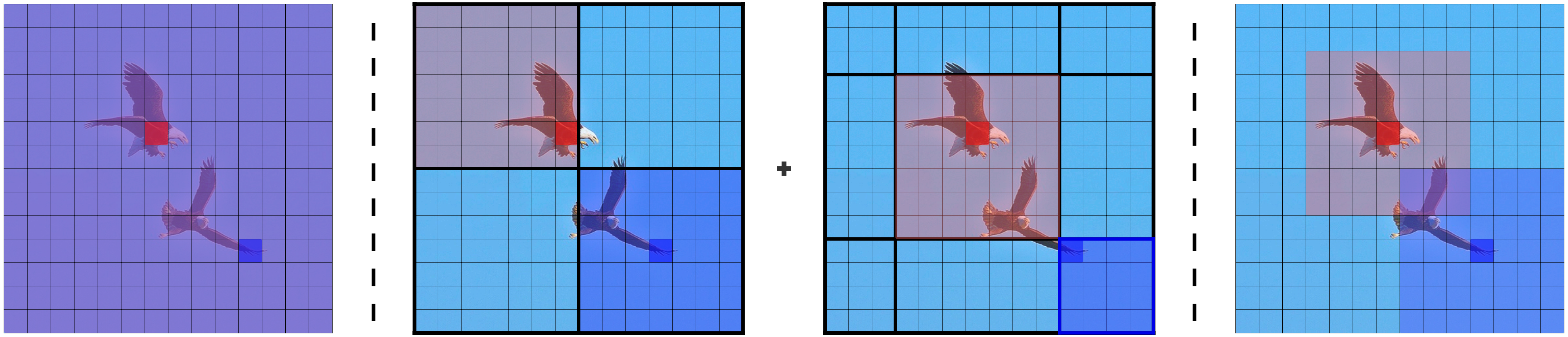
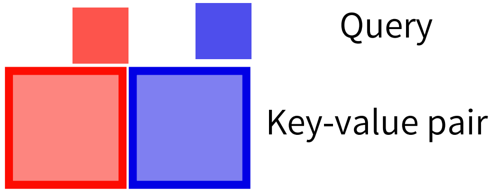

Self Attention (ViT) | Window Self Attention (Swin) + Shifted Window Self Attention (Swin) | Neighborhood Attention (NAT)

Neighborhood Attention Transformer

Ali Hassani, Steven Walton, Jiachen Li, Shen Li, and Humphrey Shi.

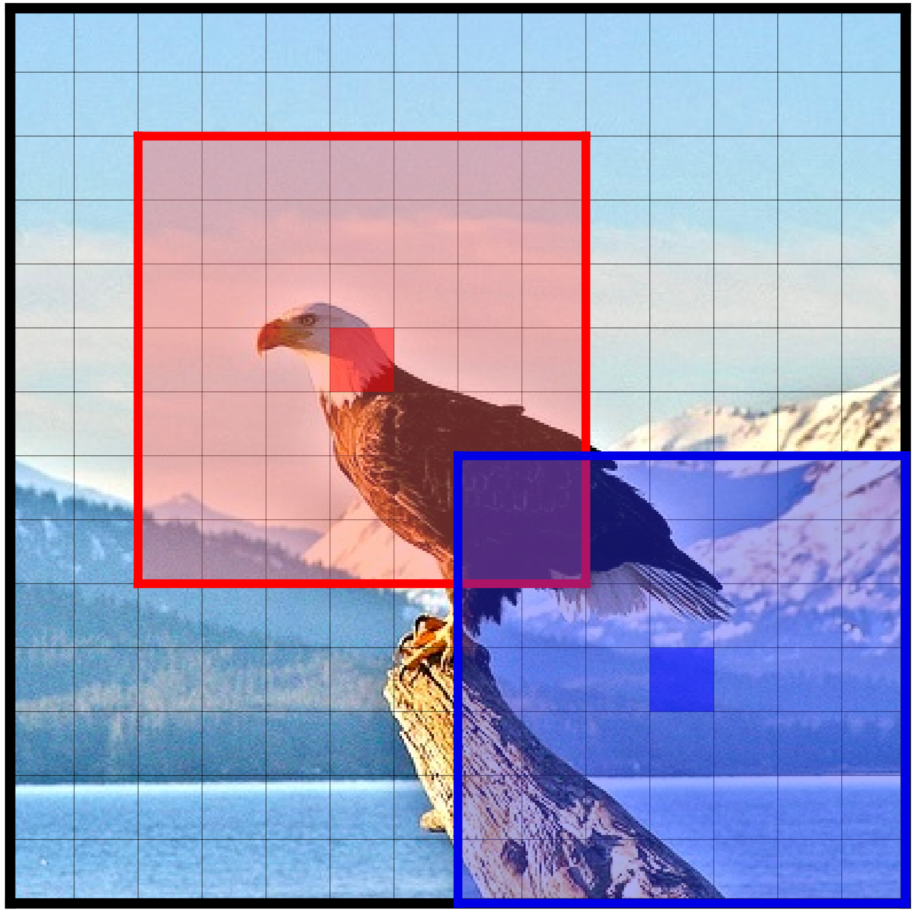
(TUE-PM-197)



Neighborhood Attention

Pixels attend to their nearest-neighboring pixels.

Linear complexity with respect to feature map size.



Neighborhood Attention

NATTEN

NEIGHBORHOOD ATTENTION EXTENSION

BRINGING ATTENTION TO A NEIGHBORHOOD NEAR YOU!

NATTEN is an extension to PyTorch, which provides the first fast sliding window attention with efficient CPU and CUDA kernels. It provides Neighborhood Attention (local attention) and Dilated Neighborhood Attention (sparse global attention, a.k.a. dilated local attention) as PyTorch modules for both 1D and 2D data.

GitHub / PyPI

Neighborhood Attention Transformers

Install with pip

Latest release: `0.14.6`

Please select your preferred PyTorch version with the correct CUDA build, or CPU build if you're not using CUDA:

PyTorch: 2.0 1.13 1.12.1 1.12 1.11 1.10.1 1.10 1.9 1.8

CUDA 11.8

CUDA 11.7

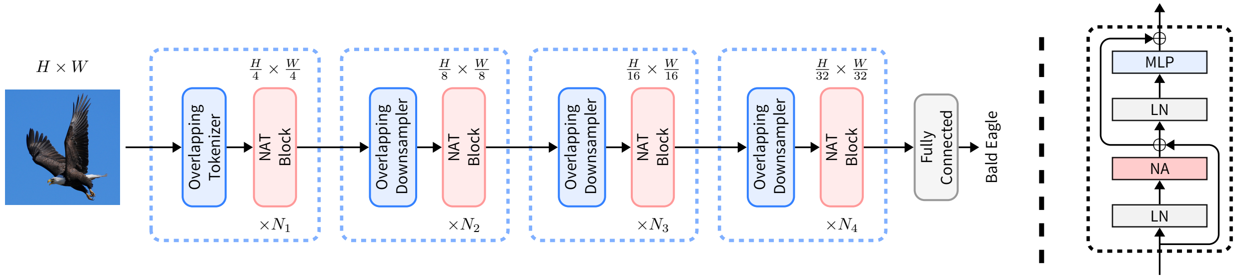
CPU

Run this command:

Copy

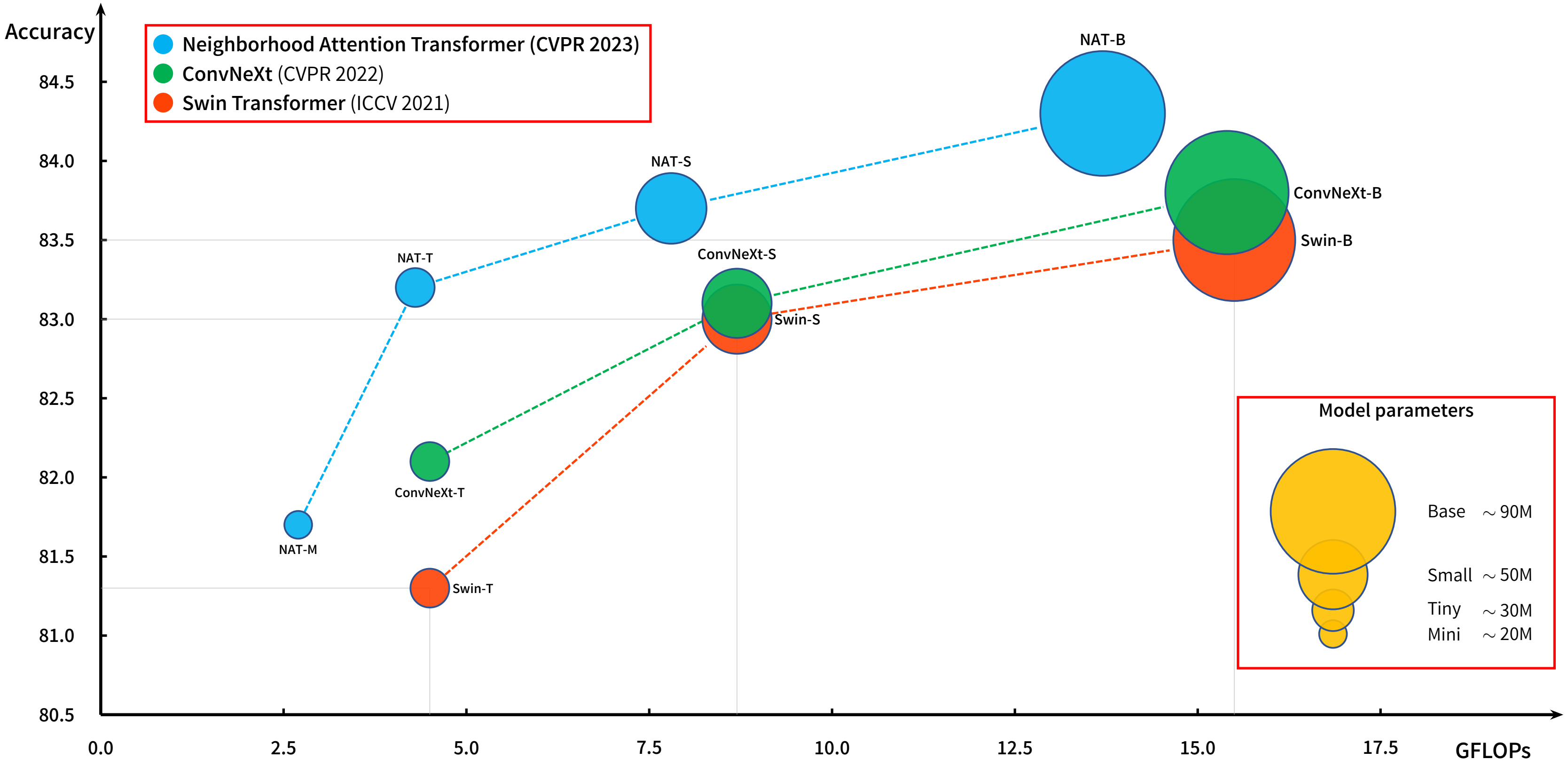
```
pip3 install natten -f https://shi-labs.com/natten/wheels/cu118/torch2.0.0/index.html
```

Neighborhood Attention Transformer



Neighborhood Attention Transformer Architecture

NAT Block



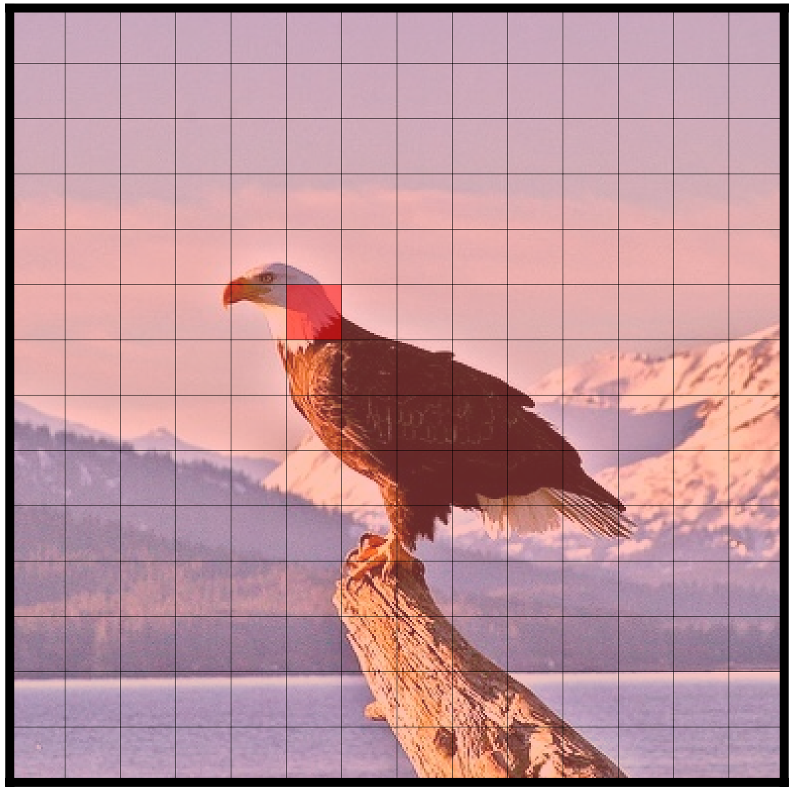
Background

Query

Key-value pair

Self attention

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



Self Attention

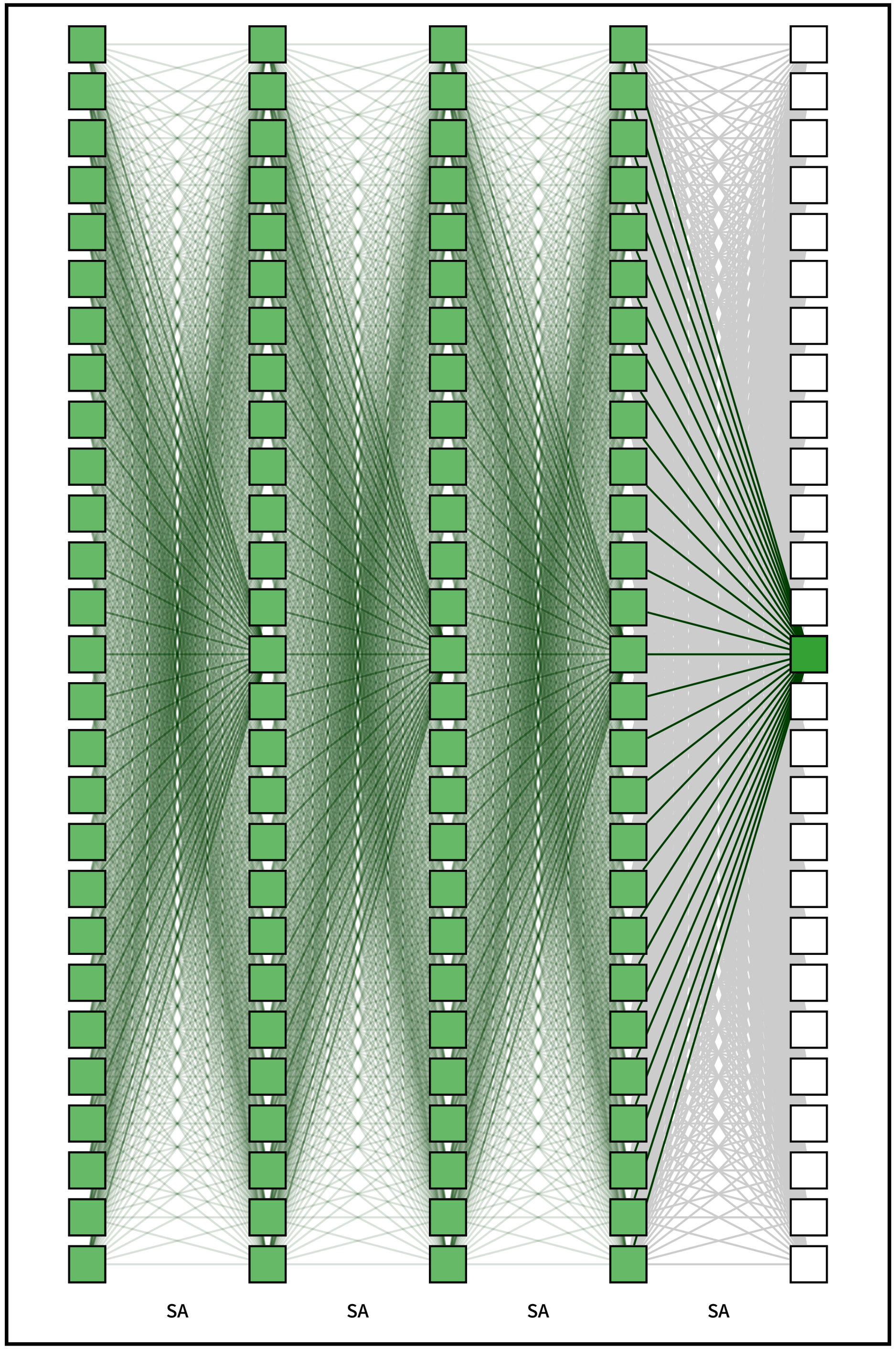
Global receptive field

$$r = n$$

where n is the number of inputs (tokens/pixels).

In this example, $n = 27$, therefore:

$$r = 27$$



Translationally equivariant

Applying a translation to the input to self attention is equal to applying the same translation to its output given the same input.

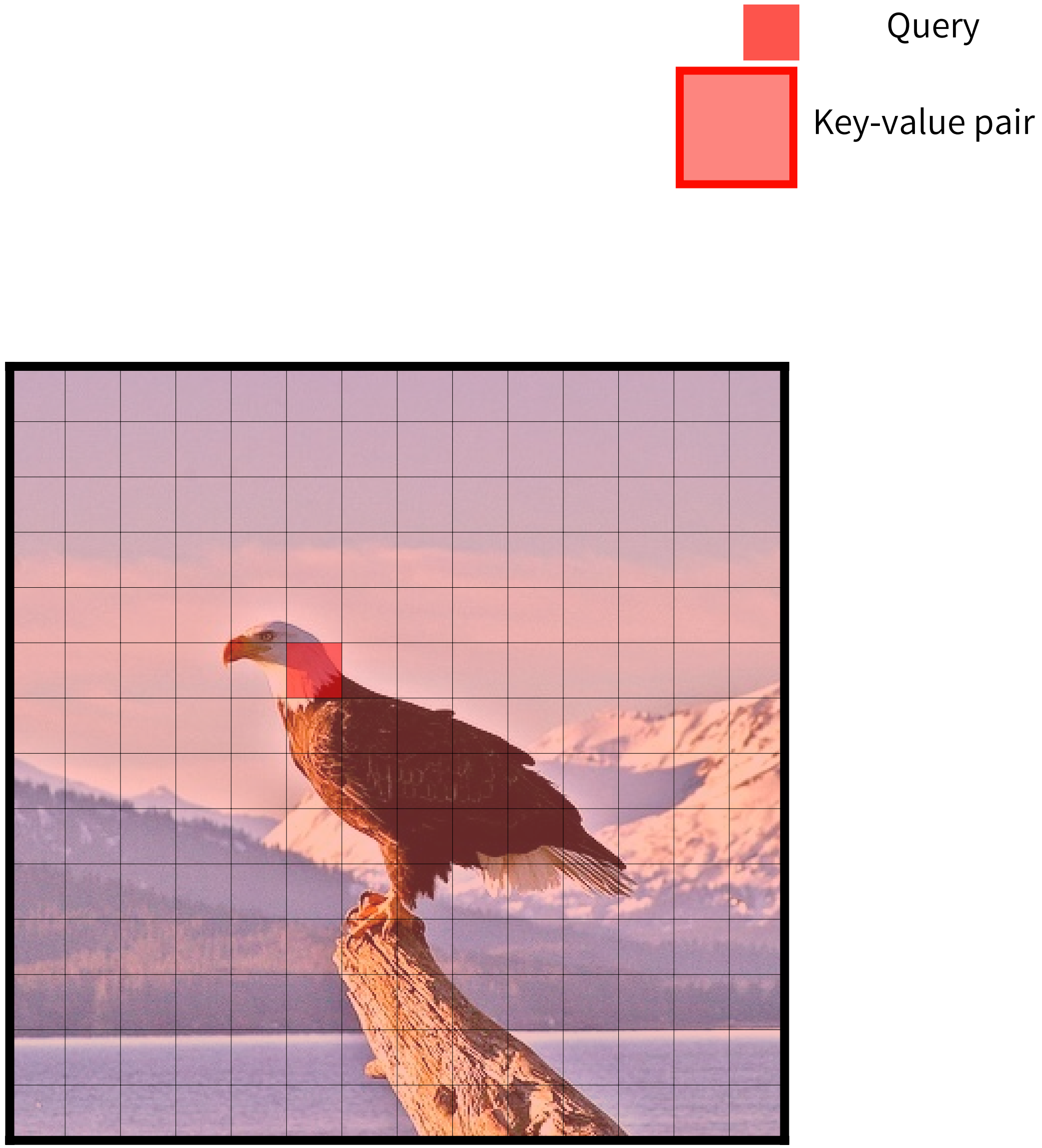
Self attention is translationally equivariant, because it is invariant to permutation.

x $\mathcal{T}(x)$ $SA\,(x)$ $\mathcal{T}(SA\,(x))$ $SA\,(\mathcal{T}(x))$

$SA^2 = SA \circ SA$

Self attention

Every pixel attending to every pixel is **quadratic** w.r.t. resolution.



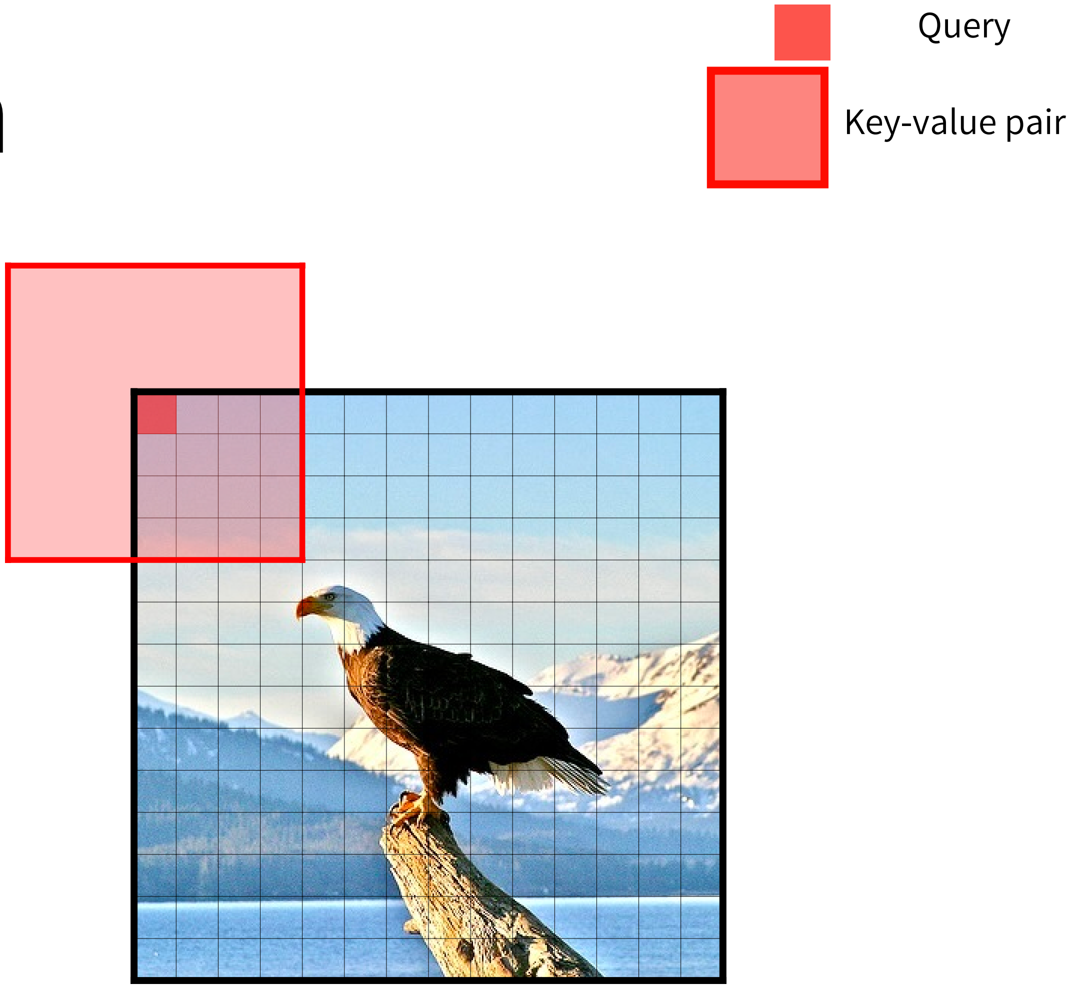
Self Attention

Sliding window attention

(a.k.a. SASA)

Every query attends to the pixels around it, with its corresponding value itself centered.

(Similar pattern to zero-padded convolutions.)



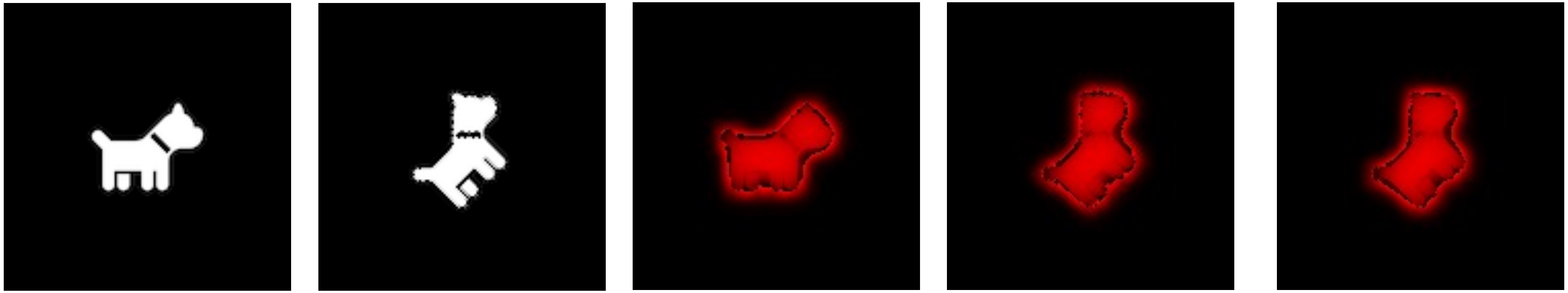
Sliding Window Attention

Ramachandran et al. "Stand-alone self-attention in vision models." In NeurIPS 2019.
Beltagy et al. "Longformer: The long-document transformer." 2020.
Zaheer et al. "Big bird: Transformers for longer sequences." In NeurIPS 2020.

Translationally equivariant

Similar to convolutions, sliding window attention is equivariant to translations.

x | $\mathcal{T}(x)$ | $SWA^2(x)$ | $\mathcal{T}(SWA^2(x))$ | $SWA^2(\mathcal{T}(x))$



$SWA^2 = SWA \circ SWA$

Receptive field

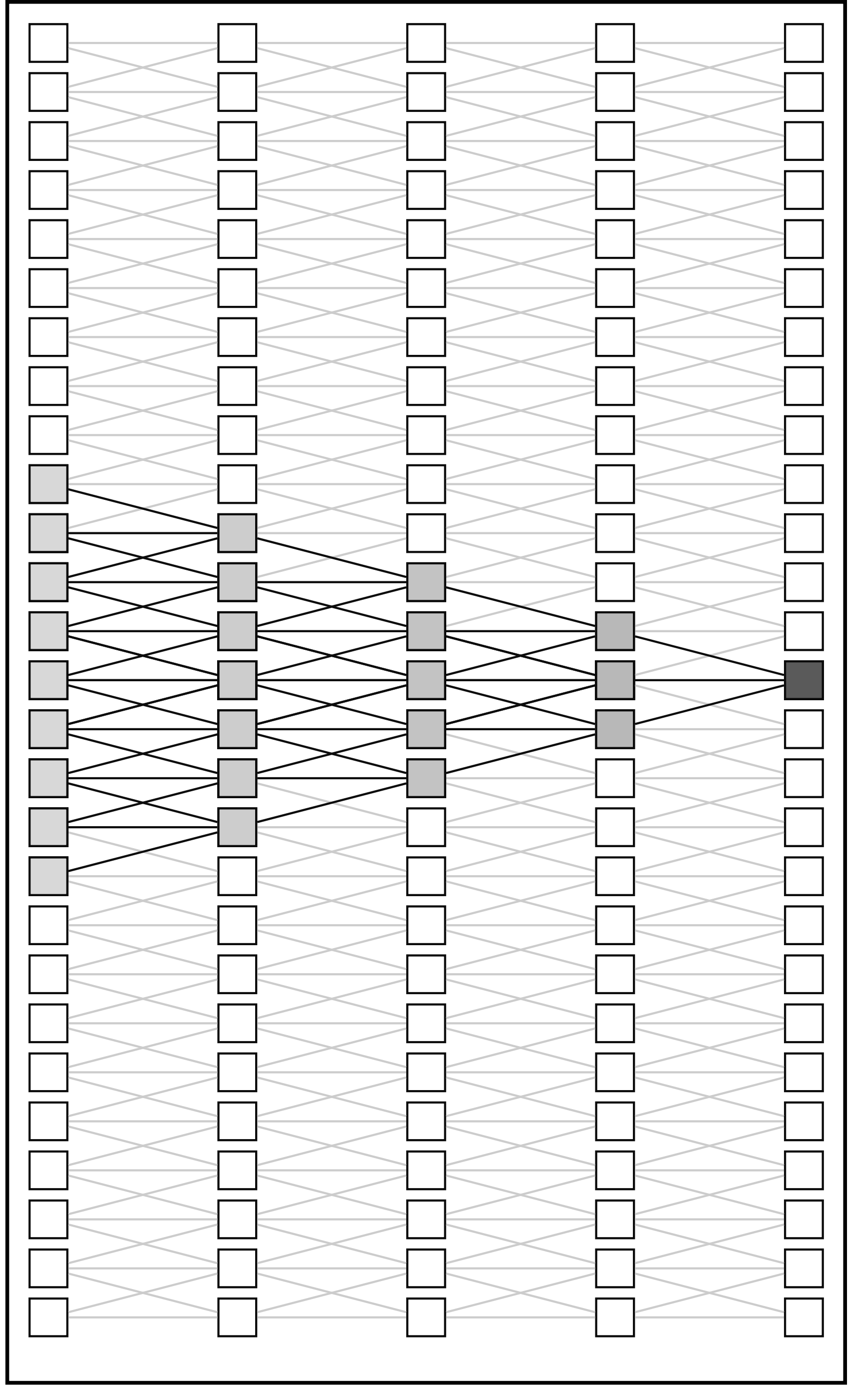
Linearly growing receptive field (identical to convolutions)

$$r = \ell(k-1) + 1$$

where k is window/kernel size, and ℓ is the number of layers.

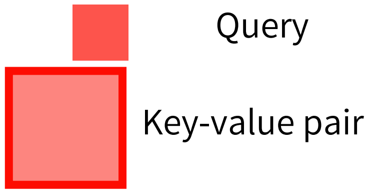
In this example, $k = 3, \ell = 4$, therefore:

$$r = 9$$

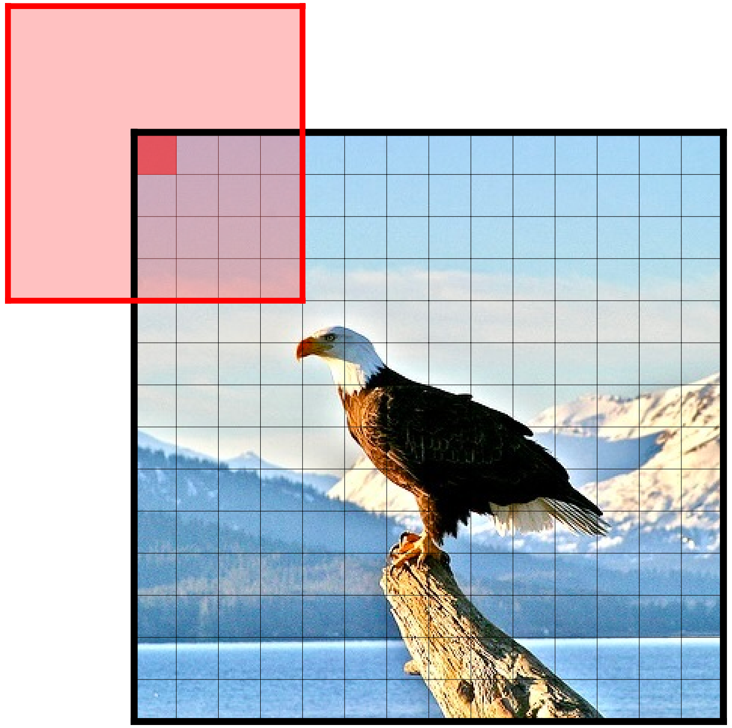


Sliding window attention

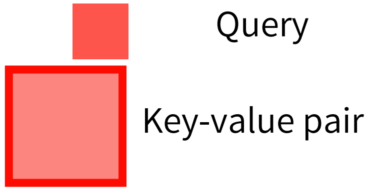
Challenge 1: Implementation



- Self attention breaks down to two matrix multiplications, with a softmax in between.
- Convolutions can be modeled as matrix multiplications (implicit GEMM.)
- Sliding window attention cannot be modeled in the same way; and even if it were, it's not practical through a Python interface.



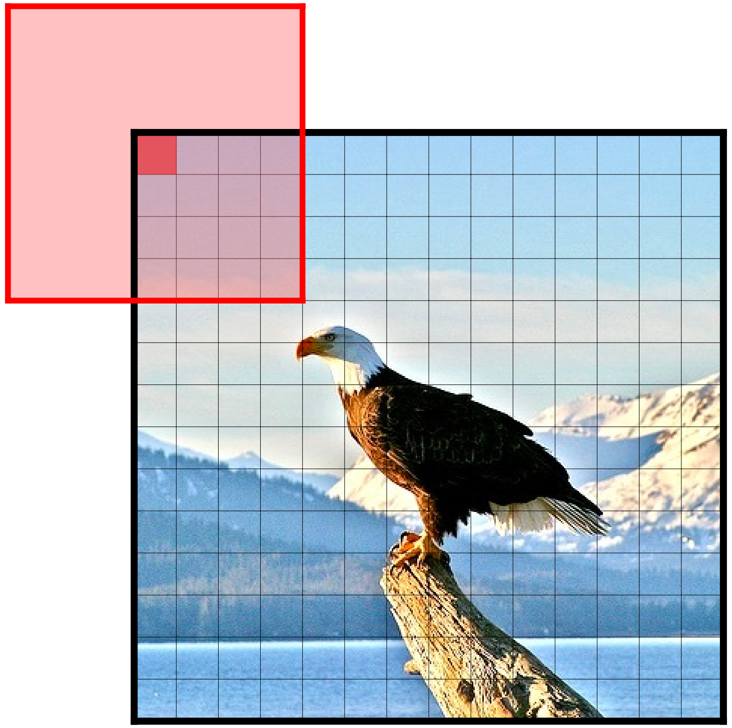
Sliding Window Attention



Challenge 2: Corner cases

- And the zero padding in the corners can quickly become a problem as window size grows…

- Larger window => larger padding => smaller average receptive field.

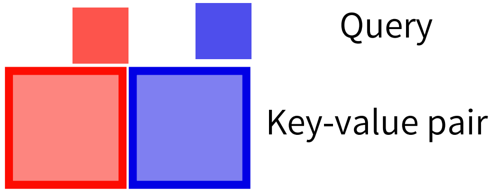


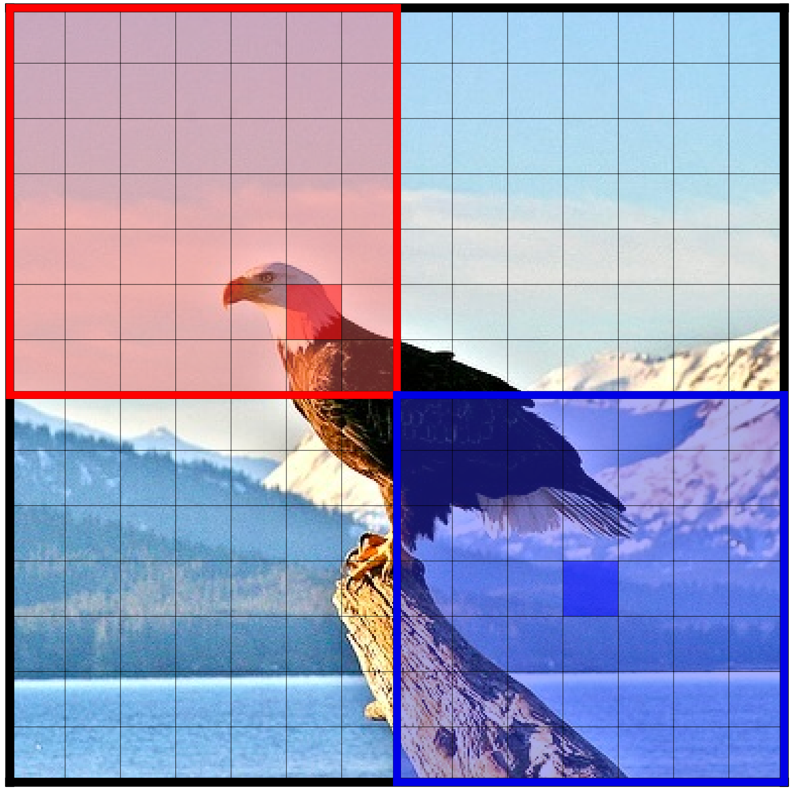
Sliding Window Attention

Window self attention

(a.k.a blocked / partitioned attention)

1. Partition inputs of fixed size
2. Apply self attention to each
3. Merge





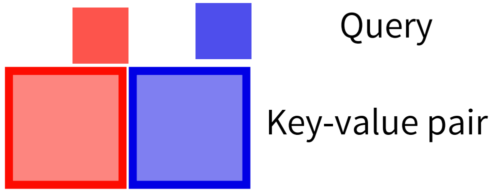
Window Self Attention

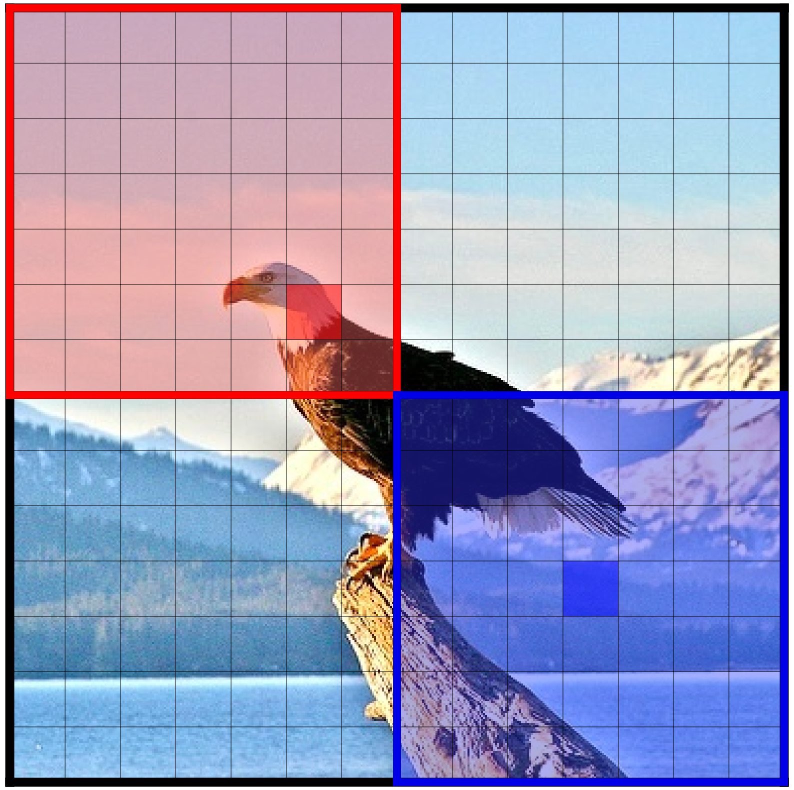
Liu et al. “Swin Transformer: Hierarchical vision transformer using shifted windows.” In ICCV 2021.
Vaswani et al. "Scaling local self-attention for parameter efficient visual backbones." In CVPR 2021.

Window self attention

(a.k.a blocked / partitioned attention)

- Linear time complexity,
- Trivial to implement without modifying the attention operator,
- Constant # of interactions per token.





Window Self Attention

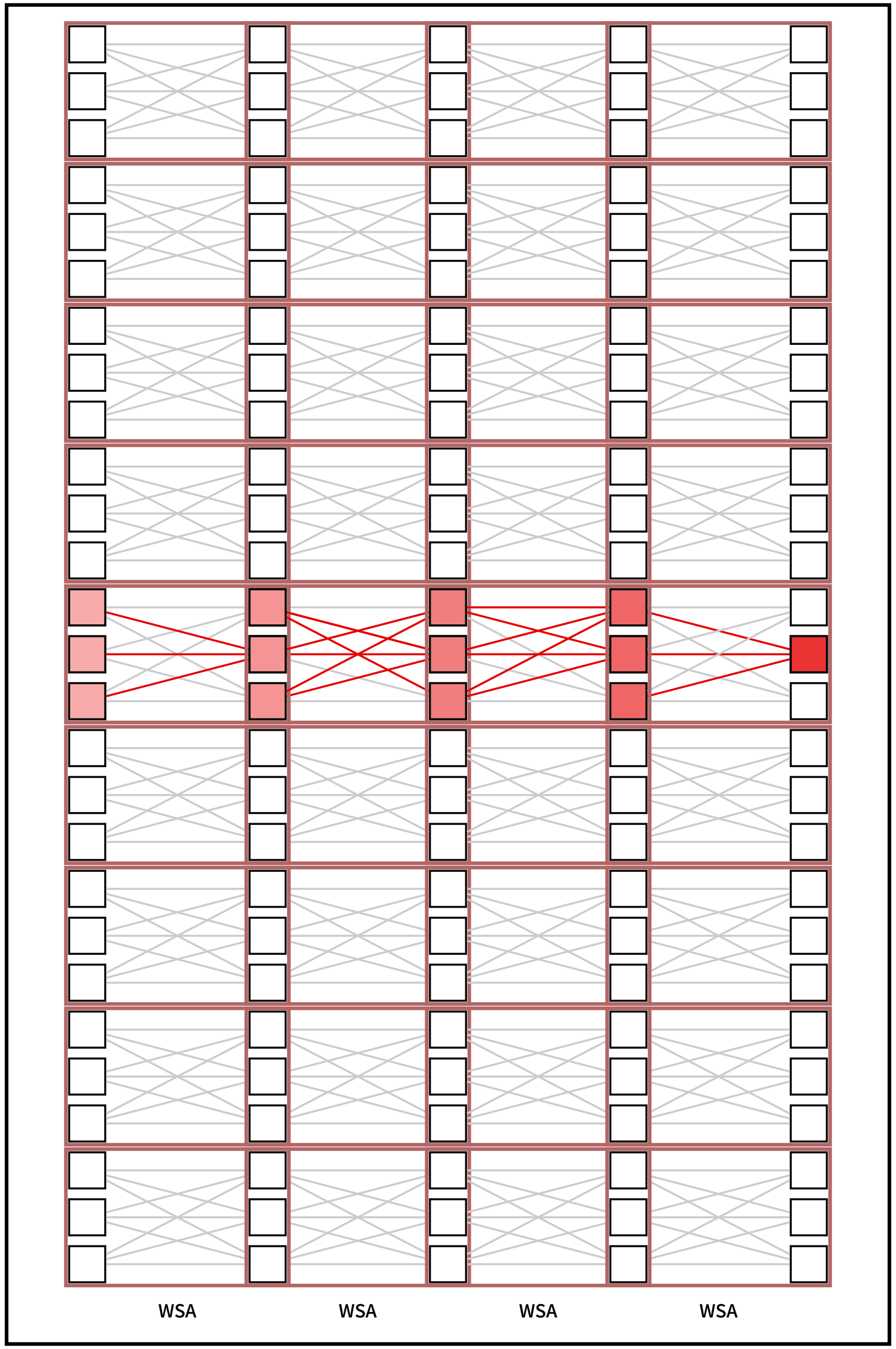
Window self attention

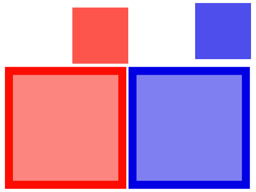
Constant receptive field...

$$r = k$$

where k is window/kernel size.

In this example, $k = 3$, therefore:
$$r = 3$$





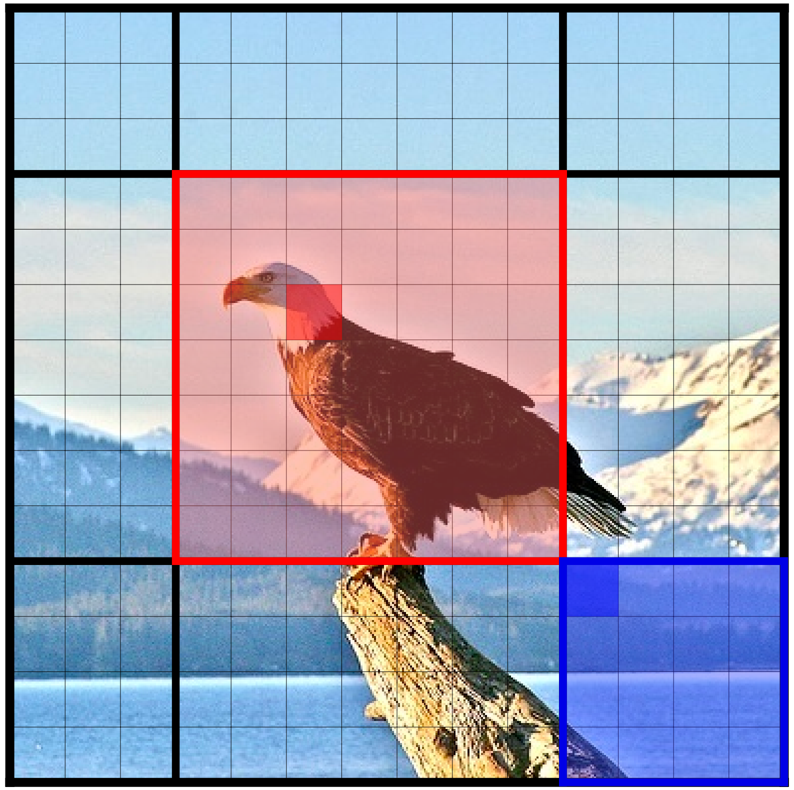
Query

Key-value pair

Cyclic shift

Liu et al. “Swin Transformer: Hierarchical vision transformer using shifted windows.” In ICCV 2021.

1. Shift pixels to get “shifted windows” ,
2. Apply masked window self attention,
3. Shift back.



Shifted Window Self Attention

WSA + SWSA

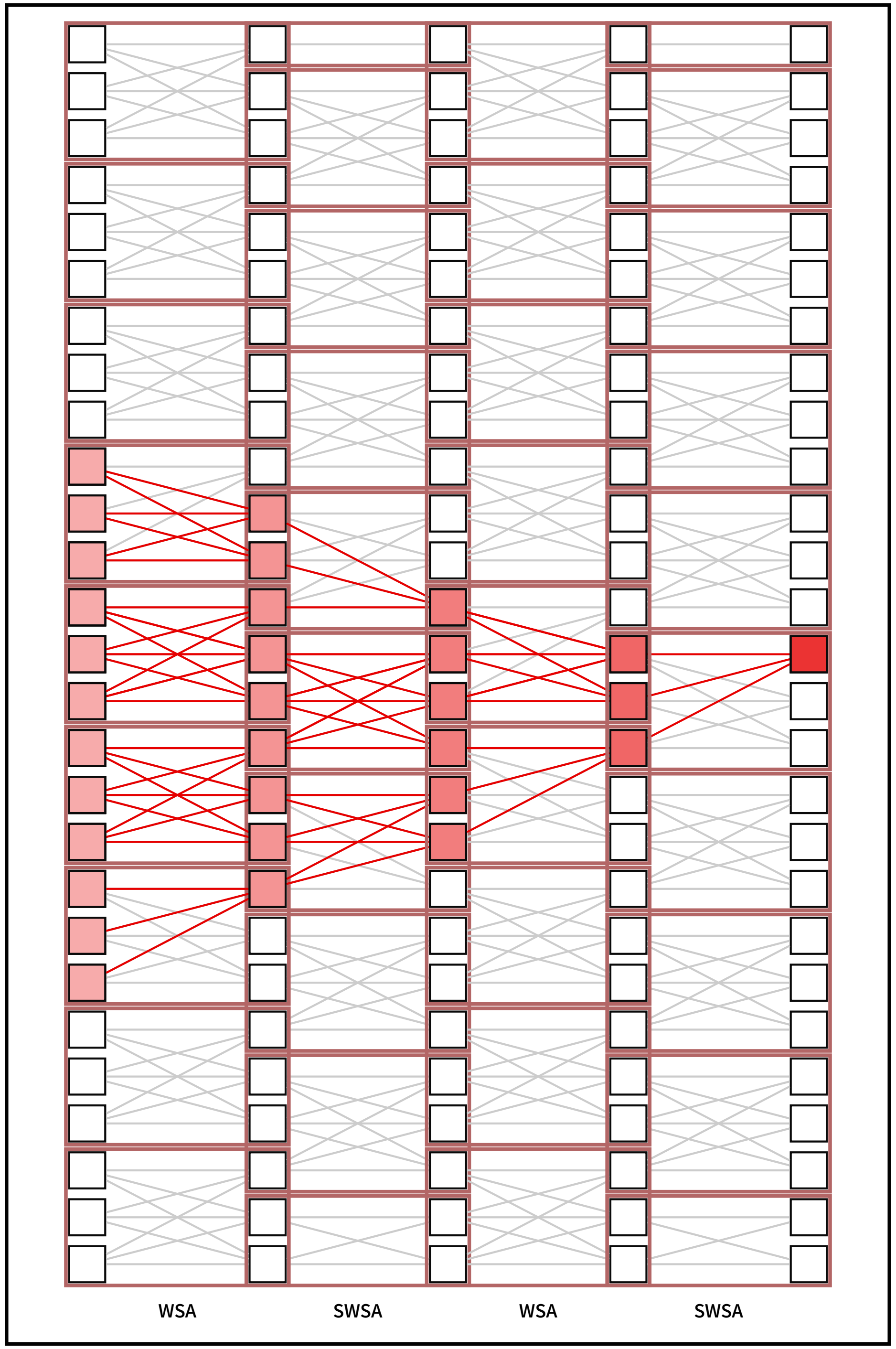
- Linearly growing receptive field

- $r = \ell k$

where k is window/kernel size, and ℓ is the number of layers.

In this example, $k = 3, \ell = 4$, therefore:

$$r = 12$$



NOT translationally equivariant

Window self attention is not translationally equivariant, primarily because the lack of overlaps, and even with the pixel shifts, the overlap is still half of the window size.

It is also correct to say that WSA+SWSA relaxes translational equivariance.

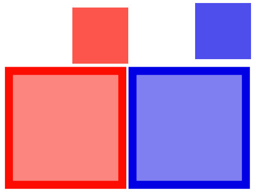
x | $\mathcal{T}(x)$ | $Sw(x)$ | $\mathcal{T}(Sw\ (x))$ | $Sw\ (\mathcal{T}(x))$

$Sw = WSA\ \circ SWSA$

Drawbacks

- Input size is required to be divisible by window size,
- Window size can only be as large as half the input size,
- Lack of symmetry and translational equivariance.

Neighborhood Attention



Neighborhood Attention

Given the query, key, and value projections (Q, K, V), and neighborhood size k , we define attention weights as:

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

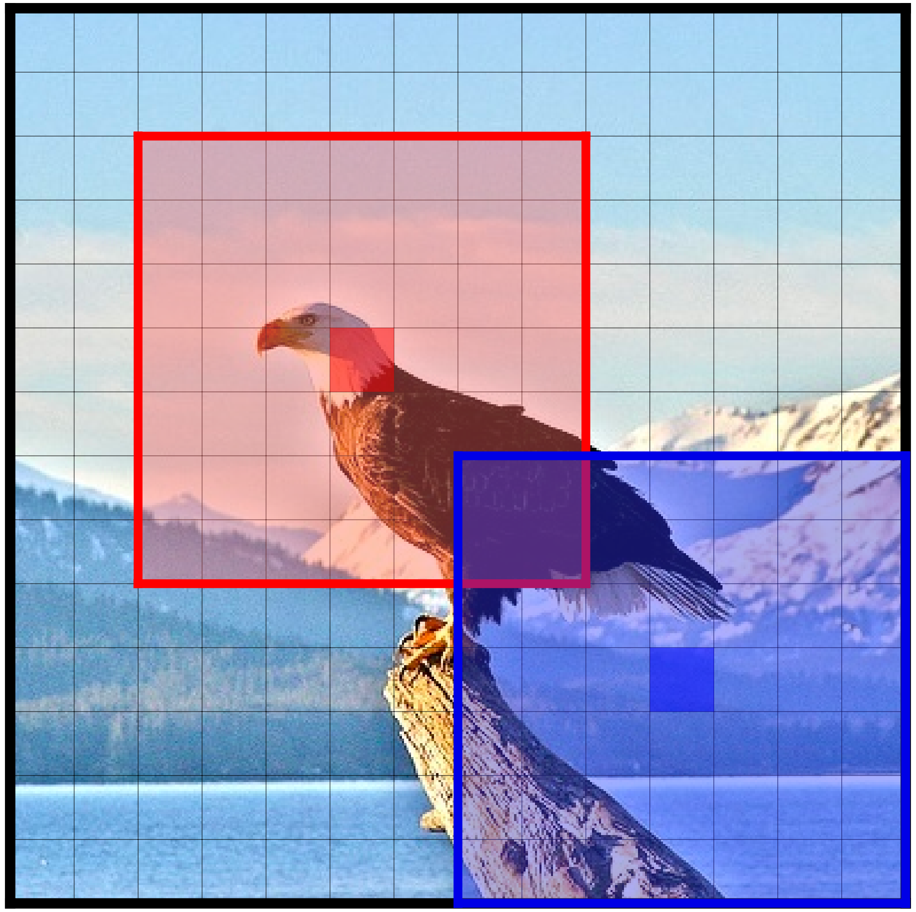
where $\rho_j(i)$ denotes the j-th nearest neighbor of token i.

We then define values as:

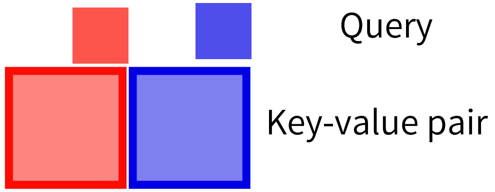
$$\mathbf{V}_i^k = \begin{bmatrix} V^T_{\rho_1(i)} & V^T_{\rho_2(i)} & \dots & V^T_{\rho_k(i)} \end{bmatrix}^T$$

Neighborhood Attention with neighborhood size k for token i is then defined as:

$$\mathrm{NA}_k(i) = softmax\left(\frac{\mathbf{A}_i^k}{\sqrt{d}}\right)\mathbf{V}_i^k.$$



Neighborhood Attention



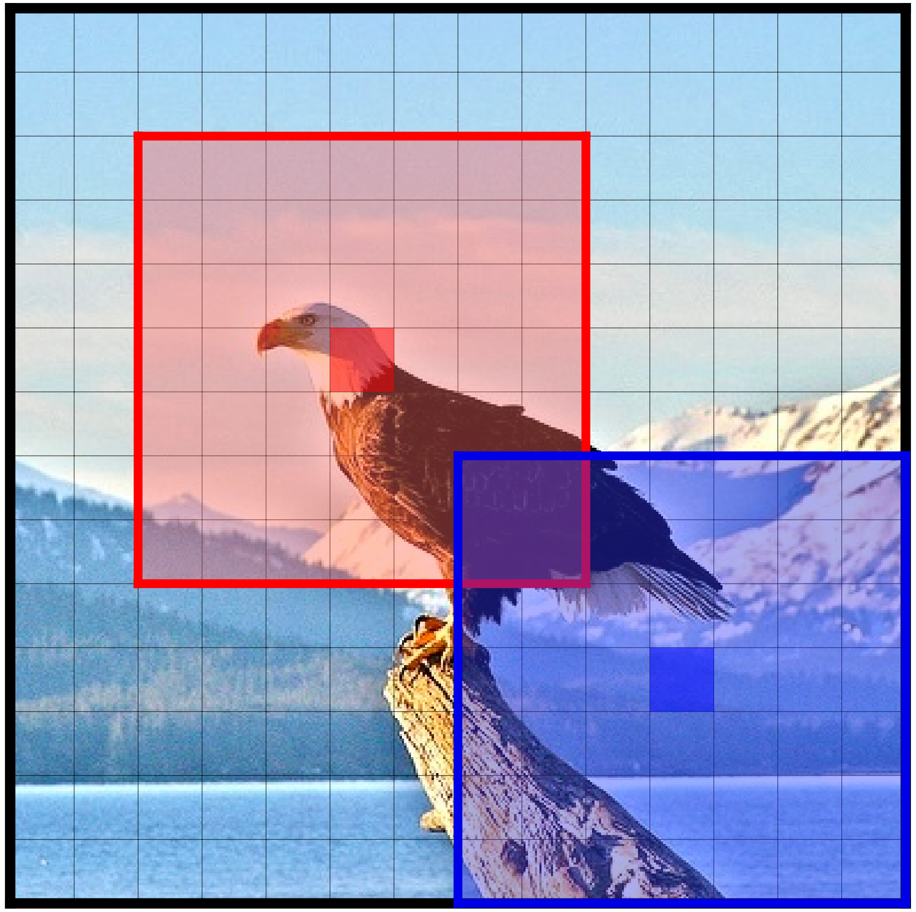
Neighborhood Attention

Pixels attend to their nearest-neighboring pixels.

Advantages:

✓Fixed # of interactions for every pixel

✓Linear complexity



Neighborhood Attention

Receptive field

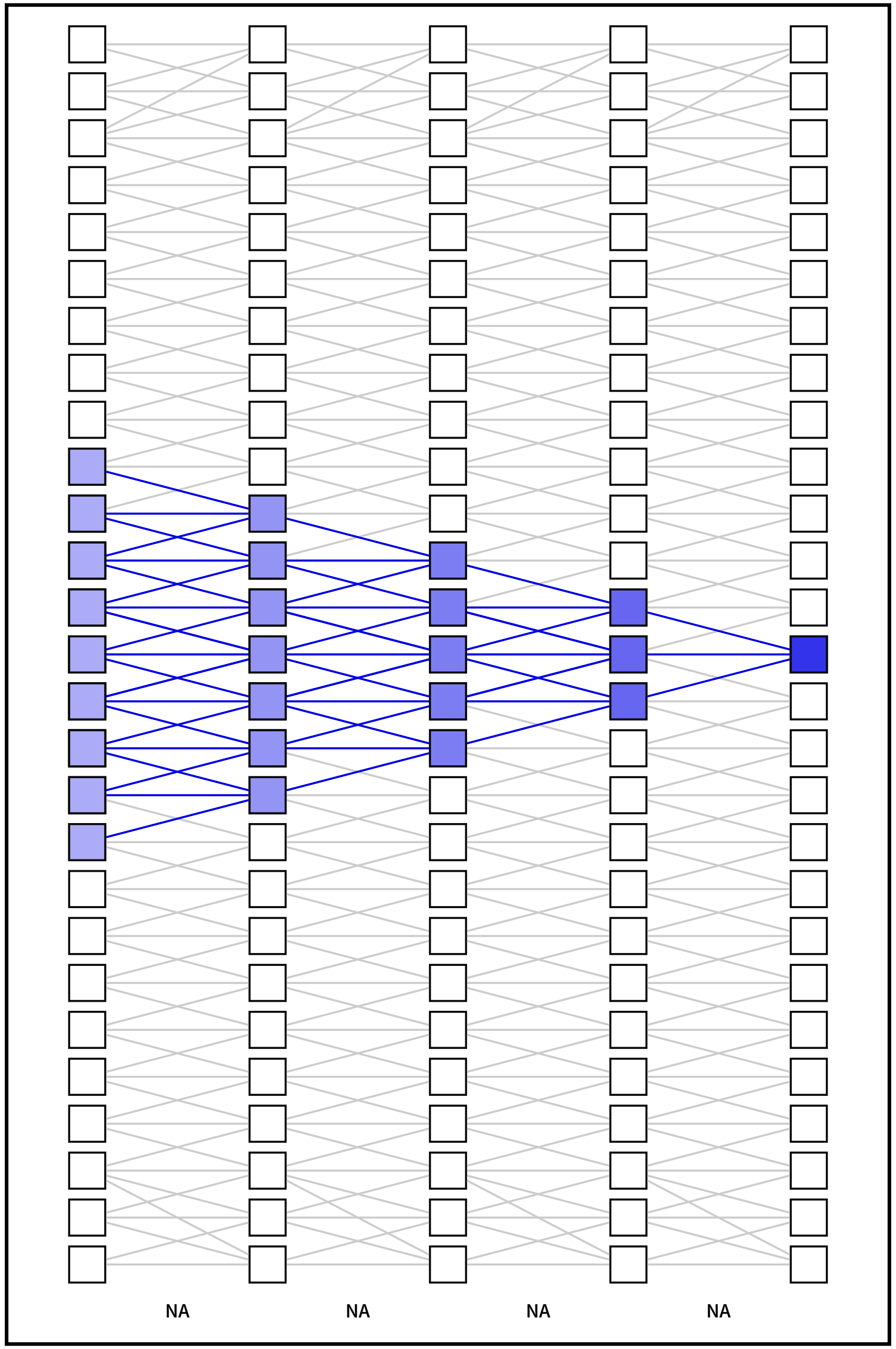
Linearly growing receptive field

$$r = \ell(k-1) + 1$$

where k is window/kernel size, and ℓ is the number of layers.

In this example, $k = 3, \ell = 4$, therefore:

$$r = 9$$



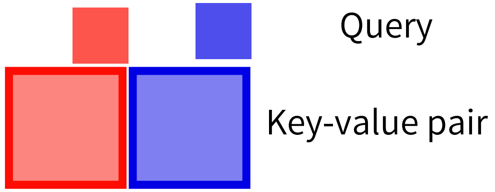
Translationally equivariant

Similar to SA and sliding window attention, NA is also translationally equivariant.

x | $\mathcal{T}(x)$ | $NA^2(x)$ | $\mathcal{T}(NA^2\,(x))$ | $NA^2\,(\mathcal{T}(x))$

$NA^2 = NA \circ NA$

Neighborhood Attention



Pixels attend to their nearest-neighboring pixels.

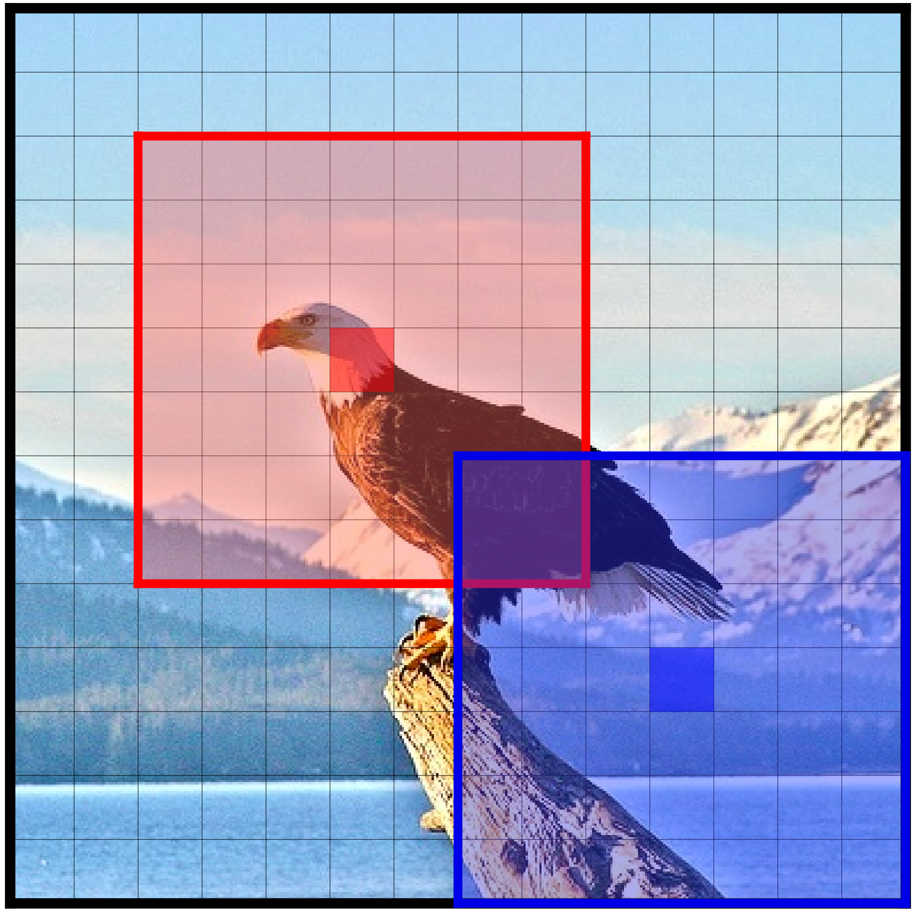
Advantages:

✓Fixed # of interactions for every pixel

✓Linear complexity

✓RF grows without cyclic shift

✓Translationally equivariant

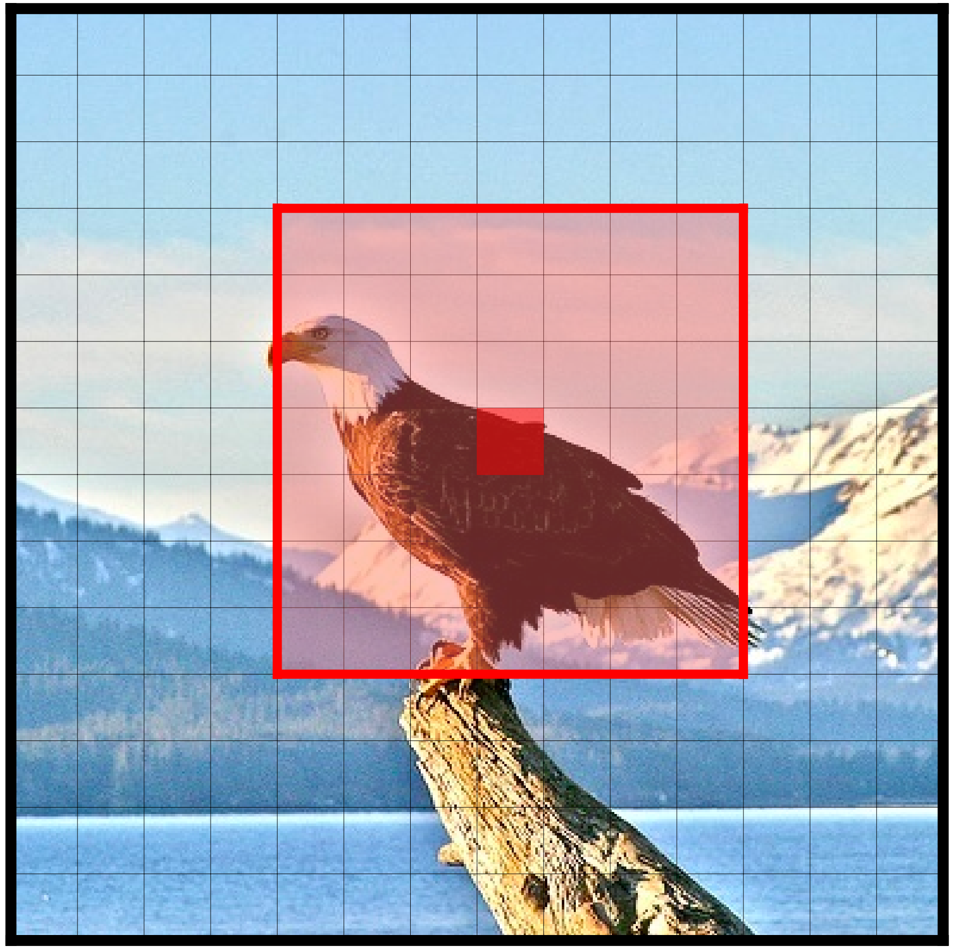


Neighborhood Attention

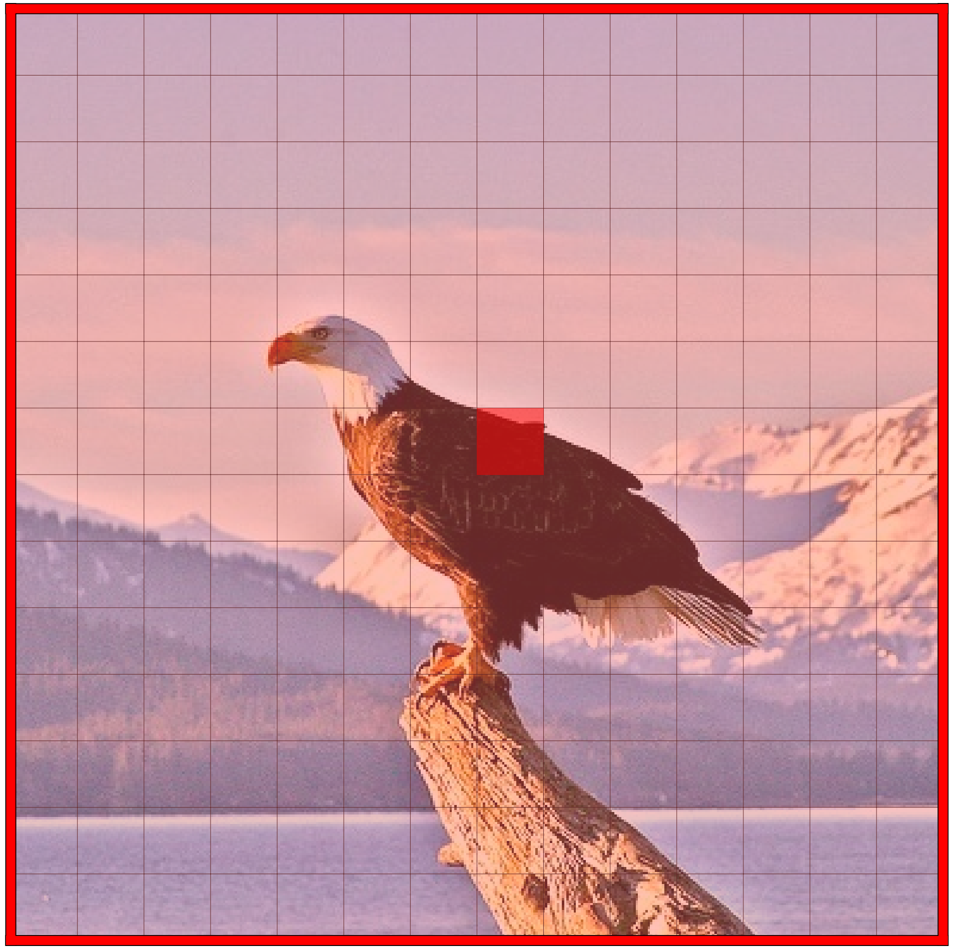
Neighborhood Attention approaches Self Attention

Query

Key-value pair

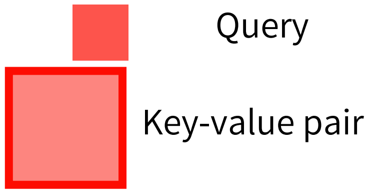


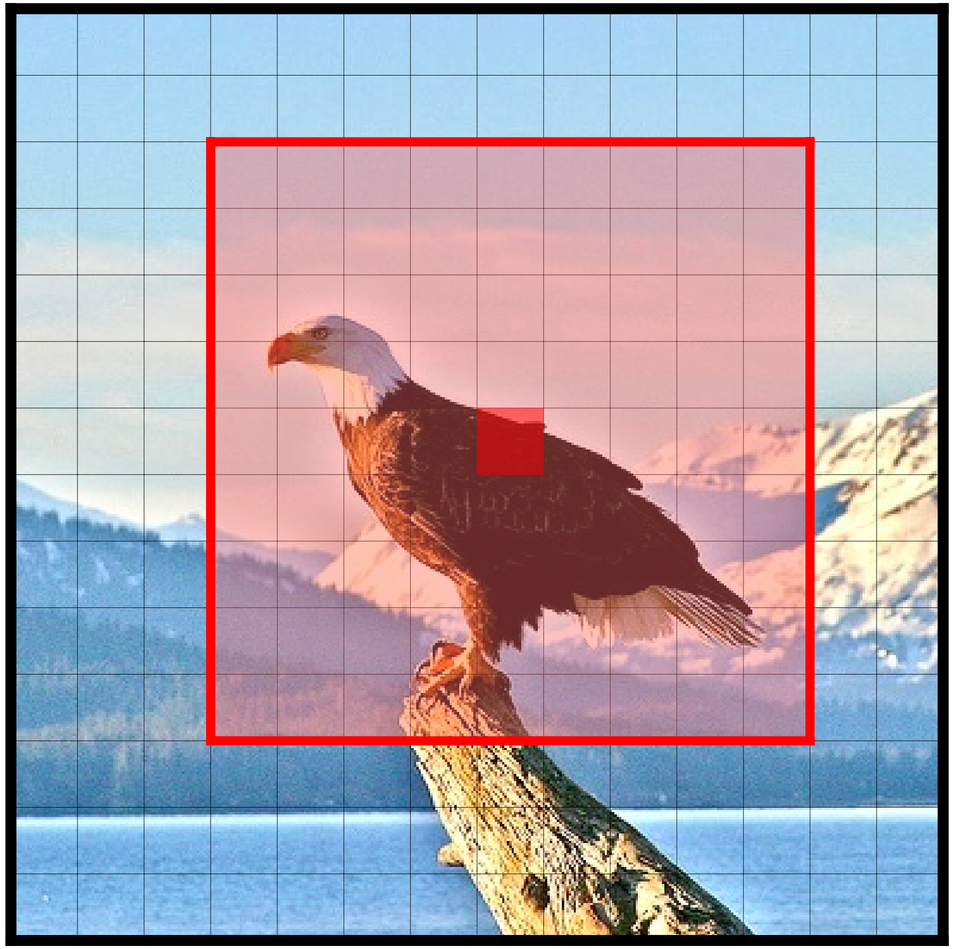
Neighborhood Attention



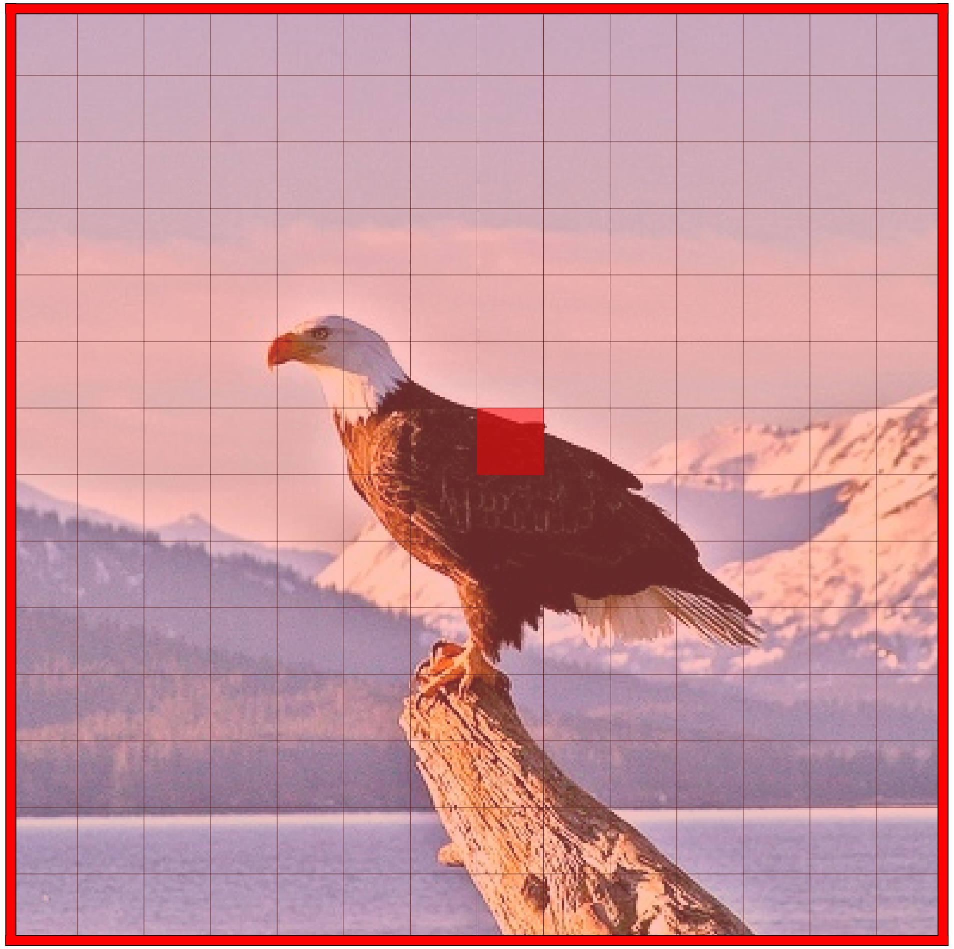
Self Attention

Neighborhood Attention approaches Self Attention

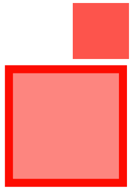




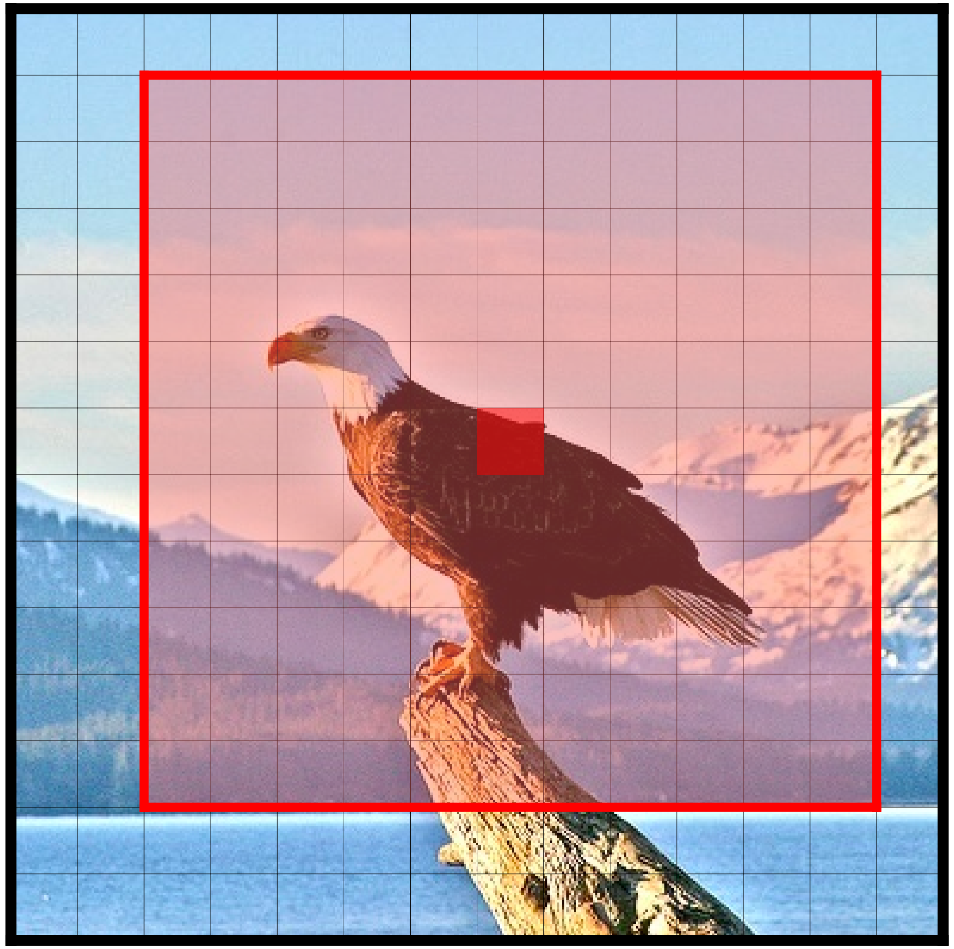
Neighborhood Attention



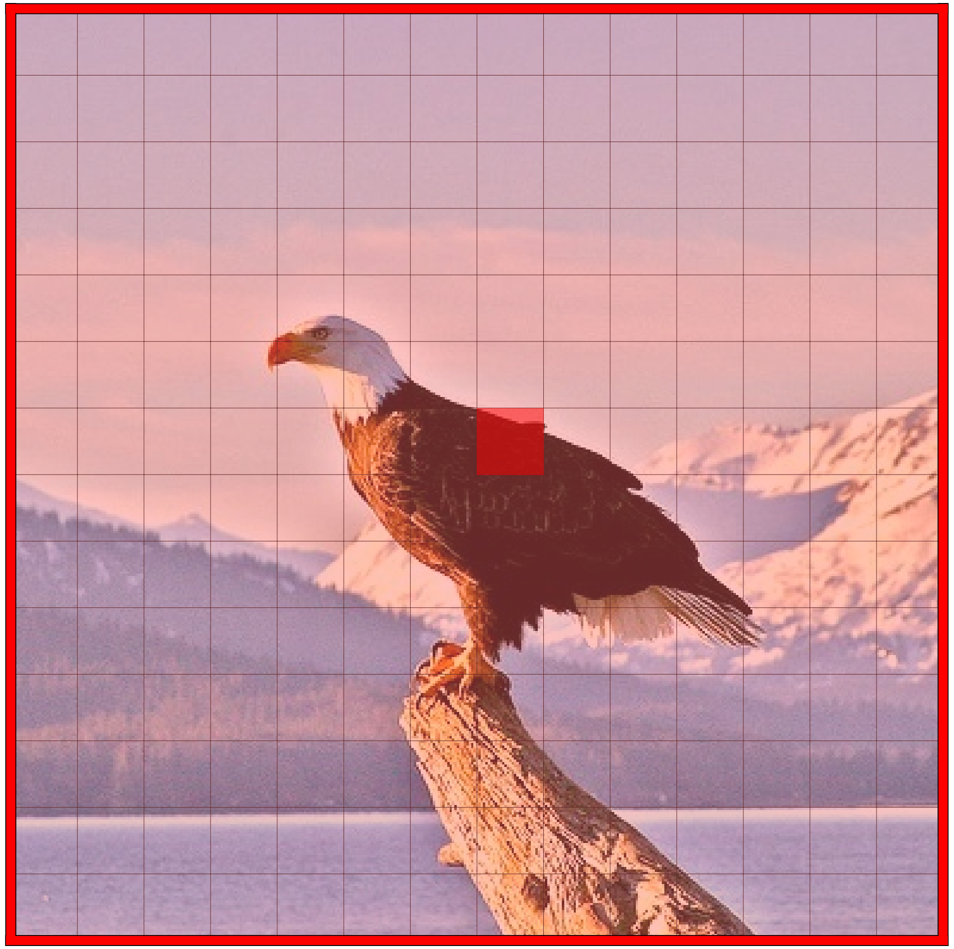
Self Attention



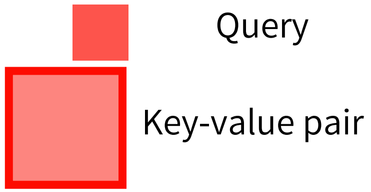
Neighborhood Attention approaches Self Attention



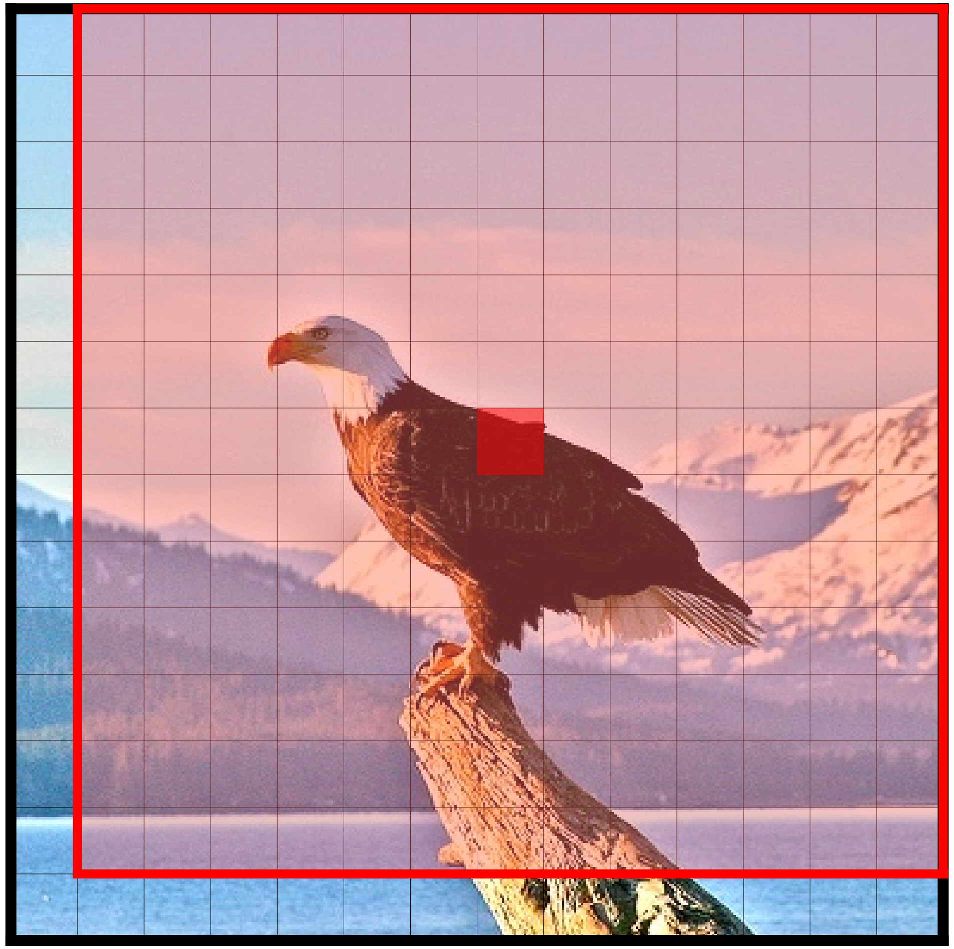
Neighborhood Attention



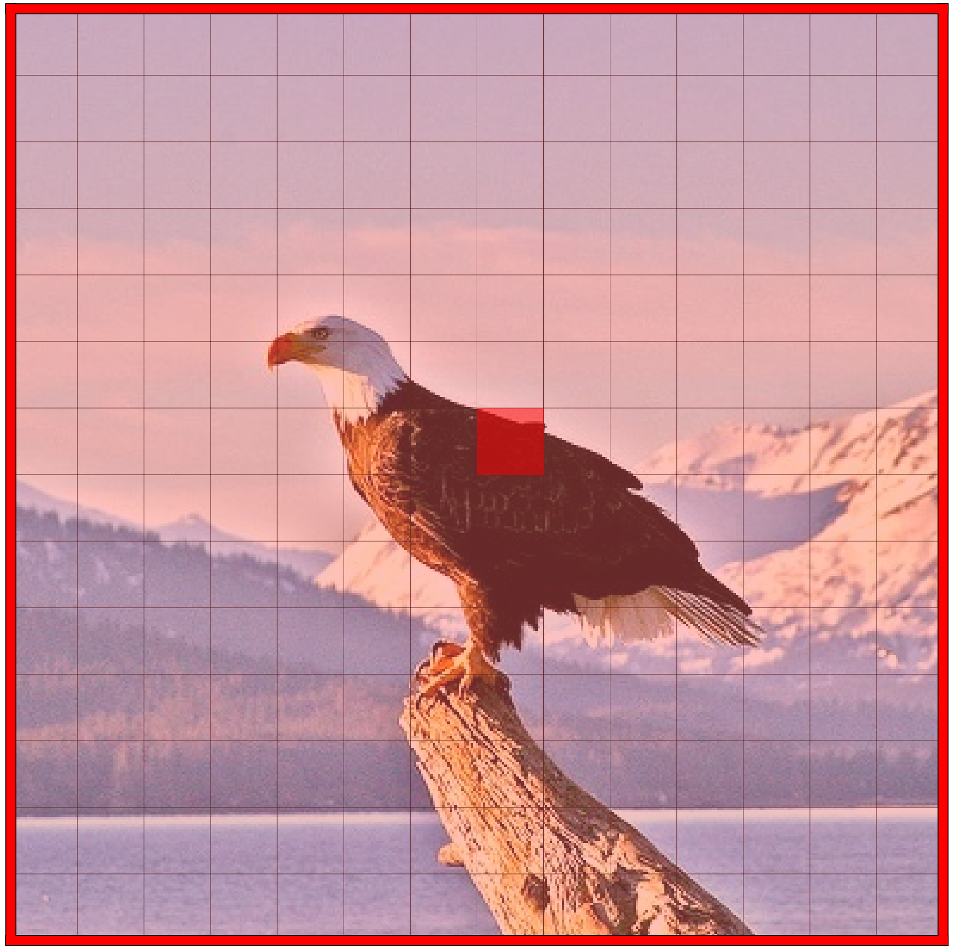
Self Attention



Neighborhood Attention approaches Self Attention



Neighborhood Attention



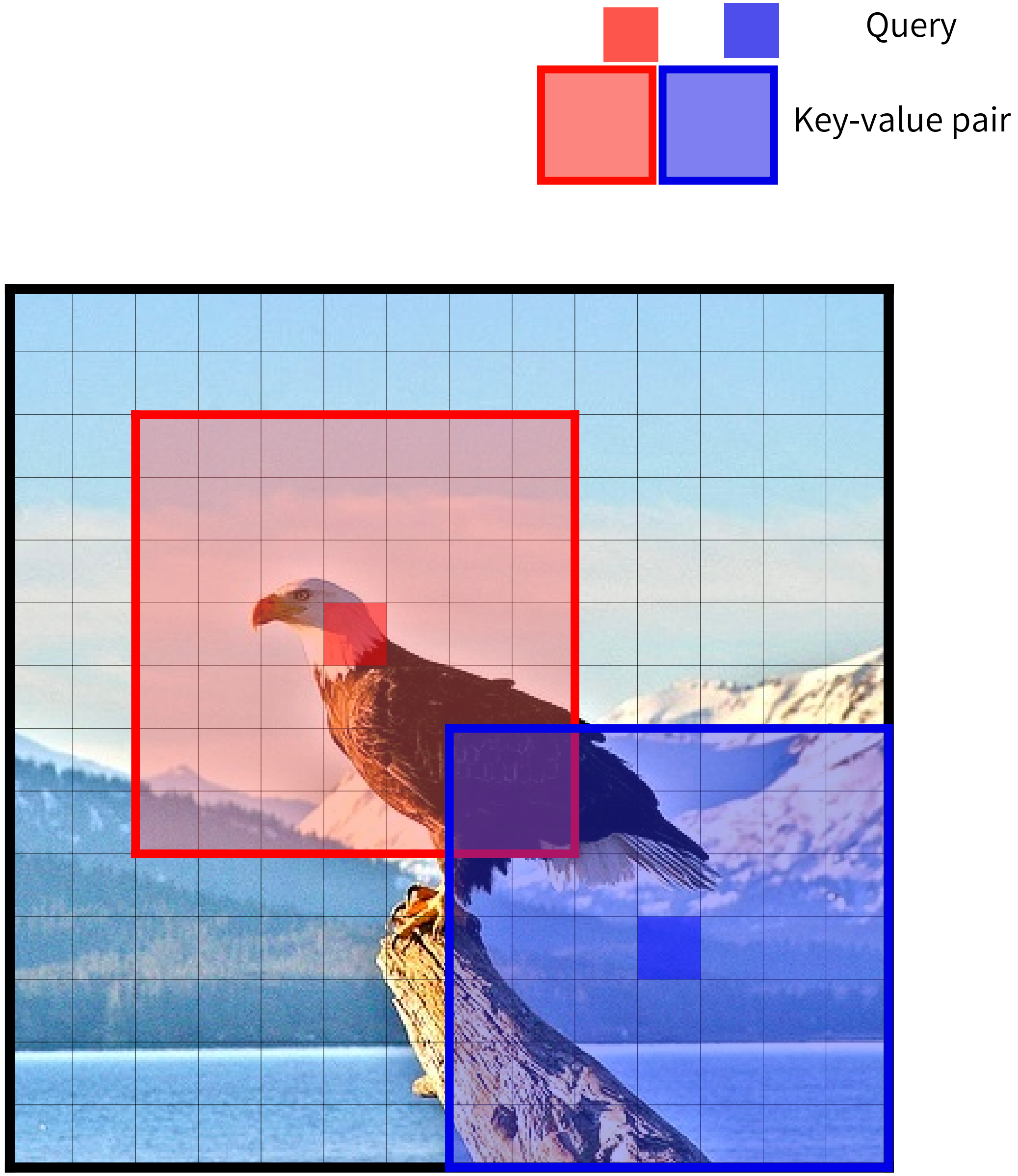
Self Attention

Implementation?

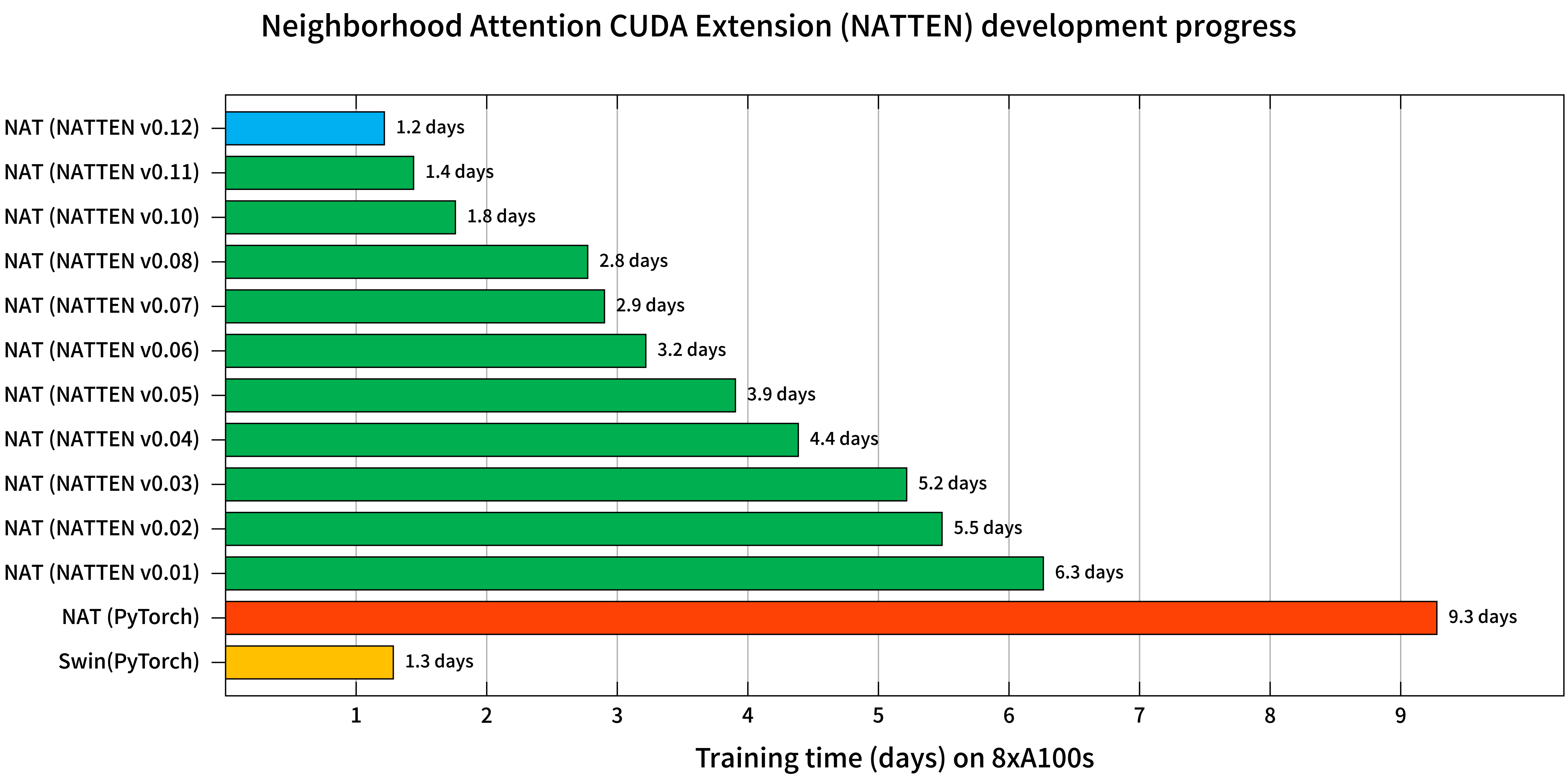
Implementation is no easier than sliding window attention.

(In fact it is is slightly more difficult than sliding window attention.)

Solution: $\mathcal{N}$ATTEN .

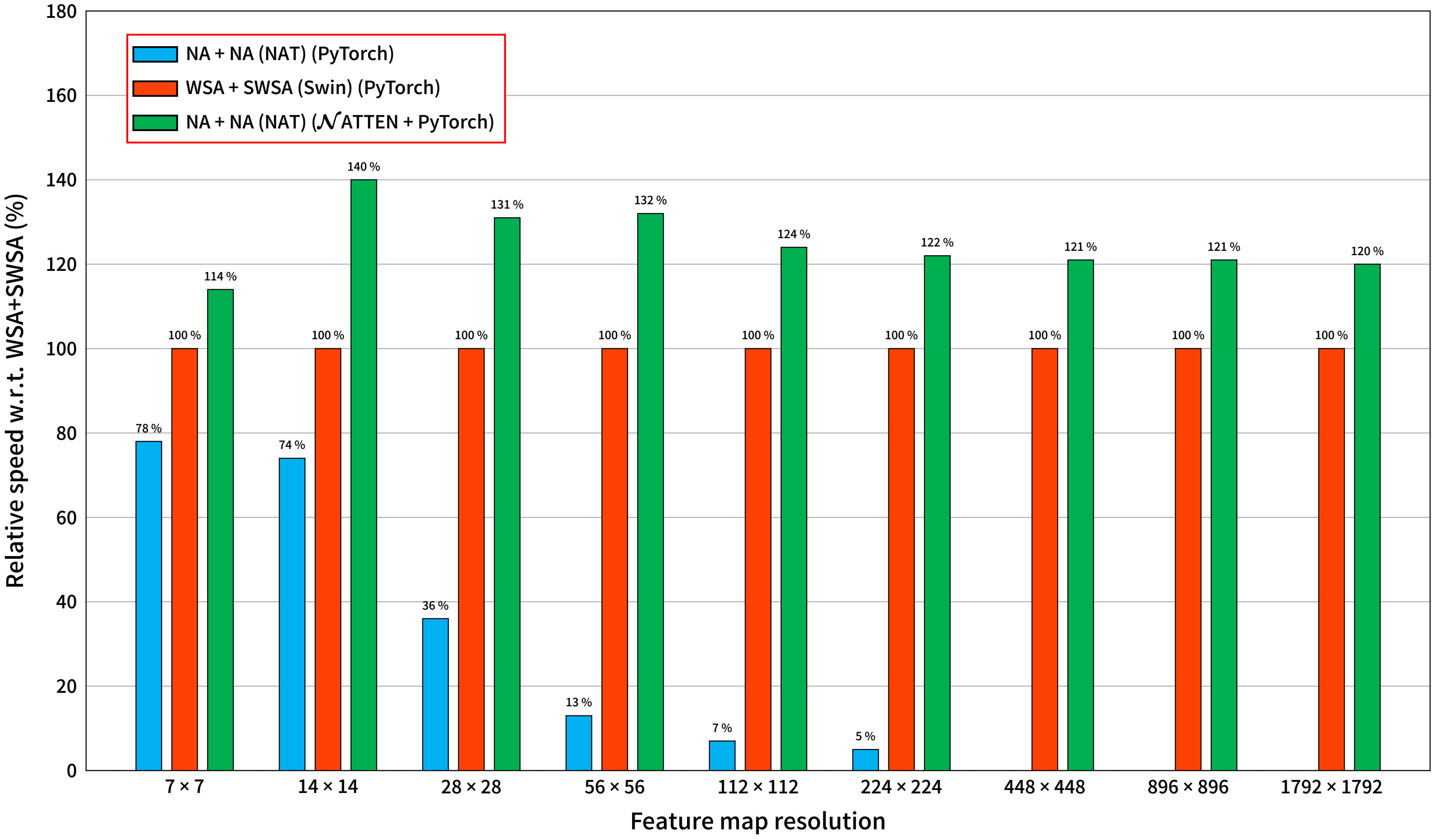


Neighborhood Attention



Neighborhood Attention CUDA Extension (NATTEN) development progress
NAT (NATTEN v0.12)
1.2 days
NAT (NATTEN v0.11)
1.4 days
NAT (NATTEN v0.10)
1.8 days
NAT (NATTEN v0.08)
2.8 days
NAT (NATTEN v0.07)
2.9 days
NAT (NATTEN v0.06)
3.2 days
NAT (NATTEN v0.05)
3.9 days
NAT (NATTEN v0.04)
4.4 days
NAT (NATTEN v0.03)
5.2 days
NAT (NATTEN v0.02)
5.5 days
NAT (NATTEN v0.01)
6.3 days
NAT (PyTorch)
9.3 days
Swin(PyTorch)
1.3 days
1
2
3
4
5
6
7
8
9
Training time (days) on 8xA100s

Per-layer latency compared to WSA+SWSA



shi-labs.com/natten

NATTEN

NEIGHBORHOOD ATTENTION EXTENSION

BRINGING ATTENTION TO A NEIGHBORHOOD NEAR YOU!

NATTEN is an extension to PyTorch, which provides the first fast sliding window attention with efficient CPU and CUDA kernels. It provides Neighborhood Attention (local attention) and Dilated Neighborhood Attention (sparse global attention, a.k.a. dilated local attention) as PyTorch modules for both 1D and 2D data.

GitHub / PyPI

Neighborhood Attention Transformers

Install with pip

Latest release: `0.14.6`

Please select your preferred PyTorch version with the correct CUDA build, or CPU build if you're not using CUDA:

PyTorch: 2.0 1.13 1.12.1 1.12 1.11 1.10.1 1.10 1.9 1.8

CUDA 11.8

CUDA 11.7

CPU

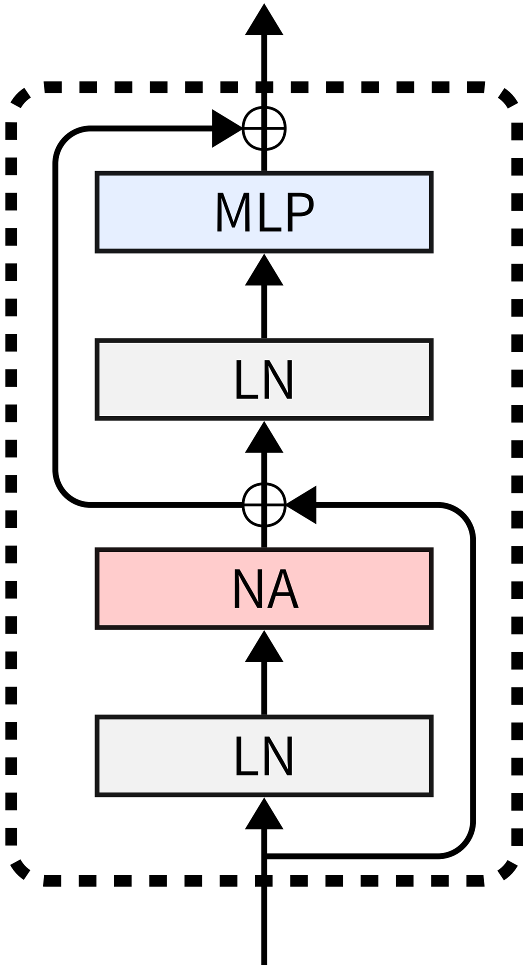
Run this command:

Copy

```
pip3 install natten -f https://shi-labs.com/natten/wheels/cu118/torch2.0.0/index.html
```

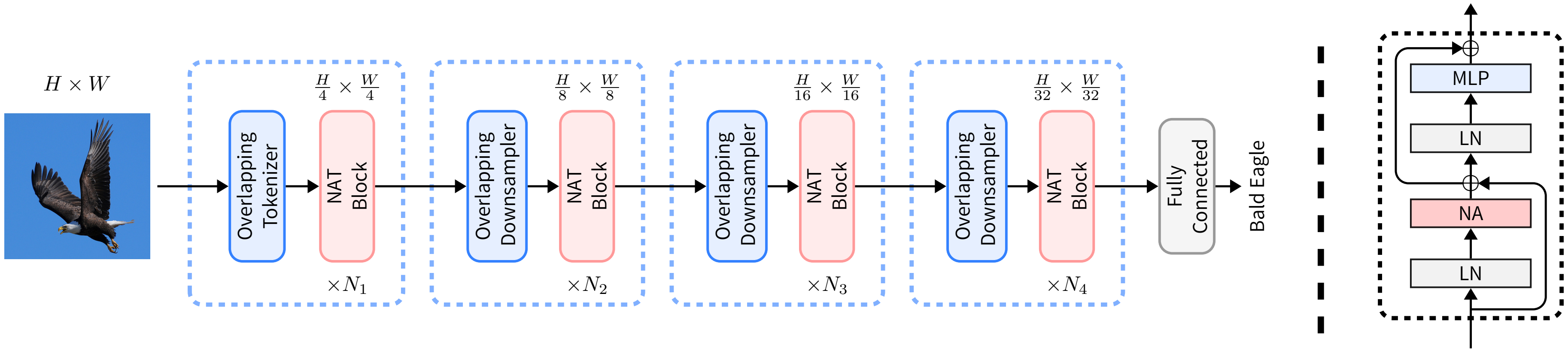
Architecture

- Transformer block with Neighborhood Attention (+ relative positional biases) instead of Self Attention / Window Self Attention,
- 7x7 NA windows (following Swin),
- Standard layer norms, skip connection, and MLP layer following the original Transformer design.



NAT Block

Architecture



Neighborhood Attention Transformer Architecture

NAT Block

- Hierarchical Vision Transformer with 4 levels,
- Conventional feature map scales (1/4, 1/8, 1/16, 1/32),
- Convolutional downsamplers instead of patched downsamplers,
- Slightly deeper, slightly thinner compared to Swin.

Image Classification

Model	# of Params	FLOPs	Thru. (imgs/sec)	Memory (GB)	Top-1 (%)
○ **NAT-M**	20 M	2.7 G	2135	2.4	81.8
○ **Swin-T**	28 M	4.5 G	1730	4.8	81.3
● **ConvNeXt-T**	28 M	4.5 G	2491	3.4	82.1
○ **NAT-T**	28 M	4.3 G	1541	2.5	**83.2**
○ **Swin-S**	50 M	8.7 G	1059	5.0	83.0
● **ConvNeXt-S**	50 M	8.7 G	1549	3.5	83.1
○ **NAT-S**	51 M	7.8 G	1051	3.7	**83.7**
○ **Swin-B**	88 M	15.4 G	776	6.7	83.5
● **ConvNeXt-B**	89 M	15.4 G	1107	4.8	83.8
○ **NAT-B**	90 M	13.7 G	783	5.0	**84.3**

ImageNet-1K image classification performance. Throughput and peak memory usage are measured from forward passes with a batch size of 256 on a single A100 GPU.

Object detection & instance segmentation

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	54.1	46.5	68.1	51.3	41.7	65.2	44.7
○ **Swin-T**	48 M	267 G	45.1	46.0	68.1	50.3	41.6	65.1	44.9
● **ConvNeXt-T**	48 M	262 G	52.0	46.2	67.0	50.8	41.7	65.0	44.9
○ **NAT-T**	48 M	258 G	44.5	47.7	69.0	52.6	42.6	66.1	45.9
○ **Swin-S**	69 M	359 G	31.7	48.5	70.2	53.5	43.3	67.3	46.6
○ **NAT-S**	70 M	330 G	34.8	48.4	69.8	53.2	43.2	66.9	46.5
				Cascade Mask R-CNN - 3x schedule					
○ **NAT-M**	77 M	704 G	27.8	50.3	68.9	54.9	43.6	66.4	47.2
○ **Swin-T**	86 M	745 G	25.1	50.4	69.2	54.7	43.7	66.6	47.3
● **ConvNeXt-T**	86 M	741 G	27.3	50.4	69.1	54.8	43.7	66.5	47.3
○ **NAT-T**	85 M	737 G	24.9	51.4	70.0	55.9	44.5	67.6	47.9
○ **Swin-S**	107 M	838 G	20.3	51.9	70.7	56.3	45.0	68.2	48.8
● **ConvNeXt-S**	108 M	827 G	23.0	51.9	70.8	56.5	45.0	68.4	49.1
○ **NAT-S**	108 M	809 G	21.7	52.0	70.4	56.3	44.9	68.1	48.6
○ **Swin-B**	145 M	982 G	17.3	51.9	70.5	56.4	45.0	68.1	48.9
● **ConvNeXt-B**	146 M	964 G	19.5	52.7	71.3	57.2	45.6	68.9	49.5
○ **NAT-B**	147 M	931 G	18.6	52.5	71.1	57.1	45.2	68.6	49.0

COCO object detection and instance segmentation performance. Throughput is measured on a single A100 GPU.

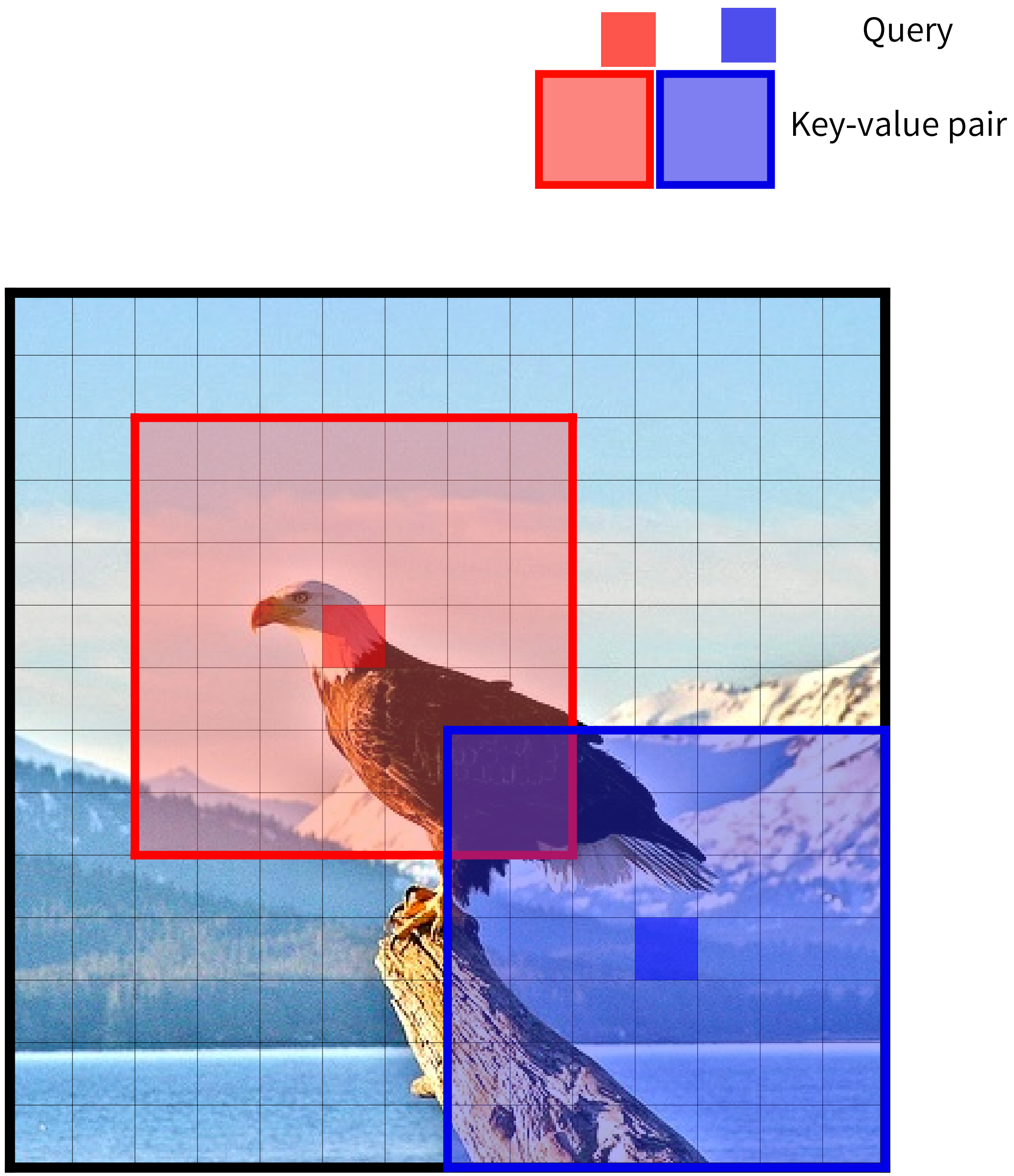
Semantic segmentation

Backbone	# of Params	FLOPs	Thru. (FPS)	mIoU single scale	mIoU multi scale
○ **NAT-M**	50 M	900 G	24.5	45.1	46.4
○ **Swin-T**	60 M	946 G	21.3	44.5	45.8
● **ConvNeXt-T**	60 M	939 G	23.3	46.0	46.7
○ **NAT-T**	58 M	934 G	21.4	47.1	48.4
○ **Swin-S**	81 M	1040 G	17.0	47.6	49.5
● **ConvNeXt-S**	82 M	1027 G	19.1	48.7	49.6
○ **NAT-S**	82 M	1010 G	17.9	48.0	49.5
○ **Swin-B**	121 M	1188 G	14.6	48.1	49.7
● **ConvNeXt-B**	122 M	1170 G	16.4	49.1	49.9
○ **NAT-B**	123 M	1137 G	15.6	48.5	49.7

ADE20K semantic segmentation performance. Throughput is measured on a single A100 GPU.

Conclusion

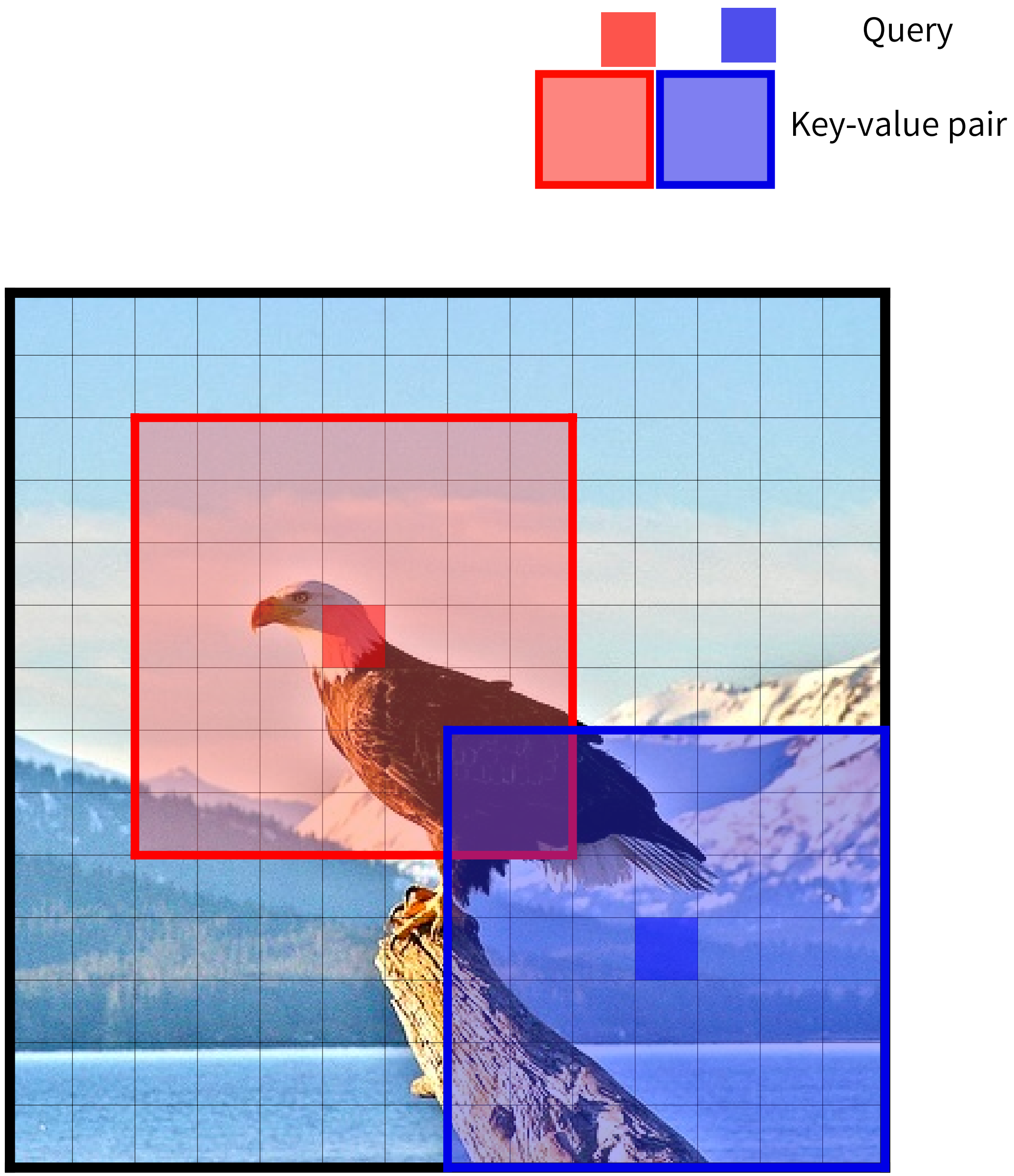
- Sliding window attention has been roadblocked by implementation and performance issues.
- We attempt to resolve the former with NATTEN, and the latter with Neighborhood Attention (NA).



Neighborhood Attention

Conclusion - Cont’d

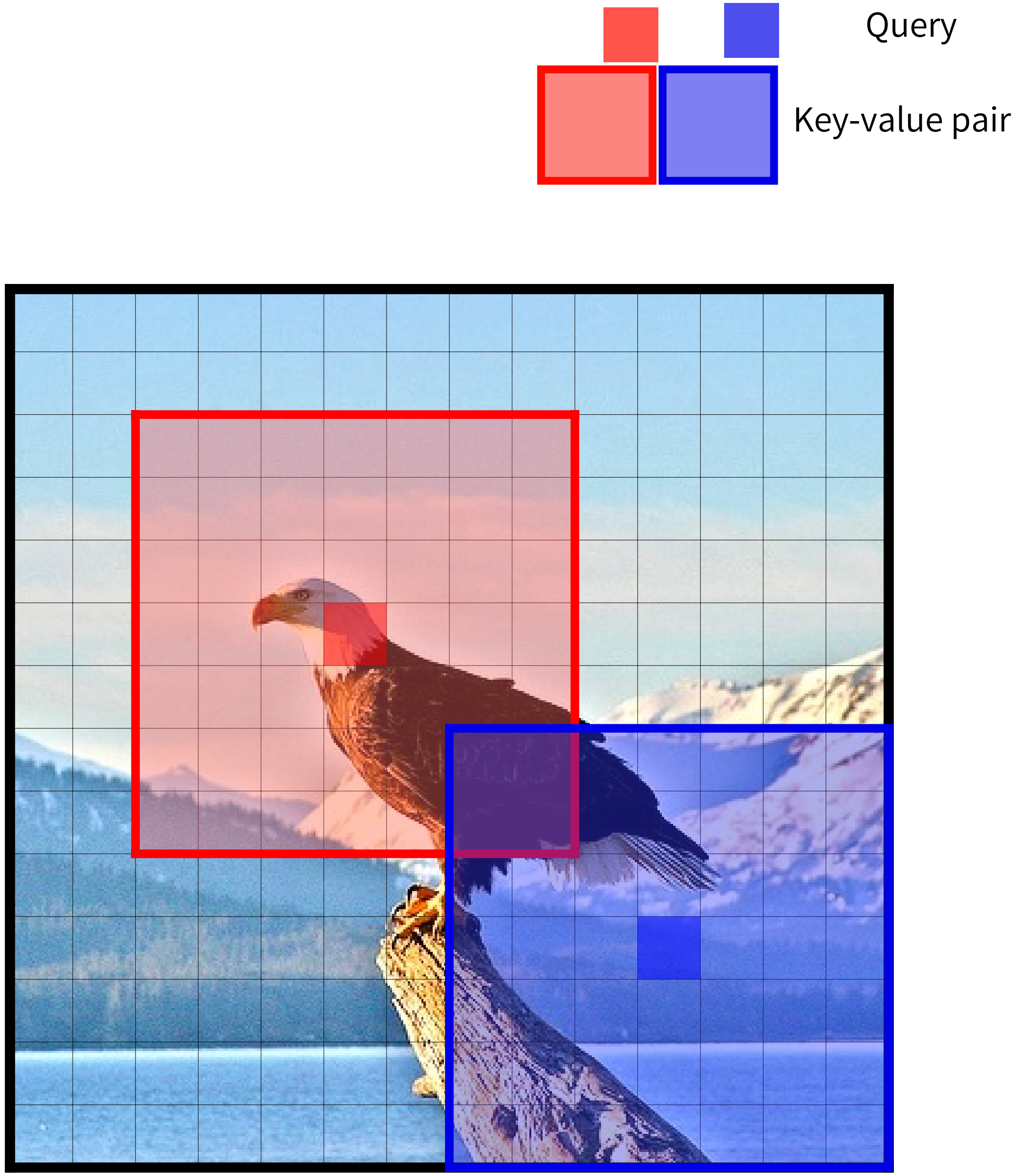
- When restricting self attention is desired, NA offers more flexibility compared to blocked attention (larger window sizes, dilation), and maintains properties such as symmetry and translational equivariance.



Neighborhood Attention

Conclusion - Cont’d

- Models based on NA are just as scalable as ones based on blocked attention (i.e. Swin) on both image classification, and downstream tasks.



Neighborhood Attention

Thank you for your *attention* !

- Please drop by and see our poster:
  - Tuesday, June 20^{th}, 4:30 – 6:00 PM
  - West Building Exhibit Halls, ABC 197

- Our GitHub page:
  - SHI-Labs/Neighborhood-Attention-Transformer
- Install $\mathcal{N}$ATTEN via pip:
  - SHI-Labs.com/NATTEN , or
  - `pip install natten`