




Minimal Convolutional Neural Networks for Temporal Anti Aliasing

K. Herveau¹ , M. Piochowiak^{1,2} , C. Dachsbacher¹ 

¹ Karlsruhe Institute of Technology, Institute for Visualization and Data Analysis, Germany

² Helmholtz Information & Data Science School for Health, Germany

Abstract

Existing deep learning methods for performing temporal anti aliasing (TAA) in rendering are either closed source or rely on upsampling networks with a large operation count which are expensive to evaluate. We propose a simple deep learning architecture for TAA combining only a few common primitives, easy to assemble and to change for application needs. We use a fully-convolutional neural network architecture with recurrent temporal feedback, motion vectors and depth values as input and show that a simple network can produce satisfactory results. Our architecture template, for which we provide code, introduces a method that adapts to different temporal subpixel offsets for accumulation without increasing the operation count. To this end, convolutional layers cycle through a set of different weights per temporal subpixel offset while their operations remain fixed. We analyze the effect of this method on image quality and present different tradeoffs for adapting the architecture. We show that our simple network performs remarkably better than variance clipping TAA, eliminating both flickering and ghosting without performing upsampling.

CCS Concepts

• *Computing methodologies* → *Antialiasing; Neural networks; Rendering;*

1. Introduction

Rendering and especially rasterization fundamentally rely on a sampling process to obtain colors for the pixels in the image frame. This inevitably results in aliasing artifacts because of the discrete nature of this process. In the past, many different techniques in the field of anti aliasing have been developed to reduce or remove these artifacts. Ideally, aliasing is reduced by increasing the sampling rate and combining multiple samples for a single pixel color, either in the form of supersampling or multi-sample anti aliasing [Ake93]. However, this approach results in higher computational load and memory consumption. Since the introduction of deferred shading rendering pipelines, such methods are no longer feasible and temporal anti aliasing (TAA) emerged as one of the most popular anti aliasing variants [YLS20]. TAA solves the aliasing problem by distributing subpixel samples over multiple frames which are accumulated after a reprojection step. Alternative approaches rely on spatial filtering by identifying aliased regions - such as jagged edges - and blurring them with an adaptive kernel [Res09]. Both methods are prone to overblurring and TAA can lead to visible ghosting artifacts.

In recent years, neural networks were utilized to solve the aliasing problem while reducing ghosting and artifacts. Neural solutions mainly rely on explicitly recreating additional subpixel samples through neural supersampling [XNC*20, Liu20]. This makes larger framebuffers and subsequently large networks necessary. In this paper, we propose a minimal, fully-convolutional neural

network architecture that utilizes kernel prediction without supersampling to perform anti aliasing. Kernel prediction allows the network to output per pixel filter kernels instead of direct colors which, after its successful usage in networks for Monte Carlo denoising [VRM*18], was shown to work well for anti aliasing too [TVLF20]. Our architecture outputs a temporal and a spatial kernel, allowing for temporal supersampling as common in TAA as well as for spatial filtering. We show that our small network reduces ghosting and overblurring artifacts in comparison with TAA and has a high temporal stability. Our implementation is compact in its description, highly optimizable and yields promising results. In particular, our contributions are:

- a minimal neural network architecture for temporal anti aliasing,
- a method to let networks adapt to a set of temporal subpixel jitters from renderers without increasing their operation count,
- an analysis of adaptations on different architecture parts to gain insights into the requirements for TAA networks.

In section 2 we give an overview of related works from anti aliasing for real time rendering and deep learning solutions from denoising and supersampling. We present fundamentals required for performing temporal anti aliasing with neural networks in section 3 and explain our architecture in section 4 and its training in section 5. Finally, we evaluate the architecture with regards to its anti aliasing properties, reduction of artifacts, and operational cost including a comparison with a typical non-neural TAA solution.

2. Related Work

Anti aliasing has a long history in computer graphics and rendering. Classically, the aliasing problem was countered using multi-sample anti aliasing [Ake93] (MSAA). But in combination with deferred rendering pipelines, MSAA lost its practicality and became less common as it leads to high loads on memory bandwidth [YLS20, CMFL15]. Single-frame methods that rely on morphological filters [Res09] (MLAA) and edge detection [Lot09] (FXAA) are usable in combination with deferred rendering. However, they are prone to overblurring and can misidentify patterns as aliasing since they do not solve the fundamental undersampling. For that reason, temporal anti aliasing [YNS*09] (TAA) is mostly used in modern real time rendering engines. Lei et al. [YLS20] present an excellent survey of TAA methods. As opposed to applying a locally higher sampling rate within a single frame as in MSAA, TAA accumulates samples over time. To this end, a moving subpixel jitter is applied on the camera projection during rendering and the reprojected output of consecutive frames is averaged. The extensions of morphological anti aliasing proposed by Jimenez et al. also include temporal accumulation for that reason [JESG12]. Commonly, a Halton 2,3 sequence is used for subpixel jitters [Kar14] which we also include in our approach. For moving cameras, reprojection using motion vectors as well as a rejection mechanism for previous samples that takes occlusion or changing illumination into account are required. This rejection is often heuristically solved, by clamping or clipping color history to a color space bounding box of pixels in a local neighborhood of the new sample [Kar14]. Salvi presents variance clipping TAA which takes the variance of color samples in such a neighborhood into account for clipping and produces comparably good results [Sal16]. TAA contains many challenges that need to be tuned manually [Ped16], including motion vector accuracy, reprojection, identifying stale history for rejection, or finding well performing blending weights [ZLY*21]. Classic approaches often suffer from blurry output images or flickering from inaccurate history rejection [Ped16]. In recent years, deep learning solutions became popular for handling different subsampling problems with early techniques already mentioning anti aliasing [NAM*17] while commonly being applied to problems like denoising [CKS*17, HMS*20, TVLF20] and upsampling [Wat20, Liu20, XNC*20].

Previous results from neural denoising showed that the prediction of local filter weights per pixel instead of a direct color yields better image quality is trained faster, a method called kernel prediction [HY21]. Kalantari et al. [KBS15] propose this method for offline denoising of path traced images while Vogel et al. [VRM*18] extended the approach with motion reprojection. Kernel prediction was subsequently adapted for neural Monte Carlo denoising in real time applications. Hasselgren et al. [HMS*20] use a combined kernel prediction denoiser and direct sample guidance network. Fan et al. [FWHB21] compute an encoding of filter kernels to reduce the typically large number of weights of the kernel output layers in their networks. While computing TAA instead of denoising, our approach uses kernel prediction instead of directly outputting pixel colors.

Supersampling differs from anti aliasing in the sense that it creates a high resolution output from a low resolution input. This high

resolution output could be filtered down to form a low resolution image without aliasing, but it is usually directly used. Anti aliasing instead directly outputs the low resolution anti aliased image without explicitly creating the additional subpixel samples for an input image. Early experiments for neural temporal anti aliasing without upsampling were presented by Salvi [Sal17]. Even though non-deep learning solutions are still part of active research [AMD20], neural solutions are becoming fast enough for real time and are currently used in a wide range of video games and game engines. DLSS [Liu20] uses motion reprojection and temporal accumulation for neural supersampling. The DLSS framework also features a variant that directly performs anti aliasing called DLAA. Xiao et al. [XNC*20] present a large architecture that contains individual networks for processing the last 8 temporal subpixel samples. Our network does not have individual network paths for different frames to keep the number of required operations for computing pixel kernels low. Instead, we rotate through different sets of network weights linked to the current subpixel jitter index to handle samples independently in the temporal accumulation. Thomas et al. [TVLF20] significantly downscale the computational complexity of an image reconstruction network through integer quantization.

Deep learning solutions for TAA solve problems that non-neural approaches face, mainly overblurring, ghosting, and flickering, but at the same time are expensive to compute because they rely on a larger number of instructions per pixel. In this paper, we investigate a minimal and simple convolutional kernel prediction network that is able to outperform variance clipping TAA without requiring manual finetuning, but at the same time does not rely on a high instruction count.

3. Background

Spatial single-frame anti aliasing uses a variety of methods and often relies on morphological information. Edge detection and smoothing are commonly used to remove aliasing artifacts [Lot09, Res09]. This produces decent results, but often leads to an over-smoothing of edges or details. The way TAA incorporates temporal information requires a variety of techniques to be viable such as motion vector reprojection and history rejection which are quite cheap to compute. It is typical to apply a cheaper anti aliasing algorithm on top of TAA, such as FXAA, which effectively produces one spatial pass and one pass using temporal information. This can result in overblurring or ghosting artifacts. To build a powerful neural network architecture, we want to allow the network to combine these two data streams, spatial and temporal, and create an output based on both at once instead of two separated passes. The reference images for training the networks are produced from renderings at higher resolutions that are downsampled. To gather multiple samples per pixel, pixel jittering is always used, i.e. the subpixel position during rasterization or ray casting is offset from the center. We discuss this in more detail in Section 5.

3.1. Temporal Accumulation

Motion vectors are obtained directly out of the renderer and can be computed using the camera projection matrix from the previous

frame. Each pixel has a motion vector $m = (x, y)$ representing the offset of that pixel from the previous to the current frame measured in decimal point pixels. The equation of the reprojected color values R as a function of the input image I is:

$$R_{i,j} = I(i - m_{i,j,x}, j - m_{i,j,y}). \quad (1)$$

This reprojection queries arbitrary positions in the input image, requiring interpolation. By default, most frameworks offer bilinear interpolation. Due to the artifacts caused by bilinear interpolation, we decided to use Catmull-Rom interpolation as a higher order interpolation scheme. This is a good compromise as it removes most artifacts from bilinear interpolation but does not introduce ringing and is relatively cheap [YLS20]. In our case, the motion vectors are first computed from the difference between the current view’s depth buffer position, and its backprojected version in the previous position’s camera UV space. All other specific motions overwrite the existing motion vectors. Typically in TAA, the color framebuffer of a new frame C is blended with the reprojected history buffer R using an exponential moving average [YNS*09] as

$$I_{i,j} = \alpha \cdot C_{i,j} + (1 - \alpha) \cdot R_{i,j} \quad (2)$$

where alpha is a constant blend factor. It is possible to vary alpha per pixel, for example for rejecting a deprecated history buffer by setting $\alpha = 1$. More commonly, the history buffer R is rectified using clamping or clipping with a bounding box of neighborhood samples from C in color space to prohibit ghosting artifacts from occlusion or illumination changes [Ped16, YLS20]. Variance Clipping TAA, for example, adapts the size of the bounding box based on the first two moments of color samples in the local neighborhood of pixels [Sal16].

3.2. Kernel Prediction

CNNs operating on image data usually directly predict pixel colors which allows the networks to “dream up” features but may induce artifacts like color shifts and lead to longer training times. With kernel prediction [KBS15, TVLF20], instead of generating a color buffer, the CNN outputs one or more K -sized per pixel kernels. These replace the constant α in eq. (2) and are used in a fixed function step to compute a weighted sum of neighboring pixels in several buffers as the final color output I . This limits the network to only transform inputs using existing color information. It is desirable to allow the network to mix multiple input buffers using $n = K \times K$ weights per kernel, requiring several per-pixel kernels. For example, two kernels can be used to interpolate colors from the new frame’s color samples C and the history buffer R :

$$I_{i,j} = \sum_n \alpha_n^C \cdot C_{i+dx_n, j+dy_n} + \sum_n \alpha_n^R \cdot R_{i+dx_n, j+dy_n}, \quad (3)$$

where $(dx_n, dy_n) \in \{x \mid x \in [-\lfloor \frac{K}{2} \rfloor, \lfloor \frac{K}{2} \rfloor]\} \times \{y \mid y \in [-\lfloor \frac{K}{2} \rfloor, \lfloor \frac{K}{2} \rfloor]\}$ and $\sum_n \alpha_n^C + \sum_n \alpha_n^R = 1$. This leads to the combined processing of spatial and temporal filtering that was motivated earlier. The choice of the different inputs is crucial to the performance of the network. The final trainable layer of the network must output the set of all weights α per pixel. This layer is memory intensive as an entire frame of size $W \times H$ pixels accumulated with N K -sized per-pixel kernels requires the generation of $W \times H \times K \times K \times N$ interpolation weights.

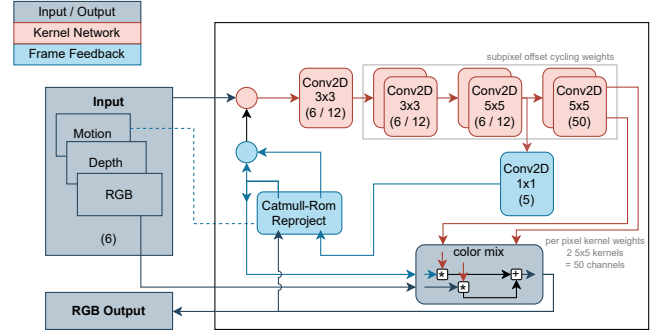


Figure 1: Architecture of the TAA network with layer counts in parentheses. The reprojected feedback uses Catmull-Rom interpolation and is implemented as a recurrent network layer. Three of the stages use a different set of layer weights depending on the Halton 2,3 index that is used for subpixel offsetting during rendering. The final layer outputs two 5×5 filter kernels applied to the new RGB buffer and previous network output.

4. CNN with kernel prediction

TAA typically makes use of motion vector and depth buffers. These are used for history rejection to drop an accumulated pixel color from the history buffer that no longer belongs to the same surface. Figure 1 shows the architecture of our network which uses the same data as input. The multi-layer convolutional neural network is embedded in a feedback loop that feeds the new frame’s output color as an additional input to the next frame’s processing. This feedback loop is implemented with a recurrent neural network cell (RNN) that we detail further down.

The main network consists of four convolutional layers that produce two 5×5 sized filter kernels per pixel (red in Figure 1). These kernels are used to compute a weighted sum of pixels in a local neighborhood from the current frame’s color input and the previous frame’s reprojected color output according to Equation (3). Before generating the kernels in the final layer, the network also outputs a five layer sized state carried over to the next frame’s processing alongside the RGB output (blue). In the following, we present the stages of the network in detail.

Network Input The kernel prediction network receives the currently rendered RGB frame in low-dynamic range, normalized to values between 0 and 1. The network works better with values in this range compared to the $[0, 255]$ range. These images are concatenated with per pixel motion vectors and depth values. Additionally, a feedback loop concatenates eight additional channels from the network output of the previous frame to this input. These feedback channels consist of the previously anti-aliased RGB output and five freely trainable channels that are conceptually equivalent to a hidden state of an RNN.

Temporal Feedback Before the last convolutional layer of the kernel prediction network generates the per-pixel kernels, we use a small 1×1 convolution to output five additional channels (blue in Figure 1). In addition to the network’s RGB output, these are carried over to the next frame to store additional temporal information.

In the next frame’s processing, we use the motion information from the renderer to reproject these feedback channels. The reprojection is performed with Catmull-Rom interpolation. This reduces typical bilinear interpolation artifacts, and improves frequency content in the images, which is especially important for convolutions. The code for this part is released separately as a side contribution of this project, as it is not available in Tensorflow. The Catmull-Rom implementation typically used in production is orders of magnitude faster than ours. This has been investigated in the past (e.g. [Kar14] for Catmull-Rom, [SN15] for other kernels), making it out of the scope of our paper. Our implementation is sufficient for proof-of-concept purposes. The RNN cell functions as a way to feed the network its own output. Note that using a loop or successive applications of the model is a valid approach. Choosing the RNN format allows us to use well documented functionalities such as different training window sizes, unrolling, managing hidden states and stateful mode. The total RNN implementation’s hidden state consists of the current anti-aliased RGB output after the *color mix* operation (detailed just after) and the five freely trainable channels. Additionally, we keep track of the currently processed index of the Halton 2,3 sequence that the renderer uses for subpixel jittering.

We unroll the network for training, but for inference a stateful implementation that iteratively processes one frame after the other is used. We observed a noticeable difference in final output quality between stateful and non stateful, the stateful version performing better.

Kernel Prediction Network The kernel prediction network (red in Figure 1) consists of four fully convolutional layers which use zero padding to retain the size of the feature maps. We evaluate the network in two variants using either six or twelve output channels in the first three layers. The last of the four layers outputs the two 5×5 filter kernels. Those kernels are later used in the kernel-weighted *color mix* operation to compute a weighted sum of the pixel neighborhoods in the RGB buffers from the currently rendered frame and the previous anti-aliased network output. The output size of this final layer is equal to the number of elements in both kernels. We use a ReLU activation after each layer except the last one, where we apply a sigmoid activation to normalize the sum of all kernel elements for a given pixel to 1. The hidden layer with a 1×1 convolution uses no activation function. We propose a method to let the network adapt to the cycle of jittering subpixel positions from the renderer: The last three layers of the kernel prediction network use eight different sets of weights $W_0 \dots W_7$. For each index in the jitter sequence j , the same set of weights W_j is always used within the layers. Note that this duplication does not increase the number of operations in the network as only one set of weights is used at a time. Thus, instead of introducing new computation paths this only leads to a slightly higher memory consumption to store the seven additional sets of weights. We show the impact of these sets of weights in section 6.

5. Training

5.1. Dataset

The training data is made of renderings of camera flythroughs in 3 scenes at a resolution of 2560×1440 pixels. These images are

resized to 640×480 pixels using a Blackmann-Harris filter to create reference images without aliasing. Renderings from the same flythroughs at a native $640 \times 480 @24$ resolution are used as training input data. The image sequences contain successions of smooth and sudden movements at different speeds. The scenes have been chosen to contain difficult cases for anti aliasing such as high-frequency geometry, frequent visibility changes, and challenging low-frequency aliasing (almost vertical doorframes etc).

We used 3 different scenes. Staircase contains doors, stairs with empty space between each step, a thin railing, large columns and wide glass windows, projecting shadows. Cafeteria contains very thin geometry (wires) that aren’t properly resolved by the input, (but the reference properly does), straight edges with tables and chairs and difficult occlusions with the cafeteria’s lampshades. Finally, the livingroom contains designer chairs that have a thin black stripe, a couch and a textured floor.

The renderer cycles through 8 different subpixel jitter offsets from a Halton 2,3 sequence. For that reason, training happens on a windows containing 8 frames, ensuring that all jitter indices were considered in the training step once. In particular, this is relevant so that backpropagation is always carried out on each of the 8 different sets of weights that we use in the last three convolutional layers of the kernel prediction network (see section 4). During training, all sequences within a batch start with the same jitter index for performance purposes, but different batches start with different indices.

5.2. Data Processing

The network is trained on patches of 8 consecutive frames with a size of $p_w \times p_h = 128 \times 128$ pixels that are extracted from the dataset sequences. Our data is augmented using channel switching, and randomized patch selection. All possible starting positions are precomputed, then shuffled which ensures that the network processes the entire dataset prior to any repetition while the order of the input varies. For each of the $n_j = 8$ different jitter positions, we store a separate dictionary containing all possible training patches that start with the respective subpixel jitter index. For a sequence of $n_f = 100$ frames and resolution $w \times h = 640 \times 480$ pixels, $(w - p_w)(h - p_h)(n_f - (n_j - 1)) = (640 - 128)(480 - 128)(100 - 7) = 16,760,832$ patches are distributed over these eight dictionaries. We use several sequences, totaling more than 500 frames. The number of possible starting positions is so large that the processing of all patches far surpasses available training time. This motivates us to define an epoch as 1600 processed patches.

Neural Networks or machine learning models are susceptible to overfitting. Main causes of overfitting in our case could be models with too many parameters or degrees of freedom, low data variety, long training times. In our case, the network typically trains on less than one percent of the total training data but still performs very well on the entire dataset. Our parameter count is very low compared to the complexity of the task and the diversity of the dataset. This leads us to believe that our network is at very low risk of overfitting, and in practice, we did not notice any overfitting.

5.3. Training Loss

The loss for training a temporal anti aliasing network has to account for several effects. We penalize undesirable artifacts and reward closeness to the reference. Our general loss function is split into three parts as known from other image reconstruction networks [KBS15, TVLF20, HMS*20]. Different weights control the influence of the individual losses:

- The *reference loss* L_r is the sum of all pixel-wise absolute L1 differences between the network output and the reference image.
- For the *gradient loss* L_g , we compute this difference on the gradient images of the reference and output images in x and y direction.
- Additionally, we compute a *temporal loss* L_t as the pixel-wise difference of temporal gradients in the reference and the network output. L_t is used to reduce flickering and other temporal artifacts.

We also experimented with using Nvidia’s \mathcal{F} LIP [ANA*20] as a training loss, but this led to poor convergence and significantly longer training times. The main reason is the gaussian filters used in the \mathcal{F} LIP metric spread out the contribution of each pixels, causing a blur that requires more training. The parameter `pixel_per_degree` was tested on a sensible range of values and we observed no obvious benefit of using \mathcal{F} LIP for training. We introduce a per pixel aliasing mask M for the training images which contains a value between 0 and 1 per pixel. Values closer to one identify pixels containing aliasing artifacts. The mask is computed by summing up all RGB channel-wise differences between the training input and reference images. Additionally, we multiply the mask with a sensitivity parameter and clip values to $[0, 1]$. As a last step, a 3×3 dilation is applied to guarantee that all pixels of aliasing regions are considered. We use this mask to create two versions of our three loss functions each by weighting them either with M or $(1 - M)$. The first identifies loss in regions containing aliasing which correlates to the network lacking anti aliasing properties. The latter identifies loss in regions where the input and reference images were already close. Here, the network should at best avoid any alteration of the input data at all. A high loss in these regions usually stems from artifacts like ghosting or overblurring. We give all these parts different weights when computing the final loss.

We choose these weights to adjust the influence of each loss, allowing us to study a balanced situation or a situation where one loss dominates the others. This reweighting is necessary to consider all losses in the backpropagation as L_t in particular is roughly ten times smaller than L_r and L_g . Finally, we adapt the loss given the aliasing mask M so that image regions containing aliasing are 1.5 times as much important as the non-aliased regions. Though we did not conduct a large scale hyperparameter investigation, all six loss weights could be fine tuned and even modified through training with dedicated frameworks. Since we train on patches consisting of 8 frames, we compute the loss only on the last output frame to let the network accumulate temporal information. The training network uses an Adam optimizer [KB14] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$ and clamping gradients to 1. Our initial learning rate of 0.001 follows an exponential decay with 1000 decay steps and a decay rate of 0.96. We train on a batch size of 24 where each batch

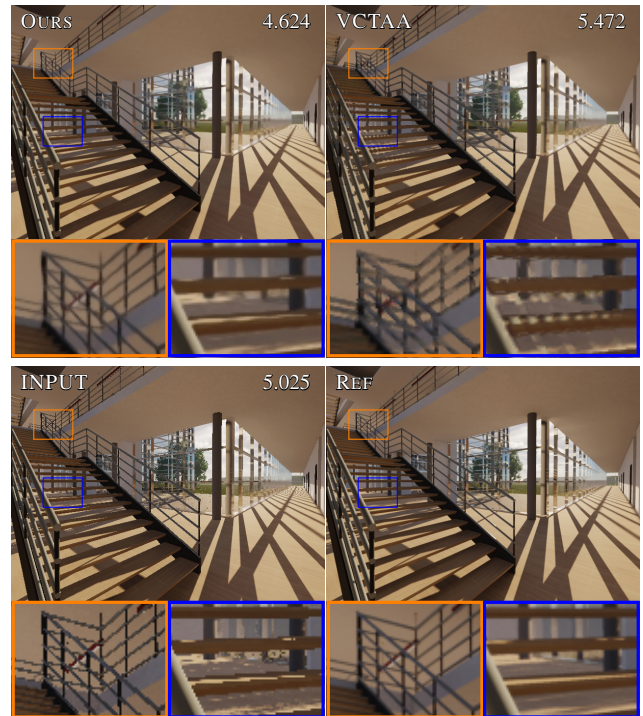


Figure 2: Numbers are RMSE. The edges are very consistently anti-aliased without blurring (left) while vctaa noticeably degrades the sharpness in the orange closeup. The artifact on the lighting of the stairs (center) is not present in our method.



Figure 3: Numbers are RMSE. Comparison of behavior during fast camera movement. Our method resolves correctly the railing and stairs that are distorted by VCTAA.

element consists of 8 successive frames of the same portion of the image in subsequent frames.

6. Results

We evaluated the network on three different scenes. A large staircase with many long and thin straight lines. A cafeteria, with difficult thin geometry on the lamps. A studio with some textures and a transition through a glass window. We compare our results with variance clipping TAA and with our reference. See also our supplementary video.

In Figure 2, we present a direct comparison between our method, variance clipping TAA [Sal16] and the input including root-mean-square errors (RMSE). Our method retains crisp edges without overblurring.

We did not detect ghosting artifacts using our method. Figure 3

shows that variance clipping TAA (VCTAA) tends to blow up thin features during fast movement, while our method retains proper edges.

The flickering produced by our final network is reduced and even better than the reference. The flickering in the reference comes from the large amount of occlusions and disocclusions happening quickly on a relatively low resolution. Our network creates a much smoother result. The numerous occlusions happening in the overlapping stairs and railing area produce high temporal frequencies that our network has been trained to reduce via the temporal weights in our loss function.

resolution	Catmull-Rom	model	color mix
2560x1440	350.241	2.892	9.204
1920x1080	264.871	2.827	8.789
640x480	180.294	2.368	8.841

Table 1: In milliseconds (ms). Our Tensorflow implementation of Catmull-Rom is comparably unoptimized. The color mix combines the 50 output channels with the the input image and the reprojected previous frame. Measured on an RTX 3060 with Tensorflow python framework using 32bit computation.

Our unoptimized implementation takes almost 200ms per frame, Table 1. This time is dominated by our naive implementation of Catmull-Rom, as this operation was not natively supported by Tensorflow. The inference time is remarkably fast with no more than 3ms. This is before any optimizations such as TensorRT [Nvi23], which performs remarkably well on convolutional networks (offering more than $6\times$ speedups) and using reduced precisions, e.g. by using half floats where applicable (half floats should not be used with sigmoids). Our feedback loop implementation as a recurrent neural network (RNN) creates compatibility issues with TensorRT which can be resolved by replacing the RNN with a simpler cell that streamlines the data of the previous frame to the next. We believe that our proof-of-concept network can be implemented in real time applications if modest optimizations can be performed.

We computed the total number of operations required for each layer, following the equations for counting multiply-add-accumulate and add operations:

$$\begin{aligned} MAC &= K^2 * I * O \\ ADD &= O \\ FLOPS &= (2 * MAC + ADD) * W * H \end{aligned}$$

with the kernel size K , the number of input and output channels I and O , and W, H the width and height of the image. We evaluate the same network we used for the measurements in section 6.

With our smaller network, with six internal channels, the count drops to $18740 * W * H$. The activation functions (ReLU) of the first layers and the softmax of the final layer add in total 2000 operations per pixel, largely dominated by the cost of exponentiation. The color mix operation takes 100 operations per pixel. The Catmull-Rom interpolation requires less than a 100 operations per pixel. Better and faster alternatives to Catmull-Rom interpolation exist, we only used it as a proof of concept. The total number of floating point operations is below 50000 per pixel. Despite being

Layer	K	I	O	Total
layer 1	3	11	12	2388
layer 2	3	12	12	2604
layer 3	5	12	12	7212
kernel	5	12	50	30050
hidden	1	12	5	125
total				42379

Table 2: Computation details for the number of FLOPs of the network itself (upper bound). 40k FLOPs/pixel

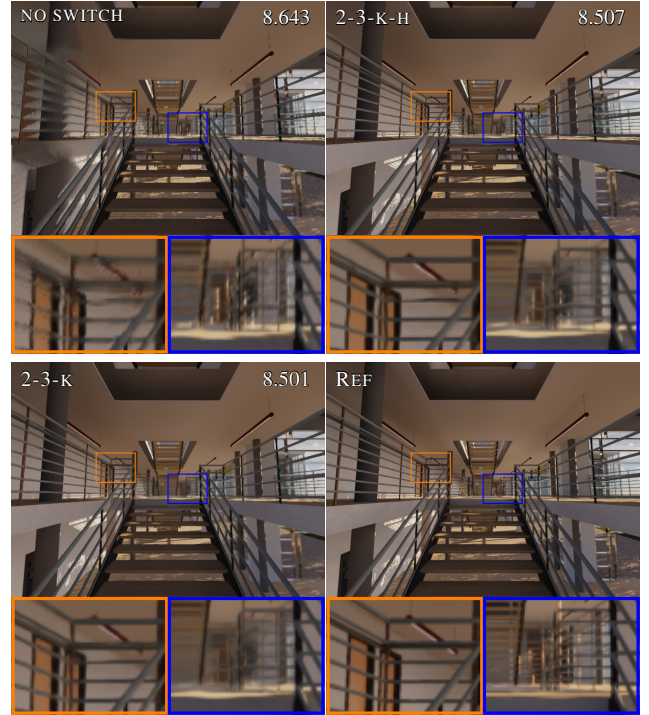


Figure 4: Top right: RMSE. Comparison of the effect of using several sets of weights for one layer and switching them in sync with the jitter id cycle. **left:** constant weights. **center** switching sets for all layers except the first. **right** only switching sets for layers 2, 3, kernel prediction (our default configuration).

sizeable amount it is by no means out of reach for a well-optimized pipeline. The vast majority of these operations is in the kernel prediction layer, which suggests that adopting another strategy could result in great improvement in performance.

6.1. Jitter

The absence of subpixel jitter specific sets of weights for the last three convolution layers heavily reduces the quality of the anti aliasing as shown in Figure 4. The previous frame being reprojected has a different jitter pattern than the current frame, causing the network to improperly combine them. The network would need to adapt to each different combination of jitter patterns and associated motion vectors. Having one set of weights per jitter position removes this burden entirely from the network. These images have been generated by networks with an equal number of train-



Figure 5: Top right: RMSE. Different layer count with equal training epochs. More layers take longer to train and take longer to compute. Less layers lead to subpar quality.

ing epochs. The center image requires more training to perform at least as well as the right image as shown by their lower RMSE. Our method of using a different set of weights for each respective camera subpixel jitter position in the convolution layers performs significantly better than using only one set of weights. These results suggest investigating the use of different sets of weights in combination with the generation of the jitter pattern by the network itself.

6.2. Internal Layers

The number of internal layers and their size plays a big role in the runtime. Chaining layers implies introducing computation dependency, which can make the process slower. Bigger layers on the other hand, benefit fully from parallelism. In our experiments, two layers were insufficient for a well-performing network, while improvement were not noticeable above three.

6.3. Hidden Layers

In addition to the anti aliased output, the network’s feedback loop contains a trainable state that is updated every iteration. The update step has its own layer through which information is stored in the per pixel state given the input data (blue convolutional layer in Figure 1). This memory area has the size of a frame, but has as many channels as we desire. Giving the network enough state space is crucial, but a larger state increases inference time and memory consumption. Our analysis of the hidden layer channels reveals that they essentially consist of various edge detection filters, seemingly

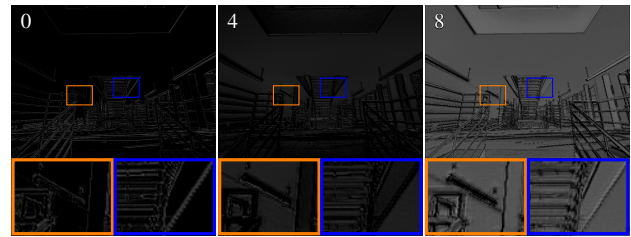


Figure 6: Left: Resembles a directional edge detection filter, the lines on the ceiling can help grasping what function is at play. Center: Another edge detection, but combined with a depth gradient although it is remapped in some way. Note that these two images are quite dark. Right: Seemingly another edge detection but also flattening the non edge parts of the scene.

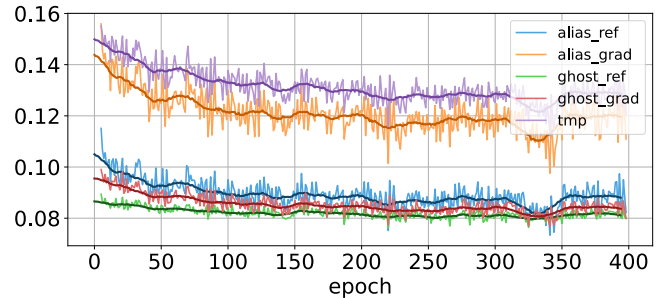


Figure 7: Weights are adjusted for the temporal loss (tmp) to be the most impactful, followed by the gradient loss. We noted better results by having the gradient loss being higher than the reference loss. The alias prefix refers to the loss in image regions with aliasing, while the ghost prefix denotes loss on regions without aliasing which typically occurs because of ghosting artifacts.

decomposing the image, as can be seen in Figure 6 which displays three of the state’s channels. An exact interpretation is impossible, but these images offer us a better understanding of what information these channels are used for, and therefore what information the network may lack in case one reduces the amount of available channels. In our experiments, using more than five channels did not offer significantly better quality.

6.4. Loss Weights

We tested different loss weight values (more details: Section 5.3, see Figure 7). The training was performed on an Nvidia Titan V, and took 8h. One noticeable finding is that having a dominant temporal loss increases image stability and reduces flickering, even to levels better than the reference. Temporal effects are difficult to convey in words, see the accompanying video, notably the ascent of the staircase. During the ascent, the pillars behind the bars on the top of the image can be seen quickly popping in and out, and later, the geometry behind the railing and stairs creates similar patterns.

6.5. Discussion

Our method only focuses on anti aliasing, but in most cases, several post processing techniques are performed on the output. It was our choice to not include denoising or upscaling, but it remains a limitation of this method for its adoption in industry context. It could

very well be adapted to perform both denoising and anti aliasing, but this is out of the scope of this paper.

One weaker point of this technique is texture reconstruction. We noted a slight loss of quality on complex and slanted textures. This could perhaps be handled by a sharpening pass.

Our method improves the quality of low frequency image features, such as doorframes that are almost vertical, or slightly slanted objects in general. Low frequency context is very difficult to deal with for almost all anti aliasing techniques because of the use of regular grids. For this reason, the helpfulness of camera subpixel offset jittering is limited in traditional anti aliasing. Since the jittering happens identically on all pixels within the same framebuffer, the frequency content only changes after each frame. Exponential moving averages and neighborhood rectifications, as commonly used, cannot capture these changes efficiently enough. Applying different subpixel offsets to pixels even within the frame would allow for a better distribution of samples and for a better subsequent reconstruction for most of these cases. One such possibility is full raytracing, which would allow the use of low discrepancy sequences or even optimized sequences such as step blue noise. One possible future work would be to use a CNN directly on the raytracing output (ray position in frame and rgb color) and allow the network to perform both antialiasing and denoising. The network would also predict optimal ray positions.

7. Conclusion

We provided a small and concise convolutional neural network architecture for temporal anti aliasing. Our analysis compares different variations of the network on different dimensions. We show that using different sets of weights for some layers is crucial and allows a lot more performance without needed additional computation, only more memory. We also suggested to take this research deeper into the real-time raytracing pipeline by having the network suggest sample positions and perform the reconstruction step. This would be an almost all-in-one Neural Network, capable of denoising, antialiasing and adaptive sampling.

Acknowledgements

The first author has been funded by the Central Innovation Programme for small and medium-sized enterprises (FKZ: ZF4720101 MS9). The second author is PhD student at the Helmholtz Information & Data Science School for Health (HIDSS4Health). We would like to thank Nym L. Colpart for their help in making the supplementary material. Our thanks extend to Stephan Bergman, Alexander Devaykin and Thomas Willberger for their insights and guidance throughout the project.

References

- [Ake93] AKELEY K.: Reality engine graphics. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (1993), pp. 109–116. doi:10.1145/166117.166131. 1, 2
- [AMD20] AMD: Fidelityfx. In www.amd.com/en/technologies/radeon-software-fidelityfx. (2020). 2
- [ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: FLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 2 (2020), 15:1–15:23. 5
- [CKS*17] CHAITANYA C. R. A., KAPLANYAN A. S., SCHIED C., SALVI M., LEFOHN A., NOWROUZEZAHRAI D., AILA T.: Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM Transactions on Graphics* 36, 4 (jul 2017). doi:10.1145/3072959.3073601. 2
- [CMFL15] CRASSIN C., MCGUIRE M., FATAHALIAN K., LEFOHN A.: Aggregate g-buffer anti-aliasing. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2015), i3D '15, Association for Computing Machinery, p. 109–119. URL: <https://doi.org/10.1145/2699276.2699285>, doi:10.1145/2699276.2699285. 2
- [FWHB21] FAN H., WANG R., HUO Y., BAO H.: Real-time Monte Carlo Denoising with Weight Sharing Kernel Prediction Network. *Computer Graphics Forum* 40, 4 (2021), 15–27. doi:10.1111/cgf.14338. 2
- [HMS*20] HASSELGREN J., MUNKBERG J., SALVI M., PATNEY A., LEFOHN A.: Neural Temporal Adaptive Sampling and Denoising. *Computer Graphics Forum* 39, 2 (2020), 147–155. doi:10.1111/cgf.13919. 2, 5
- [HY21] HUO Y., YOON S.-E.: A survey on deep learning-based Monte Carlo denoising. *Computational Visual Media* 7 (2021), 169–185. doi:10.1007/s41095-021-0209-9. 2
- [JESG12] JIMENEZ J., ECHEVARRIA J. I., SOUSA T., GUTIERREZ D.: SMAA: Enhanced Morphological Antialiasing. *Computer Graphics Forum (Proc. of Eurographics)* 31, 2 (2012). doi:10.1111/j.1467-8659.2012.03014.x. 2
- [Kar14] KARIS B.: High-quality temporal supersampling. *Advances in Real-Time Rendering in Games, SIGGRAPH Courses 1*, 10.1145 (2014), 2614028–2615455. 2, 4
- [KB14] KINGMA D. P., BA J.: Adam: A method for stochastic optimization. *arXiv preprint* (2014). doi:10.48550/arXiv.1412.6980. 5
- [KBS15] KALANTARI N. K., BAKO S., SEN P.: A Machine Learning Approach for Filtering Monte Carlo Noise. *ACM Transactions on Graphics* 34, 4 (jul 2015). doi:10.1145/2766977.2. 3, 5
- [Liu20] LIU E.: DLSS 2.0 – Image reconstruction for real-time rendering with deep learning. In *GPU Technology Conference (GTC)* (2020). 1, 2
- [Lot09] LOTTES T.: FXAA. In *Nvidia White Paper* (2009). URL: developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf. 2
- [NAM*17] NALBACH O., ARABADZHIYSKA E., MEHTA D., SEIDEL H.-P., RITSCHEL T.: Deep Shading: Convolutional Neural Networks for Screen Space Shading. *Computer Graphics Forum* 36, 4 (2017), 65–78. doi:10.1111/cgf.13225. 2
- [Nvi23] NVIDIA: Tensorrt. In <https://developer.nvidia.com/tensorrt> (2023). 6
- [Ped16] PEDERSEN L. J. F.: Temporal reprojection anti-aliasing in INSIDE. *Game Developers Conference* 3, 4 (2016), 10. 2, 3
- [Res09] RESHETOV A.: Morphological Antialiasing. In *Proc. of ACM SIGGRAPH / Eurographics conference on High Performance Graphics* (New York, NY, USA, 2009), HPG '09, Association for Computing Machinery, p. 109–116. doi:10.1145/1572769.1572787. 1, 2
- [Sal16] SALVI M.: An excursion in temporal super sampling. *Game Developers Conference* 3, 7 (2016), 12. 2, 3, 5
- [Sal17] SALVI M.: Deep learning: The future of real-time rendering. *ACM SIGGRAPH Courses: Open Problems in Real-Time Rendering 12* (2017). 2
- [SN15] SACTH L., NEHAB D.: Optimized Quasi-Interpolators for Image Reconstruction. *IEEE Transactions on Image Processing* 24, 12 (2015), 5249–5259. doi:10.1109/TIP.2015.2478385. 4

- [TVLF20] THOMAS M. M., VAIDYANATHAN K., LIKTOR G., FORBES A. G.: A Reduced-Precision Network for Image Reconstruction. *ACM Transactions on Graphics* 39, 6 (nov 2020). doi:10.1145/3414685.3417786. 1, 2, 3, 5
- [VRM*18] VOGELS T., ROUSSELLE F., MCWILLIAMS B., RÖTHLIN G., HARVILL A., ADLER D., MEYER M., NOVÁK J.: Denoising with Kernel Prediction and Asymmetric Loss Functions. *ACM Transactions on Graphics (Proc. SIGGRAPH)* 37, 4 (2018), 124:1–124:15. doi:10.1145/3197517.3201388. 1, 2
- [Wat20] WATSON A.: Deep learning techniques for super-resolution in video games. *arXiv preprint* (2020). doi:10.48550/arXiv.2012.09810. 2
- [XNC*20] XIAO L., NOURI S., CHAPMAN M., FIX A., LANMAN D., KAPLANYAN A.: Neural Supersampling for Real-Time Rendering. *ACM Transactions on Graphics* 39, 4 (aug 2020). doi:10.1145/3386569.3392376. 1, 2
- [YLS20] YANG L., LIU S., SALVI M.: A Survey of Temporal Anti-aliasing Techniques. *Computer Graphics Forum* 39, 2 (2020), 607–621. doi:10.1111/cgf.14018. 1, 2, 3
- [YNS*09] YANG L., NEHAB D., SANDER P. V., SITTHI-AMORN P., LAWRENCE J., HOPPE H.: Amortized Supersampling. *ACM Transactions on Graphics* 28, 5 (dec 2009), 1–12. doi:10.1145/1618452.1618481. 2, 3
- [ZLY*21] ZENG Z., LIU S., YANG J., WANG L., YAN L.-Q.: Temporally Reliable Motion Vectors for Real-time Ray Tracing. *Computer Graphics Forum* 40, 2 (2021), 79–90. doi:https://doi.org/10.1111/cgf.142616. 2