CS3485

# Deep Learning for Computer Vision

*Lec 7*: Convolutional Neural Networks

# Announcements

- Next week:
  - I'll be out in a conference from Monday to Friday and some important events there conflict with our lecture times.
  - Lectures will be asynchronous! I'll post their videos on Monday.
  - We'll still have a quiz (I'll test the Canvas version). You'll have to finish it on Thursday.
- Lab2:
  - Lab grades are out, let me know if you have any questions!
  - Don't forget the default parameters!
- Lab3:
  - It is out! It will be due next Thursday.
  - Things may get slower with the experiments. Make sure to use GPUs,
  - Start earlier rather than later due to errors and bugs
- A great resource for last lecture: 3Brow1Blue videos on Deep Learning!

# (Tentative) Lecture Roadmap

## Basics of Deep Learning

- Intro to Computer Vision
- Linear Classifiers and Perceptron
- Multilayer Perceptron
- Optimization in Deep Learning
- Pytorch I – MLPs
- Pytorch II – Images and Regularization
- Convolutional Neural Networks
- Data Augmentation and Deep CNNs
- Transfer Learning and Residual Nets
- Inception Net and what CNNs learn
- Adversarial Examples and Self-supervision

## Deep Learning and Computer Vision in Practice

- Intro to MLOps
- Intro to Object Detection
- Fast Object Detection
- Intro to Image Segmentation
- Applications of Detection and Segmentation
- Autoencoders
- Image Generation with GANs
- Advanced GANs
- The Attention Mechanism
- Transformers and ChatGPT
- Image Generation by Prompt
- Misc. Topics

# Fashion MNIST

- Last time, we used the MNIST dataset to try out our Multilayer Perceptron (MLP) using PyTorch.
- Today, we'll try a more challenging Dataset called **Fashion MNIST (FMNIST)**.
- It follows the same specs as MNIST: there are $70$k images ($60$k for training and $10$k for testing) of size $28 \times 28$ of $10$ classes.
- Instead of handwritten digits, the data now is of clothing articles and the classes are: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot.
- Let's see how our previous network performs in this new data and take the time to review last class.

# Loading the Data and DataLoaders

- Let’s start by downloading the dataset:

```python
import torch
from torchvision import datasets
from torch.utils.data import Dataset, DataLoader

device = 'cuda' if torch.cuda.is_available() else 'cpu'
fmnist_train = datasets.FashionMNIST('~/data/FMNIST', download=True, train=True)
fmnist_test = datasets.FashionMNIST('~/data/FMNIST', download=True, train=False)
x_train, y_train = fmnist_train.data, fmnist_train.targets
x_test, y_test = fmnist_test.data, fmnist_test.targets

class FMNISTDataset(Dataset):
   def __init__(self, x, y):
       x = x.view(-1,28*28).float()/255
       self.x, self.y = x, y
   def __getitem__(self, ix):
       return self.x[ix].to(device), self.y[ix].to(device)
   def __len__(self):
       return len(self.x)

train_dataset = FMNISTDataset(x_train, y_train)
train_dl = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_dataset = FMNISTDataset(x_test, y_test)
test_dl = DataLoader(test_dataset, batch_size=32, shuffle=True)
```

# Loading the Data and DataLoaders

- Let's start by downloading the dataset:

```
import torch
from torchvision import datasets
from torch.utils.data import Dataset, DataLoader

device = 'cuda' if torch.cuda.is_available() else 'cpu'
fmnist_train = datasets.FashionMNIST('~/data/FMNIST', download=True, train=True)
fmnist_test = datasets.FashionMNIST('~/data/FMNIST', download=True, train=False)
x_train, y_train = fmnist_train.data, fmnist_train.targets
x_test, y_test = fmnist_test.data, fmnist_test.targets

class FMNISTDataset(Dataset):
   def __init__(self, x, y):
       x = x.view(-1,28*28).float()/255
       self.x, self.y = x, y
   def __getitem__(self, ix):
       return self.x[ix].to(device), self.y[ix].to(device)
   def __len__(self):
       return len(self.x)

train_dataset = FMNISTDataset(x_train, y_train)
train_dl = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_dataset = FMNISTDataset(x_test, y_test)
test_dl = DataLoader(test_dataset, batch_size=32, shuffle=True)
```

Detects the GPU

Loads the Fashion MNIST training and testing data.

Creates a Dataset class that preprocesses the data (reshapes and rescales it).

Define the Dataloaders, that coordinate how the dat will be read.

# Visualizing the data

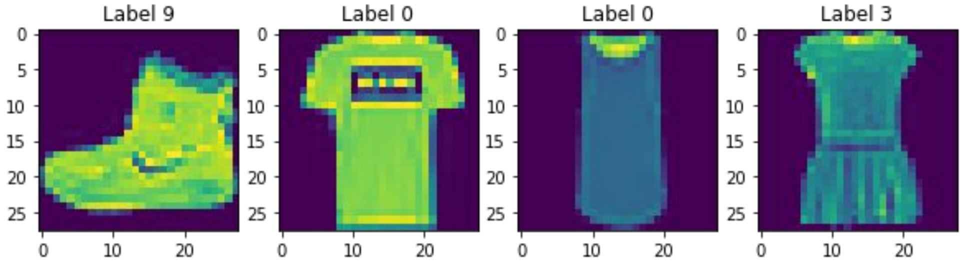
- We also check a bit of how the data looks like:

```python
print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

```
torch.Size([60000, 28, 28]) torch.Size([60000])
torch.Size([10000, 28, 28]) torch.Size([10000])
```

- Following the **(N,C,H,W)**, we know that
  - The training set has $60000$ points (N), of 1 channel each (C), and each points has height $28$ (H) and width $28$ (W). The same for the testing set, but with $10000$ samples (N).
  - There are $60000$ scalar labels for training and $10000$ for testing.
- We can furthermore read a few data points and their labels:

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(10,3))
for i in range(4):
   plt.subplot(1,4,i+1)
   plt.imshow(x_train[i])
   plt.title(f"Label {y_train[i]}")
plt.show()
```



# Creating the network and visualizing it

- Today, we'll use `nn.Sequential()` to create a NN of one hidden layer with 1k units:

```
import torch.nn as nn
model = nn.Sequential(nn.Linear(28 * 28, 1000), nn.ReLU(), nn.Linear(1000, 10)).to(device)
```

- We also introduce the `summary` function to visualize the network:

```
# In order to install torchsummary, run
# 'pip install torch summary'
from torchsummary import summary
summary(model, (1, 28*28))
```

where `(1, 28*28)` is the size of the model's input (matrices of size $1{\times}784$).

- Note that we need to learn almost $800$k weights!

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1            [-1, 1, 1, 1000]         785,000
              ReLU-2            [-1, 1, 1, 1000]               0
            Linear-3              [-1, 1, 1, 10]          10,010
================================================================
Total params: 795,010
Trainable params: 795,010
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.02
Params size (MB): 3.03
Estimated Total Size (MB): 3.05
----------------------------------------------------------------
```

# Defining the loss and the optimizer

- Today, we also choose the Cross Entropy as our loss function and ADAM as our optimizer with $0.001$ as the learning rate:

```
from torch.optim import Adam
loss_fn = nn.CrossEntropyLoss()
opt = Adam(model.parameters(), lr=1e-3)
```

- Like we did last time, we define two auxiliary functions: one to do all the steps for training and the other to compute classification accuracies.

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()

    opt.zero_grad() # Flush memory
    batch_loss = loss_fn(model(x), y) # Compute loss
    batch_loss.backward() # Compute gradients
    opt.step() # Make a GD step

    return batch_loss.detach().cpu().numpy()
```

```
@torch.no_grad()
def accuracy(x, y, model):
    model.eval()

    prediction = model(x)
    argmaxes = prediction.argmax(dim=1)
    s = torch.sum((argmaxes == y).float())/len(y)

    return s.cpu().numpy()
```

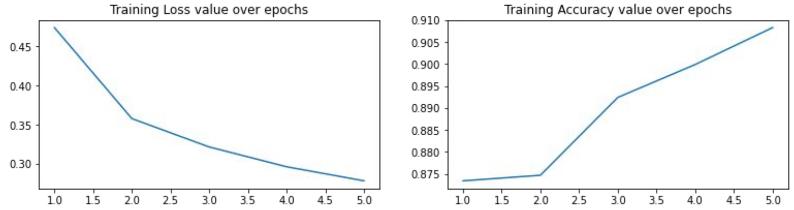
# Training the network

- Then, we train the network:

```python
import numpy as np
losses, accuracies, n_epochs = [], []
n_epochs = 5
for epoch in range(n_epochs):
   print(f"Running epoch {epoch + 1} of {n_epochs}")

   epoch_losses, epoch_accuracies = [], []
   for batch in train_dl:
       x, y = batch
       batch_loss = train_batch(x, y, model, opt, loss_func)
       epoch_losses.append(batch_loss)
   epoch_loss = np.mean(epoch_losses)

   for batch in train_dl:
       x, y = batch
       batch_acc = accuracy(x, y, model)
       epoch_accuracies.append(batch_acc)
   epoch_accuracy = np.mean(epoch_accuracies)

   losses.append(epoch_loss)
   accuracies.append(epoch_accuracy)
```

- And visualize how it did during training:

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(13,3))
plt.subplot(121)
plt.title('Training Loss value over epochs')
plt.plot(np.arange(n_epochs) + 1, losses)
plt.subplot(122)
plt.title('Testing Accuracy value over epochs')
plt.plot(np.arange(n_epochs) + 1, accuracies)
```




- This learning procedure (of $800$k weights) took around $43.6$ s.

# Testing the learned classifier

- Finally, we can test our classifier (*running again may give be slightly different results*):

```
epoch_accuracies = []
for batch in test_dl:
    x, y = batch
    batch_acc = accuracy(x, y, model)
    epoch_accuracies.append(batch_acc)

print(f"Test accuracy: {np.mean(epoch_accuracies)}")
```

```
Test accuracy: 0.8813897967338562
```

- When we did this same procedure with the same parameters on the MNIST digit dataset, **we got 96% testing accuracy**.
- This shows **how much harder Fashion MNIST is** and that we need more to use more techniques to improve its performance.
- How can we improve this testing result **without adding many more weights**?

# The Convolution Operation

- A very important operation in our solution to better our performance is the **Convolution.**
- Take two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{p \times q}$.
- The convolution between $A$ and $B$, denoted here as $A \otimes B$ is another matrix, $C$, such that:

$$(A \otimes B)_{i,j} = C_{i,j} = \sum_{k=0}^{p-1} \sum_{l=0}^{q-1} B_{k,l} \times B_{i-k,j-l}$$

$A$

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

$B$

| | |
|---|---|
| 1 | 2 |
| 3 | 4 |

$A \otimes B$

| | | |
|---|---|---|
| | | |
| | | |
| | | |

- Despite its apparent complexity, **it is quite simple**: we are simply “scanning” matrix $A$ with windows of the size of matrix $B$ and, as we scan, we multiply all elements of the window in $A$ with $B$ element wise, and finally sum the multiplication results up.
- In this case, the matrix $B$, which is sweeping over $A$, is called **kernel** or **filter** and the convolution process is sometimes called **filtering**.

# The Convolution Operation

- A very important operation in our solution to better our performance is the **Convolution.**
- Take two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times q}$.
- The convolution between $A$ and $B$, denoted here as $A \otimes B$ is another matrix, $C$, such that:

$$(A \otimes B)_{i,j} = C_{i,j} = \sum_{k=0}^{p-1}\sum_{l=0}^{q-1} B_{k,l} \times B_{i-k,j-l}$$

$A$

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

$B$

| | |
|---|---|
| 1 | 2 |
| 3 | 4 |

$A \otimes B$

| | | |
|---|---|---|
| 44 | | |
| | | |
| | | |

$1\times1 + 2\times2 + 5\times3 + 6\times4 = 44$

- Despite its apparent complexity, **it is quite simple**: we are simply "scanning" matrix $A$ with windows of the size of matrix $B$ and, as we scan, we multiply all elements of the window in $A$ with $B$ element wise, and finally sum the multiplication results up.
- In this case, the matrix $B$, which is sweeping over $A$, is called **kernel** or **filter** and the convolution process is sometimes called **filtering**.

# The Convolution Operation

- A very important operation in our solution to better our performance is the **Convolution.**
- Take two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times q}$.
- The convolution between $A$ and $B$, denoted here as $A \otimes B$ is another matrix, $C$, such that:

$$A = \begin{bmatrix} 1 & \boxed{2} & \boxed{3} & 4 \\ 5 & \boxed{6} & \boxed{7} & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad A \otimes B = \begin{bmatrix} 44 & \boxed{54} & \; \\ \; & \; & \; \\ \; & \; & \; \end{bmatrix}$$

$$(A \otimes B)_{i,j} = C_{i,j} = \sum_{k=0}^{p-1} \sum_{l=0}^{q-1} B_{k,l} \times B_{i-k,j-l}$$

$$2\times1 + 3\times2 + 6\times3 + 7\times4 = 54$$

- Despite its apparent complexity, **it is quite simple**: we are simply “scanning” matrix $A$ with windows of the size of matrix $B$ and, as we scan, we multiply all elements of the window in $A$ with $B$ element wise, and finally sum the multiplication results up.
- In this case, the matrix $B$, which is sweeping over $A$, is called **kernel** or **filter** and the convolution process is sometimes called **filtering**.

# The Convolution Operation

- A very important operation in our solution to better our performance is the **Convolution.**
- Take two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{p \times q}$.
- The convolution between $A$ and $B$, denoted here as $A \otimes B$ is another matrix, $C$, such that:

$$(A \otimes B)_{i,j} = C_{i,j} = \sum_{k=0}^{p-1} \sum_{l=0}^{q-1} B_{k,l} \times B_{i-k,j-l}$$

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad A \otimes B = \begin{bmatrix} 44 & 54 & 64 \\ & & \\ & & \end{bmatrix}$$

$$3 \times 1 + 4 \times 2 + 7 \times 3 + 8 \times 4 = 64$$

- Despite its apparent complexity, **it is quite simple**: we are simply "scanning" matrix $A$ with windows of the size of matrix $B$ and, as we scan, we multiply all elements of the window in $A$ with $B$ element wise, and finally sum the multiplication results up.
- In this case, the matrix $B$, which is sweeping over $A$, is called **kernel** or **filter** and the convolution process is sometimes called **filtering**.

# The Convolution Operation

- A very important operation in our solution to better our performance is the **Convolution.**
- Take two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{p \times q}$.
- The convolution between $A$ and $B$, denoted here as $A \otimes B$ is another matrix, $C$, such that:

$$(A \otimes B)_{i,j} = C_{i,j} = \sum_{k=0}^{p-1} \sum_{l=0}^{q-1} B_{k,l} \times B_{i-k,j-l}$$

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ \boxed{5} & \boxed{6} & 7 & 8 \\ \boxed{9} & \boxed{10} & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad A \otimes B = \begin{bmatrix} 44 & 54 & 64 \\ \boxed{84} & & \\ & & \end{bmatrix}$$

$$5 \times 1 + 6 \times 2 + 9 \times 3 + 10 \times 4 = 84$$

- Despite its apparent complexity, **it is quite simple**: we are simply “scanning” matrix $A$ with windows of the size of matrix $B$ and, as we scan, we multiply all elements of the window in $A$ with $B$ element wise, and finally sum the multiplication results up.
- In this case, the matrix $B$, which is sweeping over $A$, is called **kernel** or **filter** and the convolution process is sometimes called **filtering**.

# The Convolution Operation

- A very important operation in our solution to better our performance is the **Convolution.**
- Take two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{p \times q}$.
- The convolution between $A$ and $B$, denoted here as $A \otimes B$ is another matrix, $C$, such that:

$$(A \otimes B)_{i,j} = C_{i,j} = \sum_{k=0}^{p-1} \sum_{l=0}^{q-1} B_{k,l} \times B_{i-k,j-l}$$

$$A = \begin{array}{|c|c|c|c|} \hline 1 & 2 & 3 & 4 \\ \hline 5 & \boxed{6} & \boxed{7} & 8 \\ \hline 9 & \boxed{10} & \boxed{11} & 12 \\ \hline 13 & 14 & 15 & 16 \\ \hline \end{array} \qquad B = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} \qquad A \otimes B = \begin{array}{|c|c|c|} \hline 44 & 54 & 64 \\ \hline 84 & \boxed{94} & \\ \hline & & \\ \hline \end{array}$$

$$6 \times 1 + 7 \times 2 + 10 \times 3 + 11 \times 4 = 94$$

- Despite its apparent complexity, **it is quite simple**: we are simply "scanning" matrix $A$ with windows of the size of matrix $B$ and, as we scan, we multiply all elements of the window in $A$ with $B$ element wise, and finally sum the multiplication results up.
- In this case, the matrix $B$, which is sweeping over $A$, is called **kernel** or **filter** and the convolution process is sometimes called **filtering**.

# The Convolution Operation

- A very important operation in our solution to better our performance is the **Convolution.**
- Take two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{p \times q}$.
- The convolution between $A$ and $B$, denoted here as $A \otimes B$ is another matrix, $C$, such that:

$$(A \otimes B)_{i,j} = C_{i,j} = \sum_{k=0}^{p-1} \sum_{l=0}^{q-1} B_{k,l} \times B_{i-k,j-l}$$

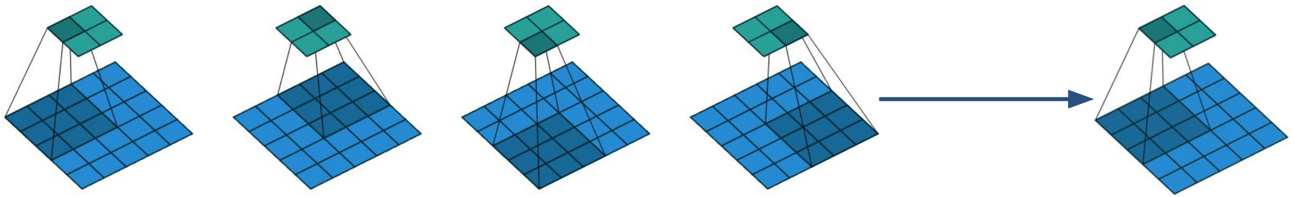
$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad A \otimes B = \begin{bmatrix} 44 & 54 & 64 \\ 84 & 94 & 104 \\ 124 & 134 & 144 \end{bmatrix}$$

And so on ...

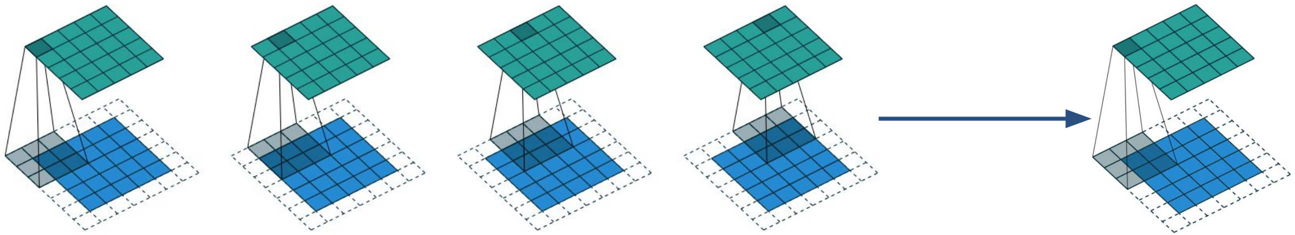
- Despite its apparent complexity, **it is quite simple**: we are simply "scanning" matrix $A$ with windows of the size of matrix $B$ and, as we scan, we multiply all elements of the window in $A$ with $B$ element wise, and finally sum the multiplication results up.
- In this case, the matrix $B$, which is sweeping over $A$, is called **kernel** or **filter** and the convolution process is sometimes called **filtering**.

# Strides and paddings

- The final **convolved matrix** $A \otimes B$ has a size that depends on the sizes of $A$ and $B$.
- In order to control the final size, we can change the convolution operation in two ways:
  - By changing the **convolutional stride**: the stride is the step you take when moving from one window to the next when sweeping. In usual convolution, the stride is $1$. Below, it is $2$:

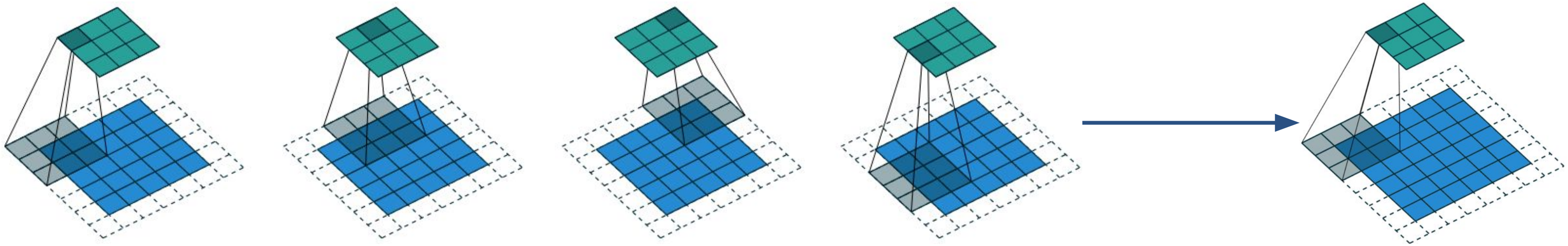


  - Via **padding**: add a frame of zero valued entries around the matrix being swept by the kernel.



# Strides and paddings

- The stride (skipping more entries as we sweep) and the mount of padding (the "thickness" of the frame) are then parameters of our convolution operation*.
- We can also have both strides and padding together:



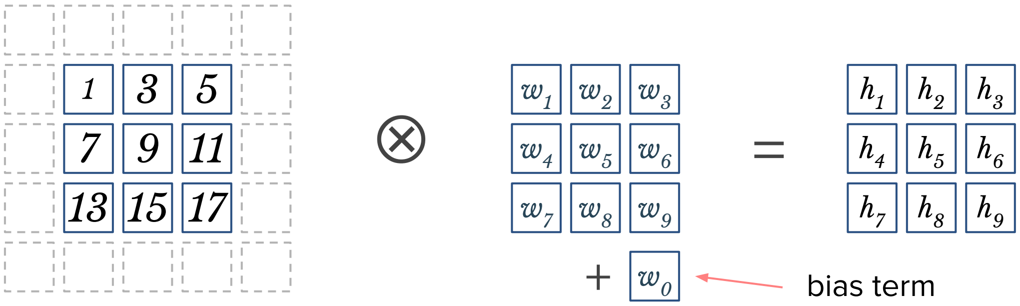
- There are other changes one can make in order **to increase** the size of the output matrix, compared to the input matrix.
- This operation is called a **Deconvolution operation**, which is crucial in many Deep Learning solutions for Computer Vision (*more on it later in the course*).

* All the animations for padding and strides were taken from this very pedagogical paper and github account.

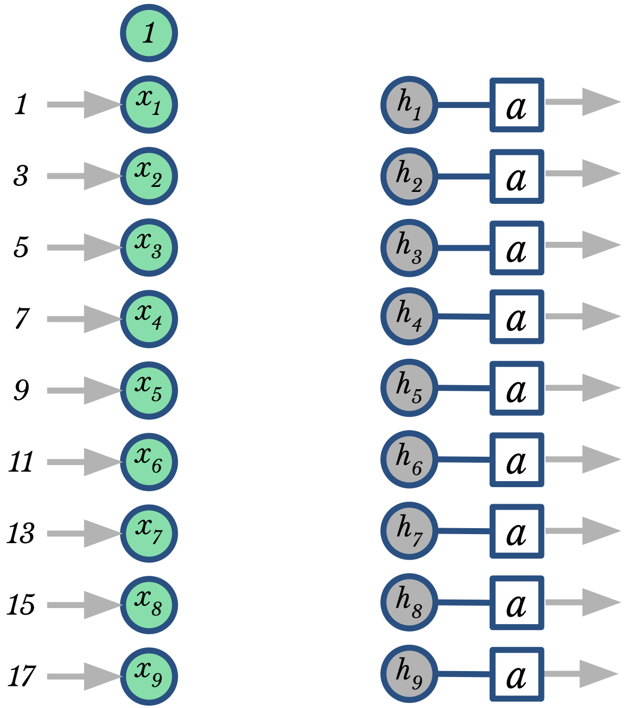
# Convolution as a Neural Network Layer

- Convolutions can be implemented as a NN layer, called **Convolutional Layer (ConvLayer)**.
- In it, we only need to zero many of the connections from its input to output (according to the convolution step) and keep the weights the same across units.

## Convolution Operation

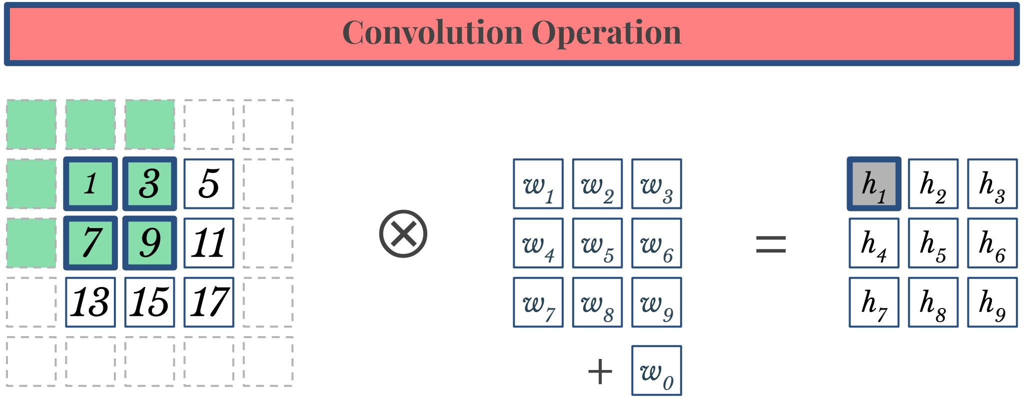


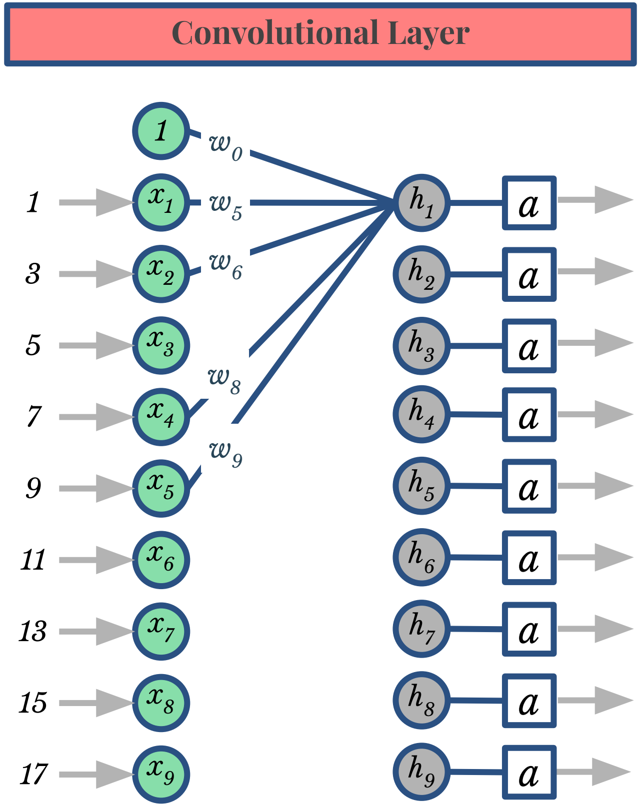

## Convolutional Layer



# Convolution as a Neural Network Layer

- Convolutions can be implemented as a NN layer, called **Convolutional Layer (ConvLayer)**.
- In it, we only need to zero many of the connections from its input to output (according to the convolution step) and keep the weights the same across units.

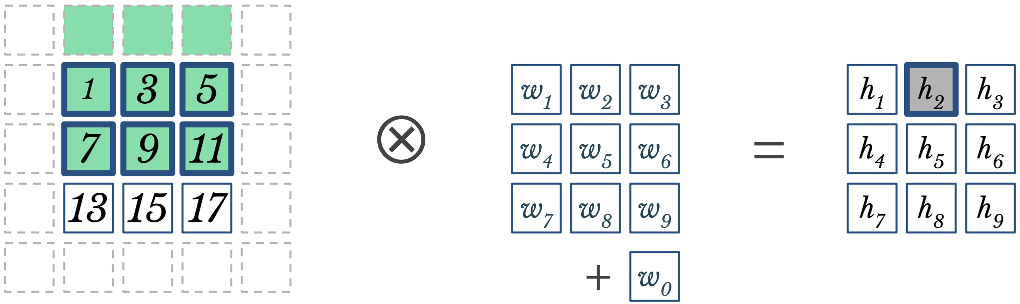


# Convolution as a Neural Network Layer

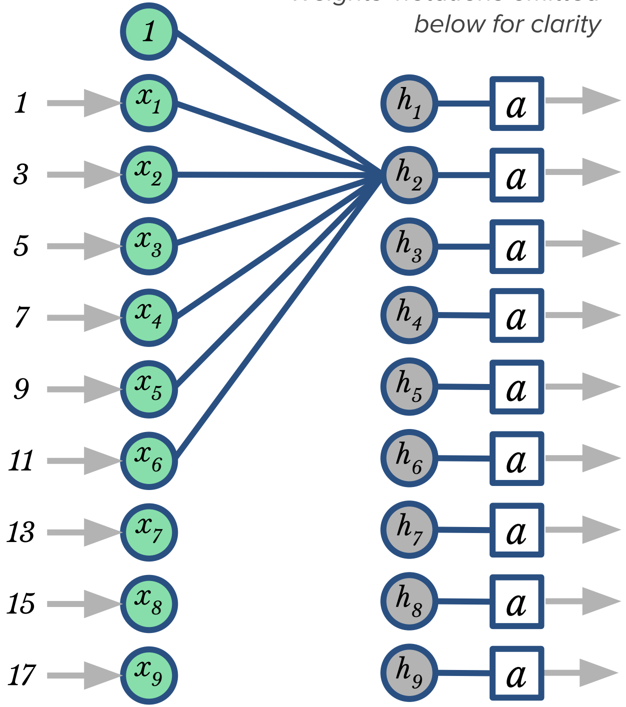
- Convolutions can be implemented as a NN layer, called **Convolutional Layer (ConvLayer)**.
- In it, we only need to zero many of the connections from its input to output (according to the convolution step) and keep the weights the same across units.

**Convolution Operation**




**Convolutional Layer**

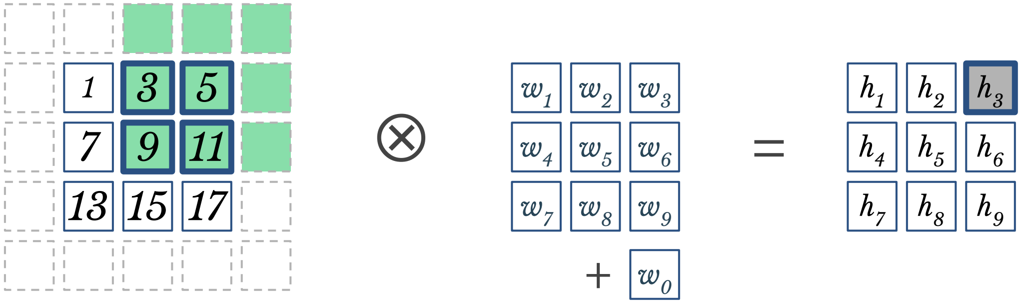
*Weights' notations omitted below for clarity*



# Convolution as a Neural Network Layer

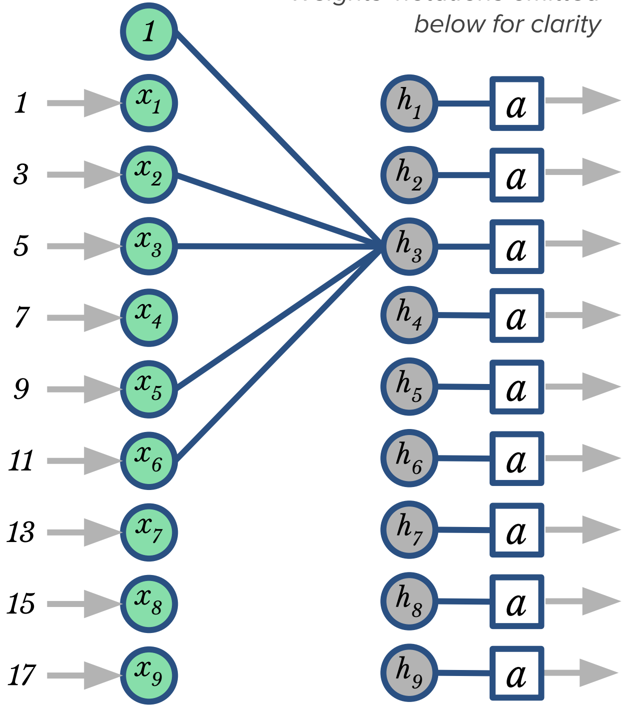
- Convolutions can be implemented as a NN layer, called **Convolutional Layer (ConvLayer)**.
- In it, we only need to zero many of the connections from its input to output (according to the convolution step) and keep the weights the same across units.

**Convolution Operation**




**Convolutional Layer**

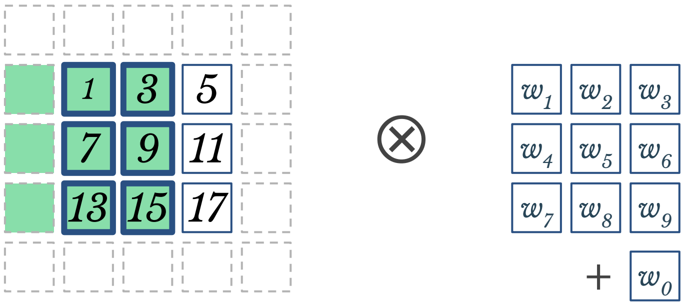
*Weights' notations omitted below for clarity*



# Convolution as a Neural Network Layer

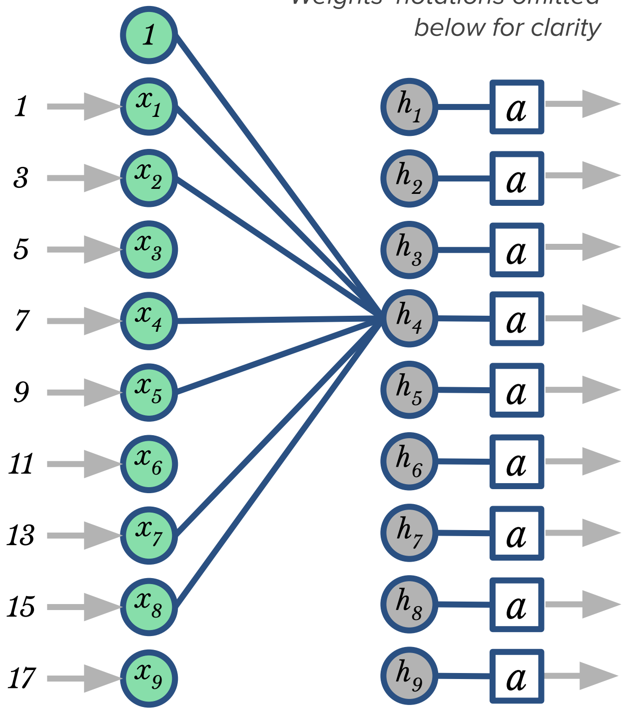
- Convolutions can be implemented as a NN layer, called **Convolutional Layer (ConvLayer)**.
- In it, we only need to zero many of the connections from its input to output (according to the convolution step) and keep the weights the same across units.

**Convolution Operation**




**Convolutional Layer**

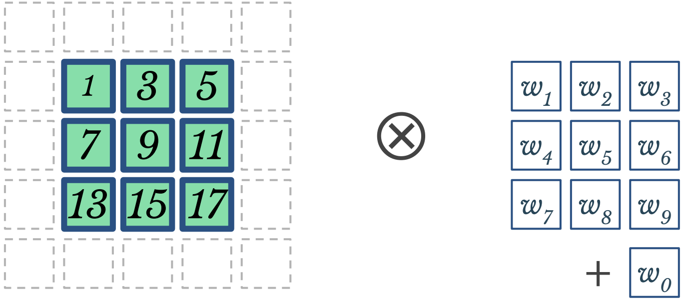
*Weights' notations omitted below for clarity*



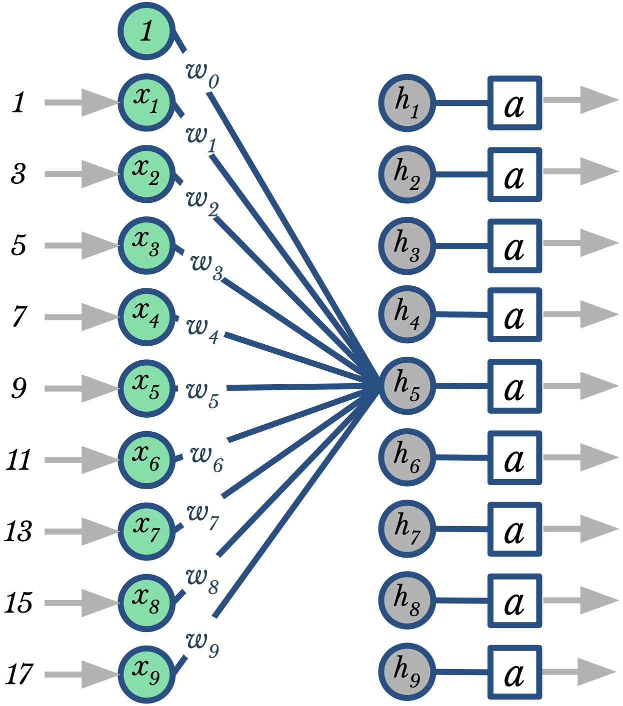
# Convolution as a Neural Network Layer

- Convolutions can be implemented as a NN layer, called **Convolutional Layer (ConvLayer)**.
- In it, we only need to zero many of the connections from its input to output (according to the convolution step) and keep the weights the same across units.

### Convolution Operation

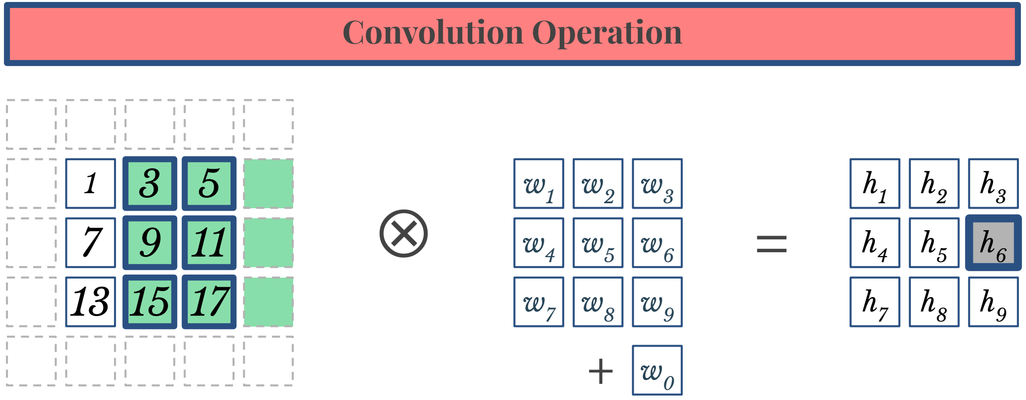


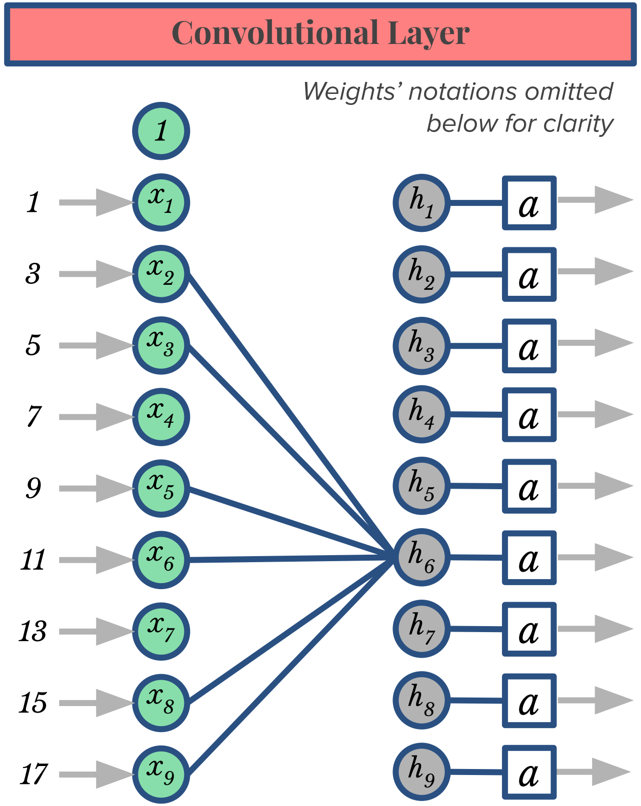

### Convolutional Layer



# Convolution as a Neural Network Layer

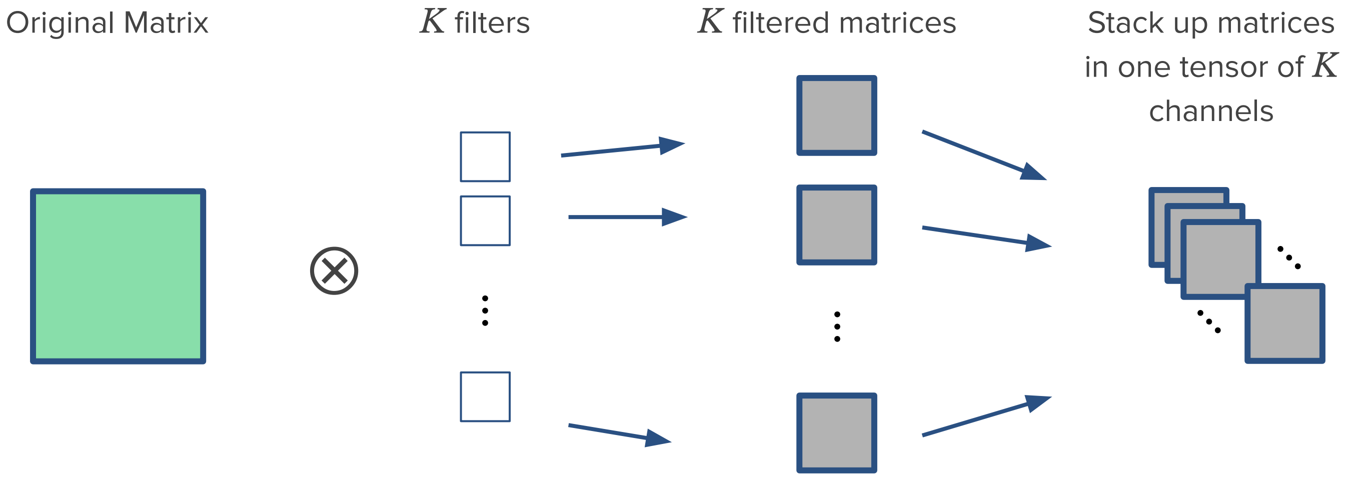
- Convolutions can be implemented as a NN layer, called **Convolutional Layer (ConvLayer)**.
- In it, we only need to zero many of the connections from its input to output (according to the convolution step) and keep the weights the same across units.



# Multiple Filters

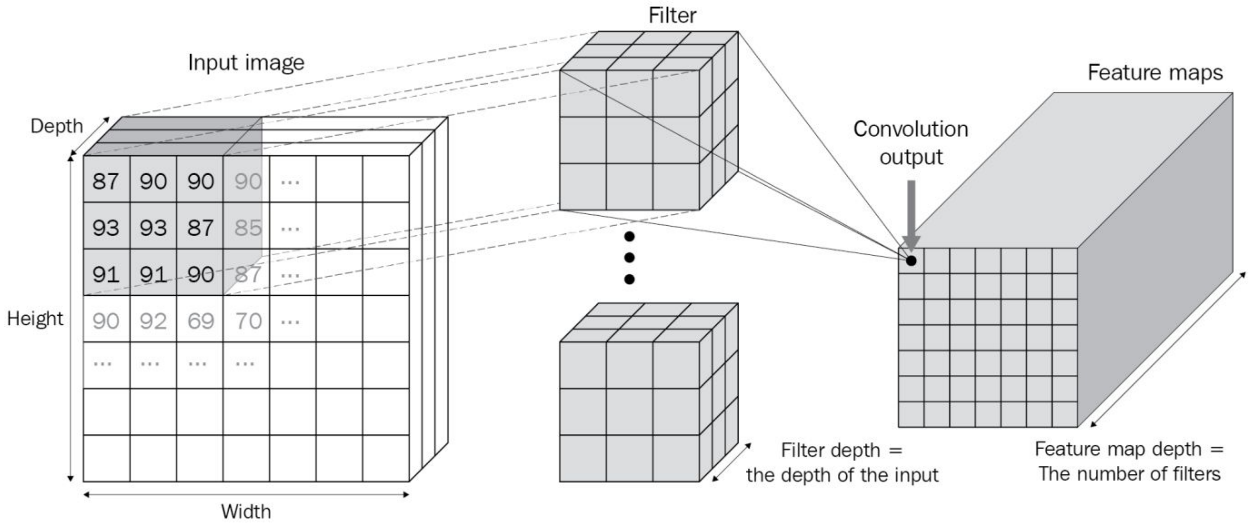
- Notice that we only had to learn $10$ **weights** in the convolutional layer from the last slide.
- In fact, that **wouldn't change** even for larger inputs, if we keep the same kernel size! This opens up the possibility of learning many filters without a big computation burden:




- Note that, in this process, we turned a matrix into a tensor of **multiple channels**.

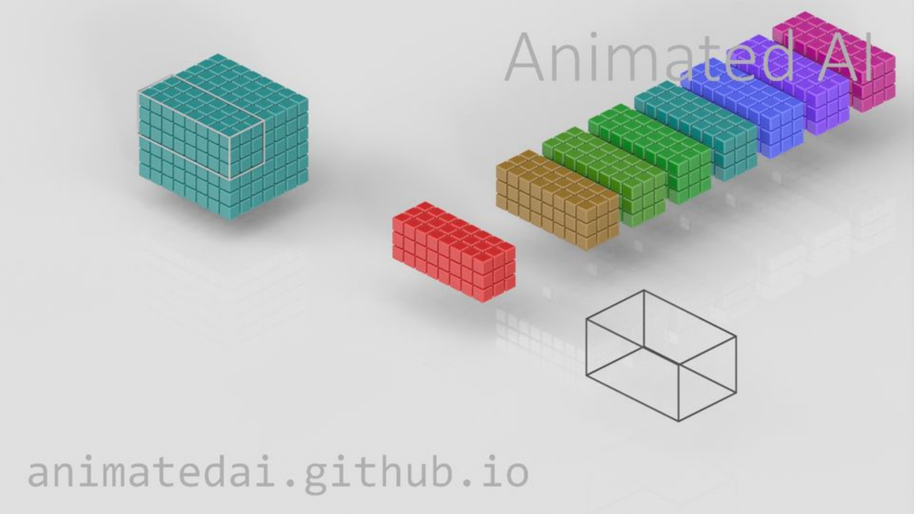
# Convolution with Tensors

- We can generalize the convolution to images (or tensors in general) that have more than one channel.
- In that case, the kernels will have the same number of channels as the input image:



# Convolution with Tensors

- We can generalize the convolution to images (or tensors in general) that have more than one channel.
- In that case, the kernels will have the same number of channels as the input image:




- In the example on the left*, we have:
  - An input tensor of shape `(8,7,6)`, i.e, $8$ channels of size $7{\times}6$.
  - A total of $8$ filters/kernels, each of shape `(8,3,3)`.
  - An output tensor of shape `(8,5,4)`.
- Each channel in of the output tensor (represented by the color of its correspondent filter) is called a feature map in deep learning lingo.

* Check out this animation and more in this link.

# The Max-pooling Operation

- Another important operation and layer in Deep Learning is called **Pooling (layer).**
- The idea is to sweep the initial matrix as in a convolution, but now you apply a standard **not-learnable** operation for each window. The common operations are:
  - **Average-pooling**: take the mean of the values in each window,
  - **Max-pooling**: pick the maximum of each window (by far the most used pooling type.)
- In the following, you have an example of the $2{\times}2$ **Max-pooling with stride** $2$:

Original Matrix

$$\begin{matrix} \boxed{1} & \boxed{2} & \boxed{3} & \boxed{4} \\ \boxed{5} & \boxed{6} & \boxed{7} & \boxed{8} \\ \boxed{9} & \boxed{10} & \boxed{11} & \boxed{12} \\ \boxed{13} & \boxed{14} & \boxed{15} & \boxed{16} \end{matrix}$$

Sweep the matrix using stride 2. $\longrightarrow$

All swept windows

$$\begin{matrix} \boxed{1} & \boxed{2} & & \boxed{3} & \boxed{4} \\ \boxed{5} & \boxed{6} & & \boxed{7} & \boxed{8} \\ & & & & \\ \boxed{9} & \boxed{10} & & \boxed{11} & \boxed{12} \\ \boxed{13} & \boxed{14} & & \boxed{15} & \boxed{16} \end{matrix}$$

Pick the largest value in each block. $\longrightarrow$

Max pooled Matrix

$$\begin{matrix} \boxed{6} & \boxed{8} \\ \boxed{14} & \boxed{16} \end{matrix}$$

- Max-pool layers are used to downscale the input by extracting the most important feature.

# Exercise (*In pairs*)

- Using the two matrices on the right, perform the following operations:
  a. Perform a $2 \times 2$ Max-pooling with stride $2$ on $A$ to create $C$.
  b. Add 2 padding on $C$ and compute its convolution with $B$ to create $D$.
  c. Compute the convolution of $D$ with $B$ as a kernel with stride $3$.
- Write down how we could define a network with batch normalization and dropout using `nn.Sequential()` instead of defining the neural net class.
- During backpropagation, one will have to the derivative of the max-pooling layer if it is present. How does one compute that derivative?

Matrix $A$

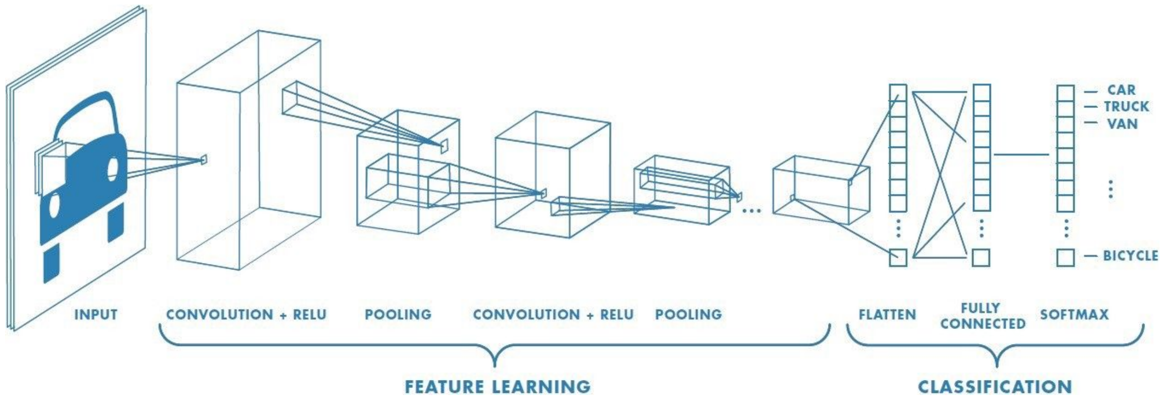
| | | | | | | |
|---|---|---|---|---|---|---|
| 12 | 2 | 29 | 4 | 1 | 2 | 3 |
| 5 | 21 | 7 | 8 | 36 | 6 | 7 |
| 9 | 10 | 11 | 12 | 9 | 10 | 11 |
| 13 | 14 | 15 | 16 | 13 | 14 | 15 |
| 7 | 5 | 10 | 4 | 28 | 20 | 3 |
| 25 | 19 | 12 | 8 | 5 | 20 | 32 |
| 9 | 10 | 11 | 12 | 21 | 10 | 11 |

Matrix $B$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | -2 | 1 |
| 1 | 1 | 1 |

# Convolutional Neural Networks

- With convolutional and pooling layers, we create a **Convolutional Neural Network (CNN)**.
- In image classification tasks, these layers are used as a **feature learning step**, as they
  - Extract relevant features (convolutional layers, *more on it later in the course*)
  - Aggregate information (polling layers).
- After the features are learned, we can flatten them and proceed with our usual Multilayer Perceptron (a series of **dense/linear layers**) for classification:



# Creating a CNN in PyTorch

- In the same way that PyTorch offers `nn.Linear()` to define dense layers, it defines the module `nn.Conv2d()` to define convolutional layers:

```
nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, bias=True)
```

- Here's what these parameters mean*:
  - **`in_channels`** and **`out_channels`**: the number of channels (not the size of each datapoint) of the input and output. For example, if the inputs are grayscale images, and you want to use $32$ filters in that layer, `in_channels = 1` and `out_channels = 32.`
  - **`kernel_size`**: the size of the kernel used in all filters of that layer. Inputting "`3`" here will give you kernels of size $3{\times}3$. All kernels in a given layer have the same size.
  - **`stride`** and **`padding`**: sets the number of stride and the padding "thickness". By default, the convolutions will sweep all the possible windows of the unpadded input.
  - **`bias`**: whether you wish to add a bias term ($w_0$ in this slide).

*Check the documentation here for more details on the layer and on other possible parameters.

# Creating a CNN in PyTorch

- Next, we use PyTorch's `nn.MaxPool2d()` module to define a Max-pooling layer*:

```
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0)
```

- The kernel size and padding parameters work exactly like in the `nn.Conv2d()` module.
- The stride parameter, however, is set to be of **the same size as the kernel size** if `stride=None`.
- In PyTorch, to flatten the output of a ConvLayer to prepare it for the Linear layers, we can use two approaches:
  - If you are defining a class that inherits `nn.Module`, you can use `torch.view()` in its `forward()` method to reshape the output of the ConvLayer.
  - When using `nn.Sequential()` to define a network, we have to add a `nn.Flatten()` module after the layer you want to flatten.

*Check the documentation here for more details on the layer and on other possible parameters.

# Creating the CNN in PyTorch

- We define the following CNN:

```
model = nn.Sequential(nn.Conv2d( 1,64,kernel_size=3),
   nn.ReLU(),
   nn.MaxPool2d(2),
   nn.Conv2d(64, 128, kernel_size=3),
   nn.ReLU(),
   nn.MaxPool2d(2),
   nn.Flatten(),
   nn.Linear(3200, 200),
   nn.ReLU(),
   nn.Linear(200, 10)
).to(device)
```

- It has only two ConvLayers, that learn $64$ and $128$ filters, resp., each followed by a $2{\times}2$ Max-pool layer.
- The CNN part is then followed by a dense layer with $200$ units.

- If we print our CNN's summary we get:

```
from torchsummary import summary
summary(model, (1, 28, 28)) # Notice the new shape
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 26, 26]             640
              ReLU-2           [-1, 64, 26, 26]               0
         MaxPool2d-3           [-1, 64, 13, 13]               0
            Conv2d-4          [-1, 128, 11, 11]          73,856
              ReLU-5          [-1, 128, 11, 11]               0
         MaxPool2d-6            [-1, 128, 5, 5]               0
           Flatten-7                 [-1, 3200]               0
            Linear-8                  [-1, 200]         640,200
              ReLU-9                  [-1, 200]               0
           Linear-10                   [-1, 10]           2,010
================================================================
Total params: 716,706
Trainable params: 716,706
Non-trainable params: 0
----------------------------------------------------------------(...)
```

Notice how many weights to learn!

# Changing the dataset class

- Because we want to use the images in their original shapes, we don't need to flatten them when defining the Dataset. So our new class definition to (compare it to the original):

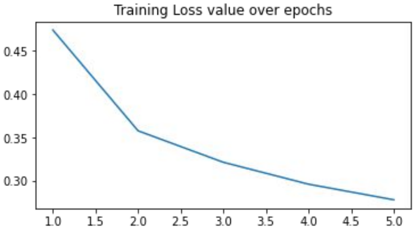
```
class FMNISTDataset(Dataset):
  def __init__(self, x, y):
      x = x.view(-1, 1, 28, 28)
      x = x.float()/255
      self.x, self.y = x, y
  def __getitem__(self, ix):
      return self.x[ix].to(device),self.y[ix].to(device)
  def __len__(self):
      return len(self.x)
```

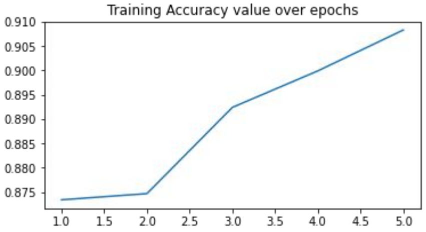
- Note that we explicitly show that each datapoint has one channel (since they are grayscale images) and size $28{\times}28$.
- Defining the data to show how many channels it has **is necessary** when using ConvLayers in the beginning of your model.

# Training and testing the CNN

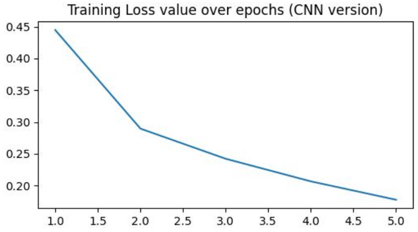
- Now, we can train our Convolutional Neural Network under the exact same parameters as the Multilayer Perceptron of the previous slides and compare their performance:

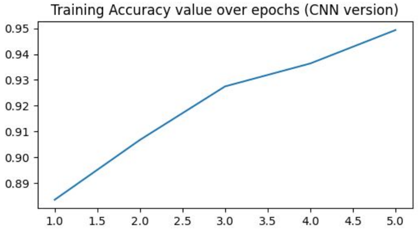
**MLP** – n. weights: 795k  training time: 43s




**CNN** – n. weights: 716k, training time: 54s




- If we test our learned CNN, we also see the improvement compared to the MLP one:

```
Test accuracy: 0.9146365523338318
```

- *In sum*: with fewer weights to be learned, we were able to improve our results by using ConvLayers and Max-pooling, all thanks to the magic of **feature learning**!

# Exercise (*In pairs*)

Click here to open code in Colab

- On the right you have the summary of today's CNN. Explain:
  - Why does the output shape progresses like depicted?
  - Why does each layer have the number of weights (parameters) it says it has.

  Here, the input shape is *(1, 28, 28)* and the convolutions have *3×3* kernels.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 26, 26]             640
              ReLU-2           [-1, 64, 26, 26]               0
         MaxPool2d-3           [-1, 64, 13, 13]               0
            Conv2d-4          [-1, 128, 11, 11]          73,856
              ReLU-5          [-1, 128, 11, 11]               0
         MaxPool2d-6            [-1, 128, 5, 5]               0
           Flatten-7                 [-1, 3200]               0
            Linear-8                  [-1, 200]         640,200
              ReLU-9                  [-1, 200]               0
           Linear-10                   [-1, 10]           2,010
================================================================
Total params: 716,706
Trainable params: 716,706
Non-trainable params: 0
----------------------------------------------------------(...)
```

# *Video*: Go AlphaGo!