency and scalability.

We pretrain InsNeXt with two major configurations and a few ablative ones, ranging from 154M to 587M parameters, supporting a maximum context window of 4096 tokens. The training is performed under a two-stage fashion: sentence-level pretraining on a BERT (Devlin et al., 2019)-style Wikipedia+books dataset and document-level pretraining on the SlimPajama (Soboleva et al., 2023) dataset. As one of the foundation works, we study on a huge variety of alternative designs in different aspects insertion-based models, revealing arguably the best practices of which. We also propose an improved re-contextualization mechanism for the insertion-based decoding to better utilize the models' bidirectional context reception. The resulting models are evaluated on a broad range of tasks, demonstrating their effectiveness both as bidirectional BERT-style representation learners for natural language understanding (NLU) and as insertion-based decoders for generative and likelihood prediction tasks. We believe these qualities distinguish InsNeXt models from autoregressive counterparts and highlighting their potential to redefine the landscape of language modeling.

2 Methodology

2.1 Revisiting InsNet with a Contrastive Study against the (Large-Scale) Autoregressive Models

InsNet (Lu et al., 2022) is one of the first works that focus on tackling the training-efficiency issue of insertion-based language models. It addresses the efficiency issues in a practical training of insertion-based models in two folds: during the *context encoding* phase and the *action prediction* phase.

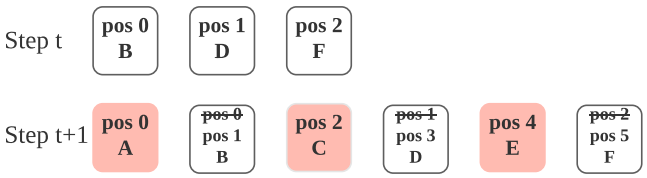


Figure 1: Illustration of the volatile position problem by Lu et al. (2022).

Context Encoding: the Volatile Position Problem and the Solution The high parallelizability and efficiency of context encoding in autoregressive transformer-based decoder-only LLMs (Vaswani et al., 2017; Radford et al., 2019) are widely considered some of the most important factors towards their success. Compared to these models, the vanilla insertion-based model Insertion Transformer (InsT) (Stern et al., 2019) falls short. Lu et al. (2022) argues that the biggest training-time performance issue for InsT comes from the position encoding inefficiency, namely the *volatile position* problem. We include their original illustration of the problem as in Figure 1. InsT relies on absolute position embeddings[1] that bind each token's representation to its index in the *partial* sequence. When a new token is inserted, the indices of the existing tokens shift, so their previously cached position embeddings become invalid. To restore consistent key/query vectors for attention, the compute-heavy

[1]Alternatively, relative position encodings with static distance, like in XLNet (Yang et al., 2019) and Roformer (Su et al., 2024)

Transformer models need to be run over the entire updated context at every step, incurring a full re-encoding pass per insertion and pushing training complexity to at most $O(n)$ per sequence. InsNet improves insertion-based text generation by introducing an insertion-oriented, relative positional encoding called the *offset* matrix. Offset matrices only concern the *relative* spatial interactions between tokens that are *already* inserted at each step. When a new token is inserted, all previously computed position encodings remain unchanged, and the incoming token reports its pairwise distance to the existing tokens into the new *offset* matrix; this lets the model reuse a single context encoding for an entire sequence, cutting training-time re-encoding from $O(n)$ to $O(1)$ and preserving full distance information. In addition, the authors develop a fast *offset-compression algorithm* that builds the offset matrix from permutation indices with simple masking and in-row ranking operations, avoiding the naive $O(n^2)$ construction cost in the sequential execution. Each transformer layer then uses sinusoidal embeddings to concern the offset values in the attention score, yielding insertion-oriented distance-aware representations without extra passes.

Action Prediction: Next-Token Prediction (NTP) vs. Next-Insertion Prediction (NiTP In autoregressive decoding, the prediction task is simple: predict the single *next token* (NTP) that will be appended to the current prefix. However, in insertion-based generation, the model must decide at least *both* **where** to insert and **what** to insert. The position choice is scalable and intuitive, because it can be cast as a single-head attention over the sequence, thus benefiting from efficient attention kernels with block-wise reduction (Rabe and Staats, 2022; Dao et al., 2022; Dao, 2023; Shah et al., 2024) for scalability. However, predicting the *next-inserted token* (NiTP) is harder. In an autoregressive decoder-only Transformer, given a training sequence, the mapping from a partial prefix to its next token is deterministic, so no extra aggregation is needed beyond the final layer. By contrast, in insertion-based generation, the representation for NiTP changes with every candidate slot, even under the same prefix. InsNet follows InsT and formalizes each candidate position as an *insertion slot*. It then builds the *slot representation* embeddings with a lightweight sparse attention over last-layer hidden vectors: the **left** and **right** neighbors, and the **most recently** inserted token.

2.2 InsNeXt: Scalable Insertion-based Language Model with Advanced Transformer Designs

InsNet inherits a lot of elements from XLNet (Yang et al., 2019), a BERT-era model with legacy designs, many of which are later proven to be suboptimal in larger scale pretraining (Brown et al., 2020; Touvron et al., 2023a). We hereby discuss both the recent advances for decoder-only transformers that we adopt to scale up the proposed model InsNeXt, and new model designs that we craft with originality to facilitate the scalability of InsNeXt.

General Design: the Major Model and the Ablated Variants Since the scaling law and best practice of such scalable insertion-based language models remains hugely underexplored, we conduct a fairly broad range of model design ablation in each of the aforementioned aspects. However, it is too computationally expensive to exhaust all of the possible combinations of designs. Thus, we first select a basic, safe combination of model designs to build a basic model for the ablation study, then combine all best practices we found in the model design study to build the final major models we deliver. The basic model is a 12-layer, 12-headed transformer with InsNet-style sinusoidal relative position encoding. It has a dimensionality of 768, and GeLU pre-LN FFN layers with intermediate size of 3072. It uses *untied* input and output embeddings. The total parameter count is 171M. It uses shallow aggregation for NiTP.

Provided our computation limitations, we present the two major setups in scaling up insertion-based models: the *base*-sized 154M model and the *advanced*-sized 573M model. InsNeXt-base is a 16-layer, 12-headed transformer model with insertion-oriented ALiBi as the position encoding. It has a hidden size of 768 and SwiGLU FFN layers with intermediate size of 2048. It uses *tied* input and output embeddings. InsNeXt-advanced shares most architectural designs with the base-sized model with only size expansion. It has 32 layers of 18-headed attention, and a hidden size of 1152. It uses SwiGLU FFN layers with intermediate size of 3072. Both models use deep aggregation for NiTP.

Residual Connection and Layer Normalization In contrast to InsNet's legacy PostLN block, we conduct an ablation study over a few more recently proposed normalization alternatives, including pre-normalization (pre-LN) (Xiong et al., 2020), two-hop pre-LN proposed in MEGALODON (Ma et al., 2024) and Deepnorm (Wang et al., 2024). Results show that most recently published normalization blocks yield observably better stability and scalability than the legacy post-LN block, which is consistent with their reported performance in autoregressive models and encoder models. For an

ablative and contrastive study on the effect of warmup iteration numbers under each normalization choice, please refer to the appendix A.4.

Position Encoding Scalable Insertion-based language models are inherently incompatible with absolute position encoding and relative position encoding with static distance assumptions. This is a direct result of the volatile position problem in insertion-based generation. Thus, common relative position encodings, *e.g.* the Rotary Position Encoding (RoPE) (Su et al., 2024) and T5-bias (Raffel et al., 2020), are not directly applicable without modification. We explore other alternatives, including the original InsNet sinusoidal relative position encoding and ALiBi (Press et al., 2021), as they both directly model the *interaction* of different positions and thus can be altered to reflect the insertion-oriented position layout. For more information on the attempts at modifying and efficiently implementing the two position encodings, please refer to the appendix A.2.

Slot Aggregation: Deep Aggregation using Two-stream Attention In our early attempt to scale up the model, we find that while the shallow aggregation proposed in InsNet is efficient and capable enough for smaller models on small datasets, it is no longer the best practice when both the training data and model sizes increase. A notable observation is that during large-scale pretraining, the model should be aware of the location of insertion after permutation. To better utilize the potential of increasing data and model capability during scaling-up, we follow the practice in XLNet and adopt the two-stream attention mechanism to *deep aggregate* the slot representation from layers of the context encoding. A detailed illustration of the insertion-oriented two-stream attention in both training and decoding can be found in the appendix Figure 3.

Position Prediction: Soft-capped Position Logits In actual language usage, when humans recursively refine or expand a sentence, it is possible that there are multiple correct slots for the next step of expansion. The original position-predicting attention in InsNet does not reflect this. In InsNeXt, we introduce a soft-cap mechanism to the position logits to encourage the model to learn a uniform distribution over all feasible next-insertion slots. Given the soft-caps $K > 0$, the modified position logit $-K < a^K_{pos} < K$ is computed by $a^K_{pos} = K \cdot \tanh(\frac{a_{pos}}{K})$.

2.3 Training Details

2.3.1 Dataset Preparation

Due to the intractable nature of exhaustive enumeration of all permutations for longer sequences, we conduct the training of InsNeXt models in a two-stage fashion: sentence-level training on a BERT-style dataset and document-level training on the first 60B of the SlimPajama dataset. A study in the earlier stage of our attempt shows that this is beneficial for the model to converge faster, compared to directly training on the document-level data.

Sentence-level Data The sentence-level data is crafted from a mixture of Wikipedia-English-2023, the Gutenberg Project dataset (PGLAF, 1971) and the BookCorpus dataset (Zhu et al., 2015). Only natural sentences that start with alphabetical characters and end with terminating punctuation are selected. Two consecutive sentences that appear in the same document are concatenated into a single training sequence for the model to learn the basic concept of *moving on* to the next sentence when one sentence is finished. The maximum sequence length is set to be 128, and models are trained to predict the likelihood of at most only the first 96 insertion operations to avoid overfitting to only complete sentence pairs.

Document-level Data The document-level data is crafted from the SlimPajama dataset (Soboleva et al., 2023). The original SlimPajama dataset consists of 627B tokens, of which we take the first 60B tokens to facilitate our training process. Due to the limit of computational resources, in the first 95% batches of training, each sequence has a token limit of 1024, and we use only the last 5% for context length expansion to at most 4096.

2.3.2 Tokenization and Permutation of Insertion Operations

Tokenizer Following the setup of prior autoregressive LMs like Pythia (Biderman et al., 2023), OlMo (Groeneveld et al., 2024; OLMo et al., 2024) and ModernBERT (Warner et al., 2024), we use the GPTNeoXTokenizer with a vocabulary size of 50254. The only notable modification is that we force the tokenizer to split each digit in numbers.

Permutation of Insertion Operations To ensure the integrity of each natural word, the tokens within the same natural word are always grouped together and generated/predicted autoregressively in the permutation. We argue that it is mostly only reasonable to assume the recursive structure and compositionality in the same natural sentence. Thus, for inter-word permutations, all permutations are limited to within natural sentences chunked by the SpaCy (Honnibal et al., 2017) sentencizer. We acknowledge that this is an potentially problematic implementation, especially for those non-text, Markdown/HTML script data in SlimPajama dataset. We leave the study of a more principled, domain-agnostic permutation algorithm for future work.

Interleaved PrefixLM Masking We adopt a partially prefix-LM (Raffel et al., 2020) paradigm of training to encourage the model to learn a representation that captures bidirectional information and can be obtained even with attention mask removed. In sentence-level training, we have a 50% chance to remove part of the attention mask for a uniformly random proportion of the first few insertion operations. In document-level training, the proportion of training sequences with this prefix-LM masking is reduced to 10% to avoid sparsity in token prediction and computation utilization. In practice, this helps improving InsNeXt robustness as a representation learner, and even brings us the possibility to resolve the distribution shift issue during decoding of insertion-based models. We will discuss the this special masking with more details and illustrations in the appendix A.3.

2.3.3 Optimization Setup

Batch Size and Distributed Training In both stages of training, we use a global batch size of 1M tokens for all configurations of the model unless otherwise stated. For more information, please refer to our appendix in A.3.

Optimizer and Learning Rate Schedule We use the AdamW (Loshchilov and Hutter, 2017) optimizer with gradient norm clipping of 1.0 and beta values of (0.9, 0.9). During sentence-level training, following the practice of ModernBERT (Warner et al., 2024), we use a trapezoidal learning rate scheduler with 10000 warmup steps and a peak learning rate of 4e-4. After 110000 further iteration steps of constant LR training, we cooldown the model in 60000 iteration steps by cosine-decaying the learning rate to 1e-4. For document-level training, we linearly warmup the model within the first 1000 steps and then fine-pretrain the model with a constant learning rate of 1e-4.

Initialization and Other Important Details For all experiments, we choose the soft position logit cap $K = 3$. Following the practice of many recently published LLMs, we initialize all parameters from a normal distribution with stddev $= \sqrt{\frac{2}{5D}}$ (D is the hidden size), unless otherwise stated.

2.4 Decoding

We hereby briefly explain the details on how to decode from a pretrained InsNeXt model.

Position Selection Unlike autoregressive models that only need to perform next-token prediction, the proposed scalable insertion-based model needs to first determine the decoding position and then aggregate the slot representations in each step of the decoding. The selection of the position can be performed deterministically by taking the argmax among all position logits, or stochastically by sampling from the softmax-ed distribution $p_{pos} \propto \exp(a^K_{pos})$ over all positions. Note that one can always choose multiple ones of the top-N ($N > 1$) positions distribution simultaneously and attempt to decode in parallel as in InsNet (Lu et al., 2022). We leave the application of this feature for future work and only focus on sequential (one token at a time) decoding in this paper.

Slot Aggregation for NiTP After the next position prediction, we generate the offset matrix and apply the two-stream attention to compute the representation for NiTP. Note that in actual deployment environment, this step can be significantly accelerated by caching KVs of the previous context encoding steps as in decoder transformers. We then take the deep-aggregated slot representation and project it using the transposed word embedding matrix to generate a distribution over the vocabulary. Any decoding algorithms or logit modifiers that work for autoregressive models can then be applied here without much adaptation.

Efficient Bidirectional Re-contextualization The original way of using the insertion-oriented position encoding and upper-triangular masking to efficiently model the spatial relation in insertion-based generation has caused two major issues. The first one is a performance issue similar to the *exposure bias* in autoregressive models. In a uni-directionally encoded insertion-based generation process, the spatial relation depicted by the interaction of the first few tokens can be inaccurate

especially if we consider longer bidirectional context. This causes the model's internal error to accumulate over time, and eventually lead to degenerated performance on the long run.

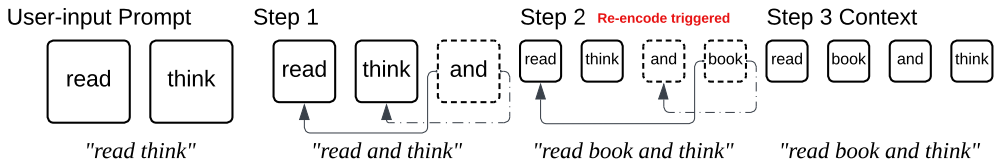


Figure 2: Illustration of when the bidirectional re-encoding happens in the proposed Efficient Bidirectional Re-contextualization. Solid line token blocks are ones encoded with a bidirectional attention; dashed line ones are encoded using the unidirectional efficient offset matrix that does not concern future insertions.

The second issue is about a proper permutation assumption for a user-input prompt. The permutation of insertion operations has an underlying effect on how the model understands the structuring of the generated contents. For model-generated contents, we always first predict the position and then the token, so the permutation is naturally present. However, this is not the case for user-input prompts. In previous insertion-based models without bidirectional encoders like InDIGO (Gu et al., 2019) and InsNet (Lu et al., 2022), a common practice is to simply assume an autoregressive permutation or randomly generate the permutation. However, this either causes train/test discrepancies or injects additional stochasticity to the context encoding, which eventually harms the reliability and controllability of the model. One essential solution is to switch to fully bidirectional context encoding as in InsT (Stern et al., 2019) and some recently published diffusion language models . However, this is inefficient, as fully re-encoding upon any context update triggered by new insertions is extremely compute-expensive.

Inspired by C++ STL's implementation of the vector container (see analysis in Cormen et al. (2022)), which allows dynamic reallocation of memory for a growing array that supports random access and appending, we propose an efficient bidirectional re-contextualization mechanism that solves the two issues of the uni-directional context encoding at once. The idea is surprisingly simple - for user-assigned input or the initial empty string context, we simply encode the sequence bidirectionally by removing the lower-triangular attention mask. We continue to generate the next few tokens with unidirectionally encoded context updates while retaining the bidirectionally encoded part in the prefixLM paradigm (Raffel et al., 2020), until the generated tokens since last re-contextualization surpass the length of the bidirectionally encoded section. Then, we discard all KV cache so far (if any) and an expensive yet spatially unbiased fully bidirectional context re-encoding will be triggered. It is expected that: 1) at least half of the context is bidirectionally encoded, so that the spatial relation won't be significantly and irreversibly corrupted by insertion operations, 2) the expensive fully-bidirectional re-encoding is not triggered very frequently. In fact, one can easily prove that the expected computational overhead of a decoding process with the proposed re-contextualization is at the same scale of the vanilla uni-directional one with only a marginal extra cost.

3 Experiments

We conduct our experiments in three major parts:

- **Natural Language Understanding Tasks** experiments aim to test the basic generalizability and quality of the learned representation. We also reuse this experimental setup in our ablative model design exploration to find the model design best practices.

- **Commonsense Reasoning Tasks** experiments aim to test whether the pretrained model is able to capture the world model knowledge in its parameters, and induce a stronger preference on the commonsensical continuation over the other ones;

- **Controllable Generation Tasks** experiments are conducted to show the unique controllability hegemony of insertion-based language models over traditional autoregressive models.

3.1 Experiment 1 - Representation Learning Study using Discriminative Natural Language Understanding Tasks

Following previous works on representation learning, we focus on two subtasks SST-2 (Socher et al., 2013) and MNLI (Williams et al., 2018) from the GLUE (Wang et al., 2018) leaderboard. We argue that these two tasks reflect the pretrained model's learned representation's generalizability for single sentence (represented by SST-2) and multi-sentence (represented by MNLI) scenarios well.

We first explore the best scalable designs for insertion-based language models. We start from a *basic* model that reproduces most architectural designs as InsNet, then replace different components of the model and train the altered variants. Due to the limit of computational resources, we only train the basic model and the altered variants on the sentence-level data for 120000 iterations without the trapezoidal cooldown. For more details, please refer to the appendix.

We report the evaluation on the development split of SST-2 and matched version of MNLI. We compose the best practices of each module and train the major models evaluated in all following experiments. A more comprehensive study of the major models' performance on the GLUE leaderboard, and a detailed ablation of different model designs can be found in the appendix.

Table 1: Representation learning study in comparison with the baseline models. The best performance of each category/group is marked with **bold** font and the notable second place winner is marked with underline. We consider both the accuracy on the downstream tasks and training efficiency for architecture selection. All results are reported as the average of models with 3 different random seeds.

Model Variant	#Params	SST-2	MNLI-m
Baselines - Generative			
GPT-2-base	124M	91.85%	81.23%
GPT-2-medium	355M	92.09%	85.23%
Pythia-160m	123M	89.30%	78.96%
Pythia-410m	354M	91.55%	83.03%
Pythia-1b	908M	91.66%	83.85%
T5-Small	61M	90.44%	82.07%
T5-Base	223M	92.54%	**85.30%**
Qwen2.5-0.5b	494M	**94.26%**	84.65%
Baselines - Discriminative			
BERT-base	108M	92.27%	84.14%
BERT-large	334M	93.73%	85.66%
RoBERTa-base	125M	94.26%	87.43%
RoBERTa-large	355M	**95.94%**	**90.26%**
InsNet (reproduced)	171M	91.85%	82.20%
InsNeXt-base (Ours)	154M	93.00%	83.23%
InsNeXt-advanced (Ours)	573M	94.15%	85.94%

Discussion Results show that the proposed InsNeXt models are solid representation learners, especially compared to peer generative models, while all of which still fall short against SOTA encoding-oriented models like RoBERTa, even with doubled size like TinyLLaMa (Zhang et al., 2024).

3.2 Experiment 2 - Zero-shot Commonsense Reasoning Tasks

Following the practice of TinyLLaMa (Zhang et al., 2024), we conduct a commonsense reasoning study of our model against a fair range of popular small-to-medium-sized language models in the community. We choose the HellaSwag (Zellers et al., 2019), Obqa (Mihaylov et al., 2018) and WinoGrande (Sakaguchi et al., 2019) datasets as our testbed. The models' choice are selected using length-normalized likelihood scores, following prior practices. Note that the commonsense reasoning tasks here are likelihood-predictive ones rather than generative ones. We will discuss the generative commonsense reasoning task CommonGen in the next section.

Table 2: Commonsense Reasoning Evaluation.

Model	#Params	HellaSwag	Obqa	WinoGrande
Random Selection	-	25.00%	25.00%	50.00%
GPT-2-base	124M	31.14%	27.20%	51.62%
GPT-2-medium	355M	39.38%	30.20%	53.20%
Pythia-160m	123M	30.17%	27.00%	51.30%
Pythia-410m	354M	40.52%	29.40%	53.04%
Pythia-1b	908M	47.10%	31.40%	53.43%
Qwen2.5-0.5b	494M	**52.17%**	**35.20%**	**56.59%**
InsNet (reproduced)	171M	27.39%	24.40%	50.71%
InsNeXt-base (Ours)	154M	33.47%	30.20%	52.37%
InsNeXt-advanced (Ours)	573M	53.63%	34.80%	55.87%

Discussion Results show that the proposed InsNeXt models are trained well enough to compress the world knowledge in the training data into its parameters, as a result having similar commonsense reasoning capabilities to similar-sized autoregressive models.

3.3 Experiment 3 - Controllable Generation Tasks

We experiment with the lexically-controlled generation tasks following the setup of InsNet to measure the performance of the proposed InsNeXt model. We examine the evaluated models on the Yelp 160K, WMT News and CommonGen datasets. Note that insertion-based models are not directly able to handle fuzzy lexical constraints that allow reflections. InsNet reports performance using the original form of the concept words without considering reflections. In this work, we train the models as insertion-based decoders that take in full keyword sequences as input, first fulfill the lexical constraints with correct reflections and then generate the rest of the context. This significantly improves their coverage of keywords, at the cost of slightly imperfect keyword coverage rates compared to some recent search-based and HMM-based methods like GeLaTo (Zhang et al., 2023).

Table 3: Controllable Generation Evaluation. Both BLEU-4 and lexical constraint coverage (shown with BLEU-4↑/Coverage↑) are reported. Note that the Yelp and News are high-variance datasets, so it's natural that all models have rather lower BLEU-4 score compared to that in CommonGen.

Model	#Params	Yelp 160K	News	CommonGen-dev
GPT-2-base	124M	5.83/77.50%	6.54/63.30%	22.90/65.4%
GPT-2-medium	355M	6.99/87.70%	7.56/80.60%	24.65/83.2%
Pythia-160m	123M	5.83/52.80%	5.50/47.00%	20.97/67.4%
Pythia-410m	354M	7.29/85.10%	6.64/72.00%	22.29/86.2%
Qwen2.5-0.5b	494M	7.46/90.30%	7.42/92.40%	23.37/92.0%
Insertion Transformer (BERT init+POINTER)	357M	3.79/100%	3.04/100%	16.70/97.9%
InsNet (Original Report)	171M	5.78/100%	4.96/100%	18.71/100%
InsNet (reproduced, w/ keyword-first perm.)	171M	5.63/100%	4.87/100%	21.36/98.3%
InsNeXt-base	154M	6.73/100%	6.60/100%	23.46/97.9%
InsNeXt-advanced	573M	**8.13/100%**	**8.01/100%**	**25.73/98.7%**

Discussion Results show that the proposed InsNeXt models are more controllable and capable language generators than its autoregressive counterparts and prior insertion-based models, in term of both constraint coverage (near-perfect to perfect) and overall generation quality.

4 Related Works

Insertion-based Language Models Insertion-based language models offer a flexible alternative by constructing sequences through token insertions, in contrast to the conventional left-to-right continuation in traditional autoregressive LMs. The Insertion Transformer (Stern et al., 2019) is an iterative, partially autoregressive model that generates sequences based on insertion operations.

Building upon this, POINTER (Zhang et al., 2020) was developed for hard-constrained text generation, progressively inserting tokens in a parallel manner to complete sequences efficiently. To enhance training efficiency and decoding flexibility, InsNet (Lu et al., 2022) introduced an insertion-oriented position encoding and a lightweight slot representation strategy, enabling both parallel and sequential decoding. Further advancements include the design of Fractional Positional Encoding (FPE) for Insertion Transformers (Zhang et al., 2021), allowing the reuse of representations from previous steps, yet introducing extra discrepancy between training and decoding.

Relative Position Encodings Recent advancements in transformer models have led to various methods for encoding positional information. Transformer-XL (Dai et al., 2019) and XLNet (Yang et al., 2019) introduced relative positional embeddings to better capture dependencies across long sequences. T5 (Raffel et al., 2020) employs learnable relative position biases, enhancing its adaptability to different sequence lengths. ALiBi (Attention with Linear Biases) (Press et al., 2021) adds linear biases directly to attention scores, facilitating the processing of longer sequences. Rotary Position Embedding (RoPE) (Su et al., 2024) integrates positional information through rotation matrices applied to token embeddings. Most of these techniques are tailored for fixed-order sequence modeling and may not be directly applicable to insertion-based generation. Notably, ALiBi and T5's relative position biases are exceptions; their designs involve rectifiers as a function of *only* the offset between positions, making them potential for insertion-oriented scenarios. However, T5 bias faces scalability issues as no mainstream efficient kernels support training with it yet.

Efficient Scaled Dot Product Attention In recent years, the development of memory and compute-efficient attention mechanisms has seen notable advancements. Initially, Rabe and Staats (2022) proposed a method to reduce memory requirements from quadratic to linear scale by partitioning attention computations into smaller blocks, fitting within a GPU's on-chip memory. Building upon this, the FlashAttention algorithm (Dao et al., 2022) further optimized efficiency by managing data movement and computation to minimize memory usage and increase computational speed. FlashAttention-2/3 (Dao, 2023; Shah et al., 2024) continue to incorporate hardware-related optimizations. More recently, FlexAttention (Dong et al., 2024) has been developed to combine the flexibility of reprogrammable attention score modifiers with the performance benefits of FlashAttention, allowing researchers to experiment with various attention variants efficiently.

5 Conclusion and Future Work

We present InsNeXt, a modern-level scalable insertion-based language model integrating recent advances in LLMs. We explore and discuss model design alternatives of each component for insertion-based generation, the basic scaling pattern of the model, as well as the best training strategies from scratch for them. We then follow the best practices from the architecture study to train two major specifications of the model, one (InsNeXt-*base*) with 154M parameters and another (InsNeXt-*advanced*) with 573M parameters. We evaluate the fully renovated insertion-based models under three major scenarios: 1) representation learning capabilities using NLU tasks; 2) commonsense reasoning capabilities using HellaSwag, WinoGrande and Obqa; and 3) controllable generation on Yelp, news and CommonGen datasets. Results show that the InsNeXt models are of competitive performance with state-of-the-art autoregressive language models, while preserving their unique advantages especially in terms of controllability.

We are aware of the recent progress in discrete-space diffusion language models with autoregressive LLM warmups. These models have the potential to provide parallelizability of generation as well as heterogeneous allocation and assignment of computational power for tokens with different level of predictability. Existing Diffusion LLMs are mostly based on the MLM objective, meaning it has to pre-determine the number of placeholder tokens and then learn to fulfill them. We argue that this is a flawed solution, and InsNeXt can be viewed as the prototype model towards a more flexible, powerful family of diffusion language models as the foundation for the next generation of artificial intelligence.

References

Marah Abdin, Mojan Javaheripi, Sébastien Bubeck, Allie Del Giorno, Ronen Eldan, Suriya Gunasekar, Yin Tat Lee, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Dynamic tables*, 4 edition, chapter 17. MIT Press, Cambridge, MA. Section 17.4.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *arXiv preprint arXiv:2205.14135*. Spotlight paper at ICLR 2023.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, and Bingxuan Wang. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2406.12345*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2024. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*.

A. Dubey, A. Goyal, A. Dey, A. Khandelwal, A. Narang, A. Rodriguez, A. Fan, B. Fuller, C. Gao, D. Bikel, D. Esiobu, E. Hambro, E. Smith, F. Azhar, G. Izacard, G. Lample, H. Inan, H. Touvron, I. Kloumann, J. Kuan, J. Lee, J. Reizenstein, K. Saladi, K. Stone, L. Martin, M. Kambadur, M. Lachaux, M. Khabsa, N. Goyal, N. Bashlykov, O. Molybog, P. Albert, P. Xu, P. Mishra, R. Silva, R. Stojnic, R. Hou, R. Rungta, S. Batra, S. Hosseini, S. Edunov, S. Bhosale, T. Lavril, T. Scialom, T. Mihaylov, V. Goswami, W. Fu, X. Martinet, X. Tan, Y. Babaei, Y. Lu, Y. Mao, Y. Nie, Y. Zhang, and Z. Yan. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, et al. 2024. Olmo: Accelerating the science of language models. *arXiv preprint arXiv:2402.00838*.

Jiatao Gu, Qi Liu, and Kyunghyun Cho. 2019. Insertion-based decoding with automatically inferred generation order. *Transactions of the Association for Computational Linguistics*, 7:661–676.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2017. spacy – industrial-strength natural language processing in python.

Mojan Javaheripi, Sébastien Bubeck, Marah Abdin, Jyoti Aneja, Caio César Teodoro Mendes, Weizhu Chen, Allie Del Giorno, Ronen Eldan, Sivakanth Gopi, et al. 2023. Phi-2: The surprising power of small language models. *Microsoft Research Blog*.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023a. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023b. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*. Poster presentation at ICLR 2019.

Sidi Lu, Tao Meng, and Nanyun Peng. 2022. Insnet: An efficient, flexible, and performant insertion-based text generation model. *Advances in Neural Information Processing Systems*, 35:7011–7023.

Xuezhe Ma, Xiaomeng Yang, Wenhan Xiong, Beidi Chen, Lili Yu, Hao Zhang, Jonathan May, Luke Zettlemoyer, Omer Levy, and Chunting Zhou. 2024. Megalodon: Efficient llm pretraining and inference with unlimited context length. *Advances in Neural Information Processing Systems*, 37:71831–71854.

Meta. 2025. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. `https://ai.meta.com/blog/llama-4-multimodal-intelligence/`. Accessed: 2025-05-10.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? A new dataset for open book question answering. *CoRR*, abs/1809.02789.

Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, et al. 2024. 2 olmo 2 furious. *arXiv preprint arXiv:2501.00656*.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. `https://openai.com/blog/chatgpt`. Initial release of ChatGPT based on GPT-3.5, November 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, pages 8024–8035.

PGLAF. 1971. Project gutenberg. https://www.gutenberg.org/. Accessed: 2025-05-14.

Ofir Press, Noah A Smith, and Mike Lewis. 2021. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*.

Markus N. Rabe and Charles Staats. 2022. Self-attention does not need $o(n^2)$ memory. *arXiv preprint arXiv:2112.05682*.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020a. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020b. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD '20)*, pages 3505–3506. ACM.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. WINOGRANDE: an adversarial winograd schema challenge at scale. *CoRR*, abs/1907.10641.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*.

Noam Shazeer. 2020. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*.

Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey. 2023. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama. https://cerebras.ai/blog/slimpajama-a-627b-token-cleaned-and-deduplicated-version-of-redpajama.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA. Association for Computational Linguistics.

Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. 2019. Insertion transformer: Flexible sequence generation via insertion operations. In *International Conference on Machine Learning*, pages 5976–5985. PMLR.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann,

Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium. Association for Computational Linguistics.

Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Dongdong Zhang, and Furu Wei. 2024. Deepnet: Scaling transformers to 1,000 layers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, et al. 2024. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. *arXiv preprint arXiv:2412.13663*.

Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122, New Orleans, Louisiana. Association for Computational Linguistics.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. 2020. On layer normalization in the transformer architecture. In *International conference on machine learning*, pages 10524–10533. PMLR.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2024b. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, Florence, Italy. Association for Computational Linguistics.

Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. 2023. Tractable control for autoregressive language generation. In *International Conference on Machine Learning*, pages 40932–40945. PMLR.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385*.

Yizhe Zhang, Guoyin Wang, Chunyuan Li, Zhe Gan, Chris Brockett, and Bill Dolan. 2020. Pointer: Constrained progressive text generation via insertion-based generative pre-training. *arXiv preprint arXiv:2005.00558*.

Zhisong Zhang, Yizhe Zhang, and Bill Dolan. 2021. Towards more efficient insertion transformer with fractional positional encoding. *arXiv preprint arXiv:2112.06295*.

Yukun Zhu, Ryan Kiros, Richard S. Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 19–27. IEEE Computer Society.

A Supplementary Details of Model and Algorithm Designs

A.1 Additional Model Design Information

Activation In addition to the legacy GeLU (Hendrycks and Gimpel, 2016) feed-forward layers, we also consider SwiGLU, which is a variant of the Gated-Linear Units (Shazeer, 2020) that reportedly provides better parameter efficiency in more recently published LLMs. Note that, to facilitate training efficiency and provide a fair comparison, we intentionally choose model dimensionality divisible by 3. This way, all matrix/vector multiplication operations can be performed on tensors with dimensionality of multiples of 128/64 for best TensorCore practices, without altering the total number of parameters compared to its GeLU counterparts.

A.2 Implementing Position Encoding with Efficient Attention Kernels

We hereby discuss the related attempts we make to implement the mentioned position encodings efficiently.

InsNet-style Sinusoidal Relative Position Encoding For the original InsNet sinusoidal relative position, no off-the-shelf efficient attention kernel is available at the moment when this paper is submitted, so we build our own memory-efficient PyTorch attention kernel with eager-mode code, and then transform it to the Triton implementation of the attention mechanism to include the block-wise reduction tricks that other efficient attention kernels provide. See code snippet 1.

Insertion-oriented ALiBi For insertion-oriented position encoding with ALiBi, we make a minor modification of it on the denominator of the bias rescaling factor $\alpha_i = \frac{1}{2^{k_i}}$. Instead of using uniformly interpolated exponentials from 1.0 to 8.0 in each attention head, as suggested by the original ALiBi paper, we simply choose k_i to be the 1-based attention head index. For example, an attention layer with 12 attention heads will have α to be $\left[\frac{1}{2^1}, \frac{1}{2^2}, \frac{1}{2^3}, ..., \frac{1}{2^{11}}, \frac{1}{2^{12}}\right]$, instead of $\left[\frac{1}{2^{(2/3)}}, \frac{1}{2^{(4/3)}}, \frac{1}{2^2}, ..., \frac{1}{2^{(22/3)}}, \frac{1}{2^8}\right]$ as in vanilla ALiBi. For larger models with more dimensions, this simply adds in extra attention heads that have slower bias-term decay when relative distance increases. This helps the larger model to utilize the additional capacity to handle longer-range dependencies or generate less position-sensitive perceptions without significantly changing its behavior in short-range dependencies. In short-range dependencies with attention heads that have larger bias decay denominators, it is approximately equivalent to omitting position information in some of the model dimensions, which is reportedly a common practice in recent position encodings like RoPE. During training, InsNeXt with insertion-oriented ALiBi is implemented with the FlexAttention (Dong et al., 2024) kernel, which is a Triton-implemented FlashAttention-2/3 alternative that facilitates scalable training of flexible attention modifier. However, up to the time of this submission, the inference-friendly version of FlexAttention is still under-optimized. During inference, we simply fall back to Memory Efficient Attention as our attention kernel choice.

A.3 Pretraining

Batch Size and Distributed Computation We train our model on a single NVIDIA H100x8 SXM node. Unfortunately, we encountered a hardware failure in the early stage of this project that the NVSwitch bridge between the first to the rest of the GPUs was compromised. As a result, we are only able to deploy a 7-way distributed training for the major experiments instead of an 8-way one. In sentence-level training, we use an equivalent batch size of 1536 per GPU along with the gradient accumulation trick, resulting in a global batch size of 10752 sequences and approximately 1M tokens. In the major part of document-level training, we use an equivalent batch size of 144 per GPU, resulting in a global batch size of 1008 sequences and approximately 1M tokens too. We reduce the batch size and learning rate accordingly for the context extension phase to keep the per-batch token number to be around 1M. We build our distributed training script using a composition of the DeepSpeed (Rasley et al., 2020b) and PyTorch (Paszke et al., 2019) libraries, adopting the ZeRO offload stage of 1 for the base-sized model and 2 for the advanced-sized model. We also use a mixed precision training, with the BFloat16 datatype for in-layer gradients, and TensorFloat32 for gradient reduction, accumulation and parameter storage.

Interleaved PrefixLM Masking We adopt a partially prefix-LM (Raffel et al., 2020) paradigm of training to encourage the model to learn a representation that captures bidirectional information and

can be obtained even with attention mask removed. Specifically, we remove part of the attention mask for a uniformly random proportion of the first few insertion operations, so that each token can attend to these unmasked tokens from both directions. See Figure 4.

A.4 Extensive Ablation Study and Model Design Exploration

In this section we report the experiments we conduct that are intended for ablation study and model design exploration purposes. We start from a reproduced InsNet architecture, replace some of the components, pretrain the resulting model on the aforementioned sentence-level data then test it on the representation learning capabilities as a reflection of model design soundness.

Position Encoding We first explore the trade-off between model capability and training scalability of different insertion-oriented position encoding. We additionally report the training throughput at a context length of 1024 tokens per sequence. See the results in 4. All available designs yield similar performance, with T5 Bias slightly outperforming the counterparts. However, as training with T5 bias is not yet supported by any efficient attention kernels, we choose **insertion-oriented ALiBi** as our major position encoding. We leave the implementation of a potentially more capable insertion-based model with T5 bias for future work.

Table 4: Position encoding study results.

Model Variant	#Params	SST-2	MNLI-m	#Tokens/s/Chip @1K Cxt.
InsNet (reproduced, @Iter. #120K)				
- Sinusoidal Position Embedding	171M	91.85%	82.20%	17384
- T5 Bias	165M	92.01%	82.73%	32331
- Insertion-oriented ALiBi	164M	91.74%	82.15%	**116144**

Residual Connection and Layer Normalization We hereby explore the stability and model capability differences impacted by the choice of different layer normalization and/or residual connection choices. We consider four major variants: the original post-LN, pre-LN, two-hop MEGALODON pre-LN and DeepNorm. For DeepNorm, we follow their guidance to change the initialization of each layer accordingly. In addition to the final performance report with 10K warmup steps, we also report the performance with less warmup steps, at 0, 100, 1000 respectively. Results are shown in Table 5.

Table 5: Residual connection and layer normalization study results.

Model Variant	SST-2	MNLI-m
InsNet (reproduced, @Iter. #120K)		
- post-LN		
w/ 0 Warmup	Divergence	Divergence
w/ 100 Warmup	89.33%	79.83%
w/ 1000 Warmup	90.14%	80.36%
w/ 10K Warmup	91.85%	82.20%
- pre-LN		
w/ 0 Warmup	90.36%	81.37%
w/ 100 Warmup	90.59%	82.25%
w/ 1000 Warmup	91.28%	81.99%
w/ 10K Warmup	91.74%	81.93%
- two-hop pre-LN		
w/ 0 Warmup	91.28%	81.76%
w/ 100 Warmup	91.51%	81.79%
w/ 1000 Warmup	91.63%	81.72%
w/ 10K Warmup	91.63%	81.88%
- DeepNorm		
w/ 0 Warmup	90.71%	80.62%
w/ 100 Warmup	90.48%	81.20%
w/ 1000 Warmup	91.51%	81.47%
w/ 10K Warmup	91.63%	81.39%

With 10K warmup steps, all variants yields similar performance, with the legacy post-LN slightly winning by a margin. However, if we adopt fewer warmup steps, the stability of post-LN is significantly compromised, while others are not very sensitive. Notably, we observe faster convergence with two-hop pre-LN with slightly degenerated final performance, and don't observe any advantages of DeepNorm against other alternatives. We acknowledge the possibility that these two layer normalization implementations are more specialized for even larger models, thus still worth trying if we further scale up the training. For our major configurations of the pretraining, we choose **pre-LN** as our layer normalization block choice.

Slot Aggregation: Deep Aggregation vs. Shallow Aggregation We concern both the representation learning and generative capabilities impacted by different slot aggregation methods. We train two variants of the *major* model InsNeXt-base under the same setup but with different aggregation method, and test their respective performances on both groups of our experiments. Results are shown as follows:

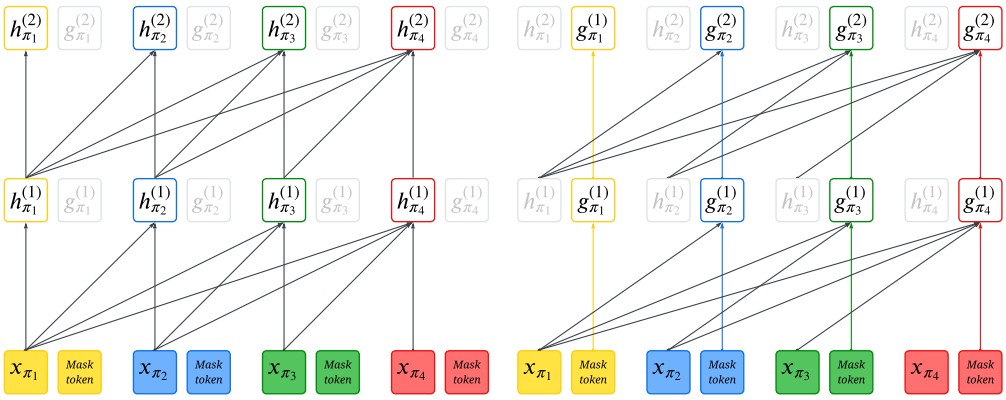
Table 6: Slot aggregation ablation study results

Model	SST-2	MNLI-m	Yelp 160K	News	CommonGen-dev
InsNet (Original Report)	-	-	5.78/100%	4.96/100%	18.71/100%
InsNet (reproduced)	91.85%	82.20%	5.63/100%	4.87/100%	21.36/98.3%
InsNeXt-base					
w/ shallow agg.	92.66%	83.07%	5.86/100%	5.13/100%	22.70/98.1%
w/ deep agg.	**93.00%**	**83.23%**	**6.73/100%**	**6.60/100%**	**23.46/97.9%**

In general, deep aggregation helps the model to better utilize model capability which consequently helps both representation learning and NiTP. We modestly argue that this will be more signficant for even larger models, but we don't have sufficient empirical results to support this claim. The only clear observation from the current results is that the usage of deep aggregation has a slightly more remarkable impact on generation/NiTP than representation learning.

A.5 Illustration of the Insertion-oriented Two-stream Attention and PrefixLM Masking

Derived from XLNet's illustration of the two-stream attention, we hereby illustrate how insertion-oriented two-stream attention works in InsNeXt. Assume we've generated a permutation π_i of all tokens x_i, the two streams of attention can be performed as shown in the following figures:



(a) Content Stream Attention (b) Query Stream Attention

Figure 3: Illustration of the two-stream attention. The content-stream attention aims to model the interaction of context tokens while the query-stream attention aims to aggregate information for NiTP.

Here, $g_{\pi_i}^{(j)}$ means the deep-aggregated representation for the i-th inserted token in the sampled permutation π from the j-th layer of the InsNeXt model.

We also illustrate the prefixLM masking adopted in our pretraining. Given a sequence of N tokens, we uniformly sample $M \sim \text{Uniform}(0, N)$ and remove this part of the lower-triangular attention mask, as illustrated in the following figure:

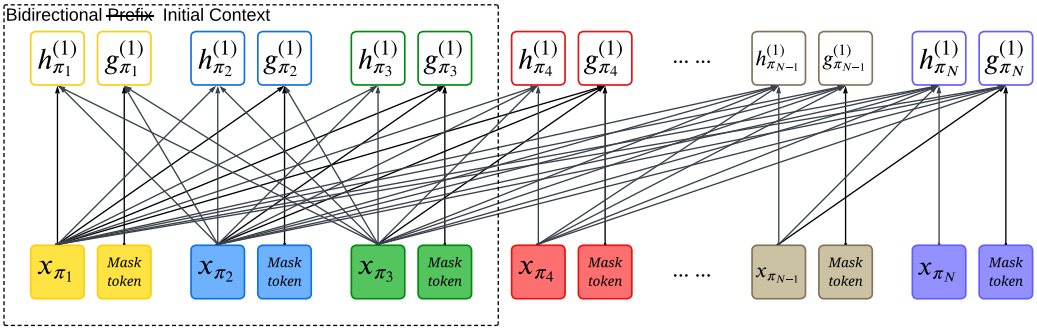


Figure 4: Illustration of the prefixLM style attention masking given a sequence with N tokens and a sampled bidirectional portion $M = 3$. Note that we stop using the notion *prefix* as in insertion-based generation, the bidirectional part is not necessarily the natural prefix, but simply the bidirectionally perceived partial context.

B Code Snippet of the Eager-mode Implementation of Sinusoidal Insertion-oriented Position Encoding

We hereby include a few of the important code snippets as more concrete description of the proposed algorithm. To facilitate the readability of code, we report the eager-mode version of the implementation. It can be transformed into an equivalent Triton implementation fairly easily.

Code 1: The eager mode implementation for the memory efficient attention with sinusoidal insertion-oriented position encoding.

```
def attn_core_logsumexp(query_chunk, key_chunk, value_chunk,
                        mask_chunk=None, offset_chunk=None,
                        query_r=None, key_r=None, offset_clip_range=128):
    attn_scores = torch.einsum('bnqd,bnkd->bnqk', query_chunk, key_chunk)
    attn_scores = attn_scores / math.sqrt(value_chunk.shape[-1])

    if mask_chunk is not None:
        attn_scores = attn_scores + mask_chunk.unsqueeze(1)
    if offset_chunk is not None:
        offset_chunk = offset_chunk.clamp(min=-offset_clip_range, max=offset_clip_range)
        indexed_offset = offset_chunk + offset_clip_range
        attn_scores_qbias = torch.einsum('ind,bnkd->bnik', query_r, key_chunk)
        attn_scores_kbias = torch.einsum('bnqd,jnd->bnqj', query_chunk, key_r)

        # Use the offset to index into the attn_scores_qbias
        # Expand indexed_offset dimensions to match attn_scores_qbias
        indexed_offset = indexed_offset.unsqueeze(1).expand(-1, query_r.size(1), -1, -1)

        # Gather the values using the computed offset index
        attn_scores_qbias = torch.gather(attn_scores_qbias, dim=2, index=indexed_offset) / math.sqrt(
            value_chunk.shape[-1])
        attn_scores_kbias = torch.gather(attn_scores_kbias, dim=3, index=indexed_offset) / math.sqrt(
            value_chunk.shape[-1])
        attn_scores = attn_scores + attn_scores_qbias + attn_scores_kbias

    # Compute logsumexp for numerical stability
    attn_scores = attn_scores.to(torch.float32)
    attn_weight = attn_scores.logsumexp(dim=-1) # bnq
    attn_distro = attn_scores.softmax(dim=-1)

    # Compute softmaxed attention weights without storing attn_scores
    # Compute weighted sum of values
    chunk_reduced_value = torch.einsum('bnkd,bnqk->bnqd', value_chunk, attn_distro.to(value_chunk.dtype
        ))

    return attn_weight, attn_distro, chunk_reduced_value

def forward_core(query, key, value, query_chunk_size, key_chunk_size,
                 mask=None, offset_matrix=None, query_r=None, key_r=None,
```

```
                offset_clip_range=128):
    reduced_values = []
    all_attn_weights = []
    mask_value = torch.finfo(query.dtype).min
    with (torch.no_grad()):
        for i in range(0, query.size(2), query_chunk_size):
            cumu_reduced_chunk = None
            cumu_attn_weight = None
            query_chunk = query[:, :, i:i + query_chunk_size, :]
            attn_weights_ = []
            if mask is not None:
                mask_qchunk = mask[:, i:i + query_chunk_size, :].to(query_chunk.device)
            else:
                mask_qchunk = None
            if offset_matrix is not None:
                offset_qchunk = offset_matrix[:, i:i + query_chunk_size, :].to(query_chunk.device)
            else:
                offset_qchunk = None

            for j in range(0, key.size(2), key_chunk_size):
                key_chunk = key[:, :, j:j + key_chunk_size, :]
                value_chunk = value[:, :, j:j + key_chunk_size, :]
                if mask is not None:
                    mask_chunk = mask_qchunk[:, :, j:j + key_chunk_size]
                else:
                    mask_chunk = None
                if offset_matrix is not None:
                    offset_chunk = offset_qchunk[:, :, j:j + key_chunk_size]
                else:
                    offset_chunk = None
                attn_weight, _, reduced_chunk = \
                attn_core_logsumexp(query_chunk, key_chunk, value_chunk,
                       mask_chunk=mask_chunk, offset_chunk=offset_chunk,
                       query_r=query_r, key_r=key_r,
                       offset_clip_range=offset_clip_range
                       )

                if cumu_reduced_chunk is None:
                    cumu_reduced_chunk = reduced_chunk.to(torch.float32)
                    cumu_attn_weight = attn_weight
                else:
                    cumu_attn_weight = torch.stack([cumu_attn_weight, attn_weight])
                    cumu_reduced_chunk = torch.stack([cumu_reduced_chunk, reduced_chunk])
                    cumu_reduced_chunk = torch.einsum(
                        "tbnqd,tbnq->bnqd",
                        cumu_reduced_chunk.to(torch.float32),
                        cumu_attn_weight.softmax(dim=0)
                    ).to(torch.float32)
                    cumu_attn_weight = cumu_attn_weight.logsumexp(dim=0)
                attn_weights_.append(attn_weight)
            reduced_values.append(cumu_reduced_chunk.to(query_chunk.dtype))
            all_attn_weights.append(cumu_attn_weight)
        reduced_values = torch.cat(reduced_values, dim=2)

        all_attn_weights = torch.cat(all_attn_weights, dim=2)

        return reduced_values, all_attn_weights

def backward_core(grad_output, query, key, value, all_attn_weights, chunk_size,
                  mask=None, offset_matrix=None, query_r=None, key_r=None,
                  offset_clip_range=128):
    grad_query = torch.zeros_like(query, dtype=torch.float32)
    grad_key = torch.zeros_like(key, dtype=torch.float32)
    grad_value = torch.zeros_like(value, dtype=torch.float32)
    grad_query_r = torch.zeros_like(query_r, dtype=torch.float32) if query_r is not None else None
    grad_key_r = torch.zeros_like(key_r, dtype=torch.float32) if key_r is not None else None
    scale = 1.0 / math.sqrt(value.shape[-1])

    with torch.no_grad():
        for i in range(0, query.size(2), chunk_size):
            query_chunk = query[:, :, i:i + chunk_size, :].contiguous()
            attn_weights_chunk = all_attn_weights[:, :, i:i + chunk_size].contiguous()
            grad_output_chunk = grad_output[:, :, i:i + chunk_size, :].contiguous()

            if mask is not None:
                mask_qchunk = mask[:, i:i + chunk_size, :].to(query.device)
            else:
                mask_qchunk = None
            if offset_matrix is not None:
                offset_qchunk = offset_matrix[:, i:i + chunk_size, :].to(query.device)
            else:
                offset_qchunk = None

            accumu_modifier = 0.

            for j in range(0, key.size(2), chunk_size):
                key_chunk = key[:, :, j:j + chunk_size, :]
                value_chunk = value[:, :, j:j + chunk_size, :]

                if mask is not None:
                    mask_chunk = mask_qchunk[:, :, j:j + chunk_size]
                else:
```

```
                    mask_chunk = None
                if offset_matrix is not None:
                    offset_chunk = offset_qchunk[:, :, j:j + chunk_size]
                else:
                    offset_chunk = None

                # Forward pass to recompute intermediates
                attn_weight, attn_distro, reduced_chunk = attn_core_logsumexp(
                    query_chunk, key_chunk, value_chunk,
                    mask_chunk=mask_chunk, offset_chunk=offset_chunk,
                    query_r=query_r, key_r=key_r,
                    offset_clip_range=offset_clip_range
                )

                adjustment = (attn_weight - attn_weights_chunk).exp().unsqueeze(dim=-1)
                attn_global = attn_distro * adjustment

                grad_attn_global = torch.einsum(
                    "bnqd,bnkd->bnqk", grad_output_chunk, value_chunk
                )
                grad_value_chunk = torch.einsum('bnqk,bnqd->bnkd',
                                                attn_global.to(grad_output_chunk.dtype),
                                                grad_output_chunk)
                grad_value[:, :, j:j + chunk_size, :] += grad_value_chunk.to(torch.float32)

                accumu_modifier -= torch.einsum("bnqk,bnqk->bnq",
                                                grad_attn_global.to(torch.float32),
                                                attn_global).unsqueeze(dim=-1)

            for j in range(0, key.size(2), chunk_size):
                key_chunk = key[:, :, j:j + chunk_size, :]
                value_chunk = value[:, :, j:j + chunk_size, :]

                if mask is not None:
                    mask_chunk = mask_qchunk[:, :, j:j + chunk_size]
                else:
                    mask_chunk = None
                if offset_matrix is not None:
                    offset_chunk = offset_qchunk[:, :, j:j + chunk_size]
                else:
                    offset_chunk = None

                # Forward pass to recompute intermediates
                attn_weight, attn_distro, reduced_chunk = attn_core_logsumexp(
                    query_chunk, key_chunk, value_chunk,
                    mask_chunk=mask_chunk, offset_chunk=offset_chunk,
                    query_r=query_r, key_r=key_r,
                    offset_clip_range=offset_clip_range
                )

                adjustment = (attn_weight - attn_weights_chunk).exp().unsqueeze(dim=-1)
                attn_global = attn_distro * adjustment

                grad_attn_global = torch.einsum(
                    "bnqd,bnkd->bnqk", grad_output_chunk, value_chunk
                )

                grad_attn_scores = attn_global * (
                    grad_attn_global + accumu_modifier
                )

                # Compute gradients w.r.t. query and key
                grad_attn_scores_qk = (grad_attn_scores * scale).to(query_chunk.dtype)

                grad_query_chunk = torch.einsum('bnqk,bnkd->bnqd', grad_attn_scores_qk, key_chunk)
                grad_key_chunk = torch.einsum('bnqk,bnqd->bnkd', grad_attn_scores_qk, query_chunk)


                # If query_r and key_r are used, compute their gradients
                if offset_chunk is not None and query_r is not None and key_r is not None:
                    offset_chunk = offset_chunk.clamp(min=-offset_clip_range, max=offset_clip_range)
                    indexed_offset = offset_chunk + offset_clip_range
                    indexed_offset = indexed_offset.unsqueeze(1).expand(-1, query_r.size(1), -1, -1)

                    # Gradients w.r.t. attn_scores_qbias and attn_scores_kbias
                    grad_attn_scores_qbias = grad_attn_scores_qk.to(torch.float32)
                    grad_attn_scores_kbias = grad_attn_scores_qk.to(torch.float32)

                    # Ungather gradients
                    grad_attn_scores_qbias_full = torch.zeros(
                        grad_attn_scores_qbias.size(0),
                        grad_attn_scores_qbias.size(1),
                        query_r.size(0),
                        grad_attn_scores_qbias.size(3),
                        device=grad_attn_scores_qbias.device,
                        dtype=torch.float32
                    ).scatter_add_(2, indexed_offset, grad_attn_scores_qbias)

                    grad_attn_scores_kbias_full = torch.zeros(
                        grad_attn_scores_kbias.size(0),
                        grad_attn_scores_kbias.size(1),
                        grad_attn_scores_kbias.size(2),
```

```
                        key_r.size(0),
                        device=grad_attn_scores_kbias.device,
                        dtype=torch.float32
                    ).scatter_add_(3, indexed_offset, grad_attn_scores_kbias)

                    # Compute gradients w.r.t. query_r and key_r
                    grad_query_r += torch.einsum('bnik,bnkd->ind', grad_attn_scores_qbias_full,
                         key_chunk.to(torch.float32))
                    grad_key_chunk += torch.einsum('bnik,ind->bnkd', grad_attn_scores_qbias_full,
                         query_r.to(torch.float32))

                    grad_key_r += torch.einsum('bnqj,bnqd->jnd', grad_attn_scores_kbias_full,
                         query_chunk.to(torch.float32))
                    grad_query_chunk += torch.einsum('bnqj,jnd->bnqd', grad_attn_scores_kbias_full,
                         key_r.to(torch.float32))

                grad_query[:, :, i:i + chunk_size, :] += grad_query_chunk.to(torch.float32)
                grad_key[:, :, j:j + chunk_size, :] += grad_key_chunk.to(torch.float32)

    return grad_query, grad_key, grad_value, grad_query_r, grad_key_r



class MemoryEfficientAttention(torch.autograd.Function):
    @staticmethod
    def forward(ctx, query, key, value, mask=None, offset_matrix=None, query_r=None, key_r=None,
         chunk_size=1024):
        dtype = value.dtype
        # query, key, value = query, key, value
        key_chunk_size = min(chunk_size * 2, key.size(2))
        query_chunk_size = min(chunk_size, query.size(2))

        with torch.no_grad():
            reduced_values, all_attn_weights = forward_core(query, key, value,
                                                            query_chunk_size, key_chunk_size,
                                                            mask, offset_matrix,
                                                            query_r, key_r
                                                           )

        # Save full query, key, and value tensors, but not the intermediates
        ctx.save_for_backward(query, key, value, mask, offset_matrix, all_attn_weights, query_r, key_r)
        ctx.chunk_size = chunk_size

        return reduced_values.to(value.dtype)

    @staticmethod
    def backward(ctx, grad_output):

        dtype = grad_output.dtype
        query, key, value, mask, offset_matrix, all_attn_weights, query_r, key_r = ctx.saved_tensors
        chunk_size = ctx.chunk_size

        # Initialize gradients
        with (torch.no_grad()):
            grad_query, grad_key, grad_value, grad_query_r, grad_key_r = \
            backward_core(grad_output, query, key, value,
                          all_attn_weights, chunk_size,
                          mask=mask, offset_matrix=offset_matrix,
                          query_r=query_r, key_r=key_r)

        return grad_query.to(dtype), grad_key.to(dtype), grad_value.to(dtype), None, None, grad_query_r
             , grad_key_r, None

def memory_efficient_attention(query, key, value, mask=None,
                               offset_matrix=None, query_r=None, key_r=None,
                               chunk_size=2048):
    return MemoryEfficientAttention.apply(query, key, value, mask,
                                          offset_matrix, query_r, key_r,
                                          chunk_size)
```

C Limitations

C.1 Data Processing and Language

As is briefly admitted in the main text of the paper, the permutation of insertion operations described in the paper assumes the major proportion of the data is natural language, and mostly implemented to adapt to natural languages with latin-alphabet writing system like English, French and German. For languages with multi-byte-multi-token characters like Japanese and Chinese, current preprocessing pipeline fall back to autoregressive/identical permutation. This allows the toleration of moderately mixed-in multiligual data, but is definitely far from being the optimal solution. We leave this for future work.

C.2 Broader Impact

As a generative model with better controllability than current autoregressive models, due to the inevitable data bias, pretrained InsNeXts are at risk of producing harmful contents that may offend people of different self-identification and/or play a negative role in misinformation spreading. We have only scaled the model to be limited sizes as large as the maximum of 0.6B parameters, so this is currently still safely contained. However, we definitely call for regarding a broader societal impact when further scaling this proposed architecture in the future.

C.3 Suboptimal Hardware-oriented Optimization

We acknowledge our limitation of ability in pushing the hardware-related optimization to the best practice like in FlashAttention-2/3. According to our statistics, given the same model scale, our training efficiency is still inferior compared to LLMs with mature architectures like LLaMA. We look forward to broader collaboration to solve this issue in the future.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction thoroughly include major details on the basic pretraining setup, evaluation experiments and ablation study experiments, as well as the major highlights of the proposed model

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: See the limitation section of the appendix.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes]

Justification: The paper contains mostly empirical and minor theoretical results that simply transplant existing algorithms in novel use cases. We've properly cited the source or the original paper.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We inherit most experimental setups from peer-reviewed prior works and have tried our best to report necessary details for reproducing our results.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The code is *partially* available as we've included the major code in the supplementary materials. We will open-source the full code after the anonymity period of the review process ends.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We confirm we've include the necessary details at the best level of our ability.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We've tried our best to use different random seeds and multiple statistically independent runs of the downstream evaluations to make sure the stochasticity is not a major negative factor in reproducibility. We've stated this point in our experiment section.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We've included some brief details in the main part of the paper and more in the appendix, including the incidents and how that affect our training setup.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We confirm there are no intentional introduction of any factors violating the NeurIPS Code of Ethics, and any inevitable factors introduced by large-scale data is acknowledged in the limitation.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: See the Limitation section of the appendix.

    Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [No]

    Justification: As a foundation work of this new class of language models, we only have scaled the model to a very limited size without instruction-following capability. It is unlikely, as of now, for us to build safeguard mechanism with the commmon practices in autoregressive LLMs. We leave this for future work.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We confirm we've properly cited all related works and resources to the best of our knowledge.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The whole repository will be documented at a basic level upon submission, but we will continue to improve it after the anonymity period.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: [NA]

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: We've cited the involved medium-to-large sized language models properly.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.