Appendix: Neighborhood Attention Transformer

Ali Hassani[1], Steven Walton[1], Jiachen Li[1], Shen Li[3], Humphrey Shi[1,2]
[1]SHI Labs @ U of Oregon & UIUC, [2]Picsart AI Research (PAIR), [3]Meta/Facebook AI
https://github.com/SHI-Labs/Neighborhood-Attention-Transformer

We present a more detailed illustration of neighborhoods in Fig. VIII. Note that the repeated windows at corner pixels is especially important to NA approaching SA. We also present details on our $\mathcal{N}ATTEN$ package in Appendix A, and additional experiments in Appendix B. Additionally, we discuss translational equivariance in self attention mechanisms in Appendix C.

A. $\mathcal{N}$ATTEN

In this section, we outline the necessity for an extension such as $\mathcal{N}ATTEN$ for research in the direction of dynamic sliding window attention patterns, and describe how it aims to resolve such problems.

A.1. Background

While many operations in deep neural networks can be broken down to matrix multiplication, certain point-wise operations, such as convolutions, require customized implementations for more optimal parallelization. As a result, convolutions, recurrent modules, and other similar operations are natively supported in most low-level computational packages, which are called by deep learning frameworks such as PyTorch. In other words, given any input set and an operation, deep learning frameworks select the most efficient implementation available for that particular case, considering the hardware and software running said operation.

This makes research significantly easier, while being inevitably constrained to operations that are well-implemented. To allow further flexibility, some deep learning frameworks also allow for extensions to be built on top of them when necessary. Extensions can therefore enjoy customized CPU and GPU implementations. Notable examples of such extensions are Deformable Convolutions [9] and Deformable Attention [41], which have been implemented as CUDA extensions to PyTorch.

Sliding window attention mechanisms are no different, in that they require manual implementation to maximize parallelization and bandwidth. Without those implementations, the only alternative is a Python implementation, which typically does not scale. For instance, implementing Neighborhood Attention with PyTorch alone would include extracting sliding windows, repeating and re-arranging them to produce neighborhoods, and then performing two batched matrix multiplications. This would mean two separate C++/CUDA calls to generate significantly large intermediary tensors, which result in an exponential memory usage and latency increases. With the most optimized plain PyTorch implementation, NA would run at **13% the speed of Swin Transformer** on a 56×56 feature map (first level of an ImageNet model), while using approximately **9 times as much memory**. With a naive CUDA implementation, the same NA module runs at **102% the speed of Swin**, while using approximately **20% less memory**. With our **Tiled NA** algorithm, that same module runs at **132% the speed of Swin**, with no change in memory usage. You can refer to Figs. I and II for more benchmarks comparing different NA implementations to Swin Transformer in terms of relative speed and memory usage.

This is why we developed $\mathcal{N}ATTEN$, which currently serves as an extension to PyTorch, and provides torch modules `NeighborhoodAttention1D` and `NeighborhoodAttention2D`. This allows any PyTorch user to integrate NA into their models, for both tokens and pixels.

Each module consists of linear projections for queries, keys, and values, and a final linear projection, which is standard in most dot-product self attention modules. $\mathcal{N}ATTEN$ provides a single autograd function for each of Eq. (2) and Eq. (4). Once tensors `q`, `k`, and `v` are generated, attention weights are computed (see Eq. (2)) by passing `q`, `k`, and positional biases `b` to the C function `QK+RPB`, which picks the appropriate kernel to call (CPU or CUDA; naive or special; half or full precision). Softmax and dropout are then applied to the output attention weights, `a`, with native torch implementations. NA's final is computed by passing `a` and `v` to the C function `AV`.

A.2. Naive CUDA Kernels

Originally, we developed 7 naive CUDA kernels: 1 for `QK+RPB`, 1 for `AV`, and 5 to compute gradients for each of `q`, `k`, `b`, `a`, and `v`. Naive kernels simply divide computa-

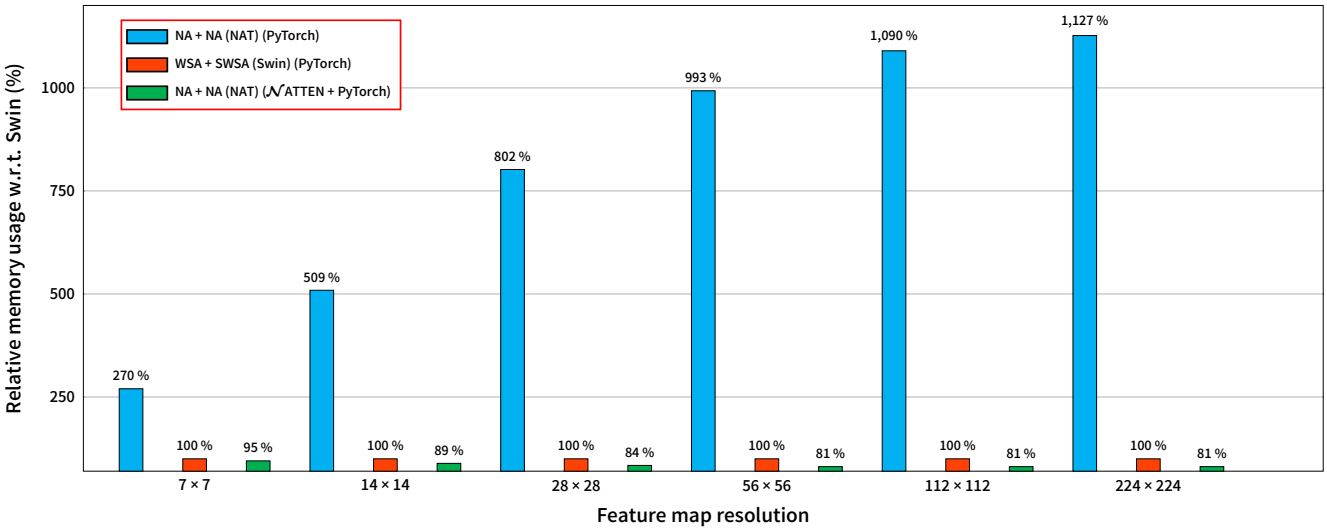

Figure I. NAT's layer-wise memory usage with respect to Swin. Without $\mathcal{N}ATTEN$, a plain PyTorch implementation of NA bears a quickly growing memory footprint compared to Swin, whereas with $\mathcal{N}ATTEN$, it uses consistently lower memory.

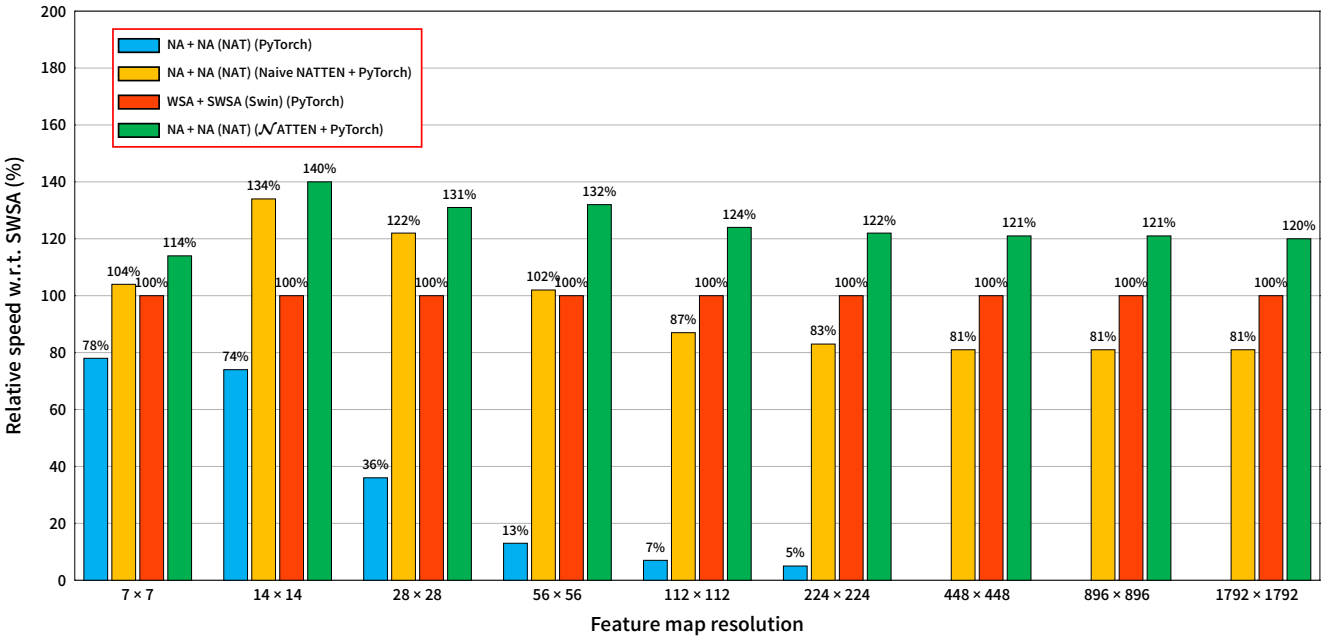

Figure II. Torch-based NA, Naive NA, and Tiled NA relative throughput comparison w.r.t. WSA+SWSA. Latency is measured on a single A100 GPU. Note that the plain PyTorch implementation of NA runs out of memory for resolutions 448^2 and higher.

tion across available threadblocks, and do not utilize shared memory or warp optimization. Despite their simplicity, they were able to benchmark between 80% up to 130% the speed of WSA+SWSA layers (both with kernel size 7×7). However, naive kernels are not optimal; they read directly from the global memory on the GPU, which bottlenecks throughput.

A.3. Half precision

Supporting mixed precision training is not too complicated. PyTorch's ATen dispatchers compile all kernels for both double and single precision by default, since tensor data type is usually templated. By choosing a different dispatcher, kernels can be easily compiled for `half` tensors. However, simply support half precision rarely results in any significant bandwidth improvement without integrat-

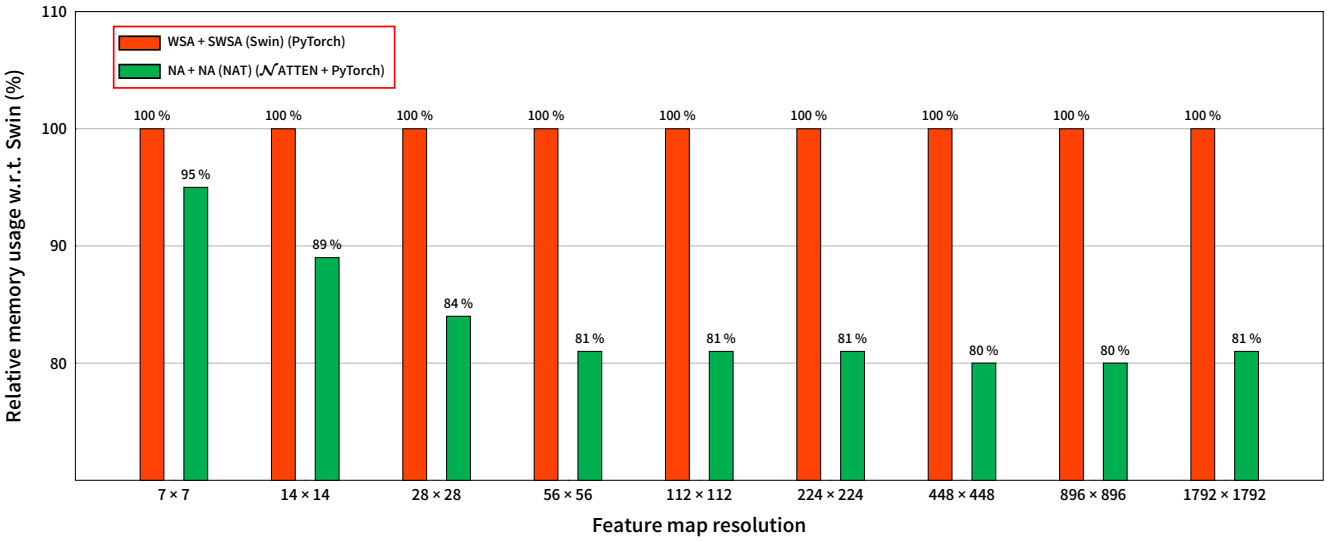

Figure III. NAT's layer-wise memory usage with respect to Swin. Since NA does not include a pixel shift and masked attention like SWSA, and the addition of positional biases is fused into the C++/CUDA kernels, NA with $\mathcal{N}ATTEN$ uses less memory compared to a similar model with WSA+SWSA.

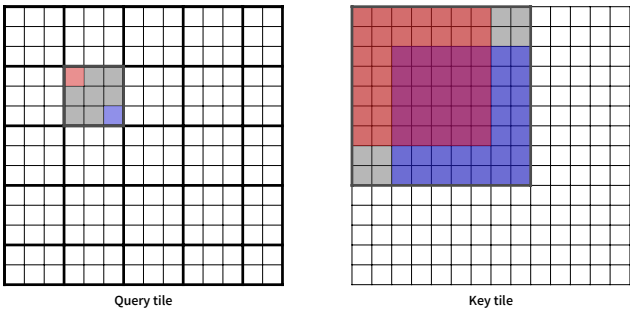

Figure IV. An illustration of tiled neighborhood attention for kernel size 7 × 7 and tile size 3 × 3. Queries are partitioned into tiles (left), and because of the large overlap in neighboring keys, it is easy to predict a tile in keys based on kernel size (right). This allows each separate threadblock, which has a shared memory between threads, to compute outputs for a specific query tile. Two queries (top left and bottom right) and their respective neighborhoods are also highlighted with different colors to visualize that the information needed to compute outputs for each tile is available in the tiles that are loaded.

ing CUDA's vectorized `half2` data type and operators. As a result, we separately define our half precision kernels to utilize vectorize load, multiply-add, and stores. This yields a more significant improvement in mixed-precision training speed.

A.4. Tiled Neighborhood Attention

CUDA allows easy allocation and utilization of shared memory between threadblocks. This, however, typically requires a change in the algorithm. Therefore, we implemented a tiled version of our attention weight kernel, and its backward kernel, which divides inputs into non-overlapping tiles, assigns each thread within the threadblock to read a specific number of adjacent cells from global memory, sync, and then compute outputs based on values in the shared memory. We present an illustration of that in Fig. IV. Using shared memory also presents new challenges, including, but not limited to: 1. Tile size bounds depending on kernel size, dimension, and shared memory limit on the GPU. 2. Bank conflicts between warps during computation. 3. Different number of reads from each input depending on tile size.

For instance, Fig. IV illustrates NA at kernel size 7 ×7, with tile size 3 × 3, which requires a key tile of size 9 × 9. The 3 × 3 tile size was chosen based on a number of factors, including the size of shared memory (48 KB), total number of threads per threadblock (1024 since compute capability 2.0), and other problem-specific factors such as embedding dimension. Key tile size is always equal to $t_q + k - 1$, where t_q is the query tile size, and k is kernel size, which is $3 + 7 - 1 = 9$ here.

Through a detailed internal analysis, we implemented and optimized Tiled NA for kernel sizes 3, 5, 7, 9, 11, and 13. Although not all bank conflicts were avoided in all use cases, they were minimized through profiling with NVIDIA Nsight™. Even though this implementation has resulted in a considerable bandwidth increase in NA training and inference, $\mathcal{N}ATTEN$ is still fairly at an early stage. We hope to improve existing kernels and add more optimal ones for different use cases, and add support for the new Hopper architecture with CUDA 12.

A.5. CPU Kernels

We extend $\mathcal{N}ATTEN$ to support CPU operations as well, both for training and inference. CPU functions for Neighborhood Attention are simple C++ implementations, with AVX vectorization support in newer PyTorch versions. As a result, they can easily utilize multi-threaded computation, which usually results in a relatively good latency compared to similar sized models on consumer CPUs. In total, there are 7 CPU kernels in the current version (similar to the naive implementations, 1 for each operation and 1 for each gradient.) We foresee further optimizations and additional CPU kernels in the near future.

A.6. Future efforts

We hope to continue supporting $\mathcal{N}ATTEN$ and help the community enjoy sliding window attention modules. Our hope is to eventually implement Neighborhood Attention with implicit GEMM (generalized matrix-matrix product), which will allow $\mathcal{N}ATTEN$ to be built on top of open-source packages (i.e. CUTLASS) and utilize the power of hardware accelerators to a greater extent.

B. Additional experiments

B.1. Ablation on RPB

We present an ablation on relative positional biases and pixel shifts (WSA only) in Tab. I.

Attention	Positional biases	Top-1 (%)	# of Params	FLOPs
○ **WSA-SWSA**	None	80.1 (+ 0.0)	28.26 M	4.51 G
○ **NA+NA**	None	80.6 (+ 0.5)	28.26 M	4.51 G
○ **WSA-WSA**	Relative Pos. Bias.	80.2 (+ 0.0)	28.28 M	4.51 G
○ **WSA-SWSA**	Relative Pos. Bias.	81.3 (+ 1.1)	28.28 M	4.51 G
○ **SASA-SASA**	Relative Pos. Bias.	81.6 (+ 0.3)	28.28 M	4.51 G
○ **NA-NA**	Relative Pos. Bias.	81.8 (+ 0.5)	28.28 M	4.51 G

Table I. Comparing NA and WSA with and without positional biases. Swin's results are directly reported from the original paper.

B.2. Saliency analysis

In an effort to further illustrate the differences between attention mechanisms and models, we present salient maps from ViT-Base, Swin-Base, and NAT-Base. We selected a few images from the ImageNet validation set, sent them through the three models, and created the salient maps based on the outputs, which are presented in Fig. VII. All images are correctly predicted (Bald Eagle, Acoustic Guitar, Hummingbird, Steam Locomotive) except ViT's Acoustic Guitar which predicts Stage. From these salient maps we can see that all models have relatively good interpretability, though they focus on slightly different areas. NAT appears to be slightly better at edge detection, which we believe is due to the localized attention mechanism, that we have presented in this work, as well as the convolutional downsamplers.

C. Notes on translational equivariance

In this section, we discuss the translational equivariance property in attention-based models, which is often referenced as a useful property in convolutional models [14]. To do that, we begin with defining equivariance and translations, and then move on to studying the existence translational equivariance in different modules.

Translation. In the context of computer vision, translation typically refers to a shift (and sometimes rotation) in pixels.

Equivariance. A function f is equivariant to a function $\mathcal{T}$ if $\mathcal{T}(f(x)) = f(\mathcal{T}(x))$.

Translational Equivariance. An operation f is equivariant to translations.

Linear projections. A single linear layer, which can also be formulated as a 1×1 convolution, is by definition equivariant to any change in the order of pixels. Therefore, they are also translationally equivariant.

Convolutions. Thanks to their dynamic sliding window structure, and their static kernel weights, convolutions are translationally equivariant [14], since every output pixel is the product of its corresponding input pixel centred in a window and multiplied by the static kernel weight.

Self Attention. SA (Eq. (1)) is translationally equivariant [25], because: 1. the linear projections maintain that property, and 2. self attention weights are also equivariant to any change in order.

SASA. SASA [25] extracts key-value pairs for every query according to the same raster-scan pattern convolutions follow, which suggests it maintains translational equivariance. However, convolutions apply static kernel weights, which allows them to maintain this property. On the other hand, even though SASA applies dynamic weights, those weights are still a function of the pixels within the window. Therefore, SASA also maintains translational equivariance. Note that SASA does not enjoy the same position-agnostic property in self attention.

HaloNet. The blocked self attention pattern described in HaloNet [30] is described to "relax" translational equivariance. It is simply due to the fact that pixels within the same region share their neighborhood, therefore their sliding window property is relaxed and with it translational equivariance.

WSA and SWSA. The basic property present in both WSA and SWSA is the partitioning, which exists in only one of two forms (regular and shifted) and therefore not dynamically sliding like SASA or convolutions. This simply breaks translational equivariance, as translations move the dividing lines. To give an example, an object within the feature map could fit within a single WSA partition, but the translation could shift the object just enough so that it falls into two different partitions. To illustrate this, we provide visualizations of activations from a single Swin block (WSA + SWSA) in Fig. VI, where we compare translations applied to input and output. We replace all linear projections with the identity function (as those are already known to be equivariant) and remove positional biases for simplicity in visualization.

NA. We note that our NA preserves translational equivariance for the most part, similar to SASA. However, NA relaxes translational equivaraince in corner cases in favor of maintaining attention span. We present translations applied to dummy inputs and their NA outputs in Fig. VI, similar to those of Swin. However, we also note that NA relaxes the translational equivariance in corner cases, particularly because of its definition of neighborhood which results in sliding windows being repeated at edge pixels. A visualization of this can be seen visualized with a larger kernel size (quarter of the image) compared to Swin and SASA in Fig. V.

The difference in how corner cases are handled is an important difference which should exist between sliding window attention mechanisms and convolutions. Repeating sliding windows at corner cases (which NA achieves with the neighborhood definition) is useful in the scope of attention, because the repeated windows are still subsets of the original self attention weights, which are being restricted. This does not hold true in convolutions, where repeated sliding windows produces repeated output pixels, because of the static kernel. On the other hand, zero padding in attention (no repetition at corner cases; like SASA) is less powerful because it limits attention span farther at corner cases. It also does not approach self attention as its window size grows, while NA does.

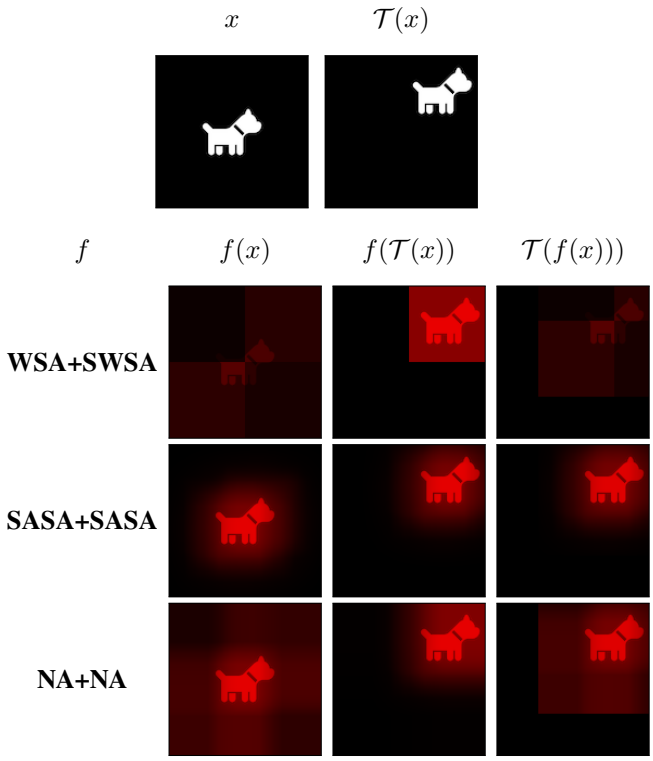

Figure V. Corner pixel visualizations with quarter size kernels. x denotes the input image with the object centered, $f(x)$ denotes the output when the function f is applied, and $\mathcal{T}$ is the translation that shifts the object to the upper right side corner of the image. While SASA does not break translational equivariance at corner pixels as much as NA, it would suffer from a reduced attention span in those areas, which is the reason it does not approach self attention. Simply looking at SASA's output for the original centered input shows the effect of the reduced attention span, when compared to NA's output.

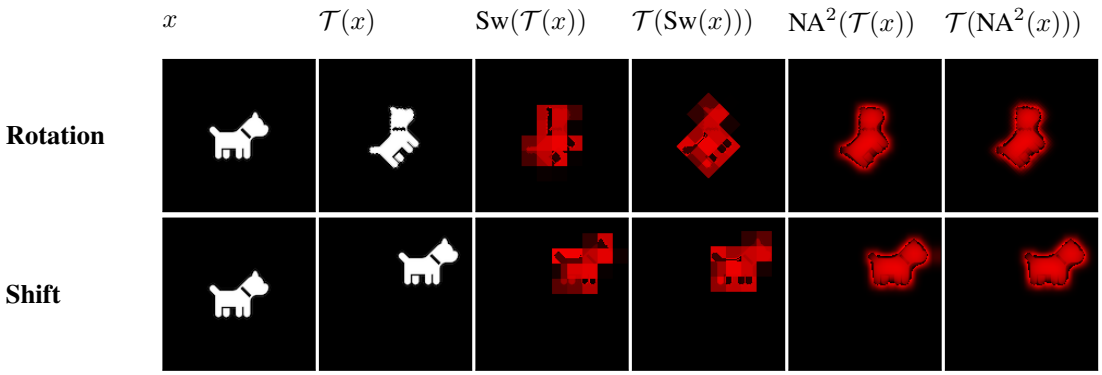

Figure VI. Visualization of translations applied to Swin and NAT. $\mathcal{T}$ denotes the translation function (top row is rotation, bottom row is shift). "Sw" denotes a WSA+SWSA applied to the input, with a residual connection in between. This pattern breaks translational equivariance. "NA^2" denotes two layers of NA applied to the input, with a residual connection in between. NA preserves translational equivariance with its sliding window property.

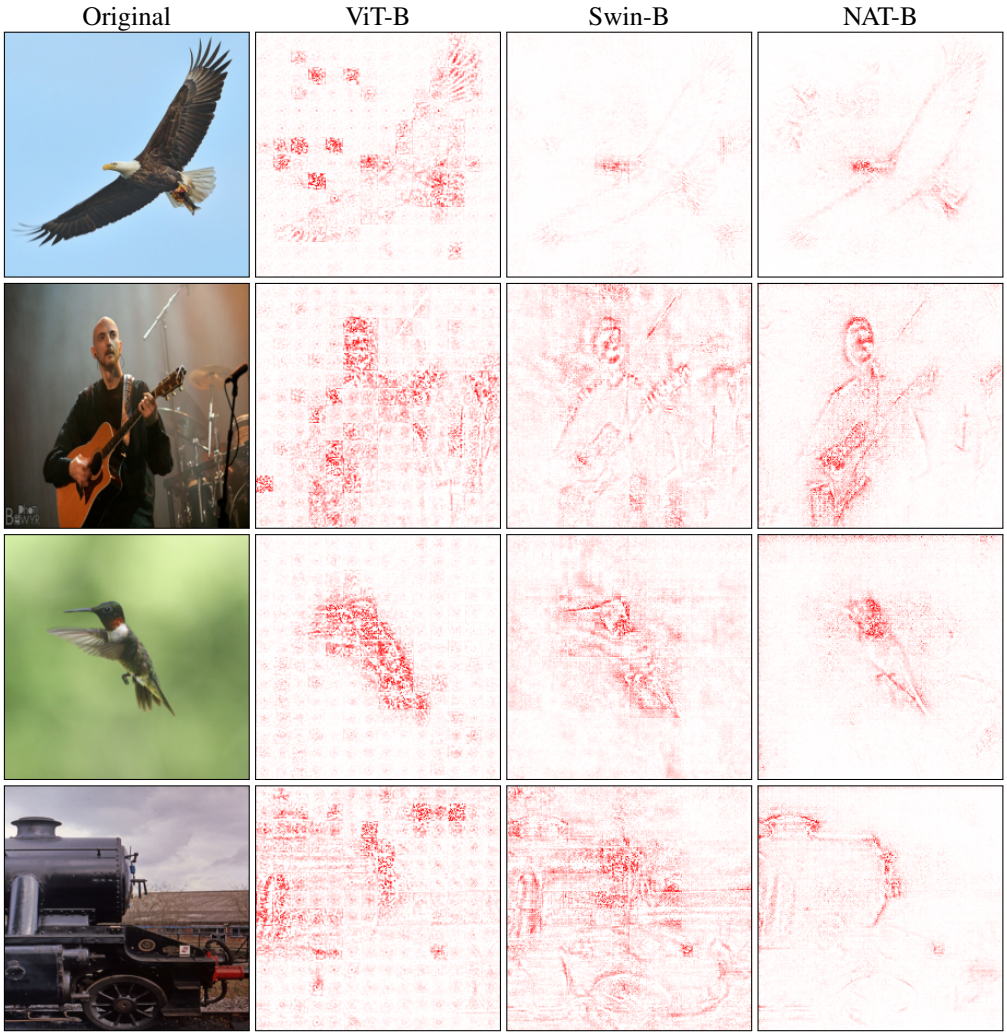

Figure VII. Salient maps of selected ImageNet validation set images, comparing ViT-Base, Swin-Base, and NAT-Base. The ground truths for these images are: Bald Eagle, Acoustic Guitar, Hummingbird, and Steam Locomotive, respectively.

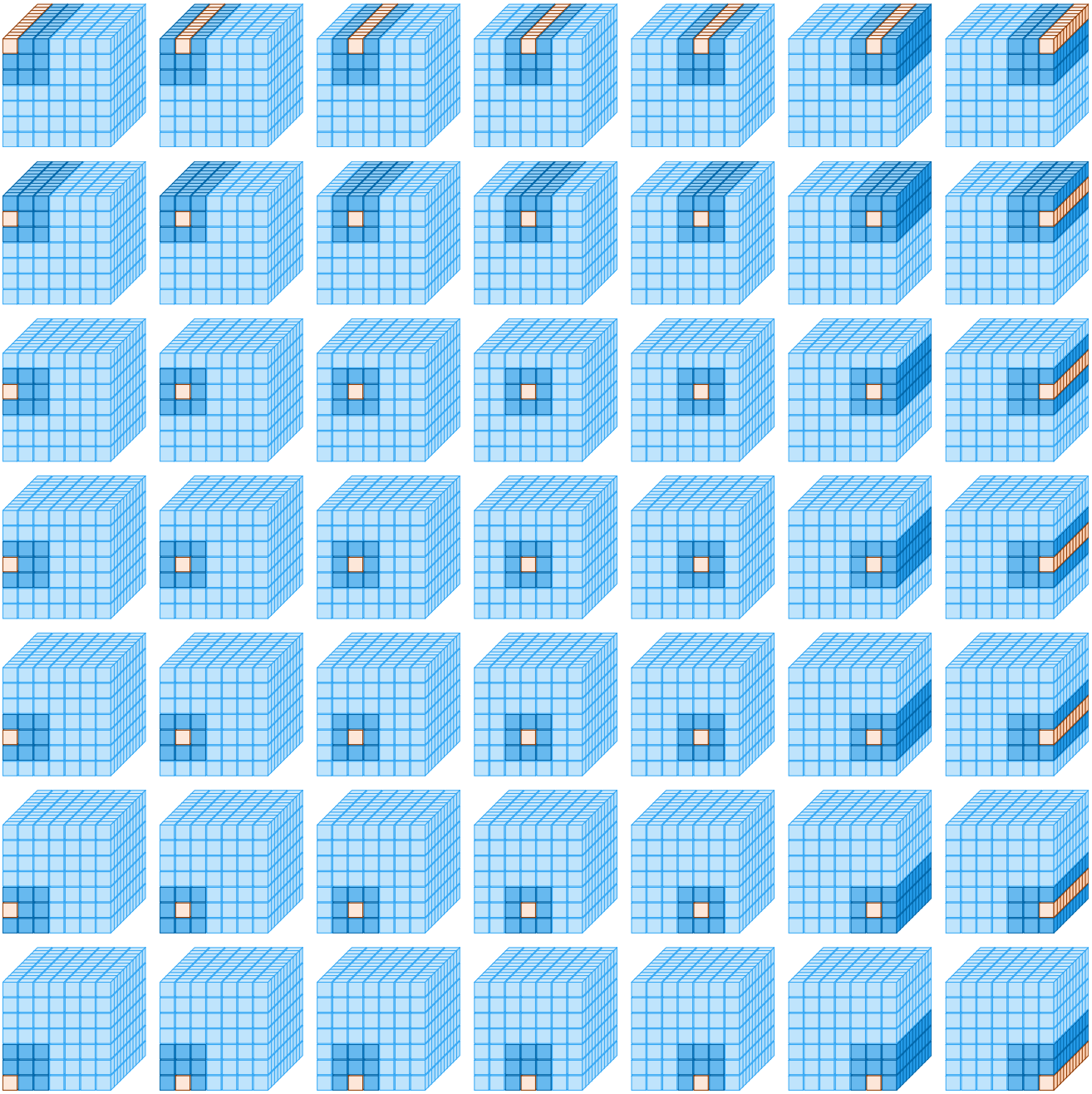

Figure VIII. An illustration of 3 × 3 neighborhood attention pattern on a 7 × 7 feature map. Query is colored orange, and its attention span (key-value pair) is dark blue. The "window" is repeated at the corners because of the neighborhood definition. This keeps attention span identical to the rest of the feature map. The alternative to this would have been smaller neighborhoods (zero padding at the corners, similar to SASA).