📖

[논문 리뷰] VGGNet (2014)

1. Introduction

🌟 VGGNet은 16 ~ 19 weight layer이다. 해당 모델은 3x3 convolution filter를 사용함으로서, depths를 증가시켰습니다.

VGGNet 논문에서는 ConvNet architecture의 depth를 중요하게 다룹니다. depth를 다루기 위하여,

Architecture의 parameter들을 변경하였고, **3x3 filter를 지속적으로 추가**함으로서,

layer의 깊이를 증가시켰습니다.

🌟 **결과적으로, 더 정확한 ConvNet구조를 구상하고, 다른 이미지 인식 dataset에도 적용가능하다.**

▼ 2. ConvNet Configurations

공정한 설정을 위해, ConvNet depth가 증가함에 따라, 정확한 evaluation을 위해,

ConvNet layer의 parameter를 동일하게 적용하였습니다.

2-1. Architecture

ConvNet에서는 (3, 224, 224) Image를 사용하였으며, 전처리 과정에서는 단지, **RGB value의 평균값을 training set의 각 pixel값 별로 substract**해주었습니다.

Image는 반복적인 3x3 filter가 쌓인, stack of convolutional layer을 통과합니다. 또한, Input channels 에 비선형성(linear transformation)을 적용하기 위해, 1x1 filter도 사용하였습니다.

대체적으로, stride=1을 적용하였으며, 공간 해상도(Spatial Resolution)를 보전하기 위해 padding을 적용하였습니다. 일부 Conv layer에는 Max-pooling(2x2, stride = 2)를 사용합니다.

Hidden layer에는 ReLU를 사용하였지만, AlexNet에서 사용한 Local Response Normalisation은 사용하지 않았습니다. 그러한 이유는, **memory 소비 및 computation time만 증가시키고, 성능향상에는 도움이 안되기 때문**입니다.

Conv layers이후에 3개의 FC가 존재하는데, 첫 번째, 두 번째는 4096 channels을 사용하고, 세 번째에서는 1000개의 channels을 사용하였습니다. 마지막 layer에서는 softmax를 사용하였습니다.

2-2. Configurations

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					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

2-3. Discussion

VGGNet에서는 3x3 filter 여러개를 전체 NeuralNet에 걸쳐 사용합니다.

3x3 filter 2개를 사용하면, 5x5의 receptive field의 성능을 가지고, 3개를 사용하면 7x7 receptive field를 가집니다.

For instance, a stack of three 3x3 conv. layers instead of single 7x7 layer?

First, 단일 layer를 사용하는 것보다, 3개의 non-linear을 사용함으로서, 더욱 차별화된 의사결정을 하는 것이 가능합니다.

Second, Parameters의 수가 감소합니다.

Second, we decrease the number of parameters: assuming that both the input and the output of a three-layer 3×3 convolution stack has C channels, the stack is parametrised by $3\left(3^2C^2\right) = 27C^2$ weights; at the same time, a single 7×7 conv. layer would require $7^2C^2 = 49C^2$ parameters, i.e. 81% more. This can be seen as imposing a regularisation on the 7×7 conv. filters, forcing them to have a decomposition through the 3×3 filters (with non-linearity injected in between).

▼ 3. Classification FrameWork

3-1. Training

Initialisation of the network weigths는 매우 중요합니다.

Random한 initialize로 훈련이 가능하게 끔 하였습니다. 4개의 ConvNet과 3개의 FC를 사용하여, 학습을 시작합니다. 중간 layer들의 값을 0 ~ 10^{-2}의 가중치 값으로 초기화하였다.

224*224의 이미지를 사용하기 위해서, training image내에서 무작위로 cropped 하였다. 더 나아가, random horizontal flipping 과 random RGB colour shift를 사용하였습니다.

▼ 4. Classification Experiments

First, VGGNet에서 local response normalisation(A-LRN network)은 모든 normalisation layer가 없는 model에서 개선되지 않습니다. 그러므로, B ~ E 에서는 정규화를 사용하지 않습니다.

Second, ConvNet 깊이가 증가함에 따라, Classification Error가 감소하는 것을 알 수 있습니다.

✨ Scale jittering Augmentation에 의한 training은 Multi-scale image를 capturing하는데 도움을 줍니다.

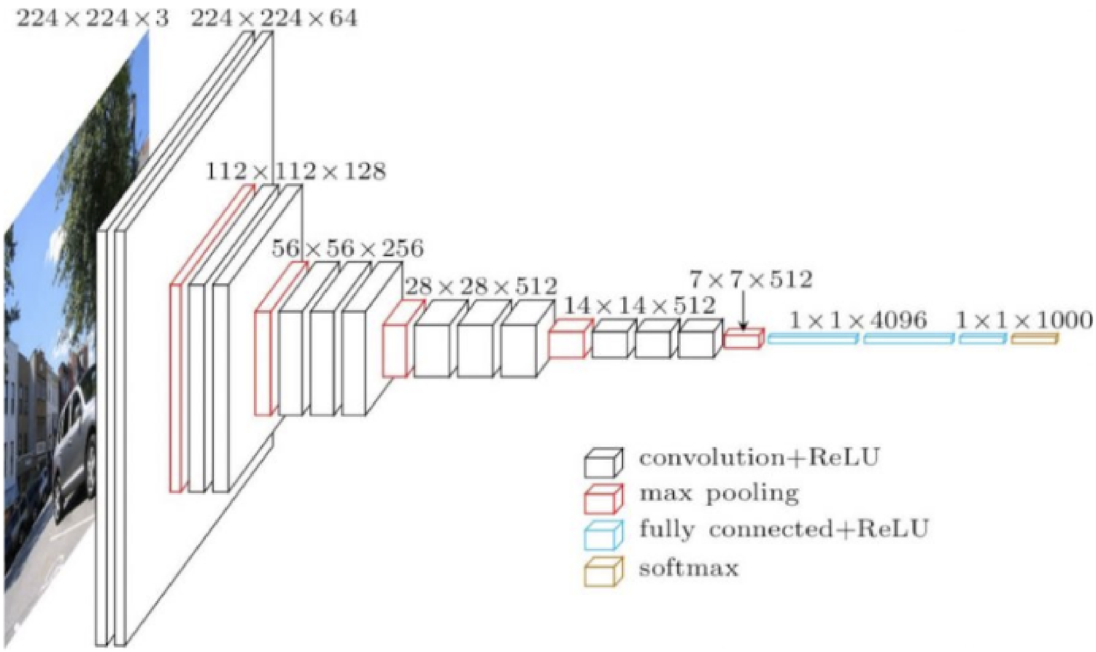
Table 3: **ConvNet performance at a single test scale.**

ConvNet config. (Table 1)	smallest image side train (S)	smallest image side test (Q)	top-1 val. error (%)	top-5 val. error (%)
A	256	256	29.6	10.4
A-LRN	256	256	29.7	10.5
B	256	256	28.7	9.9
C	256	256	28.1	9.4
	384	384	28.1	9.3
	[256;512]	384	27.3	8.8
D	256	256	27.0	8.8
	384	384	26.8	8.7
	[256;512]	384	25.6	8.1
E	256	256	27.3	9.0
	384	384	26.9	8.7
	[256;512]	384	**25.5**	**8.0**

5. Conclusion

Larage-scale image classification을 위해서 19 weights layers CNN을 활용하였습니다. Representation depth는 Classification Accuracy에서 매우 유익하며, SOTA에 달성하는데 도움을 준다는 것을 알 수 있었습니다.

▼ **VGG16 MODEL [Pytorch]**



VGG16 Architecture

center). In one of the configurations we also utilise 1×1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1 pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchsummary import summary

device = "cuda:0" if torch.cuda.is_available() else "cpu"

VGG_16 = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']

class VGG16(nn.Module):
    def __init__(self, in_channels, n_classes):
        super(VGG16, self).__init__()
        self.in_channels = in_channels

        self.feature_extractor = self.create_conv_layers(VGG_16)

        self.classification = nn.Sequential(
            nn.Linear(512*7*7, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, n_classes)
        )

    def forward(self, x):
        x = self.feature_extractor(x)
        x = x.view(-1, 512*7*7)
        x = self.classification(x)

        return x

    def create_conv_layers(self, architecture):
```

```
        layers = []
        in_channels = self.in_channels

        for x in architecture:
            if type(x) == int:
                out_channels = x

                layers += [nn.Conv2d(in_channels = in_channels, out_channels = out_channels,
                                    kernel_size = (3, 3), stride = (1, 1), padding = (1, 1)),
                          nn.BatchNorm2d(x),
                          nn.ReLU()]
                in_channels = x
            elif x == 'M':
                layers += [nn.MaxPool2d(kernel_size = (2, 2), stride = (2, 2))]

        return nn.Sequential(*layers)



model = VGG16(in_channels = 3, n_classes= 1000).to(device)
summary(model, input_size = (3, 224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
       BatchNorm2d-2         [-1, 64, 224, 224]             128
              ReLU-3         [-1, 64, 224, 224]               0
            Conv2d-4         [-1, 64, 224, 224]          36,928
       BatchNorm2d-5         [-1, 64, 224, 224]             128
              ReLU-6         [-1, 64, 224, 224]               0
         MaxPool2d-7         [-1, 64, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]          73,856
       BatchNorm2d-9        [-1, 128, 112, 112]             256
             ReLU-10        [-1, 128, 112, 112]               0
           Conv2d-11        [-1, 128, 112, 112]         147,584
      BatchNorm2d-12        [-1, 128, 112, 112]             256
             ReLU-13        [-1, 128, 112, 112]               0
        MaxPool2d-14          [-1, 128, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         295,168
      BatchNorm2d-16          [-1, 256, 56, 56]             512
             ReLU-17          [-1, 256, 56, 56]               0
           Conv2d-18          [-1, 256, 56, 56]         590,080
      BatchNorm2d-19          [-1, 256, 56, 56]             512
             ReLU-20          [-1, 256, 56, 56]               0
           Conv2d-21          [-1, 256, 56, 56]         590,080
      BatchNorm2d-22          [-1, 256, 56, 56]             512
             ReLU-23          [-1, 256, 56, 56]               0
        MaxPool2d-24          [-1, 256, 28, 28]               0
           Conv2d-25          [-1, 512, 28, 28]       1,180,160
      BatchNorm2d-26          [-1, 512, 28, 28]           1,024
             ReLU-27          [-1, 512, 28, 28]               0
           Conv2d-28          [-1, 512, 28, 28]       2,359,808
      BatchNorm2d-29          [-1, 512, 28, 28]           1,024
             ReLU-30          [-1, 512, 28, 28]               0
           Conv2d-31          [-1, 512, 28, 28]       2,359,808
      BatchNorm2d-32          [-1, 512, 28, 28]           1,024
             ReLU-33          [-1, 512, 28, 28]               0
        MaxPool2d-34          [-1, 512, 14, 14]               0
           Conv2d-35          [-1, 512, 14, 14]       2,359,808
      BatchNorm2d-36          [-1, 512, 14, 14]           1,024
             ReLU-37          [-1, 512, 14, 14]               0
           Conv2d-38          [-1, 512, 14, 14]       2,359,808
      BatchNorm2d-39          [-1, 512, 14, 14]           1,024
             ReLU-40          [-1, 512, 14, 14]               0
           Conv2d-41          [-1, 512, 14, 14]       2,359,808
      BatchNorm2d-42          [-1, 512, 14, 14]           1,024
             ReLU-43          [-1, 512, 14, 14]               0
        MaxPool2d-44            [-1, 512, 7, 7]               0
           Linear-45                 [-1, 4096]     102,764,544
             ReLU-46                 [-1, 4096]               0
          Dropout-47                 [-1, 4096]               0
           Linear-48                 [-1, 4096]      16,781,312
             ReLU-49                 [-1, 4096]               0
          Dropout-50                 [-1, 4096]               0
           Linear-51                 [-1, 1000]       4,097,000
================================================================
Total params: 138,365,992
Trainable params: 138,365,992
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 321.95
Params size (MB): 527.82
Estimated Total Size (MB): 850.35
----------------------------------------------------------------
```

▼ Baseline [Pytorch]

```
import torch
import torch.nn as nn
from torch import optim
from torch.optim.lr_scheduler import StepLR

from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import os
```

```
from torchvision import utils
import matplotlib.pyplot as plt
%matplotlib inline

import numpy as np
from datetime import datetime
import copy
```

```
path2data = 'data'
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((224, 224)),
    transforms.Normalize(RGB_mean, RGB_std)
])
train_dataset = datasets.STL10(path2data, split ='train',
                               download = True, transform = transform)
valid_dataset = datasets.STL10(path2data, split = 'test',
                               download = True, transform = transform)
```

```
meanRGB = [np.mean(x.numpy(), axis = (1, 2)) for x, _ in train_dataset]
stdRGB = [np.std(x.numpy(), axis = (1, 2)) for x, _ in train_dataset]

meanR = np.mean([m[0] for m in meanRGB])
meanG = np.mean([m[1] for m in meanRGB])
meanB = np.mean([m[2] for m in meanRGB])

stdR = np.mean([s[0] for s in stdRGB])
stdG = np.mean([s[1] for s in stdRGB])
stdB = np.mean([s[2] for s in stdRGB])

RGB_mean = [meanR, meanG, meanB]
RGB_std = [stdR, stdG, stdB]
```

```
train_dataloader = DataLoader(train_dataset, batch_size = 4, shuffle = True)
valid_dataloader = DataLoader(valid_dataset, batch_size = 4, shuffle = False)
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
# from torchsummary import summary

device = "cuda:0" if torch.cuda.is_available() else "cpu"

VGG_16 = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']

class VGG16(nn.Module):
    def __init__(self, in_channels, n_classes):
        super(VGG16, self).__init__()
        self.in_channels = in_channels

        self.feature_extractor = self.create_conv_layers(VGG_16)

        self.classification = nn.Sequential(
            nn.Linear(512*7*7, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, n_classes)
        )

    def forward(self, x):
```

```
        x = self.feature_extractor(x)
        x = x.view(-1, 512*7*7)
        x = self.classification(x)

        return x

    def create_conv_layers(self, architecture):
        layers = []
        in_channels = self.in_channels

        for x in architecture:
            if type(x) == int:
                out_channels = x

                layers += [nn.Conv2d(in_channels = in_channels, out_channels = out_channels,
                                     kernel_size = (3, 3), stride = (1, 1), padding = (1, 1)),
                          nn.BatchNorm2d(x),
                          nn.ReLU()]
                in_channels = x
            elif x == 'M':
                layers += [nn.MaxPool2d(kernel_size = (2, 2), stride = (2, 2))]

        return nn.Sequential(*layers)



model = VGG16(in_channels = 3, n_classes= 10).to(device)
# summary(model, input_size = (3, 224, 224))
```

```
def get_accuracy(model, data_loader, device):
    correct_pred = 0
    n = 0

    with torch.no_grad():
        model.eval()
        for X, y_true in data_loader:
            X, y_true = X.to(device), y_true.to(device)
            y_pred = model(X)

            _, predicted_labels = torch.max(y_pred, 1)

            n += y_true.size(0)
            correct_pred += (predicted_labels == y_true).sum()

    return correct_pred.float() / n
```

```
def train(train_loader, model, criterion, optimizer, device):
    model.train()
    running_loss = 0
    for X, y_true in train_loader:
        X, y_true = X.to(device), y_true.to(device)
        y_hat = model(X)
        loss = criterion(y_hat, y_true)
        running_loss += loss.item() * X.size(0)
        loss.backward()
        optimizer.step()

    epoch_loss = running_loss / len(train_loader.dataset)
    return model, optimizer, epoch_loss
```

```
def validation(valid_loader, model, criterion, device):
    model.eval()
    running_loss = 0
    for X, y_true in valid_loader:
        X, y_true = X.to(device), y_true.to(device)
```

```
        y_hat= model(X)
        loss = criterion(y_hat, y_true)
        running_loss += loss.item() * X.size(0)

    epoch_loss = running_loss / len(valid_loader.dataset)

    return model, epoch_loss
```

```
def training_loop(model, criterion, optimizer, train_loader, valid_loader, epochs, device, print_every = 1):
    best_loss = 1e10
    train_losses = []
    valid_losses = []

    for epoch in range(epochs):
        model, optimizer, train_loss = train(train_loader, model, criterion, optimizer, device)
        train_losses.append(train_loss)

        with torch.no_grad():
            model, valid_loss = validation(valid_loader, model, criterion, device)
            valid_losses.append(valid_loss)

        if epoch % print_every == (print_every - 1):
            train_acc = get_accuracy(model, train_loader, device)
            valid_acc = get_accuracy(model, valid_loader, device)

            print(f'{datetime.now().time().replace(microsecond=0)} --- '
                  f'Epoch: {epoch}\t'
                  f'Train loss: {train_loss:.4f}\t'
                  f'Valid loss: {valid_loss:.4f}\t'
                  f'Train accuracy: {100 * train_acc:.2f}\t'
                  f'Valid accuracy: {100 * valid_acc:.2f}')

    return model, optimizer, (train_losses, valid_losses)
```

```
device = "cuda:0" if torch.cuda.is_available() else "cpu"

model = VGG16(3, 10).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr = 1e-1)

model, optimizer, _ = training_loop(model, criterion, optimizer,
                                    train_dataloader, valid_dataloader, 15, device)
```

Very Deep Convolutional Networks for Large-Scale Image Recognition

In this work we investigate the effect of the convolutional network depth on its accuracy in the large-scale image recognition setting. Our main contribution is a thorough evaluation of networks of increasing depth using an architecture with very small (3x3)

https://arxiv.org/abs/1409.1556