Neighborhood Attention: dynamic restriction of self attention

Ali Hassani
Department of Computer Science
University of Oregon
alih@uoregon.edu

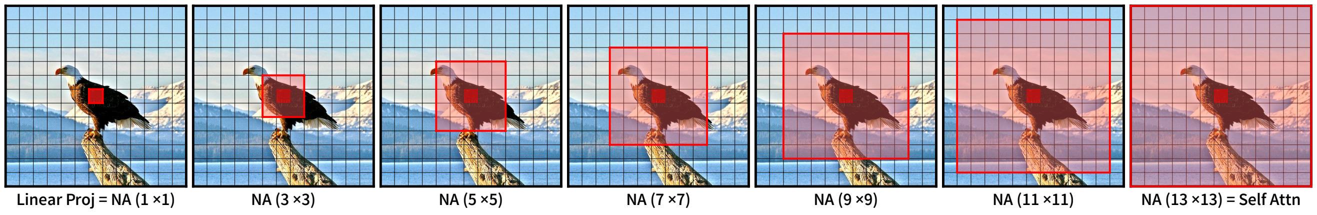

Figure 1. Illustration of neighborhood attention with different kernel sizes. Neighborhood attention localizes self attention to each token's nearest neighbors, which eases the quadratic complexity, introduces inductive biases such as locality, and allows flexibility in choosing the degree to which self attention is localized or restricted.

Abstract

Transformers, and more generally attention-based models, are omnipresent in modern deep learning frameworks, dominating a wide range of applications from language to vision and speech. Self attention, one of the primary operators in these models, is often cited for its quadratic complexity with respect to input size. Avoiding this complexity has often been done through local and sparse patterns, which can be effective, but usually either eliminate useful properties and inductive biases, or are often difficult to implement and scale. We propose neighborhood attention, a restriction of self attention to nearest neighbors, which results in linear time complexity, and can maintain many of the properties present in self attention. This pattern can be thought of as a flexible sliding window pattern, aimed at capturing consistent local context throughout the input. Previous attempts at sliding window patterns in attention were roadblocked by the relative difficulty in implementing such patterns for parallel hardware, which dominate training deep learning models. To that end, we propose and implement a range of different algorithms for neighborhood attention, starting with naive implementations. We then formulate neighborhood attention as an implicit general matrix-matrix multiplication (GEMM) problem, and implement its kernels in CUTLASS. This implementation provides up to 6× improvement in latency in 1D problems, and 4× improvement in 2D problems compared to naive GPU kernels. We package and release all of our implementations as a Python package, $\mathcal{NATTEN}$, which would allow researchers to quickly set up and use neighborhood attention. We finally show some of many possible applications of neighborhood attention, by introducing a set of hierarchical vision transformers, Neighborhood Attention Transformer, and present their performance on image classification, object detection, and image segmentation.

1. Introduction

Transformers [48] have made a significant contribution to AI research, starting with machine translation [48] and more generally natural language processing [14, 38], and later applied to other modalities such as speech [17] and vision [15, 34]. Their omnipresence in AI today can be attributed to their universal architecture built upon attention. In computer vision, the original Transformer architecture has been applied directly to image classification [15] as well as dense tasks [19, 27]. Computer vision research in transformer-like [30, 31, 50], transformer-inspired [32], and attention-based [24, 31, 39, 47, 49] architectures has arguably overtaken the former de facto standard, Convolutional Neural Networks (CNNs) [21, 25, 26]. Applications vary, from simple vision backbone networks [39, 47], to more specific tasks such as image generation and density modeling [8, 34], object detection [5], image segmentation [24, 49], and more recently, denoising diffusion models [22, 36, 41].

The attention operator, a key component in the Transformer, bears with it many highly desirable properties, including, but not limited to, global inter-dependency modeling, a global receptive field, permutation invariance, and therefore translational equivariance, and relatively easy implementation and parallelization. Attention is typically used in two different settings: weighting a single set of inputs (projected into a key-value pair) with respect to themselves (query) (self attention), or weighting the single set of inputs with respect to a different set of inputs (cross attention). In the original Transformer architecture, encoder layers were comprised of a self attention and multi-layer perceptron (MLP) block, while in the decoder, a cross attention block between the input sequence, and a masked embedding of the output sequence is placed between the two. Self attention is known for its quadratic time complexity with respect to the number of inputs (memory depends on implementation [12, 37]), which, to some extent, has limited its applications to longer documents in language processing, and larger resolutions in vision. In addition, certain inductive biases that were found useful particularly in vision, such as locality, a property that convolution bears inherently, are typically only achievable in self attention through learning, which tends to require more training data [15, 18]. As a result, researchers started experimenting with the idea of localizing self attention, producing a sliding window pattern similar to that in discrete finite convolution [1, 39]. Such an approach would maintain many of the useful properties in self attention, such as translational equivariance, interdependency modeling and dynamic weighting, and introduce inductive biases such as locality, while reducing the quadratic time complexity to a linear time complexity with respect to the number of inputs (tokens in language, feature map size in vision). Such an approach would however eliminate the global receptive field and global inter-dependency modeling. Despite this, the primary limitation of these methods was scaling, both in terms of speed [31, 39, 47] and performance [31, 54].

Sliding window attention, meaning an attention operator where every input token has a different key-value pair (the window around it instead of every other token as in self attention), is difficult to implement for a number of reasons. Firstly, such an operation requires lower-level implementation, meaning implementing it through Python interfaces of deep learning libraries that researchers typically utilize is not nearly optimal [39]. Secondly, lower-level implementations of operations that compare with this approach favorably (i.e. convolution, and the like), have been highly optimized as a result of years of continued development of computational packages, as well as research into more efficient linear algebra and deep learning routines. This means that naive low-level implementations of such an approach is not likely to run as efficiently as most similar components used in deep neural networks, such as convolutions. Finally, this operation cannot be modeled simply after convolutions, despite the seemingly similar pattern. This is due to multiple reasons, including the fact that the kernel is different for every point, and said kernel requires to be computed in the first place, and softmaxed. As for performance, we find that a major issue is in how this operation was originally defined, which is modeled after zero-padded convolution primarily used in deep learning. The padding, which ensures queries are centered and not avoided, reduces the receptive field and therefore the context captured along corner cases, which limits the ability to scale this operation to larger window sizes. This is specifically a concern when the upper bound is self attention (maximum window size). In other words, localizing self attention by using sliding windows will not approach self attention itself as the window size grows. As a result of these issues, the community seemingly lost interest in studying these methods further. Follow up works by the same researchers abandoned the idea for alternatives with relaxed and less-frequent sliding windows [47], and other works building hierarchical vision transformers for higher-resolution downstream tasks, such as object detection, and image segmentation, also opted for non-sliding window alternatives [31], or other localized attention variants.

In this work, we aim to re-introduce explicit sliding window attention by addressing the two issues in depth. We first introduce Neighborhood Attention, a new pattern which avoids zero padding by simply removing a built-in constraint. It does so by restricting attention to an input token's nearest neighbors, and not those within the window around it, relaxing the requirement for queries to be centered within the local window. It is proven that by making this change, Neighborhood Attention becomes a flexible and direct approach towards restricting self attention, while maintaining all the properties present in the former sliding window attention. We introduce dilations, which can allow for sparse global receptive fields, and potentially reintroduce the global receptive field and global context from self attention. We then introduce our efforts towards implementing the concept, starting from the most naive approach, which is implementation using Python interfaces. We then move on to naive GPU kernels written in CUDA, and improve upon them by utilizing higher bandwidth memory. We then introduce our formulation of neighborhood attention as an implicit General Matrix-Matrix Multiplication (implicit GEMM), which means it can run on specialized matrix multiply and accumulate (MMA) cores in hardware accelerators (i.e. NVIDIA's Tensor Cores). Through these, we find that models based on neighborhood attention can be trained as efficiently as state of the art architectures, scale to competitive performance levels in image classification, object detection, and image segmentation, while allowing for a flexible pruning of the self attention graph.

In summary, the contributions of this paper are:

1. We introduce Neighborhood Attention; A flexible explicit sliding window attention pattern that bridges the line between linear projection and self attention. We investigate its properties, such as complexity, receptive field size, translational equivariance, and its relationship with self attention. We additionally introduce dilation into this pattern, and show that it provides many advantages such as a faster growing receptive field, and capturing global context, and it does so without imposing any additional computational cost.

2. We thoroughly examine ways to implement neighborhood attention, starting with the most naive of approaches, going all the way to formulating it, and more generally sliding window attention, as a GEMM problem, which allows it to utilize hardware accelerators more efficiently. We package these implementations as a Python package to ease the process of using this pattern in existing deep learning platforms.

3. We experiment with different vision architectures, and introduce one of our own, all equipped with Neighborhood Attention, and apply them to vision tasks such as classification, detection, and segmentation.

2. Background

In this section, we briefly review dot product self attention (DPSA), the Transformer [48], and Vision Transformer [15]. We then revisit previous attempts at restricting self attention through localized and sparse approaches, and discuss the shortcomings of each. More specifically, we revisit sliding window attention [1, 39], blocked attention [31, 47], sparse attention [1, 8], and blocked and sparse attention [23, 46].

2.1. Self Attention and the Transformer

In the original paper proposing the Transformer architecture, Vaswani et al. [48] defined dot product attention as an operation between a query, and a set of key-value pairs. The dot product of the query and keys is scaled and softmaxed, which produce the final attention weights. Said attention weights are then applied to the values:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d}}\right)V, \quad (1)$$

where $\sqrt{d}$ is the scaling parameter, and d is the number of dimension for which the dot product is computed (the embedding dimension of Q and K). Dot product self attention is simply a case of this operation where the queries, keys, and values are all linear projections of the same input. This operation bears a number of advantages, including, but not limited to, the ability to model global inter-dependencies, and therefore a global receptive field, as well as invariance to permutations to the order of the input sequence, therefore making it invariant to translations.

Vision Transformer (ViT) [15], one of the earliest works applying a pure transformer encoder to vision, showed the power of large-scale self attention based models in computer vision. Follow up works extended the study with minimal changes to training techniques [44], architectural changes [45], and applications to small data regimes [18].

Self attention's most cited bottleneck is its quadratic complexity with respect to the number of input tokens. Its memory footprint can be reduced to linear [12, 37] thanks to kernel fusion, but more importantly, online computation of softmax statistics [33]. However, its time complexity can still limit scaling to longer input sequences, such as long documents or context in language processing, and larger resolutions in vision. In addition, it can be argued that modeling global interactions can in some cases be unnecessary, especially in the presence of redundant tokens [2, 3, 28, 40]. As a result of these, especially the quadratic cost, restricting interactions in self attention is a problem that continues to be of interest to the research community.

2.2. Local Attention

2.2.1 Sliding window attention

Stand Alone Self Attention [39] is one of the earliest local attention patterns that was specifically geared towards vision. It defines a different key-value pair for every query token with respect to the token's spatial location. For every query, its key-value pair is reduced from the entire input sequence into only those that fall within a local window around the query token. In other words, the same raster scan pattern which is seen in convolution is used to extract the tokens to which a query will attend. This operation, often referred to as sliding window attention, maintains translational equivariance, and introduces inductive biases such as locality. The original paper showed that this operation could replace convolution in existing CNNs, such as ResNets, and even reduce computational complexity. Despite the promise it showed, the authors found that the resulting model suffers from higher latency compared to the baseline, due to the inefficient implementation of this module. Works succeeding it therefore switched to alternative methods that could run more efficiently. An example is HaloNet [47], which proposed relaxing the sliding window stride for the sake of efficiency.

However, implementation wasn't the only roadblock in studying such methods. This operation also eliminates the global receptive field, and the ability to model global inter-dependencies, which would hold true for any localized attention pattern. In addition, the same raster scan pattern introduces another issue: reduced interactions when windows

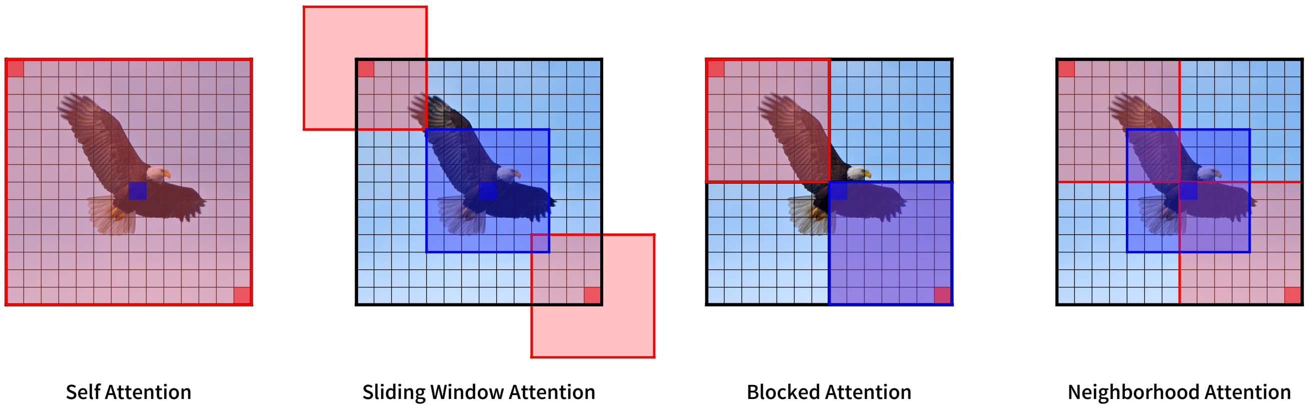

Figure 2. Illustration of different attention patterns compared to our neighborhood attention. Each image represents a 14 × 14 feature map, and the highlighted windows represent the receptive field from their corresponding query pixels. Self attention allows global interactions between every pair of tokens / pixels. Sliding window attention restricts interactions for every token to a local window around it, which bears local inductive biases present in convolutional models. Blocked attention partitions tokens, and restricts global interactions to each partition, which is easier to implement and parallelize compared to sliding window, but eliminates the dynamically shifting receptive field among other properties present in the former two. Neighborhood attention restricts interactions for every token to its nearest neighbors, capturing more context than sliding window attention, while preserving its useful properties, and allowing for more flexibility.

go out of bounds. This issue can become more serious as window sizes grow, or when patterns such as dilation are introduced, since the reduced interactions are a direct result of the "zero padding" handling of corner cases, and the number of those cases grows linearly with window size and dilation.

2.2.2 Blocked attention

Blocked attention is simply a pattern in which input tokens are partitioned, followed by a self attention operation on each partition. Also referred to as partitioned self attention and window self attention, this pattern can be implemented trivially, unlike sliding window attention. The partitioning operation has a relatively small overhead, similar to the reverse, making this pattern's implementation independent of the attention operator. This further eases implementation across different applications and platforms. This pattern also introduces a few issues of its own. The first issue is that it is no longer translationally equivariant, which simply means it lacks a useful property present in not only CNNs, but also attention-based models using self attention or sliding window attention. Another issue with this approach is that implementation can get more complicated if partitions are either not of the same size (i.e. partitions that contain more than half of the tokens.) Finally, among the most important issues is the fact that without any change in the partition size or the sequence order, tokens in different partitions would never interact. This means that the receptive field will be bound by window size and not grow. As a result, some additional operation following this pattern that allows for those interactions is necessary.

Liu et al. [31] proposed shifting pixels and masking invalid interactions, which is relatively trivial to implement, bears limited overhead, and works relatively well across a variety of applications. The same paper proposed Swin Transformer, a hierarchical vision transformer that followed ViT [15], but instead focused on applications to downstream vision tasks. It was also pointed out in this paper that sliding window attention was among their choices for a localized self attention pattern, but eventually not pursued due to issues in implementation as well as inferior performance.

2.3. Sparse Attention

Some works in language processing, such as Longformer [1], explored the idea of not only using a sliding window attention, but introducing dilation as well, which leads to a sparse and global pattern depending on the dilation value. Child et al. [8] proposed Sparse Transformers, which included different sparse and local patterns, used across different attention heads. There have been other works in sparse attention, including, but not limited to, Routing Transformers [42], and CCNet [24], all of which share a common feature: reducing the cost of self attention in cases where longer sequences are inevitable, but a global context is still necessary. In addition to those, more recently Tu et al. [46] proposed MaxViT, a hybrid architecture comprised of convolution, blocked attention, and sparse blocked attention with interlaced partitioning [23], which would result in a sparse global attention pattern. These patterns, and their hybrid architecture, led to state-of-the-art image classification performance.

In summary, local and sparse patterns in attention have

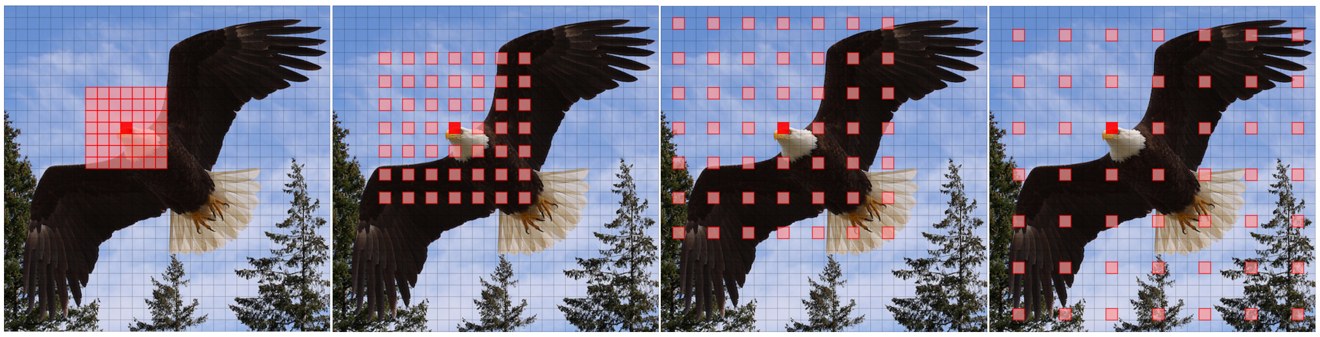

Figure 3. Illustration of our proposed neighborhood attention pattern with different dilation values.

been sought-after for long, but mostly limited to the blocked attention pattern. In this work, we aim to revisit sliding window attention, and to some extent address the shortcomings that prevented researchers from fully utilizing such methods at scale.

3. Methodology

Herein, we formally define Neighborhood Attention, and describe its properties and its extensions. We then move on to describe the difficulty in implementing neighborhood attention, and more generally, in implementing sliding window attention. We present multiple implementation approaches, describe their advantages and limitations, before moving on to formulating neighborhood attention as a set of smaller matrix multiplications. Through our final formulation, we introduce our GEMM-based implementation in NVIDIA's open-source CUTLASS library [43], which allows neighborhood attention to run on Tensor Cores. Finally, we introduce a hierarchical vision transformer based on our pattern, which we dub Neighborhood Attention Transformer (NAT).

3.1. Neighborhood Attention

As described previously, restricting self attention is a topic of interest not only due to its quadratic complexity with respect to input length, but also due to potential emergence of redundant tokens [2, 3, 28, 40], or lack of enough training data to model inductive biases such as locality [15, 18]. While blocked attention patterns are among the most commonly used, in part thanks to their ease of implementation, they eliminate properties such as translational equivariance, and dynamic per-token receptive fields.

Setting aside implementation, we revisit sliding window attention [39] and discuss a key issue the original formulation bears in theory, and then propose neighborhood attention which is aimed at resolving said issue. We then study neighborhood attention's properties, and compare it to existing methods. We finally move on to challenges in implementation, and present a wide range of possible implementations, ranging from naive implementations using Python interfaces, to naive CUDA kernels, and finally, a GEMM-based implementation in CUTLASS.

3.1.1 Definition

For simplicity, we use 1-D notations with only a single attention head.

Neighborhood attention. Given an input sequence in the form of a matrix, $X \in \mathbb{R}^{n \times d}$, whose rows are d-dimensional token vectors, we first project X into queries, and key-value pairs, similar to the original dot product attention formulation in Eq. (1), which we denote with matrices Q, K, and V, which are of the same shape as X. We then define neighborhood attention weights for the i-th token, $\mathbf{A}_i^k$, with neighborhood size k as the following vector-matrix multiplication:

$$\mathbf{A}_i^k = \begin{bmatrix} Q_i K_{\rho_1(i)}^T \\ Q_i K_{\rho_2(i)}^T \\ \vdots \\ Q_i K_{\rho_k(i)}^T \end{bmatrix}, \tag{2}$$

where $\rho_j(i)$ denotes token i's j-th nearest neighbor. We similarly define neighboring values, $\mathbf{V}_i^k$, to which attention weights would be applied, as a matrix whose rows are the i-th token's k nearest neighboring value projections:

$$\mathbf{V}_i^k = \begin{bmatrix} V_{\rho_1(i)}^T & V_{\rho_2(i)}^T & \dots & V_{\rho_k(i)}^T \end{bmatrix}^T. \tag{3}$$

The final output from neighborhood attention for the i-th token with neighborhood size k is :

$$\text{NA}_k(i) = softmax\left(\frac{\mathbf{A}_i^k}{\sqrt{d}}\right)\mathbf{V}_i^k, \tag{4}$$

where similar to Eq. (1), $\sqrt{d}$ is the scaling parameter, and d is the embedding dimension.

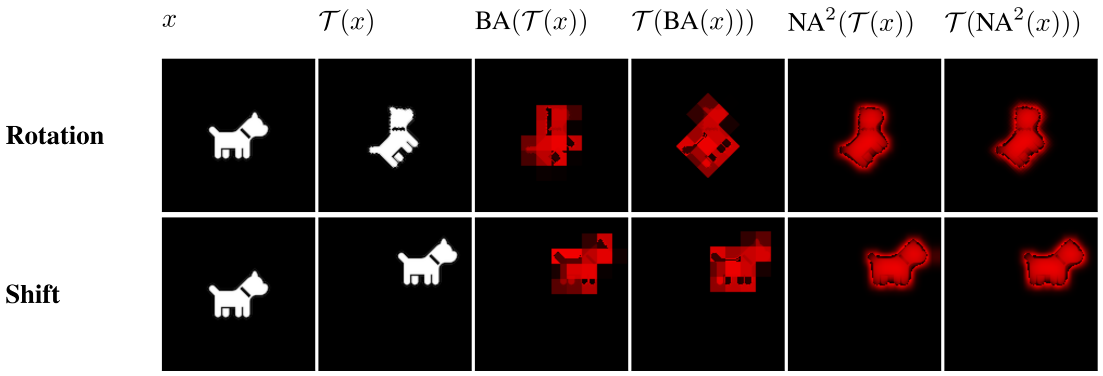

Figure 4. Visualization of translations applied to blocked and neighborhood attention. $\mathcal{T}$ denotes the translation function (top row is rotation, bottom row is shift). "BA" denotes blocked attention with shifts applied to the input per Swin Transformer, with a residual connection in between. This pattern breaks translational equivariance. "NA^2" denotes two consecutive neighborhood attention operations applied to the input, again with a residual connection in between. NA preserves translational equivariance.

Dilated neighborhood attention. Given a dilation value δ, we define $\rho_j^\delta(i)$ as the i-th token's j-th nearest neighbor that also satisfies: $j \bmod \delta = i \bmod \delta$. We can then define **$\delta$-dilated** neighborhood attention weights for the i-th token with neighborhood size k, $\mathbf{A}_i^{(k,\delta)}$, as follows:

$$\mathbf{A}_i^{(k,\delta)} = \begin{bmatrix} Q_i K^T_{\rho_1^\delta(i)} \\ Q_i K^T_{\rho_2^\delta(i)} \\ \vdots \\ Q_i K^T_{\rho_k^\delta(i)} \end{bmatrix}. \tag{5}$$

We similarly define δ-dilated neighboring values for the i-th token with neighborhood size k, $\mathbf{V}_i^{(k,\delta)}$:

$$\mathbf{V}_i^{(k,\delta)} = \begin{bmatrix} V^T_{\rho_1^\delta(i)} & V^T_{\rho_2^\delta(i)} & \cdots & V^T_{\rho_k^\delta(i)} \end{bmatrix}^T. \tag{6}$$

Finally, dilated neighborhood attention for the i-th token with neighborhood size k is defined as:

$$\text{DiNA}_k^\delta(i) = softmax\left(\frac{\mathbf{A}_i^{(k,\delta)}}{\sqrt{d_k}}\right)\mathbf{V}_i^{(k,\delta)}. \tag{7}$$

3.1.2 Properties

Relation to self attention and linear projection. Neighborhood attention can exhibit certain properties under different kernel sizes. Given a kernel size of 1 (or 1 × 1 in two dimensions), each token can only attend to itself, resulting in only a single attention weight. Since attention weights are mapped to a probability distribution through softmax, the final attention weight for every token will be 1.0, and therefore the every output token would be 1.0 times its corresponding value token. In other words, with a kernel size of 1, the attention operator becomes the identity function, and as a result, an attention layer would only perform a linear projection.

On the other hand, given the maximum kernel size, which is the same as input size, each token will attend to every other token, therefore effectively computing self attention. This can be easily seen by increasing kernel size k in Eq. (4). This property is useful because any self attention operation can be restricted step by step by simply reducing the number of interactions through reducing kernel size.

Translational equivariance. One of the properties present in both self attention, and convolution is translational equivariance [16]. To study whether this property exists in neighborhood attention, and other restricted attention patterns, we begin with the definition of translational equivariance in the context of vision, where translation typically refers to a shift or rotation function. Any function f is equivariant to a translation function $\mathcal{T}$ if $\mathcal{T}(f(x)) = f(\mathcal{T}(x))$. Linear projection is invariant to permutations in order of pixels, simply because pixels would serve as rows in the underlying matrix multiplication, and is therefore translationally equivariant. The same holds for self attention [39] simply because deriving attention weights is a simple matrix multiplication, and applying attention weights is a weighted sum of rows, therefore maintaining permutation invariance. As a result, both linear projection and self attention are invariant to permutations in the order of tokens, and therefore equivariant to translations. Convolutions are translationally equivariant [16], since every output pixel is the product of its corresponding input pixel centred in a window and multiplied by a static kernel. Note that convolutions are not invariant to permutations. Sliding window attention [39], and by extension neighborhood at-

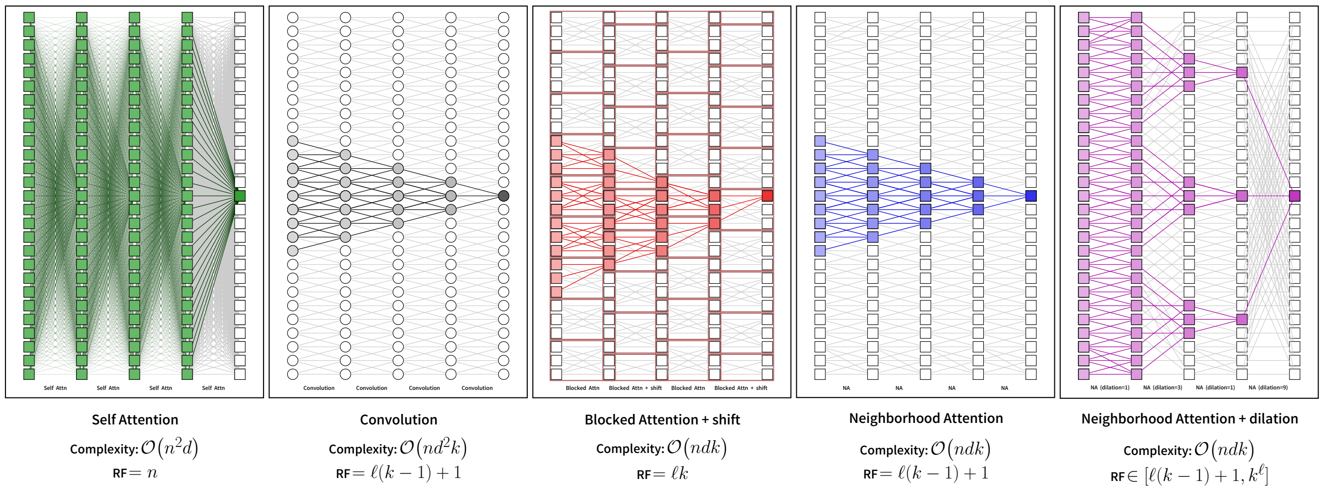
Self Attn
Convolution
Blocked Attn
Blocked Attn + shift
NA
NA (dilation=1)
NA (dilation=3)
NA (dilation=9)
Self Attention
Complexity: $\mathcal{O}(n^2d)$
RF$= n$
Convolution
Complexity: $\mathcal{O}(nd^2k)$
RF$= \ell(k-1)+1$
Blocked Attention + shift
Complexity: $\mathcal{O}(ndk)$
RF$= \ell k$
Neighborhood Attention
Complexity: $\mathcal{O}(ndk)$
RF$= \ell(k-1)+1$
Neighborhood Attention + dilation
Complexity: $\mathcal{O}(ndk)$
RF$\in [\ell(k-1)+1, k^\ell]$

Figure 5. Receptive fields from self attention (per ViT), convolution, blocked attention with shifts (per Swin), and neighborhood attention both with and without dilation. n denotes the number of tokens, d denotes the embedding dimension, and k denotes kernel/window size. All receptive fields are bounded by input size, n. Self attention's global receptive field yields the maximum receptive field per every layer. Convolution, blocked attention, and neighborhood attention grow their receptive fields linearly with more layers. With dilation, neighborhood attention's receptive field growth can range from linear, $\ell(k-1)+1$, to exponential, k^ℓ.

tention, break permutation invariance, as permutations will affect each pixel's neighborhood. However, translational equivariance is still maintained, because of the raster scan pattern, which is similar to that in convolutional layers. As for blocked attention, the partitioning breaks translational equivariance across the entire feature map, while permutation invariance within partitions is maintained. To better illustrate these, we visualize applying blocked attention (as proposed in Liu et al. [31]) and neighborhood attention with translations in Fig. 4.

Receptive field. Receptive field growth rate is considered as one of the contributing factors to a model's theoretical performance. As a result, we present an analysis of not only neighborhood attention's receptive field, but the effect dilation has on its growth rate. For simplicity, we will continue to use 1-dimensional notation. We denote the number of layers with ℓ, kernel size with k, and number of tokens with n. Self attention is known for its global receptive field, meaning every token can contribute to the outcome of every other token, therefore receptive field size in self attention is always the upper bound, n. On the other hand, convolutional layers and neighborhood attention start out with a receptive field of size k, since k tokens at a maximum can contribute to the outcome of any given token. They also expand by $k-1$ tokens per layer. As a result, both will end up with a receptive field growth of $k+(k-1)*(\ell-1)$, which can be simplified into $\ell(k-1)+1$. Blocked attention on its own maintains a constant receptive field of the same k tokens per layer. This can be problematic if used throughout the model without any operation introducing interactions between tokens in different partitions, meaning the receptive field will not grow. To introduce cross-partition interactions, Liu et al. [31] proposed shifting inputs in order to create a shifted partitioning effect with minimal overhead, which they dubbed cyclic shift. Alternating between blocked attention with and without cyclic shift resolves this issue, and expands receptive fields by exactly one window per layer, which is an expansion of k per layer. This results in a growth rate of ℓk, which is slightly larger than that of convolutions and neighborhood attention by $\ell-1$.

As for neighborhood attention's receptive field growth with dilation, it can range anywhere from the original $\ell(k-1)+1$, to an exponentially growing receptive field of k^ℓ. The lower bound is simply the growth rate from neighborhood attention with no dilation, meaning every layer's dilation value is set to 1. The upper bound is derived as follows: Regardless of dilation, the first layer always yields a receptive field of size k. Each one of the k tokens in the first layer can target a maximum of k new tokens in the following layer, given a dilation value that minimizes overlap. Because of this, the upper bound for growth in the second layer is k^2. This can be easily extended to ℓ layers, into a growth rate of k^ℓ. As a result, by introducing dilation into neighborhood attention, one can achieve an exponentially growing receptive field, which is a step in the right direction in terms of capturing global context. It is important to note that convolution can be dilated as well, but dilation values are much more constrained compared to neighborhood attention. This will be further discussed in Sec. 3.2. An illustration of the receptive field growth in different operators is also presented in Fig. 5.

Complexity. As stated, self attention has a quadratic complexity with respect to the number of tokens, which can be easily derived from Eq. (1). Computing attention weights is a matrix multiplication of an $n \times d$ matrix by the transpose of a matrix of the same size, which is $\mathcal{O}(n^2 d)$. The softmax operation requires computing element-wise exponential values followed by n summations of d elements, and an element-wise division, which is simply $\mathcal{O}(nd)$. Applying attention weights is similar in complexity to computing the weights, but is an element-wise multiplication of n weights by and $n \times d$ matrix ($\mathcal{O}(nd)$) for every one of n tokens, which is $\mathcal{O}(n^2 d)$. Neighborhood attention, blocked attention, and sliding window attention, share the same computational complexity given the same window size, despite their differences. For these methods, per token interactions, therefore weights, drop from n tokens to k tokens, reducing the original complexity to a linear $\mathcal{O}(ndk)$. Convolutions on the other hand apply static kernels of size k to the input, which is effectively applying a linear projection, $k \times d \to d$, to n activations of size $k \times d$ (assuming no padding). This is simply a matrix multiplication of an $n \times kd$ matrix by a $kd \times d$ matrix, which is of $\mathcal{O}(nd^2 k)$. It is worth noting that depth-wise convolutions are of $\mathcal{O}(ndk)$, and that attention also requires three linear projections, which would add $\mathcal{O}(nd^2)$ to the complexity. Fig. 5 includes computational complexity in addition to receptive field growth across the methods discussed.

3.2. Why not convolutions?

Neighborhood attention's seemingly similar pattern to convolutional layers may raise the question: what would be the advantage in using neighborhood attention instead of convolutions? This question is best addressed by pointing out both the differences between windowed attention and convolution in general, and by looking at properties specific to neighborhood attention that cannot be transferred to convolutions.

Windowed attention versus convolution. While at first glance, the pattern used to extract key-value pairs for tokens in sliding window and neighborhood attention seems similar to that in convolutions, the operation being performed is very different. In convolutions, activations from the raster scan are linearly projected into an output, which captures local context within the window. The same does not hold true for attention, where there's two sets of activations, one from each of the key-value projections, and the first is used to compute attention weights between a single token, and the tokens within its corresponding activation. This operation not only captures local context, it models local inter-dependencies between the query token, and its corresponding key tokens. Once attention weights are computed, and softmaxed, they are applied element-wise to the corresponding value activations. This is the primary difference between convolution and attention; the former applies a static linear projection on local regions, while the latter is using local interactions to compute outputs.

Properties specific to neighborhood attention. As stated, neighborhood attention's key difference compared to sliding window attention is the sliding window pattern itself. Neighborhood attention restricts attention to nearest neighbors, which is not necessarily a window centered around each token. This difference is very important, because explicitly centering attention window around every token results in reduced receptive field for tokens that cannot be centered, which potentially degrades performance at scale. This is because the ratio of such tokens increases linearly with window size and dilation, which would limit the ability to capture meaningful interactions at scale. Defining neighborhood attention completely resolves this issue without imposing any additional computation, but it may introduce additional challenges in implementation, which will be discussed in Sec. 3.3. Note that this difference is the sole reason why neighborhood attention windows can be dilated to greater extents compared to sliding window attention and convolution; given n tokens, kernel size can range from 1 to n, and dilation can range from $\lfloor n/k \rfloor$. This property cannot be extended to static convolution, simply because the neighborhood attention pattern results in repeated pixels in corner cases, due to the static kernel. As a result, static convolution is limited in terms of choices for kernel size and dilation, as larger kernel sizes or dilation values lead to linearly increasing padding in order to maintain the spatial size.

3.3. Implementation

Many existing works pointed out that implementing a sliding window attention is difficult, primarily citing the dynamic kernel as the reason why. Within this subsection, we aim to first provide more evidence to support claims regarding difficulty, before moving on to describing our implementations, each of which aim to improve upon the previous by creating more optimal implementations for hardware accelerators, which in the case is limited to general-purpose GPUs. Deep learning frameworks such as PyTorch [35] are generally interfaces to many underlying computational packages, as well other frameworks providing the infrastructure, all of which are to some extent abstracted away from users. In other words, such frameworks rarely implement deep learning primitives and operations, such as convolution, or even matrix multiplication. Such operations are typically implemented by computational packages (i.e. LAPACK, cuDNN). In rare cases, state of the art algorithms [12] are also interfaced to provide enhanced performance prior to being fully adopted into standard computational packages.

As a result, primitives such as attention, convolution, linear projection, recurrent layers, and the like tend to provide close-to-optimal performance in terms of throughput, particularly those that are comprised of matrix multiplications. Among the reasons why matrix multiplications in particular are well-optimized are high parallelizability, and the fact that many hardware accelerators geared towards deep learning are equipped with dedicated matrix multiply and accumulate (MMA) cores (i.e. Tensor Cores in NVIDIA GPUs.) In many platforms, different forms of matrix multiplication are computed through the General Matrix-Matrix Multiplication (GEMM) routine. In addition, (discrete) convolution can be modeled as an implicit GEMM problem with limited overhead, and as a result is typically implemented using GEMM routines in many platforms. Implicit GEMM is simply a GEMM routine designed to multiply the convolution activation (extracted sliding windows) matrix by the weight matrix (reshaped kernels). The reason why this can be done efficiently is that the sliding windows are not explicitly extracted and stored in global memory, and are instead loaded into registers or high-bandwidth memory only prior to each block of computation. This also means convolution operations also enjoy increased throughput by running on dedicated MMA cores, such as Tensor Cores.

Implementing neighborhood attention faces a few challenges, the first being its requirement of lower level implementations. Because of the nuances of the method, particularly that in corner cases, implementing it through the typical Python interface is not going to scale. Extracting sliding windows alone will consume exponentially larger amounts of global memory, leading to very limited scalability in terms of training. In addition, this implementation leads to orders of magnitude higher latency compared to alternatives with identical FLOPs. As a result, we carefully implemented GPU kernels to avoid both the excessive memory footprint and latency. We also implemented trivial CPU procedures in C++.

3.3.1 Operations

In order to allow for automatic differentiation for backpropagation in packages such as PyTorch, we implement both forward pass and backward pass kernels in every scenario. There are two forward pass kernels, one for computing attention weights per Eq. (2), and one for computing the output per Eq. (4). As a result there would be four backward pass kernels, one for each of Q, K, V, and $\mathbf{A}$. This makes a total of 6 kernels, which we found can be reduced to 3 kernels. Herein, we describe the 3 operations, and specify their respective kernels. For ease of implementation and portability, we exclude the softmax operation, as it is challenging to fuse into kernels, and if not fused, the routine can simply be imported from existing libraries.

Pointwise-Neighborhood (PN). Computing neighborhood attention weights (Eq. (2)) is a per-token general matrix-vector multiplication (GEMV) between a token's neighboring keys, and its query vector. Given kernel size k, every token's query is a d dimensional vector, and has k neighboring keys, each of which are also d dimensional vectors, which are packed into a $k \times d$ matrix. This is essentially a $1 \times k \times d$ shaped GEMM. Computing the gradient of attention weights is also a per-token GEMV between an output token's gradient, and its neighboring values, with the same shapes. This means that any routine computing neighborhood attention weights can be used to compute the gradient for those weights as well, simply by plugging in the relevant inputs. We label this operation Pointwise-Neighborhood (NN) to illustrate the vector-matrix multiplication between a token (point) and its neighborhood.

Neighborhood-Neighborhood (NN). Applying attention weights (Eq. (4)) can also be expressed as a per-token GEMV. The difference however is in the problem size, which is a $1 \times d \times k$ shaped GEMM. Each token's k attention weights are multiplied by k value vectors, all of which are d dimensional, and then a summation over the k vectors. This can be simplified into multiplying a k dimensional vector by a $k \times d$ dimensional matrix, resulting in a single d dimensional output vector. Computing the gradient of queries, Q, is similarly a GEMV between the gradient of each token's k attention weights, and its k key vectors, resulting in a single d dimensional query gradient vector. We label this operation Neighborhood-Neighborhood (NN), which highlights the key difference with PN: k is the number of accumulations (length of the inner-most loop).

Inverse-Neighborhood (IN). This operation computes gradients for the key-value pair (K and V), and differs from the first two due to the fact that the neighbor relationship is not commutative. Simply put, a token x being in another token y's neighborhood given a fixed neighborhood/kernel size does not imply y is in x's neighborhood with the same neighborhood/kernel size. This is simply due to the very definition of neighborhood attention, and the elimination of the constraint on queries being centered in the neighborhood. This operation is therefore defined as follows: for any given token x, we find the set of *inverse* neighbors, which is comprised of all tokens y where x falls in y's neighborhood. This is a superset of x's neighbors, and in non-corner cases would be equivalent to it. In corner cases, the superset can contain up to 50% more tokens. We label this operation Inverse-Neighborhood (IN), as it is similar in concept as well as the GEMM shape to NN, but requires additional index mapping and additional tokens being loaded.

3.3.2 Challenges

There are a number of challenges that make implementing neighborhood attention in the same manner as other deep learning primitives and operations non-trivial. Those include the following:

1. GEMV cannot scale as well as problems that are formulated as GEMMs (i.e. attention and convolution). This is especially true in the presence of MMA cores (i.e. Tensor Cores).

2. Inverse-Neighborhood requires a more complicated index mapping compared to the other two operations, and the difference in the size of the set of inverse neighbors for every token is difficult to predict at compile time.

3. Extending the operations to two dimensions complicates tiling-based approaches, because in that case tiling would have to be done with respect to the two dimensions, possibly creating additional overhead.

Because of these, we initially started with developing very naive kernels without using caching or higher-bandwidth memory, focusing only on correctness.

3.3.3 Naive kernels

Our naive kernels are simple CUDA kernels that implement the three operations in single-instruction multiple-threads (SIMT) style. PN threads compute the neighborhood range (indices of their corresponding token's neighbors), and perform one vector-vector multiply and accumulate (MAC) operation, and therefore store a single neighborhood attention weight. NN threads similarly compute the neighborhood range, and perform a single vector-vector MAC between neighborhood attention weights, and their corresponding values in a single output dimension. IN threads compute the inverse neighborhood range for the token corresponding to their index, and compute a single vector-vector MAC between the weights from the inverse neighborhood and their corresponding values in a single dimension, following NN in that regard.

Tiling and shared memory. We experimented with re-implementing our PN kernel to prefetch contiguous query and key tokens and compute attention weights for multiple tokens per thread, which is possible due to the assumption that sets of neighbors corresponding to consecutive queries largely intersect (queries close to each other share most of their neighborhoods.) While this resulted in a significant drop in latency, and was further optimized by minimizing bank conflicts, the formulation simply cannot scale to larger problem sets, and extension to different problems sizes requires heavy use of templated arguments as well as code duplication. These make this implementation difficult to extend and maintain. Motivated by this, in addition to the fact that naive kernels cannot reach peak throughput due to frequent reads from lower-bandwidth memory, we attempted to model neighborhood attention as an implicit GEMM problem. Such an implementation would also allow utilization of MMA cores.

3.3.4 Implicit GEMM NA

As previously mentioned, convolutions can be formulated as GEMMs by implicitly forming sliding window activations. This means the global data iterators within the GEMM kernel, which load tiles of the operand matrices, are modified to map activation indices to indices in the original input, therefore forming only tiles of the activation matrix in higher bandwidth memory prior to the main loop performing MMAs. This is preferable to pre-computing the relatively large activation matrix from the input and storing it in global memory. We attempt to formulate neighborhood attention operations (PN, NN, and IN) as implicit GEMM problems. While seemingly impossible, given that all three operations are GEMV problems, we found that the very assumption used in our previous tiling approach (neighborhoods corresponding to consecutive tokens largely intersect) can be transferred to implicit GEMM kernels. In addition, by customizing the epilogue, which is responsible for copying MMA results from registers into global memory, we can mask out invalid attention weights, as long as all valid attention weights[1] are computed within the same cooperative thread array (CTA). Our approach is to implicitly tile inputs as to prevent launching an excessive number of CTAs. This simply means we would launch fewer CTAs compared to a self attention kernel. In our formulation, each CTA effectively computes cross attention between the loaded query tokens, and the key tokens. We design global data iterators that load enough key tokens that would cover all loaded queries, and repeat the CTA with the same query tokens and new key tokens if necessary (useful when not all corresponding keys fit into shared memory.) The main loop is a standard GEMM main loop, performing MMA operations on loaded tiles. This allows for any GEMM routine to replace our existing one without having to re-implement the neighborhood attention logic. Finally, the epilogue loops through computed attention weights between the loaded queries and keys, and only proceeds to store valid ones into global memory (valid refers to those that fall within the defined neighborhood/kernel size). An illustration of this process in a 2-dimensional PN kernel is

[1]In practice we split attention weights into multiple CTAs if the problem size exceeds the bounds of GEMM tile shape. This will not result in any race conditions in our formulation.

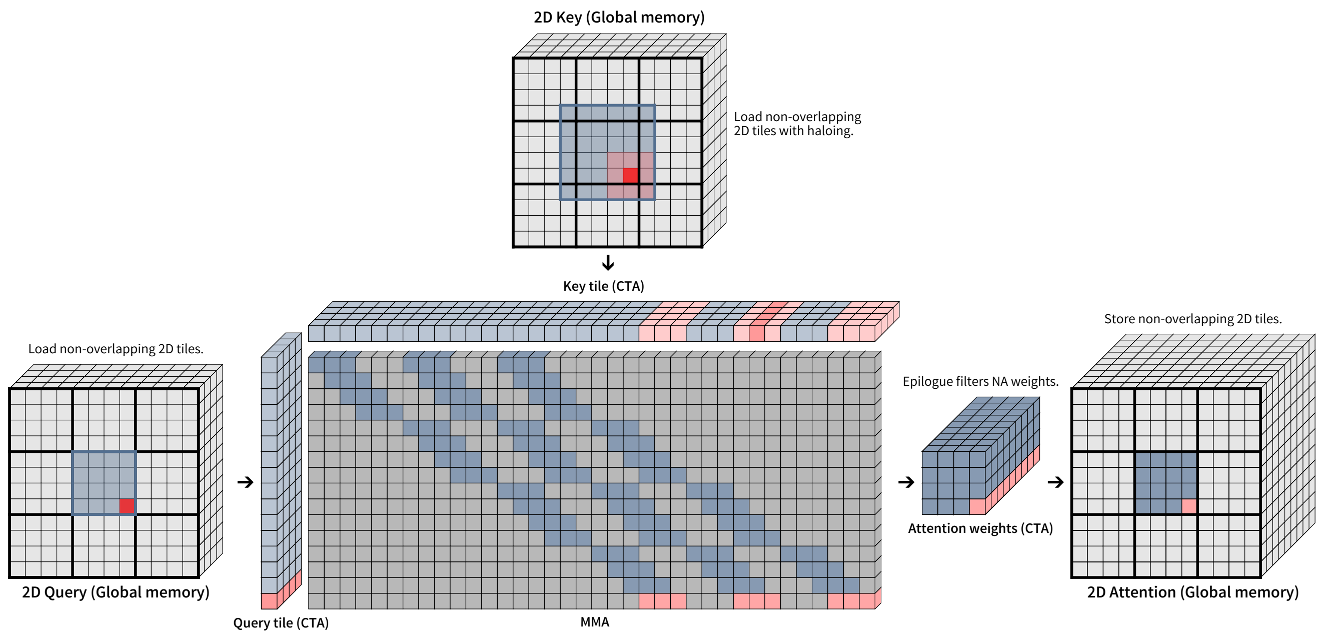
2D Key (Global memory)
Load non-overlapping 2D tiles with haloing.
Key tile (CTA)
Load non-overlapping 2D tiles.
Store non-overlapping 2D tiles.
Epilogue filters NA weights.
Attention weights (CTA)
2D Query (Global memory)
2D Attention (Global memory)
Query tile (CTA)
MMA

Figure 6. Visualization of tiling in implicit GEMM NA. This visualizes the pointwise-neighborhood (PN) operation. In this example, the input is a 12^2 feature map with 4 channels, window size is 3^2. Each CTA will load one non-overlapping query tile and a haloed key tile, computes attention weights, and the epilogue only stores valid weights, ignoring the rest.

presented in Fig. 6. This concept was implemented for both 1-dimensional (token sequence) and 2-dimensional (feature map) neighborhood attention using NVIDIA's CUTLASS library. We followed the GEMM hierarchy in CUTLASS 2.9 (the latest release at the time of development), but hope to extend the implementation to the new design language introduced in CUTLASS 3.0, which would allow targeting the latest architectures from NVIDIA (Ada Lovelace and Hopper.)

3.3.5 Performance improvements from GEMM-based kernels

To evaluate performance improvements from our implicit GEMM kernels, we profiled them against our naive kernels under different practical window sizes, dilation values, and input sizes. Latency values were divided to derive relative improvement in forward pass (QK^T and $\mathbf{A}V$) and backward pass (∇Q, ∇K, ∇V, $\nabla \mathbf{A}$) functions, as well as the total of one forward and backward pass. Those results are presented in Tab. 1 grouped by kernel size, where we can see the GEMM kernels reach approximately 500% to 1300% the throughput of naive kernels in 1-D, and 300% to 500% of their throughput in 2-D. We found that across different problem sizes, implicit GEMM kernels consistently reach significantly higher throughput levels compared to naive kernels, which is to no surprise. In addition, we found that these kernels can also outperform our original tiled PN kernel, which is excluded from the results here for consistency. We've also verified the correctness of and used both 1-dimensional and 2-dimensional neighborhood attention with up to kernel size 63^2.

Kernel size	**Forward pass** QK^T (PN)	$\mathbf{A}V$ (NN)	Total	**Backward pass** $\nabla\mathbf{A}$ (PN)	∇Q (NN)	∇K (IN)	∇V (IN)	Total	**Total**
1-dimensional neighborhood attention									
9	1377 %	105 %	667 %	1401 %	105 %	243 %	243 %	456 %	527 %
25	3208 %	213 %	1573 %	3338 %	213 %	402 %	402 %	958 %	1156 %
49	3544 %	298 %	1796 %	3718 %	298 %	567 %	567 %	1119 %	1327 %
81	3338 %	398 %	1857 %	3584 %	398 %	648 %	648 %	1169 %	1373 %
121	3045 %	407 %	1706 %	3373 %	407 %	698 %	698 %	1141 %	1310 %
169	2157 %	402 %	1386 %	2333 %	402 %	715 %	715 %	1041 %	1150 %
Total	2002 %	155 %	985 %	2060 %	155 %	337 %	337 %	653 %	762 %
2-dimensional neighborhood attention									
$\mathbf{3^2}$	1009%	75%	430%	1026%	75%	153%	153%	257%	304%
$\mathbf{5^2}$	2266%	106%	943%	2274%	106%	202%	202%	436%	558%
$\mathbf{7^2}$	2317%	140%	952%	2332%	140%	235%	235%	442%	556%
$\mathbf{9^2}$	1101%	196%	684%	1114%	196%	268%	268%	404%	473%
$\mathbf{11^2}$	1128%	265%	726%	1150%	265%	286%	286%	424%	495%
$\mathbf{13^2}$	1007%	286%	714%	1021%	286%	305%	305%	436%	504%
Total	1322%	188%	745%	1339%	188%	257%	257%	413%	493%

Table 1. Relative throughput comparison between naive and implicit GEMM neighborhood attention kernels. Kernels were profiled across different problem sizes and grouped by kernel size.

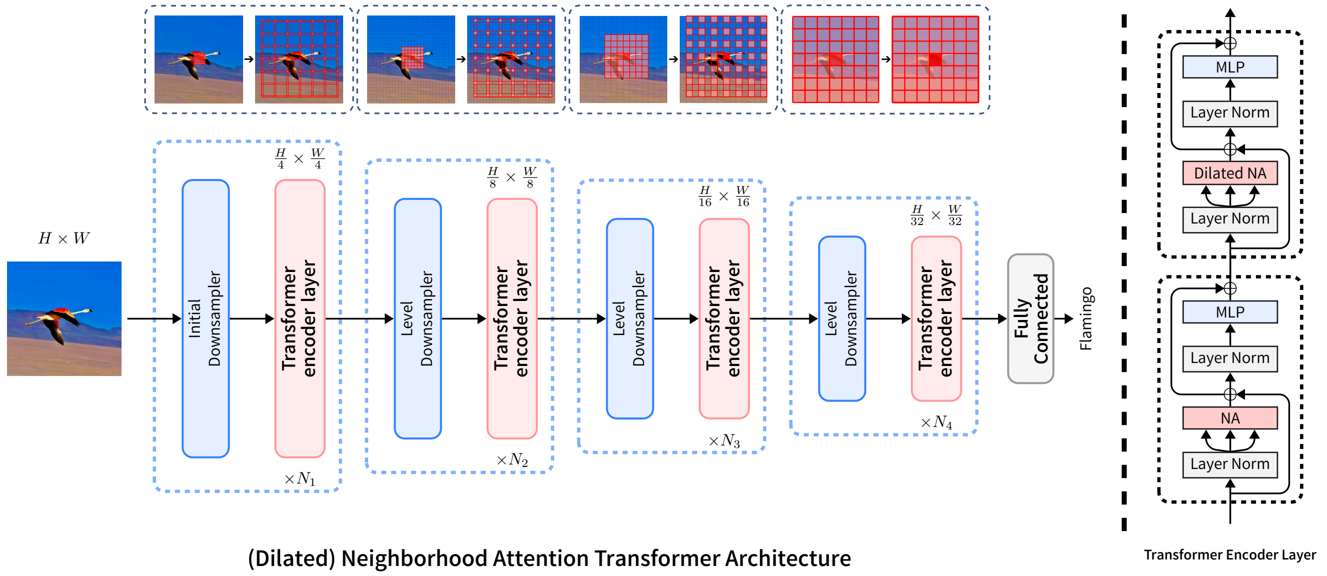
$H \times W$
Initial Downsampler
Transformer encoder layer
$\frac{H}{4} \times \frac{W}{4}$
$\times N_1$
Level Downsampler
$\frac{H}{8} \times \frac{W}{8}$
$\times N_2$
$\frac{H}{16} \times \frac{W}{16}$
$\times N_3$
$\frac{H}{32} \times \frac{W}{32}$
$\times N_4$
Fully Connected
Flamingo
(Dilated) Neighborhood Attention Transformer Architecture
MLP
Layer Norm
Dilated NA
NA
Transformer Encoder Layer

Figure 7. An illustration of the hierarchical NAT/DiNAT architecture. Inputs are downsampled to a quarter of their original spatial resolution, and go through 4 levels of Transformer encoders. Feature maps are downsampled to half their spatial size and doubled in channels between levels. In NAT variants, self attention is replaced with neighborhood attention. In DiNAT variants, we alternate between non-dilated neighborhood attention, and dilated neighborhood attention.

3.3.6 $\mathcal{N}$ATTEN

Our implementations are primarily C++ and CUDA kernels, which require compilation and binding to Python in order to support running as part of frameworks such as PyTorch. In order to minimize the effort required to do so, we packaged all of our implementations, along with Python bindings and autograd support for PyTorch, as a Python package, which can be installed via `pip`. The package, Neighborhood Attention Extension ($\mathcal{N}ATTEN$), comes with pre-built binaries for the latest PyTorch releases, allowing researchers to plug neighborhood attention into their work within seconds. $\mathcal{N}ATTEN$'s latest public release only includes our naive kernels, and our early tiled PN kernels, which is sufficient for many scenarios, but certainly not always competitive in terms of throughput. However, upon releasing our implicit GEMM backend, users should observe a significant improvement in latency, both in 1-dimensional, and 2-dimensional neighborhood attention, simply by upgrading $\mathcal{N}ATTEN$.

3.4. Architecture design

In order to evaluate neighborhood attention, we create vision architectures based on it, which follow the trend of hierarchical vision transformers, such as Swin [31]. While architecture design is not the focus of this paper, we tweaked our architecture to further enhance performance by re-introducing convolutional downsamplers (as opposed to non-overlapping, a.k.a patched, downsampling), and reducing the additional complexity and memory footprint by reducing the size of inverted bottlenecks. We label the new architectural configuration Neighborhood Attention Transformer (NAT), and summarize its variants ranging from 20 million parameters to 200 million parameters in Tab. 2, with an illustration of the design in Fig. 7. Variants using dilated neighborhood attention are dubbed DiNAT.

Variant	Layers per level	Dim × Heads	MLP ratio	# of Params	FLOPs
● (Di)NAT-Mini	3, 4, 6, 5	32 × 2	3	20 M	2.7 G
● (Di)NAT-Tiny	3, 4, 18, 5	32 × 2	3	28 M	4.3 G
● (Di)NAT-Small	3, 4, 18, 5	32 × 3	2	51 M	7.8 G
● (Di)NAT-Base	3, 4, 18, 5	32 × 4	2	90 M	13.7 G
● (Di)NAT-Large	3, 4, 18, 5	32 × 6	2	200 M	30.6 G

Table 2. NAT/DiNAT variants. Channels (number of attention heads and embedding dimension) double after every level except the last. No dilation is used in NAT variants, whereas in DiNAT variants we alternate between neighborhood attention with and without dilation.

4. Experiments

Herein, we present experiments with multiple vision architectures, including our proposed configuration from Sec. 3.4, NAT and DiNAT. We trained these models on image classification, region-based object detection and image segmentation. Unless otherwise stated, we used neighborhood attention with a kernel size 7^2.

4.1. Image classification

Following existing and related works, we train our image classification models on ImageNet-1K [13] for 300 epochs, the first 20 of which warm up to a learning rate of 1e-3. We used a batch size of 1024, and an additional 10 cooldown epochs in addition to the 300, again following the standard practice [51]. Our larger scale variants on the other hand are pre-trained on the superset, ImageNet-22K, for 90 epochs, and fine-tuned on the original ImageNet-1K for an additional 30 epochs. For this task, we experimented with not only our proposed architecture, but also with the Swin Transformer architecture [31], which is similar in concept to our NAT/DiNAT architecture, but differs in downsampling, as well as the number of layers and inverted bottleneck sizes (see Sec. 3.4). We additionally experimented with the MaxViT architecture [46], which is a state-of-the-art hybrid architecture comprised of attention and convolution. We also experimented with the original ViT architecture [15], in order to observe both performance changes from restricting self attention, and to observe the impact of dilation.

4.1.1 NAT and DiNAT

Tab. 3 summarizes our ImageNet-1K experiments with NAT and DiNAT. Within these experiments, we see that dilation has little effect on throughput, but performance levels vary. In smaller scale variants, dilated variants either match or fall short of non-dilated variants. However, we found these instances to be the exception, as we will see variants with dilation significantly outperform those without dilation, both in image classification at scale, and downstream tasks. Moreover, we did not observe the same effect in other architectures, such as Swin Transformer's architecture.

Model	**# of Params**	**FLOPs**	**Thru.** (imgs/sec)	**Memory** (GB)	**Top-1** (%)
○ **NAT-M**	20 M	2.7 G	2297	2.4	**81.8**
● **DiNAT-M**	20 M	2.7 G	2245	2.4	**81.8**
○ **NAT-T**	28 M	4.3 G	1664	2.5	**83.2**
● **DiNAT-T**	28 M	4.3 G	1619	2.5	82.7
○ **NAT-S**	51 M	7.8 G	1130	3.7	83.7
● **DiNAT-S**	51 M	7.8 G	1142	3.7	**83.8**
○ **NAT-B**	90 M	13.7 G	856	5.0	84.3
● **DiNAT-B**	90 M	13.7 G	864	5.0	**84.4**

Table 3. ImageNet classification performance with NAT/DiNAT architecture. Dilation almost always improves accuracy, with very limited effect on latency, and no additional computational cost. The exception here are the Mini and Tiny variants, where dilation does not improve classification accuracy. Despite this, we see the same variants outperform its non-dilated counterpart in downstream tasks. Throughput and peak memory usage are measured from forward passes with a batch size of 256 on a single A100 GPU.

4.1.2 With Swin Transformer

We replaced blocked attention in Swin Transformer variants with neighborhood attention, and dub the resulting model NAT$_s$. In addition, we created a dilated counterpart for said model, where we replace layers with blocked attention with neighborhood attention, but replace layers with shifted blocked attention with dilated neighborhood attention, and dub these models DiNAT$_s$. Results are presented in Tab. 4. In every instance, variants powered by neighborhood attention can run noticeably faster, use less memory (from lack of pixel shifts, and the like), and enjoy improved classification accuracy. We also experimented with replacing only the shifted blocked attention with sparse blocked attention (per MaxViT [46].) We observed that while this change reduces memory usage and improves throughput compared to shifted blocked attention, it simply does not catch up with variants using dilated neighborhood attention in terms of performance, as its performance at scale can be even worse than the original Swin counterparts.

Model	**Attn**	**# of Params**	**FLOPs**	**Thru.** (imgs/sec)	**Memory** (GB)	**Top-1** (%)
○ **Swin-T**	BA / shifted BA	28 M	4.5 G	1988	4.8	81.2
●	BA / sparse BA	28 M	4.5 G	2091	4.5	81.1
○ **NAT$_s$-T**	NA / NA	28 M	4.5 G	**2146**	**4.0**	**81.8**
● **DiNAT$_s$-T**	NA / DiNA	28 M	4.5 G	2100	**4.0**	**81.8**
○ **Swin-S**	BA / shifted BA	50 M	8.7 G	1225	5.0	83.0
●	BA / sparse BA	50 M	8.7 G	1292	4.7	82.9
○ **NAT$_s$-S**	NA / NA	50 M	8.7 G	**1323**	**4.1**	83.2
● **DiNAT$_s$-S**	NA / DiNA	50 M	8.7 G	1293	**4.1**	**83.5**
○ **Swin-B**	BA / shifted BA	88 M	15.4 G	892	6.7	83.5
●	BA / sparse BA	88 M	15.4 G	934	6.3	83.1
○ **NAT$_s$-B**	NA / NA	88 M	15.4 G	**979**	**5.5**	83.5
● **DiNAT$_s$-B**	NA / DiNA	88 M	15.4 G	957	**5.5**	**83.8**

Table 4. ImageNet classification performance with Swin architecture. In addition to Swin's blocked attention with shifts (local), and our neighborhood attention with dilation (local and sparse global) patterns, we include MaxViT's local and sparse blocked attention pattern (also local and sparse global). Throughput and peak memory usage are measured from forward passes with a batch size of 256 on a single A100 GPU.

4.1.3 Large-scale pre-training.

We scaled the aforementioned variants to 200M parameters and pre-trained them on ImageNet-22K for 90 epochs. We then fine-tuned each model on ImageNet-1K for 30 epochs, with a batch size of 512, and a linear learning rate schedule with no warmup, and a base learning rate of 5e-5, and weight decay rate of 1e-4, following standard practice. Final ImageNet-1K validation set accuracy levels, along with number of parameters, FLOPs, throughput, and memory usage are provided in Tab. 5. We observe that models based on neighborhood attention continue to outperform Swin with competitive throughput.

Model	Res.	# of Params	FLOPs	Thru. (imgs/sec)	Memory (GB)	Top-1 (%)
○ **Swin-L**	224^2	197 M	34.5 G	540	10.4	86.3
● **DiNAT$_s$-L**	224^2	197 M	34.5 G	585	8.6	86.5
● **DiNAT-L**	224^2	200 M	30.6 G	545	7.8	**86.6**
○ **Swin-L**†	384^2	197 M	104.0 G	189	32.7	87.3
● **DiNAT$_s$-L**	384^2	197 M	101.5 G	185	22.6	87.4
● **DiNAT-L**	384^2	200 M	89.7 G	172	20.1	87.4
● **DiNAT-L**†	384^2	200 M	92.4 G	135	26.9	**87.5**

Table 5. ImageNet classification performance with ImageNet-22K pre-training. †indicates increased window size from 7^2 to 11^2 (DiNAT) and 12^2 (Swin). Throughput and peak memory usage are measured from forward passes with a batch size of 256 on a single A100 GPU.

4.1.4 With MaxViT

As discussed in Sec. 2, MaxViT [46] is a state-of-the-art hybrid architecture comprised of convolutions and attention. Similar to Swin, MaxViT includes blocked attention, but unlike Swin follows it with a sparse blocked attention pattern. Replacing blocked attention with neighborhood attention, and sparse blocked attention with dilated neighborhood attention is therefore a natural fit, which is what we present in Tab. 6. We still observe a slight throughput improvement in speed and at times memory, which can be attributed to blocked attention implementations still requiring frequent pixel permutations, while neighborhood attention based on $\mathcal{NATTEN}$ does not. That said, both approaches have identical computational cost, similar to experiments in Tab. 4, and the throughput difference is simply related to implementation. Note that performance gaps here are different from those in Tab. 4, because this is a hybrid architecture, and attention is not the primary operator throughout the model.

Model	Attn	# of Params	FLOPs	Thru. (imgs/sec)	Memory (GB)	Top-1 (%)
● **MaxViT-T**	BA / sparse BA	31 M	5.6 G	912	10.7	83.9
●	NA / DiNA	31 M	5.6 G	**990**	10.7	**84.0**
● **MaxViT-S**	BA / sparse BA	69 M	11.7 G	603	15.6	84.7
●	NA / DiNA	69 M	11.7 G	**655**	15.6	**84.8**
● **MaxViT-B**	BA / sparse BA	120 M	24.1 G	331	16.0	84.8
●	NA / DiNA	120 M	24.1 G	**361**	15.8	**84.9**

Table 6. ImageNet classification performance with MaxViT architecture. Models based on neighborhood attention can enjoy slightly improved accuracy. Performance gap difference between this table and Tab. 4 highlights the role that the hybrid architecture plays. Throughput and peak memory usage are measured from forward passes with a batch size of 256 on a single A100 GPU.

4.1.5 With ViT

We also experimented with restricting self attention in the original ViT architecture. We present those experiments in Tab. 7, where we simply replace self attention (with relative positional biases) once with neighborhood attention alone, and once with our non-dilated and dilated combination. We used a 7^2 kernel size, similar to previous architectures, and feature map sizes in this setting are 14^2. We observe that without dilation, neighborhood attention alone results in a more significant drop in accuracy compared to the combination of dilated and non-dilated patterns. This further highlights the importance of local and sparse global attention patterns being used in conjunction, because together they can capture more global context, and reach a larger receptive field with the same number of layers.

Model	Attn	# of Params	FLOPs	Thru. (imgs/sec)	Memory (GB)	Top-1 (%)
○	**NA/NA**	22 M	4.3 G	3348	1.3	80.0
●	**NA/DiNA**	22 M	4.3 G	3255	1.3	80.8
● **ViT-S**	**SA/SA**	22 M	4.6 G	3070	1.9	**81.2**
○	**NA/NA**	86 M	16.9 G	1413	2.7	81.6
●	**NA/DiNA**	86 M	16.9 G	1386	2.7	82.1
● **ViT-B**	**SA/SA**	86 M	17.5 G	1288	3.7	**82.5**

Table 7. ImageNet classification performance with ViT architecture. NA and DiNA restrict SA, therefore performance drops are expected. However, NA/DiNA do not bear the quadratic complexity and memory footprint. Throughput and peak memory usage are measured from forward passes with a batch size of 256 on a single A100 GPU.

4.1.6 Comparison to ConvNeXt

ConvNeXt [32] is a relatively modern baseline for convolutional neural networks. Its adoption of the transformer design language, and more specifically that of hierarchical transformers such as Swin [31], make it a good candidate for comparison to attention-based models. The key difference in ConvNeXt is that instead of a windowed attention operator and its linear projections (Q, K, V), a depthwise convolution is used, which is followed by a linear projection, similar to attention, making it a depthwise separable convolution (linear projection is equivalent to a 1×1 convolution.) However, this design contains far fewer parameters and FLOPs per layer compared to attention-based models, which is why ConvNeXt variants are noticeably deeper models. Because of this difference, it is impractical to compare to ConvNeXt by solely replacing the convolution operator with neighborhood attention. As a result, we simply compare some of the models already presented, DiNAT$_s$ and DiNAT, to ConvNeXt directly. ConvNeXt's primary attention-based baseline was Swin Transformer [31], and DiNAT$_s$ has the same architecture and configuration as

Swin, with the exception of the attention pattern. In addition, DiNAT also closely follows Swin's design. This comparison is presented in Tab. 8, where we see DiNAT$_s$ reach competitive performance with ConvNeXt, and DiNAT outperforming ConvNeXt without large-scale pre-training. In larger scale models which were pre-trained on ImageNet-22K, we see DiNAT matching ConvNeXt's performance.

Model	**Res.**	**# of Params**	**FLOPs**	**Thru.** (imgs/sec)	**Memory** (GB)	**Top-1** (%)
			ImageNet-1K trained models			
● **DiNAT-M**	224^2	20 M	2.7 G	2245	2.4	**81.8**
○ **Swin-T**	224^2	28 M	4.5 G	1988	4.8	81.3
● **DiNAT$_s$-T**	224^2	28 M	4.5 G	2100	4.0	81.9
● **ConvNeXt-T**	224^2	28 M	4.5 G	1976	3.4	82.1
● **DiNAT-T**	224^2	28 M	4.3 G	1619	2.5	**82.7**
○ **Swin-S**	224^2	50 M	8.7 G	1225	5.0	83.0
● **DiNAT$_s$-S**	224^2	50 M	8.7 G	1293	4.1	83.5
● **ConvNeXt-S**	224^2	50 M	8.7 G	1229	3.5	83.1
● **DiNAT-S**	224^2	51 M	7.8 G	1142	3.7	**83.8**
○ **Swin-B**	224^2	88 M	15.4 G	892	6.7	83.5
● **DiNAT$_s$-B**	224^2	88 M	15.4 G	957	5.5	83.8
● **ConvNeXt-B**	224^2	89 M	15.4 G	887	4.8	83.8
● **DiNAT-B**	224^2	90 M	13.7 G	864	5.0	**84.4**
			ImageNet-22K pre-trained models			
○ **Swin-L**	224^2	197 M	34.5 G	540	10.4	86.3
● **DiNAT$_s$-L**	224^2	197 M	34.5 G	585	8.6	86.5
● **ConvNeXt-L**	224^2	198 M	34.4 G	531	7.5	**86.6**
● **DiNAT-L**	224^2	200 M	30.6 G	545	7.8	**86.6**
○ **Swin-L**[†]	384^2	197 M	104.0 G	189	32.7	87.3
● **DiNAT$_s$-L**	384^2	197 M	101.5 G	185	22.6	87.4
● **ConvNeXt-L**	384^2	198 M	101.1 G	179	19.2	**87.5**
● **DiNAT-L**	384^2	200 M	89.7 G	172	20.1	87.4
● **DiNAT-L**[†]	384^2	200 M	92.4 G	135	26.9	**87.5**

Table 8. ImageNet-1K image classification performance. [†]indicates increased window size from 7^2 to 11^2 (DiNAT) and 12^2 (Swin). Throughput and peak memory usage are measured from forward passes with a batch size of 256 on a single A100 GPU.

4.2. Object detection and instance segmentation

We extend variants based on Swin and our NAT/DiNAT architecture to region-based object detection and instance segmentation, which we present in Tab. 9. We observe that DiNAT shows noticeable improvement over NAT, with little-to-no drop in throughput. There are even instances where DiNAT even surpasses NAT's throughput, but within the margin of error. We also find that similar to classification, neighborhood attention in the Swin architecture (DiNAT$_s$) enjoys higher throughput and improved performance compared to the original Swin variants based on blocked attention. Additionally, we observe that DiNAT stays ahead of ConvNeXt [32], including in large scale variants, which was not the case in classification.

Backbone	**# of Params**	**FLOPs**	**Thru.** (FPS)	**AP^b**	**AP$^b_{50}$**	**AP$^b_{75}$**	**AP^m**	**AP$^m_{50}$**	**AP$^m_{75}$**
				Mask R-CNN - 3x schedule					
○ **NAT-M**	40 M	225 G	55.4	46.5	68.1	51.3	41.7	65.2	44.7
● **DiNAT-M**	40 M	225 G	54.2	**47.2**	**69.1**	**51.9**	**42.5**	**66.0**	**45.9**
○ **Swin-T**	48 M	267 G	48.9	46.0	68.1	50.3	41.6	65.1	44.9
● **DiNAT$_s$-T**	48 M	263 G	55.0	46.6	68.8	51.3	42.1	65.7	45.4
● **ConvNeXt-T**	48 M	262 G	53.1	46.2	67.0	50.8	41.7	65.0	44.9
○ **NAT-T**	48 M	258 G	44.7	47.7	69.0	52.6	42.6	66.1	45.9
● **DiNAT-T**	48 M	258 G	44.8	**48.6**	**70.2**	**53.4**	**43.5**	**67.3**	**46.8**
○ **Swin-S**	69 M	359 G	34.2	48.5	70.2	53.5	43.3	67.3	46.6
● **DiNAT$_s$-S**	69 M	350 G	40.7	48.6	70.4	53.2	43.5	67.6	46.9
○ **NAT-S**	70 M	330 G	35.7	48.4	69.8	53.2	43.2	66.9	46.5
● **DiNAT-S**	70 M	330 G	35.7	**49.3**	**70.8**	**54.2**	**44.0**	**68.0**	**47.4**
				Cascade Mask R-CNN - 3x schedule					
○ **NAT-M**	77 M	704 G	28.4	50.3	68.9	54.9	43.6	66.4	47.2
● **DiNAT-M**	77 M	704 G	28.2	**51.2**	**69.8**	**55.7**	**44.4**	**67.3**	**47.8**
○ **Swin-T**	86 M	745 G	26.6	50.4	69.2	54.7	43.7	66.6	47.3
● **DiNAT$_s$-T**	86 M	742 G	28.0	51.0	69.9	55.4	44.1	67.3	47.6
● **ConvNeXt-T**	86 M	741 G	27.7	50.4	69.1	54.8	43.7	66.5	47.3
○ **NAT-T**	85 M	737 G	24.8	51.4	70.0	55.9	44.5	67.6	47.9
● **DiNAT-T**	85 M	737 G	25.3	**52.2**	**71.0**	**56.8**	**45.1**	**68.3**	**48.8**
○ **Swin-S**	107 M	838 G	21.6	51.8	70.4	56.3	44.7	67.9	48.5
● **DiNAT$_s$-S**	107 M	829 G	24.0	52.3	71.2	56.7	45.2	68.6	49.1
● **ConvNeXt-S**	108 M	827 G	23.5	51.9	70.8	56.5	45.0	68.4	49.1
○ **NAT-S**	108 M	809 G	22.2	52.0	70.4	56.3	44.9	68.1	48.6
● **DiNAT-S**	108 M	809 G	22.1	**52.9**	**71.8**	**57.6**	**45.8**	**69.3**	**49.9**
○ **Swin-B**	145 M	982 G	18.6	51.9	70.9	56.5	45.0	68.4	48.7
● **DiNAT$_s$-B**	145 M	966 G	20.7	52.6	71.5	57.2	45.3	68.8	49.1
● **ConvNeXt-B**	146 M	964 G	19.7	52.7	71.3	57.2	45.6	68.9	49.5
○ **NAT-B**	147 M	931 G	19.1	52.3	70.9	56.9	45.1	68.3	49.1
● **DiNAT-B**	147 M	931 G	18.7	**53.4**	**72.1**	**58.2**	**46.2**	**69.7**	**50.2**
○ **Swin-L**[⋆‡]	253 M	1393 G	14.1	53.7	72.2	58.7	46.4	69.9	50.7
● **DiNAT$_s$-L**[‡]	253 M	1357 G	15.8	54.8	74.2	59.8	47.2	71.3	51.2
● **ConvNeXt-L**[‡]	253 M	1354 G	15.1	54.8	73.8	59.8	47.6	71.3	51.7
● **DiNAT-L**[‡]	258 M	1276 G	14.1	**55.3**	**74.3**	**60.2**	**47.8**	**71.8**	**52.0**

Table 9. COCO object detection and instance segmentation performance using Cascade Mask R-CNN [4]. [‡]indicates that the model was pre-trained on ImageNet-22K. [⋆]Swin-L was not reported with Cascade Mask R-CNN, therefore we trained it with the checkpoint released in their official repository. Throughput was measured from single-batch forward passes on a single A100 GPU.

4.3. Semantic segmentation

Tab. 10 presents results in training a semantic segmentation framework, UPerNET [52], with the aforementioned architectures. We observe again that DiNAT always outperforms NAT with very little drop in throughput, and DiNAT$_s$ improves Swin in both throughput and performance. In addition, we observe DiNAT maintains its place ahead of both models, as well as ConvNeXt, at scale with ImageNet-22K pre-training.

4.4. Image segmentation with Mask2Former

To analyze image segmentation performance further, we conducted experiments with Mask2Former [7], which is a Transformer-based segmentation framework capable of targeting instance, semantic, and panoptic segmentation. In this set of experiments, we specifically focus on large scale image segmentation, following the original paper in us-

Backbone	Res.	# of Params	FLOPs	Thru. (FPS)	mIoU single scale	mIoU multi scale
○ NAT-M	2048 × 512	50 M	900 G	24.9	45.1	46.4
● DiNAT-M	2048 × 512	50 M	900 G	24.7	**45.8**	**47.2**
○ Swin-T	2048 × 512	60 M	946 G	23.0	44.5	45.8
● DiNAT$_s$-T	2048 × 512	60 M	941 G	24.6	46.0	47.4
● ConvNeXt-T	2048 × 512	60 M	939 G	23.9	46.0	46.7
○ NAT-T	2048 × 512	58 M	934 G	22.3	47.1	48.4
● DiNAT-T	2048 × 512	58 M	934 G	22.0	**47.8**	**48.8**
○ Swin-S	2048 × 512	81 M	1040 G	18.6	47.6	49.5
● DiNAT$_s$-S	2048 × 512	81 M	1030 G	20.6	48.6	**49.9**
● ConvNeXt-S	2048 × 512	82 M	1027 G	19.6	48.7	49.6
○ NAT-S	2048 × 512	82 M	1010 G	18.6	48.0	49.5
● DiNAT-S	2048 × 512	82 M	1010 G	18.6	**48.9**	**49.9**
○ Swin-B	2048 × 512	121 M	1188 G	16.0	48.1	49.7
● DiNAT$_s$-B	2048 × 512	121 M	1173 G	17.9	49.4	50.2
● ConvNeXt-B	2048 × 512	122 M	1170 G	16.9	49.1	49.9
○ NAT-B	2048 × 512	123 M	1137 G	16.1	48.5	49.7
● DiNAT-B	2048 × 512	123 M	1137 G	15.9	**49.6**	**50.4**
○ Swin-L†‡	2560 × 640	234 M	2585 G	12.0	-	53.5
● DiNAT$_s$-L‡	2560 × 640	234 M	2466 G	12.6	53.4	54.6
● ConvNeXt-L‡	2560 × 640	235 M	2458 G	12.4	53.2	53.7
● DiNAT-L‡	2560 × 640	238 M	2335 G	11.7	**54.0**	**54.9**

Table 10. ADE20K semantic segmentation performance using UPerNet [52]. ‡indicates that the model was pre-trained on ImageNet-22K. †indicates increased window size from the default 7^2 to 12^2. Throughput was measured from single-batch forward passes on a single A100 GPU.

ing a model with approximately 200 million parameters pre-trained on ImageNet-22K. The original paper experimented with Swin Transformer's large variant, and we in turn experiment with our large DiNAT variant. We trained Mask2Former on MS-COCO [29], ADE20K [57], and Cityscapes [10], on all segmentation objectives for which the datasets provided annotations. We present instance segmentation results in Tab. 11, semantic segmentation results in Tab. 12, and panoptic segmentation results in Tab. 13. We note that DiNAT-L is using an 11^2 kernel size, instead of Swin-L's 12^2, since even-sized windows break the symmetry in NA and are therefore not defined. DiNAT-L outperforms Swin-L on all three tasks and datasets in the primary metrics.

4.5. Neighborhood size, dilation size, and order of layers

In this subsection, we present experiments aimed at finding the effects of dilation values, the NA-DiNA order, kernel sizes, and changes in dilation during inference.

Dilation values. Tab. 14 summarizes effects of different dilation values on classification, detection, instance segmentation and semantic segmentation. Dilation can only be extended only up to the bounds of the input and cannot overflow, and as a result, increased dilation (16, 8, 4, 2) is only applicable to downstream tasks. Therefore, "8, 4, 2, 1" is the maximum applicable dilation to ImageNet at 224 × 224 resolution. Larger dilation values are therefore possible when dealing with larger resolutions, which is the case for downstream tasks. We explored input-dependent dilation values, where DiNA layers apply the maximum possible dilation, which is the floor of resolution divided by kernel size ("Maximum" in Tab. 14). We finally chose settle on "gradual" dilation, in which we gradually increase dilation to the maximum level defined. More details are presented in Sec. 4.6.

Backbone	Win. Size	# of Params	FLOPs	AP	AP^{50}	AP^{S}	AP^{M}	AP^{L}
MS-COCO								
○ Swin-L	12 × 12	216 M	641 G	50.1	-	29.9	53.9	**72.1**
● DiNAT-L	11 × 11	220 M	522 G	**50.8**	**75.0**	**30.9**	**54.7**	**72.1**
ADE20K								
○ Swin-L	12 × 12	216 M	654 G	34.9	-	**16.3**	**40.0**	54.7
● DiNAT-L	11 × 11	220 M	535 G	**35.4**	-	**16.3**	39.0	**55.5**
Cityscapes								
○ Swin-L	12 × 12	216 M	641 G	43.7	71.4	-	-	
● DiNAT-L	11 × 11	220 M	522 G	**45.1**	**72.6**	-	-	-

Table 11. Instance segmentation performance with Mask2Former. All backbones were pre-trained on ImageNet-22K. FLOPs are reported with respect to resolution 800^2.

Backbone	Win. Size	# of Params	FLOPs	mIoU single scale	mIoU multi scale
ADE20K					
○ Swin-L	12 × 12	215 M	636 G	56.1	57.3
● DiNAT-L	11 × 11	220 M	518 G	**57.3**	**58.1**
Cityscapes					
○ Swin-L	12 × 12	215 M	627 G	83.3	84.3
● DiNAT-L	11 × 11	220 M	509 G	**83.9**	**84.5**

Table 12. Semantic segmentation performance with Mask2Former. All backbones were pre-trained on ImageNet-22K. FLOPs are reported with respect to resolution 800^2.

Backbone	Win. Size	# of Params	FLOPs	PQ	PQTh	PQSt	AP$^{Th}_{pan}$	mIoU$_{pan}$
MS-COCO								
○ Swin-L	12 × 12	216 M	658 G	57.8	64.2	48.1	48.6	67.4
● DiNAT-L	11 × 11	220 M	540 G	**58.5**	**64.9**	**48.8**	**49.2**	**68.3**
ADE20K								
○ Swin-L	12 × 12	216 M	660 G	48.1	-	-	34.2	54.5
● DiNAT-L	11 × 11	220 M	542 G	**49.4**	-	-	**35.0**	**56.3**
Cityscapes								
○ Swin-L	12 × 12	216 M	643 G	66.6	-	-	43.6	82.9
● DiNAT-L	11 × 11	220 M	525 G	**67.2**	-	-	**44.5**	**83.4**

Table 13. Panoptic segmentation performance with Mask2Former. All backbones were pre-trained on ImageNet-22K. FLOPs are reported with respect to resolution 800^2.

Model	Dilation per level	ImageNet Top-1 (%)	MSCOCO AP^b	AP^m	ADE20K mIoU
○ **Swin-Tiny**	Not Applicable	81.3	46.0	41.6	45.8
○ **NAT$_s$-Tiny**	1, 1, 1, 1	81.8	46.1	41.5	46.2
● **DiNAT$_s$-Tiny**	8, 4, 2, 1	81.8	46.3	41.6	46.7
● **DiNAT$_s$-Tiny**	16, 8, 4, 2	-	**46.4**	**41.8**	47.1
● **DiNAT$_s$-Tiny**	Maximum	81.8	**46.4**	**41.9**	47.0
● **DiNAT$_s$-Tiny**	Gradual	-	**46.6**	**42.1**	**47.4**
○ **NAT-Tiny**	1, 1, 1, 1	83.2	47.7	42.6	48.4
● **DiNAT-Tiny**	8, 4, 2, 1	82.7	48.0	42.9	48.5
● **DiNAT-Tiny**	16, 8, 4, 2	-	48.3	43.4	48.5
● **DiNAT-Tiny**	Maximum	82.7	**48.6**	**43.5**	48.7
● **DiNAT-Tiny**	Gradual	-	**48.6**	**43.5**	**48.8**

Table 14. Dilation impact on performance. Dilation values beyond ”8, 4, 2, 1” are only applicable to downstream tasks, as their larger resolution allows for it. Maximum dilation indicates it is set to the maximum possible value based on input size. It would be the same as ”8, 4, 2, 1” for ImageNet. Gradual dilation indicates that dilation values in DiNA layers increase gradually.

NA-DiNA *vs*. DiNA-NA. We also experimented with models with DiNA layers before NA layers, as opposed to our final choice in which we begin with NA layers. While the local-global order (NA-DiNA) was our initial choice, we've also found it to be the more effective choice. We also tried a model with only DiNA modules, and found that it performs significantly worse than other combinations. This highlights the importance of having a combination of both local and sparse global attention patterns in the model. Results are summarized in Tab. 15.

Variant	Layer structure	ImageNet Top-1 (%)	MSCOCO AP^b	AP^m	ADE20K mIoU
○○ **Swin-Tiny**	WSA-SWSA	81.3	46.0	41.6	45.8
○○ **NAT$_s$-Tiny**	NA-NA	**81.8**	46.1	41.5	46.2
○● **DiNAT$_s$-Tiny**	NA-DiNA	**81.8**	46.4	**41.8**	**47.1**
●○	DiNA-NA	81.5	**46.5**	**41.8**	46.9
●●	DiNA-DiNA	79.7	39.8	36.8	40.7
○○ **NAT-Tiny**	NA-NA	**83.2**	47.7	42.6	48.4
○● **DiNAT-Tiny**	NA-DiNA	82.7	48.3	43.4	**48.5**
●○	DiNA-NA	82.6	**48.5**	**43.5**	47.9
●●	DiNA-DiNA	82.2	44.9	40.5	45.8

Table 15. Layer structure impact on performance. Our final model has the local-global (NA-DiNA) order.

Kernel size. We summarize experiments with different kernel sizes in Tab. 16. We observed that a DiNAT-Tiny sees a significant decay in performance with a smaller kernel size across all three tasks. However, we find increasing kernel size beyond the default 7×7 does not result in a significant increase in performance.

Test-time dilation changes. We present an analysis of sensitivity to dilation values, in which we attempt different

Model	Win. size	ImageNet Top-1	Thru.	MSCOCO AP^b	AP^m	Thru.	ADE20K mIoU	Thru.
○ **NAT-T**	5^2	**81.6**	1828 imgs/sec	46.8	42.0	48.4 fps	46.3	23.4 fps
● **DiNAT-T**	5^2	81.3	1788 imgs/sec	**47.6**	**42.7**	48.1 fps	**46.4**	23.2 fps
○ **NAT-T**	7^2	**83.2**	1664 imgs/sec	47.7	42.6	44.7 fps	48.4	22.3 fps
● **DiNAT-T**	7^2	82.7	1619 imgs/sec	**48.3**	**43.4**	44.8 fps	**48.5**	22.0 fps
○ **NAT-T**	9^2	**83.1**	1263 imgs/sec	48.5	43.3	40.7 fps	48.1	20.6 fps
● **DiNAT-T**	9^2	**83.1**	1245 imgs/sec	**48.8**	**43.5**	40.2 fps	**48.4**	20.4 fps

Table 16. Kernel size impact on performance. Note that we set dilation to the maximum values possible in each block based on the default resolutions. Therefore, the variant with kernel size 5 has larger dilation values compared to the one with kernel size 7.

dilation values on already trained models, and evaluate their performance. This can be particularly important to cases with varying resolutions (i.e. multi-scale testing). The results are presented in Tab. 17.

Model	Dilation Train	Test	ImageNet Top-1 (%)	MSCOCO AP^b	AP^m	ADE20K mIoU
○ **NAT-T**	1, 1, 1, 1	1, 1, 1, 1	83.2	47.7	42.6	48.4
	1, 1, 1, 1	8, 4, 2, 1	81.0	42.6	39.5	46.3
	1, 1, 1, 1	16, 8, 4, 2	-	36.0	34.4	40.2
	1, 1, 1, 1	Maximum	-	31.7	30.7	38.2
	8, 4, 2, 1	1, 1, 1, 1	78.2	43.0	38.6	41.5
● **DiNAT-T**	8, 4, 2, 1	8, 4, 2, 1	82.7	48.0	42.9	48.5
	8, 4, 2, 1	16, 8, 4, 2	-	45.6	41.3	47.1
	8, 4, 2, 1	Maximum	-	40.2	37.3	45.8
	16, 8, 4, 2	1, 1, 1, 1	-	29.0	26.7	26.2
	16, 8, 4, 2	8, 4, 2, 1	-	42.6	38.6	43.3
● **DiNAT-T**	16, 8, 4, 2	16, 8, 4, 2	-	48.3	43.4	48.5
	16, 8, 4, 2	Maximum	-	47.4	42.5	48.6

Table 17. Test time dilation change and its impact on performance. Dilation values larger than 8, 4, 2, 1 are inapplicable to ImageNet at 224^2.

4.6. Training settings

Herein we summarize our experimental settings. We followed Swin Transformer [31] and ConvNeXt [32] in choosing the following hyperparameters and training settings. All image classification models, with the exception of "Large" variants, are trained on ImageNet-1K [13] using the training procedure from `timm` [51] (Apache License v2). All such training jobs follow the aforementioned papers in training configurations, regularization techniques, augmentations (CutMix [53], Mixup [55], RandAugment [11], Random Erasing [56]) and other hyperparameters (300 epochs with a batch size of 1024, iteration-wise cosine learning rate schedule, 20 epoch warmup, 1e-3 learning rate, 5e-2 weight decay, extra 10 cooldown epochs.)

As previously mentioned, most models, unless explicitly stated, use a 7 × 7 kernel size neighborhood attention, and dilation values are selected with respect to input resolution.

Resolution	Dilation pattern			
	Level 1	Level 2	Level 3	Level 4
ImageNet classification.				
224^2	1, 8	1, 4	1, 2	1, 1
384^2	1, 13	1, 6	1, 3	1, 1
MS-COCO detection and instance segmentation.				
800^2	1, 28	1, 14	1, 3, 1, 5, 1, 7	1, 3
800^2	1, 28	1, 14	1, 3, 1, 5, 1, 7	1, 3
ADE20K semantic segmentation.				
512^2	1, 16	1, 8	1, 2, 1, 3, 1, 4	1, 2
640^2	1, 20	1, 10	1, 2, 1, 3, 1, 4, 1, 5	1, 2

Table 18. Dilation patterns used for different resolutions. Due to ImageNet's relatively small input resolution, level 4 layers cannot go beyond a dilation value of 1. Also note that at ImageNet's 224×224 resolution, level 4 inputs will be exactly 7×7, therefore NA will be equivalent to self attention. This is not true in downstream tasks where resolutions are noticeably higher and levels 2 and 3 have *gradually* increasing dilation values, which are repeated in deeper models. This corresponds to the highlighted rows in Tab. 14 labeled "Gradual". These configurations apply to all downstream experiments (excluding those in Sec. 4.5).

For example, ImageNet classification at 224×224 is downsampled to a quarter of the original size initially, therefore Level 1 layers take feature maps of resolution 56×56 as input. With a kernel size of 7×7, the maximum possible dilation value is $\lfloor 56/7 \rfloor = 8$. Level 2 will take feature maps of resolution 28×28 as input, leading to a maximum possible dilation value of 4. Because of this, we change dilation values depending on the task and resolution. We present the final dilation values we used in classification, detection, and segmentation in Tab. 18. Note that we only change dilation values for DiNA layers, since we found that fine-tuning NA layers to DiNA layers may result in a slight decrease in initial performance (see Sec. 4.5, Tab. 17).

We conducted our object detection and instance segmentation experiments with Mask R-CNN [20] and Cascade Mask R-CNN [4] on MS-COCO [29], again following Swin [31] and ConvNeXt [32] and their training settings in `mmdetection` [6] (Apache License v2), and trained with the same accelerated 3× LR schedule. The same goes for our semantic segmentation experiments with UPerNet [52] on ADE20K [57] which follow said papers training configurations for `mmsegmentation` [9] (Apache License v2).

5. Future directions

Neighborhood attention provides a spectrum of possible attention patterns, ranging from linear projection to self attention. Applying neighborhood attention to any problem where self attention has already been found useful can be thought of as a primary research direction. In theory, any self attention operation can be replaced with neighborhood attention, without requiring re-training or fine-tuning. While seemingly simple, because of the two key hyperparameters, window size and dilation, there's a relatively large set of combinations that are possible. This, however, is not the only application of neighborhood attention; Another avenue is applying neighborhood attention with less aggressive downsampling, in order to enhance performance, while still maintaining a tractable computational cost.

Another exceedingly important area of work is in implementation. Despite the performance improvements gained from neighborhood attention's implicit GEMM kernels, these kernels, and more generally $\mathcal{NATTEN}$ require continued development as to catch up with other computational software, as well as new hardware architectures. Moreover, the rise of more complicated kernel fusion, as well as advanced integer quantization methods, can also present challenges of their own when applied to neighborhood attention kernels. In addition, our existing implicit GEMM formulation is not perfect; it has limited scalability to mixed-precision and lower-precision calls, due to the fact that the PN epilogue is required to store attention weights one by one, making vectorized stores impossible. For a similar reason, reading attention weights in NN and IN kernels is also limited to one by one reads, which not only are slower compared to vectorized reads, but also restrict the ability to use caching and asynchronous copies. In spite of these issues, the current set of kernels have significantly improved runtime for sliding window attention patterns, raising them to competitive levels with alternatives.

6. Conclusion

In this paper, we introduced neighborhood attention, which restricts self attention to nearest neighboring tokens, resulting in an explicit sliding window behavior. We discussed prior sliding window attention patterns, and discussed a key issue in their formulation (handling of corner tokens), and their implementation challenges. Neighborhood attention's definition resolves the aforementioned issue in formulation, and results in a spectrum of possible attention patterns between no attention (linear projection) and self attention. We then introduced different levels of implementations, ranging from high level implementations using Python interfaces, and naive CUDA kernels, down to a GEMM-based kernel following the implicit GEMM approach used in implementing convolutions. Through our implementations, which we package as a Python extension, $\mathcal{NATTEN}$, models based on neighborhood attention can scale up to 200 million parameters, and achieve competitive performance compared to former and current state-of-the-art architectures. We finally discuss possible future applications of neighborhood attention, and challenges ahead.

References

[1] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020. 2, 3, 4

[2] Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. Token merging: Your ViT but faster. In *International Conference on Learning Representations (ICLR)*, 2023. 3, 5

[3] Daniel Bolya and Judy Hoffman. Token merging for fast stable diffusion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2023. 3, 5

[4] Zhaowei Cai and Nuno Vasconcelos. Cascade r-cnn: Delving into high quality object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 15, 18

[5] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European Conference on Computer Vision (ECCV)*, 2020. 1

[6] Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, et al. Mmdetection: Open mmlab detection toolbox and benchmark. *arXiv:1906.07155*, 2019. 18

[7] Bowen Cheng, Ishan Misra, Alexander G Schwing, Alexander Kirillov, and Rohit Girdhar. Masked-attention mask transformer for universal image segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 15

[8] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv:1904.10509*, 2019. 1, 3, 4

[9] MMSegmentation Contributors. MMSegmentation: Openmmlab semantic segmentation toolbox and benchmark. https://github.com/open-mmlab/mmsegmentation, 2020. 18

[10] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 16

[11] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2020. 17

[12] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. 2, 3, 8

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009. 13, 17

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019. 1

[15] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2020. 1, 2, 3, 4, 5, 13

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org. 6

[17] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, et al. Conformer: Convolution-augmented transformer for speech recognition. *Interspeech*, 2020. 1

[18] Ali Hassani, Steven Walton, Nikhil Shah, Abulikemu Abuduweili, Jiachen Li, and Humphrey Shi. Escaping the big data paradigm with compact transformers. *arXiv:2104.05704*, 2021. 2, 3, 5

[19] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 1

[20] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2017. 18

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 1

[22] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 1

[23] Lang Huang, Yuhui Yuan, Jianyuan Guo, Chao Zhang, Xilin Chen, and Jingdong Wang. Interlaced sparse self-attention for semantic segmentation. *arXiv preprint arXiv:1907.12273*, 2019. 3, 4

[24] Zilong Huang, Xinggang Wang, Yunchao Wei, Lichao Huang, Humphrey Shi, Wenyu Liu, and Thomas S. Huang. Ccnet: Criss-cross attention for semantic segmentation. In *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2020. 1, 4

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012. 1

[26] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1989. 1

[27] Yanghao Li, Hanzi Mao, Ross Girshick, and Kaiming He. Exploring plain vision transformer backbones for object detection. In *European Conference on Computer Vision (ECCV)*, 2022. 1

[28] Youwei Liang, Chongjian Ge, Zhan Tong, Yibing Song, Jue Wang, and Pengtao Xie. Not all patches are what you need: Expediting vision transformers via token reorganizations. In *International Conference on Learning Representations (ICLR)*, 2022. 3, 5

[29] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision (ECCV)*, 2014. 16, 18

[30] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, et al. Swin transformer v2: Scaling up capacity and resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 1

[31] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021. 1, 2, 3, 4, 7, 12, 13, 14, 17, 18

[32] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 1, 14, 15, 17, 18

[33] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv:1805.02867*, 2018. 3

[34] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning (ICML)*, 2018. 1

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. 8

[36] William Peebles and Saining Xie. Scalable diffusion models with transformers. *arXiv:2212.09748*, 2022. 1

[37] Markus N Rabe and Charles Staats. Self-attention does not need $O(n^2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021. 2, 3

[38] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training, 2018. 1

[39] Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. Stand-alone self-attention in vision models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. 1, 2, 3, 5, 6

[40] Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. Dynamicvit: Efficient vision transformers with dynamic token sparsification. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. 3, 5

[41] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 1

[42] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics (TACL)*, 9, 2021. 4

[43] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, et al. Cutlass, 2023. 5

[44] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning (ICML)*, 2020. 3

[45] Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going deeper with image transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021. 3

[46] Zhengzhong Tu, Hossein Talebi, Han Zhang, Feng Yang, Peyman Milanfar, Alan Bovik, and Yinxiao Li. Maxvit: Multi-axis vision transformer. In *European Conference on Computer Vision (ECCV)*, 2022. 3, 4, 13, 14

[47] Ashish Vaswani, Prajit Ramachandran, Aravind Srinivas, Niki Parmar, Blake Hechtman, and Jonathon Shlens. Scaling local self-attention for parameter efficient visual backbones. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. 1, 2, 3

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. 1, 3

[49] Huiyu Wang, Yukun Zhu, Bradley Green, Hartwig Adam, Alan Yuille, and Liang-Chieh Chen. Axial-deeplab: Stand-alone axial-attention for panoptic segmentation. In *European Conference on Computer Vision (ECCV)*, 2020. 1

[50] Wenhai Wang, Enze Xie, Xiang Li, Deng-Ping Fan, Kaitao Song, Ding Liang, Tong Lu, Ping Luo, and Ling Shao. Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021. 1

[51] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019. 13, 17

[52] Tete Xiao, Yingcheng Liu, Bolei Zhou, Yuning Jiang, and Jian Sun. Unified perceptual parsing for scene understanding. In *European Conference on Computer Vision (ECCV)*, 2018. 15, 16, 18

[53] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. 17

[54] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 2

[55] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *International Conference on Learning Representations (ICLR)*, 2018. 17

[56] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2020. 17

[57] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ade20k dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. 16, 18