C2FAR: Coarse-to-Fine Autoregressive Networks for Precise Probabilistic Forecasting: Supplementary Materials

Shane Bergsma **Timothy Zeyl** **Javad Rahimipour Anaraki** **Lei Guo**
Huawei Cloud, Alkaid Lab Canada
`{shane.bergsma,timothy.zeyl,javad.anaraki,leiguo}@huawei.com`

A Experimental details

A.1 Architecture

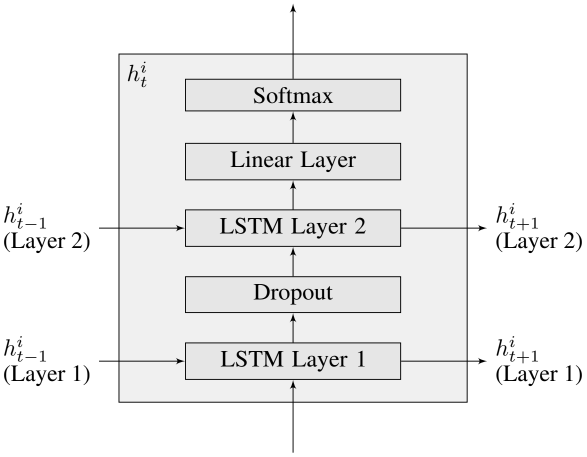


Figure 1: LSTM architecture for h_t^i: one level of C2FAR at time t and level i (see Fig. 4 in main paper). Network layers do include *bias weights*. The same number of hidden units are used in each LSTM layer and is given by hyperparameter *n_hidden*. Dropout probability is given by hyperparameter *lstm_dropout*.

Regarding the sequence model, all C2FAR-RNN models use 2-layer LSTMs [10] with intra-layer dropout [19, 12], as depicted in Fig. 1. Multi-level C2FAR-RNN models use the same LSTM architecture at each level, with the same number of hidden units in each LSTM layer of each level. We follow DeepAR [18] in using the same network to encode (i.e., process the conditioning range) and decode (i.e., generate values in the prediction range). However, unlike DeepAR, during training we only compute loss over the prediction range.

For generating the parameters of the Pareto distribution, we use a feed-forward neural network with a single hidden layer (followed by a *softplus* output transformation). The number of units in this hidden layer is also controlled by the *n_hidden* hyperparameter.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

A.2 Features and input/output preparation

Toward our goal of universal forecasting (§4.3 of the main paper), we exclude covariates for series-specific meta data, series "age", and lagged historical values (all of which are used in [18]). We use min-max scaling [14] to normalize conditioning ranges prior to forecasting. Autoregressive inputs are represented with 1-hot-encodings.

A.3 Training and testing details

We use Optuna [1] for tuning, with the TPESampler [4], and use both MedianPruning (pruning unpromising trials compared to previous trials) and early stopping (stopping trials when results no longer improve). We stop early if we see *n_stop_evals_no_improve* evaluations without a new top score (currently set to 37, see Table 5).

We filter instances with constant *conditioning* ranges from training and testing. We do not oversample training instances from the higher-amplitude series, as DeepAR does [18].

We vectorize across multiple series during both the computing of likelihood (in training) and during the sampling of future values (in prediction). The number of series that we parallelize over is referred to as our "batch size" (e.g. *n_train_batch_size*).

We also vectorize across different multi-step-ahead rollouts during the Monte Carlo procedure to generate the forecast distribution. We use 500 separate rollouts during the forecasting process (*n_rollouts* = 500), unless otherwise stated. We compute rolling evaluations with a stride of 1, i.e., we forecast and evaluate over overlapping prediction ranges, as in [9].

A.4 Computational resources

C2FAR is implemented in `PyTorch` [13], version `1.9.1+cu102`. We use GPUs from Nvidia: four Tesla P100 GPUs with 16280MiB and two Tesla K80 GPUs with 11441MiB. To maximize the utilization of the GPUs, we usually ran two trials in parallel on each of the six GPUs, for 12 trials running in parallel in total at any one time for a given tuning study.

B Distribution recovery

Here we expand on the results in §5.1 of the paper, evaluating each of our our implemented systems on the task of recovering synthetic distributions.

B.1 Training details

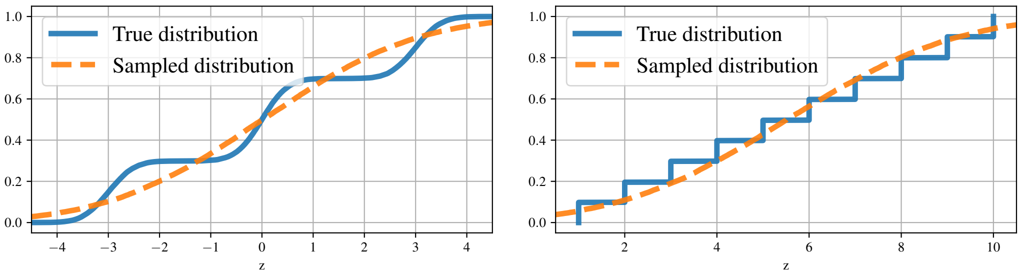
Rather than tuning the models for this (simple) task, we use a fixed learning rate of 2e-2, a weight decay of 1e-6, and 64 hidden units, which are settings that worked well during development experiments on the *elec* validation set. We also used an *lstm_dropout* of 1e-3 and training batch size of 1024 (as in Table 5). The (normalized) binning extent is taken from -0.01 to 1.01. We use a conditioning range of 96 and a prediction range of 24, following [8]. We take the final four days as the testing period.

B.2 Computational performance

The time to execute the training runs varied between 53 minutes and 84 minutes for all systems on the synthetic Gaussian Mixture Model (GMM) data, and between 10 minutes and 30 minutes for all systems on the synthetic discrete data.

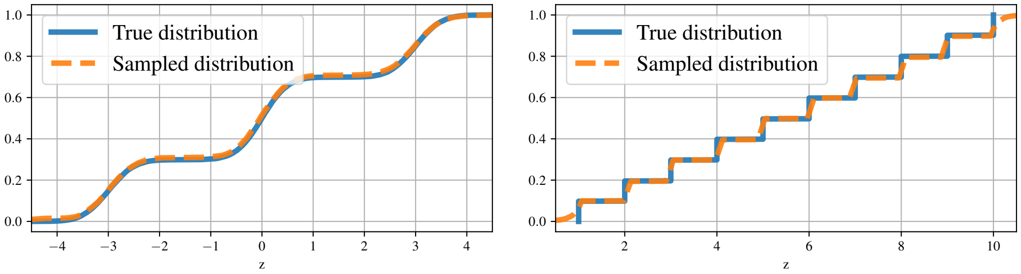
B.3 Further results

Fig. 2 shows the ability of each of our implemented systems to recover the synthetic distribution (repeating the C2FAR-RNN$_3$ results, also given in Fig. 5 in the main paper). The frequently-used Gaussian output distribution [11, 18, 5] cannot recover the Gaussian mixture since it has only a single mixture component, while it fits the discrete data as well as can be expected. For the C2FAR-RNN models, we use 60 total *bins* across all the levels, which amounts to much greater precision for the



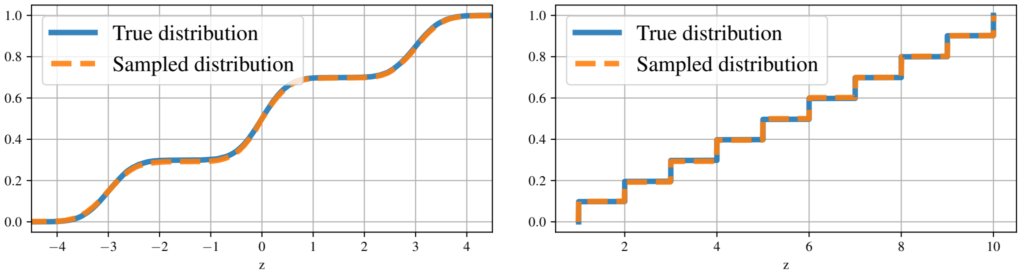
(a) GMM, DeepAR-Gaussian

(b) Discrete, DeepAR-Gaussian



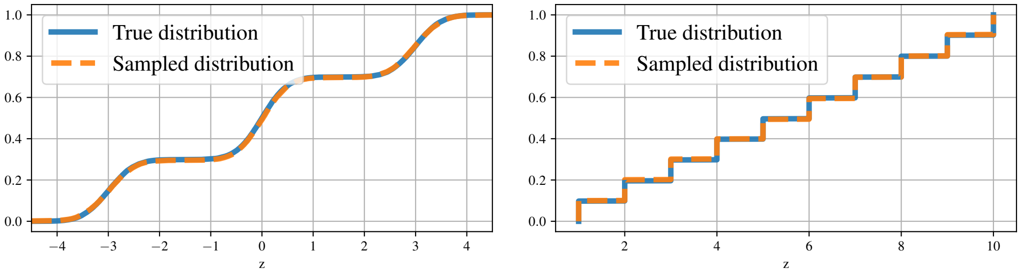
(c) GMM, C2FAR-RNN$_1$ with 60 bins

(d) Discrete, C2FAR-RNN$_1$ with 60 bins



(e) GMM, C2FAR-RNN$_2$ with $30 + 30$ bins

(f) Discrete, C2FAR-RNN$_2$ with $30 + 30$ bins



(g) GMM, C2FAR-RNN$_3$ with $20 + 20 + 20$ bins

(h) Discrete, C2FAR-RNN$_3$ with $20 + 20 + 20$ bins

Figure 2: Distribution recovery supplemental results on the Gaussian mixture model synthetic data (GMM, left) and the discrete uniform synthetic data (Discrete, right) for each of our implemented systems.

Table 1: Negative log likelihood (NLL) of test data for different models on synthetic data

	GMM	Discrete
DeepAR-Gaussian	0.2526	0.2775
C2FAR-RNN$_1$	-0.4150	-1.564
C2FAR-RNN$_2$	-0.4198	-4.479
C2FAR-RNN$_3$	-0.4203	-6.664

Table 2: Information about the *azure* dataset. Flavor *names* are automatically generated from each VM's allocated *VCPUs* and *Memory* requirement via the template *az-{VCPUs}-{Memory/VCPUs}*. Flavor *types* are defined purely by the Memory:VCPUs ratio via the template *az-{Memory/VCPUs}*.

Duration	30 days
Num. unique VMs	2,013,767
Num. unique subscriptions	5,958, with top 250 used for customer-specific series.
Unique VM categories	{Delay-insensitive, Interactive, Unknown}
Unique flavor names	{az-1-1.75, az-1-2, az-16-7, az-2-1.75, az-2-2, az-2-7, az-2-8, az-4-1.75, az-4-2, az-4-7, az-4-8, az-8-1.75, az-8-2, az-8-7, az-8-8}
Unique flavor types	{az-0.75, az-1.75, az-2, az-7, az-8}

multi-level models. Qualitatively, C2FAR-RNN$_1$ fits the data fairly well, not quite overlapping the true GMM distribution, and struggling to generate the sharp increases in the CDF seen in the discrete data. The C2FAR-RNN$_2$ and C2FAR-RNN$_3$ models fit each distribution nearly perfectly.

Quantitatively, we can evaluate each of these fits by computing the negative log likelihood (NLL) of the test data, as computed by each model. These results are presented in Table 1, which shows C2FAR-RNN$_3$ is only very slightly better on GMM, but significantly better on the discrete data. Of course, by adding more layers and bins, we may achieve arbitrarily low NLL on the discrete dataset. Whether such precision is beneficial will depend on the application.

C Empirical study on real-world data: further details

C.1 Azure VM demand dataset

We generated the *azure* dataset in order to provide real-world data reflecting the types of time series seen in the context of large-scale compute clouds. Cloud decision making can benefit from predicted future resource demand, e.g., for capacity planning or scheduling optimization [3]. A dominant workload type in compute clouds is the virtual machine (VM), which is typically available in one of a limited number of specific configurations or *flavors*; essentially, a flavor represents a specific bundle of resource requirements, including VCPU and Memory needs. Forecasts for the total demand (in terms of VCPUs or Memory, in GB) of specific flavors, customers, and workload categories are all valuable. Furthermore, aggregations of these basic time series are also valuable. For example, VM flavors with a common VCPUs:Memory ratio, known as a VM flavor *type*, are often run on shared physical servers, and therefore forecasts of flavor type demand are directly actionable in terms of provisioning of server resources.

We obtained real-world data reflecting these dimensions of cloud resource demand by leveraging the publicly-available *Azure Public Dataset*[1], originally released in [7] under a Creative Commons Attribution 4.0 International Public License. This dataset contains create, delete, and CPU utilization information (as a % of allocated VCPUs, over time) for over 2 million cloud VMs, reflecting both internal and external customer workload over a 30-day period. We converted this data into time series by counting the total VCPUs and Memory requirements over time for different combinations of flavors, subscriptions (customer IDs), categories, and flavor types, as noted in Table 2. Although we count VMs from all customers, we build customer-specific time series for the top 250 subscription IDs by VM frequency. We also combined lower-level time series to form a hierarchy of time series, as is common in real-world demand data [20, 15]. We also defined a *stopped* VM as any VM whose

---

[1] `https://github.com/Azure/AzurePublicDataset/blob/master/AzurePublicDatasetV1.md`

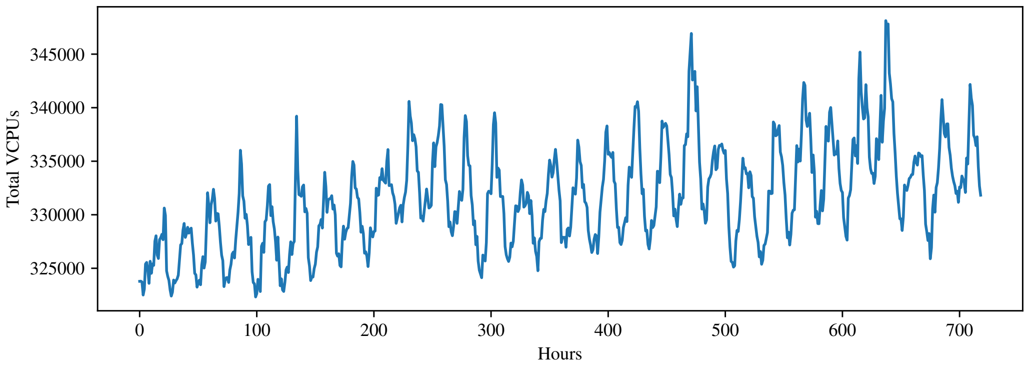


Figure 3: *azure* sample time series: total *aggregate* VCPUs demand. A strong daily seasonality is apparent.

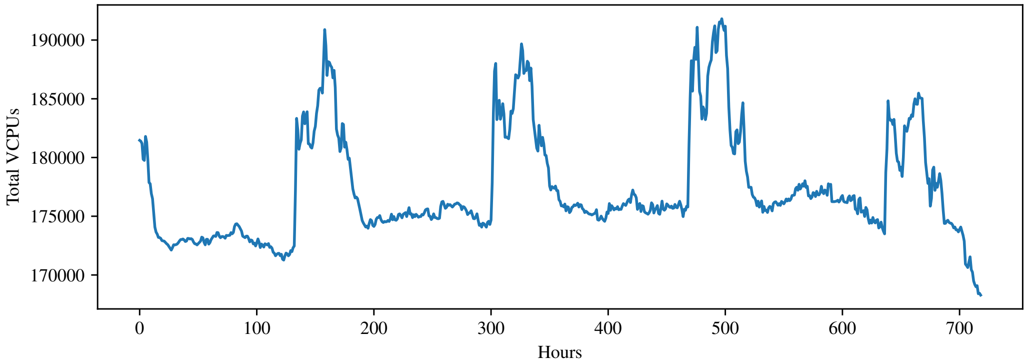


Figure 4: *azure* sample time series: total VCPUs demand of *delay-insensitive* VMs of flavor type *az-1.75*. *Delay-insensitive* VMs exhibit strong weekly seasonality, apparently being used mostly on weekends.

CPU utilization% drops below 1%. We then created two different versions of our time series, in each case aggregating the data with a 1-hour sampling period:

1. The time series aggregated with the *maximum* of each 1-hour period.
2. The time series aggregated with the *minimum* of each 1-hour period, and excluding stopped VMs at each time point.

In a way, the first set of time series represents a *pessimistic* view of how many resources we require each hour, while the second version represents an *optimistic* view, as we only provision for the minimum and assume we can re-use the resources from stopped VMs.

We provide some examples of these time series in Figs. 3 through 6. Fig. 3 represents the total demand across all VMs, and we can see the strong daily seasonality. Demand of the *delay-insensitive* category, for a particular flavor type, is shown in Fig. 4, while Fig. 5 gives the demand for the same category and flavor type, but for customer 1108 specifically (note all the original IDs are anonymized and represented in the dataset using placeholder values). Demand of the *unknown* category, for flavor name az-8-1.75, is given in Fig. 6.

Although the *azure* dataset covers 30 days, for experiments we use 20 days as training, 3 days for validation, and 3 final days for testing, leaving the remaining few days unseen and available for future experiments.

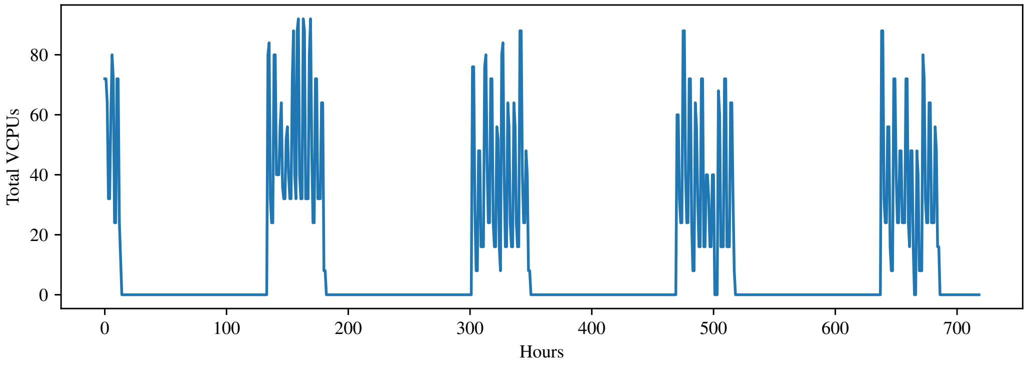


Figure 5: *azure* sample time series: customer 1108, total VCPUs demand of *delay-insensitive* VMs of flavor type az-1.75. This particular customer seems to consume resources exclusively on weekends.

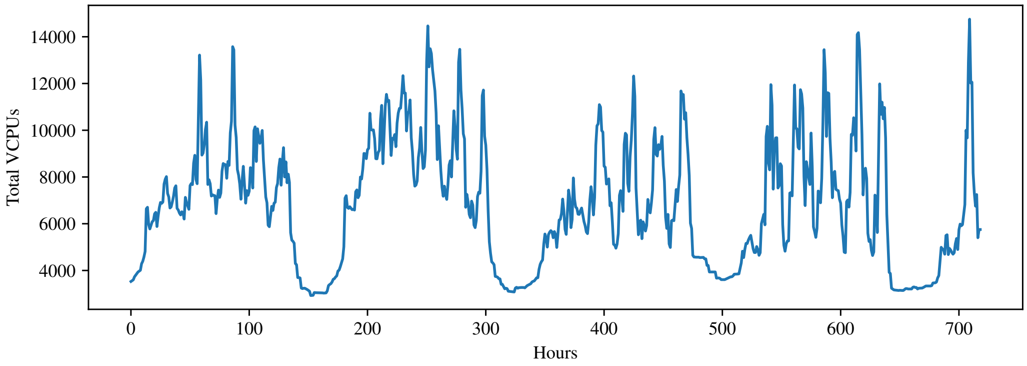


Figure 6: *azure* sample time series: total VCPUs demand of *unknown* VMs of flavor name az-8-1.75. *Unknown* VMs exhibit strong daily and weekly seasonality, requiring resources during the periods where *delay-insensitive* VMs are lower (i.e., during weekdays).

Table 3: Dataset details for the empirical study.

Dataset	Binning extent start	Binning extent end	Num. series	Domain	Freq.	Num. vals per series	Num. validation vals per series	Num. test vals per series	Prediction range size
elec	-0.01	1.01	321	Discrete	Hourly	21212	168	168	24
traff	-0.01	1.01	862	Real	Hourly	14204	168	168	24
wiki	-0.2	3.1	9535	Discrete	Daily	912	50	150	30
azure	-0.1	1.3	4098	Discrete	Hourly	719	72	72	24

C.2 Other datasets and dataset details

Beyond *azure*, the *elec*, *traff*, and *wiki* datasets were obtained using scripts in GluonTS [2], with the objective of replicating the training/validation/test splits used in prior work [17, 14, 9]. Table 3 provides the details of these datasets and *azure*. Note the binning extent was selected in order to cover from roughly the 1% to the 99% percentiles of normalized values in the training data for each series (normalized using min-max scaling on the *conditioning* range; the prediction range can go below the min and above the max).

C.3 Metrics

Let i index the time series, and t index the time step. Let N be the total number of points $z_{i,t}$ on which we evaluate. Let $\mathcal{I}(\cdot)$ denote the indicator function. Let $\hat{z}_{i,t}^{(q_\#)}$ be the $q_\#$ quantile of the forecast distribution for time series i at point t, e.g. $\hat{z}_{i,t}^{(q_{0.8})}$ is the value such that 80% of possible values for point $z_{i,t}$ are expected to be below this value.

We define *pinball loss* and *quantile loss* as part of the derivation of *weighted quantile loss*. *Weighted quantile loss*, *normalized deviation*, *calibration*, and *sharpness* are reported in the main paper.

Pinball loss:

$$\Lambda_\alpha(q, z) = (\alpha - \mathcal{I}(z < q))(z - q)$$

Quantile loss:

$$\text{QL\#} = \frac{\sum_{i,t} 2\Lambda_{(q_\#)}(\hat{z}_{i,t}^{(q_\#)}, z_{i,t})}{\sum_{i,t} |z_{i,t}|}$$

Weighted quantile loss:

$$\text{wQL} = \frac{1}{9}(\text{QL0.1} + \text{QL0.2} + \cdots + \text{QL0.9})$$

Normalized deviation:

$$\text{ND} = \frac{\sum_{i,t} |z_{i,t} - \hat{z}_{i,t}|}{\sum_{i,t} |z_{i,t}|}$$

Calibration:

$$\text{calibration} = \frac{\sum_{i,t} \mathcal{I}(\hat{z}_{i,t}^{(q_l)} < z_{i,t} \leq \hat{z}_{i,t}^{(q_u)})}{N}$$

Sharpness:

$$\text{sharpness} = \frac{\sum_{i,t} |\hat{z}_{i,t}^{(q_u)} - \hat{z}_{i,t}^{(q_l)}|}{\sum_{i,t} |z_{i,t}|}$$

Where calibration and sharpness are reported together as the CovX metric. E.g., Cov80 gives the coverage and sharpness of the 80% prediction range where $q_l = q_{0.1}$ and $q_u = q_{0.9}$. Note that for all metrics we use 500 Monte Carlo samples in order to generate our forecast distribution, except for generating the Cov99 metrics in Table 1 of the main paper, for which we used 5000 samples, as the extreme percentiles are by definition more rare and require more samples for good estimation.

Table 4: Tuning ranges for hyperparameter optimization.

Hyperparameter	Range	Sampling type
n_hidden	[16, 288]	integer
learning_rate	[1e-5, 1e-1]	loguniform
weight_decay	[1e-7, 1e-2]	loguniform
n_bins, for C2FAR-RNN$_1$	[4, 1024]	integer
n_bins_level_i, ith level of C2FAR-RNN$_B$, $B > 1$	[4, 128]	integer

Table 5: Fixed hyperparameters used in the empirical study.

Hyperparameter	Value	Note
n_max_total_parms	1,000,000	Enforced via a cap on n_hidden
lstm_dropout	1e-3	Intra-layer dropout
n_lstm_layers	2	
n_conditioning_range-hourly	168	For *elec*, *traff*, *azure*, as in prior work
n_conditioning_range-daily	30	For *wiki*, as in prior work
n_rollouts (validation)	25	For computing ND on validation set
n_rollouts (test)	500	For evaluation on test set
n_train_batch_size	1024	Total num. prediction ranges per batch
n_train_ranges_per_checkpoint	32768	Total num. prediction ranges in one *checkpoint* (train loss reported)
n_validation_set	32768	Total num. prediction ranges per validation evaluation
n_max_checkpoints	750	Maximum num. checkpoints (sets of n_train_ranges_per_checkpoint)
n_validation_eval_period	2	Num. train checkpoints per validation evaluation
n_validation_eval_warmup	11	Num. train checkpoints before first validation evaluation
n_stop_evals_no_improve	37	Num. validation evaluations without improvement before stopping

C.4 Background and motivation for our experimental setup

As mentioned in the main paper, multi-level C2FAR models actually include shallower models as special cases (i.e., a three-level C2FAR model is equivalent to a one-level model with a single bin in both the second and the third levels). So given enough tuning, C2FAR is strictly more powerful than a flat binning. Moreover, any shallower model (including a flat binning) could potentially be improved by adding additional finer-grained C2FAR levels, increasing the precision without making the problem any more complex for the original shallower model. We initially used this approach, adding a second C2FAR level to our production flat binning system and finding gains across all our internal and publicly-available datasets. In essence, the finer-grained C2FAR level acts as a kind of "reconstruction function" [14] for the coarse, flat binning model; the lower-level network maps the top-level bin to a more precise location. Adding another lower-level model increases the precision further, again, without any cost to the coarser predictions, and further levels can be added recursively until the optimum level of precision is obtained for the task at hand.

However, adding additional levels does have two practical costs: each level increases both the number of parameters (via the added networks at each level) and the number of hyperparameters (that is, the number of bins at each new level). Based on both theory and initial experimental results, we are therefore confident that we can pay this cost in order to achieve superior forecasting accuracy. However, for the purposes of this paper, we elected to pursue a different experimental question: for a fixed parameter budget (counting parameters across all levels in the C2FAR hierarchy), can we jointly tune the number of bins at each level in order to achieve superior predictions compared to traditional approaches? And can we do this without any additional cost for hyperparameter tuning? To answer this question, we established a fixed parameter budget of one million parameters for each system, and a budget of 100 tuning trials for each system.

C.5 Tuned and fixed hyperparameters

The specific parameters that are tuned are given in Table 4. Given tuning search space grows exponentially with added hyperparameters, we elected to fix some hyperparameters to values that proved effective in preliminary experiments (e.g., *lstm_dropout* and *n_lstm_layers*); other hyperparameters are either set to follow prior work (e.g., the conditioning range parameters), or are set for practical reasons in order to help maximize the use of our available compute resources. See Table 5 for a list of all fixed hyperparameters.

Table 6: Tuning results, tuning for normalized deviation, in the empirical study.

Dataset	System	nhidden	NBins1	NBins2	NBins3	Total bins	Total intervals
elec	DeepAR-Gaussian	141	-	-	-	-	-
elec	C2FAR-RNN$_1$	248	110	-	-	110	110
elec	C2FAR-RNN$_2$	189	12	35	-	47	420
elec	C2FAR-RNN$_3$	156	5	25	21	51	2625
traff	DeepAR-Gaussian	165	-	-	-	-	-
traff	C2FAR-RNN$_1$	236	163	-	-	163	163
traff	C2FAR-RNN$_2$	184	28	13	-	41	364
traff	C2FAR-RNN$_3$	146	17	9	5	31	765
wiki	DeepAR-Gaussian	153	-	-	-	-	-
wiki	C2FAR-RNN$_1$	146	968	-	-	968	968
wiki	C2FAR-RNN$_2$	176	21	18	-	39	378
wiki	C2FAR-RNN$_3$	139	79	16	11	106	13904
azure	DeepAR-Gaussian	249	-	-	-	-	-
azure	C2FAR-RNN$_1$	83	75	-	-	75	75
azure	C2FAR-RNN$_2$	64	16	71	-	87	1136
azure	C2FAR-RNN$_3$	130	16	11	94	121	16544

Table 7: Training time in hours for top system on validation set in empirical study.

	elec	*traff*	*wiki*	*azure*	Average
DeepAR-Gaussian	18.2	31.3	1.0	5.0	13.9
C2FAR-RNN$_1$	28.7	27.1	12.8	12.8	20.3
C2FAR-RNN$_2$	70.0	72.1	22.5	6.8	42.9
C2FAR-RNN$_3$	29.5	57.1	32.3	31.5	37.6

Recall that for forecasting, we tune for multi-step-ahead normalized deviation (ND) on the validation set. Whether tuning C2FAR models or tuning DeepAR-Gaussian, this requires running the Monte Carlo sampling procedure to generate a forecast distribution; we use the median of this forecast distribution as the point forecast for evaluation. Since sampling is relatively expensive, we evaluate only every second checkpoint, only after 11 warm-up checkpoints, on only 32768 validation prediction ranges, and only using 25 rollouts to generate the forecast distribution (note corresponding hyperparameters in Table 5). We also use smaller batch sizes for generating forecasts (testing) than we do in training (since we simultaneously vectorize over Monte Carlo rollouts for each series).

C.6 Tuning results

Table 6 provides the results of our tuning procedure in terms of the selected number of hidden units and bins. While prior work used a fixed 1024 bins in their flat binning [14], our tuner often selected quite fewer bins for C2FAR-RNN$_1$ on our datasets. C2FAR-RNN$_2$ and C2FAR-RNN$_3$ models generally use fewer total bins than flat binning, while having very many more total actual intervals.

C.7 Computational performance and resource requirements

Table 7 has the training times for the top systems found on the validation set. Training time naturally reflects both the speed of convergence in learning (number of training epochs) *and* the speed of operating the specific architecture.[2]

Testing time roughly follows a similar pattern (Table 8), taking longer on the multi-level C2FAR models, although more efficient implementations than ours are certainly possible. Meanwhile, Table 9 gives the memory requirements of the different systems. Overall, we may say that in our implementation, multi-level C2FAR models run slower than a flat binning, but with less memory.

---

[2] Note that the baseline systems Naïve, Seasonal-naïve, and ETS do not require training; ETS parameters are fit separately for each input history at *inference* time.

Table 8: Time per 100 forecasts in *seconds* (running on NVIDIA Tesla P100) by top system on test set. All systems ran with common test batch sizes (60 for daily *wiki* dataset, 22 for hourly datasets) and number of samples (500) for each dataset.

	elec	*traff*	*wiki*	*azure*	Average
DeepAR-Gaussian	1.28	1.57	0.55	2.72	1.53
C2FAR-RNN$_1$	3.73	3.79	1.71	1.34	2.64
C2FAR-RNN$_2$	4.22	4.12	1.50	1.68	2.88
C2FAR-RNN$_3$	4.81	4.45	1.92	4.73	3.98

Table 9: Amount of memory consumed for prediction in MiB (measured via *nvidia-smi* on NVIDIA Tesla P100) by top system on test set. All systems ran with common *test batch sizes* (60 for daily *wiki*, 22 for hourly datasets) and number of samples (500) for each dataset. The flat binning, C2FAR-RNN$_1$, consumes significantly more memory on three of the four datasets; memory requirements depend directly on the number of bins selected by the tuner (see Table 6).

	elec	*traff*	*wiki*	*azure*	Average
DeepAR-Gaussian	4295	4873	3113	4845	4281.50
C2FAR-RNN$_1$	8264	7465	12619	3389	7934.25
C2FAR-RNN$_2$	6233	6147	5577	4209	5541.50
C2FAR-RNN$_3$	5657	5111	6372	6037	5794.25

C.8 Evaluation by forecast horizon

Figure 7 shows the forecast error of the systems as a function of the forecast horizon. Compared to three baselines (Naïve, Seasonal-naïve and DeepAR-Gaussian), we find C2FAR-RNN$_3$ is the best performer across virtually all horizons on all datasets, except the last couple horizons on *azure*. Interestingly, before C2FAR, practictioners faced a very nuanced problem when selecting a forecast method for their own dataset. E.g., on the cloud demand data that we are most interested in (*azure*), and ignoring C2FAR, it seems that on some horizons, Naïve is best, on others, DeepAR, and on others still, it is Seasonal-naïve. C2FAR combines the best aspects of all of these systems, outputting highly-precise predictions when doing so makes sense (similar to Naïve), but generating seasonally-adjusted estimates during the middle forecast horizons.

D Stability of Empirical Results

In this section, we investigate the stability of our empirical results. Random seeds are used in both our testing process (via Monte Carlo sampling of predicted future values) and our tuning process (via sampling of hyperparameters), and it is important to quantify the stability of these sources of randomness separately [6]. Ideally, we would repeat our entire tuning procedure multiple times with different random seeds, allowing us to determine the reliability of our process for fitting both model parameters and hyperparameters. While such repetition is not practical given the total time required, we can nevertheless investigate tuning randomness in other ways, and use this to assess the stability of our empirical results. Note that stability of results may depend on both the systems under investigation (e.g., models with more hyperparameters may be more unstable with respect to tuning), the datasets and splits used in the experiments (e.g., larger data sets may be more stable), and the tuning/training/testing processes (e.g., more tuning runs may lead to more stable results).

We summarize the results as follows, but provide full details in the following subsections:

- The Monte Carlo sampling process is very stable with respect to the random seed: we repeated the sampling 6 times and found negligible differences in normalized deviation (§D.1).
- Results are fairly stable when moving from the validation set to the test set: on all validation sets and all test sets, the top C2FAR models improve on the top DeepAR-Gaussian model. On all validation sets and all test sets, C2FAR-RNN$_2$ performed better than C2FAR-RNN$_1$.

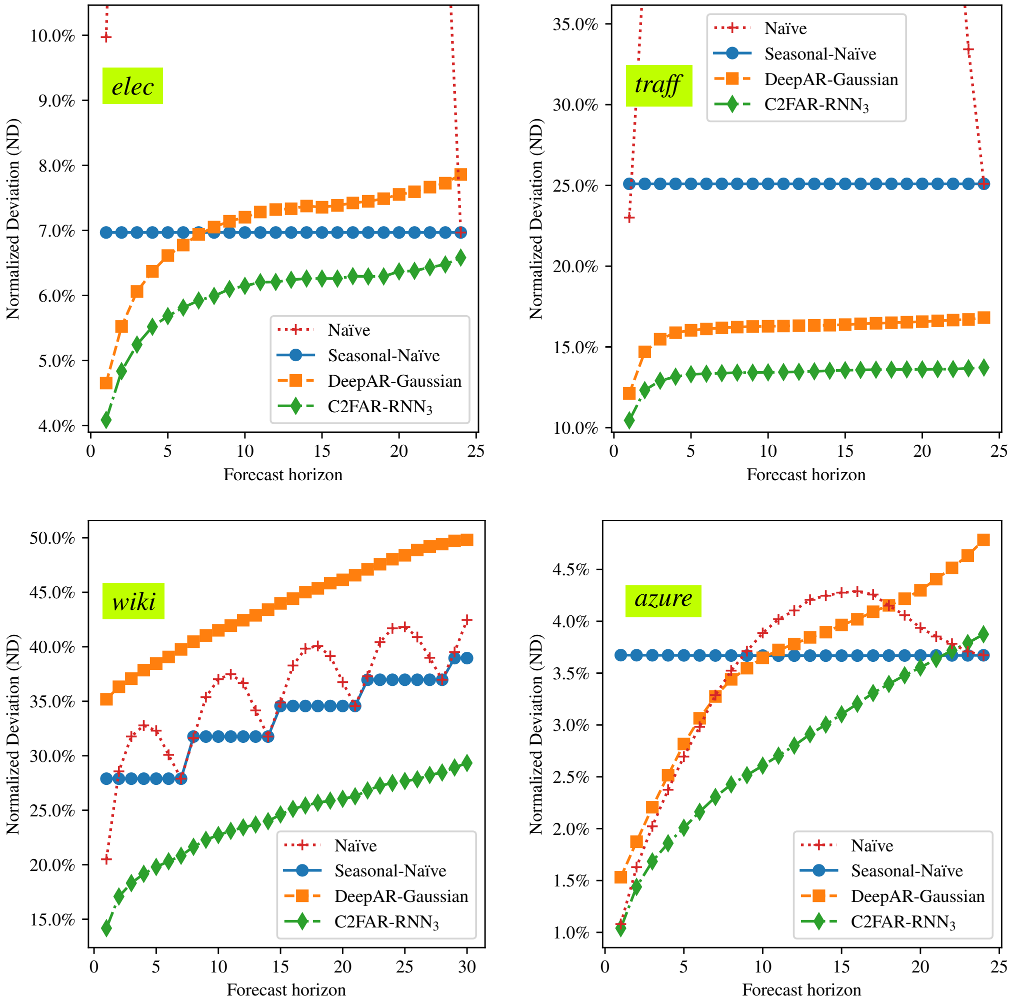


Figure 7: Comparison of different forecasting systems with normalized deviation (ND) calculated separately at each forecast horizon. For *elec*, *traff*, and *azure*, we forecast forward for one 24-hour seasonal cycle, while for *wiki*, we predict for slightly-more-than-four 7-day cycles. Seasonal-naïve is flat over a cycle because we evaluate using rolling predictions: every datapoint is forecast once at every horizon, and always gets the same prediction. Vanilla Naïve becomes first less accurate, then more accurate as we approach the end of the cycle, at which point it becomes equivalent to Seasonal-naïve. Aside from *wiki*, where DeepAR-Gaussian fails to learn a good model, DeepAR-Gaussian is competitive with C2FAR-RNN$_3$ at earlier horizons, but the gap widens over time.

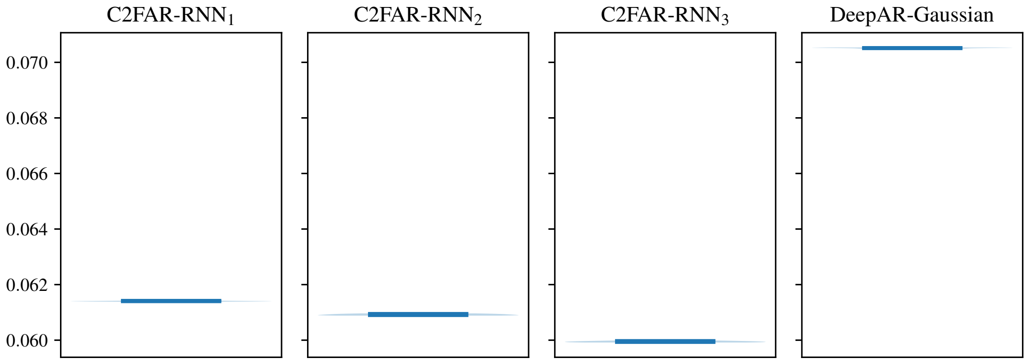


Figure 8: *elec* sampling stability: distribution of normalized deviation (on test set) across different random seeds as a violin plot.

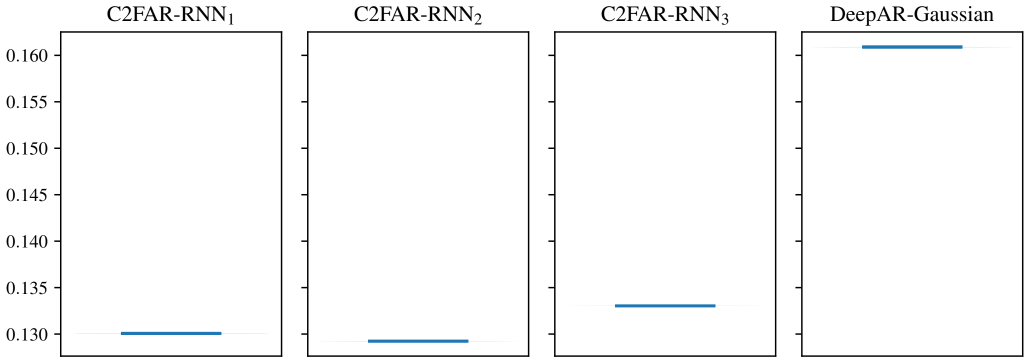


Figure 9: *traff* sampling stability: distribution of normalized deviation (on test set) across different random seeds as a violin plot.

However, in testing three of the four datasets, C2FAR-RNN$_3$ improved over its ranking on the validation set (§D.2).

- Results are less stable with respect to the tuning. If we use the second-best validation-set model on each *test* set, C2FAR-RNN$_3$ and C2FAR-RNN$_2$ suffer larger drops on test than C2FAR-RNN$_1$, while DeepAR-Gaussian improves in two cases. However, even with the second-best models, C2FAR models perform better than DeepAR-Gaussian across all test sets, and C2FAR-RNN$_3$ remains the top system on three of the four test sets (§D.3).

Note we do not assess the stability of re-training the models with the same hyperparameters, but different shuffling of the training data, as such re-training is not currently part of our operations. However, as future work, we plan to investigate the stability of re-training-without-re-tuning in the context of slight increases in the training data over time, which is very common in real-world ML pipelines.

D.1 Sampling stability

Recall that prediction of each example creates a forecast distribution using 500 different Monte Carlo rollouts of the time series. We take the median of this distribution to compute the normalized deviation (ND). Overall ND is the average over all test examples, and we report this in Table 1 of the main paper. We repeated the ND evaluation of our models 6 times using different random seeds, and plotted the distribution of the 6 results as *violin plots* in Figures 8 to 11. The distributions are very narrow, showing that average ND is very stable with respect to the random seed. We conclude that evaluation is very stable with respect to the random seed used in Monte Carlo sampling.

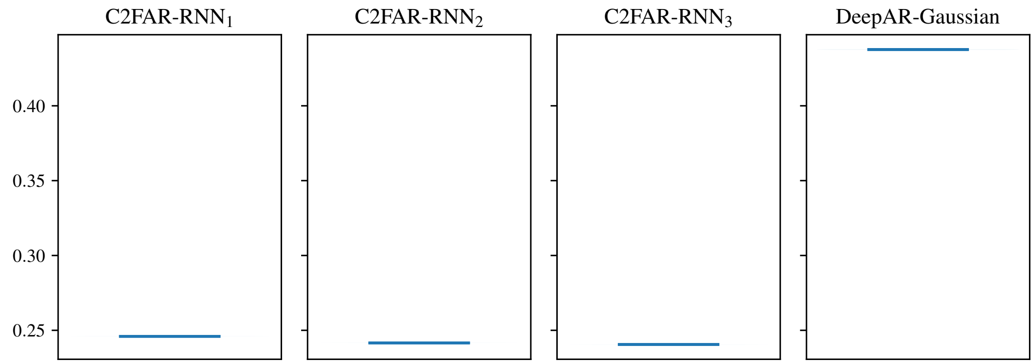


Figure 10: *wiki* sampling stability: distribution of normalized deviation (on test set) across different random seeds as a violin plot.

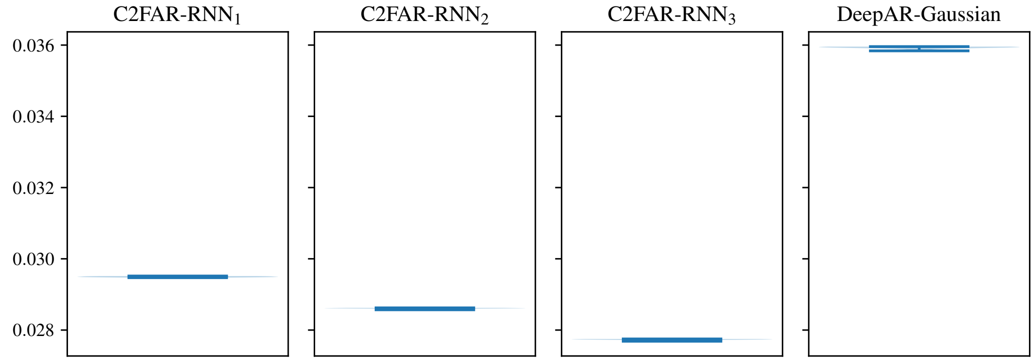


Figure 11: *azure* sampling stability: distribution of normalized deviation (on test set) across different random seeds as a violin plot.

D.2 Validation/test stability

Another possible source of instability is different behavior of systems on the validation set versus the test set. We summarize this behavior in our data by showing the ranking of the systems on the validation set and test set in Table 10 (note we exclude DeepAR-Gaussian from this table as it is always ranked 4th on all validation and test sets). Results are fairly stable moving from validation to test. On all validation sets and all test sets, all C2FAR models improve over DeepAR-Gaussian. Also, on all validation sets and all test sets, C2FAR-RNN$_2$ improves over C2FAR-RNN$_1$. The ranking in *traff* is the same on validation and test set, but on the others, the only difference is C2FAR-RNN$_3$ moved to first place on test.

Table 10: Validation/test stability

		elec			*traff*	
	1st	2nd	3rd	1st	2nd	3rd
Val.	C2FAR-RNN$_2$	**C2FAR-RNN$_3$**	C2FAR-RNN$_1$	C2FAR-RNN$_2$	C2FAR-RNN$_1$	C2FAR-RNN$_3$
Test	**C2FAR-RNN$_3$**	C2FAR-RNN$_2$	C2FAR-RNN$_1$	C2FAR-RNN$_2$	C2FAR-RNN$_1$	C2FAR-RNN$_3$
		wiki			*azure*	
	1st	2nd	3rd	1st	2nd	3rd
Val.	C2FAR-RNN$_2$	C2FAR-RNN$_1$	**C2FAR-RNN$_3$**	C2FAR-RNN$_2$	**C2FAR-RNN$_3$**	C2FAR-RNN$_1$
Test	**C2FAR-RNN$_3$**	C2FAR-RNN$_2$	C2FAR-RNN$_1$	**C2FAR-RNN$_3$**	C2FAR-RNN$_2$	C2FAR-RNN$_1$

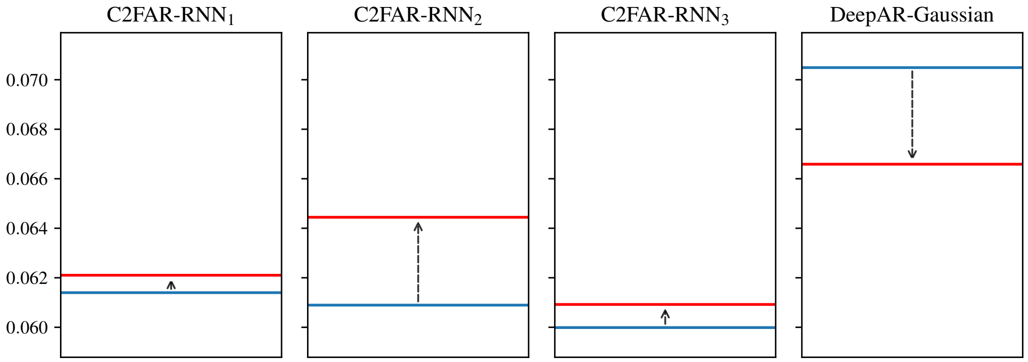


Figure 12: *elec* tuning stability: difference in normalized deviation (on test set) from top-1 tuned model (in blue) to second-best tuned model (in red).

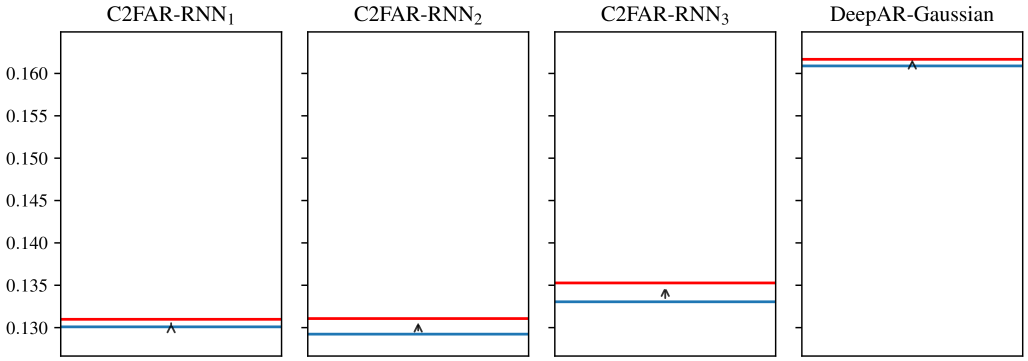


Figure 13: *traff* tuning stability: difference in normalized deviation (on test set) from top-1 tuned model (in blue) to second-best tuned model (in red).

D.3 Tuning stability

We assess the stability of our tuning results by considering the consequences had the optimizer not found the top model on the validation set. We therefore evaluate the second-best models from the validation set on the test set. We hypothesize that the second-best models may have larger drops for C2FAR-RNN$_2$ and C2FAR-RNN$_3$, as these models have more hyperparameters and are therefore more vulnerable to sub-optimal tuning. Figures 12 to 15 show the results, with the blue lines indicating the original top-model result, and the red lines indicating the result after switching to the second-best validation model.

We do see larger drops for the multi-level C2FAR models, especially C2FAR-RNN$_2$, which drops significantly on the *elec* dataset. However, even with the second-best models, C2FAR models perform better than DeepAR-Gaussian across all test sets, and C2FAR-RNN$_3$ remains, as before, the top system on three of the four test sets (§D.3). Overall, this suggests that we may wish to use more tuning trials for the multi-level C2FAR models, or, as suggested in the main paper, simply use the same number of bins at each level, reducing the number of bins to a single hyperparameter, as in the flat binning.

E Test-set likelihood experiments

In this section, we report some supplementary results comparing our systems for their ability to estimate the log-likelihood of held-out test data. As mentioned in the main paper in §4.2, log-likelihood is sometimes regarded as the de facto standard for evaluating generative models [21], and

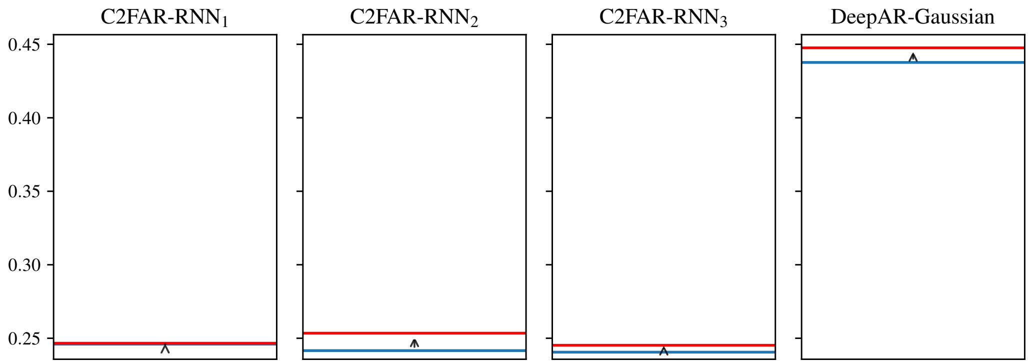


Figure 14: *wiki* tuning stability: difference in normalized deviation (on test set) from top-1 tuned model (in blue) to second-best tuned model (in red).

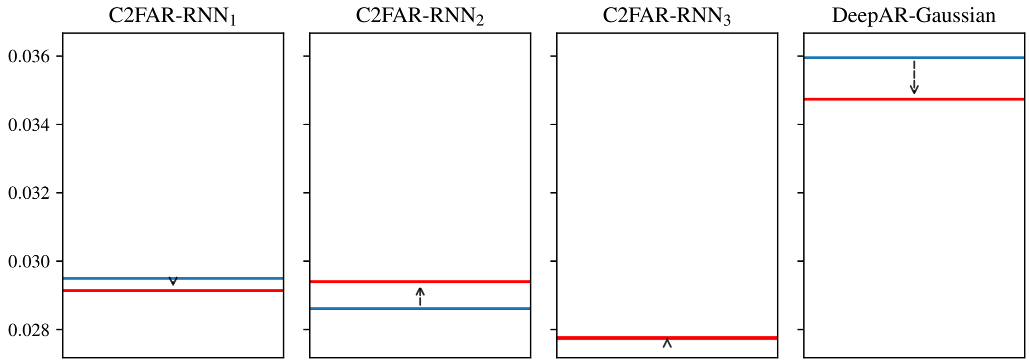


Figure 15: *azure* tuning stability: difference in normalized deviation (on test set) from top-1 tuned model (in blue) to second-best tuned model (in red).

Table 11: Tuning results, tuning for NLL, on *azure* with noise added. Compare to Table 6 for tuning for ND.

Dataset	System	nhidden	NBins1	NBins2	NBins3	Total bins	Total intervals
azure	DeepAR-Gaussian	74	-	-	-	-	-
azure	C2FAR-RNN$_1$	165	638	-	-	638	638
azure	C2FAR-RNN$_2$	126	70	31	-	101	2170
azure	C2FAR-RNN$_3$	141	6	14	67	87	5628

Table 12: NLL results on noisy and original datasets

	Experimental configuration					System			
Dataset	Trained for	Tested for	Tuned for	Trained on	Tested on	DeepAR-Gaussian	C2FAR-RNN$_1$	C2FAR-RNN$_2$	C2FAR-RNN$_3$
azure	NLL	NLL	NLL	+Noise	+Noise	1.355	-2.011	**-2.075**	**-2.075**
azure	NLL	NLL	ND	Original	+Noise	0.766	-1.739	-1.574	-1.504
azure	NLL	NLL	NLL	+Noise	Original	2.307	-3.300	-4.110	-4.043
azure	NLL	NLL	ND	Original	Original	0.094	-2.533	-4.475	**-6.309**

results on log-likelihood estimation are regarded as a proxy for the effectiveness of models on other tasks such as anomaly detection or missing value imputation. Since we cannot *tune* C2FAR models directly for log-likelihood on discrete data (as this leads to narrower and narrower density spikes), we investigate models trained under two other regimes:

1. Models tuned for ND (that is, the same models used in the forecasting experiments)
2. Models tuned for negative log likelihood (NLL), but on *dequantized* data, i.e., data with `Uniform[0,1]` noise added, as in prior work [16]

For the models tuned for NLL on the noise-added data, we use the same tuning setup as in forecasting, doing 100 tuning evaluations for each system and using the same tuned and fixed hyperparameters as described in §C.5. We performed this experiment on *azure* data only.

The resulting tuned hyperparameters are given in Table 11. Interestingly, the tuner selects quite many more bins for the C2FAR-RNN$_1$ model as were selected when tuning for ND (Table 6), suggesting precision is important even in noise-added data.

We evaluated both the NLL- and ND-tuned systems on both the original test data and the test data with `Uniform[0,1]` noise added. Results are given in Table 12. Likelihood is computed in the normalized domain (after min-max scaling) for all time series.

We can divide the evaluation into two objectives: NLL on the noise-added data, and NLL on the original data. If our objective is NLL on the noise-added data, then we see that multi-level C2FAR models still offer benefits over a flat binning. One may have expected this to not be the case, as adding noise removes some of the precision in the data and thus one of the advantages of C2FAR, but we see that both C2FAR-RNN$_2$ and C2FAR-RNN$_3$ still prove superior to the flat binning in this case. Even on noise-added data, multi-level C2FAR models use many more total intervals than those used by a flat binning (Table 11), illustrating that precision is still important even in noise-added data.

Now, regarding NLL of the *original* data, we see that multi-level C2FAR models are again superior to flat binning, and moreover, multi-level C2FAR models trained for ND are superior to those trained for NLL on the noise-added data. This illustrates that, if our objective is to achieve maximum likelihood on the original data, adding noise to dequantize the data may not be a good solution; here it results in worse NLL than simply tuning for ND on the original data. We repeat the point in the main paper that, in reality, for tasks such as anomaly detection, missing value imputation, denoising, compression, etc., we should not tune for NLL at all, but rather tune for the application-specific metric of interest.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2623–2631.

[2] Alexander Alexandrov, Konstantinos Benidis, Michael Bohlke-Schneider, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Danielle C Maddix, Syama Rangapuram, David Salinas, Jasper Schulz, Stella Lorenzo, Ali Caner Türkmen, and Yuyang Wang. 2020. GluonTS: Probabilistic and Neural Time Series Modeling in Python. *Journal of Machine Learning Research* 21, 116 (2020), 1–6.

[3] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J Christopher Beck. 2021. Generating Complex, Realistic Cloud Workloads using Recurrent Neural Networks. In *ACM SIGOPS Symposium on Operating Systems Principles*. 376–391.

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. *Advances in Neural Information Processing Systems* 24 (2011).

[5] Yitian Chen, Yanfei Kang, Yixiong Chen, and Zizhuo Wang. 2020. Probabilistic Forecasting with Temporal Convolutional Neural Network. *Neurocomputing* 399 (2020), 491–501.

[6] Jonathan H Clark, Chris Dyer, Alon Lavie, and Noah A Smith. 2011. Better Hypothesis Testing for Statistical Machine Translation: Controlling for Optimizer Instability. In *Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. 176–181.

[7] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Symposium on Operating Systems Principles*. 153–167.

[8] Jan Gasthaus, Konstantinos Benidis, Yuyang Wang, Syama Sundar Rangapuram, David Salinas, Valentin Flunkert, and Tim Januschowski. 2019. Probabilistic Forecasting with Spline Quantile Function RNNs. In *International conference on artificial intelligence and statistics*. 1901–1910.

[9] Adèle Gouttes, Kashif Rasul, Mateusz Koren, Johannes Stephan, and Tofigh Naghibi. 2021. Probabilistic Time Series Forecasting with Implicit Quantile Networks. *arXiv preprint arXiv:2107.03743* (2021).

[10] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural computation* 9, 8 (1997), 1735–1780.

[11] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. 2019. Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting. *Advances in Neural Information Processing Systems* 32 (2019).

[12] Gábor Melis, Chris Dyer, and Phil Blunsom. 2017. On the State of the Art of Evaluation in Neural Language Models. *arXiv preprint arXiv:1707.05589* (2017).

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Tejani Alykhan, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (2019).

[14] Stephan Rabanser, Tim Januschowski, Valentin Flunkert, David Salinas, and Jan Gasthaus. 2020. The Effectiveness of Discretization in Forecasting: An Empirical Study on Neural Time Series Models. *arXiv preprint arXiv:2005.10111* (2020).

[15] Syama Sundar Rangapuram, Lucien D Werner, Konstantinos Benidis, Pedro Mercado, Jan Gasthaus, and Tim Januschowski. 2021. End-to-End Learning of Coherent Probabilistic Forecasts for Hierarchical Time Series. In *International Conference on Machine Learning*. 8832–8843.

[16] Kashif Rasul, Abdul-Saboor Sheikh, Ingmar Schuster, Urs Bergmann, and Roland Vollgraf. 2020. Multi-Variate Probabilistic Time Series Forecasting via Conditioned Normalizing Flows. *arXiv preprint arXiv:2002.06103* (2020).

[17] David Salinas, Michael Bohlke-Schneider, Laurent Callot, Roberto Medico, and Jan Gasthaus. 2019. High-dimensional Multivariate Forecasting with Low-Rank Gaussian Copula Processes. *Advances in Neural Information Processing Systems* 32 (2019).

[18] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. 2020. DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks. *International Journal of Forecasting* 36, 3 (2020), 1181–1191.

[19] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[20] Souhaib Ben Taieb, James W Taylor, and Rob J Hyndman. 2017. Coherent Probabilistic Forecasts for Hierarchical Time Series. In *International Conference on Machine Learning*. 3348–3357.

[21] Lucas Theis, Aäron van den Oord, and Matthias Bethge. 2015. A note on the evaluation of generative models. *arXiv preprint arXiv:1511.01844* (2015).