COSE474 Deep Learning

Project #2: CNN Architecture Implementation

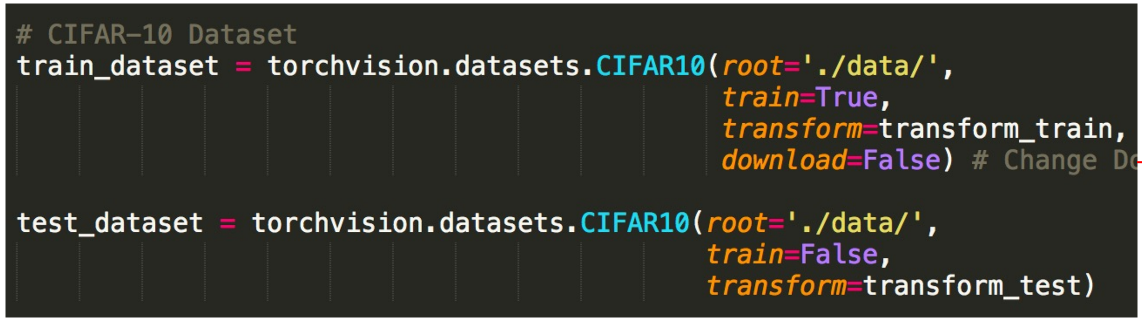
Seungryong Kim

Computer Vision Lab. (CVLAB)
Department of Computer Science and Engineering
Korea University

How to Train Models Using PyTorch

- **Train**

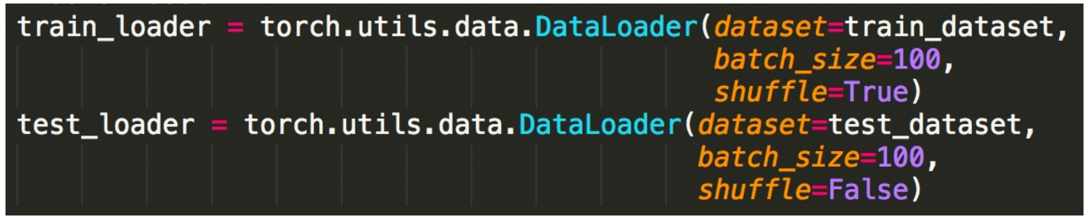
- Get CIFAR-10 Dataset



```
# CIFAR-10 Dataset
train_dataset = torchvision.datasets.CIFAR10(root='./data/',
                                             train=True,
                                             transform=transform_train,
                                             download=False) # Change Do

test_dataset = torchvision.datasets.CIFAR10(root='./data/',
                                            train=False,
                                            transform=transform_test)
```

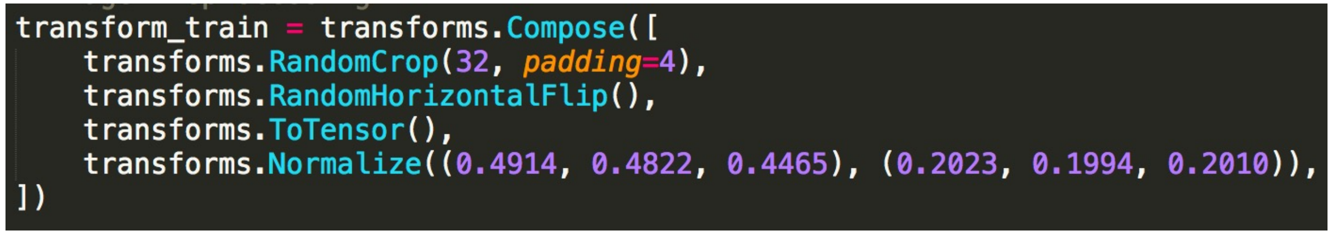
You have to change *download = True* at the first excution

- Load data using DataLoader()



```
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=100,
                                          shuffle=False)
```

- Data augmentation and preprocessing using transform function



```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

Random Crop
Random HorizontalFlip
Convert image to Tensor
Image Normalization

You can execute it at once!

How to Train Models Using PyTorch

- Set device

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

Check if device has GPU & CUDA is installed using torch.cuda.is_available()

- Choose model

```
model = ResNet(ResidualBlock, [3, 4, 6]).to(device)
```

```
model = vgg16().to(device)
```

- Choose Loss and optimizer

```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

- Loss : get difference between ground truth and predicted output

➢ nn.L1loss : get element-wise absolute difference $\ell(x,y) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = |x_n - y_n|,$

➢ nn.MSELoss : get mean squared error or squared L2 norm $\ell(x,y) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = (x_n - y_n)^2,$

➢ nn.CrossEntropyLoss : get Cross Entropy Loss, contains nn.LogSoftmax() and nn.NLLLoss()

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

➢ nn.NLLLoss : get Negative log likelihood loss $\ell(x,y) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$

➢ nn. BCELoss : get Binary Cross Entropy loss $\ell(x,y) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = -w_n\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right],$

How to Train Models Using PyTorch

- Set model to train mode

```
model.train()
```

- Get both index and data using enumerate()

```
for batch_index, (images, labels) in enumerate(train_loader):
```

- Get predicted output by putting input into the model, and calculate loss by criterion()

```
# Forward pass
outputs = model(images)
loss = criterion(outputs, labels)
```

- Set gradients to zero and conduct backpropagation by loss.backward()

```
# Backward and optimize
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

$\rightarrow$ we need to *set the gradients to zero* before starting to do backpropragation because PyTorch *accumulates the gradients on subsequent backward passes*

- Set batch size

```
batch_size=100,
```

$\rightarrow$ Number of training examples utilized in one iteration

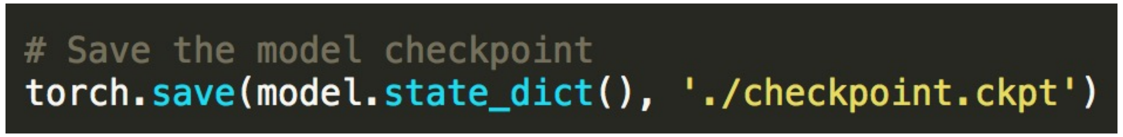
- Learning rate decay

```
if (epoch + 1) % 20 == 0:
    current_lr /= 3
    update_lr(optimizer, current_lr)
```

$\rightarrow$ Decrease the learning rate in order to reach the optimal point

How to Train Models Using PyTorch

- Save/Load model



```
# Save the model checkpoint
torch.save(model.state_dict(), './checkpoint.ckpt')
```

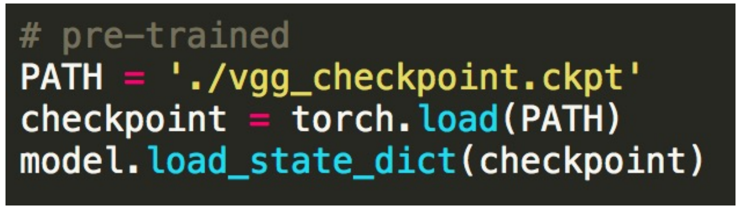
- *torch.save(obj, f, pickle_module=<module 'pickle' from '/opt/conda/lib/python3.6/pickle.py'>, pickle_protocol=2)*
  - *obj* : saved object
  - *f* : a file-like object or a string containing a file name
  - e.g.)
    1. torch.save(model.state_dict(), PATH)
    2. torch.save({'epoch': epoch,
       'model_state_dict': model.state_dict(),
       'optimizer_state_dict': optimizer.state_dict(),
       'loss': loss}, PATH)

① : Only save model's parameters

② : Save checkpoint file with other variables

How to Train Models Using PyTorch

- Save/Load model



```
# pre-trained
PATH = './vgg_checkpoint.ckpt'
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint)
```

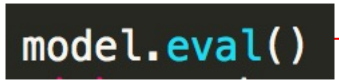
- *torch.load(f, map_location=None, pickle_module=<module 'pickle' from '/opt/conda/lib/python3.6/pickle.py'>, **pickle_load_args)*
  - *f* : a file-like object or a string containing a file name
  - e.g.)
    ① model.load_state_dict(torch.load(PATH))
    ② checkpoint = torch.load(PATH)
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    epoch = checkpoint['epoch']
    loss = checkpoint['loss']

① : If saved check point only include model's parameters
② : Load variables in saved checkpoint file

How to Test Models Using PyTorch

- **Test**

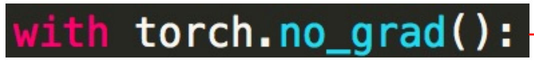
- Set model to evaluation mode



```
model.eval()
```

notify all your layers that you are in eval mode,
batchnorm or dropout layers will work in eval mode

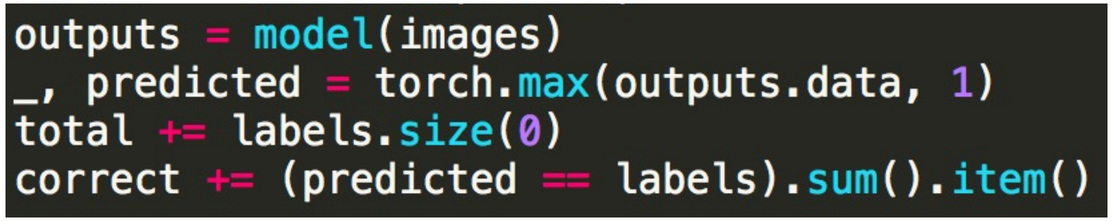
- torch.no_grad() impacts the autograd engine and deactivate it



```
with torch.no_grad():
```

It will reduce memory usage and speed up computations
but you won't be able to backprop (which you don't want in an eval script).

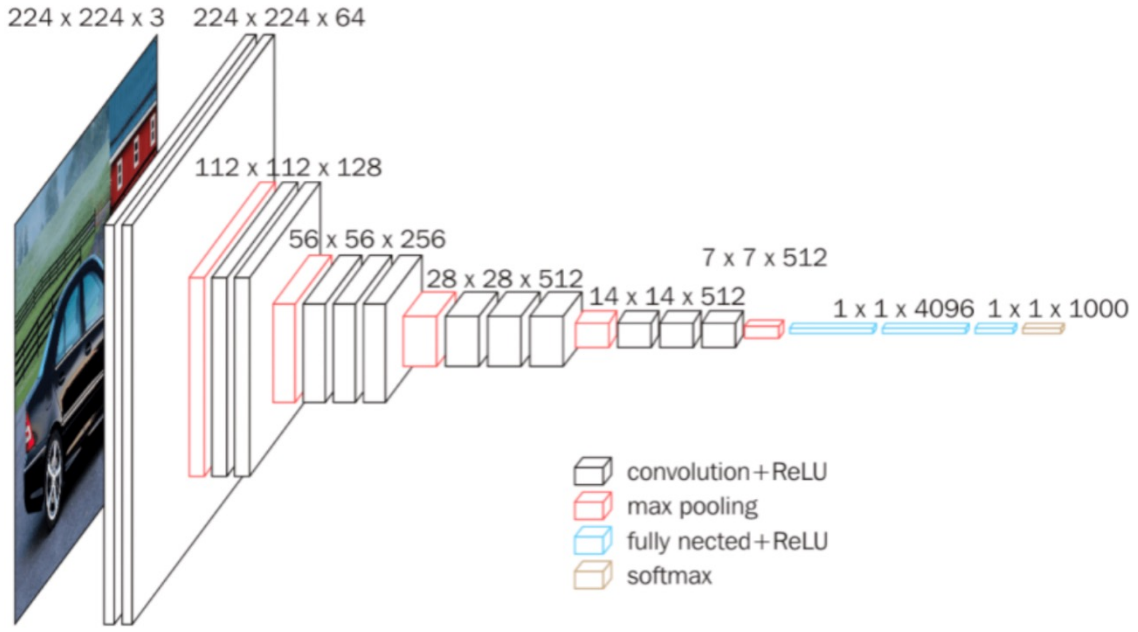
- You don't backpropagate loss in test

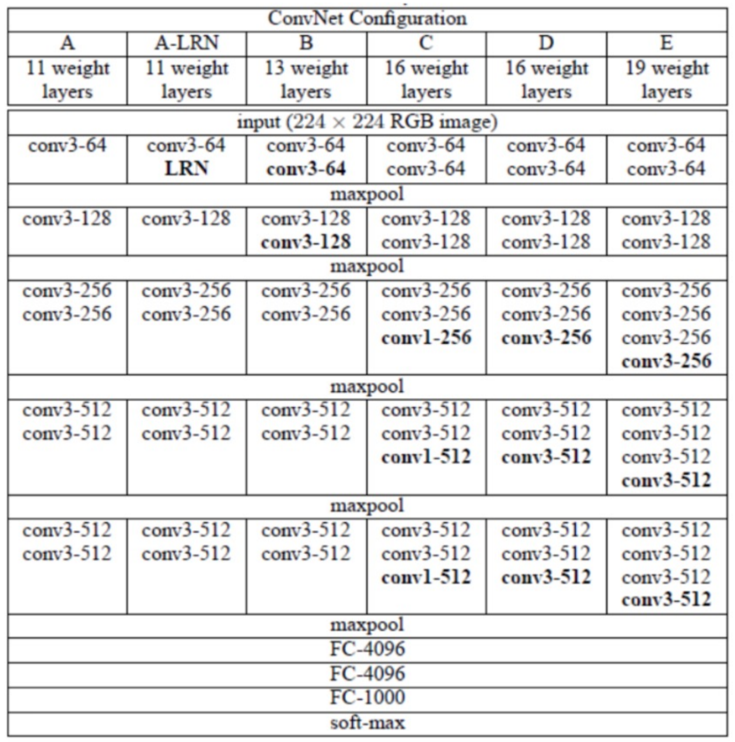


```
outputs = model(images)
_, predicted = torch.max(outputs.data, 1)
total += labels.size(0)
correct += (predicted == labels).sum().item()
```

VGG Networks- *Practice*

- **Train “VGG-16”**
  - Network Architecture



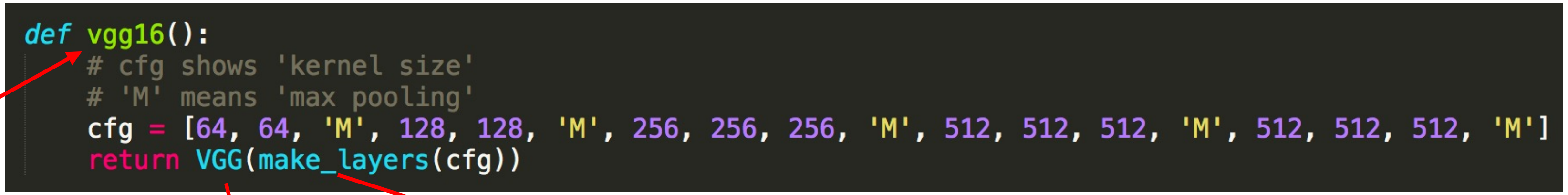


ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	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					

1. The nets are referred to their names (A-E)
2. All configurations follow the generic design present in architecture and differ only in the depth: from 11 weight layers in the network A (8 conv. and 3 FC layers) to 19 weight layers in the network E (16 conv. and 3 FC layers).
3. The width of conv. layers (the number of channels) is rather small, starting from 64 in the first layer and then increasing by a factor of 2 after each max-pooling layer, until it reaches 512.

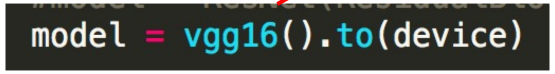
VGG Networks- *Practice*

- **Train “VGG-16”**
  - Code Flow



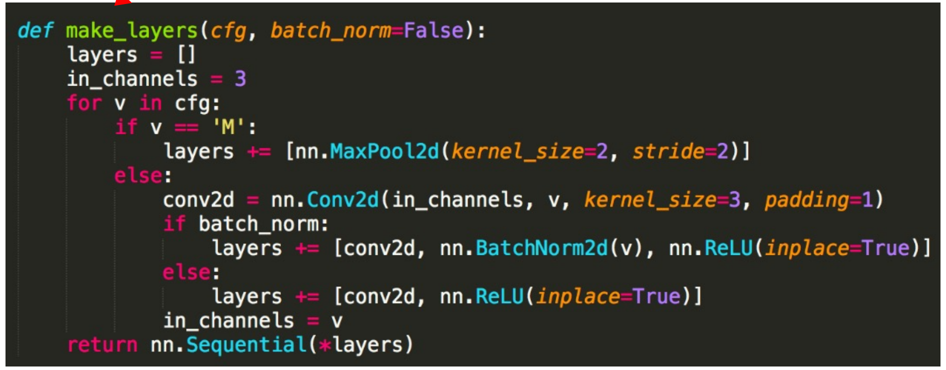
```python
def vgg16():
    # cfg shows 'kernel size'
    # 'M' means 'max pooling'
    cfg = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']
    return VGG(make_layers(cfg))
```

VGG(make_layers()) is called



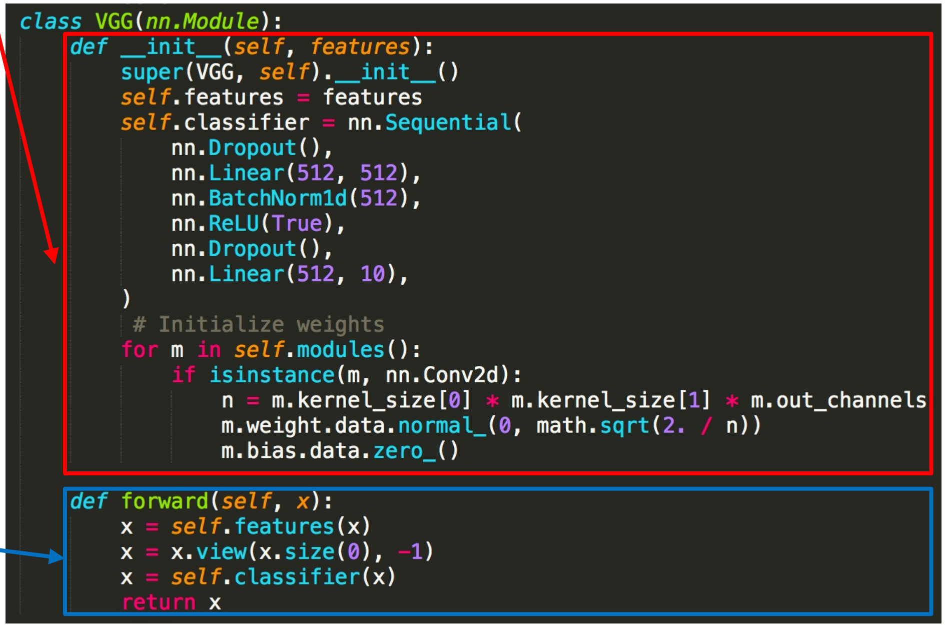
```python
model = vgg16().to(device)
```

Initialize



```python
def make_layers(cfg, batch_norm=False):
    layers = []
    in_channels = 3
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
```

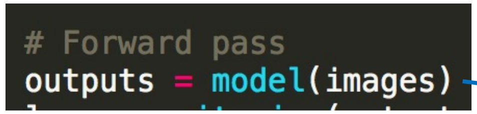
‘M’ : maxpool,
Else : Convolution + ReLu



```python
class VGG(nn.Module):
    def __init__(self, features):
        super(VGG, self).__init__()
        self.features = features
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(512, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(512, 10),
        )
         # Initialize weights
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                m.bias.data.zero_()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

At initialization time,
__init__ is called

Forward



```python
# Forward pass
outputs = model(images)
```

At Network forward,
forward() is called

VGG Networks- *Practice*

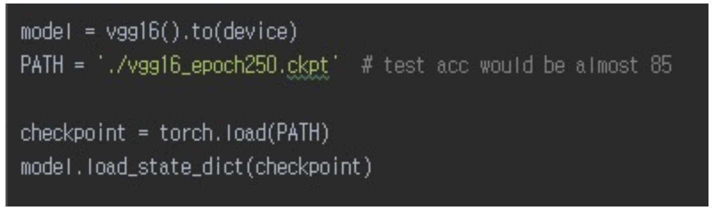
- **Train “VGG-16”**
  - Train “VGG-16” model with “CIFAR-10” datasets
    - Optimize parameters with *Adam optimizer* and *cross Entropy Loss*
      - ■ Use “VGG-16” model with torch.nn library
      - ■ Get “CIFAR-10” Dataset with torchvision library
    - Procedure

      1) Load the trained model (which is given)

      2) Train it with CPU or GPU, and screen capture the test accuracy.

      3) You can use a trained checkpoint parameters of 250 epochs. You will train model only 1 epoch.

      ** Please Understand VGG class in “vgg16_full.py”.



```
model = vgg16().to(device)
PATH = './vgg16_epoch250.ckpt'  # test acc would be almost 85

checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint)
```

VGG Networks- *Practice*

- **Train "VGG-16"**
  - Optimizer
    - *torch.optim.Adam*(params, lr, momentum)*
      - ➢ *params* : iterable of parameters to optimize or dicts defining parameter groups
      - ➢ *lr* : learning rate
      - ➢ *momentum* : momentum factor (default : 0)
  - Loss
    - *torch.nn.CrossEntropyLoss*(weight, size_average, ignore_index, reduce, reduction)*
      - ➢ *weight* : a manual rescaling weight given to each class
      - ➢ *reduction* : specifies the reduction to apply to the output ('none' | 'mean' | 'sum')

Residual Networks

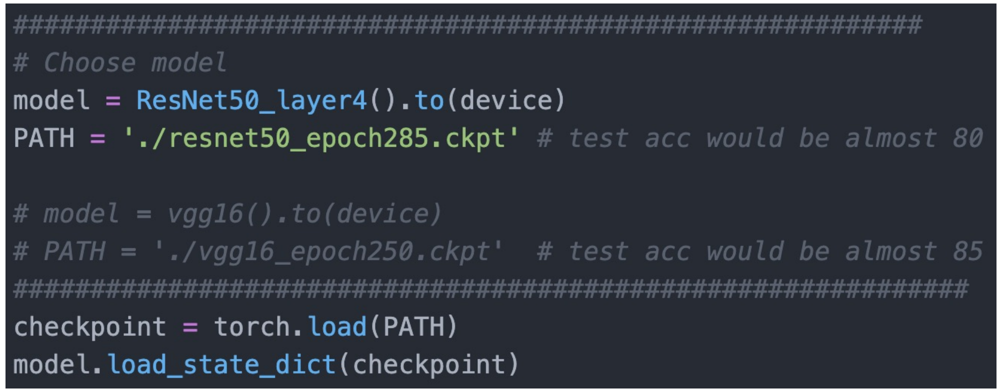
- **Implement “ResNet-50”**
  - Train “ResNet-50” model with “CIFAR-10” datasets
    - Optimize parameters with *Adam optimizer* and *cross Entropy Loss*
      - Get “CIFAR-10” dataset with torchvision library
    - Procedure

      1) Load the trained model (which is given)

      2) Complete the *class ResNet50_layer4* in *“resnet50_skeleton.py”* .

      3) Train it with CPU or GPU and submit the screen capture of test accuracy as a result.

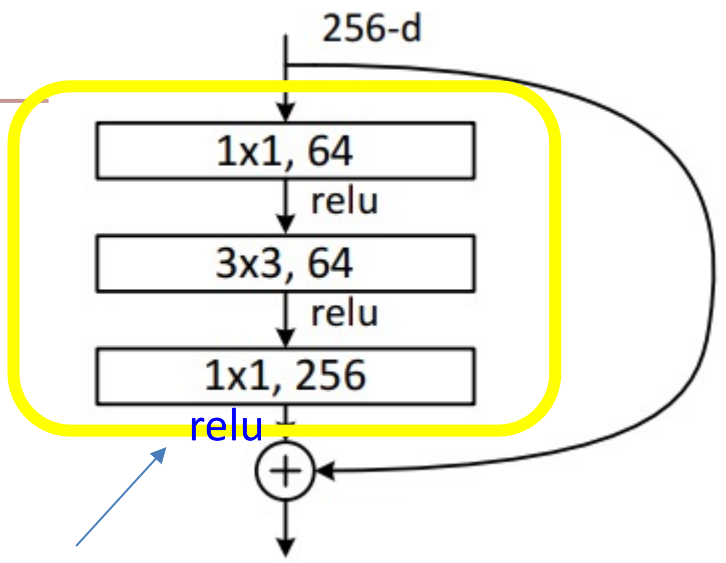
      3) You can use a trained checkpoint parameters of 285 epochs.
      You will train model only 1 epoch.



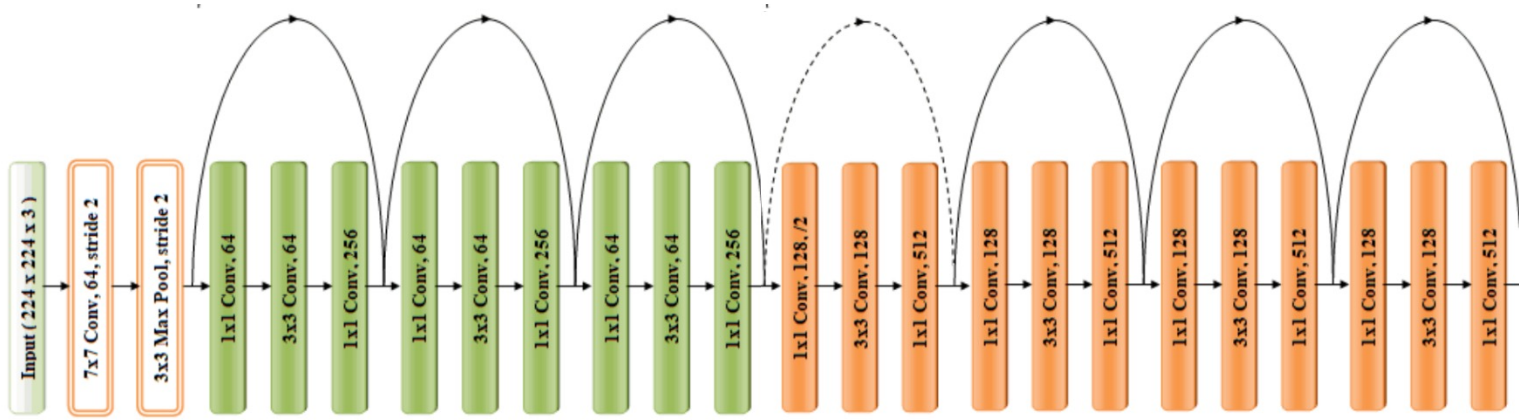
```
############################################################
# Choose model
model = ResNet50_layer4().to(device)
PATH = './resnet50_epoch285.ckpt' # test acc would be almost 80

# model = vgg16().to(device)
# PATH = './vgg16_epoch250.ckpt'  # test acc would be almost 85
###############################################################
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint)
```

Residual Networks

- **Implement “ResNet-50”**
  - ✓ **Question 1:** Implement the “bottleneck building block”
  - Bottleneck building block (residual block)
    - For each residual function F, we use a stack of 3 layers. The three layers are 1x1, 3x3 and 1x1 convolutions.
    - Input: x
    - Output: $\mathrm{relu}(F(x) + x)$
    - *You can see how building blocks make up the network below. This diagram is a part of Resnet50 network.*



Note) This structure is different form the original Building block on the Resnet. In the original ResNet, “relu” is executed after ⊕ operation”.



Note) The homework codes are different from the original Resnet50 structure. When you do homework, please refer to the table on page 28, not left figure.

K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition.” In CVPR, 2016

Residual Networks

- **Implement "ResNet-50"**
  - ✓ **Question 2:** Implement the "ResNet50_layer4"
  - Network Architecture
    - It is different from the original ResNet50.
    - Complete the network code by looking at the table.
    - You can study the original ResNet50 network architecture from the link below.
    - Fill in the #blank# in the code.
    - There are 25 blanks in the code

Layer number	Network	Output Image size
Layer 1	7x7 conv, channel = 64, stride = 2 3x3 max pool, stride = 2	8 x 8
Layer 2	[1x1 conv, channel = 64, 3x3 conv, channel = 64, 1x1 conv, channel = 256] x 2 [1x1 conv, channel = 64, stride = 2 3x3 conv, channel = 64, 1x1 conv, channel = 256] x 1	4 x 4
Layer 3	[1x1 conv, channel = 128, 3x3 conv, channel = 128, 1x1 conv, channel = 512] x 3 [1x1 conv, channel = 128, stride = 2 3x3 conv, channel = 128, 1x1 conv, channel = 512] x 1	2 x 2
Layer 4	[1x1 conv, channel = 256, 3x3 conv, channel = 256, 1x1 conv, channel = 1024] x 6	2 x 2
	AvgPool	1 x 1
	Fully connected layer	?

Note) In layer 2 and 3, reduce an activation map in half by using the strided convolution (stride = 2) at the last Residual block.

K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition." In CVPR, 2016 (https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf)

Residual Networks

- **Implement "ResNet-50"**
  - torchvision provide some pre-trained(or not) models
    - *torchvision.models.resnet18(pretrained, progress, kwargs)*
      - ➢ *pretrained :* If True, returns a model pre-trained on ImageNet
      - ➢ *progress* : If True, displays a progress bar of the download to stderr
  - torchvision also provide some datasets(ImageNet, CIFAR-10, Pascal VOC, CityScapes, …)
    - *torchvision.datasets.CIFAR10(root, train, transform, target_transform, download)*
      - ➢ *root* : Root directory of dataset where directory "cifar-10-batches-py" exists or will be saved to if download is set to True
      - ➢ *train* : If True, creates dataset from training set, otherwise creates from test set
      - ➢ *transform* : A function/transform that takes in an PIL image and returns a transformed version
      - ➢ *target_transform* : A function/transform that takes in the target and transforms it
      - ➢ *download* : If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again

Residual Networks

- **Implement "ResNet-50"**
  - DataLoader
    - *torch.utils.data.DataLoader(dataset, batch_size, shuffle, sampler, batch_sampler, num_workers, collate_fn, pin_memory, drop_last, timeout, worker_init_fn)*
      - *dataset* : dataset from which to load the data
      - *batch_size* : how many samples per batch to load (default: 1)
      - *shuffle* : set to True to have the data reshuffled at every epoch (default: False)
      - *num_workers* : how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process (default: 0)
      - *drop_last* : set to True to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If False and the size of dataset is not divisible by the batch size, then the last batch will be smaller (default: False)

❖ " __get__item(index)" will return (image, target) where target is index of the target class

Residual Networks

- **Implement "ResNet-50"**
  - Optimizer
    - *torch.optim.Adam*(params, lr, momentum)*
      - ➢ *params* : iterable of parameters to optimize or dicts defining parameter groups
      - ➢ *lr* : learning rate
      - ➢ *momentum* : momentum factor (default : 0)
  - Loss
    - *torch.nn.CrossEntropyLoss*(weight, size_average, ignore_index, reduce, reduction)*
      - ➢ *weight* : a manual rescaling weight given to each class
      - ➢ *reduction* : specifies the reduction to apply to the output ('none' | 'mean' | 'sum')

CNN Architecture Implementation

Due on Nov. 15 (Mon.), 01:59 pm (in Blackboard)

(late policy: 25% off per a day late)

You must submit the **code** with the **report**.
(1 page with free format, including the description of your code, results, and discussions)

The report should be written in **English**.

Please do NOT copy your friends' and internet sources.

Please start your project EARLY.

Thank you!

Q & A