A Appendix

Algorithm 1: Training the ViTCA update rule with a "pool sampling"-based approach

Input : ViTCA cell update rule F_θ, hyper-parameters $\Omega = \{I \in \mathbb{N}^+, b \in \mathbb{N}^+, \sigma \in [0,1], C_h \in \mathbb{N}^+, \eta \in \mathbb{R}^+, \alpha \in \mathbb{R}, \beta \in \mathbb{R}, N_\mathcal{P} \in \mathbb{N}^+\}$, dataset of images $\mathcal{D} = \{\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_{N_b \in \mathbb{N}^+}\}$

Output : Optimal update rule parameters θ_I

$\theta_0 \leftarrow$ initial update rule parameters; // *E.g.*, He initialization [46]

$\mathcal{P} \leftarrow \emptyset$; // Pool of cell grids and their respective ground truth images

for $i \leftarrow 1$ **to** I **do**

  $\mathbf{X} \leftarrow (\mathbf{X}_j, ..., \mathbf{X}_{j+b})$ where $j \sim \mathcal{U}\{1, N_b - b\}$; // $(\cdot, ..., \cdot)$ is batch-wise concatenation

  if $|\mathcal{P}| > b$ **and** $i \bmod 2 = 0$ **then**

    $\mathbf{P} \leftarrow \{(\mathbf{Z}_1, \mathbf{X}_1), ..., (\mathbf{Z}_b, \mathbf{X}_b)\} \subset \mathcal{P}$; // Retrieve first b elements from the pool

    $\mathbf{Z} \leftarrow (\mathbf{Z}_1, ..., \mathbf{Z}_b)$; $\mathbf{X} \leftarrow (\mathbf{X}_1, ..., \mathbf{X}_b)$; // Retrieve cell grids and images from $\mathbf{P}$

  else

    // Zero-initialize grids of cells and inject noisy inputs

    // i is used for determining noise shape and coverage

    // C_h determines the number of cell hidden channels

    $\mathbf{Z} \leftarrow \texttt{seed}(\texttt{mask}(\mathbf{X}, i), C_h)$;

  end

  $T \sim \mathcal{U}\{8, 32\}$; // Randomly sample number of cell updates to perform

  for $t \leftarrow 1$ **to** T **do**

    $\mathbf{Z} \leftarrow F_{\theta_{i-1}}(\mathbf{Z}, \sigma)$; // Iteratively update cell grids with cell update prob. σ

  end

  // $\mathbf{Z}_o$ and $\mathbf{Z}_h$ are output and hidden channels of cell grids, respectively

  $L_{o\_overflow} \leftarrow \frac{1}{C_o}\|\mathbf{Z}_o - \min(\max(\mathbf{Z}_o, 0), 1)\|_1$; // Output channels overflow loss

  $L_{h\_overflow} \leftarrow \frac{1}{C_h}\|\mathbf{Z}_h - \min(\max(\mathbf{Z}_h, -1), 1)\|_1$; // Hidden channels overflow loss

  $L_{rec} \leftarrow \frac{1}{C_o}\|\mathbf{Z}_o - \mathbf{X}\|_1$; // Image reconstruction loss

  $L \leftarrow \frac{1}{bHW}(\alpha L_{rec} + \beta(L_{o\_overflow} + L_{h\_overflow}))$;

  $Q \leftarrow \nabla L / (\|\nabla L\|_F + 10^{-8})$; // Normalize gradients. $\|\cdot\|_F$ is Frob. norm

  $\theta_i \leftarrow \theta_{i-1} - \eta Q$; // Update the update rule parameters

  $\mathcal{P} \leftarrow \mathcal{P} \cup \{(\mathbf{Z}_1, \mathbf{X}_1), ..., (\mathbf{Z}_b, \mathbf{X}_b)\}$; // Append updated cell grids and ground truths

  $\mathcal{P} \leftarrow \texttt{trunc}(\texttt{shuffle}(\mathcal{P}), N_\mathcal{P})$; // Shuffle pool and retain first $N_\mathcal{P}$ elements

end

A.1 Training on high-resolution imagery with fusion and mitosis

As an alternative to gradient checkpointing for reducing memory usage, we briefly experimented with a downsampling scheme inspired by cell fusion and mitosis when training on CelebA at 64×64. Specifically, we split the T applications of the update rule (within a training iteration) into multiple stages: 1) We apply the update rule twice so that cells will have, at minimum, some amount of knowledge of their neighbours. 2) We stash the masked input for a later re-injection. 3) *Fusion*—we apply a 2×2 average pooling with a stride of 2 across the cell grid, combining 2×2 groups of cells into singular cells. 4) We apply the update rule $T - 4$ times at this 32×32 downsampled cell grid resolution. 5) *Mitosis*—we perform a 2×2 duplication of cells (each cell is duplicated to its right, bottom-right, and bottom). 6) We re-inject the stashed masked input. 7) We apply the update rule twice to adapt the cells to the 64×64 resolution and to fill in any missing information.

We found that performing this fusion and mitosis scheme decreased training memory consumption to levels similar to our gradient checkpointing scheme ($\sim 50\%$ memory reduction) while having a $\sim 70\%$ faster backward pass. Loss-wise, we observed a $\sim 33\%$ increase in the average validation reconstruction loss during training, which can qualitatively be observed in the example provided in Fig. 5 (*bottom*). Although the results shown are not ideal—*i.e.*, we did not perform a hyper-parameter search here, for example, finding the optimal number of iterations preceding fusion and following mitosis—this brief experiment tests the feasibility of reducing memory consumption while maintaining denoising capability and avoiding gradient checkpointing. As shown in the figure, ViTCA with fusion and mitosis is able to successfully denoise the input despite applying updates at two

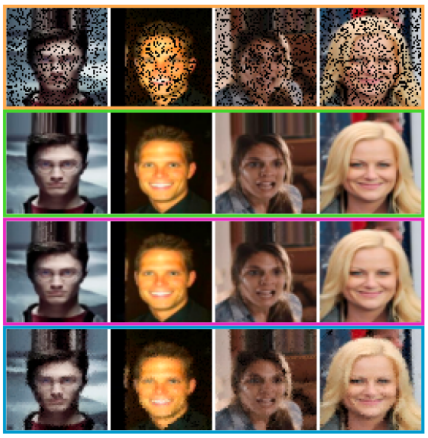

Figure 5: Qualitative results using cell fusion and mitosis as an alternative to gradient checkpointing. Gold boxes are inputs, green ground truths, purple ViTCA outputs, and blue ViTCA w. fusion and mitosis outputs. Outputs are after 64 CA iterations.

		PSNR ↑	SSIM ↑	LPIPS ↓	# Params.
Pool size	128	26.51	0.914	0.065	92.5K
	256	26.40	0.912	0.067	92.5K
	512	**26.61**	**0.915**	**0.064**	92.5K
	1024	26.53	0.913	0.066	92.5K
	2048	26.54	**0.915**	**0.064**	92.5K
	4096	26.48	0.912	0.066	92.5K
	8192	26.30	0.910	0.069	92.5K
Cell init.	*constant*	**26.53**	**0.913**	**0.066**	92.5K
	random	25.90	0.905	0.074	92.5K
Patch size	*1×1*	**26.53**	**0.913**	**0.066**	92.5K
	2×2	25.85	0.906	0.076	96.0K
	4×4	24.54	0.882	0.113	109.8K
	8×8	21.62	0.803	0.212	165.3K
	16×16	18.71	0.687	0.279	387.0K

Table 4: Quantitative ablation on pool size $N_{\mathcal{P}}$, cell initialization method, and patch size $P_H \times P_W$ for denoising autoencoding with ViTCA on CelebA. Boldface and underlining denote best and second best results. Italicized items denote baseline configuration settings.

	PSNR ↑	SSIM ↑	LPIPS ↓
3×3	**23.25**	**0.827**	**0.145**
5×5	22.34	0.817	**0.145**
7×7	21.65	0.792	0.168

Table 5: Quantitative ablation on attention neighbourhood size $N_H \times N_W$ for denoising autoencoding with ViTCA on FashionMNIST. Boldface and underlining denote best and second best results. Italicized items denote baseline configuration settings.

	LandCoverRep			**MNIST**			**CelebA**			**FashionMNIST**		
	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓
asynchronous	**33.80**	**0.932**	**0.102**	**27.01**	**0.940**	**0.028**	**26.53**	**0.913**	**0.066**	**23.80**	**0.855**	**0.117**
synchronous	33.68	0.931	0.104	26.00	0.927	0.034	23.76	0.870	0.105	23.12	0.832	0.132

Table 6: Quantitative ablation comparing test results with ViTCA trained using asynchronous ($\sigma = 50\%$) *vs.* synchronous ($\sigma = 100\%$) cell updates for denoising autoencoding. During testing, cells are updated at the rate they were trained in. Boldface denotes best results. Italicized items denote baseline configuration settings.

different scales. This scale agnostic behaviour reveals potentially interesting research directions beyond the scope of this work, such as allowing an NCA update rule to dynamically and locally modify cell grid resolution based on a compute budget, which could see applications in signal (image, video, or audio) compression.

A.2 Extended ablation study

Here we present an extension of our ablation study in Sec. 4.1.1, using the baseline ViTCA model as our reference. As before, the ablation examines the effects certain training configuration parameters have on test performance.

Pool size, cell initialization, and patch size. In Tab. 4, we examine the impact of varying the (max) pool size $N_{\mathcal{P}}$, cell initialization method, and patch size $P_H \times P_W$ on CelebA. As shown in the table, it is difficult to correlate pool size with test performance. However, when pool size $N_{\mathcal{P}} = 8192$,

	LandCoverRep			MNIST			CelebA			FashionMNIST		
	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓
disabled	**33.80**	0.932	0.102	**27.01**	**0.940**	**0.028**	**26.53**	**0.913**	**0.066**	23.80	0.855	0.117
enabled	33.76	0.932	0.102	26.77	0.938	0.029	26.05	0.904	0.075	**23.89**	0.855	0.117

Table 7: Quantitative ablation comparing test results of ViTCA trained with gradient checkpointing disabled *vs*. enabled. Boldface denotes best results. Italicized items denote baseline configuration settings.

there is a noticeable reduction in performance. Test performance also degrades when initializing cells such that their output and hidden channels receive random values sampled from $\mathcal{U}(0,1)$ and $\mathcal{U}(-1,1)$, respectively, as opposed to receiving constant values (0.5 for output channels and 0 for hidden). Finally, we see a consistent decrease in performance when the input image is divided into non-overlapping patches $> 1\times1$, as well as an increase in the number of model parameters.

Attention neighbourhood size. In Tab. 5, we examine the impact of attention neighbourhood size $N_H \times N_W$ on FashionMNIST. Interestingly, increasing the neighbourhood size past 3×3 causes a degradation in performance. This is most likely attributed to the increase in complexity caused by incorporating more information into ViTCA's self-attention. One would expect explicitly increasing the receptive field of spatially localized self-attention to result in better performance, but it can also complicate the process of figuring out which neighbours to attend to. We believe this may be alleviated by increasing model capacity and/or training duration. As described in Sec. 4, we use the Moore neighbourhood (3×3) as it requires less computation while still demonstrating ViTCA's effectiveness.

Asynchronous *vs*. synchronous cell updates. In Tab. 6, we compare between training with asynchronous cell updates ($\sigma = 50\%$) and training with synchronous cell updates ($\sigma = 100\%$) on LandCoverRep, MNIST, CelebA, and FashionMNIST. Training with asynchronous cell updates provides a meaningful increase in performance compared to training with synchronous cell updates and comes with several benefits, such as not requiring cells in a neighbourhood to be in sync with each other and serving as additional data augmentation. Similarly mentioned in related work [21], this allows ViTCA to be used in a distributed system where cells need not exist under a global clock and can be updated at varying rates. Thus making it easier to scale up or down within a non-homogeneous compute environment. This was somewhat demonstrated in Fig. 4 (d) where ViTCA was able to adapt to varying update rates despite being trained on a fixed asynchronous update rate ($\sigma = 50\%$).

Effects of gradient checkpointing. In Tab. 7, we compare between training with gradient checkpointing disabled and with gradient checkpointing enabled on LandCoverRep, MNIST, CelebA, and FashionMNIST. Similarly shown in Tab. 2, we see here that training with gradient checkpointing has an adverse effect on test performance. As mentioned in Sec. 5, NCAs—during training—require all activations from each recurrent iteration to be stored in memory before performing backpropagation. This results in memory usage being proportional to the amount of recurrent iterations. As such, depending on ViTCA's configuration, gradient checkpointing may be required to be able to train on a single GPU. We make use of PyTorch's `checkpoint_sequential`, which we use as follows: given the number of CA iterations T, we divide the sequential (forward) application of the update rule into $\lfloor T/2 \rfloor$ segments of roughly the same length (depending on whether T is even or odd). Then, all segments are executed in sequence, where activations from only the first and last segments are stored as well as the inputs to each intermediate segment. The intermediate inputs are used for re-running the segments without stored activations during the backward pass to compute gradients. This results in a trade-off between memory consumption and backpropagation duration since each intermediate segment's forward pass needs to be re-computed during its backward pass. Moreover, and not mentioned in the documentation of PyTorch at the time of writing, there exists a subtle yet meaningful side-effect which we have observed and confirmed through the use of GNU Debugger (GDB) and Python Debugger (PDB): Without gradient checkpointing, gradients are accumulated all at once at the end of backpropagating through the entire computation graph, resulting in the expected round-offs due to limitations in machine precision (`float32` in our case). At this point, PyTorch may

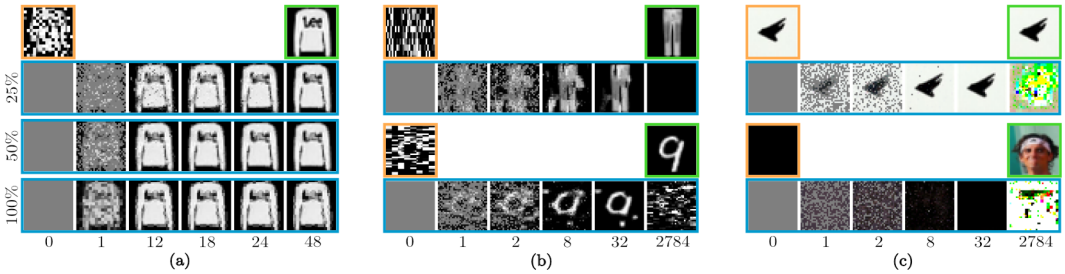

Figure 6: Qualitative results showcasing UNetCA's inductive biases in terms of adapting to: **(a)** varying cell update rates; **(b)** noise configurations unseen during training, and; **(c)** unmasked and completely masked inputs. Gold boxes are inputs, green ground truths, and blue UNetCA outputs.

use a variety of numerical techniques to minimize round-off, such as cascade summation (verified to be used for CPU-based summation, see `SumKernel.cpp` in PyTorch) which recursively sums two halves of a sequence of summands as opposed to naively summing them in sequence. *With* gradient checkpointing, gradients are accumulated at each segment. This means that round-offs are forced to (potentially) occur at each checkpoint/segment instead of once at the end of the entire computation graph. Even if cascade summation is used when summing gradients within each segment, the segment-wise ordering may reduce its effectiveness. We verified this behaviour by observing an exact machine epsilon difference ($\epsilon \approx 1.19 \times 10^{-7}$ in IEEE 754 standard) in the gradient—when compared to the non-checkpointed scheme—of the final operation of the update rule at the second-last segment, once the loss started to diverge.

It is important to note that despite the difference in gradients, the accuracy of the forward pass remains unchanged between the checkpointed and non-checkpointed models. Also, we must remind ourselves that round-offs are unavoidable when performing floating-point arithmetic, meaning that gradients computed within a deep learning library such as PyTorch are always an *estimation* of the true gradient. Importantly, both checkpointed and non-checkpointed models exhibited the same spikes and dips in their validation losses over the course of training, also decreasing at similar rates.

A.3 Extended analysis of cell state and update rule inductive biases

Here we present an extension of the analyses provided in Sec. 4.1.2 and Sec. 4.1.3.

Adaptation to varying update rates (UNetCA). Fig. 6 (a) shows UNetCA capable of adapting to a slower ($\sigma = 25\%$) cell update rate despite being trained with a $\sigma = 50\%$ cell update rate. Interestingly, UNetCA experiences difficulty synchronously updating all cells ($\sigma = 100\%$), producing a noticeably lower quality output compared to its outputs at asynchronous rates. This is in contrast to ViTCA (Fig. 4 (d)), where the quality of output remains the same across all update rates. Also, not shown in Fig. 4 (d), but is important to note, are the number of ViTCA iterations from left-to-right, which are as follows: 1, 8, 12, 16, 32. We point attention to the fact that UNetCA required 48 iterations to converge with $\sigma = 25\%$, 24 iterations to converge with $\sigma = 50\%$, and could not converge to a good solution with $\sigma = 100\%$, while ViTCA required 32 iterations to converge with $\sigma = 25\%$, 16 iterations to converge with $\sigma = 50\%$, and 8 iterations to converge with $\sigma = 100\%$.

Generalization to noise unseen during training (UNetCA). As shown in Fig. 6 (b), UNetCA is incapable of generalizing to noise configurations unseen during training, inducing a divergence in cell states. This is in contrast to ViTCA as shown in Fig. 4 (e). ViTCA not only produces a higher fidelity output mid-denoising, but it also maintains cell state stability.

Effects of not *vs*. completely masking input (UNetCA). Fig. 6 (c; *top*): Although UNetCA is able to successfully autoencode the unmasked input image, it eventually induces a divergence amongst cell states. This is in contrast to ViTCA as shown in Fig. 4 (g; *left*). ViTCA not only produces a higher fidelity output mid-denoising, but it also maintains cell state stability. Fig. 6 (c; *bottom*): Unlike ViTCA (Fig. 4 (g; *right*)), UNetCA does not output the median image when attempting to denoise a completely masked input and instead causes cells to diverge.

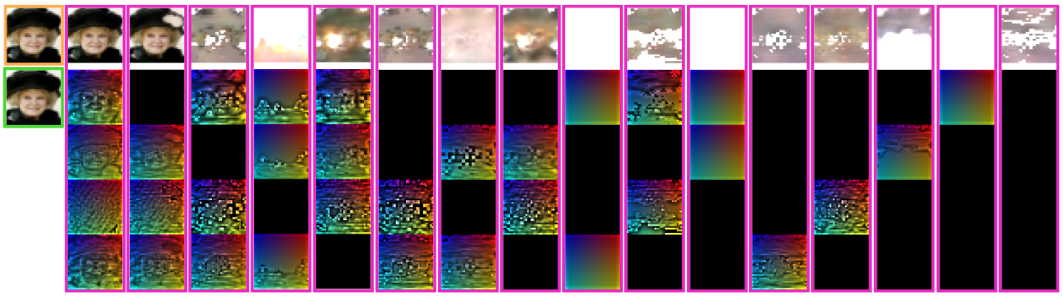

Figure 7: Qualitative results showcasing ViTCA's inductive biases in terms of adapting to masking one or several of its self-attention heads. Gold boxes are inputs, green ground truths, and purple ViTCA outputs (after 2784 iterations). For reference, the first column of outputs does not contain any head masking.

Effect of masking heads. Fig. 7 shows how ViTCA reacts to having its self-attention heads masked during autoencoding (no noise) an example from CelebA. The purpose of this experiment is to observe each head's contribution to the output. We can see that when none of the heads are masked, they attend to facial features and contours, and the output is as expected. However, once heads are masked, the unmasked heads stop attending to the features they once did and instead deteriorate. In some cases, the unmasked heads stop attending to anything at all. There are a couple of interesting cases: 1) When only the first head is masked, ViTCA is still able to successfully autoencode the input, although there is a slight degradation in quality. This is consistent with examples from the other datasets as well as when there is noise involved. 2) When certain heads are masked, the noise that the model was trained to denoise starts to appear (*e.g.*, fourth column from left and fifth column from right).

	Fwd. ↘	Bwd. ↘	Mem. ↘
disabled	**229ms**	**355ms**	17.0GB
enabled	232ms	576ms	**2.5GB**

Table 8: Profiling results showcasing ViTCA's runtime performance (forward and backward in milliseconds) and memory usage (in GB) while training on a minibatch of random $32 \times 3 \times H \times W$ images with gradient checkpointing disabled *vs*. enabled. We use $T = 32$ ViTCA iterations and 16 checkpoint segments. Boldface denotes best results. Italicized items denote baseline configuration settings.

A.4 Runtime analysis of ViTCA

Here we provide a brief analysis of ViTCA's runtime performance and memory usage while training on a minibatch of random $32 \times 3 \times H \times W$ images through measurements of forward pass duration (ms), backward pass duration (ms), and training memory usage (GB), with and without using gradient checkpointing. We use $T = 32$ ViTCA iterations and 16 checkpoint segments. Results are shown in Tab. 8. Gradient checkpointing provides substantial memory savings at the cost of proportionally increasing the duration of the backward pass.