More convolutional nets

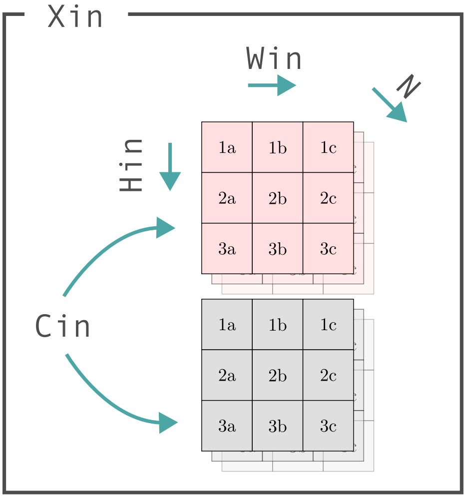
Lecture 09 — CS 577 Deep Learning

Instructor: Yutong Wang

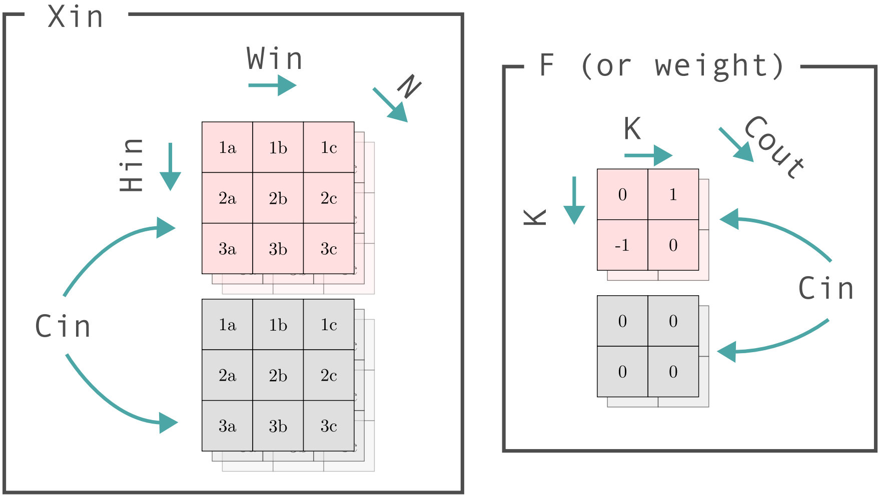
Computer Science
Illinois Institute of Technology

October 16, 2024

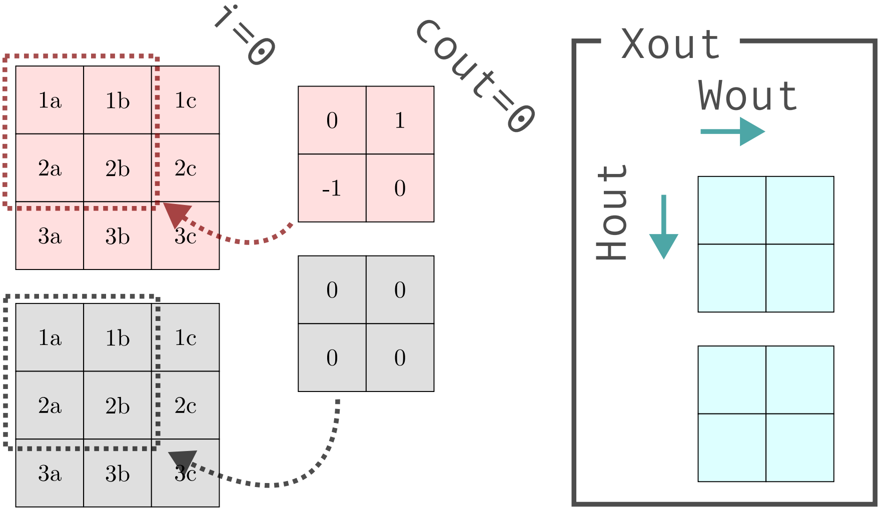
Input image tensor with shape $(N, C_{\text{in}}, H_{\text{in}}, W_{\text{in}})$



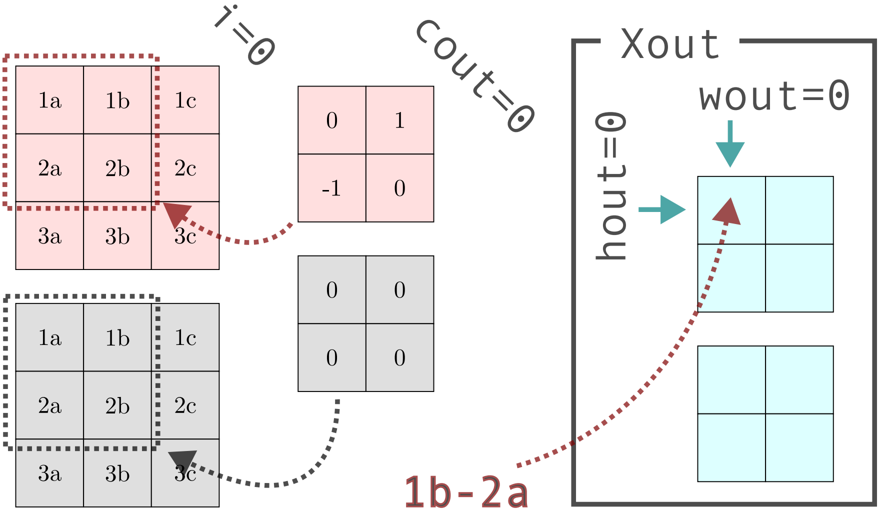
A single convolution layer $(C_{\text{out}}, C_{\text{in}}, K, K)$



Output: `Xout` with shape $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$

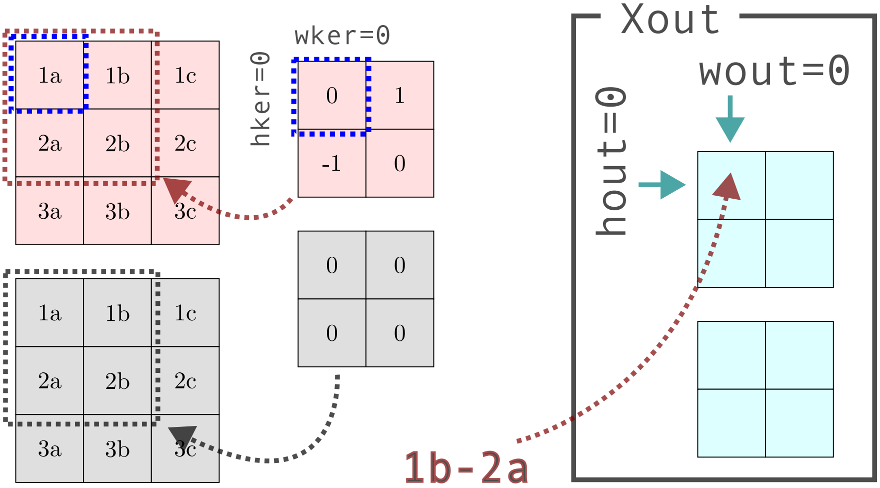


patch_idx = 0



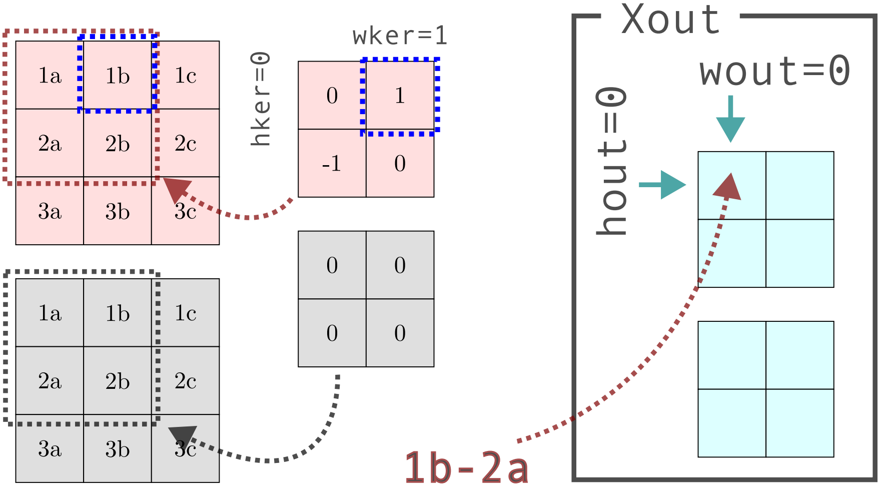
i=0
cout=0
Xout
wout=0
hout=0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
1b-2a

patch_idx = 0

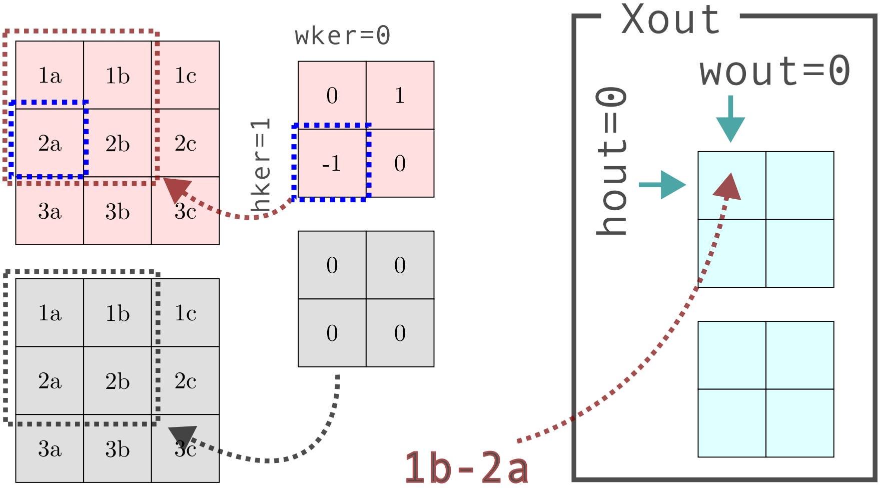


1a
1b
1c
2a
2b
2c
3a
3b
3c
wker=0
hker=0
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
Xout
wout=0
hout=0
1b-2a

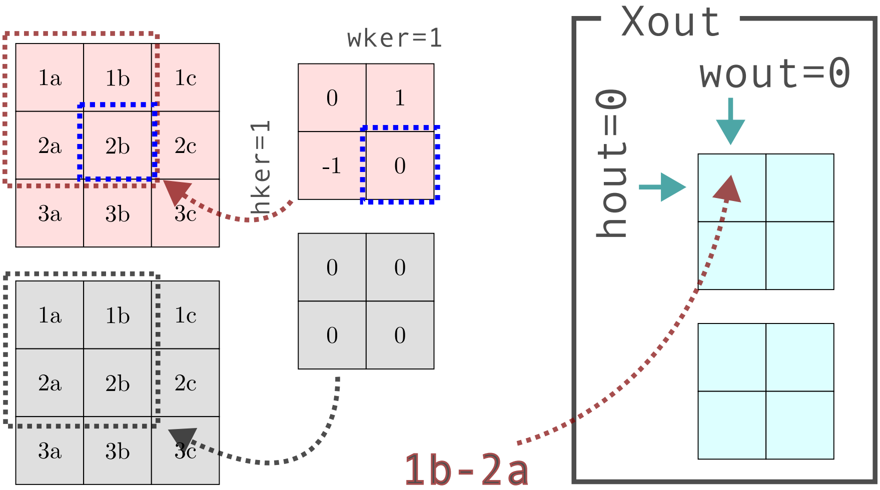
patch_idx = 0


1a
1b
1c
2a
2b
2c
3a
3b
3c
hker=0
wker=1
0
1
-1
0
0
0
0
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
Xout
wout=0
hout=0
1b-2a

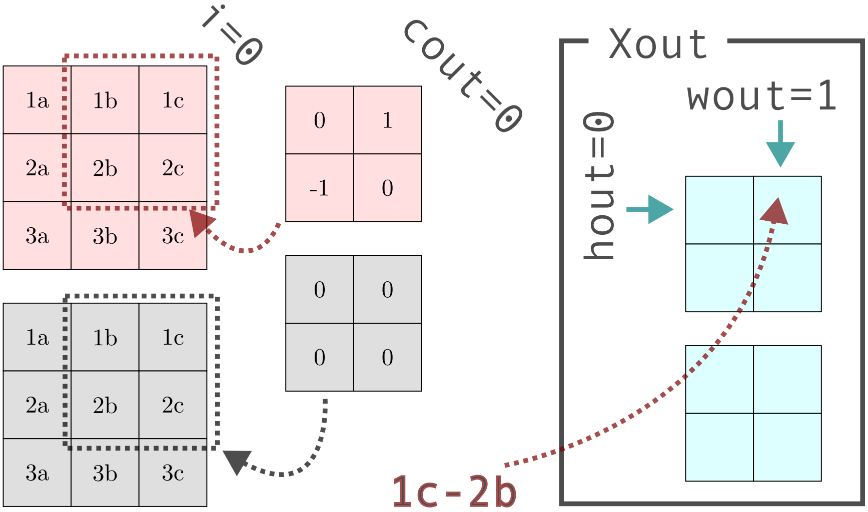
patch_idx = 0


1a
1b
1c
2a
2b
2c
3a
3b
3c
hker=1
wker=0
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
Xout
wout=0
hout=0
1b-2a

patch_idx = 0

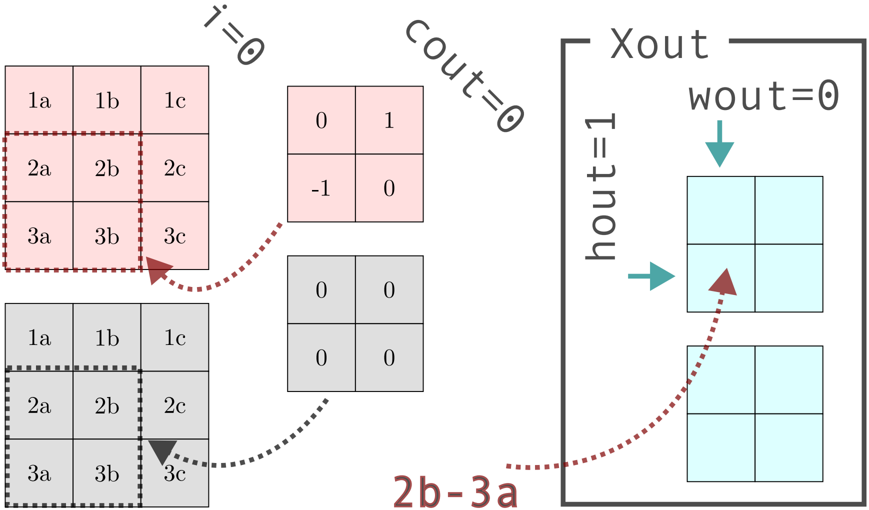

wker=1
hker=1
Xout
wout=0
hout=0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
1
-1
0
0
0
0
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
1b-2a

patch_idx = 1



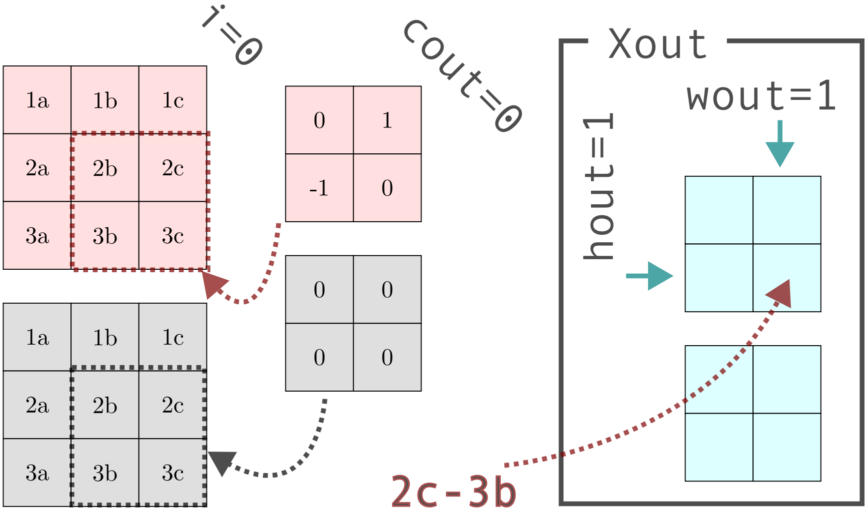
i=0
cout=0
Xout
wout=1
hout=0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
1c-2b

patch_idx = 2



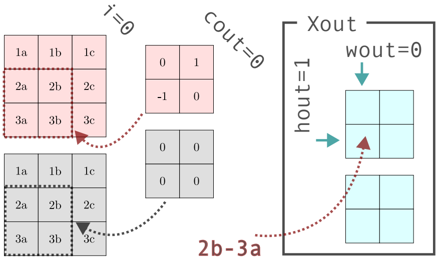
i=0
cout=0
Xout
wout=0
hout=1
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
1
-1
0
1a
1b
1c
2a
2b
2c
3a
3b
3c
0
0
0
0
2b-3a

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

Sliding window (simple but slow)




```python
# 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])
```

There's gotta be a better way! "`im2col`" (Next)

"im2col"

Main idea (in theory)

- Keep the window fixed in one spot
- Move the image tensor

Main idea (in theory)

- Create patches of the data
- Each patch has the same shape as the convolution filter
- Do matrix multiplication between the (flattened) patches and the (flattened) convolutional filter weights

Flatten the weights

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

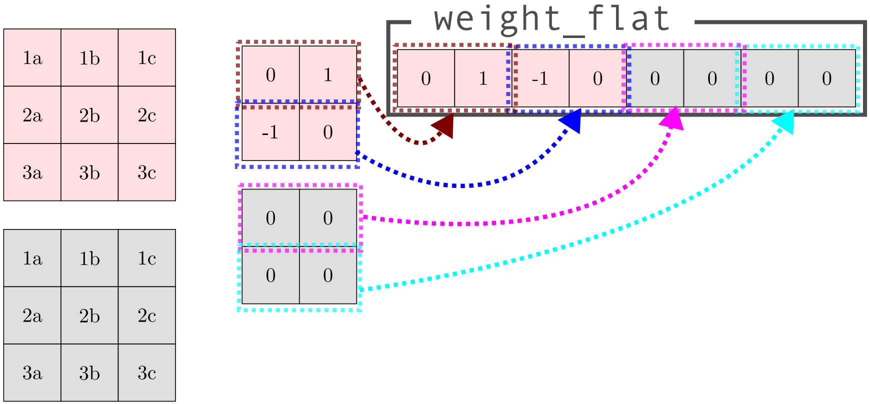
0	1
-1	0

`weight_flat`

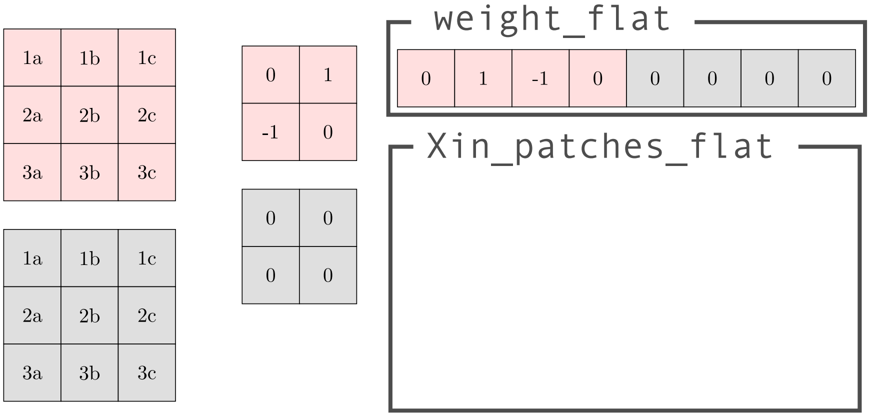
1a	1b	1c
2a	2b	2c
3a	3b	3c

0	0
0	0

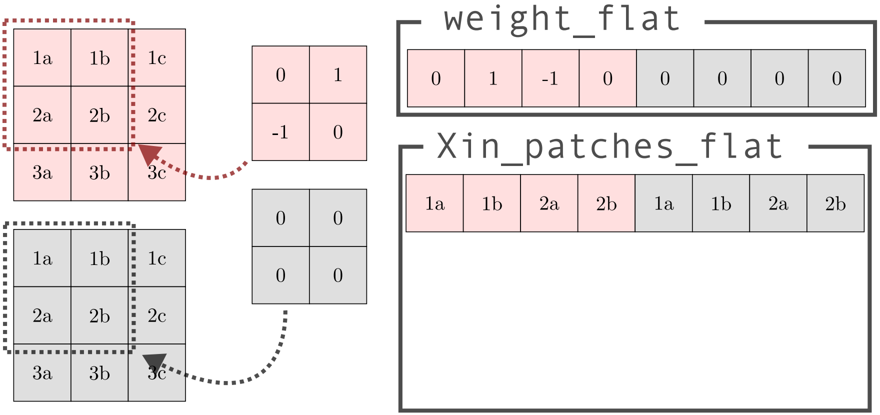
Flatten the weights



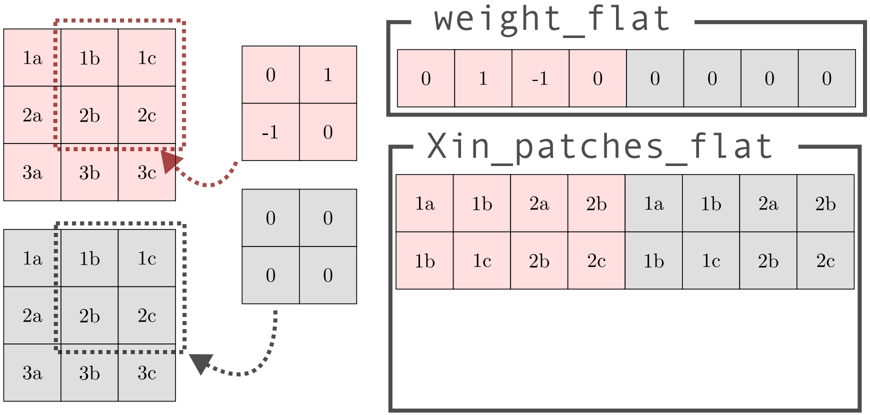
Flatten the weights



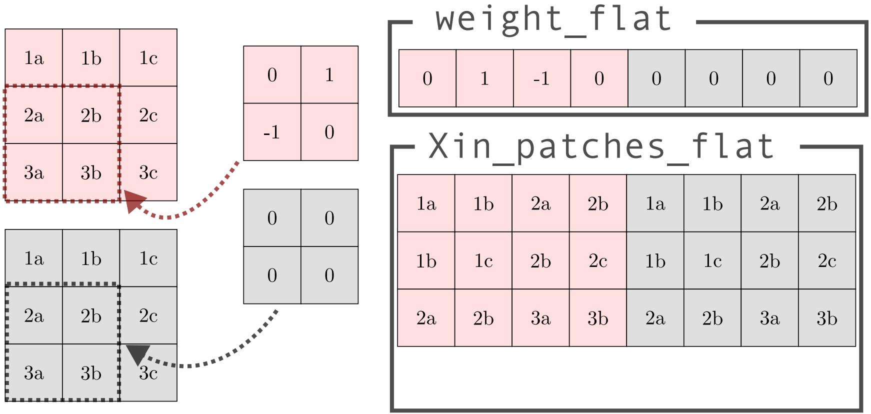
Flatten the weights



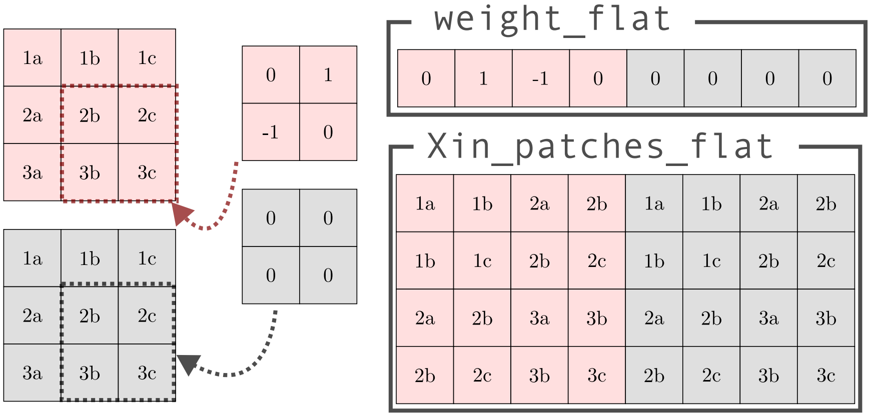
Flatten the weights



Flatten the weights



Flatten the weights



Sliding window via matmul

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

@

weight_flat

0
1
-1
0
0
0
0
0

matmul

Sliding window via matmul

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

@

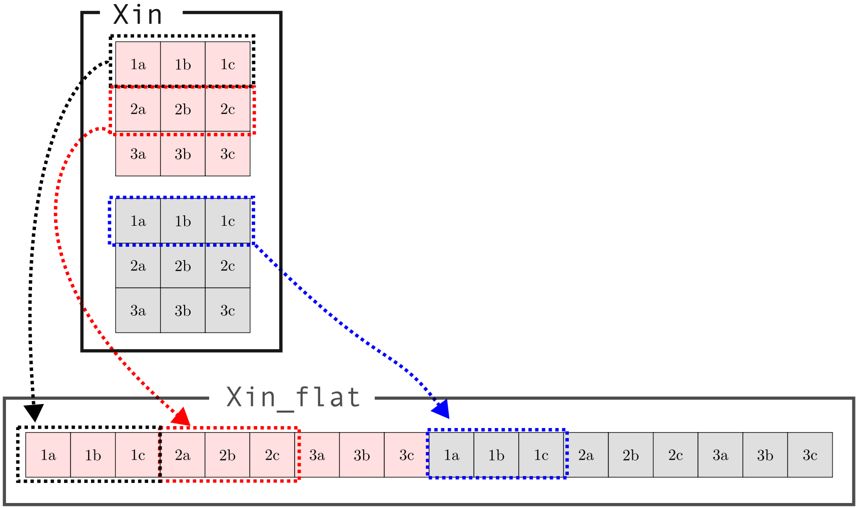
weight_flat

0	0
1	0
-1	0
0	0
0	1
0	0
0	-1
0	0

matmul

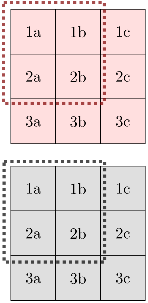
Cout

Flattening `Xin` into `Xin_flat`

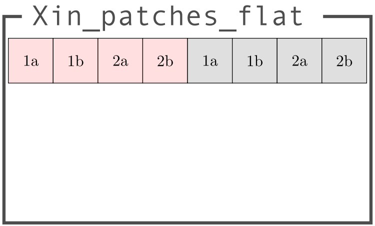


Flattening `Xin` into `Xin_flat`

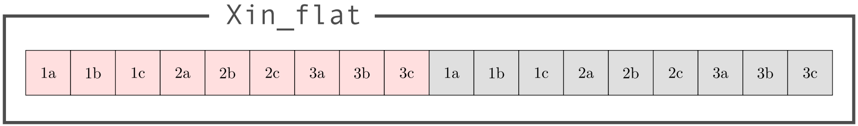
`Recall patch_idx=0:`



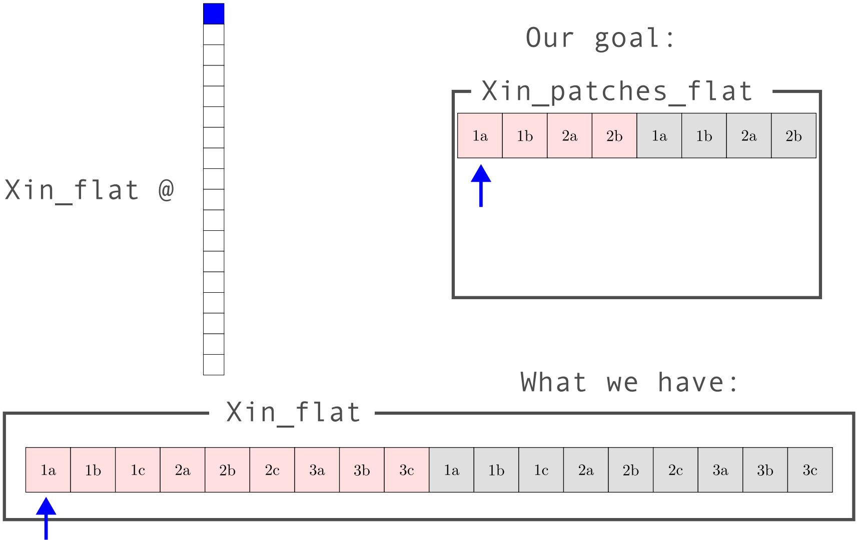
`Our goal:`



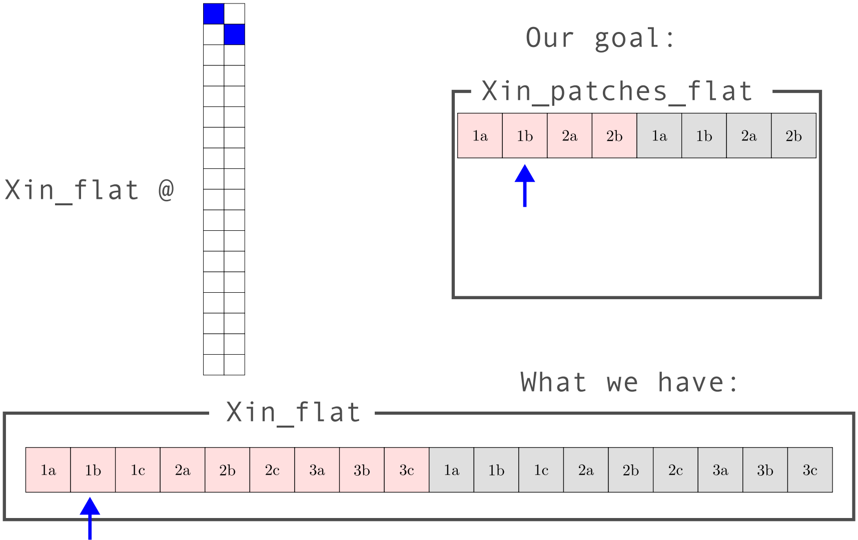
`What we have:`



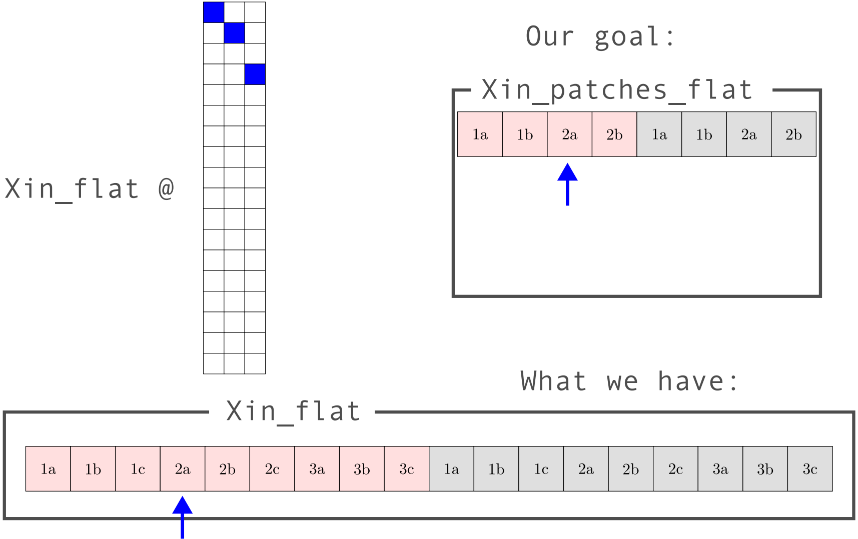
The “im2col” matrix



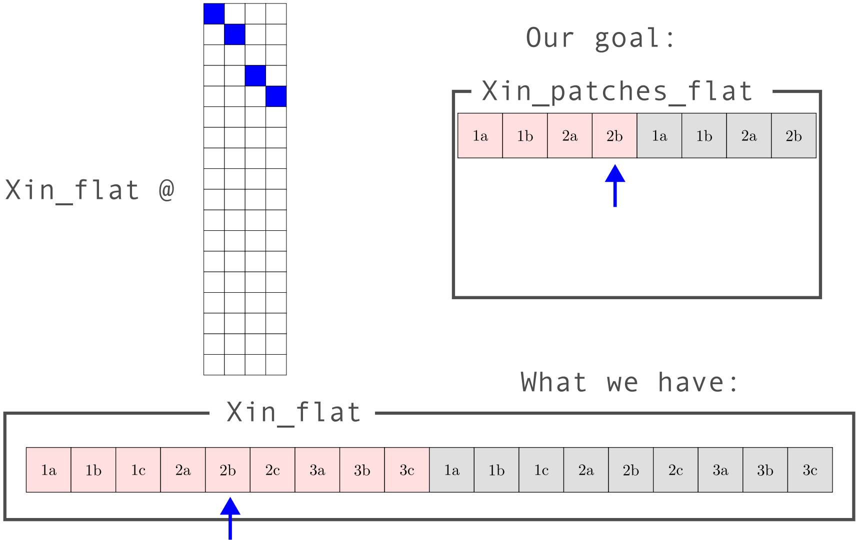
The "im2col" matrix



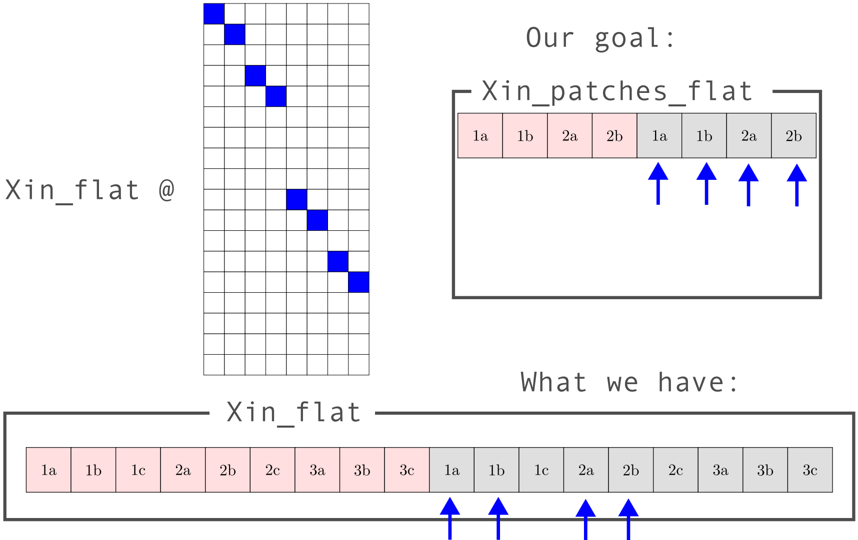
The “im2col” matrix



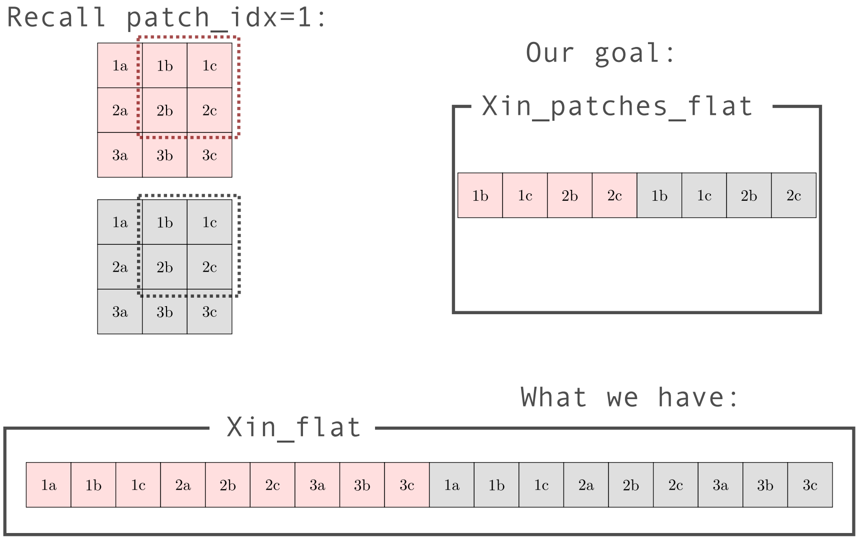
The “im2col” matrix



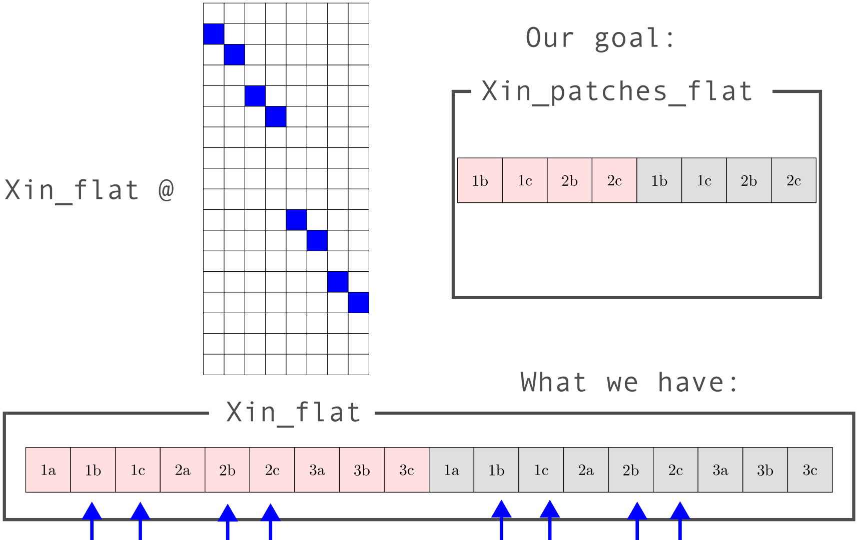
The “im2col” matrix



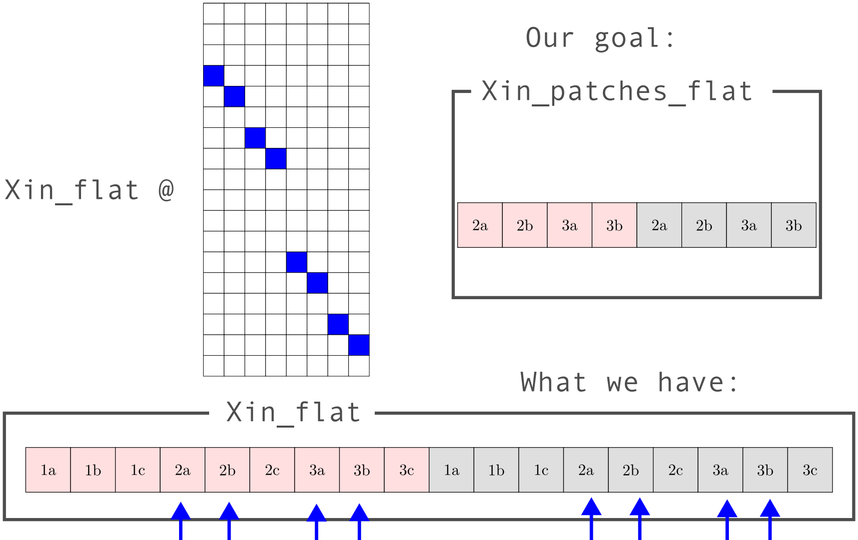
The "im2col" matrix



The “im2col” matrix

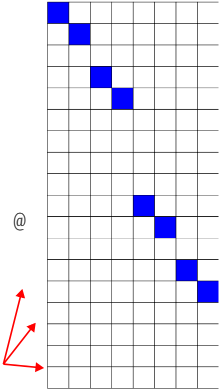


The “im2col” matrix



The “im2col” matrix

Xin_flat @



im2col

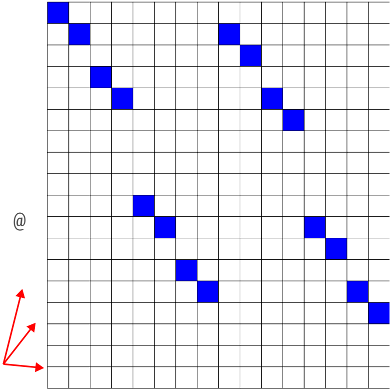
=

Xin_im2col

1a	1b	2a	2b	1a	1b	2a	2b

The "im2col" matrix

Xin_flat @



im2col

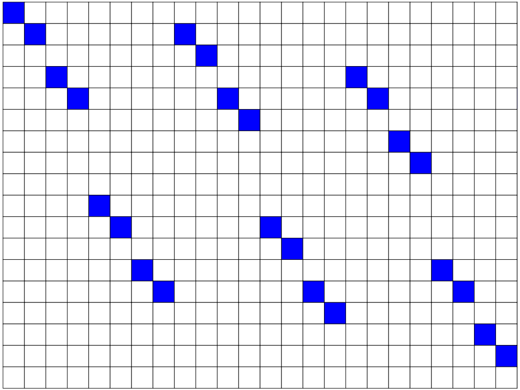
=

Xin_im2col

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

The “im2col” matrix

Xin_flat @



im2col

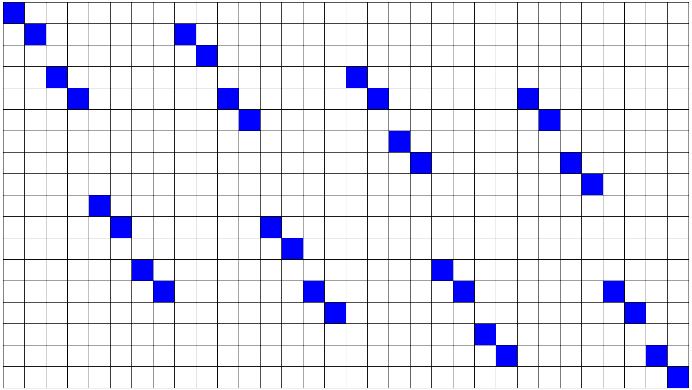
=

Xin_im2col

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

The “im2col” matrix

Xin_flat @



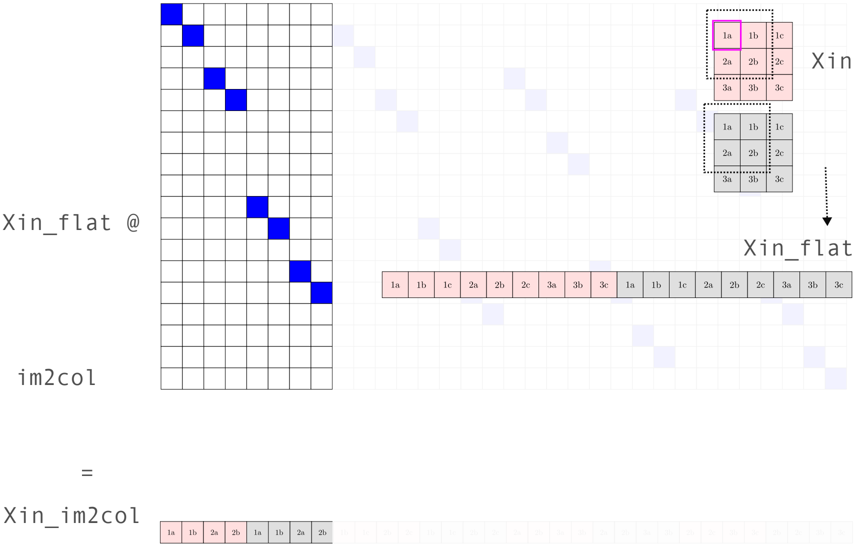
im2col

=

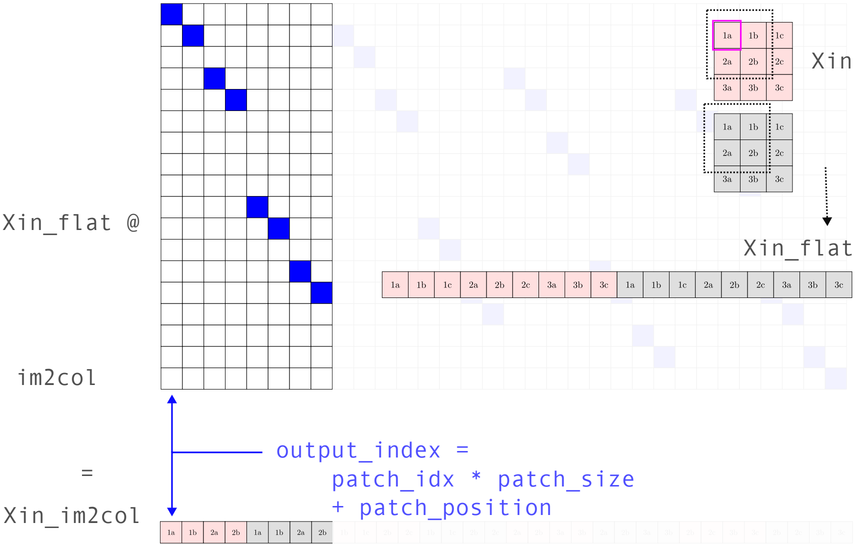
Xin_im2col

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

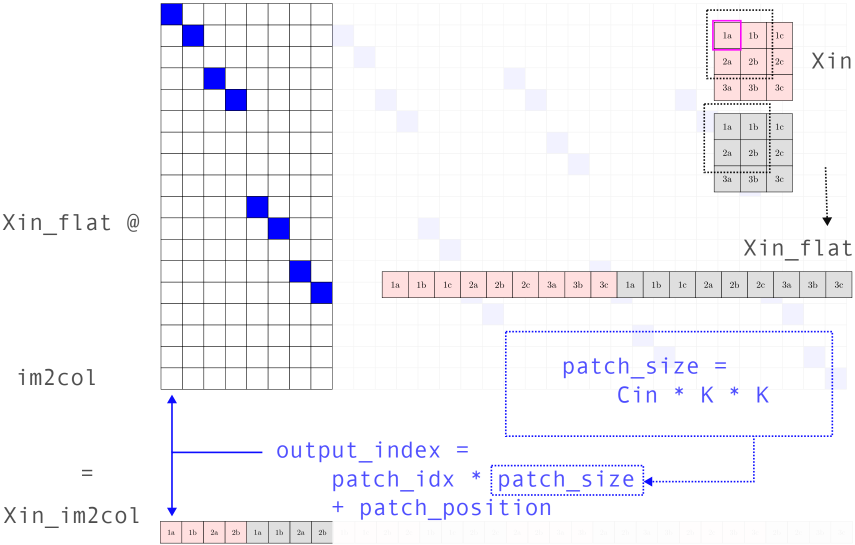
The “im2col” matrix



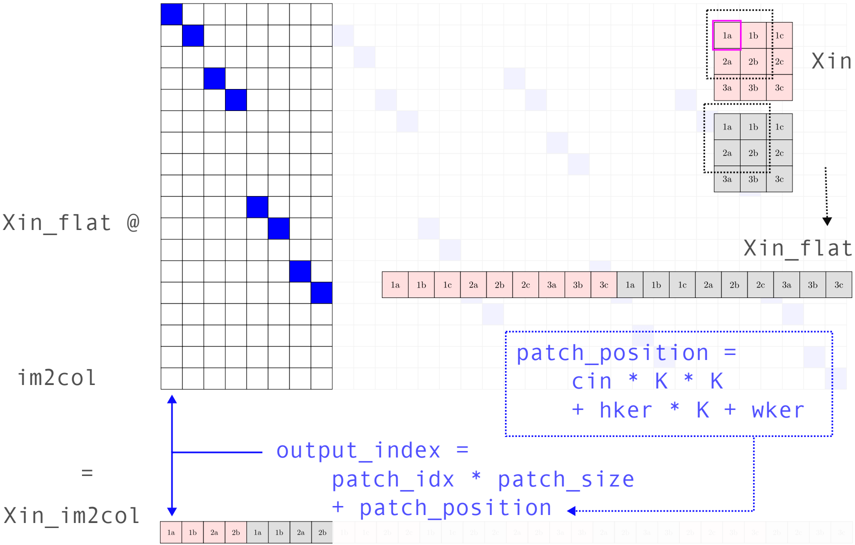
The "im2col" matrix



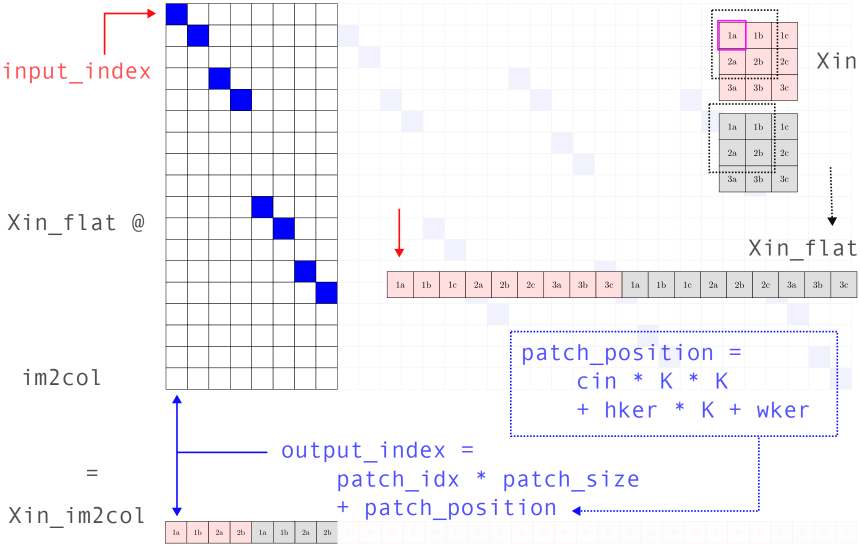
The "im2col" matrix



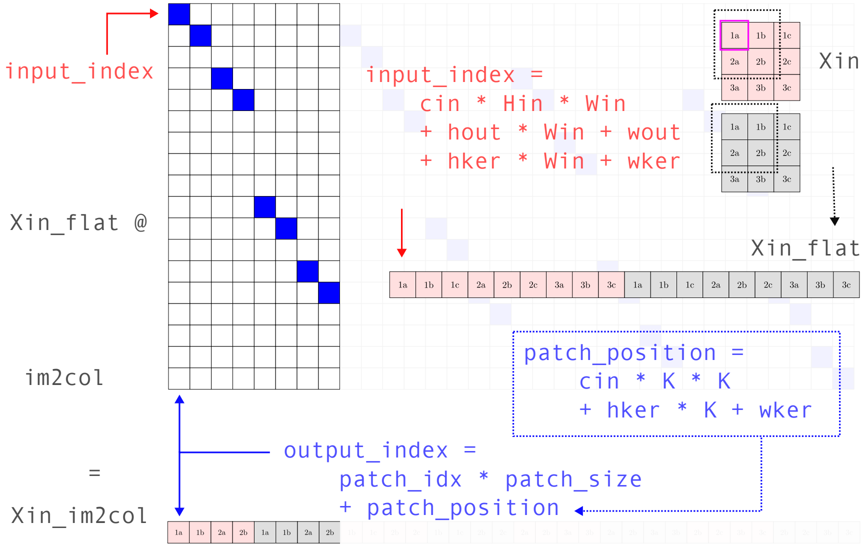
The “im2col” matrix



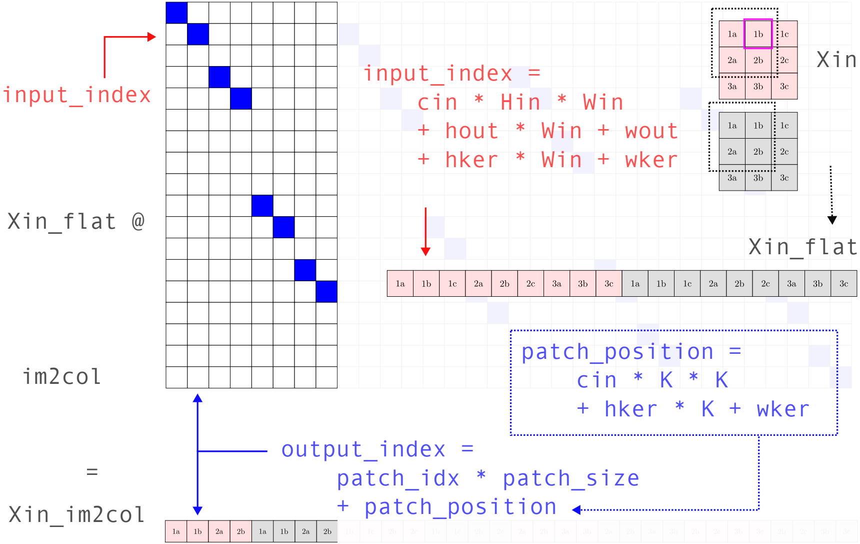
The “im2col” matrix



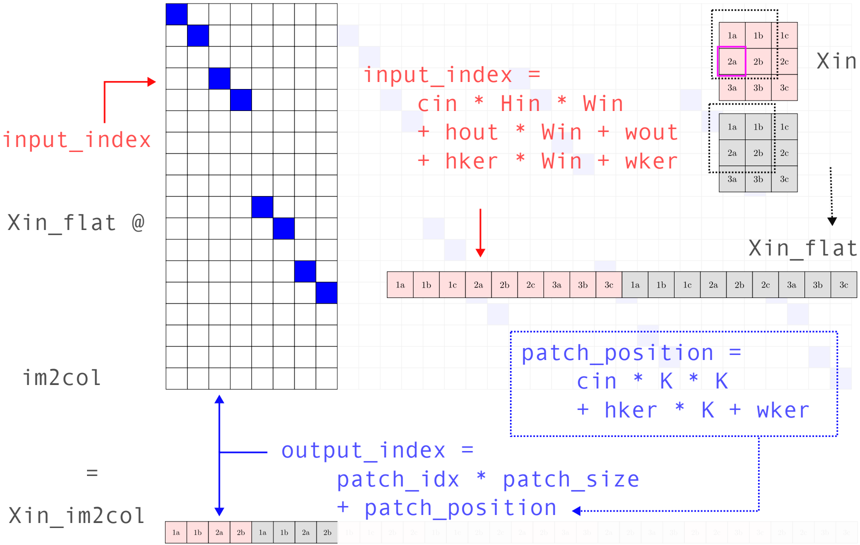
The “im2col” matrix



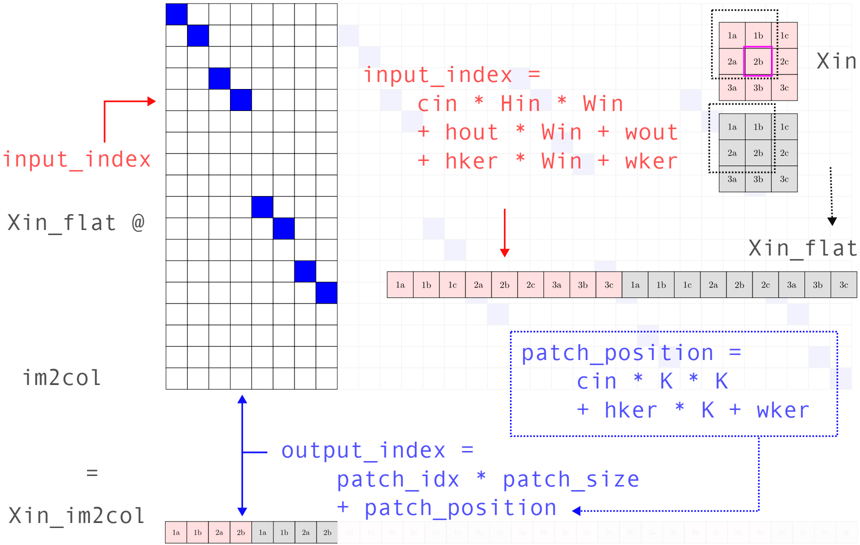
The “im2col” matrix



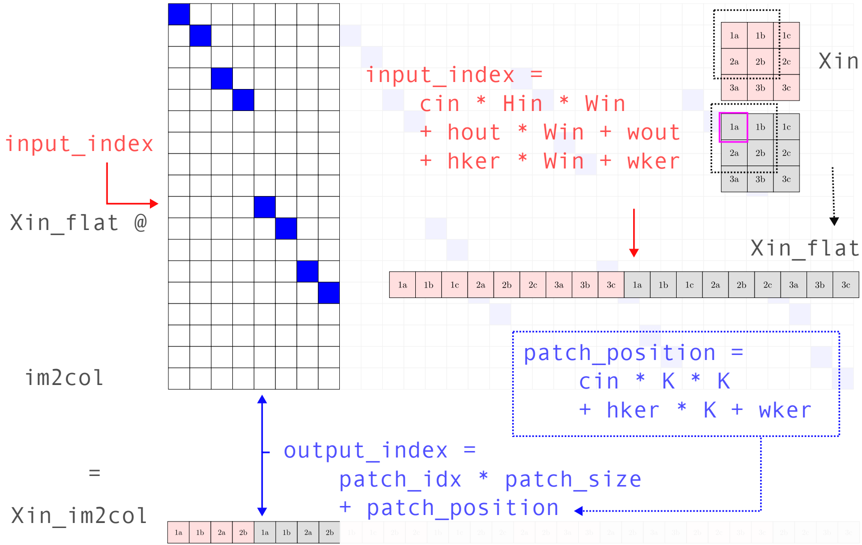
The “im2col” matrix



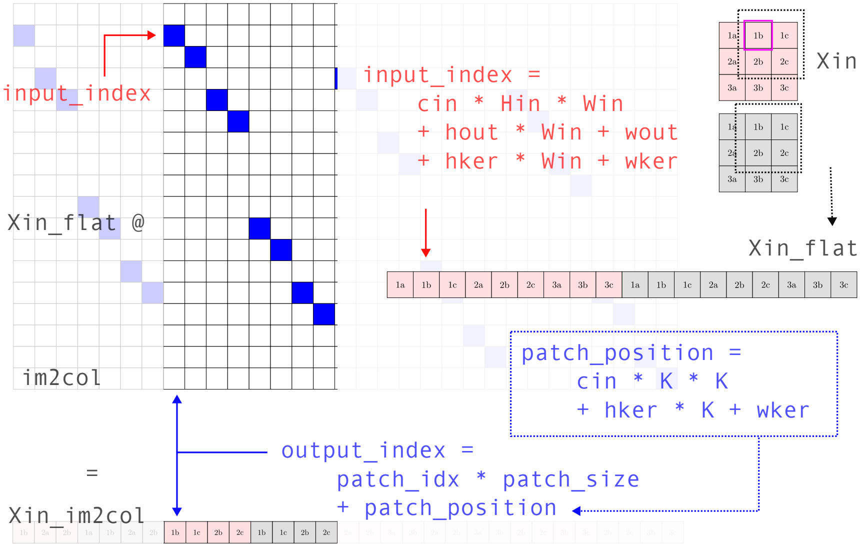
The "im2col" matrix



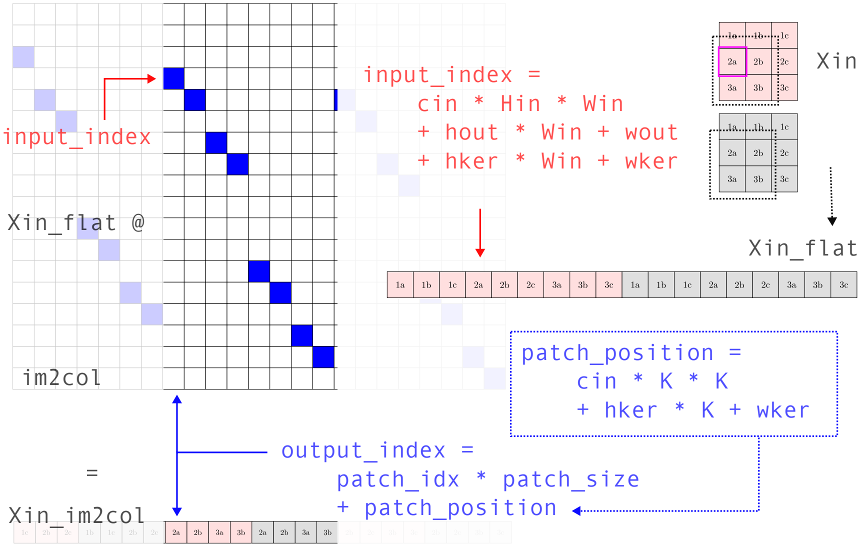
The “im2col” matrix



The "im2col" matrix



The "im2col" matrix



Putting it all together

```python
def im2col_matrix_dense(Xin, K, S=1):
    N, Cin, Hin, Win = Xin.shape
    Hout, Wout = Hin - K + 1, Win - K + 1
    P = Hout * Wout  # Total number of patches per image
    patch_size = Cin * K * K # Size of each flattened patch
    im2col_mat_dense = np.zeros((Cin * Hin * Win, P * patch_size))

    # [main loop on next slide...]

    return im2col_mat_dense
```

Putting it all together

```
    # [continued from previous slide...]
    patch_idx = 0
    for hout in range(Hout):
        for wout in range(Wout):
            for cin in range(Cin):
                for hker in range(K):
                    for wker in range(K):
                        input_index = cin * Hin * Win + hout * Win + wout + hker * Win + wker
                        patch_position = cin * K * K + hker * K + wker
                        output_index = patch_idx * patch_size + patch_position
                        im2col_mat_dense[input_index, output_index] = 1
            patch_idx += 1
```

Stride = 2

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e
3a	3b	3c	3d	3e
4a	4b	4c	4d	4e
5a	5b	5c	5d	5e

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e

Stride = 2

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e
3a	3b	3c	3d	3e
4a	4b	4c	4d	4e
5a	5b	5c	5d	5e

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e

Stride = 2

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e
3a	3b	3c	3d	3e
4a	4b	4c	4d	4e
5a	5b	5c	5d	5e

1a	1b	1c	1d	1e
2a	2b	2c	2d	2e

Implement `im2col_matrix_dense` with stride

See `lec09-in-class-ex1-im2col.ipynb`
End of part 1.

Goal for today: implement LeNet5

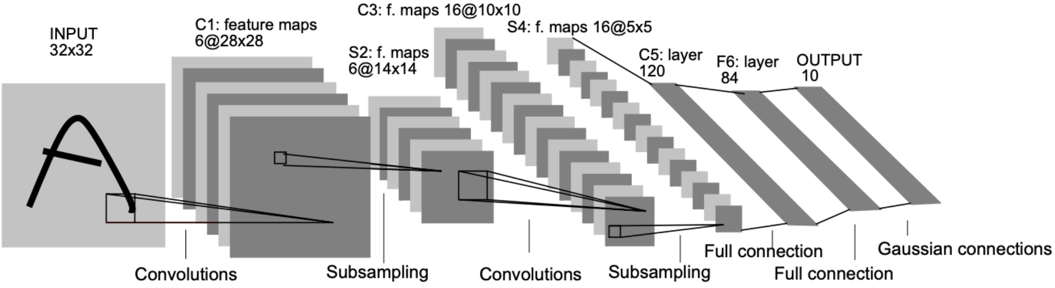
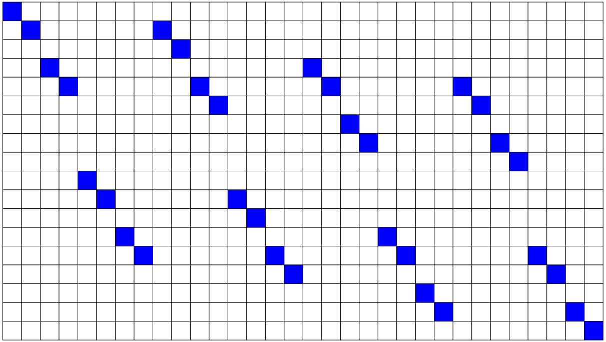


Image from LeCun et al. 1998

im2col.shape = $C_{\text{in}} \cdot H_{\text{in}} \cdot W_{\text{in}}$-**by**-$P \cdot C_{\text{out}} \cdot K^2$

Xin_flat @



im2col

=

Xin_im2col

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

`im2col.shape =` $C_{\text{in}} \cdot H_{\text{in}} \cdot W_{\text{in}}$**-by-**$P \cdot C_{\text{in}} \cdot K^2$

For 1st convolution layer in LeNet5.

- $H_{\text{in}} = W_{\text{in}} = 32$
- $C_{\text{in}} = 1$
- $K = 5$
- $S = 1$
- $H_{\text{out}} = H_{\text{in}} - K + 1 = 28$
- $W_{\text{out}} = W_{\text{in}} - K + 1 = 28$
- $P = 28 * 28$
- Final shape of `im2col` = 1024 -by- 19600
- 20,070,400 float64's to store ≈ 160 MB

Sparse matrices

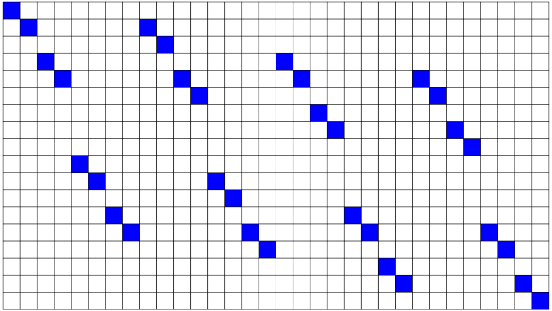
Pros

- Store just the non-zero values of the matrix (memory efficient)
- Faster than dense matmul when matrices are mostly zeros (speedier)

Cons

- Irregular memory access (less cache-optimization friendly)
- Nightmare for GPUs (as of now, to the best of my knowledge)

from scipy.sparse import csr_matrix



```
data = [1,1,1,1,1,...] # len(data) = P * patch_size
row_indices = [0,1,3,4,9,...]
col_indices = [0,1,2,3,4,...]
im2col_mat_sparse = csr_matrix((data,
                                (row_indices, col_indices)),
                               shape=(n_rows, n_cols))
```

Example

```
data = [1,2,3,1]
row_indices = [1,2,0,1]
col_indices = [0,1,2,1]
sparse_mat_example = csr_matrix((data, (row_indices, col_indices)),
    shape=(3, 3))
sparse_mat_example.toarray()
```

Output:

```
array([[0, 0, 3],
       [1, 1, 0],
       [0, 2, 0]])
```

Dense-sparse multiplication

```
X = np.arange(9).reshape(3,3)
## this won't work
# np.matmul(X, sparse_mat_example)
## this works bc method resolution order
X                    @   sparse_mat_example
```

```
array([[0, 1, 2],        array([[0, 0, 3],        array([[ 1,  5,  0],
       [3, 4, 5],    @          [1, 1, 0],    =          [ 4, 14,  9],
       [6, 7, 8]])              [0, 2, 0]])              [ 7, 23, 18]])
```

Implement `im2col_matrix_sparse`

- `lec09-in-class-ex2-sparse.ipynb`

Quick update

Can add/mul constants both on left and right now.

```
        def __radd__(self, other):
            return self + other

        def __rmul__(self, other):
            return self * other

        def __rsub__(self, other):
            return (-self) + other

        def __rtruediv__(self, other):
            return ag.Tensor(other) / self
```

Quick update

```python
    def moveaxis(input, source, destination):
        output = ag.Tensor(np.moveaxis(input.value, source, destination), inputs=[input], op="moveaxis")

        def _backward():
            input.grad += np.moveaxis(output.grad, source, destination)
            return None
        output._backward = _backward
        return output
```

Enabling autograd for dense-sparse matmul

- Problem: our current autograd-enabled matmul only supports when both inputs are numpy arrays
- Observation: we don't need to track gradients on the `im2col_mat`
- Solution:

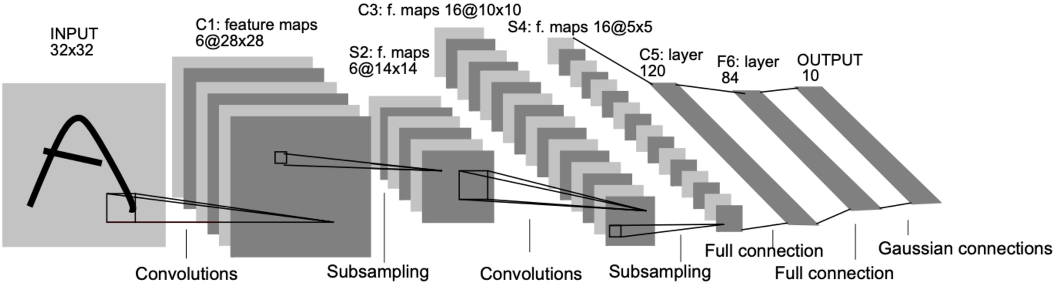
```python
# [sp]arse [c]onstant (non-AG-enabled) [mat]rix [mul]tiplication
def spcmatmul(input, sparse_mat: csr_matrix):
    output = ag.Tensor(input.value @ sparse_mat,
                       inputs = [input],
                       op="spcmatmul")

    def _backward():
        input.grad += output.grad @ sparse_mat.T
        return None

    output._backward = _backward
    return output
```

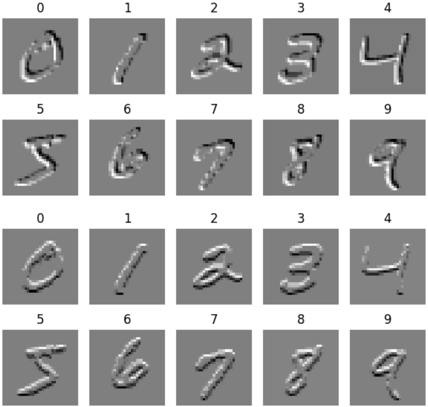
Exercise 3

- Create the `Conv2d` layer
- Create the `AvgPool2d` layer
- Create the `LeNet5` model



- Create the `nn.CrossEntropyLoss`

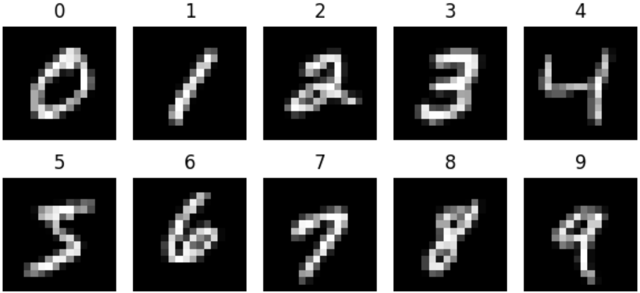
Sanity check for conv2d



Channel 1

Channel 2

Sanity check for pool



Sanity check for training run

You should achieve 100 percentage training accuracy

References I

[LeC+98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.