



Нейронні мережі

Лекція 5: Згорткові мережі

Кочура Юрій Петрович
iuriy.kochura@gmail.com
[@y_kochura](https://twitter.com/y_kochura)

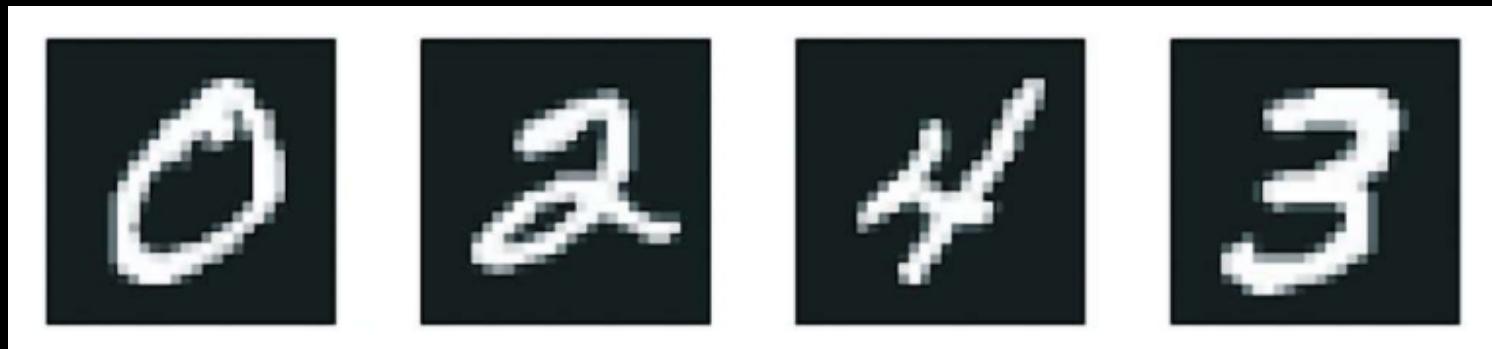
Сьогодні

Розуміння згорткових нейронних мереж (convnets або CNNs):

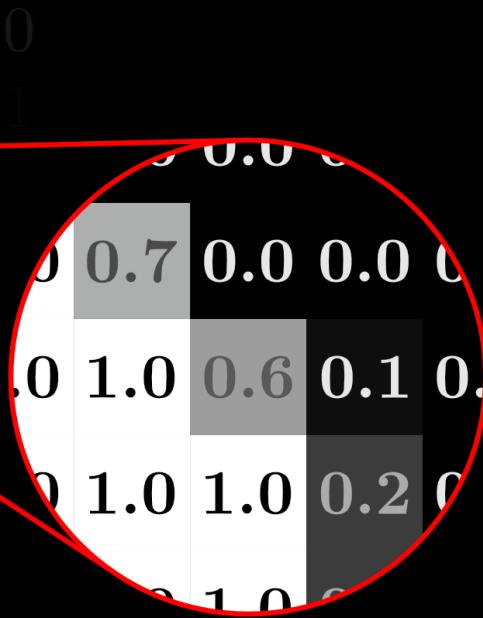
- Повнозв'язна мережа
- Згорткова мережа:
 1. Візуальне сприйняття
 2. Операція згортки
 3. Крок згортки
 4. Ефект доповнення (padding)
 5. Операція агрегації (pooling)

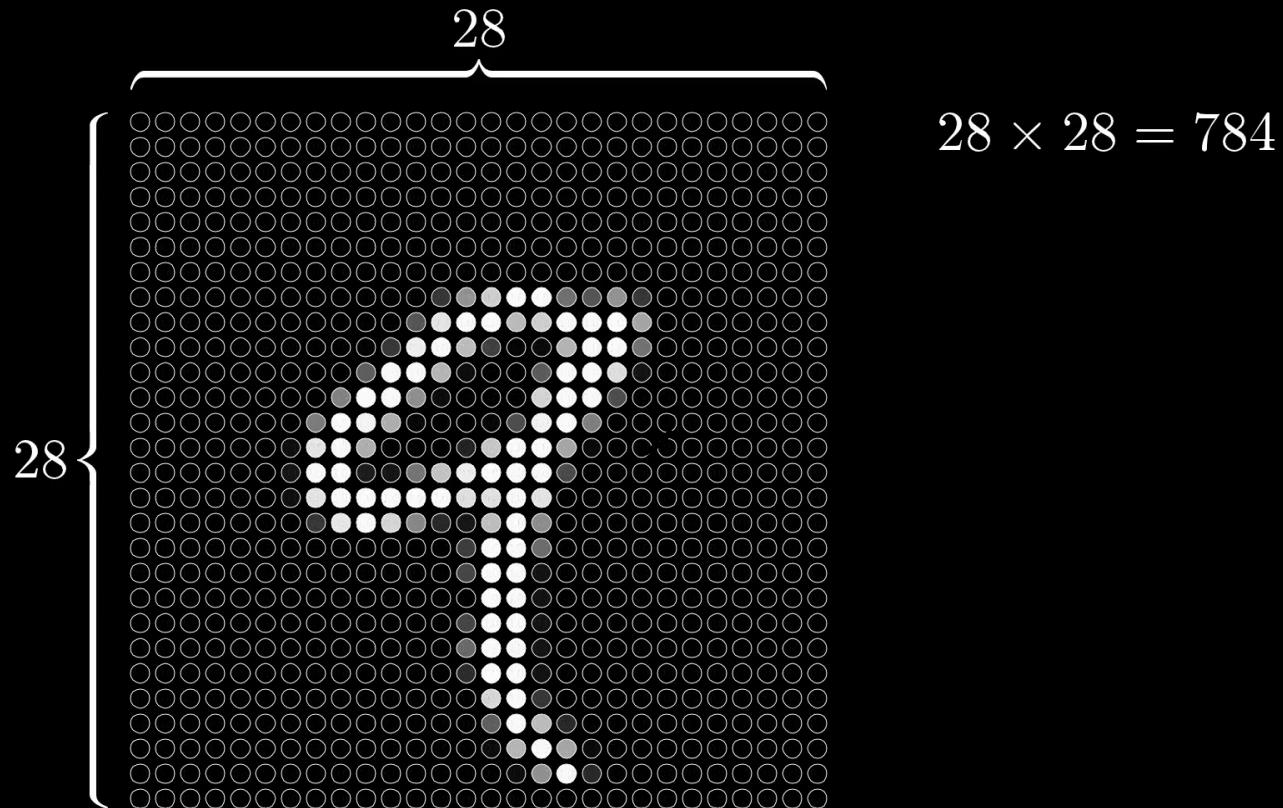
Повнозв'язна мережа

MNIST: приклади



Примітка! У машинному навчанні для задачі класифікації категорія даних називається **класом**. Кожна одиниця даних називається **прикладом**. Клас, який пов'язаний із певним прикладом називається **міткою (label)**.





▶ 0:00 / 0:09

🔊 ⏸ ⏹

Імпортування набору даних MNIST у Keras

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.lo
```

- `train_images` та `train_labels` – навчальний набір даних (дані на яких модель буде навчатись)
- `test_images` та `test_labels` – тестовий набір (дані на яких буде оцінено продуктивність моделі)

Навчальний набір

```
train_images.shape
```

```
(60000, 28, 28)
```

```
len(train_labels)
```

```
60000
```

```
train_labels
```

```
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

Тестовий набір

```
test_images.shape
```

```
(10000, 28, 28)
```

```
len(test_labels)
```

```
10000
```

```
test_labels
```

```
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Архітектура мережі

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
    layers.Dense(512, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

Робочий процес буде таким: спочатку ми передамо нейронній мережі навчальні дані `train_images` та `train_labels`. Таким чином мережа навчиться пов'язувати зображення з мітками. Потім ми попросимо мережу створити прогнози для `test_images` та перевіримо, чи відповідають ці прогнози міткам з `test_labels`.

Готуємо мережу до навчання

Щоб підготувати мережу до навчання, нам потрібно визначити на етапі компіляції:

- Оптимізатор – алгоритм за допомогою якого модель оновлюватиметься на основі навчальних даних, які надаються моделі для покращення свої продуктивності.
- Функція втрат – спосіб виміру втрат моделі. Оптимізатор намагається мінімізувати втрати моделі.
- Метрики для моніторингу під час навчання та тестування – у цій задачі ми слідкуватимо за точністю (відсоток зображень, які були правильно класифіковані).

Компіляція моделі

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=[ "accuracy" ])
```

Підготовка даних

```
train_images = train_images.reshape(60000, 28 * 28)
train_images = train_images.astype("float32") / 255

test_images = test_images.reshape(10000, 28 * 28)
test_images = test_images.astype("float32") / 255
```

Раніше наші навчальні зображення зберігалися в масиві $(60000, 28, 28)$ типу `uint8` зі значеннями в інтервалі $[0, 255]$. Ми перетворюємо його на масив `float32` форми $(60000, 28 * 28)$ зі значеннями від `0` до `1`. Тестовий набір даних перетворюємо аналогічним чином.

Навчання моделі

```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
Epoch 1/5  
469/469 [=====] - 3s 5ms/step - loss: 0.2  
Epoch 2/5  
469/469 [=====] - 2s 5ms/step - loss: 0.10  
Epoch 3/5  
469/469 [=====] - 2s 5ms/step - loss: 0.01  
Epoch 4/5  
469/469 [=====] - 3s 6ms/step - loss: 0.00  
Epoch 5/5  
469/469 [=====] - 3s 6ms/step - loss: 0.01  
  
<keras.src.callbacks.History at 0x7e81cba25e70>
```

Виконання прогнозу

```
test_digits = test_images[0:10]
prediction = model.predict(test_digits)
prediction[0]
```

```
array([3.9224375e-08, 4.9862869e-09, 9.4487259e-06, 8.6249776e-05,
       2.9801717e-11, 3.3959207e-08, 4.4558798e-13, 9.9990362e-01,
       1.0565784e-07, 3.3271664e-07], dtype=float32)
```

Кожне індекс i в цьому масиві відповідає ймовірності того, що наше тестове зображення `test_digits[0]` належить до класу i .

Виконання прогнозу

Перше тестове зображення має найбільшу ймовірність (`0.99990362`, майже 1) для індекса масива `7`, тому відповідно до цього прогнозу моделі це має бути `7`:

```
prediction[0].argmax()
```

7

```
prediction[0][7]
```

0.9999036

Перевіримо на відповідність тестовій мітці:

```
test_labels[0]
```

7

Оцінка моделі на нових даних

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.0
```

```
print(f"test accuracy: {test_acc}")
```

```
test accuracy: 0.9817000031471252
```

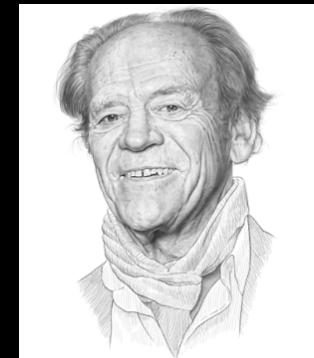
Згорткові мережі

Візуальне сприйняття

У 1959-1962 роках Девід Г'юбел і Торстен Візел зробили відкриття, що стосується принципів переробки інформації в нейронних структурах. За своє відкриття вони отримали Нобелівську премію з медицини в 1981 році.



Девід Г'юбел



Торстен Візел





Hubel and Wiesel Cat Experiment



Share





04 2 Simple Complex Cells



Share

0H09M41S



SP



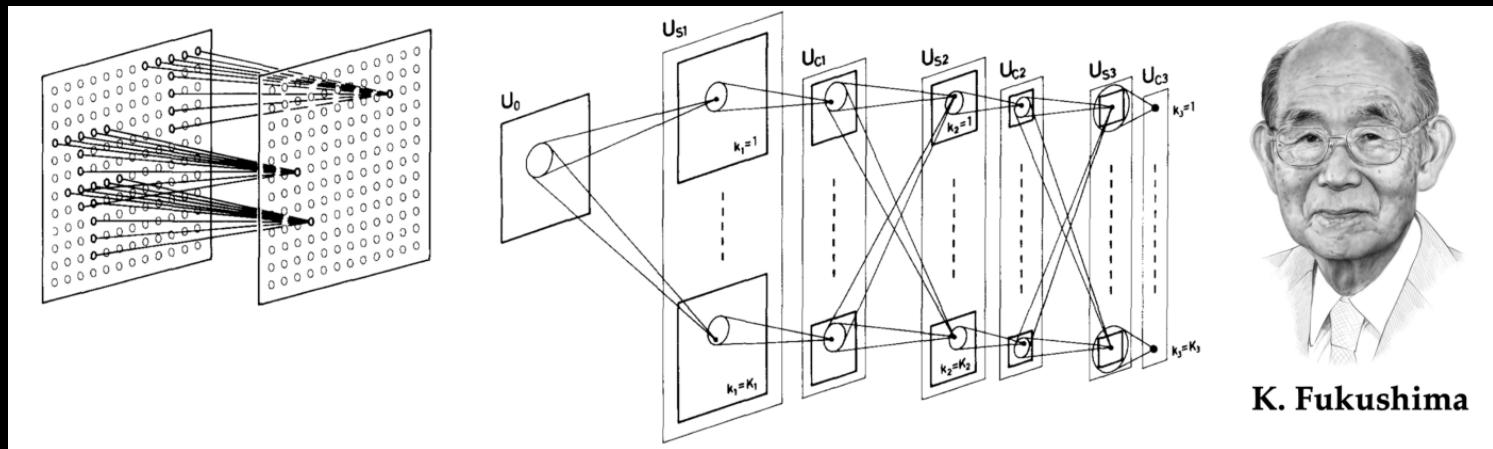
Індуктивний зсув

Чи можемо ми оснастити нейронні мережі **індуктивними зсувами**, характерними для зорової системи?

- Локальність (як у простих нейронів)
- Інваріантність зміщень (як у складних нейронах)
- Ієрархічна композиційність (як у гіперкомплексних нейронах)

Неокогніtron

У 1979 році Фукусіма пропонує реалізацію ієрархічної багатошарової штучної нейронної мережі для моделі зорової нервової системи Г'юбеля та Візеля.

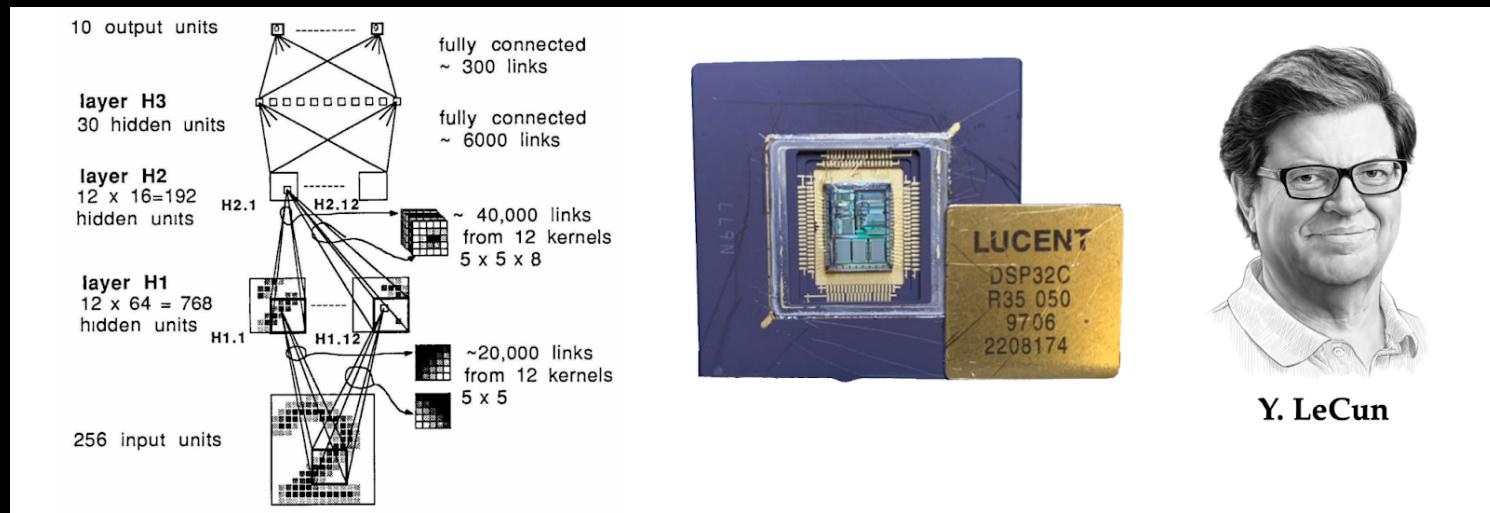


Куніхіко Фукусіма та неокогніtron, рання геометрична архітектура глибокого навчання та попередник сучасних згорткових нейронних мереж

- Побудований на основі **згорток** і дозволяє створювати **ієрархію ознак**.

LeNet-1

У 1989 році Лекун навчає згорткову мережу методом зворотного поширення.



Джерело: Towards Geometric Deep Learning



Convolutional Network Demo from 1989



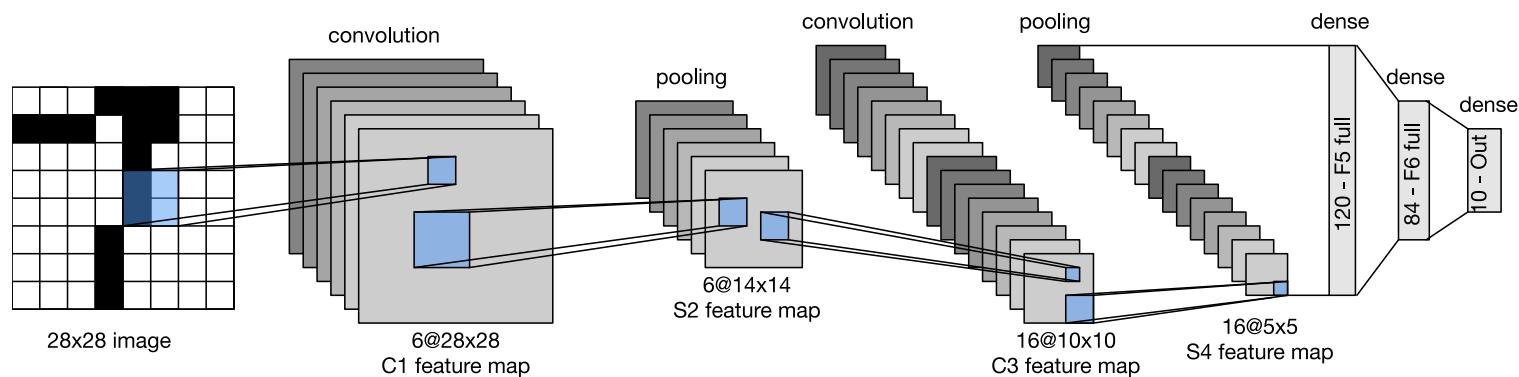
Share



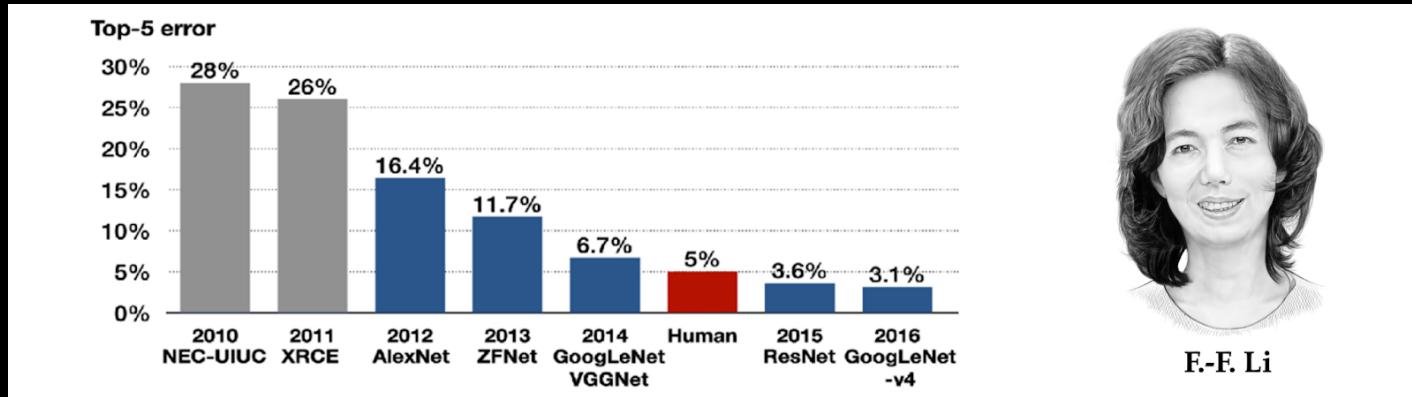
LeNet-1 (LeCun et al, 1993)

LeNet-5 (LeCun et al, 1998)

Композиція з двох шарів **CONV + POOL** за якими йде блок повністю пов'язаних шарів.



Тріумф глибокого навчання



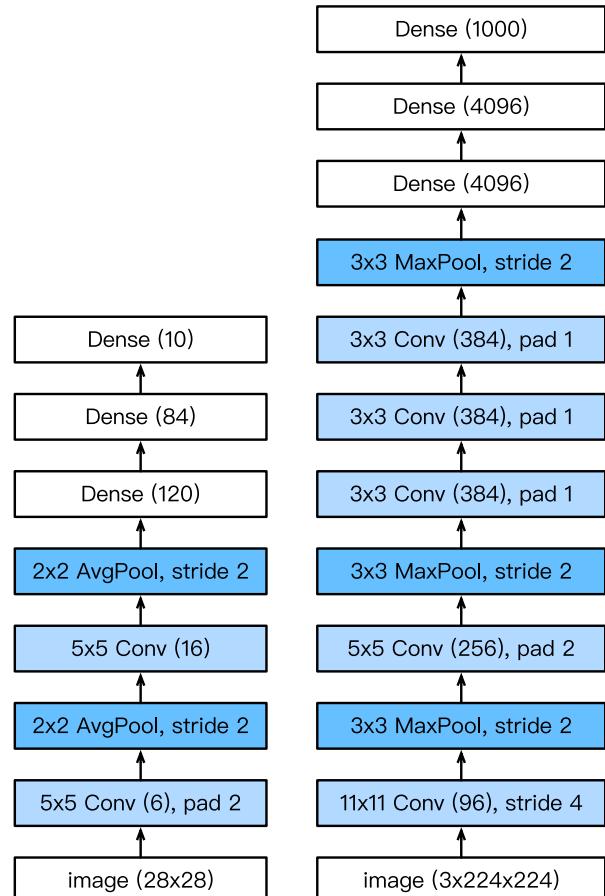
Результати ImageNet Large Scale Visual Recognition Challenge (ILSVRC). У 2012 році AlexNet стала першою архітектурою глибокого навчання, яка перевершила підходи, що базувались на «ручному» видобуванню ознак; відтоді всі методи-переможці базувалися на глибокому навченні.



AlexNet ([Krizhevsky et al, 2012](#))

Композиція 8-шарової згорткової нейронної мережі з 3-шарами MLP.

Оригінальна реалізація складалася з двох частин, щоб вона могла поміститися в два графічні прискорювачі GeForce NVIDIA.

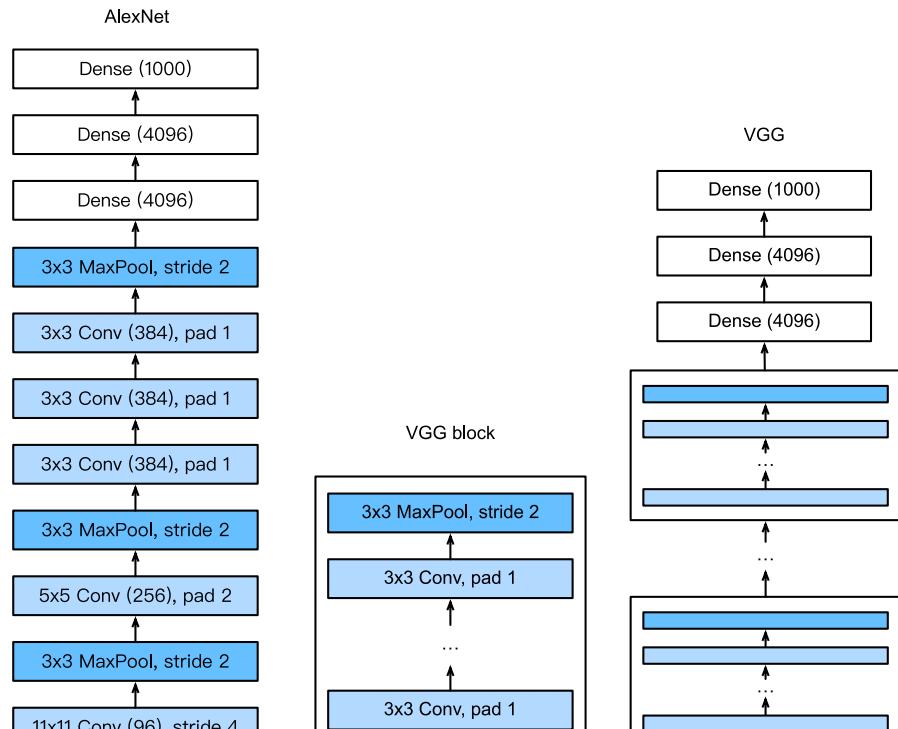


LeNet vs. AlexNet

VGG (Simonyan and Zisserman, 2014)

Композиція з 5 VGG блоків, що складається з шарів CONV + POOL, за якими йде блок повністю звязаних шарів.

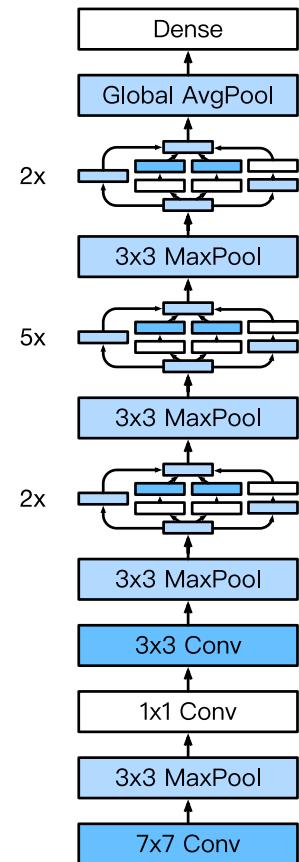
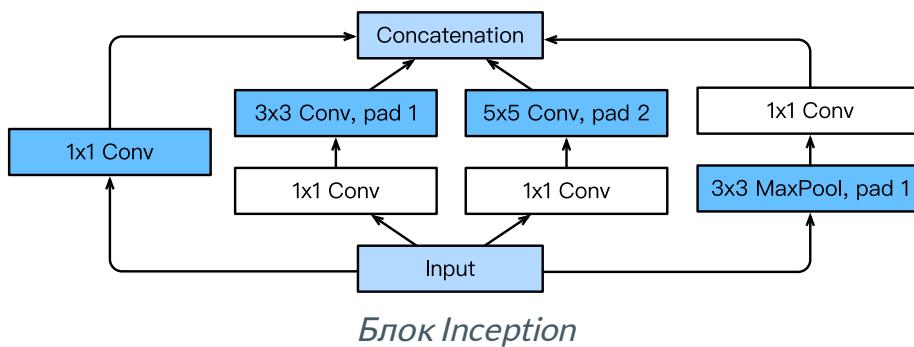
Глибина мережі зросла до 19 шарів, а розмір ядра зменшився до 3.



GoogLeNet (Szegedy et al, 2014)

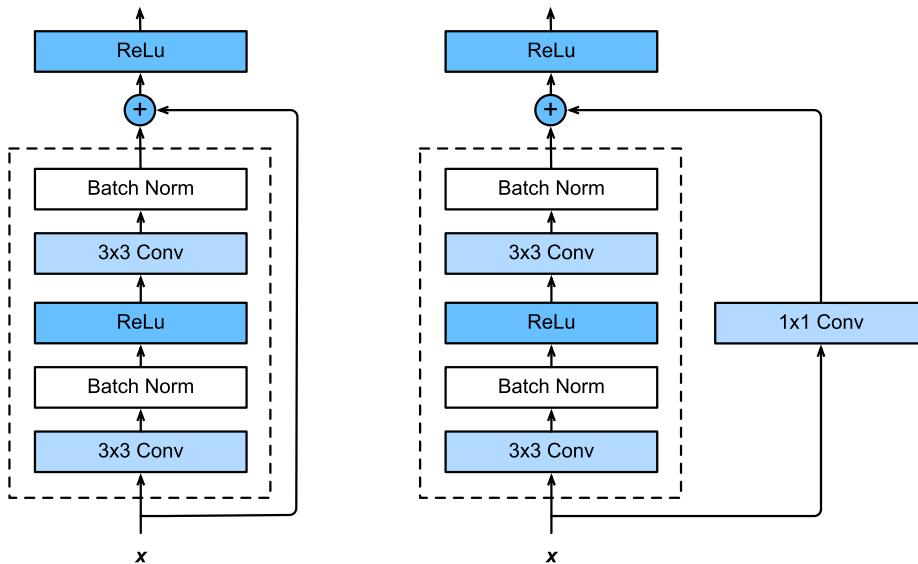
Композиція з двох шарів **CONV + POOL**, стеку з 9 блоків **Inception** і глобального об'єднання за середнім.

Кожен **Inception** блок сам по собі визначається як згорткова мережа з 4 паралельними шляхами.

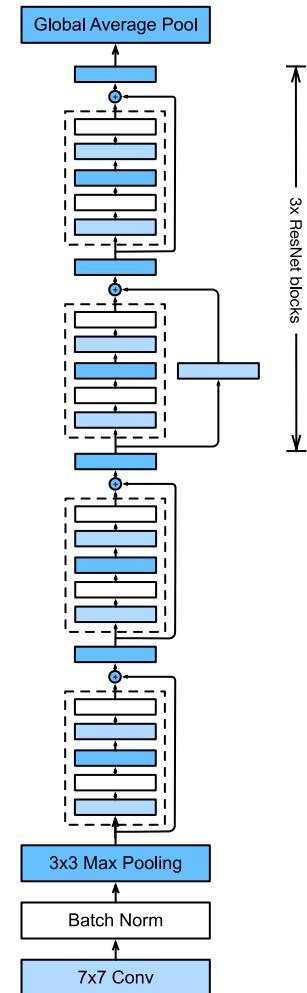


ResNet (He et al, 2015)

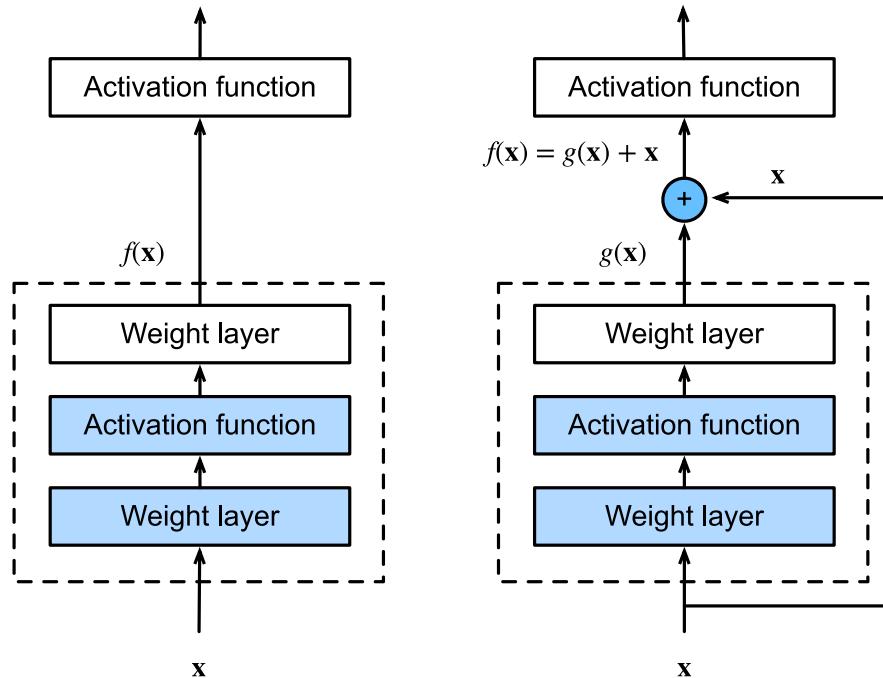
Композиція шарів **CONV**, організованих у стек з блоків **residual**. Можна додавати більше блоків **residual**, загалом до 152 шарів (ResNet-152).



ResNet vs. ResNet з 1×1 згорткою.



Навчальні мережі такої глибини стали можливими завдяки пропущеним з'єднанням $f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x})$ у блоках **residual**. Вони дозволяють градієнтам обходити шари і проходити крізь мережу, не зникаючи.



У звичайному блокі (ліворуч) частина в полі з пунктирною лінією має безпосередньо вивчати відображення $f(\mathbf{x})$. У залишковому блокі (справа) частина в межах пунктирної лінії має вивчити залишкове відображення $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$, що полегшує вивчення відображення ідентичності $f(\mathbf{x}) = \mathbf{x}$.

Створюємо згорткову мережу

```
from tensorflow import keras
from tensorflow.keras import layers
input = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu") (input)
x = layers.MaxPool2D(pool_size=2) (x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu") (x)
x = layers.MaxPool2D(pool_size=2) (x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu") (x)
x = layers.Flatten() (x)
output = layers.Dense(10, activation="softmax") (x)
model = keras.Model(inputs=input, outputs=output)
```

Важливо, що convnet приймає на вхід тензори форми (`image_height`, `image_width`, `image_channels`), не включаючи розмірність пакету. У цьому випадку ми налаштуємо convnet для обробки вхідних даних розміром `(28, 28, 1)`, що відповідає формату зображень MNIST.

```
model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_2 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_3 (Conv2D)	(None, 3, 3, 128)	73856
flatten (Flatten)	(None, 1152)	0
dense (Dense)	(None, 10)	11530
<hr/>		

Total params: 104202 (407.04 KB)

Trainable params: 104202 (407.04 KB)

Non-trainable params: 0 (0.00 Byte)

Підготовка даних, компіляція та навчання моделі

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape(60000, 28, 28, 1)
train_images = train_images.astype("float32") / 255

test_images = test_images.reshape(10000, 28, 28, 1)
test_images = test_images.astype("float32") / 255

model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Оцінка згорткової моделі на нових даних

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

```
313/313 [=====] - 2s 6ms/step - loss: 0.01
```

```
print(f"test accuracy: {test_acc}")
```

```
test accuracy: 0.991100013256073
```

Будівельні блоки

Тензор (**tensor**)

масив чисел, розташованих у сітці зі змінною кількістю осей





0D tensor

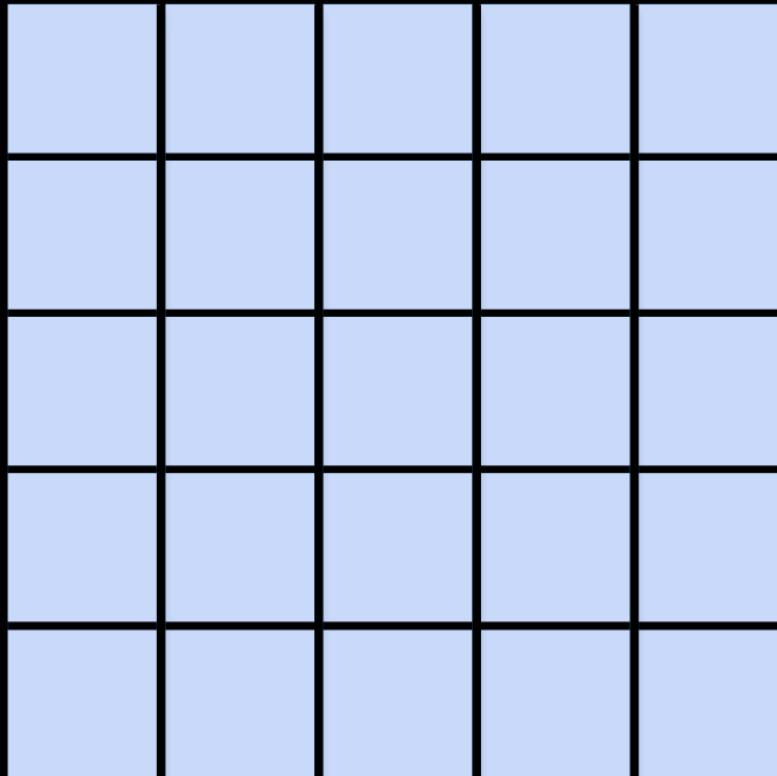


$\text{rows} \times 1$

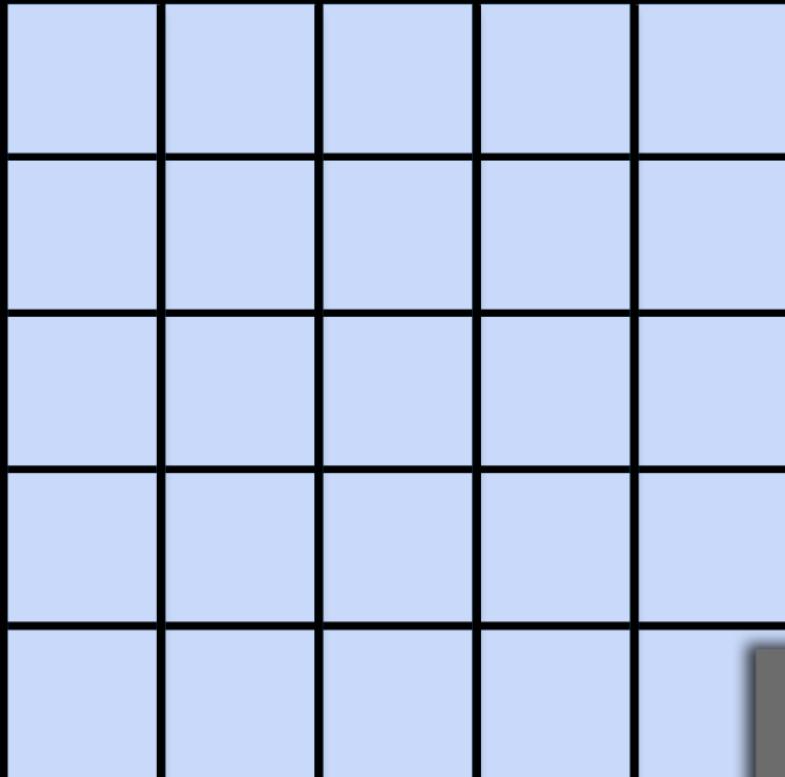


1D tensor

rows × 1

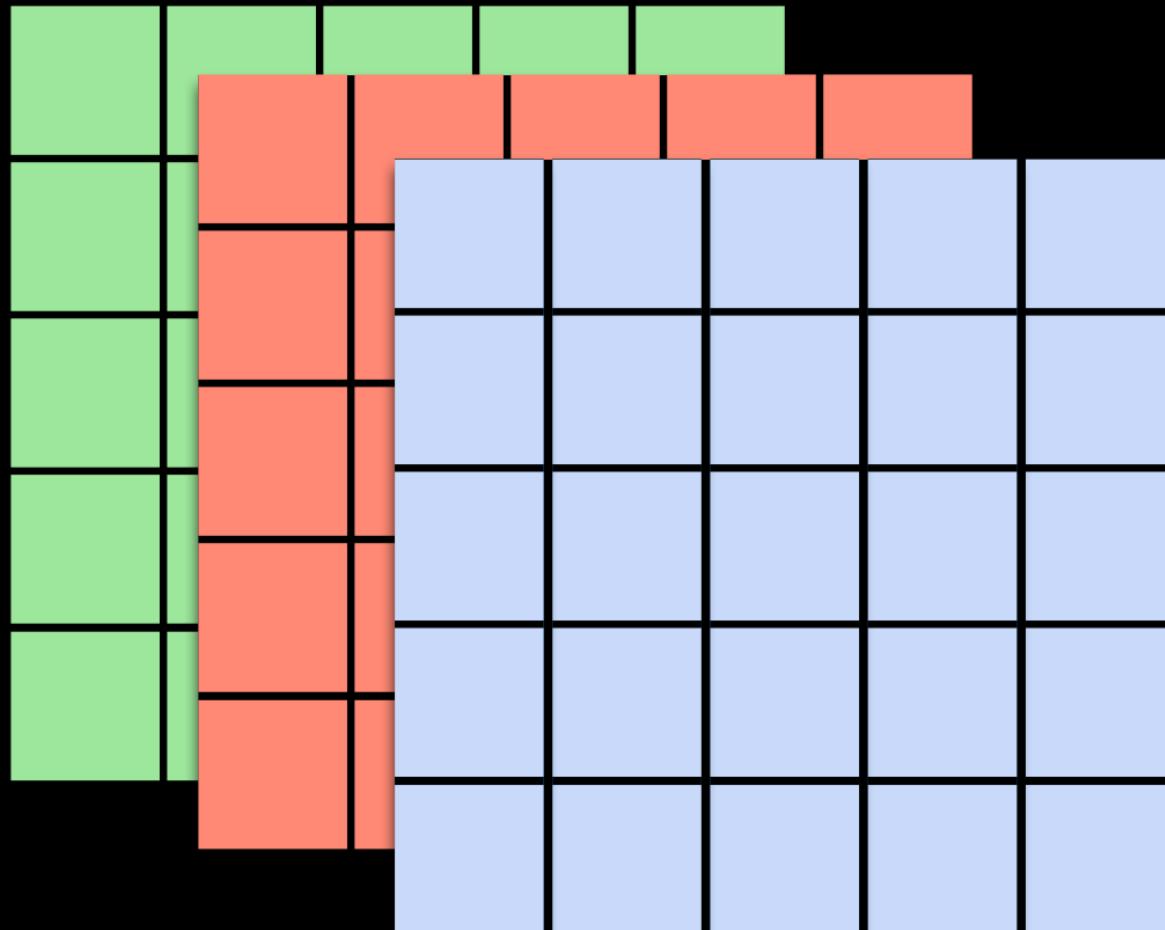


$\text{rows} \times \text{cols}$

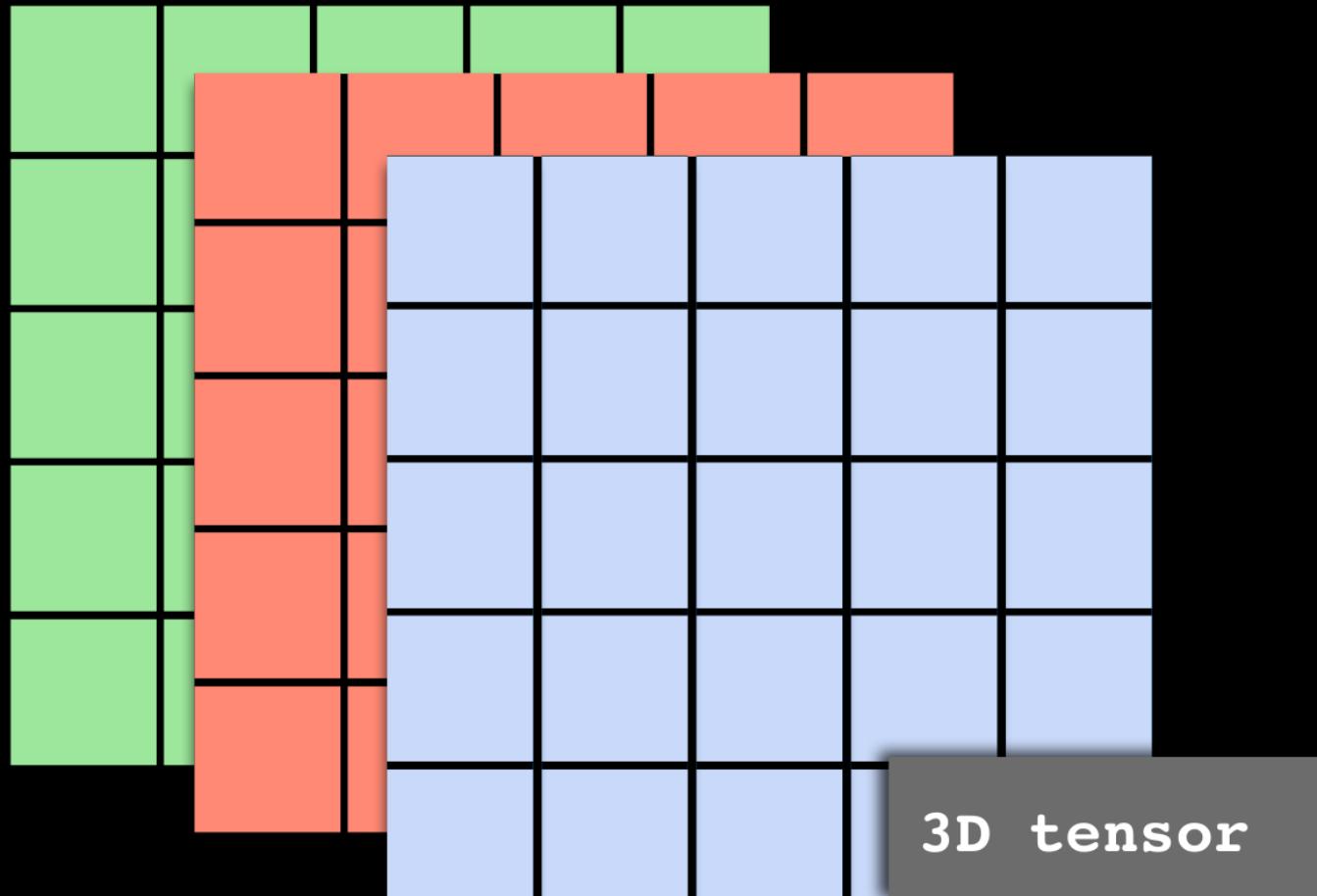


2D tensor

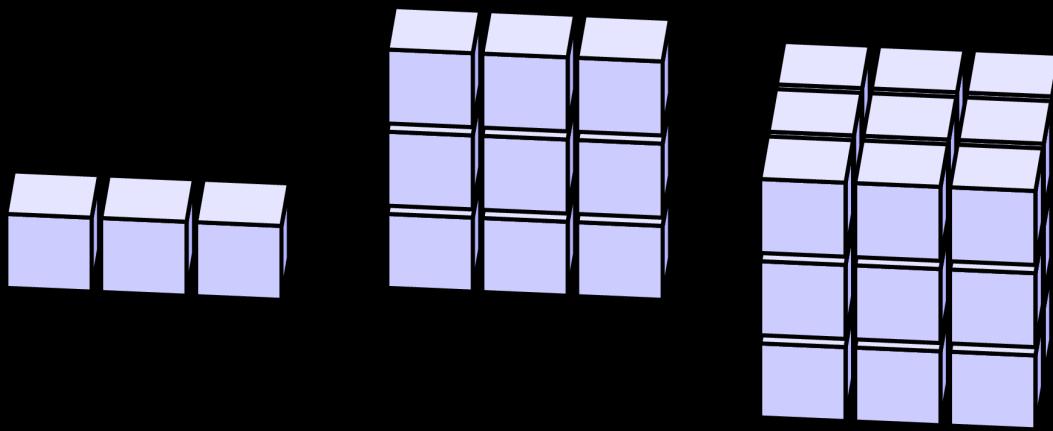
rows × cols



rows \times cols \times channels

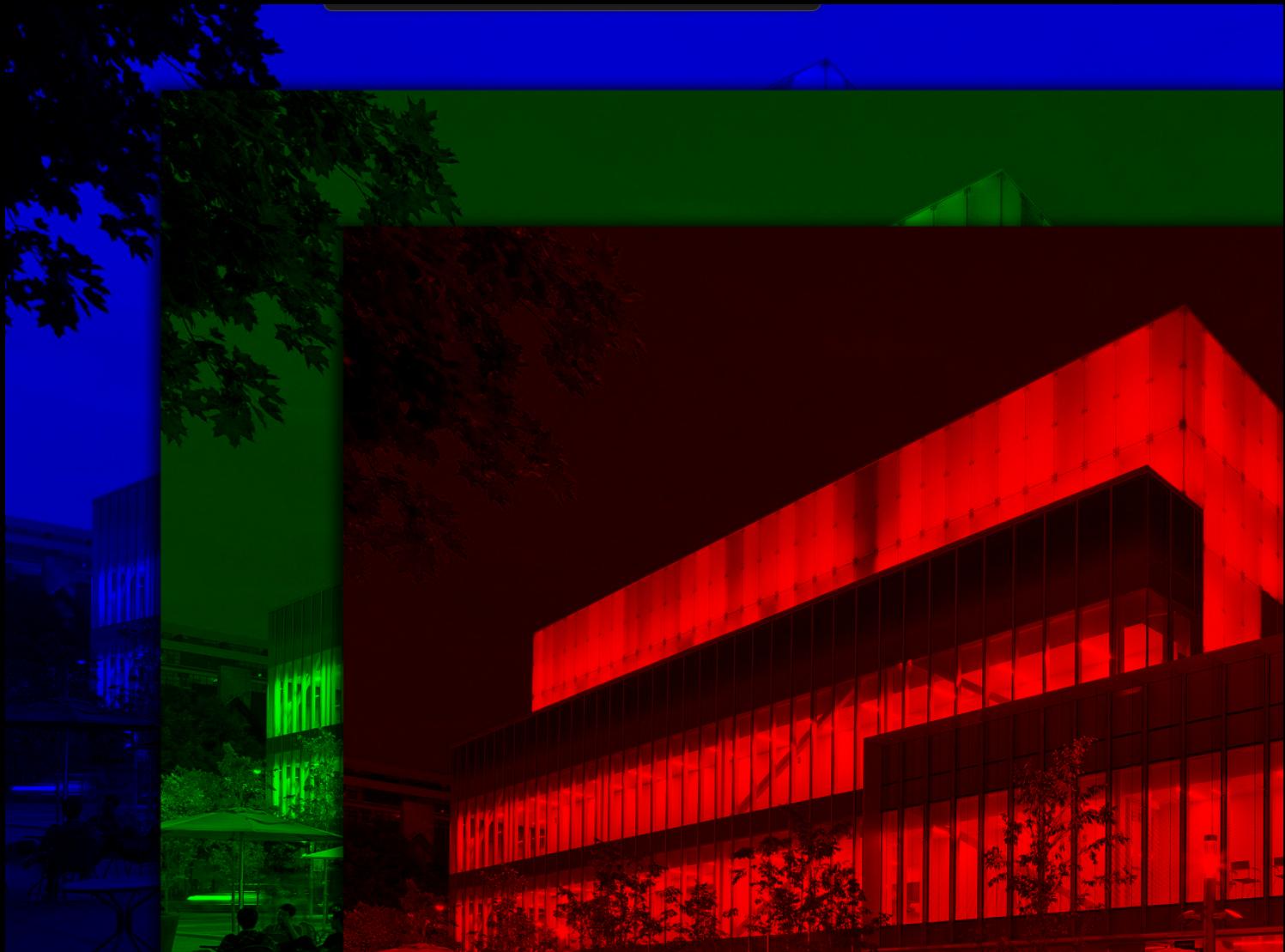


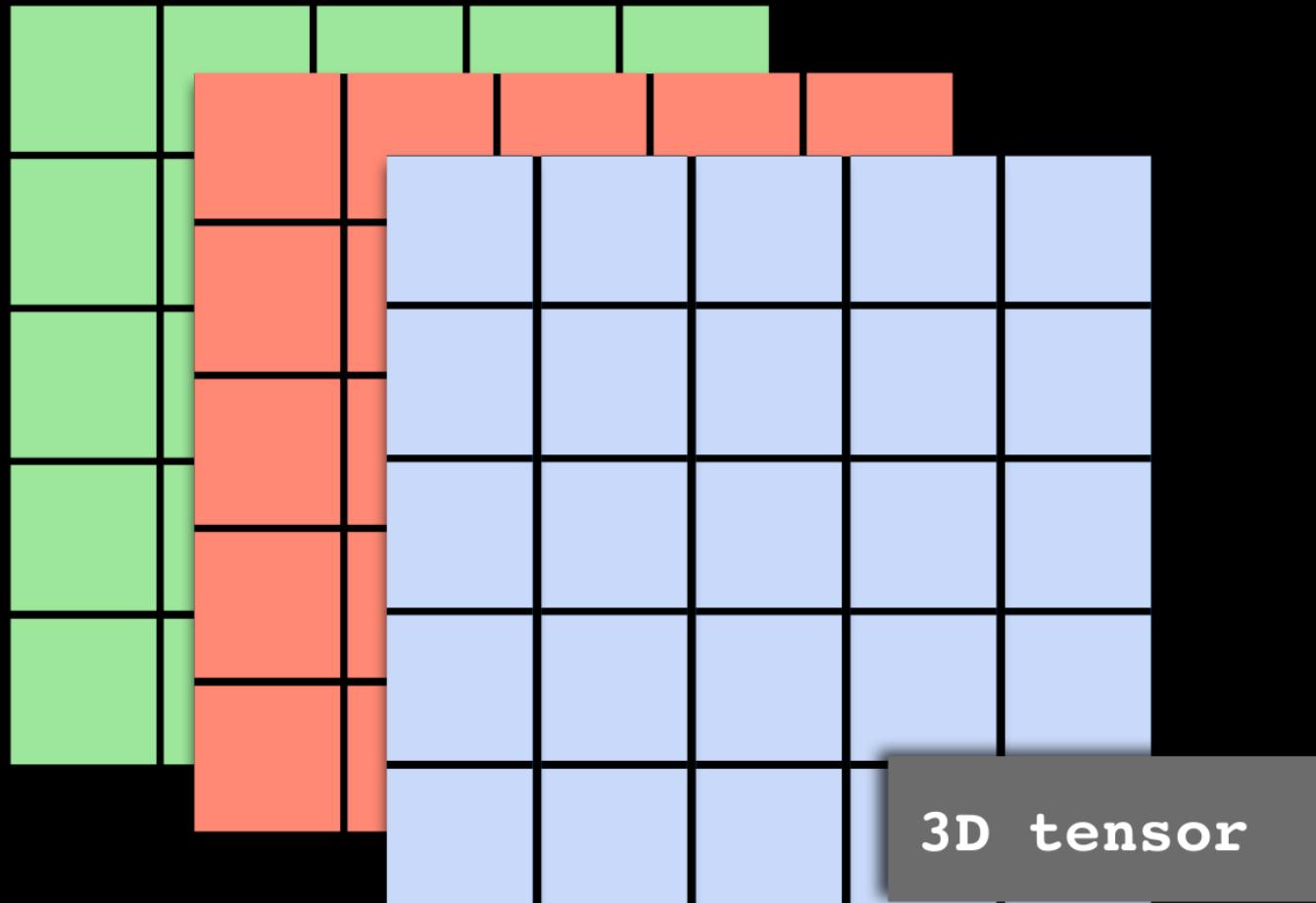
$\text{rows} \times \text{cols} \times \text{channels}$



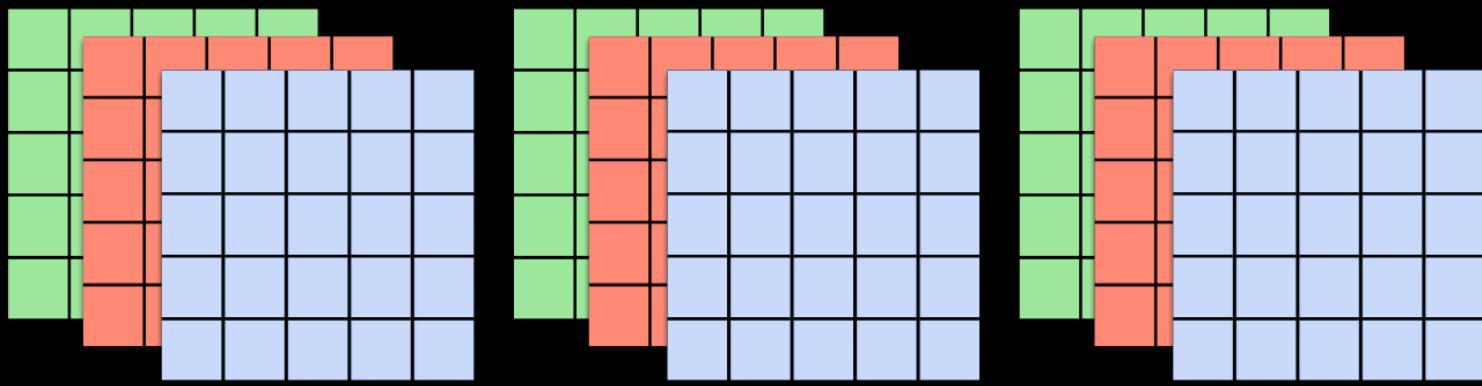


$768 \times 1024 \times 3$

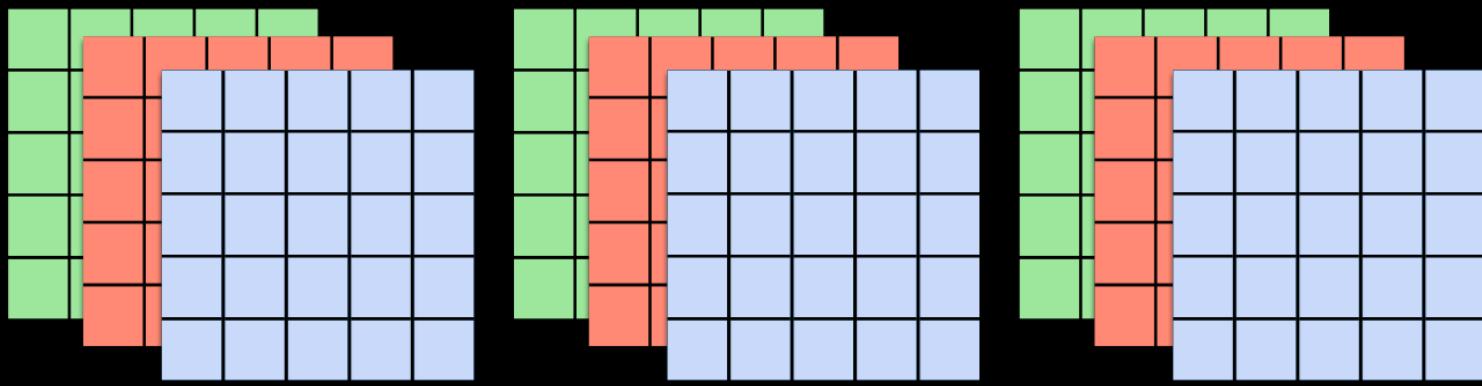




$\text{rows} \times \text{cols} \times \text{channels}$



$\text{rows} \times \text{cols} \times \text{channels} \times t$



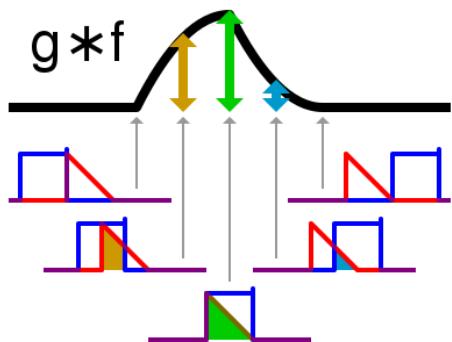
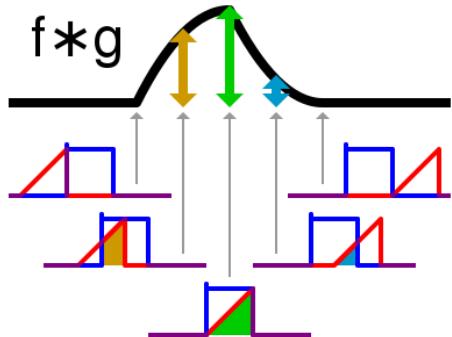
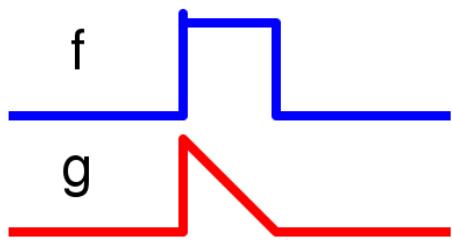
$\text{rows} \times \text{cols} \times \text{channels} \times t$

4D tensor

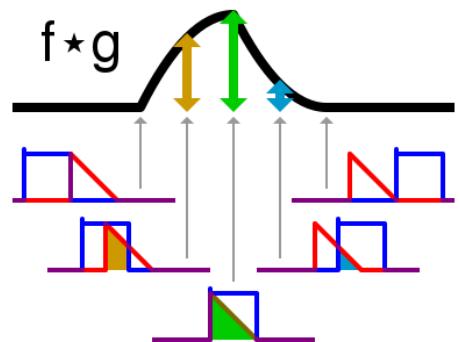
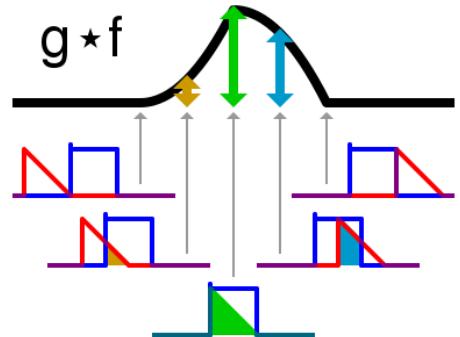
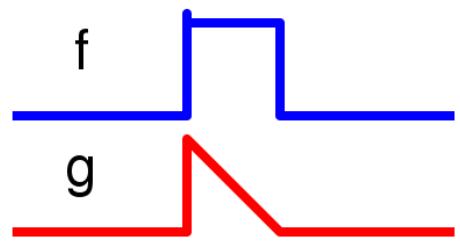
Згортка (convolution)

$$(f * g)(x) = \int f(z)g(x - z) dz$$

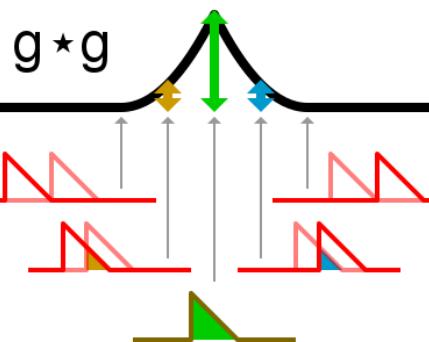
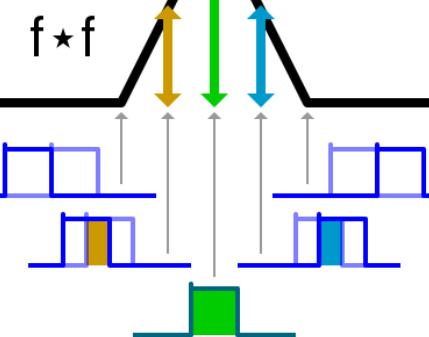
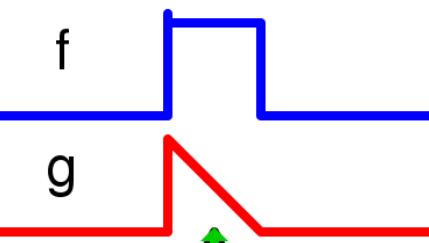
Convolution



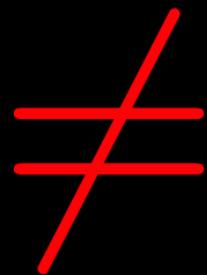
Cross-correlation



Autocorrelation



convolution



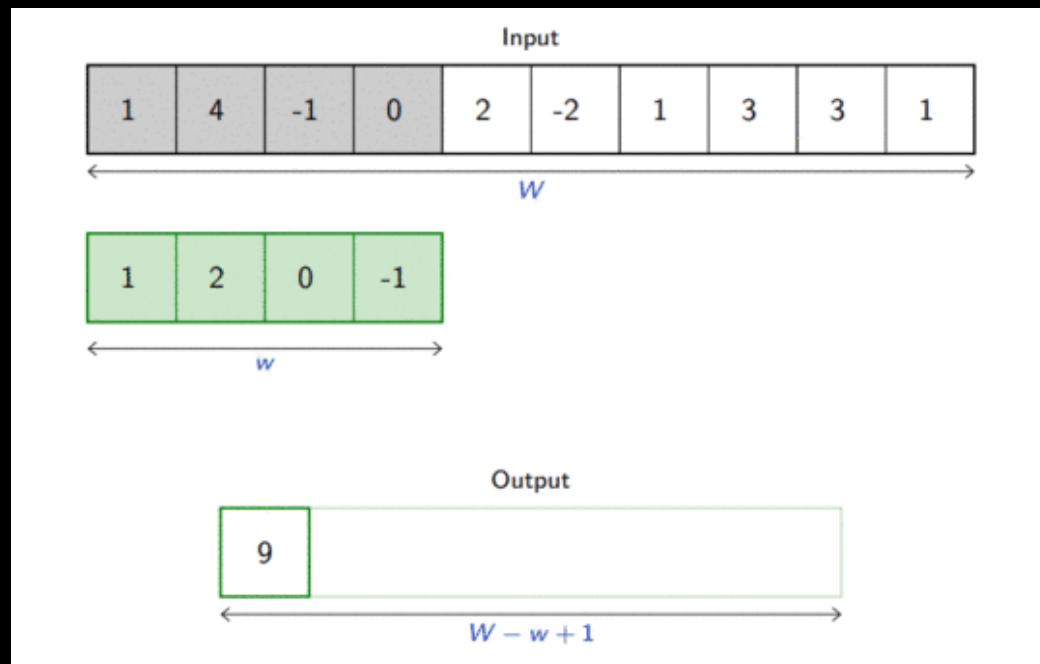
correlation

Демо

Відмінність згортки та взаємної
кореляції?

1d згортка

Згортковий шар застосовує те саме лінійне перетворення локально всюди, зберігаючи структуру сигналу.



1d згортка

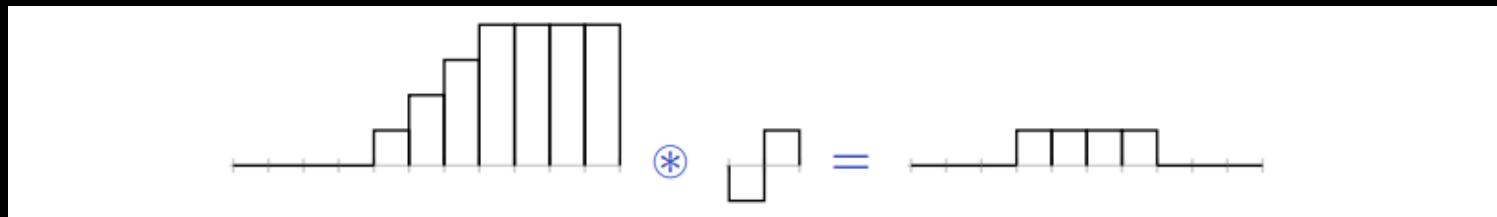
Для одновимірного вхідного сигналу $\mathbf{x} \in \mathbb{R}^W$ та згорткового фільтра $\mathbf{u} \in \mathbb{R}^w$, дискретна згортка $\mathbf{x} \circledast \mathbf{u}$ є вектором розміром $W - w + 1$:

$$(\mathbf{x} \circledast \mathbf{u})[i] = \sum_{m=0}^{w-1} \mathbf{x}_{m+i} \mathbf{u}_m.$$

Технічно, \circledast позначає оператор взаємної кореляції. Однак більшість бібліотек машинного навчання називають це згорткою.

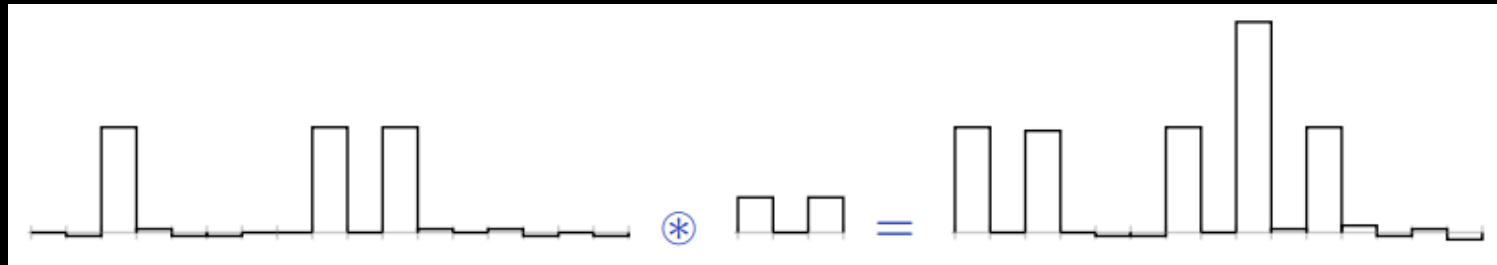
Взаємна кореляція може реалізовувати диференціальні оператори:

$$(0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4) \circledast (-1, 1) = (0, 0, 0, 1, 1, 1, 1, 0, 0, 0)$$



або прості порівняння шаблонів:

$$(0, 0, 3, 0, 0, 0, 0, 0, 3, 0, 3, 0, 0, 0) \circledast (1, 0, 1) = (3, 0, 3, 0, 0, 0, 3, 0, 6, 0, 3, 0)$$



Взаємна кореляція

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Вхід

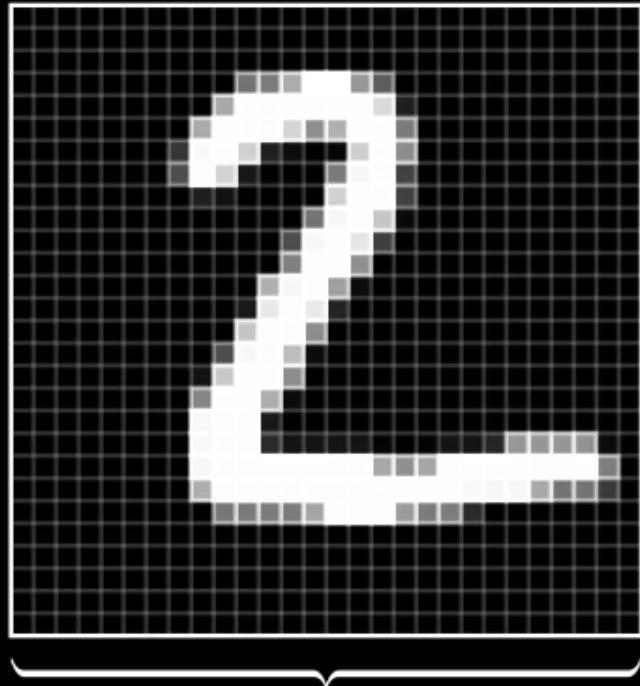
0	1	2
3	4	5
6	7	8

.

Фільтр

258	294
357	438

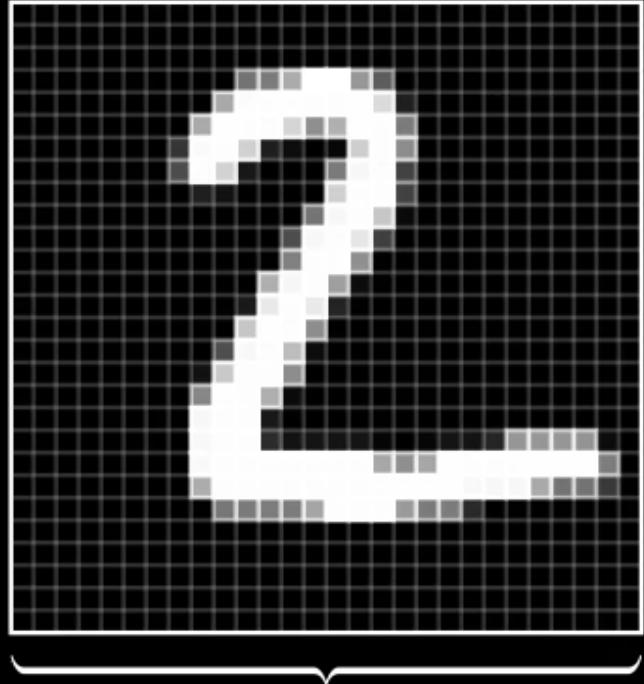
Вихід



Input Layer (28 x 28 x 1)

▶ 0:00 / 0:22





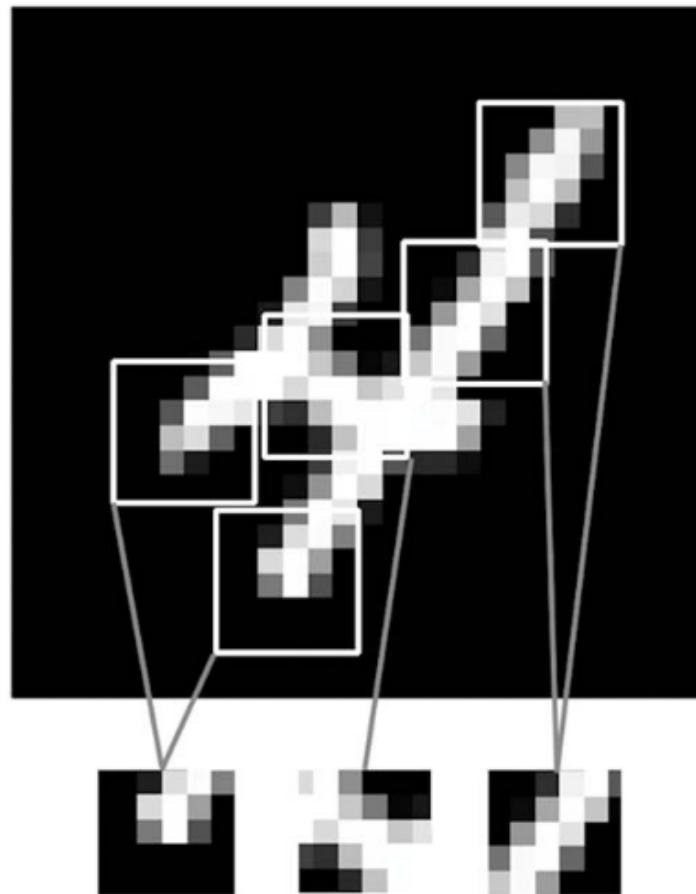
Input Layer (28 x 28 x 1)

-0.074	0.203	0.128	-0.033	0.289
-0.082	-0.198	-0.104	0.209	-0.184
0.112	0.231	0.104	0.259	-0.121
-0.124	-0.169	-0.116	-0.189	0.242
-0.055	0.102	0.116	-0.194	-0.222

Kernels = 2 x (5 x 5 x 1)
Stride = 1, Activation = ReLU

0.118	-0.003	-0.099	0.043	-0.1
-0.168	-0.096	0.228	0.237	0.192
0.335	-0.235	0.035	0.097	-0.028
0.117	0.068	0.094	-0.009	-0.137
-0.169	0.244	-0.146	0.189	-0.196

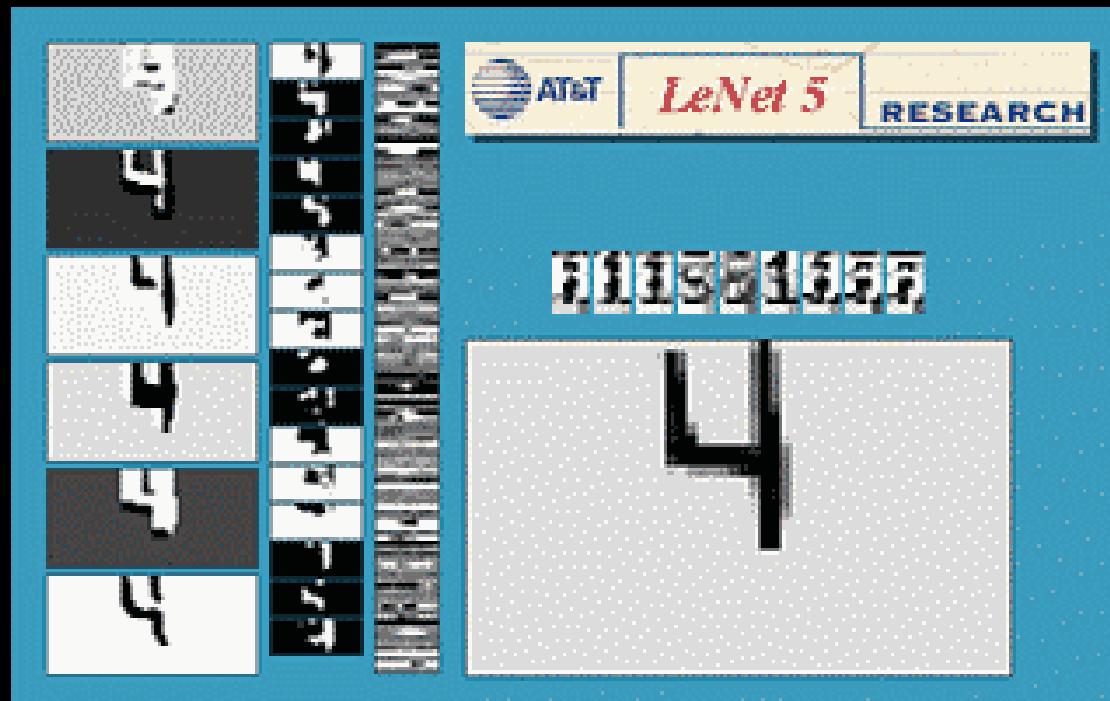
Згортковий шар



Еквіваріантність

Формально, функція f є еквіваріантною відносно перетворення g , якщо для будь-якого \mathbf{x} виконується рівність: $f(g(\mathbf{x})) = g(f(\mathbf{x}))$.

Використання спільних параметрів у згортковому шарі спричиняє, що цей шар стає еквіваріантним до трансляцій.

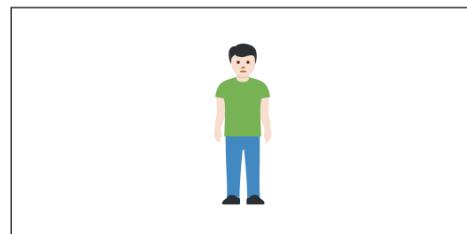


Якщо об'єкт змінює розташування у вхідному зображення, його представлення рухатиметься на таку саму величину у вихідному.

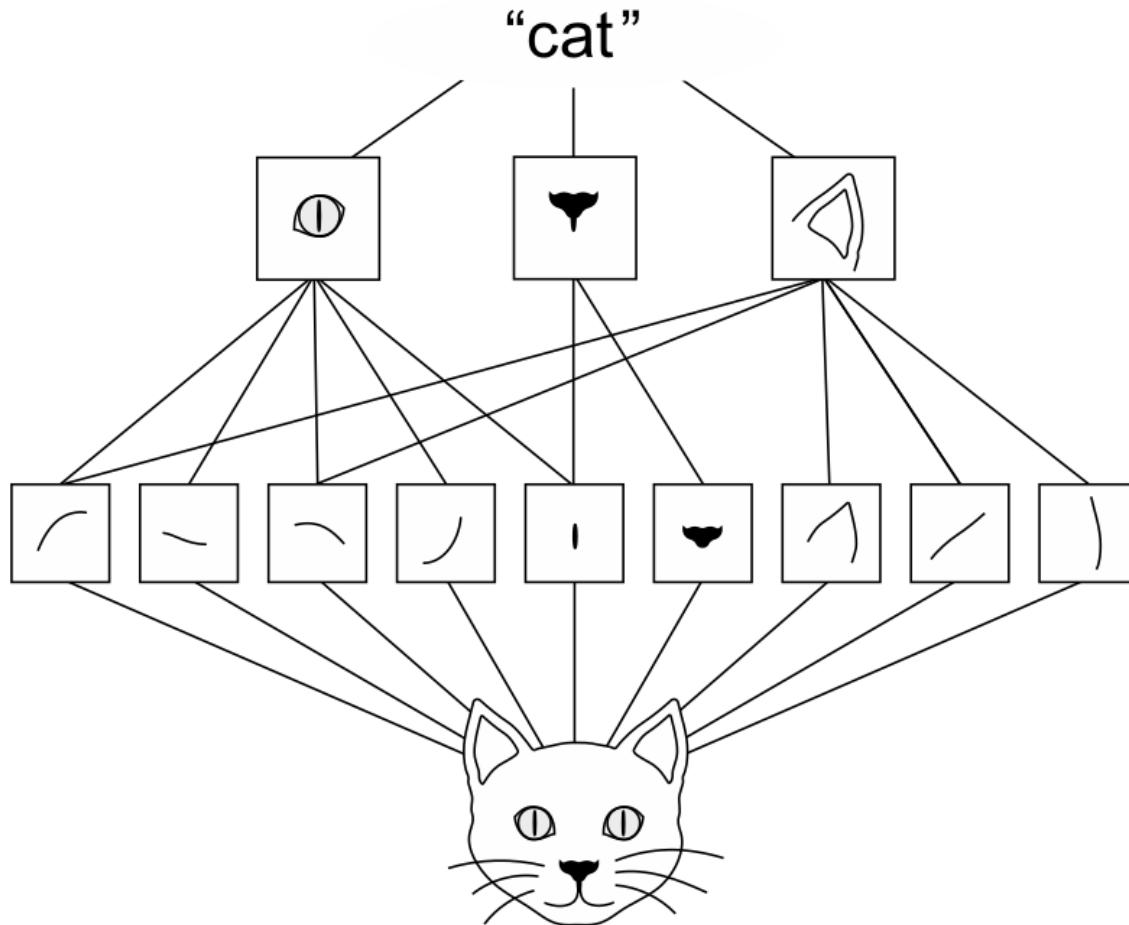
Інваріантнти відносно зміщень

Функція f є інваріантною відносно перетворення g , якщо для будь-якого \mathbf{x} виконується рівність: $f(g(\mathbf{x})) = f(\mathbf{x})$.

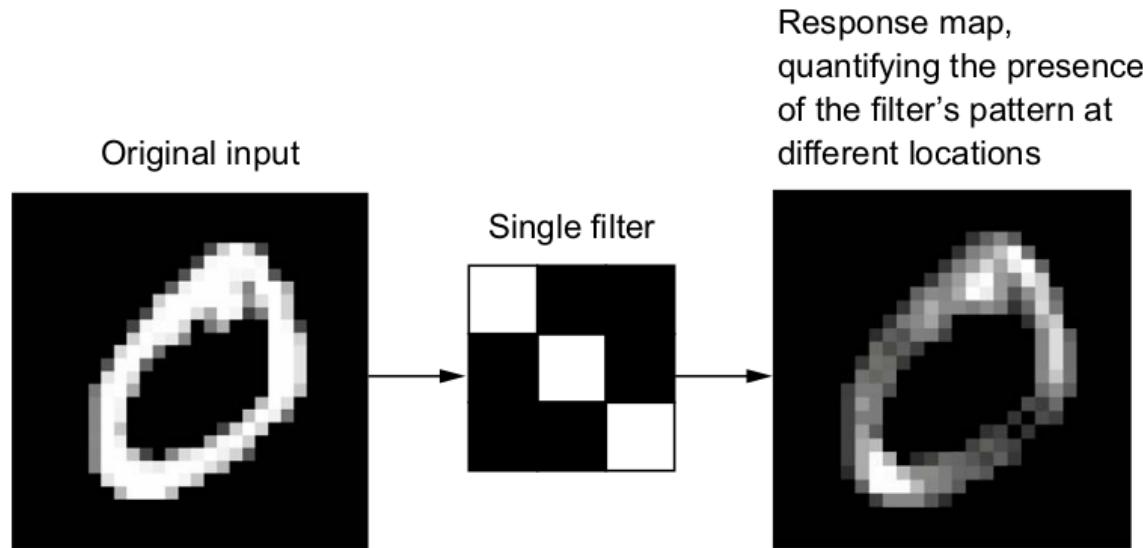
CNN інваріантнти відносно зміщень



Вивчають просторову ієрархію шаблонів

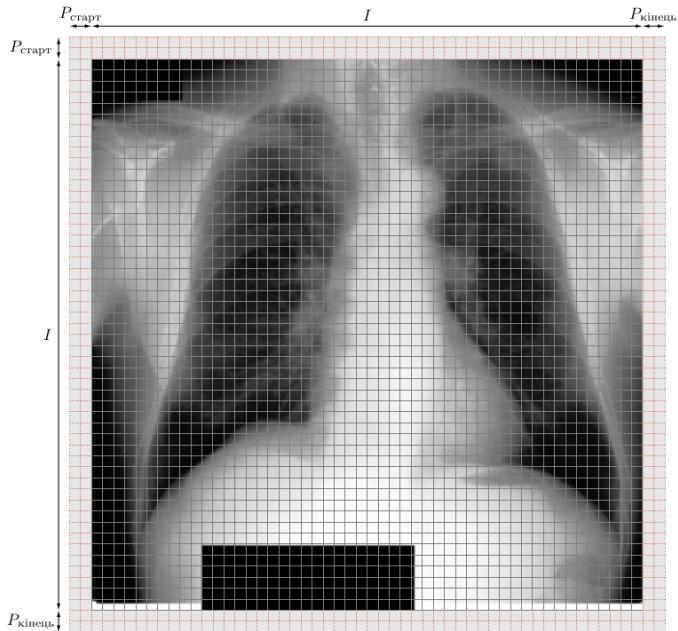


Вихідна карта ознак: двовимірна карта присутності візерунка в різних місцях вхідного тезора

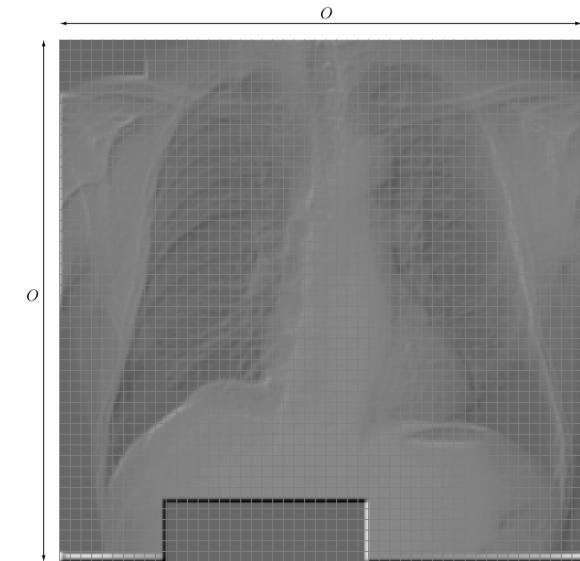
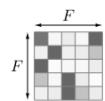


У прикладі MNIST перший шар згортки приймає карту ознак розміром **(28, 28, 1)** і виводить карту ознак розміром **(26, 26, 32)**: він обчислює 32 фільтри над своїм входом. Кожен із цих 32 вихідних каналів містить сітку значень 26×26 , яка є картою відгуку фільтра над входом, що вказує на відповідь цього шаблону фільтра в різних місцях входу (див. малюнок вище).

Розмір виходу згортки



Вхід



Фільтр

Вихід

$$O = \frac{I - F + 2P}{S} + 1$$

Демо

Як працює згортка?

Згортки визначаються двома ключовими параметрами:

- **Розмір патчів, отриманих із вхідних даних** – зазвичай це 3×3 або 5×5 . У прикладі вони були 3×3 , що є звичайним вибором.
- **Глибина вихідної карти ознак** – це кількість фільтрів, обчислених згорткою. Приклад почався з глибини 32 і закінчився глибиною 128.

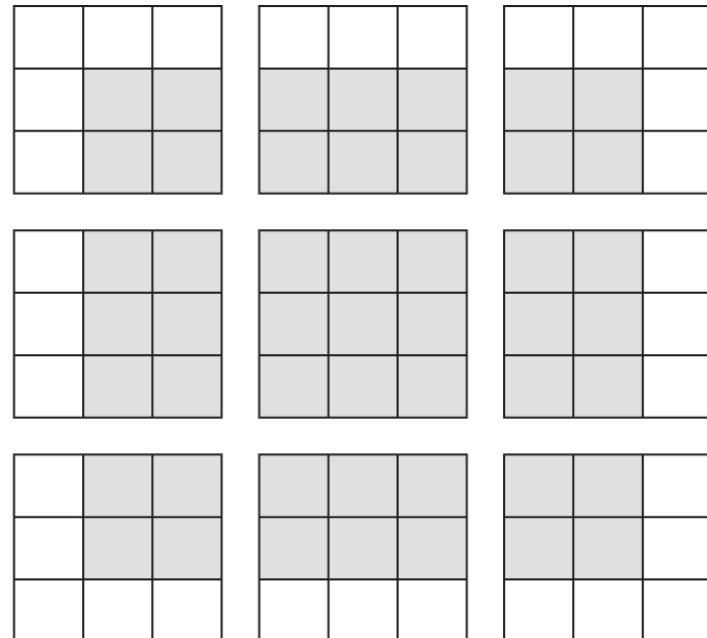
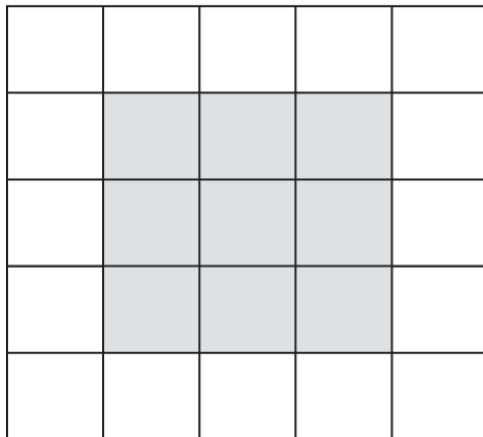
```
from tensorflow import keras
from tensorflow.keras import layers
input = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu") (input)
x = layers.MaxPool2D(pool_size=2) (x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu") (x)
x = layers.MaxPool2D(pool_size=2) (x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu") (x)
x = layers.Flatten() (x)
output = layers.Dense(10, activation="softmax") (x)
model = keras.Model(inputs=input, outputs=output)
```

У Keras шар Conv2D має наступні параметри ([Глибина вихідної карти ознак, Розмір патчів, отриманих із вхідних даних](#)):

```
Conv2D (output_depth, (window_height, window_width))
```

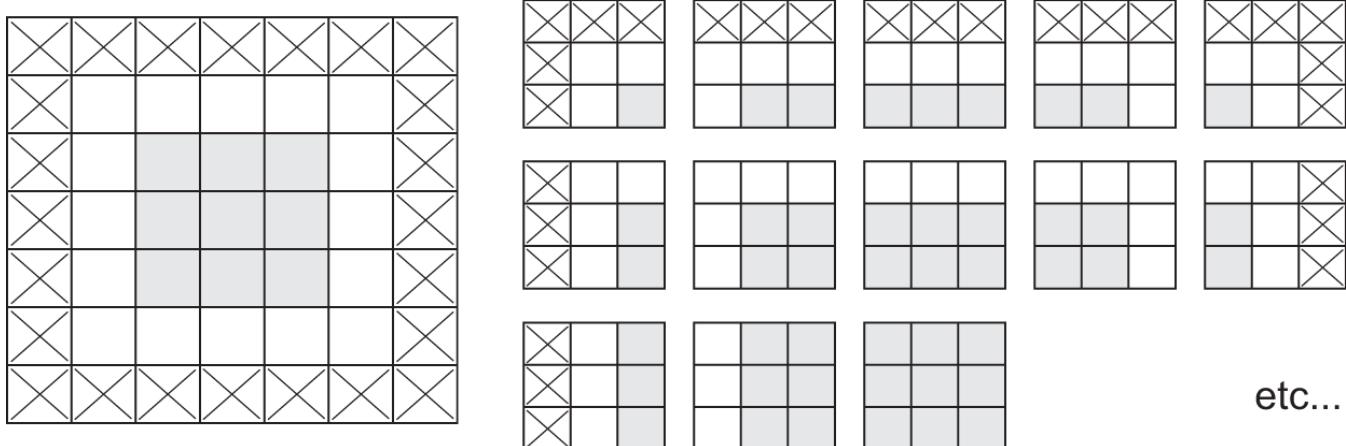
Розуміння ефекту додавання (padding)

Ядро розміром 3×3 , вхідна карта ознак 5×5



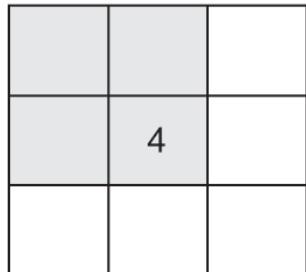
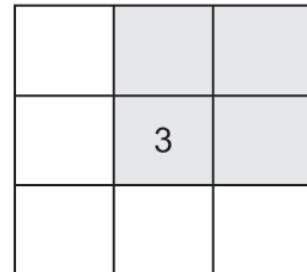
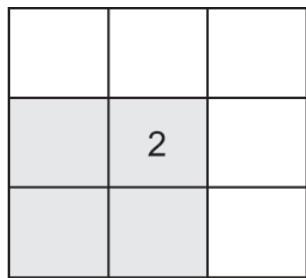
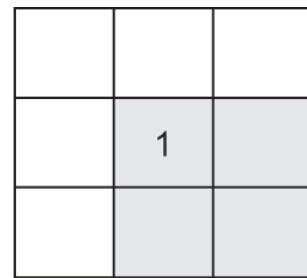
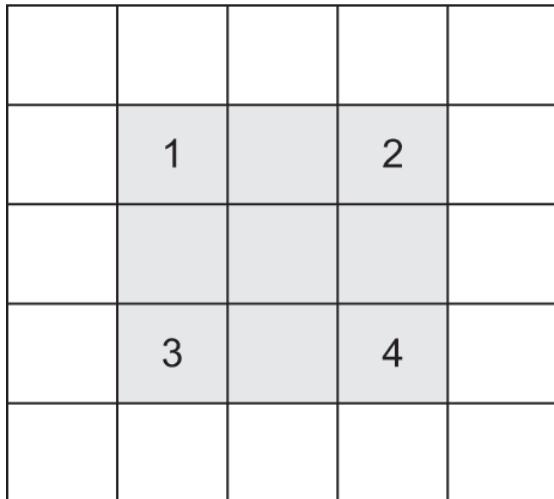
Розуміння ефекту доповнення

Доповнююємо вхідну карту ознак 5×5 для того, щоб можна було витягнути **25** патчів розміром 3×3



Розуміння кроку згортки

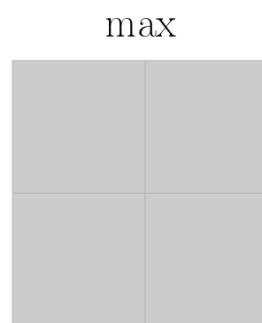
3×3 ядро з кроком 2×2



Операція максимізаційного агрегування (max-pooling)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Вхід



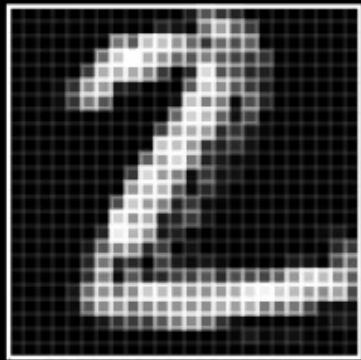
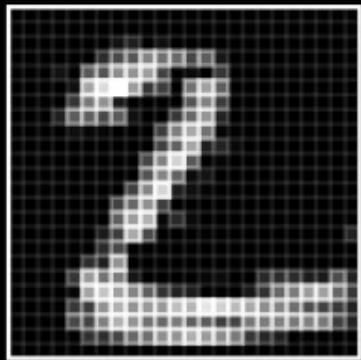
max

=

5	7
13	15

Вікно

Вихід



▶ 0:00 / 0:41



Операція максимізаційного агрегування (max-pooling)

```
from tensorflow import keras
from tensorflow.keras import layers
input = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu") (input)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu") (x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu") (x)
x = layers.Flatten() (x)
output = layers.Dense(10, activation="softmax") (x)
model_without_max_pool = keras.Model(inputs=input, outputs=output)
```

```
model_without_max_pool.summary()
```

Model: "model_1"

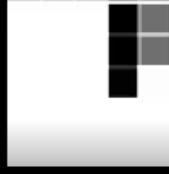
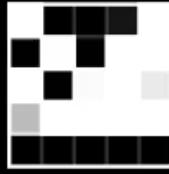
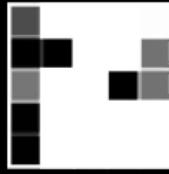
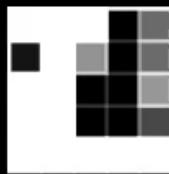
Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[None, 28, 28, 1]	0
conv2d_3 (Conv2D)	(None, 26, 26, 32)	320
conv2d_4 (Conv2D)	(None, 24, 24, 64)	18496
conv2d_5 (Conv2D)	(None, 22, 22, 128)	73856
flatten_1 (Flatten)	(None, 61952)	0
dense_1 (Dense)	(None, 10)	619530
<hr/>		

Total params: 712202 (2.72 MB)

Trainable params: 712202 (2.72 MB)

Non-trainable params: 0 (0.00 Byte)

FLATTEN



0:00 / 0:19



