Convolutions & Vectorization

CSE 493G1, Spring 2026

Materials prepared by Tanush Yadav
Edited by Kirandeep Kaur

Agenda

- Course Logistics
- Convolutions
- Vectorization

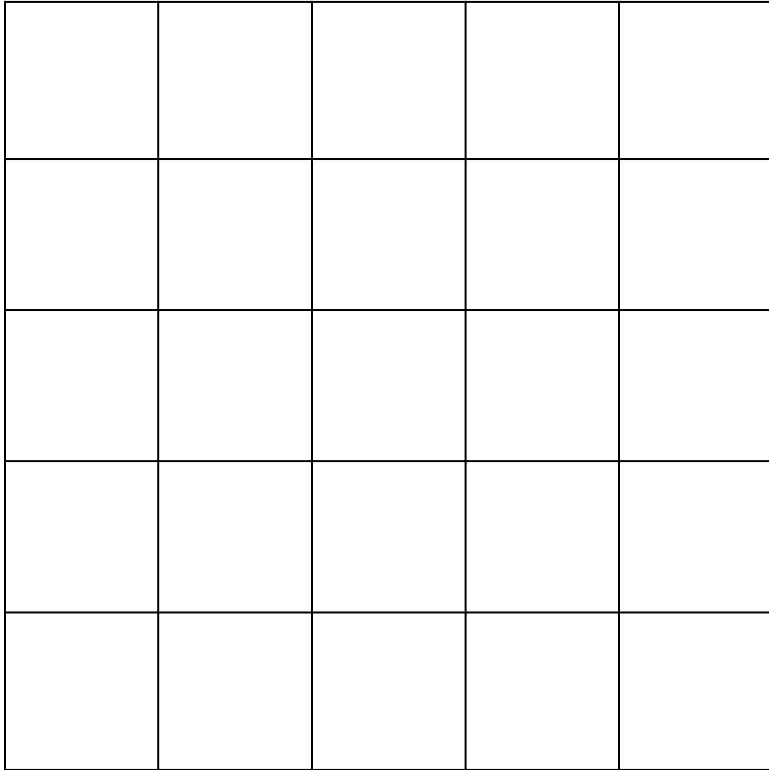
Course Logistics

- A2 released and due April 28
- Mid term on April 28
- Project Proposal due on April 30

Convolutions

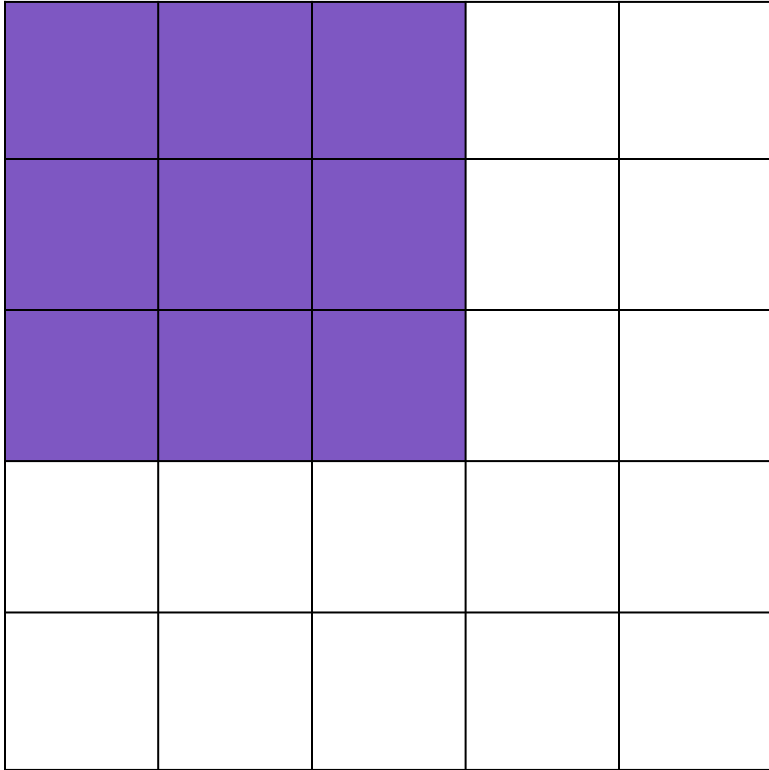
Input: 5x5x1

Width: 5
Height: 5
Depth: 1



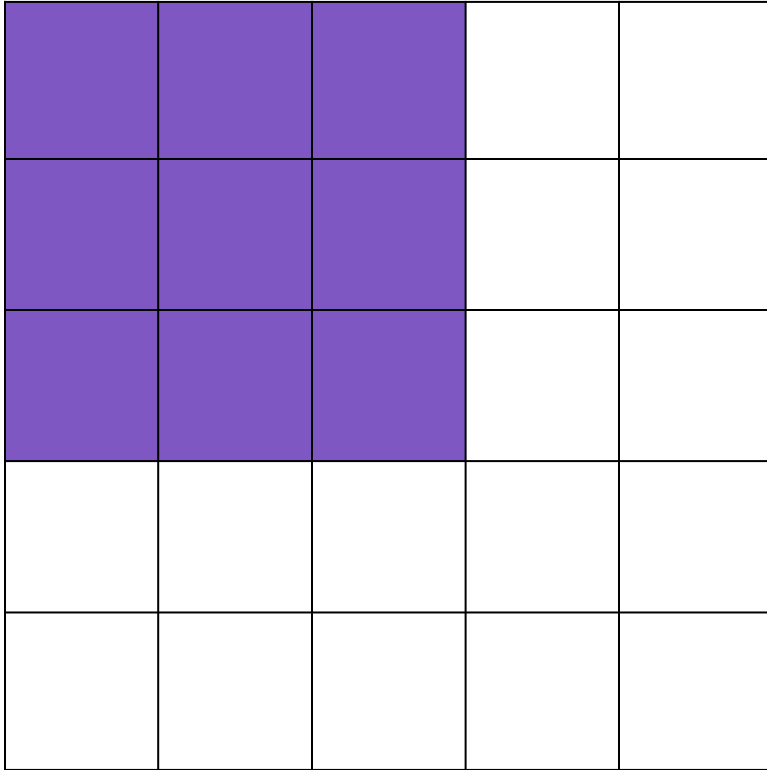
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



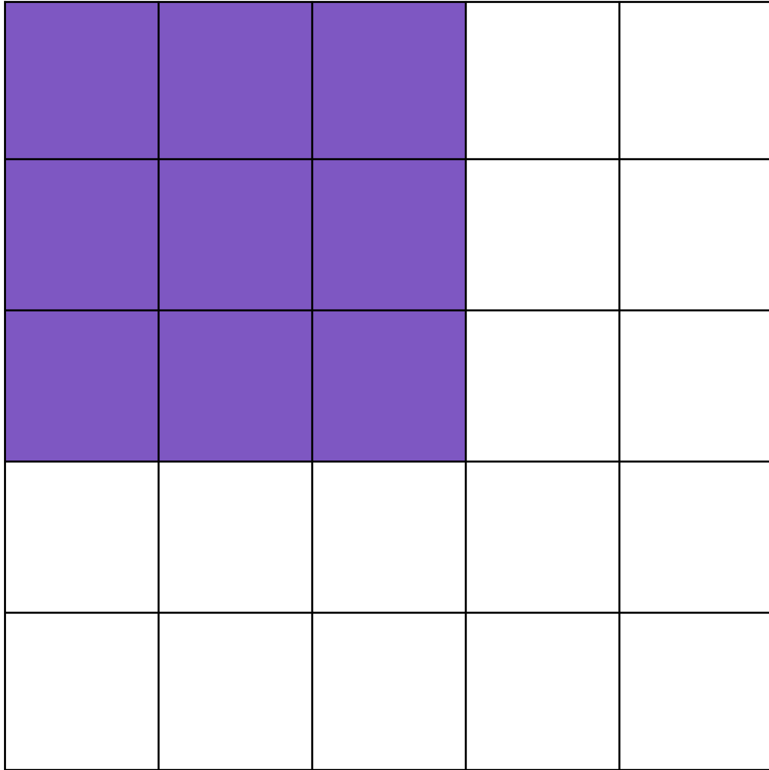
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



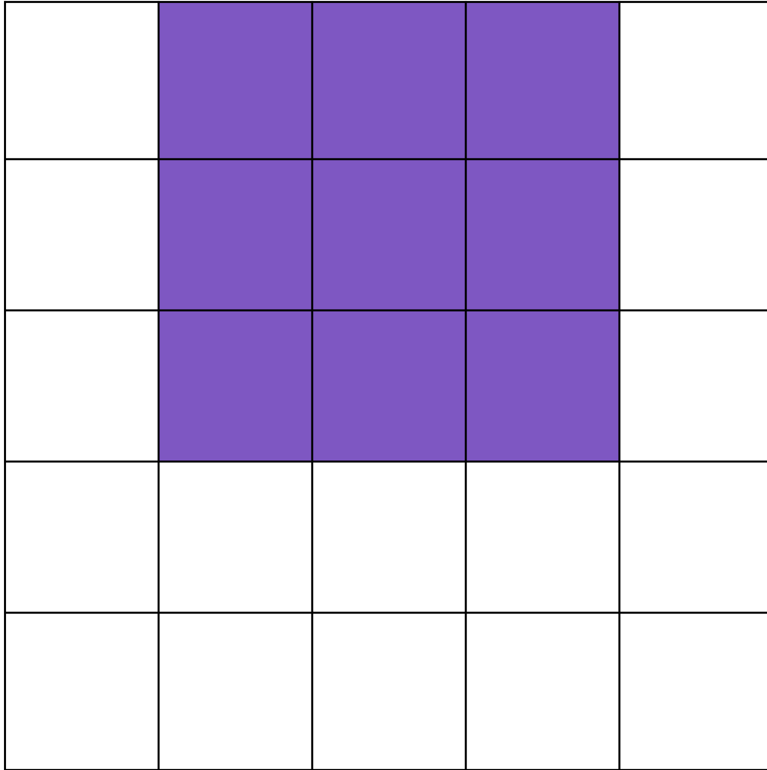
A note on terminology.

- filter
- neuron's receptive field



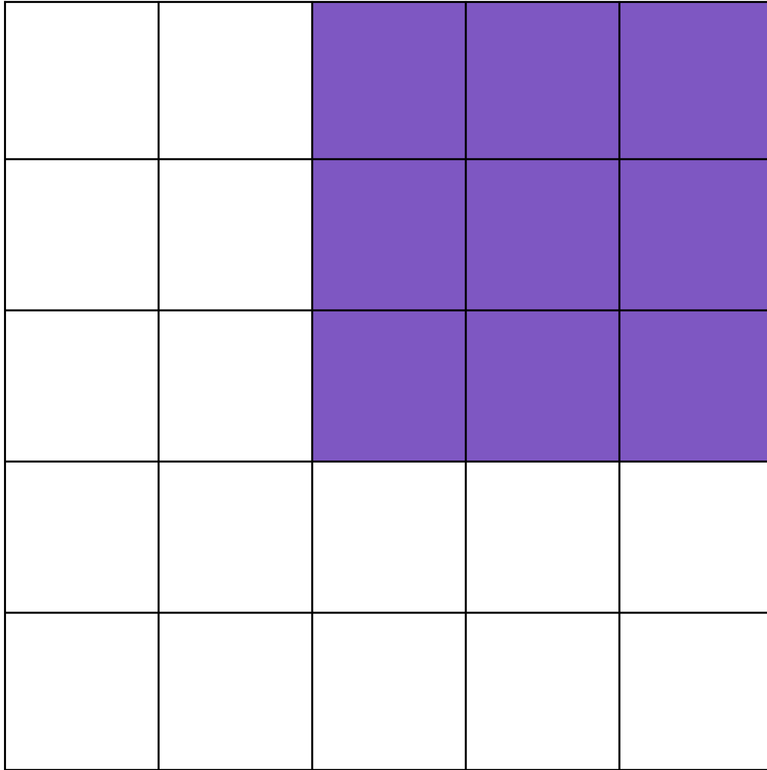
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



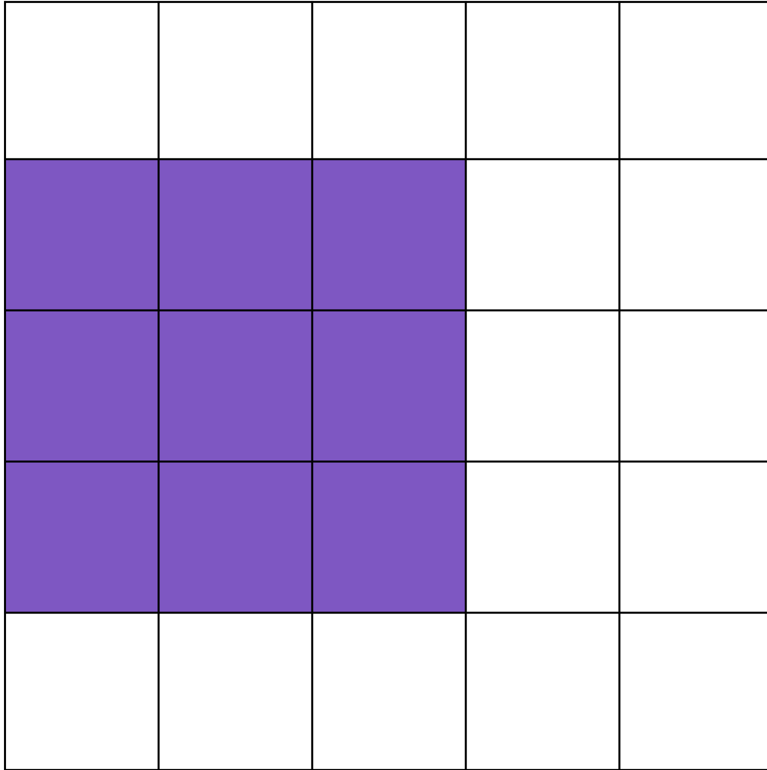
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



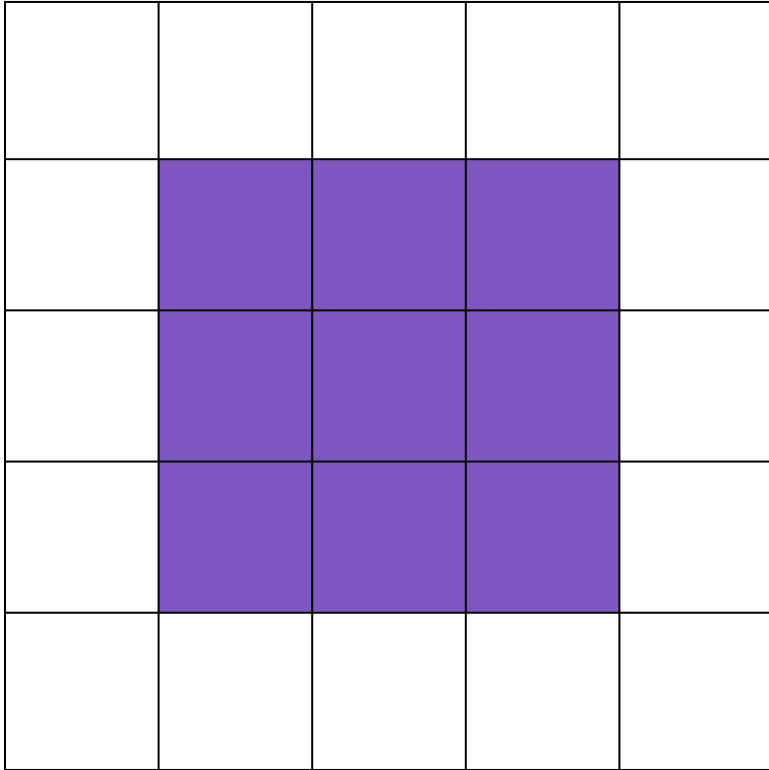
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



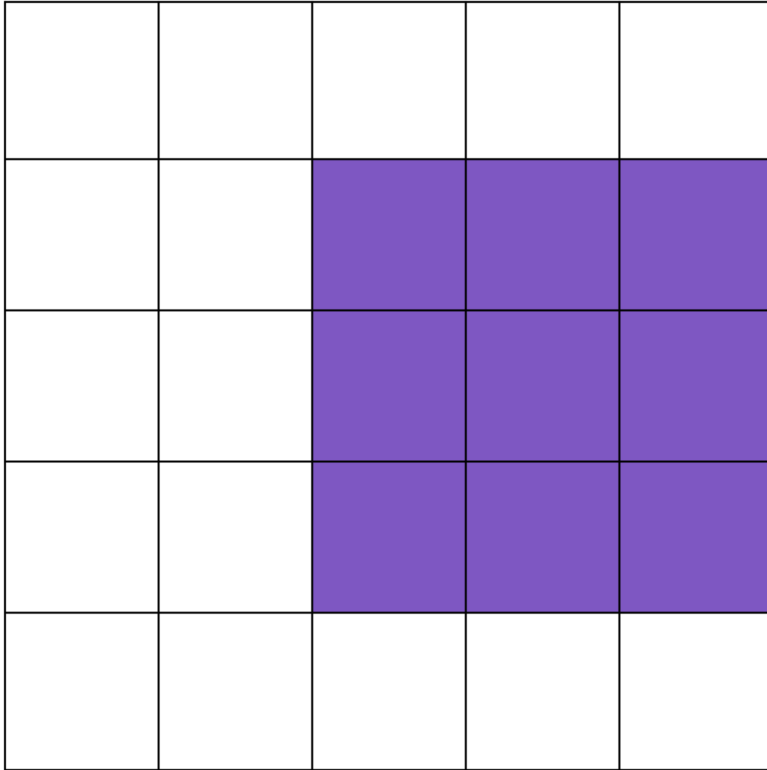
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



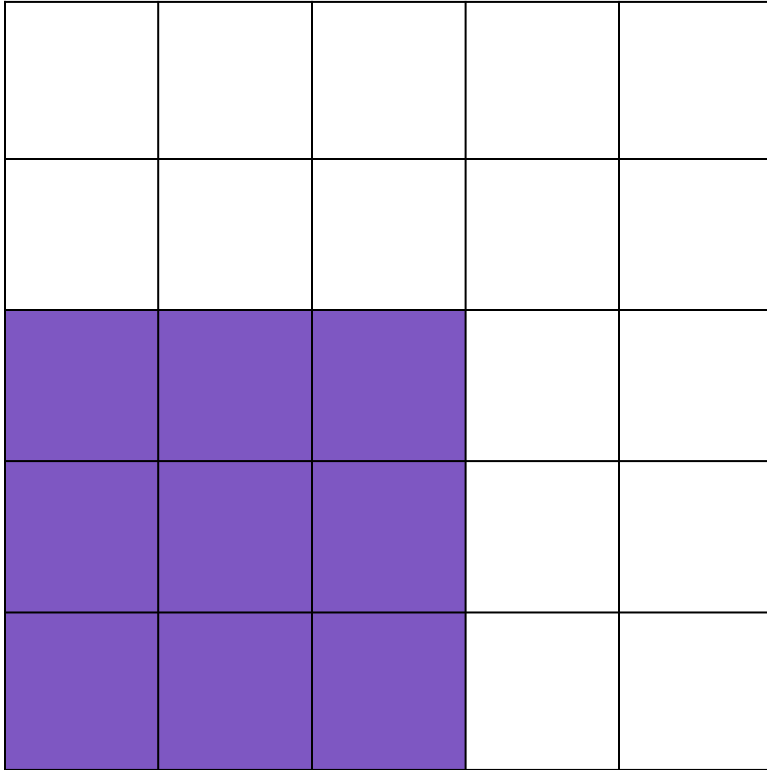
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



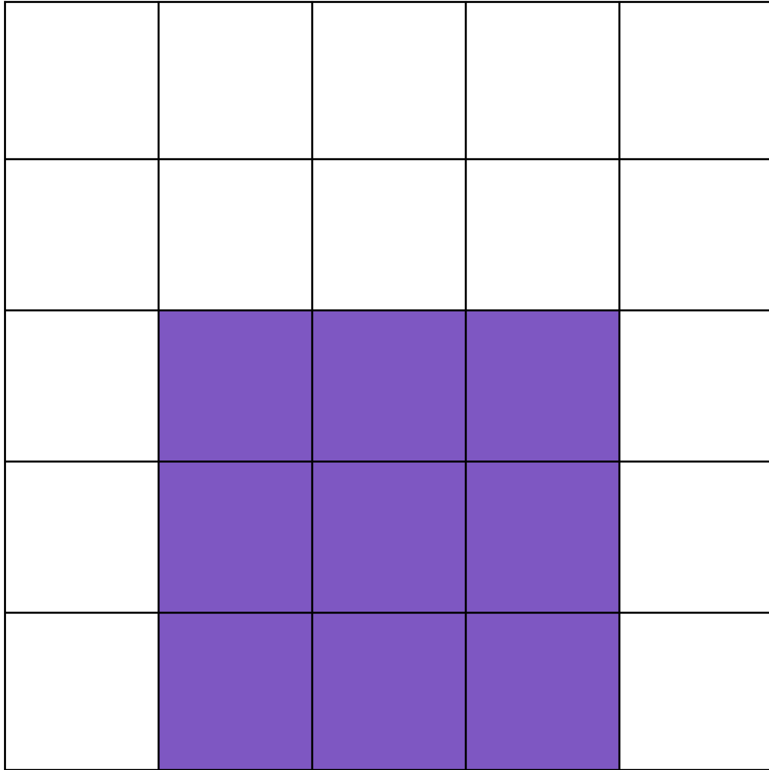
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



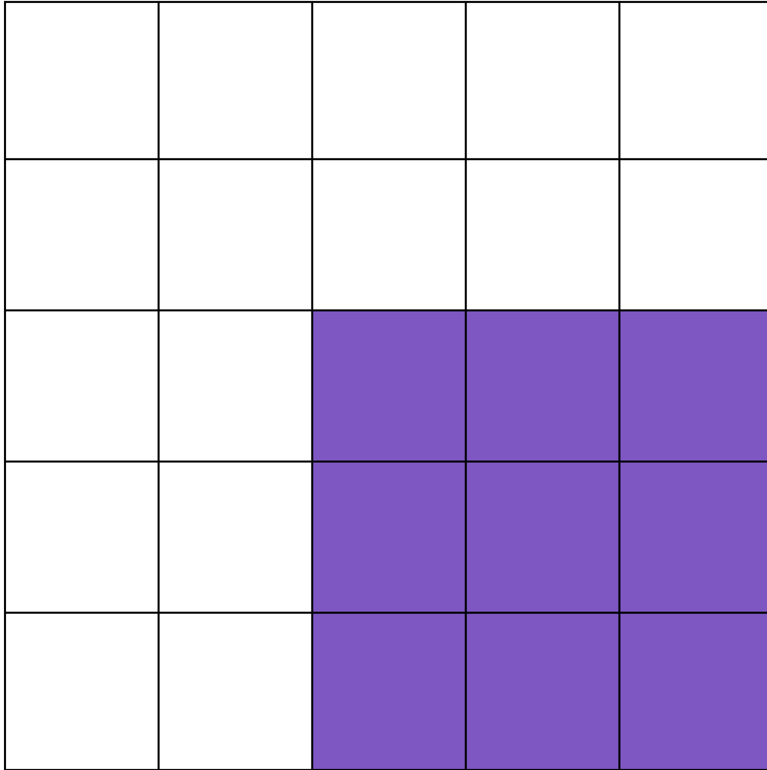
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3

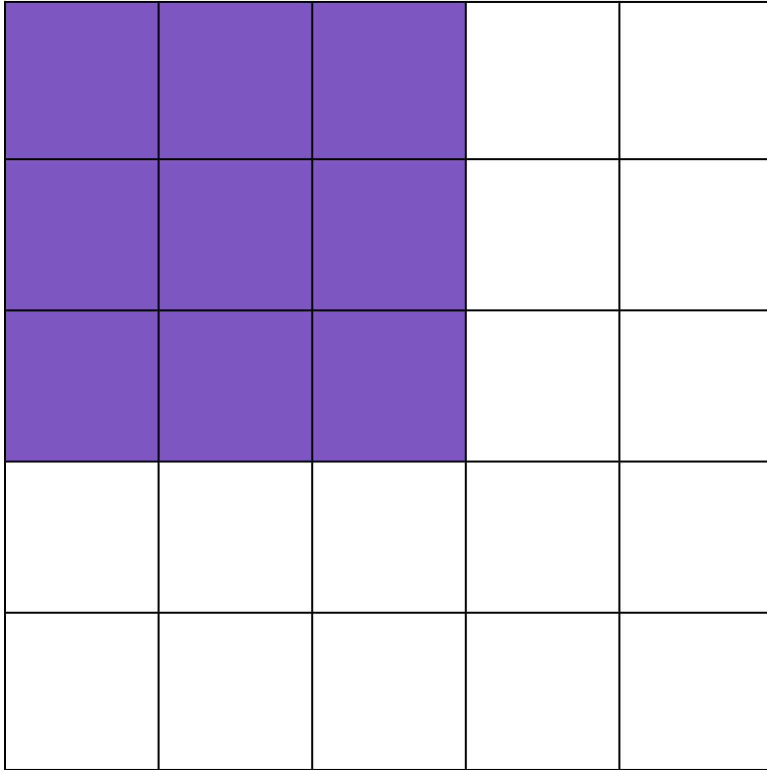


Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3

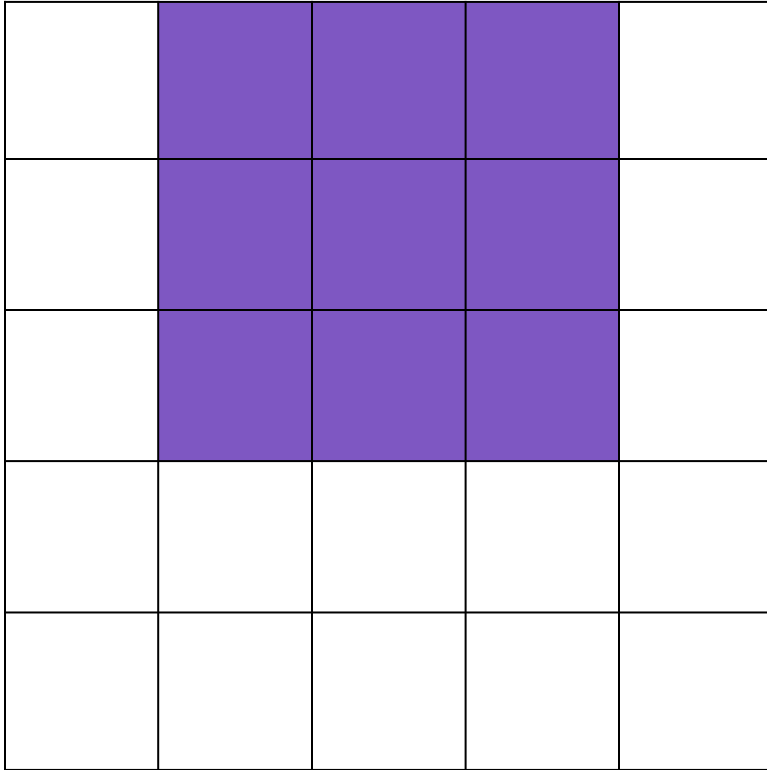
Shift horizontally and vertically is the same.

Let’s focus on studying one.



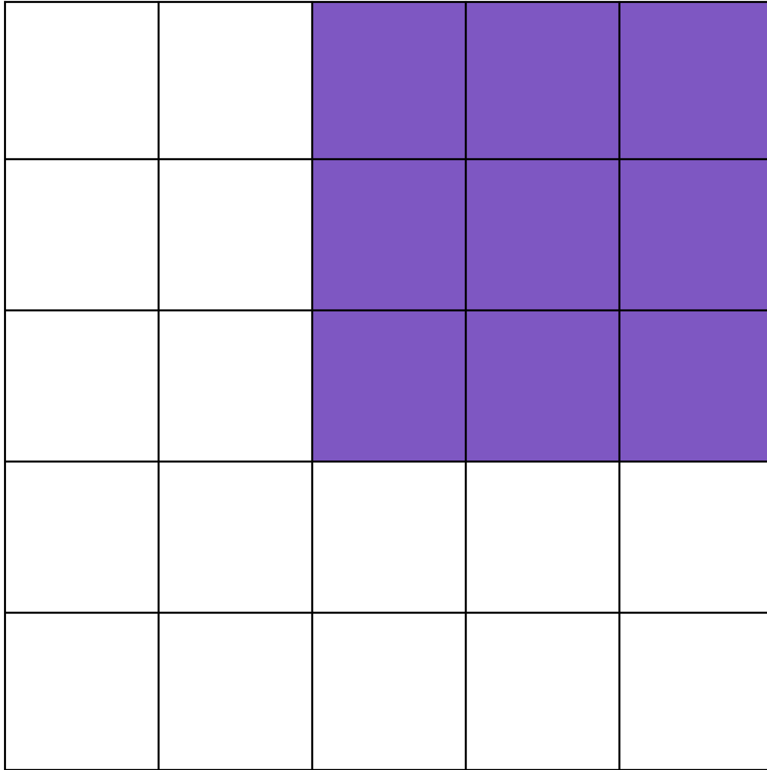
Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3



Input: 5x5x1
Filter: 3x3x1

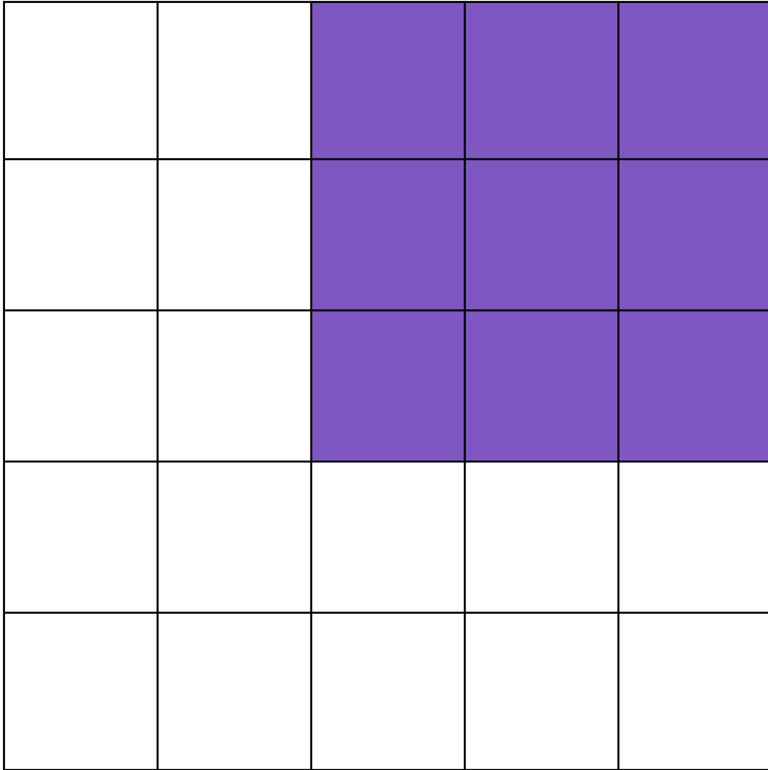
W = 5
F = 3



Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3

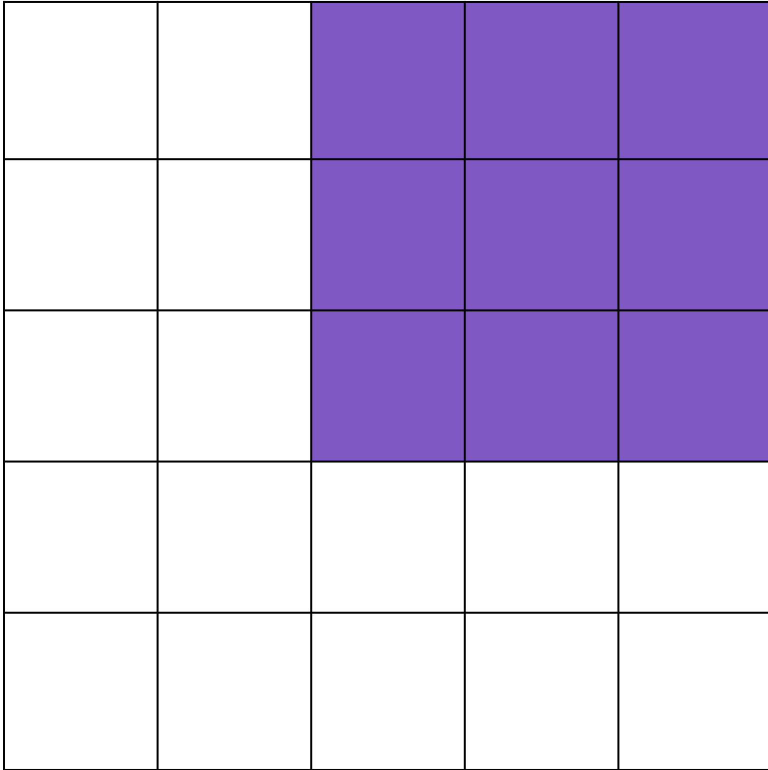
What could a good equation be?



Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3

$$W - F + 1$$

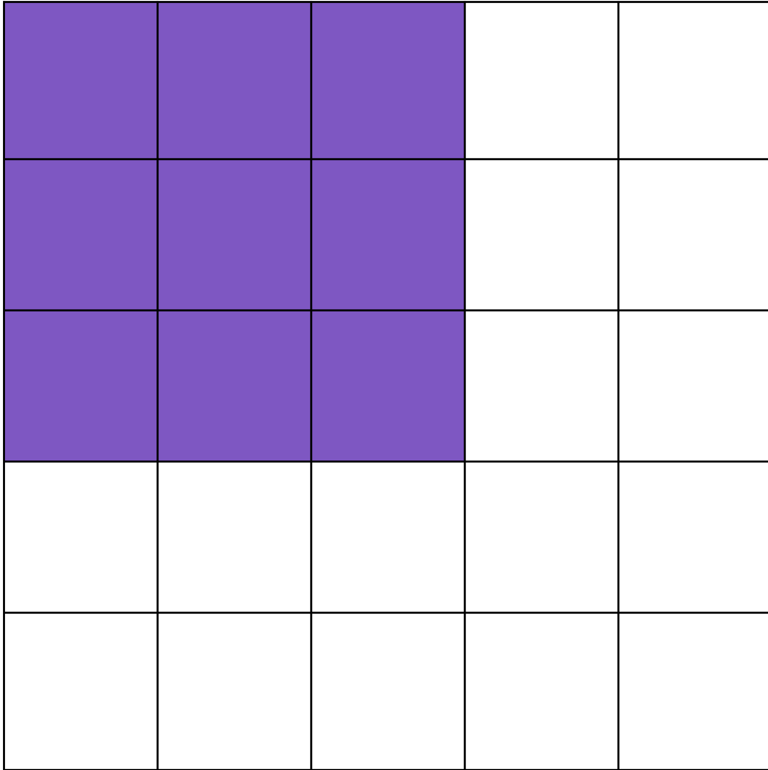


Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3

$$\frac{W - F}{1} + 1$$

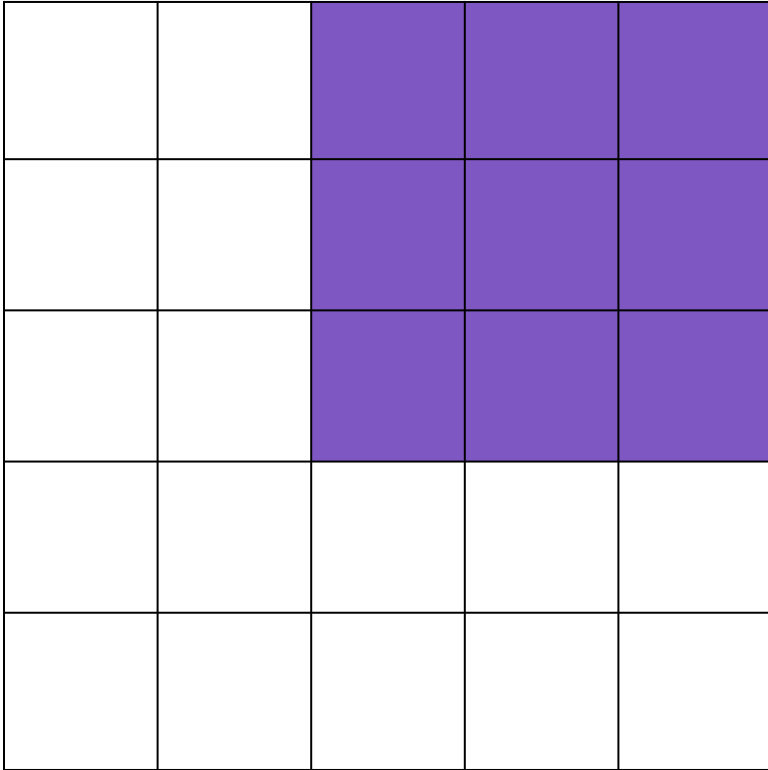
Let’s now consider stride.



Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3
S = 2

$$\frac{W - F}{1} + 1$$



Input: 5x5x1
Filter: 3x3x1

W = 5
F = 3
S = 2

$$\frac{W - F}{1} + 1$$

The math ain't mathing.

Input: 5x5x1
Filter: 3x3x1

This equation gives us 3.

But we just saw that our output will have width 2!

W = 5
F = 3
S = 2

$$\frac{W - F}{1} + 1$$

Here's one way to make the math work.

W = 5
F = 3
S = 2

$$\frac{5-3}{2}+1=1+1=2$$

What changed?

W = 5
F = 3
S = 2

$$\frac{5-3}{1}+1=2+1=3$$

$$\frac{5-3}{2}+1=1+1=2$$

What changed?

W = 5
F = 3
S = 2

$$\frac{5-3}{2}+1 \qquad \frac{W-F}{2}+1$$

2 is our stride!! Let's adjust our equation.

$$\frac{W - F}{2} + 1$$

2 is our stride!! Let's adjust our equation

$$\frac{W - F}{S} + 1$$

Alright great, now what about padding?

Alright great, now what about padding?

Let P represent our padding.

If set P = 1, then we’re adding a zero value on the left and right of our image.

In other words … we’re increasing the width of our image by 2.

Alright great, now what about padding?

Let P represent our padding.

If set P = 1, then we’re adding a zero value on the left and right of our image.

In other words … we’re increasing the width of our image by 2.

Adding 2*P to our value for W should do the trick!

Taking padding into account...

$$\frac{W - F}{S} + 1$$

Taking padding into account...

$$\frac{W - F}{S} + 1$$

$$\frac{W + 2P - F}{S} + 1$$

Our final equation 🙂

$$\frac{W - F + 2P}{S} + 1$$

Remember that conv layers operate along the *entire* depth.

Our final equation 🙂

$$\frac{W - F + 2P}{S} + 1$$

"The connections are local in 2D space (along width and height), but always full along the entire depth of the input volume" ~ cs231n notes

Vectorization

What is vectorization?

A **parallelization** technique to make your code run faster.

A simple example.

$$5 + \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 7 & 8 \end{bmatrix}$$

```
arr = np.array([1, 2, 3])
for i in range(arr.shape[0]):
  arr[i] += 5
```

Naive approach

```
arr = np.array([1, 2, 3])
arr += 5
```

Vectorized approach

(hint: recall broadcasting)

A simple example.

$$5 + \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 7 & 8 \end{bmatrix}$$

```
arr = np.array([1, 2, 3])
for i in range(arr.shape[0]):
  arr[i] += 5
```

Naive approach

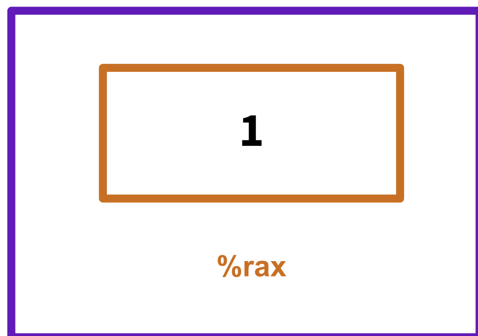
```
arr = np.array([1, 2, 3])
arr += np.array([5, 5, 5])
```

Vectorized approach
(hint: recall broadcasting)

The naive approach.

```
arr = np.array([1, 2, 3])
for i in range(arr.shape[0]):
  arr[i] += 5
```

CPU Registers



Memory

Assembly

Load 1 into rax

`add rax, 5`

Save answer to memory

–

Load 2 into rax

`add rax, 5`

Save answer to memory

–

Load 3 into rax

`add rax, 5`

Save answer to memory

~9 instructions

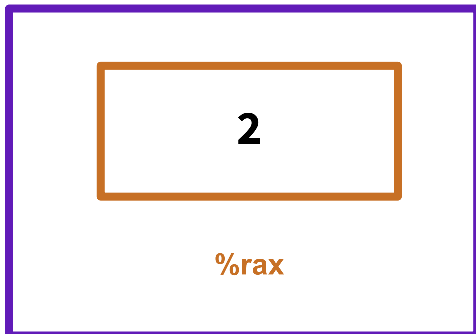
Assumes Intel AVX 2 with 64 bit floats

This is not a perfect comparison

The naive approach.

```
arr = np.array([1, 2, 3])
for i in range(arr.shape[0]):
  arr[i] += 5
```

CPU Registers



Memory

Assembly

Load 1 into rax

`add rax, 5`

Save answer to memory

–

Load 2 into rax

`add rax, 5`

Save answer to memory

–

Load 3 into rax

`add rax, 5`

Save answer to memory

~9 instructions

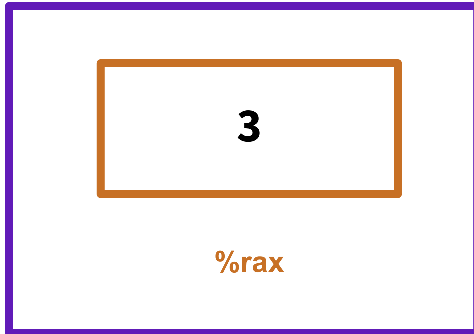
Assumes Intel AVX 2 with 64 bit floats

This is not a perfect comparison

The naive approach.

```
arr = np.array([1, 2, 3])
for i in range(arr.shape[0]):
  arr[i] += 5
```

CPU Registers



Memory

Assembly

Load 1 into rax
`add rax, 5`
Save answer to memory
–
Load 2 into rax
`add rax, 5`
Save answer to memory
–
Load 3 into rax
`add rax, 5`
Save answer to memory

~9 instructions

Assumes Intel AVX 2 with 64 bit floats

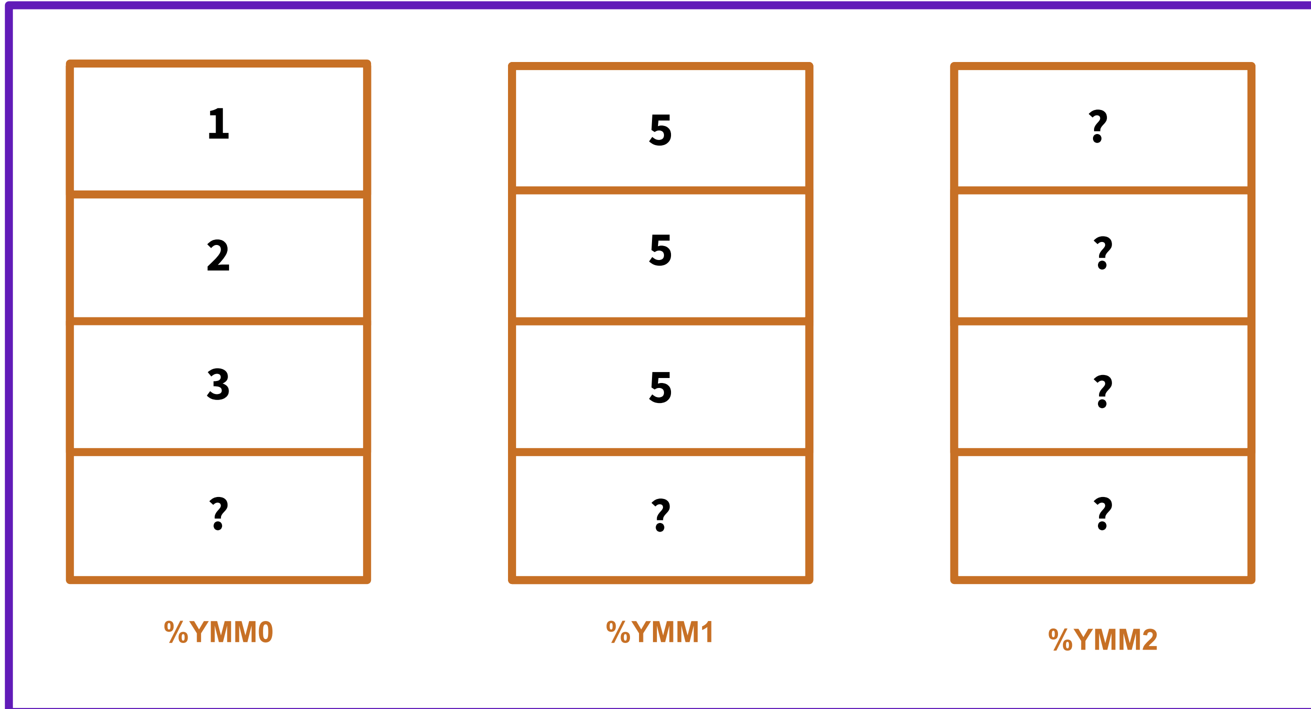
This is not a perfect comparison

Single Instruction, Multiple Data (SIMD)

The vectorized approach.

```
arr = np.array([1, 2, 3])
arr += np.array([5, 5, 5])
```

CPU Registers (*before*)



Assembly

Load array [1,2,3] into YMM0

Load array [5,5,5] into YMM1

`vaddp ymm2, ymm1, ymm2`

Save YMM2 to memory

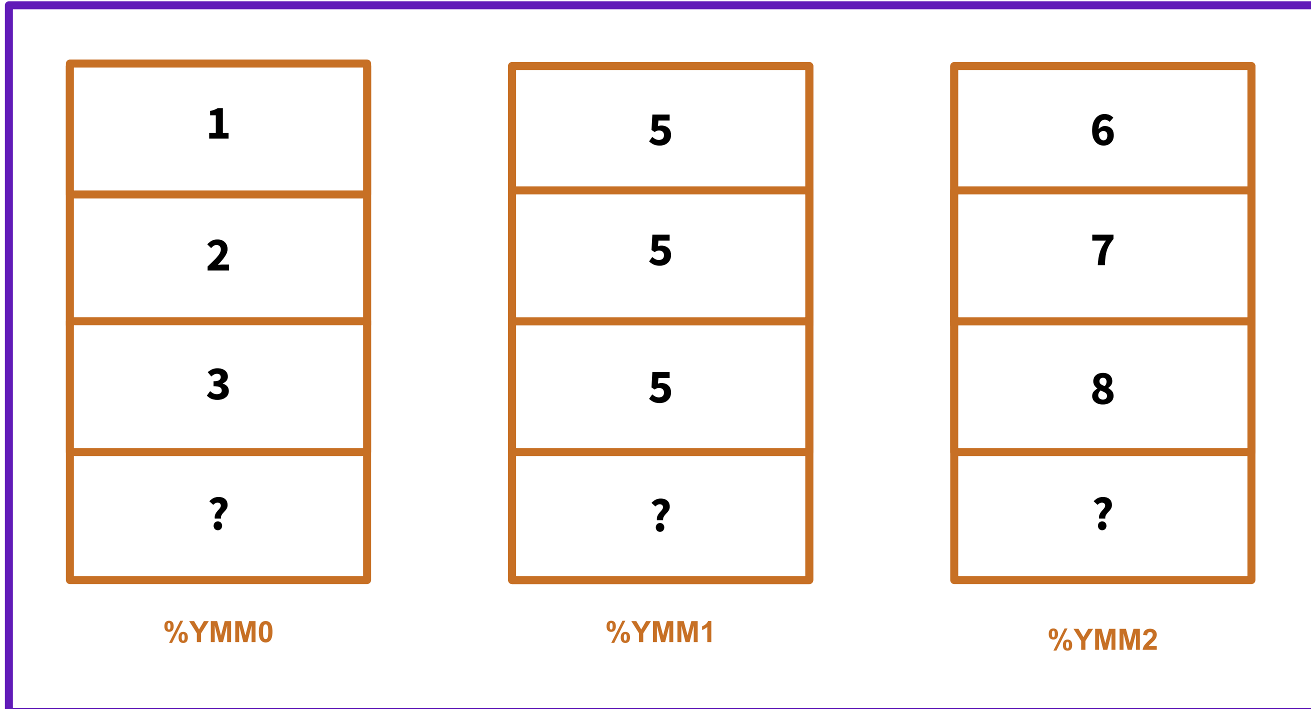
~4 instructions

Assumes Intel AVX 2 with 64 bit floats

This is not a perfect comparison

The vectorized approach.

```
arr = np.array([1, 2, 3])
arr += np.array([5, 5, 5])
```



Assembly

Load array [1,2,3] into YMM0

Load array [5,5,5] into YMM1

`vaddp ymm2, ymm1, ymm2`

Save YMM2 to memory

~4 instructions

Assumes Intel AVX 2 with 64 bit floats

This is not a perfect comparison

This is a generous comparison.

For the naive approach, we didn't take into account…

- The overhead of looping
  - Keeping a counter, incrementing the counter, checking if the loop condition is met, etc.
- Having to load the second array's contents
  - The vectorized approach loaded [5,5,5] into a register, whereas the naive version simply added 5 each time. If we were doing an element-wise addition of two arrays, the naive version would require reading the second array's element from memory.

Counterargument: compilers to the rescue.

A modern C or C++ compiler can vectorize your naive loops for you.

However, in this class (and more generally) you should vectorize your ML code yourself when possible.

- Python interpreters are not as good at optimizing code as C compilers
- NumPy offloads array operations to C
    - Take advantage of this by only working with vectors

A note on GPUs.

The aforementioned optimizations all work on CPUs.

GPUs are wonderful for machine learning (more on that in a moment), but HW1 - HW3 are done in NumPy, which does not support GPU acceleration.

Takeaway: even when we only use CPUs, vectorization is still extremely useful to us.

Vectorization and GPUs go hand-in-hand.

SIMD has been around since 1966.

GPU's take this paradigm to the next level, allowing *even more* parallel processing.

This is inherently conducive to working with matrices, since operations like matmul are just a bunch of small operations that can be done in parallel.

Lots of really smart people have been paid a lot of money to make matrix operations as efficient as possible on GPUs — reap the benefits by vectorizing your code whenever possible

Implementing L1 Distance

Quick review of L1 Distance

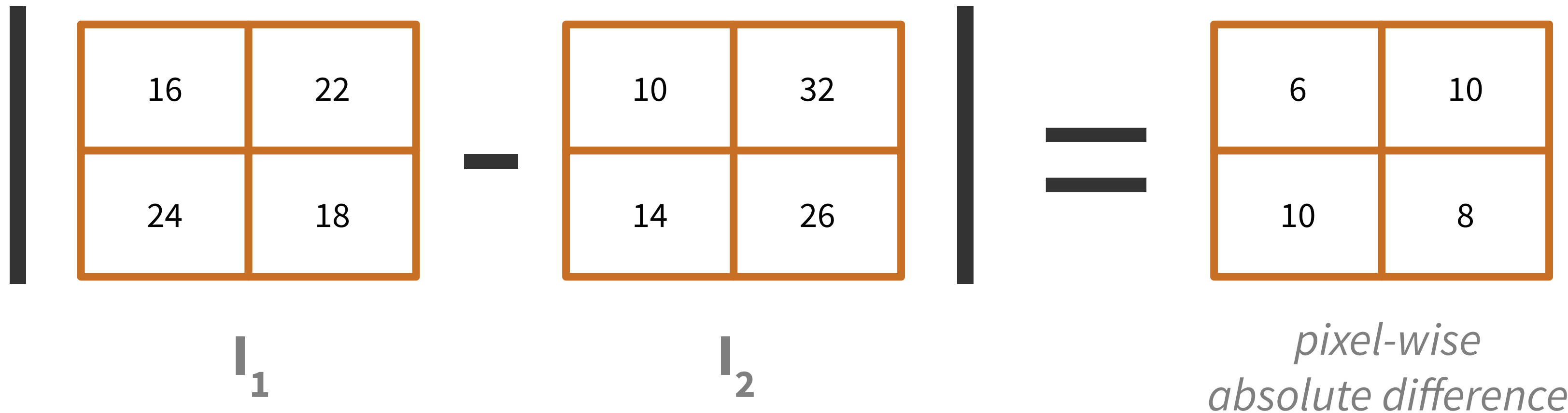
Also called the "Manhattan Distance"

$$d(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

- **d** is a function computing L1 distance
- $\mathbf{I_j}$ is the $\mathbf{j}^{th}$ image
- $\mathbf{\Sigma_p}$ sums across all pixels in an image
- $\mathbf{I_j^k}$ gets the value of the $\mathbf{k}^{th}$ pixel in the the $\mathbf{j}^{th}$ image

Quick review of L1 Distance

$$d(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



Quick review of L1 Distance

$$d(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

d (I₁ , I₂) = 34

I_1

16	22
24	18

I_2

10	32
14	26

$d(I_1, I_2) = 34$

Your Task

Given training data `X_train` and test data `X_test`, compute the L1 distance.

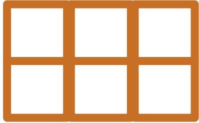
Your results should be stored in the `dists` array, where `dists[i, j]` stores the distance between the `i`th test point and `j`th training point.

In other words, each row `i` will store the L1 distance from test point `i` to all `j` training points.

Visualizing our variables

Say we have 4 training images and 2 test images. Each image consists of 3 pixels*.

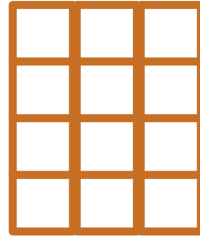
*Assumes each pixel is represented by 1 number.



`X_test`

2 test images → 2 rows

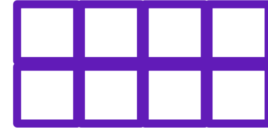
3 pixels → 3 columns



`X_train`

4 training images → 4 rows

3 pixels → 3 columns



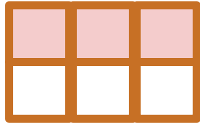
`dists`

2 test images → 2 rows

4 training images → 4 columns

Visualizing our variables

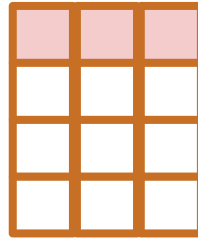
Say we have 4 training images and 2 test images. Each image consists of 3 pixels.



`X_test`

2 test images → 2 rows

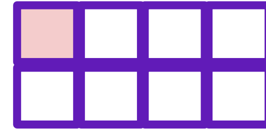
3 pixels → 3 columns



`X_train`

4 training images → 4 rows

3 pixels → 4 columns



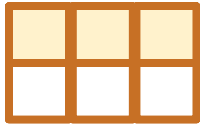
`dists`

2 test images → 2 rows

4 training images → 4 columns

Visualizing our variables

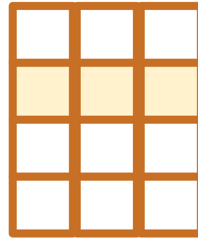
Say we have 4 training images and 2 test images. Each image consists of 3 pixels.



`X_test`

2 test images → 2 rows

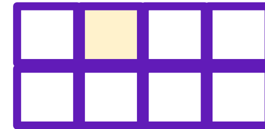
3 pixels → 3 columns



`X_train`

4 training images → 4 rows

3 pixels → 4 columns



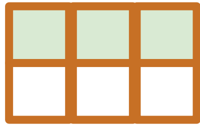
`dists`

2 test images → 2 rows

4 training images → 4 columns

Visualizing our variables

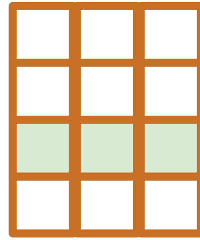
Say we have 4 training images and 2 test images. Each image consists of 3 pixels*.



X_test

2 test images → 2 rows

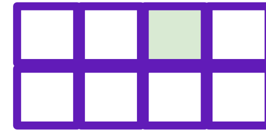
3 pixels → 3 columns



X_train

4 training images → 4 rows

3 pixels → 4 columns



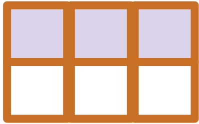
dists

2 test images → 2 rows

4 training images → 4 columns

Visualizing our variables

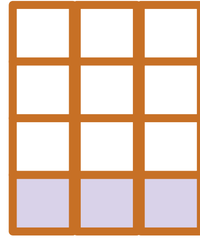
Say we have 4 training images and 2 test images. Each image consists of 3 pixels*.



`X_test`

2 test images → 2 rows

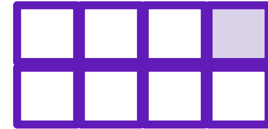
3 pixels → 3 columns



`X_train`

4 training images → 4 rows

3 pixels → 4 columns



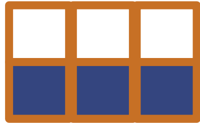
`dists`

2 test images → 2 rows

4 training images → 4 columns

Visualizing our variables

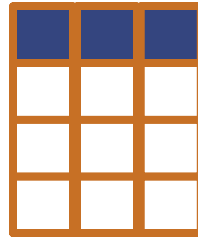
Say we have 4 training images and 2 test images. Each image consists of 3 pixels*.



`X_test`

2 test images → 2 rows

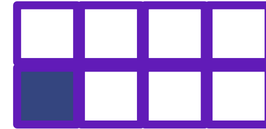
3 pixels → 3 columns



`X_train`

4 training images → 4 rows

3 pixels → 4 columns



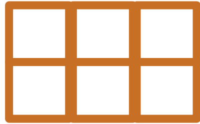
`dists`

2 test images → 2 rows

4 training images → 4 columns

Visualizing our variables

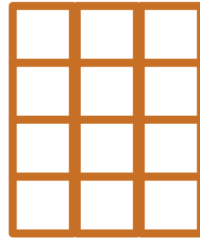
Say we have 4 training images and 2 test images. Each image consists of 3 pixels*.



`X_test`

2 test images → 2 rows

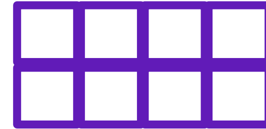
3 pixels → 3 columns



`X_train`

4 training images → 4 rows

3 pixels → 4 columns



`dists`

2 test images → 2 rows

4 training images → 4 columns

Implement L1 naively (hint: 2 `for` loops)

Given training data `X_train` and test data `X_test`, compute the L1 distance.

Your results should be stored in the `dists` array, where `dists[i,j]` stores the distance between the `i`th test point and `j`th training point.

```
X_test = np.random.rand(2, 3)                 # initialize random test points
X_train = np.random.rand(4, 3)                # initialize random training points

num_test = X_test.shape[0]                    # the number of test points
num_train = X_train.shape[0]                  # the number of training points

dists = np.zeros((num_test, num_train))       # initialize dists to an empty (i,j) array

# ***** START OF SOLUTION CODE *****

# ***** END OF SOLUTION CODE *****

return dists
```

Implement L1 naively

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists.`

```
X_test = np.random.rand(2, 3)                    # initialize random test points
X_train = np.random.rand(4, 3)                   # initialize random training points

num_test = X_test.shape[0]                       # the number of test points
num_train = X_train.shape[0]                     # the number of training points

dists = np.zeros((num_test, num_train))          # initialize dists to an empty (i,j) array

# ***** START OF SOLUTION CODE *****

 for i in range(num_test):
    for j in range(num_train):
        # store L1 distance of this unique (test image, training image) pair in dists[i,j]
        pass

# ***** END OF SOLUTION CODE *****

return dists
```

Implement L1 naively

$$d(I_1, I_2) = \sum_p \left| I_1^p - I_2^p \right|$$

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists`.

We can breakdown the L1 calculation into 3 simple steps that we'll repeat for each unique (test image, training image) pair:

1. For each pixel, calculate difference between the test image and training image.

```
raw_diff = X_test[i] - X_train[j]
```

An even more naive approach would loop through each pixel!

2. Take the absolute value of each pixelwise difference.

```
abs_diff = np.abs(raw_diff)
```

3. Add up these difference to arrive at your total "distance".

```
dists[i,j] = np.sum(abs_diff)
```

Implement L1 naively

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists`.

```
X_test = np.random.rand(2, 3)
X_train = np.random.rand(4, 3)

num_test = X_test.shape[0]
num_train = X_train.shape[0]

dists = np.zeros((num_test, num_train))

# ***** START OF SOLUTION CODE *****

for i in range(num_test):
   for j in range(num_train):
     raw_diff = X_test[i] - X_train[j]
     abs_diff = np.abs(raw_diff)
     dists[i,j] = np.sum(abs_diff)

# ***** END OF SOLUTION CODE *****

return dists
```

Implement L1 naively

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists.`

```python
X_test = np.random.rand(2, 3)
X_train = np.random.rand(4, 3)

num_test = X_test.shape[0]
num_train = X_train.shape[0]

dists = np.zeros((num_test, num_train))

# ***** START OF SOLUTION CODE *****

for i in range(num_test):
    for j in range(num_train):
        raw_diff = X_test[i] - X_train[j]
        abs_diff = np.abs(raw_diff)
        dists[i,j] = np.sum(abs_diff)

# ***** END OF SOLUTION CODE *****

return dists
```

```python
X_test = np.random.rand(2, 3)
X_train = np.random.rand(4, 3)

num_test = X_test.shape[0]
num_train = X_train.shape[0]

dists = np.zeros((num_test, num_train))

# ***** START OF SOLUTION CODE *****

for i in range(num_test):
    for j in range(num_train):
        dists[i,j] = np.sum(np.abs(X_test[i] - X_train[j]))

# ***** END OF SOLUTION CODE *****

return dists
```

Implement L1 naively

Given training data X_train and test data X_test, compute the L1 distance and store it in dists.

```
X_test = np.random.rand(2, 3)
X_train = np.random.rand(4, 3)

num_test = X_test.shape[0]
num_train = X_train.shape[0]

dists = np.zeros((num_test, num_train))

# ***** START OF SOLUTION CODE *****

for i in range(num_test):
    for j in range(num_train):
        dists[i,j] = np.sum(np.abs(X_test[i] - X_train[j]))

# ***** END OF SOLUTION CODE *****

return dists
```

Implement L1 with 1 loops & with 0 loops

Given training data X_train and test data X_test, compute the L1 distance and store it in dists.

```
X_test = np.random.rand(2, 3)                   # initialize random test points
X_train = np.random.rand(4, 3)                  # initialize random training points

num_test = X_test.shape[0]                      # the number of test points
num_train = X_train.shape[0]                    # the number of training points

dists = np.zeros((num_test, num_train))         # initialize dists to an empty (i,j) array

# ***** START OF SOLUTION CODE *****

# ***** END OF SOLUTION CODE *****

return dists
```

Hint: Remove one loop instead of comparing one test image to one training image at a time, we compare one test image to **all training images at once**.

Implement L1 with 1 loops

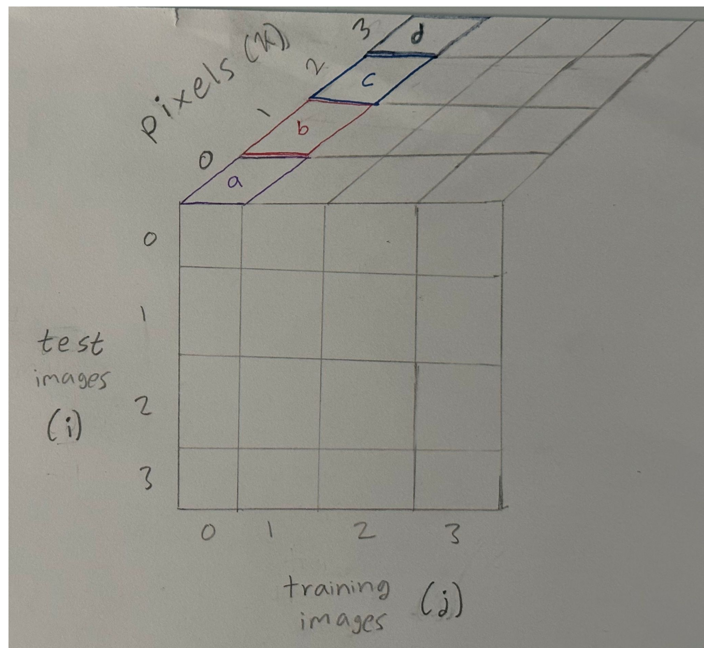
We won’t cover the one-loop version as its very similar to the one-loop L2 calculation from A1, but it is available on the section solutions on Ed Discussion.

Implement L1 with 0 loops

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists`.

Create an intermediary variable where we add another dimension `k`, such that `[i,j,k]` stores the difference between the `k`th pixel of the `i`th test and `j`th training image.

Then, we can pick an `[i,j]` value and sum across all associated `k` values to get the L1 distance between `i` and `j`.

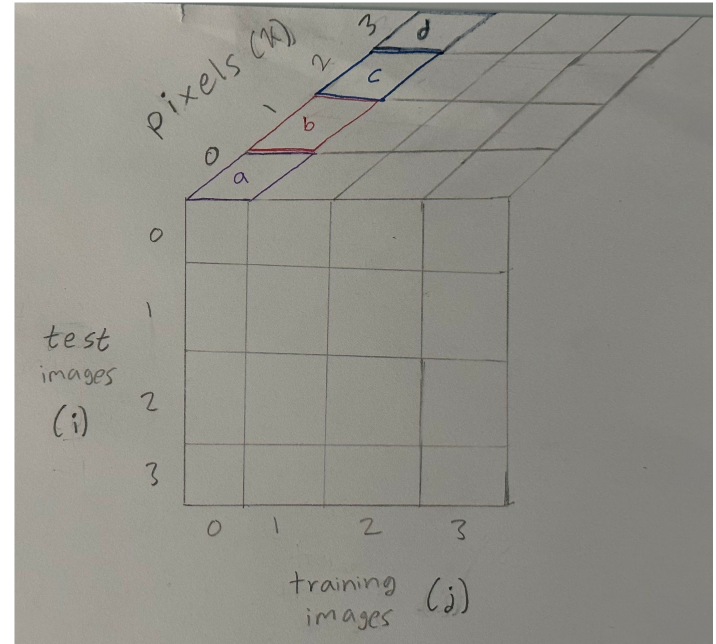


Implement L1 with 0 loops

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists`.

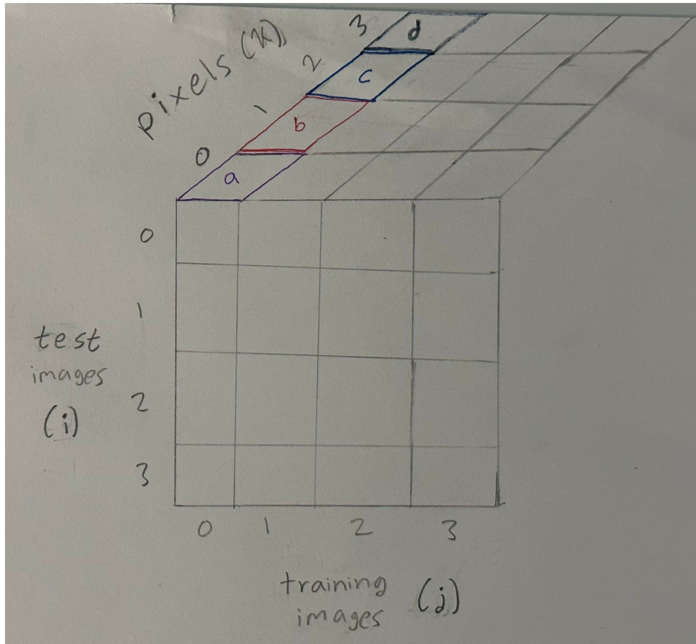
Create an intermediary variable where we add another dimension `k`, such that `[i,j,k]` stores the difference between the `k`th pixel of the `i`th test and `j`th training image.

Then, we can pick an `[i,j]` value and sum across all associated `k` values to get the L1 distance between `i` and `j`.



Implement L1 with 0 loops

Goal: create an intermediary variable where we add another dimension `k`, such that `[i,j,k]` stores the difference between the `k`th pixel of the `i`th test and `j`th training image.



this box stores the difference in this pixel
between these images

label	test img	training img	pixel
a	0	0	0
b	0	0	1
c	0	0	2 (two)
d	0	0	3

Implement L1 with 0 loops

Originally, `X_test.shape = (2, 3)` and `X_train.shape = (4, 3)`

```
X_test[:, np.newaxis, :] - X_train
```

Now, `X_test.shape = (2, 1, 3)` and `X_train.shape = (4, 3)`

Broadcast Rule #1, `X_test.shape = (2, 1, 3)` and `X_train.shape = (1, 4, 3)`

Broadcast Rule #2, `X_test.shape = (2, 4, 3)` and `X_train.shape = (2, 4, 3)`

Final Shape: 2 (test images) x 4 (training images) x 3 (pixels)

$$d(I_1, I_2) = \sum_p \left| I_1^p - I_2^p \right|$$

Implement L1 (naive)

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists`.

We can breakdown the L1 calculation into 3 simple steps **that we'll repeat for each unique (test image, training image) pair**:

1. For each pixel, calculate difference between the test image and training image.

```
raw_diff = X_test[i] - X_train[j]
```

2. Take the absolute value of each pixelwise difference.

```
abs_diff = np.abs(raw_diff)
```

3. Add up these difference to arrive at your total "distance".

```
dists[i,j] = np.sum(abs_diff)
```

Implement L1 (vectorized)

$$d(I_1, I_2) = \sum_p \left|I_1^p - I_2^p\right|$$

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists`.

We can breakdown the L1 calculation into 3 simple steps:

1. For each pixel, calculate difference between the test image and training image.

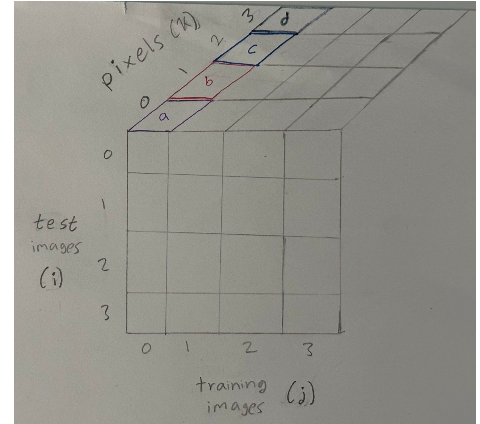
```
raw_diff = X_test[:, np.newaxis, :] - X_train
```

2. Take the absolute value of each pixelwise difference.

```
abs_diff = np.abs(raw_diff)
```

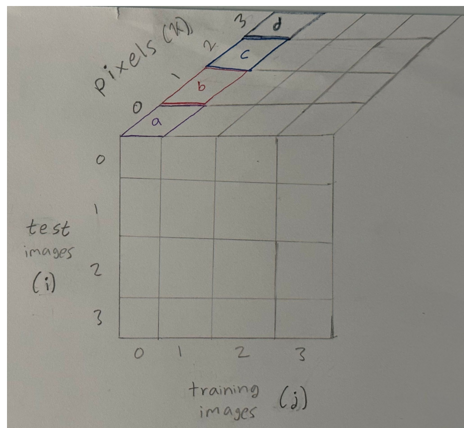
3. Add up these difference to arrive at your total "distance".

```
dists = np.sum(abs_diff, axis=2)
```



Implement L1 with 0 loops

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists.`

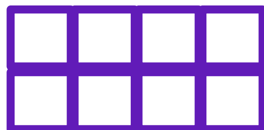



```
X_test = np.random.rand(2, 3)
X_train = np.random.rand(4, 3)

num_test = X_test.shape[0]
num_train = X_train.shape[0]

dists = np.zeros((num_test, num_train))

# ***** START OF SOLUTION CODE *****

abs_diff = np.abs(X_test[:, np.newaxis, :] - X_train)
dists = np.sum(abs_diff, axis=2)

# ***** END OF SOLUTION CODE *****

return dists
```

`X_test[:, None]`

Extension: Farthest Neighbor

For each test image, find the training image it is least similar to.



`dists`

2 test images → 2 rows

4 training images → 4 columns

Naive

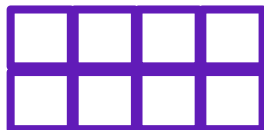
```
farthest = np.zeros(num_test)

for i in range(num_test):
     farthest[i] = np.max(dists[i])
```

Vectorized

```
farthest = np.max(dists, axis = 1)
```

Extension: Farthest Neighbor

For each test image, find the training image it is least similar to.



`dists`

2 test images → 2 rows

4 training images → 4 columns

Vectorized

```
farthest = np.max(dists, axis = 1)
```

Why Axis = 1?

Recall that the axis with index 1 represents the columns.

1. For each test image, we are trying to find the training image (i.e., the column) with the highest value; in other words, we are taking the max along the columns.

2. After our operation we are getting rid of the columns and preserving the rows. Thus, we are taking the max along the columns.

L1 is much faster when vectorized.

Given training data `X_train` and test data `X_test`, compute the L1 distance and store it in `dists`.

In [38]:

```
# This is an amalgamation of solution code from Questions 2 and 3.
# Please do not waste time re-reading it. Instead, focus on internalizing how important vectorization is.

import time

# Naive
dists = np.zeros([num_test, num_train])
tic = time.time()
for i in range(num_test):
    for j in range(num_train):
        dists[i,j] = np.sum( np.abs(X_test[i] - X_train[j]) )
farthest = np.zeros(num_test)
for i in range(num_test):
    farthest[i] = np.max(dists[i])
toc = time.time()
print('Naive version took %d microseconds' % int(1e6*(toc - tic)))

# Vectorized
tic = time.time()
abs_diff = np.abs(X_test[:, np.newaxis] - X_train)
dists = np.sum(abs_diff, axis=2)
farthest = np.max(dists, axis = 1)
toc = time.time()
print('Vectorized version took %d microseconds' % int(1e6*(toc - tic)))
```

Out [38]:

```
Naive version took 221 microseconds
Vectorized version took 90 microseconds
```

Naive: 221 microseconds

Vectorized: 90 microseconds

An alarm should go off in your head every time you see a `for` loop.

They're not always avoidable, but getting rid of them via vectorization is something you should always try to do.

Questions?

Thanks for coming to section!