Classification (1)

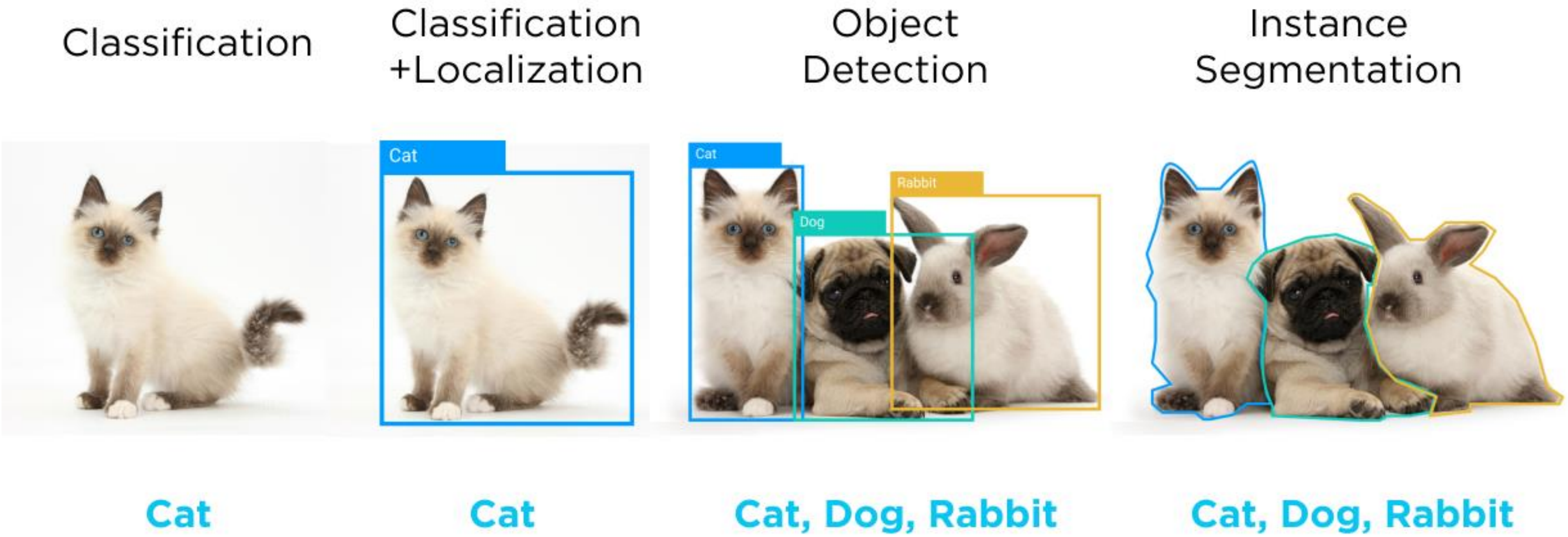
POSTECH MIP Lab.

TA: Joonhyuk Park, Seunghun Baek, Soojin Hwang

Goal

- Understand classification task
- Understand the development of CNN based classification models (**AlexNet, VGG**, GoogLeNet, ResNet)
- Learn how to implement CNN models from architecture

Classification task



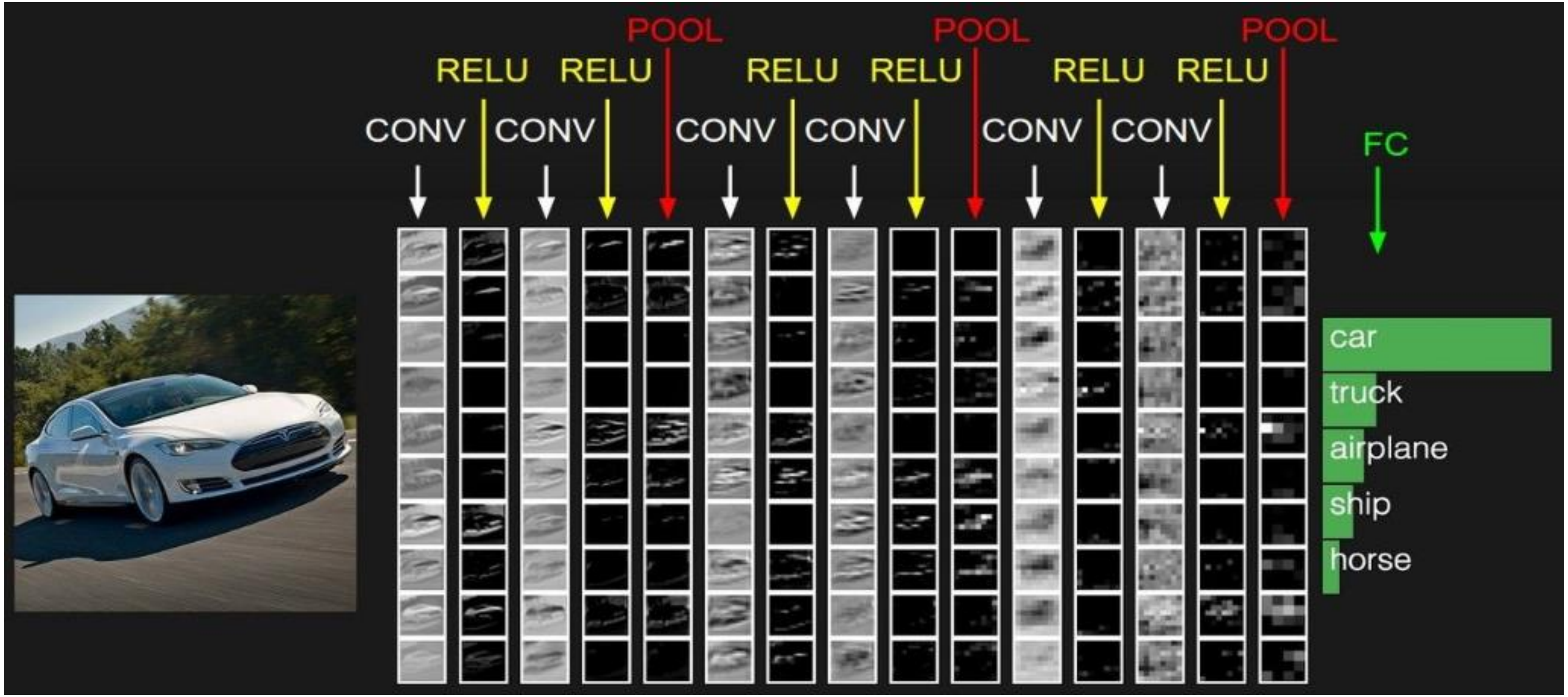
Classification task

```
def classify_image(image):
  # Some magic here?
  return class_label
```

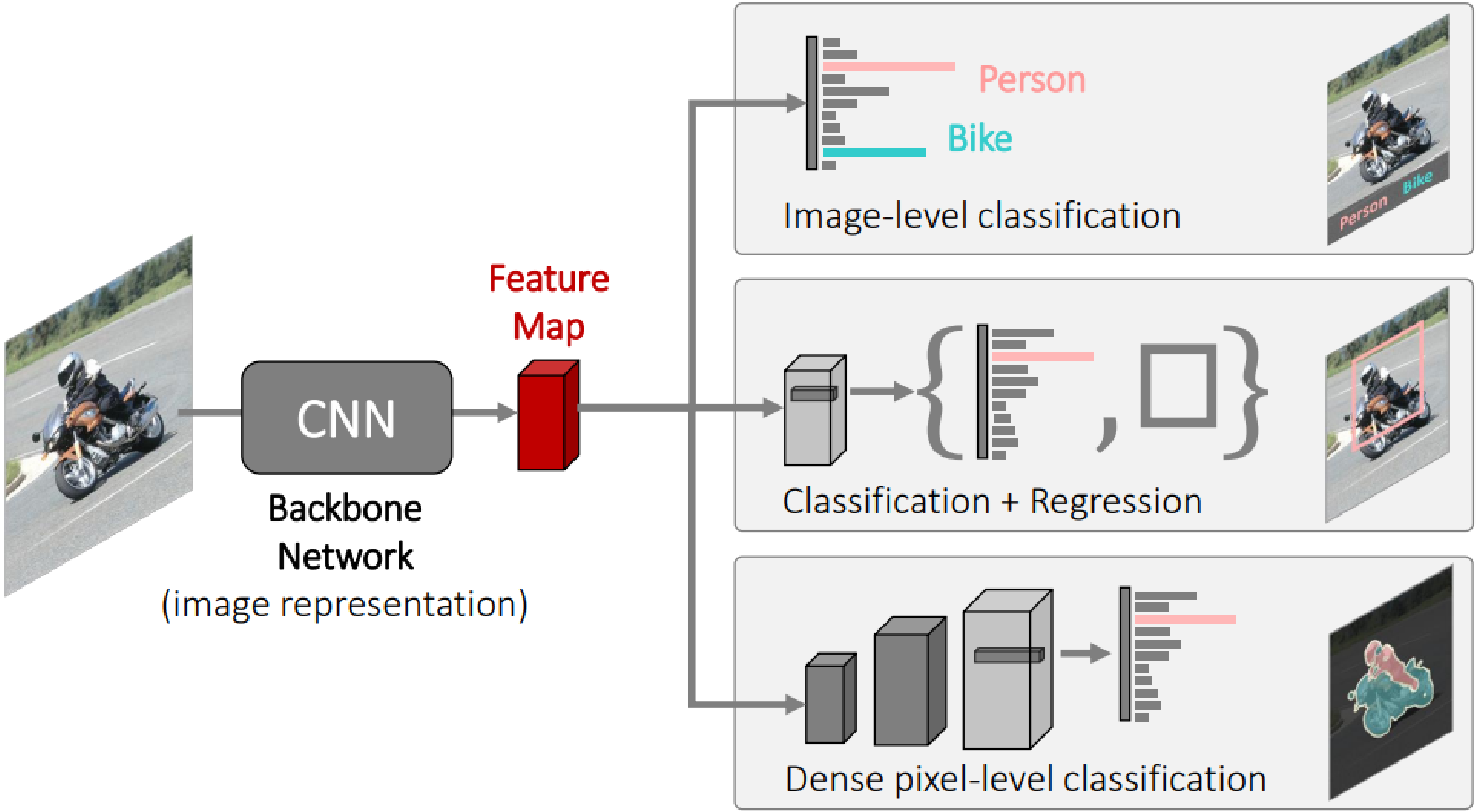
Cat

Goal : Classify the label for each image

Classification task



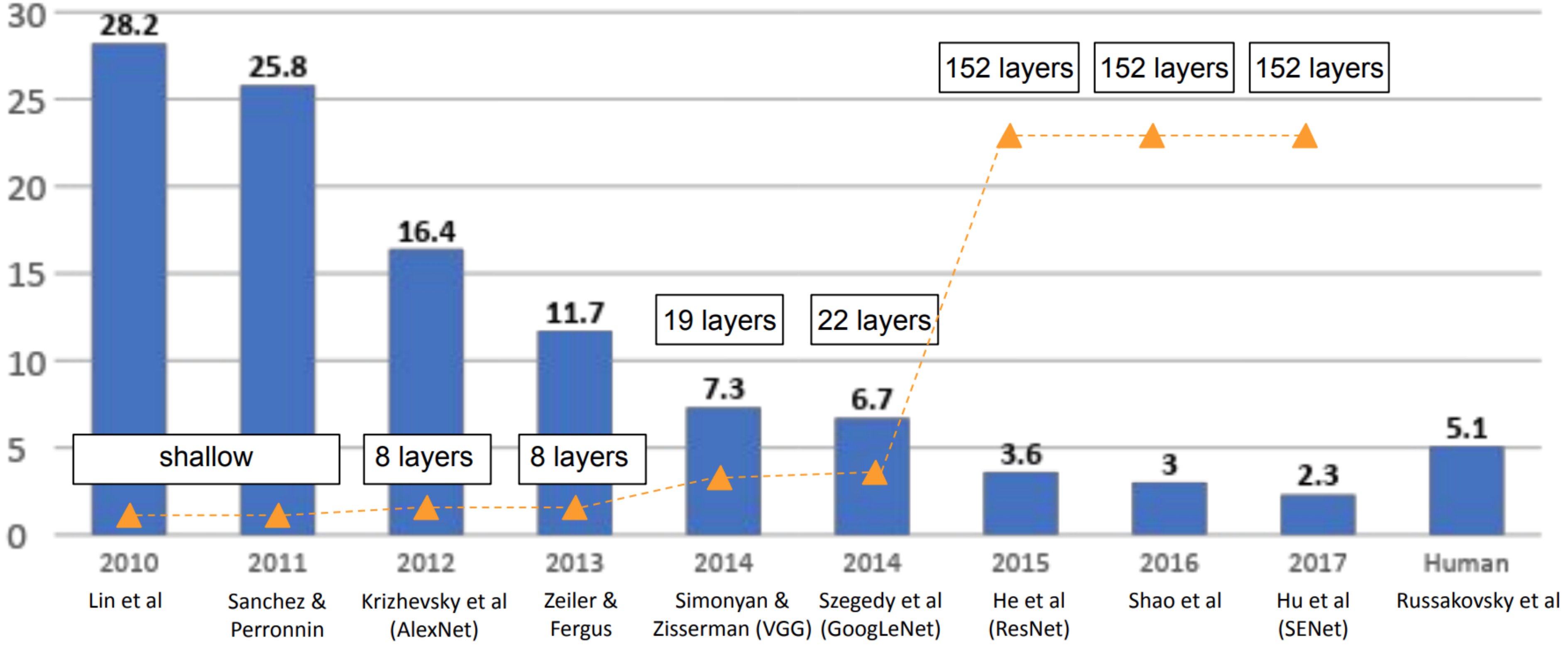
Classification CNN: Backbone Network



CNN Architectures

- AlexNet
- VGG
- GoogLeNet
- ResNet
- …

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



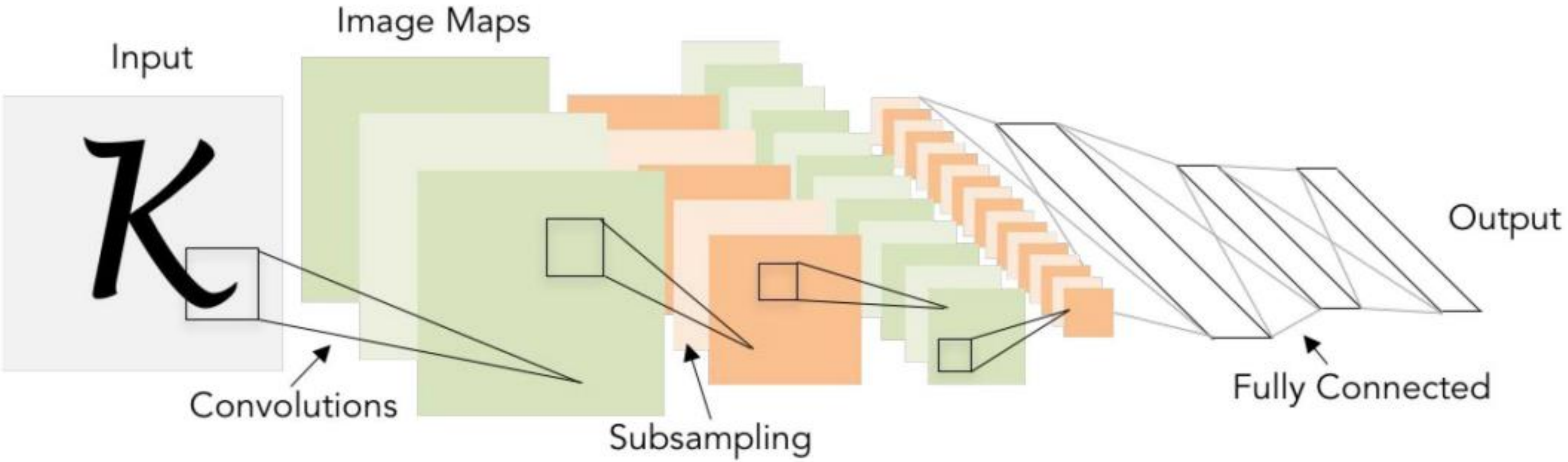
AlexNet (2012)

- Successful CNN image classification model
  - Based on LeNet5 CNN design (Yann Lecun et al, 1989)
  - **Computationally expensive, but feasible due to GPUs**
    - Parallel computation
  - Winner of ILSVRC-2012 competition by a large margin,
    - ImageNet Large-Scale Visual Recognition Challenge
    - a top-5 error of 15.3%, more than 10.8 percentage points lower than that of the runner up.
  - Transfer to significant gains in a variety of domains

LeNet-5 (1998)

- A very simple CNN architecture by Yann LeCun in 1998

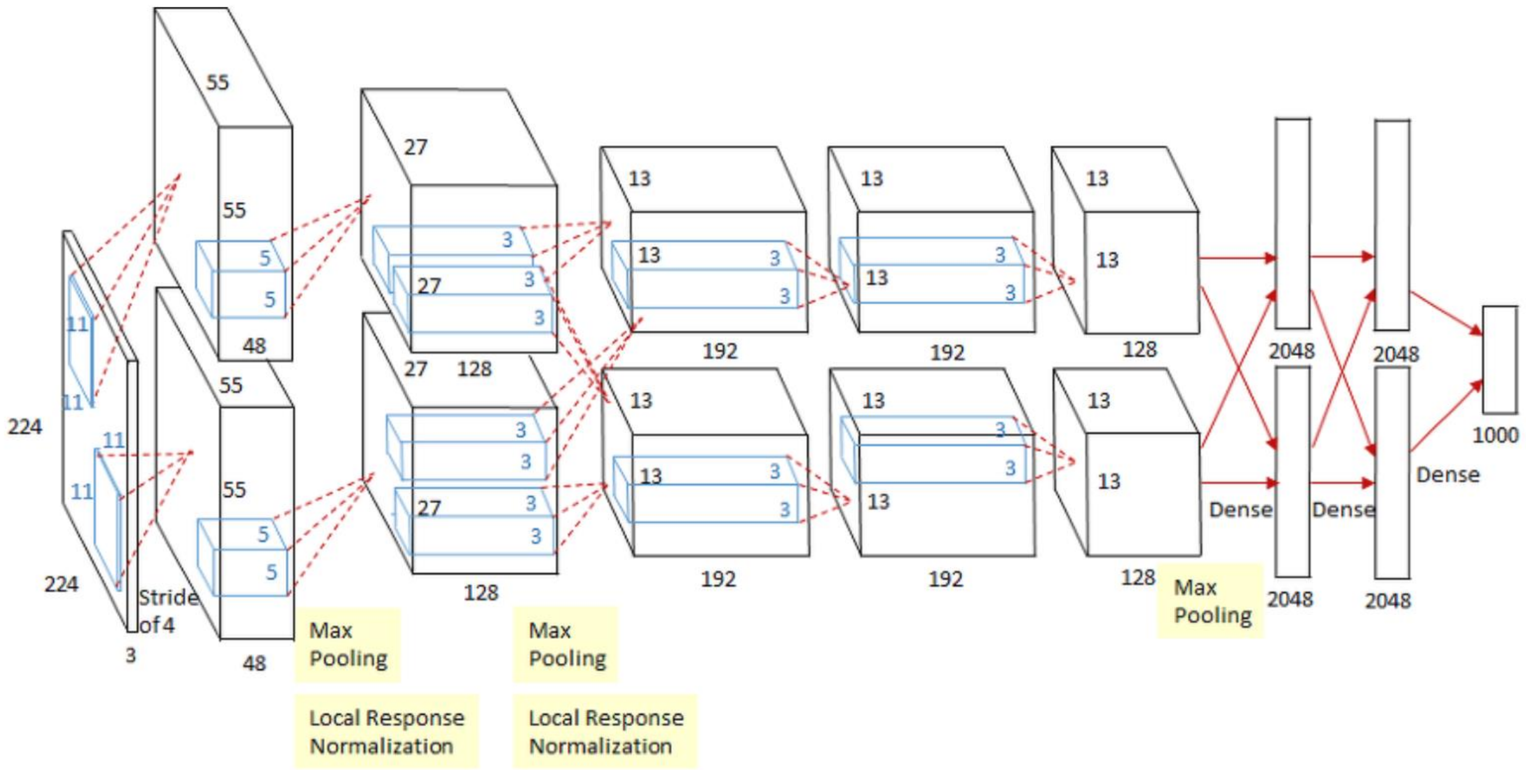
Overall architecture: Conv –Pool – Conv –Pool – FC – FC



[Lecun89] Y. LeCun et al.: **Handwritten Digit Recognition with a Back-Propagation Network**. NIPS 1989

AlexNet

- Architecture
  - Conv1
  - Maxpool1
  - Norm1
  - Conv2
  - Maxpool2
  - Norm2
  - Conv3
  - Conv4
  - Conv5
  - Maxpool3
  - FC6
  - FC7
  - FC8

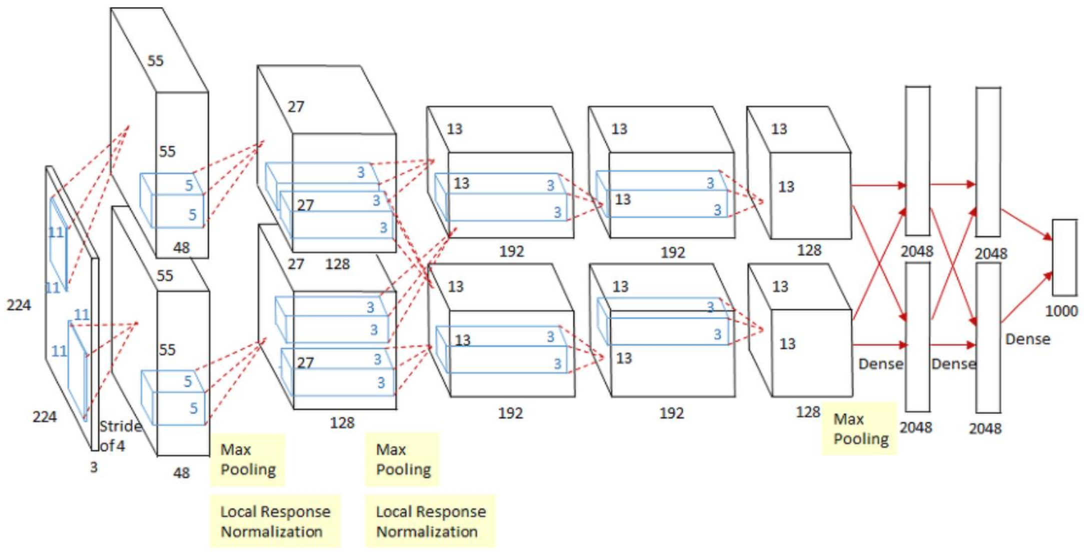


Overall architecture:
Conv – Pool – LRN – Conv – Pool – LRN – Conv – Conv – Conv – Pool – FC – FC – FC

LeNet-5 vs AlexNet

- Compared to LeNet-5, AlexNet is …
  - Bigger (more layers, more parameters)
  - Trained with a larger amount of data (ImageNet)
  - Use different activation function & regularization techniques (dropout)
  - Local Response Normalization
  - Parallel computation via GPUs

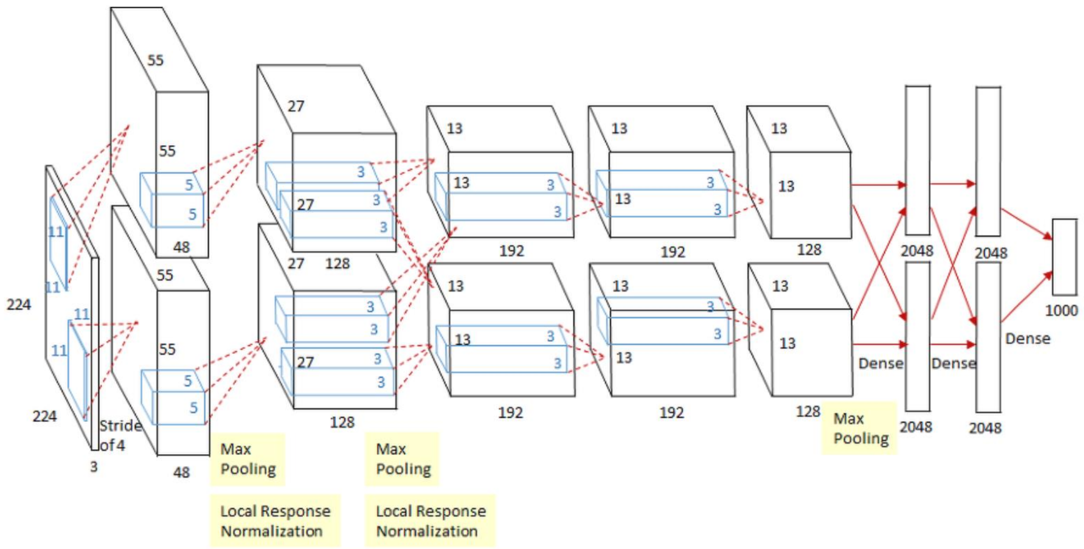
AlexNet



- Input: **227 * 227 * 3** images
- After Conv1 (96 11*11 filters, stride 4): **55 * 55 * 96**
- After Pool1 (3*3 filters, stride 2): **27*27*96**
- …

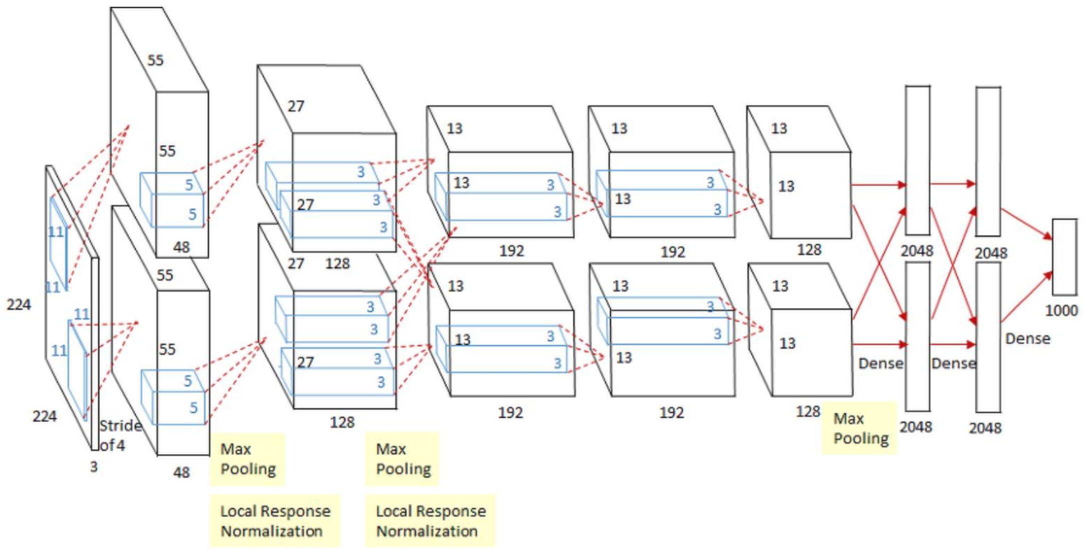
- Output size of convolution = $\frac{input\ size - filter\ size + (2 \times padding)}{stride} + 1$

AlexNet



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

AlexNet

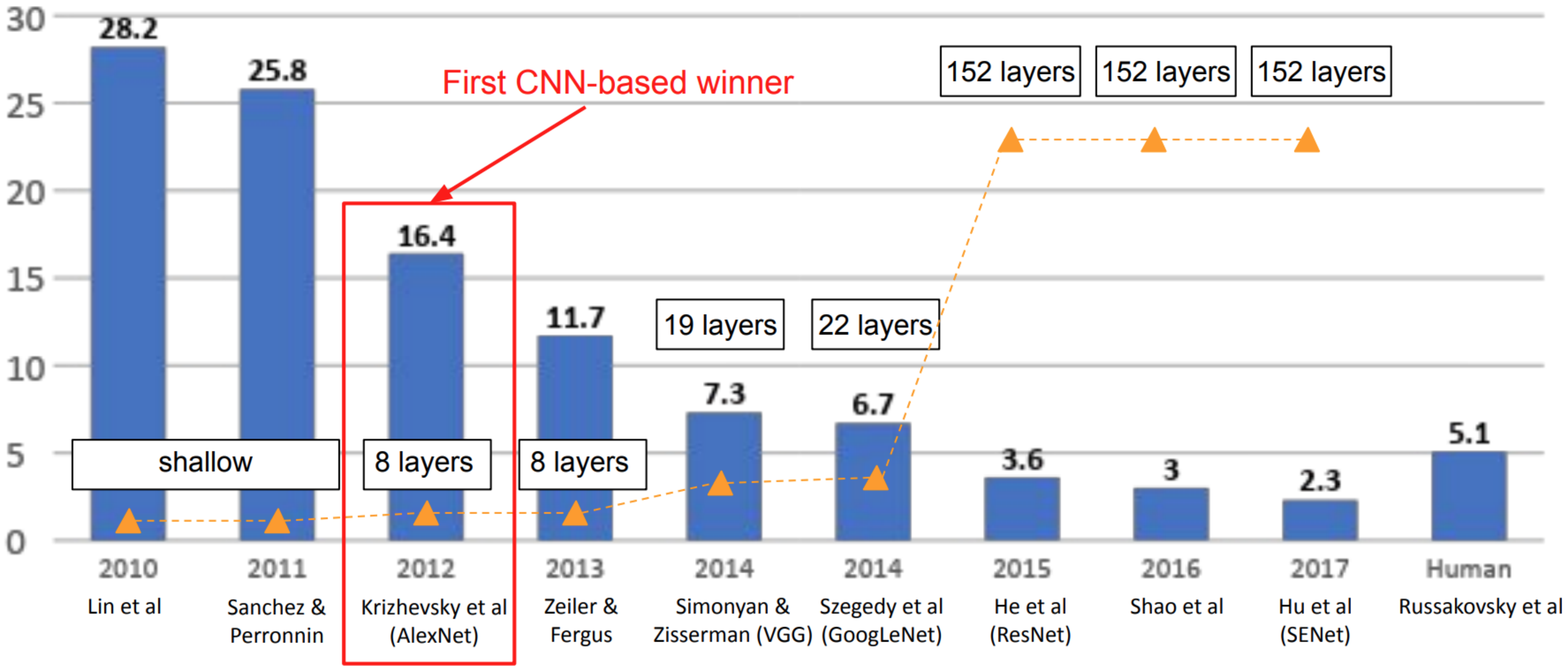


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Other Details:

- First use of ReLU
- Used Local Response Normalization
- Heavy data augmentation
- Dropout 0.5
- Batch size 128
- Learning rate 1e-2, reduced by 10 when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4 %

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



VGG network (2014)

- Proposed by Oxford VGG team in 2014 ILSVRC (2^{nd} rank)
- Comparison
  - More deeper network than AlexNet (Alexnet 8 layers -> 16 – 19 layers)
  - More simple structure than GoogleNet (1^{st} rank)
- Filter
  - Only **3 x 3** filter & 1 x 1 filter
    - Less parameter for same receptive field -> Regularization
- 11.7% top 5 error in ILSVRC’13 -> 7.3% top 5 error in ILSVRC’14

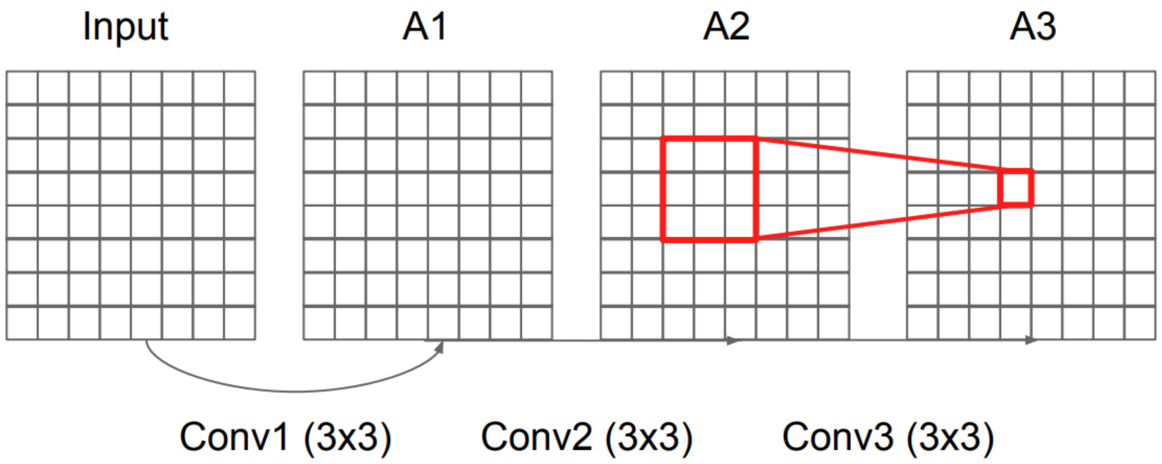
VGG network

- Why use smaller filters? (3*3 conv)
  - Stack of there 3*3 conv (stride 1) layers has the same receptive field as one 7*7 conv layer
  - Receptive field: The region in the input space that a particular CNN’s feature is looking at

VGG network

- Why use smaller filters? (3*3 conv)
  - Stack of there 3*3 conv (stride 1) layers has the same receptive field as one 7*7 conv layer
  - Receptive field: The region in the input space that a particular CNN's feature is looking at

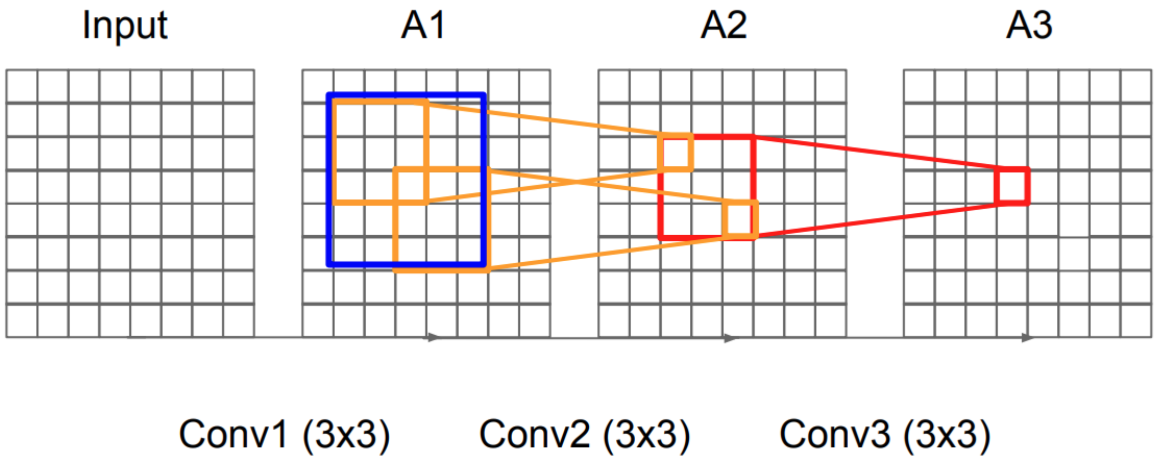
- ex)



VGG network

- Why use smaller filters? (3*3 conv)
  - Stack of there 3*3 conv (stride 1) layers has the same receptive field as one 7*7 conv layer
  - Receptive field: The region in the input space that a particular CNN's feature is looking at

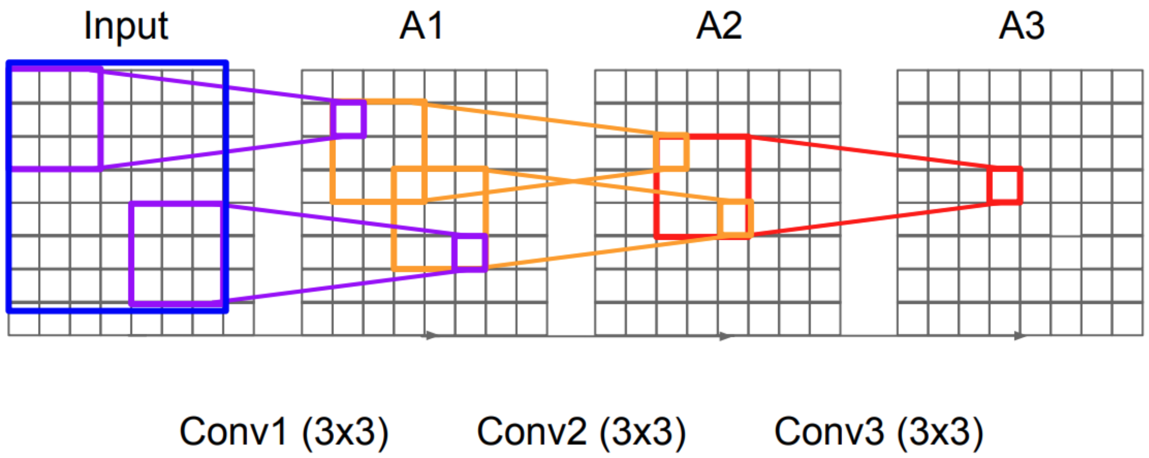
- ex)



VGG network

- Why use smaller filters? (3*3 conv)
  - Stack of there 3*3 conv (stride 1) layers has the same receptive field as one 7*7 conv layer
  - Receptive field: The region in the input space that a particular CNN's feature is looking at

\- ex)



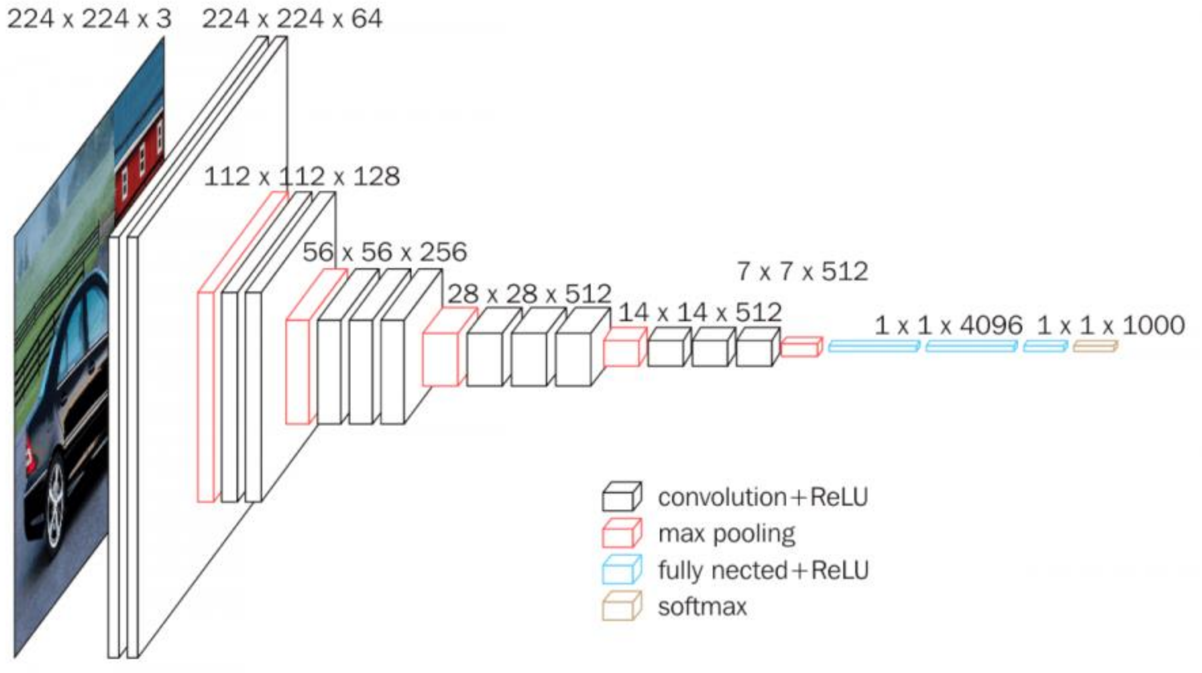
VGG network

- Why use smaller filters? (3*3 conv)
  - Stack of there 3*3 conv (stride 1) layers has the same receptive field as one 7*7 conv layer
  - Receptive field: The region in the input space that a particular CNN's feature is looking at

- But we have deeper, more non-linearities
- And fewer parameters:
  - 3-(3*3) filters = 27
  - 1-(7*7) filter = 49

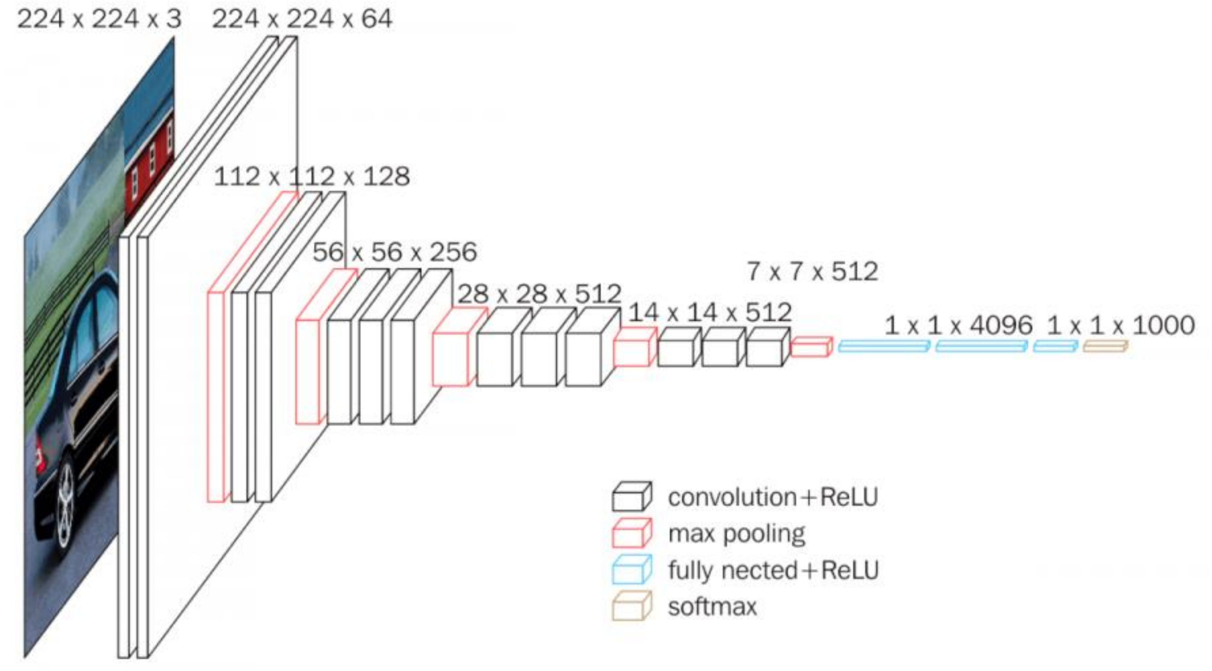
VGG16 network

- Configuration
  - Image size : 224 * 224 * 3
  - All conv layer: Stride 1, Padding 1, filter 3*3 or 1*1
    - Maintain resolution (H * W size) of feature
  - Maxpooling: 2 x 2 window, 2 stride



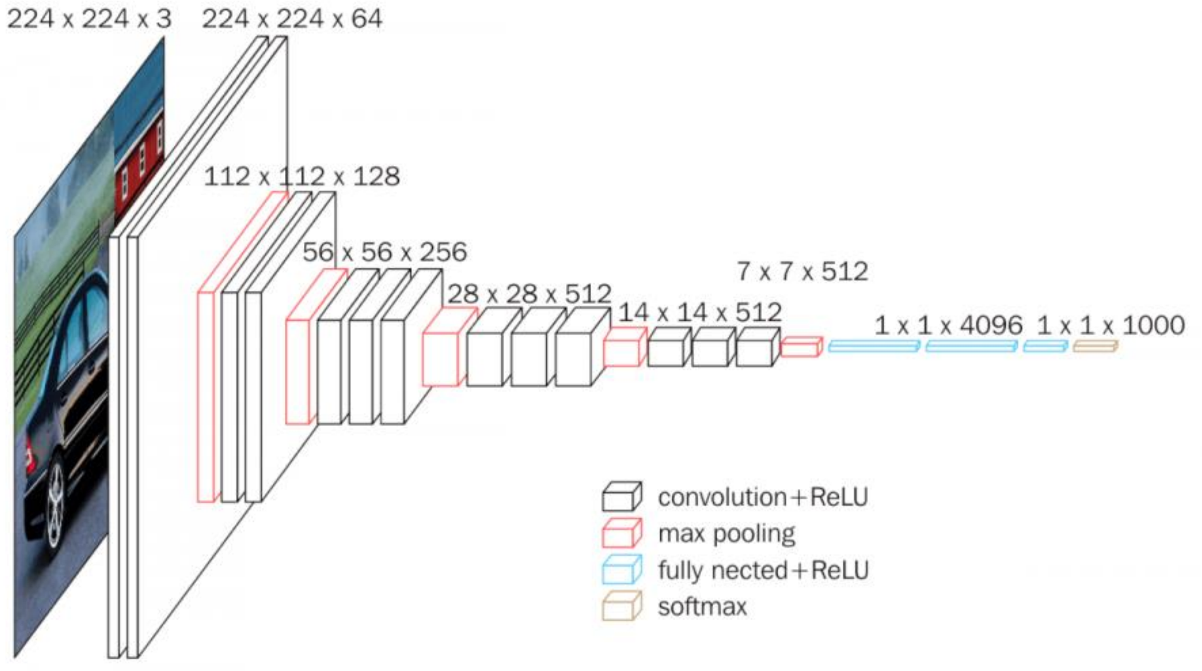
VGG16 network

Input: [224*224*3]
Conv3-64: [224*224*64]
Conv3-64: [224*224*64]
Pool2: [112*112*64]
Conv3-128: [112*112*128]
Conv3-128: [112*112*128]
Pool2: [56*56*128]
Conv3-256: [56*56*256]
Conv3-256: [56*56*256]
Conv3-256: [56*56*256]
Pool2: [28*28*256]
Conv3-512: [28*28*512]
Conv3-512: [28*28*512]
Conv3-512: [28*28*512]
Pool2: [14*14*512]
Conv3-512: [14*14*512]
Conv3-512: [14*14*512]
Conv3-512: [14*14*512]
Pool2: [7*7*512]
FC: [1*1*4096]
FC: [1*1*4096]
FC: [1*1*1000]



VGG16 network

- Other details
  - ILSVRC'14 2nd winner
  - No Local Response Normalization
  - Use VGG16 or VGG19 (VGG19 slightly better)
  - Use ensembles for best results



VGG network

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Pytorch code pipeline

- Define Dataset -> 학습하고 싶은 데이터셋을 정의
- Define DataLoader -> 데이터셋의 배치화
- Define model -> 학습할 모델 정의
- Define loss (criterion) -> loss 계산
- Define optimizer -> gradient descent 수행

-> Start Training and Testing

Pytorch code example

- **Define Dataset →**
- Define DataLoader →
- Define model →
- Define loss (criterion) →
- Define optimizer →
- Training and Testing →

```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms

train_data = MNIST('./data/train', train=True, download=True, transform=transforms.ToTensor())
test_data = MNIST('./data/test', train=False, download=True, transform=transforms.ToTensor())
```

Pytorch code example

- Define Dataset →
- **Define DataLoader →**
- Define model →
- Define loss (criterion) →
- Define optimizer →
- Training and Testing →

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(train_data, batch_size=16, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=16, shuffle=False)
```

Pytorch code example

- Define Dataset →
- Define DataLoader →
- **Define model →**
- Define loss (criterion) →
- Define optimizer →
- Training and Testing →

```python
from torch import nn

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_layer = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_layer(x)
        return logits

model = Model()
```

Pytorch code example

- Define Dataset →
- Define DataLoader →
- Define model →
- **Define loss (criterion) →**
- **Define optimizer →**
- Training and Testing →

```
import torch.optim as optim

optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=0.001)
loss_function = nn.MSELoss()
```

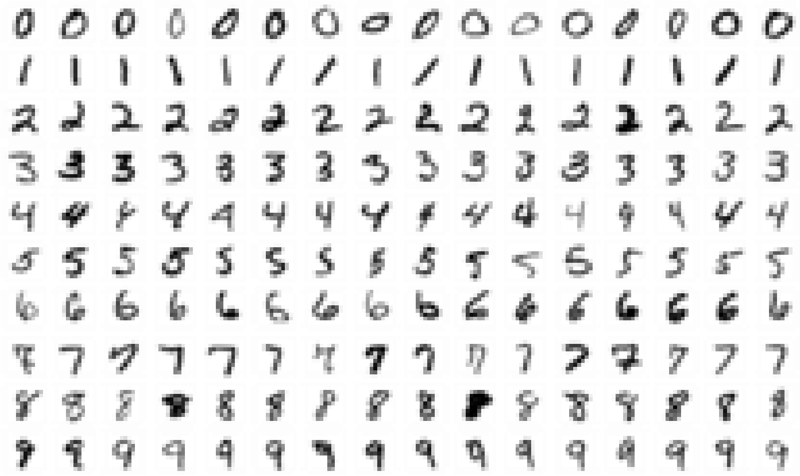
Pytorch code example

- Define Dataset →
- Define DataLoader →
- Define model →
- Define loss (criterion) →
- Define optimizer →
- **Training and Testing →**

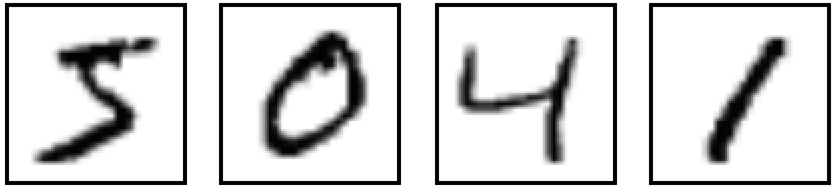
```
model.train()
for x, y in train_dataloader:
    prediction = model(x)
    loss = loss_function(prediction, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
model.eval()
for x, y in test_dataloader:
    prediction = nn.Softmax(dim=1)(model(x))
    y_pred = prediction.argmax(1)
    print("Predicted class: {y_pred}, True label: {y}")
```

Ex) MNIST dataset



Image

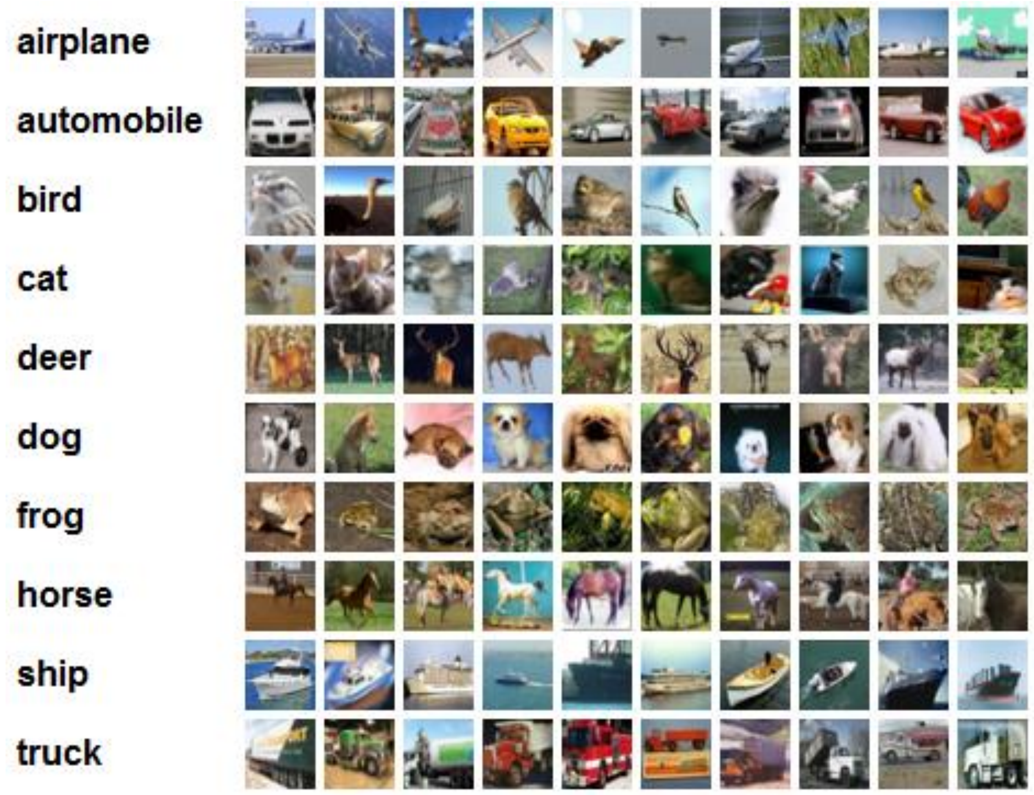


Label

5 0 4 1

- **Hand-written number dataset**
- **Total 10 class (0 ~ 9)**
- **28X28 size, grayscale**
- **Training sample : 60,000**
- **Test sample : 10,000**

Ex) CIFAR-10 dataset



- **10-class dataset**
- **32 x 32 size, RGB image**
- **Training sample : 50,000**
- **Test sample : 10,000**

Softmax function

- Function that takes as input a vector of K real numbers, and normalizes it into a probability distribution

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

- Also known as, normalized exponential function

Cross entropy loss

- To measure the distance between probability p and q

$$H(P,Q) = -\sum_x P(x)\log Q(x)$$

- Ex) Given that ground truth label P(x) = (1, 0)

  if Q(x) = (0, 1), $-P(x)\log Q(x) = -\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}\log 0 \\ \log 1\end{bmatrix} = -(-\infty + 0) = \infty$

  if Q(x) = (1, 0), $-P(x)\log Q(x) = -\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}\log 1 \\ \log 0\end{bmatrix} = -(0 + 0) = 0$

Datasets and DataLoader

- All datasets are subclass of torch.utils.data.Dataset
  - You can make your own datasets to inherit Dataset class
  - Some famous datasets is provided in torchvision.datasets
    - CIFAR-10, MNIST, etc
- DataLoader reads datasets and make the batch

torch.utils.data.DataLoader

- It reads datasets and make batch for training and inference
- Arguments
  - dataset – target dataset (torch.utils.data.Dataset)
  - batch_size – batch size for training or inference
  - shuffle – if true, dataloader shuffle the data in datasets randomly
  - sampler, batch_sampler, num_workers, pin_memorys
- for batch_idx, (data, label) in enumerate(trainloader)

Exercise 1.
Training a classifier on CIFAR-10

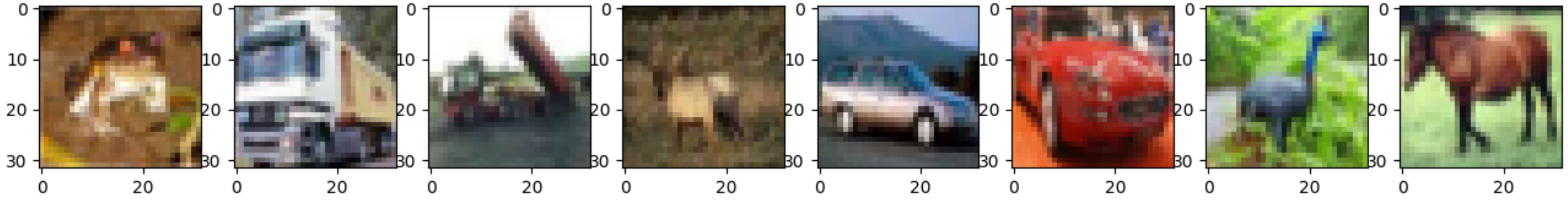
- Step)
  - Load and normalizing the CIFAR 10 training and test datasets using torchvision
    - Visualize first 8 images from the train dataset
  - **1. Define a convolutional Neural Network**
  - **2. Define a loss function and optimizer (CrossEntropy, SGD)**
  - **3. Train the network on the training data**
  - Test the network on the test data

Visualize first 8 images from the train dataset

```
import matplotlib.pyplot as plt
from torchvision.transforms.functional import to_pil_image
```

import library for plotting/visualizing data

```
plt.figure(figsize=(16,4))
for i in range(8):
    img = to_pil_image(train_data[i][0]/2 + 0.5)
    plt.subplot(1,8,i+1)
    plt.imshow(img)
plt.show()

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
print("Labels: ", end='')
for i in range(8): print(classes[train_data[i][1]], end=' | ')
```

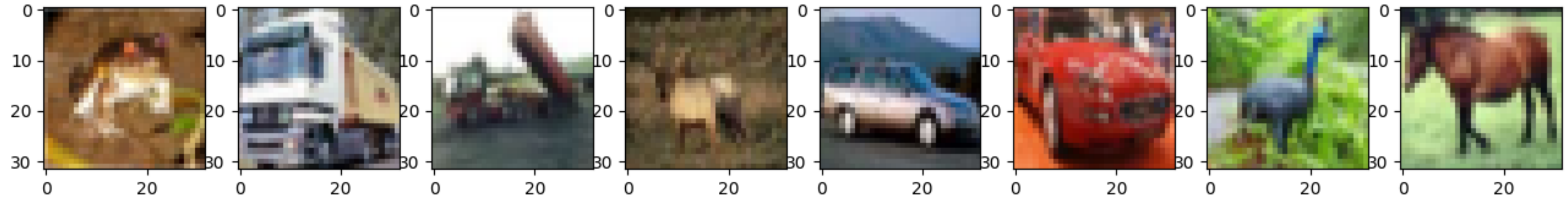



```
Labels: frog | truck | truck | deer | car | car | bird | horse |
```

Visualize first 8 images from the train dataset

```
import matplotlib.pyplot as plt
from torchvision.transforms.functional import to_pil_image

plt.figure(figsize=(16,4))
for i in range(8):
    img = to_pil_image(train_data[i][0]/2 + 0.5)
    plt.subplot(1,8,i+1)
    plt.imshow(img)
plt.show()

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
print("Labels: ", end='')
for i in range(8): print(classes[train_data[i][1]], end=' | ')
```

Define figure and plot images



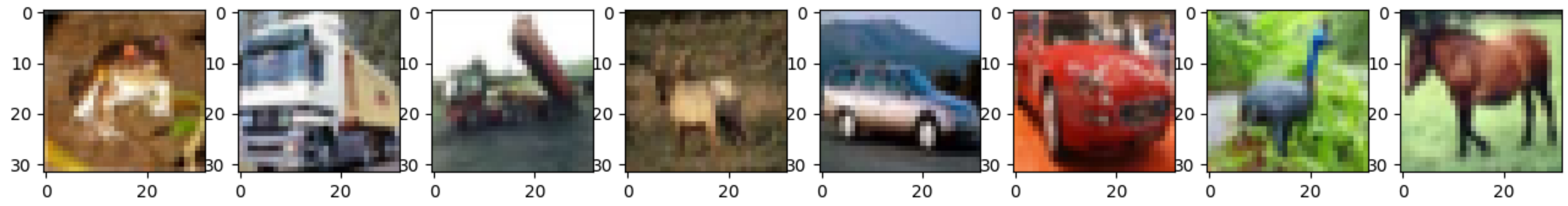

```
Labels: frog | truck | truck | deer | car | car | bird | horse |
```

Visualize first 8 images from the train dataset

```
import matplotlib.pyplot as plt
from torchvision.transforms.functional import to_pil_image

plt.figure(figsize=(16,4))
for i in range(8):
    img = to_pil_image(train_data[i][0]/2 + 0.5)
    plt.subplot(1,8,i+1)
    plt.imshow(img)
plt.show()
```

```
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
print("Labels: ", end='')
for i in range(8): print(classes[train_data[i][1]], end=' | ')
```

Print labels corresponding to the images




```
Labels: frog | truck | truck | deer | car | car | bird | horse |
```

- Load and normalizing the CIFAR 10 training and test datasets using torchvision

```
from torchvision.datasets import CIFAR10
import torchvision.transforms as transforms

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

train_data = CIFAR10("./data", train=True, download=True, transform=transform)
test_data = CIFAR10("./data", train=False, download=True, transform=transform)

print(len(train_data))
print(len(test_data))
```

```
Files already downloaded and verified
Files already downloaded and verified
50000
10000
```

- Load and normalizing the CIFAR 10 training and test datasets using torchvision

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(train_data, batch_size=8, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=8, shuffle=False)
```

1. Define a convolutional Neural Network

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 8, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2,2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(8, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2,2)
        )

        self.fc_layer = nn.Linear(16 * 8 * 8, 10)

    def forward(self, x):
        x1 = self.layer1(x)
        x2 = self.layer2(x1)

        x2 = torch.flatten(x2, 1)
        x3 = self.fc_layer(x2)
        out = F.softmax(x3, dim=1)
        return out


net = CNN()
```

```python
if torch.cuda.is_available(): device = torch.device('cuda')
else: device = torch.device('cpu')

net = net.to(device)
```

✓ 0.3s

2. Define a loss function and optimizer (CrossEntropy, SGD)

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

3. Train the network on the training data

```python
for epoch in range(4):   # 데이터셋을 수차례 반복합니다.

    running_loss = 0.0
    net.train()
    for i, data in enumerate(train_dataloader, 0):
        # [inputs, labels]의 목록인 data로부터 입력을 받은 후;
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # 변화도(Gradient) 매개변수를 0으로 만들고
        optimizer.zero_grad()

        # 순전파 + 역전파 + 최적화를 한 후
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # 통계를 출력합니다.
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}',end=' | ')
            running_loss = 0.0
```

Test the network on the test data

```python
    correct = 0
    total = 0
    net.eval()
    # 학습 중이 아니므로, 출력에 대한 변화도를 계산할 필요가 없습니다
    with torch.no_grad():
        for data in test_dataloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)

            # 신경망에 이미지를 통과시켜 출력을 계산합니다
            outputs = net(images)
            # 가장 높은 값(energy)를 갖는 분류(class)를 정답으로 선택하겠습니다
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Accuracy on the 10000 test images: {100 * correct // total} %')

print('Finished Training')
```

```
[1,  2000] loss: 2.279 | [1,  4000] loss: 2.187 | [1,  6000] loss: 2.149 | Accuracy on the 10000 test images: 33 %
[2,  2000] loss: 2.122 | [2,  4000] loss: 2.101 | [2,  6000] loss: 2.072 | Accuracy on the 10000 test images: 40 %
[3,  2000] loss: 2.050 | [3,  4000] loss: 2.036 | [3,  6000] loss: 2.028 | Accuracy on the 10000 test images: 44 %
[4,  2000] loss: 2.014 | [4,  4000] loss: 1.999 | [4,  6000] loss: 2.002 | Accuracy on the 10000 test images: 47 %
Finished Training
```

Exercise 2.
Train VGG (A) Network on CIFAR-10

Step:

1. Implement VGG11 network

2. Change the fully connected part (**use only one fc**)

- Conv – input channels: 3, output channels: 64
- Maxpooling
- Conv – input channels: 64, output channels: 128
- Maxpooling
- Conv – input channels: 128, output channels: 256
- Conv – input channels: 256, output channels: 256
- Maxpooling
- Conv – input channels: 256, output channels: 512
- Conv – input channels: 512, output channels: 512
- Maxpooling
- Conv – input channels: 512, output channels: 512
- Conv – input channels: 512, output channels: 512
- Maxpooling

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Every conv layer: kernel size: 3, padding: 1
Every max pooling layer: kernel size 2, stride 2

Exercise 2-2.
Train VGG (D) Network on CIFAR-10

Step:

1. Implement VGG16 network
2. Change the fully connected part (**use three fc layers**)
3. Apply dropout of 0.5 to fc layers

- Conv – input channels: 3, output channels: 64
- Conv – input channels: 64, output channels: 64
- Maxpooling
- Conv – input channels: 64, output channels: 128
- Conv – input channels: 128, output channels: 128
- Maxpooling
- Conv – input channels: 128, output channels: 256
- Conv – input channels: 256, output channels: 256
- Conv – input channels: 256, output channels: 256
- Maxpooling
- Conv – input channels: 256, output channels: 512
- Conv – input channels: 512, output channels: 512
- Conv – input channels: 512, output channels: 512
- Maxpooling
- Conv – input channels: 512, output channels: 512
- Conv – input channels: 512 ,output channels: 512
- Conv – input channels: 512 ,output channels: 512
- Maxpooling

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weigh layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-12 conv3-12	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-25 conv3-25 **conv1-25**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-51 conv3-51 **conv1-51**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-51 conv3-51 **conv1-51**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Every conv layer: kernel size: 3, padding: 1
Every max pooling layer: kernel size 2, stride 2