Convolutional nets

Lecture 08 — CS 577 Deep Learning

Instructor: Yutong Wang

Computer Science
Illinois Institute of Technology

October 9, 2024

Autograd-enabled tensors

Previously

- `ag.Scalar`

Today:

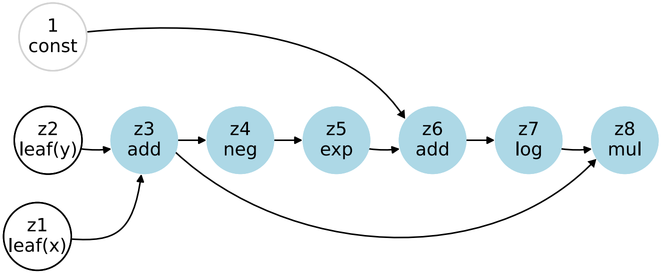
- Use `ag.Scalar` for training small relu networks
- `ag.Tensor`
- Overflow in cross entropy and how to avoid it
- Use `ag.Tensor` to train convolutional neural networks (CNNs)

Autograd-enabled scalars: `ag.Scalar`

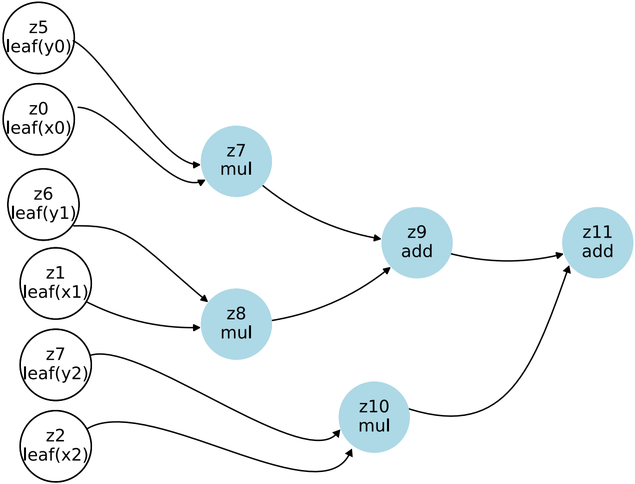
```python
    class Scalar: # Scalars with grads
        def __init__(self,
                     value,
                     op="",
                     _backward= lambda : None,
                     inputs=[],
                     label=""):

            self.value = float(value)
            self.grad = 0.0

            self._backward = _backward
            self.inputs = inputs

            self.op = op
            self.label = label
```

Computational graph

$f(x, y) = \log(1 + e^{-(x+y)}) \cdot (x + y).$

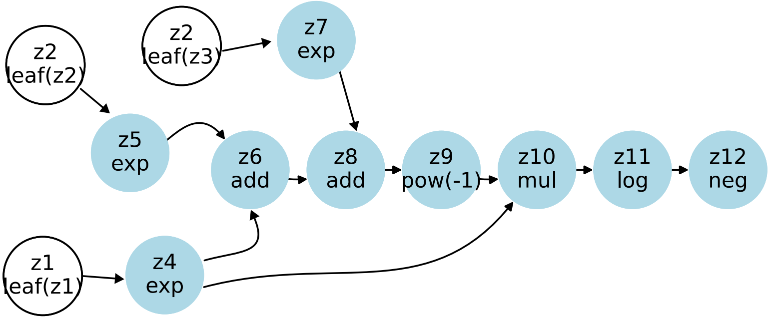


Computational graph
$f(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top \mathbf{y} = x_1 y_1 + x_2 y_2 + x_3 y_3$



Computational graph

$-\log(e^{z_1}/(e^{z_1}+e^{z_2}+e^{z_3}))$




```
expz1 = ag.exp(z1)
expz2 = ag.exp(z2)
expz3 = ag.exp(z3)
final_output = -ag.log( expz1/(expz1+expz2+expz3))
```

What can we do with `ag.Scalar`?

PyTorch-"lite" consisting of:

- A `Model` class (params and forward)
- A `Loss` class (just Mean Squared Error (MSE) for now)
- An `Optimizer` class (just gradient descent for now)
- A training loop

We will fit a 1-hidden layer neural network with 1-dimensional input

$$f(x; \mathbf{w}_1, \mathbf{b}_1, \mathbf{w}_2, b_2) = \mathbf{w}_2^\top \mathrm{relu}(\mathbf{w}_1 x + \mathbf{b}_1) + b_2$$

where $\mathbf{w}_1$ and $\mathbf{w}_2$ are both vectors of `n_hidden` dimensions.

Training Loop

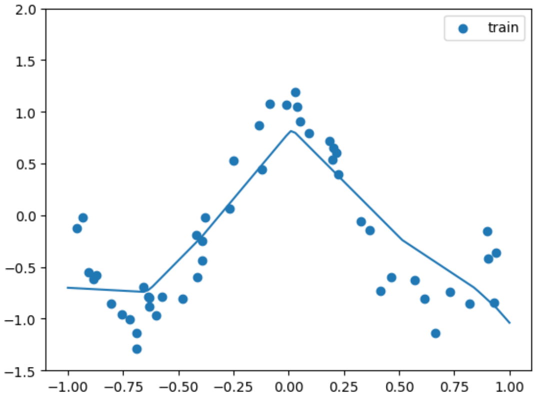
```python
model = Model(n_hidden=20)
loss_fn = Loss()
optimizer = Optimizer(model.parameters, lr=0.1)

for epoch in range(100):
    optimizer.zero_grad()
    output = model.forward(xnp)
    loss = loss_fn.mse(output, ynp)
    loss.backward()
    optimizer.step()
    if epoch % 10 == 0:
        print(f"Iteration {epoch}, Loss: {loss.value}")
```

Exercise 1

`lec08-in-class-ex1-framework.ipynb`

$$f(x; \mathbf{w}_1, \mathbf{b}_1, \mathbf{w}_2, b_2) = \mathbf{w}_2^\top \mathrm{relu}(\mathbf{w}_1 x + \mathbf{b}_1) + b_2$$

where $\mathbf{w}_1$ and $\mathbf{w}_2$ are both vectors of `n_hidden` dimensions.



The `Model` class

```python
class Model:
    def __init__(self, n_hidden, rng_seed=42):
        np.random.seed(rng_seed)

        w1np = np.random.randn(n_hidden)
        # ...
        b2np = np.random.randn(1)

        self.w1 = [ag.Scalar(val) for val in w1np]
        # ...
        self.b2 = [ag.Scalar(val) for val in b2np]

        self.parameters = self.w1 + self.b1 + self.w2 + self.b2

    def forward(self, x): # ...
        # x is a 1-dimensional numpy array
```

The `forward` function

```python
    def forward(self, x):
        # x is a 1-dimensional numpy array
        # "upgrade" x into ag.Scalars
        x_scalar = [ag.Scalar(val) for val in x]
        n_samples = len(x_scalar)

        # calculate the forward

        ## YOUR CODE HERE
        return [ag.Scalar(0.0) for i in range(n_samples)]
```

The `Loss` class

```python
class Loss:
    def mse(self, predictions, targets):
        # mean squared error
        assert len(predictions) == len(targets)
        n_samples = len(predictions)
        loss = ag.Scalar(0.0)

        # YOUR CODE HERE

        return loss
```

The `Optimizer` class

```python
class Optimizer:
    def __init__(self, parameters, lr=0.01):
        self.parameters = parameters
        self.lr = lr

    def zero_grad(self):
        for param in self.parameters:
            param.grad = 0.0

    def step(self):
        for param in self.parameters:
            param.value -= self.lr * param.grad
```

Check gradients!

```
Gradients w.r.t...
{'w1': array([-0.77812733, -0.          , ...

 'b1': array([ 2.35879461, -0.          , ...

 'w2': array([ 4.15815372e+00, -0.00000000e+00, ...

 'b2': 3.194179224246 4686}
```

Upshot

- It seems to work just fine.

Autograd-enabled tensors

Previously

- `ag.Scalar`

Today:

- $\boxed{\checkmark}$ Use `ag.Scalar` for training small relu networks
- `ag.Tensor`
- Use `ag.Tensor` to train convolutional neural networks (CNNs)
- Overflow in cross entropy and how to avoid it

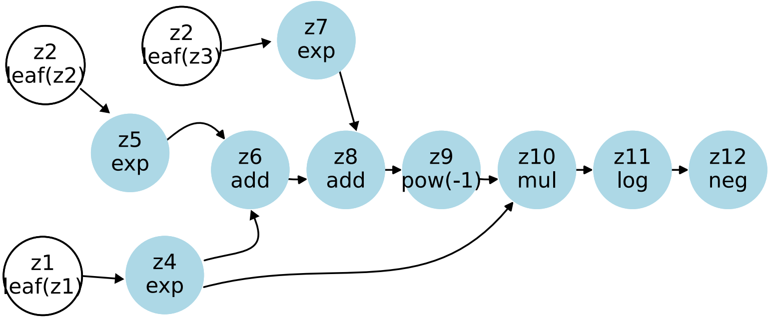
ag.Scalar

```python
class Scalar: # Scalars with grads
    def __init__(self,
                 value,
                 op="",
                 _backward= lambda : None,
                 inputs=[],
                 label=""):

        self.value = float(value)
        self.grad = 0.0
```

ag.Tensor

```
    class Tensor: # Tensor with grads
        def __init__(self,
                     value,
                     op="",
                     _backward= lambda : None,
                     inputs=[],
                     label=""):

            if type(value) in [float ,int]:
                value = np.array(value)
            self.value = value # <--------------------\
#                                                      |- same shape!
            self.grad = np.zeros_like(value) #  <-----/
```

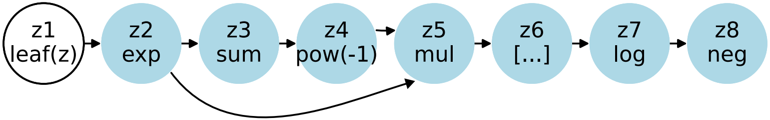
Computational graph
$-\log(e^{z_1}/(e^{z_1}+e^{z_2}+e^{z_3}))$




```
expz1 = ag.exp(z1)
expz2 = ag.exp(z2)
expz3 = ag.exp(z3)
final_output = -ag.log( expz1/(expz1+expz2+expz3))
```

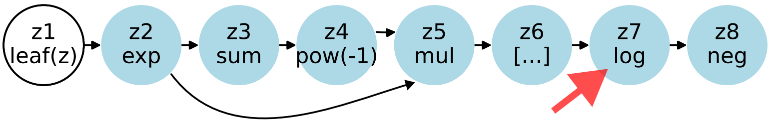
Computational graph

$-\log(e^{z_1}/(e^{z_1}+e^{z_2}+e^{z_3})) = L(\mathbf{z},1)$ $\leftarrow$ cross entropy



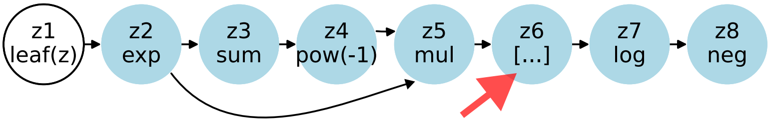

```python
z = ag.Tensor(np.random.randn(3),label="z1\nleaf(z)")
expX = ag.exp(z)
p = expX/ ag.sum(expX)
nll = -ag.log(p[0]) # negative log-likelihood
```

Computational graph
Entrywise ops



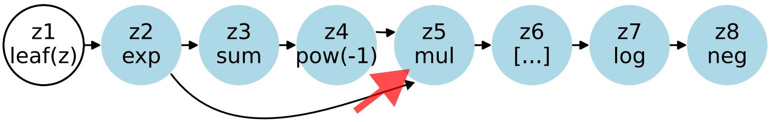

```python
def log(input):
    output = ag.Tensor(np.log(input.value), inputs=[input], op="log")
    def _backward():
        input.grad += output.grad / input.value
        return None
    output._backward = _backward
    return output
```

Computational graph
Slicing




```python
def __getitem__(self, idx):
    output = ag.Tensor(np.array(self.value[idx]),
                       inputs = [self],
                       op=f"[...]")
    def _backward():
        self.grad[idx] += output.grad # idx must not have repeats!
        return None
    output._backward = _backward
    return output
```

Computational graph
Pairwise ops — multiplication (NOT matmul)




```
def __mul__(self, other):
    # ... lines skipped
    output = ag.Tensor(self.value * other.value,
                       inputs=[self, other], op="mul")
    def _backward():
        self.grad += unbroadcast(output.grad*other.value, ax1, pad1)
        other.grad += unbroadcast(output.grad*self.value, ax2, pad2)
```

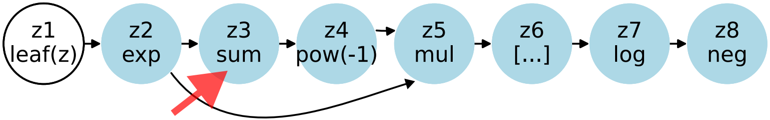
What is "`unbroadcast`" and why define backward this way? ... next

Computational graph
Pairwise ops — multiplication (NOT matmul)

What is "`unbroadcast`" and why define backward this way?

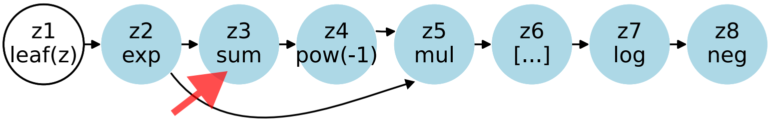
- `X1.shape = (2,3,1,5,6)`
- `X2.shape = (4,1,6)`
- What is `(X1 * X2).shape`?

Computational graph
Reductive ops — sum



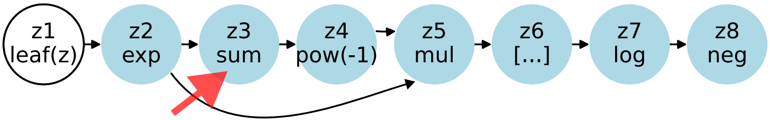

```python
def sum(input,axis = None, keepdims = False):
    output = ag.Tensor(np.sum(input.value, axis = axis, keepdims =
    keepdims), inputs = [input], op='sum')
# ...
# ...
# ...
# ...
# ...
# ...
# ...
# ...
```

Computational graph
Reductive ops — sum




```python
def sum(input,axis = None, keepdims = False):
    output = ag.Tensor(np.sum(input.value, axis = axis, keepdims =
    keepdims), inputs = [input], op='sum')
    def _backward():
        if axis == None or keepdims:
            input.grad += output.grad
        else:
# ...
# ...
# ...
# ...
```

Computational graph
Reductive ops — sum




```
def sum(input,axis = None, keepdims = False):
    output = ag.Tensor(np.sum(input.value, axis = axis, keepdims =
   keepdims), inputs = [input], op='sum')
    def _backward():
        if axis == None or keepdims:
            input.grad += output.grad
        else:
            input.grad += np.expand_dims(output.grad, axis = axis)
        return None
# ...
# ...
```

Exercises

- Reshape (Exercise 2.1)
  — Fix the backward function for reshape

Exercises

- Reshape (Exercise 2.1)
  — Fix the backward function for reshape
- Indexing (Exercise 2.2)
  — Fix the loss derivative autograd calculation

Exercises

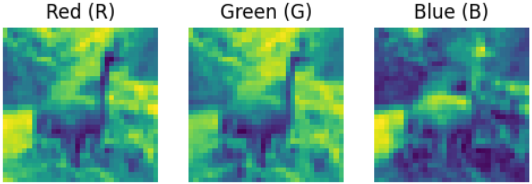
- Reshape (Exercise 2.1)
  — Fix the backward function for reshape
- Indexing (Exercise 2.2)
  — Fix the loss derivative autograd calculation
- Implementing BCE (Exercise 2.3)
  — Implement the binary cross entropy.

Convolution

$$\texttt{X.shape} = (N, C, H, W) = (2, 3, 32, 32) \quad \text{(2 images from CIFAR10)}$$

Convolution

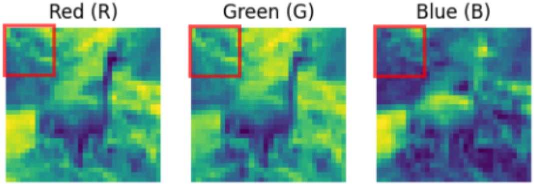
$$\texttt{X.shape} = (N, C, H, W) = (2, 3, 32, 32)$$



$$\texttt{X[0].shape} = (C, H, W) = (3, 32, 32)$$

Convolution

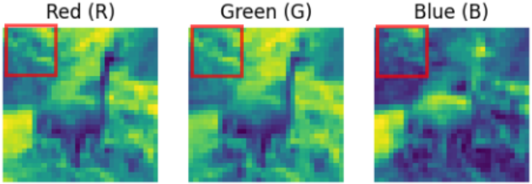
$$\texttt{X.shape} = (N, C, H, W) = (2, 3, 32, 32)$$



$$\texttt{X[0].shape} = (C, H, W) = (3, 32, 32)$$

Convolution

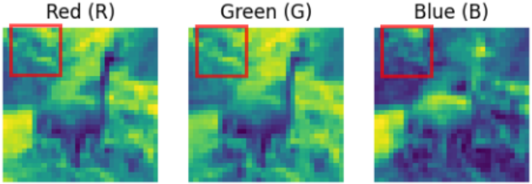
$$\texttt{X.shape} = (N, C, H, W) = (2, 3, 32, 32)$$



$$\texttt{X[0].shape} = (C, H, W) = (3, 32, 32)$$

Convolution

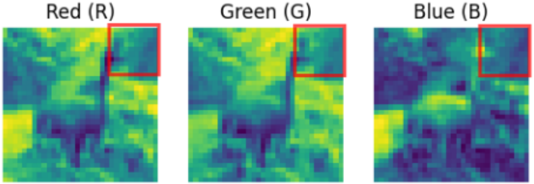
$$\texttt{X.shape} = (N, C, H, W) = (2, 3, 32, 32)$$



$$\texttt{X[0].shape} = (C, H, W) = (3, 32, 32)$$

Convolution

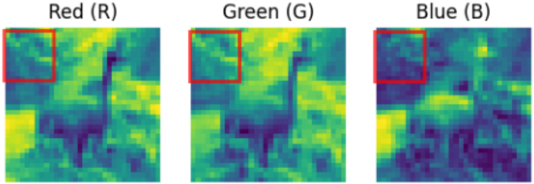
$$\texttt{X.shape} = (N, C, H, W) = (2, 3, 32, 32)$$



$$\texttt{X[0].shape} = (C, H, W) = (3, 32, 32)$$

Convolution

$$\texttt{X.shape} = (N, C, H, W) = (2, 3, 32, 32)$$



$$\texttt{X[0].shape} = (C, H, W) = (3, 32, 32)$$

LeNet5

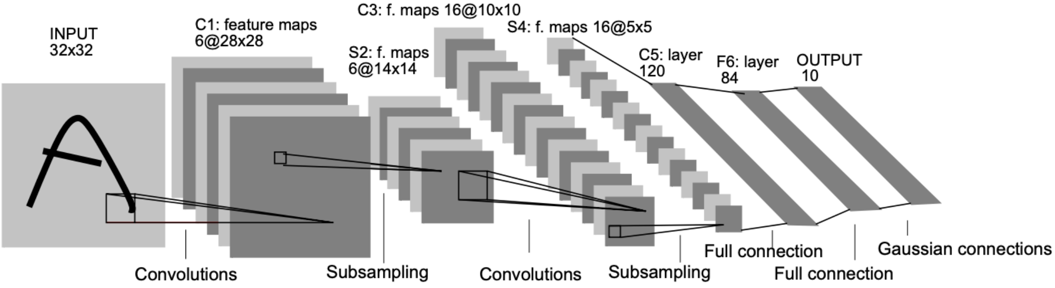
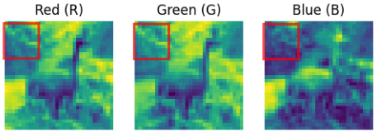
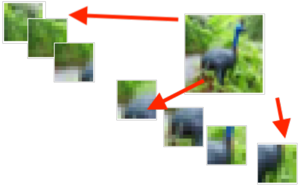


Image from LeCun et al. 1998

Convolution



$$\texttt{X[0].shape} = (C, H, W) = (3, 32, 32)$$



$$\texttt{X\_patches[0].shape} = (P, C, K, K) = (3, 32, 32)$$

K = size of window, P = number of patches

A simple example

```
C, H, W = 2, 4, 5
K = 2  # Kernel size
N = 2  # Batch size
X = np.arange(N * C * H * W).reshape(N, C, H, W)
X[0]
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]]])
```

Goal: compute `X_patches`. Okay, but what is `X_patches`?

A simple example

```
X[0]
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]]])
```

Goal: compute `X_patches`. Okay, but what is `X_patches`?

```
X_patches[0,0]
array([[[ 0.,  1.],
        [ 5.,  6.]],

       [[20., 21.],
        [25., 26.]]])
```

A simple example

```
X[0]
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]]])
```

Goal: compute `X_patches`. Okay, but what is `X_patches`?

```
X_patches[0,1]
array([[[ 1.,  2.],
        [ 6.,  7.]],

       [[21., 22.],
        [26., 27.]]])
```

A simple example

```
X[0]
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]]])
```

Goal: compute `X_patches`. Okay, but what is `X_patches`?

```
X_patches[0,3]
array([[[ 3.,  4.],
        [ 8.,  9.]],

       [[23., 24.],
        [28., 29.]]])
```

A simple example

```
X[0]
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]]])
```

Goal: compute `X_patches`. Okay, but what is `X_patches`?

```
X_patches[0,4]
array([[[ 5.,  6.],
        [10., 11.]],

       [[25., 26.],
        [30., 31.]]])
```

A simple example

```
X[1]
array([[[40, 41, 42, 43, 44],
         [45, 46, 47, 48, 49],
         [50, 51, 52, 53, 54],
         [55, 56, 57, 58, 59]],

        [[60, 61, 62, 63, 64],
         [65, 66, 67, 68, 69],
         [70, 71, 72, 73, 74],
         [75, 76, 77, 78, 79]]])
```

Goal: compute `X_patches`. Okay, but what is `X_patches`?

```
X_patches[1,0]
array([[[40., 41.],
         [45., 46.]],

        [[60., 61.],
         [65., 66.]]])
```

X_patches

$$\texttt{X\_patches[0].shape} = (P, C, K, K) = (3, 32, 32)$$

K = size of window, P = number of patches

```
CHW = C * H * W
out_H = H - K + 1 # Output height
out_W = W - K + 1 # Output width
P = out_H * out_W  # Total number of patches per image
```

Strategy:

- First flatten `X` to be a matrix, call it `X_flat`

X_patches

$$\texttt{X\_patches[0].shape} = (P, C, K, K) = (3, 32, 32)$$

K = size of window, P = number of patches

```
CHW = C * H * W
out_H = H - K + 1 # Output height
out_W = W - K + 1 # Output width
P = out_H * out_W  # Total number of patches per image
```

Strategy:

- First flatten `X` to be a matrix, call it `X_flat`
- Pick out the patches using this "`im2col_mat`" matrix

X_patches

$$\texttt{X\_patches[0].shape} = (P, C, K, K) = (3, 32, 32)$$

K = size of window, P = number of patches

```
CHW = C * H * W
out_H = H - K + 1 # Output height
out_W = W - K + 1 # Output width
P = out_H * out_W  # Total number of patches per image
```

Strategy:

- First flatten `X` to be a matrix, call it `X_flat`
- Pick out the patches using this "`im2col_mat`" matrix
- (there are two flavors: dense and sparse. More on that later)

X_patches

$$\texttt{X\_patches[0].shape} = (P, C, K, K) = (3, 32, 32)$$

K = size of window, P = number of patches

```
CHW = C * H * W
out_H = H - K + 1 # Output height
out_W = W - K + 1 # Output width
P = out_H * out_W  # Total number of patches per image
```

Strategy:

- First flatten `X` to be a matrix, call it `X_flat`
- Pick out the patches using this "`im2col_mat`" matrix
- (there are two flavors: dense and sparse. More on that later)
- Get `X_patches_flat`. Reshape them to get `X_patches`.

A simple example

```
C, H, W = 2, 4, 5
K,N = 2,2  # Kernel size, Batch size
X = np.arange(N * C * H * W).reshape(N, C, H, W)
X[0]
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
# ...
X_flat = X.reshape(N, -1)  # Shape (N, C*H*W)
```

A simple example

```
C, H, W = 2, 4, 5
K,N = 2,2  # Kernel size, Batch size
X = np.arange(N * C * H * W).reshape(N, C, H, W)
X[0]
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
# ...
X_flat = X.reshape(N, -1)  # Shape (N, C*H*W)
X_flat[0]
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
       16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
       33, 34, 35, 36, 37, 38, 39])
```

A simple example

```
X_flat = X.reshape(N, -1)  # Shape (N, C*H*W)
X_flat[0]
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
       16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
       33, 34, 35, 36, 37, 38, 39])

im2col_mat_dense = im2col_matrix_dense(X, K)    # assume this given
X_out_dense = np.matmul(X_flat, im2col_mat_dense)
X_out_dense
array([[ 0.,  1.,  5.,  6., 20., 21., 25., 26.,  1.,  2.,  6.,  7., 21.,
        22., 26., 27.,  2.,  3.,  7.,  8., 22., 23., 27., 28.,  3.,  4.,
         8.,  9., 23., 24., 28., 29.,  5.,  6., 10., 11., 25., 26., 30.,
        31.,  6.,  7., 11., 12., 26., 27., 31., 32.,  7.,  8., 12., 13.,
...
```

A simple example

```
X_out_dense = np.matmul(X_flat, im2col_mat_dense)
X_out_dense
array([[ 0.,  1.,  5.,  6., 20., 21., 25., 26.,  1.,  2.,  6.,  7., 21.,
        22., 26., 27.,  2.,  3.,  7.,  8., 22., 23., 27., 28.,  3.,  4.,
         8.,  9., 23., 24., 28., 29.,  5.,  6., 10., 11., 25., 26., 30.,
        31.,  6.,  7., 11., 12., 26., 27., 31., 32.,  7.,  8., 12., 13.,
...
X_patches = X_patches_flat.reshape(N,P,C,K,K)
```

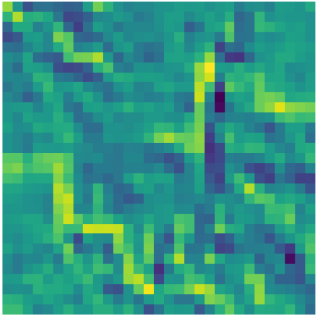
A simple example

```
X_out_dense = np.matmul(X_flat, im2col_mat_dense)
X_out_dense
array([[ 0.,  1.,  5.,  6., 20., 21., 25., 26.,  1.,  2.,  6.,  7., 21.,
        22., 26., 27.,  2.,  3.,  7.,  8., 22., 23., 27., 28.,  3.,  4.,
         8.,  9., 23., 24., 28., 29.,  5.,  6., 10., 11., 25., 26., 30.,
        31.,  6.,  7., 11., 12., 26., 27., 31., 32.,  7.,  8., 12., 13.,
...
X_patches = X_patches_flat.reshape(N,P,C,K,K)
X_patches[0,0]
array([[[ 0.,  1.],
         [ 5.,  6.]],

        [[20., 21.],
         [25., 26.]]])
```

Exercise 3

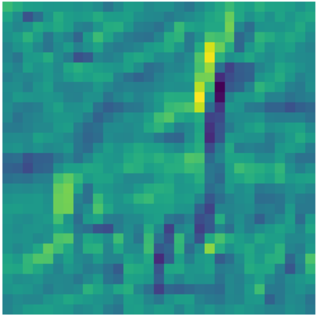
```
    # input
X = np.arange(N * C * H * W).reshape(N, C, H, W)
X[0]
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],
    # ...

    # desired output
    # (2 times lower left - 1 times top right, ignore second channel)
X_convolved.shape = (2, 3, 4)
X_convolved[0]
array([[ 9., 10., 11., 12.],
       [14., 15., 16., 17.],
       [19., 20., 21., 22.]])
```

Smoothness



```
Conv_kernel = np.array([[[ 0,  -1.], [ 1.,  0.]],  [[0., 0.], [0., 0.]],
     [[0., 0.], [0., 0.]]])
```

Smoothness



```
Conv_kernel = np.array([[[ 1,  0.], [ 0.,  -1.]],  [[0., 0.], [0., 0.]],
     [[0., 0.], [0., 0.]]])
```

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.