Predictive Sampling with Forecasting Autoregressive Models

Auke Wiggers 1 Emiel Hoogeboom 2 3

Abstract

Autoregressive models (ARMs) currently hold state-of-the-art performance in likelihood-based modeling of image and audio data. Generally, neural network based ARMs are designed to allow fast inference, but sampling from these models is impractically slow. In this paper, we introduce the predictive sampling algorithm: a procedure that exploits the fast inference property of ARMs in order to speed up sampling, while keeping the model intact. We propose two variations of predictive sampling, namely sampling with ARM fixed-point iteration and learned forecasting modules. Their effectiveness is demonstrated in two settings: i) explicit likelihood modeling on binary MNIST, SVHN and CIFAR10, and ii) discrete latent modeling in an autoencoder trained on SVHN, CIFAR10 and Imagenet32. Empirically, we show considerable improvements over baselines in number of ARM inference calls and sampling speed.

1. Introduction

Deep generative models aim to approximate the joint distribution \( P(x) \) of high-dimensional objects, such as images, video and audio. When a model of the distribution is available, it may be used for numerous applications such as anomaly detection, inpainting, super-resolution and denoising. However, modeling high-dimensional objects remains a notoriously challenging task.

A powerful class of distribution models called deep autoregressive models (ARMs) (Bengio & Bengio, 2000; Larochelle & Murray, 2011) decomposes the high-dimensional joint distribution into single-dimensional conditional distributions, using the chain rule from probability theory. Neural network based ARMs currently hold state-of-the-art likelihood in image and audio domains (van den Oord et al., 2016a;b; Salimans et al., 2017; Chen et al., 2018; Menick & Kalchbrenner, 2018; Child et al., 2019).

A major limitation of ARMs is that the autoregressive computation can be parallelized only in a single direction: either evaluation or sampling. Generally, these models are trained using likelihood evaluation and empirical computational cost of training is substantially higher than that of sampling. As such, ARMs are designed to allow for fast evaluation, but sampling from these models is prohibitively slow. In the literature, there are methods that try to accelerate sampling by breaking autoregressive structure a priori, but consequently suffer from a decrease in likelihood performance (Reed et al., 2017). Another approach approximates an autoregressive density model via distillation (van den Oord et al., 2018), but this method provides no guarantees that samples from the distilled model originate from the original model distribution.

This paper proposes a new algorithm termed predictive sampling, which i) accelerates discrete ARM sampling, ii) keeps autoregressive structure intact, and iii) samples from the true model distribution. Predictive sampling forecasts which values are likely to be sampled, and uses the parallel inference property of the ARM to reduce the total number of required ARM forward passes. To forecast future values,
we introduce two methods: ARM fixed-point iteration and learned forecasting. These methods rely on two insights: i) the ARM sampling procedure can be reparametrized into a deterministic function and independent noise, and ii) activations of the penultimate layer of the ARM can be utilized for computationally efficient forecasting. We demonstrate a considerable reduction in the number of forward passes, and consequently, sampling time, on binary MNIST, SVHN and CIFAR10. Additionally, we show on the SVHN, CIFAR10 and Imagenet32 datasets that predictive sampling can be used to speed up ancestral sampling from a discrete latent autoencoder, when an ARM is used to model the latent space. For a visual overview of the method, see Figure 1.

2. Methodology

Consider a variable \( x \in \mathcal{X} \), where \( \mathcal{X} \) is a discrete space, for example \( \mathcal{X} = \{0, 1, \ldots, 255\}^d \) for 8 bit images, where \( d \) is the dimensionality of the data. An autoregressive model views \( x \) as a sequence of 1-dimensional variables \((x_i)_{i=1}^d\), which suggests the following universal probability model (Bengio & Bengio, 2000; Larochelle & Murray, 2011):

\[
P_{\text{ARM}}(x) = \prod_{i=1}^d P_{\text{ARM}}(x_i | x_{<i}), \quad (1)
\]

Universality follows from the chain rule of probability theory, and \( x_{<i} \) denotes the values \((x_j)_{j=1}^{i-1}\). Samples from the model \( x \sim P_{\text{ARM}} \) are typically obtained using ancestral sampling:

\[
x_i \sim P_{\text{ARM}}(x_i | x_{<i}) \quad \text{for} \quad i = 1, \ldots, d \quad (2)
\]

In practice, ARMs can be implemented efficiently using deep neural networks. Let \( f \) be a strictly autoregressive function such that when \( h = f(x) \), the representation \( h_i \) depends only on input values \( x_{<i} \). The parameters \( \theta_i \) for the distribution over \( x_i \) are then an autoregressive function of the representation such that \( \theta_i \) depends on \( h_{<i} \). Using this formulation it is possible to parallelize ARM inference, i.e., to obtain a log-likelihood for every variable in parallel. However, in this setting, naive sampling from an ARM requires \( d \) forward calls that cannot be parallelized.

2.1. Predictive Sampling

Consider the naive sampling approach. First, it computes an intermediate ARM representation \( h_1 = f_1(\emptyset) \) and distribution parameters \( \theta_1(h_1) \), then samples the first value \( x_1 \sim P(x_1 | \theta_1) \). Only then can the next representation \( h_2 = f_2(x_1) \) be computed.

In the setting of predictive sampling, suppose now that we can obtain a forecast \( \hat{x}_1 \), which is equal to \( x_1 \) with high probability. In this case \([h_1, h_2] = f(\hat{x}_1)\) can be computed in parallel. We say that \( h_2' \) is valid, i.e., it is equal to \( h_2 \), if \( \hat{x}_1 \) equals \( x_1 \). We proceed as before and sample \( x_1 \sim P(x_1 | \theta_1(h_1)) \). If the sampled \( x_1 \) is indeed equal to the forecast \( \hat{x}_1 \), the representation \( h_2' \) is valid, and we can immediately sample \( x_2 \sim P(x_2 | h_2) \) without additional calls to \( f \). More generally, for a sequence of \( n \) correct
Isolating stochasticity via reparametrization The sampling step introduces unpredictability for each dimension, which may fundamentally limit the number of subsequent variables that can be predicted correctly. For example, even if every forecasted variable has a 60% chance of being correct, the expected length of a correct sequence will only be $\sum_{k=1}^{\infty} 0.6^k = 1.5$. To solve this issue, we reparametrize the sampling procedure from the ARM using a deterministic function $g$ and a stochastic noise variable $\epsilon$. A sample $x_i \sim P_{\text{ARM}}(\cdot \mid x_{<i})$ can equivalently be computed using the deterministic function $g$ conditioned on random noise $\epsilon$:

$$x_i = g_i(x_{<i}, \epsilon) \quad \text{where} \quad \epsilon \sim p(\epsilon). \quad (4)$$

Such a reparametrization always exists for discrete distributions. As a consequence the sampling procedure from the ARM has become deterministic given noise $\epsilon$. This is an important insight, because the reparametrization resolves the aforementioned fundamental limit of predicting a stochastic sequence. For instance, consider an ARM that models categorical distributions over $x$ using log-probabilities $\mu$. One method to reparametrize categorical distributions is the Gumbel-Max trick (Gumbel, 1954), which has recently become popular in machine learning (Maddison et al., 2014; 2017; Jang et al., 2017; Kool et al., 2019). By sampling standard Gumbel noise $\epsilon \sim G_{d \times K}$ the categorical sample $x_i \sim P_{\text{ARM}}(\cdot \mid x_{<i})$ can be computed using:

$$x_i = \arg\max_c \left( \mu_{i,c}(x_{<i}) + \epsilon_{i,c} \right), \quad (5)$$

where $c \in \{1, \ldots, K\}$ represents a category and $\mu_{i,c} = \log P_{\text{ARM}}(x_i = c \mid x_{<i})$ is its log probability for location $i$.

Although we focus on the discrete setting here, reparametrization noise can be obtained for many common continuous probability distributions (Ruiz et al., 2016). Note that in the continuous setting, verifying that the forecast is equal to the ARM output will depend on numerical precision, and a margin of error must be used.

**Shared Representation** In theory, the future sequence can be predicted perfectly using $\epsilon$ and $x_{<i}$, as it turns the ARM into a deterministic function $g$. In practice however, the amount of computation that is required to predict the future sequence perfectly may exceed the computational cost of the ARM.

Let $m$ be the new number of iterations that predictive sampling requires to converge. It is desirable to design F such that the added computational cost of $F$ is lower than the computational cost that was saved by the reduced number of ARM calls. Specifically, this requires that $m \cdot \text{cost}(F) \leq (d - m) \cdot \text{cost}(\text{ARM})$. 

**Algorithm 1 Predictive Sampling**

| Input: ARM $P_{\text{ARM}}$, forecasting function $F$ |
| Output: $x$ |
| let $i \leftarrow 0$, $x \leftarrow 0$ |
| while $i < d$ do |
| $\hat{x}_i, \hat{x}_{i+1}, \ldots \leftarrow F_i(x)$ // Forecast |
| $x \leftarrow x_{<i} \circ [\hat{x}_i, \hat{x}_{i+1}, \ldots]$ |
| $x', x'_{i+1}, \ldots \sim P_{\text{ARM}}(\cdot \mid x)$ |
| // While forecast $\hat{x}_i$ is correct, output $x'_{i+1}$ is valid |
| while $\hat{x}_i = x'_i$ and $i < d$ do |
| $i \leftarrow i + 1$ |
| end while |
| if $i < d$ then |
| $x[i] \leftarrow x'_i$ // Overwrite the input vector |
| $i \leftarrow i + 1$ |
| end if |
| end while |

forecasts, $n$ re-computations of $f$ are saved. A general description of these steps can be found in Algorithm 1, where $\circ$ denotes vector concatenation and $F_i$ denotes a function that outputs forecasts starting at location $i$. Note that the ARM returns probability distributions $P_{\text{ARM}}(\cdot \mid x)$ for all locations simultaneously, even if the input partially consists of placeholder values. A corresponding visualization is shown in Figure 2.

### 2.2. Forecasting

The forecasting function $F_i$ can be formalized as follows. Specifically, consider the vector $x$ in Algorithm 1, which contains valid samples until position $i$, i.e., the variables $x_{<i}$ are valid samples from the ARM. Let $\hat{x}_i$ denote a forecast for the variable $x_i$. A forecasting function $F_i$ aims to infer the most likely future sequence starting from position $i$ given all available information thus far:

$$\hat{x}_i, \hat{x}_{i+1}, \ldots = F_i(x). \quad (3)$$

Using this notion, we can define the predictive sampling algorithm, given in Algorithm 1. It uses the forecasting function $F_i$ to compute a future sequence $[\hat{x}_i, \hat{x}_{i+1}, \ldots]$, and combines known values $x_{<i}$ with this sequence to form the ARM input. Utilizing this input the ARM can compute an output sequence for future time steps $x'_i, x'_{i+1}, \ldots$ in parallel. The ARM output $x'_i$ is valid: it is a sample from the true model distribution, as the conditioning $x_{<i}$ does not contain forecasts. For each consecutive step $t$ where forecast $\hat{x}_{i+t}$ is equal to the ARM output $x'_{i+t}$, the subsequent output $x'_{i+t+1}$ is valid as well. When the forecasting function does not agree with the ARM, we write the last valid output to the input vector $x$, and proceed to the next iteration of predictive sampling. This process is repeated until all $d$ variables have been sampled.
To keep the computational cost of $\mathbf{F}$ low, the ARM representation $\mathbf{h} = f(\mathbf{x})$ from the previous iteration of predictive sampling is shared with the forecasting function:

$$\tilde{x}_i, \tilde{x}_{i+1}, \ldots = F_i(\mathbf{x}, \mathbf{h}, \epsilon).$$

(6)

When forecasts starting from location $i$ are required, the variables $\mathbf{x}_{\leq i}$ are already valid model outputs. In the previous iteration of predictive sampling, input $\mathbf{x}_{\leq i-1}$ was valid, and therefore the representation $\mathbf{h}_{\leq i}$ is valid as well. Although it is possible to obtain an unconditional forecast for $i = 0$, we use a zero vector as initial forecast.

Conditioning on $\mathbf{h}$ should in theory have no effect on performance, as the data processing inequality states that no post-processing function can increase the information content. However, in practice $\mathbf{h}$ is a convenient representation that summarizes the input at no extra computational cost.

2.3. ARM Fixed-Point Iteration

The first forecasting method we introduce is ARM Fixed-Point Iteration (FPI), which utilizes the ARM itself as forecasting function. Specifically, a forecast $\tilde{x}_{i+t}$ at step $t$ is obtained using the ARM reparametrization $g$, where noise $\epsilon$ is isolated:

$$\tilde{x}_{i+t} = g_{i+t}(\mathbf{x}_{\leq i+t}, \epsilon) \text{ for } t = 0, 1, \ldots$$

(7)

Note that $\mathbf{x}_{\leq i+t}$ is a concatenation of the valid samples thus far $\mathbf{x}_{\leq i}$ and the forecasts $\mathbf{x}_{i:i+t-1}$ from the previous iteration of predictive sampling (as in Algorithm 1). In other words, current forecasts are obtained using ARM inputs that may turn out to be invalid. Nevertheless, the method is compelling because it is computationally inexpensive and requires no additional learned components: it simply reuses the ARM output.

Interestingly, the combination of forecasting with Equation 7 and Algorithm 1 is equivalent to a reformulation as a fixed-point iteration using the function $g$ defined over $\mathbf{x}$:

$$\mathbf{x}^{(n+1)} = g(\mathbf{x}^{(n)}, \epsilon),$$

(8)

where $n$ denotes the iteration number of predictive sampling. We show this reformulation in Algorithm 2. This equivalence follows because ARM outputs $\mathbf{x}'_i$ are fixed if their conditioning consists of samples $\mathbf{x}_{\leq i}$ that are valid, i.e., for $\mathbf{x}' = g(\mathbf{x}, \epsilon)$ the outputs $\mathbf{x}'_{\leq i}$ equal the inputs $\mathbf{x}_{\leq i}$. The future outputs are automatically used as forecasts. The algorithm is guaranteed to converge in $d$ steps because the system has strictly triangular dependence, and may converge much faster if variables do not depend strongly on adjacent previous variables.

2.4. Learned forecasting

ARM fixed-point iteration makes use of the fact that the ARM outputs distributions $P_{\text{ARM}}(x_i|x_{\leq i})$ for every location $i \in \{1, \ldots, d\}$. However, many output distributions are conditioned on forecasts $x_{i:i+t-1}$ from the previous iteration of predictive sampling, and these may turn out to be incorrect. For example, if in the first iteration of the algorithm we find that forecast $\tilde{x}_1$ does not match $x_1$, the procedure will still use the sampled $x'_2 \sim P_{\text{ARM}}(x_2|\tilde{h}_1(\tilde{x}_1))$ as input in the second iteration. In turn, this may result in an incorrect forecast $\tilde{x}_3$. In the worst case, this leads to cascading errors, and $d$ ARM inference calls are required.

To address this problem, we introduce learned forecasting, an addition to ARM fixed-point iteration. We construct forecasting modules: small neural networks that are trained to match the distribution $P_{\text{ARM}}(x_i|x_{\leq i})$. These networks may only utilize information that will be available during sampling. For that reason, they are only conditioned on the available valid information, $\mathbf{x}_{\leq i}$, $\mathbf{h}_{\leq i}$, and $\epsilon$.

In particular, a forecasting module at timestep $i + t$ outputs a distribution $P_{F}^{(t)}(\tilde{x}_{i+t}|x_{\leq i})$ that will be trained to match the ARM distribution at that location $P_{\text{ARM}}(x_{i+t}|x_{\leq i})$.

The important difference is that $P_F$ is conditioned only on $\mathbf{x}_{\leq i}$ and $\mathbf{h}_{\leq i}$, whereas the ARM output for that location is based on $\mathbf{x}_{\leq i+t}$. We minimize the KL divergence between corresponding distributions $P_{\text{ARM}}$ and $P_F$:

$$\sum_{i} \text{KL} \left[ P_{\text{ARM}}(x_{i+t}|x_{\leq i}) \parallel P_{F}^{(t)}(\tilde{x}_{i+t}|x_{\leq i}) \right],$$

(9)

with respect to the forecasting module $P_F^{(t)}$ for each future step $t$. The gradient path to the model $P_{\text{ARM}}$ is detached in this divergence.

After training, forecasts can be obtained via the forecasting distributions and reparametrization noise. For example, when $P_{\text{ARM}}$ and $P_F$ are categorical distributions:

$$\tilde{x}_{i+t} = \arg \max_c \left( \tilde{\mu}_{i+t,c}(x_{\leq i}) + \epsilon_{i,c} \right),$$

(10)

where $\tilde{\mu}_{i+t,c}(x_{\leq i}) = \log P_F(x_{i+t} = c|x_{\leq i})$ is the log-probability that $x_{i+t} = c$ according to the forecasting distribution. In practice, a sequence of forecasts is obtained by concatenating forecasting modules $F_{i,t}$, where $t = 0, \ldots, T - 1$ and $T$ is the window in which we forecast future values.

---

**Algorithm 2** ARM Fixed-Point iteration

**Input:** $g$, $\epsilon$

**Output:** $\mathbf{x}$

1. let $\mathbf{x}^{(0)} = 0$, $n = 0$
2. repeat
   - $\mathbf{x}^{(n+1)} = g(\mathbf{x}^{(n)}, \epsilon)$
   - $n = n + 1$
3. until $\mathbf{x}^{(n)} = \mathbf{x}^{(n-1)}$
We find that explicitly conditioning on $x_{<i}$ in combination with $h_{<i}$ does not result in a noticeable effect on performance for the forecasting module capacity we consider. Instead it suffices to solely condition on $h_{<i}$. The representation $h$ is shared and trained jointly for both the ARM and the forecasting modules, but the forecasting objective is down-weighed with a factor of 0.01 so that the final log-likelihood performance is not affected. While it is possible to train forecasting modules on samples from the model distribution, we only train on samples from the data distribution as the sampling process is relatively slow.

3. Related work

Neural network based likelihood methods in generative modelling can broadly be divided into VAEs (Kingma & Welling, 2014; Rezende et al., 2014), Flow based models (Dinh et al., 2017) and autoregressive models (Bengio & Bengio, 2000; Larochelle & Murray, 2011). VAEs and Flows are attractive when fast sampling is important, as they can be constructed without autoregressive components that need inverses. However, in terms of likelihood performance, ARMs currently outperform VAEs and Flows and hold state-of-the-art in image and audio domains (van den Oord et al., 2016b; Salimans et al., 2017; Chen et al., 2018; Child et al., 2019).

One of the earliest neural network architectures for autoregressive probability estimation of image data is NADE (Larochelle & Murray, 2011). This model employs a causal structure, i.e., nodes of the network are connected in such a way that layer output $h_i$ only depends on a set of inputs $x_{<i}$. Numerous follow up works by Germain et al. (2015); van den Oord et al. (2016b); Akoury & Nguyen (2017); Salimans et al. (2017); Menick & Kalchbrenner (2018); Sadeghi et al. (2019) improve on this idea, and increase likelihood performance by refining training objectives and improving network architectures.

There are various approaches that aim to capture the performance of the ARM while keeping sampling time low. The autoregressive dependencies can be broken between some of the dimensions, which allows some parts of the sampling to run in parallel, but comes at the cost of decreased likelihood performance (Reed et al., 2017). It is possible to train a student network using distillation (van den Oord et al., 2018), but in this case samples from the student network will not come from the (teacher) model distribution.

An alternative method that does preserve the model structure relies on caching of layer activations to avoid duplicate computation (Ramachandran et al., 2017). To apply this algorithm, the caching strategy must be specified in accordance with the ARM architecture. In addition, activations of the network must be stored during sampling, resulting in larger memory overhead. In contrast, our method does not require knowledge of model-specific details and does not require extra memory.

Finally, a method that predicts what a language model will output in order to save runtime has been proposed in (Stern et al., 2018). A key difference between this method and ours is that we sample from the model distribution instead of decoding greedily via argmax.

4. Experiments

Predictive sampling is evaluated in two settings. First, an ARM is trained on images, we refer to this task as explicit likelihood modeling. Second, an ARM is trained on the discrete latent space of an autoencoder.

The used datasets are Binary MNIST (Larochelle & Murray, 2011), SVHN (Netzer et al., 2011), CIFAR10 (Krizhevsky et al., 2009), and ImageNet32 (van den Oord et al., 2016b). We use the standard test split as test data, except for ImageNet32, for which no test split is available and we use the validation split as test data. As validation data, we use the last 5000 images of the train split for MNIST and CIFAR10, we randomly select 8527 images from the train split for SVHN, and we randomly select 20000 images from the train split for ImageNet32. For all datasets, the remainder of the train split is used as training data.

The ARM architecture is based on (Salimans et al., 2017), with the fully autoregressive categorical output distribution of (van den Oord et al., 2016b). The categorical output distribution allows us to scale to an arbitrary number of channels without substantial changes to the network architecture. The autoregressive order is a raster-scan order, and in each spatial location an output channel is dependent on all preceding input channels.

All experiments were performed using PyTorch version 1.1.0 (Paszke et al., 2019). Training took place on Nvidia Tesla V100 GPUs. To obtain sampling times, measurements were taken on a single Nvidia GTX 1080Ti GPU, with Nvidia driver 410.104, CUDA 10.0, and cuDNN v7.5.1. Sampling results are obtained without caching (Ramachandran et al., 2017). For a full list of hyperparameters and data preprocessing steps, see Appendix A.

4.1. Predictive sampling of image data

Setting In this section the performance of predictive sampling for explicit likelihood modelling tasks is tested on binary MNIST, SVHN and CIFAR10. We use the same architecture for all datasets but binary MNIST, for which we decrease the number of layers and filters to prevent overfitting. Each ARM is optimized using the log-likelihood objective and performance is reported in bits per dimension (bpd), which is the negative log-likelihood in base two di-
Table 1. Performance of predictive sampling for ARMs trained on explicit likelihood modeling tasks, in terms of percentage of forward passes with respect to the original sampling procedure, and total time to sample. All reported times are based on own implementation. Reported means and (Bessel-corrected) standard deviations are based on sampling of 10 batches with random seeds \{0, \ldots, 9\}.

<table>
<thead>
<tr>
<th></th>
<th>Batch size 1</th>
<th>Batch size 32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARM calls</td>
<td>Time (s)</td>
</tr>
<tr>
<td>MNIST (1 bit)</td>
<td>Baseline</td>
<td>100.0% ± 0.0</td>
</tr>
<tr>
<td></td>
<td>Forecast zeros</td>
<td>14.5% ± 5.0</td>
</tr>
<tr>
<td></td>
<td>Predict last</td>
<td>7.8% ± 1.5</td>
</tr>
<tr>
<td></td>
<td>Fixed-point iteration</td>
<td>3.3% ± 0.9</td>
</tr>
<tr>
<td></td>
<td>+ Forecasting (T = 20)</td>
<td>3.3% ± 0.6</td>
</tr>
<tr>
<td>SVHN (8 bit)</td>
<td>Baseline</td>
<td>100.0% ± 0.0</td>
</tr>
<tr>
<td></td>
<td>Fixed-point iteration</td>
<td>22.0% ± 1.2</td>
</tr>
<tr>
<td></td>
<td>+ Forecasting (T = 1)</td>
<td>36.9% ± 2.7</td>
</tr>
<tr>
<td>CIFAR10 (5 bit)</td>
<td>Baseline</td>
<td>100.0% ± 0.0</td>
</tr>
<tr>
<td></td>
<td>Fixed-point iteration</td>
<td>15.6% ± 2.1</td>
</tr>
<tr>
<td></td>
<td>+ Forecasting (T = 1)</td>
<td>23.2% ± 2.9</td>
</tr>
<tr>
<td>CIFAR10 (8 bit)</td>
<td>Baseline</td>
<td>100.0% ± 0.0</td>
</tr>
<tr>
<td></td>
<td>Fixed-point iteration</td>
<td>22.0% ± 2.0</td>
</tr>
<tr>
<td></td>
<td>+ Forecasting (T = 1)</td>
<td>43.1% ± 5.5</td>
</tr>
<tr>
<td></td>
<td>+ Forecasting (T = 5)</td>
<td>59.8% ± 2.9</td>
</tr>
</tbody>
</table>

(a) Samples from the model distribution \(x \sim P_{\text{ARM}}(\cdot)\).

(b) Forecasting mistakes by the forecasting modules.

(c) Forecasting mistakes by fixed-point iteration.

Figure 3. Samples from the 1-bit ARM. Forecasting mistakes are shown in red.

Table 2. Performance of ARMs on explicit likelihood modeling tasks, in terms of percentage of forward passes with respect to the original sampling procedure, and total time to sample. All reported times are based on own implementation. Reported means and (Bessel-corrected) standard deviations are based on sampling of 10 batches with random seeds \{0, \ldots, 9\}.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ARM calls</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST (1 bit)</td>
<td>Baseline</td>
<td>100.0% ± 0.0</td>
<td>16.6 ± 0.1</td>
</tr>
<tr>
<td>Forecast zeros</td>
<td>14.5% ± 5.0</td>
<td>2.4 ± 0.8</td>
<td>6.9×</td>
</tr>
<tr>
<td>Predict last</td>
<td>7.8% ± 1.5</td>
<td>1.5 ± 0.4</td>
<td>11.1×</td>
</tr>
<tr>
<td>Fixed-point iteration</td>
<td>3.3% ± 0.9</td>
<td>0.6 ± 0.1</td>
<td>27.6×</td>
</tr>
<tr>
<td>+ Forecasting (T = 20)</td>
<td>3.3% ± 0.6</td>
<td>0.7 ± 0.2</td>
<td>23.7×</td>
</tr>
<tr>
<td>SVHN (8 bit)</td>
<td>Baseline</td>
<td>100.0% ± 0.0</td>
<td>145.7 ± 0.8</td>
</tr>
<tr>
<td>Fixed-point iteration</td>
<td>22.0% ± 1.2</td>
<td>32.2 ± 1.7</td>
<td>4.5×</td>
</tr>
<tr>
<td>+ Forecasting (T = 1)</td>
<td>36.9% ± 2.7</td>
<td>57.3 ± 4.2</td>
<td>2.5×</td>
</tr>
<tr>
<td>CIFAR10 (5 bit)</td>
<td>Baseline</td>
<td>100.0% ± 0.0</td>
<td>148.2 ± 0.5</td>
</tr>
<tr>
<td>Fixed-point iteration</td>
<td>15.6% ± 2.1</td>
<td>23.3 ± 3.0</td>
<td>6.4×</td>
</tr>
<tr>
<td>+ Forecasting (T = 1)</td>
<td>23.2% ± 2.9</td>
<td>35.6 ± 4.3</td>
<td>4.2×</td>
</tr>
<tr>
<td>CIFAR10 (8 bit)</td>
<td>Baseline</td>
<td>100.0% ± 0.0</td>
<td>145.7 ± 0.8</td>
</tr>
<tr>
<td>Fixed-point iteration</td>
<td>22.0% ± 2.0</td>
<td>32.0 ± 2.9</td>
<td>4.6×</td>
</tr>
<tr>
<td>+ Forecasting (T = 1)</td>
<td>43.1% ± 5.5</td>
<td>65.1 ± 8.2</td>
<td>2.2×</td>
</tr>
<tr>
<td>+ Forecasting (T = 5)</td>
<td>59.8% ± 2.9</td>
<td>94.5 ± 4.4</td>
<td>1.5×</td>
</tr>
</tbody>
</table>

Performance Sampling performance for ARMs is presented in Table 1. For each dataset, we list the percentage of ARM calls with respect to the default sampling procedure, as well as the total runtime during sampling of a batch. Results are reported for batch sizes 1 and 32. In this implementation, the slowest image determines the number of ARM inference passes. We leave the implementation of a scheduling system to future work, which would allow sampling at an average rate equal to the batch size 1 setting.

Fixed-point iteration and learned forecasting greatly outperform the standard baseline on all datasets. To put the improvements in perspective, we introduce two additional baselines for binary MNIST: forecast zeros and predict last. The first baseline simply forecasts \(\hat{x}_{t+T} = 0\) for all future timesteps \(t\), and the second baseline repeats the last observed value \(\hat{x}_{t+T} = x_t\). On binary MNIST, both fixed-point iteration and learned forecasting outperform these baselines.
Comparing the sampling speed for 5-bit and 8-bit CIFAR, we observe that when data has a lower-bit depth, it is generally easier to predict future variables. This can likely be attributed to the lower number of categories. Typically SVHN is considered to be an easier dataset to model than CIFAR10, a claim which is also supported by the bpd of 1.81 for SVHN versus 3.05 for CIFAR10. Interestingly, Table 2. we find that SVHN is not necessarily easier in the case of predictive sampling. Comparing the ARM calls for SVHN and CIFAR10 when using fixed-point iteration, both models require approximately 22% of the ARM calls. This suggests that the performance of predictive sampling depends mostly on the number of categories and less on the modeling difficulty of the data. Furthermore, while forecasting seems to work well for binary MNIST, the results do not transfer to the more complicated datasets. For CIFAR10, we observe that increasing the number of forecasting modules decreases performance. Note also that for binary MNIST the runtime overhead of the forecasting modules negates the effect of the reduced number of ARM inference.

Note that results are obtained without caching (Ramachandran et al., 2017). Combining predictive sampling with caching has the potential to further reduce sampling time. For example, (Ramachandran et al., 2017) report a speedup of 2× for batch size 1 and 33× for batch size 32 for PixelCNN++ models trained on CIFAR10.

To aid quantitative analysis, model samples and corresponding forecasting mistakes are visualized in Figure 3 and 4. In these figures, red pixels highlight in which locations in the image the forecast was incorrect, for both forecasting modules and fixed-point iteration. As color images consist of three channels, mistakes are visualized using $\frac{1}{3}$, $\frac{2}{3}$ or $\frac{3}{3}$ red, depending on the number of channels that were predicted incorrectly. For binary MNIST samples (Figure 3) it is noticeable that forecasting mistakes do not only lie on the edge of the digits, and that transitions from digit to background pixel are often predicted correctly. This indicates that more sophisticated patterns are used than, for example, simply repeating the last observed value. For more complicated 5-bit CIFAR data (Figure 4) we observe more mistakes in the top row and on the right side of the images. An explanation for this may be that the ARM dependency structure is from left to right, and top to bottom. The left-most pixels are conditioned more strongly on pixels directly above, and these are generally further away in the sequence. Hence, even if the last pixel of the preceding row contains a wrong value, pixels in the left-most column can be predicted with high accuracy.

### 4.2. Predictive sampling of latent variables

**Setting** In this section we explore autoencoders with a probabilistic latent space (Theis et al., 2017; Ballé et al., 2017). Typically these methods weigh a distortion component $\ell$ and a rate component $\log P(z)$:

$$\left|\ell(x, G(z)) - \beta \log P(z)\right|_{z=Q(x)}$$

where $Q$ is an encoder, $G$ is a decoder and $\beta$ is a tunable parameter. In our experiments we use the Mean Squared Error (MSE) as distortion metric and set $\beta = 0.1$. Following (van den Oord et al., 2017; Habibian et al., 2019; Razavi et al., 2019) we model the latent distribution $P(z)$ using an ARM. The (deterministic) encoder $Q$ has an architecture consisting of two $3 \times 3$ convolutional layers, two strided convolutions and two residual blocks, following PyTorch BasicBlock implementation (He et al., 2016). The decoder $G$ mirrors this architecture with two residual blocks, two transposed convolutions and two standard convolutional layers. The latent space is quantized using an argmax of a softmax, where the gradient is obtained using the straight-through estimator. We use a latent space of 4 channels, with height and width equal to 8, and 128 categories per latent variable. Further details on the architecture are given in Appendix A.
Predictive Sampling with Forecasting Autoregressive Models

Figure 5. Samples from the VAE and corresponding forecasting mistakes for a $4 \times 8 \times 8$ latent space.

Following van den Oord et al. (2017), we separate the training of autoencoder and ARM. We first train the discrete autoencoder for 50000 iterations, then freeze its weights, and train an ARM on the latents generated by the encoder for another 150000 iterations. We find that this scheme results in more stability than joint training. The obtained MSE is 0.0129 for Imagenet32, 0.0065 for CIFAR10, and 0.0008 for SVHN. The obtained bits per image dimension are 0.223 for Imagenet32, 0.244 for CIFAR10, and 0.191 for SVHN (To obtain the bits per latent dimension, multiply these by the dimensionality reduction factor 12). Note that the prior likelihood depends on the latent variables produced by the encoder, and cannot be compared directly with results from explicit likelihood modeling.

Performance The sampling performance for PixelCNN trained on the latent space of the discrete-latent autoencoder is presented in Table 2. Similar to the explicit likelihood modeling setting, predictive sampling with fixed-point iteration and learned forecasting modules both outperform the baseline, and fixed-point iteration outperforms learned forecasting across all three datasets.

Samples and predictive sampling mistakes of forecasting methods are depicted in Figure 5 for an autoencoder trained on CIFAR10 (8 bit). Samples $z \sim P(z)$ are generated in the latent representation and subsequently $\hat{x} = G(z)$ is visualized. In addition, the latent representation is visualized on a scale from black to red, where the amount of red indicates the number of mistakes at that location, averaged over the channel dimension. The latent representation has an $8 \times 8$ resolution and is resized to match the $32 \times 32$ images.

Finally, the convergence behavior of fixed-point iteration is visualized in Figure 6. In this figure, the color indicates the iteration of sampling from which the variable remained the same, i.e., the iteration at which that variable converged. For example, because there is strict triangular dependence and the top-left variable in the first channel is at the beginning of the sequence, this variable will converge at step one.

The converging iterations are averaged over channels and a batch of 32 images. The right image of Figure 6 shows the baseline, where the total number of iterations is exactly equal to the number of dimensions. The left image shows the convergence of the ARM fixed-point iteration procedure, which needs 53 iterations on average for this batch of data. We observe that pixels on the left of the image tend to converge earlier than those on the right. This matches the conditioning structure of the ARM, where values in the leftmost column depend strongly on pixel values directly above, and right-most variables also depend on pixels to their left.

4.3. Ablations

We perform ablations on 8 bit CIFAR10 data to show the effect of the isolation of stochasticity via reparametrization, and the sharing of the ARM representation. First, to quantify the effect of reparametrization, the sampling procedure is run again for an ARM without learned forecasting modules. As forecast, the most likely value according the forecasting distribution $P_F$ is used. For categorical distributions, this is done by removing the $\epsilon_{i,c}$ term from Equation 10.

In addition, we show the importance of sharing the ARM representation by training forecasting modules conditioned
only on $x_{<i}$ and reparametrization noise $\epsilon$, i.e., Equation 6 where $h$ is removed. Results are shown in Table 3. These experiments indicate that the both reparametrization and the shared representation improve performance considerably, with reparametrization having the biggest effect.

5. Conclusion

We introduce predictive sampling, an algorithm speeds up sampling for autoregressive models (ARMs), while keeping the model intact. The algorithm aims to forecast likely future values and exploits the parallel inference property of neural network based ARMs. We propose two variations to obtain forecasts, namely ARM fixed-point iteration and learned forecasting modules. In both cases, the sampling procedure is reduced to a deterministic function by a reparametrization. We train ARMs on image data and on the latent space of a discrete autoencoder, and show in both settings that predictive sampling provides a considerable increase in sampling speed. ARM fixed-point iteration, a method that requires no training, obtains the best performance overall.

References


Gumbel, E. J. Statistical theory of extreme values and some practical applications. NBS Applied Mathematics Series, 33, 1954.


Predictive Sampling with Forecasting Autoregressive Models


