PARALLEL TOKEN GENERATION FOR LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Autoregressive transformers are the backbone of modern large language models. Despite their success, inference remains slow due to strictly sequential prediction. Prior attempts to predict multiple tokens per step typically impose independence assumptions across tokens, which limits their ability to match the full expressiveness of standard autoregressive models. In this work, we break this paradigm by proposing an efficient and universal framework to jointly predict multiple tokens in a single transformer call, without limiting the representational power. Inspired by ideas from inverse autoregressive normalizing flows, we convert a series of random variables deterministically into a token sequence, incorporating the sampling procedure into a trained model. This allows us to train parallelized models both from scratch and by distilling an existing autoregressive model. Empirically, our distilled model matches its teacher's output for an average of close to 50 tokens on toy data and 5 tokens on a coding dataset, all within a single forward pass.

1 INTRODUCTION

Autoregressive transformers (Vaswani et al., 2017) are the foundation of today's large language models (LLMs) (Brown et al., 2020). Their sequential generation process, however, remains a major bottleneck: each token depends on the full history, requiring one forward pass per token. For long outputs, this increases the inference latency significantly when compared to what a single transformer call would achieve.

Many recent efforts aim to bypass this bottleneck by predicting multiple tokens at once. Broadly, they can be categorized into two lines of work: The first, speculative decoding, takes a systems approach, making predictions in a lightweight model that is verified by a large model (Leviathan et al., 2023; Chen et al., 2023; Sun et al., 2023; Zhong et al., 2025). The second line of work makes use of predicting several tokens independent of each other. This significantly reduces the search

Figure 1: **Our parallelized model generates the same text as its teacher in a fraction of the steps.** By the time our model *(bottom)* has generated an entire function, an autoregressive model *(top)* only generates the method's signature. Prompt: `Write a Python function that computes the factorial of a number`. Green tokens are accepted tokens in that step, red tokens are incorrect. Semitransparent tokens are rejected after the first mistake.

space for sequences and improves overall model quality (Qi et al., 2020; Gloeckle et al., 2024; DeepSeek-AI et al., 2025). Similarly, discrete diffusion iteratively refines generated sequences, again not modeling conditional dependencies between tokens in each denoising step (Hoogeboom et al., 2021; Austin et al., 2021). However, all of these methods still contain an irreducible sequential component to generate sequences.

Our work takes a step towards filling this gap. We propose a framework that, in theory, can generate arbitrary length sequences in parallel. This is enabled by a small but fundamental architectural change: Instead of sampling from the distributions predicted by an autoregressive model in a post-processing step, we feed the involved random variables as an input to the model: the model learns to sample. This enables it to anticipate which tokens will be sampled and predict them jointly. Similar frameworks have been formulated in the normalizing flow literature: Inverse Autoregressive Flows (Kingma et al., 2016) generate samples of many continuous dimensions in parallel, which we transfer to sampling discrete sequences.

Our contributions are threefold:

- We propose *Parallel Token Prediction* (PTP), a modeling approach for discrete data that generates multiple tokens in parallel (section 2.1). We theoretically confirm its universality to model arbitrary distributions (Theorems 1 and 2).
- PTP can be trained to predict several tokens either by distilling an existing teacher, effectively parallelizing it, or via cross-entropy on training data (section 2.2).
- Experimentally, we distill models on toy sequence data and real-world coding datasets, achieving an average number of close to 50 respectively 5 tokens identical to their teachers (section 3).

Together, our framework opens a design space to build models that accurately predict several tokens in parallel, ultimately reducing latency in language model output.

2 Parallel Token Prediction

2.1 Parallel Sampling

To construct our *Parallel Token Prediction* framework, let us recap how a classical transformer decoder generates text. It iteratively predicts the categorical distribution of the next token $t_i \in \{1, \dots, V\}$ based on all previous tokens $t_{<i} = (t_1, \dots, t_{i-1})$,

$$P_i := P(t_i | t_{<i}) \tag{1}$$

For simplicity, we assume this distribution is the final distribution that is used to generate tokens, in that it already reflects temperature scaling (Guo et al., 2017), top-k and top-p sampling (Holtzman et al., 2020), or other approaches trading sample diversity with quality. To sample a token from this distribution, one draws an auxiliary random variable $u_i \sim \mathcal{U}[0, 1]$ and looks up the corresponding token from the cumulative distribution function as follows:

$$t_i = \mathrm{Pick}(u_i, P_i) = \min_{j \in \{1, \dots V\}} \{j : F_{ij} > u_i\}, \quad \text{where } F_{ij} = \sum_{l=1}^{j} P_{il}. \tag{2}$$

Here, j iterates possible token choices, P_{il} is the probability to sample $t_i = l$, and F_{ij} is the cumulative distribution to sample a token $t_i \in \{1, \dots j\}$.

Figure 2(a) illustrates how, in traditional autoregressive models, we first sample t_i from P_i before moving on to predicting the next token t_{i+1}, as the distribution P_{i+1} depends on the selected token t_i. Every new token involves another model call, increasing latency. To break this iterative nature, note that while eq. (1) defines a distribution over possible next tokens, eq. (2) is a deterministic rule once the auxiliary variable u_i is drawn. Thus, write this rule as an explicit deterministic function:

$$t_i = f_P(t_{<i}; u_i) = \mathrm{Pick}(u_i, P(\cdot\,|t_{<i})). \tag{3}$$

Figure 3 illustrates how this function jumps from token to token as a function of u_i.

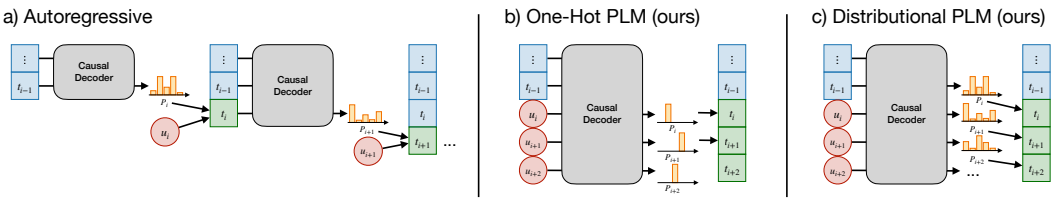

Figure 2: **Parallel Token Prediction Models predict several tokens in one model call.** (a) An autoregressive model predicts the distribution for token t_i, then uniformly samples an auxiliary variable u_i to select a token. This results in one model call per token. (b) One-Hot Parallel Token Prediction Models feed auxiliary variables into the model, making all tokens a deterministic choice. This allows the model to be executed only once. (c) Categorical Parallel Token Prediction Models model the distribution of each token, but predict them in parallel using the auxiliary variables.

This is all we need to perform parallel generation of text: All information about which token t_i we are going to select is available to the model if it has access to u_i as one of its inputs. By repeating the above argument and feeding all the auxiliary variables into the model, any subsequent token $t_{>i}$ can be predicted deterministically (proof in appendix B.1):

Theorem 1. *Given any probabilistic model P for next token prediction. Then, the future token t_k can be selected as a deterministic function f_P of previous tokens $t_{<i}$ and auxiliary variables $u_i, \dots, u_k \sim \mathcal{U}[0,1]$:*

$$t_k = f_P(t_{<i}; u_i, \dots u_k), \qquad \textit{for all } k \geq i. \quad (4)$$

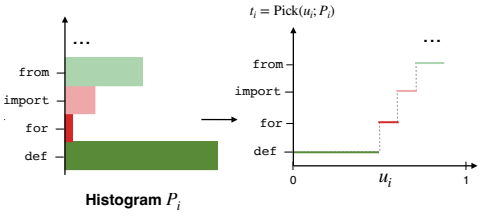

Figure 3: **Sampling from a discrete distribution.** Given a histogram P_i *(left)*, compute the inverse cumulative distribution function *(right)* and look up the token at a random location $u_i \in \mathcal{U}[0,1]$. Our framework relies on considering both parts jointly.

Theorem 1 shows a clear path to build a model that can sample many tokens in parallel: Instead of learning the distribution $P(t_k|t_{<k})$, we propose to directly fit the function $f_P(t_{<i}; u_i, \dots, u_k)$, which jointly predicts future tokens t_k.

Figure 2(b) visualizes how this can be implemented with a standard transformer (Vaswani et al., 2017) backbone: Alongside the previous tokens, simply feed the auxiliary random variables for the next N tokens into the model. It then predicts a discrete distribution over tokens $P(t_k|t_{<i}; u_i, \dots, u_k)$. Since by theorem 1, this distribution is singular at t_k, we take the argmax to get each token. We refer to this model as a **One-Hot Parallel Token Prediction Model** (O-PTP). O-PTPs can be trained to replicate an existing autoregressive model P, see section 2.2.1 for details.

An existing autoregressive model to train an O-PTP may not be available, however. For this case, and to allow access to the token distributions P_i, we propose **Categorical Parallel Token Prediction** (C-PTP). Instead of predicting future tokens t_j directly, it predicts their distributions in parallel. This recovers training directly from data, see section 2.2.2. The central difference to O-PTP is that we do not inform the prediction of a token t_i about the auxiliary variable u_i we will use to sample it. For the first token, the best prediction recovers the original autoregressive distribution in eq. (1):

$$P_i = P(t_i|t_{<i}, \cancel{u_i}) = P(t_i|t_{<i}). \quad (5)$$

Moving to the next token t_{i+1}, we now do pass in the auxiliary variable u_i used to sample the first token t_i. Since P_i and u_i uniquely determine t_i, u_i and t_i contain the same information. By the law of total probability, this recovers the same distribution as conditioning on the previous token:

$$P_{i+1} = P(t_i|t_{<i}, u_i) = P(t_{i+1}|t_{<i}, t_i). \quad (6)$$

Repeating this argument, we find that the distribution of every future tokens is available if we condition on all preceding auxiliary variables (proof in appendix B.2):

Theorem 2. *Given any probabilistic model P for next token prediction. Then, the distribution of a token t_k is fully determined by context tokens $t_{<i}$ and the past auxiliary variables $u_i, \ldots, u_{k-1}$:*

$$P(t_k|t_{<k}, u_i, \ldots, u_{k-1}) = P(t_k|t_{<k}), \qquad \textit{for all } k \geq i. \tag{7}$$

Figure 2(c) shows how this can be used to predict the distribution of the tokens $t_i, \ldots t_N$ in parallel. Just like for the O-PTP, first sample all required auxiliary variables $u_i, \ldots u_{N-1}$, and then predict all $P_k = P(t_k|t_{<i}, u_i, \ldots, u_{k-1})$ in parallel. Sampling from these distributions is done via $t_k = \text{Pick}(u_k, P_k)$. By using a causal decoder architecture, we can properly mask which token has access to which auxiliaries.

Both One-Hot and Categorical Parallel Token Prediction Models are constructions that allow predicting several tokens in parallel in a single model call. By Theorems 1 and 2, there are no fundamental restrictions apart from model capacity as to which distributions they can learn. In the next section, we propose two approaches to train these models, either by training from scratch (only C-PTP) or by distillation an existing model (both O-PTP and C-PTP).

2.2 Training

Before deriving the training paradigms for Parallel Token Prediction Models, let us quickly recall that autoregressive models are trained by minimizing the cross-entropy between samples from the training data $t \sim P(t)$ and the model P_θ

$$\mathcal{L}(\theta) = \mathbb{E}_{t\sim P(t)}\left[-\sum_{i=1}^{N} \log P_\theta(t_i|t_{1\ldots i-1})\right]. \tag{8}$$

Using a causal model such as a transformer (Vaswani et al., 2017) this loss can be evaluated on an entire sequence of tokens in a single model call (Radford et al., 2018). We first present how to distill both One-Hot and Categorical Parallel Token Prediction Models from a trained autoregressive model. We then show how the latter can be self-distilled from data alone via eq. (8).

2.2.1 Distillation

Both PTP variants can be trained to emulate the token-level predictions of an autoregressive teacher Q_φ, allowing for efficient, parallel generation of several tokens. We then call the PTP a student model P_θ. With enough data and model capacity, our algorithm leads to a student model that produces the same sequence of tokens in a single model call as the teacher does in high-latency autoregressive sampling (Theorems 1 and 2). We defer correcting errors arising from finite resources to the subsequent section 2.3.

To train the student for a given training sequence $t_1, \ldots, t_T$, we reverse engineer the auxiliary variables $u_1, \ldots, u_N$ under which the teacher would have generated it, split the sequence into context and prediction sequences, and then evaluate a loss that leads the student towards the correct generation. This process is summarized in algorithm 2 in appendix F.1.

Auxiliary variables. First, we extract the auxiliary variables that the teacher model would use to generate the training sequence. We evaluate the teacher distributions of each training token to get the cumulative discrete distributions $F_1, \ldots, F_T$ for each token. Inverting eq. (2), we find for every $k = 1, \ldots, T$:

$$u_k \in [F_{k,t_k-1}, F_{k,t_k}). \tag{9}$$

Since u_k is continuous, while t_k is discrete, we can randomly pick any compatible value. See appendix C for details.

Sequence splitting. Second, we split the training sequence into a context part $t_1, \ldots, t_{i-1}$ and a prediction part $t_i, \ldots, t_T$. We usually pick $i \sim P(i|t)$ randomly and predict a fixed subsequent window of tokens.

Loss evaluation. Third, while both parallel models depend on the auxiliary variables just extracted from the teacher, the training paradigm depends on concrete variant to distill.

For C-PTP, our model predicts a categorical distribution $P_{\theta,k}$ for each future token that we can compare to the distribution of the teacher model. We can distill with any loss $d(Q, P)$ that measures

divergence between categorical distributions. This could be the Kullback–Leibler divergence $d = \mathrm{KL}(Q \,\|\, P)$ or its reverse variant $d = \mathrm{KL}(P \,\|\, Q)$.

$$\mathcal{L}(\theta, t) = \mathbb{E}_{i \sim P(i|t)} \left[\sum_{k=i}^{T} d(Q_\varphi(t_k|t_{<k}), P_\theta(t_k|t_{<i}, u_{i \ldots k-1})) \right], \tag{10}$$

with u_k as in eq. (9). Note that while different losses have different convergence properties, crucially, $d = 0$ implies identical conditional distributions and a perfectly distilled model.

For O-PTP, remember from section 2.1 that our model predicts a distribution over all tokens of which we take the argmax to get our discrete prediction. To train, we can use the cross-entropy loss

$$\mathcal{L}(\theta, t) = \mathbb{E}_{i \sim P(i|t)} \left[-\sum_{k=i}^{T} \log P_\theta(t_k|t_{<i}, u_i, \ldots u_k) \right], \tag{11}$$

which can go to zero since t_k is a deterministic prediction by Theorem 1.

Sequence proposal distribution. We can optimize the above losses by sampling sequences $t \sim P(t)$ from any data source. From a theoretical standpoint, any proposal distribution with the same support as the teacher will train the student to replicate the teacher everywhere. In contrast to training with eq. (8), the student will learn to approximate $Q_\varphi(t)$ and not $P(t)$. We have a great degree of freedom in this choice and test several options empirically in section 3.1.2. If our goal is to deploy our parallelized student as a drop-in replacement of our teacher model, the lowest-variance option is to sample training sequences from the teacher. Another possibility is to directly sample training sequences from a dataset, such as the one that was used to train the teacher model in the first place. This might increase performance in transfer-learning settings, where we can focus on learning just the parts of the teacher model that are needed to complete the new task. This also has the further advantage that we can compute the teacher predictions $Q_\varphi(t_k|t_{<k})$ in parallel over a full sequence instead of iteratively having to generate it. Finally, we can sample sequences directly from the student model by first sampling auxiliary variables $u_i, \ldots, u_T \sim \mathcal{U}[0, 1]$ and then using our student model at its current state to sample training sequences in parallel. As the student's prediction gets closer to that of the teacher during training, this approaches the same training sequence distribution as if we had sampled the teacher directly. When sampling training sequences from the student, we can save a second call to the student model by swapping the roles of the teacher and student in the training algorithm. In particular, choosing auxiliary variables that are compatible with the student (instead of the teacher) and comparing how the teacher's output would have compared to the student's ground truth - with the exact same losses as before.

2.2.2 Inverse Autoregressive Training

Categorical Parallel Token Prediction Models can also be trained directly via eq. (8), avoiding the need to have a teacher model as target. For a given training sequence $t_1, \ldots, t_T$, we again split it into the context $t_{<i}$ and the following prediction $t_{\geq i}$. By picking i at random we allow each token t_k to be in any position of the parallel token prediction.

Exactly as during distillation, we have to find auxiliary variables that are compatible with every $t_k, k \geq i$. We can do this by selecting, randomly, any $u_k \in [F_{k,t_k-1}, F_{k,t_k})$, equivalently to eq. (9), where F_k, t_k now is the cumulative probability under P_θ (instead of the teacher model) to choose t_k when predicting that token. As this probability depends on the previous auxiliary variables $u_i, \ldots, u_{k-1}$, we select them iteratively. Specifically, we can alternate between computing the logits of $P_\theta(t_k|t_{<i}, u_i, \ldots, u_{k-1})$, and drawing u_k using equation 9.

Finally, we can train our model using the cross-entropy loss

$$\mathcal{L}(\theta) = \mathbb{E}_{t \sim P(t), i \sim P(i|t)} \left[-\sum_{k=i}^{N} \log P_\theta(t_k|t_{<i}, u_i, \ldots, u_{k-1}) \right]. \tag{12}$$

Algorithm 3 in appendix F.1 summarizes the procedure. A similar approach of iteratively determining latent variables (our auxiliaries) was proposed by Inverse Autoregressive Flows (Kingma et al., 2016), although they considered continuous variables that are traced through an invertible neural network.

2.3 Error Correction

The distillation procedure proposed in section 2.2.1 in theory leads to perfectly distilled parallel model. Practically, finite model capacity and compute limit infinite parallel sequence generation. In this section, we leverage ideas from speculative decoding (Leviathan et al., 2023) to obtain models that generate long sequences in as few model calls as possible while exactly adhering to the teacher.

A Parallel Token Prediction Model generates a sequence of tokens $t_i, \ldots, t_N$ from context $t_{<i}$ using auxiliary variables $u_i, \ldots, u_N$. To verify that the parallel token prediction is accurate, we can verify that eq. (4) is fulfilled by computing the distributions $P_i, \ldots, P_N$ in a single model call, and checking that indeed $u_k \in [F_{k,t_k}, F_{k,t_k-1})$. If there is a spurious token t_{k^*}, we replace it by the teacher prediction and roll out our model again, this time with context $t_{\leq k^*}$ and the remaining auxiliary variables $u_{k^*+1}, \ldots, u_N$. By repeating this, we obtain the same sequence as the teacher would have generated sequentially. This is made explicit in algorithm 1 in appendix F.1.

Intuitively, if PTP on average predicts c correct tokens before it first makes a mistake, we can expect the total number of model calls (including the verification step) to be close to $2/(c+1)$ per token instead of 1, significantly less if $c > 1$, greatly reducing latency.

Furthermore, if reducing latency is more important than total compute, we can already start predicting more tokens by another PTP call, for example by prematurely accepting the first c tokens, while the teacher is verifying the predicted sequence. Since we can always use the first token of the teacher, this ensures the wall-clock time to generate text is never slower than the autoregressive counterpart regardless of the student's quality.

Latency can be further decreased by running several PTP models in parallel with different offsets, minimizing the chances that none of the generated sequences will be accepted by the teacher. We discuss how to leverage additional computing resources that allow us to run many parallel PTPs to further decrease the total number of model calls in appendix D.

2.4 Limitations of Independent Prediction

Our Parallel Token Prediction framework removes an important limitation of the models in prior work such as discrete diffusion models (Hoogeboom et al., 2021; Austin et al., 2021) and multi-token prediction (Qi et al., 2020; Gloeckle et al., 2024): Whenever these models predict several tokens in parallel, they model these tokens independent of each other. This limits the maximum speedup they can achieve. Note that this in addition to any deficiencies arising from finite compute and model capacity.

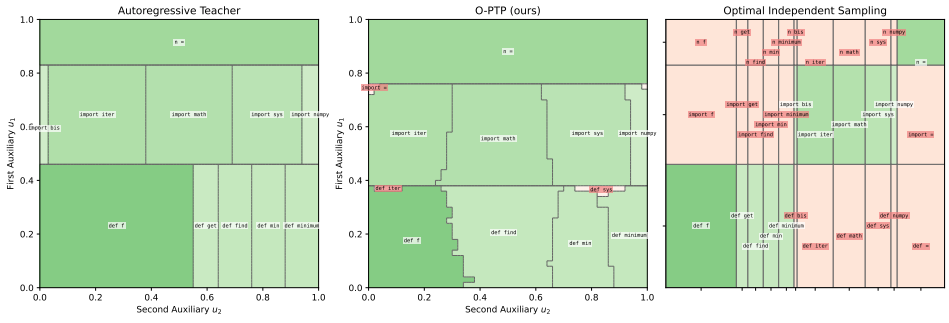

Figure 4: **Parallel Token Prediction generates meaningful pairs of tokens.** *(Left)* In a coding problem, autoregressive sampling first selects one of `def`, `import` or `n`, and then continues with meaningful predictions: function name to declare, package to import, or variable assignment. *(Center)* Our code completion model from section 3.2 also reliably predicts sensible combinations of tokens, but in a single model call. *(Red)* Only in rare cases (<1%), it produces incompatible predictions such as `def sys`. *(Right)* A model that independently predicts future tokens is bound to fail: In 60% of the cases, it combines incompatible tokens because the second token is not informed about the first.

Figure 4 shows how this limitation becomes evident in the task of writing a Python program to do some numerical computation. If you had to solve this problem, you might first import an external library via `import numpy`, or start defining a function as in `def f():`. For an autoregressive language model, this is an easy task. Sample the first token, that is `import` or `def`, and the second token can be identified depending on the first instruction.

For a model that predicts both tokens simultaneously, the prediction of both tokens has to be coordinated, or we will end up sampling code like `def numpy` or `import f`, which do not make sense in this context. Unfortunately, this is exactly what a model that predicts next tokens independently ends up doing in a significant number of cases: The best model can identify which tokens are good candidates for prediction, but it cannot coordinate which combinations go together:

$$P^{\text{indep}}(t_i, t_{i+1}|t_{<i}) = P(t_i|t_{<i})P(t_{i+1}|t_{<i}) \neq P(t_i|t_{<i})P(t_{i+1}|t_{<i}, t_i) = P(t_i, t_{i+1}|t_{<i}). \quad (13)$$

In the example in fig. 4, even the closest possible model to an autoregressive teacher predicts invalid tokens in 60% of the cases. Comparing this with our framework, Theorems 1 and 2 guarantee that a PTP can in principle exactly replicate any dependencies between tokens. The only remaining approximation is the finite model capacity. In the above example, 99% of the token pairs predicted by our trained model for code prediction are useful.

3 Experiments

We now verify empirically that our framework for *Parallel Token Prediction* not only is theoretically sound but enables meaningful parallel inference in practice. We first extensively test our framework on a computationally efficient discrete real-world dataset with a small vocabulary in section 3.1, and then demonstrate that our method scales to a practical language-prediction task where we parallelize a language-model from the LLama family (Zhang et al., 2024) in section 3.2. We give all details to replicate experiments in appendix F.2.

3.1 Exploring Design Choices

We now provide some of the specific choices we made when implementing the general framework of Parallel Token Prediction. Specifically, we discuss the empirical difference between O-PTP and C-PTP and which specific loss to choose. We will specify our model architecture and how to embed both tokens and auxiliary variables in the same embedding space, and lastly compare the proposal distributions our training sequences can be sampled from.

We test our framework by training a model that predicts pick-up locations for taxis in New York City. Based on a dataset (NYC TLC, 2017) that contains latitudes and longitudes for pick-up locations for all taxi rides in 2016, we divide the city into 25 neighborhoods via k-Means clustering to obtain a discrete-valued time-series that we can split into overlapping chunks of length N. This is a common benchmark dataset in the literature of marked temporal point processes (Xue et al., 2024).

As a teacher model, we pretrain a 29M-parameter autoregressive causal transformer based on the architecture of GPT-2 (Radford et al., 2019), using the cross-entropy loss in eq. (8). For our PLM we choose the same GPT-style transformer architecture as the teacher. This allows us to use the teacher's parameters as a warm-start. We evaluate all our parallel models in terms of the average number of leading tokens predicted by our student model that are identical to the teacher. In the end, this is the quantity that limits the maximum latency reduction that can be achieved, see section 2.3.

3.1.1 Auxiliary Variable Embeddings

In our experiments we use transformers that embedded tokens into a higher-dimensional embedding space via a learned embedding before adding a positional embedding. This doesn't work out-of-the box for our auxiliary variables since they are one-dimensional continuous variables. Thus we learn a separate embedding. We combine two components, for each of which we test several variants: (1) A learned affine linear transform [**lin**] or a fully connected neural network [**NN**]. (2) Feed either the scalar u [**fl**], a n-dimensional threshold-embedding $e_i = 1\{u \leq i/n\}$ [**th**], or an n-dimensional embedding $e_i = 1\{u2^{i-1} \bmod 1 \leq 0.5\}$ [**ar**] inspired by arithmetic coding (Witten et al., 1987).

Empirically, all methods work reasonably well, but a structured embedding leads to faster and more stable training convergence. This is similar to the transformer's positional embedding were both

Proposal Distribution $P(t)$	**kl**	**kl-rev**	**bce**	**ce**	MTP
Teacher	40	41	**45**	44	10.1
Student	44	39	**45**	**45**	10.1
Dataset	29	36	44	43	10.1

Table 1: **Our framework is compatible with several losses.** Average number of correct tokens (↑) on the taxi dataset, evaluated on 16000 samples. O-PTP are distilled with KL or reverse KL loss (**kl**, **kl-rev**), C-PTP with binary or categorical cross entropy loss (**bce**, **ce**). Independent prediction (MTP) (Gloeckle et al., 2024) achieves 10.1. Numbers rounded to reflect level of statistical certainty.

Model	all tokens	t_1	t_2	t_3	t_4	t_8	t_{16}
C-PTP	19.88	20.0	19.8	20.1	20.0	20.0	19.7
Autoregressive Teacher	19.81	19.81	–	–	–	–	–

Table 2: **The sample quality of a Categorical Parallel Token Prediction Model (C-PTP) matches that of autoregressive when trained on only the dataset**. Model perplexity (↓) for several positions within the prediction, on the taxi dataset, evaluated on 16000 samples. t_1 is the first token predicted after the context, t_2 the one after that. Numbers rounded to reflect level of statistical certainty.

learned and fixed embeddings work well but the later is preferred in practice (Vaswani et al., 2017). For further experiments we use the [**ar + lin**] embedding. Table 4 in appendix A shows the detailed effect of different embedding strategies.

3.1.2 Distillation Losses and Proposal Distributions

As our framework is deliberately general, it is compatible with a wide selection of losses. We here compare the distillation losses (section 2.2.1), focusing on KL and cross-entropy losses in eqs. (10) and (11). Specifically the KL loss (**kl**), reverse KL loss (**kl-rev**), binary cross-entropy loss (**bce**), and categorical cross-entropy loss (**ce**). During training we sample training sequences from a dataset and continuations $t_{\geq i}$ either from the teacher model Q_φ, the student model P_θ, or directly from a dataset. Table 1 shows the results for different losses. Empirically we note, that O-PTPs are easier to train than C-PTPs and achieve a higher number of average correct tokens. This is most likely due to the fact that O-PTPs do not have to predict the full token distribution accurately, which includes tail behavior, as long as they learn which token is the most likely given the auxiliary variable. In the following, we choose to sample training sequences from the teacher model for best results.

3.1.3 Inverse Autoregressive Training

Here, we confirm the ability of Categorical Parallel Token Prediction Models to be trained using just a dataset without having to be guided by a teacher model. We train our model as described in section 2.2.2 on the cross entropy loss in eq. (12). Table 2 shows a comparison of sample quality as measured by model perplexity. Our PLM is able to closely match the sample quality of a next-token prediction model while generating multiple tokens in parallel. The zero-shot average number of tokens that the PLM matches with the teacher model is 24 (c.f. 40 when trained via distillation) further indicating that the PLM has learned to predict future tokens well. This reduced performance is expected since the student never learned to exactly mimic the teacher; they make different mistakes.

3.2 Code Generation with TinyLlama 1.1B

We now scale our framework by adding parallelization to an existing autoregressive model in a realistic yet computationally feasible setting. To this end, we use TinyLlama 1.1B-Chat-v1.0 (Zhang et al., 2024) as a teacher and distill a O-PTP as explained in section 2.2.1. We distill a student model to replicate the teacher in solving CodeContests coding problems (Li et al., 2022). During training and inference of our student model we provide the full problem description as context, compute a continuation from our teacher and chose the starting position of our student's prediction at random within the training sequence. Table 3 compares our distilled model to one trained to

Parallelization technique	Avg. correct tokens (↑)
None	1.0
Independent prediction	2.5 ± 0.3
O-PTP (ours)	$\mathbf{5.1} \pm \mathbf{0.3}$

Table 3: **Our One-Hot Parallel Token Prediction Model (O-PTP) predicts significantly more tokens identical to its teacher than baselines.** We distill TinyLlama-1.1B (Zhang et al., 2024) on coding problems (Li et al., 2022) and compare against the naive autoregressive baseline (Brown et al., 2020), as well as independently predicting tokens (Qi et al., 2020; Gloeckle et al., 2024; DeepSeek-AI et al., 2025). Errors indicate the standard deviation over three runs.

independently predict the next tokens (Gloeckle et al., 2024). Figure 1 shows a qualitative sample of our model's predictions, and Figure 4 shows how it can outperform a model predicting several tokens independently.

4 Related Work

Speeding up the generation of autoregressive models and discrete sequence models in particular has been the focus on a broad body of work, see (Khoshnoodi et al., 2024) for an overview.

Our framework combines two ideas from the Normalizing Flow literature and imports them to modeling discrete data: Inverse Autoregressive Flows (IAF) are trained with fast prediction in mind (Kingma et al., 2016) by iteratively identifying latent variables (our auxiliary variables) that generate a particular continuous one-dimensional value, and Free-Form Flows (FFF) train a generating function when a fast parallel sampler is not available (Draxler et al., 2024).

In the LLM literature, speeding up generation has been approached from various angles. **Speculative decoding** takes a system perspective, using a small draft model to propose multiple tokens and a large target model to verify them (Leviathan et al., 2023; Chen et al., 2023). Variants verify entire sequences (Sun et al., 2023) or use a smaller verifier network Zhong et al. (2025) to improve quality and speed. **Latent variable methods** first sample latent codes from the prompt so that the distribution of subsequent tokens factorizes given latent codes (Gu et al., 2018; Ma et al., 2019). **Diffusion language models** leave autoregressive sampling behind by iteratively refining the text starting from a noisy or masked variant (Hoogeboom et al., 2021; Austin et al., 2021). **Multi-head output models** predict several next tokens independent of each other (Qi et al., 2020; Gloeckle et al., 2024; DeepSeek-AI et al., 2025), narrowing down on the possible set of next tokens. Both diffusion and multi-head models assume independence of tokens, which is fundamentally limited in modeling capacity (section 2.4).

In contrast to the above, our work introduces a new class of fast language models that are universal in the sense that can approximate arbitrary dependence between several tokens in a single model call. Our new method is complementary to existing approaches, and we leave exploring these combinations open for future research.

5 Conclusion

In this paper, we introduce *Parallel Token Prediction*, a framework that permits consistent generation of several tokens in a single autoregressive model call. It eliminates the independence assumptions that limited prior approaches, allowing us to model tokens with arbitrary dependency between them. Empirically, we show that existing models can be distilled into efficient parallel samplers. With error correction, these models produce identical output as a teacher while significantly reducing latency.

This speedup makes language models more practical for real-time applications. Future work includes extending our framework to large scale models, multimodal generation, combining it with complementary acceleration strategies, and exploring theoretical limits on parallelization.

Overall, our results suggest that the sequential bottleneck in autoregressive transformers is not inherent, and that universal, efficient parallel generation is within reach.

Ethics Statement

Our work focuses on reducing the inference time of Large Language Models, enabling more computations per unit time and supporting large-scale or real-time applications. While this can improve responsiveness and resource efficiency, it may also increase the potential for misuse, such as generating misinformation or automated spam at higher volumes. Faster inference does not mitigate underlying model biases, so responsible deployment, monitoring, and safeguards are critical to balance performance gains with societal risks.

Reproducibility Statement

We include proofs for all theoretical results introduced in the main text in appendix B. We include further experimental and implementation details (including model architectures and other hyperparameter choices) in section 3.1 and appendix F. Our code will be made available by the time of publication.

References

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in neural information processing systems*, 34:17981–17993, 2021.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in neural information processing systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan

Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. DeepSeek-V3 Technical Report, February 2025. URL http://arxiv.org/abs/2412.19437.

Felix Draxler, Peter Sorrenson, Lea Zimmermann, Armand Rousselot, and Ullrich Köthe. Free-form Flows: Make Any Architecture a Normalizing Flow. In *Artificial Intelligence and Statistics*, 2024.

William Falcon and The PyTorch Lightning team. PyTorch lightning, March 2019.

Fabian Gloeckle, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. Better & faster large language models via multi-token prediction. In *Proceedings of the 41st international conference on machine learning*, pp. 15706–15734, 2024.

Jiatao Gu, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. Non-autoregressive neural machine translation. In *International conference on learning representations*, 2018. URL https://openreview.net/forum?id=B1l8BtlCb.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International conference on machine learning*, pp. 1321–1330. PMLR, 2017.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International conference on learning representations*, 2020.

Emiel Hoogeboom, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. Argmax flows and multinomial diffusion: Learning categorical distributions. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=6nbpPqUCIi7.

J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3): 90–95, 2007.

Mahsa Khoshnoodi, Vinija Jain, Mingye Gao, Malavika Srikanth, and Aman Chadha. A Comprehensive Survey of Accelerated Generation Techniques in Large Language Models, May 2024. URL http://arxiv.org/abs/2405.13019. arXiv:2405.13019 [cs].

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 2015.

Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (eds.), *Advances in neural information processing systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/ddeebdeefdb7e7e7a697e1c3e3d8ef54-Paper.pdf.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th international conference on machine learning*, volume 202 of *Proceedings of machine learning research*, pp. 19274–19286. PMLR, July 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

Xuezhe Ma, Chunting Zhou, Xian Li, Graham Neubig, and Eduard Hovy. FlowSeq: Non-autoregressive conditional sequence generation with generative flow. In *Proceedings of the 2019 conference on empirical methods in natural language processing*, Hong Kong, November 2019.

Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman (eds.), *9th Python in Science Conference*, 2010.

New York City Taxi and Limousine Commission. 2016 yellow taxi trip data, 2017. City of New York, OpenData portal.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

Weizhen Qi, Yu Yan, Yeyun Gong, Dayiheng Liu, Nan Duan, Jiusheng Chen, Ruofei Zhang, and Ming Zhou. ProphetNet: Predicting future n-gram for sequence-to-SequencePre-training. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the association for computational linguistics: EMNLP 2020*, pp. 2401–2410, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.217. URL https://aclanthology.org/2020.findings-emnlp.217/.

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Technical Report*, 2019.

Ziteng Sun, Ananda Theertha Suresh, Jae Hun Ro, Ahmad Beirami, Himanshu Jain, and Felix Yu. SpecTr: Fast speculative decoding via optimal transport. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in neural information processing systems*, volume 36, pp. 30222–30242. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/6034a661584af6c28fd97a6f23e56c0a-Paper-Conference.pdf.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Daniel Bikel, Lukas Blecher, Nikolay Bogoychev, William Brannon, Anthony Brohan, Humberto Caballero, Andy Chadwick, Jenny Lee, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

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

Ian H. Witten, Radford M. Neal, and John G. Cleary. *Arithmetic Coding for Data Compression*, volume 30. Communications of the ACM, 1987.

Siqiao Xue, Xiaoming Shi, Zhixuan Chu, Yan Wang, Hongyan Hao, Fan Zhou, Caigao Jiang, Chen Pan, James Y. Zhang, Qingsong Wen, Jun Zhou, and Hongyuan Mei. EasyTPP: Towards open benchmarking temporal point processes. In *International conference on learning representations (ICLR)*, 2024. URL https://arxiv.org/abs/2307.08097.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. TinyLlama: An open-source small language model, 2024. arXiv: 2401.02385 [cs.CL].

Meiyu Zhong, Noel Teku, and Ravi Tandon. Speeding up speculative decoding via sequential approximate verification. In *ES-FoMo III: 3rd workshop on efficient systems for foundation models*, 2025. URL https://openreview.net/forum?id=Y4KcfotBkf.

Model	**fl + NN**	**th + lin**	**th + NN**	**ar + lin**	**ar + NN**	MTP
O-PTP	35.9	40.9	39.1	45.4	**46.1**	10.1
C-PTP	28.8	36.8	36.6	**40.4**	35.7	10.1

Table 4: **Structured embeddings of auxiliary variables u_k are more stable than fully-learned embeddings.** Average number of correct tokens (↑) on the taxi dataset, evaluated on 16000 samples. Trained using the KL loss (C-PTP) and binary cross-entropy loss (O-PTP), respectively. Independent prediction (MTP) (Gloeckle et al., 2024) achieves 10.1. Numbers rounded to reflect level of statistical certainty.

b	2	1	0.5	MTP
O-PTP	12.9	**13.9**	13.8	8.4
P-PTP	13.6	**13.8**	13.5	8.4

Table 5: **Different sampling strategies for u_k are available.** Average number of correct tokens (↑) for $\tilde{u}_k \sim \mathrm{Beta}(b, b)$ on the taxi dataset, evaluated on 16000 samples, with $N = 16$. Trained using the KL loss (C-PTP) and binary cross-entropy loss (O-PTP), respectively. Independent prediction (MTP) (Gloeckle et al., 2024) achieves 8.4. Numbers rounded to reflect level of statistical certainty.

A Additional Ablation Results

B Proofs

B.1 Proof of theorem 1

Proof. By theorem 2, it holds that the distribution of token $t_k, k \geq i$ is fully determined by $t_1, \ldots, t_{k-1}$ and $u_i, \ldots, u_{k-1}$, showing that the categorical distribution P_k of token t_k is fully determined.

Thus, the function to compute token t_k is given by eq. (2):

$$t_k = f_P(t_1, \ldots, t_{i-1}; u_i, \ldots u_k) = \mathrm{Pick}(u_k; P_k). \tag{14}$$

□

B.2 Proof of theorem 2

Proof. We prove by induction over $k, k \geq i$.

For $k = i$, there is nothing to show, since there are no auxiliaries involved in the statement.

For $k \mapsto k + 1$, assume the statement holds for k. This gives us access to the distribution P_k of the token t_k. Since token t_k is uniquely determined from P_k and u_k via eq. (3), any distribution conditioning on P_k, t_k can instead condition on P_k, u_k via the law of total probability. □

C Sampling of Auxiliary Variables

Our framework conditions, for a prompt $t_{<i}$, not on token t_k directly but on the auxiliary variable $u_k \in [F_{k,t_k-1}, F_{k,t_k})$ that contains the same information. During inference we sample $u_k \sim \mathcal{U}[0, 1]$ as to not bias our predictions. During training on the other hand, we have more flexibility and can sample the permissible interval using $u_k = F_{k,t_k-1} + \tilde{u}_k \left[F_{k,t_k} - F_{k,t_k-1}\right]$, where $\tilde{u}_k \sim \mathrm{Beta}(b, b)$. For $b = 1$ this simplifies to a uniform distribution while $b \neq 1$ puts more or less weight on predictions that land closer to the border of the permissible interval and thus are more difficult to predict. Training results for different values of b can be found in Table 5. Empirically, we find that while the choice of b does not seem to effect the final average number of correct samples, a larger b might speeds up the earlier stages of training while a smaller b might yield slightly better sample quality during inference, as measured by model perplexity.

M	1	4	16	64	256	1024	10^6	∞
Avg. correct tokens	45.36	49.67	54.76	55.74	56.91	59.79	46.01	90.17

Table 6: **Additional compute increases correctness**. Average number of correct tokens (↑) for M O-PTPs running in parallel on the taxi dataset, with sequence length $N = 100$.

N	1	2	4	8	16	64	100	∞
Avg. correct tokens	1.00	1.99	3.93	7.59	13.90	36.60	45.36	48.76
Best MTP	1.00	1.91	3.53	5.86	8.40	10.08	10.07	10.20

Table 7: **Less compute decreases correctness**. Average number of correct tokens (↑) for limited number of predicted tokens N per O-PTP call on the taxi dataset, $M = 1$.

D Abundant Computational Resources

In section 2.3 we discussed how to leverage several PTPs that run in parallel to further reduce latency. Another way to leverage several models run at once is to use them to improve the expected number of correct tokens directly. Specifically, for a fixed context we can let M PTPs compute M independent predictions using independently drawn auxiliary variables $u_{i,m}, \ldots, u_{i+N,m}$. By choosing the best prediction, i.e. the one that gives us the best chance of a higher number of correct tokens, we can improve latency further.

Crucially, we have to choose the best prediction in a way that doesn't bias the marginal distribution over future tokens. If we, for example, naively choose the sequence that is correct for the most amount of tokens, we will bias our prediction towards sequences that are easier to predict. On way to achieve bias-free improvements is to pick the set of auxiliary variables that lands, on average, closest to the center of a token's valid interval $I_k(t_k) = [F_{k,t_k}, F_{k,t_k-1})$ where F_{k,t_k} is the cumulative probability under Q_φ to choose t_k when predicting that token. Specifically, choose

$$\text{argmax}_m \sum_{k=i}^{i+N} \left| \frac{u_{k,m} - F_{k,t_{k,m}}}{F_{k,t_{k.m}-1} - F_{k,t_{k,m}}} - \frac{1}{2} \right|. \tag{15}$$

This does not bias the marginal distribution but does bias the distribution of the selected u_k to be closer to the center of it's interval $I_k(t_k)$. making the prediction less prone to small differences in the teacher's and student's logits. In the limit $M \to \infty$ we always select the middle point of $I_k(t_k)$ yielding an upper bound to the possible improvement. Table 6 shows the performance gains on the taxi dataset.

We can combine both techniques, avoiding the additional latency of verification while still keeping the higher expected number of correct tokens. Because the selection in eq. (15) relies on the teacher logits it can only be made after the verification step. To avoid waiting for the verification we assume, as before, that after a model call one of $n = 1 \ldots S$ tokens are correct and pre-compute the future tokens based on this assumption. Instead of one call as before, we now have to make M-many calls for each n. After the verification step we discard all but the best call from the correct n^*. As we have to repeat this for all M viable calls, that are yet to being verified, in parallel this approach benefits from M^2S PTPs running in parallel.

E Restricted Computational Resources

Limiting the number N of token's our PTP predicts at once to a smaller number will reduce the total number of floating point operations, increasing energy efficiency. This, of course, negatively effects the possible latency gains, especially since N is an upper bound on the average number of correct tokens. Table 7 shows the result for different values of N on the taxi dataset.

F Experimental Details

F.1 Algorithms

Algorithm 2 shows how to distill a PTP from a teacher, algorithm 3 shows how to train directly from data.

Algorithm 1 Sampling with error correction

Require: Sequence proposal distribution $P(t)$ (teacher, student, dataset, or combination), teacher model $Q_\varphi(t)$, one-hot or categorical PTP P_θ. Input sequence $(t_1, \dots, t_{i-1})$.
  Sample $u_k \sim \mathcal{U}[0,1]$ for all $k \geq i$.
  while $i < T$ **do**
    if P_θ is one-hot PTP **then**
      $P_k \leftarrow P_\theta(t_k|t_{<i}, u_i, \dots, u_k)$, jointly for all $k \geq i$. ▷ Student one-hot distributions
      $t_k = \mathrm{argmax}_l P_{kl}$, for all $k \geq i$.
    else
      $P_k \leftarrow P_\theta(t_k|t_{<i}, u_i, \dots, u_{k-1})$, jointly for all $k \geq i$. ▷ Student categorical distributions
      $t_k = \mathrm{Pick}(u_k, P_k)$, for all $k \geq i$.
    end if
    $Q_k \leftarrow Q_\varphi(t_k|t_{<k})$, jointly for all $k \geq i$. ▷ Teacher categorical distributions.
    $\tilde{t}_k \leftarrow \mathrm{Pick}(u_k, Q_k)$, for all $k \geq i$.
    $i \leftarrow \min_{k>i}\{k : t_k \neq \tilde{t}_k\}$. ▷ First error
    $t_i = \tilde{t}_i$.
  end while

Algorithm 2 Training PTP (distillation)

Require: Sequence proposal distribution $P(t)$ (teacher, student, dataset, or combination), cutoff distribution $P(i|t)$, teacher model $Q_\varphi(t)$, one-hot or categorical PTP P_θ.
  while not converged **do**
    Sample $t \sim P(t)$
    $P_k = Q_\varphi(t_k|t_{<k})$ in single model call.
    Sample $u_k \in [F_{k,t_k-1}, F_{k,t_k})$.
    Sample $i \sim P(i|t)$.
    Compute $\nabla_\theta \mathcal{L}(\theta, t)$ using eq. (10) or eq. (11).
    Gradient step.
  end while

Algorithm 3 Training PTP (inverse autoregressive)

Require: Dataset $P(t)$, cutoff distribution $P(i|t)$, categorical PTP P_θ.
  while not converged **do**
    Sample $t \sim P(t)$
    Sample $i \sim P(i|t)$.
    for $k = i, \dots, N$ **do**
      $P_k = P_\theta(t_k|t_{<i}, u_i, \dots, u_{k-1})$, with auxiliary available form previous iterations.
      Sample $u_k \in [F_{k,t_k-1}, F_{k,t_k})$.
    end for
    Compute $\nabla_\theta \mathcal{L}(\theta, t)$ using eq. (12).
    Gradient step.
  end while

F.2 Training details

The teacher used in section 3.1 is a GPT-2–style transformer language model with 4 transformer layers, a hidden size of 1536, and approximately 29 million trainable parameters. Each layer follows

the standard GPT-2 architecture, consisting of multi-head self-attention and position-wise feedforward sublayers, combined with residual connections and layer normalization. The vocabulary size is set 25. Unless otherwise noted, all other hyperparameters and initialization schemes follow the original GPT-2 specification (Radford et al., 2019). During training and inference of our student model we don't provide any context and evaluate the correctness of the next $N = 100$ tokens, by comparing $Q_\varphi(t_k|t_{<k})$ and $P_\theta(t_k)$. For results on a smaller $N = 16$, see appendix E. We train every model for 150k steps with a batch size of 32 with the Adam optimizer (Kingma & Ba, 2015) and learning rate 0.0001.

The teacher model used in section 3.2 is a dialogue-tuned variant of the TinyLlama (Zhang et al., 2024) 1.1 billion parameter model, adopting the same architecture and tokenizer as LLaMA 2 (Touvron et al., 2023). The model uses a transformer architecture comprising 22 transformer layers, each with standard multi-head self-attention, SwiGLU feedforward blocks, residual connections, and layer normalization. The embedding and hidden dimension is 2048, and the intermediate (feedforward) dimension is 5632, consistent with a LLaMA-style scaling. The vocabulary size is $32,000$. The parameters are available via `https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0`. During training and inference, we evaluate the correctness of the next $N = 64$ tokens. We train every model for 100k steps with a batch size of 64 with the AdamW optimizer (Loshchilov & Hutter, 2019) on eq. (11) and learning rate 0.0001. We generate training and validation data by generating code completions of maximum length 320 tokens from the teacher, with $P(i|t)$ randomly sampling a sequence of length N in the completion. The teacher is prompted with the training respectively validation data from (Li et al., 2022). We use a teacher sampling temperature of 0.7, top-$k = 50$ and top-$p = 0.9$, as is recommended for this model. The student is traine don these adapted logits.

For the MTP baseline, we use eq. (10) with uninformative us in otherwise identical code for a fair comparison.

We base our code on `PyTorch` (Paszke et al., 2019), PyTorch Lightning (Falcon & The PyTorch Lightning team, 2019), Numpy (Harris et al., 2020), Matplotlib (Hunter, 2007) for plotting and Pandas (McKinney, 2010) for data evaluation.

F.3 Prompt for Figure 4

```
You are given a permutation p_1, p_2, ..., p_n.

In one move you can swap two adjacent values.

You want to perform a minimum number of moves, such that in the end there
     will exist a subsegment 1,2,..., k, in other words in the end there
    should be an integer i, 1 <= i <= n-k+1 such that p_i = 1, p_{i+1} =
    2, ..., p_{i+k-1}=k.

Let f(k) be the minimum number of moves that you need to make a
    subsegment with values 1,2,...,k appear in the permutation.

You need to find f(1), f(2), ..., f(n).

Input

The first line of input contains one integer n (1 <= n <= 200 000): the
    number of elements in the permutation.

The next line of input contains n integers p_1, p_2, ..., p_n: given
    permutation (1 <= p_i <= n).

Output

Print n integers, the minimum number of moves that you need to make a
    subsegment with values 1,2,...,k appear in the permutation, for k=1,
    2, ..., n.

Examples
```

```
Input


5
5 4 3 2 1

Output


0 1 3 6 10

Input


3
1 2 3

Output


0 0 0
```