Build a CNN from scratch

Practice – CNN

- Run “7.2. MyCNN.ipynb"

Build my own CNN model

Suppose input image size = 64 x 64x 3

```
class MyCNN(nn.Module):
  def __init__(self):
    super(MyCNN, self).__init__()
    self.features = nn.Sequential(    #Assume input image H/W=64
        nn.Conv2d(3, 32, 3, 1, 1), #feature map H/W=(64+2*1-3)/1+1 = 64
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2, 2, 0),     #H/W=(64+2*0-2)/2+1 = 32
        nn.Conv2d(32, 8, 3, 1, 1), #H/W=(32+2*1-3)/1+1 = 32
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2, 2, 0),     #H/W=(32+2*0-2)/2+1 = 16
    )
    self.classifier = nn.Sequential(
        nn.Dropout(),
        nn.Linear(8 * 16 * 16, 500),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(500, 100),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(100, 2),
    )

  def forward(self, x):
    x = self.features(x)
    x = torch.flatten(x, 1)
    x = self.classifier(x)
    return x
```

The MLP used in "4.2. Classification with CE loss"

```
MyNet = nn.Sequential(
    nn.Linear(2, 50),
    nn.ReLU(),
    nn.Linear(50, 100),
    nn.ReLU(),
    nn.Linear(100, 50),
    nn.ReLU(),
    nn.Linear(50, 2),
)
MyNet.to(device)
```

Practice: Draw the structure of MyCNN

```
model = MyCNN().to(device)
print(model)
```

Suppose input image size = 64 x 64x 3

```
MyCNN(
  (features): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
    (3): Conv2d(32, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=2048, out_features=500, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=500, out_features=100, bias=True)
    (5): ReLU(inplace=True)
    (6): Dropout(p=0.5, inplace=False)
    (7): Linear(in_features=100, out_features=2, bias=True)
  )
)
```

My own CNN

```
from torchsummary import summary
summary(model, input_size=(3, 64, 64))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 64, 64]             896
              ReLU-2           [-1, 32, 64, 64]               0
         MaxPool2d-3           [-1, 32, 32, 32]               0
            Conv2d-4            [-1, 8, 32, 32]           2,312
              ReLU-5            [-1, 8, 32, 32]               0
         MaxPool2d-6            [-1, 8, 16, 16]               0
           Dropout-7                 [-1, 2048]               0
            Linear-8                  [-1, 500]       1,024,500
              ReLU-9                  [-1, 500]               0
          Dropout-10                  [-1, 500]               0
           Linear-11                  [-1, 100]          50,100
             ReLU-12                  [-1, 100]               0
          Dropout-13                  [-1, 100]               0
           Linear-14                    [-1, 2]             202
================================================================
Total params: 1,078,010
Trainable params: 1,078,010
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 2.42
Params size (MB): 4.11
Estimated Total Size (MB): 6.58
----------------------------------------------------------------
```

MLP in "4.2. Classification with CE loss"

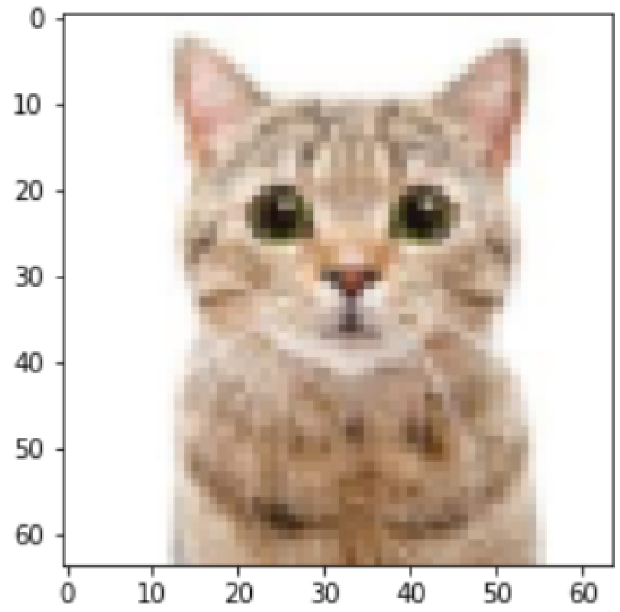
```
BATCH_SIZE = 30
summary(MyNet, input_size=(BATCH_SIZE, 2))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 30, 50]             150
              ReLU-2               [-1, 30, 50]               0
            Linear-3              [-1, 30, 100]           5,100
              ReLU-4              [-1, 30, 100]               0
            Linear-5               [-1, 30, 50]           5,050
              ReLU-6               [-1, 30, 50]               0
            Linear-7                [-1, 30, 2]             102
================================================================
Total params: 10,402
Trainable params: 10,402
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.09
Params size (MB): 0.04
Estimated Total Size (MB): 0.13
----------------------------------------------------------------
```

Input image after pre-processing

In [13]:
```
#visualize the image after pre-processing
# Tensor is channel first, to plot, we need to convert to channel last
import numpy as np
PILImgArray = np.zeros((PILImg.shape[1], PILImg.shape[2], 3))
PILImgArray[:,:,0] = PILImg[0, :, :]
PILImgArray[:,:,1] = PILImg[1, :, :]
PILImgArray[:,:,2] = PILImg[2, :, :]
PILImgArray = PILImgArray*0.5+0.5  # change N(0, 1) to [0, 1]
print(PILImgArray.shape, PILImgArray.min(), PILImgArray.max())
plt.imshow(PILImgArray)
plt.show()
```

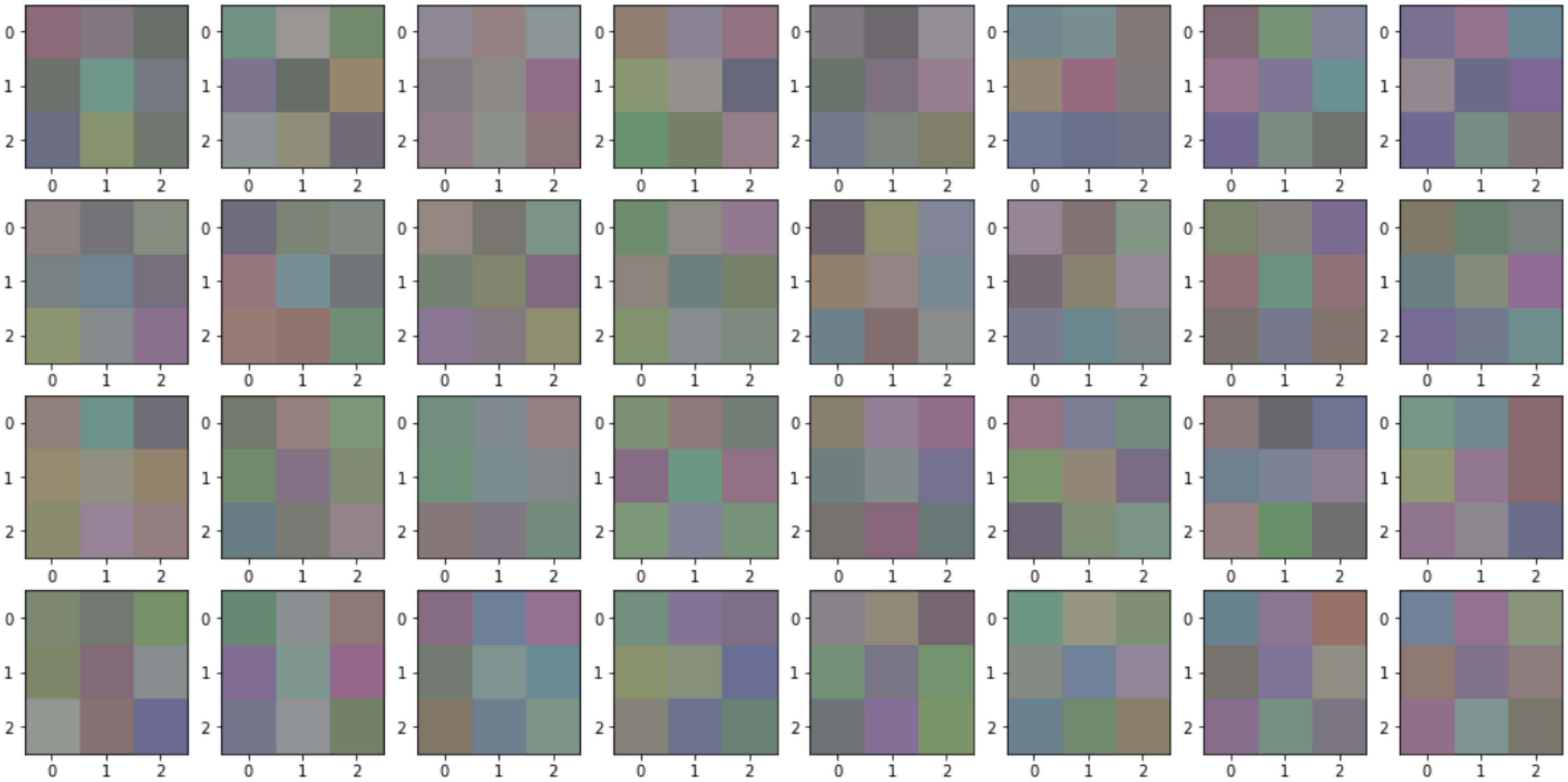
```
(64, 64, 3) 0.027450978755950928 1.0
```



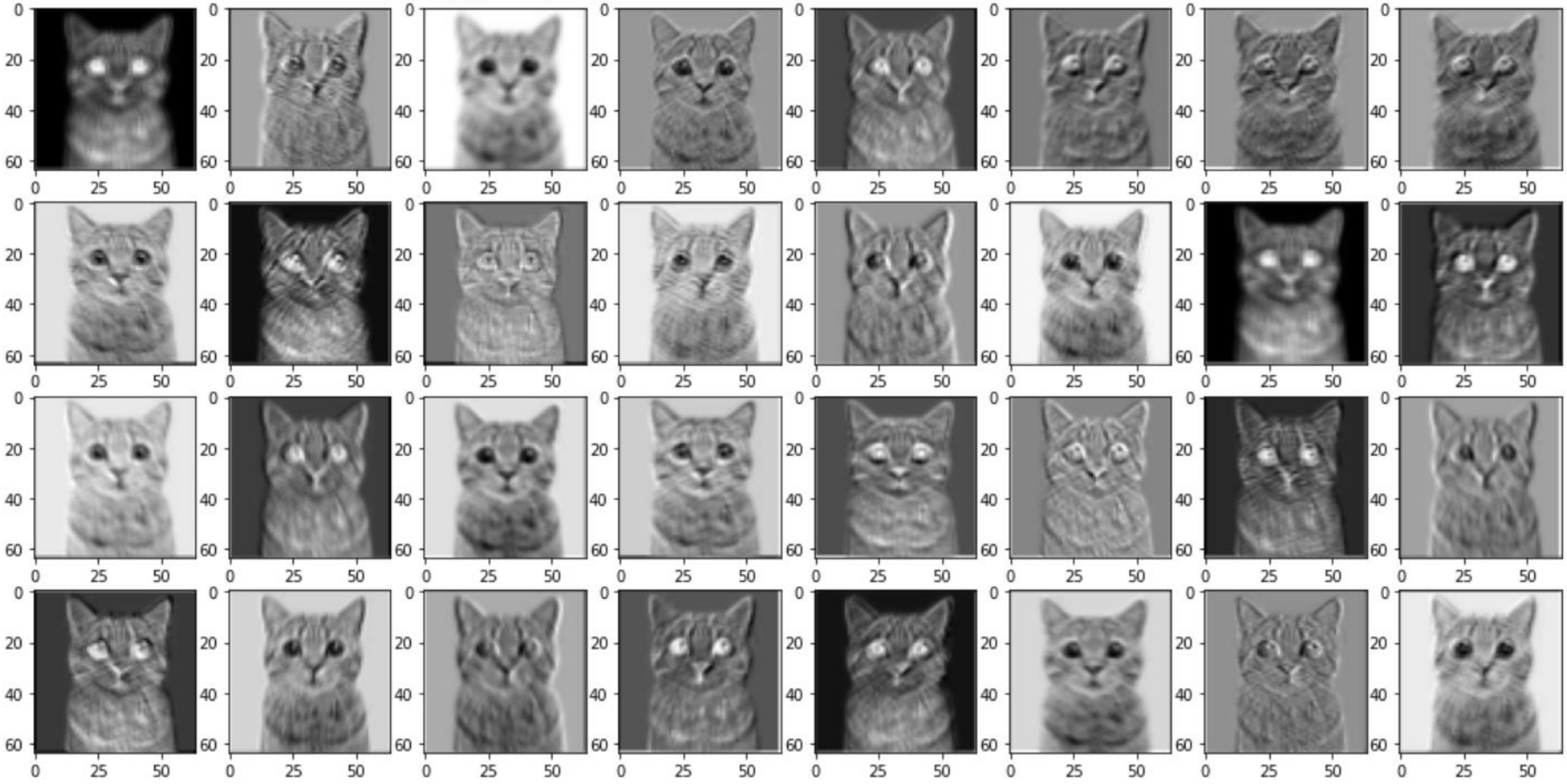
Input image size = 64 x 64x 3

Initial filter weights

```
MyCNN(
  (features): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1
    (3): Conv2d(32, 8, kernel_size=(3, 3), stride=(1, 1), padding
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1
```

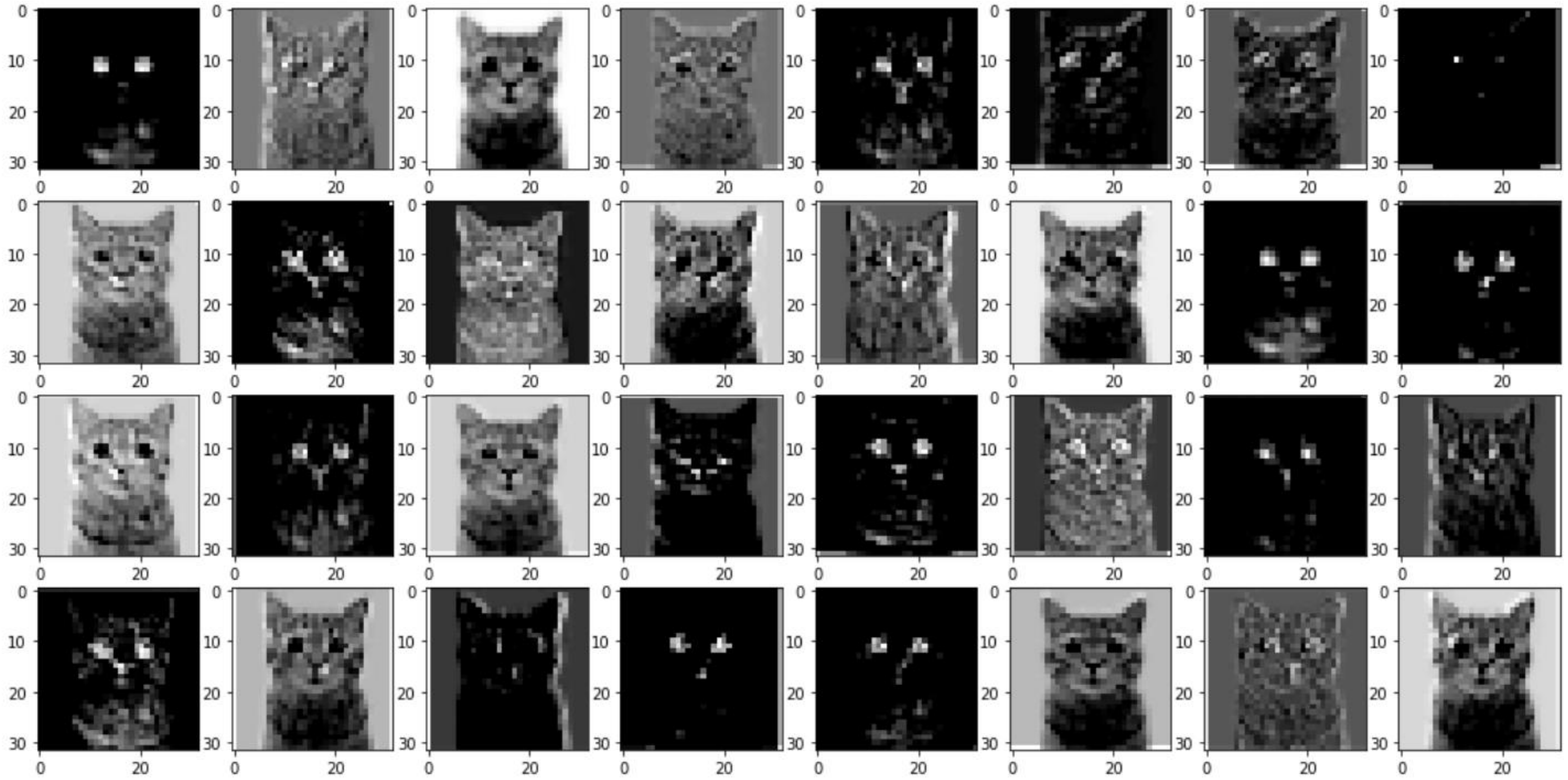


Output feature map, shape = 64x64x32



Feature map after max pooling, shape = 32x32x32

```
torch.Size([1, 32, 32, 32])
```



Flatten

```
class MyCNN(nn.Module):
  def __init__(self):
    super(MyCNN, self).__init__()
    self.features = nn.Sequential(    #Assume input image H/W=64
        nn.Conv2d(3, 32, 3, 1, 1), #feature map H/W=(64+2*1-3)/1+1 = 64
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2, 2, 0),     #H/W=(64+2*0-2)/2+1 = 32
        nn.Conv2d(32, 8, 3, 1, 1), #H/W=(32+2*1-3)/1+1 = 32
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2, 2, 0),     #H/W=(32+2*0-2)/2+1 = 16
    )
    self.classifier = nn.Sequential(
        nn.Dropout(),
        nn.Linear(8 * 16 * 16, 500),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(500, 100),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(100, 2),
    )

  def forward(self, x):
    x = self.features(x)
    x = torch.flatten(x, 1)
    x = self.classifier(x)
    return x
```

```
model = MyCNN().to(device)
print(model)
```

```
MyCNN(
  (features): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
    (3): Conv2d(32, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=2048, out_features=500, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=500, out_features=100, bias=True)
    (5): ReLU(inplace=True)
    (6): Dropout(p=0.5, inplace=False)
    (7): Linear(in_features=100, out_features=2, bias=True)
  )
)
```

In [22]:
```
WholeConvLayers = model.features
out1 = WholeConvLayers(imageTensor.to(device))
print(out1.shape)
```

```
torch.Size([1, 8, 16, 16])
```

In [23]:
```
out2 = torch.flatten(out1, 1)
print(out2.shape)
```

```
torch.Size([1, 2048])
```

In [24]:
```
ClassifierMLP = model.classifier
out = ClassifierMLP(out2)
```

HW5 (1)

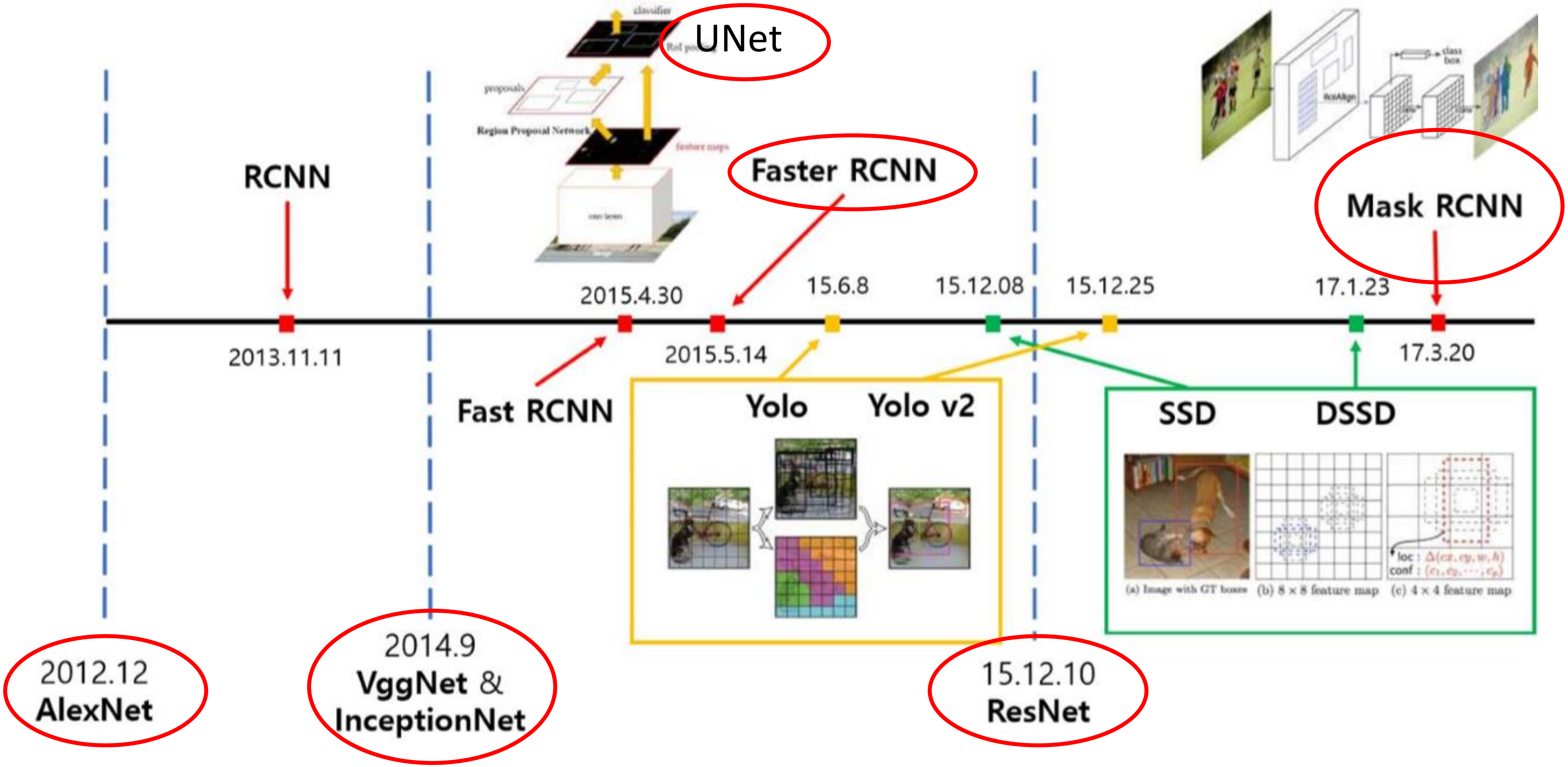
- Let the input image size be 224x224x3. Modify your CNN.

```python
class MyCNN(nn.Module):
  def __init__(self):
    super(MyCNN, self).__init__()
    self.features = nn.Sequential(    #Assume input image H/W=64
        nn.Conv2d(3, 32, 3, 1, 1), #feature map H/W=(64+2*1-3)/1+1 = 64
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2, 2, 0),    #H/W=(64+2*0-2)/2+1 = 32
        nn.Conv2d(32, 8, 3, 1, 1), #H/W=(32+2*1-3)/1+1 = 32
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2, 2, 0),    #H/W=(32+2*0-2)/2+1 = 16
    )
    self.classifier = nn.Sequential(
        nn.Dropout(),
        nn.Linear(8 * 16 * 16, 500),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(500, 100),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(100, 2),
    )

  def forward(self, x):
    x = self.features(x)
    x = torch.flatten(x, 1)
    x = self.classifier(x)
    return x
```

```python
[10]: from torchvision import transforms
transformer = transforms.Compose([
      transforms.Resize(64),
      transforms.CenterCrop(64),
      transforms.ToTensor(),
      transforms.Normalize(mean=[0.5, 0.5
```

VGG16

CNN family



圖來源: 李春煌 FasterRCNN講義 https://youtu.be/2i9CcmJp2yI

Practice – Load ImageNet pre-trained VGG

```
import torchvision
model = torchvision.models.vgg16(pretrained=True)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth"
100% ██████████ 528M/528M [00:10<00:00, 54.9MB/s]
```

Practice: Draw the structure of VGG16

```
model.eval()
model.to(device)
```

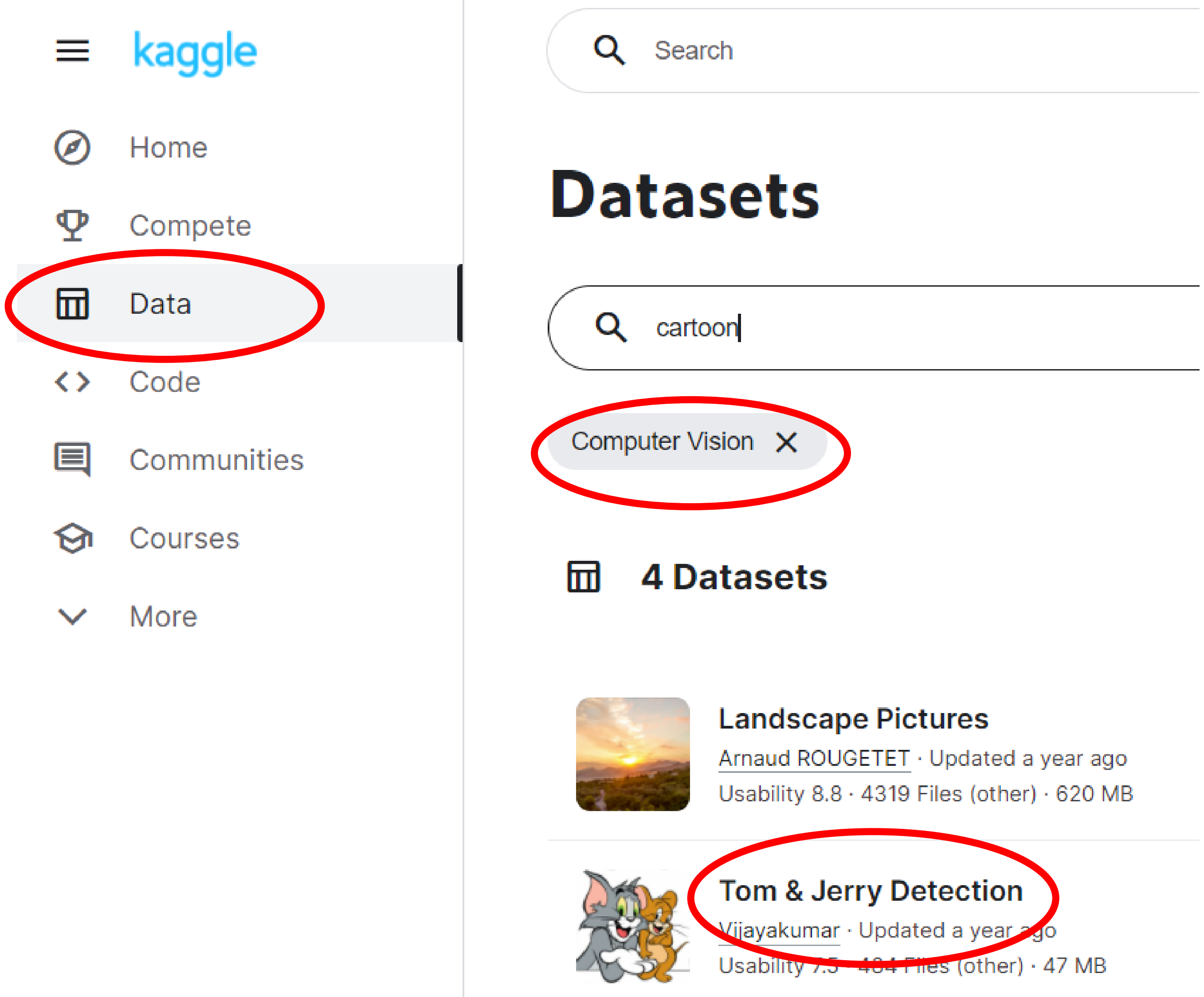
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
```

Practice: Draw the structure of VGG16

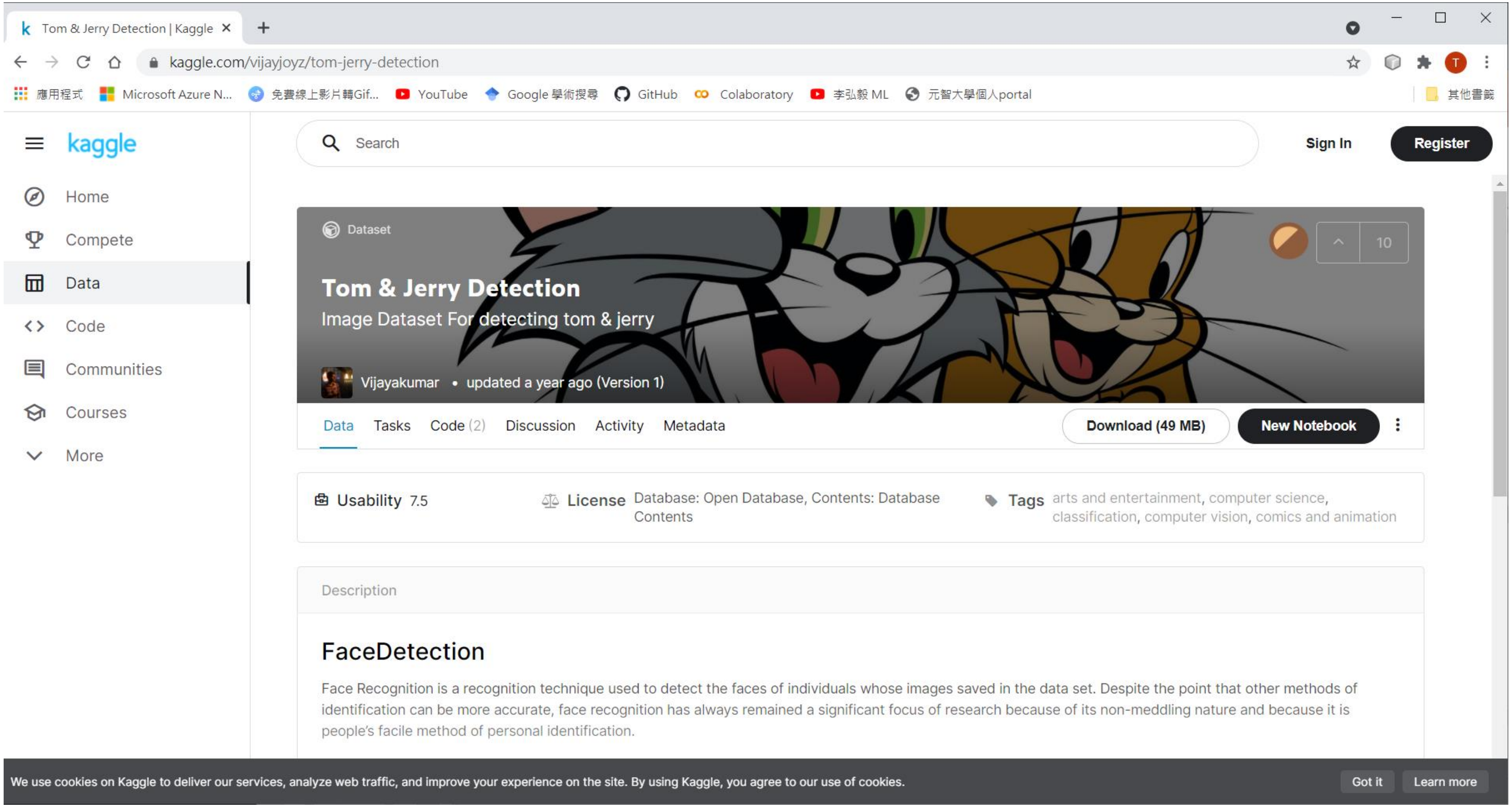
```
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Transfer learning

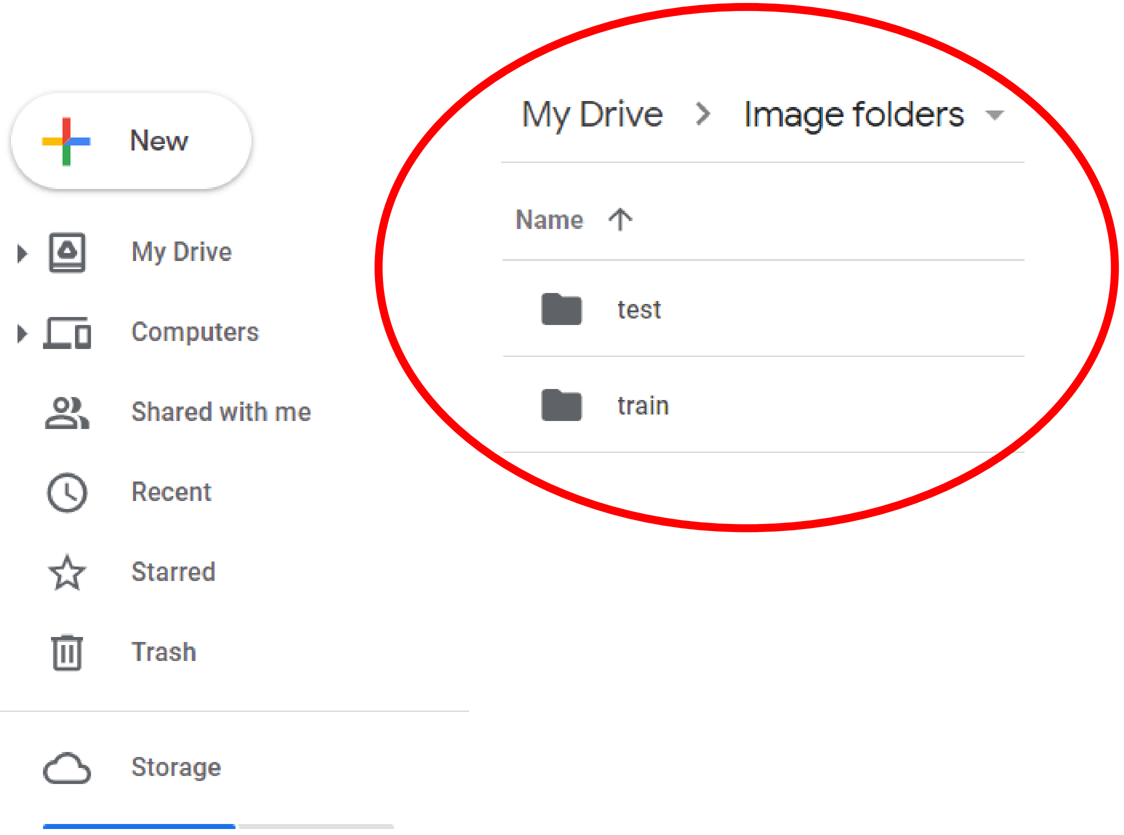
Download images from Kaggle

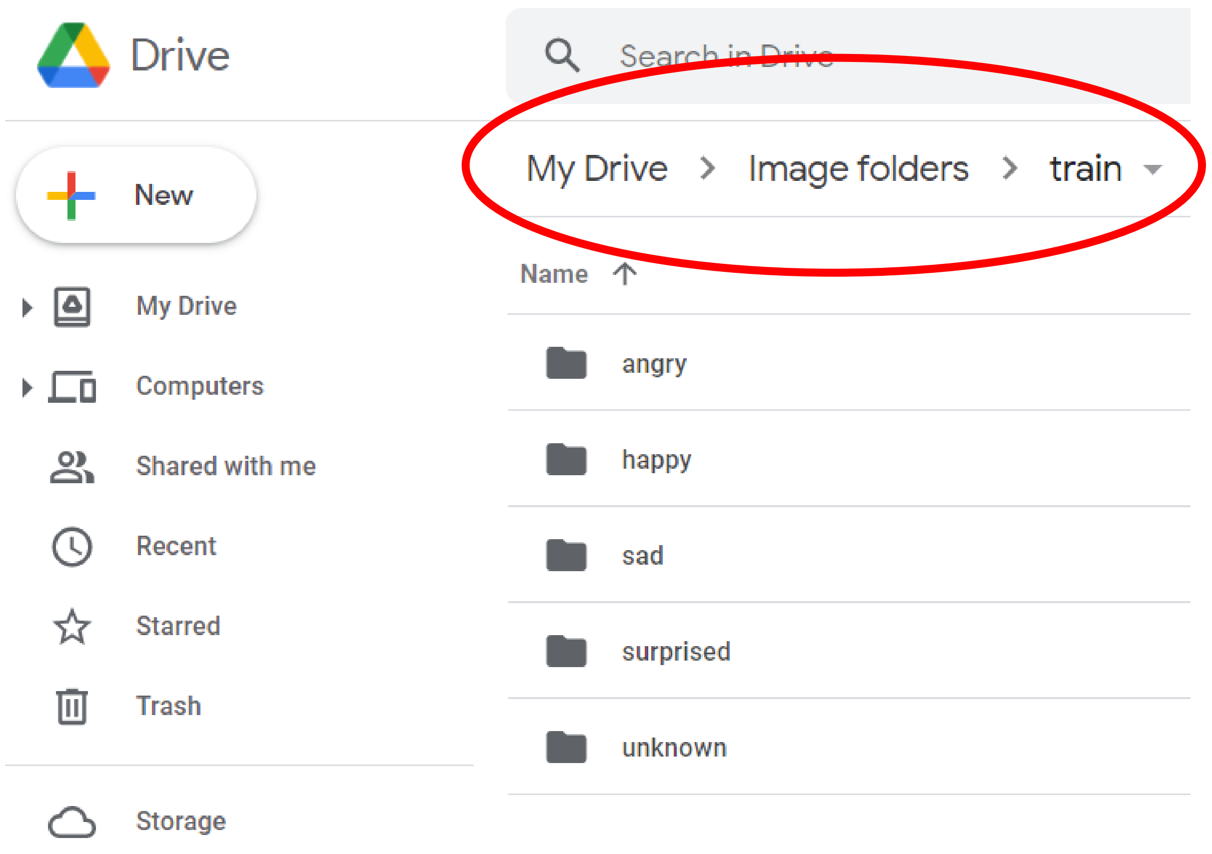


Tom & Jerry

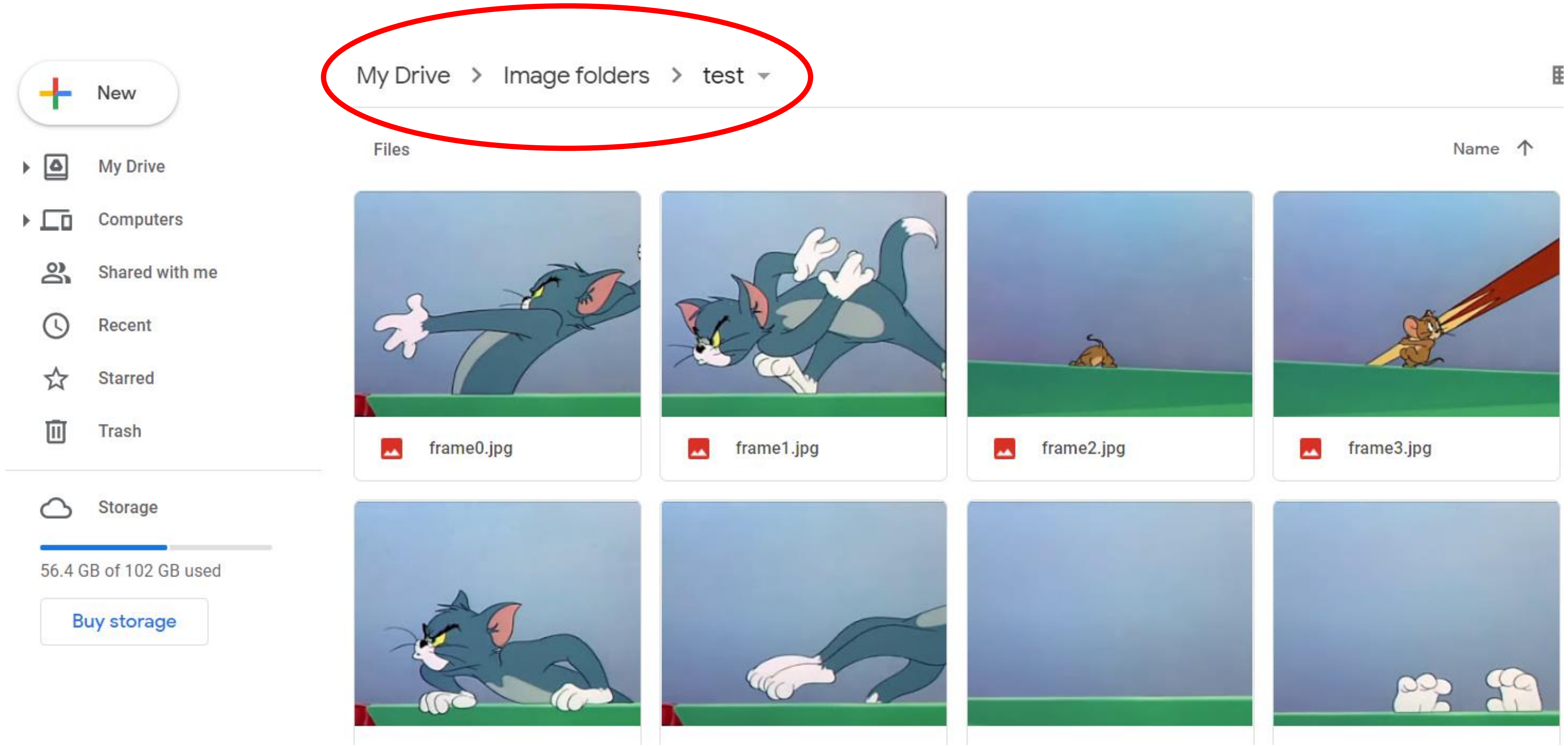

Tom & Jerry Detection | Kaggle
kaggle.com/vijayjoyz/tom-jerry-detection
應用程式
Microsoft Azure N...
免費線上影片轉Gif...
YouTube
Google 學術搜尋
GitHub
Colaboratory
李弘毅 ML
元智大學個人portal
其他書籤
kaggle
Search
Sign In
Register
Home
Compete
Data
Code
Communities
Courses
More
Dataset
Tom & Jerry Detection
Image Dataset For detecting tom & jerry
Vijayakumar • updated a year ago (Version 1)
10
Data
Tasks
Code (2)
Discussion
Activity
Metadata
Download (49 MB)
New Notebook
Usability 7.5
License Database: Open Database, Contents: Database Contents
Tags arts and entertainment, computer science, classification, computer vision, comics and animation
Description
FaceDetection
Face Recognition is a recognition technique used to detect the faces of individuals whose images saved in the data set. Despite the point that other methods of identification can be more accurate, face recognition has always remained a significant focus of research because of its non-meddling nature and because it is people's facile method of personal identification.
We use cookies on Kaggle to deliver our services, analyze web traffic, and improve your experience on the site. By using Kaggle, you agree to our use of cookies.
Got it
Learn more

Save images in your Google drive





Save images in your Google drive



Practice

- Run “7.3. Transfer learning.ipynb"

Build our own image classifier

- Suppose input image size = (224, 224, 3)
- Output has 5 classes: Angry, Happy, Sad, Surprised, Unknown

In [3]:
```
import torch.nn as nn
# fix the weight of convolution layers
model.features.eval()

# modify classifier
model.classifier = torch.nn.Sequential(
  nn.Linear(25088, 4096),
  nn.ReLU(inplace=True),
  nn.Dropout(p=0.5, inplace=False),
  nn.Linear(4096, 4096),
  nn.ReLU(inplace=True),
  nn.Dropout(p=0.5, inplace=False),
  torch.nn.Linear(4096, 5))
```

Summary of parameters

```
Total params: 139,590,725
Trainable params: 139,590,725
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 238.68
Params size (MB): 532.50
Estimated Total Size (MB): 771.75
----------------------------------------------------------------
```

MLP in "4.2. Classification with CE loss"

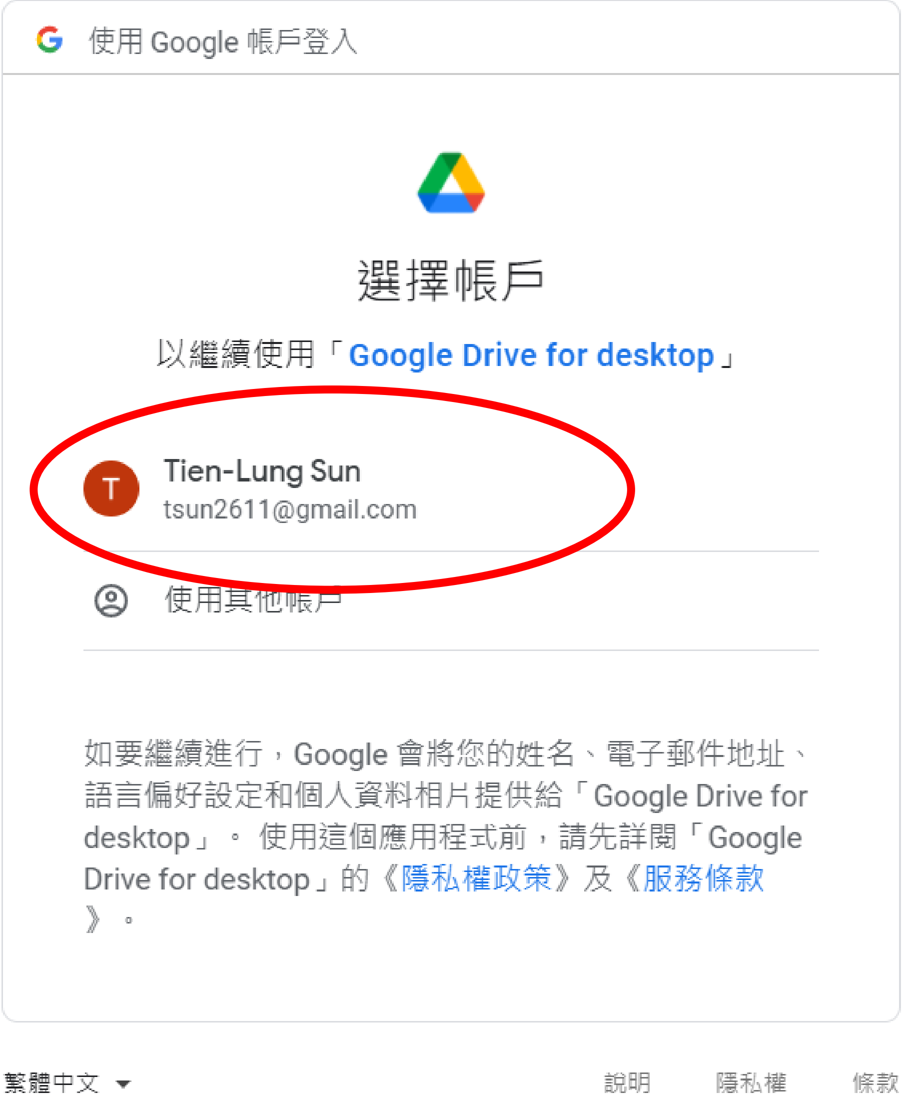
```
BATCH_SIZE = 30
summary(MyNet, input_size=(BATCH_SIZE, 2))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 30, 50]             150
              ReLU-2               [-1, 30, 50]               0
            Linear-3              [-1, 30, 100]           5,100
              ReLU-4              [-1, 30, 100]               0
            Linear-5               [-1, 30, 50]           5,050
              ReLU-6               [-1, 30, 50]               0
            Linear-7                [-1, 30, 2]             102
================================================================
Total params: 10,402
Trainable params: 10,402
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.09
Params size (MB): 0.04
Estimated Total Size (MB): 0.13
----------------------------------------------------------------
```

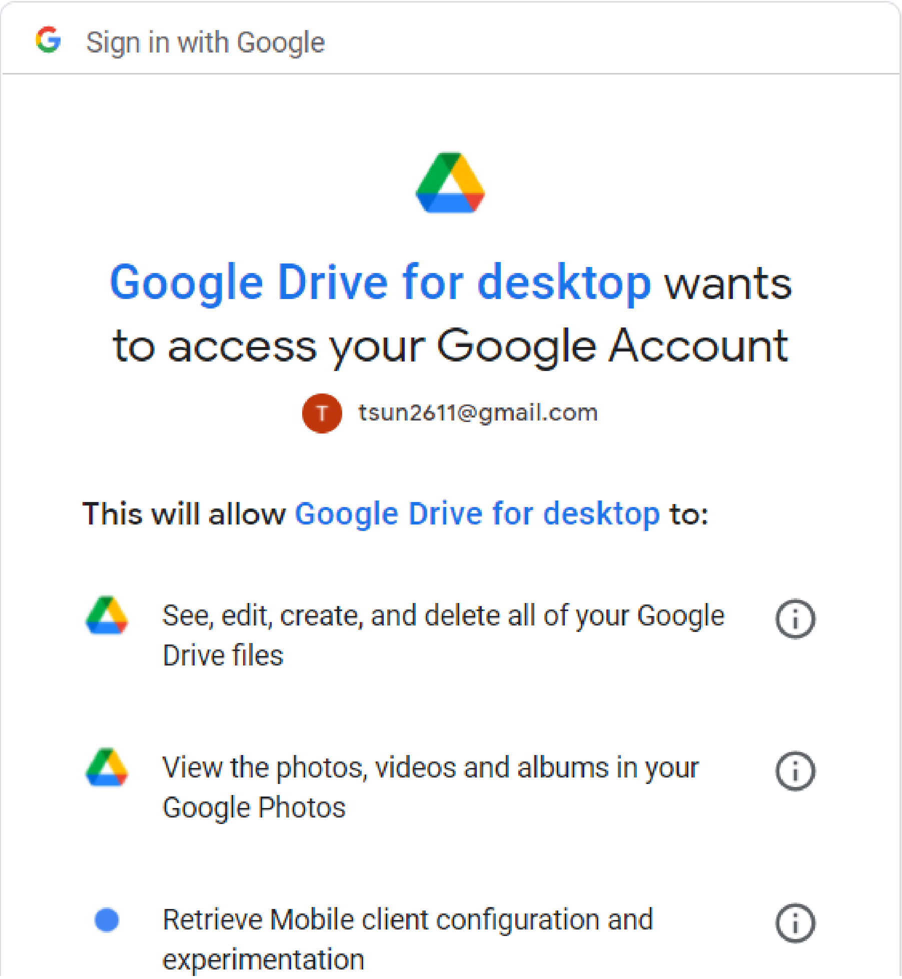
Connect to Google drive

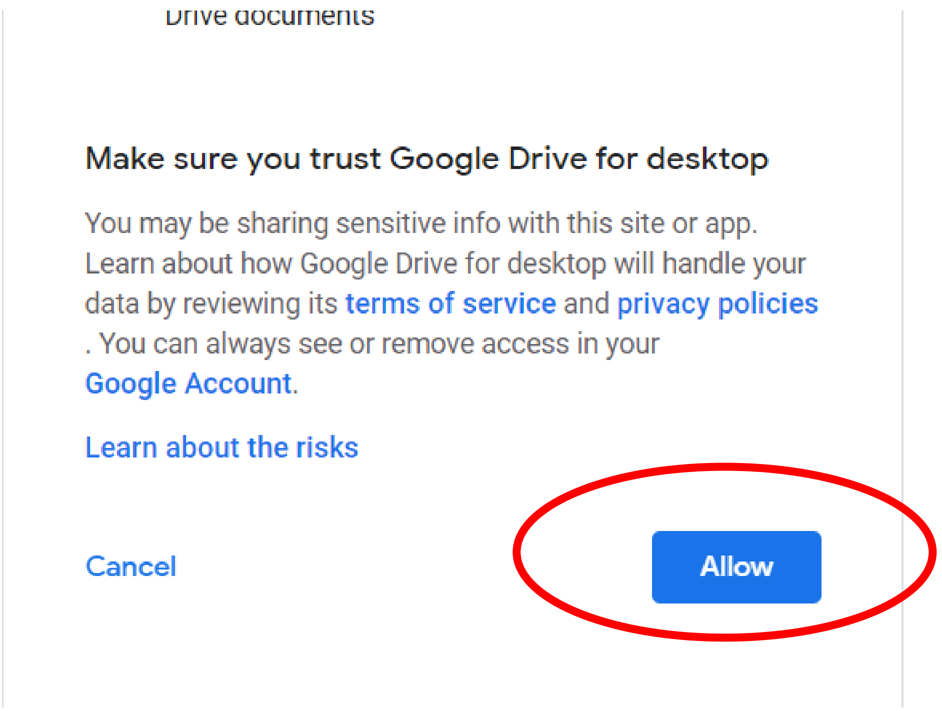
```python
from google.colab import drive
drive.mount("/content/gdrive")
```

```
Go to this URL in a browser: https://accounts.google.com/o/

Enter your authorization code:
```

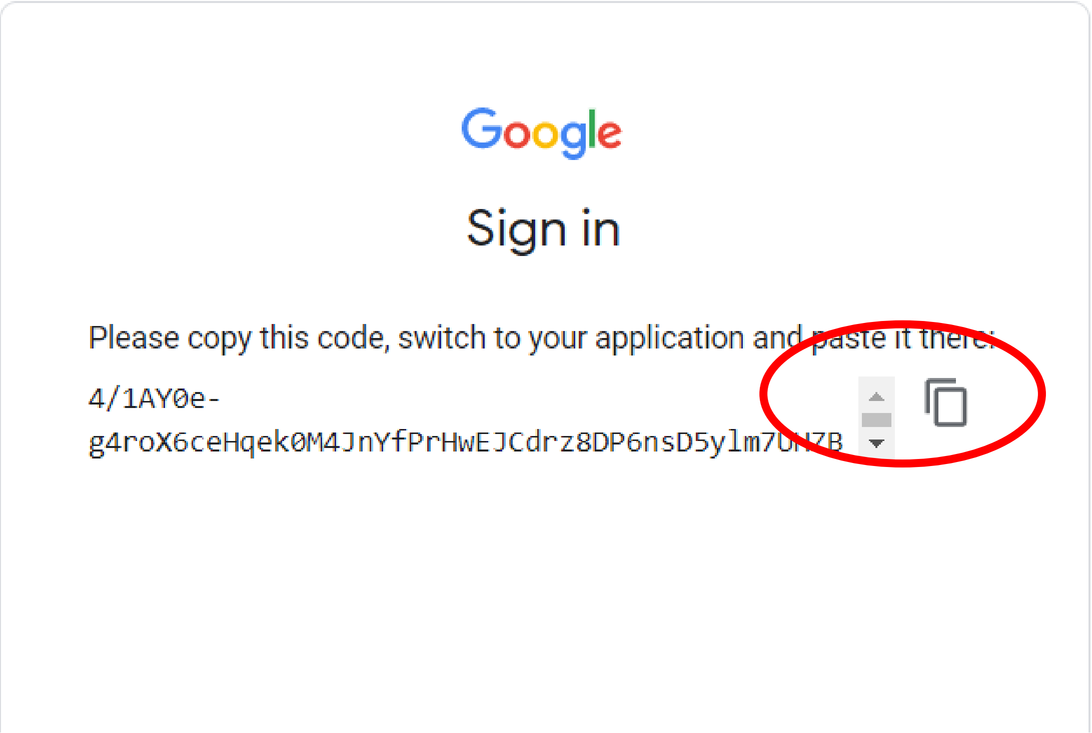


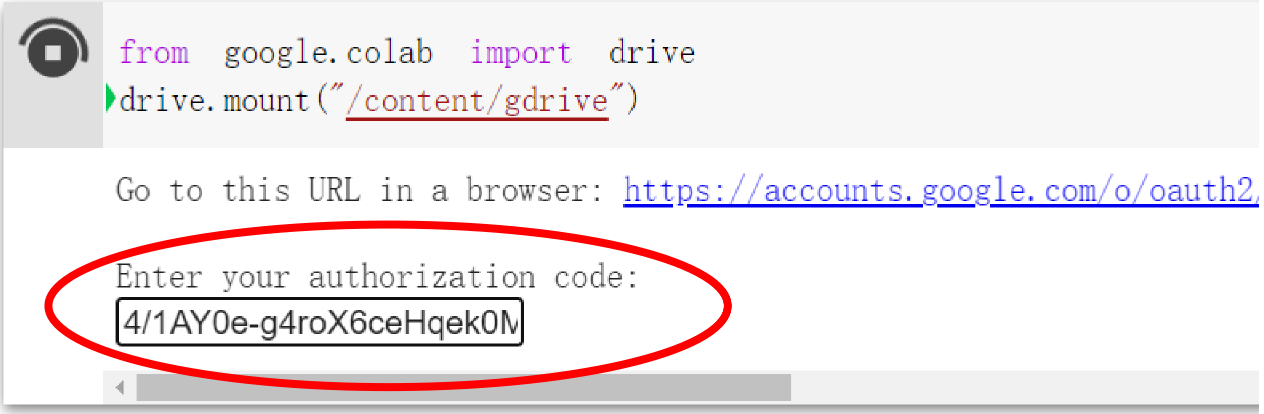
Connect to Google drive





Connect to Google drive






```
[7]  from google.colab import drive
     drive.mount("/content/gdrive")

     Mounted at /content/gdrive
```

Batch training using Image Folder

In [8]:
```
from torchvision import transforms
transformer = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5] )])
```

In [9]:
```
from torchvision import datasets
train_dataset = datasets.ImageFolder(root = "/content/gdrive/MyDrive/Image folders/train", transform = transformer)
```

n [10]:
```
classes = train_dataset.classes
classes_index = train_dataset.class_to_idx
print(classes)
print(classes_index)
```

```
['angry', 'happy', 'sad', 'surprised', 'unknown']
{'angry': 0, 'happy': 1, 'sad': 2, 'surprised': 3, 'unknown': 4}
```

n [11]:
```
import torch.utils.data as Data
loader = Data.DataLoader(dataset=train_dataset,batch_size=4,shuffle=True)
```

Batch training using data in RAM

In [9]:
```
tensorX = torch.FloatTensor(trainX).to(device)
tensorY_hat = torch.LongTensor(trainY_hat).to(device)
print(tensorX.shape, tensorY_hat.shape)
```

```
torch.Size([128, 2]) torch.Size([128])
```

In [10]:
```
torch_dataset = Data.TensorDataset(tensorX, tensorY_hat)
```

In [11]:
```
loader = Data.DataLoader(
    dataset=torch_dataset,
    batch_size=5,
    shuffle=True,
    num_workers=0,    # subprocesses for loading data
)
```

In [12]:
```
for (batchX, batchY_hat) in loader:
    break
print(batchX.shape, batchY_hat)
```

```
torch.Size([5, 2]) tensor([0, 0, 0, 1, 1], device='cuda:0')
```

One batch has 4 images

```
[12]: for batchX, batchY_hat in loader:
        break;
      print(batchX.shape, batchY_hat.shape, batchY_hat)
```

```
torch.Size([4, 3, 224, 224]) torch.Size([4]) tensor([3, 2, 3, 2])
```

```
[13]: import numpy as np
      import matplotlib.pyplot as plt
      imgTensor = torchvision.utils.make_grid(batchX)
      imgArray = imgTensor.numpy()
      imgArray1 = np.zeros((imgArray.shape[1], imgArray.shape[2], 3))
      imgArray1[:,:,0] = imgArray[0, :, :]
      imgArray1[:,:,1] = imgArray[1, :, :]
      imgArray1[:,:,2] = imgArray[2, :, :]
      imgArray1 = imgArray1*0.5+0.5
      plt.figure(figsize=(12, 6))
      plt.imshow(imgArray1)
      plt.show()
      print([classes[i] for i in batchY_hat])
```


```
['surprised', 'sad', 'surprised', 'sad']
```

Batch training loop

[16]:
```
lossLst = []
accuracyLst = []
for epoch in range(1, 4):
  print("\nepoch = ", epoch, end = ", ")
  print("batch: ", end="")
  for step, (batch_x, batchY_hat) in enumerate(loader):
    if(step%5==0):
      print(step, end = ", ")
    tensorY = model(batch_x.to(device))
    loss = loss_func(tensorY, batchY_hat.to(device))
    lossLst.append(float(loss))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    correct = 0
    tensorY = torch.softmax(tensorY, 1)
    MaxIdxOfEachRow = torch.max(tensorY, 1)[1]
    for i in range(batchY_hat.shape[0]):
      if (int(MaxIdxOfEachRow[i]) == int(batchY_hat[i])):
        correct += 1
    accuracy = correct/batchY_hat.shape[0]
    accuracyLst.append(accuracy)
```

```
epoch =  1, batch: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 
epoch =  2, batch: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 
epoch =  3, batch: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 
```

MLP in "4.2. Classification with CE loss"

```
lossLst = []
accuracyLst = []
for epoch in range(1, 500):
  for (batchX, batchY_hat) in loader:
    tensorY = MyNet(batchX)
    loss = loss_func(tensorY, batchY_hat)
    lossLst.append(float(loss))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    correct = 0
    tensorY = torch.softmax(tensorY, 1)
    MaxIdxOfEachRow = torch.max(tensorY, 1)[1]
    for i in range(batchY_hat.shape[0]):
      if (int(MaxIdxOfEachRow[i]) == int(batchY_hat[i])):
        correct += 1
    accuracy = correct/batchY_hat.shape[0]
    accuracyLst.append(accuracy)
```

Transfer learning design 2

Use first 10 layers in convolution section

Let input image size = (224, 224, 3), Output has 5 classes: Angry, Happy, Sad, Surprised, Unknown

```
[3] import torch.nn as nn
    class MyCNN(nn.Module):
        def __init__(self):
            super(MyCNN, self).__init__()
            self.features = vgg19.features[0:10] #layer 0-9
            self.classifier = nn.Sequential(
                nn.Dropout(),
                nn.Linear(56*56*128, 4096),
                nn.ReLU(inplace=True),
                nn.Dropout(p=0.5, inplace=False),
                nn.Linear(4096, 4096),
                nn.ReLU(inplace=True),
                nn.Dropout(p=0.5, inplace=False),
                nn.Linear(4096, 5),
            )
        def forward(self, x):
            x = self.features(x)
            x = torch.flatten(x, 1)
            x = self.classifier(x)
            return x
```

1,661M parameters !

```
[5] from torchsummary import summary
    summary(model, input_size=(3, 224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
          Dropout-11               [-1, 401408]               0
           Linear-12                 [-1, 4096]   1,644,171,264
             ReLU-13                 [-1, 4096]               0
          Dropout-14                 [-1, 4096]               0
           Linear-15                 [-1, 4096]      16,781,312
             ReLU-16                 [-1, 4096]               0
          Dropout-17                 [-1, 4096]               0
           Linear-18                    [-1, 5]          20,485
================================================================
Total params: 1,661,233,221
Trainable params: 1,661,233,221
Non-trainable params: 0
```

```
Total params: 139,590,725
Trainable params: 139,590,725
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 238.68
Params size (MB): 532.50
Estimated Total Size (MB): 771.75
----------------------------------------------------------------
```

CUDA out of memory!

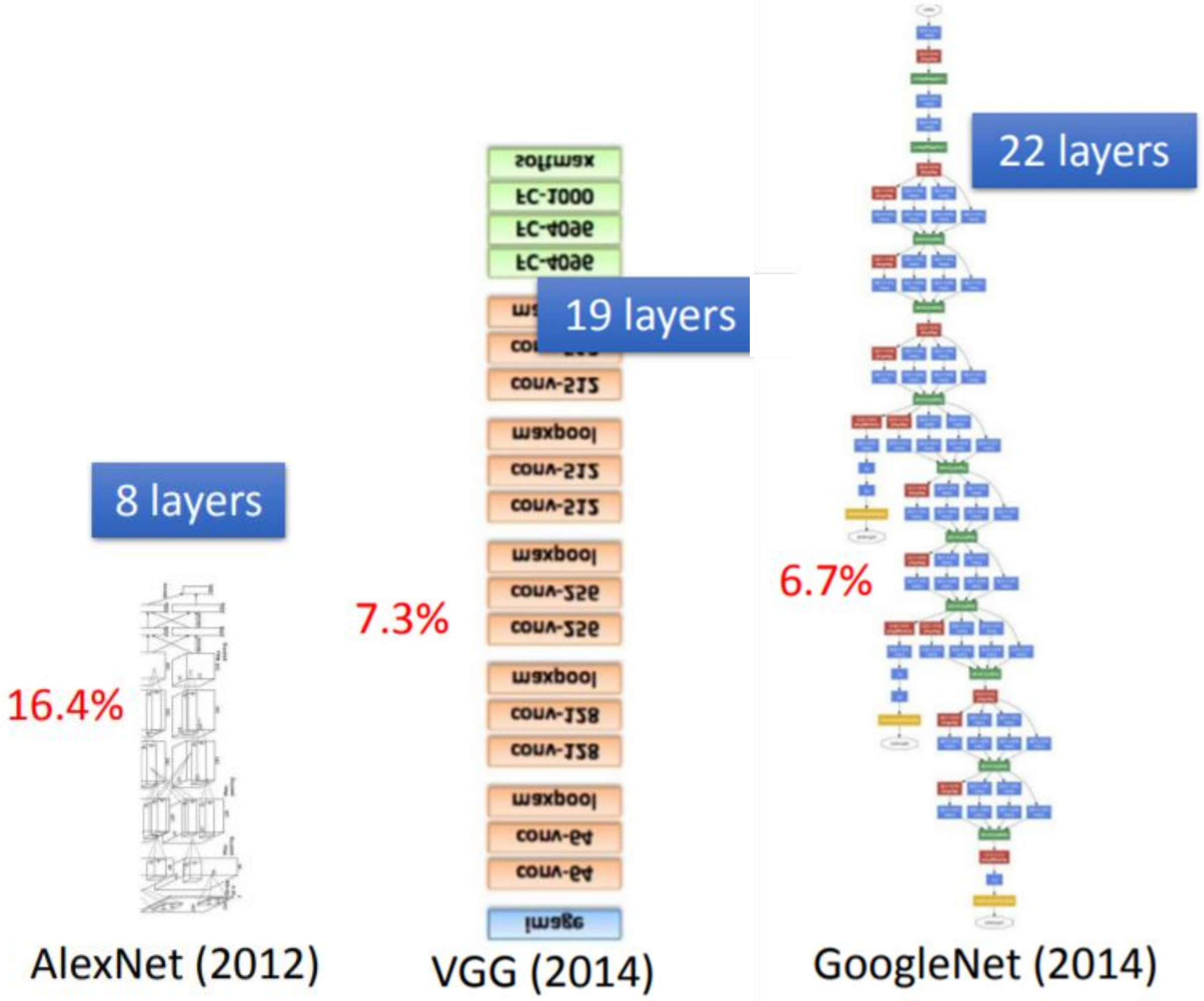
```
epoch =  1, batch: 0,
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-17-94eca5998520> in <module>()
     11     lossLst.append(float(loss))
     12     optimizer.zero_grad()
---> 13     loss.backward()
     14     optimizer.step()
     15

                              1 frames

/usr/local/lib/python3.7/dist-packages/torch/autograd/__init__.py in backward(tensors,
grad_tensors, retain_graph, create_graph, grad_variables, inputs)
    145     Variable._execution_engine.run_backward(
    146         tensors, grad_tensors_, retain_graph, create_graph, inputs,
--> 147         allow_unreachable=True, accumulate_grad=True)  # allow_unreachable flag
    148
    149

RuntimeError: CUDA out of memory. Tried to allocate 6.12 GiB (GPU 0; 11.17 GiB total
capacity; 6.46 GiB already allocated; 4.27 GiB free; 6.47 GiB reserved in total by PyTorch)
```

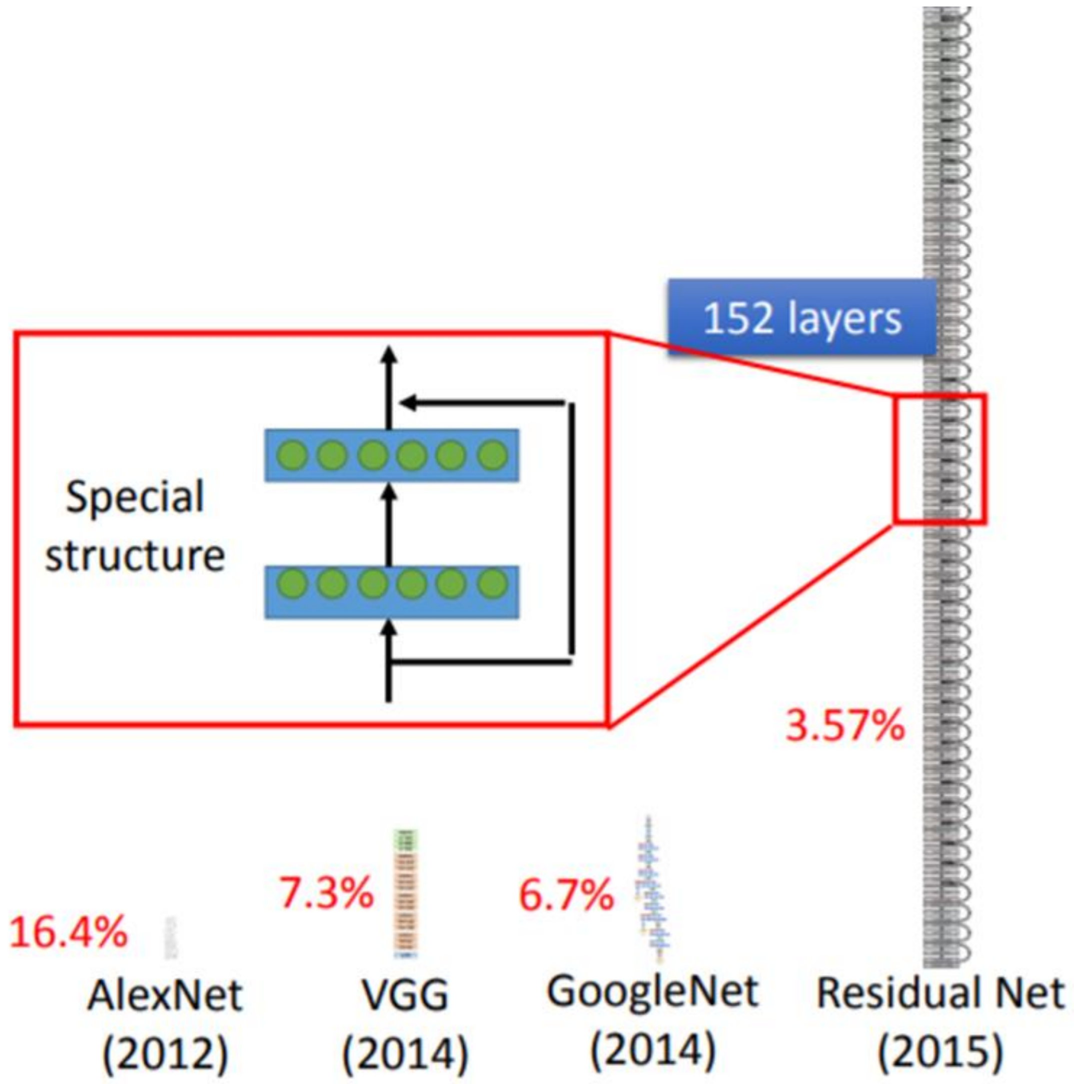
HW5 (2)

- Use transfer learning to train your own image classifier with other image data set, e.g., your face vs your friend's face.

ResNet

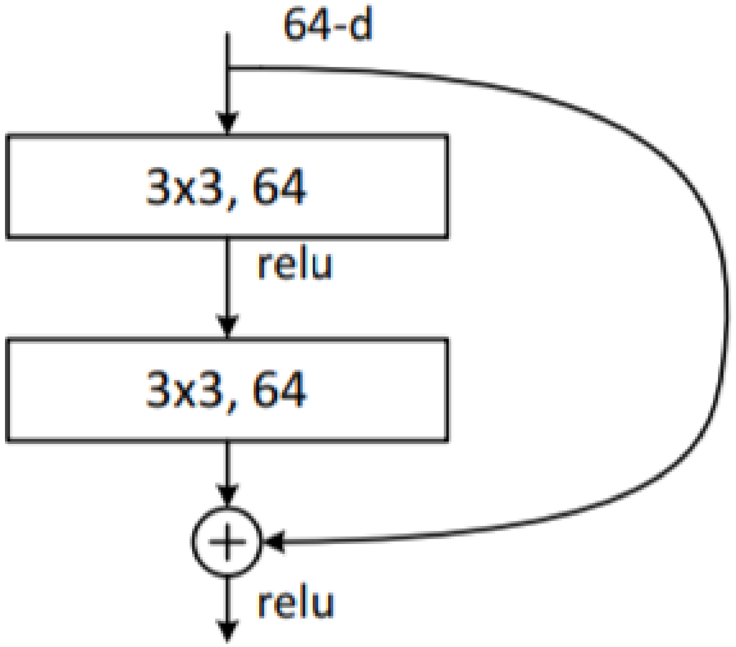
Going deeper and deeper…



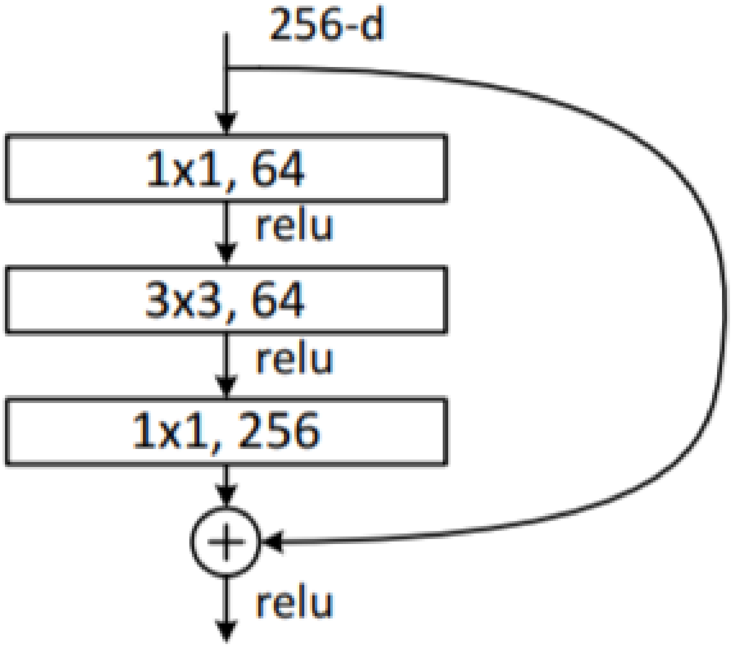


Reference: 李弘毅 ML Lecture 6 https://youtu.be/Dr-WRlEFefw

ResNet



Basic block



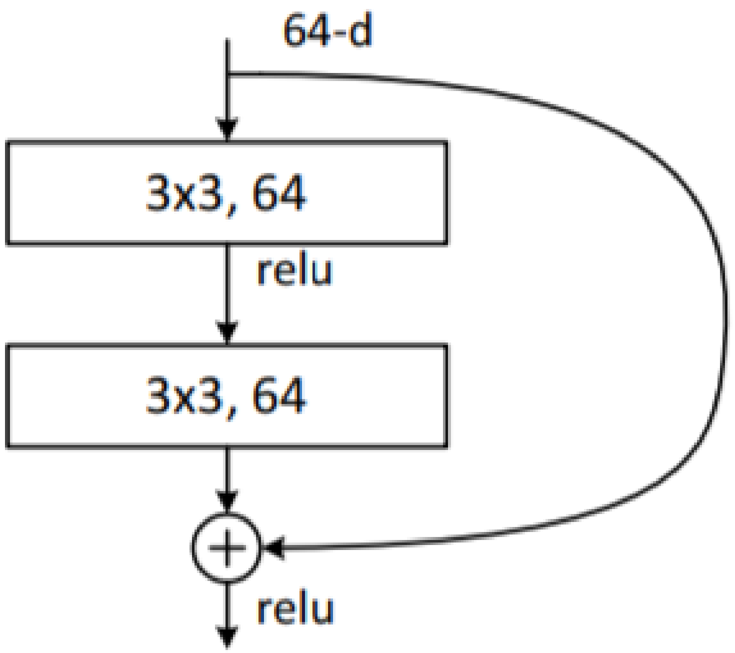
Bottleneck block

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

Practice

- Run “7.4. Build ResNet from scratch.ipynb"

Basic loop




```python
class BasicBlock(nn.Module):
  expansion = 1
  def __init__(self, inplanes, planes, stride=1, downsample=None,):
    super(BasicBlock, self).__init__()
    self.conv1=conv3x3(inplanes,planes,stride)
    self.bn1=nn.BatchNorm2d(planes)
    self.relu=nn.ReLU(inplace=True)
    self.conv2=conv3x3(planes,planes)
    self.bn2=nn.BatchNorm2d(planes)
    self.downsample=downsample
    self.stride=stride

    if(stride!=1 or inplanes!=planes*self.expansion):
      self.downsample=nn.Sequential(
        nn.Conv2d(inplanes,planes*self.expansion,kernel_size=1,stride
        nn.BatchNorm2d(planes*self.expansion),
      )

  def forward(self, x):
    residual = x
    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)
    out = self.conv2(out)
    out = self.bn2(out)

    # Downsample:feature Map size/2 || Channel increase
    if (self.downsample is not None):
      residual = self.downsample(x)
    print("out= ", out.shape, "residual= ", residual.shape)
    out+=residual
    out=self.relu(out)
    return out
```

Batch Normalization

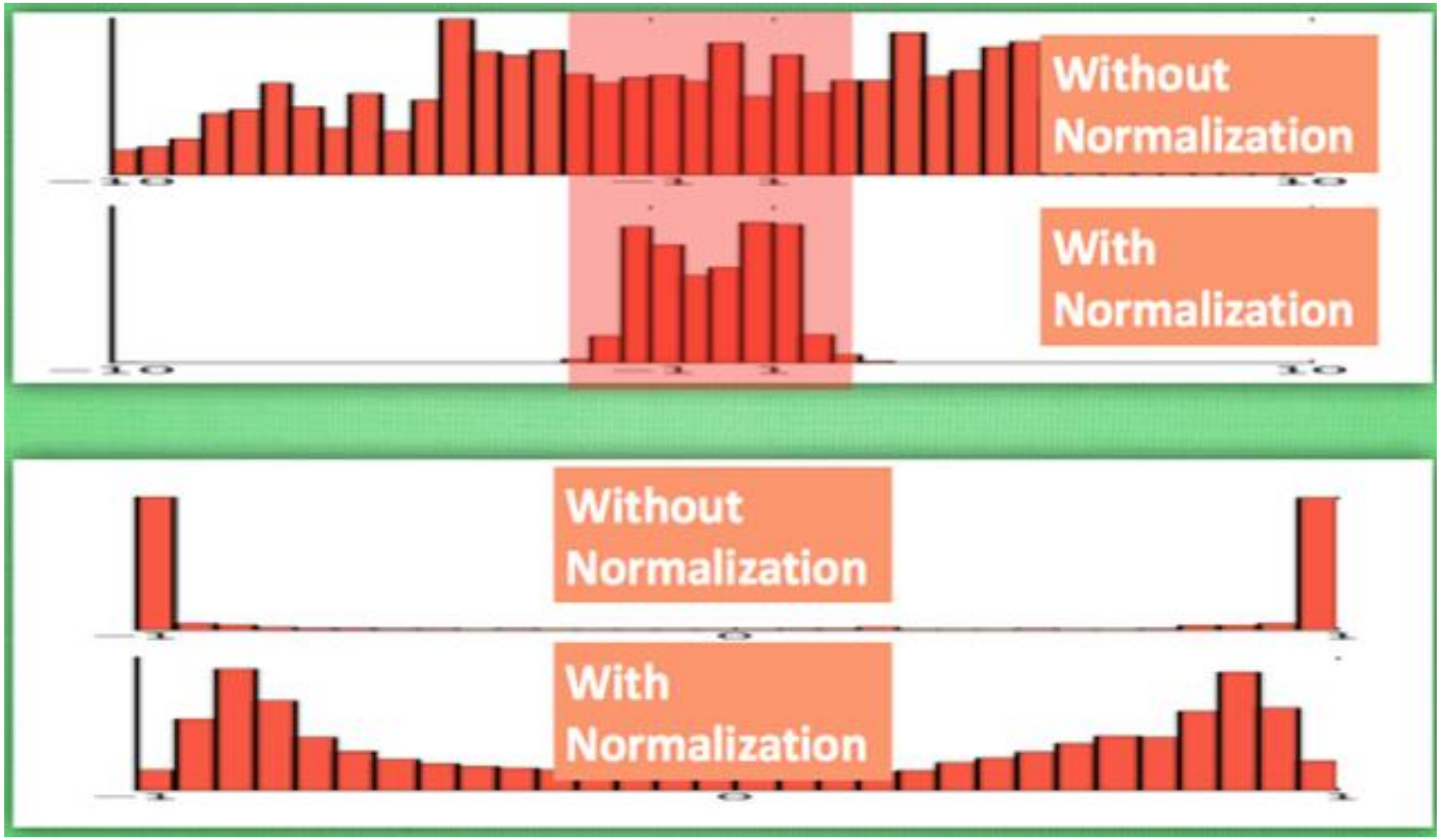
Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

- The mean and standard-deviation are calculated per-dimension over the mini-batches.
- By default, the elements of γare set to 1 and the elements of β are set to 0.

https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html

Batch Normalization



https://medium.com/ching-i/batch-normalization-%E4%BB%8B%E7%B4%B9-135a24928f12

```python
class MyResNet(nn.Module):
  def __init__(self, block, layers, num_classes=2):
    super(MyResNet, self).__init__()
    self.inplanes = 64
    self.dilation = 1
    self.conv1=nn.Conv2d(3,self.inplanes,kernel_size=7,stride=2,
    self.maxpool=nn.MaxPool2d(kernel_size=3,stride=2, padding=1)
    self.layer1=self._make_layer(block,64,layers[0])
    self.layer2=self._make_layer(block,128,layers[1],stride=2)
    self.avgpool=nn.AdaptiveAvgPool2d((1,1))
    self.fc=nn.Linear(128*block.expansion,num_classes)
    self.linear=nn.Linear(128*block.expansion,num_classes)

  def _make_layer(self, block, planes, blocks, stride=1):
    layers=[]
    layers.append(block(self.inplanes,planes,stride))
    self.inplanes=planes*block.expansion

    for i in range(1,blocks):
      layers.append(block(self.inplanes,planes))
    return nn.Sequential(*layers)

  def forward(self, x):
    x=self.conv1(x)
    x=self.maxpool(x)
    x=self.layer1(x)
    x=self.layer2(x)
    x=self.avgpool(x)
    x=torch.flatten(x, 1)
    x=self.fc(x)
    return x
```

Practice - Build ResNet from scratch

```python
model=MyResNet(BasicBlock,[1,1]).to(device)
print(model)
```

```
MyResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=128, out_features=2, bias=True)
  (linear): Linear(in_features=128, out_features=2, bias=True)
)
```

Practice – Load ImageNet pre-trained ResNet

```
In [2]: import torchvision
        model = torchvision.models.resnet18(pretrained=True)

Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" t

HBox(children=(FloatProgress(value=0.0, max=46827520.0), HTML(value='')))
```

ResNet

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

ResNet

```
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
```

What does CNN learn?

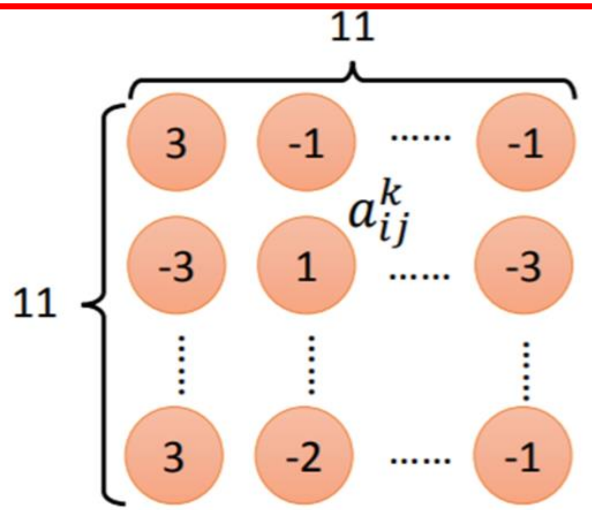
Only the weight of the 1st convolution filters can be directly visualized. How to interpret the filter weights of other convolution layers?

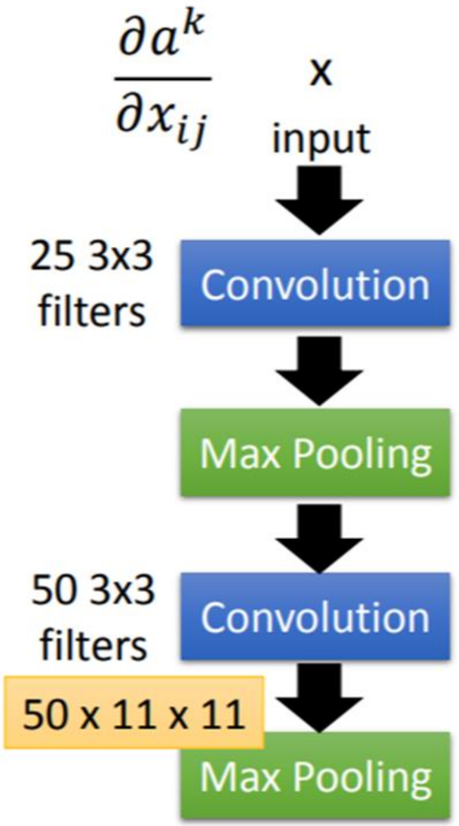
How to use gradient ascent to implement this in PyTorch?

The output of the k-th filter is a 11 x 11 matrix.

Degree of the activation of the k-th filter: $a^k = \sum_{i=1}^{11}\sum_{j=1}^{11} a_{ij}^k$

$\boxed{x^* = \arg\max_x a^k \text{ (gradient ascent)}}$





Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g

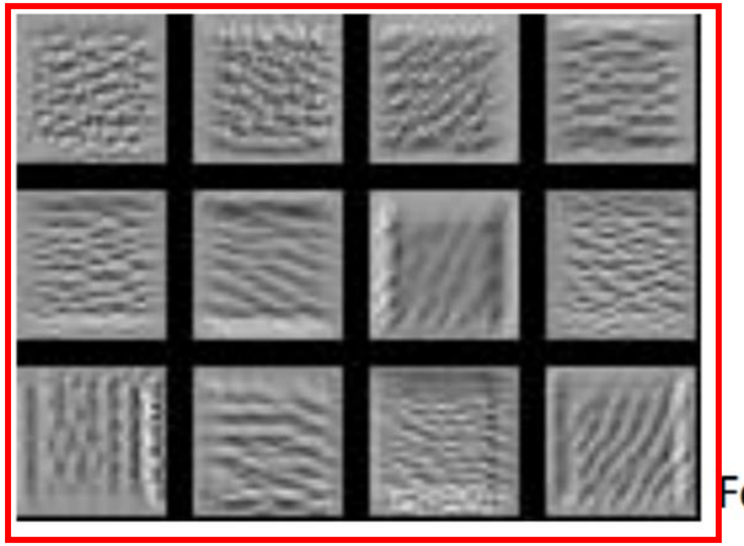
With MNIST data set, in the convolution layer, the filters detects a particular texture pattern.

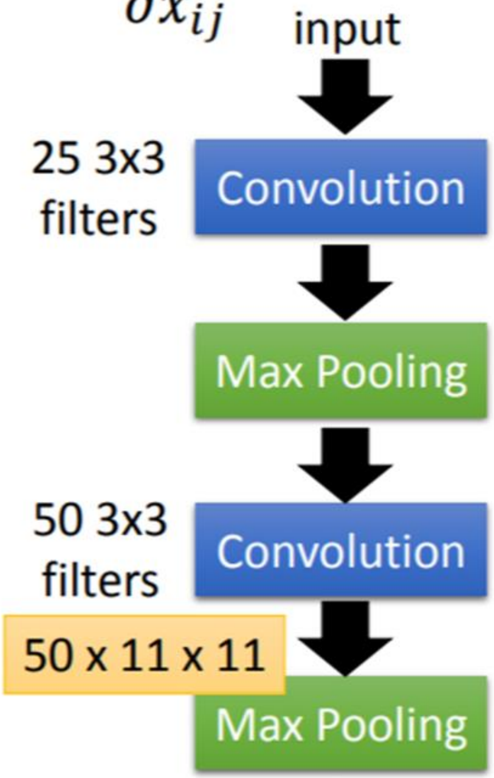
The output of the k-th filter is a 11 x 11 matrix.

Degree of the activation of the k-th filter: $a^k = \sum_{i=1}^{11}\sum_{j=1}^{11} a_{ij}^k$

$x^* = arg \max_x a^k$ (gradient ascent)

$$\frac{\partial a^k}{\partial x_{ij}}$$





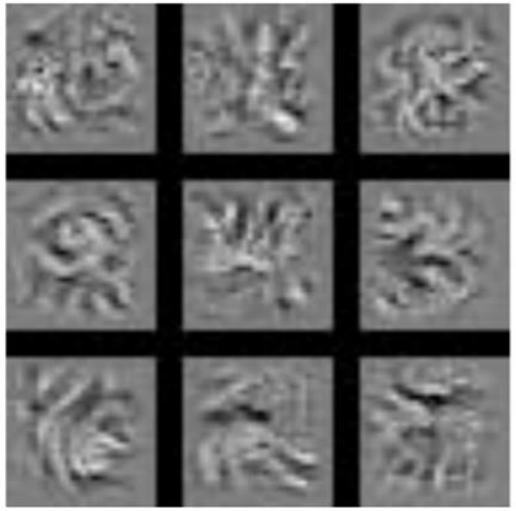
For each filter

How to implement this in PyTorch?

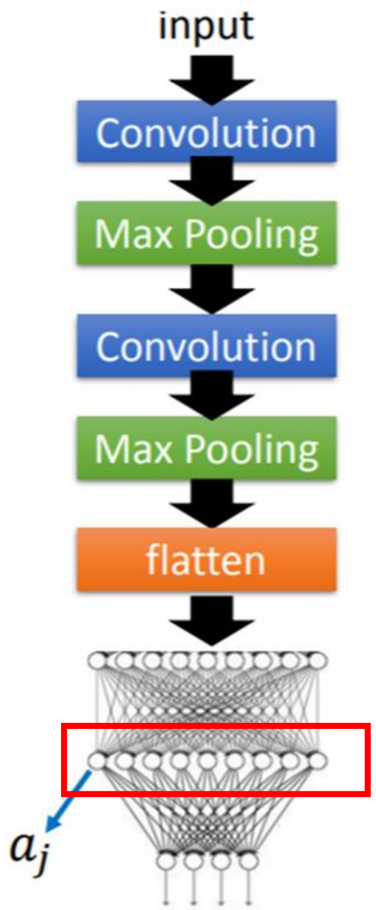
Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g

In the hidden layer of the fully-connected NN, each neuron detects an overall pattern in the picture rather than a particular texture pattern.

Find an image maximizing the output of neuron: $x^* = arg\max_x a^j$

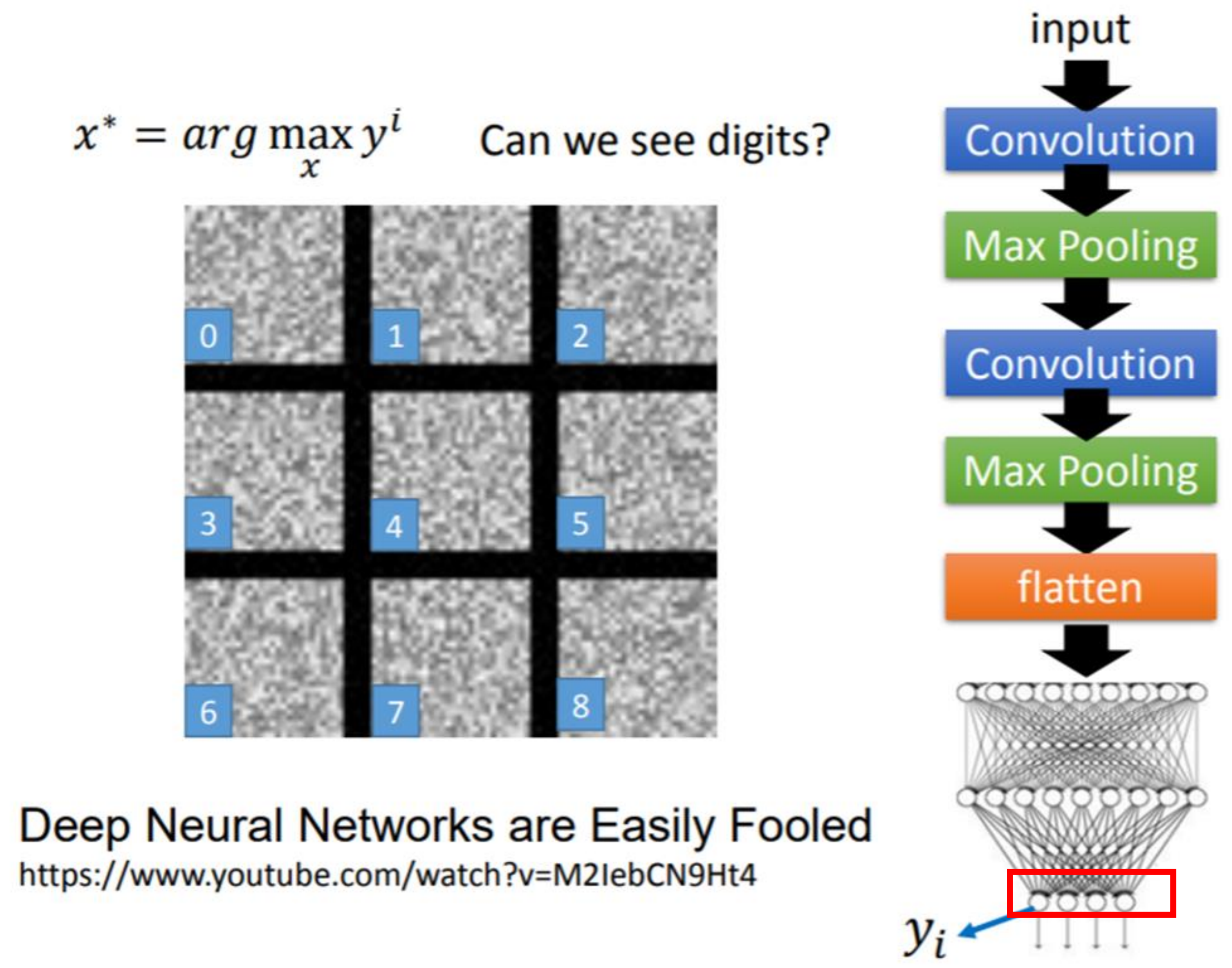


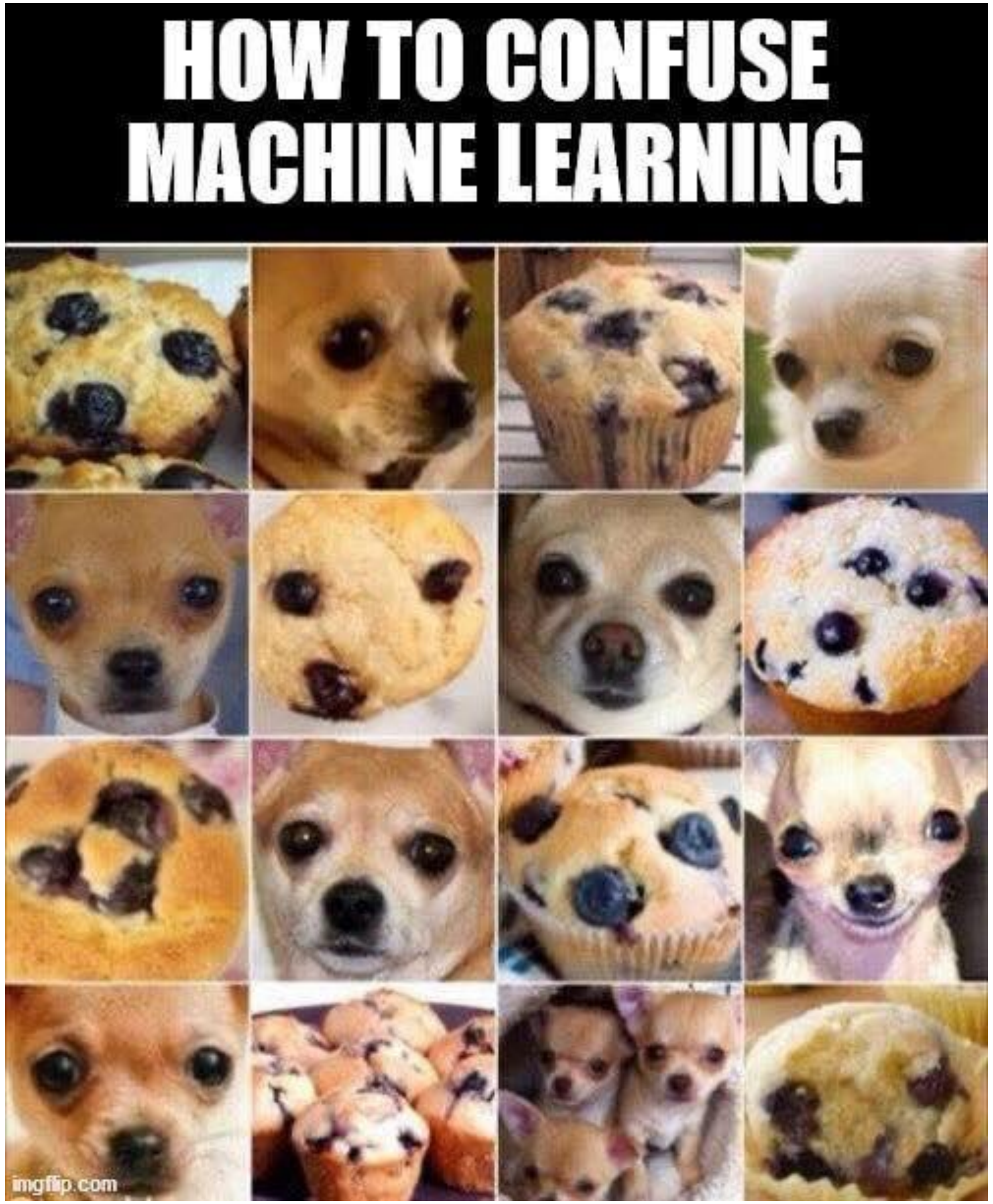
Each figure corresponds to a neuron



Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g

If we watch the output layer node, it is easy to see that CNN is easily fooled.





https://www.facebook.com/105366834575253/photos/a.106433371135266/217334733378462/

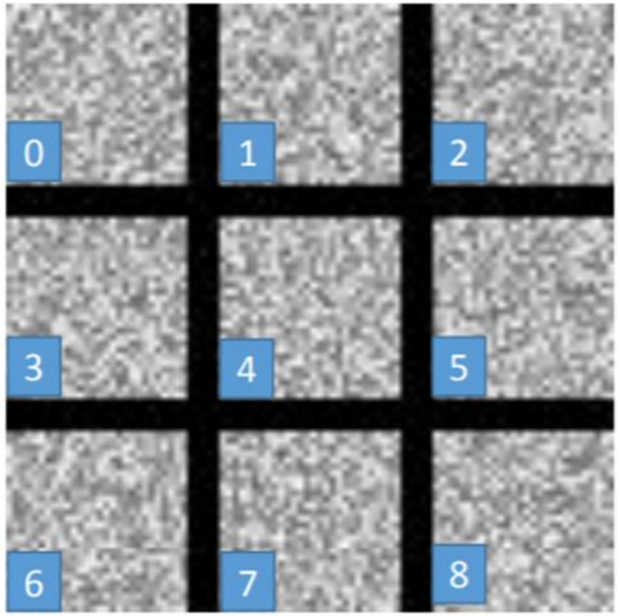
Adding regularization to the objective function to force most pixels be "NO INK"

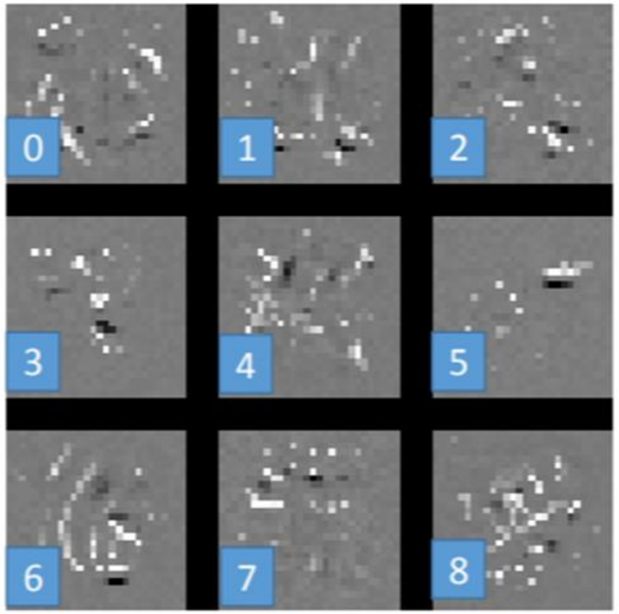
$$x^* = arg\max_x y^i$$

Over all pixel values

$$x^* = arg\max_x \left(y^i - \boxed{\sum_{i,j} |x_{ij}|} \right)$$

Here white pixels indicate ink, and black pixels indicate “NO INK”.



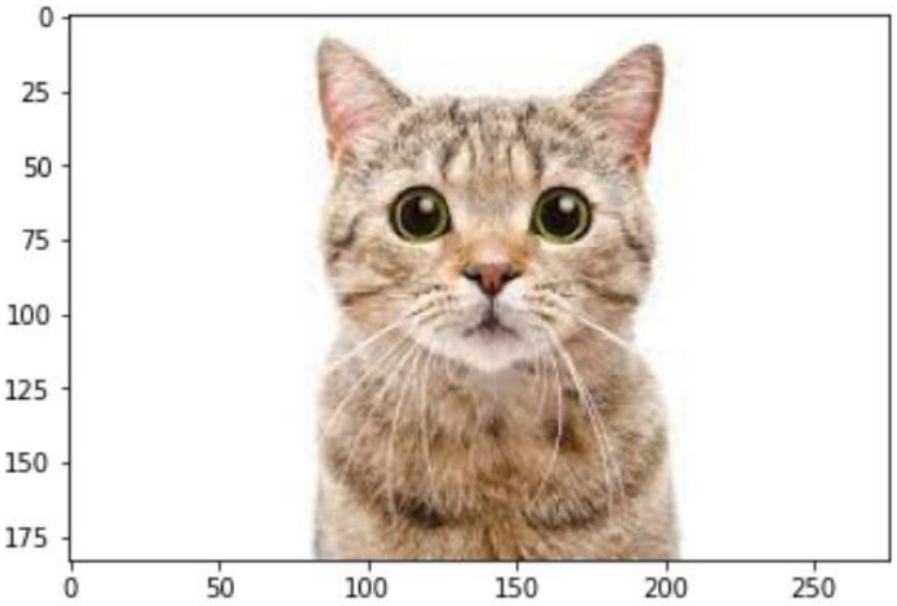


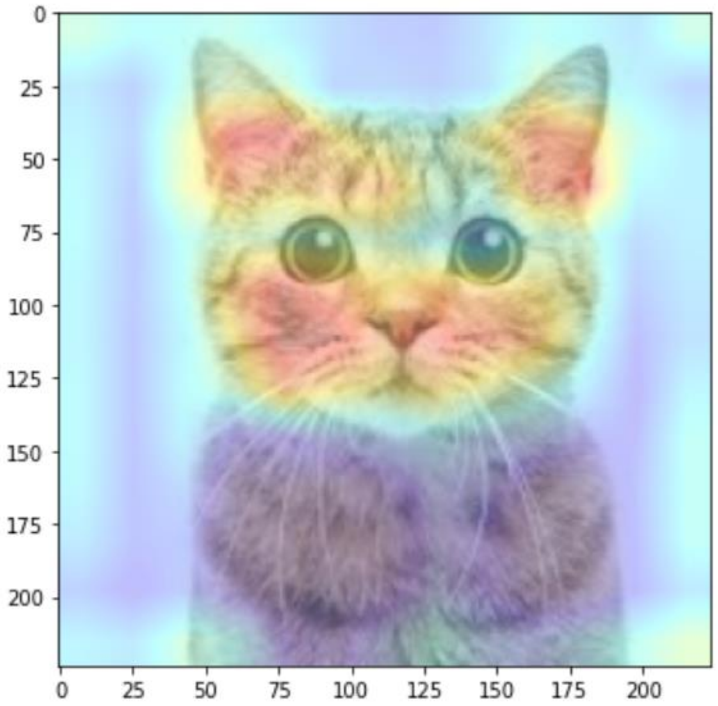
L1 regularization to force xij=0, i.e., force most pixels to be black, NO INK (as only small part of the image has ink)

Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g

Practice – What does CNN learn?

- Run “7.3 GradCAM.ipynb"





HW5 (3)

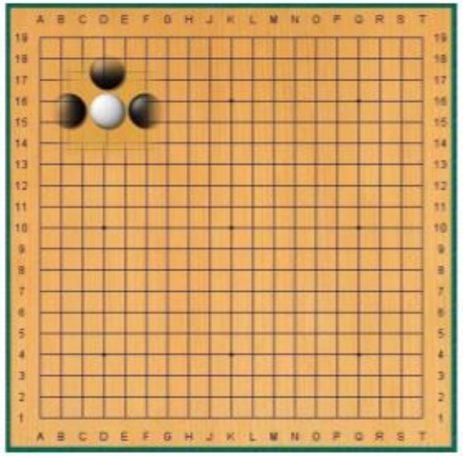
	Class index predicted by the model	Class index you assigned
AlexNet		
VGG		
ResNet18		

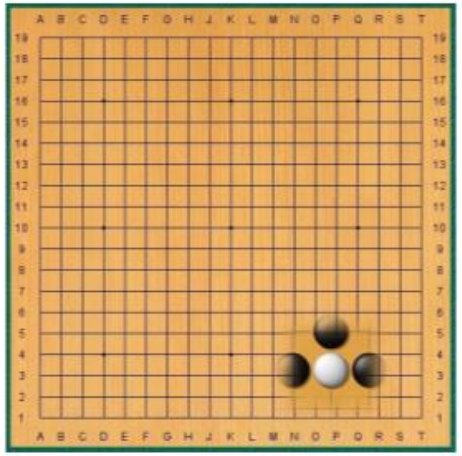
Use CNN in Alpha GO

- Some patterns are much smaller than the whole image

Alpha Go uses 5 x 5 for first layer

- The same patterns appear in different regions.





Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g

Use CNN in Alpha GO

Neural network architecture. The input to the policy network is a 19 × 19 × 48 image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23 × 23 image, then convolves *k* filters of kernel size 5 × 5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21 × 21 image, then convolves *k* filters of kernel size 3 × 3 with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size 1 × 1 with strid … max func-
tion. The … Extended Data Table 3 additionally show the results of training with $k = 128$, 256 and 384 filters.

Alpha Go does not use Max Pooling ……

Reference: 李弘毅 ML Lecture 10 https://youtu.be/FrKWiRv254g