Convolutional Neural Network (CNN)'s Implementation

Prof. Pei-Jun Lee

Course: Deep Learning based image recognition

Video Signal Processing and Application Lab.

Outline

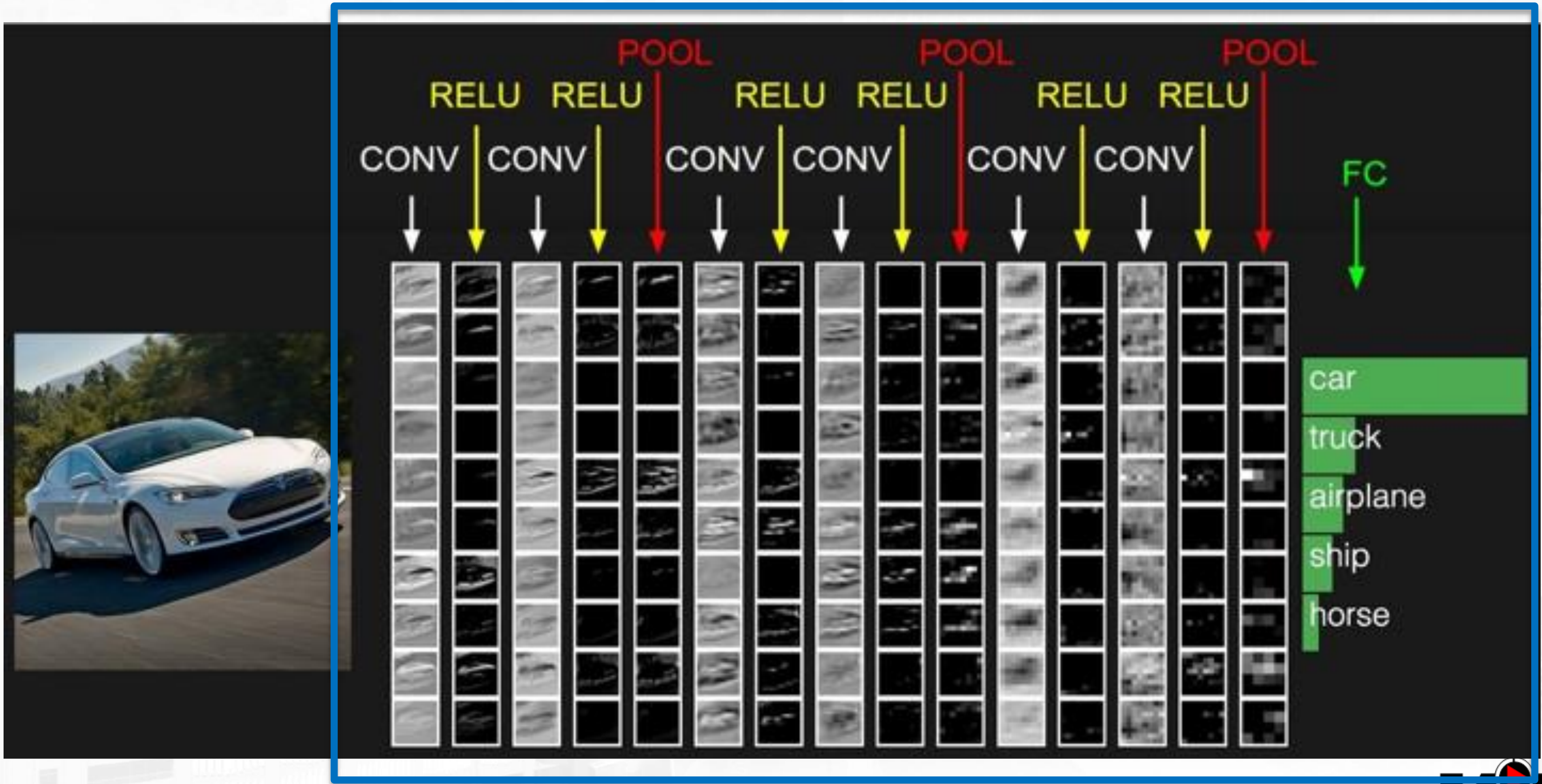
- **Convolution layers implementation**
- Simple Convolution Neural Network Implementation
- EfficientNets implementation

PyTorch: TORCH.NN

PyTorch provides the elegantly designed *modules* and *classes* torch.nn , torch.optim , Dataset , and DataLoader to create and train neural networks.

- NN Module.
- NN Classes: Convolution, ReLu, Pooling…

CNN: Structure



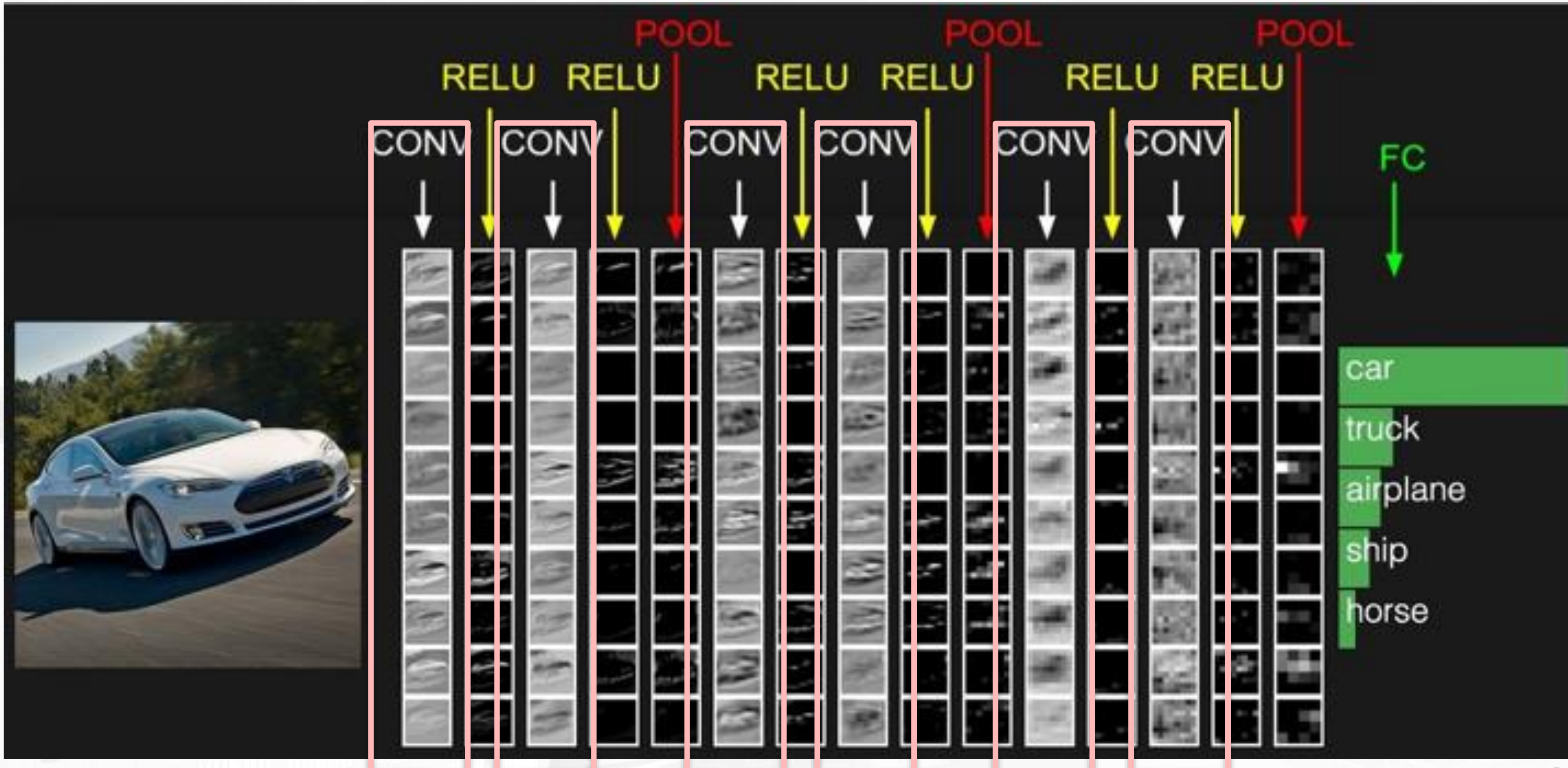
NN Module

Course: Deep Learning based image recognition

Prof. Pei-Jun Lee

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

CNN: Structure



Convolution Layer

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

elementwise multiplication and sum of a filter and the signal (image)

Activation maps

NN CONV2D

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

NN CONV2D

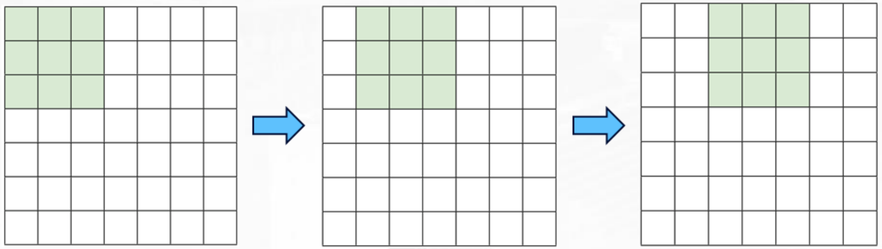
CONV2D

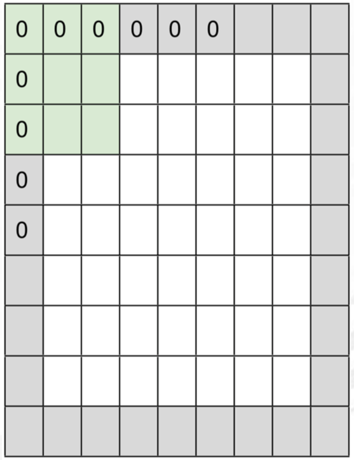
CLASS torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

[2]:
```
# With square kernels and equal stride
m = nn.Conv2d(16, 33, 3, stride=2)
# non-square kernels and unequal stride and with padding
m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
# non-square kernels and unequal stride and with padding and dilation
m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
```

NN CONV2D

Stride **1**





e.g. input 7x7, **3x3** filter, applied with **stride 1** **pad with 1 pixel** border

[2]:
```
# With square kernels and equal stride
m = nn.Conv2d(16, 33, 3, stride=2)
# non-square kernels and unequal stride and with padding
m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
# non-square kernels and unequal stride and with padding and dilation
m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
```

NN CONV2D

QUESTION 1: ***Implementation the Python code on your PC, and explain the reason why the output is different.***

- **RANDOM A MATRIX**
- Flowing the different CONV2D on previous slide, show 3 different output of Conv2d in the case of
  - stride = 2
  - stride=(2, 1), padding=(4, 2)
  - stride=(2, 1), padding=(4, 2), dilation=(3, 1)

```
[27]: input = torch.randn(20, 16, 50, 100)
      input

[27]: tensor([[[[ 1.8860e-01,  2.4087e-01,  1.5497e+00,  ..., -3.0938e-01,
                  4.0887e-01, -2.9717e-01],
                [ 9.2806e-01,  1.0157e+00,  5.1952e-01,  ...,  2.8216e-02,
                 -6.6362e-03, -5.2349e-01],
                [-9.6317e-02,  4.6547e-01,  1.3554e-02,  ..., -8.3604e-02,
                  2.6012e-01, -2.9451e-01],
                ...,
                [-1.1337e+00, -6.8217e-02, -1.4225e+00,  ...,  9.0001e-02,
                 -2.3670e-02, -1.6538e-01],
                [-4.3143e-01, -7.8885e-01,  6.1664e-02,  ..., -3.4750e-01,
                  9.4764e-01, -1.7202e+00],
                [ 1.4547e+00,  1.5792e+00, -3.3765e-01,  ...,  1.4484e+00,
                 -2.2056e+00,  2.4060e+00]],
```

NN CONV2D

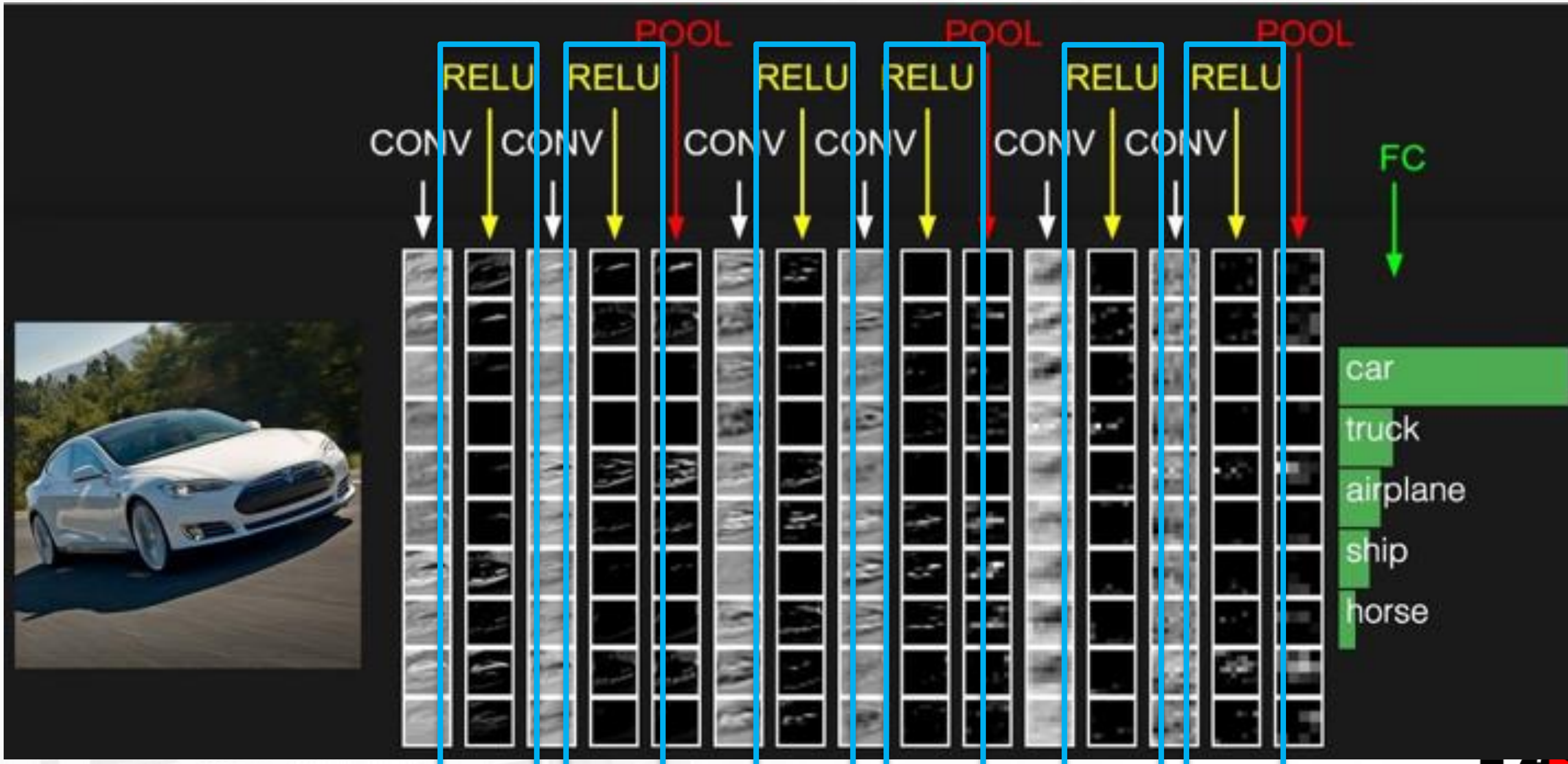
QUESTION 1: ***Implementation the Python code on your PC, and explain the reason why the output is different.***

- RANDOM A MATRIX
- Flowing the different CONV2D on previous slide, **show 3 different output of Conv2d in the case of**
  - **stride = 2**
  - **stride=(2, 1), padding=(4, 2)**
  - **stride=(2, 1), padding=(4, 2), dilation=(3, 1)**

```
[28]: output = m(input)
      output
```

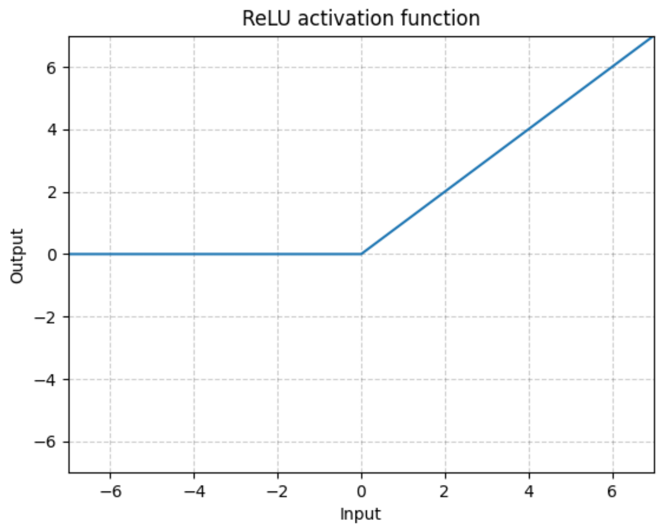
```
[28]: tensor([[[[ 1.1569e-01, -1.0741e-01, -7.0935e-01,  ...,  2.1281e-01,
              -3.6033e-01, -1.9710e-01],
             [-6.5989e-03, -1.6566e-01, -2.7000e-01,  ..., -3.0469e-01,
              -3.7350e-01, -4.8443e-01],
             [ 1.1702e-01,  6.3829e-01, -6.9597e-03,  ..., -3.6604e-01,
              -3.0093e-01, -1.0985e-01],
             ...,
```

CNN: RELU



POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

RELU function



CLASS `torch.nn.ReLU(inplace=False)` [SOURCE]

Applies the rectified linear unit function element-wise:

$$\mathrm{ReLU}(x) = (x)^+ = \max(0, x)$$

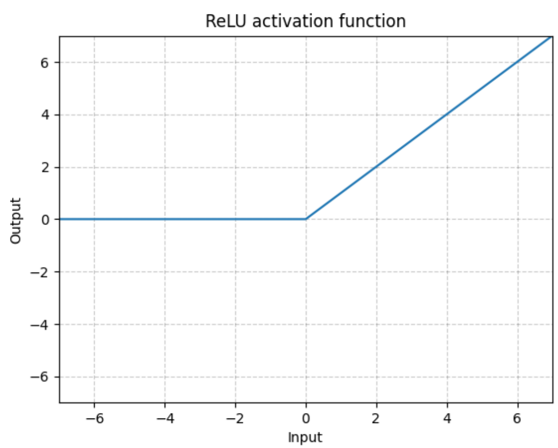
Parameters

inplace – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(*)$, where $*$ means any number of dimensions.
- Output: $(*)$, same shape as the input.

RELU function




```
[34]: m = nn.ReLU()
      input = torch.randn(5,5)
      input
```

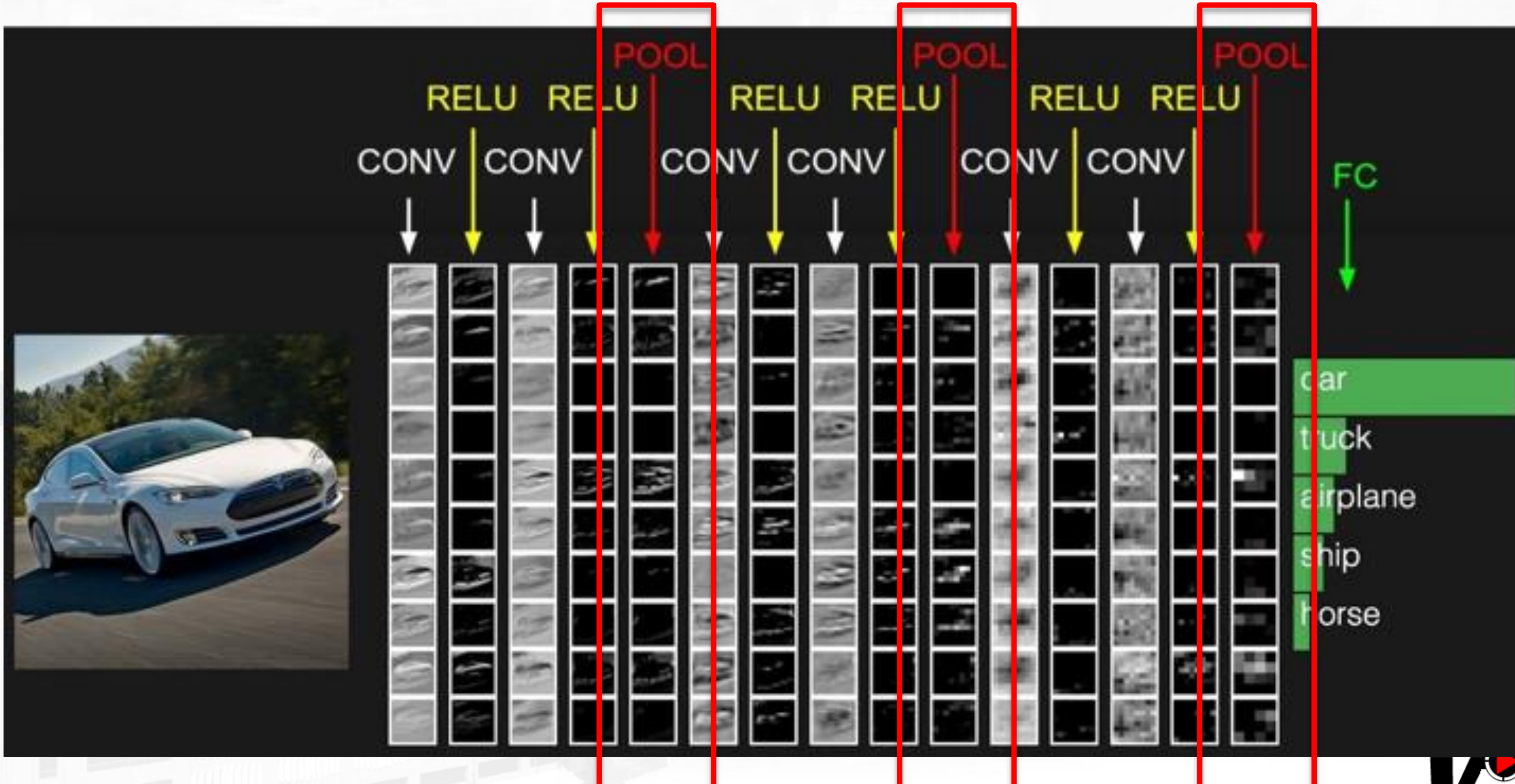
```
[34]: tensor([[ 0.7753, -1.6358,  1.1919, -0.6742, -0.4419],
              [ 0.3367, -0.8557,  0.8647, -1.6617,  1.0797],
              [-0.3697, -0.6958, -0.0910,  1.1365,  0.5092],
              [-2.4236,  0.0714, -0.6627, -0.0876,  1.3807],
              [ 0.5123, -0.9011,  1.3277,  0.3821, -1.7388]])
```

```
[35]: output = m(input)
      output
```

```
[35]: tensor([[0.7753, 0.0000, 1.1919, 0.0000, 0.0000],
              [0.3367, 0.0000, 0.8647, 0.0000, 1.0797],
              [0.0000, 0.0000, 0.0000, 1.1365, 0.5092],
              [0.0000, 0.0714, 0.0000, 0.0000, 1.3807],
              [0.5123, 0.0000, 1.3277, 0.3821, 0.0000]])
```

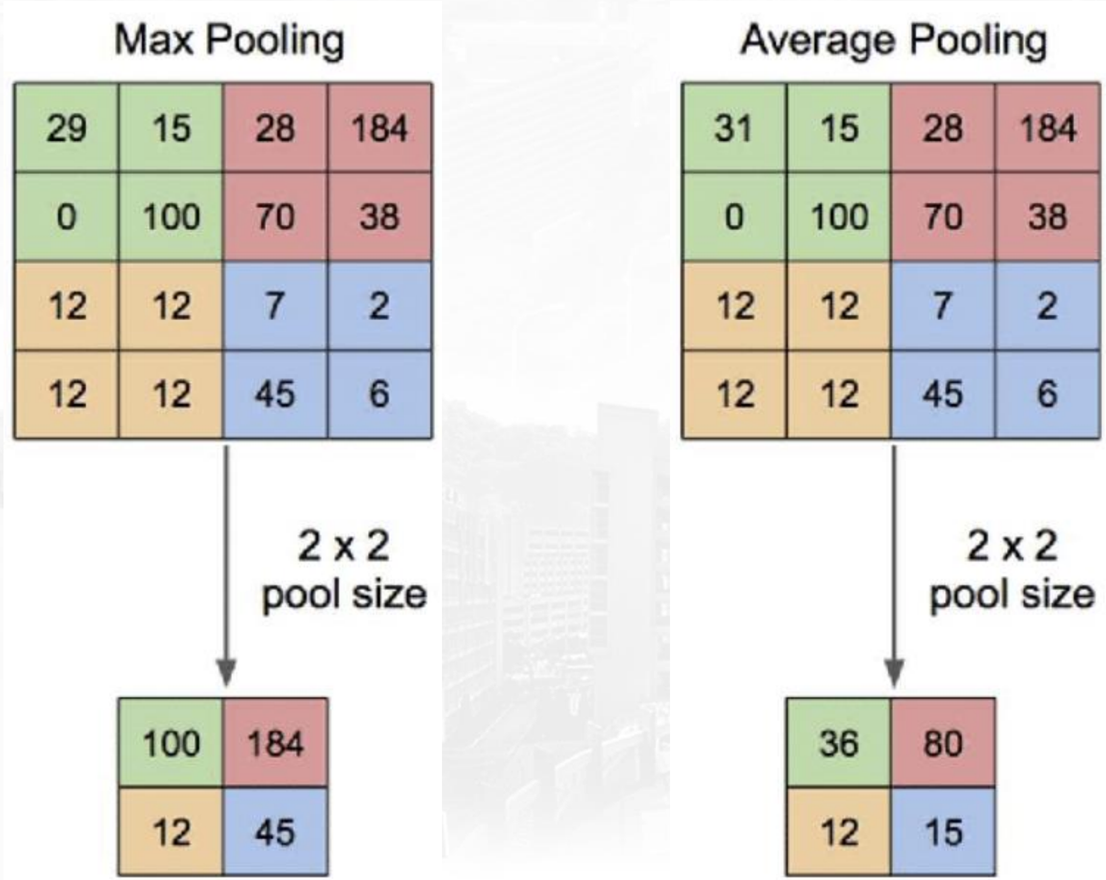
<u>QUESTION 2:</u> ***Implementation the Python code on your PC***

CNN: POOL



POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

Pooling layer



MAX Pooling function

MAXPOOL2D

CLASS `torch.nn.MaxPool2d(`*kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False*`)` [SOURCE]

Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

If `padding` is non-zero, then the input is implicitly padded with negative infinity on both sides for `padding` number of points. `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

MAX Pooling function

```
[50]: # pool of square window of size=3, stride=2
m = nn.MaxPool2d(3, stride=2)
# pool of non-square window
m = nn.MaxPool2d((3, 2), stride=(2, 1))
```

Shape:

- Input: (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

MAX Pooling function

Input

```
[51]: input = torch.randn(2, 3, 4, 4)
      input
```

```
[51]: tensor([[[[ 1.8795, -0.9712, -0.9392, -0.4232],
          [ 0.8934, -0.1838, -0.6962,  0.7811],
          [-0.7759,  0.6899,  1.2669, -0.4593],
          [-0.3234,  0.1564,  0.3186, -0.6218]],

         [[ 0.1849, -0.8552, -0.4428, -0.0164],
          [ 0.9378, -0.0277,  0.7987, -0.5503],
          [ 1.1780,  0.5824,  0.1628, -0.2823],
          [-0.8981, -1.6407, -0.1553, -0.4629]],

         [[ 0.4457,  0.3343,  0.2223,  0.0144],
          [-0.6155,  0.8187, -0.9193,  0.7510],
          [ 1.1858, -1.3562, -2.2173,  0.3620],
          [-0.8036, -1.3456, -1.8544,  1.3941]]],


        [[[-0.0728, -1.2540,  0.9493,  0.2052],
          [ 0.4544, -0.1516, -0.8139, -2.2012],
          [ 0.2120,  0.1947,  1.2858, -1.4137],
          [ 1.6568, -1.3123,  0.6329,  0.1649]],

         [[-0.0990, -0.9462, -0.0781,  0.2328],
          [-0.7766,  0.2643, -0.9084, -0.1353],
          [ 0.9885, -0.0519, -2.2123, -0.9353],
          [-0.7118,  0.1378, -0.8299, -0.5661]],

         [[-0.2325,  1.4443, -1.2123,  1.0013],
          [-1.0646, -0.5381,  0.4471, -0.7720],
          [ 0.1712, -0.4545, -0.7615, -0.4085],
          [-0.3185, -1.9137, -1.5205,  0.1778]]]])
```

Output of

m = nn.MaxPool2d(3, stride=2)

```
[52]: output = m(input)
      output
```

```
[52]: tensor([[[[1.8795]],

          [[1.1780]],

          [[1.1858]]],


         [[[1.2858]],

          [[0.9885]],

          [[1.4443]]]])
```

Output of

m = nn.MaxPool2d((3, 2), stride=(2, 1))

```
[54]: output = m(input)
      output
```

```
[54]: tensor([[[[1.8795, 1.2669, 1.2669]],

          [[1.1780, 0.7987, 0.7987]],

          [[1.1858, 0.8187, 0.7510]]],


         [[[0.4544, 1.2858, 1.2858]],

          [[0.9885, 0.2643, 0.2328]],

          [[1.4443, 1.4443, 1.0013]]]])
```

QUESTION 3:

Implementation the Python code on your PC, and explain the reason why the output is different.

AVERAGE Pooling function

AVGPOOL2D

CLASS `torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)` [SOURCE]

Applies a 2D average pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points.

AVERAGE Pooling function

Input

```
[51]: input = torch.randn(2, 3, 4, 4)
      input

[51]: tensor([[[[ 1.8795, -0.9712, -0.9392, -0.4232],
                [ 0.8934, -0.1838, -0.6962,  0.7811],
                [-0.7759,  0.6899,  1.2669, -0.4593],
                [-0.3234,  0.1564,  0.3186, -0.6218]],

               [[ 0.1849, -0.8552, -0.4428, -0.0164],
                [ 0.9378, -0.0277,  0.7987, -0.5503],
                [ 1.1780,  0.5824,  0.1628, -0.2823],
                [-0.8981, -1.6407, -0.1553, -0.4629]],

               [[ 0.4457,  0.3343,  0.2223,  0.0144],
                [-0.6155,  0.8187, -0.9193,  0.7510],
                [ 1.1858, -1.3562, -2.2173,  0.3620],
                [-0.8036, -1.3456, -1.8544,  1.3941]]],


              [[[-0.0728, -1.2540,  0.9493,  0.2052],
                [ 0.4544, -0.1516, -0.8139, -2.2012],
                [ 0.2120,  0.1947,  1.2858, -1.4137],
                [ 1.6568, -1.3123,  0.6329,  0.1649]],

               [[-0.0990, -0.9462, -0.0781,  0.2328],
                [-0.7766,  0.2643, -0.9084, -0.1353],
                [ 0.9885, -0.0519, -2.2123, -0.9353],
                [-0.7118,  0.1378, -0.8299, -0.5661]],

               [[-0.2325,  1.4443, -1.2123,  1.0013],
                [-1.0646, -0.5381,  0.4471, -0.7720],
                [ 0.1712, -0.4545, -0.7615, -0.4085],
                [-0.3185, -1.9137, -1.5205,  0.1778]]]])
```

Output of average pooling 1

```
output = m(input)
output

tensor([[[[ 0.1293]],

         [[ 0.2799]],

         [[-0.2335]]],


        [[[ 0.0893]],

         [[-0.4244]],

         [[-0.2446]]]])
```

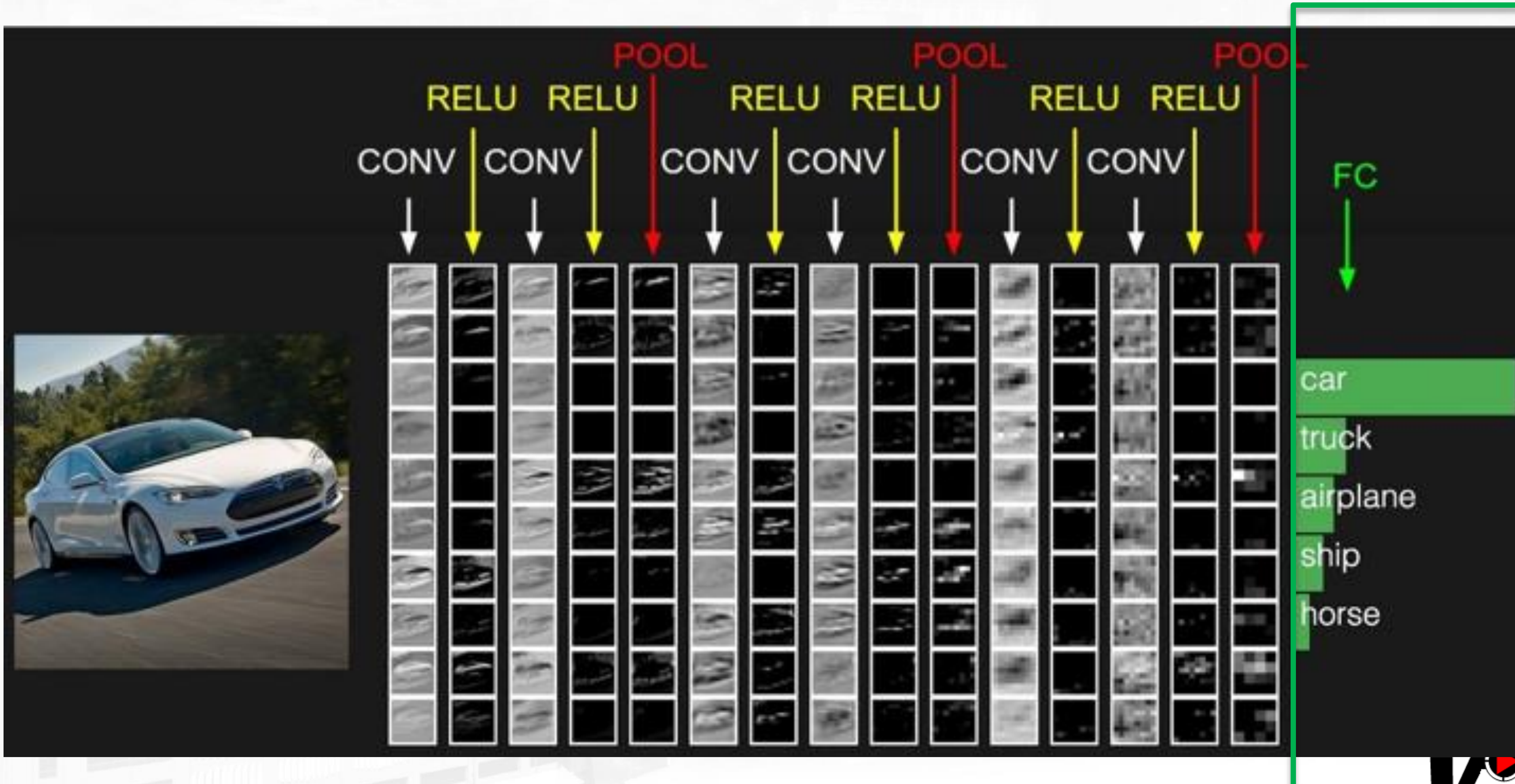
Output of average pooling 2

```
output = m(input)
output

tensor([[[[ 0.2553, -0.1390, -0.0783]],

         [[ 0.3334,  0.0364, -0.0551]],

         [[ 0.1355, -0.5196, -0.2978]]],


        [[[-0.1029,  0.0350, -0.3314]],

         [[-0.1035, -0.6554, -0.6728]],

         [[-0.1124, -0.1792, -0.2843]]]])
```

QUESTION 4:

Write the Python code for Average pooling 1 and 2

CNN: FC


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

Fully Connected Layer

This ***Flattened vector*** is then connected to a few fully connected layers (same as Artificial Neural Networks) and perform the same mathematical operations.

Artificial Neural Networks (last week contents)

$$Y = \sum (weight * input) + bias$$

$$g(Wx + b)$$

x is the input vector with dimension $[p_l, 1]$
W is the weight matrix with dimensions $[p_l, n_l]$ where, p_l is the number of neurons in the previous layer and n_l is the number of neurons in the current layer.
b is the bias vector with dimension $[p_l, 1]$
g — Is the activation function, which is usually ReLU.

Fully Connected Layer function

LINEAR

CLASS `torch.nn.Linear(`*in_features, out_features, bias=True, device=None, dtype=None*`)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Fully Connected Layer function

```
[63]: m = nn.Linear(20, 30)

[65]: input = torch.randn(2, 20)
      input

[65]: tensor([[-0.6686, -0.6767, -0.0565, -0.5262,  0.3592, -1.1600,  0.3646,  0.7910,
               -2.0160,  0.8577,  0.1639, -0.7672, -0.9179, -0.9620,  0.3510, -0.6575,
               -0.3823,  0.3325,  0.1125,  0.0372],
              [ 0.2882, -0.7986, -2.0156, -1.7547,  1.9898,  0.0356, -0.4335, -0.1389,
                1.1506,  0.1132, -0.1702, -0.8904, -0.9243, -0.5073, -0.8827, -0.1358,
                0.1609,  0.4834,  0.5100, -1.3671]])

[66]: output = m(input)
      output

[66]: tensor([[ 0.2792,  1.1089,  0.2182, -0.5451,  0.1285,  0.5692, -0.0080,  0.2321,
               -0.0058,  1.0021,  0.1688, -0.0619, -0.2699, -0.1167,  0.1608, -0.2449,
                0.0687, -0.4368,  0.3151, -0.7890,  0.4795,  0.3152, -0.1806, -0.4197,
                0.5918, -0.0309,  0.0733, -0.5721, -0.0938,  0.0247],
              [ 0.5794,  0.4321,  0.6116,  0.4074, -0.8108, -0.1720, -0.0687, -0.2666,
               -0.0355, -0.1920, -0.0296,  0.3457, -0.6254,  0.0858,  0.2384,  0.4128,
               -0.0604, -0.0173,  0.1590, -0.1716,  0.3281,  0.0714, -0.3995, -0.3633,
               -0.7026, -0.5622, -0.8118,  0.3799, -0.6657,  0.9333]],
             grad_fn=<AddmmBackward0>)
```

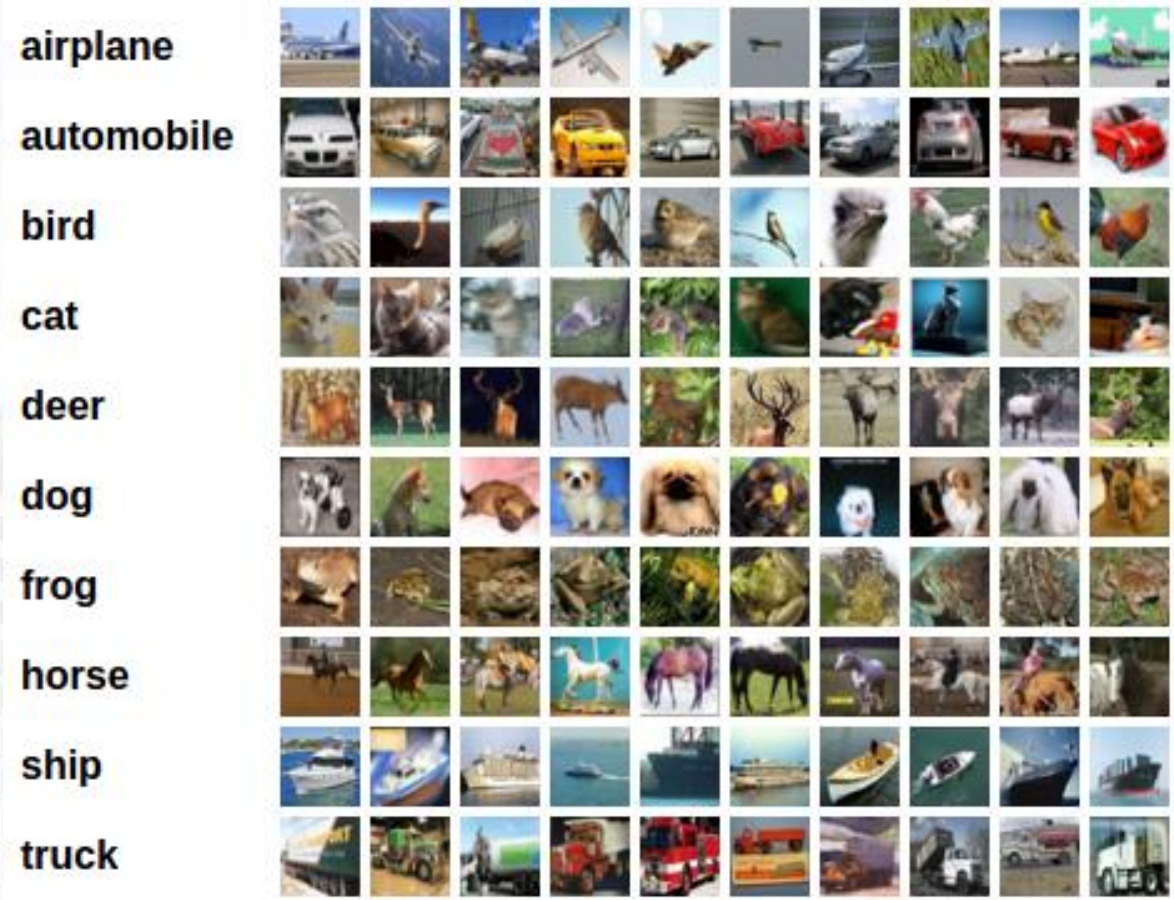
QUESTION 5:

Write the Python code for FC layer

Outline

- Convolution layers implementation
- **Simple Convolution Neural Network Implementation**
- EfficientNets implementation

W8: Image classification: CFAR-10



W8: Image classification: CFAR-10

Load and normalize the CIFAR10 training and test datasets using torchvision

[2]:
```
import torch
import torchvision
import torchvision.transforms as transforms
```

[*]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

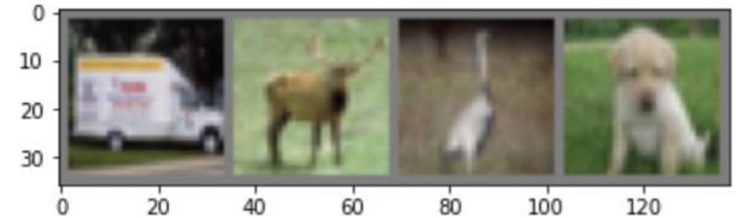
```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
```

26% 43664384/170498071 [00:04<00:09, 13784498.68it/s]

W8: Image classification: CFAR-10

Verify CIFAR10 datasets

```
[4]: import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```




```
truck deer  bird  dog
```

Image classification: CFAR-10

From the implementation on Week 8, change the simple CNN code to a new CNN architecture (LeNET):

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

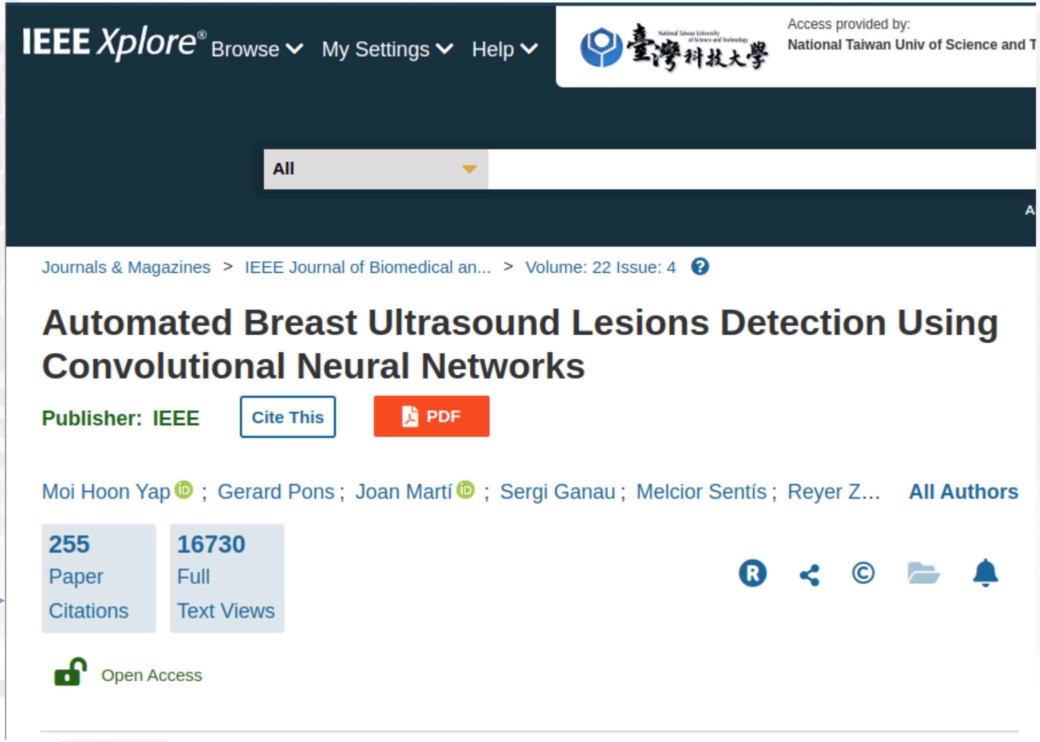
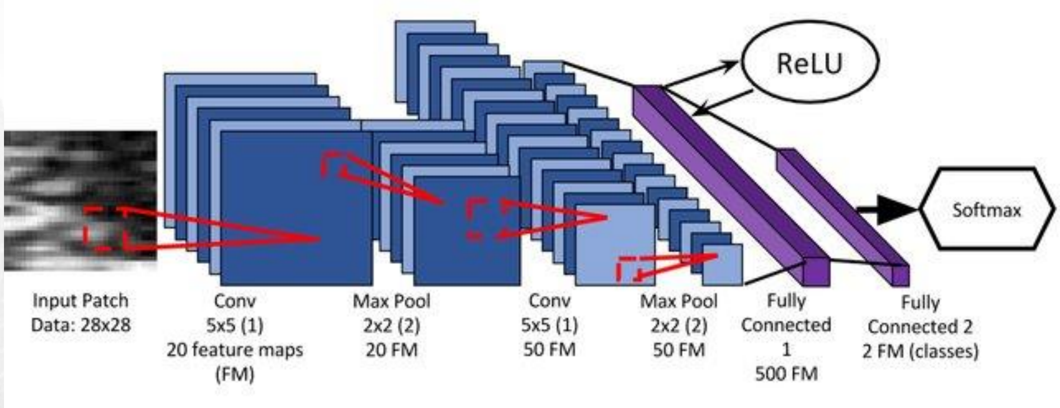


Image classification: CFAR-10

From the implementation on Week 8, change the simple CNN code to a new CNN architecture (**LeNET**):

[5]:
```python
import torch.nn as nn
import torch.nn.functional as func

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = func.relu(self.conv1(x))
        x = func.max_pool2d(x, 2)
        x = func.relu(self.conv2(x))
        x = func.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = func.relu(self.fc1(x))
        x = func.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```



Compare Accuracy

```
[16]:
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

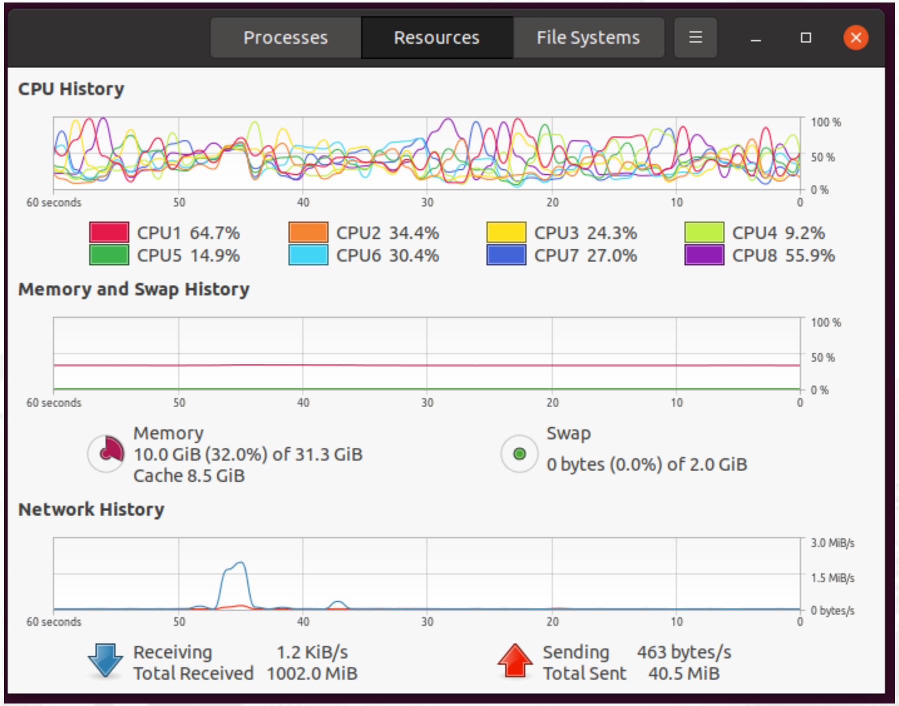
```
Accuracy for class: plane is 59.6 %
Accuracy for class: car   is 68.7 %
Accuracy for class: bird  is 48.1 %
Accuracy for class: cat   is 49.6 %
Accuracy for class: deer  is 61.7 %
Accuracy for class: dog   is 49.3 %
Accuracy for class: frog  is 68.3 %
Accuracy for class: horse is 66.5 %
Accuracy for class: ship  is 75.5 %
Accuracy for class: truck is 75.3 %
```

Week 8:
Simple NN

```
Accuracy for class: plane is 69.1 %
Accuracy for class: car   is 72.8 %
Accuracy for class: bird  is 44.9 %
Accuracy for class: cat   is 44.9 %
Accuracy for class: deer  is 55.1 %
Accuracy for class: dog   is 49.3 %
Accuracy for class: frog  is 79.6 %
Accuracy for class: horse is 70.1 %
Accuracy for class: ship  is 78.4 %
Accuracy for class: truck is 70.5 %
```

Current:
LeNet

W8: Simple CNN implementation




```
Every 2.0s: nvidia-smi

Sun May  8 12:37:29 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.103.01   Driver Version: 470.103.01   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:01:00.0  On |                  N/A |
| 35%   37C    P2    45W / 260W |   2158MiB / 11018MiB |     15%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1340      G   /usr/lib/xorg/Xorg                 53MiB |
|    0   N/A  N/A      5782      G   /usr/lib/xorg/Xorg                347MiB |
|    0   N/A  N/A      5909      G   /usr/bin/gnome-shell               98MiB |
|    0   N/A  N/A      6490      G   ...AAAAAAAAA= --shared-files       23MiB |
|    0   N/A  N/A      6522      G   ...847102908930284502,131072       63MiB |
|    0   N/A  N/A     10114      G   /usr/lib/firefox/firefox          161MiB |
|    0   N/A  N/A     12738      G   .../debug.log --shared-files       19MiB |
|    0   N/A  N/A     19034      G   ...RendererForSitePerProcess       10MiB |
|    0   N/A  N/A     30220      C   ...ngan93/pt-gpu/bin/python3     1363MiB |
+-----------------------------------------------------------------------------+
```

GPUs executing

Question 6:

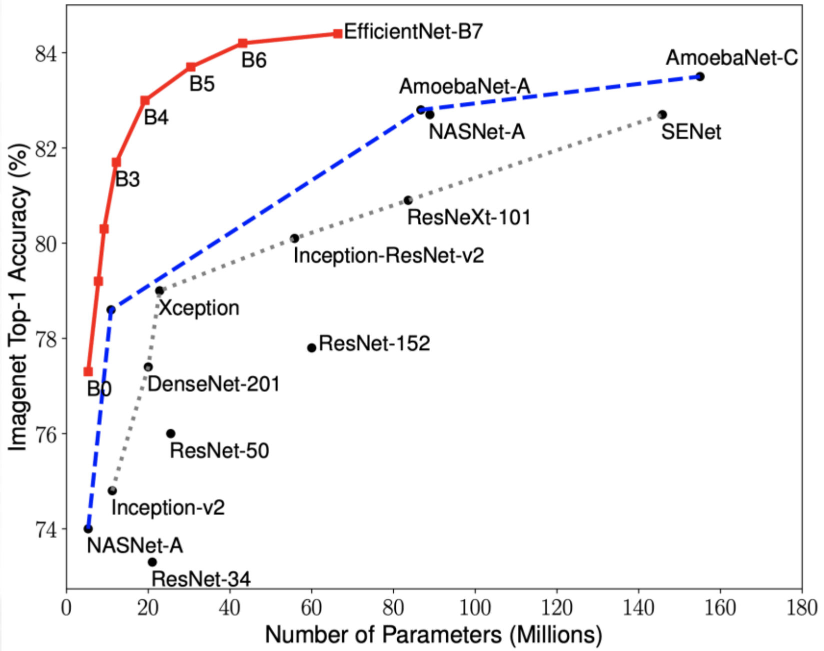
Write the Python code to implements the LeNET with CIFAR-10 dataset.

- Show the accuracy per classes of CIFAR-10 dataset: plane, car, bird, cat, deer, dog, frog, horse, ship, truck
- Compare with the old Neural Network model (on Week 8's implementation): following the example in slide 33.

Outline

- Convolution layers implementation
- Simple Convolution Neural Network Implementation
- **EfficientNets implementation**

Efficient Nets



https://arxiv.org/abs/1905.11946

EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

Mingxing Tan, Quoc V. Le

Convolutional Neural Networks (ConvNets) are commonly developed at a fixed resource budget, and then scaled up for better accuracy if more resources are available. In this paper, we systematically study model scaling and identify that carefully balancing network depth, width, and resolution can lead to better performance. Based on this observation, we propose a new scaling method that uniformly scales all dimensions of depth/width/resolution using a simple yet highly effective compound coefficient. We demonstrate the effectiveness of this method on scaling up MobileNets and ResNet.

To go even further, we use neural architecture search to design a new baseline network and scale it up to obtain a family of models, called EfficientNets, which achieve much better accuracy and efficiency than previous ConvNets. In particular, our EfficientNet-B7 achieves state-of-the-art 84.3% top-1 accuracy on ImageNet, while being 8.4x smaller and 6.1x faster on inference than the best existing ConvNet. Our EfficientNets also transfer well and achieve state-of-the-art accuracy on CIFAR-100 (91.7%), Flowers (98.8%), and 3 other transfer learning datasets, with an order of magnitude fewer parameters. Source code is at this https URL.

Question 7: (Homework)

Write the Python code to implements the Efficient Nets with CIFAR-10 dataset.

- Show the accuracy per classes of CIFAR-10 dataset: plane, car, bird, cat, deer, dog, frog, horse, ship, truck
- Compare with the **old Neural Network** model (on Week 8's implementation) and ***Efficient*** : following the example in slide 33.

Convolution Neural Network

Question and Answer!