A Appendix: Additional Proof Details

In this section, we (1) provide formal proof for CMAB's constant computation updates property, (2) include practical considerations to avoid numerical issues in the computation, (3) show that CMANPs uphold context and target invariance properties, and (4) include complexity analysis for CMANP-AND.

A.1 CMAB's Constant Computation Updates Proof

Recall, CMAB works as follows:

$$\mathbf{CMAB}(L_I, \mathcal{D}) = \mathbf{SA}(\,\mathbf{CA}(\,L_I,\ \mathrm{SA}(\,\mathrm{CA}(\,L_B, \mathcal{D}\,))\,))$$

where $\mathbf{SA}$ represents $\mathrm{SelfAttention}$ and $\mathbf{CA}$ represents $\mathrm{CrossAttention}$. The two cross-attentions have a linear complexity of $\mathcal{O}(N|L_B|)$ and a constant complexity $\mathcal{O}(|L_B||L_I|)$, respectively where $N = |\mathcal{D}|$. The self-attentions have constant complexities of $\mathcal{O}(|L_B|^2)$ and $\mathcal{O}(|L_I|^2)$, respectively. As such, the total computation required to compute the output of the block is $\mathcal{O}(N|L_B| + |L_B|^2 + |L_B||L_I| + |L_I|^2)$ where $|L_B|$ and $|L_I|$ are hyperparameter constants which bottleneck the amount of information which can be encoded.

Importantly, since $|L_B|$ and $|L_I|$ are constants (hyperparameters), CMAB's complexity is constant except for the contributing complexity part of the first attention block: $\mathrm{CrossAttention}(L_B, \mathcal{D})$, which has a complexity of $\mathcal{O}(N|L_B|)$. To achieve constant computation updates, it suffices that the updated output of this cross-attention can be updated in constant computation per datapoint. Simplified, $\mathrm{CrossAttention}(L_B, \mathcal{D})$ is computed as follows:

$$\mathrm{emb} = \mathrm{CrossAttention}(L_B, \mathcal{D}) = \mathrm{softmax}(QK^T)V$$

where K and V are key, value matrices respectively that represent the embeddings of the context dataset $\mathcal{D}_C$ and Q is the query matrix representing the embeddings of the block-wise latent vectors L_B. When an update with $\mathcal{D}_U$ new datapoints occurs, $|\mathcal{D}_U|$ rows are added to the key, value matrices. However, the query matrix is constant due to L_B being a fixed set of latent vectors whose values are learned.

Without loss of generality, for simplicity, we consider the $j-th$ output vector of the cross-attention (emb_j). Let $s_i = Q_{j,:}(K_{i,:})^T$ and $v_i = V_{i,:}$, then we have the following:

$$\mathrm{emb}_j = \sum_{i=1}^{N} \frac{\exp(s_i)}{C} v_i$$

where $C = \sum_{i=1}^{N} \exp(s_i)$. Performing an update with a set of new inputs D_U, results in adding $|\mathcal{D}_U|$ rows to the K, V matrices:

$$\mathrm{emb'}_j = \sum_{i=1}^{N+|\mathcal{D}_U|} \frac{\exp(s_i)}{C'} v_i$$

where $C' = \sum_{i=1}^{N+|\mathcal{D}_U|} \exp(s_i) = C + \sum_{i=N+1}^{N+|\mathcal{D}_U|} \exp(s_i)$. As such, the updated embedding $\mathrm{emb'}_j$ can be computed via a rolling average:

$$\mathrm{emb'_j} = \frac{C}{C'} \times \mathrm{emb_j} \ + \sum_{i=N+1}^{N+|\mathcal{D}_U|} \frac{e^{s_i}}{C'} v_i$$

Computing $\mathrm{emb'}_j$ and C' via this rolling average only requires $\mathcal{O}(|\mathcal{D}_U|)$ operations when given C and emb as required. In practice, however, this is not stable. The computation can quickly run into numerical issues such as overflow problems.

Practical Implementation: In practice, instead of computing and storing C and C', we instead compute and store $\log(C)$ and $\log(C')$.

The update is instead computed as follows: $\log(C') = \log(C) + \text{softplus}(T)$ where $T = \log(\sum_{i=N+1}^{N+|\mathcal{D}_U|} \exp(s_i - \log(C)))$. T can be computed efficiently and accurately using the log-sum-exp trick in $\mathcal{O}(|\mathcal{D}_U|)$. This results in an update as follows:

$$\text{emb}'_{\text{j}} = \exp(\log(C) - \log(C')) \times \text{emb}_{\text{j}} + \sum_{i=N+1}^{N+|\mathcal{D}_U|} \exp(s_i - \log(C'))v_i$$

This method of implementation avoids the numerical issues that will occur while resulting in computing the same emb'. We detail how to derive the practical implementation below:

Practical Implementation (Derivation):

$$C = \sum_{i=1}^{N} \exp(s_i) \qquad C' = \sum_{i=1}^{N+|\mathcal{D}_U|} \exp(s_i)$$

$$\begin{aligned}
\log(C') - \log(C) &= \log(\sum_{i=1}^{N+|\mathcal{D}_U|} \exp(s_i)) - \log(\sum_{i=1}^{N} \exp(s_i)) \\
\log(C') &= \log(C) + \log(\frac{\sum_{i=1}^{N+|\mathcal{D}_U|} \exp(s_i)}{\sum_{i=1}^{N} \exp(s_i)}) \\
\log(C') &= \log(C) + \log(1 + \frac{\sum_{i=N+1}^{N+|\mathcal{D}_U|} \exp(s_i)}{\sum_{i=1}^{N} \exp(s_i)}) \\
\log(C') &= \log(C) + \log(1 + \frac{\sum_{i=N+1}^{N+|\mathcal{D}_U|} \exp(s_i)}{\exp(\log(C))}) \\
\log(C') &= \log(C) + \log(1 + \sum_{i=N+1}^{N+|\mathcal{D}_U|} \exp(s_i - \log(C)))
\end{aligned}$$

Let $T = \log(\sum_{i=N+1}^{N+|\mathcal{D}_U|} \exp(s_i - \log(C)))$. Note that T can be computed efficiently using the log-sum-exp trick in $\mathcal{O}(|\mathcal{D}_U|)$. Also, recall the softplus function is defined as follows: $\text{softplus}(k) = \log(1 + \exp(k))$. As such, we have the following:

$$\begin{aligned}
\log(C') &= \log(C) + \log(1 + \exp(T)) \\
&= \log(C) + \text{softplus}(T)
\end{aligned}$$

Recall:

$$\text{emb}'_{\text{j}} = \frac{C}{C'} \times \text{emb}_{\text{j}} \; + \sum_{i=N+1}^{N+|\mathcal{D}_U|} \frac{\exp(s_i)}{C'} v_i$$

Re-formulating it using $\log(C)$ and $\log(C')$ instead of C and C' we have the following update:

$$\text{emb}'_{\text{j}} = \exp(\log(C) - \log(C')) \times \text{emb}_{\text{j}} + \sum_{i=N+1}^{N+|\mathcal{D}_U|} \exp(s_i - \log(C'))v_i$$

which only requires $\mathcal{O}(|\mathcal{D}_U|)$ computation (i.e., constant computation per datapoint) while avoiding numerical issues.

A.2 Additional Properties

In this section, we show that CMANPs uphold the context and target invariance properties.

Property: Context Invariance. A Neural Process p_θ is context invariant if for any choice of permutation function π, context datapoints $\{(x_i, y_i)\}_{i=1}^N$, and target datapoints $x_{N+1:N+M}$,

$$p_\theta(y_{N+1:N+M}|x_{N+1:N+M}, x_{1:N}, y_{1:N}) = p_\theta(y_{N+1:N+M}|x_{N+1:N+M}, x_{\pi(1):\pi(N)}, y_{\pi(1):\pi(N)})$$

Proof Outline: Since CMANPs retrieve information from a compressed encoding of the context dataset computed by CMAB (Constant Memory Attention Block). It suffices to show that CMABs compute their output while being order invariant in their input (i.e., context dataset in CMANPs) ($\mathcal{D}$).

Recall CMAB's work as follows:

$$\mathbf{CMAB}(L_I, \mathcal{D}) = \mathbf{SA}(\mathbf{CA}(L_I, \mathbf{SA}(\mathbf{CA}(L_B, \mathcal{D}))))$$

where L_I is a set of vectors outputted by prior blocks, L_B is a set of vectors whose values are learned during training, and $\mathcal{D}$ are the set of inputs in which we wish to be order invariant in.

The first cross-attention to be computed is $\mathbf{CA}(L_B, \mathcal{D})$. A nice feature of cross-attention is that its order-invariant in the keys and values; in this case, these are embeddings of $\mathcal{D}$. In other words, the output of $\mathbf{CA}(L_B, \mathcal{D})$ is order invariant in the input data $\mathcal{D}$.

Since the remaining self-attention and cross-attention blocks take as input: L_I and the output of $\mathbf{CA}(L_B, \mathcal{D})$, both of which are order invariant in $\mathcal{D}$, therefore the output of CMAB is order invariant in $\mathcal{D}$.

As such, CMANPs are also context invariant as required.

Property: Target Equivariance. A model p_θ is target equivariant if for any choice of permutation function π, context datapoints $\{(x_i, y_i)\}_{i=1}^N$, and target datapoints $x_{N+1:N+M}$,

$$p_\theta(y_{N+1:N+M}|x_{N+1:N+M}, x_{1:N}, y_{1:N}) = p_\theta(y_{\pi(N+1):\pi(N+M)}|x_{\pi(N+1):\pi(N+M)}, x_{1:N}, y_{1:N})$$

Proof Outline: The vanilla variant of CMANPs makes predictions similar to that of LBANPs (Feng et al., 2023) by retrieving information from a set of latent vectors via cross-attention and uses an MLP (Predictor). The architecture design of LBANPs ensure that the result is equivalent to making the predictions independently. As such, CMANPs preserve target equivariance the same way LBANPs do.

However, for the Autoregressive Not-Diagonal variant (CMANP-AND), the target equivariance is not held as it depends on the order in which the datapoints are processed. This is in common with that of prior methods by Nguyen & Grover (2022) and Bruinsma et al. (2023).

A.3 Complexity Analysis for CMANP-AND

For a batch of M datapoints and a prediction block size of b_Q (hyperparameter constant), there are $\lceil \frac{M}{b_Q} \rceil$ batches of datapoints whose predictions are made autoregressively. Each batch incurs a constant complexity of $\mathcal{O}(b_Q)^2$ due to predicting a full covariance matrix. As such for a batch of M target datapoints, CMANP-AND requires a sub-quadratic total computation of $\mathcal{O}(\lceil \frac{M}{b_Q} \rceil b_Q^2) = \mathcal{O}(Mb_Q)$ with a sequential computation length of $\mathcal{O}(\frac{M}{b_Q})$. Crucially, CMANP-AND only requires constant memory in $|\mathcal{D}_C|$ and linear memory in M, making it significantly more efficient than prior works which required at least quadratic memory.

B Appendix: Additional Experiments and Analyses

In this section, we (1) showcase the versatility of CMABs by applying them to Temporal Point Processes, (2) show results for CMANPs on Contextual Bandits, a setting where the amount of data increases over time, (3) include a memory complexity table which includes all baselines, and (4) analyse the time cost and performance relative to several hyperparameters.

	Mooc			Reddit		
	RMSE	NLL	ACC	RMSE	NLL	ACC
THP	0.202 ± 0.017	**0.267 ± 0.164**	**0.336 ± 0.007**	**0.238 ± 0.028**	**0.268 ± 0.098**	**0.610 ± 0.002**
CMHP	**0.168 ± 0.011**	**-0.040 ± 0.620**	0.237 ± 0.024	**0.262 ± 0.037**	**0.528 ± 0.209**	**0.609 ± 0.003**

Table 4: Temporal Point Processes Experiments.

B.1 Applying CMABs to Temporal Point Processes (TPPs)

In this section, we highlight the effectiveness of our proposed Constant Memory Attention Block by applying it to settings beyond that of Neural Processes. Specifically, we apply CMABs to Temporal Point Processes (TPPs). In brief, Temporal Point Processes are stochastic processes composed of a time series of discrete events. Recent works have proposed to model this via a neural network. Notably, models such as THP (Zuo et al., 2020) encode the history of past events to predict the next event, i.e., modelling the predictive distribution of the next event $p_\theta(\tau_{l+1}|\tau_{\leq l})$ where θ are the parameters of the model, τ represents an event, and l is the number of events that have passed. Typically, an event comprises a discrete temporal (time) stamp and a mark (categorical class).

B.1.1 Constant Memory Hawkes Processes (CMHPs)

Building on CMABs, we introduce the Constant Memory Hawkes Process (CMHPs) (Figure 5), a model which replaced the transformer layers in Transformer Hawkes Process (THP) (Zuo et al., 2020) with Constant Memory Attention Blocks. However, unlike THPs which summarise the information for prediction in a single vector, CMHPs summarise it into a set of latent vectors. As such, a flattening operation is added at the end of the model. Following prior work (Bae et al., 2023; Shchur et al., 2020), we use a mixture of log-normal distribution as the decoder for both THP and CMHP.

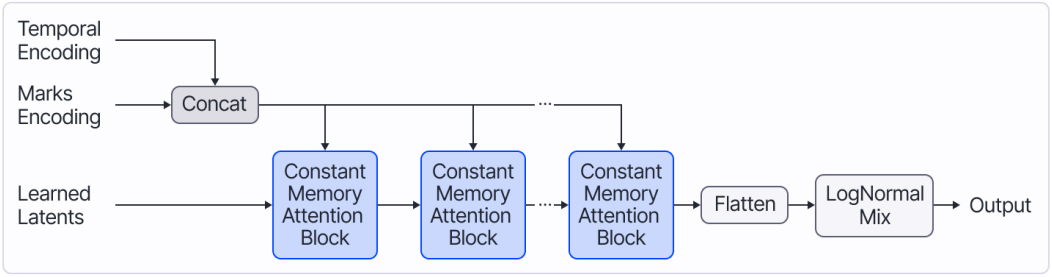


Figure 5: Constant Memory Hawkes Processes

B.1.2 CMHPs: Experiments

In this experiment, we compare CMHPs against THPs on standard TPP datasets: Mooc and Reddit.

Mooc Dataset. comprises of $7,047$ sequences. Each sequence contains the action times of an individual user of an online Mooc course with 98 categories for the marks.

Reddit Dataset. comprises of $10,000$ sequences. Each sequence contains the action times from the most active users with marks being one of the 984 the subreddit categories of each sequence.

The results (Table 4) suggest that replacing the transformer layer with CMAB (Constant Memory Attention Block) results in a small drop in performance. Crucially, unlike THP, CMHP has the ability to efficiently update the model with new data as it arrives over time which is typical in time series data such as in Temporal Point Processes. CMHP only pays constant computation to update the model unlike the quadratic computation required by THP.

B.2 Additional CMANPs Experiments: Contextual Bandits

In the Contextual Bandit setting introduced by Riquelme et al. (2018), a unit circle is divided into 5 sections which contain 1 low reward section and 4 high reward sections δ defines the size of the low

Method	$\delta = 0.7$	$\delta = 0.9$	$\delta = 0.95$	$\delta = 0.99$	$\delta = 0.995$
Uniform	100.00 ± 1.18	100.00 ± 3.03	100.00 ± 4.16	100.00 ± 7.52	100.00 ± 8.11
CNP	4.08 ± 0.29	8.14 ± 0.33	8.01 ± 0.40	26.78 ± 0.85	38.25 ± 1.01
CANP	8.08 ± 9.93	11.69 ± 11.96	24.49 ± 13.25	47.33 ± 20.49	49.59 ± 17.87
NP	1.56 ± 0.13	2.96 ± 0.28	4.24 ± 0.22	18.00 ± 0.42	25.53 ± 0.18
ANP	1.62 ± 0.16	4.05 ± 0.31	5.39 ± 0.50	19.57 ± 0.67	27.65 ± 0.95
BNP	62.51 ± 1.07	57.49 ± 2.13	58.22 ± 2.27	58.91 ± 3.77	62.50 ± 4.85
BANP	4.23 ± 16.58	12.42 ± 29.58	31.10 ± 36.10	52.59 ± 18.11	49.55 ± 14.52
TNP-D	**1.18 ± 0.94**	**1.70 ± 0.41**	2.55 ± 0.43	**3.57 ± 1.22**	**4.68 ± 1.09**
LBANP	**1.11 ± 0.36**	**1.75 ± 0.22**	**1.65 ± 0.23**	6.13 ± 0.44	8.76 ± 0.15
CMANP (Ours)	**0.93 ± 0.12**	**1.56 ± 0.10**	**1.87 ± 0.32**	9.04 ± 0.42	13.02 ± 0.03

Table 5: Contextual Multi-Armed Bandit Experiments with varying δ. Models are evaluated according to cumulative regret (lower is better). Each model is run 50 times for each value of δ.

reward section while the 4 high reward sections have equal sizes. In each round, the agent has to select 1 of 5 arms that each represent one of the regions. For context during the selection, the agent is given a 2-D coordinate X and the actions it selected and rewards it received in previous rounds.

If $||X|| < \delta$, then the agent is within the low reward section. If the agent pulls arm 1, then the agent receives a reward of $r \sim \mathcal{N}(1.2, 0.012)$. Otherwise, if the agent pulls a different arm, then it receives a reward $r \sim \mathcal{N}(1.0, 0.012)$. Consequently, if $||X|| \geq \delta$, then the agent is within one of the four high-reward sections. If the agent is within a high reward region and selects the corresponding arm to the region, then the agent receives a large reward of $N \sim \mathcal{N}(50.0, 0.012)$. Alternatively, pulling arm 1 will reward the agent with a small reward of $r \sim \mathcal{N}(1.2, 0.012)$. Pulling any of the other 3 arms rewards the agent with an even smaller reward of $r \sim \mathcal{N}(1.0, 0.012)$.

During each training iteration, $B = 8$ problems are sampled. Each problem is defined by $\{\delta_i\}_{i=1}^{B}$ which are sampled according to a uniform distribution $\delta \sim \mathcal{U}(0, 1)$. $N = 512$ points are sampled as context datapoints and $M = 50$ points are sampled for evaluation. Each datapoint comprises of a tuple (X, r) where X is the coordinate and r is the reward values for the 5 arms. The objective of the model during training is to predict the reward values for the 5 arms given the coordinates (context datapoints).

During the evaluation, the model is run for 2000 steps. At each step, the agent selects the arm which maximizes its UCB (Upper-Confidence Bound). After which, the agent receives the reward value corresponding to the arm. The performance of the agent is measured by cumulative regret. For comparison, we evaluate the modes with varying δ values and report the mean and standard deviation for 50 seeds.

Results. In Table 5, we compare CMANPs with other NP baselines, including the recent state-of-the-art methods TNP-D, EQTNP, and LBANP. We see that CMANP achieves competitive performance with state-of-the-art for $\delta \in \{0.7, 0.9, 0.95\}$. However, the performance degrades as δ reaches extreme values close to the limit such as 0.99 and 0.995 – settings that are at the edge of the training distribution.

B.3 Additional Analyses

Memory Complexity: In Table 6, we include a comparison of CMANPs with all NP baselines, showing that CMANPs are amongst the best in terms of memory efficiency when compared to prior NP methods. Notably, the methods with a similar memory complexity to CMANPs perform significantly worse in terms of performance across the various experiments (Tables 2 and 3)). As such, CMANPs provide the best trade-off in terms of memory and performance.

Time Cost and Performance Scatterplot: In Figure 6, we evaluate the empirical time cost of CMANP-AND with varying number of context datapoints ($N = |\mathcal{D}_C|$), number of target datapoints (M), and block size (b_Q). The number of context datapoints and the number of target datapoints are shown as labels in the scatterplot. The colour of the points on the scatterplot represents its respective block size. Depending on the amount of available resources (e.g., time), the value of the block size can be chosen equivalently.

	Conditioning	Querying		Updating	
In Terms of	$\lvert\mathcal{D}_C\rvert$	$\lvert\mathcal{D}_C\rvert$	M	$\lvert\mathcal{D}_C\rvert$	$\lvert\mathcal{D}_U\rvert$
CNP	✓	✓	✓	✓	✓
CANP	✗	✗	✓	✗	✗
NP	✓	✓	✓	✓	✓
ANP	✗	✗	✓	✗	✗
BNP	✓	✓	✓	✓	✓
BANP	✗	✗	✓	✗	✗
TNP-D	N/A	✗	✗	N/A	N/A
LBANP	✓	✓	✓	✓	✓
CMANP (Ours)	✓	✓	✓	✓	✓
TNP-ND	N/A	✗	✗	N/A	N/A
LBANP-ND	✓	✓	✗	✓	✓
CMANP-AND (Ours)	✓	✓	✓	✓	✓

Table 6: Comparison of Memory Complexities of Neural Processes with respect to the number of context datapoints $|\mathcal{D}_C|$, number of target datapoints in a batch M, and the number of new datapoints in an update $|\mathcal{D}_U|$. (Green) Checkmarks represent requiring constant memory, (Orange) half checkmarks represent requiring linear memory, and (Red) Xs represent requiring quadratic or more memory. A table with all baselines are included in the Appendix.

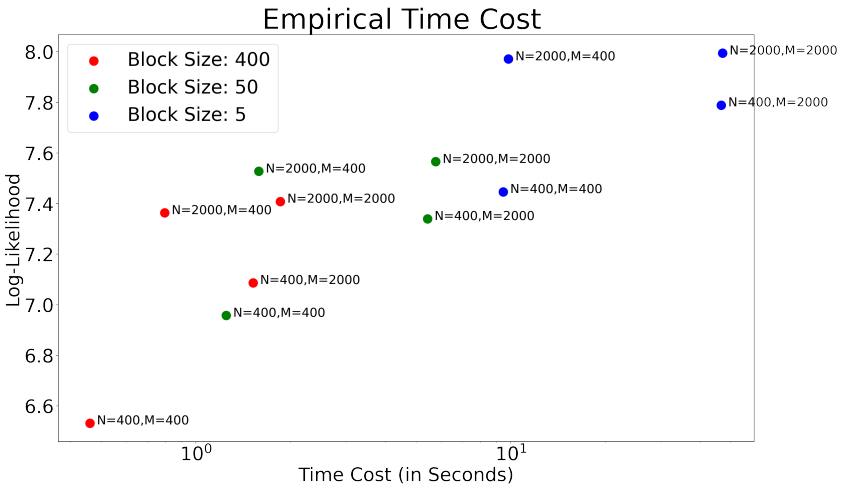


Figure 6: Scatterplot comparing the empirical time cost of CMANP-AND with respect to the block size (b_Q), number of context datapoints (N), and number of target datapoints (M).

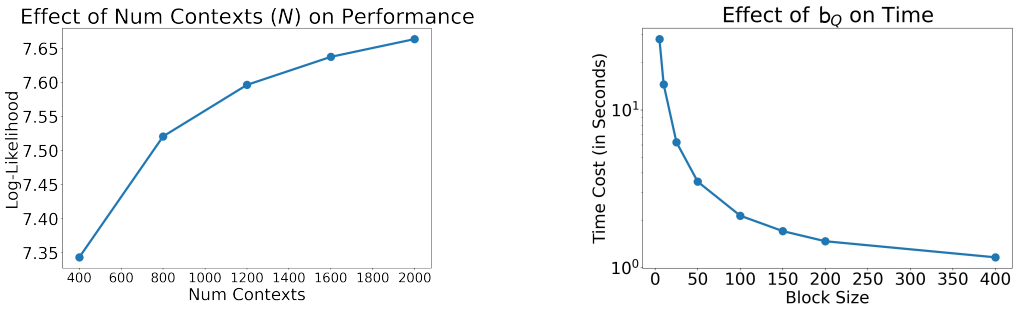
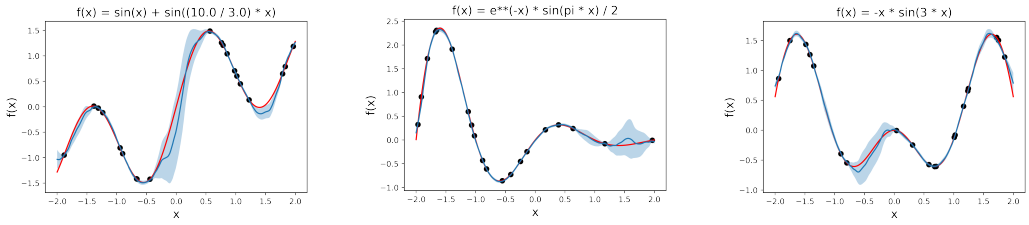


Figure 7: Additional Analyses Graphs



(a) $y = sin(x) + sin(10/3) * x$ (b) $y = e^{-x} * sin(\pi * x)/2$ (c) $y = -x * sin(3 * x)$

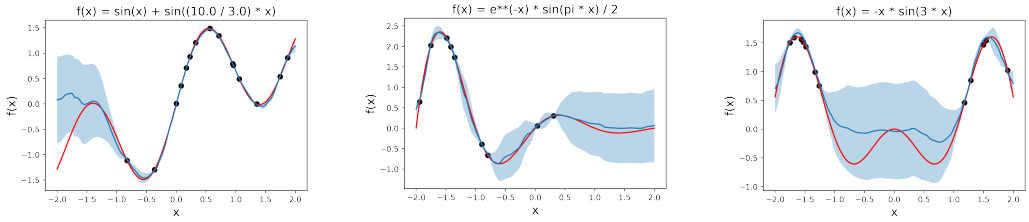
Figure 8: CMANPs 1-D Regression Visualizations

Generalisation Ability: In Figure 7, we evaluated CMANP-AND's potential to generalize to settings with significantly more context datapoints than originally trained on. During training, the model was trained on tasks with a maximum of 800 context datapoints. In contrast, during evaluation, we conditioned on up to 2000 context datapoints and evaluated on 800 target datapoints. Empirically, we found that the model's performance grows consistently as the number of context datapoints increases. However, the performance slows down at large number of contexts. We hypothesize that the cause of the saturation is due to two main factors: (1) the information gained from new context datapoints is dependent on the size of the current context dataset. For example, adding 400 new datapoints to a context dataset of size 400 results in 100% more data. Alternatively, adding 400 new datapoints to a context dataset of size 1600 results in 25% more data. As such, it is expected to see such saturation with a linear x-axis scaling. (2) in this case, CelebA (64 x 64) comprising of only 4096 pixels in total. 2000 comprises of a substantial amount of the data, i.e., approximately half. As such, saturation is expected as the amount of information gained by additional datapoints is minimal.

Effect of Block Size (b_Q) on Empirical Time Cost: In Figure 7, we evaluated the time required for CMANP-AND with respect to the block size (b_Q). The results are as expected, showing that the time required during deployment is lower as the block size increases. In the main paper, we showed that lower block sizes improve the model's performance. In conjunction, these plots show that there is a trade-off between the time cost and performance. These results suggest that during deployment it is advisable to select smaller block sizes if allowed for the time constraint.

Visualizations: In Figures 8 and 10, we show visualizations for the 1-D regression and Image Completion tasks respectively. Figure 9 show out-of-distribution visualizations where the context datapoints are only sampled from part of the distribution.

Number of Latents Comparison with LBANPs: A major factor that affects the performance in iterative attention-based models is the size of the bottleneck (i.e., the number of latents). Feng et al. (2023) showed that the performance of LBANPs (iterative attention based Neural Process) can change significantly depending on $|L_{LBANPs}|$ (the number of latents). As such, for the sake of fairness, in our paper, we similarly set the number of latents in CMANPs to match the same number of latents used in LBANPs' paper, i.e., $|L_I| = |L_B| = |L_{LBANPs}|$.



(a) $y = sin(x) + sin(10/3) * x$ (b) $y = e^{-x} * sin(\pi * x)/2$ (c) $y = -x * sin(3 * x)$

Figure 9: CMANPs 1-D Out-of-Distribution Regression Visualizations. The model is evaluated between $[-2.0, 2.0]$. However, context datapoints are sampled from only (a) $[-1.0, 2.0]$, (b) $[-2.0, 1.0]$, and (c) $[-2.0, -1.0] \cup [1.0, 2.0]$.

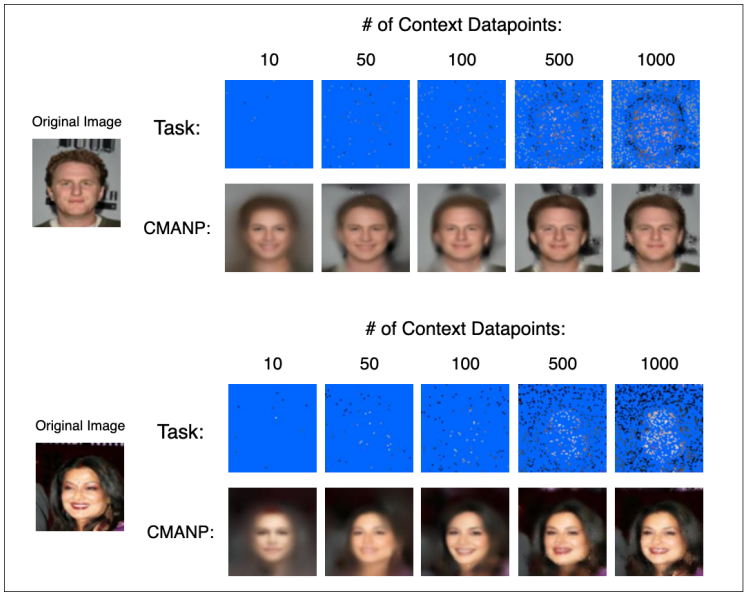
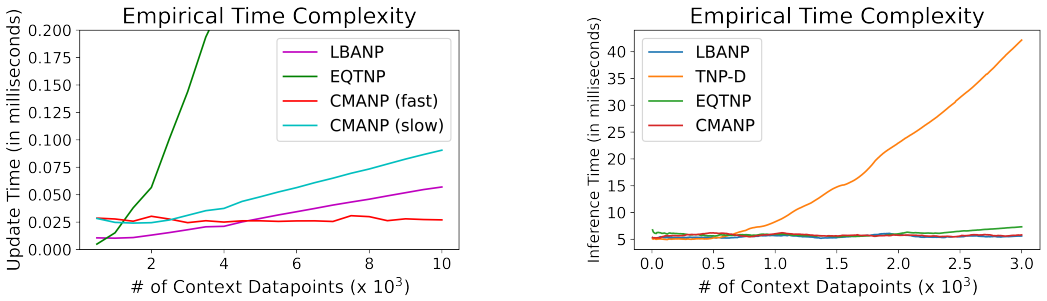


Figure 10: CMANPs Image Completion Visualizations

Num Latents	CMANPs	LBANPs
8	3.49 ± 0.02	3.54 ± 0.01
16	3.60 ± 0.03	3.64 ± 0.02
32	3.73 ± 0.03	3.77 ± 0.01
64	3.79 ± 0.06	3.88 ± 0.01
128	3.92 ± 0.03	3.97 ± 0.02

Table 7: Comparison of CMANPs with LBANPs for varying number of latents on the CelebA (32x32) image completion task. The number of latents in CMANPs matches the same number of latents used in LBANPs' paper, i.e., $|L_I| = |L_B| = |L_{LBANPs}|$. We see that CMANPs are competitive with LBANPs performing slightly worse. However, unlike LBANPs, CMANPs (1) computes their output in constant memory, and (2) perform updates in constant computation given new context tokens (in this case, pixels)



(a) Runtime analysis of the update process.

(b) Runtime analysis of the query process.

Figure 11: Analyses Graphs comparing the runtime of CMANPs with various baselines. (a) Comparison of the update procedure of CMAB-based NP (CMANPs) with Perceiver's iterative attention-based NP model (LBANPs) and a transformer-based NP model (EQTNP). CMANP (fast) refers to the CMAB's efficient update mechanism. CMANP (slow) refers to the traditional update mechanism. (b) Comparison of the query/inference process of CMANPs with LBANPs (Perceiver's iterative attention-based model), TNPs (Transformer-based model), and EQTNPs (Transformer-based model with an efficient query mechanism).

For completeness, we have included in Table 7 a comparison of the performance of CMANPs and LBANPs for varying number of latents for the CelebA (32x32) image completion task. We see that CMANPs are competitive with LBANPs performing slightly worse. However, unlike LBANPs, CMANPs (1) computes their output in constant memory, and (2) perform updates in constant computation given new context tokens (in this case, pixels).

Empirical Time Comparison with Baselines: In Figure 11a, we compare CMANP using the efficient update process with CMANP using the traditional update process, showing that the efficient update process is initially similar in runtime to the traditional update process. However, as the number of context datapoints increases (i.e., updates are performed) over time, the traditional update process requires linear runtime while our proposed efficient update process still only requires constant runtime.

In Figure 11a, we also compare the runtime of the update process of CMAB-based NP (CMANPs) with Perceiver's iterative attention-based NP model (LBANPs) and a transformer-based NP model. We see that the CMAB-based model only requires a constant amount of time to perform the update. In contrast, Perceiver's iterative attention-based model's update runtime scales linearly and Transformer model's update runtime scales quadratically.

In Figure 11b, we compare the querying (inference) runtime of CMANP with LBANPs (Perceiver's iterative attention-based model), TNPs (Transformer-based model). We see that CMANPs and LBANPs stay constant while the transformer-based model (TNP) scales quadratically in runtime.

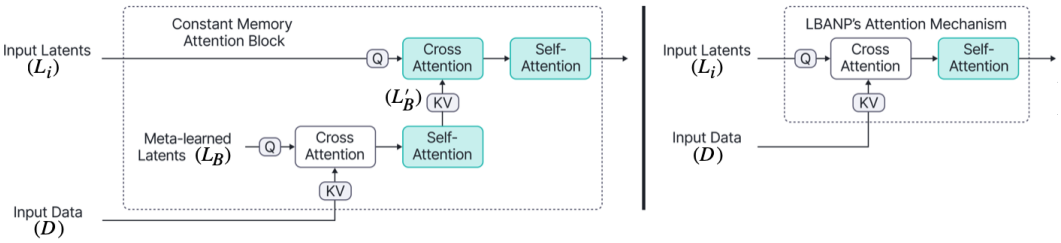


Figure 12: Comparison of our proposed Constant Memory Attention Block and that of LBANP's Attention Block (i.e., Perceiver's iterative attention). The green blocks indicate constant complexity. Naively computing the outputs, the white blocks indicate linear complexity. CMAB, however, can compute its white cross attention block in constant memory via a rolling average. LBANP's Attention block (Perceiver's iterative attention) cannot compute their white cross attention block in constant memory.

We would like to note, however, that runtime is highly dependent on the efficiency of the implementation and the hardware. Since our work focused primarily on the memory aspect rather than runtime, our implementation was that of a simple sequential version of CMABs and CMANPs. However, CMANPs have an architecture which allows for several modules within CMABs to be parallelized when performing updates for improved runtime. As such, we expect that an optimized codebase will be able to significantly improve CMAB's and CMANP's runtime.

C Appendix: Discussion

In this section, we (1) compare Perceiver's iterative attention with CMABs, detailing why Perceiver cannot achieve the efficiency properties of CMAB, (2) compare the likelihood computation of Autoregressive Not-Diagonal extension with Not-Diagonal extension, and (3) compare NPs with other existing methods for uncertainty estimation.

C.1 Comparison of Iterative Attention with CMABs

Figure 12 compares Perceiver's iterative attention (used in LBANPs) with CMABs (used in CMANPs). In this subsection, we detail why Perceiver's iterative attention cannot achieve computing its output in constant memory and performing updates in constant computation. Notably, the property of constant memory is dependent on constant computation updates. Below, we detail why Perceiver's iterative attention does not have the constant computation updates property. Previously, we proved that the output of CrossAttention can be updated in constant computation per datapoint via a rolling summation given that the query vectors are constants. The efficiency gains revolve around CMABs' block-wise learnable latent vectors denoted as L_B being a learned constant.

When stacked, CMABs work as follows: $L_{i+1} = \mathrm{SA}(\mathrm{CA}(L_i, L'_B))$ where $L'_B = \mathrm{SA}(\mathrm{CA}(L^i_B, \mathcal{D}))$ and L^i_B denote the block-wise latent vectors for the i-th CMAB.

Perceiver's iterative attention block works as follows: $L_{i+1} = \mathrm{SA}(\mathrm{CA}(\mathrm{L_i}, \mathcal{D}))$.

When new datapoints $\mathcal{D}_U$ is added to the input, i.e., $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_U$, the input latents ($L_i^{updated} \neq L_i$ where $i > 0$) change and is thus not a constant. As such, Perceiver's iterative attention do not allow for (1) constant computation updates and (2) computing output in constant memory, making it more expensive in terms of memory compared to CMABs.

For CMABs, computing $L_{i+1} = \mathrm{SA}(\mathrm{CA}(L_i, L'_B))$ is always constant in computation since $|L_i|$ and $|L'_B|$ are constant in size. Computing the updated output: $L^i_B = \mathrm{SA}(\mathrm{CA}(L_B, \mathcal{D} \cup D_U))$ can always be computed in constant computation because L_B is a constant.

C.2 COMPARISON OF THE LIKELIHOOD COMPUTATION OF AUTOREGRESSIVE NOT-DIAGONAL EXTENSION WITH NOT-DIAGONAL EXTENSION:

In brief, the Autoregressive Not-Diagonal extension is different from Not-Diagonal extension in that the predictions are made autoregressively which allows for more flexible distributions than prior Not-Diagonal variants. As such, it is expected that the autoregressive not-diagonal variant's likelihood is higher than that of the non-autoregressive baselines which only model an unimodal gaussian distribution. Consider the following didactic example where $B_Q = 1$ (the block prediction size).

Since -AND feeds earlier samples back into the model for making predictions, the likelihood of the target datapoints: $\{(x_i, y_i)\}_{i=1}^{M}$ for our -AND model is computed as follows:

$$\begin{aligned}\log p_{AND}(y_{1:M}|x_{1:M}, D_{context}) &= \log \prod_{i=1}^{M} p(y_i|x_{1:i-1}, y_{1:i-1}, x_i, D_{context}) \\ &= \sum_{i=1}^{M} \log p(y_i|x_{1:i-1}, y_{1:i-1}, x_i, D_{context})\end{aligned}$$

In contrast, consider the likelihood of -ND: $\log p_{ND}(y_{1:M}|x_{1:M}, D_{context})$. By Boole's Inequality (or Union Bound), we have that

$$\log p_{ND}(y_{1:M}|x_{1:M}, D_{context}) \leq \sum_{i=1}^{M} \log p(y_i|x_{1:M}, D_{context}) = \sum_{i=1}^{M} \log p(y_i|x_i, D_{context})$$

$(x_{1:i-1}, y_{1:i-1})$ provides relevant information for predicting the value of the function at x_i, e.g., nearby pixel values in image completion. As a result, it is likely the case that:

$$p(y_i|x_i, D_{context}) \leq p(y_i|x_{1:i-1}, y_{1:i-1}, x_i, D_{context})$$

Summing from $i = 1 \ldots M$, this means:

$$\log p_{ND}(y_{1:M}|x_{1:M}, D_{context}) \leq \log p_{AND}(y_{1:M}|x_{1:M}, D_{context})$$

As such, it is expected that the autoregressive not-diagonal variant's likelihood is higher than that of the non-autoregressive baselines.

C.3 COMPARISON OF NPS WITH OTHER EXISTING METHODS FOR UNCERTAINTY ESTIMATION

Other popular methods which can perform uncertainty estimation, include and are not limited to MC-Dropout, Ensembles, Gaussian Processes (GPs), and Bayesian Neural Networks (BNNs).

Ensembles is an approximate Bayesian method which trains a group of neural networks on the same set of datapoints. The predictions of this group of neural networks are used to provide uncertainty predictions. Ensembles require retraining several models with gradient descent when new datapoints are received which is very costly.

GPs specify a Gaussian distribution over the function values that fit the datapoints. However, GPs scale cubically with the number of datapoints, making it only practical in settings with a small number of datapoints.

Bayesian Neural Networks is a stochastic neural network with a prior over weights trained using Bayesian inference. BNNs suffer their own respective challenges such as difficulty in tuning, difficulty in specifying weight priors, and cold posteriors. They also often perform worse compared to approximate bayesian methods.

D Appendix: Implementation, Hyperparameter Details, and Compute

D.1 Implementation and Hyperparameter Details

We use the implementation of the baselines from the official repository of TNPs (https://github.com/tung-nd/TNP-pytorch) and LBANPs (https://github.com/BorealisAI/latent-bottlenecked-anp). The datasets are standard for Neural Processes and are available in the same link. We follow closely the hyperparameters of TNPs and LBANPs. In CMANP, the number of blocks for the conditioning phase is equivalent to the number of blocks in the conditioning phase of LBANP. Similarly, the number of cross-attention blocks for the querying phase is equivalent to that of LBANP. We used an ADAM optimizer with a standard learning rate of $5e-4$. We performed a grid search over the weight decay term $\{0.0, 0.00001, 0.0001, 0.001\}$. Consistent with prior work (Feng et al., 2023) who set their number of latents $L = 128$, we also set the number of latents to the same fixed value $L_I = L_B = 128$ without tuning. Due to CMANPs and CMABs architecture, they allow for varying embedding sizes for the learned latent values (L_I and L_B). For simplicity, we set the embedding sizes to 64 consistent with prior works (Nguyen & Grover, 2022; Feng et al., 2023). The block size for CMANP-AND is set as $B_Q = 5$. During training, CelebA (128x128), (64x64), and (32x32) used a mini-batch size of 25, 50, and 100 respectively. All experiments are run with 5 seeds. For the Autoregressive Not-Diagonal experiments, we follow TNP-ND and LBANP-ND (Nguyen & Grover, 2022; Feng et al., 2023) and use cholesky decomposition for our LBANP-AND experiments. Focusing on the efficiency aspect, we follow LBANPs in the experiments and consider the conditional variant of NPs, optimizing the log-likelihood directly.

D.2 Compute

All experiments were run on a Nvidia GTX 1080 Ti (12 GB) or Nvidia Tesla P100 (16 GB) GPU. 1-D regression experiments took 4 hours to train. EMNIST took 2 hours to train. CelebA (32x32) took 16 hours to train. CelebA (64x64) took 2 days to train. CelebA (128x128) took 3 days to train.

D.3 Comparison of Model parameters

Each CMAB consists of 2 self-attention blocks and 2 cross-attention blocks compared to LBANP's attention block which consists of 1 self-attention block and 1 cross-attention block. In our experiments, the models have 6 encoder layers (e.g., 6 CMABs) and 6 querying decoder layers (i.e., CrossAttention). As a result, CMANP uses an overall 30 attention blocks and LBANP uses an overall 18 attention blocks, i.e., CMANP uses approximately 67% more parameters than LBANPs. Although CMANPs use more parameters than LBANPs, CMANPs ultimately use less memory (only constant!) since the number of inputs is the bottleneck in terms of memory usage for attention-based methods.

D.4 Runtime

Previously, we analyzed the runtime for our method. Unfortunately, comparing the runtime of existing baselines is difficult as they have been optimized differently, making it hard to compare the runtimes fairly. NP models such as LBANPs and CMANPs have an architecture which interleaves modules, allowing for different modules to be computed in parallel at the same time for improved efficiency. For example, CMANPs compute encodings of the context dataset via: $L_i = \mathrm{CrossAttention}(\mathrm{SelfAttention}(L_{i-1}, \mathcal{D}_C)$ and retrieves information from this context dataset for prediction via: $X^i_{query} = \mathrm{CrossAttention}(X^{i-1}_{query}, L_i)$. In an optimized codebase, computing L_{i+1} and X^{query}_i can actually be computed in parallel, resulting in a significantly more efficient model in terms of runtime. However, the publicly available codebase for LBANPs does not support this. Another example is that of Conditional Neural Processes (CNPs), a variant of NPs which leverages DeepSets. CNPs are able to efficiently compute updates via a rolling averaging mechanism. However, the available codebases do not support this by default either. Specialized implementations for comparing the runtime of NP methods are outside the scope of our work. Nonetheless, we detail below how to implement an efficient version of CMANPs.

D.5 EFFICIENT IMPLEMENTATION

In our code, we implemented a sequential variant of CMANPs that computes each CrossAttention and Self-Attention module sequentially. However, computing parts of the stacked CMAB blocks in a model can be done in parallel to improve the processing speed. All CMAB blocks can compute the following costly operation $L'_B = \mathrm{SelfAttention}(\mathrm{CrossAttention}(L_B, \mathcal{D}_C))$ in parallel. In addition, CMAB can perform all updates to L'_B in parallel as well. This is particularly important for the Autoregressive Not-Diagonal extension. When the prediction block size (b_Q) decreases, this corresponds to performing more CMAB updates since the predictions are made autoregressively. As such, a properly optimized codebase which computes in parallel would significantly reduce the runtime. Note that using the model this way would still be constant memory since the number of stacked CMAB blocks is a fixed hyperparameter.