Memory and IO in deep learning

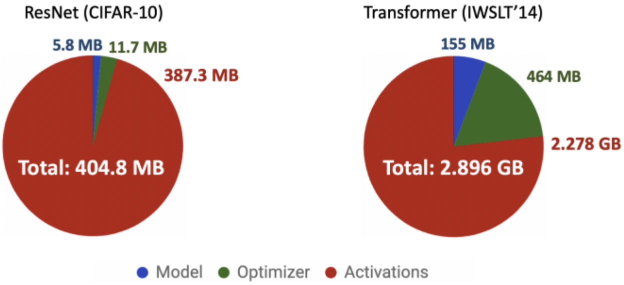
Lecture 12 — CS 577 Deep Learning

Instructor: Yutong Wang

Computer Science
Illinois Institute of Technology

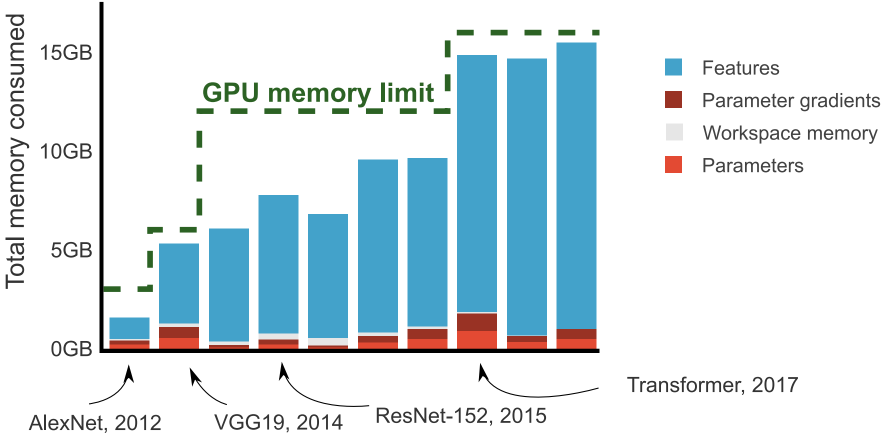
November 6, 2024

The memory wall



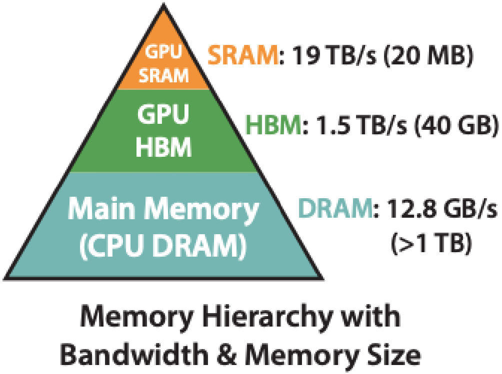
From Sohoni et al. 2019 "Low-memory neural network training: A technical report"

The memory wall



From Jain et al. 2020 “Checkmate: Breaking the memory wall with optimal tensor rematerialization”

Flash attention



From Dao et al. 2022 "FlashAttention: Fast and memory-efficient exact attention with io-awareness"

Memory and IO

- DL workloads often memory limited
- Moving data (IO) is costly

Roadmap for today

Memory:

- Warm-up: `requires_grad`
- Gradient checkpointing

IO:

- FlashAttention: how to compute the attention module without moving around so much data

Next: `requires_grad`

Size of NumPy array

```
import numpy as np
import sys

n = 1000
X_np = np.random.randn(n,n)
sys.getsizeof(X_np)

# >>> 8000128      # <- unit in bytes
```

Array in C

```
double *X;

X = (double *)malloc(n * n * sizeof(double));
```

```
n*n*8

# >>> 8000000
```

Size of NumPy array

```python
import numpy as np
import sys

n = 1000
X_np = np.random.randn(n,n)
sys.getsizeof(X_np)

# >>> 8000128     # <--- extra bytes for metadata
```

```python
n*n*8      # <--- size_of_float64  times  num_of_elements_in_X_np

# >>> 8000000
```

Aside: Size of PyTorch array

```python
import torch

n = 1000
X_torch = torch.randn(n,n)
sys.getsizeof(X_torch.untyped_storage())

# >>> 4000072
```

```python
n*n*4      # <--- size_of_float32  times  num_of_elements_in_X_torch

# >>> 4000000
```

tracemalloc

Allows tracking only memory allocated by numpy

```
import tracemalloc

# [...]
print("Simple example")

start_trace()

a = 1.0*np.zeros((n,n))     # n = 1000

print_trace_stats(snapshot_trace())
end_trace()
# >>> Simple example
# >>> memory allocated: 8 MB
```

tracemalloc

Allows tracking only memory allocated by numpy

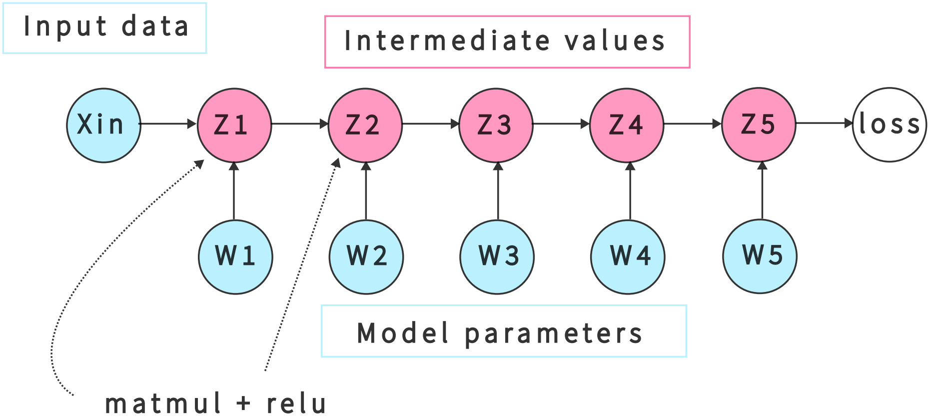
```
import tracemalloc

# [...]
print("No discarding")
start_trace()

a = 1.0*np.zeros((n,n))     # n = 1000
b = a*a
c = np.sum(b,axis=1)

print_trace_stats(snapshot_trace())
end_trace()
# >>> No discarding
# >>> memory allocated: 16 MB
```

tracemalloc

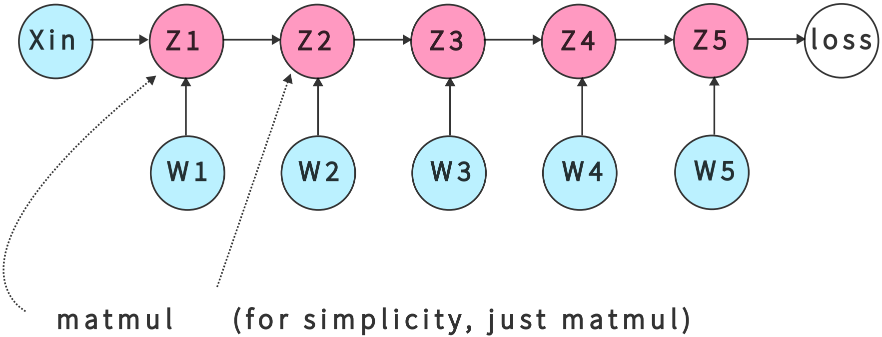
Allows tracking only memory allocated by numpy

```
import tracemalloc

# [...]
print("Discarding")
start_trace()

a = 1.0*np.zeros((n,n))     # n = 1000
b = a*a
c = np.sum(b,axis=1)
b = None

print_trace_stats(snapshot_trace())
end_trace()
# >>> With discarding
# >>> memory allocated: 8 MB
```

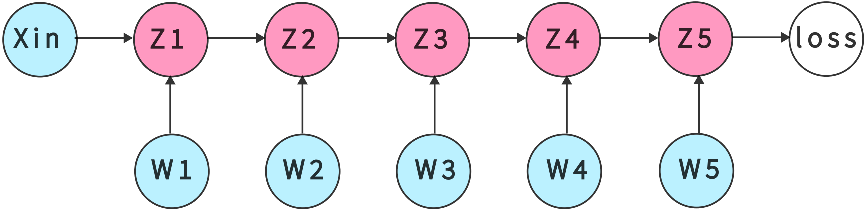
What is needed to calculate the backward?



What is needed to calculate the backward?

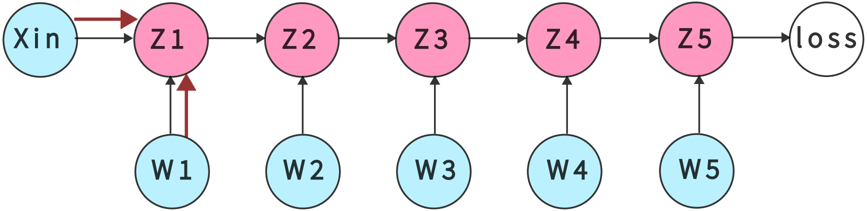


What is needed to calculate the backward?



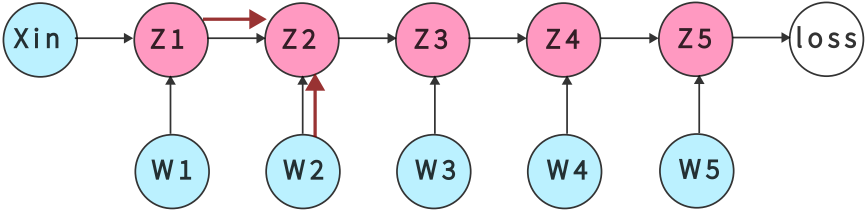
What is needed to calculate the backward?

```
Z1 = ag.matmul(Xin, W1)
```



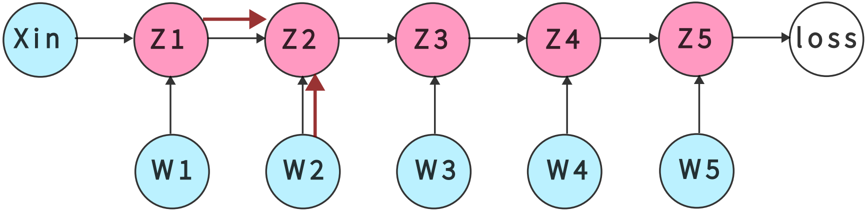
What is needed to calculate the backward?

```
Z2 = ag.matmul(Z1, W2)
```



What is needed to calculate the backward?

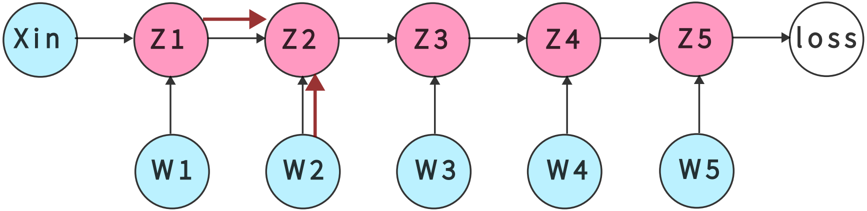
```
(Inside constructor for Z2)...

self.value = 1.0*value
self.grad = np.zeros_like(self.value)
```



Array sizes!

(Inside constructor for Z2)...

```
self.value = 1.0*value
self.grad = np.zeros_like(self.value)
```



What is needed to calculate the backward?

```python
def forward(x, weights):
    for w in weights:
        x = ag.matmul(x, w)
    return ag.sum(x)
```

What is needed to calculate the backward?

```python
def forward_traced(x, weights):
    start_trace()                # tracing
    mem_usage = []               # tracing
    for w in weights:
        x = ag.matmul(x, w)
        mem_usage.append(snapshot_trace()) # tracing
    l = ag.sum(x)
    mem_usage.append(snapshot_trace()) # tracing
    end_trace() # tracing
    return l, mem_usage
```

What is needed to calculate the backward?

```python
for i, trace_stats in enumerate(mem_usage_forward):
    print(f"layer {i}")
    print_trace_stats(trace_stats)
```

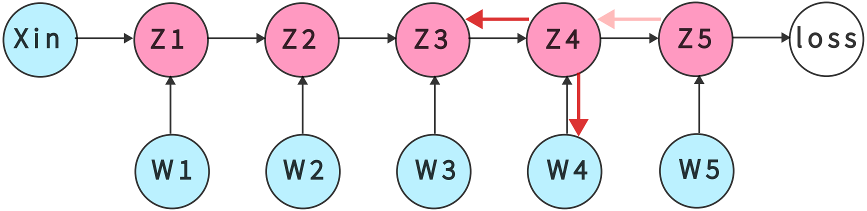
Output:

```
layer 0
memory allocated: 64 MB
layer 1
memory allocated: 128 MB
layer 2
memory allocated: 192 MB
```

Observation: double the amount of memory actually needed!

What is needed to calculate the backward?

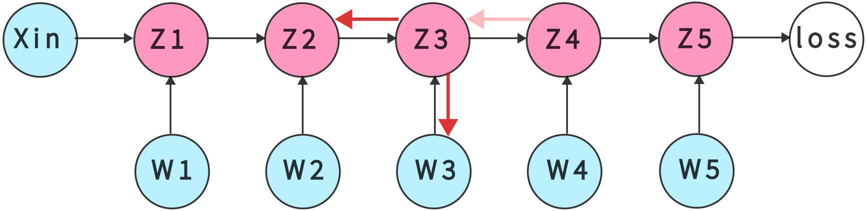
```
Z3.grad += np.matmul(Z4.grad, W4.value.T)
```




```
W4.grad += np.matmul(Z3.value.T, Z4.grad)
```

What is needed to calculate the backward?

```
Z2.grad += np.matmul(Z3.grad, W3.value.T)
```




```
W3.grad += np.matmul(Z2.value.T, Z3.grad)
```

Z4.grad never used again

Backward

```
        def backward(self):
            self.grad = np.array(1.0)

            topo_order = self.topological_sort()

            for node in reversed(topo_order):
                node._backward()
            return None
```

Backward with tracing

```
    def backward(self):
        self.grad = np.array(1.0)

        topo_order = self.topological_sort()

        start_trace()  # tracing
        mem_usage = [] # tracing

        for node in reversed(topo_order):
            node._backward()
            mem_usage.append(snapshot_trace()) # tracing
        end_trace() # tracing
        return mem_usage
```

Backward with tracing

```python
for i, trace_stats in enumerate(mem_usage_backward):
    print(f"backward step {i}")
    print_trace_stats(trace_stats)
```

Output:

```
backward step 0
memory allocated: 0 MB
backward step 1
memory allocated: 0 MB
backward step 2
memory allocated: 0 MB
```

Observation: After calling backward on a tensor, its grad is no longer needed

Backward with tracing

```python
for i, trace_stats in enumerate(mem_usage_backward):
    print(f"backward step {i}")
    print_trace_stats(trace_stats)
```

Output:

```
backward step 0
memory allocated: 0 MB
backward step 1
memory allocated: 0 MB
backward step 2
memory allocated: 0 MB
```

Observation: After calling backward on a tensor, its grad is no longer needed

Next: Shift the task of initialize the grad to the backward pass

Keeping track of grad — is it needed?

```
class Tensor: # Tensor with grads
    def __init__(self,
                 value,
        # [...]

        self.value = 1.0*value
        self.grad = np.zeros_like(self.value) # <---- WASTEFUL!
```

Keeping track of grad — is it needed?

```
class Tensor: # Tensor with grads
    def __init__(self,
                 value,
                 requires_grad=False, # <-- Flag for keeping grad
        # [...]

        self.value = 1.0*value

        self.grad = None
        if self.requires_grad:
            self.grad = np.zeros_like(self.value)
```

Keeping track of grad — is it needed?

```
    class Tensor: # Tensor with grads
        def __init__(self,
                     value,
                     requires_grad=False, # <-- Flag for keeping grad
            # [...]

            self.value = 1.0*value

            self.grad = None
            if self.requires_grad:
                self.grad = np.zeros_like(self.value)
# [...]

weights = [ag.Tensor(0.02*np.random.randn(dim_hidden, dim_hidden),
                     requires_grad = True) for _ in range(num_layers)]
#                    ^^^^^^^^^^^^^^^^^^^^
```

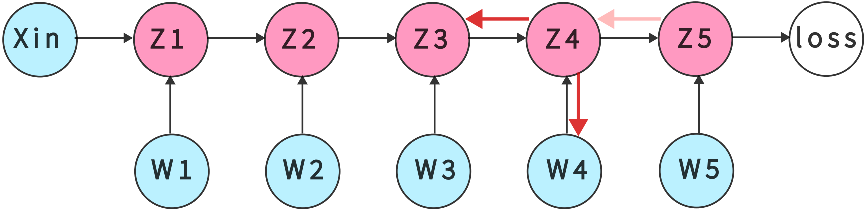
Keeping track of grad — is it needed?

```
    class Tensor: # Tensor with grads
        def __init__(self,
                     value,
                     requires_grad=False, # <-- Flag for keeping grad
            # [...]

            self.value = 1.0*value

            self.grad = None
            if self.requires_grad:
                self.grad = np.zeros_like(self.value)
# [...]

    for w in weights:
        x = ag.matmul(x, w)
        # x.requires_grad == False by default
```

Problem: during backward, we might encounter `None`'s when we should expect `grad`.

What is needed to calculate the backward?

```
Z3.grad += np.matmul(Z4.grad, W4.value.T)
```




```
W4.grad += np.matmul(Z3.value.T, Z4.grad)
```

Exercise 1: discarding the grad

```
# inside ag.__matmul__
            def _backward():
                # YOUR CODE HERE FOR initializing the grad

                self.grad += np.matmul(output.grad, other.value.T)
                other.grad += np.matmul(self.value.T, output.grad)

                # YOUR CODE HERE FOR discarding the grad
                return None
```

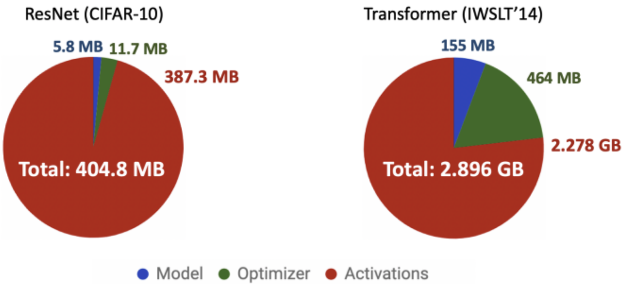
If your answer is correct, then the “sanity checks” should have the expected output.

The memory wall

- We've reduced the memory consumption of forward by a factor 2 without incurring much additional backward memory burden.

The memory wall

- We've reduced the memory consumption of forward by a factor 2 without incurring much additional backward memory burden.
- But recall that...



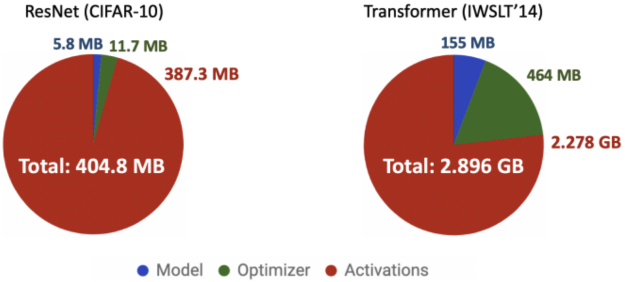
From Sohoni et al. 2019 "Low-memory neural network training: A technical report"

The memory wall

- We've reduced the memory consumption of forward by a factor 2 without incurring much additional backward memory burden.

The memory wall

- We've reduced the memory consumption of forward by a factor 2 without incurring much additional backward memory burden.
- But recall that...



From Sohoni et al. 2019 "Low-memory neural network training: A technical report"

The memory wall

Can we do something about this?

```
layer 0
memory allocated: 32 MB
layer 1
memory allocated: 65 MB
layer 2
memory allocated: 98 MB
layer 3
memory allocated: 131 MB
layer 4
memory allocated: 163 MB
...
```

Tensor Rematerialization

- Discard the activations as soon as you can

Tensor Rematerialization

- Discard the activations as soon as you can
- "Beam" them back/recompute them only when you need them

From Berman et al. 1995 "Star Trek: Voyager"

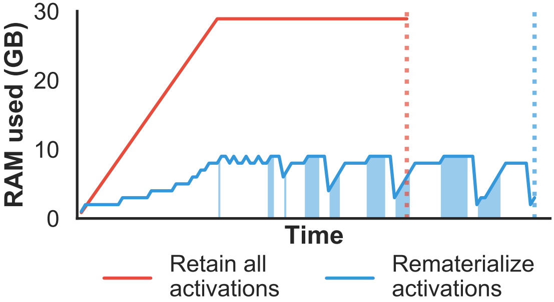
Tensor Rematerialization

- Discard the activations as soon as you can
- "Beam" them back/recompute them only when you need them

From Berman et al. 1995 "Star Trek: Voyager"

- Save *some* activations as "checkpoints" to speed things up

Tensor Rematerialization

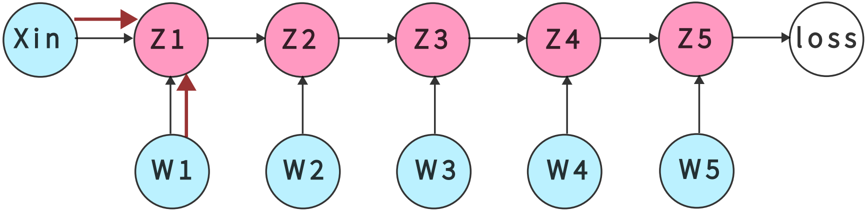


From Jain et al. 2020 “Checkmate: Breaking the memory wall with optimal tensor rematerialization”

- Also known as **gradient checkpoint**, activation checkpointing, recomputation...

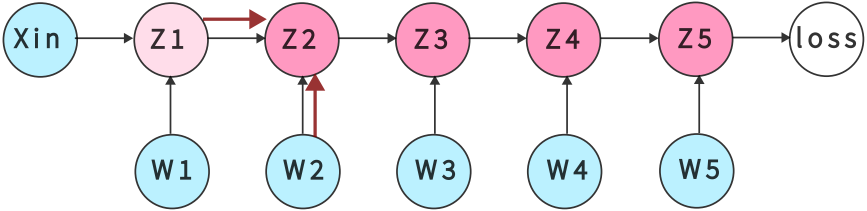
Rematerialization: forward phase

```
Z1 = ag.matmul(Xin, W1)
```

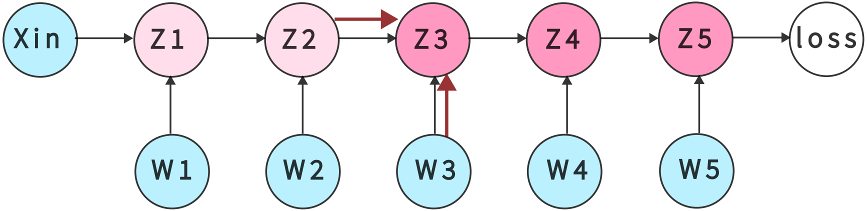


Rematerialization: forward phase

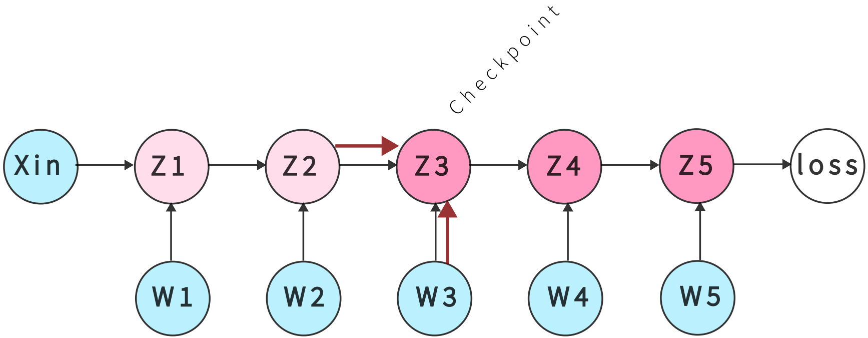
```
Z2 = ag.matmul(Z1, W2)
```



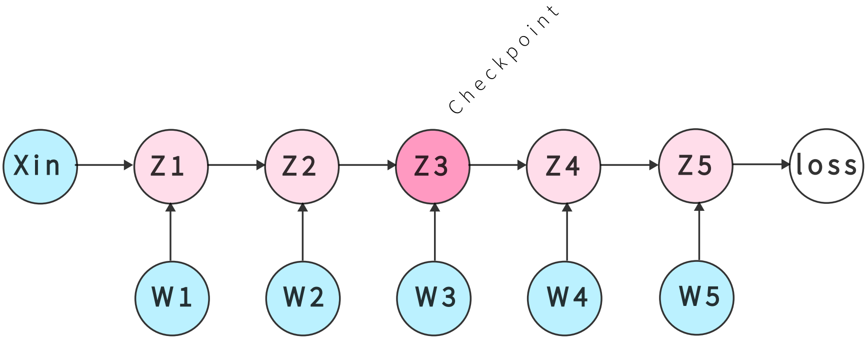
Rematerialization: forward phase



Rematerialization: forward phase

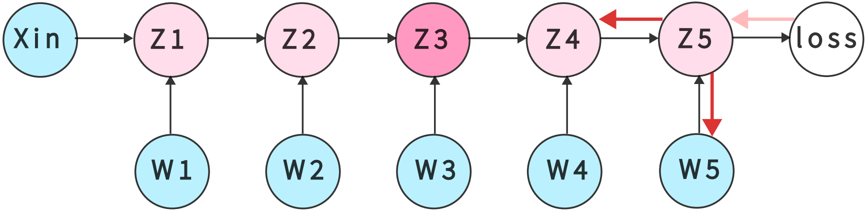


Rematerialization: forward phase



Rematerialization: backward phase

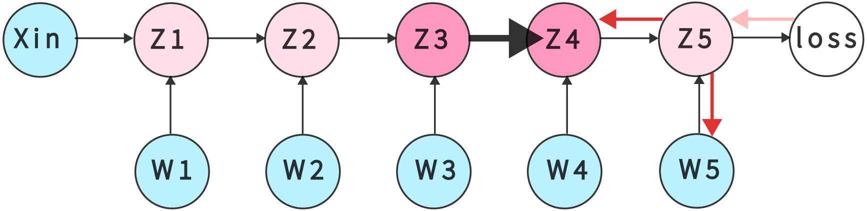
```
Z4.grad += np.matmul(Z5.grad, W5.value.T)
```




```
W5.grad += np.matmul(Z4.value.T, Z5.grad)
```

Rematerialization: backward phase

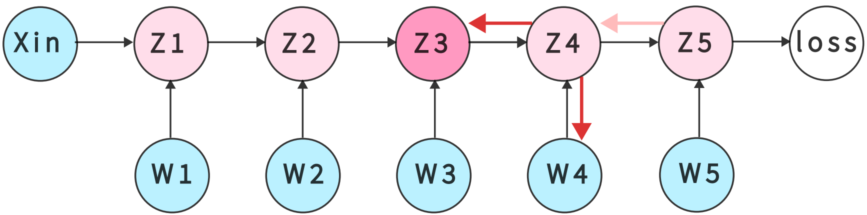
```
Z4.grad += np.matmul(Z5.grad, W5.value.T)
```




```
W5.grad += np.matmul(Z4.value.T, Z5.grad)
```

Rematerialization: backward phase

```
Z3.grad += np.matmul(Z4.grad, W4.value.T)
```




```
W4.grad += np.matmul(Z3.value.T, Z4.grad)
```

Exercise 2: rematerialization

```
    class Tensor: # Tensor with grads
        def __init__(self,
                     value,
                     requires_grad=False,
                     rematerializer = None, # None means don't
   rematerialize, KEEP
# [...]
```

Exercise 2: rematerialization

```python
def forward_traced_with_rematerializer(x, weights):
    start_trace()
    mem_usage = [] # tracing

    checkpoints = [5] # these layers are checkpoints

    farthest_checkpoint = 0 # this is the input data
    x_at_farthest_checkpoint = x

    for i, w in enumerate(weights): # main training loop
        if i in checkpoints:
            x = ag.matmul(x, w)
            farthest_checkpoint = i
            x_at_farthest_checkpoint = x
# [...]
```

Exercise 2: rematerialization

```
for i, w in enumerate(weights): # main training loop
    if i in checkpoints:
        x = ag.matmul(x, w)
        farthest_checkpoint = i
        x_at_farthest_checkpoint = x
    else:
        def _rematerializer():
            xval = x_at_farthest_checkpoint.value
            for w in weights[farthest_checkpoint:(i+1)]:
                xval = np.matmul(xval,w.value)
            return xval
        x = ag.matmul(x, w)
        x.rematerializer = _rematerializer
```

Exercise 2: task 1: rematerialize!

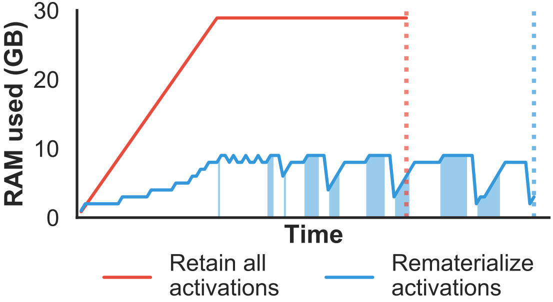
```
def backward(self):
    # [...]
    for node in reversed(topo_order):
        # YOUR CODE HERE FOR rematerializing "input.value", if it is none

        node._backward()
```

Exercise 2: task 2: discard!

```python
    def __matmul__(self,other):
        # [...]
        def _backward():
            # [...]
            self.grad += np.matmul(output.grad, other.value.T)
            other.grad += np.matmul(self.value.T, output.grad)

            # YOUR CODE HERE FOR discarding activations for "self" and "other"
            # hint: add a helper function to make it neater
            # hint: see "discard_value_if_has_rematerializer" below
            # [lines skipped ...]
            return None

        output._backward = _backward
        # YOUR CODE HERE FOR discarding activations for "self" and "other"
```

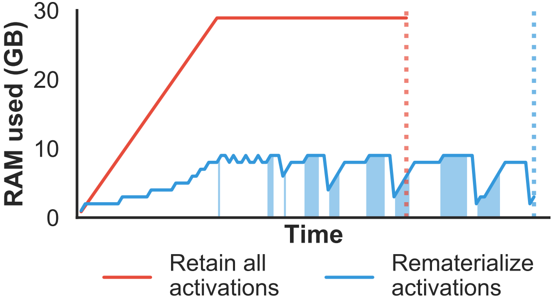
Summary

- Less memory used in the forward



Summary

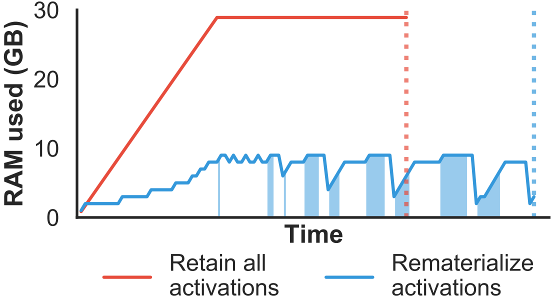
- Less memory used in the forward



- More computation (for rematerialization).
  (75 matmuls w/ rematerialization vs. 30 matmuls w/o rematerialization).

Summary

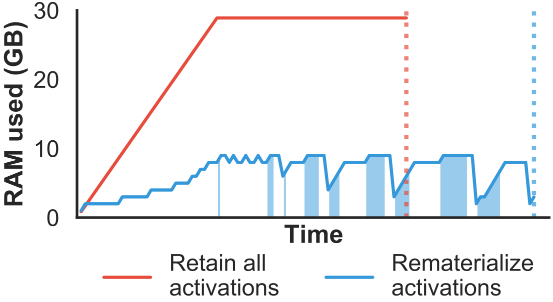
- Less memory used in the forward



- More computation (for rematerialization).
  (75 matmuls w/ rematerialization vs. 30 matmuls w/o rematerialization).
- **Question**: what about more complicated computational graph?

Summary

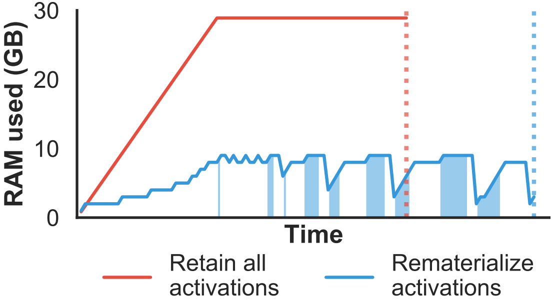
- Less memory used in the forward



- More computation (for rematerialization).
  (75 matmuls w/ rematerialization vs. 30 matmuls w/o rematerialization).
- **Question**: what about more complicated computational graph?
- **Question**: optimal way of selecting the checkpoints?

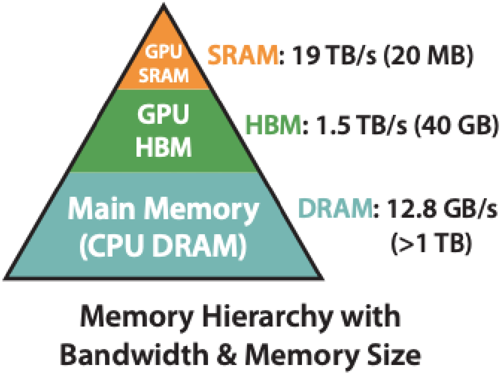
Summary

- Less memory used in the forward



- More computation (for rematerialization).
  (75 matmuls w/ rematerialization vs. 30 matmuls w/o rematerialization).
- **Question**: what about more complicated computational graph?
- **Question**: optimal way of selecting the checkpoints?
- **Next**: IO and FlashAttention

Flash attention



From Dao et al. 2022 "FlashAttention: Fast and memory-efficient exact attention with io-awareness"

Self-attention

- Let $\mathbf{X} \in \mathbb{R}^{C \times d}$ (note everything is transposed compared to last time)

$$\texttt{attention}(\mathbf{X}; \theta) := \mathrm{softmax}\left(\mathbf{X}\mathbf{W}^{(Q)}\mathbf{W}^{(K)\top}\mathbf{X}^{\top}\right)\mathbf{X}\mathbf{W}^{(V)} \in \mathbb{R}^{d \times C}.$$

Self-attention

- Let $\mathbf{X} \in \mathbb{R}^{C \times d}$ (note everything is transposed compared to last time)

$$\texttt{attention}(\mathbf{X}; \theta) := \operatorname{softmax}\left(\mathbf{X}\mathbf{W}^{(Q)}\mathbf{W}^{(K)\top}\mathbf{X}^\top\right)\mathbf{X}\mathbf{W}^{(V)} \in \mathbb{R}^{d \times C}.$$

- parameters

$$\theta^{(\texttt{att})} = [\mathbf{W}^{(Q)}, \mathbf{W}^{(K)}, \mathbf{W}^{(V)}]$$

Self-attention

- Let $\mathbf{X} \in \mathbb{R}^{C\times d}$ (note everything is transposed compared to last time)

$$\texttt{attention}(\mathbf{X};\theta) := \operatorname{softmax}\left(\mathbf{X}\mathbf{W}^{(Q)}\mathbf{W}^{(K)\top}\mathbf{X}^{\top}\right)\mathbf{X}\mathbf{W}^{(V)} \in \mathbb{R}^{d\times C}.$$

- parameters

$$\theta^{(\texttt{att})} = [\mathbf{W}^{(Q)}, \mathbf{W}^{(K)}, \mathbf{W}^{(V)}]$$

- ("Q, K, V" stands for "query", "key", "value", respectively) where

$$\mathbf{W}^{(Q)} \text{ and } \mathbf{W}^{(K)} \text{ and } \mathbf{W}^{(V)} \in \mathbb{R}^{d\times d}.$$

Self-attention

- Let $\mathbf{X} \in \mathbb{R}^{C \times d}$ (note everything is transposed compared to last time)

$$\texttt{attention}(\mathbf{X}; \theta) := \operatorname{softmax}\left(\mathbf{X}\mathbf{W}^{(Q)}\mathbf{W}^{(K)\top}\mathbf{X}^\top\right)\mathbf{X}\mathbf{W}^{(V)} \in \mathbb{R}^{d \times C}.$$

- parameters

$$\theta^{(\texttt{att})} = [\mathbf{W}^{(Q)}, \mathbf{W}^{(K)}, \mathbf{W}^{(V)}]$$

- (“Q, K, V” stands for “query”, “key”, “value”, respectively)
  where

$$\mathbf{W}^{(Q)} \text{ and } \mathbf{W}^{(K)} \text{ and } \mathbf{W}^{(V)} \in \mathbb{R}^{d \times d}.$$

- “**seq-2-seq**”: maps the sequence $\mathbf{X}$ to another sequence $\texttt{attention}(\mathbf{X}; \theta)$

Self-attention

- Let $\mathbf{X} \in \mathbb{R}^{C \times d}$ (note everything is transposed compared to last time)

$$\texttt{attention}(\mathbf{X}; \theta) := \mathrm{softmax}\Big(\underbrace{\mathbf{X}\mathbf{W}^{(Q)}}_{\mathbf{Q}} \underbrace{\mathbf{W}^{(K)\top}\mathbf{X}^{\top}}_{\mathbf{K}^{\top}} \Big) \underbrace{\mathbf{X}\mathbf{W}^{(V)}}_{\mathbf{V}} \in \mathbb{R}^{d \times C}.$$

- parameters

$$\theta^{(\texttt{att})} = [\mathbf{W}^{(Q)}, \mathbf{W}^{(K)}, \mathbf{W}^{(V)}]$$

- (“Q, K, V” stands for “query”, “key”, “value”, respectively)
  where

$$\mathbf{W}^{(Q)} \text{ and } \mathbf{W}^{(K)} \text{ and } \mathbf{W}^{(V)} \in \mathbb{R}^{d \times d}.$$

- “**seq-2-seq**”: maps the sequence $\mathbf{X}$ to another sequence $\texttt{attention}(\mathbf{X}; \theta)$

Self-attention

- Let $\mathbf{X} \in \mathbb{R}^{C \times d}$ (note everything is transposed compared to last time)

$$\texttt{attention}(\mathbf{X}; \theta) := \text{softmax}\Big(\underbrace{\mathbf{Q}\mathbf{K}^\top}_{\mathbf{S}}\Big)\mathbf{V} \in \mathbb{R}^{d \times C}.$$

- parameters

$$\theta^{(\texttt{att})} = [\mathbf{W}^{(Q)}, \mathbf{W}^{(K)}, \mathbf{W}^{(V)}]$$

- ("Q, K, V" stands for "query", "key", "value", respectively) where

$$\mathbf{W}^{(Q)} \text{ and } \mathbf{W}^{(K)} \text{ and } \mathbf{W}^{(V)} \in \mathbb{R}^{d \times d}.$$

- "**seq-2-seq**": maps the sequence $\mathbf{X}$ to another sequence $\texttt{attention}(\mathbf{X}; \theta)$

Self-attention

- Let $\mathbf{X} \in \mathbb{R}^{C \times d}$ (note everything is transposed compared to last time)

$$\mathtt{attention}(\mathbf{X}; \theta) := \underbrace{\mathrm{softmax}\Big(\underbrace{\mathbf{Q}\mathbf{K}^\top}_{\mathbf{S}}\Big)}_{\mathbf{P}=\mathrm{softmax}(\mathbf{S})} \mathbf{V} \in \mathbb{R}^{d \times C}.$$

- parameters

$$\theta^{(\mathtt{att})} = [\mathbf{W}^{(Q)}, \mathbf{W}^{(K)}, \mathbf{W}^{(V)}]$$

- ("Q, K, V" stands for "query", "key", "value", respectively) where

$$\mathbf{W}^{(Q)} \text{ and } \mathbf{W}^{(K)} \text{ and } \mathbf{W}^{(V)} \in \mathbb{R}^{d \times d}.$$

- "**seq-2-seq**": maps the sequence $\mathbf{X}$ to another sequence $\mathtt{attention}(\mathbf{X}; \theta)$

Self-attention

- Let $\mathbf{X} \in \mathbb{R}^{C\times d}$ (note everything is transposed compared to last time)

$$\texttt{attention}(\mathbf{X};\theta) := \text{softmax}\left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V} \in \mathbb{R}^{d\times C}.$$

- parameters

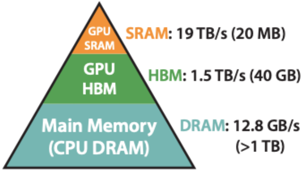
$$\theta^{(\texttt{att})} = [\mathbf{W}^{(Q)}, \mathbf{W}^{(K)}, \mathbf{W}^{(V)}]$$

- (“Q, K, V” stands for “query”, “key”, “value”, respectively) where

$$\mathbf{W}^{(Q)} \text{ and } \mathbf{W}^{(K)} \text{ and } \mathbf{W}^{(V)} \in \mathbb{R}^{d\times d}.$$

- “**seq-2-seq**”: maps the sequence $\mathbf{X}$ to another sequence $\texttt{attention}(\mathbf{X};\theta)$

Flash attention



Memory Hierarchy with
Bandwidth & Memory Size

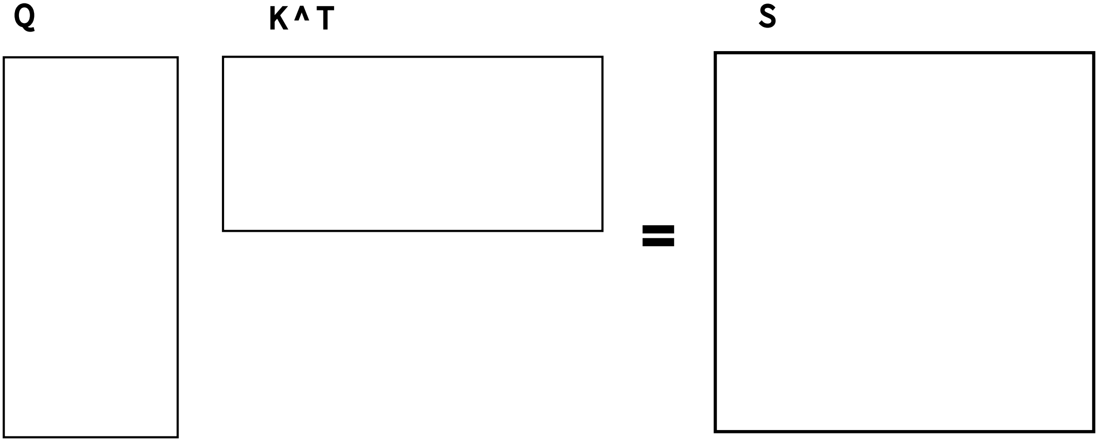
Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

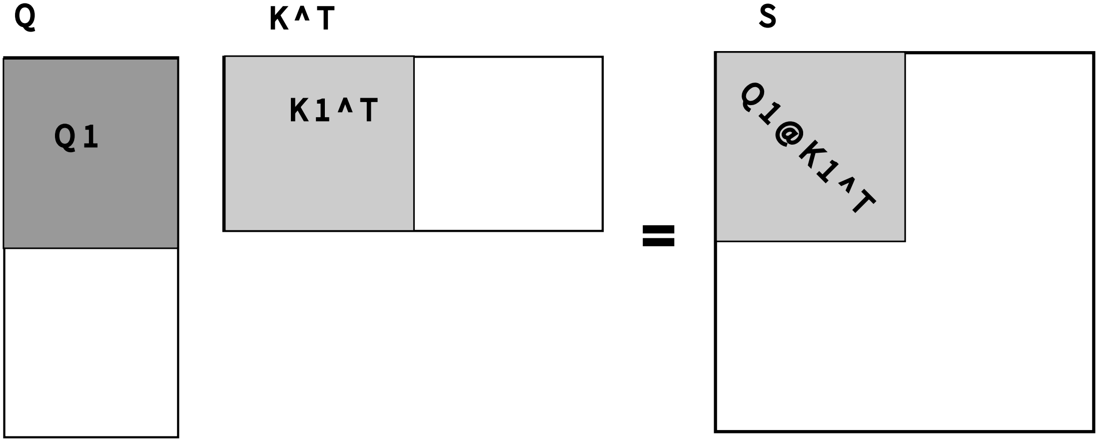
1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

From Dao et al 2022 (Flash Attention)

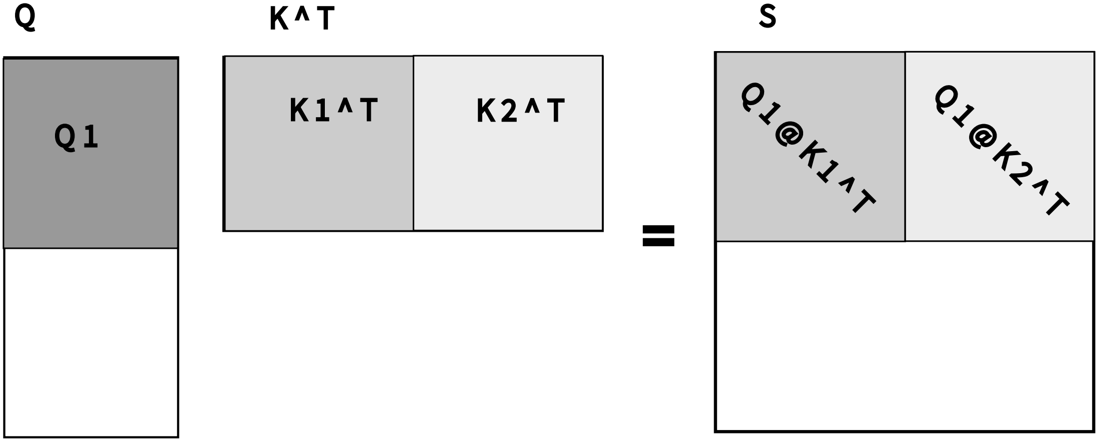
Tiling



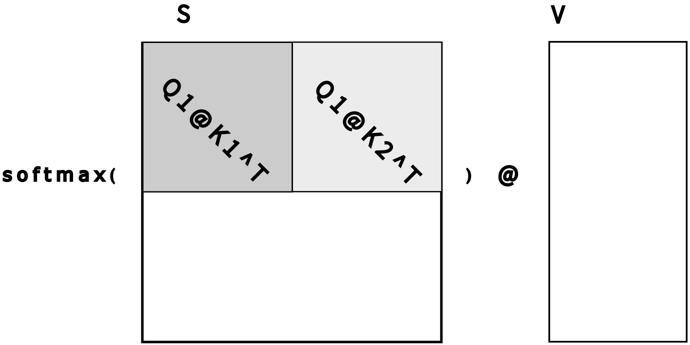
Tiling



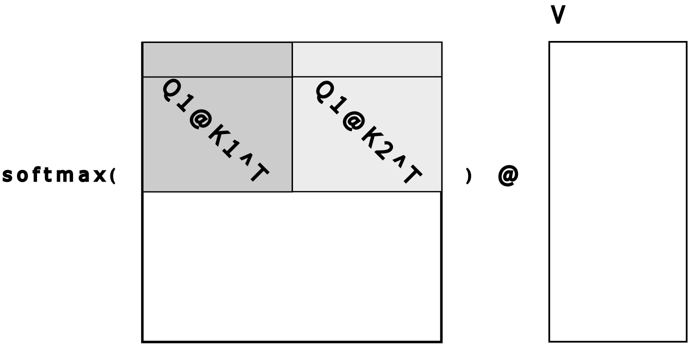
Tiling



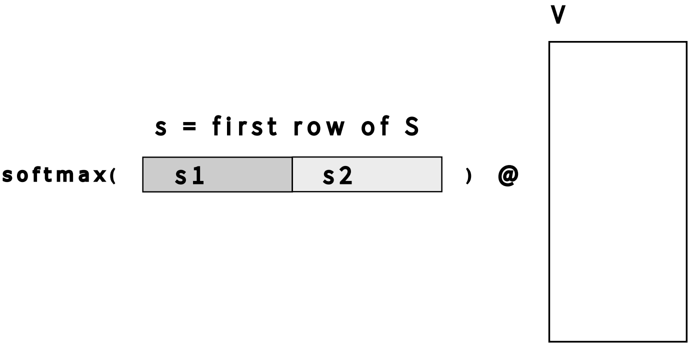
Tiling



Tiling



Online softmax



Online softmax

```
C = 100

s = np.random.rand(C)

I1 = np.arange(0, C//2)
I2 = np.arange(C//2, C)
s1 = s[I1]
s2 = s[I2]

s - np.hstack([s1,s2]) # make sure we didn't make a silly error
```

Online softmax

```python
def softmax(z):
    expz = np.exp(z)
    normalizer = np.sum(expz)
    return expz / normalizer, normalizer
```

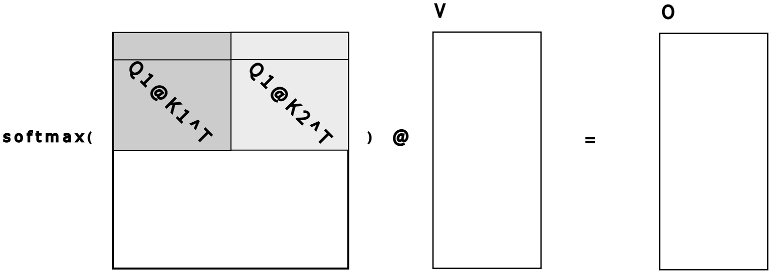
Online softmax

```
d = 3
np.random.seed(42)
V = np.random.randn(C, d)

p, n = softmax(s)
p @ V
# >>> array([ 0.13778369, -0.16034761,  0.04310764])
```

Online softmax

```python
# block 1
p1, n1 = softmax(s1)
O = p1 @ V[I1,:]

# block 2
p2, n2 = softmax(s2)
O = None # YOUR CODE HERE
O
```

Summary



- Load $\mathbf{Q}_i$, $\mathbf{K}_j$, $\mathbf{V}_j$ one block at a time
- Compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^\top$
- Compute $\mathrm{softmax}(\mathbf{S}_{ij})\mathbf{V}_j$
- Update $\mathbf{O}_j$ using online softmax

References I

- Dao, Tri, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré (2022). "FlashAttention: Fast and memory-efficient exact attention with io-awareness". In: *Advances in Neural Information Processing Systems* 35, pp. 16344–16359.
- Jain, Paras, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica (2020). "Checkmate: Breaking the memory wall with optimal tensor rematerialization". In: *Proceedings of Machine Learning and Systems* 2, pp. 497–511.
- Sohoni, Nimit S, Christopher R Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré (2019). "Low-memory neural network training: A technical report". In: *arXiv preprint arXiv:1904.10631*.