Finals Review

Lecture 13 — CS 577 Deep Learning

Instructor: Yutong Wang

Computer Science
Illinois Institute of Technology

November 20, 2024

What we've done

- Computing gradients by hand

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum
- Three key building blocks of deep learning models:

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum
- Three key building blocks of deep learning models:
  - convolution layer

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum
- Three key building blocks of deep learning models:
  - convolution layer
  - recurrent layer

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum
- Three key building blocks of deep learning models:
  - convolution layer
  - recurrent layer
  - transformer layer

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum
- Three key building blocks of deep learning models:
  - convolution layer
  - recurrent layer
  - transformer layer
- Performance issues:

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum
- Three key building blocks of deep learning models:
  - convolution layer
  - recurrent layer
  - transformer layer
- Performance issues:
  - Efficiently computing convolution ("lower" to BLAS, i.e., im2col)

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - `ag.Scalar`
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum
- Three key building blocks of deep learning models:
  - convolution layer
  - recurrent layer
  - transformer layer
- Performance issues:
  - Efficiently computing convolution ("lower" to BLAS, i.e., im2col)
  - Memory usage (tensor rematerialization/gradient checkpointing)

What we've done

- Computing gradients by hand
  - perceptron, linear regression, 1-layer neural network, transformer
- Computing gradients automatically
  - ♡ `ag.Scalar` ♡
  - `ag.Tensor`
- Two optimzers:
  - Stochastic gradient descent (SGD)
  - SGD with momentum
- Three key building blocks of deep learning models:
  - convolution layer
  - recurrent layer
  - ♡ transformer layer ♡
- Performance issues:
  - ♡ Efficiently computing convolution ("lower" to BLAS, i.e., im2col) ♡
  - ♡ Memory usage (tensor rematerialization/gradient checkpointing) ♡

Automatic differentiation, manually

- What you will be given

To ensure that everyone's answer do not vary wildly, please follow the following format!

Automatic differentiation, manually

- What you will be given
  - The code for the forward computation

To ensure that everyone's answer do not vary wildly, please follow the following format!

Automatic differentiation, manually

- What you will be given
  - The code for the forward computation
  - The (non-reversed) topological ordering

To ensure that everyone's answer do not vary wildly, please follow the following format!

Automatic differentiation, manually

- What you will be given
  - The code for the forward computation
  - The (non-reversed) topological ordering
- What you'll have to do

To ensure that everyone's answer do not vary wildly, please follow the following format!

Automatic differentiation, manually

- What you will be given
  - The code for the forward computation
  - The (non-reversed) topological ordering
- What you'll have to do
  - Draw the computational graph

To ensure that everyone's answer do not vary wildly, please follow the following format!

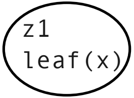
Automatic differentiation, manually

- What you will be given
  - The code for the forward computation
  - The (non-reversed) topological ordering
- What you'll have to do
  - Draw the computational graph
  - Calculate gradient at the leaves using backpropagation, iteration-by-iteration

To ensure that everyone's answer do not vary wildly, please follow the following format!

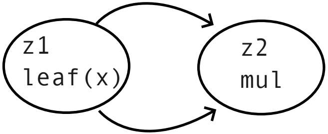
Problem 1.1 forward pass

```
x = ag.scalar(2.0, label="z1\nleaf(x)")
z1 = x
z2 = z1*z1 # z2.label = "z2:mul"
z3 = z2*z2
z3.backward()
print(x.grad)
```



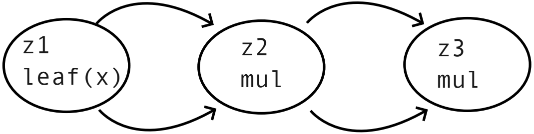
Problem 1.1 forward pass

```
x = ag.scalar(2.0, label="z1\nleaf(x)")
z1 = x
z2 = z1*z1
z3 = z2*z2
z3.backward()
print(x.grad)
```



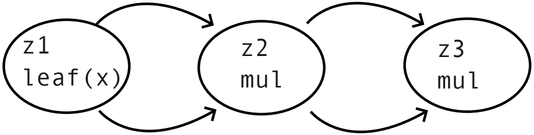
Problem 1.1 forward pass

```
x = ag.scalar(2.0, label="z1\nleaf(x)")
z1 = x
z2 = z1*z1
z3 = z2*z2
z3.backward()
print(x.grad)
```



Problem 1.1 topological ordering

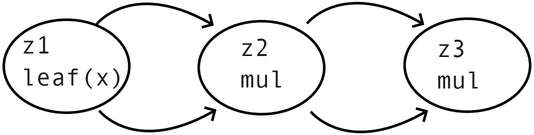
```
def topological_sort(self):
    topo_order = []
    visited = set()
    def dfs(node):
        if node not in visited:
            visited.add(node)
            for input in node.inputs:
                dfs(input)
            topo_order.append(node)
    dfs(self)
    return topo_order
```




```
visited   : set()
topo_order: []
```

Problem 1.1 topological ordering

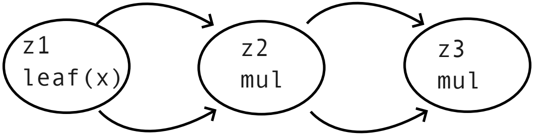
```python
def topological_sort(self):
    topo_order = []
    visited = set()
    def dfs(node):
        if node not in visited:
            visited.add(node)
            for input in node.inputs:
                dfs(input)
            topo_order.append(node)
    dfs(self)
    return topo_order
```




```
visited   : {z3:mul}
topo_order: []
```

Problem 1.1 topological ordering

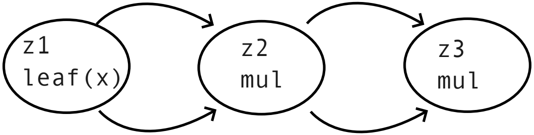
```
def topological_sort(self):
    topo_order = []
    visited = set()
    def dfs(node):
        if node not in visited:
            visited.add(node)
            for input in node.inputs:
                dfs(input)
            topo_order.append(node)
    dfs(self)
    return topo_order
```




```
visited   : {z3:mul, z2:mul}
topo_order: []
```

Problem 1.1 topological ordering

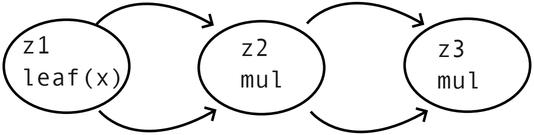
```python
def topological_sort(self):
    topo_order = []
    visited = set()
    def dfs(node):
        if node not in visited:
            visited.add(node)
            for input in node.inputs:
                dfs(input)
            topo_order.append(node)
    dfs(self)
    return topo_order
```




```
visited   : {z3:mul, z2:mul,
             z1:leaf(x)}
topo_order: [z1:leaf(x)]
```

Problem 1.1 topological ordering

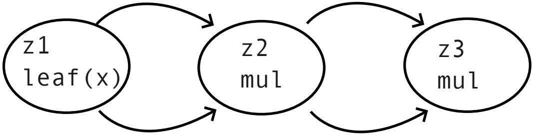
```python
def topological_sort(self):
    topo_order = []
    visited = set()
    def dfs(node):
        if node not in visited:
            visited.add(node)
            for input in node.inputs:
                dfs(input)
            topo_order.append(node)
    dfs(self)
    return topo_order
```




```
visited   : {z3:mul, z2:mul,
             z1:leaf(x)}
topo_order: [z1:leaf(x), z2:mul]
```

Problem 1.1 topological ordering

```
def topological_sort(self):
    topo_order = []
    visited = set()
    def dfs(node):
        if node not in visited:
            visited.add(node)
            for input in node.inputs:
                dfs(input)
            topo_order.append(node)
    dfs(self)
    return topo_order
```

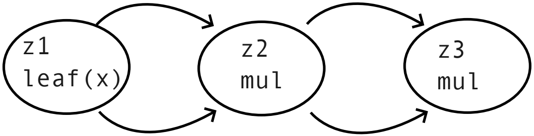



```
visited   : {z3:mul, z2:mul,
             z1:leaf(x)}
topo_order: [z1:leaf(x), z2:mul,
             z3:mul]
```

Problem 1.1 backward pass

```python
def backward(self):
    self.grad = 1.0
    topo_order = self.topological_sort()
    for node in reversed(topo_order):
        node._backward()
```

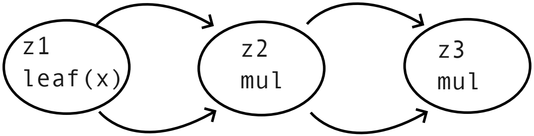
topo_order: [z1:leaf(x), z2:mul, z3:mul]



Problem 1.1 backward pass iteration 0

```python
def __mul__(self, other):
    output = ag.Scalar(self.value * other.value, inputs=[self, other])
    def _backward():
        self.grad += other.value * output.grad
        other.grad += self.value * output.grad    # [lines omitted...]
```

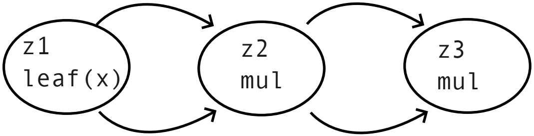
topo_order: [z1:leaf(x), z2:mul, z3:mul]



Problem 1.1 backward pass iteration 0

```python
def __mul__(self, other):
    output = ag.Scalar(self.value * other.value, inputs=[self, other])
    def _backward():
        self.grad += other.value * output.grad
        other.grad += self.value * output.grad    # [lines omitted...]
```

```
topo_order: [z1:leaf(x), z2:mul, z3:mul]
```

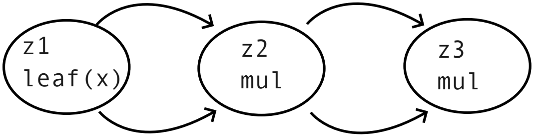



```
iteration 0:
z2.grad += z2.value * z3.grad
z2.grad += z2.value * z3.grad
```

Problem 1.1 backward pass iteration 0

```python
def __mul__(self, other):
    output = ag.Scalar(self.value * other.value, inputs=[self, other])
    def _backward():
        self.grad += other.value * output.grad
        other.grad += self.value * output.grad    # [lines omitted...]
```

topo_order: [z1:leaf(x), z2:mul, z3:mul]

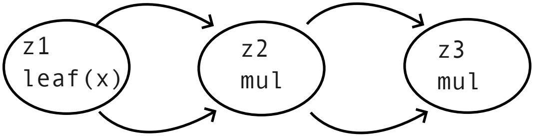


after iteration 0:
z2.grad = 8

Problem 1.1 backward pass iteration 1

```python
def __mul__(self, other):
    output = ag.Scalar(self.value * other.value, inputs=[self, other])
    def _backward():
        self.grad += other.value * output.grad
        other.grad += self.value * output.grad    # [lines omitted...]
```

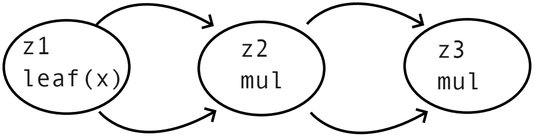
topo_order: [z1:leaf(x), z2:mul, z3:mul]



Problem 1.1 backward pass iteration 1

```python
def __mul__(self, other):
    output = ag.Scalar(self.value * other.value, inputs=[self, other])
    def _backward():
        self.grad += other.value * output.grad
        other.grad += self.value * output.grad     # [lines omitted...]
```

```
topo_order: [z1:leaf(x), z2:mul, z3:mul]
```

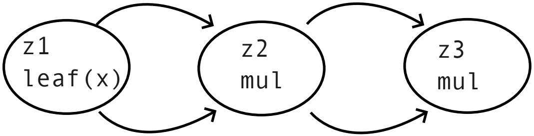



```
iteration 1:
z1.grad += z1.value * z2.grad
z1.grad += z1.value * z2.grad
```

Problem 1.1 backward pass iteration 1

```python
def __mul__(self, other):
    output = ag.Scalar(self.value * other.value, inputs=[self, other])
    def _backward():
        self.grad += other.value * output.grad
        other.grad += self.value * output.grad    # [lines omitted...]
```

```
topo_order: [z1:leaf(x), z2:mul, z3:mul]
```

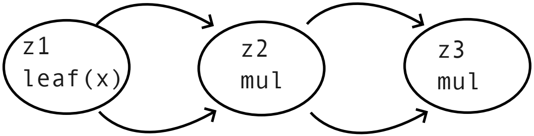



```
after iteration 1:
z1.grad = 2*8 + 2*8 = 32
```

Problem 1.1 backward pass iteration 2

```python
def __mul__(self, other):
    output = ag.Scalar(self.value * other.value, inputs=[self, other])
    def _backward():
        self.grad += other.value * output.grad
        other.grad += self.value * output.grad    # [lines omitted...]
```

topo_order: [z1:leaf(x), z2:mul, z3:mul]



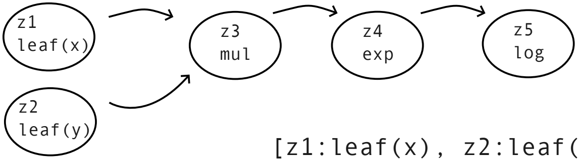
Problem 1.1a (not on the guide)

```
x = ag.Scalar(2.0, label="z1:leaf(x)")
z1 = x
z2 = z1*z1
z3 = z1*z2
z4 = z1*z3
z4.backward()
print(x.grad)
```

Problem 1.2

```
x = ag.Scalar(2.0, label="z1\nleaf(x)")
y = ag.Scalar(3.0, label="z2\nleaf(y)")

z1 = x
z2 = y
z3 = z1*z2
z4 = ag.exp(z3)
z5 = ag.log(z4)
z5.backward()
print(x.grad, y.grad)
```



[z1:leaf(x), z2:leaf(y), z3:mul, z4:exp, z5:log]

Problem 1.3

```
x1 = ag.Scalar(2.0, label="z1:leaf(x1)")
h0 = ag.Scalar(3.0, label="z2:leaf(h0)")
wr = ag.Scalar(4.0, label="z3:leaf(wr)")
wi = ag.Scalar(5.0, label="z4:leaf(wi)")
wo = ag.Scalar(6.0, label="z5:leaf(wo)")

z1 = x1
z2 = h0
z3 = wr
z4 = wi
z5 = z3*z2 # wr*h0
z6 = z4*z1 # wi*x1
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)
z9 = wo
z10 = z8*z9
z10.backward()
print(wr.grad, wi.grad, wo.grad)
```

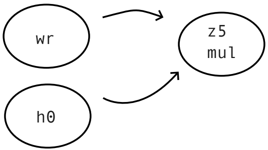
Problem 1.3

```python
x1 = ag.Scalar(2.0, label="z1:leaf(x1)")
h0 = ag.Scalar(3.0, label="z2:leaf(h0)")
wr = ag.Scalar(4.0, label="z3:leaf(wr)")
wi = ag.Scalar(5.0, label="z4:leaf(wi)")
wo = ag.Scalar(6.0, label="z5:leaf(wo)")

z1 = x1
z2 = h0
z3 = wr
z4 = wi
z5 = z3*z2 # wr*h0      <---- start here
z6 = z4*z1 # wi*x1
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)
z9 = wo
z10 = z8*z9
z10.backward()
print(wr.grad, wi.grad, wo.grad)
```

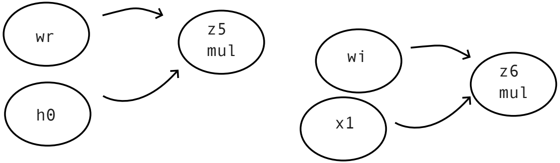
Problem 1.3

```
z5 = z3*z2 # wr*h0      <----
z6 = z4*z1 # wi*x1
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)
z9 = wo
z10 = z8*z9
```



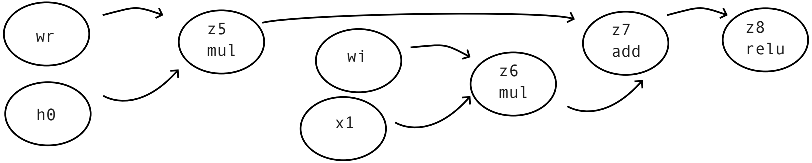
Problem 1.3

```
z5 = z3*z2 # wr*h0
z6 = z4*z1 # wi*x1      <-----
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)
z9 = wo
z10 = z8*z9
```



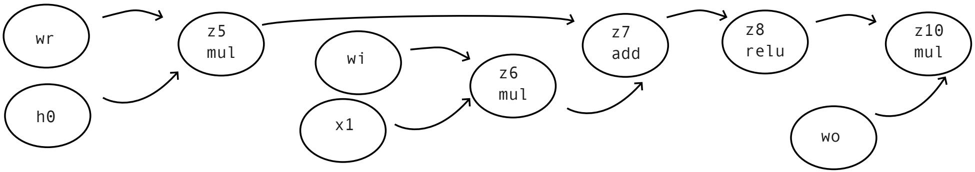
Problem 1.3

```
z5 = z3*z2 # wr*h0
z6 = z4*z1 # wi*x1
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)      <----
z9 = wo
z10 = z8*z9
```



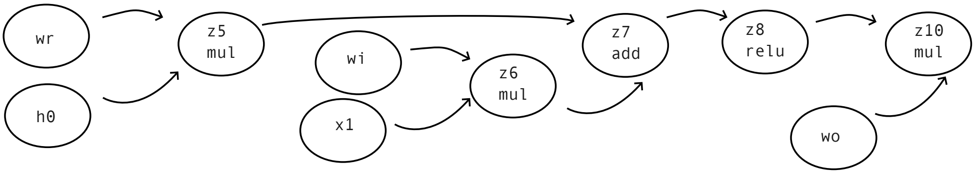
Problem 1.3

```
z5 = z3*z2 # wr*h0
z6 = z4*z1 # wi*x1
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)
z9 = wo
z10 = z8*z9      <----
```



Problem 1.3

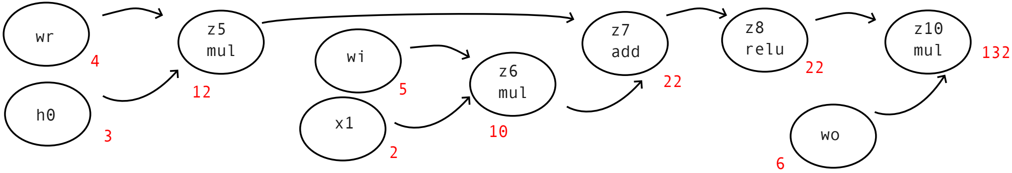
```
z5 = z3*z2 # wr*h0
z6 = z4*z1 # wi*x1
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)
z9 = wo
z10 = z8*z9      <----
```



[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

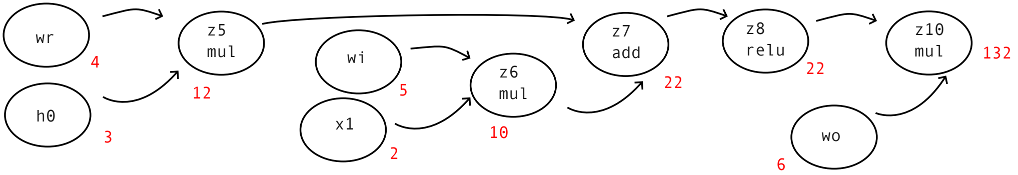
Problem 1.3

```
z5 = z3*z2 # wr*h0
z6 = z4*z1 # wi*x1
z7 = z5+z6
z8 = ag.relu(z7) # relu(wr*h0 + wi*x1)
z9 = wo
z10 = z8*z9      <----
```



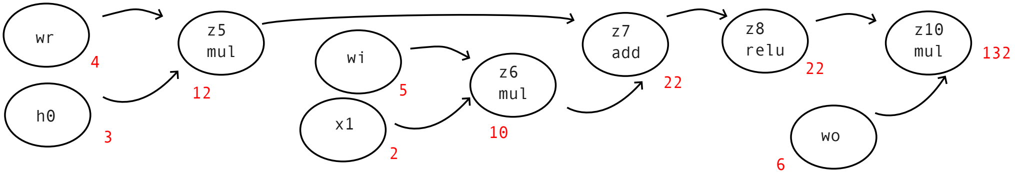
[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Problem 1.3 backward iteration 0



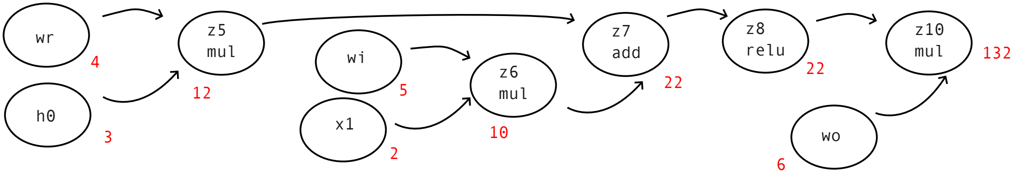
`[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]`

Problem 1.3 backward iteration 1



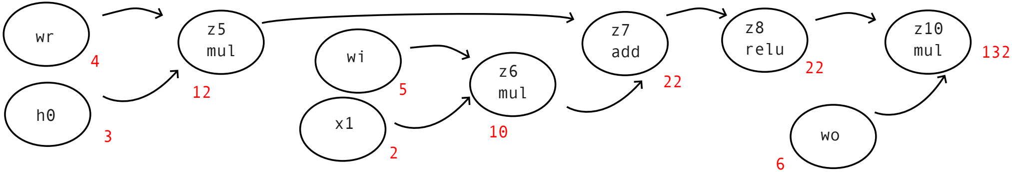
[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Problem 1.3 backward iteration 2



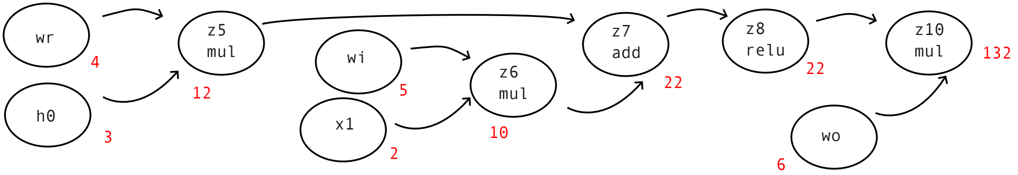
[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Problem 1.3 backward iteration 3



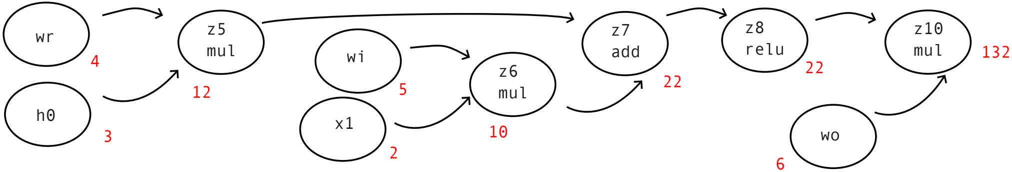
`[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]`

Problem 1.3 backward iteration 4



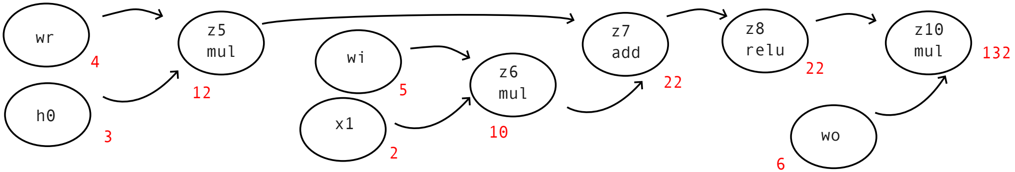
[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Problem 1.3 backward iteration 5



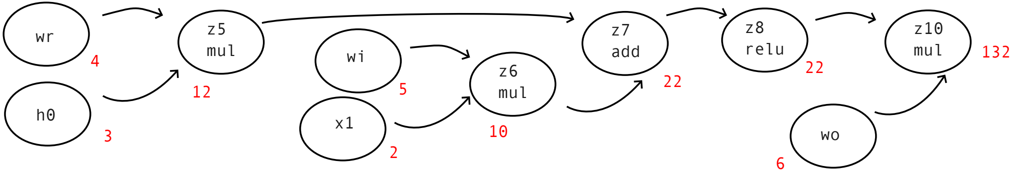
[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Problem 1.3 backward iteration 6



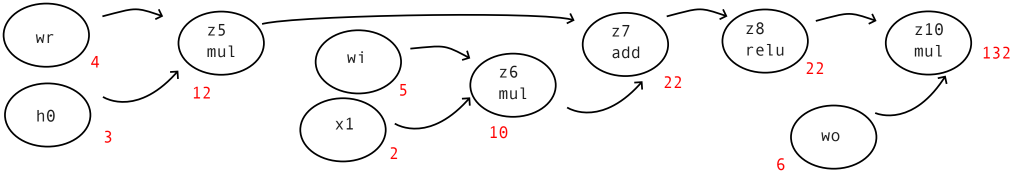
[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Problem 1.3 backward iteration 7



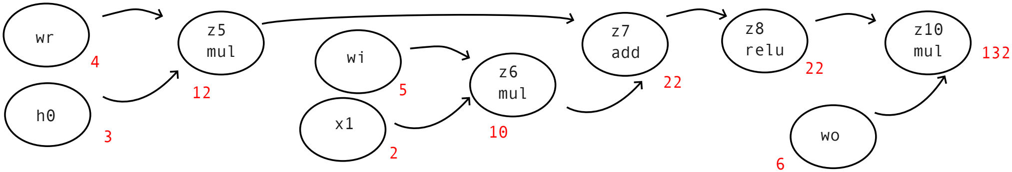
[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Problem 1.3 backward iteration 8



[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Problem 1.3 backward iteration 9



[z3:leaf(wr), z2:leaf(h0), z5:mul, z4:leaf(wi), z1:leaf(x1), z6:mul, z7:add, z8:relu, z5:leaf(wo), z10:mul]

Make sure that...

- Draw your graph from left (input) to right (output)
- Write down intermediate values
- If a node has two inputs (e.g., add or mul):
  - left input: top
  - right input: bottom
- Explain what happens during each iteration of the backward pass. Either...
  - Some computation, or
  - Do nothing (leaf node)

What you should know about convolutions

- `Xin.shape == (N, Cin, Hin, Win)`

What you should know about convolutions

- `Xin.shape == (N, Cin, Hin, Win)`
- `Xout.shape == (N, Cout, Hout, Wout)`

What you should know about convolutions

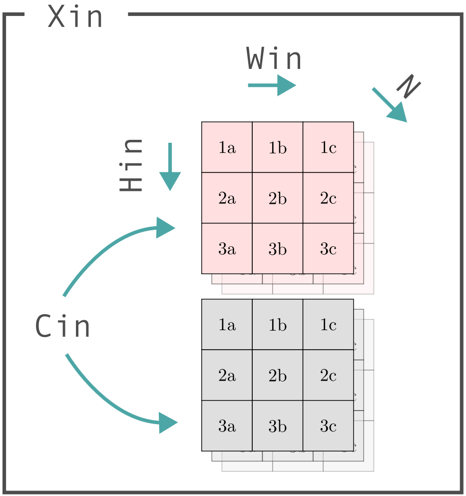
- `Xin.shape == (N, Cin, Hin, Win)`
- `Xout.shape == (N, Cout, Hout, Wout)`
- `F.shape == (Cout, Cin, K, K)`

What you should know about convolutions

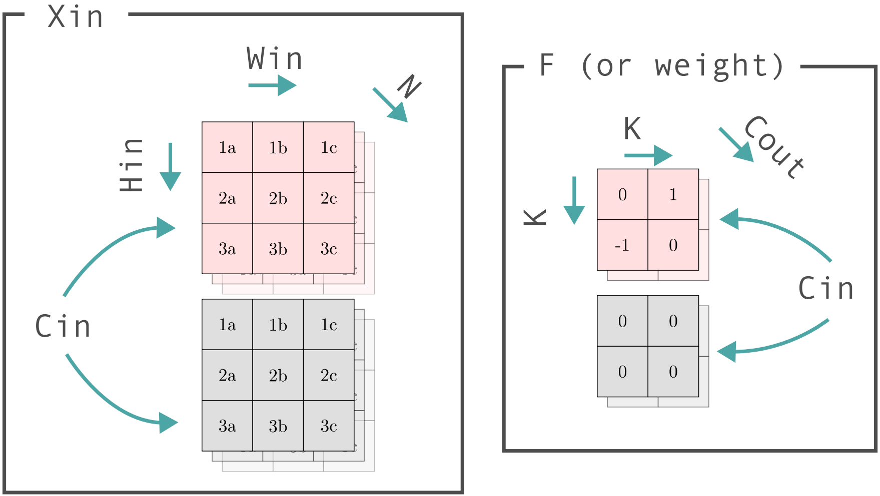
- `Xin.shape == (N, Cin, Hin, Win)`
- `Xout.shape == (N, Cout, Hout, Wout)`
- `F.shape == (Cout, Cin, K, K)`
- Convolution naively: 7 nested for-loops

```
# Suppose i, c_out, h_out, w_out are already defined, and
# X_out is initialized to all zeros
for cin in range(Cin):  # Input channels
    for hker in range(K):  # Kernel height
        for wker in range(K):  # Kernel width
            Xout[i, cout, hout, wout] += (
                F[cout, cin, hker, wker] *
                Xin[i, cin, hout + hker, wout + wker])
```

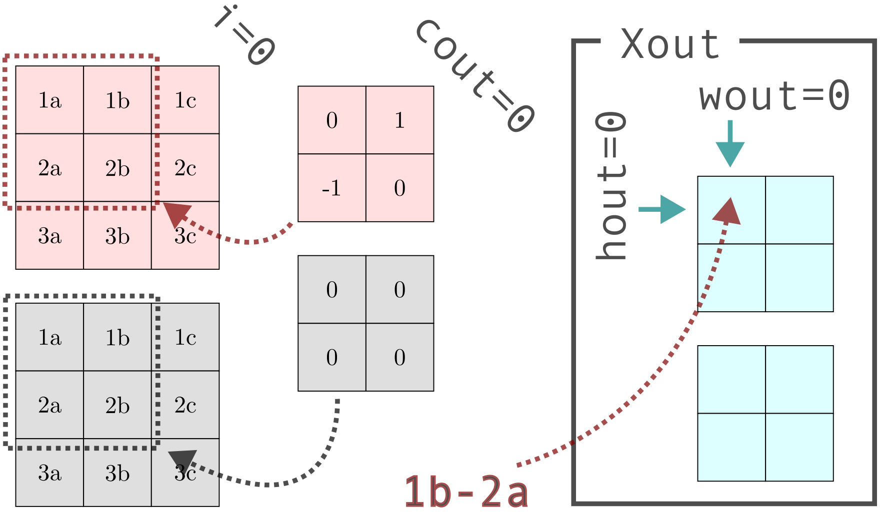
Input image tensor with shape $(N, C_{\text{in}}, H_{\text{in}}, W_{\text{in}})$



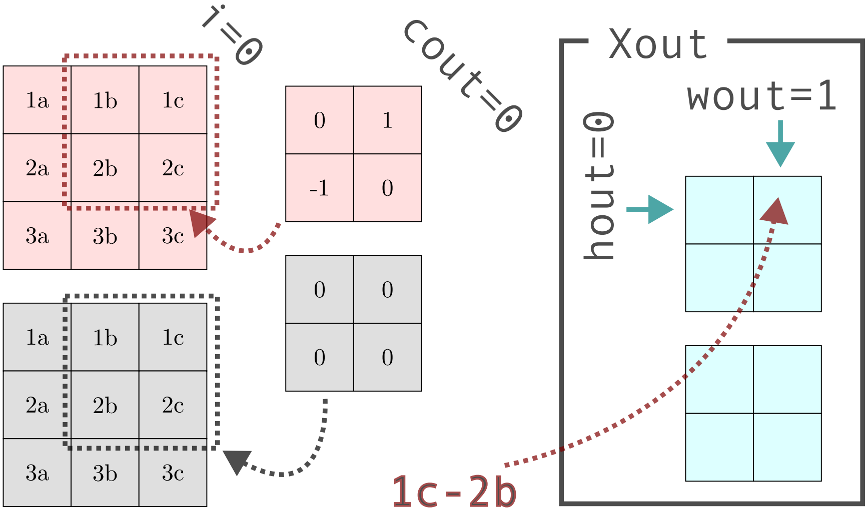
A single convolution layer $(C_{\text{out}}, C_{\text{in}}, K, K)$



patch_idx = 0

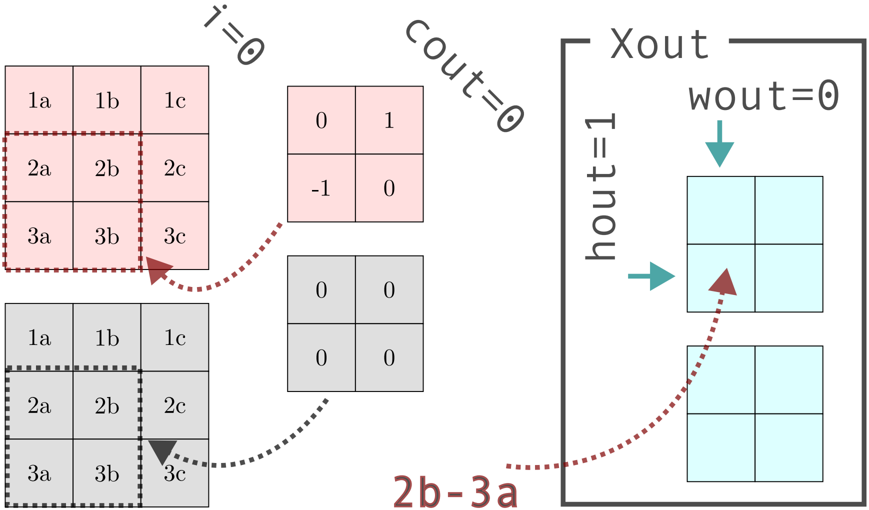

i=0
cout=0
Xout
wout=0
hout=0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
1b-2a

patch_idx = 1

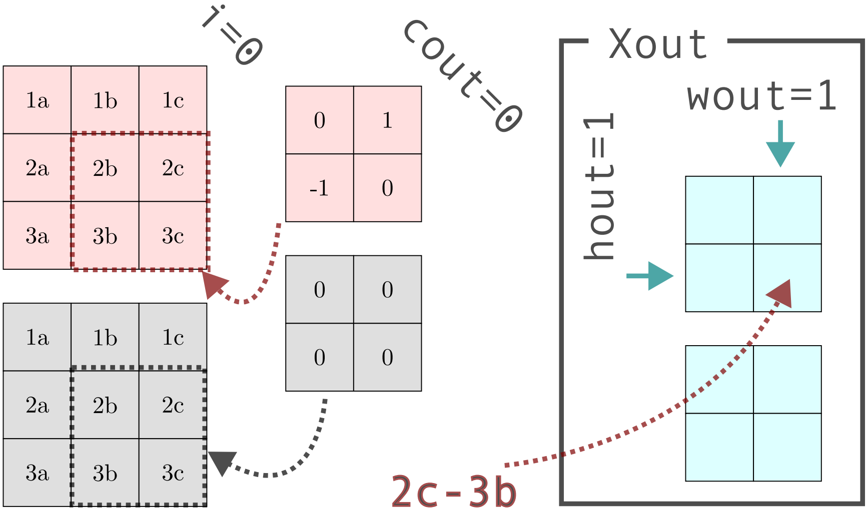


i=0
cout=0
Xout
wout=1
hout=0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
1c-2b

patch_idx = 2


i=0
cout=0
Xout
wout=0
hout=1
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
2b-3a

patch_idx = 3



i=0
cout=0
Xout
wout=1
hout=1
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
2c-3b

Flatten the weights

1a	1b	1c
2a	2b	2c
3a	3b	3c

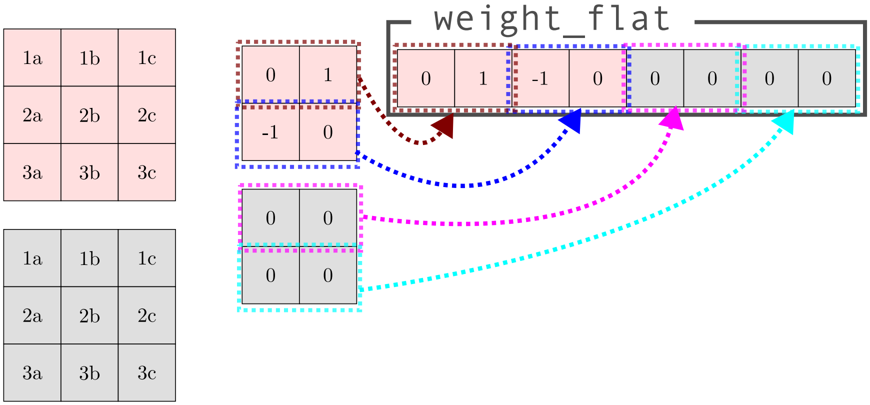
0	1
-1	0

1a	1b	1c
2a	2b	2c
3a	3b	3c

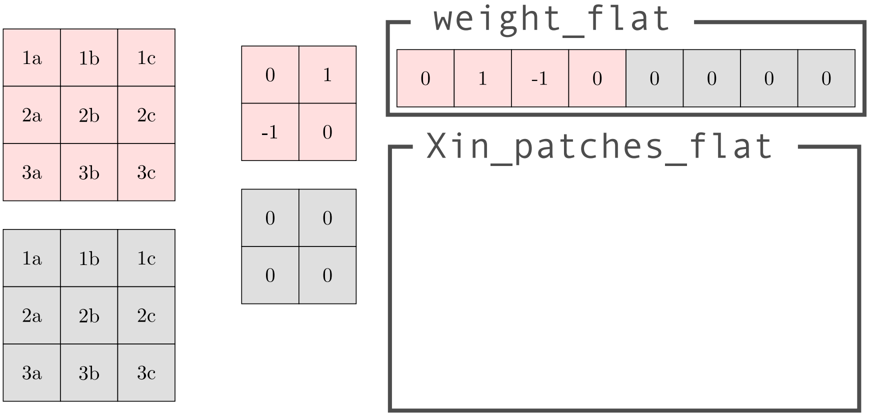
0	0
0	0

`weight_flat`

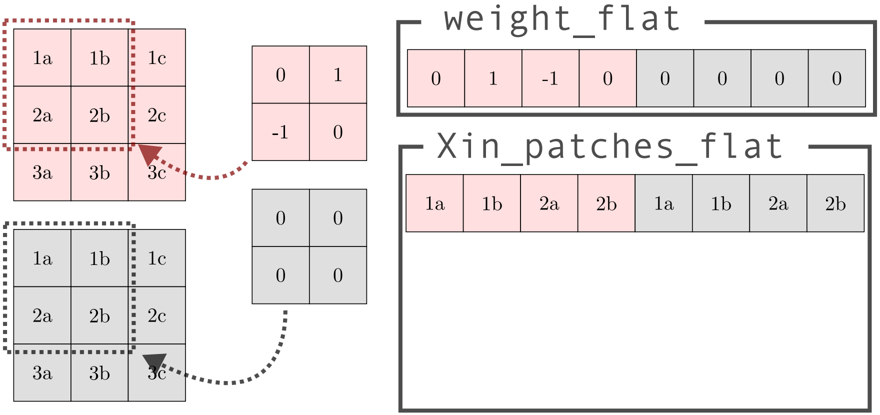
Flatten the weights



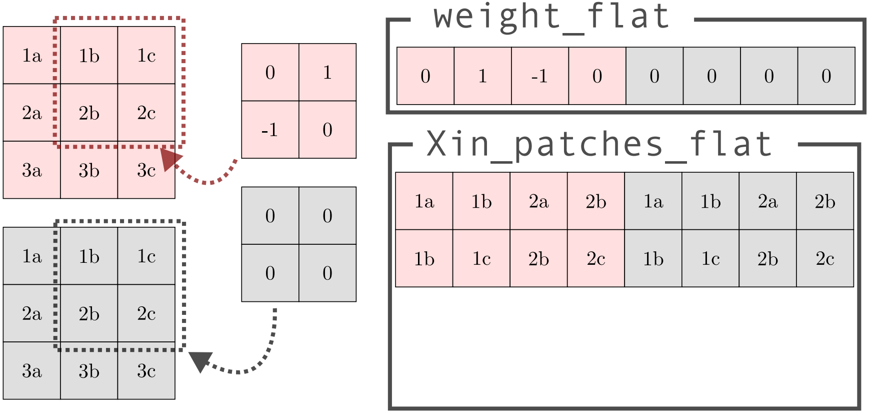
Flatten the weights



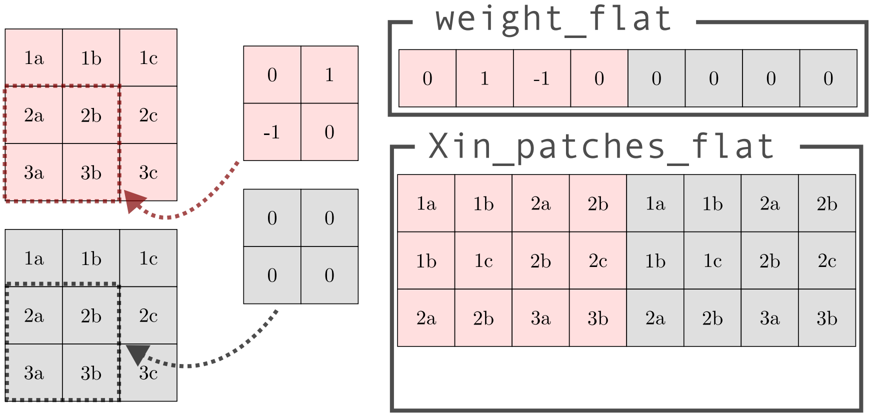
Flatten the weights



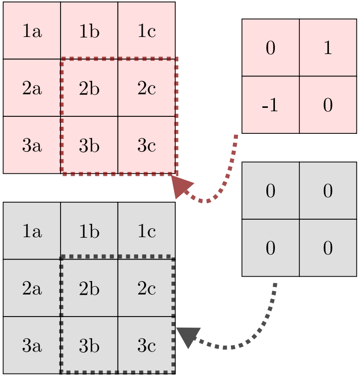
Flatten the weights



Flatten the weights



Flatten the weights



weight_flat

0	1	-1	0	0	0	0	0

Xin_patches_flat

1a	1b	2a	2b	1a	1b	2a	2b
1b	1c	2b	2c	1b	1c	2b	2c
2a	2b	3a	3b	2a	2b	3a	3b
2b	2c	3b	3c	2b	2c	3b	3c

What you should know about im2col

- im2col “lowers” convolution to linear algebra

What you should know about im2col

- im2col "lowers" convolution to linear algebra
  - `Xin`

What you should know about im2col

- im2col “lowers” convolution to linear algebra
  - `Xin`
  - `Xin_flat = Xin.reshape(-1, Cin * Hin * Win)`
    — (reshape is “free”)

What you should know about im2col

- im2col “lowers” convolution to linear algebra
  - `Xin`
  - `Xin_flat = Xin.reshape(-1, Cin * Hin * Win)`
    — (reshape is “free”)
  - `Xin_im2col = ag.spcmatmul(Xin_flat, self.im2col_mat)`
    — (“memory expensive”)

What you should know about im2col

- im2col "lowers" convolution to linear algebra
  - `Xin`
  - `Xin_flat = Xin.reshape(-1, Cin * Hin * Win)`
    — (reshape is "free")
  - `Xin_im2col = ag.spcmatmul(Xin_flat, self.im2col_mat)`
    — ("memory expensive")
  - `Xin_patches_flat = Xin_im2col.reshape(N, P, patch_size)`
    — ("free")

What you should know about im2col

- im2col "lowers" convolution to linear algebra
  - `Xin`
  - `Xin_flat = Xin.reshape(-1, Cin * Hin * Win)`
    — (reshape is "free")
  - `Xin_im2col = ag.spcmatmul(Xin_flat, self.im2col_mat)`
    — ("memory expensive")
  - `Xin_patches_flat = Xin_im2col.reshape(N, P, patch_size)`
    — ("free")
  - `Xout_flat = (Xin_patches_flat @ self.weight) + self.bias`
    — ("compute expensive")

What you should know about im2col

- im2col "lowers" convolution to linear algebra
  - `Xin`
  - `Xin_flat = Xin.reshape(-1, Cin * Hin * Win)`
    — (reshape is "free")
  - `Xin_im2col = ag.spcmatmul(Xin_flat, self.im2col_mat)`
    — ("memory expensive")
  - `Xin_patches_flat = Xin_im2col.reshape(N, P, patch_size)`
    — ("free")
  - `Xout_flat = (Xin_patches_flat @ self.weight) + self.bias`
    — ("compute expensive")
  - `Xout_flat = ag.moveaxis(Xout_flat, 1, 2)`
    — ("free")

What you should know about im2col

- im2col "lowers" convolution to linear algebra
  - `Xin`
  - `Xin_flat = Xin.reshape(-1, Cin * Hin * Win)`
    — (reshape is "free")
  - `Xin_im2col = ag.spcmatmul(Xin_flat, self.im2col_mat)`
    — ("memory expensive")
  - `Xin_patches_flat = Xin_im2col.reshape(N, P, patch_size)`
    — ("free")
  - `Xout_flat = (Xin_patches_flat @ self.weight) + self.bias`
    — ("compute expensive")
  - `Xout_flat = ag.moveaxis(Xout_flat, 1, 2)`
    — ("free")
  - `Xout = Xout_flat.reshape(N, Cout, Hout, Wout)`
    — ("free")

What you should know about im2col

- im2col "lowers" convolution to linear algebra
  - `Xin`
  - `Xin_flat = Xin.reshape(-1, Cin * Hin * Win)`
    — (reshape is "free")
  - `Xin_im2col = ag.spcmatmul(Xin_flat, self.im2col_mat)`
    — ("memory expensive")
  - `Xin_patches_flat = Xin_im2col.reshape(N, P, patch_size)`
    — ("free")
  - `Xout_flat = (Xin_patches_flat @ self.weight) + self.bias`
    — ("compute expensive")
  - `Xout_flat = ag.moveaxis(Xout_flat, 1, 2)`
    — ("free")
  - `Xout = Xout_flat.reshape(N, Cout, Hout, Wout)`
    — ("free")
- `self.im2col_mat.shape =`

What you should know about im2col

- im2col "lowers" convolution to linear algebra
  - `Xin`
  - `Xin_flat = Xin.reshape(-1, Cin * Hin * Win)`
    — (reshape is "free")
  - `Xin_im2col = ag.spcmatmul(Xin_flat, self.im2col_mat)`
    — ("memory expensive")
  - `Xin_patches_flat = Xin_im2col.reshape(N, P, patch_size)`
    — ("free")
  - `Xout_flat = (Xin_patches_flat @ self.weight) + self.bias`
    — ("compute expensive")
  - `Xout_flat = ag.moveaxis(Xout_flat, 1, 2)`
    — ("free")
  - `Xout = Xout_flat.reshape(N, Cout, Hout, Wout)`
    — ("free")
- `self.weight.shape =`
- Cost of computing `Xout_flat =`

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)

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

- 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)

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

- 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)

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

- 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)

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

- parameters

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

- ("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)

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

- 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 $\texttt{attention}(\mathbf{X};\theta)$
- `Queries`, `Keys`, `KQ`, `expKQ`, `softmaxKQ`, `P`.

Self-attention

```
class SingleHeadAttention:
    def __init__(self, n_features):
        self.Wq = ag.Tensor(np.random.randn(n_features, n_features),
   label="Wq")
        self.Wk = ag.Tensor(np.random.randn(n_features, n_features),
   label="Wk")
        self.Wv = ag.Tensor(np.random.randn(n_features, n_features),
   label="Wv")
    def __call__(self, Xin):
        # Xin is a (n_samples, n_context, n_features) tensor
        # Xout is *also* a (n_samples, n_context, n_features) tensor
        Queries = Xin @ self.Wq
        Keys = Xin @ self.Wk
        KQ = (Keys @ ag.moveaxis(Queries, 1,2))
        expKQ = ag.exp(KQ)
        softmaxKQ = expKQ / ag.sum(expKQ, axis=1, keepdims=True)
        P = ag.moveaxis(Xin,1,2) @ softmaxKQ
        Xout = ag.moveaxis(P, 1,2) @ self.Wv
        return Xout
```

Self-attention

```python
class TransformerBlock:
    def __init__(self, n_features, n_hidden):
        self.att = SingleHeadAttention(n_features)
        self.mlp = MLP(n_features, n_hidden)
    def __call__(self, Xin):
        return self.mlp(self.att(Xin))
```

Self-attention

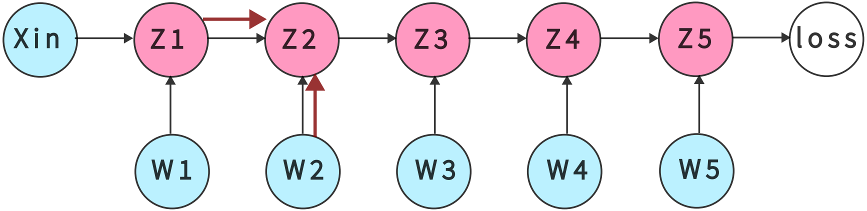
```
class MLP:
    def __init__(self, n_features, n_hidden):
        self.Wh = ag.Tensor(np.random.randn(n_features, n_hidden), label
    ="Whidden")
        self.bh = ag.Tensor(np.random.randn(n_hidden), label="bhidden")
        self.wo = ag.Tensor(np.random.randn(n_hidden, n_features), label
    ="Wout")
        self.bo = ag.Tensor(np.random.randn(n_features), label="bout")

    def __call__(self, Xin):
        hidden = ag.relu((Xin @ self.Wh) + self.bh)
        return hidden @ self.wo + self.bo
```

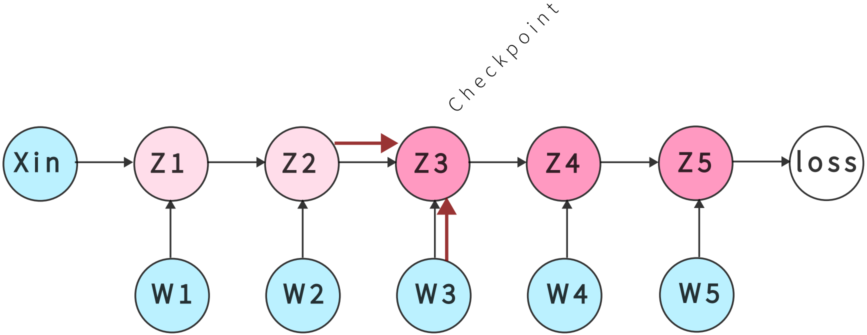
Forward without rematerialization

(Inside constructor for Z2)...

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



Rematerialization: forward phase



What we didn't cover?

- Architecture design

What we didn't cover?

- Architecture design
    - Skip connection

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization
- Optimizer design

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization
- Optimizer design
  - Pruning

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization
- Optimizer design
  - Pruning
  - Adam

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization
- Optimizer design
  - Pruning
  - Adam
  - Shampoo

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization
- Optimizer design
  - Pruning
  - Adam
  - Shampoo
  - Soap (ShampoO with Adam in the Preconditioner's) Vya et. al., 2024

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization
- Optimizer design
  - Pruning
  - Adam
  - Shampoo
  - Soap (ShampoO with Adam in the Preconditioner's) Vya et. al., 2024
- Theory of deep learning

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization
- Optimizer design
  - Pruning
  - Adam
  - Shampoo
  - Soap (ShampoO with Adam in the Preconditioner's) Vya et. al., 2024
- Theory of deep learning
  - Implicit regularization

What we didn't cover?

- Architecture design
  - Skip connection
  - U-net
  - Batch normalization
  - Quantization
- Optimizer design
  - Pruning
  - Adam
  - Shampoo
  - Soap (ShampoO with Adam in the Preconditioner's) Vya et. al., 2024
- Theory of deep learning
  - Implicit regularization
  - Benign overfitting

References I