Supplementary Material for Predictive Sampling with Forecasting Autoregressive Models

Auke Wiggers [1] Emiel Hoogeboom [2 3]

A. Architecture and hyperparameters

A.1. Autoregressive model architecture

We base our PixelCNN implementation on a Pytorch implementation of PixelCNN++ (Salimans et al., 2017) by GitHub user pclucas14 (https://github.com/pclucas14/pixel-cnn-pp, commit 16c8b2f). We make the following modifications.

Instead of the discretized mixture of logistics loss as described in (Salimans et al., 2017), we utilize the categorical distributions as described in (van den Oord et al., 2016), which allows us to model distributions with full autoregressive dependence. This is particularly useful when training the PixelCNN for discrete-latent autoencoders, as the number of input channels can be altered without substantial changes to the implementation. We model dependencies between channels as in the PixelCNN architecture, by masking convolutions so that the causal structure is preserved. That is, the output corresponding to input $x_{c,h,w}$ is conditioned on all previous rows $x_{0,:,:}, \dots, x_{h-1,:,:}$, on all previous columns of the same row $x_{h,0,:}, \dots, x_{h,w-1,:}$ and all previous channels of the same spatial location $x_{h,w,0}, \dots, x_{h,w,c-1}$.

The original implementation normalizes the input data to a range between -1 and 1. Instead, we follow (van den Oord et al., 2016) and use a one-hot encoding for inputs. Additionally, we do not use weight normalization.

A.2. Forecasting module architecture

The forecasting module used in this work consists of a single strictly triangular 3×3 convolution followed by a 1×1 convolution, where the number of output channels is equal to the number of data channels multiplied by the number of categories. The masked convolution is applied to the last activation of the *up-left stack* of the PixelCNN, $\mathbf{h}$. We set the number of channels for this layer to 162 for image space experiments, and 160 for latent space experiments.

We experimented with variations of forecasting modules that use x (one-hot) or truncated Gumbel noise, obtained as described in Section B, as additional inputs. For the forecasting module capacity we considered, this did not lead to improved sampling performance.

[1]Qualcomm AI Research, Qualcomm Technologies Netherlands B.V.. Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc. [2]University of Amsterdam, Netherlands. [3]Research done while completing an internship at Qualcomm AI Research.. Correspondence to: Auke Wiggers <auke@qti.qualcomm.com>.

Proceedings of the 37th International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020. Copyright 2020 by the author(s).

Table 1. Hyperparameters for the trained PixelCNN models.

Hyperparameter	Binary MNIST	Default
Learning rate	0.0002	0.0002
Learning rate decay	0.999995	0.999995
Batch size	64	64
Max iterations	200000	200000
Weight decay	1e-6	1e-6
Optimizer	Adam	Adam
Number of gated resnets	2	5
Filters per layer	60	162
Dropout rate	0.5	0.5
Nonlinearity	concat_elu	concat_elu
Forecasting modules	20	1
Forecasting filters	60	162
Forecasting loss weight	0.01	0.01

A.3. Default hyperparameters

Explicit likelihood modeling Hyperparameter settings for the PixelCNNs trained on image data are given in Table 1. We use the same parameters across all datasets to maintain consistency, and did not alter the architecture for likelihood performance. The exception is binary MNIST, where we observed strong overfit if the size was not changed.

Latent space modeling For the latent space experiments, we use an encoder and decoder with bottleneck structure, and a PixelCNN to model the resulting latent space. The *width* of the encoder and decoder, *i.e.*, the parameter that controls the number of channels at every layer, is 512 for all experiments. The used loss function is Mean Squared Error, and the input data is normalized to the range $[-1, 1]$.

The encoder consists of the following layers. First, two 3×3 convolutional layers with padding 1 and half width. Then, one strided 4×4 convolution of half width with padding 1 and stride 2, followed by a similar layer of full width. We then apply two residual blocks (PyTorch BasicBlock implementation (He et al., 2016)). Finally, a 1×1 convolution layer maps to the desired number of latent channels.

The decoder architecture mirrors the encoder architecture. First, a 1×1 convolution layer maps from the (one-hot) latents to the desired width. Two residual blocks are applied, followed by a full width transpose convolution and a half width transpose convolution, both having the same parameters as their counterparts in the encoder. Lastly, two 3×3 convolution layers of half width are applied, where the last layer has three output channels.

The latent space is quantized by taking the argmax over a softmax, and one-hot encoding the resulting latent variable. As quantization is non-differentiable, the gradient is obtained using a straight-through estimator, *i.e.*, the softmax gradient is used in the backward pass. We use a latent space of 4 channels, with height and width equal to 8, and 128 categories per latent variable.

Optimization parameters and the parameters of the PixelCNN that is used to model the latent space are kept the same as in the explicit likelihood setting, see Table 1. We do not train the autoencoder and ARM jointly. Instead, we train an autoencoder for 50000 iterations, then freeze the autoencoder weights and train an ARM on the latent space for an additional 200000 iterations.

A.4. Infrastructure

Software used includes Pytorch (Paszke et al., 2019) version 1.1.0, CUDA 10.0, cuDNN 7.5.1. All sampling time measurements were obtained on a single Nvidia 1080Ti GPU using CUDA events, and we only compute runtime after calling *torch.cuda.synchronize*. Training was performed on Nvidia TeslaV100 GPUs, with the same software stack as the evaluation system.

B. Posterior Reparametrization Noise

To condition forecasting modules on reparametrization noise when training on the data distribution, sample noise pairs $(\mathbf{x}, \boldsymbol{\epsilon})$ are needed. In principle, these can be created by sampling $\boldsymbol{\epsilon}$ and computing the corresponding $\mathbf{x}$ using the ARM, *i.e.*, by computing the autoregressive inverse. However, this process may be slow, and does not allow for joint training of the ARM and forecasting module. Alternatively, one can use the assumption that the model distribution $\mathrm{P}_{\mathrm{ARM}}(\mathbf{x})$ will sufficiently approximate $\mathrm{P}_{\mathrm{data}}(\mathbf{x})$. In this case, $(\mathbf{x}, \boldsymbol{\epsilon})$ pairs can be sampled using the data distribution $\mathrm{P}_{\mathrm{data}}(\mathbf{x})$, and the posterior of the noise $p(\boldsymbol{\epsilon}|\mathbf{x})$:

$$\mathbf{x}, \boldsymbol{\epsilon} \sim \mathrm{P}_{\mathrm{ARM}}(\mathbf{x}|\boldsymbol{\epsilon})p(\boldsymbol{\epsilon}) \approx p(\boldsymbol{\epsilon}|\mathbf{x})\mathrm{P}_{\mathrm{data}}(\mathbf{x}), \tag{1}$$

where $\mathrm{P}_{\mathrm{ARM}}(\mathbf{x}|\boldsymbol{\epsilon})$ is a Dirac delta peak on the output of the reparametrization, and $p(\boldsymbol{\epsilon}|\mathbf{x})$ denotes the posterior of the noise given a sample $\mathbf{x}$:

$$p(\boldsymbol{\epsilon}|\mathbf{x}) = \frac{\mathrm{P}_{\mathrm{ARM}}(\mathbf{x}|\boldsymbol{\epsilon})p(\boldsymbol{\epsilon})}{\mathrm{P}_{\mathrm{ARM}}(\mathbf{x})}. \tag{2}$$

In the case of the Gumbel-Max reparametrization, the posterior Gumbel noise $p(\boldsymbol{\epsilon}|\mathbf{x})$ can be computed straightforwardly by using the notion that the maximum and the location of the maximum are independent (Maddison et al., 2014; Kool et al., 2019). First, we sample from the Gumbel distribution for the arg max locations, *i.e.*, the locations that resulted in the sample $\mathbf{x}$:

$$\epsilon_{i,x_i} \sim G. \tag{3}$$

Subsequently, the remaining values can be sampled using truncated Gumbel distributions (TG) (Maddison et al., 2014; Kool et al., 2019). The truncation point is located at the maximum value $\mu_{i,x_i} + \epsilon_{i,x_i}$:

$$\epsilon_{i,c} \sim TG(\mu_{i,c}|\mu_{i,x_i} + \epsilon_{i,x_i}) - \mu_{i,c} \quad \text{for all} \quad c \neq x_i. \tag{4}$$

Here, $\mu_{i,c}$ denotes the logit from the model distribution $\mathrm{P}_{\mathrm{ARM}}(\mathbf{x})$ of dimension i for category c.

To summarize, a sample $(\mathbf{x}, \boldsymbol{\epsilon})$ is created by first sampling $\mathbf{x} \sim \mathrm{P}_{\mathrm{data}}$ from data, and then sampling $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}|\mathbf{x})$ using the Gumbel and Truncated Gumbel distributions as described above. This technique allows simultaneous training of the ARM and forecasting module conditioned on Gumbel noise without the need to create a dataset of samples.

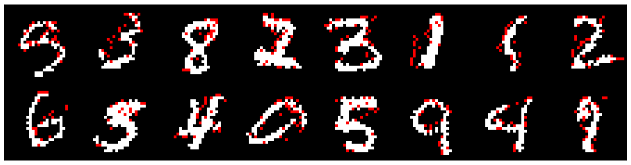
C. Generated samples

We show 16 samples for each of the models trained with forecasting modules, as well as forecasting mistakes. To find forecasting mistakes made by ARM fixed-point iteration, we simply disable the forecasting modules during sampling. All samples were generated using the same random seed (10) and were not cherry-picked for perceptual quality or sampling performance. 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., 2016).

Samples generated by the model trained on binary MNIST are shown in Figure 1, Figure 2 shows SVHN 8-bit samples, and Figures 3 and 4 show samples generated by the ARM trained on CIFAR10 for 5-bit and 8-bit data, repsectively.

Samples from the VAE are generated by decoding a sample from the ARM trained on the latent space. That is, we

(a) Samples from the model distribution $\mathbf{x} \sim \mathrm{P}_{\mathrm{ARM}}(\,\cdot\,)$.


(b) Forecasting mistakes by the forecasting modules.

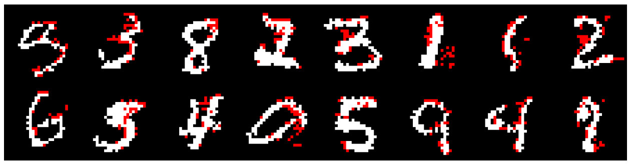
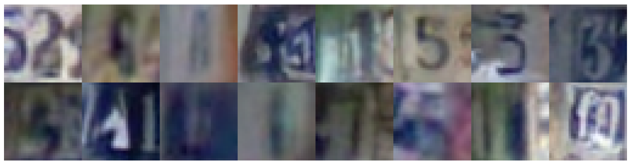
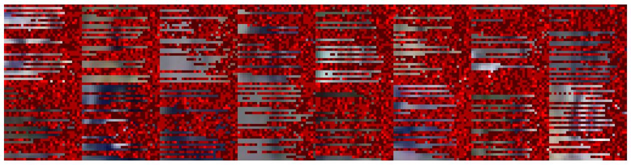

(c) Forecasting mistakes by fixed-point iteration.

Figure 1. Samples from the 1-bit ARM and forecasting mistakes.


(a) Samples from the model distribution $\mathbf{x} \sim \mathrm{P}_{\mathrm{ARM}}(\,\cdot\,)$.


(b) Forecasting mistakes by the forecasting modules.

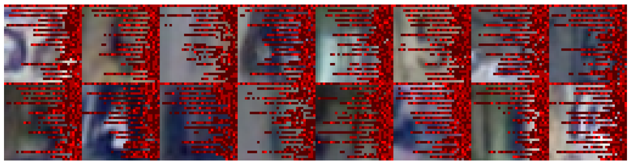
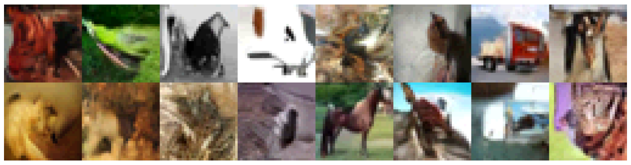
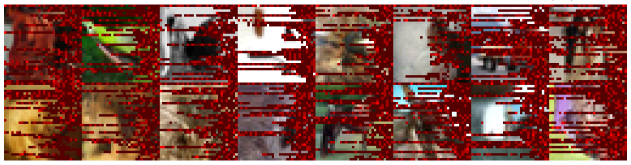

(c) Forecasting mistakes by fixed-point iteration.

Figure 2. Samples from the 8-bit ARM and forecasting mistakes.

first generate a latent variable from the trained ARM $z \sim P(z)$. This sample is then decoded to image-space using the decoder G as $\hat{x} = G(z)$. We show samples for SVHN in Figure 5, for CIFAR10 in Figure 6, and for Imagenet32 in Figure 7.


(a) Samples from the model distribution $\mathbf{x} \sim \mathrm{P}_{\mathrm{ARM}}(\,\cdot\,)$.


(b) Forecasting mistakes by the forecasting modules.

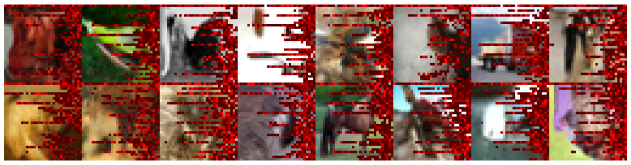
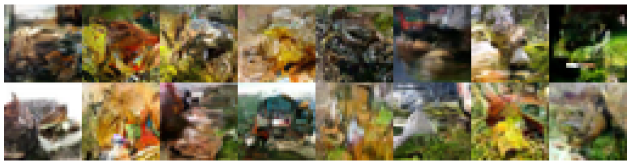
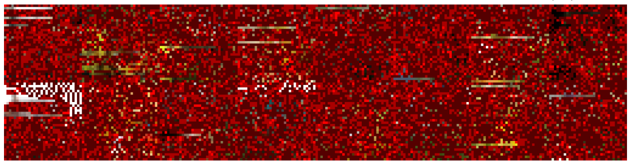

(c) Forecasting mistakes by fixed-point iteration.

Figure 3. Samples from the 5-bit ARM and forecasting mistakes.


(a) Samples from the model distribution $\mathbf{x} \sim \mathrm{P}_{\mathrm{ARM}}(\,\cdot\,)$.


(b) Forecasting mistakes by the forecasting modules.

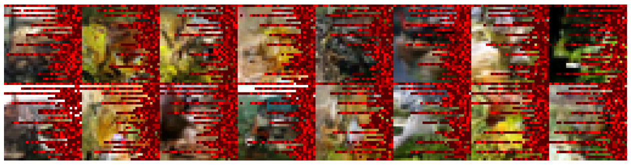
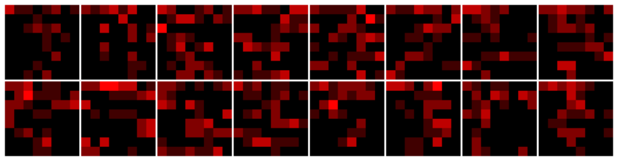

(c) Forecasting mistakes by fixed-point iteration.

Figure 4. Samples from the 8-bit ARM and forecasting mistakes.

(a) Decoded samples $G(\mathbf{z})$, where $\mathbf{z} \sim \mathrm{P}(\mathbf{z})$.


(b) Forecasting mistakes by learned forecasting modules.

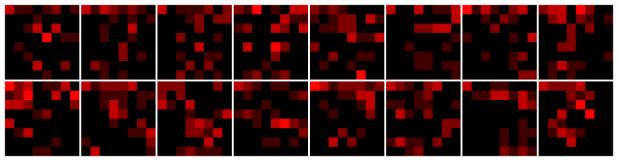
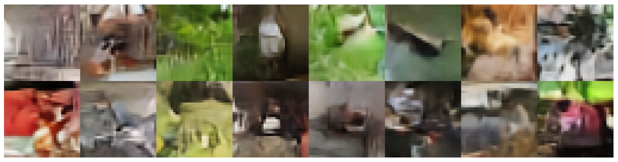
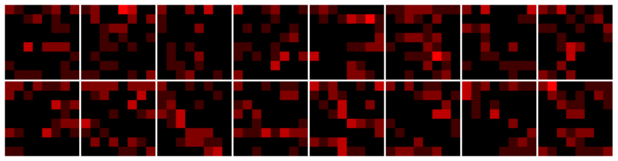

(c) Forecasting mistakes by fixed-point iteration.

Figure 5. VAE samples, and forecasting mistakes in latent space.


(a) Decoded samples $G(\mathbf{z})$, where $\mathbf{z} \sim \mathrm{P}(\mathbf{z})$.


(b) Forecasting mistakes by learned forecasting modules.

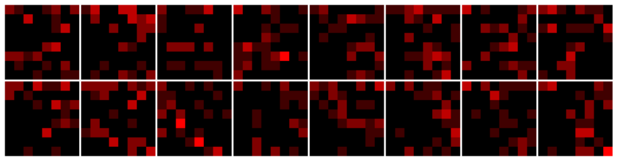
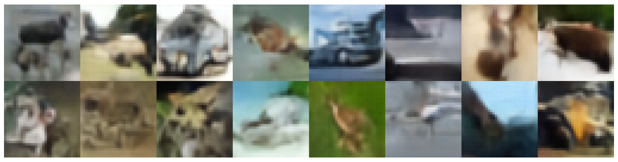
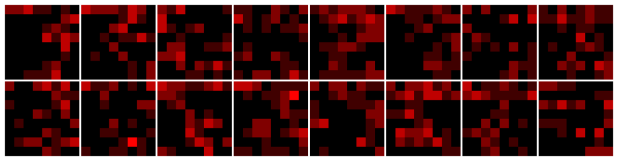

(c) Forecasting mistakes by fixed-point iteration.

Figure 7. VAE samples, and forecasting mistakes in latent space.


(a) Decoded samples $G(\mathbf{z})$, where $\mathbf{z} \sim \mathrm{P}(\mathbf{z})$.


(b) Forecasting mistakes by learned forecasting modules.

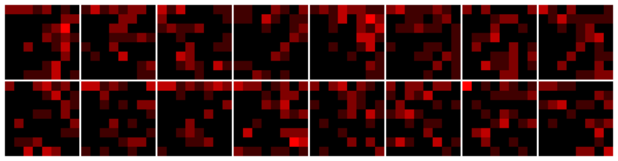

(c) Forecasting mistakes by fixed-point iteration.

Figure 6. VAE samples, and forecasting mistakes in latent space.

References

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Kool, W., van Hoof, H., and Welling, M. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. In *Proceedings of the 36th International Conference on Machine Learning, ICML*, pp. 3499–3508, 2019.

Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.

Larochelle, H. and Murray, I. The neural autoregressive distribution estimator. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 29–37, 2011.

Maddison, C. J., Tarlow, D., and Minka, T. A* sampling. In *Advances in Neural Information Processing Systems*, pp. 3086–3094, 2014.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning. 2011.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d' Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Salimans, T., Karpathy, A., Chen, X., and Kingma, D. P. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

van den Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.