Supplementary Materials
Clockwork Diffusion: Efficient Generation With Model-Step Distillation

Amirhossein Habibian* Amir Ghodrati* Noor Fathima* Guillaume Sautiere
Risheek Garrepalli Fatih Porikli Jens Petersen
Qualcomm AI Research†
{ahabibia, ghodrati, noor, gsautie, rgarrepa, fporikli, jpeterse}@qti.qualcomm.com

A. Clockwork details

UNet Architecture In Fig. 1 we show a detailed schematic of the SD UNet architecture. The parts in pink are replaced by our lightweight adaptor. We also show parameter counts and GMACs per block. In ablations we varied the level at which we introduce the adaptor, as shown in Table 3 of the main body. There we compare "Stage 1 (res 32x32)" (our default setup) and "Stage 2 (res 16x16)" (a variant where DOWN-1 and UP-2 remain in the model), finding better performance for the former. Interestingly, our sampling analysis suggested that introducing the adaptor at such a high resolution, replacing most parts of the UNet, should lead to poor performance. However, this is only true if we replace multiple consecutive steps (see adaptor clock ablations in Table 3 of the main body). By alternating adaptor and full UNet passes we recover much of the performance, and can replace more parts of the UNet than would otherwise be possible.

Adaptor Architecture In Fig. 2 we show a schematic of our UNet-like adaptor architecture, as discussed in ablations (Section 5.4 of the main paper). In addition to our ResNet-like architecture (Fig. 3 of the main paper) We tried 1) a UNet-like convolutional architecture with 640 channels in each block and 4 ResNet blocks in the middle level ($N = 4$), 2) a lighter variant of it with 96 channels and 2 ResNet blocks in the middle level. While all adaptors provide comparable performance, the ResNet-like adaptor obtains better quality-complexity trade-off.

Training We provide pseudocode for our unrolled training in Algorithm 1.

*Equal contribution

†Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc

Algorithm 1 Adaptor training with unrolled trajectories

Require: Teacher model ϵ
Require: Adaptor ϕ_θ
Require: Prompt set P
Require: Clock schedule $\mathcal{C}(t)$
  for N_e epochs **do**
    $P_D \leftarrow \text{RandomSubset}_D(P)$ ▷ optional
    $D \leftarrow \text{GenerateTrajectories}(P_D, \epsilon)$
    for all Trajectory & prompt $(T, text) \in D$ **do**
      for all $(t, \boldsymbol{r}_t^{in}, \boldsymbol{r}_t^{out} \boldsymbol{r}_{t+1}^{out}) \in T$ **do**
        if $\mathcal{C}(t) = 1$ **then**
          $\hat{\boldsymbol{r}}_t^{out} \leftarrow \phi_\theta \left(\boldsymbol{r}_t^{in}, \boldsymbol{r}_{t+1}^{out}, \boldsymbol{t}_{emb}, \boldsymbol{text}_{emb}\right)$
          $\mathcal{L} \leftarrow \|\boldsymbol{r}_t^{out} - \hat{\boldsymbol{r}}_t^{out}\|_2$
          $\theta \leftarrow \theta - \gamma \nabla \mathcal{L}$
        end if
      end for
    end for
  end for

	Steps	FID	CLIP	TFLOPs
DPM++	8	24.22	0.302	9.5
+ Clockwork	8	23.21	0.296	5.9
DPM	8	24.32	0.301	9.5
+ Clockwork	8	23.24	0.296	5.9
PNDM	8	35.64	0.272	9.5
+ Clockwork	8	33.15	0.280	5.9
DDIM	8	34.72	0.287	9.5
+ Clockwork	8	38.38	0.280	5.9

Table 1. Clockwork works with different schedulers.

B. Ablations

Scheduler. We evaluate Clockwork across multiple schedulers: DPM++, DPM, PNDM, and DDIM. With the exception of DDIM, Clockwork improves FID at negligible change to the CLIP score, while reducing FLOPs by 38%.

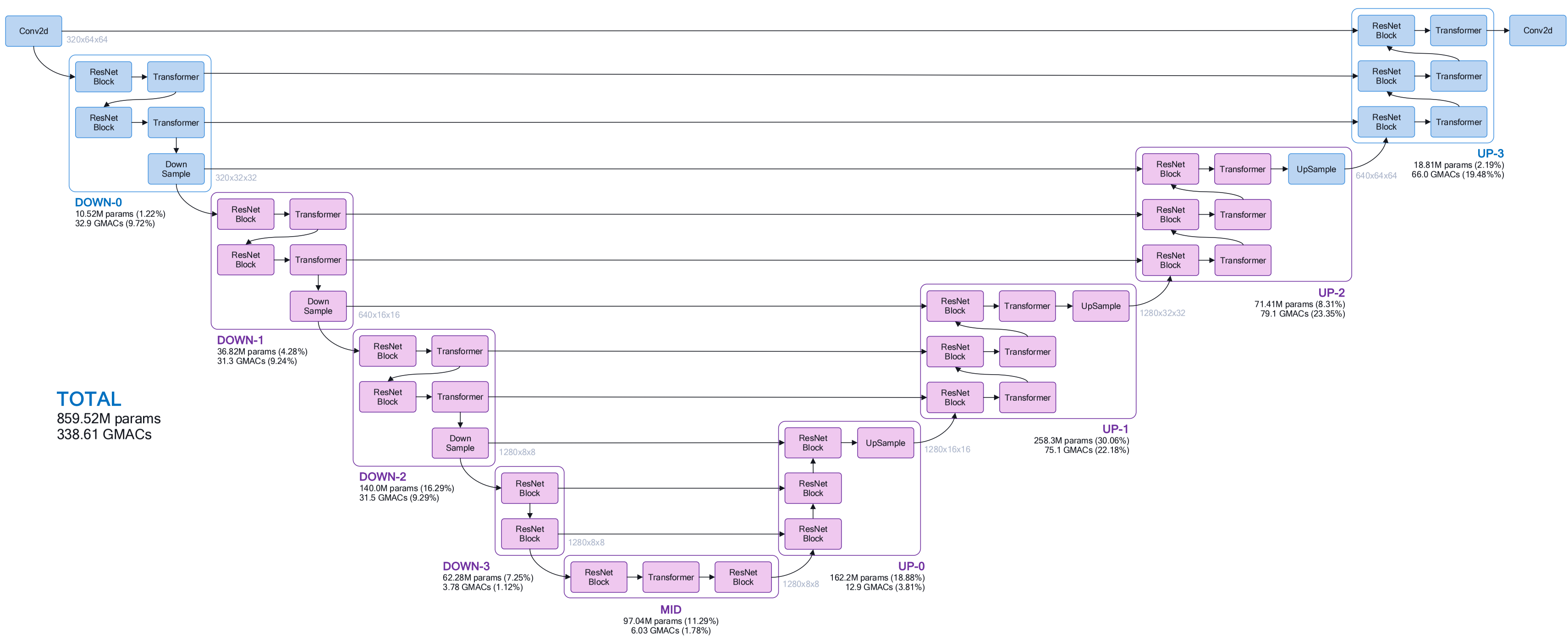


Figure 1. Detailed view of the SD UNet architecture. We replace the pink/purple parts with a lightweight adaptor, the input to which has 32×32 spatial resolution. For the ablations in the main body we also tried leaving DOWN-1 and UP-2 in the higher-resolution path, only replacing blocks below.

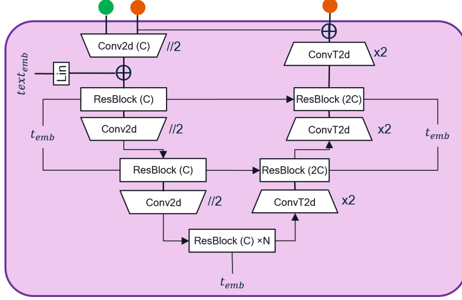


Figure 2. Architecture of a variant of the adaptor: UNet and UNet-light. For UNet we set $C = 640$ and $N = 4$, while for UNet-light we set $C = 96$ and $N = 2$.

	Steps	FID [↓]	CLIP [↑]	GFLOPs
Distilled Efficient UNet	8	25.75	0.297	150
Adaptor Input				
$r_t^{IN} + t_{emb}$	8	40.73	0.262	150
$r_{t+1}^{OUT} + t_{emb}$	8	24.76	0.295	150
$+ r_t^{IN}$	8	24.45	0.295	150
$+ text_{emb}$	8	24.45	0.295	150
Model Distillation				
$r_t^{IN} + t_{emb} + text_{emb}$	8	117.64	0.06	150

Table 2. **Ablation** of adaptor inputs. We use the MSCOCO-2017 dataset, Distilled Efficient UNet as backbone and a clock of 2 (except for Model distillation where we use adaptor for all the steps) . FLOPs are reported for 1 forward step of UNet with adaptor.

Adaptor inputs. We vary the inputs to the adaptor ϕ_θ. In the simplest version, we only input $\boldsymbol{r}_t^{in}$ and the time embedding. It leads to a poor FID and CLIP. Using only $\boldsymbol{r}_{t+1}^{out}$ provides good performance, indicating the importance of using features from previous steps. Adding $\boldsymbol{r}_t^{in}$ helps for a better performance, showcasing the role of the early high-res layers of the UNet. Finally, adding the pooled prompt embedding $text_{emb}$ doesn't change FID and CLIP scores.

Model Distillation In Tab. 2 In previous ablation, we used clock of 2. In this ablation, we explore the option to distill the low resolution layers of ϵ into the adaptor ϕ_θ, for all the steps. Here we train the adaptor in a typical model distillation setting - i.e., the adaptor ϕ_θ receives as input the downsampled features at current timesteps $\boldsymbol{r}_t^{in}$ along with the time and text embeddings t_{emb} and $text_{emb}$. It learns to predict upsampled features at current timestep $\boldsymbol{r}_t^{out}$. During inference, we use the adaptor during all sampling steps. Replacing the lower-resolution layers of ϵ with a lightweight adaptor results in worse performance. It is crucial that the adaptor be used with a clock schedule along with input from a previous upsampled latent.

Timings for different GPU models In Tab. 3 we report latency of different UNet backbones on different GPU models.

Latency [ms]	RTX 3080	RTX 2080Ti	V100	A100
SD v1.5	454	589	453	235
+ Clockwork	341	440	360	183
	(−24.9%)	(−25.3%)	(−20.5%)	(−22.1%)
Eff. UNet	330	427	312	176
+ Clockwork	213	268	212	118
	(−35.5%)	(−37.2%)	(−32.1%)	(−33.0%)
Eff. UNet (distilled)	240	302	245	191
+ Clockwork	154	190	159	122
	(−35.8%)	(−37.1%)	(−35.1%)	(−36.1%)

Table 3. Latency improvements [ms] using Clockwork on different GPU models. All measurements are averaged over 10 runs, using DPM++ with 8 steps and batch size 1 (distilled) or 2 (for classifier-free guidance).

C. Additional perturbation analyses

In Section 3 of the main body, we introduced perturbation experiments to demonstrate how lower-resolution features in diffusion UNets are more robust to perturbations, and thus amenable to distillation with more lightweight components. As a quick reminder, we mix a given representation with a random noise sample by assuming that the feature map is normal $\boldsymbol{f} \sim \mathcal{N}(\mu_f, \sigma_f^2)$. We then draw a random sample $\boldsymbol{z} \sim \mathcal{N}(0, \sigma_f^2)$ and update the feature map with:

$$\boldsymbol{f} \leftarrow \mu_f + \sqrt{\alpha} \cdot (\boldsymbol{f} - \mu_f) + \sqrt{1-\alpha} \cdot \boldsymbol{z} \tag{1}$$

For the example in the main body we set $\alpha = 0.3$, so that the signal is dominated by the noise. In Fig. 3 we show the same plot, but using weaker perturbations with $\alpha = 0.6$. The general behaviour is the same: lower-resolution perturbations result in semantic changes, and are less influential in later steps. High-resolution perturbations result in artifacts, but overall changes are less pronounced than with stronger perturbations.
Interestingly, we can also fully replace feature maps with noise, *i.e.* use $\alpha = 0.0$. The result is shown in Fig. 4. Changes are much stronger than before, but lower-resolution perturbations still result mostly in semantic changes. However, the output is of lower perceptual quality.
For the analysis in the main body, as well as Fig. 3 and Fig. 4, we perturb the output of the three upsampling layers in the SD UNet. We perform the same analysis for other layers in Fig. 5. Specifically, there we perturb the output of the bottleneck layer, the three downsampling layers, and the first convolutional layer of the network (which is also one of the skip connections). Qualitatively, findings remain the same, but perturbation of a given downsampling layer output leads to more semantic changes (as opposed to artifacts) compared to its upsampling counterpart.

Inversion	
SD version	1.5
Sampler	DDIM
Inversion prompt	"a <style> of an <instance>"
Extract reverse	False
Generation	
SD version	1.5
Sampler	DDIM
Guidance scale	15.0 (for both real and fake images)
Negative prompt	"ugly, blurry, black, low res, unrealistic"
τ_A	0.5
τ_f	0.8

Table 4. Plug-and-Play hyper-parameters in inversion and generation. τ_A and τ_f are expressed as fraction of the sampling trajectory. For instance, $\tau_f = 0.8$ means that for the first 80% steps in the generation, convolutional features will be injected. If one uses 10 DDIM steps, this means that for the first 8 steps, convolutional features will be injected.

Finally, we quantify the L2 distance to the unperturbed output as a function of the step where we start perturbation. Fig. 6 corresponds to the perturbations from the main body, while Fig. 7 shows the same but corresponds to the downsampling perturbations of Fig. 5. Confirming what we saw visually, perturbations to low-resolution layers result in smaller changes to the final output than the same perturbations to higher-resolution features.

D. Text-Guided Image Editing

D.1. Implementation Details

We base our implementation of Plug-and-Play (PnP) [2] off of the official pnp-diffusers implementation. We summarize the different hyper-parameters used to generate the results for both the baseline and Clockworkvariant of PnP in Tab. 4. Additionally, while conceptually similar we outline a couple of important differences between what the original paper describes and what the code implements. Since we use this code to compute latency and FLOP, we will go over the differences and explain how both are computed. We refer to Fig. 8 for a visual reference of the implementation of the "pnp-diffusers". For a better understanding, we encourage the reader to compare it to Fig. 2 from the PnP [2] paper.

When are features cached? The paper describes that the source image is first inverted, and only then features are cached during DDIM sampling. They are only cached at sampling step t falling within the injection schedule, which is defined by the two hyper parameters τ_f and τ_A which corresponds to the sampling steps until which feature and self-attention will be injected respectively. The code, instead of caching features during DDIM generation at time steps corresponding to injection schedule, caches during all DDIM inversion steps. This in theory could avoid running DDIM sampling using the source or no prompt However as we will see in the next paragraph, since the features are not directly cached but the latents are, we end up spending the compute on DDIM sampling anyway.

What is cached? The paper describes the caching of spatial features from decoder layers $\mathbf{f}_t^4$ along with their self-attention $\mathbf{A}_t^l$, where 4 and l indicate layer indices. The implementation trades off memory for compute by caching the latent x_t instead, and recomputes the activations on the fly by stacking the cached latent along the batch axis along with an empty prompt. The code does not optimize this operation and stacks such latent irrespective of whether it will be injected, which results in a batch size of 3 throughout the whole sampling trajectory: (1) unconditional cached latent forward (2) latent conditioned on target prompt and (3) latent conditioned on negative prompt. This has implications on the latency and complexity of the solution, and we reflected it on the FLOP count, we show the formula we used in Eq. (3).

Of note, this implementation which caches latents instead of features has a specific advantage for Clockwork, as it enables mismatching inversion and generation steps and clock. During inversion, when the latents are cached, it does not matter whether it is obtained using a full UNet pass or an adaptor pass. During generation, when the features should be injected, the cached latent is simply ran through the UNet to obtain the features on-the-fly. This is illustrated in Fig. 8 where features are injected at step $t+1$ during the generation although the adaptor was used at the corresponding step during inversion.

How do we compute FLOP for PnP? To compute FLOP, we need to understand what data is passed through which network during inversion and generation. Summarizing previous paragraphs, we know that:

- inversion is ran with a batch size of 1 with the source prompt only.
- generation is ran with a batch size of 3. The first element in the batch corresponds to the cached latent and the empty prompt. The other two corresponds to the typical classifier-free guidance UNet calls using the target prompt and negative prompt.
- both during inversion and generation, if the adaptor is used, only the higher-res part of the original UNet will be run, ϵ_H.

Let us denote N and C the number of steps and the clock, the indices I and G standing for inversion and generation respectively. We first count the number of full UNet pass in each, using integer division $N_I^{full} = N_I \,\mathbf{div}\, C_I$ (we follow similar logic for N_G^{full}. Additionally, we use FLOP estimate for a single forward pass with batch size of 1 in

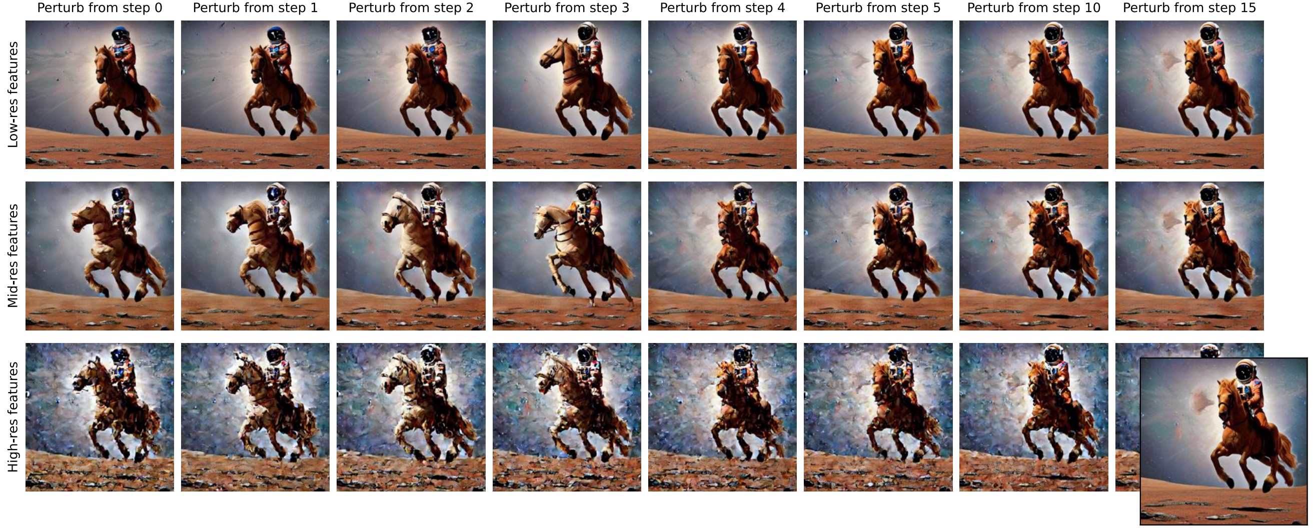


Figure 3. Reproduction of Figure 2 from the main body, using $\alpha = 0.6$ (where Figure 2 uses $\alpha = 0.3$). This corresponds to a weaker perturbation, which results in outputs that are visually closer to the reference.

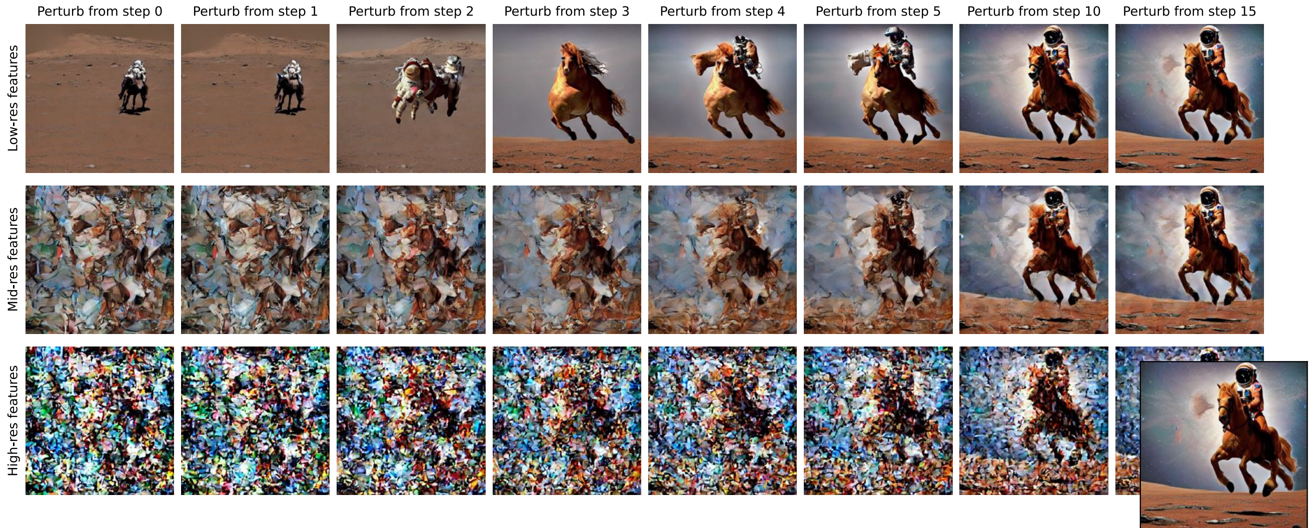


Figure 4. Reproduction of Figure 2 from the main body, using $\alpha = 0.0$ (where Figure 2 uses $\alpha = 0.3$). This corresponds to full perturbation of the representation, *i.e.* the representation is completely replaced by noise in each step. Perturbation of low-resolution features still mostly results in semantic changes, whereas perturbation of higher-resolution features leads to artifacts.

UNet, $F_{\epsilon} = 677.8$ GFLOPs, and UNet with identity adaptor, $F_{\epsilon_H+\phi} = F_{\epsilon_H} = 228.4$ GFLOPs. The estimates are obtained using the DeepSpeed library [1]. Finally, we obtain the FLOP count F as follows:

$$F_I = N_I^{full} \cdot F_{\epsilon} + (N_I - N_I^{full}) \cdot F_{\epsilon_H} \quad (2)$$

$$F_G = 3 \cdot \left(N_G^{full} \cdot F_{\epsilon} + (N_G - N_G^{full}) \cdot F_{\epsilon_H} \right) \quad (3)$$

$$F = F_I + F_G \quad (4)$$

How do we compute latency for PnP? As described in Section 5, we only compute latency of the inversion and generation loops using PyTorch's benchmark utilities. In particular, we exclude from latency computation any "fixed" cost like VAE decoding and text encoding. Additionally, similar to the FLOP computation, we did not perform optimization over the official PnP implementation, which leads to a batch size of 1 in the inversion loop, and a batch size of 3 in the generation loop.

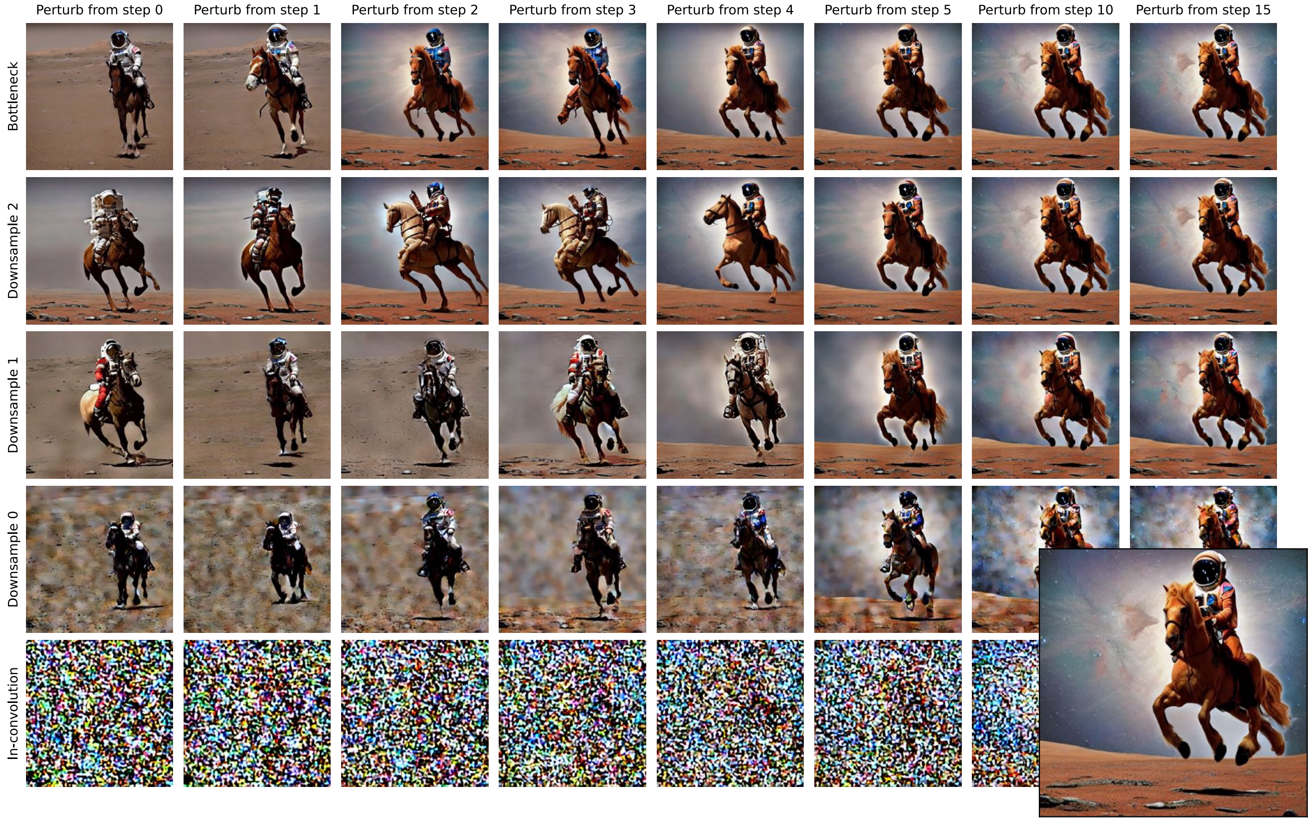


Figure 5. Reproduction of Figure 2 from the main body, perturbation different layers. Figure 2 perturbs the outputs of the 3 upsampling layers in the SD UNet, here we perturb the outputs of the 3 downsampling layers as well as the bottleneck and the first input convolution. Qualitative findings remain the same.

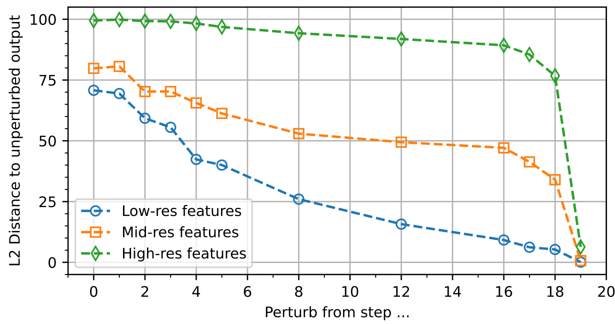


Figure 6. L2 distance to the unperturbed output, when perturbing representations with noise ($\alpha = 0.7$), starting after a given number of steps. This quantifies what is shown visually in Figure 2 in the main body. Lower-resolution representations are much more robust to perturbations, and converge to the unperturbed output faster.

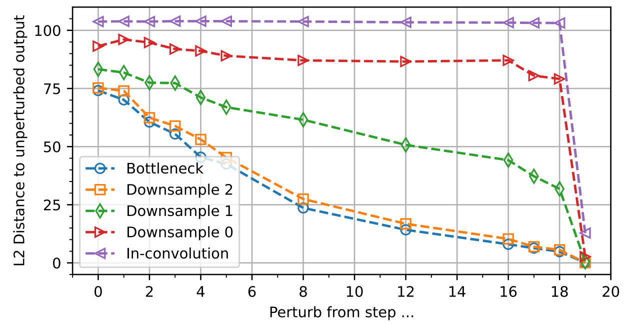


Figure 7. L2 distance to the unperturbed output, when perturbing representations with noise ($\alpha = 0.7$), starting after a given number of steps. This quantifies what is shown visually in Figure 5. Lower-resolution representations are much more robust to perturbations, and converge to the unperturbed output faster.

Interplay between injection and adaptor. The adaptor replaces the lower resolution part of the UNet ϵ_L. Based on where we split the UNet between low- and high-res, it turns out all layers which undergo injection are skipped if adaptor ϕ is ran instead of ϵ_L. Hence, when adaptor is ran during generation it means no features are being injected. As the number of inversion and generation steps decrease,

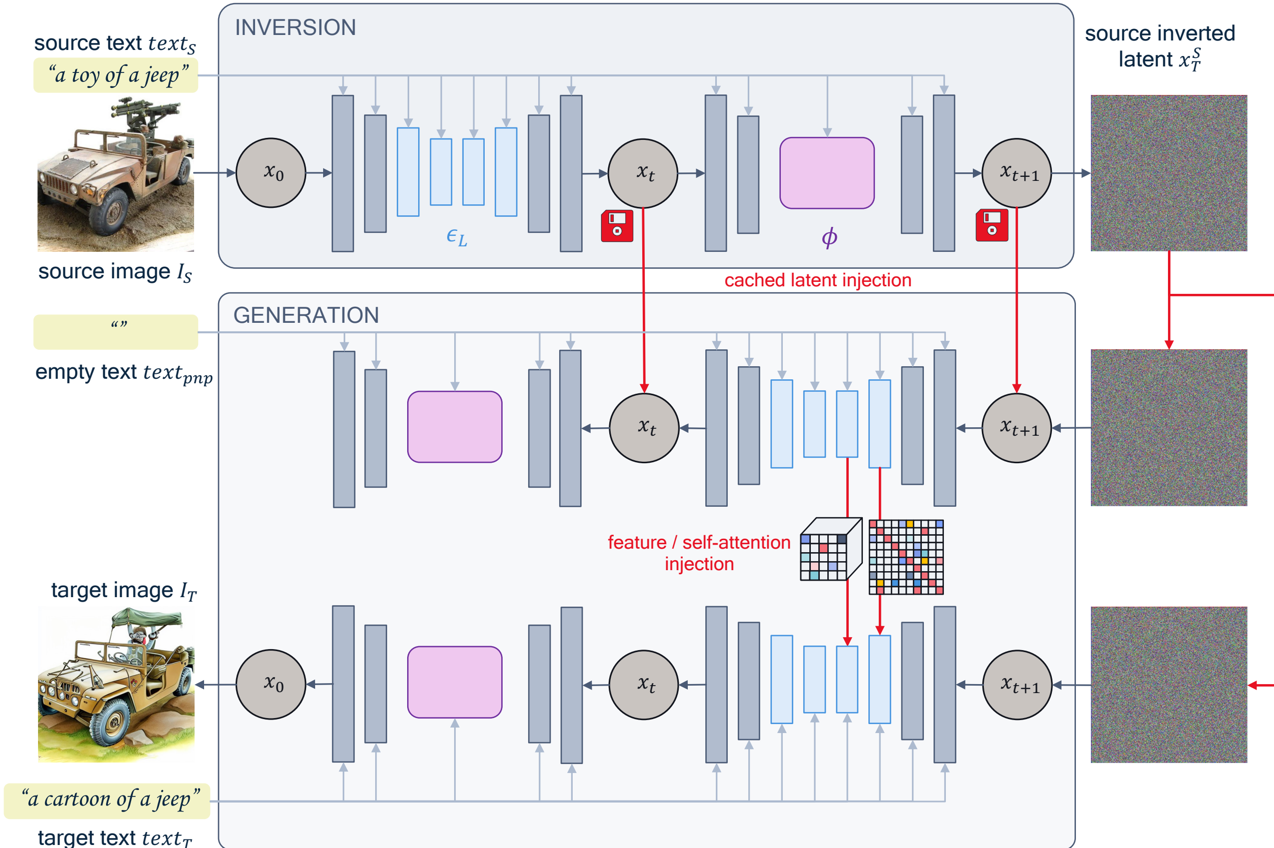


Figure 8. Overview of the actual diffusers implementation of Plug-and-Play, which contrary to what the paper describes *caches latent during inversion, not intermediate features during generation*. The features to be injected are re-computed from the cached latents on-the-fly during DDIM generation sampling. The red arrows indicate injection, the floppy disk icon indicate that only the latent gets cached / saved to disk. Inversion and generation are ran separately, all operations within each are ran in-memory.

the effect of skipping injection are more and more visible, in particular structural guidance degrades. One could look into caching and injecting adaptor features to avoid losing structural guidance. Note however that this would have no effect on complexity, and might only affect PnP + Clockworkperformance in terms of CLIP and DINO scores at lower number of steps. Since optimizing PnP's performance at very low steps was not a focus of the paper, we did not pursue this thread of work.

Possible optimizations. The careful reader might understand that there are low hanging fruits in terms of both latency and FLOP optimizations for PnP. First, if memory would allow, one could cache the actual activations instead of the latent during inversion, which would allow not re-running the latent through the UNet at generation time. Second, it would be simple to modify the generation loop code *not* to stack the cached latent when t does not fall within the injection schedule. If implemented, a substantial amount of FLOP and latency could be saved on the generation, as the default PnP hyper parameters τ_f and τ_A lead to injection in only the first 80% of the sampling trajectory. Note however that both of these optimizations are orthogonal to Clockwork, and would benefit both the baseline and Clockworkimplementations of PnP, which is why we did not implement them.

D.2. Additional Quantitative Results

We provide additional quantitative results for PnP and its Clockworkvariants. In particular, we provide CLIP and DINO scores at different clocks and with a learned ResNet adaptor. In addition to the ImageNet-R-TI2I *real* dataset results, we report scores on ImageNet-R-TI2I *fake* [2].

In the Fig. 9, we can see how larger clock size of 4 enables bigger FLOP savings compared to 2, yet degrade performance at very low number of steps, where both CLIP and DINO scores underperform at 10 inversion and generation steps. It is interesting to see that the learned ResNet adaptor does not outperform nor match the baseline, which is line with our ablation study which shows that Clockwork-

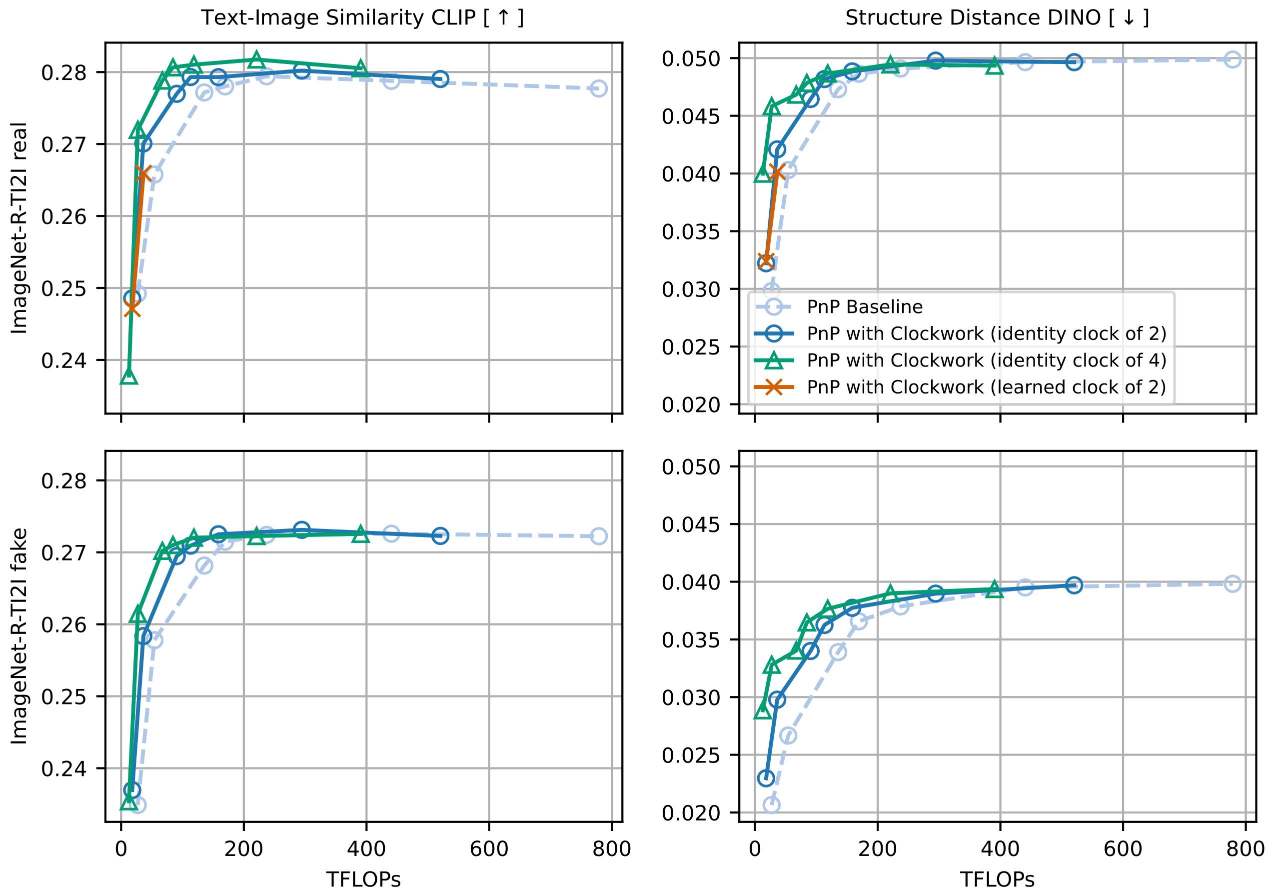


Figure 9. Additional quantitative results on ImageNet-R-TI2I real (top) and fake (bottom) for varying number of DDIM inversion steps: $[10, 20, 25, 50, 100, 200, 500, 1000]$. We use 50 generation steps except for inversion steps below 50 where we use the same number for inversion and generation.

works best for all schedulers but DDIM at very low number of steps, see Tab. 1.

We can see that results transfer well across datasets, where absolute numbers change when going from ImageNet-R-TI2I real (top row) to fake (bottom row) but the relative difference between methods stay the same.

D.3. Additional Qualitative Results

We provide additional qualitative examples for PnP for ImageNet-R-TI2I real in Fig. 10 and Fig. 11. We show examples at 50 DDIM inversion and generation steps.

E. Additional examples

We provide additional example generations in this section. Examples for SD UNet are given in Fig. 12, examples for Efficient UNet in Fig. 13, and those for the distilled Efficient UNet in Fig. 14. In each case the top panel shows the reference without Clockwork and the bottom panel shows generations with Clockwork. Fig. 12 includes the same examples already shown in the main body so that the layout is the same as for the other models for easier comparison.

We also include additional generations in Fig. 15 for $Clockwork$ that operates at 16×16 feature resolution. The generations with $Clockwork$ are relatively not as impressive as $Clockwork$ at 32×32 resolution. We speculate that due to higher dimensional feature representation, we may need an adaptor with higher capacity.

The prompts that were used for the generations are the following (left to right, top to bottom), all taken from the MS-COCO 2017 validation set:

- "a large white bear standing near a rock."
- "a kitten laying over a keyboard on a laptop."
- "the vegetables are cooking in the skillet on the stove."
- "a bright kitchen with tulips on the table and plants by the window "
- "cars waiting at a red traffic light with a dome shaped building in the distance."
- "a big, open room with large windows and wooden floors."

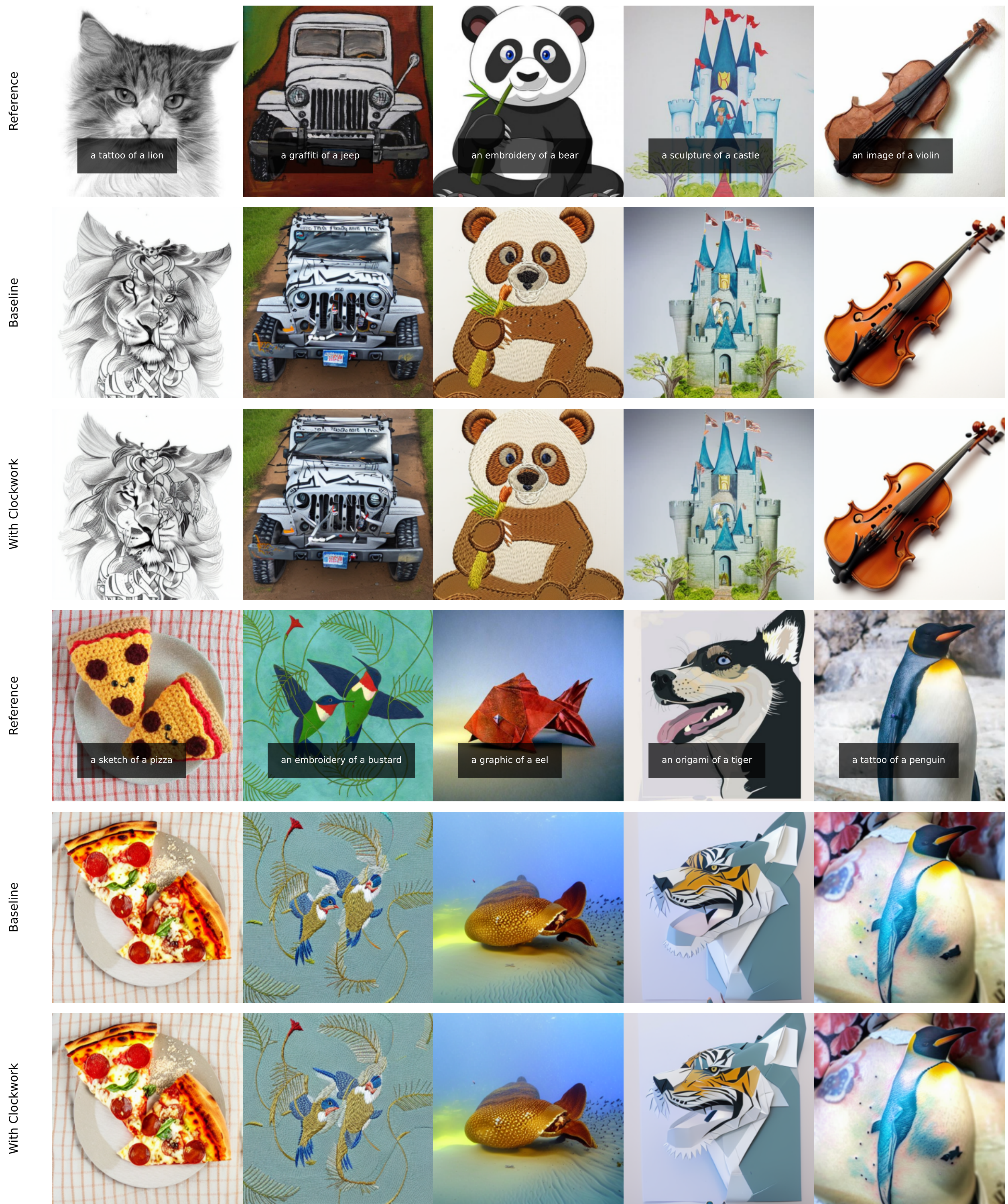


Figure 10. Examples from ImageNet-R-TI2I *real* from Plug-and-Play [2] and its Clockworkvariant. We use 50 DDIM inversion and generation steps, and a clock of 2. Images synthesized with Clockworkare generated 34% faster than the baseline, while being perceptually close if at all distinguishable from baseline.

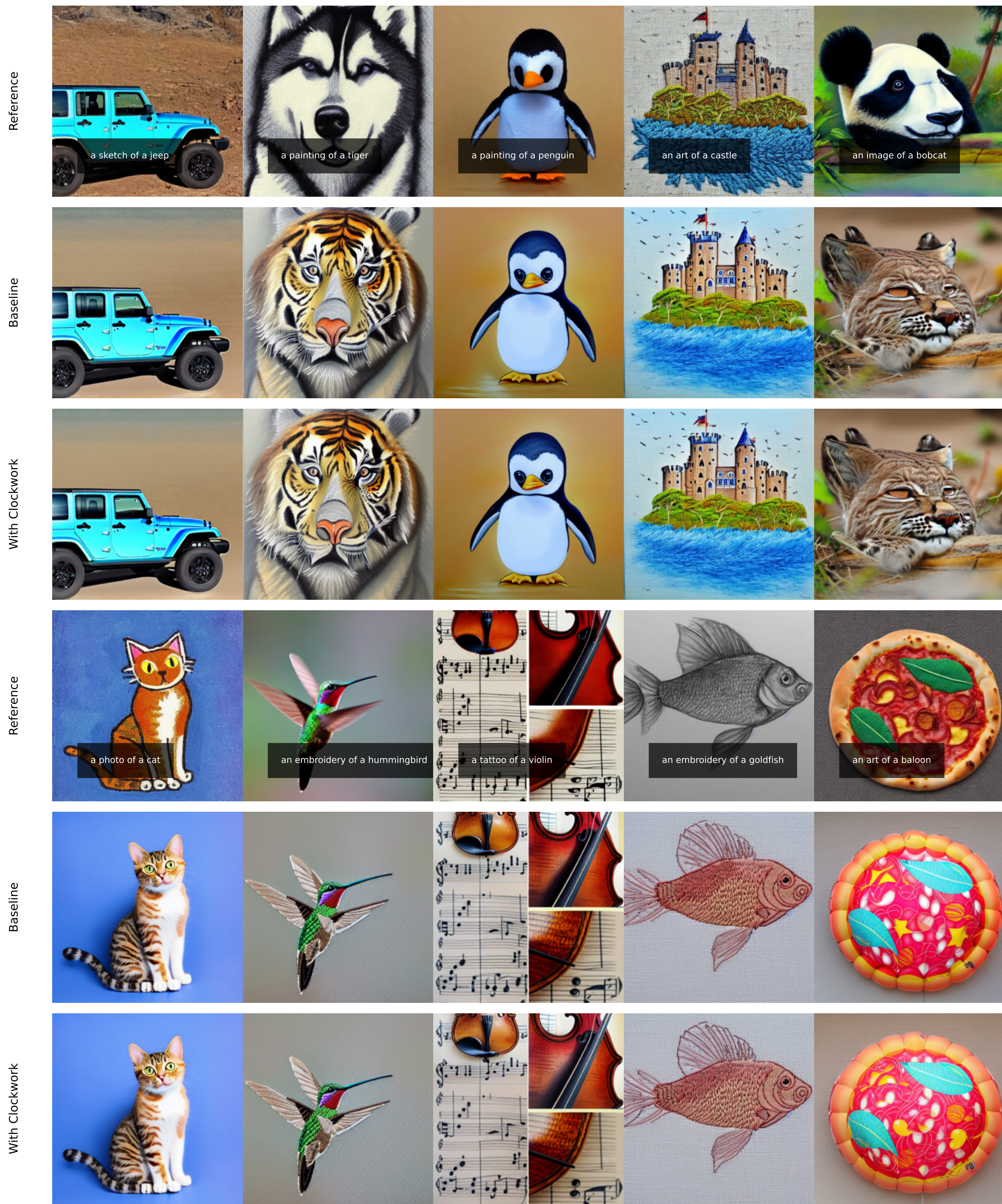


Figure 11. Examples from ImageNet-R-TI2I *fake* from Plug-and-Play [2] and its Clockworkvariant. We use 50 DDIM inversion and generation steps, and a clock of 2. Images synthesized with Clockworkare generated 34% faster than the baseline, while being perceptually close if at all distinguishable from baseline.

- “a grey cat standing in a window with grey lining.”
- “red clouds as sun sets over the ocean”
- “a picnic table with pizza on two trays ”
- “a couple of sandwich slices with lettuce sitting next to condiments.”
- “a piece of pizza sits next to beer in a bottle and glass. ”
- “the bust of a man’s head is next to a vase of flowers.”
- “a view of a bathroom that needs to be fixed up.”
- “a picture of some type of park with benches and no people around.”
- “two containers containing quiche, a salad, apples and a banana on the side.”

References

[1] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020. 5

[2] Narek Tumanyan, Michal Geyer, Shai Bagon, and Tali Dekel. Plug-and-play diffusion features for text-driven image-to-image translation. In *CVPR*, 2023. 4, 7, 9, 10

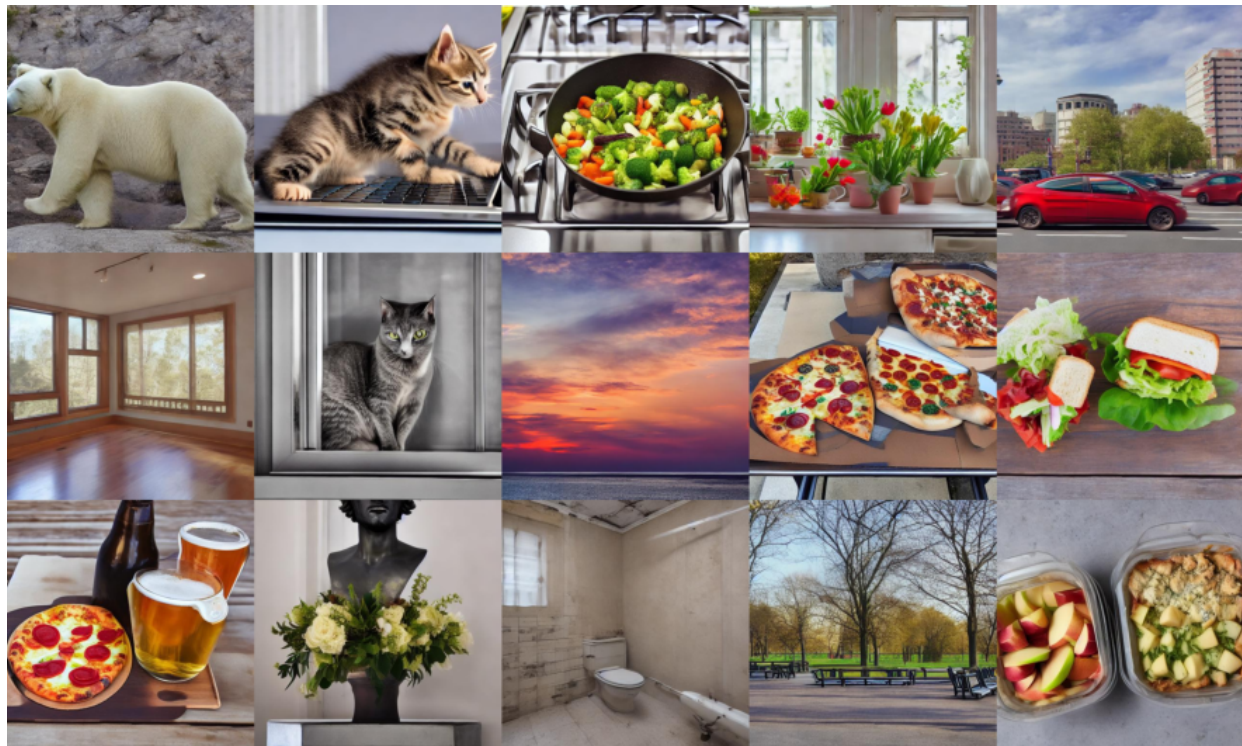
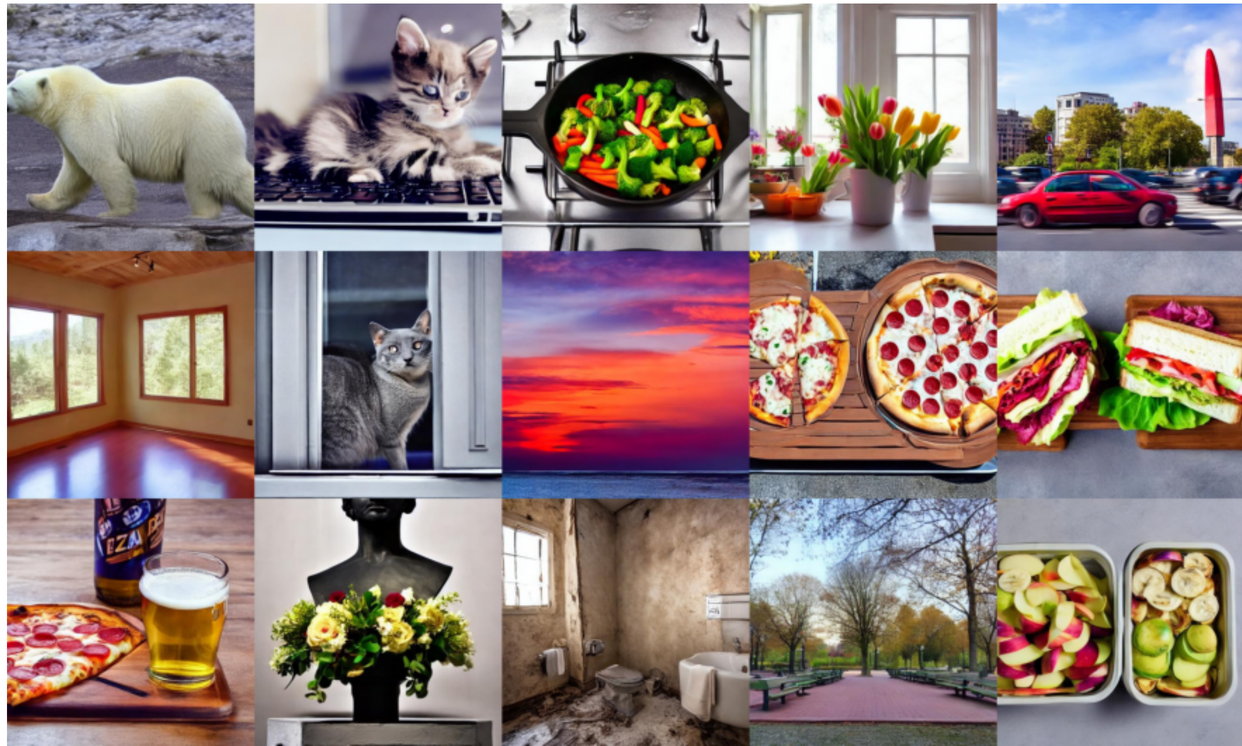


Figure 12. Additional example generations for SD UNet without (top) and with (bottom) Clockwork. We include the examples shown in the main body so that the layout of this figure matches that of Fig. 13 and Fig. 14.



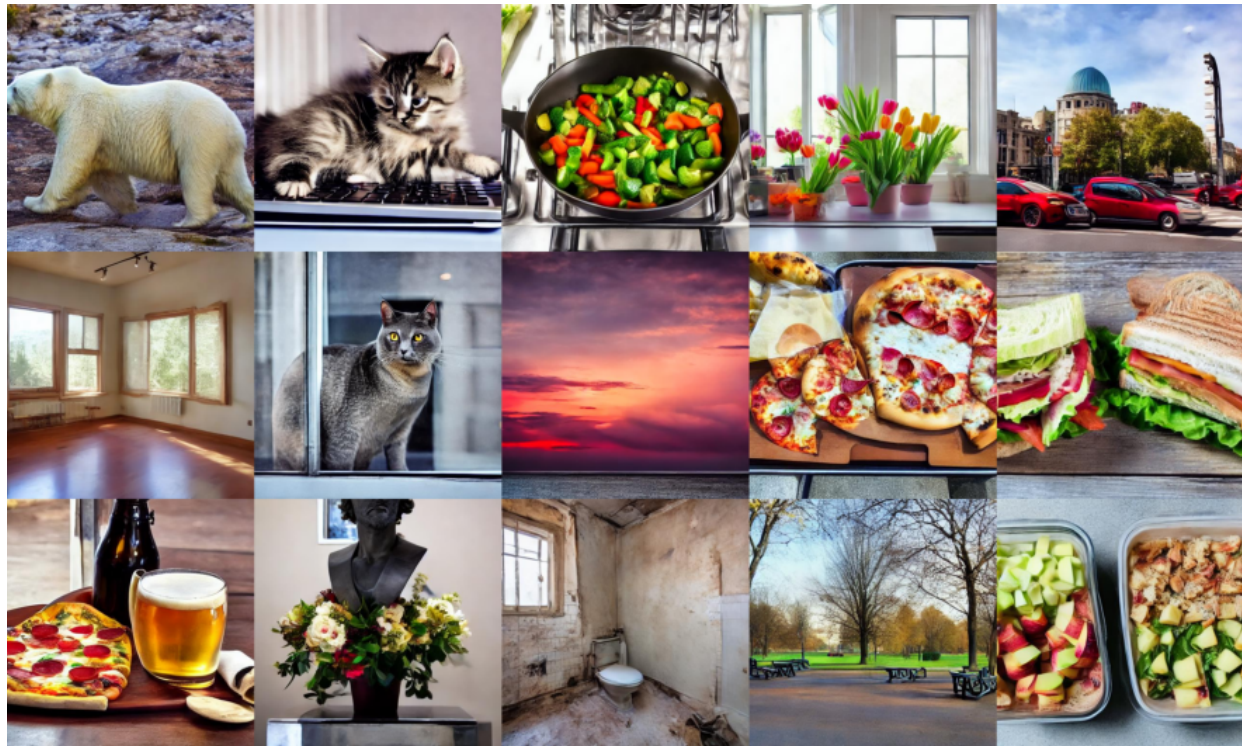


Figure 13. Example generations for Efficient UNet without (top) and with (bottom) Clockwork.

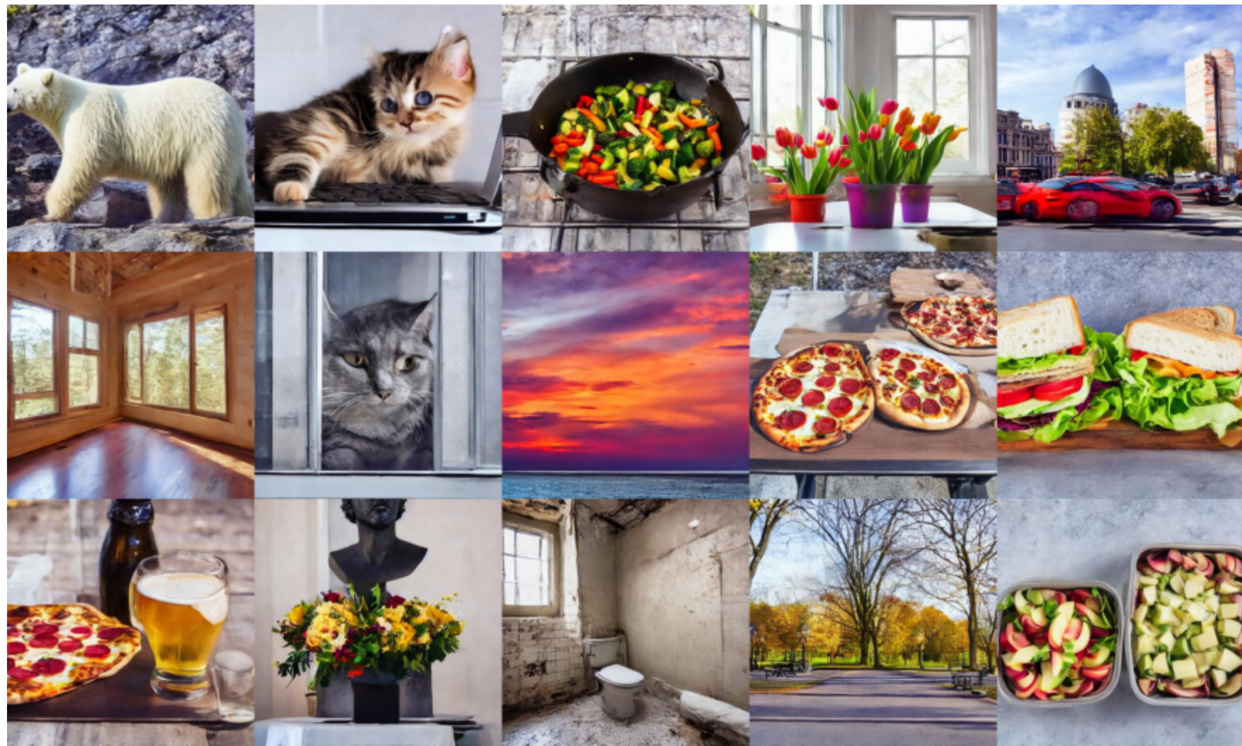
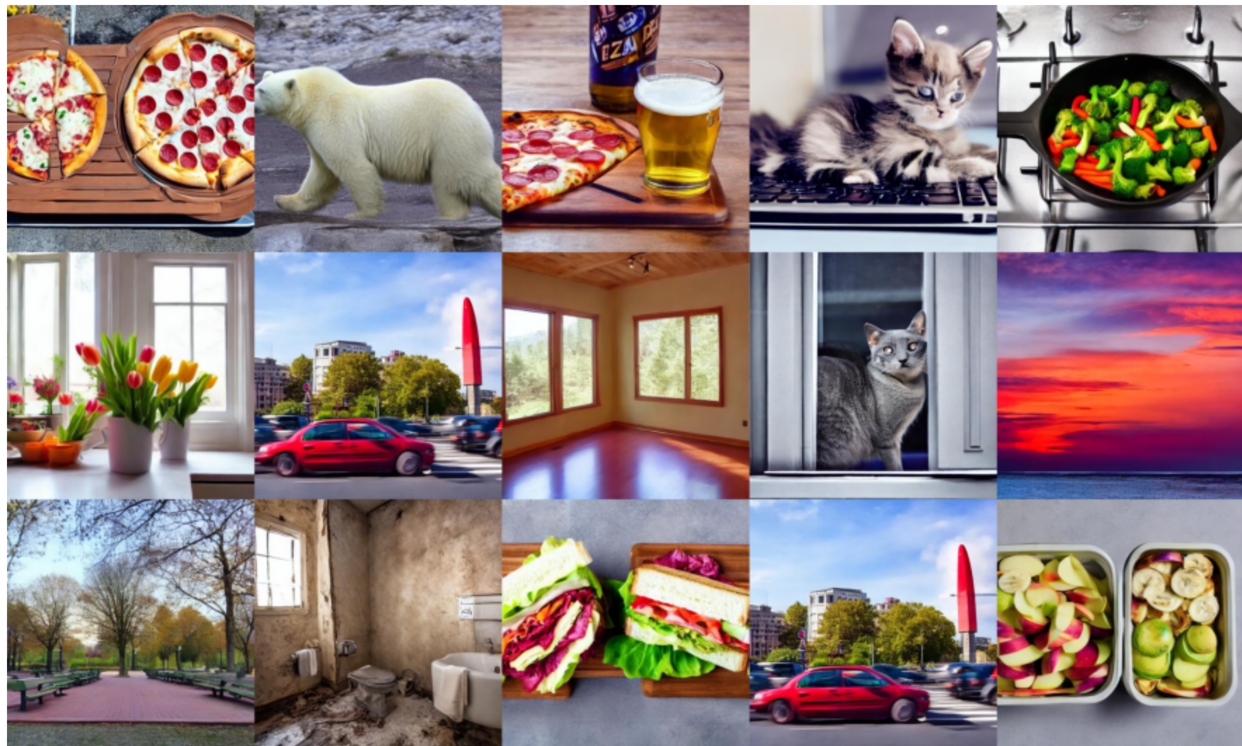


Figure 14. Example generations for Distilled Efficient UNet without (top) and with (bottom) Clockwork.



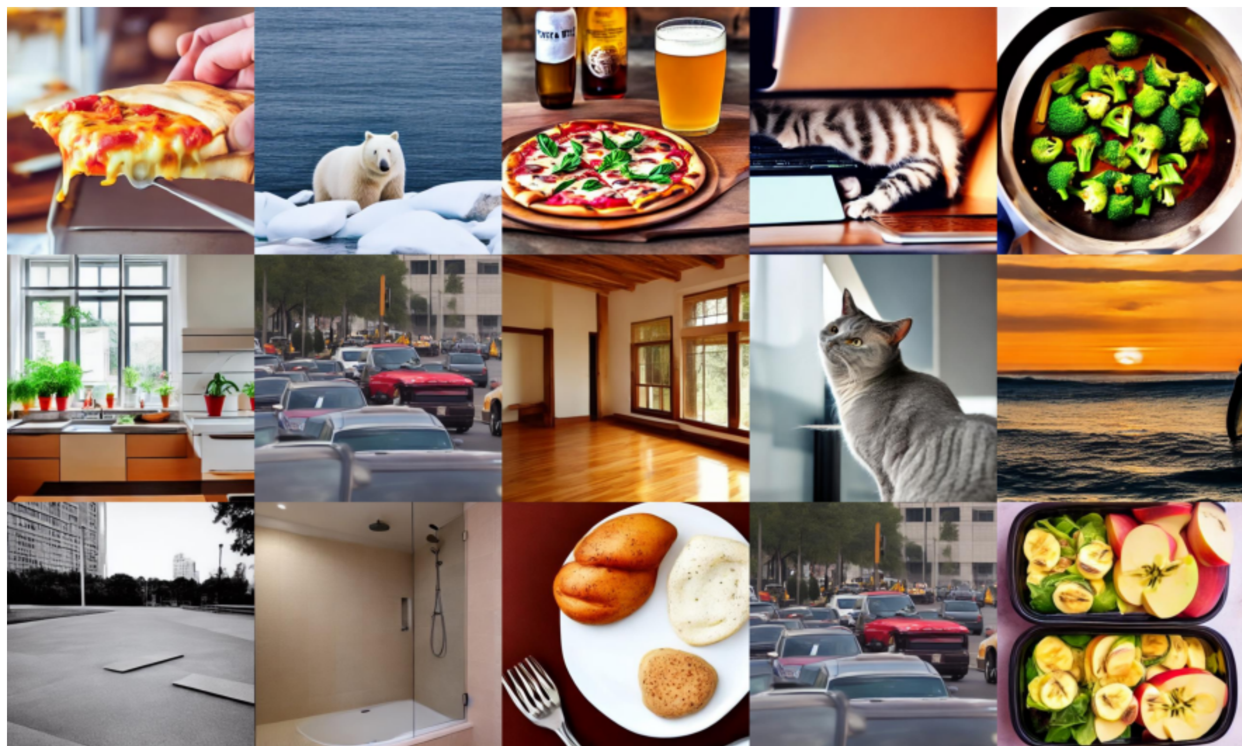


Figure 15. Additional example generations for Efficient Unet without (top) and with (bottom) Clockworkthat is trained for 16×16 feature representation.