June 14th, 2021

AI+BD ML Lab. Day 4

Convolutional Neural Network (CNN)

YoungIn Kim
<youngkim21@postech.edu>

Contents

Goals

1. Understanding **basic components of CNN**

2. Composing **a CNN model** and test it with CIFAR-10 dataset

3. Watching **Class Activation Map** using CNN

Kaggle competition

Let’s get the highest score!

Kaggle

- **Link**
  - https://www.kaggle.com/t/c5ba45a8b05443f3b1a62c50cb9188a6

Kaggle

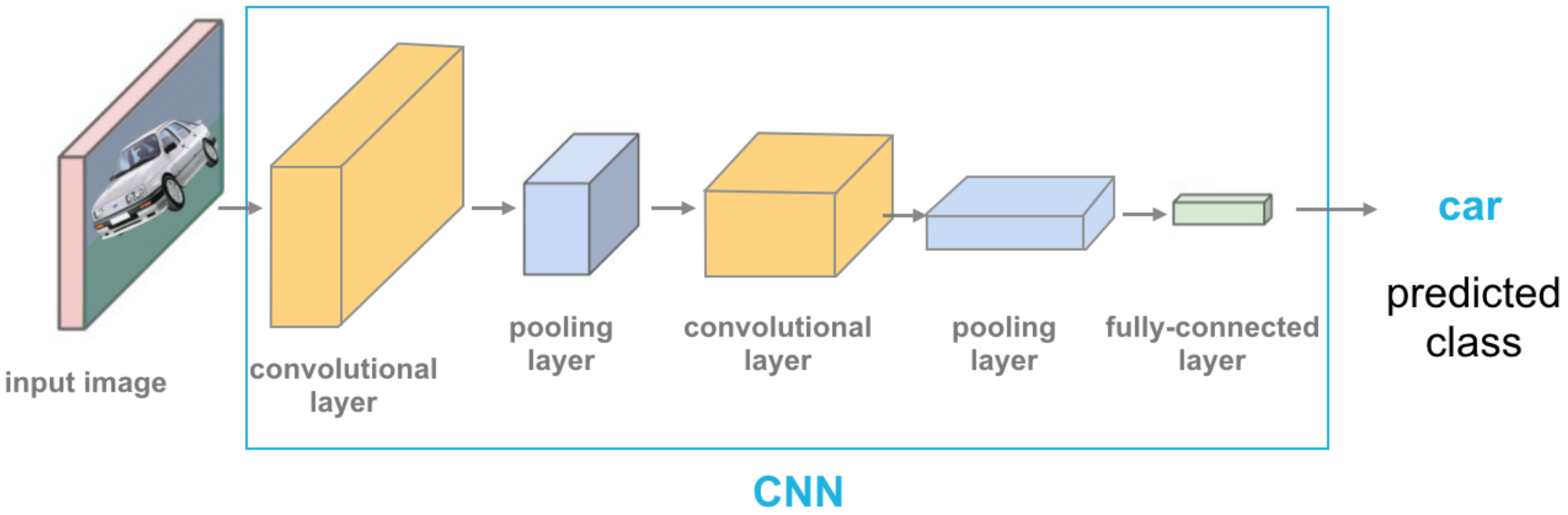
- **Sample code**
  - https://git.io/kaggle-sample
  - need to change “root” & “save_root”

Backgrounds

For your information & reminding

Convolutional Neural Network

- A good network for image



Components of CNN

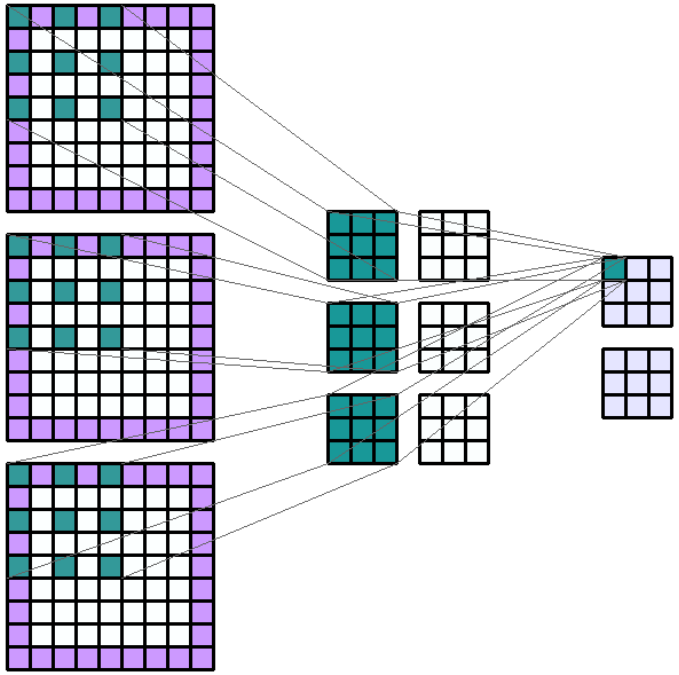
- Convolution Operation (Conv2d)

0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

Kernel

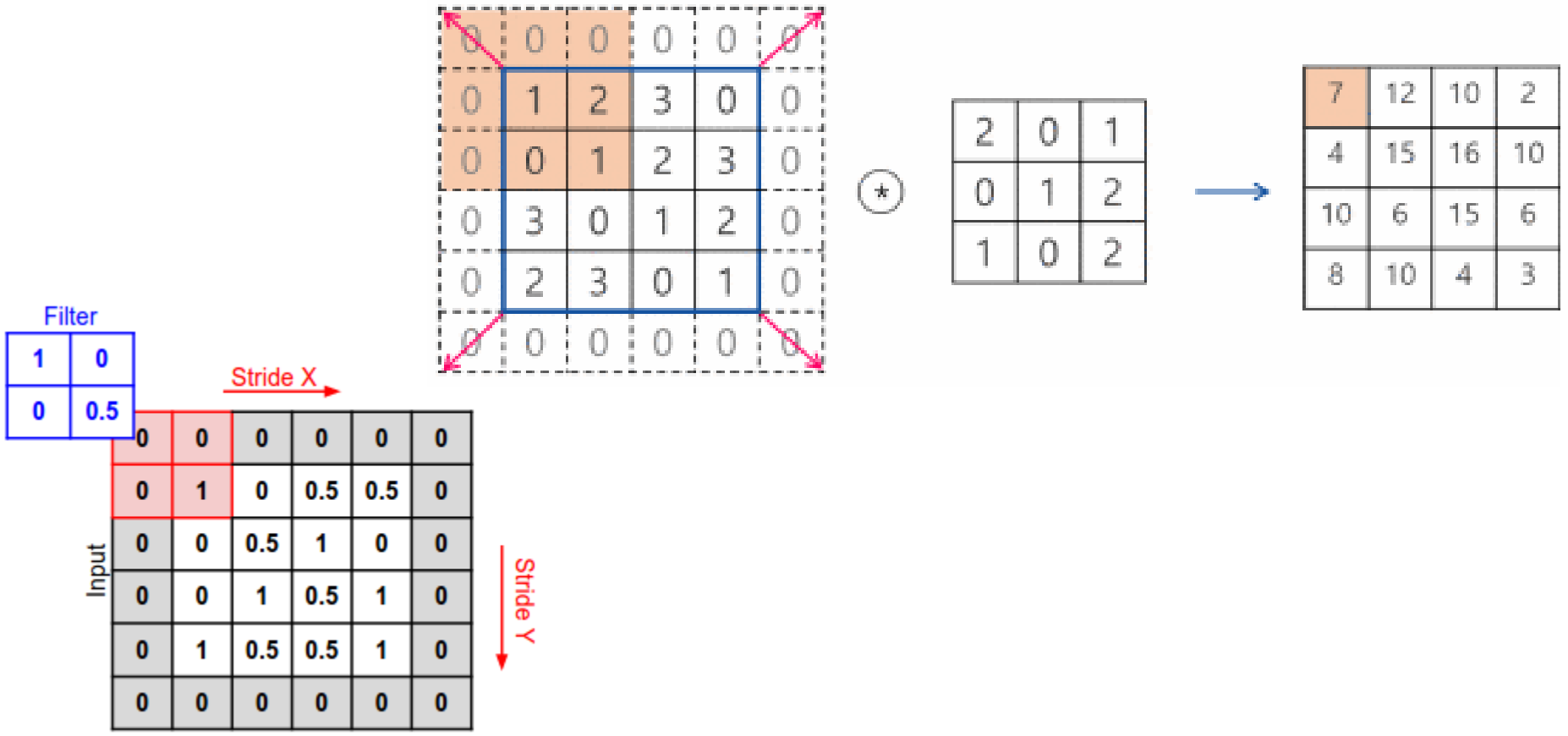
0	-1	0
-1	5	-1
0	-1	0

114				



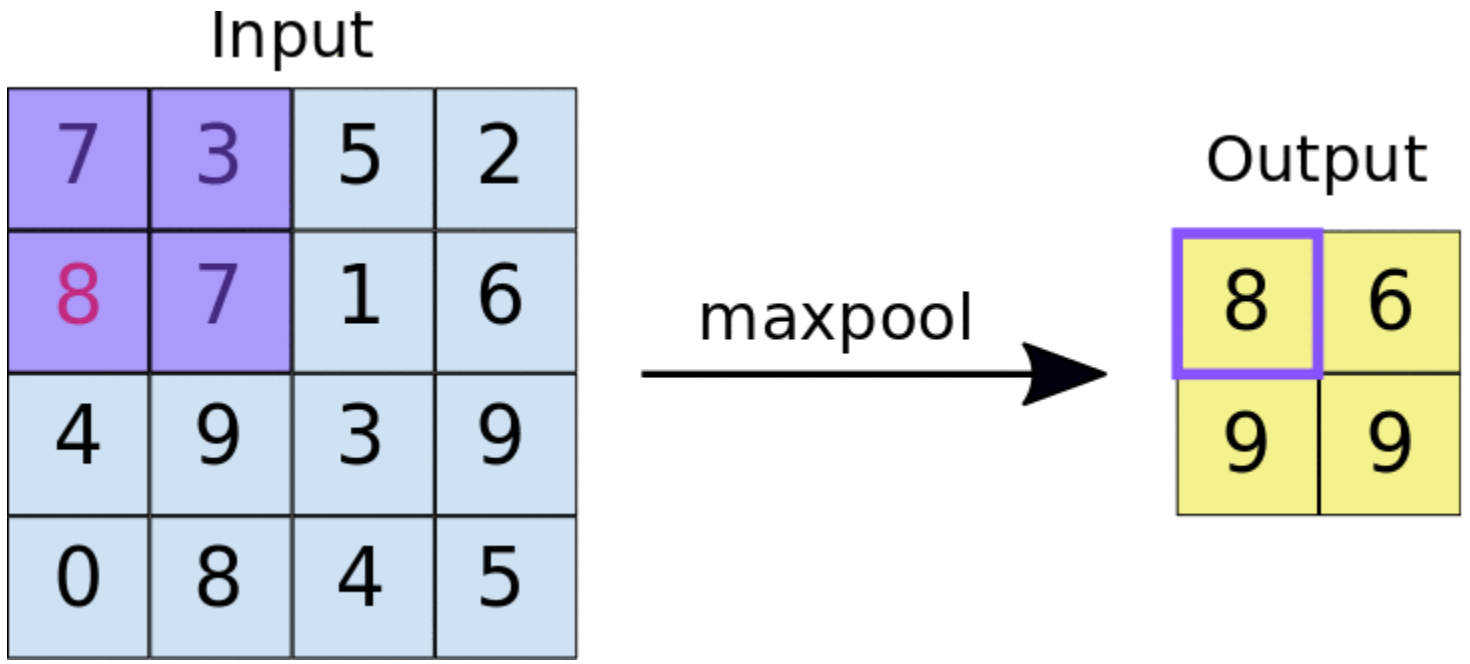
Components of CNN

- Convolution Operation (Conv2d) - Stride, Padding



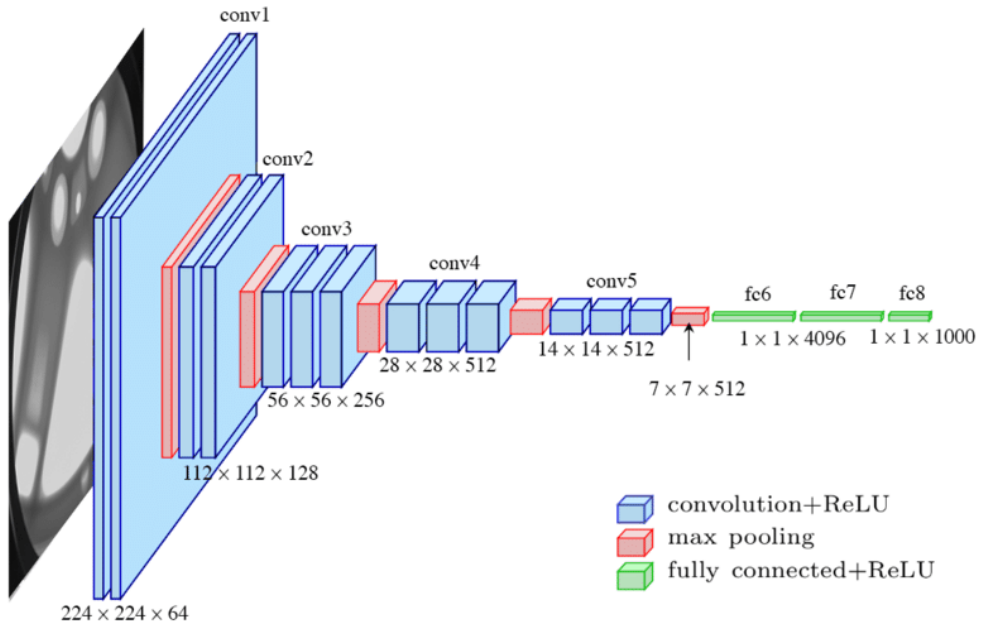
Components of CNN

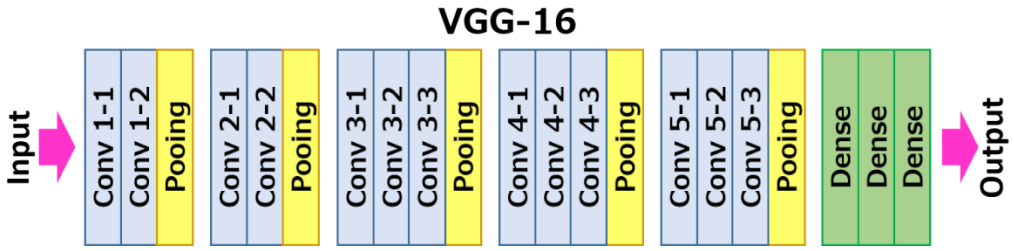
- **Max-pooling (MaxPool2d)**

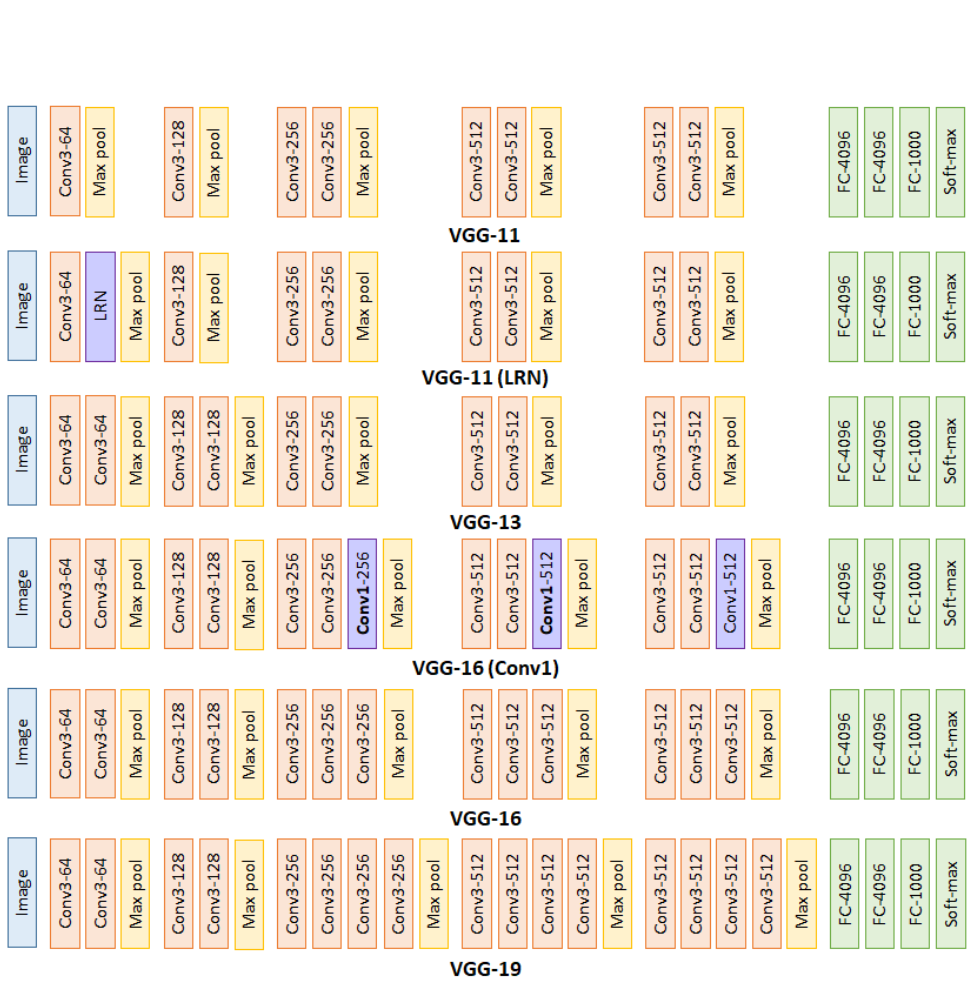


Famous CNNs

VGG11-19



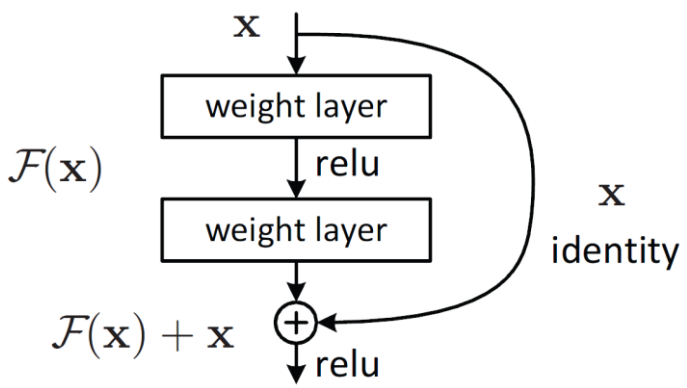


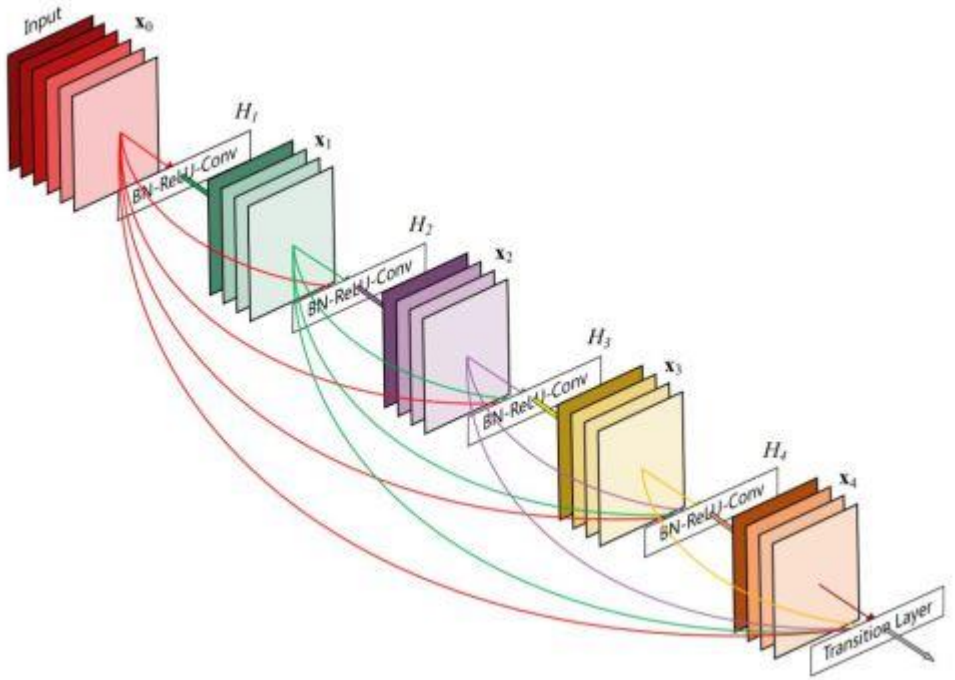


	Number of Parameters (millions)	Top-5 Error Rate (%)
VGG-11	133	10.4
VGG-11 (LRN)	133	10.5
VGG-13	133	9.9
VGG-16 (Conv1)	134	9.4
VGG-16	138	8.8
VGG-19	144	9.0

Famous CNNs

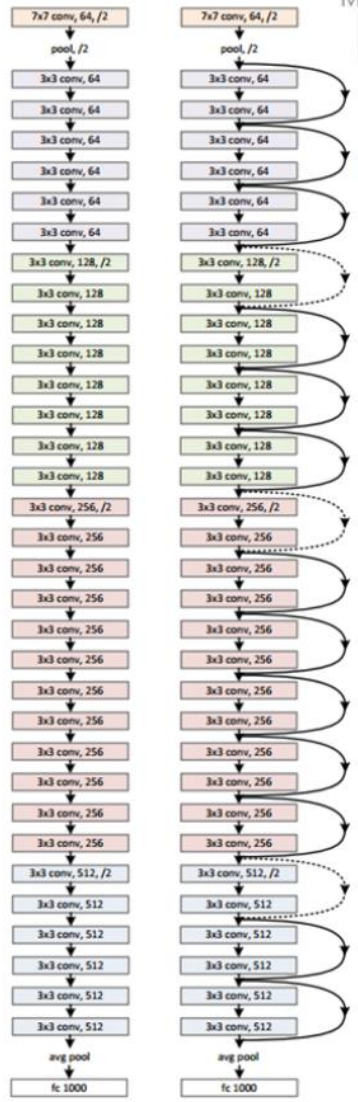
ResNet





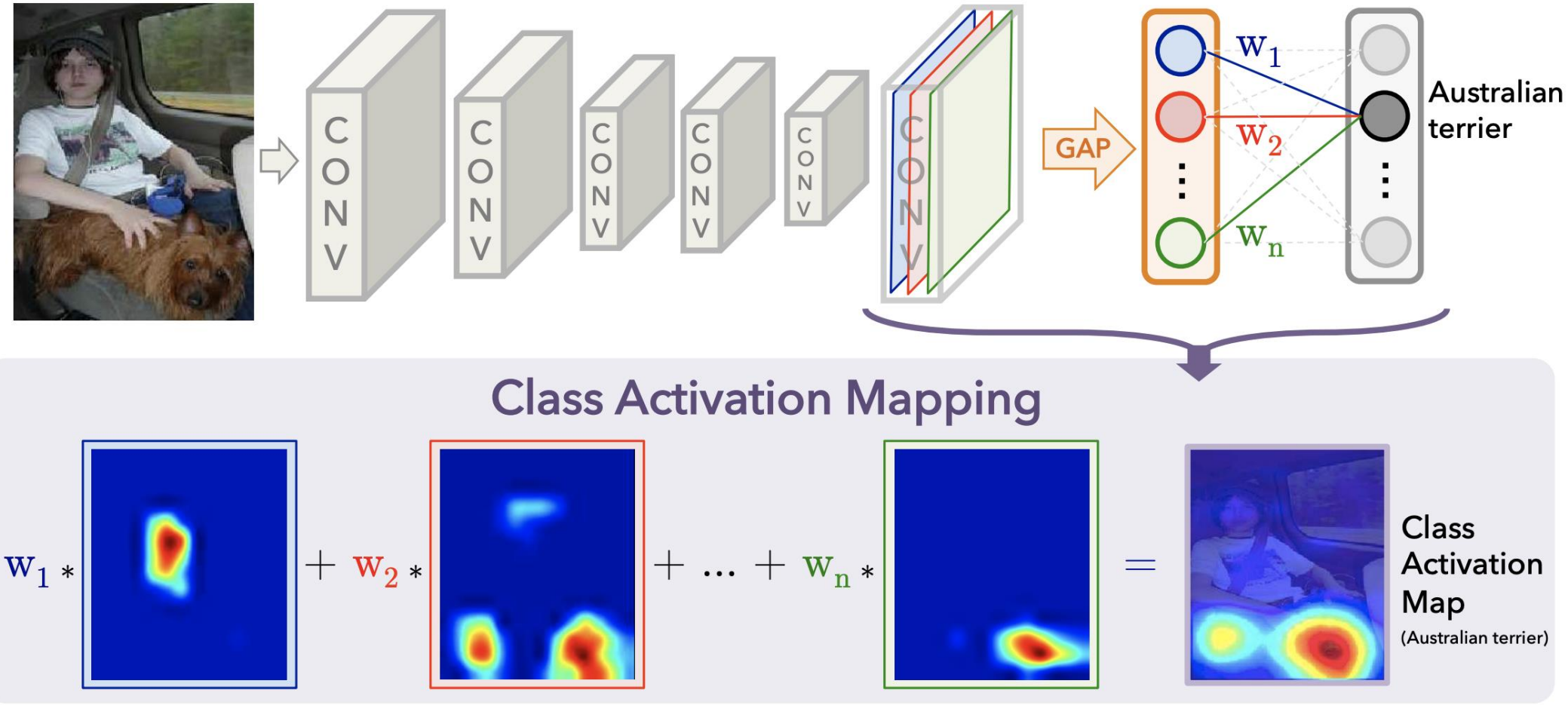
plain net

ResNet



Microsoft Research

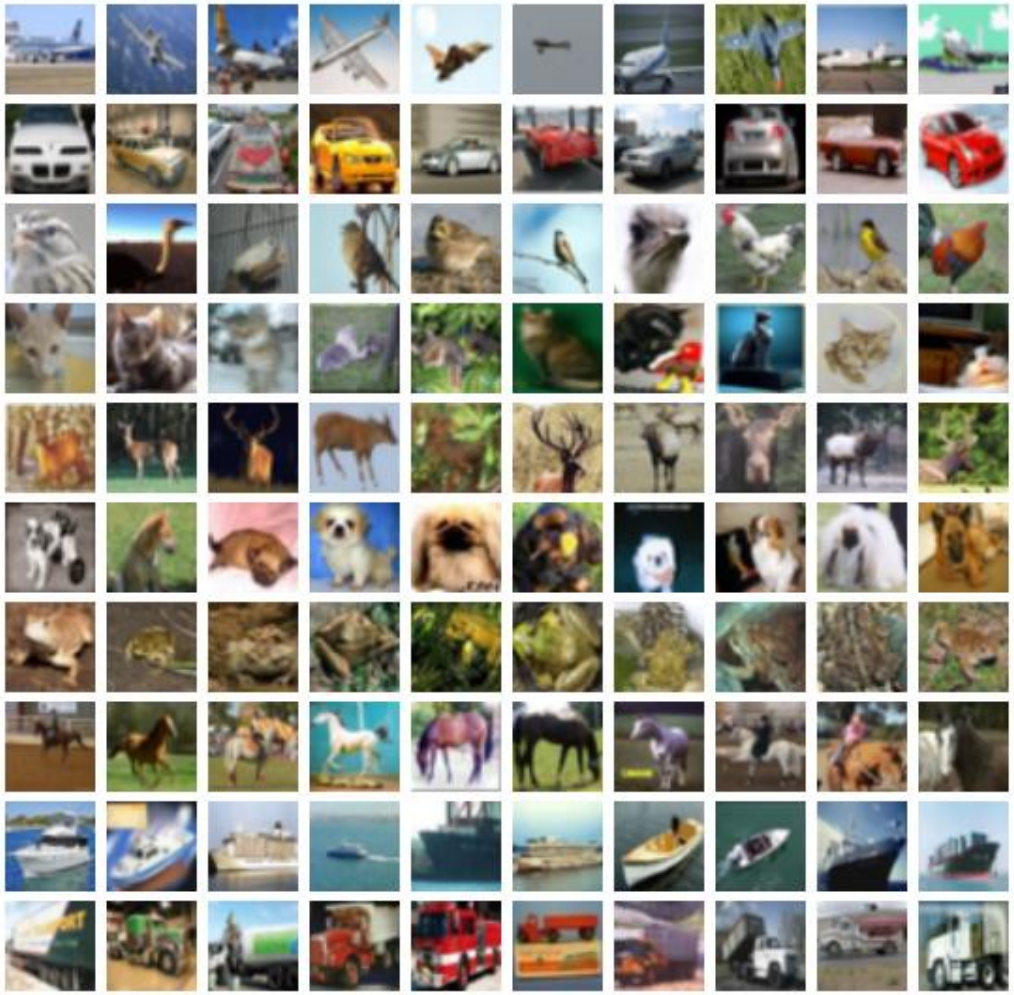
Class Activation Map



CIFAR 10 Dataset

10 classification dataset by Canadian Institute For Advanced Research

비행기
자동차
새
고양이
사슴
개
개구리
말
배
트럭



32X32 RGB image
50000 training data pairs
10000 test data pairs

Today we will upscale this 32x32 image to 224x224 RGB image

It’s coding time

Let’s fill the I.P.Y.N.B

Preparation

Base .ipynb :

https://git.io/aibd-cnn-student

Required Python Packages :

- matplotlib numpy scikit-learn **scikit-image**
- torch torchvision **torchsummary**

1. Lib & Dev Prep.

Already filled by TA

```
# You don't need to edit this section today.
import gc
import matplotlib.pyplot as plt
import numpy as np
import time
import torch
import torch.nn as nn
import torchvision.transforms as transforms

from IPython.display import clear_output
from skimage.transform import resize as OVResize
from torch.cuda import memory_allocated, empty_cache
from torch.optim import Adam
from torch.utils.data import DataLoader, random_split
from torchsummary import summary as Summary
from torchvision.datasets import CIFAR10
from torchvision.transforms import Compose, ToTensor, Normalize, RandomCrop, RandomHorizontalFlip, \
                                   ToPILImage, Resize, Grayscale

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'{"CPU" if device == "cpu" else "GPU"} will be used in training/validation.')
```

Today, we will use a lot of preprocessing techniques (torchvision.transforms)

2. Hyperparams

No Model-related Hyper Parameters Today!

```
# Data Load
input_norm_mean = (.4914, .4822, .4465)
input_norm_std = (.2023, .1994, .2010)
batch_size = 16

# Learning
logging_dispfig = True
maximum_epoch = 15
learning_rate = 1e-3
```

- input_norm_mean and input_norm_std normalizes R/G/B channel to make various images be looked similarly.
- Those values are fit to CIFAR-10, calculated based on computer vision theory.

- Today's batch size is small (16), because of the CUDA memory problem.

2. Hyperparams

No Model-related Hyper Parameters Today!

```
# Load dataset into python variable
input_transform = transforms.Compose([
  Resize(224),
  RandomCrop(224, padding=24),
  RandomHorizontalFlip(),
  ToTensor(),
  Normalize(mean=input_norm_mean, std=input_norm_std),
])

train_data = CIFAR10("./", train=True, transform=input_transform, download=True)
train_data, valid_data = random_split(train_data, [45000, 5000])
test_data = CIFAR10("./", train=False, transform=input_transform, download=False)
```

- Today, we will use CNN model based on VGG11. VGG11 receives images with size of 224x224, so in same way, we will convert our 32x32 CIFAR10 images into 224x224 images. (“Resize(224)”)
- Then we will apply data augmentation technique for more generic model. (“RandomCrop”, “RandomHorizontalFlip”)
- Then we will convert our data onto Tensor and will apply RGB normalizing as I mentioned in previous slide.

3. Data Load/Prepp.

Dataset loading

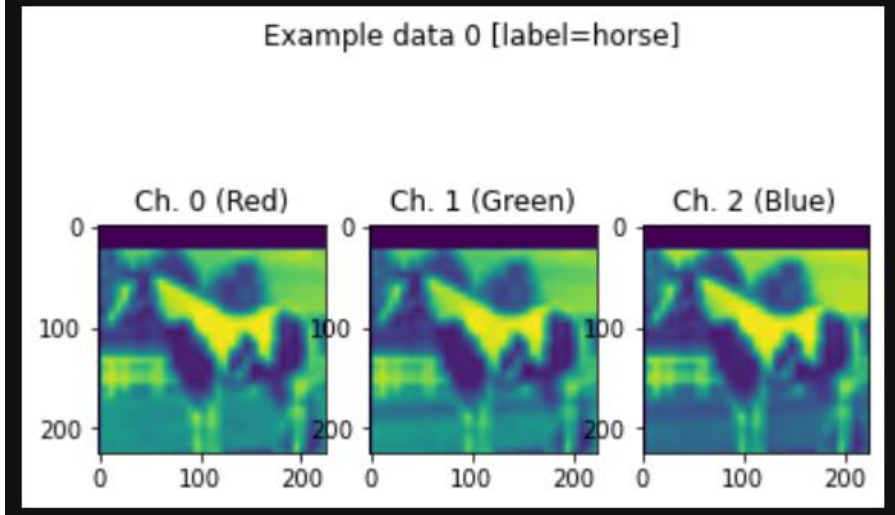
```
# Check the data
print(f'Train dataset length = {len(train_data)}')
print(f'Valid dataset length = {len(valid_data)}')
print(f'Test dataset length = {len(test_data)}\n')

print(f'Classes = {test_data.classes}\n')

train_0_x, train_0_y = train_data[0]
print(f'Label ({type(train_0_y).__name__}) = {train_0_y} ({test_data.classes[train_0_y]})')
print(f'Data ({type(train_0_x).__name__}) = {train_0_x.shape}\n')

fig, (ax1, ax2, ax3) = plt.subplots(1, 3)
fig.suptitle(f'Example data 0 [label={test_data.classes[train_0_y]}]')
ax1.imshow(train_0_x[0, :, :])
ax1.set_title('Ch. 0 (Red)')
ax2.imshow(train_0_x[1, :, :])
ax2.set_title('Ch. 1 (Green)')
ax3.imshow(train_0_x[2, :, :])
ax3.set_title('Ch. 2 (Blue)')
plt.show()
```

- These code will show you R/G/B channel of an example image, in separated.

Dataset loading

Expected Result

```
Train dataset length = 45000
Valid dataset length = 5000
Test dataset length = 10000

Classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

Label (int) = 7 (horse)
Data (Tensor) = torch.Size([3, 224, 224])
```



- Look. As we applied RandomCrop, the upper part of the horse image is cropped! (empty section colored as purple).

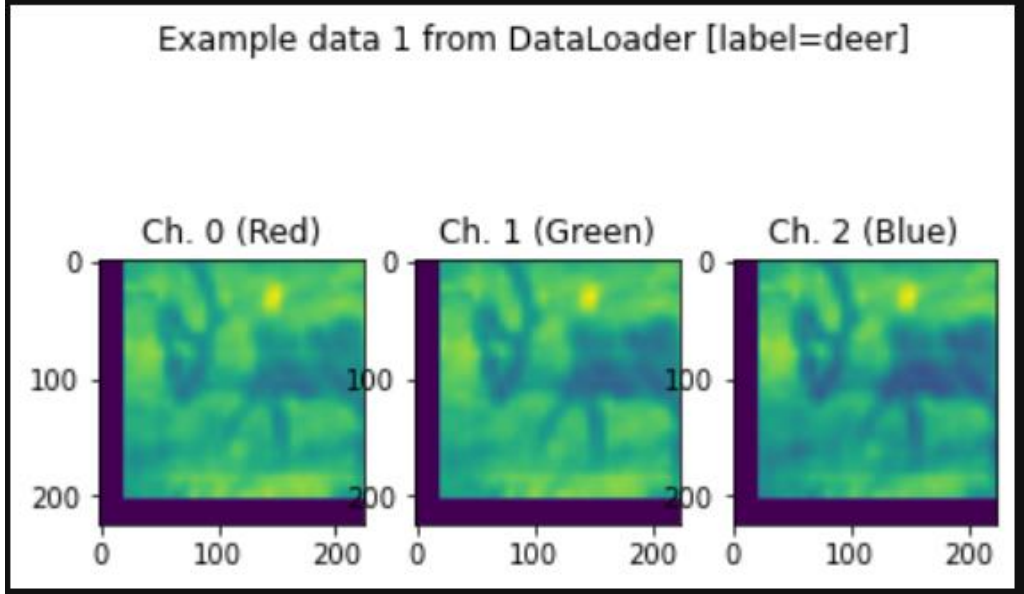
DataLoader Wrapping

```
# Examine the data loader
train_enumerator = enumerate(train_loader)
ex_batch_idx, (ex_data, ex_label) = next(train_enumerator)
print(f'Idx: {ex_batch_idx} / X.shape = {ex_data.shape} / Y.shape = {ex_label.shape}\n')
print(f'Y[0:{batch_size}] = {ex_label.tolist()}')
print(f'→ Label = {np.array(test_data.classes)[ex_label]}\n')

preview_index = 1
fig, (ax1, ax2, ax3) = plt.subplots(1, 3)
fig.suptitle(f'Example data {preview_index} from DataLoader [label={test_data.classes[ex_label[preview_index]]}]')
ax1.imshow(ex_data[preview_index, 0, :, :])
ax1.set_title('Ch. 0 (Red)')
ax2.imshow(ex_data[preview_index, 1, :, :])
ax2.set_title('Ch. 1 (Green)')
ax3.imshow(ex_data[preview_index, 2, :, :])
ax3.set_title('Ch. 2 (Blue)')
plt.show()
```

DataLoader Wrapping

Expected Result

```
Idx: 0 / X.shape = torch.Size([16, 3, 224, 224]) / Y.shape = torch.Size([16])

Y[0:16] = [7, 4, 3, 5, 6, 6, 2, 0, 1, 9, 0, 7, 7, 0, 3, 2]
→ Label = ['horse' 'deer' 'cat' 'dog' 'frog' 'frog' 'bird' 'airplane' 'automobile'
 'truck' 'airplane' 'horse' 'horse' 'airplane' 'cat' 'bird']
```



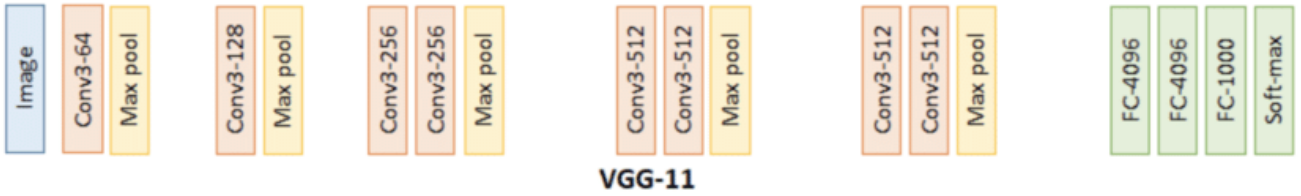
4. Fn. Definitions

Model Definition

```
def init_model():
  global net, loss_fn, optim
  net = MyLittleCNN().to(device)
  loss_fn = nn.CrossEntropyLoss()
  optim = Adam(net.parameters(), lr=learning_rate)
```

- As we did yesterday, we will define our model in class structure.
- Loss function is still crossentropy
- Today, we will use “Adam” optimizer, which is widely used these days instead of SGD.

4. Fn. Definitions



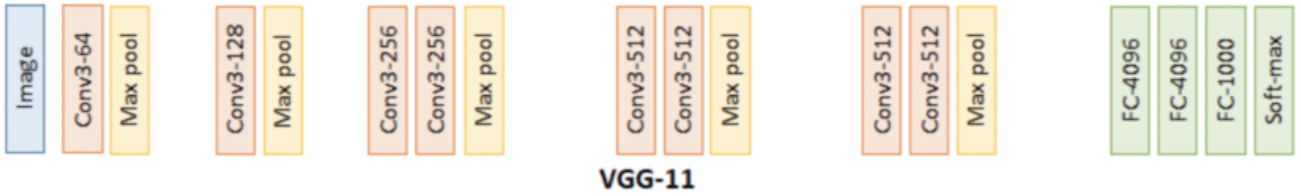
Model Definition

```
def __init__(self):
  super(MyLittleCNN, self).__init__()
  self.convolution_part = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=3, padding=1), nn.BatchNorm2d(64), nn.LeakyReLU(0.2),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(64, 128, kernel_size=3, padding=1), nn.BatchNorm2d(128), nn.LeakyReLU(0.2),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(128, 256, kernel_size=3, padding=1), nn.BatchNorm2d(256), nn.LeakyReLU(0.2),
    nn.Conv2d(256, 256, kernel_size=3, padding=1), nn.BatchNorm2d(256), nn.LeakyReLU(0.2),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(256, 512, kernel_size=3, padding=1), nn.BatchNorm2d(512), nn.LeakyReLU(0.2),
    nn.Conv2d(512, 512, kernel_size=3, padding=1), nn.BatchNorm2d(512), nn.LeakyReLU(0.2),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(512, 512, kernel_size=3, padding=1), nn.BatchNorm2d(512), nn.LeakyReLU(0.2),
    nn.Conv2d(512, 512, kernel_size=3, padding=1), nn.BatchNorm2d(512), nn.LeakyReLU(0.2),
    nn.MaxPool2d(kernel_size=2, stride=2)
  )
  self.channelavg_part = nn.AvgPool2d(7)
  self.classifier_part = nn.Linear(512, 10, bias=False)
```

- Compare top-left VGG11 image and above code.

4. Fn. Definitions

Model Definition




```python
def forward(self, data):
    conv_out = self.convolution_part(data)
    avg_out = self.channelavg_part(conv_out)
    avg_out_flatten = avg_out.reshape(avg_out.size(0), -1)
    classifier_out = self.classifier_part(avg_out_flatten)
    return classifier_out, conv_out
```

- Forward function only connects “convolution_part -> channelavg_part -> classifier_part”, with reshaping output of channelavg part and taking away convolution part output for purpose of drawing CAM.

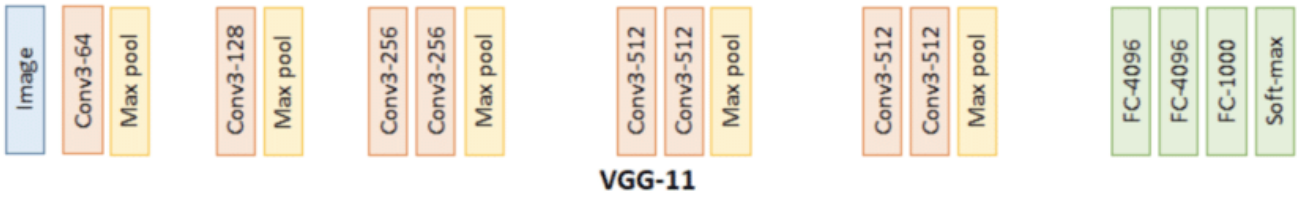
4. Fn. Definitions

Model Definition

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
       BatchNorm2d-2         [-1, 64, 224, 224]             128
         LeakyReLU-3         [-1, 64, 224, 224]               0
         MaxPool2d-4         [-1, 64, 112, 112]               0
            Conv2d-5        [-1, 128, 112, 112]          73,856
       BatchNorm2d-6        [-1, 128, 112, 112]             256
         LeakyReLU-7        [-1, 128, 112, 112]               0
         MaxPool2d-8          [-1, 128, 56, 56]               0
            Conv2d-9          [-1, 256, 56, 56]         295,168
      BatchNorm2d-10          [-1, 256, 56, 56]             512
        LeakyReLU-11          [-1, 256, 56, 56]               0
           Conv2d-12          [-1, 256, 56, 56]         590,080
      BatchNorm2d-13          [-1, 256, 56, 56]             512
        LeakyReLU-14          [-1, 256, 56, 56]               0
        MaxPool2d-15          [-1, 256, 28, 28]               0
           Conv2d-16          [-1, 512, 28, 28]       1,180,160
      BatchNorm2d-17          [-1, 512, 28, 28]           1,024
        LeakyReLU-18          [-1, 512, 28, 28]               0
           Conv2d-19          [-1, 512, 28, 28]       2,359,808
      BatchNorm2d-20          [-1, 512, 28, 28]           1,024
        LeakyReLU-21          [-1, 512, 28, 28]               0
        MaxPool2d-22          [-1, 512, 14, 14]               0
           Conv2d-23          [-1, 512, 14, 14]       2,359,808
      BatchNorm2d-24          [-1, 512, 14, 14]           1,024
        LeakyReLU-25          [-1, 512, 14, 14]               0
           Conv2d-26          [-1, 512, 14, 14]       2,359,808
      BatchNorm2d-27          [-1, 512, 14, 14]           1,024
        LeakyReLU-28          [-1, 512, 14, 14]               0
        MaxPool2d-29            [-1, 512, 7, 7]               0
        AvgPool2d-30            [-1, 512, 1, 1]               0
           Linear-31                  [-1, 10]           5,120
================================================================
Total params: 9,231,104
Trainable params: 9,231,104
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 181.65
Params size (MB): 35.21
Estimated Total Size (MB): 217.44
----------------------------------------------------------------
```

Expected Result

- Compare left TorchSummary output and below VGG 11 block diagram.



VGG-11

4. Fn. Definitions

GPU Memory Cleaner

```
# Memory cleaner to prevent CUDA out of memory error
def clear_memory():
    if device != 'cpu':
        empty_cache()
    gc.collect()
```

- Today's model is based on VGG and VGG is very memory-intensive model. So we're defining GPU/CPU useless cache memory cleaning function.

4. Fn. Definitions

Epoch Function

```python
def init_epoch():
  global epoch_cnt
  epoch_cnt = 0


def epoch(data_loader):
  # One epoch : gets data_loader as input and returns loss / accuracy, and
  #             last prediction value / its label(truth) value for future use
  global epoch_cnt
  iter_loss, iter_acc = [], []

  last_grad_performed = False

  # Mini-batch iterations
  for _data, _label in data_loader:
    data, label = _data.to(device), _label.to(device)

    # 1. Feed-forward
    onehot_out, _ = net(data)

    # 2. Calculate accuracy
    _, out = torch.max(onehot_out, 1)
    acc_partial = (out == label).float().sum()
    acc_partial = acc_partial / len(label)
    iter_acc.append(acc_partial.item())

    # 3. Calculate loss
    loss = loss_fn(onehot_out, label)
    iter_loss.append(loss.item())

    # 4. Backward propagation if not in `torch.no_grad()`
    if onehot_out.requires_grad:
      optim.zero_grad()
      loss.backward()
      optim.step()
      last_grad_performed = True

  # Up epoch count if backward propagation is done
  if last_grad_performed:
    epoch_cnt += 1

  # Clear memory to prevent CUDA memory error
  clear_memory()

  return np.average(iter_loss), np.average(iter_acc)


def epoch_not_finished():
  # For now, let's repeat training fixed times, e.g. 25 times.
  # We will learn how to determine training stop or continue later.
  return epoch_cnt < maximum_epoch
```

Filled by TA

- Same as yesterday.

4. Fn. Definitions

Logging Function

```python
# Logging (same with day 2)
def init_log():
  global log_stack, iter_log, tloss_log, tacc_log, vloss_log, vacc_log, time_log
  iter_log, tloss_log, tacc_log, vloss_log, vacc_log = [], [], [], [], []
  time_log, log_stack = [], []


def record_train_log(_tloss, _tacc, _time):
  # Push time, training loss, training accuracy, and epoch count into lists
  time_log.append(_time)
  tloss_log.append(_tloss)
  tacc_log.append(_tacc)
  iter_log.append(epoch_cnt)


def record_valid_log(_vloss, _vacc):
  # Push validation loss and validation accuracy into each list
  vloss_log.append(_vloss)
  vacc_log.append(_vacc)


def last(log_list):
  # Get the last member of list. If empty, return -1.
  if len(log_list) > 0: return log_list[len(log_list) - 1]
  else: return -1


def print_log():
  # Generate log string and put it into log stack
  log_str = f'Iter: {last(iter_log):>4d} >> T_loss {last(tloss_log):<8.5f}   ' \
          + f'T_acc {last(tacc_log):<6.5f}   V_loss {last(vloss_log):<8.5f}   ' \
          + f'V_acc {last(vacc_log):<6.5f}   🕒 {last(time_log):5.3f}s'
  log_stack.append(log_str)

  # Draw figure if want
  if logging_dispfig:
    hist_fig, loss_axis = plt.subplots(figsize=(10, 3), dpi=99)
    hist_fig.patch.set_facecolor('white')

    # Draw loss lines
    loss_t_line = plt.plot(iter_log, tloss_log, label='Train Loss', color='#FF9999', marker='o')
    loss_v_line = plt.plot(iter_log, vloss_log, label='Valid Loss', color='#99B0FF', marker='s')
    loss_axis.set_xlabel('epoch')
    loss_axis.set_ylabel('loss')

    # Draw accuracy lines
    acc_axis = loss_axis.twinx()
    acc_t_line = acc_axis.plot(iter_log, tacc_log, label='Train Acc.', color='#FF0000', marker='+')
    acc_v_line = acc_axis.plot(iter_log, vacc_log, label='Valid Acc.', color='#003AFF', marker='x')
    acc_axis.set_ylabel('accuracy')

    # Append annotations
    hist_lines = loss_t_line + loss_v_line + acc_t_line + acc_v_line
    loss_axis.legend(hist_lines, [l.get_label() for l in hist_lines])
    loss_axis.grid()
    plt.title(f'Learning history until epoch {last(iter_log)}')
    plt.draw()

  # Print log
  clear_output(wait=True)
  if logging_dispfig: plt.show()
  for idx in reversed(range(len(log_stack))):
    print(log_stack[idx])
```

Filled by TA

- Same as yesterday.

5. Training Iter.

Yeah

```
# Training Initialization (same with day 2)
init_model()
init_epoch()
init_log()

# Training Iteration (similar to day 2)
while epoch_not_finished():
  start_time = time.time()
  tloss, tacc = epoch(train_loader)
  end_time = time.time()
  time_taken = end_time - start_time
  record_train_log(tloss, tacc, time_taken)
  with torch.no_grad():
    vloss, vacc = epoch(valid_loader)
    record_valid_log(vloss, vacc)
  print_log()

print('\n Training completed!')
```

Filled by TA

- Same as yesterday.

5. Training Iter.

Yeah

```
Iter:   15 >> T_loss 0.39752   T_acc 0.86513   V_loss 0.44669   V_acc 0.84565   96.090s
Iter:   14 >> T_loss 0.41736   T_acc 0.85895   V_loss 0.45768   V_acc 0.83746   96.118s
Iter:   13 >> T_loss 0.43601   T_acc 0.85220   V_loss 0.45802   V_acc 0.84285   96.044s
Iter:   12 >> T_loss 0.46626   T_acc 0.84173   V_loss 0.47743   V_acc 0.83407   96.128s
Iter:   11 >> T_loss 0.48864   T_acc 0.83253   V_loss 0.50081   V_acc 0.83087   96.179s
Iter:   10 >> T_loss 0.52613   T_acc 0.82101   V_loss 0.53792   V_acc 0.81510   95.971s
Iter:    9 >> T_loss 0.55588   T_acc 0.81041   V_loss 0.55347   V_acc 0.81130   96.406s
Iter:    8 >> T_loss 0.60472   T_acc 0.79523   V_loss 0.57173   V_acc 0.80232   96.224s
Iter:    7 >> T_loss 0.64848   T_acc 0.77869   V_loss 0.64989   V_acc 0.77656   96.172s
Iter:    6 >> T_loss 0.70538   T_acc 0.75727   V_loss 0.67705   V_acc 0.76957   96.181s
Iter:    5 >> T_loss 0.79176   T_acc 0.72648   V_loss 0.71801   V_acc 0.75120   96.270s
Iter:    4 >> T_loss 0.89577   T_acc 0.68794   V_loss 0.81597   V_acc 0.70747   96.073s
Iter:    3 >> T_loss 1.04208   T_acc 0.63111   V_loss 0.95706   V_acc 0.66014   96.143s
Iter:    2 >> T_loss 1.23487   T_acc 0.56203   V_loss 1.14193   V_acc 0.59505   96.072s
Iter:    1 >> T_loss 1.72277   T_acc 0.36240   V_loss 1.37549   V_acc 0.50459   95.061s

 Training completed!
```

- Beautiful graph.

Expected Result

6. Result Analysis

Accuracy

```
# Accuracy for test dataset (similar to day 2)
with torch.no_grad():
  test_loss, test_acc = epoch(test_loader)
  print(f'Test accuracy = {test_acc}\nTest loss = {test_loss}')
```

```
Test accuracy = 0.8406
Test loss = 0.4767277945458889
```

Expected Result

6. Result Analysis

Class Activation Map

```
# Data preparation (load test data again, but without data augmentation)
cam_data_iterable = CIFAR10(root='./', train=False, download=False)
cam_data = list(cam_data_iterable)
```

- Now we're trying to draw class activation map (CAM).
- Because we want draw the original picture and overlay CAM on it, first load image data without any transformations (first line).
- Then make it as a python list (second line)

Class Activation Map

For the full picture of “Drawing CAM” code,
Look at here : https://i.imgur.com/tlhLuS7.png
Also check for “#comments” the description.

Class Activation Map

```
# Draw CAMs
cam_figidx = 1
cam_random_order = np.arange(len(cam_data), dtype=int)
np.random.shuffle(cam_random_order)

fig = plt.figure(figsize=(23, 47))
for class_idx, class_label in enumerate(cam_data_iterable.classes):
  # We will draw 5 CAMs for each classes
  class_axes = [
    fig.add_subplot(10, 5, cam_figidx),
    fig.add_subplot(10, 5, cam_figidx + 1),
    fig.add_subplot(10, 5, cam_figidx + 2),
    fig.add_subplot(10, 5, cam_figidx + 3),
    fig.add_subplot(10, 5, cam_figidx + 4)
  ]
  cam_figidx = cam_figidx + 5

  # Find 5 proper data with a specified "label"
  class_figidx = 0
```

Class Activation Map

```
  # We will draw 5 CAMs for each classes
  class_axes = [
    fig.add_subplot(10, 5, cam_figidx),
    fig.add_subplot(10, 5, cam_figidx + 1),
    fig.add_subplot(10, 5, cam_figidx + 2),
    fig.add_subplot(10, 5, cam_figidx + 3),
    fig.add_subplot(10, 5, cam_figidx + 4)
  ]
  cam_figidx = cam_figidx + 5

  # Find 5 proper data with a specified "label"
  class_figidx = 0
  for random_order in cam_random_order:
    data, label = cam_data[random_order]
    if label == class_idx:
      # 1. Preprocess original image to put it into the model
      temp_input = ToTensor()(Resize(224)(data))
      temp_input = Normalize(mean=input_norm_mean, std=input_norm_std)(temp_input)
      temp_input = temp_input.reshape((1, 3, 224, 224)).to(device)

      # 2. Put the data into model and get convolutional part output and classifier output
      with torch.no_grad():
        temp_classifier_out, temp_conv_out = net(temp_input)
      _, temp_predicted_class = torch.max(temp_classifier_out, 1)

      # 3. If the predicted label is same with the truth, draw CAM of it
      if temp_predicted_class == label:
```

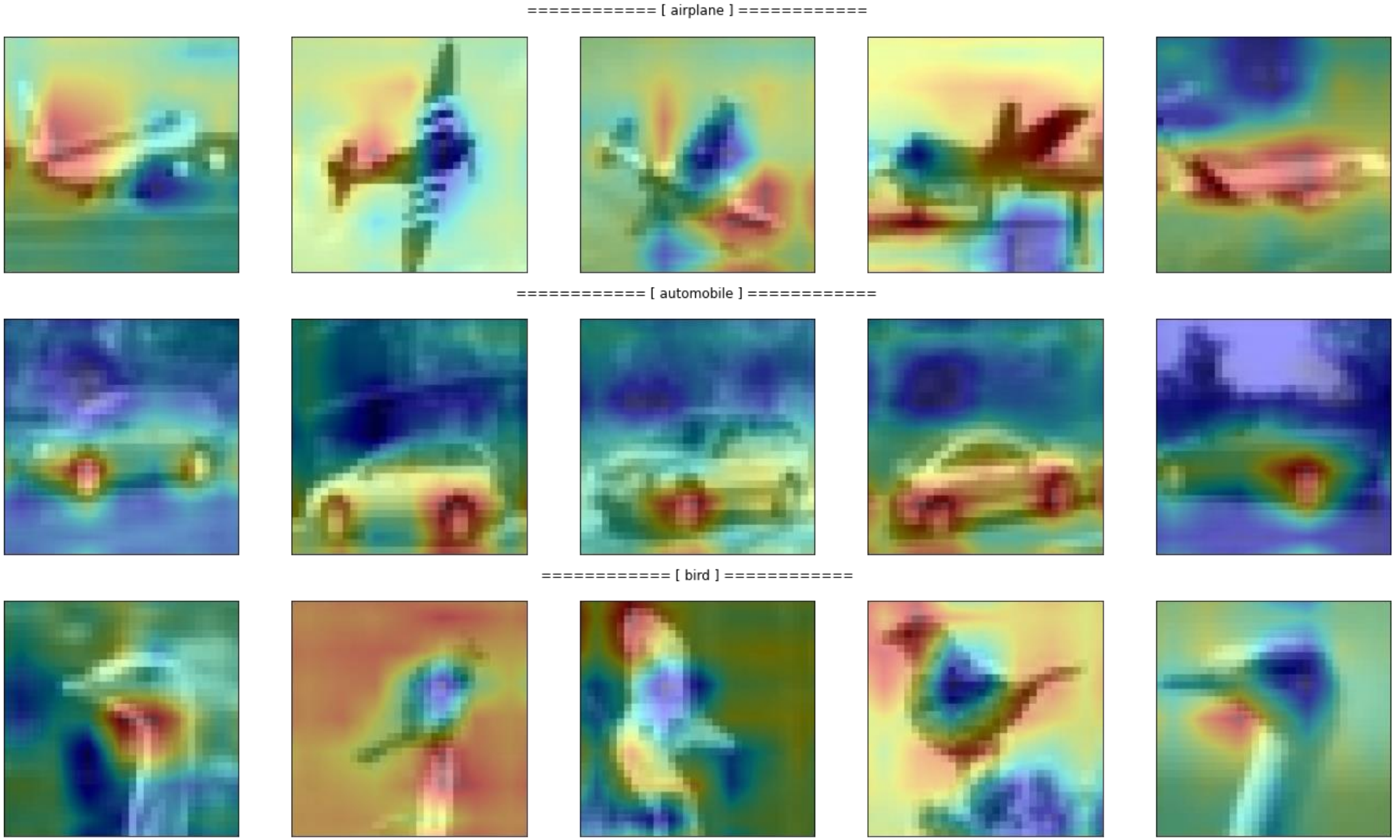
Class Activation Map

```
    temp_input = temp_input.reshape((1, 3, 224, 224)).to(device)

    # 2. Put the data into model and get convolutional part output and classifier output
    with torch.no_grad():
      temp_classifier_out, temp_conv_out = net(temp_input)
    _, temp_predicted_class = torch.max(temp_classifier_out, 1)

    # 3. If the predicted label is same with the truth, draw CAM of it
    if temp_predicted_class == label:
      conv_class_weight = list(net.parameters())[-1].squeeze()[label]
      temp_conv_out = temp_conv_out.reshape(512, 49)
      cam = conv_class_weight.matmul(temp_conv_out).reshape(7, 7).cpu().data.numpy()
      cam = cam - np.min(cam)
      cam = cam / np.max(cam)
      class_axes[class_figidx].imshow(
        Grayscale()(Resize((512, 512), interpolation=0)(data)),
        cmap='gray',
      )
      class_axes[class_figidx].imshow(OVResize(cam, [512, 512]), cmap='jet', alpha=0.4)
      class_axes[class_figidx].set_xticks([], minor=False)
      class_axes[class_figidx].set_yticks([], minor=False)
      if class_figidx == 2:
        class_axes[class_figidx].set_title(f'\n============ [ {class_label} ] ============\n')
      class_figidx = class_figidx + 1
      if class_figidx >= 5:
        break

  # Tiny error message
```

6. Result Analysis

Class Activation Map

```
        if temp_predicted_class == label:
          conv_class_weight = list(net.parameters())[-1].squeeze()[label]
          temp_conv_out = temp_conv_out.reshape(512, 49)
          cam = conv_class_weight.matmul(temp_conv_out).reshape(7, 7).cpu().data.numpy()
          cam = cam - np.min(cam)
          cam = cam / np.max(cam)
          class_axes[class_figidx].imshow(
            Grayscale()(Resize((512, 512), interpolation=0)(data)),
            cmap='gray',
          )
          class_axes[class_figidx].imshow(OVResize(cam, [512, 512]), cmap='jet', alpha=0.4)
          class_axes[class_figidx].set_xticks([], minor=False)
          class_axes[class_figidx].set_yticks([], minor=False)
          if class_figidx == 2:
            class_axes[class_figidx].set_title(f'\n============ [ {class_label} ] ============\n')
          class_figidx = class_figidx + 1
          if class_figidx >= 5:
            break

    # Tiny error message
    if class_figidx != 5:
      print(f'Failed to find proper data for {class_label}[{class_idx}].')

plt.show()
```



Expected Result

You can check the full version of today's .ipynb at

https://git.io/aibd-cnn-full

Early Stopping

Just see it and code

Overfitting problem

- When the model is too fit to the training dataset so when the performance for non-training data degrades, we call it “overfitted”.

- So how can we prevent it? One way is “early stopping”.

Earlystopping

- **Code**
  - https://git.io/aibd-early-stop

Just for testing purpose

```
# Load dataset into python variable
input_transform = transforms.Compose([
    RandomCrop(32, padding=4),
    RandomHorizontalFlip(),
    ToTensor(),
    Normalize(mean=input_norm_mean, std=input_norm_std),
])

train_data = CIFAR10("./", train=True, transform=input_transform, download=True)
train_data, valid_data = random_split(train_data, [2000, 48000])
valid_data, trash_data = random_split(valid_data, [2000, 46000])
test_data = CIFAR10("./", train=False, transform=input_transform, download=False)
```

Earlystopping

- **Earlystopping can be done with two mechanism.**
  - Wait for “patience” times of no validation loss decreasing.
  - If the era of no validation loss decreasing continues for “patience” times, then stop the training.
  - After stopping the training, set the parameters of network as its best parameter (who shown the lowest validation loss).

Earlystopping

- **Earlystopping can be done with two mechanism.**

```
def init_epoch():
    global epoch_cnt, earlystop_cnt
    epoch_cnt = 0
    earlystop_cnt = 0
```

```
def epoch_not_finished():
    return earlystop_cnt < earlystop_patience
```

```
# Earlystopping
def copy_weights(src, dst):
    dst.load_state_dict(src.state_dict())
```

```
# Training Iteration
while epoch_not_finished():
    start_time = time.time()
    tloss, tacc = epoch(train_loader)
    end_time = time.time()
    time_taken = end_time - start_time
    record_train_log(tloss, tacc, time_taken)
    with torch.no_grad():
        vloss, vacc = epoch(valid_loader)
        if len(vloss_log) and np.min(vloss_log) < vloss:
            earlystop_cnt = earlystop_cnt + 1
        else:
            earlystop_cnt = 0
            copy_weights(net, best_net)
        record_valid_log(vloss, vacc)
    print_log()

copy_weights(best_net, net)
```

Earlystopping

- **You can get two advantages from earlystopping**
  - You don’t have to wait for a long time for meaningless training.
  - You can prevent overfitting.

- **But sometimes earlystopped model has a poor performance than more-trained model.**
  - So, just use earlystopping when you taking aim about the appropriate epoch count.