SOTA deep CNN architectures and their principles: from AlexNet to EfficientNet (1)

Nikolas Adaloglou

https://theaisummer.com/cnn-architectures/

Artificial Intelligence

Creating the Future

Dong-A University

Division of Computer Engineering & Artificial Intelligence

References

Main

- https://theaisummer.com/cnn-architectures/

blog Sub

- https://deep-learning-study.tistory.com/
- https://hoya012.github.io/blog/DenseNet-Tutorial-1/
- https://kjhov195.github.io/2020-02-11-CNN_architecture_3/
- https://poddeeplearning.readthedocs.io/ko/latest/CNN/GoogLeNet/
- https://bskyvision.com/539
- https://m.blog.naver.com/laonple/220686328027

PyTorch tutorial

- website : https://pytorch.org/
- Korea website : https://pytorch.kr/

github

- https://github.com/pytorch
- https://github.com/9bow/PyTorch-tutorials-kr
- torchvision : https://github.com/pytorch/vision
- https://github.com/weiaicunzai/pytorch-cifar100

SOTA of CNN Architecture

Part1

- AlexNet (2012)
- VGG (2014)
- InceptionNet / GoogleNet (2014)
- Inception V2 (2015)
- Inception V3 (2016)
- ResNet (2015)
- Inception V4, Inception-ResNet (2016)
- DenseNet (2017)
- BigTransfer (BiT) (2020)

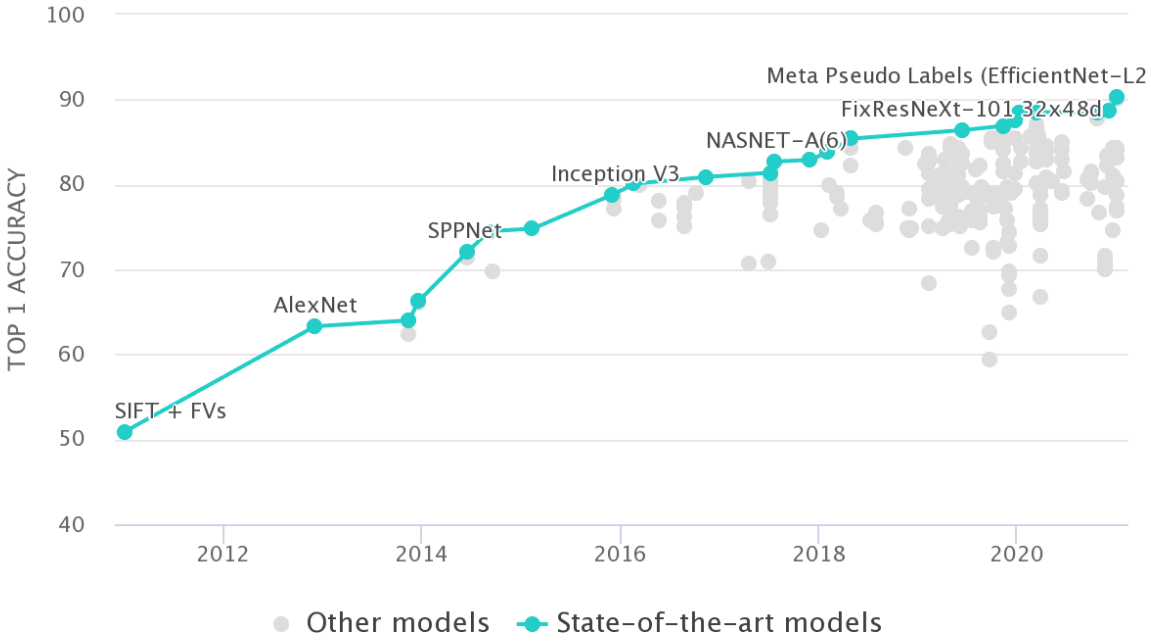
Part2

- EfficientNet (2019)
- Noisy Student (2020)
- Meta - Pseudo Labels (2020)
- EfficientDet (2021)

- GoogleNet (2014 ImageNet winner) : 74.8% top-1 accuracy, about 6.8M parameters
- SENet (2017 ImageNet winner) : 82.7% top-1 accuracy, about 145M parameters
- GPipe (2018, SOTA IMageNet) : 84.% top-1 accuracy, about 557M parameters

1 Introduction

- What a rapid progress in ~8.5 years of deep learning! Back in 2012, Alexnet scored 63.3% Top-1 accuracy on ImageNet. Now, we are over 90% with EfficientNet architectures and teacher-student training.
- If we plot the accuracy of all the reported works on Imagenet, we would get something like this:



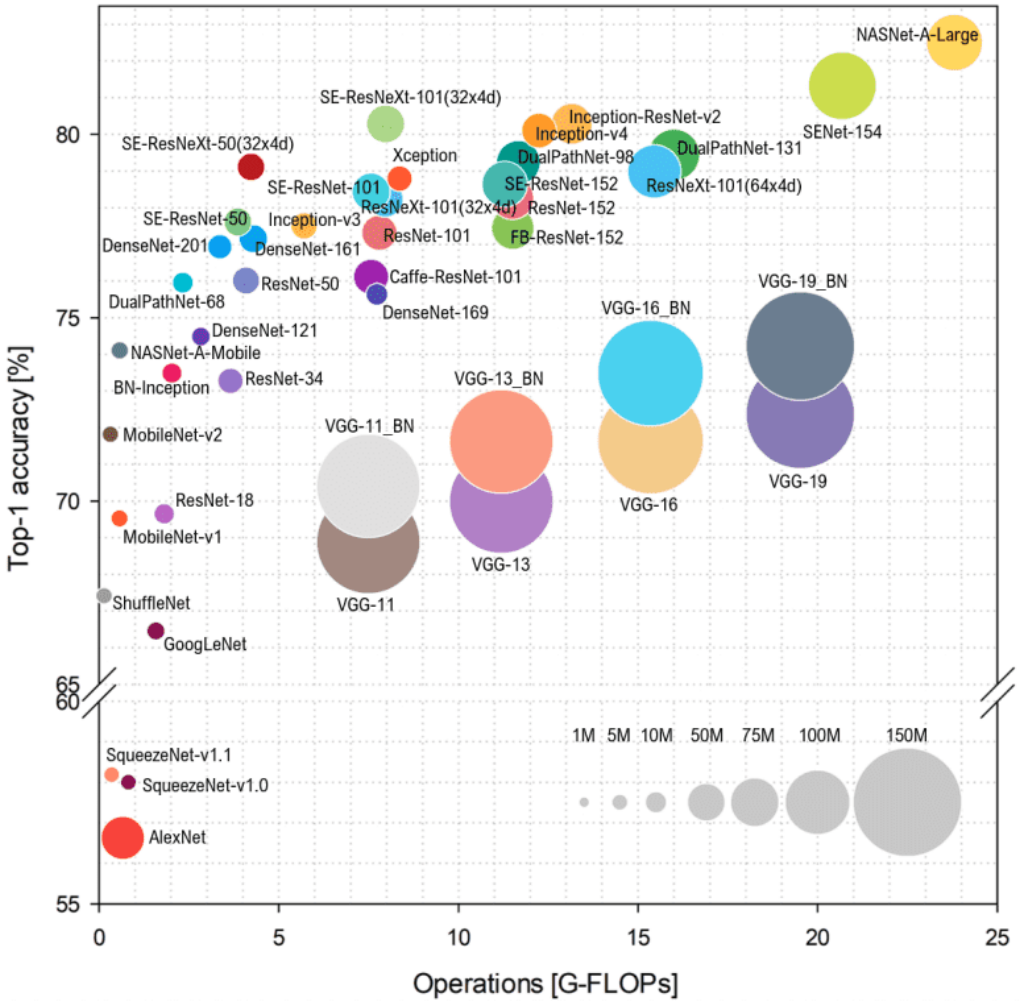
RANK	MODEL	TOP 1 ACCURACY	TOP 5 ACCURACY	NUMBER OF PARAMS	EXTRA TRAINING DATA	PAPER	CODE	RESULT	YEAR
1	Meta Pseudo Labels (EfficientNet-L2)	90.2%	98.8%	480M	✓	Meta Pseudo Labels			2021
2	Meta Pseudo Labels (EfficientNet-B6-Wide)	90%	98.7%	390M	✓	Meta Pseudo Labels			2021
3	NFNet-F4+	89.2%		527M	✓	High-Performance Large-Scale Image Recognition Without Normalization			2021
4	ALIGN (EfficientNet-L2)	88.64%	98.67%	480M	✓	Scaling Up Visual and Vision-Language Representation Learning With Noisy Text Supervision			2021
5	EfficientNet-L2-475 (SAM)	88.61%		480M	✓	Sharpness-Aware Minimization for Efficiently Improving Generalization			2020
6	ViT-H/14	88.55%		632M	✓	An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale			2020
7	FixEfficientNet-L2	88.5%	98.7%	480M	✓	Fixing the train-test resolution discrepancy: FixEfficientNet			2020

Source: Papers with Code - Imagenet Benchmark
https://paperswithcode.com/sota/image-classification-on-imagenet

1 Introduction

- In this article, we will focus on the evolution of convolutional neural networks (CNN) architectures. Rather than reporting plain numbers, we will focus on the fundamental principles. To provide another visual overview, one could capture top-performing CNNs until 2018 in a single image:

- Don’t freak-out. All the depicted architectures are based on the concepts that we will describe.
- Note that, the FLoating point Operations Per second (FLOPs) indicate the complexity of the model, while on the vertical axis we have the ImageNet accuracy. The radius of the circle indicates the number of parameters.
- From the above graph, it is evident that **more parameters do not always lead to better accuracy**. We will attempt to encapsulate a broader perspective on CNNs and see why this holds true.



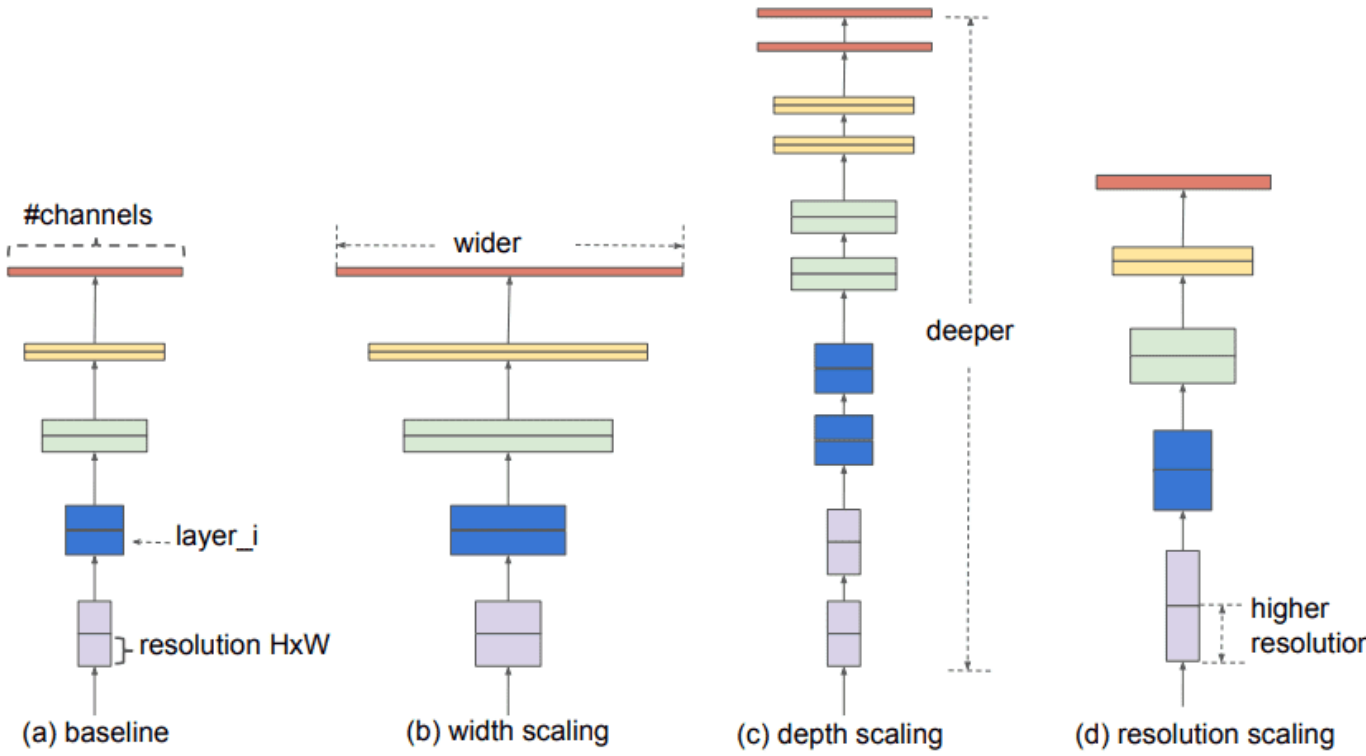
Overview of architectures until 2018. Source: Simone Bianco et al. 2018 Benchmark Analysis of Representative Deep Neural Network Architectures, https://arxiv.org/abs/1810.00736

1 Terminology

But first, we have to define some terminology:

- A **wider** network means more feature maps (filters) in the convolutional layers
- A **deeper** network means more convolutional layers
- A network with **higher resolution** means that it processes input images with larger width and depth (spatial resolutions). That way the produced feature maps will have higher spatial dimensions.

Architecture engineering is all about scaling. We will thoroughly utilize these terms so be sure to understand them before you move on.

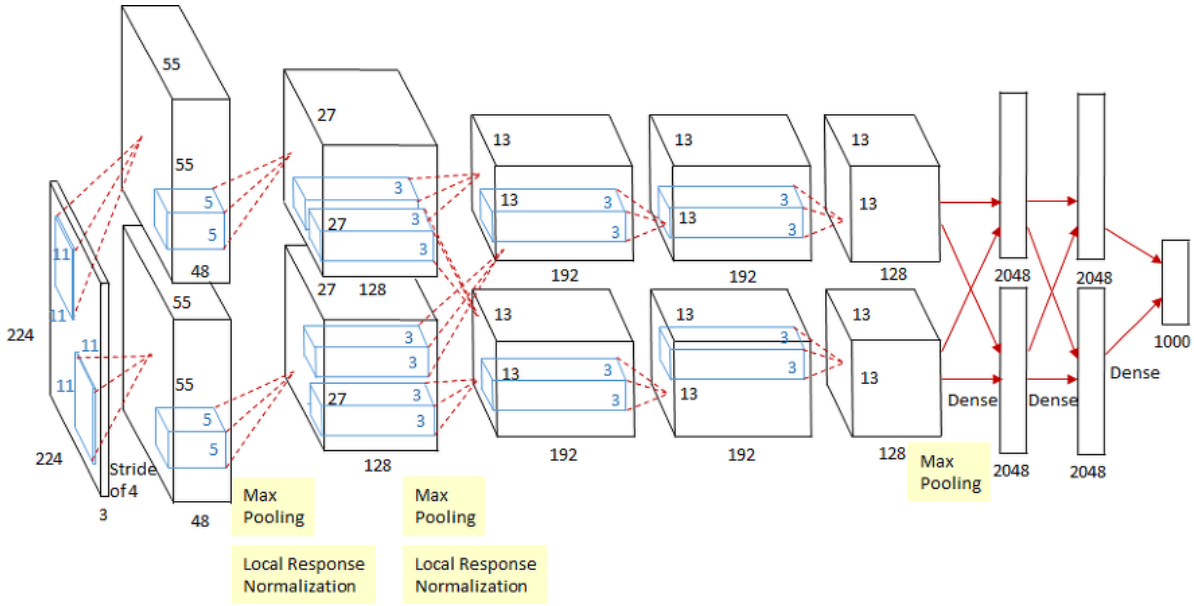


Architecture scaling. Source: Mingxing Tan, Quoc V. Le 2019
https://arxiv.org/abs/1905.11946
EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

2 AlexNet: ImageNet Classification with Deep Convolutional Neural Networks (2012)

- Alexnet [1] is made up of 5 conv layers starting from an 11x11 kernel. It was the first architecture that employed max-pooling layers, ReLu activation functions, and dropout for the 3 enormous linear layers. The network was used for image classification with 1000 possible classes, which for that time was madness.

- It was the first convolutional model that was successfully trained on Imagenet and for that time, it was much more difficult to implement such a model in CUDA. Dropout is heavily used in the enormous linear transformations to avoid overfitting. Before 2015-2016 that auto-differentiation came out, it took months to implement backprop on the GPU.



- AlexNet은 8개의 레이어로 구성되어 있다. 5개의 컨볼루션 레이어와 3개의 full-connected 레이어로 구성되어 있다. 두번째, 네번째, 다섯번째 컨볼루션 레이어들은 전 단계의 같은 채널의 특성맵들과만 연결되어 있는 반면, 세번째 컨볼루션 레이어는 전 단계의 두 채널의 특성맵들과 모두 연결되어 있다.

[출처] https://bskyvision.com/421

[1] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90.

2 AlexNet: ImageNet Classification with Deep Convolutional Neural Networks (2012)

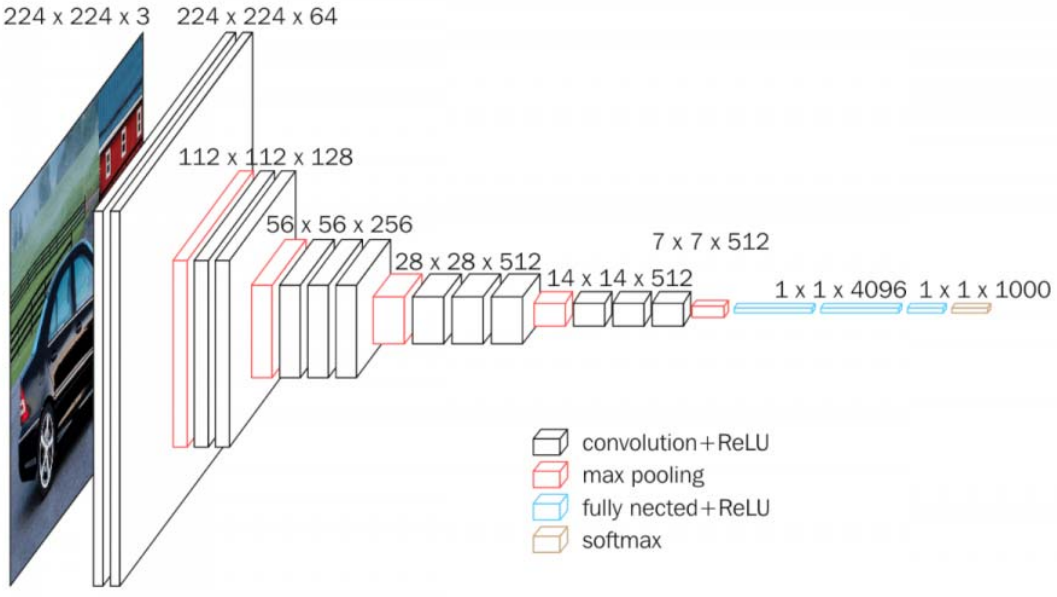
- Now, you can implement it in 35 lines of PyTorch code: **https://github.com/pytorch/vision/tree/master/torchvision/models**

```
import torch
import torch.nn as nn
from .utils import load_state_dict_from_url
from typing import Any

__all__ = ['AlexNet', 'alexnet']


model_urls = {
    'alexnet': 'https://download.pytorch.org/models/alexnet-owt-7be5be79.pth',
}

class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 1000) -> None:
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
```

```
self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

def alexnet(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> AlexNet:
    r"""AlexNet model architecture from the
    `"One weird trick..." <https://arxiv.org/abs/1404.5997>`_ paper.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    model = AlexNet(**kwargs)
    if pretrained:
        state_dict = load_state_dict_from_url(model_urls['alexnet'],
                                              progress=progress)
        model.load_state_dict(state_dict)
    return model
```

2 VGG (2014)

- The famous paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" [2] made the term deep viral. It was the first study that provided undeniable evidence that simply adding more layers increases the performance. Nonetheless, this assumption holds true up to a certain point. To do so, they use only 3x3 kernels, as opposed to AlexNet. The architecture was trained using 224 × 224 RGB images.
- The main principle is that a stack of three 3×3 conv. layers are similar to a single 7×7 layer. And maybe even better! Because they use three non-linear activations in between (instead of one), which makes the function more discriminative.
- Secondly, this design decreases the number of parameters. Specifically, you need $3*(3^2)C^2 = 27xC^2$ weights, compared to a 7×7 conv. layer that would require $1*(7^2)C^2 = 49C^2$ parameters (81% more).
- Intuitively, it can be regarded as a regularization on the 7×7 conv. filters, constricting them to have a 3x3 non-linear decomposition.
- Finally, it was the first architecture that normalization started to become quite an issue.
- Nevertheless, pretrained VGGs are still used for feature matching loss in Generative adversarial Networks, as well as neural style transfer and feature visualizations.



VGG16 구조 (https://bskyvision.com/504)

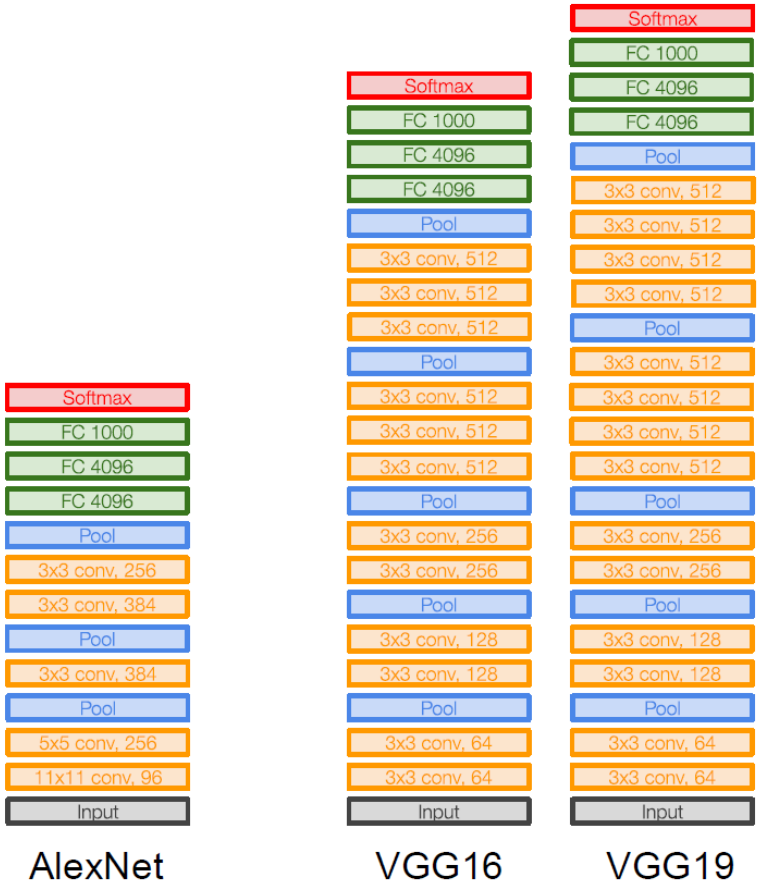
[2] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

2 VGG (2014)

- In my humble opinion, it is very interesting to inspect the features of a convnet with respect to the input, as shown in the following video:

Source: https://www.youtube.com/watch?v=RNnKtNrsrmg

- Finally to get a visual comparison next to Alexnet:



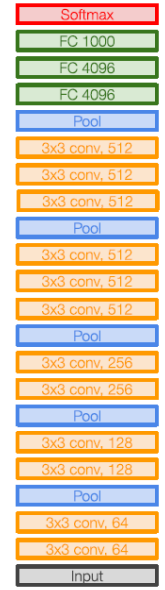
Source: Standford 2017 Deep Learning Lectures: CNN architectures

2 VGG (2014)

- 각 Layer마다 소요되는 Memory와 parameter의 수를 살펴보면 다음과 같다.
- VGGNet의 경우 전체적으로 모델의 뒷단으로 갈수록 feature map의 size는 점차 줄어들면서 filter의 개수는 점차 늘어나도록 구성되어 있다는 특징을 가지고 있다.
- VGGNet의 첫 번째 FC Layer의 parameter의 개수를 눈여겨 볼 만 한데, 하나의 layer에서 약 1억개에 육박하는 parameter를 가지고 있는 것을 확인할 수 있다. 이 때문에 굉장히 비효율적인 training이 이루어지게 되며, overfitting을 해결하는데에도 전혀 도움을 주지 못한다.

INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 96MB / image (for a forward pass)
TOTAL params: 138M parameters



VGG16

[출처] https://kjhov195.github.io/2020-02-11-CNN_architecture_3/

2 VGG (2014)

- PyTorch code: **https://github.com/pytorch/vision/blob/master/torchvision/models/vgg.py**

```
import torch
import torch.nn as nn
from .utils import load_state_dict_from_url
from typing import Union, List, Dict, Any, cast

__all__ = [
    'VGG', 'vgg11', 'vgg11_bn', 'vgg13', 'vgg13_bn', 'vgg16', 'vgg16_bn',
    'vgg19_bn', 'vgg19',
]

model_urls = {
    'vgg11': 'https://download.pytorch.org/models/vgg11-bbd30ac9.pth',
    'vgg13': 'https://download.pytorch.org/models/vgg13-c768596a.pth',
    'vgg16': 'https://download.pytorch.org/models/vgg16-397923af.pth',
    'vgg19': 'https://download.pytorch.org/models/vgg19-dcbb9e9d.pth',
    'vgg11_bn': 'https://download.pytorch.org/models/vgg11_bn-6002323d.pth',
    'vgg13_bn': 'https://download.pytorch.org/models/vgg13_bn-abd245e5.pth',
    'vgg16_bn': 'https://download.pytorch.org/models/vgg16_bn-6c64b313.pth',
    'vgg19_bn': 'https://download.pytorch.org/models/vgg19_bn-c79401a0.pth',
}

class VGG(nn.Module):

    def __init__(
        self,
        features: nn.Module,
        num_classes: int = 1000,
        init_weights: bool = True
    ) -> None:
        super(VGG, self).__init__()
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            self._initialize_weights()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

    def _initialize_weights(self) -> None:
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)
```

2 VGG (2014)

```
def make_layers(cfg: List[Union[str, int]], batch_norm: bool = False) -> nn.Sequential:
    layers: List[nn.Module] = []
    in_channels = 3
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            v = cast(int, v)
            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)

cfgs: Dict[str, List[Union[str, int]]] = {
    'A': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'B': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'D': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512,
'M'],
    'E': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512,
512, 512, 512, 'M'],
}
```

```
def _vgg(arch: str, cfg: str, batch_norm: bool, pretrained: bool, progress: bool, **kwargs: Any)
-> VGG:
    if pretrained:
        kwargs['init_weights'] = False
    model = VGG(make_layers(cfgs[cfg], batch_norm=batch_norm), **kwargs)
    if pretrained:
        state_dict = load_state_dict_from_url(model_urls[arch],
                                              progress=progress)
        model.load_state_dict(state_dict)
    return model


def vgg11(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 11-layer model (configuration "A") from
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition"
<https://arxiv.org/pdf/1409.1556.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg11', 'A', False, pretrained, progress, **kwargs)


def vgg11_bn(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 11-layer model (configuration "A") with batch normalization
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition"
<https://arxiv.org/pdf/1409.1556.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg11_bn', 'A', True, pretrained, progress, **kwargs)
```

2 VGG (2014)

```
def vgg13(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 13-layer model (configuration "B")
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition" <https://arxiv.org/pdf/1409.1556.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg13', 'B', False, pretrained, progress, **kwargs)


def vgg13_bn(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 13-layer model (configuration "B") with batch normalization
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition" <https://arxiv.org/pdf/1409.1556.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg13_bn', 'B', True, pretrained, progress, **kwargs)


def vgg16(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 16-layer model (configuration "D")
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition" <https://arxiv.org/pdf/1409.1556.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg16', 'D', False, pretrained, progress, **kwargs)
```

```
def vgg16_bn(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 16-layer model (configuration "D") with batch normalization
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition" <https://arxiv.org/pdf/1409.1556.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg16_bn', 'D', True, pretrained, progress, **kwargs)


def vgg19(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 19-layer model (configuration "E")
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition" <https://arxiv.org/pdf/1409.1556.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg19', 'E', False, pretrained, progress, **kwargs)

def vgg19_bn(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 19-layer model (configuration 'E') with batch normalization
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition" <https://arxiv.org/pdf/1409.1556.pdf>`_.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg19_bn', 'E', True, pretrained, progress, **kwargs)
```

3 InceptionNet/GoogleNet (2014)

- After VGG, the paper "Going Deeper with Convolutions" [3] by Christian Szegedy et al. was a huge breakthrough.

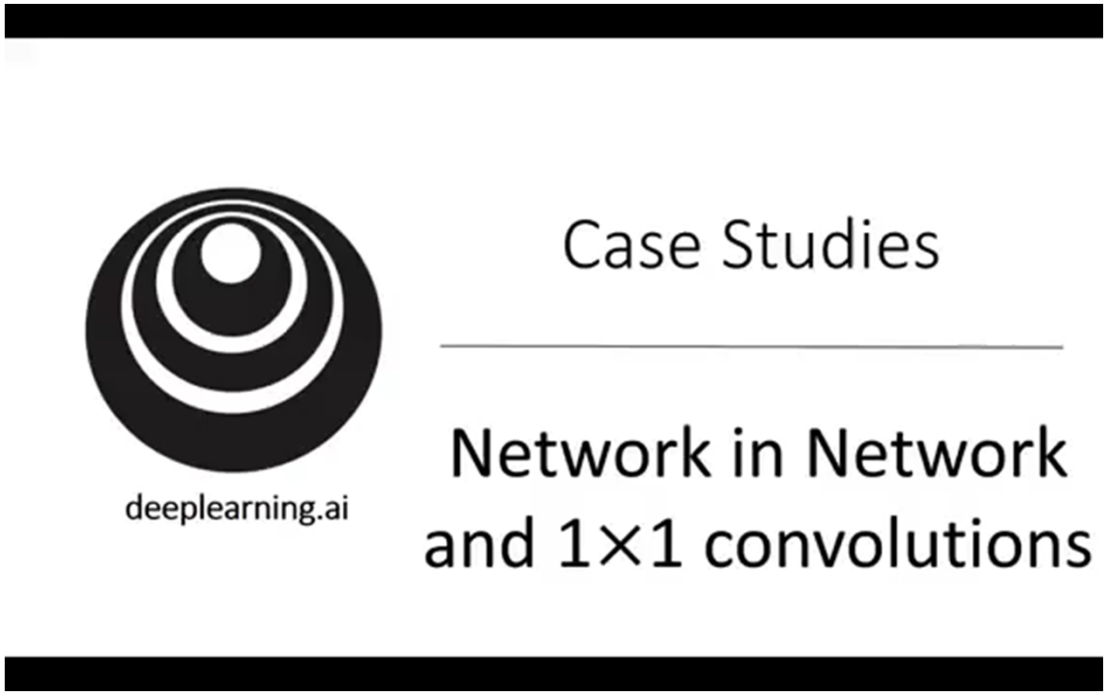
- **Motivation**: Increasing the depth (number of layers) is not the only way to make a model bigger. What about increasing **both** the depth and width of the network while keeping computations to a constant level?
- This time the inspiration comes from the human visual system, wherein information is processed at multiple scales and then aggregated locally [3]. How to achieve this without a memory explosion?
- The answer is with 1×1 convolutions! The main purpose is dimension reduction, by reducing the output channels of each convolution block. Then we can process the input with different kernel sizes. As long as the output is padded, it is the same as in the input.

[3] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

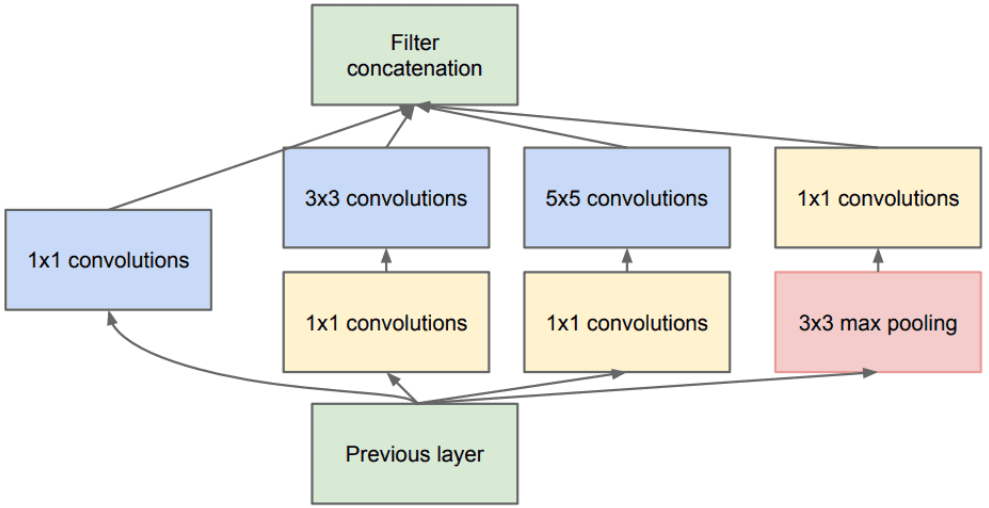
- To find the appropriate padding with single stride convs without dilation, padding p and kernel k are defined so that $out = in$ (input and output spatial dims):
- $out = in + 2 \times p - k + 1$, which means that $p = (k - 1)/2$. In Keras you simply specify padding='same'. This way, we can concatenate features convolved with different kernels.
- Then we need the 1×1 convolutional layer to 'project' the features to fewer channels in order to win computational power. And with these extra resources, we can add more layers. Actually, the 1×1 convs work similar to a low dimensional embedding.

3 InceptionNet/GoogleNet (2014)

- For a quick overview on 1x1 convs advise this video from the famous Coursera course:
- Neural Networks - Networks in Networks and 1x1 Convolutions



- This in turn allows to not only increase the depth, but also the width of the famous GoogleNet by using Inception modules. The core building block, called the inception module, looks like this:

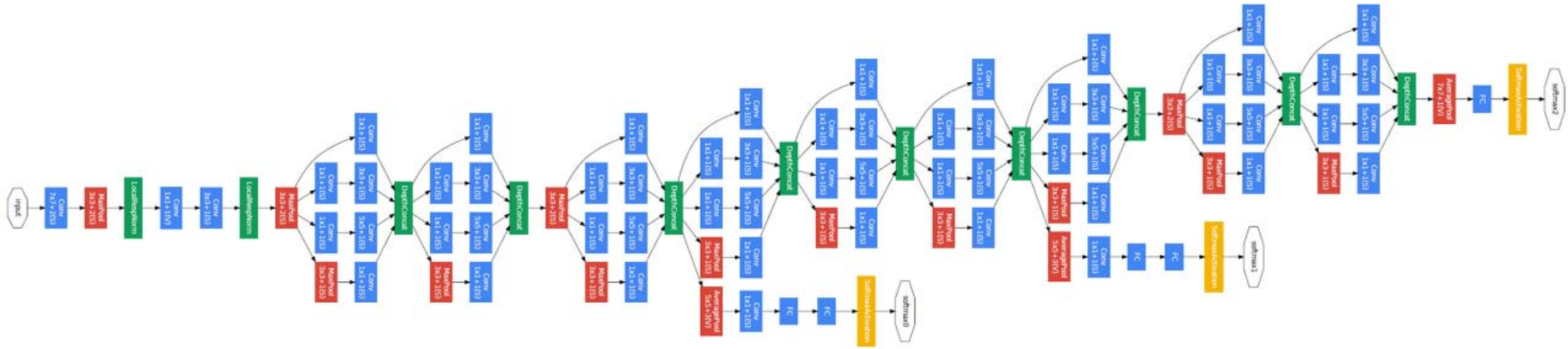


- The whole architecture is called *GoogLeNet* or *InceptionNet*. In essence, the authors claim that they try to approximate a *sparse convnet* with normal dense layers (as shown in the figure).
- Why? Because they believe that only a small number of neurons are effective. This comes in line with the Hebbian principle: "Neurons that fire together, wire together".
- Moreover, it uses convolutions of different kernel sizes (5×5, 3×3, 1×1) to capture details at multiple scales.

3 InceptionNet/GoogleNet (2014)

- In general, a larger kernel is preferred for information that resides globally, and a smaller kernel is preferred for information that is distributed locally.

- Besides, 1 × 1 convolutions are used to compute reductions before the computationally expensive convolutions (3 × 3 and 5 × 5).
- The InceptionNet/GoogLeNet architecture consists of 9 inception modules stacked together, with max-pooling layers between (to halve the spatial dimensions). It consists of 22 layers (27 with the pooling layers). It uses global average pooling after the last inception module.



GoogLeNet의 구조도 [출처: original 논문]

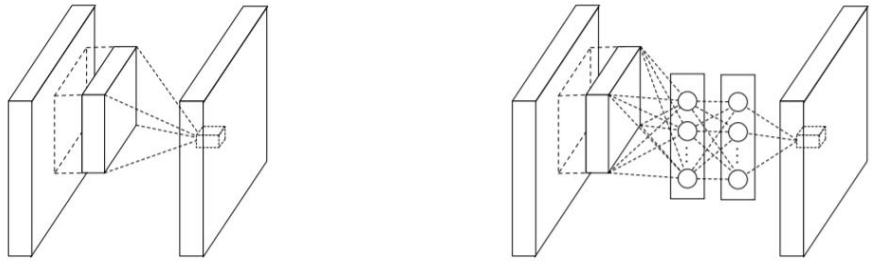
Convolution
Pooling
Softmax
Concatation/Normalization

3 InceptionNet/GoogleNet (2014)

NIN (Network in Network) [출처] https://kangbk0120.github.io/articles/2018-01/inception-googlenet-review

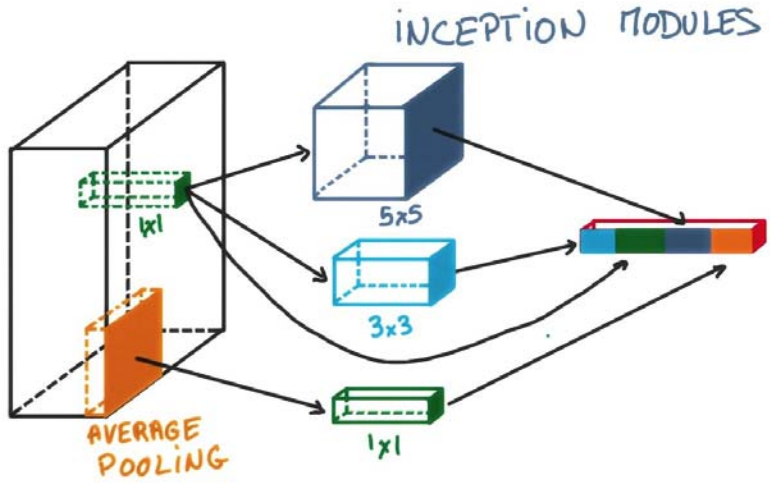
- https://arxiv.org/pdf/1312.4400.pdf

- Convolution layer는 필터가 움직이면서 해당하는 입력 부분과 곱/합 연산을 수행하고, 활성화함수를 거쳐 결과를 만들어냄.
- But 만약 데이터의 분포가 저러한 선형 관계로 표현될 수 없는 비선형적인 관계라면?
- 논문에서 비선형적 관계를 표현할 수 있도록, 단순한 곱/합 연산이 아니라 Multi Layer Perceptron, 즉 MLP를 중간에 넣음. → Network in Network(NIN)
- MLP 또한 Convolution의 필터처럼 역전파를 통해 학습되고, 그 자체가 깊은 구조를 가질 수 있으므로 MLP를 사용했다고 함.



(a) Linear convolution layer (b) Mlpconv layer

Figure 1: Comparison of linear convolution layer and mlpconv layer. The linear convolution layer includes a linear filter while the mlpconv layer includes a micro network (we choose the multilayer perceptron in this paper). Both layers map the local receptive field to a confidence value of the latent concept.



3 InceptionNet/GoogleNet (2014)

NIN (Network in Network) [출처] https://kangbk0120.github.io/articles/2018-01/inception-googlenet-review

➢ 실제 NIN 구조

- MLP Conv를 세 개 쌓고, 마지막에 Fully-Connected Layer를 넣는 대신 Global Average Pooling을 넣음. 이는 오버피팅 방지 효과가 있다고 함.
- 여기서 MLP Conv의 연산을 식으로 나타내면 (i,j)는 feature map에서 pixel 위치, k는 feature map의 k번째 채널, n은 MLP Conv의 n번째 레이어)

$$
\begin{aligned}
f^1_{i,j,k_1} &= \max({w^1_{k_1}}^T x_{i,j} + b_{k_1}, 0). \\
&\vdots \\
f^n_{i,j,k_n} &= \max({w^n_{k_n}}^T f^{n-1}_{i,j} + b_{k_n}, 0).
\end{aligned}
$$

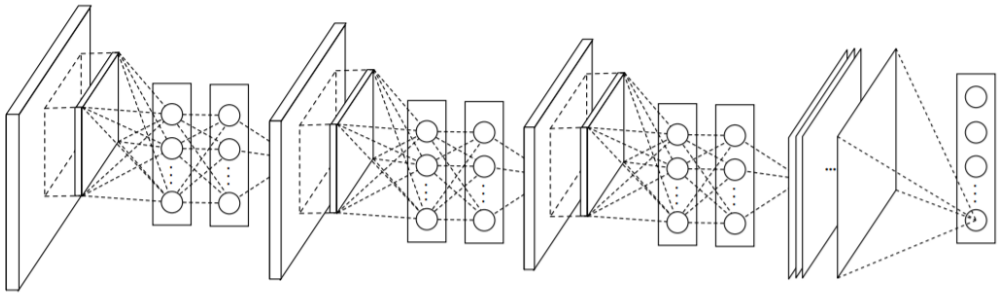
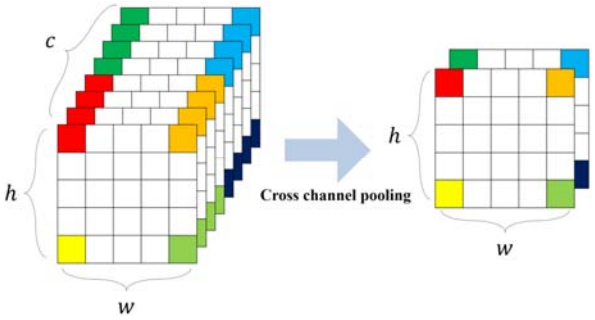
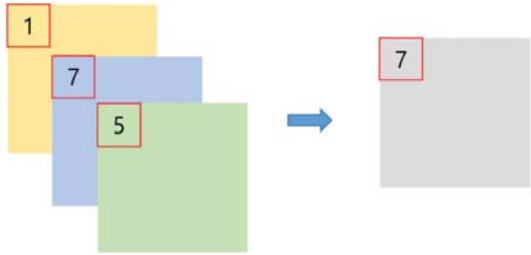


Figure 2: The overall structure of Network In Network. In this paper the NINs include the stacking of three mlpconv layers and one global average pooling layer.

- 첫번째식 : 일반적인 CNN의 feature map $f_{i,j,k} = \max(w_k^T x_{i,j}, 0)$((i,j)는 그 점을 중심으로 하는 input patch, k는 채널의 index)과 동일하다고 볼 수 있음. (물론 bias 고려)
- 마지막 식 : cross channel pooling(CCCP, feature map 크기는 유지하고, 채널 수만 줄임, channel reduction)을 일반적인 Conv 레이어에 적용한 것과 같음. (이는 첫번째 식이 일반적인 CNN과 동일하기 때문)



2개 채널로 cross channel pooling함

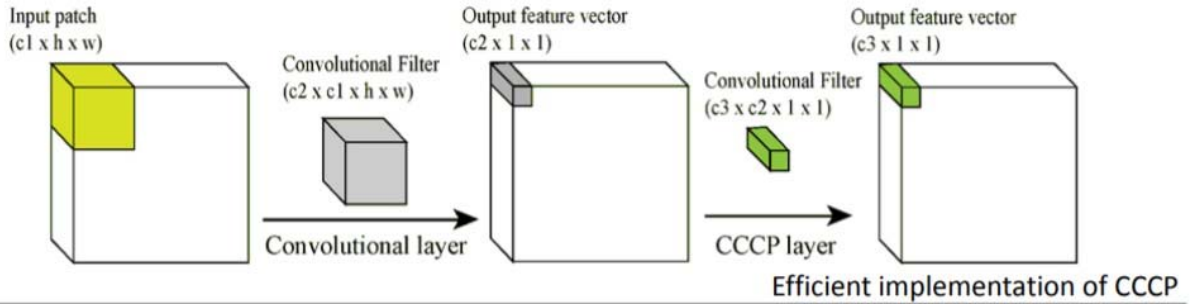


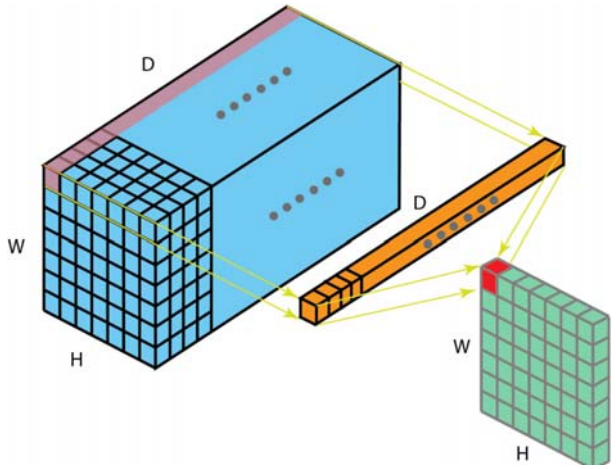
cross channel pooling (max pooling)

3 InceptionNet/GoogleNet (2014)

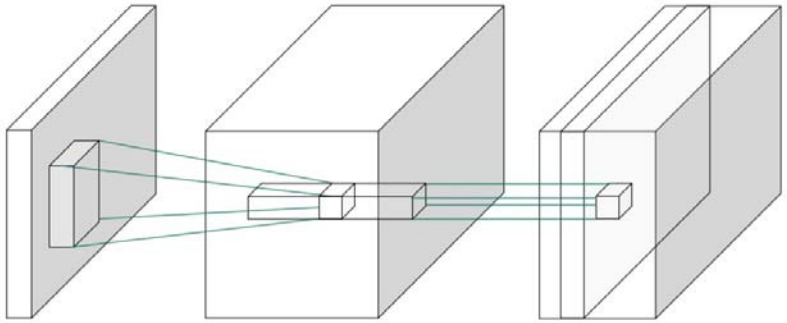
NIN (Network in Network) [출처] https://kangbk0120.github.io/articles/2018-01/inception-googlenet-review

✓ 이렇게 일반적인 Convolution 연산을 거치고, 마지막에 하나의 값으로 매핑하는 과정을 CCCP라고 볼 수 있음.





- 위 과정(n개의 레이어를 사용하는 MLP Conv)은 일반적인 Convolution 연산을 적용한 다음 필터 크기가 1x1인 Convolution을 적용한 것과 동일함.
- 앞에서 말한 *NIN의 목적*이 무엇이었는지 기억하시나요? MLP의 도입을 통해 비선형적인 관계를 더 잘 표현하는 것이었습니다. 바로 위에서 보였듯, 결국 MLP를 통해 구하는 관계는 일반적인 CNN과 1x1 Conv의 결합으로도 표현할 수 있습니다. 즉 NIN의 의의는 이러한 1x1 Conv의 도입이라고 할 수 있죠. 결과적으로 1x1 Conv를 적절하게 사용하면 비선형적 함수를 더 잘 만들어낼 수 있게 되는 것입니다. 또한 1x1 Conv의 장점은 이것 만이 아닙니다. 1x1 Conv는 채널 단위에서 Pooling을 해줍니다. 즉 1x1 Conv의 수를 입력의 채널보다 작게 하면 dimension reduction, 차원 축소가 가능한 것이죠



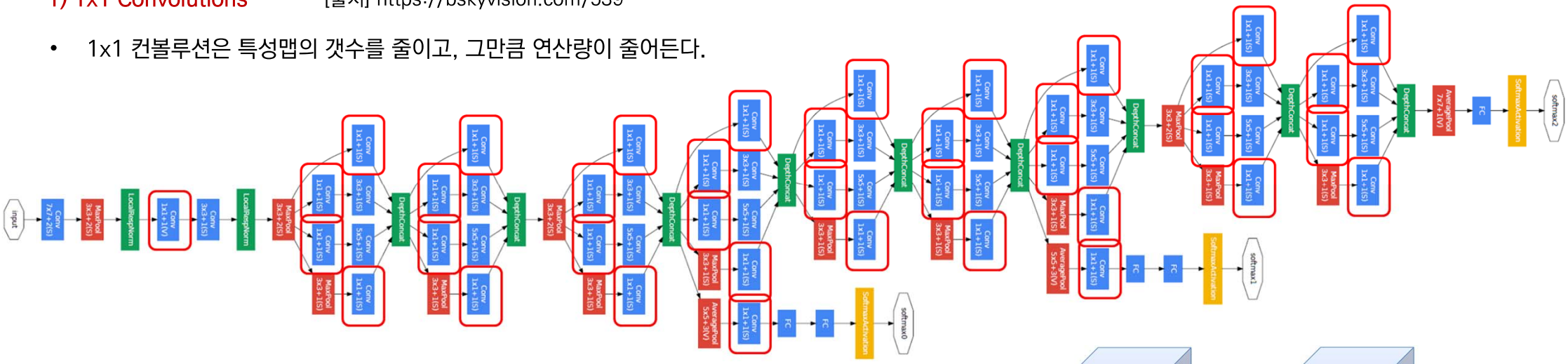
1x1 conv layer

3 InceptionNet/GoogleNet (2014)

1) 1x1 Convolutions

[출처] https://bskyvision.com/539

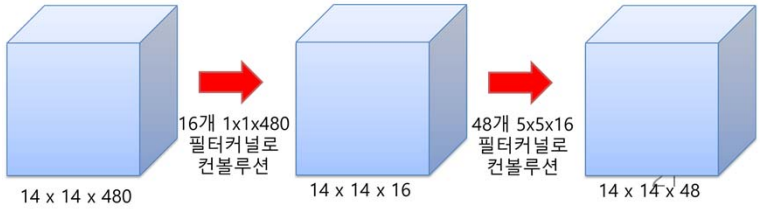
- 1x1 컨볼루션은 특성맵의 갯수를 줄이고, 그만큼 연산량이 줄어든다.



112.9M

V

5.3M



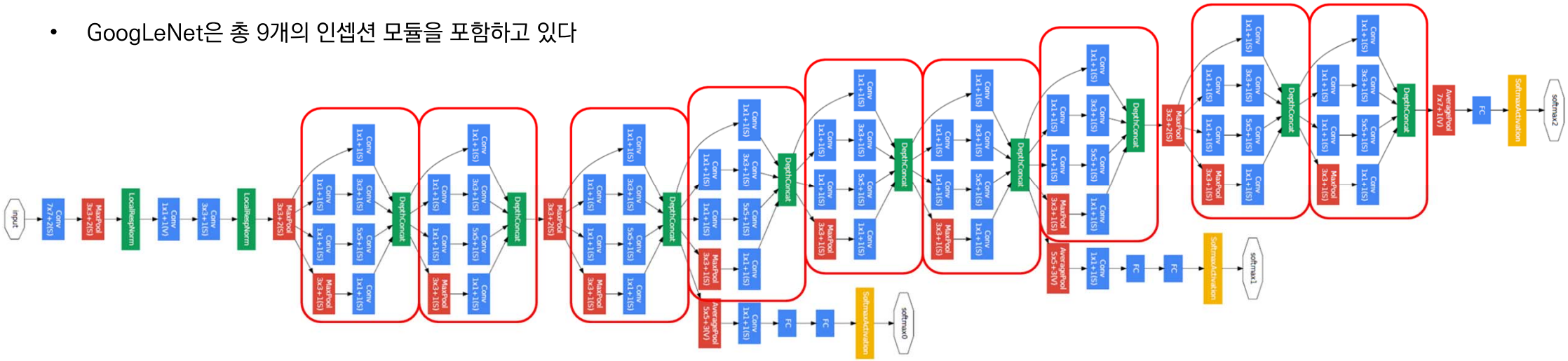
✓ 480장의 14 x 14 사이즈의 특성맵(14 x 14 x 480)이 있다고 가정해보자. 이것을 48개의 5 x 5 x 48의 필터 커널로 컨볼루션을 해주면 48장의 14 x 14의 특성맵(14 x 14 x 48)이 생성된다. (zero padding을 2로, 컨볼루션 보폭은 1로 설정했다고 가정했다.) 이때 필요한 연산횟수는 얼마나 될까? 바로 (14 x 14 x 480) x (5 x 5 x 48) = 약 112.9M이 된다.

✓ 이번에는 480장의 14 x 14 특성맵(14 x 14 x 480)을 먼저 16개의 1 x 1 x 16의 필터커널로 컨볼루션을 해줘 특성맵의 갯수를 줄여보자. 결과적으로 16장의 14 x 14의 특성맵(14 x 14 x 16)이 생성된다. 480장의 특성맵이 16장의 특성맵으로 줄어든 것에 주목하자. 이 14 x 14 x 16 특성맵을 48개의 5 x 5 x 48의 필터커널로 컨볼루션을 해주면 48장의 14 x 14의 특성맵(14 x 14 x 48)이 생성된다. 위에서 1 x 1 컨볼루션이 없을 때와 결과적으로 산출된 특성맵의 크기와 깊이는 같다는 것을 확인하자. 그럼 이때 필요한 연산횟수는 얼마일까? (14 x 14 x 480)*(1 x 1 x 16) + (14 x 14 x 16)*(5 x 5 x 48) = 약 5.3M이다. 112.9M에 비해 훨씬 더 적은 연산량을 가짐을 확인할 수 있다. 연산량을 줄일 수 있다는 점은 네트워크를 더 깊이 만들수 있게 도와준다는 점에서 중요하다

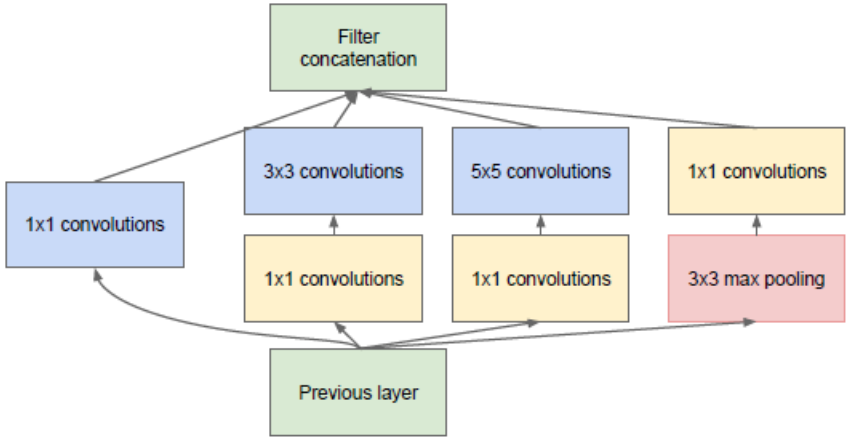
3 InceptionNet/GoogleNet (2014)

2) Inception Module

- GoogLeNet은 총 9개의 인셉션 모듈을 포함하고 있다



- GoogLeNet에 실제로 사용된 모듈은 1x1 컨볼루션이 포함된 (b) 모델이다.
- 노란색 블럭으로 표현된 1x1 컨볼루션을 제외한 나이브(naive) 버전을 살펴보면, 이전 층에서 생성된 특성맵을 1x1 컨볼루션, 3x3 컨볼루션, 5x5 컨볼루션, 3x3 최대 풀링해준 결과 얻은 특성맵들을 모두 함께 쌓아준다. AlexNet, VGGNet 등의 이전 CNN 모델들은 한 층에서 동일한 사이즈의 필터커널을 이용해서 컨볼루션을 해줬던 것과 차이가 있다. 따라서 좀 더 다양한 종류의 특성이 도출된다. 여기에 1x1 컨볼루션이 포함되었으니 당연히 연산량은 많이 줄어들었을 것이다.



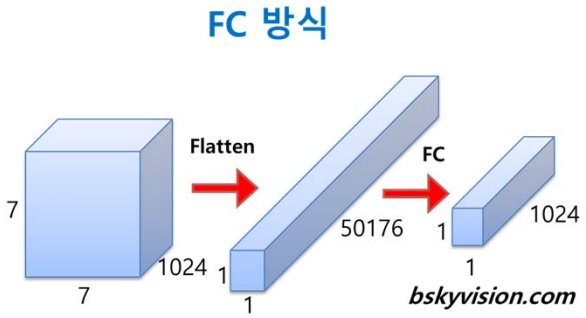
(b) Inception module with dimensionality reduction

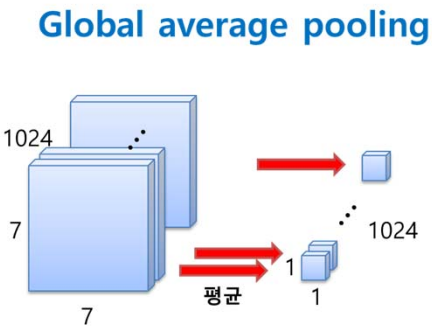
3 InceptionNet/GoogleNet (2014)

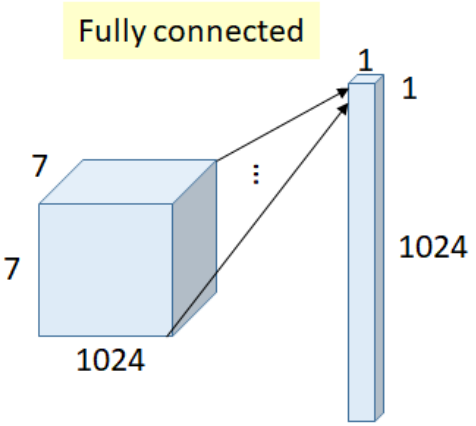
3) Global average pooling

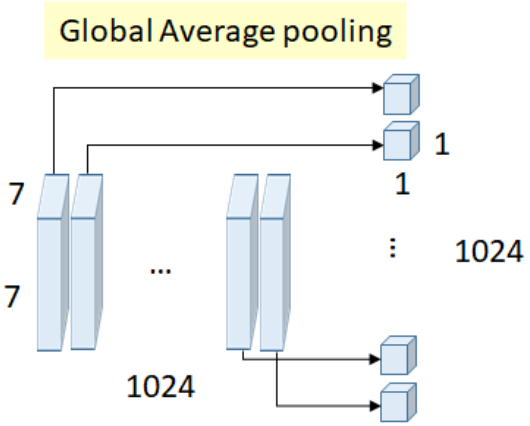
- AlexNet, VGGNet 등에서는 fully connected (FC) 층들이 망의 후반부에 연결되어 있다. 그러나 GoogLeNet은 FC 방식 대신에 *global average pooling*이란 방식을 사용한다. global average pooling은 전 층에서 산출된 특성맵들을 각각 평균낸 것을 이어서 1차원 벡터를 만들어주는 것이다. 1차원 벡터를 만들어줘야 최종적으로 이미지 분류를 위한 softmax 층을 연결해줄 수 있기 때문이다. 만약 전 층에서 1024장의 7 x 7의 특성맵이 생성되었다면, 1024장의 7 x 7 특성맵 각각 평균내주어 얻은 1024개의 값을 하나의 벡터로 연결해주는 것이다.

- 이렇게 해줌으로 얻을 수 있는 장점은 가중치의 갯수를 상당히 많이 없애준다는 것이다. 만약 FC 방식을 사용한다면 훈련이 필요한 가중치의 갯수가 7 x 7 x 1024 x 1024 = 51.3M이지만 global average pooling을 사용하면 가중치가 단 한개도 필요하지 않다.







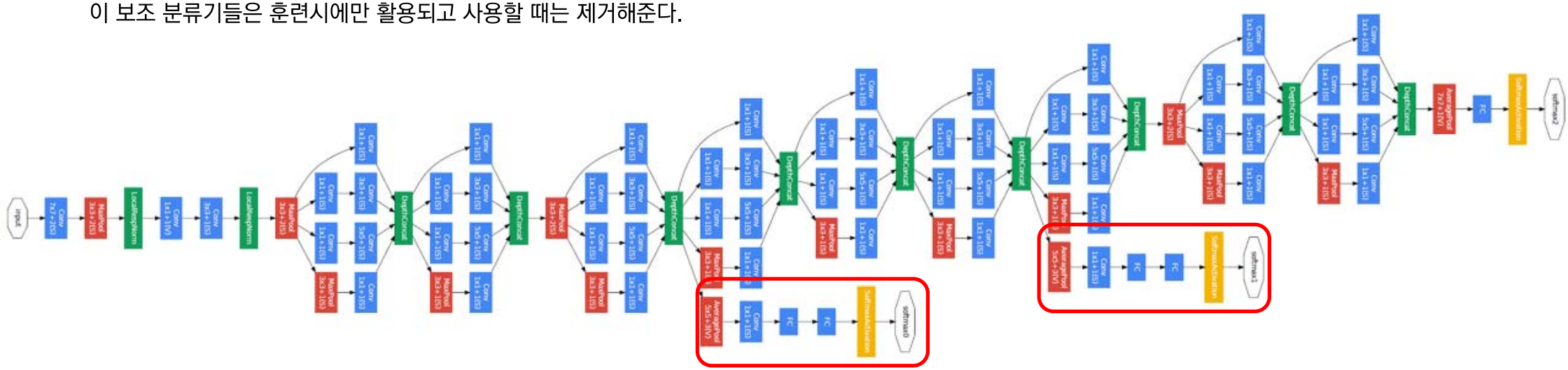


[출처] https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7

3 InceptionNet/GoogleNet (2014)

4) Auxiliary Classifier

- 네트워크의 깊이가 깊어지면 깊어질수록 *vanishing gradient* 문제를 피하기 어려워진다. 그러니까 가중치를 훈련하는 과정에 역전파(back propagation)를 주로 활용하는데, 역전파 과정에서 가중치를 업데이트하는데 사용되는 gradient가 점점 작아져서 0이 되어버리는 것이다. 따라서 네트워크 내의 가중치들이 제대로 훈련되지 않는다. 이 문제를 극복하기 위해서 GoogLeNet에서는 네트워크 중간에 두 개의 보조 분류기(*auxiliary classifier*)를 달아주었다.
- 보조 분류기의 구성을 살펴보면, 5 x 5 평균 풀링(stride 3) –〉 128개 1x1 필터 커널로 컨볼루션 –〉 1024 FC 층 –〉 1000 FC 층 –〉 softmax 순이다. 이 보조 분류기들은 훈련시에만 활용되고 사용할 때는 제거해준다.



① 네트워크의 얕은 부분, 입력과 가까운 부분에는 Inception 모듈을 사용하지 않음. 논문에 따르면 이 부분에는 Inception의 효과가 없다고 함. 따라서 일반적으로 CNN하면 떠올리는, Conv와 Pooling 연산을 수행함.

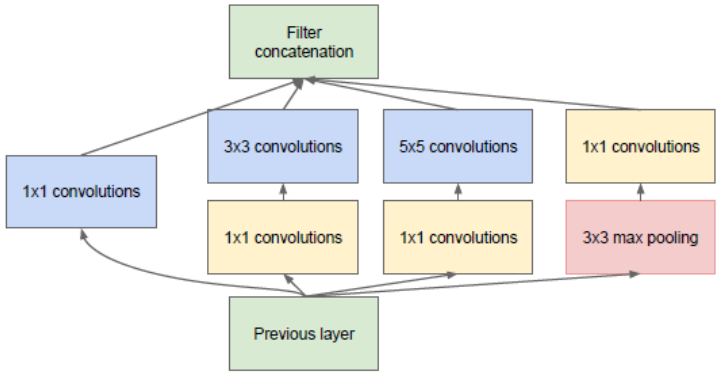
② softmax를 통해 결과를 뽑아내는 부분이 맨 끝과 중간 중간에 있다는 점임 (auxiliary classifier). 엄청나게 깊은 네트워크에서 Vanishing Gradient 문제 극복을 위하여 추가. Loss를 맨 끝뿐만 아니라 중간 중간에서 구하기 때문에 gradient가 적절하게 역전파된다고 함. 대신 지나치게 영향을 주는 것을 막기 위해 auxiliary classifier의 loss는 0.3을 곱함. 물론 실제로 테스트하는 과정에서는 auxiliary classifier를 제거하고 맨 끝, 제일 마지막의 softmax만을 사용함.

[출처] https://kangbk0120.github.io/articles/2018-01/inception-googlenet-review

3 InceptionNet/GoogleNet (2014)

- PyTorch code: **https://colab.research.google.com/drive/1f1HBi7piJM3za2kUTErL1qH9WJwocfv7?usp=sharing**

```
import torch
import torch.nn as nn

class InceptionModule(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(InceptionModule, self).__init__()
        relu = nn.ReLU()
        self.branch1 = nn.Sequential(
                nn.Conv2d(in_channels, out_channels=out_channels, kernel_size=1, stride=1, padding=0),
                relu)

        conv3_1 = nn.Conv2d(in_channels, out_channels=out_channels, kernel_size=1, stride=1, padding=0)
        conv3_3 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.branch2 = nn.Sequential(conv3_1, conv3_3,relu)

        conv5_1 = nn.Conv2d(in_channels, out_channels=out_channels, kernel_size=1, stride=1, padding=0)
        conv5_5 = nn.Conv2d(out_channels, out_channels, kernel_size=5, stride=1, padding=2)
        self.branch3 = nn.Sequential(conv5_1,conv5_5,relu)

        max_pool_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        conv_max_1 = nn.Conv2d(in_channels, out_channels=out_channels, kernel_size=1, stride=1, padding=0)
        self.branch4 = nn.Sequential(max_pool_1, conv_max_1,relu)

    def forward(self, input):
        output1 = self.branch1(input)
        output2 = self.branch2(input)
        output3 = self.branch3(input)
        output4 = self.branch4(input)
        return torch.cat([output1, output2, output3, output4], dim=1)

model = InceptionModule(in_channels=3,out_channels=32)
inp = torch.rand(1,3,128,128)
print(model(inp).shape)
```



3 InceptionNet/GoogleNet (2014)

- PyTorch code: **torchvision**

GoogLeNet | PyTorch : https://pytorch.org/hub/pytorch_vision_googlenet/

https://colab.research.google.com/github/pytorch/pytorch.github.io/blob/master/assets/hub/pytorch_vision_googlenet.ipynb

```
import torch
model = torch.hub.load('pytorch/vision:v0.9.0', 'googlenet', pretrained=True)
model.eval()
```

All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape `(3 x H x W)`, where `H` and `W` are expected to be at least `224`. The images have to be loaded in to a range of `[0, 1]` and then normalized using `mean = [0.485, 0.456, 0.406]` and `std = [0.229, 0.224, 0.225]`.

Here's a sample execution.

```
# Download an example image from the pytorch website
import urllib
url, filename = ("https://github.com/pytorch/hub/raw/master/images/dog.jpg", "dog.jpg")
try: urllib.URLopener().retrieve(url, filename)
except: urllib.request.urlretrieve(url, filename)
```

```
# sample execution (requires torchvision)
from PIL import Image
from torchvision import transforms
input_image = Image.open(filename)
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
input_tensor = preprocess(input_image)
input_batch = input_tensor.unsqueeze(0) # create a mini-batch as expected by the model

# move the input and model to GPU for speed if available
if torch.cuda.is_available():
    input_batch = input_batch.to('cuda')
    model.to('cuda')

with torch.no_grad():
    output = model(input_batch)
# Tensor of shape 1000, with confidence scores over Imagenet's 1000 classes
print(output[0])
# The output has unnormalized scores. To get probabilities, you can run a softmax on it.
probabilities = torch.nn.functional.softmax(output[0], dim=0)
print(probabilities)
```

3 InceptionNet/GoogleNet (2014)

- PyTorch code: **torchvision**

GoogLeNet | PyTorch : https://pytorch.org/hub/pytorch_vision_googlenet/

https://colab.research.google.com/github/pytorch/pytorch.github.io/blob/master/assets/hub/pytorch_vision_googlenet.ipynb

```
# Download ImageNet labels
!wget https://raw.githubusercontent.com/pytorch/hub/master/imagenet_classes.txt
```

```
# Read the categories
with open("imagenet_classes.txt", "r") as f:
    categories = [s.strip() for s in f.readlines()]
# Show top categories per image
top5_prob, top5_catid = torch.topk(probabilities, 5)
for i in range(top5_prob.size(0)):
    print(categories[top5_catid[i]], top5_prob[i].item())
```

3 InceptionNet/GoogleNet (2014)

- PyTorch code: **https://github.com/pytorch/vision/blob/master/torchvision/models/googlenet.py**

```
class GoogLeNet(nn.Module):
    __constants__ = ['aux_logits', 'transform_input']

    def __init__(
        self,
        num_classes: int = 1000,
        aux_logits: bool = True,
        transform_input: bool = False,
        init_weights: Optional[bool] = None,
        blocks: Optional[List[Callable[..., nn.Module]]] = None
    ) -> None:
        super(GoogLeNet, self).__init__()
        if blocks is None:
            blocks = [BasicConv2d, Inception, InceptionAux]
        if init_weights is None:
            warnings.warn('The default weight initialization of GoogleNet will be changed in future releases of '
                          'torchvision. If you wish to keep the old behavior (which leads to long initialization times'
                          ' due to scipy/scipy#11299), please set init_weights=True.', FutureWarning)
            init_weights = True
        assert len(blocks) == 3
        conv_block = blocks[0]
        inception_block = blocks[1]
        inception_aux_block = blocks[2]

        self.aux_logits = aux_logits
        self.transform_input = transform_input.
```

```
        self.conv1 = conv_block(3, 64, kernel_size=7, stride=2, padding=3)
        self.maxpool1 = nn.MaxPool2d(3, stride=2, ceil_mode=True)
        self.conv2 = conv_block(64, 64, kernel_size=1)
        self.conv3 = conv_block(64, 192, kernel_size=3, padding=1)
        self.maxpool2 = nn.MaxPool2d(3, stride=2, ceil_mode=True)

        self.inception3a = inception_block(192, 64, 96, 128, 16, 32, 32)
        self.inception3b = inception_block(256, 128, 128, 192, 32, 96, 64)
        self.maxpool3 = nn.MaxPool2d(3, stride=2, ceil_mode=True)

        self.inception4a = inception_block(480, 192, 96, 208, 16, 48, 64)
        self.inception4b = inception_block(512, 160, 112, 224, 24, 64, 64)
        self.inception4c = inception_block(512, 128, 128, 256, 24, 64, 64)
        self.inception4d = inception_block(512, 112, 144, 288, 32, 64, 64)
        self.inception4e = inception_block(528, 256, 160, 320, 32, 128, 128)
        self.maxpool4 = nn.MaxPool2d(2, stride=2, ceil_mode=True)

        self.inception5a = inception_block(832, 256, 160, 320, 32, 128, 128)
        self.inception5b = inception_block(832, 384, 192, 384, 48, 128, 128)

        if aux_logits:
            self.aux1 = inception_aux_block(512, num_classes)
            self.aux2 = inception_aux_block(528, num_classes)
        else:
            self.aux1 = None  # type: ignore[assignment]
            self.aux2 = None  # type: ignore[assignment]

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout(0.2)
        self.fc = nn.Linear(1024, num_classes)

        if init_weights:
            self._initialize_weights()
```

3 InceptionNet/GoogleNet (2014)

- PyTorch code: **https://github.com/pytorch/vision/blob/master/torchvision/models/googlenet.py**

```
    def _initialize_weights(self) -> None:
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                import scipy.stats as stats
                X = stats.truncnorm(-2, 2, scale=0.01)
                values = torch.as_tensor(X.rvs(m.weight.numel()), dtype=m.weight.dtype)
                values = values.view(m.weight.size())
                with torch.no_grad():
                    m.weight.copy_(values)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def _transform_input(self, x: Tensor) -> Tensor:
        if self.transform_input:
            x_ch0 = torch.unsqueeze(x[:, 0], 1) * (0.229 / 0.5) + (0.485 - 0.5) / 0.5
            x_ch1 = torch.unsqueeze(x[:, 1], 1) * (0.224 / 0.5) + (0.456 - 0.5) / 0.5
            x_ch2 = torch.unsqueeze(x[:, 2], 1) * (0.225 / 0.5) + (0.406 - 0.5) / 0.5
            x = torch.cat((x_ch0, x_ch1, x_ch2), 1)
        return x

    def _forward(self, x: Tensor) -> Tuple[Tensor, Optional[Tensor], Optional[Tensor]]:
        # N x 3 x 224 x 224
        x = self.conv1(x)
        # N x 64 x 112 x 112
        x = self.maxpool1(x)
        # N x 64 x 56 x 56
        x = self.conv2(x)
        # N x 64 x 56 x 56
        x = self.conv3(x)
        # N x 192 x 56 x 56
        x = self.maxpool2(x)
        # N x 192 x 28 x 28
        x = self.inception3a(x)
        # N x 256 x 28 x 28
        x = self.inception3b(x)
        # N x 480 x 28 x 28
        x = self.maxpool3(x)
        # N x 480 x 14 x 14
        x = self.inception4a(x)
        # N x 512 x 14 x 14
        aux1: Optional[Tensor] = None
        if self.aux1 is not None:
            if self.training:
                aux1 = self.aux1(x)

        x = self.inception4b(x)
        # N x 512 x 14 x 14
        x = self.inception4c(x)
        # N x 512 x 14 x 14
        x = self.inception4d(x)
        # N x 528 x 14 x 14
        aux2: Optional[Tensor] = None
        if self.aux2 is not None:
            if self.training:
                aux2 = self.aux2(x)
        x = self.avgpool(x)
        # N x 1024 x 1 x 1
        x = torch.flatten(x, 1)
        # N x 1024
        x = self.dropout(x)
        x = self.fc(x)
        # N x 1000 (num_classes)
        return x, aux2, aux1
```

3 InceptionNet/GoogleNet (2014)

- PyTorch code:

```
    @torch.jit.unused
    def eager_outputs(self, x: Tensor, aux2: Tensor, aux1: Optional[Tensor]) -> GoogLeNetOutputs:
        if self.training and self.aux_logits:
            return _GoogLeNetOutputs(x, aux2, aux1)
        else:
            return x   # type: ignore[return-value]

    def forward(self, x: Tensor) -> GoogLeNetOutputs:
        x = self._transform_input(x)
        x, aux1, aux2 = self._forward(x)
        aux_defined = self.training and self.aux_logits
        if torch.jit.is_scripting():
            if not aux_defined:
                warnings.warn("Scripted GoogleNet always returns GoogleNetOutputs Tuple")
            return GoogLeNetOutputs(x, aux2, aux1)
        else:
            return self.eager_outputs(x, aux2, aux1)

class Inception(nn.Module):

    def __init__(
        self,
        in_channels: int,
        ch1x1: int,
        ch3x3red: int,
        ch3x3: int,
        ch5x5red: int,
        ch5x5: int,
        pool_proj: int,
        conv_block: Optional[Callable[..., nn.Module]] = None
    ) -> None:
        super(Inception, self).__init__()
        if conv_block is None:
            conv_block = BasicConv2d
        self.branch1 = conv_block(in_channels, ch1x1, kernel_size=1)

        self.branch2 = nn.Sequential(
            conv_block(in_channels, ch3x3red, kernel_size=1),
            conv_block(ch3x3red, ch3x3, kernel_size=3, padding=1)
        )

        self.branch3 = nn.Sequential(
            conv_block(in_channels, ch5x5red, kernel_size=1),
            # Here, kernel_size=3 instead of kernel_size=5 is a known bug.
            # Please see https://github.com/pytorch/vision/issues/906 for details.
            conv_block(ch5x5red, ch5x5, kernel_size=3, padding=1)
        )

        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1, ceil_mode=True),
            conv_block(in_channels, pool_proj, kernel_size=1)
        )

    def _forward(self, x: Tensor) -> List[Tensor]:
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)

        outputs = [branch1, branch2, branch3, branch4]
        return outputs

    def forward(self, x: Tensor) -> Tensor:
        outputs = self._forward(x)
        return torch.cat(outputs, 1)
```

3 InceptionNet/GoogleNet (2014)

- PyTorch code: **https://github.com/pytorch/vision/blob/master/torchvision/models/googlenet.py**

```
class InceptionAux(nn.Module):

    def __init__(
        self,
        in_channels: int,
        num_classes: int,
        conv_block: Optional[Callable[..., nn.Module]] = None
    ) -> None:
        super(InceptionAux, self).__init__()
        if conv_block is None:
            conv_block = BasicConv2d
        self.conv = conv_block(in_channels, 128, kernel_size=1)

        self.fc1 = nn.Linear(2048, 1024)
        self.fc2 = nn.Linear(1024, num_classes)

    def forward(self, x: Tensor) -> Tensor:
        # aux1: N x 512 x 14 x 14, aux2: N x 528 x 14 x 14
        x = F.adaptive_avg_pool2d(x, (4, 4))
        # aux1: N x 512 x 4 x 4, aux2: N x 528 x 4 x 4
        x = self.conv(x)
        # N x 128 x 4 x 4
        x = torch.flatten(x, 1)
        # N x 2048
        x = F.relu(self.fc1(x), inplace=True)
        # N x 1024
        x = F.dropout(x, 0.7, training=self.training)
        # N x 1024
        x = self.fc2(x)
        # N x 1000 (num_classes)

        return x
```

```
class BasicConv2d(nn.Module):

    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        **kwargs: Any
    ) -> None:
        super(BasicConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, bias=False, **kwargs)
        self.bn = nn.BatchNorm2d(out_channels, eps=0.001)

    def forward(self, x: Tensor) -> Tensor:
        x = self.conv(x)
        x = self.bn(x)
        return F.relu(x, inplace=True)
```

3 InceptionNet/GoogleNet (2014)

torchvision 기반 학습 코드 예제

GoogleNet

https://deep-learning-study.tistory.com/523

Inception v4 (2016)

https://deep-learning-study.tistory.com/537

3 Inception V2 (2015)

- Later on, in the paper "Rethinking the Inception Architecture for Computer Vision" the authors improved the Inception model based on the following principles:

✓ Factorize 5x5 and 7x7 (in InceptionV3) convolutions to two and three 3x3 sequential convolutions respectively. This improves *computational speed*. This is the same principle as VGG.

✓ They used spatially separable convolutions. Simply, a 3x3 kernel is decomposed into two smaller ones: a 1x3 and a 3x1 kernel, which are applied sequentially.

✓ The inception modules became wider (more feature maps).

✓ They tried to distribute the computational budget in a balanced way between the depth and width of the network.

✓ They added batch normalization.

- Later versions of the inception model are InceptionV4 and Inception-Resnet.

[1] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna, "Rethinking the Inception Architecture for Computer Vision," arXiv preprint, arXiv:1512.00567, 2015.

[2] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alex Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," arXiv preprint, arXiv:1602.07261, 2016.

- Neural-Net 설계 원칙

1) Avoid representational bottlenecks, especially early in the network
2) Higher dimensional representations are easier to process locally within a network.
3) Spatial aggregation can be done over lower dimensional embeddings without much or any loss in representational power
4) Balance the width and depth of the network.

- Inception-v2의 3가지 핵심 요소

1) Conv Filter Factorization
2) Rethinking Auxiliary Classifier
3) Avoid representational bottleneck → Grid Size Reduction

[출처1] https://norman3.github.io/papers/docs/google_inception.html
[출처2] https://hoya012.github.io/blog/deeplearning-classification-guidebook-2/

3 Inception V2 (2015)

[출처] https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728

- Spatial Separable Convolutions

Note

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Image 1: Separating a 3x3 kernel spatially

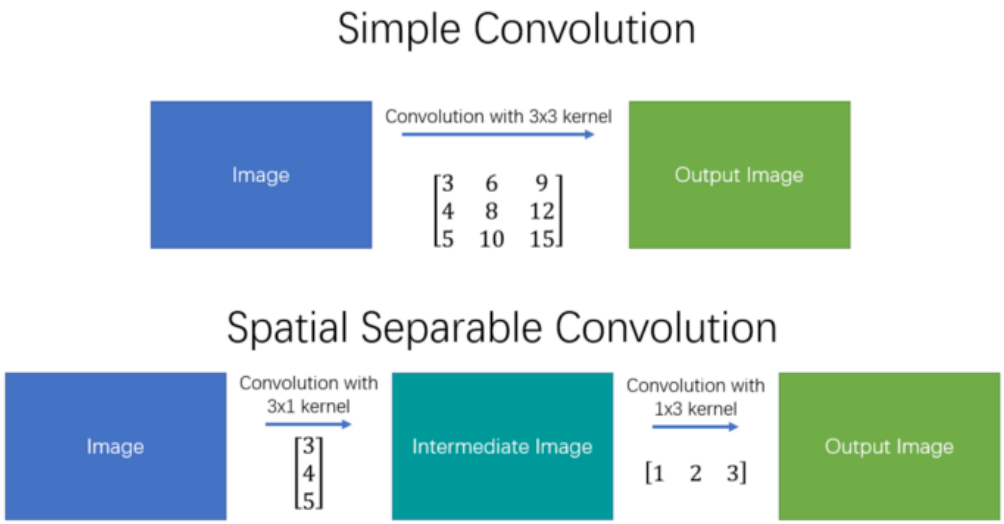


Image 2: Simple and spatial separable convolution

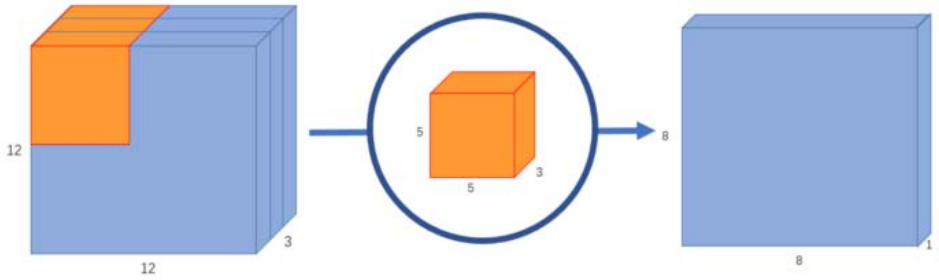


Image 4: A normal convolution with 8x8x1 output

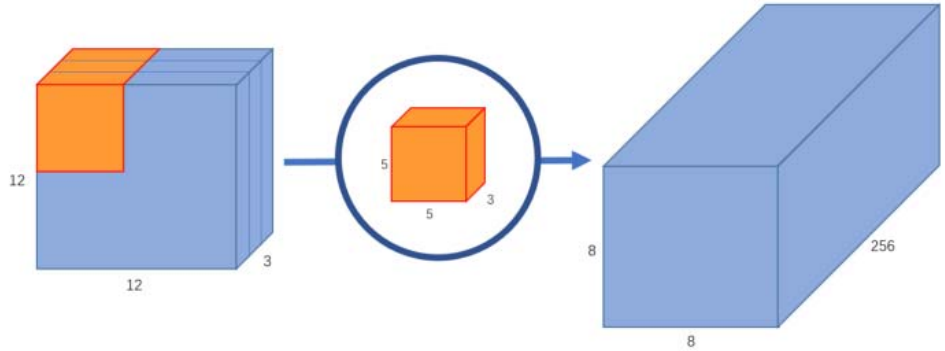


Image 5: A normal convolution with 8x8x256 output (256 kernels)

3 Inception V2 (2015)

[출처] https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728

- Depthwise Convolutions

  ✓ Give the input image a convolution without changing the depth.

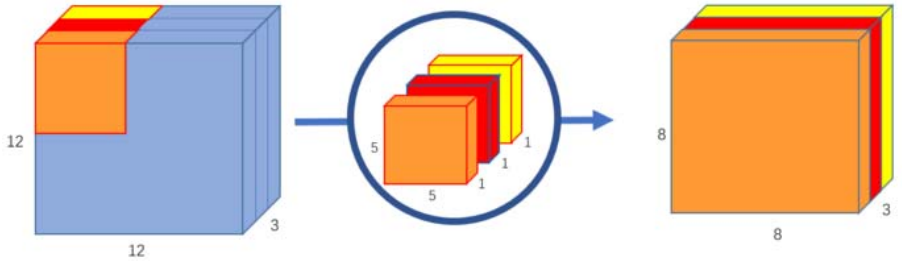


Image 6: Depthwise convolution, uses 3 kernels to transform a 12x12x3 image to a 8x8x3 image

  ✓ Each 5x5x1 kernel iterates 1 channel of the image (note: 1 channel, not all channels), getting the scalar products of every 25 pixel group, giving out a 8x8x1 image. Stacking these images together creates a 8x8x3 image.

- The original convolution transformed a 12x12x3 image to a 8x8x256 image. Currently, the depthwise convolution has transformed the 12x12x3 image to a 8x8x3 image. Now, we need to increase the number of channels of each image.

Note

- Pointwise Convolutions

  ✓ It uses a 1x1 kernel, or a kernel that iterates through every single point. This kernel has a depth of however many channels the input image has; in our case, 3. Therefore, we iterate a 1x1x3 kernel through our 8x8x3 image, to get a 8x8x1 image.

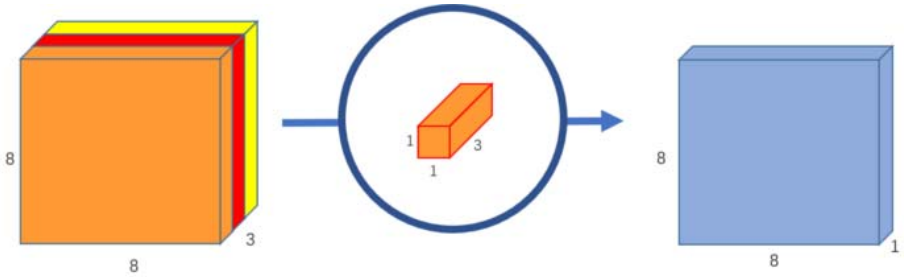


Image 7: Pointwise convolution, transforms an image of 3 channels to an image of 1 channel

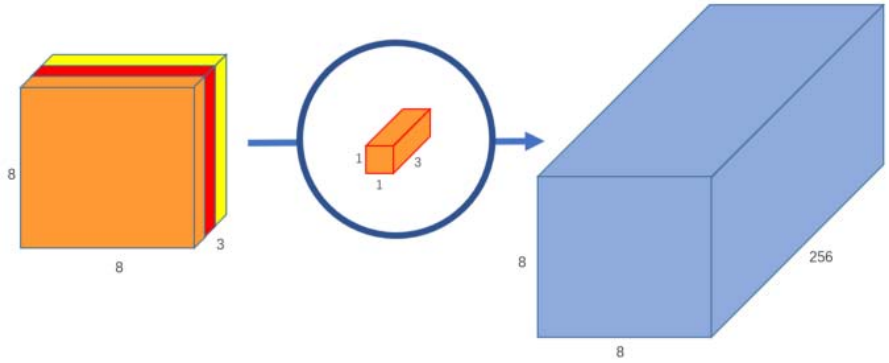


Image 8: Pointwise convolution with 256 kernels, outputting an image with 256 channels

3 Inception V2 (2015)

[출처1] https://norman3.github.io/papers/docs/google_inception.html
[출처2] https://hoya012.github.io/blog/deeplearning-classification-guidebook-2/

1) Conv Filter Factorization

- Inception-v1(GoogLeNet)은 VGG, AlexNet에 비해 parameter수가 굉장히 적지만, 여전히 많은 연산량을 필요로 함.
- Inception-v2에서는 연산의 복잡도를 줄이기 위한 여러 Conv Filter Factorization 방법을 제안하고 있음. 우선 VGG에서 했던 것처럼 5x5 conv를 3x3 conv 2개로 대체하는 방법을 적용하고, 나아가 연산량은 줄어들지만 receptive field는 동일한 점을 이용하여 n x n conv를 1 x n + n x 1 conv로 factorization하는 방법을 제안함
- 계산량이 33% 줄어듬

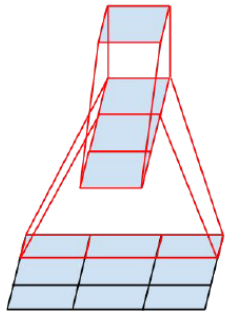


Figure 3. Mini-network replacing the 3×3 convolutions. The lower layer of this network consists of a 3×1 convolution with 3 output units.

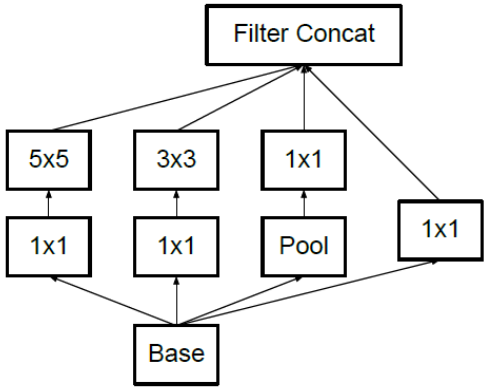


Figure 4. Original Inception module as described in [20].

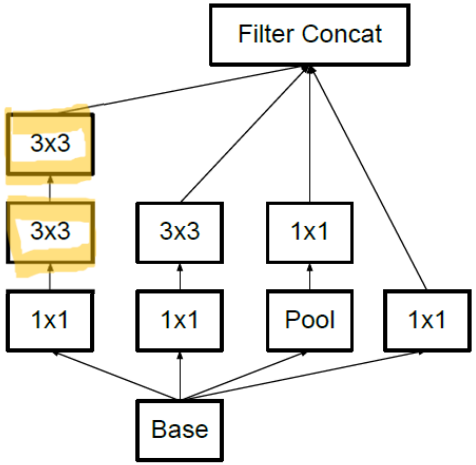


Figure 5. Inception modules where each 5×5 convolution is replaced by two 3×3 convolution, as suggested by principle 3 of Section 2

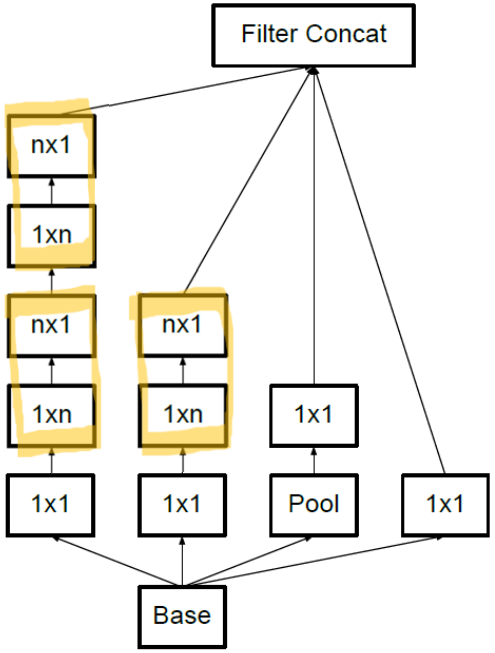


Figure 6. Inception modules after the factorization of the $n \times n$ convolutions. In our proposed architecture, we chose $n = 7$ for the 17×17 grid. (The filter sizes are picked using principle 3)

3 Inception V2 (2015)

[출처1] https://norman3.github.io/papers/docs/google_inception.html
[출처2] https://hoya012.github.io/blog/deeplearning-classification-guidebook-2/

2) Auxiliary Classifiers

- Inception-v1(GoogLeNet)에서 (Backprop시 weight 갱신을 더 잘하라는) 적용했던 auxiliary classifier에 대한 재조명
- 여러 실험과 분석을 통해 auxiliary classifier가 학습 초기에는 수렴성을 개선시키지 않음을 보였고, 학습 후기에 약간의 정확도 향상을 얻을 수 있음을 보였음.
- 또한 기존엔 2개의 auxiliary classifier를 사용하였으나, 실제로 초기 단계(lower)의 auxiliary classifier는 있으나 없으나 큰 차이가 없어서 제거함.

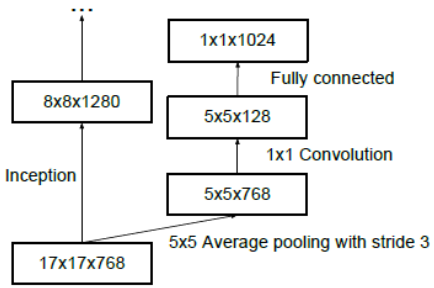


Figure 8. Auxiliary classifier on top of the last 17×17 layer. Batch normalization[7] of the layers in the side head results in a 0.4% absolute gain in top-1 accuracy. The lower axis shows the number of itertions performed, each with batch size 32.

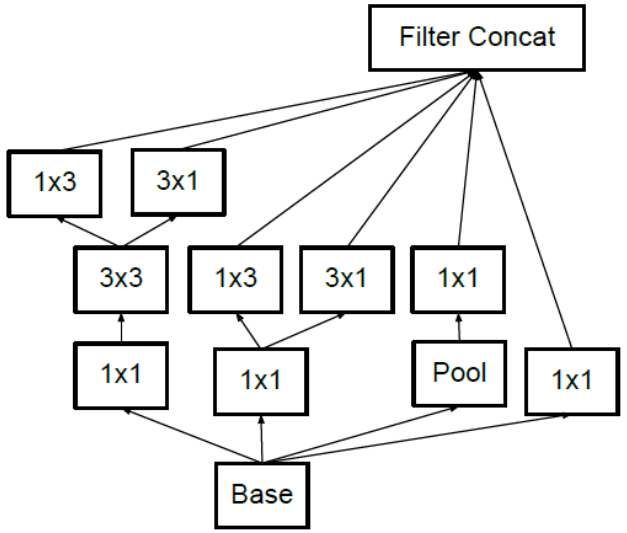


Figure 7. Inception modules with expanded the filter bank outputs. This architecture is used on the coarsest (8×8) grids to promote high dimensional representations, as suggested by principle 2 of Section 2. We are using this solution only on the coarsest grid, since that is the place where producing high dimensional sparse representation is the most critical as the ratio of local processing (by 1×1 convolutions) is increased compared to the spatial aggregation.

3 Inception V2 (2015)

[출처1] https://norman3.github.io/papers/docs/google_inception.html
[출처2] https://hoya012.github.io/blog/deeplearning-classification-guidebook-2/

3) Grid Size Reduction (Avoid representational bottleneck)

- Representational bottleneck : CNN에서 주로 사용되는 pooling 으로 인해 feature map의 size가 줄어들면서 정보량이 줄어드는 것
- 즉, 그림9의 왼쪽과 같이 pooling을 먼저 하면 Representational bottleneck이 발생하고, 오른쪽과 같이 pooling을 뒤에 하면 연산량이 많아짐. 그래서 연산량도 줄이면서 Representational bottleneck도 피하기 위해 가운데와 같은 방식을 제안하였고, 최종적으론 맨 오른쪽과 같은 방식을 이용함.

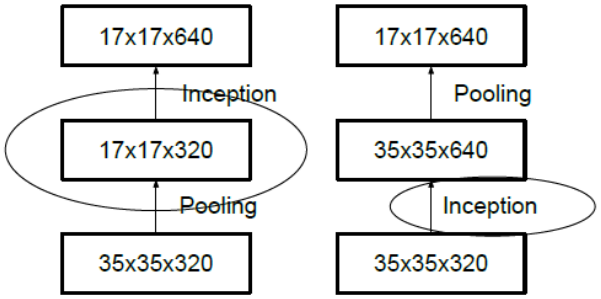


Figure 9. Two alternative ways of reducing the grid size. The solution on the left violates the principle 1 of not introducing an representational bottleneck from Section 2. The version on the right is 3 times more expensive computationally.

- 예제에서 연산은 (d,d,k) 를 (d/2,d/2,2k) 로 변환하는 Conv 로 확인. (여기서는 d=35, k=320)
- 실제 연산 수는
  - ✓ pooling + stride 1 conv with 2k filter : $2(d/2)^2k^2$ 연산 수
  - ✓ strid.1 conv with 2k fileter + pooling : $2d^2k^2$ 연산 수

  왼쪽은 연산량이 좀 더 작지만 Representational Bottleneck 이 발생.
  오른쪽은 정보 손실이 더 적지만 연산량이 2배.

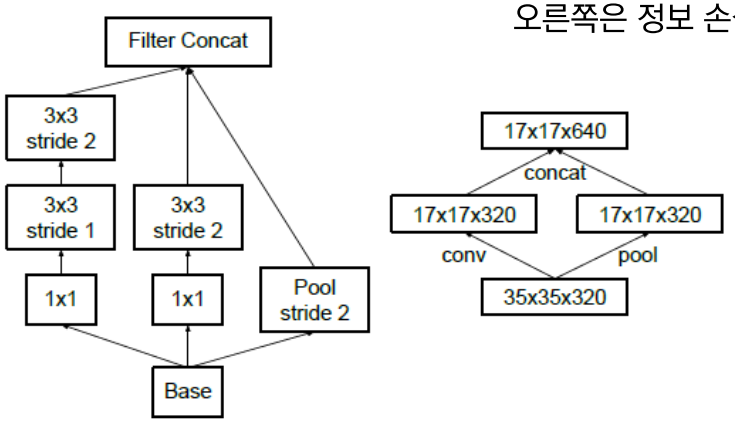


Figure 10. Inception module that reduces the grid-size while expands the filter banks. It is both cheap and avoids the representational bottleneck as is suggested by principle 1. The diagram on the right represents the same solution but from the perspective of grid sizes rather than the operations.

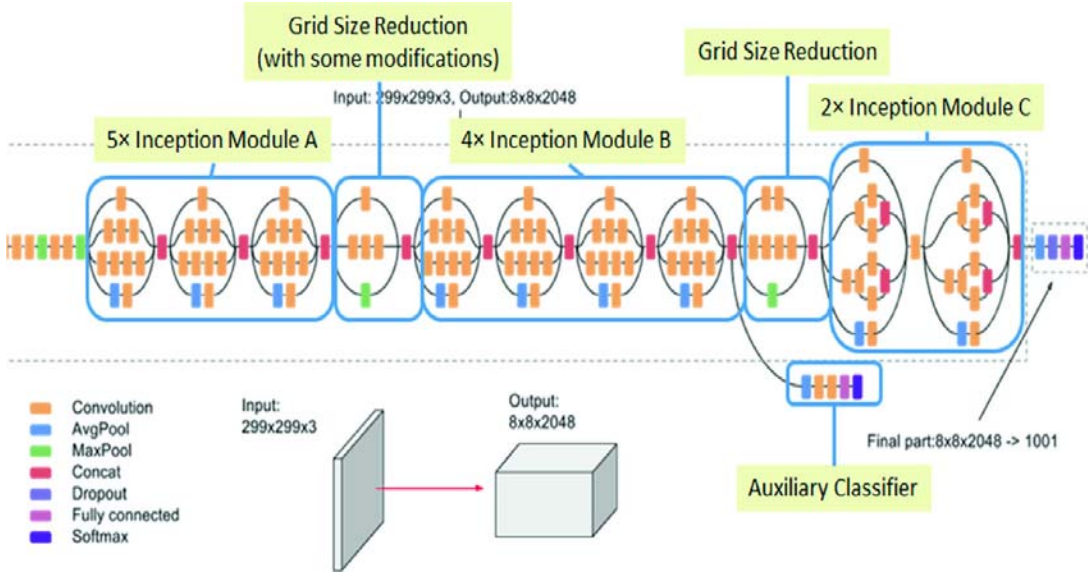
- 2개를 병렬로 수행한 뒤 합치는 것.
- 연산량은 좀 줄이면서 Conv 레이어를 통해 Representational Bottleneck을 줄인다.

3 Inception V3 (2016)

[출처1] https://norman3.github.io/papers/docs/google_inception.html
[출처2] https://hoya012.github.io/blog/deeplearning-classification-guidebook-2/

- Inception V3는 Inception V2 구조에 각종 기능을 추가한 것

- **RMSProp** : Optimizer 변경
- **Label Smoothing**
  - ✓ Target 값을 one-hot encoding을 사용하는 것이 아니라,
  - ✓ 값이 0 인 레이블에 대해서도 아주 작은 값 e를 배분하고 정답은 1−(n−1)∗e 로 값을 반영
- **Factorized 7-7**
  - ✓ conv 7x7 레이어를 (3x3)-(3x3) 2 레이어로 Factorization
- **BN(Batch Normalization)-auxiliary**
  - ✓ 마지막 Fully Connected 레이어에 Batch Normalization(BN) 적용



Network	Top-1 Error	Top-5 Error	Cost Bn Ops
GoogLeNet [20]	29%	9.2%	1.5
BN-GoogLeNet	26.8%	-	**1.5**
BN-Inception [7]	25.2%	7.8	2.0
Inception-v2	23.4%	-	3.8
Inception-v2 RMSProp	23.1%	6.3	3.8
Inception-v2 Label Smoothing	22.8%	6.1	3.8
Inception-v2 Factorized 7 × 7	21.6%	5.8	4.8
Inception-v2 BN-auxiliary	**21.2%**	**5.6%**	4.8

single-crop

Network	Crops Evaluated	Top-5 Error	Top-1 Error
GoogLeNet [20]	10	-	9.15%
GoogLeNet [20]	144	-	7.89%
VGG [18]	-	24.4%	6.8%
BN-Inception [7]	144	22%	5.82%
PReLU [6]	10	24.27%	7.38%
PReLU [6]	-	21.59%	5.71%
Inception-v3	12	19.47%	4.48%
Inception-v3	144	**18.77%**	**4.2%**

multi-crop

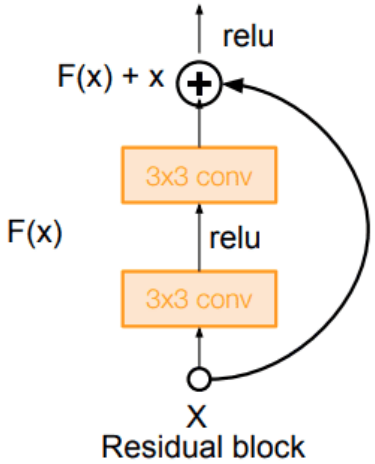
Network	Models Evaluated	Crops Evaluated	Top-1 Error	Top-5 Error
VGGNet [18]	2	-	23.7%	6.8%
GoogLeNet [20]	7	144	-	6.67%
PReLU [6]	-	-	-	4.94%
BN-Inception [7]	6	144	20.1%	4.9%
Inception-v3	4	144	**17.2%**	**3.58%***

4 ResNet: Deep Residual Learning for Image Recognition (2015)

- All the predescribed issues such as vanishing gradients were addressed with two tricks:
  - ✓ **batch normalization**
  - ✓ **short skip connection**
    https://theaisummer.com/skip-connections/

Is learning better networks as simple as stacking more layers?

- Instead of $H(x) = F(x)$, we ask them model to learn the difference (residual) $H'(x) = F(x) + x$, which means $\mathrm{H}(x) - x = F(x)$ will be the residual part [4].



Source: Standford 2017 Deep Learning Lectures: CNN architectures

- With that simple but yet effective block, the authors designed deeper architectures ranging from 18 (Resnet-18) to 150 (Resnet-150) layers.
- For the deepest models they adopted 1x1 convs, as illustrated on the right:

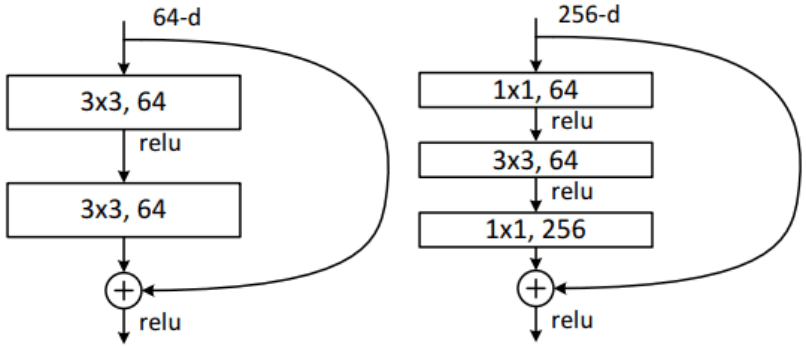


Image by Kaiming He et al. 2015.

Source: Deep Residual Learning for Image Recognition

The bottleneck layers (1×1) first reduce and then restore the channel dimensions, leaving the 3×3 layer with fewer input and output channels.

4 ResNet (2015)

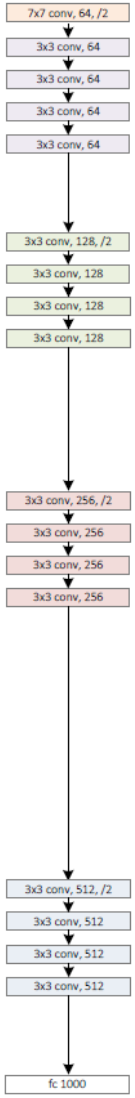
[출처]
- https://towardsdatascience.com/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection-e39402bfa5d8
- https://hohodu.tistory.com/23
- https://sike6054.github.io/blog/paper/first-post/
- https://jxnjxn.tistory.com/22

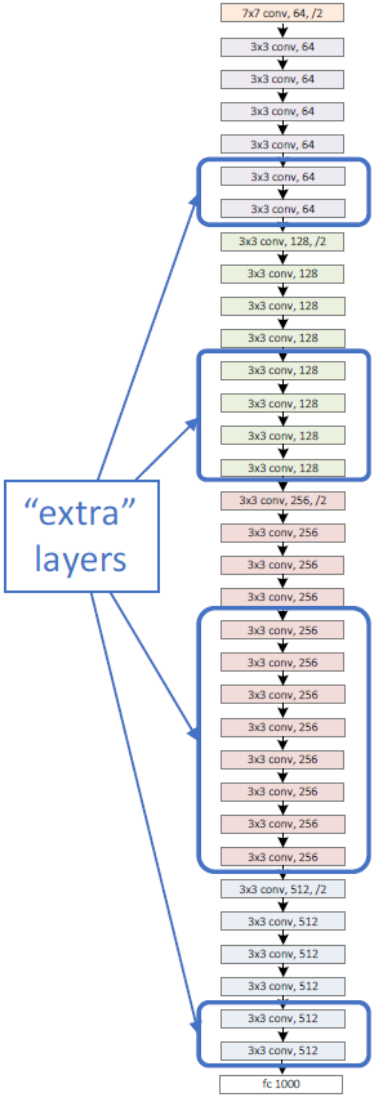
- Watch an awesome video from Henry AI Labs on ResNets:

- Problems of Plain Network (Vanishing/Exploding Gradient)
- Skip / Shortcut Connection in Residual Network (ResNet)
- ResNet Architecture
- Bottleneck Design
- Ablation Study
- Comparison with State-of-the-art Approaches (Image Classification)
- Comparison with State-of-the-art Approaches (Object Detection)

4 ResNet (2015)

A shallower model (18 layers)





A deeper counter (34 layers)

- Richer solution space
- A deeper model should not have **higher training error**
- A solution *by construction*:
  - original layers: copied from a learned shallower model
  - extra layers: set as identity
  - at least the same training error
- Optimization difficulties: solvers cannot find the solution when going deeper...

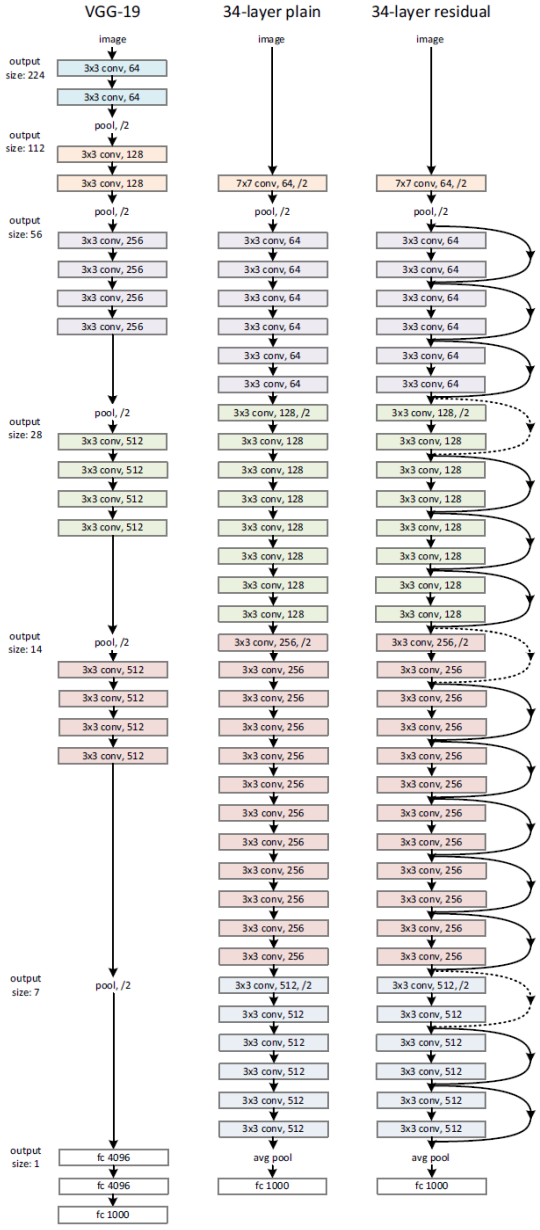


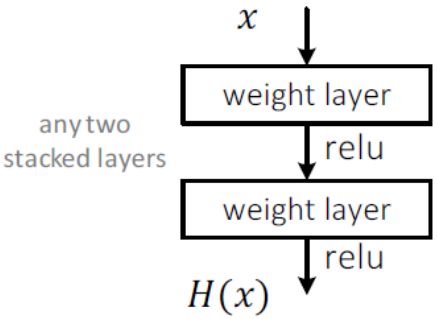
Figure 3. Example network architectures for ImageNet. Left: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. Middle: a ***plain network*** with 34 parameter layers (3.6 billion FLOPs). Right: a ***residual network*** with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. Table 1 shows more details and other variants.

4 ResNet (2015)

1) Problems of Plain Network (Vanishing/Exploding Gradient)

- **For conventional deep learning networks**, they usually have conv layers then fully connected (FC) layers for classification task like **AlexNet**, **ZFNet** and **VGGNet**, without any skip / shortcut connection, we call them ***plain networks*** here. **When the plain network is deeper (layers are increased), the problem of vanishing/exploding gradients occurs.**

- **Plain net**



$H(x)$ is any desired mapping,
Hope the 2 weights layers fit $H(x)$

- We expect deeper network will have more accurate prediction. However, below shows an example, **20-layer *plain network* got lower training error and test error than 56-layer plain network**, a **degradation problem** occurs due to **vanishing gradients**. (overfitting 문제가 아닌 것을 보여줌)

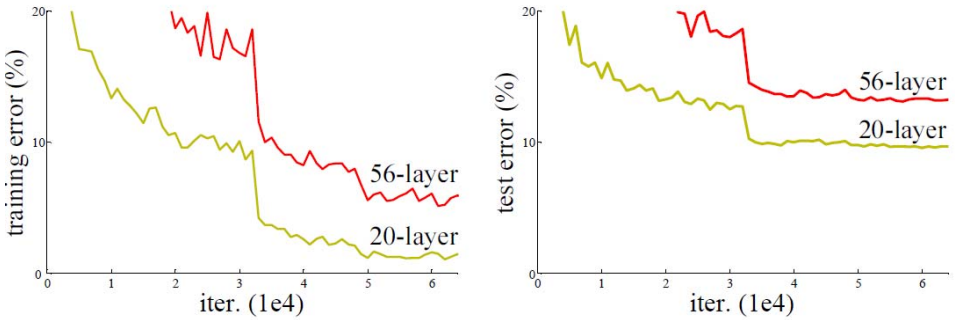


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

4 ResNet (2015)

1) Problems of Plain Network (Vanishing/Exploding Gradient)

Vanishing / Exploding Gradients

- During backpropagation, when partial derivative of the error function with respect to the current weight in each iteration of training, this has the effect of multiplying n of these small / large numbers to compute gradients of the “front” layers in an n-layer network.
- **Vanished** : When the network is deep, and multiplying n of these small numbers will become zero (vanished).
- **Exploded** : When the network is deep, and multiplying n of these large numbers will become too large (exploded).

Degradation Problem

- Gradient Vanishing /Exploding 문제 때문에 depth가 증가할지라도 어느 정도에 다다르면 성능이 떨어지는 현상을 "Degradation"라 함.
- Depth가 증가함에 따라 accuracy가 포화되어 degrade가 점점 빨라짐.
- Overfitting이 아니라, Model의 depth가 깊어짐에 따라 training error 높아짐

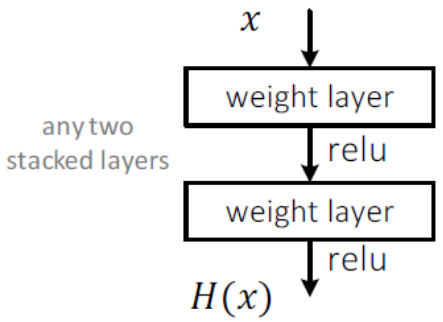
Deep residual learning

- 기존에 학습된 shallower architecture에다가 ***identity* mapping** 인 layer만 추가하여 deeper architecture를 고려하자. 실험을 통해 현재의 solver들은 이 constructed solution이나 그 이상의 성능을 보이는 solution을 찾을 수 없다는 것을 보여준다. (identity mapping : 기존 학습된 layers에서 생성된 output을 추가된 layers에서 동일한 output을 생성하는 것)
- Degradation 문제를 풀기 위한 **Deep residual learning** 제안 : To solve the problem of vanishing/exploding gradients, a **skip / shortcut connection** is added to add the input x to the output after few weight layers as below:

4 ResNet (2015)

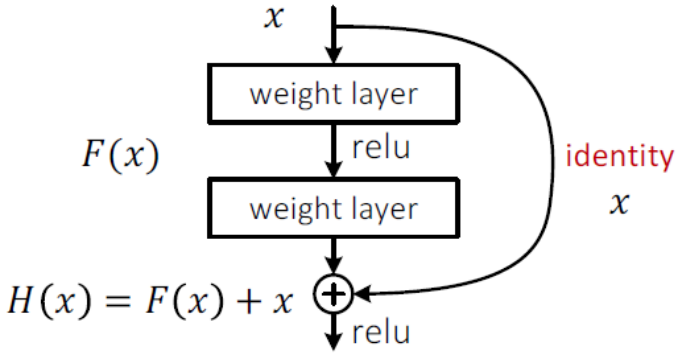
2) Skip / Shortcut Connection in Residual Network (ResNet)

Plain net



$H(x)$ is any desired mapping,
Hope the 2 weights layers fit $H(x)$

Residual net



$H(x)$ is any desired mapping,
~~Hope the 2 weight layers fit $H(x)$~~
Hope the 2 weight layers fit $F(x)$
let $H(x) = F(x) + x$

$F(x)$ is residual mapping with respect to identity

- If identity were optimal, easy to set weights as o.
- If optimal mapping is closer to identity, easier to find small fluctuations. 다음페이지 참조

❖ Short connection은 한 개 이상의 layer를 skipping 하는 것, direct mapping을 하는 대신에 identity mapping을 함. Identity shortcut connection은 extra parameter를 추가하는 것도 아니고, computational complexity가 증가하지 않게 된다. 그 전체 network는 SGD와 backpropagation을 통해 학습될 수 있다.

- Hence, the output $H(x) = F(x) + x$. The weight layers actually is to learn a kind of residual mapping: $F(x) = H(x) - x$.
- Even if there is vanishing gradient for the weight layers, we always still have the identity x to transfer back to earlier layers.

4 ResNet (2015)

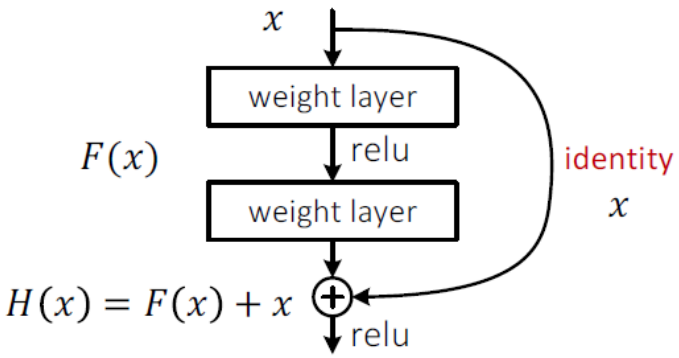
Residual Learning

- Few stacked layers마다 residual learning을 사용함.
- stacked layer를 H(x)-x에 mapping함으로써 original mapping을 F(x)+x로 reformulation하는 것은, 성능 저하 문제를 해결하기 위함이다.
- 실제 상황에서 H(x)의 optimal이 identity mapping이 아닐지라도, 이 reformulation은 문제에 precondition을 제공하는 효과를 준다. 만약 optimal function이 zero mapping보다 identity mapping에 더 가깝다면, solver가 identity mapping을 참조하여 작은 변화 F(x)를 학습하는 것이 새로운 function을 생으로 학습하는 것보다 쉬울 것이다. 실험에서는 학습된 residual function에서 일반적으로 작은 반응이 있다는 결과를 보여준다(Fig.7 참조). 이 결과는 identity mapping이 합리적인 preconditioning을 제공한다는 것을 시사한다.

4 ResNet (2015)

Identity (x) Mapping by Shortcuts

Residual net



$F(x)$ is residual mapping with respect to identity

```
shorcut = x

out = self.conv1(x)
out = self.bn1(out)
out = self.relu(out)

out = self.conv2(out)
out = self.bn2(out)

out += shortcut
out = self.relu(out)
```

- A defined building block

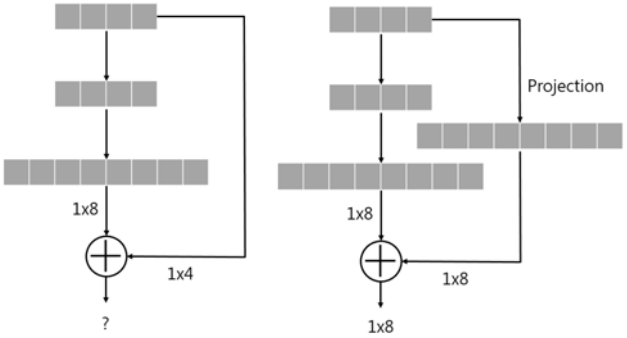
$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}$$

✓ $\mathbf{x}$와 $\boldsymbol{y}$는 각각 building block에서 input과 output이다. $\mathcal{F}(\mathbf{x}, \{W_i\})$는 학습 되어야 할 residual mapping을 나타낸다. Fig.2와 같이 layer가 두 개 있는 경우를 예로 들면, $\mathcal{F} = W_2\sigma(W_1\mathbf{x})$ 로 나타낼 수 있다. 여기서 σ는 ReLU를 나타내며, bias는 표기법 간소화를 위해 생략된다. $\mathcal{F} + \mathbf{x}$ 연산은 shortcut connection 및 element-wise addition으로 수행되며, addition 후에는 second nonlinearity로 ReLU를 적용한다.

- $\mathcal{F}, \mathbf{x}$의 Dimension이 같아야 하며, 이를 위해 linear projection W_s을 수행할 수 있다

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x}$$

✓ W_s는 dimension matching 용도로만 사용 (Degradation 문제 해결에 identity mapping만으로 충분함을 실험에 보여줌)



$FC\ layer$ 상에서는 이와 같이 $node$의 개수만 맞추면 되지만, $conv\ layer$의 경우에는 $feature\ map\ size$와 $channel$의 개수까지 맞춰야 한다.

4 ResNet (2015)

3) ResNet Architecture

1) The VGG-19 (left) is a state-of-the-art approach in ILSVRC 2014.
2) 34-layer plain network (middle) is treated as the deeper network of VGG-19, i.e. more conv layers.
3) 34-layer residual network (ResNet) (right) is the plain one with addition of skip / shortcut connection.

Plain network는 VGG의 초반 몇 개의 3x3 convolution을 7x7 convolution으로 대체했다. 또한 global average pooling을 마지막 convolutional layer 출력에 적용하기 때문에 VGG에 비해 fully connected layer가 적다.

ResNet은 plain network에서 두 개의 convolutional block을 skipping하는 shortcut connection을 추가한 것이다.

- identity shortcut (수식1)은 intput과 outpu이 동일한 dimension인 경우 (실선)
- dimension이 증가할 경우 (점선) : feature map이 2 size씩 건너뛰기 때문에 stride을 2로 사용
  - ✓ identity shortcut connection 계속 실행하는 경우 -- zero padding
  - ✓ projection shortcut connection은 1x1 conv 사용

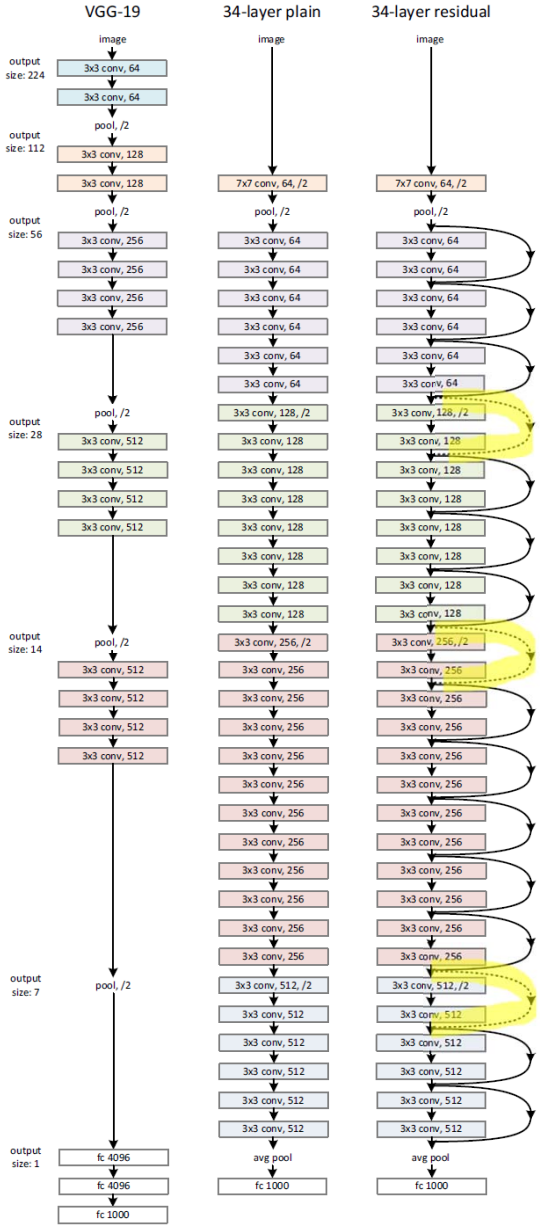


Figure 3. Example network architectures for ImageNet. Left: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. Middle: a ***plain network*** with 34 parameter layers (3.6 billion FLOPs). Right: a ***residual network*** with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. Table 1 shows more details and other variants.

4 ResNet (2015)

3) ResNet Architecture

Implementation

- 이미지는 scale augmentation를 위해 [256, 480]에서 무작위하게 샘플링 된 shorter side를 사용하여 rescaling된다. 224x224 crop은 horizontal flip with per-pixel mean subtracted 이미지 중에 무작위로 샘플링 되며, standard color augmentation도 사용된다.
- 각각의 conv layer와 activation 사이에는 batch normalization을 사용하며, He initialization 기법으로 weight를 초기화하여 모든 plain/residual nets을 학습한다.
- batch normalization에 근거해 dropout을 사용하지 않는다.
- learning rate는 0.1에서 시작하여, error plateau 상태마다 rate를 10으로 나누어 적용하며, decay는 0.0001, momentum은 0.9로 한 SGD를 사용했다.
- mini-batch size는 256로 했으며, iteration은 총 600K회 수행된다.

4) Bottleneck Design

- Since the network is very deep now, the time complexity is high. A bottleneck design is used to reduce the complexity as follows:

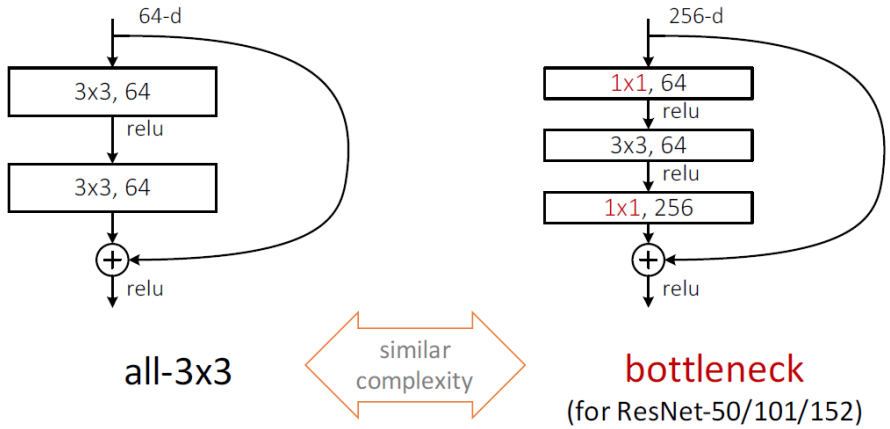


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

- 50-layer ResNet : 34-layer ResNet의 2-layer block들을 3-layer bottleneck block으로 대체하여 50-layer ResNet을 구성했다. dimension matching을 위해서는 위의 옵션 2를 사용한다. 이 모델은 3.8 billion FLOPs 이다.
- 101-layer and 152-layer ResNets : 여기에 3-layer bottleneck block을 추가하여 101-layer 및 152-layer ResNet을 구성했다. depth가 상당히 증가했음에도 상당히 높은 정확도가 결과로 나왔다. depth의 이점이 모든 evaluation metrics에서 발견됐다.

4 ResNet (2015)

- PyTorch code: **https://dnddnjs.github.io/cifar10/2018/10/09/resnet/**

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

class IdentityPadding(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super(IdentityPadding, self).__init__()

        self.pooling = nn.MaxPool2d(1, stride=stride)
        self.add_channels = out_channels - in_channels

    def forward(self, x):
        out = F.pad(x, (0, 0, 0, 0, 0, self.add_channels))
        out = self.pooling(out)
        return out

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, down_sample=False):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.stride = stride

        if down_sample:
            self.down_sample = IdentityPadding(in_channels, out_channels, stride)
        else:
            self.down_sample = None
```

```
    def forward(self, x):
        shortcut = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.down_sample is not None:
            shortcut = self.down_sample(x)

        out += shortcut
        out = self.relu(out)
        return out

class ResNet(nn.Module):
    def __init__(self, num_layers, block, num_classes=10):
        super(ResNet, self).__init__()
        self.num_layers = num_layers

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)

        # feature map size = 32x32x16
        self.layers_2n = self.get_layers(block, 16, 16, stride=1)
        # feature map size = 16x16x32
        self.layers_4n = self.get_layers(block, 16, 32, stride=2)
        # feature map size = 8x8x64
        self.layers_6n = self.get_layers(block, 32, 64, stride=2)
```

4 ResNet (2015)

- PyTorch code: **https://dnddnjs.github.io/cifar10/2018/10/09/resnet/**

```
        # output layers
        self.avg_pool = nn.AvgPool2d(8, stride=1)
        self.fc_out = nn.Linear(64, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
                                        nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def get_layers(self, block, in_channels, out_channels, stride):
        if stride == 2:
            down_sample = True
        else:
            down_sample = False

        layers_list = nn.ModuleList(
            [block(in_channels, out_channels, stride, down_sample)])

        for _ in range(self.num_layers - 1):
            layers_list.append(block(out_channels, out_channels))

        return nn.Sequential(*layers_list)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layers_2n(x)
        x = self.layers_4n(x)
        x = self.layers_6n(x)

        x = self.avg_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc_out(x)
        return x

def resnet():
    block = ResidualBlock
    # total number of layers if 6n + 2. if n is 5 then the depth of network is 32.
    model = ResNet(5, block)
    return model
```

4 ResNet (2015)

ResNet | PyTorch : https://pytorch-tutorial.readthedocs.io/en/latest/tutorial/chapter03_intermediate/3_2_2_cnn_resnet_cifar10/

```
# ---------------------------------------------------------------------- #
# An implementation of https://arxiv.org/pdf/1512.03385.pdf              #
# See section 4.2 for the model architecture on CIFAR-10                 #
# Some part of the code was referenced from below                        #
# https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py   #
# ---------------------------------------------------------------------- #

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
num_epochs = 80
learning_rate = 0.001

# Image preprocessing modules
transform = transforms.Compose([
    transforms.Pad(4),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
    transforms.ToTensor()])

# CIFAR-10 dataset
train_dataset = torchvision.datasets.CIFAR10(root='../../data/',
                                             train=True,
                                             transform=transform,
                                             download=True)
```

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

# Data loader
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)

# 3x3 convolution
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)

# Residual block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample
```

4 ResNet (2015)

ResNet | PyTorch : https://pytorch-tutorial.readthedocs.io/en/latest/tutorial/chapter03_intermediate/3_2_2_cnn_resnet_cifar10/

```
    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)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out

# ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[1], 2)
        self.layer3 = self.make_layer(block, 64, layers[2], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, num_classes)
```

```
    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

model = ResNet(ResidualBlock, [2, 2, 2]).to(device)

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

4 ResNet (2015)

ResNet | PyTorch : https://pytorch-tutorial.readthedocs.io/en/latest/tutorial/chapter03_intermediate/3_2_2_cnn_resnet_cifar10/

```
# For updating learning rate
def update_lr(optimizer, lr):
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr

# Train the model
total_step = len(train_loader)
curr_lr = learning_rate
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

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

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

        if (i+1) % 100 == 0:
            print ("Epoch [{}/{}], Step [{}/{}] Loss: {:.4f}"
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))

    # Decay learning rate
    if (epoch+1) % 20 == 0:
        curr_lr /= 3
        update_lr(optimizer, curr_lr)
```

```
# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Accuracy of the model on the test images: {} %'.format(100 * correct / total))

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

4 Inception V4, Inception-ResNet (2016)

Paper : Inception-v4, Inception-RestNet and the Impact of Residual Connections on Learning
[참고] https://norman3.github.io/papers/docs/google_inception.html

Version
- Inception v4 : Inception v3 확장
- Inception v3 + resnet : Inception-resnet v1
- Inception v4 + resnet : Inception-resnet v2

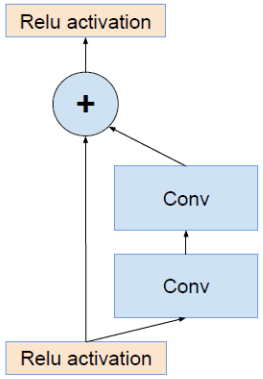


Figure 1. Residual connections as introduced in He et al. [5].

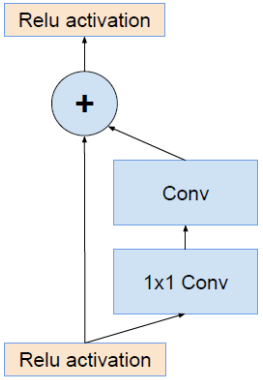


Figure 2. Optimized version of ResNet connections by [5] to shield computation.

- 그림 1. Residual connection
- 그림 2. 1x1 conv 추가하여 연산량 줄인 residual connection

Inception v4 Network

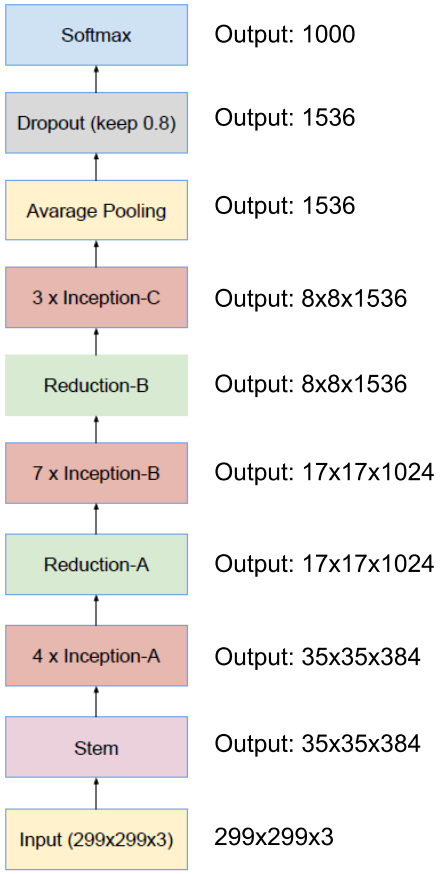


Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

- Inception v3와 거의 유사한 형태이나, 세부 inception 레이어가 조금 다름

4 Inception V4, Inception-ResNet (2016)

Inception v4

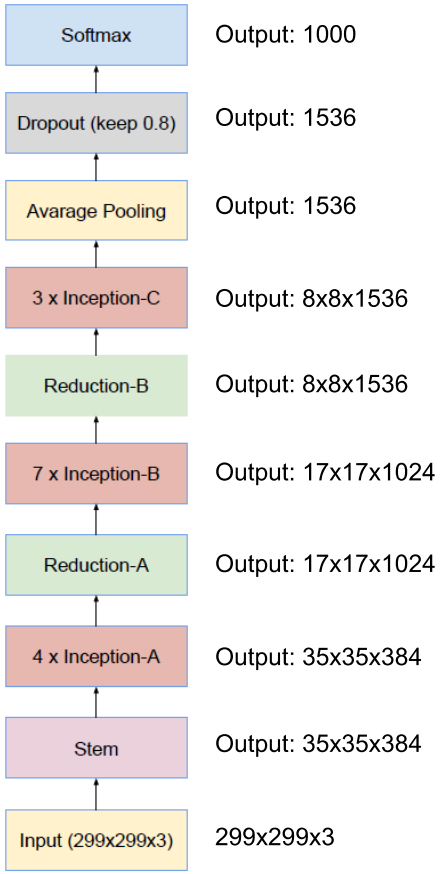


Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

Inception-Resnet V1,V2

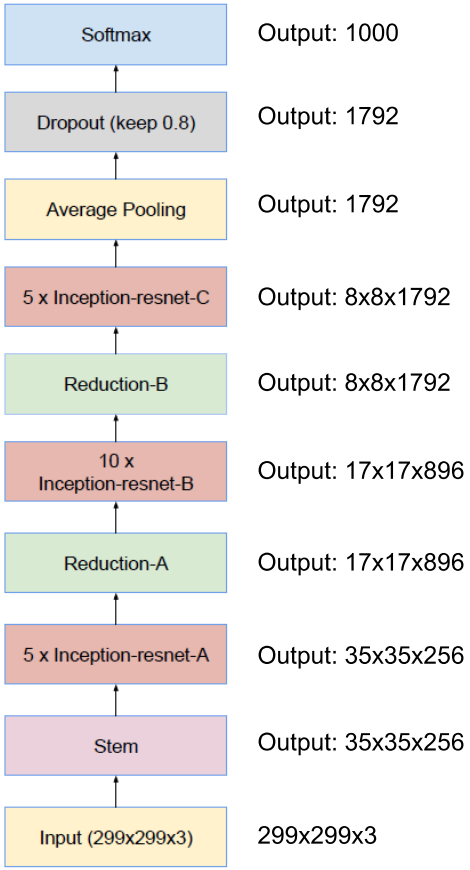


Figure 15. Schema for Inception-ResNet-v1 and Inception-ResNet-v2 networks. This schema applies to both networks but the underlying components differ. Inception-ResNet-v1 uses the blocks as described in Figures 14, 10, 7, 11, 12 and 13. Inception-ResNet-v2 uses the blocks as described in Figures 3, 16, 7, 17, 18 and 19. The output sizes in the diagram refer to the activation vector tensor shapes of Inception-ResNet-v1.

4 Inception V4, Inception-ResNet (2016)

1) Inception v4

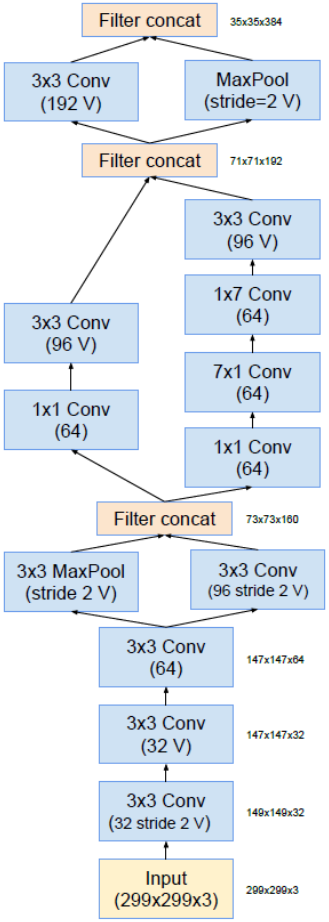


Figure 3. The schema for stem of the pure Inception-v4 and Inception-ResNet-v2 networks. This is the input part of those networks. Cf. Figures 9 and 15

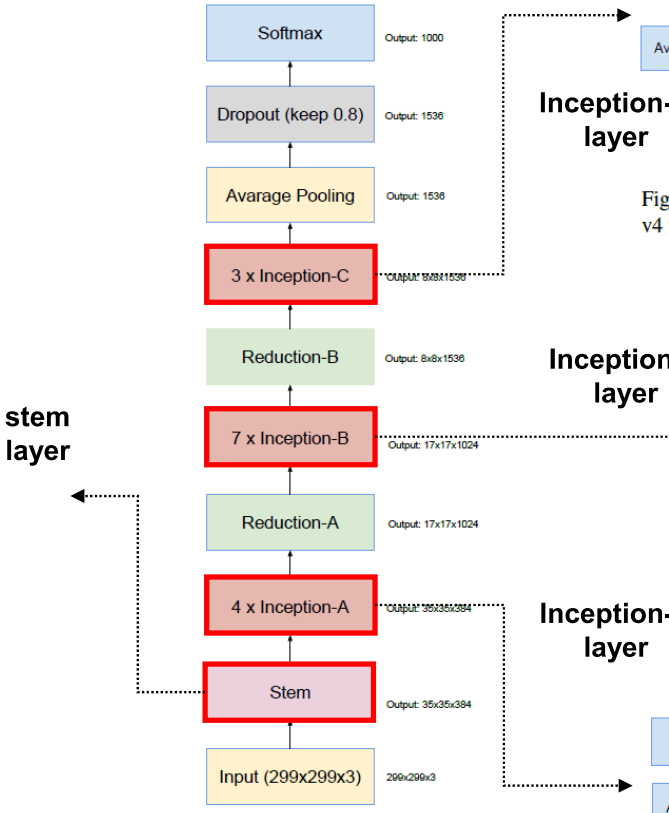


Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

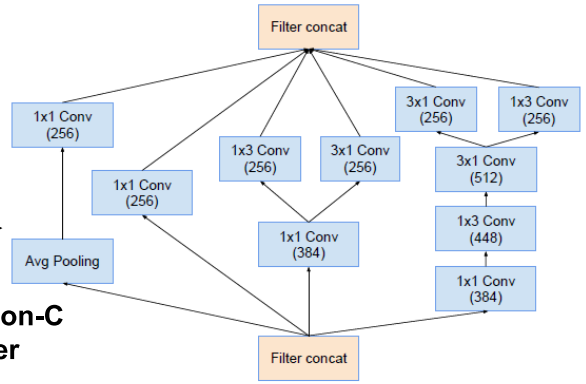


Figure 6. The schema for 8×8 grid modules of the pure Inception-v4 network. This is the Inception-C block of Figure 9.

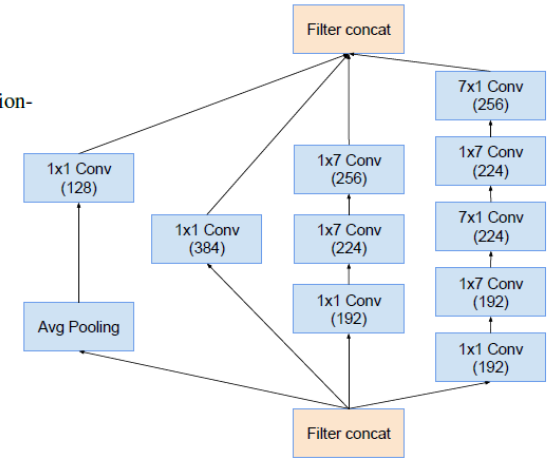


Figure 5. The schema for 17×17 grid modules of the pure Inception-v4 network. This is the Inception-B block of Figure 9.

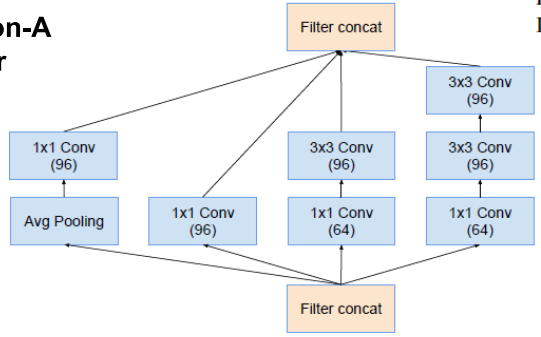


Figure 4. The schema for 35×35 grid modules of the pure Inception-v4 network. This is the Inception-A block of Figure 9.

Inception 모듈은 모두 입출력 크기 변화가 없음

4 Inception V4, Inception-ResNet (2016)

1) Inception v4

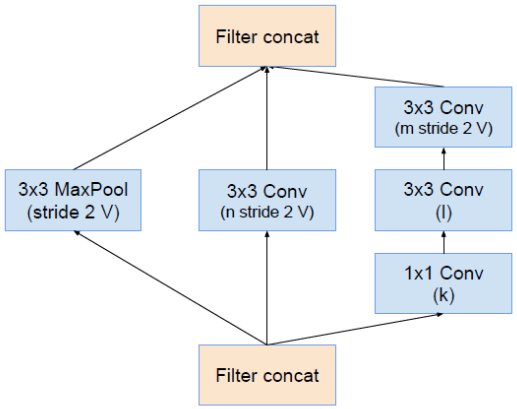


Figure 7. The schema for 35×35 to 17×17 reduction module. Different variants of this blocks (with various number of filters) are used in Figure 9, and 15 in each of the new Inception(-v4, -ResNet-v1, -ResNet-v2) variants presented in this paper. The k, l, m, n numbers represent filter bank sizes which can be looked up in Table 1.

Reduction Module

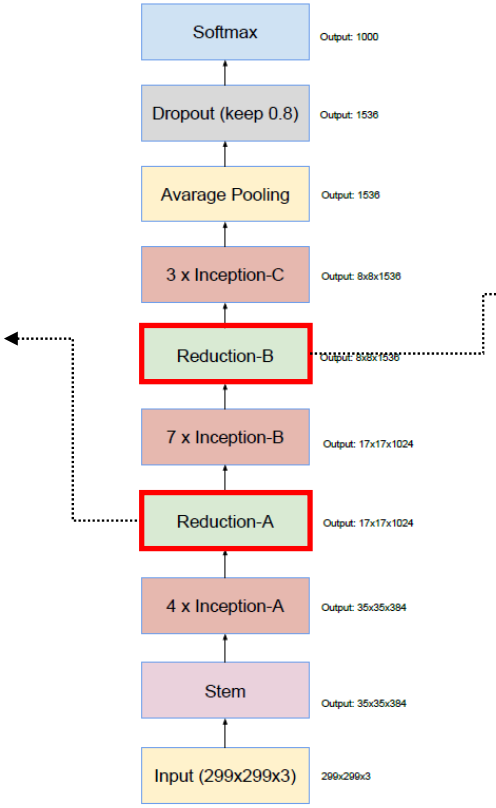


Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

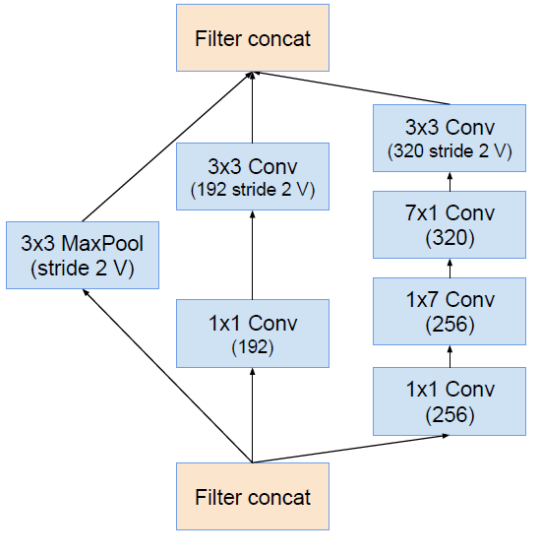


Figure 8. The schema for 17×17 to 8×8 grid-reduction module. This is the reduction module used by the pure Inception-v4 network in Figure 9.

4 Inception V4, Inception-ResNet (2016)

2) Inception ResNet v1

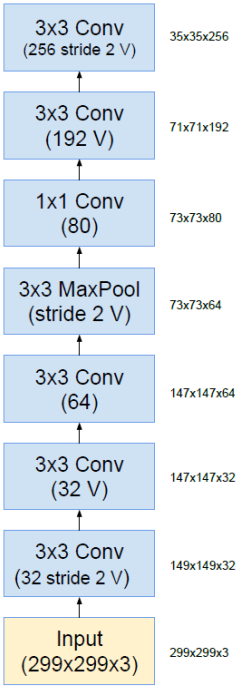


Figure 14. The stem of the Inception-ResNet-v1 network.

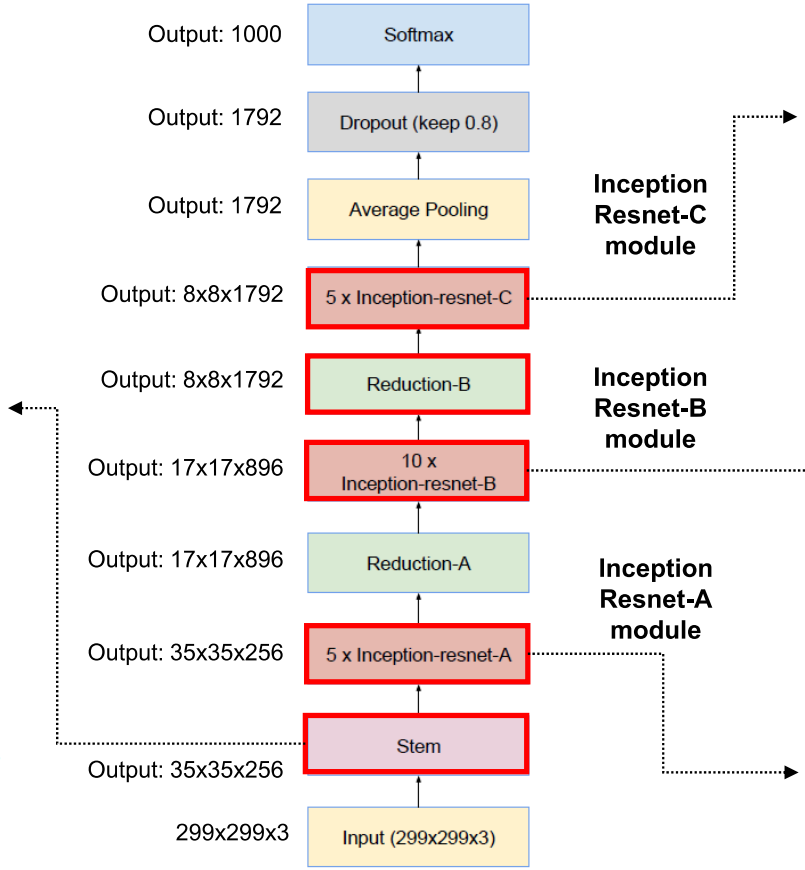


Figure 15. Schema for Inception-ResNet-v1 and Inception-ResNet-v2 networks. This schema applies to both networks but the underlying components differ. Inception-ResNet-v1 uses the blocks as described in Figures 14, 10, 7, 11, 12 and 13. Inception-ResNet-v2 uses the blocks as described in Figures 3, 16, 7, 17, 18 and 19. The output sizes in the diagram refer to the activation vector tensor shapes of Inception-ResNet-v1.

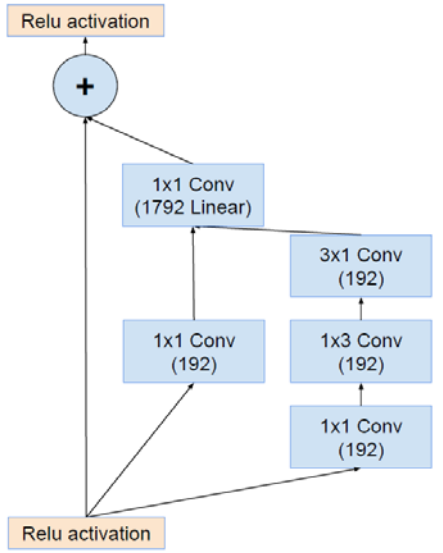


Figure 13. The schema for 8×8 grid (Inception-ResNet-C) module of Inception-ResNet-v1 network.

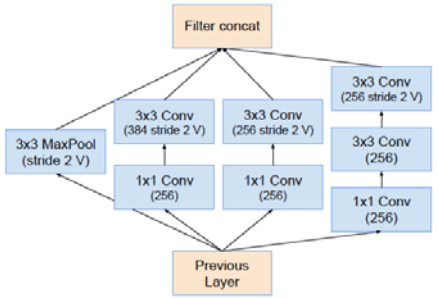


Figure 12. "Reduction-B" 17 × 17 to 8 × 8 grid-reduction module. This module used by the smaller Inception-ResNet-v1 network in Figure 15.

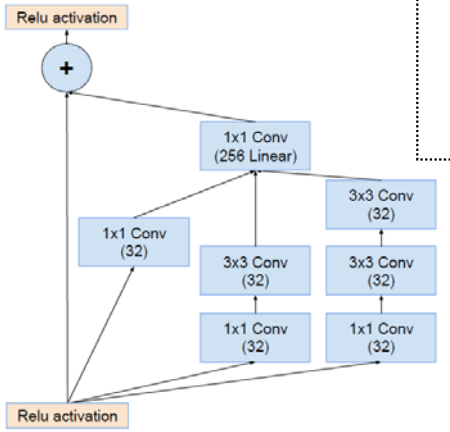


Figure 10. The schema for 35 × 35 grid (Inception-ResNet-A) module of Inception-ResNet-v1 network.

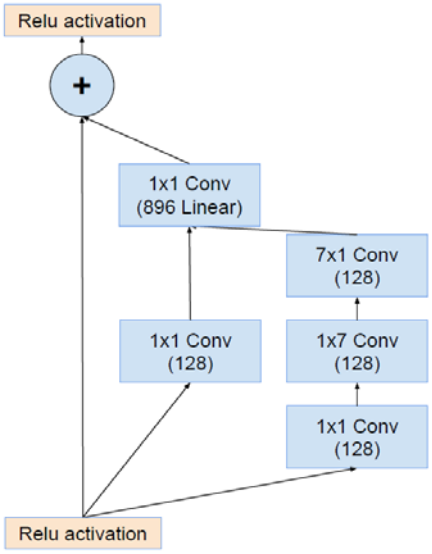
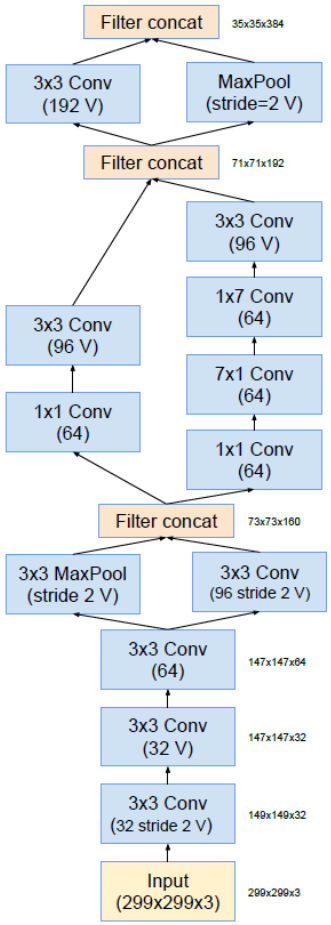


Figure 11. The schema for 17 × 17 grid (Inception-ResNet-B) module of Inception-ResNet-v1 network.

4 Inception V4, Inception-ResNet (2016)

2) Inception Restnet v2



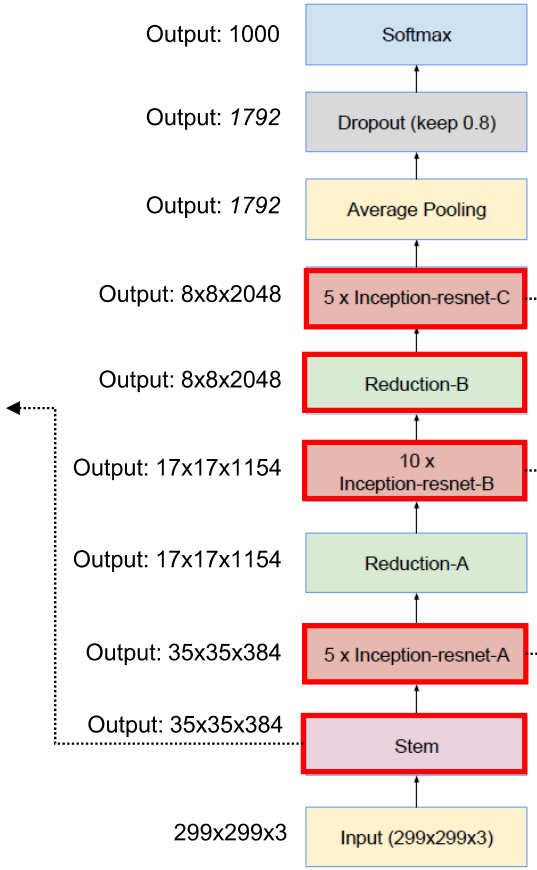


Figure 15. Schema for Inception-ResNet-v1 and Inception-ResNet-v2 networks. This schema applies to both networks but the underlying components differ. Inception-ResNet-v1 uses the blocks as described in Figures 14, 10, 7, 11, 12 and 13. Inception-ResNet-v2 uses the blocks as described in Figures 3, 16, 7, 17, 18 and 19. The output sizes in the diagram refer to the activation vector tensor shapes of Inception-ResNet-v1.

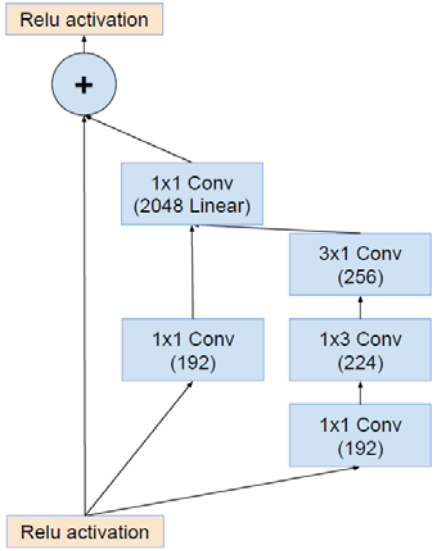


Figure 19. The schema for 8×8 grid (Inception-ResNet-C) module of the Inception-ResNet-v2 network.

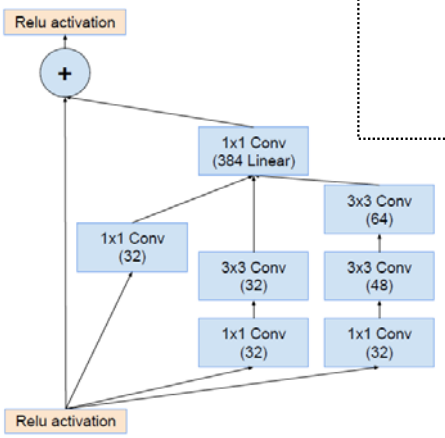


Figure 16. The schema for 35 × 35 grid (Inception-ResNet-A) module of the Inception-ResNet-v2 network.

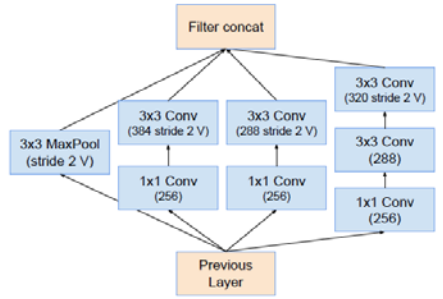


Figure 18. The schema for 17 × 17 to 8 × 8 grid-reduction module. Reduction-B module used by the wider Inception-ResNet-v1 network in Figure 15.

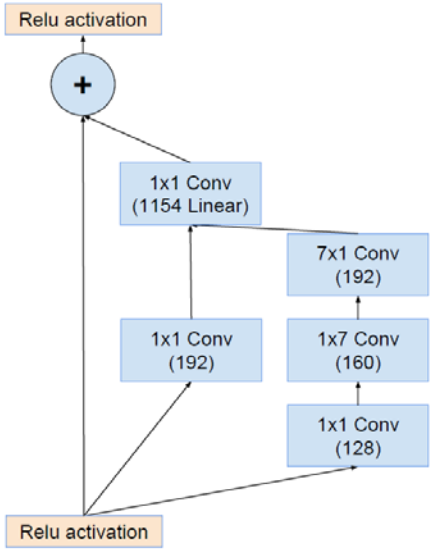


Figure 17. The schema for 17 × 17 grid (Inception-ResNet-B) module of the Inception-ResNet-v2 network.

4 Inception V4, Inception-ResNet (2016)

Inception v4 | PyTorch : https://github.com/zhulf0804/Inceptionv4_and_Inception-ResNetv2.PyTorch

```
import torch
import torch.nn as nn

class Conv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, padding, stride=1, bias=True):
        super(Conv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride=stride, padding=padding, bias=bias)
        self.bn = nn.BatchNorm2d(out_channels, eps=0.001, momentum=0.1)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        return x
```

```
class Reduction_A(nn.Module):
    # 35 -> 17
    def __init__(self, in_channels, k, l, m, n):
        super(Reduction_A, self).__init__()
        self.branch_0 = Conv2d(in_channels, n, 3, stride=2, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, k, 1, stride=1, padding=0, bias=False),
            Conv2d(k, l, 3, stride=1, padding=1, bias=False),
            Conv2d(l, m, 3, stride=2, padding=0, bias=False),
        )
        self.branch_2 = nn.MaxPool2d(3, stride=2, padding=0)

    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        return torch.cat((x0, x1, x2), dim=1) # 17 x 17 x 1024
```

4 Inception V4, Inception-ResNet (2016)

Inception v4 | PyTorch : https://github.com/zhulf0804/Inceptionv4_and_Inception-ResNetv2.PyTorch

```
import torch
import torch.nn as nn
from model.units import Conv2d, Reduction_A

class Stem(nn.Module):
    def __init__(self, in_channels):
        super(Stem, self).__init__()
        self.conv2d_1a_3x3 = Conv2d(in_channels, 32, 3, stride=2, padding=0, bias=False)

        self.conv2d_2a_3x3 = Conv2d(32, 32, 3, stride=1, padding=0, bias=False)
        self.conv2d_2b_3x3 = Conv2d(32, 64, 3, stride=1, padding=1, bias=False)

        self.mixed_3a_branch_0 = nn.MaxPool2d(3, stride=2, padding=0)
        self.mixed_3a_branch_1 = Conv2d(64, 96, 3, stride=2, padding=0, bias=False)

        self.mixed_4a_branch_0 = nn.Sequential(
            Conv2d(160, 64, 1, stride=1, padding=0, bias=False),
            Conv2d(64, 96, 3, stride=1, padding=0, bias=False),
        )
        self.mixed_4a_branch_1 = nn.Sequential(
            Conv2d(160, 64, 1, stride=1, padding=0, bias=False),
            Conv2d(64, 64, (1, 7), stride=1, padding=(0, 3), bias=False),
            Conv2d(64, 64, (7, 1), stride=1, padding=(3, 0), bias=False),
            Conv2d(64, 96, 3, stride=1, padding=0, bias=False)
        )

        self.mixed_5a_branch_0 = Conv2d(192, 192, 3, stride=2, padding=0, bias=False)
        self.mixed_5a_branch_1 = nn.MaxPool2d(3, stride=2, padding=0)
```

```
def forward(self, x):
        x = self.conv2d_1a_3x3(x) # 149 x 149 x 32
        x = self.conv2d_2a_3x3(x) # 147 x 147 x 32
        x = self.conv2d_2b_3x3(x) # 147 x 147 x 64
        x0 = self.mixed_3a_branch_0(x)
        x1 = self.mixed_3a_branch_1(x)
        x = torch.cat((x0, x1), dim=1) # 73 x 73 x 160
        x0 = self.mixed_4a_branch_0(x)
        x1 = self.mixed_4a_branch_1(x)
        x = torch.cat((x0, x1), dim=1) # 71 x 71 x 192
        x0 = self.mixed_5a_branch_0(x)
        x1 = self.mixed_5a_branch_1(x)
        x = torch.cat((x0, x1), dim=1) # 35 x 35 x 384
        return x
```

4 Inception V4, Inception-ResNet (2016)

Inception v4 | PyTorch : https://github.com/zhulf0804/Inceptionv4_and_Inception-ResNetv2.PyTorch

```
class Inception_A(nn.Module):
    def __init__(self, in_channels):
        super(Inception_A, self).__init__()
        self.branch_0 = Conv2d(in_channels, 96, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 64, 1, stride=1, padding=0, bias=False),
            Conv2d(64, 96, 3, stride=1, padding=1, bias=False),
        )
        self.branch_2 = nn.Sequential(
            Conv2d(in_channels, 64, 1, stride=1, padding=0, bias=False),
            Conv2d(64, 96, 3, stride=1, padding=1, bias=False),
            Conv2d(96, 96, 3, stride=1, padding=1, bias=False),
        )
        self.brance_3 = nn.Sequential(
            nn.AvgPool2d(3, 1, padding=1, count_include_pad=False),
            Conv2d(384, 96, 1, stride=1, padding=0, bias=False)
        )

    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        x3 = self.brance_3(x)
        return torch.cat((x0, x1, x2, x3), dim=1)
```

```
class Inception_B(nn.Module):
    def __init__(self, in_channels):
        super(Inception_B, self).__init__()
        self.branch_0 = Conv2d(in_channels, 384, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False),
            Conv2d(192, 224, (1, 7), stride=1, padding=(0, 3), bias=False),
            Conv2d(224, 256, (7, 1), stride=1, padding=(3, 0), bias=False),
        )
        self.branch_2 = nn.Sequential(
            Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False),
            Conv2d(192, 192, (7, 1), stride=1, padding=(3, 0), bias=False),
            Conv2d(192, 224, (1, 7), stride=1, padding=(0, 3), bias=False),
            Conv2d(224, 224, (7, 1), stride=1, padding=(3, 0), bias=False),
            Conv2d(224, 256, (1, 7), stride=1, padding=(0, 3), bias=False)
        )
        self.branch_3 = nn.Sequential(
            nn.AvgPool2d(3, stride=1, padding=1, count_include_pad=False),
            Conv2d(in_channels, 128, 1, stride=1, padding=0, bias=False)
        )
    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        x3 = self.branch_3(x)
        return torch.cat((x0, x1, x2, x3), dim=1)
```

4 Inception V4, Inception-ResNet (2016)

Inception v4 | PyTorch : https://github.com/zhulf0804/Inceptionv4_and_Inception-ResNetv2.PyTorch

```
class Reduction_B(nn.Module):
    # 17 -> 8
    def __init__(self, in_channels):
        super(Reduction_B, self).__init__()
        self.branch_0 = nn.Sequential(
            Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False),
            Conv2d(192, 192, 3, stride=2, padding=0, bias=False),
        )
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 256, 1, stride=1, padding=0, bias=False),
            Conv2d(256, 256, (1, 7), stride=1, padding=(0, 3), bias=False),
            Conv2d(256, 320, (7, 1), stride=1, padding=(3, 0), bias=False),
            Conv2d(320, 320, 3, stride=2, padding=0, bias=False)
        )
        self.branch_2 = nn.MaxPool2d(3, stride=2, padding=0)

    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        return torch.cat((x0, x1, x2), dim=1)  # 8 x 8 x 1536
```

```
class Inception_C(nn.Module):
    def __init__(self, in_channels):
        super(Inception_C, self).__init__()
        self.branch_0 = Conv2d(in_channels, 256, 1, stride=1, padding=0, bias=False)

        self.branch_1 = Conv2d(in_channels, 384, 1, stride=1, padding=0, bias=False)
        self.branch_1_1 = Conv2d(384, 256, (1, 3), stride=1, padding=(0, 1), bias=False)
        self.branch_1_2 = Conv2d(384, 256, (3, 1), stride=1, padding=(1, 0), bias=False)

        self.branch_2 = nn.Sequential(
            Conv2d(in_channels, 384, 1, stride=1, padding=0, bias=False),
            Conv2d(384, 448, (3, 1), stride=1, padding=(1, 0), bias=False),
            Conv2d(448, 512, (1, 3), stride=1, padding=(0, 1), bias=False),
        )
        self.branch_2_1 = Conv2d(512, 256, (1, 3), stride=1, padding=(0, 1), bias=False)
        self.branch_2_2 = Conv2d(512, 256, (3, 1), stride=1, padding=(1, 0), bias=False)

        self.branch_3 = nn.Sequential(
            nn.AvgPool2d(3, stride=1, padding=1, count_include_pad=False),
            Conv2d(in_channels, 256, 1, stride=1, padding=0, bias=False)
        )

    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x1_1 = self.branch_1_1(x1)
        x1_2 = self.branch_1_2(x1)
        x1 = torch.cat((x1_1, x1_2), 1)
        x2 = self.branch_2(x)
        x2_1 = self.branch_2_1(x2)
        x2_2 = self.branch_2_2(x2)
        x2 = torch.cat((x2_1, x2_2), dim=1)
        x3 = self.branch_3(x)
        return torch.cat((x0, x1, x2, x3), dim=1) # 8 x 8 x 1536
```

4 Inception V4, Inception-ResNet (2016)

Inception-ResNet v2 | PyTorch : https://github.com/zhulf0804/Inceptionv4_and_Inception-ResNetv2.PyTorch

```
import torch
import torch.nn as nn
from model.units import Conv2d, Reduction_A

class Stem(nn.Module):
    def __init__(self, in_channels):
        super(Stem, self).__init__()
        self.features = nn.Sequential(
            Conv2d(in_channels, 32, 3, stride=2, padding=0, bias=False), # 149 x 149 x 32
            Conv2d(32, 32, 3, stride=1, padding=0, bias=False), # 147 x 147 x 32
            Conv2d(32, 64, 3, stride=1, padding=1, bias=False), # 147 x 147 x 64
            nn.MaxPool2d(3, stride=2, padding=0), # 73 x 73 x 64
            Conv2d(64, 80, 1, stride=1, padding=0, bias=False), # 73 x 73 x 80
            Conv2d(80, 192, 3, stride=1, padding=0, bias=False), # 71 x 71 x 192
            nn.MaxPool2d(3, stride=2, padding=0), # 35 x 35 x 192
        )
        self.branch_0 = Conv2d(192, 96, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(192, 48, 1, stride=1, padding=0, bias=False),
            Conv2d(48, 64, 5, stride=1, padding=2, bias=False),
        )
        self.branch_2 = nn.Sequential(
            Conv2d(192, 64, 1, stride=1, padding=0, bias=False),
            Conv2d(64, 96, 3, stride=1, padding=1, bias=False),
            Conv2d(96, 96, 3, stride=1, padding=1, bias=False),
        )
        self.branch_3 = nn.Sequential(
            nn.AvgPool2d(3, stride=1, padding=1, count_include_pad=False),
            Conv2d(192, 64, 1, stride=1, padding=0, bias=False)
        )
```

```
def forward(self, x):
        x = self.features(x)
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        x3 = self.branch_3(x)
        return torch.cat((x0, x1, x2, x3), dim=1)
```

4 Inception V4, Inception-ResNet (2016)

Inception-ResNet v2 | PyTorch : https://github.com/zhulf0804/Inceptionv4_and_Inception-ResNetv2.PyTorch

```
class Inception_ResNet_A(nn.Module):
    def __init__(self, in_channels, scale=1.0):
        super(Inception_ResNet_A, self).__init__()
        self.scale = scale
        self.branch_0 = Conv2d(in_channels, 32, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 32, 1, stride=1, padding=0, bias=False),
            Conv2d(32, 32, 3, stride=1, padding=1, bias=False)
        )
        self.branch_2 = nn.Sequential(
            Conv2d(in_channels, 32, 1, stride=1, padding=0, bias=False),
            Conv2d(32, 48, 3, stride=1, padding=1, bias=False),
            Conv2d(48, 64, 3, stride=1, padding=1, bias=False)
        )
        self.conv = nn.Conv2d(128, 320, 1, stride=1, padding=0, bias=True)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        x_res = torch.cat((x0, x1, x2), dim=1)
        x_res = self.conv(x_res)
        return self.relu(x + self.scale * x_res)
```

```
class Inception_ResNet_B(nn.Module):
    def __init__(self, in_channels, scale=1.0):
        super(Inception_ResNet_B, self).__init__()
        self.scale = scale
        self.branch_0 = Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 128, 1, stride=1, padding=0, bias=False),
            Conv2d(128, 160, (1, 7), stride=1, padding=(0, 3), bias=False),
            Conv2d(160, 192, (7, 1), stride=1, padding=(3, 0), bias=False)
        )
        self.conv = nn.Conv2d(384, 1088, 1, stride=1, padding=0, bias=True)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x_res = torch.cat((x0, x1), dim=1)
        x_res = self.conv(x_res)
        return self.relu(x + self.scale * x_res)
```

4 Inception V4, Inception-ResNet (2016)

Inception-ResNet v2 | PyTorch : https://github.com/zhulf0804/Inceptionv4_and_Inception-ResNetv2.PyTorch

```
class Reduciton_B(nn.Module):
    def __init__(self, in_channels):
        super(Reduciton_B, self).__init__()
        self.branch_0 = nn.Sequential(
            Conv2d(in_channels, 256, 1, stride=1, padding=0, bias=False),
            Conv2d(256, 384, 3, stride=2, padding=0, bias=False)
        )
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 256, 1, stride=1, padding=0, bias=False),
            Conv2d(256, 288, 3, stride=2, padding=0, bias=False),
        )
        self.branch_2 = nn.Sequential(
            Conv2d(in_channels, 256, 1, stride=1, padding=0, bias=False),
            Conv2d(256, 288, 3, stride=1, padding=1, bias=False),
            Conv2d(288, 320, 3, stride=2, padding=0, bias=False)
        )
        self.branch_3 = nn.MaxPool2d(3, stride=2, padding=0)

    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x2 = self.branch_2(x)
        x3 = self.branch_3(x)
        return torch.cat((x0, x1, x2, x3), dim=1)
```

```
class Inception_ResNet_C(nn.Module):
    def __init__(self, in_channels, scale=1.0, activation=True):
        super(Inception_ResNet_C, self).__init__()
        self.scale = scale
        self.activation = activation
        self.branch_0 = Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False)
        self.branch_1 = nn.Sequential(
            Conv2d(in_channels, 192, 1, stride=1, padding=0, bias=False),
            Conv2d(192, 224, (1, 3), stride=1, padding=(0, 1), bias=False),
            Conv2d(224, 256, (3, 1), stride=1, padding=(1, 0), bias=False)
        )
        self.conv = nn.Conv2d(448, 2080, 1, stride=1, padding=0, bias=True)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x0 = self.branch_0(x)
        x1 = self.branch_1(x)
        x_res = torch.cat((x0, x1), dim=1)
        x_res = self.conv(x_res)
        if self.activation:
            return self.relu(x + self.scale * x_res)
        return x + self.scale * x_res
```

4 Inception V4, Inception-ResNet (2016)

Inception-ResNet v2 | PyTorch : https://github.com/zhulf0804/Inceptionv4_and_Inception-ResNetv2.PyTorch

```
class Inception_ResNetv2(nn.Module):
    def __init__(self, in_channels=3, classes=1000, k=256, l=256, m=384, n=384):
        super(Inception_ResNetv2, self).__init__()
        blocks = []
        blocks.append(Stem(in_channels))
        for i in range(10):
            blocks.append(Inception_ResNet_A(320, 0.17))
        blocks.append(Reduction_A(320, k, l, m, n))
        for i in range(20):
            blocks.append(Inception_ResNet_B(1088, 0.10))
        blocks.append(Reduciton_B(1088))
        for i in range(9):
            blocks.append(Inception_ResNet_C(2080, 0.20))
        blocks.append(Inception_ResNet_C(2080, activation=False))
        self.features = nn.Sequential(*blocks)
        self.conv = Conv2d(2080, 1536, 1, stride=1, padding=0, bias=False)
        self.global_average_pooling = nn.AdaptiveAvgPool2d((1, 1))
        self.linear = nn.Linear(1536, classes)

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

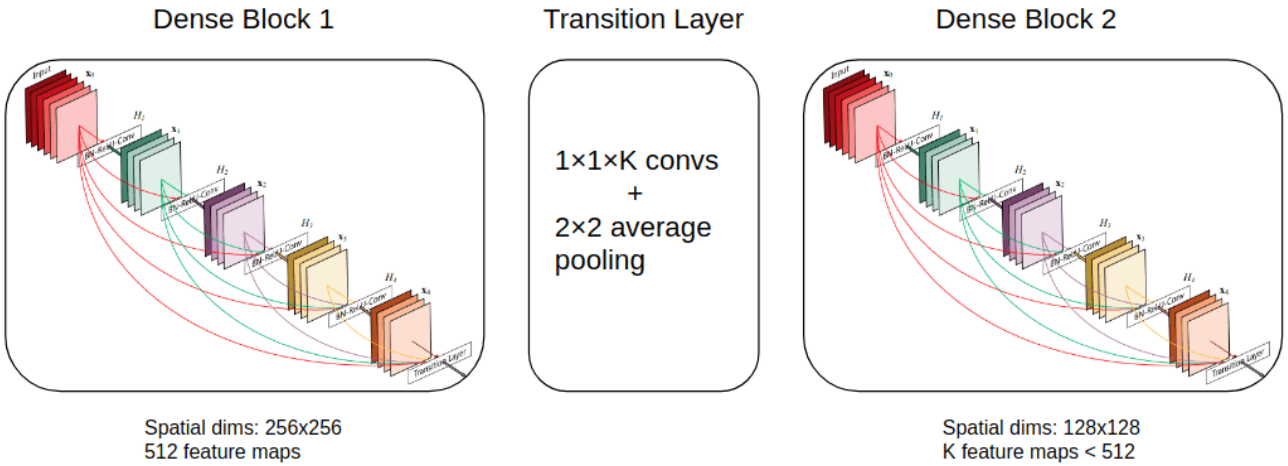
5 DenseNet: Densely Connected Convolutional Networks (2017)

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks.
In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4700-4708).

- Skip connections are a pretty cool idea. Why don't we just skip-connect everything?
- *Densenet* is an example of pushing this idea into the extremity. Of course, the main difference with ResNets is that we will concatenate instead of adding the feature maps.
- Thus, the core idea behind it is *feature reuse*, which leads to very compact models. As a result, it requires fewer parameters than other CNNs, as there are no repeated feature-maps.

- Two concerns here:
  1) The **feature maps** have to be of **the same size.**
  2) The **concatenation with all the previous feature maps** may result in **memory explosion.**

- To address the first issue, we have two solutions:
  1) Use conv layers with appropriate padding that maintain the spatial dims or
  2) Use dense skip connectivity only inside blocks called *Dense Blocks*.



- The *transition layer* can down-sample the image dimensions with average pooling.
- To address the second concern which is memory explosion, the feature maps are reduced (kind of compressed) with 1x1 convs. Notice that I used K in the diagram, but densenet uses $K = featuremaps/2$.
- Furthermore, they add a dropout layer with p=0.2 after each convolutional layer when no data augmentation is used.

5 DenseNet: Densely Connected Convolutional Networks (2017)

[출처1] https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803
[출처2] https://hoya012.github.io/blog/DenseNet-Tutorial-1/

❖ Dense Connectivity

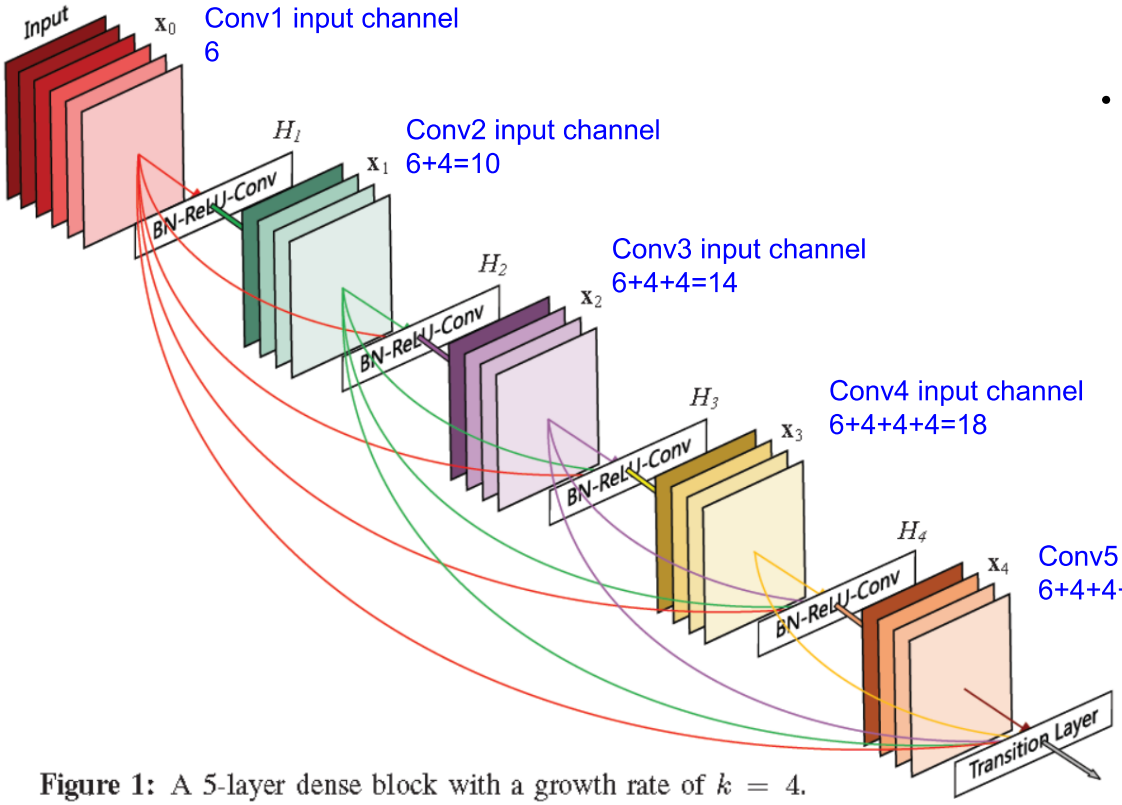


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

- 이전 layer들의 feature map을 계속해서 다음 layer의 입력과 연결 (ResNet)
  - ✓ ResNet은 feature map들을 Add
  - ✓ *DenseNet*은 feature map들을 *Concatenation*
- 이러한 구조의 이점
  - ➢ Vanishing Gradient 개선
  - ➢ Feature Propagation 강화
  - ➢ Feature Reuse
  - ➢ Parameter 수 절약

- l^{th} layer receives the feature maps of all preceding layers, $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}$ as input:

$$\mathbf{x}_l = H_l([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}])$$

- where $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}]$ refers to the concatenation of the feature-maps produced in layers $0, \ldots, l-1$.
- For easy implementation, concatenate the multiple inputs of H_l in the above eq. into a single tensor.

5 DenseNet: Densely Connected Convolutional Networks (2017)

[출처1] https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803
[출처2] https://hoya012.github.io/blog/DenseNet-Tutorial-1/

❖ Composite function

- Define $H_l()$ as a composite function of 3 consecutive operations: batch normalization (BN), ReLU, and 3x3 Conv.

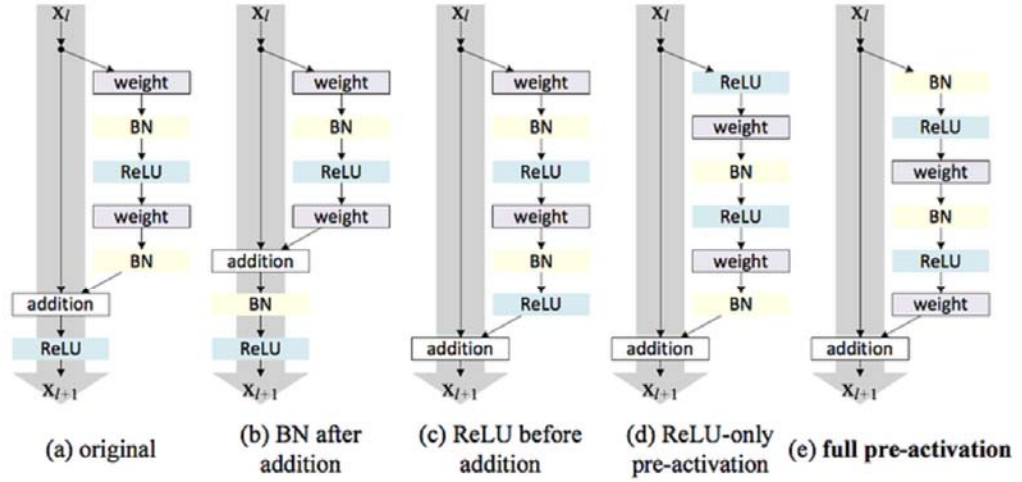
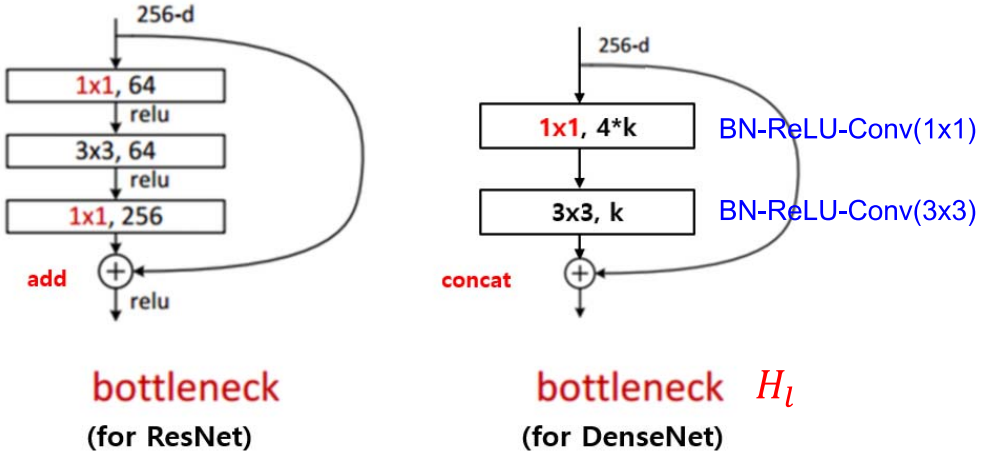


Figure 4. Various usages of activation in Table 2. All these units consist of the same components — only the orders are different.

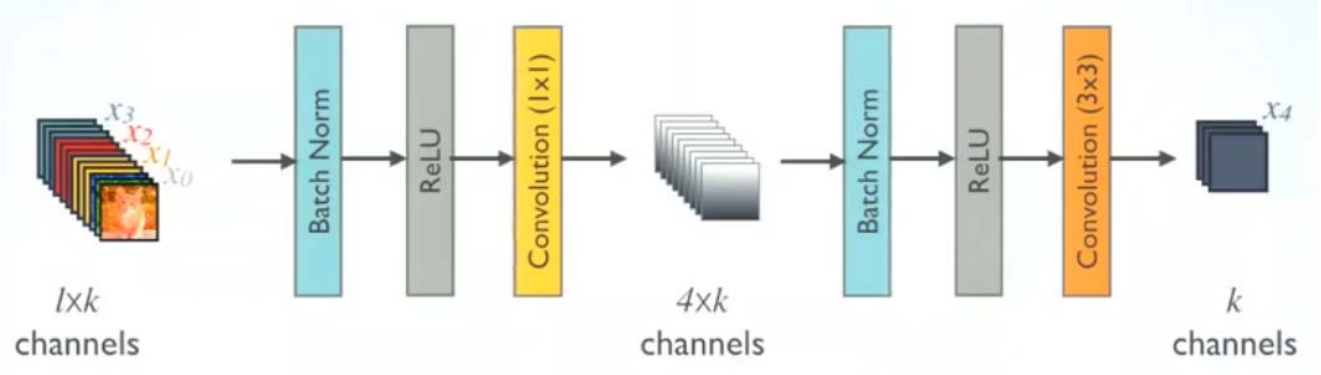
5 DenseNet: Densely Connected Convolutional Networks (2017)

[출처1] https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803
[출처2] https://hoya012.github.io/blog/DenseNet-Tutorial-1/

❖ **Bottleneck Layer (DenseNet-B)**



- ResNet과 Inception 등에서 사용되는 bottleneck layer의 아이디어는 DenseNet에서도 찾아볼 수 있음
- 1x1 convolution can be introduced as bottleneck layer before 3x3 convolution to reduce the number of input feature maps (improve computational efficiency). a bottleneck for DenseNet : BN-ReLU-Conv(1x1)-BN-ReLU-Conv(3x3) version of H_l, as DenseNet-B.
- 3x3 convolution 전에 1x1 convolution을 거쳐서 입력 feature map의 channel 개수를 줄이는 것 까지는 같으나, 그 뒤로 다시 입력 feature map의 channel 개수 만큼을 생성하는 대신 growth rate 만큼의 feature map을 생성하는 것이 차이점이며 이를 통해 computational cost를 줄일 수 있다고 함.
- DenseNet의 Bottleneck Layer는 1x1 convolution 연산을 통해 4* k(growth rate) 개의 feature map을 만들고 그 뒤에 3x3 convolution을 통해 k(growth rate) 개의 feature map으로 줄여줌 (실험적으로 결정)



- To reduce the model complexity and size, BN-ReLU-1×1 Conv is done before BN-ReLU-3×3 Conv.

5 DenseNet: Densely Connected Convolutional Networks (2017)

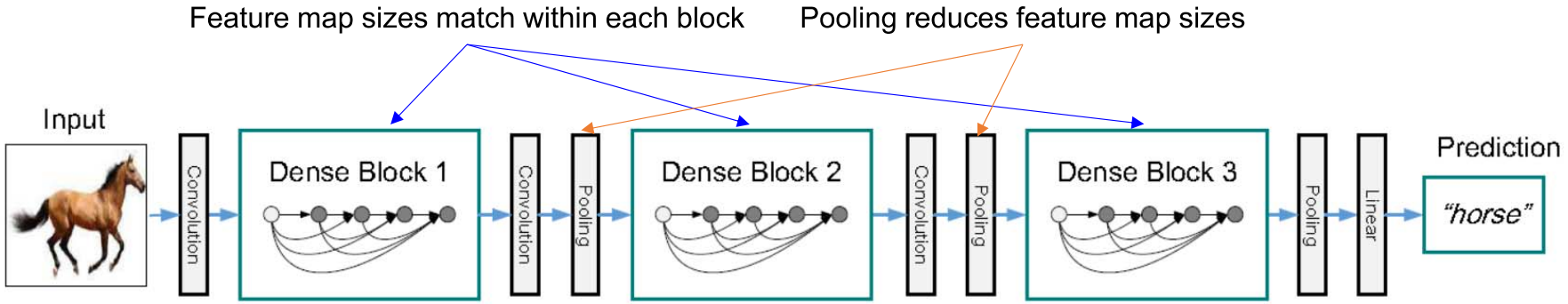


Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

❖ **Pooling Layers (Transition layer)**

- To facilitate *down-sampling* in our architecture, divide the network into multiple densely connected dense blocks; see Figure 2. We refer to layers between blocks as *transition layers*, which do *convolution* and *pooling*.
- The ***transition layers*** used in our experiments consist of a **batch normalization layer** and an **1×1 convolutional layer** followed by a **2×2 average pooling layer**.

❖ **Compression (DenseNet-BC)**

- To further improve model compactness, reduce the number of feature-maps at transition layers.
- If a dense block contains m feature-maps, the following transition layer generate $\lfloor \theta m \rfloor$ output feature maps, as the compression factor θ ($0 < \theta \leq 1$).
- When $\theta = 1$, the number of feature-maps across transition layers remains unchanged.
- Refer the DenseNet with $\theta < 1$ as **DenseNet-C** ($\theta = 0.5$).
- When both the bottleneck and transition layers with $\theta < 1$ are used, refer to our model as **DenseNet-BC**.

5 DenseNet: Densely Connected Convolutional Networks (2017)

[출처1] https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803
[출처2] https://hoya012.github.io/blog/DenseNet-Tutorial-1/

❖ Growth Rate

- Growth rate (Important parameter) : Control the number of feature maps of the total architecture. It specifies the output features of each extra dense conv layer.
- Given k_0 initial feature maps and k growth rate, one can calculate the number of input feature maps of each layer l as $k_0 + k(l-1)$. In frameworks, the number k is in multiples of 4, called *bottleneck size* (bn_size).
- Finally, referencing here DenseNet's most important arguments from torchvision as a sum up:

```
import torchvision
model = torchvision.models.DenseNet(
    growth_rate = 16,                # how many filters to add each layer (`k` in paper)
    block_config = (6, 12, 24, 16),  # how many layers in each pooling block
    num_init_features = 16,          # the number of filters to learn in the first convolution layer (k0)
    bn_size= 4,                      # multiplicative factor for number of bottleneck (1x1 cons) layers
    drop_rate = 0,                   # dropout rate after each dense conv layer
    num_classes = 30                 # number of classification classes
)
print(model) # see snapshot below
```

- 각 feature map끼리 dense 연결 구조이다 보니 자칫 feature map의 channel 개수가 많은 경우 계속해서 channel-wise로 concat이 되면서 channel이 많아 질 수 있음. 그래서 DenseNet에서는 각 layer의 feature map의 channel 개수를 굉장히 작은 값을 사용하며, 이 때 각 layer의 feature map의 channel 개수를 growth rate(k) 이라 부름.
- 위의 그림 1은 k(growth rate) = 4 인 경우를 의미하며 그림 1의 경우로 설명하면 6 channel feature map 입력이 dense block의 4번의 convolution block을 통해 (6 + 4 + 4 + 4 + 4 = 22) 개의 channel을 갖는 feature map output으로 계산이 되는 과정을 보여주고 있음. 위의 그림의 경우를 이해하실 수 있으면 실제 논문에서 구현한 DenseNet의 각 DenseBlock의 각 layer마다 feature map의 channel 개수 또한 간단한 등차수열로 나타낼 수 있습니다.

5 DenseNet: Densely Connected Convolutional Networks (2017)

[출처1] https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803
[출처2] https://hoya012.github.io/blog/DenseNet-Tutorial-1/

❖ Growth Rate

- Inside the “dense” layer (denselayer5 and 6 in the snapshot) there is a bottleneck (1x1) layer that reduces the channels to bn_size * growth_rate = 64 in our case. Otherwise, the number of input channels would explode.
- As demonstrated below each layer adds up 16=growth_rate channels.

```
(denselayer5): _DenseLayer(
  (norm1): BatchNorm2d(120, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): ReLU(inplace=True)
  (conv1): Conv2d(120, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (norm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): ReLU(inplace=True)
  (conv2): Conv2d(64, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
)
(denselayer6): _DenseLayer(
  (norm1): BatchNorm2d(136, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): ReLU(inplace=True)
  (conv1): Conv2d(136, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (norm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): ReLU(inplace=True)
  (conv2): Conv2d(64, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

5 DenseNet: Densely Connected Convolutional Networks (2017)

[출처1] https://towardsdatascience.com/review-densenet-image-classification-b6631a8ef803
[출처2] https://hoya012.github.io/blog/DenseNet-Tutorial-1/

❖ DenseNet architecture for ImageNet

Layers	Output Size	DenseNet-121($k = 32$)	DenseNet-169($k = 32$)	DenseNet-201($k = 32$)	DenseNet-161($k = 48$)
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv; 3 × 3 conv] × 24	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48	[1 × 1 conv; 3 × 3 conv] × 36
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv; 3 × 3 conv] × 16	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 24
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Table 1: DenseNet architectures for ImageNet. The growth rate for the first 3 networks is $k = 32$, and $k = 48$ for DenseNet-161. Note that each "conv" layer shown in the table corresponds the sequence BN-ReLU-Conv.

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ DenseNet Pytorch

[참고1] https://hoya012.github.io/blog/DenseNet-Tutorial-2/
[참고2] https://github.com/pytorch/vision/blob/master/torchvision/models/densenet.py
[참고3] https://pytorch.org/hub/pytorch_vision_densenet/

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

Preparation Step

```
import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.utils import save_image
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import os
import glob
import PIL
from PIL import Image
from torch.utils import data as D
from torch.utils.data.sampler import SubsetRandomSampler
import random
import torchsummary

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

batch_size = 64
validation_ratio = 0.1
random_seed = 10
initial_lr = 0.1
num_epoch = 300
```

Data loader 생성

```
transform_train = transforms.Compose([
        #transforms.Resize(32),
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])

transform_validation = transforms.Compose([
        #transforms.Resize(224),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])

transform_test = transforms.Compose([
        #transforms.Resize(32),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])

trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_train)

validset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_validation)

testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform_test)

num_train = len(trainset)
indices = list(range(num_train))
split = int(np.floor(validation_ratio * num_train))
```

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

Data loader 생성

```
np.random.seed(random_seed)
np.random.shuffle(indices)

train_idx, valid_idx = indices[split:], indices[:split]
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(
    trainset, batch_size=batch_size, sampler=train_sampler, num_workers=0
)

valid_loader = torch.utils.data.DataLoader(
    validset, batch_size=batch_size, sampler=valid_sampler, num_workers=0
)

test_loader = torch.utils.data.DataLoader(
    testset, batch_size=batch_size, shuffle=False, num_workers=0
)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

이 부분은 training, validation, test set으로 split하여 각각의 data loader를 생성해주는 부분을 의미하고, 학습 시에는 random crop, random horizontal flip augmentation을 사용함.

또한 입력 이미지를 CIFAR-10 데이터셋의 평균, 분산으로 normaliz를 해주는 전처리 또한 포함함. [4]번 셀까지 실행을 하면 CIFAR-10 데이터셋을 불러와서 torch data loader class를 생성함

Module Class 생성 : bn-relu-conv 클래스

```
class bn_relu_conv(nn.Module):
    def __init__(self, nin, nout, kernel_size, stride, padding, bias=False):
        super(bn_relu_conv, self).__init__()
        self.batch_norm = nn.BatchNorm2d(nin)
        self.relu = nn.ReLU(True)
        self.conv = nn.Conv2d(nin, nout, kernel_size=kernel_size, stride=stride,
padding=padding, bias=bias)

    def forward(self, x):
        out = self.batch_norm(x)
        out = self.relu(out)
        out = self.conv(out)

        return out
```

Composite function : Batch Normalization – ReLU – Convolution 연산을 차례대로 수행하는 역할을 하며, DenseNet에 이러한 composite function이 자주 사용되어서 편의상 별도의 class로 생성함.

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

Module Class 생성 : bottleneck layer 클래스

```
class bottleneck_layer(nn.Sequential):
  def __init__(self, nin, growth_rate, drop_rate=0.2):
      super(bottleneck_layer, self).__init__()

      self.add_module('conv_1x1', bn_relu_conv(nin=nin, nout=growth_rate*4, kernel_size=1,
stride=1, padding=0, bias=False))
      self.add_module('conv_3x3', bn_relu_conv(nin=growth_rate*4, nout=growth_rate,
kernel_size=3, stride=1, padding=1, bias=False))

      self.drop_rate = drop_rate

  def forward(self, x):
      bottleneck_output = super(bottleneck_layer, self).forward(x)
      if self.drop_rate > 0:
          bottleneck_output = F.dropout(bottleneck_output, p=self.drop_rate,
training=self.training)

      bottleneck_output = torch.cat((x, bottleneck_output), 1)

      return bottleneck_output
```

bottleneck layer는 DenseNet-BC에서 사용되며, 1x1 convolution 과 3x3 convolution 연산이 차례대로 수행됨. 또한 각 연산의 output feature map 크기가 변하고 drop out도 구현됨.

torch.cat 함수 : DenseNet의 핵심인 부분이며 매 bottleneck layer를 거칠 때 마다 feature map이 channel-wise로 누적되는 것을 한 줄로 구현할 수 있음. Cat 함수의 2번째 parameter의 argument로 넣어준 1 값은 concatenation을 해주는 dimension을 의미하며 1번째 차원은 channel을 의미합니다

Module Class 생성 : Transition Layer 클래스

```
class Transition_layer(nn.Sequential):
  def __init__(self, nin, theta=0.5):
      super(Transition_layer, self).__init__()

      self.add_module('conv_1x1', bn_relu_conv(nin=nin, nout=int(nin*theta), kernel_size=1,
stride=1, padding=0, bias=False))
      self.add_module('avg_pool_2x2', nn.AvgPool2d(kernel_size=2, stride=2, padding=0))
```

Transition Layer는 간단하게 구현이 가능합니다. 1x1 convolution 이후 2x2 average pooling을 사용하며, Compression hyper parameter인 theta에 따라 1x1 convolution의 output feature map의 개수를 조절할 수 있습니다.

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

Module Class 생성 : DenseBlock 클래스

```
class DenseBlock(nn.Sequential):
  def __init__(self, nin, num_bottleneck_layers, growth_rate, drop_rate=0.2):
      super(DenseBlock, self).__init__()

      for i in range(num_bottleneck_layers):
          nin_bottleneck_layer = nin + growth_rate * i
          self.add_module('bottleneck_layer_%d' % i,
bottleneck_layer(nin=nin_bottleneck_layer, growth_rate=growth_rate, drop_rate=drop_rate))
```

DenseBlock은 위에서 생성한 bottleneck layer를 각 DenseBlock의 layer 길이에 맞게 차례대로 이어주는 방식으로 구현이 가능하며 for 문을 통해 간단히 구현할 수 있음.

각 bottleneck layer의 input feature map(nin)은 첫 feature map의 개수에 growth rate을 더해주는 방식으로 계산하며, 이는 nin_bottleneck_layer 라는 variable로 정의함.

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

DenseNet-BC 구성

```
class DenseNet(nn.Module):
    def __init__(self, growth_rate=12, num_layers=100, theta=0.5, drop_rate=0.2, num_classes=10):
        super(DenseNet, self).__init__()

        assert (num_layers - 4) % 6 == 0

        num_bottleneck_layers = (num_layers - 4) // 6            # (num_layers-4)//6

        # 32 x 32 x 3 --> 32 x 32 x (growth_rate*2)
        self.dense_init = nn.Conv2d(3, growth_rate*2, kernel_size=3, stride=1, padding=1, bias=True)

        # 32 x 32 x (growth_rate*2) --> 32 x 32 x [(growth_rate*2) + (growth_rate * num_bottleneck_layers)]
        self.dense_block_1 = DenseBlock(nin=growth_rate*2, num_bottleneck_layers=num_bottleneck_layers, growth_rate=growth_rate, drop_rate=drop_rate)

        # 32 x 32 x [(growth_rate*2) + (growth_rate * num_bottleneck_layers)] --> 16 x 16 x [(growth_rate*2) + (growth_rate * num_bottleneck_layers)]*theta
        nin_transition_layer_1 = (growth_rate*2) + (growth_rate * num_bottleneck_layers)
        self.transition_layer_1 = Transition_layer(nin=nin_transition_layer_1, theta=theta)

        # 16 x 16 x nin_transition_layer_1*theta --> 16 x 16 x [nin_transition_layer_1*theta + (growth_rate * num_bottleneck_layers)]
        self.dense_block_2 = DenseBlock(nin=int(nin_transition_layer_1*theta), num_bottleneck_layers=num_bottleneck_layers, growth_rate=growth_rate, drop_rate=drop_rate)

        # 16 x 16 x [nin_transition_layer_1*theta + (growth_rate * num_bottleneck_layers)] --> 8 x 8 x [nin_transition_layer_1*theta + (growth_rate * num_bottleneck_layers)]*theta
        nin_transition_layer_2 = int(nin_transition_layer_1*theta) + (growth_rate * num_bottleneck_layers)
        self.transition_layer_2 = Transition_layer(nin=nin_transition_layer_2, theta=theta)

        # 8 x 8 x nin_transition_layer_2*theta --> 8 x 8 x [nin_transition_layer_2*theta + (growth_rate * num_bottleneck_layers)]
        self.dense_block_3 = DenseBlock(nin=int(nin_transition_layer_2*theta), num_bottleneck_layers=num_bottleneck_layers, growth_rate=growth_rate, drop_rate=drop_rate)

        nin_fc_layer = int(nin_transition_layer_2*theta) + (growth_rate * num_bottleneck_layers)

        # [nin_transition_layer_2*theta + (growth_rate * num_bottleneck_layers)] --> num_classes
        self.fc_layer = nn.Linear(nin_fc_layer, num_classes)
```

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

DenseNet-BC 구성

```
def forward(self, x):
    dense_init_output = self.dense_init(x)

    dense_block_1_output = self.dense_block_1(dense_init_output)
    transition_layer_1_output = self.transition_layer_1(dense_block_1_output)

    dense_block_2_output = self.dense_block_2(transition_layer_1_output)
    transition_layer_2_output = self.transition_layer_2(dense_block_2_output)

    dense_block_3_output = self.dense_block_3(transition_layer_2_output)

    global_avg_pool_output = F.adaptive_avg_pool2d(dense_block_3_output, (1, 1))
    global_avg_pool_output_flat = global_avg_pool_output.view(global_avg_pool_output.size(0), -1)

    output = self.fc_layer(global_avg_pool_output_flat)

    return output
```

첫 번째 Convolution 이후 Dense Block과 Transition Layer들을 차례로 통과시키고 마지막에 global average pooling을 거친 후 fully-connected layer로 연결해주는 부분이 위의 코드에 나와있습니다.

각 module 마다 feature map의 shape이 어떻게 변하는지를 각 모듈의 선언 부분 위의 주석을 통해 기재하였으며 이렇게 각 layer 마다 shape이 변하는 것을 계산하는 과정을 통해 architecture에 대한 이해도가 높아질 수 있습니다..

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

DenseNet-BC 모델

```
def DenseNetBC_100_12():
    return DenseNet(growth_rate=12, num_layers=100, theta=0.5, drop_rate=0.2,
num_classes=10)

def DenseNetBC_250_24():
    return DenseNet(growth_rate=24, num_layers=250, theta=0.5, drop_rate=0.2,
num_classes=10)

def DenseNetBC_190_40():
    return DenseNet(growth_rate=40, num_layers=190, theta=0.5, drop_rate=0.2,
num_classes=10)

net = DenseNetBC_100_12()
net.to(device)
```

논문에서 제시하고 있는 3가지 DenseNet-BC 모델들을 위에서 생성한 DenseNet class를 통해 만들어주는 함수임. Growth rate, num_layer를 통해 조절이 가능하며 또한 transition layer의 compression 정도, drop out rate 등도 조절할 수 있음.

생성한 architecture를 맨 처음 생성한 torch.device에 넣어주면 GPU에서 학습을 할 수 있음

toch summary를 통해 DenseNet의 Model Summary를 수행하고, 그 중 첫번째 DenseBlock의 feature map shape을 요약하고 있음. Input feature map이 bn_relu_conv block을 거쳐서 channel이 어떻게 변하는지를 한눈에 볼 수 있으며, bottleneck layer에서 48 채널로 변환이 되었다가 이전 feature map과 concat이 되어서 feature map 개수가 증가하는 과정을 확인하실 수 있음.

Architecture Summary

```
torchsummary.summary(net, (3, 32, 32))
```

```
torchsummary.summary(net, (3, 32, 32))
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 24, 32, 32]             672
       BatchNorm2d-2           [-1, 24, 32, 32]              48
              ReLU-3           [-1, 24, 32, 32]               0
            Conv2d-4           [-1, 48, 32, 32]           1,152
      bn_relu_conv-5           [-1, 48, 32, 32]               0
       BatchNorm2d-6           [-1, 48, 32, 32]              96
              ReLU-7           [-1, 48, 32, 32]               0
            Conv2d-8           [-1, 12, 32, 32]           5,184
      bn_relu_conv-9           [-1, 12, 32, 32]               0
       BatchNorm2d-10          [-1, 36, 32, 32]              72
             ReLU-11           [-1, 36, 32, 32]               0
           Conv2d-12           [-1, 48, 32, 32]           1,728
                                      ⋮
      bn_relu_conv-391         [-1, 48, 8, 8]                 0
       BatchNorm2d-392         [-1, 48, 8, 8]                96
             ReLU-393          [-1, 48, 8, 8]                 0
           Conv2d-394          [-1, 12, 8, 8]             5,184
      bn_relu_conv-395         [-1, 12, 8, 8]                 0
          Linear-396                 [-1, 10]             3,430
================================================================
Total params: 768,502
Trainable params: 768,502
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 87.35
Params size (MB): 2.93
Estimated Total Size (MB): 90.30
----------------------------------------------------------------
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
       BatchNorm2d-2           [-1, 24, 32, 32]              48
       BatchNorm2d-6           [-1, 48, 32, 32]              96
      BatchNorm2d-10           [-1, 36, 32, 32]              72
      BatchNorm2d-14           [-1, 48, 32, 32]              96
      BatchNorm2d-18           [-1, 48, 32, 32]              96
      BatchNorm2d-22           [-1, 48, 32, 32]              96
      BatchNorm2d-26           [-1, 60, 32, 32]             120
      BatchNorm2d-30           [-1, 48, 32, 32]              96
      BatchNorm2d-34           [-1, 72, 32, 32]             144
      BatchNorm2d-38           [-1, 48, 32, 32]              96
      BatchNorm2d-42           [-1, 84, 32, 32]             168
      BatchNorm2d-46           [-1, 48, 32, 32]              96
      BatchNorm2d-50           [-1, 96, 32, 32]             192
      BatchNorm2d-54           [-1, 48, 32, 32]              96
      BatchNorm2d-58          [-1, 108, 32, 32]             216
      BatchNorm2d-62           [-1, 48, 32, 32]              96
      BatchNorm2d-66          [-1, 120, 32, 32]             240
      BatchNorm2d-70           [-1, 48, 32, 32]              96
      BatchNorm2d-74          [-1, 132, 32, 32]             264
      BatchNorm2d-78           [-1, 48, 32, 32]              96
      BatchNorm2d-82          [-1, 144, 32, 32]             288
      BatchNorm2d-86           [-1, 48, 32, 32]              96
      BatchNorm2d-90          [-1, 156, 32, 32]             312
      BatchNorm2d-94           [-1, 48, 32, 32]              96
      BatchNorm2d-98          [-1, 168, 32, 32]             336
     BatchNorm2d-102           [-1, 48, 32, 32]              96
     BatchNorm2d-106          [-1, 180, 32, 32]             360
     BatchNorm2d-110           [-1, 48, 32, 32]              96
     BatchNorm2d-114          [-1, 192, 32, 32]             384
     BatchNorm2d-118           [-1, 48, 32, 32]              96
     BatchNorm2d-122          [-1, 204, 32, 32]             408
     BatchNorm2d-126           [-1, 48, 32, 32]              96
     BatchNorm2d-130          [-1, 216, 32, 32]             432
```

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

DenseNet-BC 모델

```
def DenseNetBC_100_12():
    return DenseNet(growth_rate=12, num_layers=100, theta=0.5, drop_rate=0.2,
num_classes=10)

def DenseNetBC_250_24():
    return DenseNet(growth_rate=24, num_layers=250, theta=0.5, drop_rate=0.2,
num_classes=10)

def DenseNetBC_190_40():
    return DenseNet(growth_rate=40, num_layers=190, theta=0.5, drop_rate=0.2,
num_classes=10)

net = DenseNetBC_100_12()
net.to(device)
```

논문에서 제시하고 있는 3가지 DenseNet-BC 모델들을 위에서 생성한 DenseNet class 를 통해 만들어주는 함수임. Growth rate, num_layer를 통해 조절이 가능하며 또한 transition layer의 compression 정도, drop out rate 등도 조절할 수 있음.

생성한 architecture를 맨 처음 생성한 torch.device에 넣어주면 GPU에서 학습을 할 수 있음

Training

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=initial_lr, momentum=0.9)
lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer=optimizer,
milestones=[int(num_epoch * 0.5), int(num_epoch * 0.75)], gamma=0.1, last_epoch=-1)

for epoch in range(num_epoch):
    lr_scheduler.step()

    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        show_period = 100
        if i % show_period == show_period-1:    # print every "show_period" mini-batches
            print('[%d, %5d/50000] loss: %.7f' %
                  (epoch + 1, (i + 1)*batch_size, running_loss / show_period))
            running_loss = 0.0
```

5 DenseNet: Densely Connected Convolutional Networks (2017)

❖ **DenseNet Pytorch : https://hoya012.github.io/blog/DenseNet-Tutorial-2/**

Training

```
    # validation part
    correct = 0
    total = 0
    for i, data in enumerate(valid_loader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs)

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

    print('[%d epoch] Accuracy of the network on the validation images: %d %%' %
          (epoch + 1, 100 * correct / total)
         )

print('Finished Training')
```

SGD optimizer에 momentum은 0.9 사용하였고 learning rate는 pytorch의 learning rate scheduling을 이용하여 구현함. 전체 epoch의 50%인 지점과 75%인 지점에서 각각 0.1배씩 곱해주는 방식이므로 lr_scheduler.MultiStepLR 을 이용하여 구현함.

그리고 1 epoch이 끝날 때 마다 validation set으로 accuracy를 측정하고 있으며 이 예제에서는 model selection, learning curve plotting은 구현하지 않았음. 필요할 경우 추가하여 사용가능.

Test set에 대해 test를 한 뒤 10가지 클래스마다 정확도를 각각 구하고, 또한 전체 정확도를 구하는 과정이 코드로 구현됨

Test

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

correct = 0
total = 0

with torch.no_grad():
    for data in test_loader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()

        for i in range(labels.shape[0]):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

            total += labels.size(0)
            correct += (predicted == labels).sum().item()

print('Accuracy of the network on the test images: %d %%' % (100 * correct / total))

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

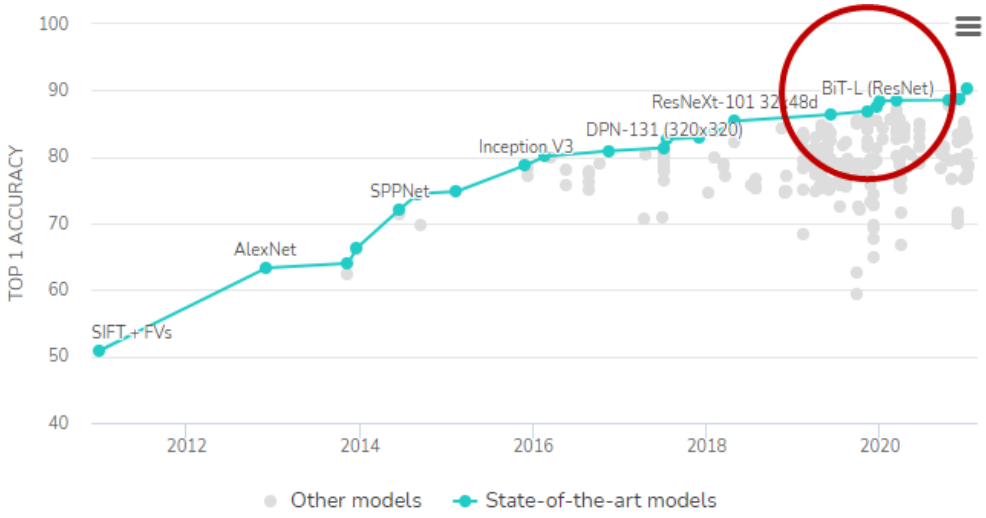
6 Big Transfer (BiT): General Visual Representation Learning (2020)

Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, Neil Houlsby, "Big Transfer (BiT): General Visual Representation Learning," arXiv:1912.11370, 2020.

[출처] https://theaisummer.com/cnn-architectures/
[출처] https://ai.googleblog.com/2020/05/open-sourcing-bit-exploring-large-scale.html
[출처2] https://kmhana.tistory.com/7

Pytorch : https://github.com/google-research/big_transfer

- Transfer Learning에 대한 Insight : 대용량 데이터를 사전학습 할 때와 그 모델을 Fine-tuning할 때에 대한 고찰, But, Google 전용 데이터를 사용했으며, 엄청난 자원이 필요함.



Best performance on ImageNet

- Even though many variants of ResNet have been proposed, the most recent and famous one is BiT.
- **Big Transfer (BiT)** is **a scalable ResNet-based model for effective image pre-training** [5].
- They developed 3 BiT models (small, medium and large) based on ResNet152. For the large variation of BiT they used **ResNet152x4**, which means that each layer has 4 times more channels. They pretrained that model once in far more bigger datasets than ImageNet. The largest model was trained on the insanely large JFT dataset, which consists of 300M labeled images.
- The major contribution in the architecture is the choice of normalization layers. To this end, the authors replaced batch normalization (BN) with group normalization (GN) and weight standardization (WS).

6 Big Transfer (BiT): General Visual Representation Learning (2020)

Big Transfer

- 효과적 전이학습 설계 위한 주요 요소 분석
- Pre-training 동안 사용되는 *Upstream 요소* (Scale, GN&WS)
- New task에 대한 fine-tuning 동안 사용되는 *Downstream 요소*

➢ **Upstream Pre-training**

① Transfer learning 의 Scale (architecture size, dataset size, training time etc) 분석

✓ 3 BiT 모델 학습 : BiT-S (ILSVRC-2012, 1.3M images), BiT-M (ImageNet-21k, 14M images), BiT-L (JFT, 300M images)

② Group normalization (GN) & Weight Standardization (WS)

Summary of our pre-training strategy: take a standard ResNet, increase depth and width, replace BatchNorm (BN) with GroupNorm and Weight Standardization (GNWS), and train on a very large and generic dataset for many more iterations.

- Replaced batch normalization (BN) with **group normalization (GN)** and **weight standardization (WS)**.
- Because first BN's parameters (means and variances) need adjustment between pre-training and transfer. On the other hand, GN doesn't depend on any parameter states.
- Another reason is that BN uses batch-level statistics, which become unreliable for distributed training in small devices like TPU's. Since GN does not compute batch-level statistics, it also side-steps this issue. A 4K batch distributed across 500 TPU's means 8 batches per worker, which does not give a good estimation of the statistics. By changing the normalization technique to GN+WS they avoid synchronization across workers.

6 Big Transfer (BiT): General Visual Representation Learning (2020)

➢ **Transfer to Downstream Tasks**

- Following the methods established in the language domain by BERT, we fine-tune the pre-trained BiT model on data from a variety of "*downstream*" tasks of interest, which may come with very little labeled data.
- **Fine-tuning** comes with a lot of hyper-parameters to be chosen, such as l*earning-rate*, *weight-decay*, etc. We propose a heuristic for selecting these hyper-parameters that we call "**BiT-HyperRule**", which is based only on high-level dataset characteristics, such as *image resolution* and *the number of labeled examples*. We successfully apply the BiT-HyperRule on more than 20 diverse tasks, ranging from natural to medical images.

Once the BiT model is pre-trained, it can be fine-tuned on any task, even if only few labeled examples are available.

- Try one hyper-parameter per task to avoid expensive hyper-parameter search
- heuristic rule(call *BiT-HyperRule*) 사용 : Task의 이미지 해상도와 데이터 수의 간단한 함수로 Tuning 위한 가장 중요한 hyper-parameter 선택; 학습 스케줄 길이, 해상도, MixUp regularization 사용 여부
- Task 마다의 Data Augmentation이 다름 : 예) object orientation Task 같은 경우 Flip이나 Crop이 악영향을 줌
- Training과 Test에서 발생되는 해상도 차이 해결 : Fine-Tuning 단계에서 해결
- Transfer 진행 시, weight decay나 Dropout 같은 Regularization 사용안 함; 매우 큰 모델 + 매우 작은 데이터의 Downstream Task에서도, 이와 같은 Regularization을 사용하지 않아도 좋은 성능

6 Big Transfer (BiT): General Visual Representation Learning (2020)

HyperRule 및 모델구성 설명

1. Upstream Pre-training

- 모델 Architecture : ResNet-v2, 모델 크기(디폴트) : ResNet 152x4d
- Optimizer : SGD with momentum (Lr = 0.03, m = 0.9)
  - ❖ Adaptive 계열을 써도 더 좋은 성능 개선은 없음
- 학습 데이터 : BiT-S - 130만 장(ILSVRC-2012) / BiT-M - 1400만 장 (ImageNet-21k) / BiT-L - 3억 장(JFT)
  - ❖ *JFT dataset* : 구글이 만든 dataset이고 오픈되지 않음. 개인적인 견해로, 형평성 관점에서 SOTA에서 JFT dataset 사용 여부를 분리하는게 좋을것 같음. Google 팀에서, JFT dataset을 활용하여, 엄청난 성과를 보이고 있다.
- 이미지 크기 : 224x224
- Train Schedule : BiT-S와 BiT-M - 90 epochs(30,60, 80에서 Lr decay 10), BiT-L - 40 epochs(10, 23, 30, 37에서 Lr decay 10)
- Warmup : 5000 steps (매 step마다 batch size / 256 씩 곱함)
  - ❖ Warmup은 요즘 CNN에서 필수 덕목처럼 보임.
- Batch Size : 4096 - 엄청난 숫자이며.. 사용한 TPU는 512 장 (chip 당 8 이미지 학습)
- Weight decay : 0.0001 (Transfer learning에서는 사용하지 않는다)

2. Downstream Fine-tuning

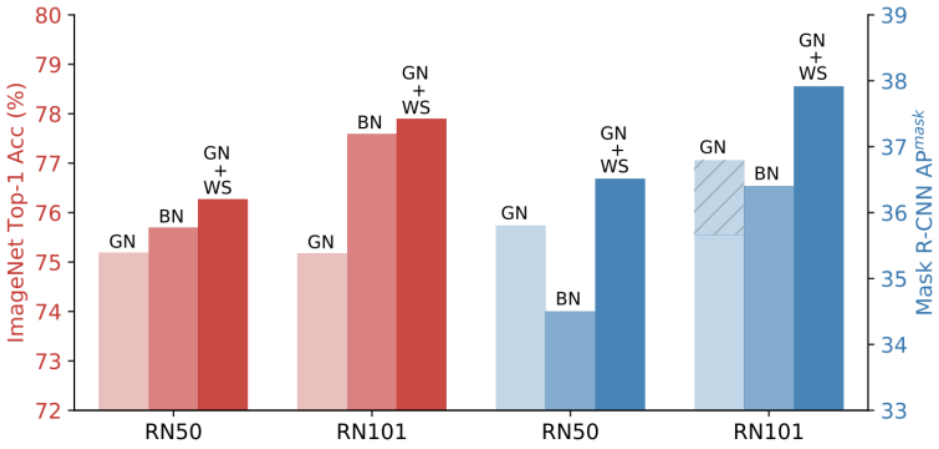
- 기준 : Small task (2만장), Medium task (50만 장 미만), Large task (50만 장 이상)라고 정의
- Optimizer : SGD with momentum (Lr = 0.003, m = 0.9) → 사전 학습보다 0.1 낮음
- Batch Size : 512
- 이미지 크기 : 96x96↓ ▶ 160x160변환 ▶ 128 crop
  448x448↑ ▶ 384 crop
- Fine-Tuning Schedule : 공통 - 30, 60, 90%에서 Lr decay (10 factor)
  S task - 500 steps (최소 약 12.8 epoch)
  M task - 1만 steps (최소 약 10.2 epoch)
  L task - 2만 steps (최대 20.5 epoch)
- Mixup 유무 : M과 L task에서만 Mixup 사용(alpha = 0.1)

참고 : Bag of Tricks for Image Classification with Convolutional Neural Networks
https://norman3.github.io/papers/docs/bag_of_tricks_for_image_classification.html

6 Big Transfer (BiT): General Visual Representation Learning (2020)

Weight Standardization (WN) [출처] https://blog.lunit.io/2019/05/28/weight-standardization/

- Normalization은 머신러닝에서 데이터의 불필요한 정보를 제거하고 학습을 용이하게 하기 위해 매우 중요한 요소임
- Batch Normalization (BN)은 모델 학습을 안정시키고 가속화함으로써 성능을 크게 향상시켰고, ResNet을 비롯한 대부분의 SOTA 모델에서 필수적인 요소로 사용됨. 그러나 BN은 minibatch 단위로 normalization을 수행하므로, 모델 성능이 large batch size에 의존하게 되는 단점이 있음.
- 이러한 문제를 해결하기 위해 Group Normalization (GN) 등 다양한 기법이 제안되었지만, 일반적인 large-batch training 상황에서는 BN의 성능에 미치지 못하여 활용도가 현저히 떨어짐.
- Weight Standardization (WS) 은 GN과 같이 minibatch dependency를 완전히 제거하면서 large-batch 상황에서의 BN보다도 좋은 성능을 달성함.



- Batch/Layer/Instance/Group Normalization 과 같은 기존의 기법들은 주로 feature activation을 대상으로 normalization을 수행하는 반면, WS는 weight (convolution filter)을 대상으로 normalization을 수행

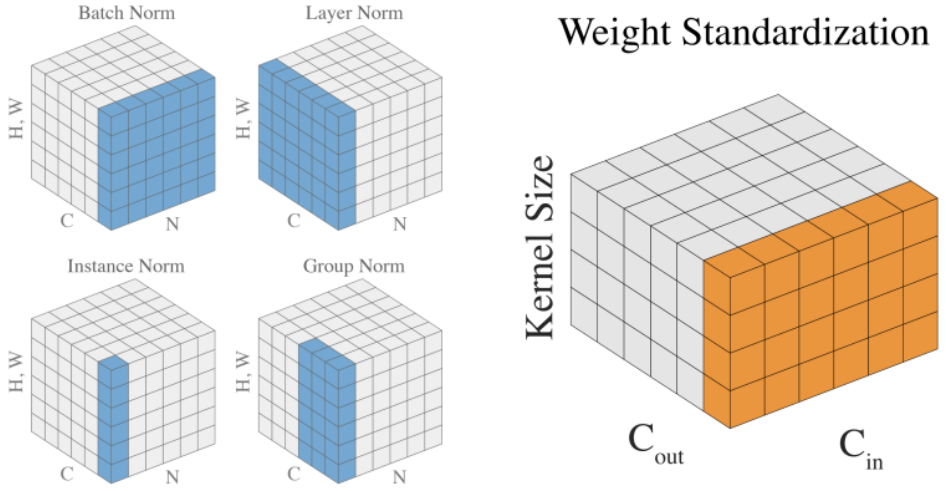


그림 3. Comparing normalization methods on activations and Weight Standardization.

$$\hat{\boldsymbol{W}} = \left[\hat{\boldsymbol{W}}_{i,j} \mid \hat{\boldsymbol{W}}_{i,j} = \frac{\boldsymbol{W}_{i,j} - \mu_{\boldsymbol{W}_{i,\cdot}}}{\sigma_{\boldsymbol{W}_{i,\cdot}+\epsilon}}\right] \qquad \mu_{\boldsymbol{W}_{i,\cdot}} = \frac{1}{I}\sum_{j=1}^{I}\boldsymbol{W}_{i,j}, \quad \sigma_{\boldsymbol{W}_{i,\cdot}} = \sqrt{\frac{1}{I}\sum_{i=1}^{I}(\boldsymbol{W}_{i,j} - \mu_{\boldsymbol{W}_{i,\cdot}})^2}$$

$$\boldsymbol{y} = \hat{\boldsymbol{W}} * \boldsymbol{x}$$

6 Big Transfer (BiT): General Visual Representation Learning (2020)

❖ **BiT Pytorch : https://github.com/google-research/big_transfer, big_transfer_pytorch.ipynb**

Colab : big_transfer_pytorch.ipynb

- This colab demonstrates how to:
  1) Load BiT models in PyTorch
  2) Make predictions using BiT pre-trained on ImageNet
  3) Fine-tune BiT on 5-shot CIFAR10 and get amazing results!
- It is good to get an understanding or quickly try things. However, to run longer training runs, we recommend using the commandline scripts at http://github.com/google-research/big_transfer

```
from functools import partial
from collections import OrderedDict

%config InlineBackend.figure_format = 'retina'

import numpy as np

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import torchvision as tv

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

Reading weight data from the Cloud bucket

```
import requests
import io

def get_weights(bit_variant):
  response = requests.get(f'https://storage.googleapis.com/bit_models/{bit_variant}.npz')
  response.raise_for_status()
  return np.load(io.BytesIO(response.content))

weights = get_weights('BiT-M-R50x1')  # You could use other variants, such as R101x3
or R152x4 here, but it is not advisable in a colab.
```

6 Big Transfer (BiT): General Visual Representation Learning (2020)

❖ **BiT Pytorch : https://github.com/google-research/big_transfer, big_transfer_pytorch.ipynb**

Defining the architecture and loading weights

```
class StdConv2d(nn.Conv2d):

  def forward(self, x):
    w = self.weight
    v, m = torch.var_mean(w, dim=[1, 2, 3], keepdim=True, unbiased=False)
    w = (w - m) / torch.sqrt(v + 1e-10)
    return F.conv2d(x, w, self.bias, self.stride, self.padding, self.dilation, self.groups)


def conv3x3(cin, cout, stride=1, groups=1, bias=False):
  return StdConv2d(cin, cout, kernel_size=3, stride=stride,
                   padding=1, bias=bias, groups=groups)

def conv1x1(cin, cout, stride=1, bias=False):
  return StdConv2d(cin, cout, kernel_size=1, stride=stride,
                   padding=0, bias=bias)

def tf2th(conv_weights):
  """Possibly convert HWIO to OIHW."""
  if conv_weights.ndim == 4:
    conv_weights = conv_weights.transpose([3, 2, 0, 1])
  return torch.from_numpy(conv_weights)
```

```
class PreActBottleneck(nn.Module):
  """
  Follows the implementation of "Identity Mappings in Deep Residual Networks" here:
  https://github.com/KaimingHe/resnet-1k-layers/blob/master/resnet-pre-act.lua
  Except it puts the stride on 3x3 conv when available.
  """
  def __init__(self, cin, cout=None, cmid=None, stride=1):
    super().__init__()
    cout = cout or cin
    cmid = cmid or cout//4

    self.gn1 = nn.GroupNorm(32, cin)
    self.conv1 = conv1x1(cin, cmid)
    self.gn2 = nn.GroupNorm(32, cmid)
    self.conv2 = conv3x3(cmid, cmid, stride)  # Original ResNetv2 has it on conv1!!
    self.gn3 = nn.GroupNorm(32, cmid)
    self.conv3 = conv1x1(cmid, cout)
    self.relu = nn.ReLU(inplace=True)

    if (stride != 1 or cin != cout):
      # Projection also with pre-activation according to paper.
      self.downsample = conv1x1(cin, cout, stride)
```

6 Big Transfer (BiT): General Visual Representation Learning (2020)

❖ **BiT Pytorch : https://github.com/google-research/big_transfer, big_transfer_pytorch.ipynb**

```
  def forward(self, x):
    # Conv'ed branch
    out = self.relu(self.gn1(x))

    # Residual branch
    residual = x
    if hasattr(self, 'downsample'):
      residual = self.downsample(out)

    # The first block has already applied pre-act before splitting, see Appendix.
    out = self.conv1(out)
    out = self.conv2(self.relu(self.gn2(out)))
    out = self.conv3(self.relu(self.gn3(out)))

    return out + residual

  def load_from(self, weights, prefix=''):
    with torch.no_grad():
      self.conv1.weight.copy_(tf2th(weights[prefix + 'a/standardized_conv2d/kernel']))
      self.conv2.weight.copy_(tf2th(weights[prefix + 'b/standardized_conv2d/kernel']))
      self.conv3.weight.copy_(tf2th(weights[prefix + 'c/standardized_conv2d/kernel']))
      self.gn1.weight.copy_(tf2th(weights[prefix + 'a/group_norm/gamma']))
      self.gn2.weight.copy_(tf2th(weights[prefix + 'b/group_norm/gamma']))
      self.gn3.weight.copy_(tf2th(weights[prefix + 'c/group_norm/gamma']))
      self.gn1.bias.copy_(tf2th(weights[prefix + 'a/group_norm/beta']))
      self.gn2.bias.copy_(tf2th(weights[prefix + 'b/group_norm/beta']))
      self.gn3.bias.copy_(tf2th(weights[prefix + 'c/group_norm/beta']))
      if hasattr(self, 'downsample'):
        self.downsample.weight.copy_(tf2th(weights[prefix +
'a/proj/standardized_conv2d/kernel']))
    return self
```

```
class ResNetV2(nn.Module):
  """Implementation of Pre-activation (v2) ResNet mode."""

  def __init__(self, block_units, width_factor, head_size=21843, zero_head=False):
    super().__init__()
    wf = width_factor  # shortcut 'cause we'll use it a lot.

    # The following will be unreadable if we split lines.
    # pylint: disable=line-too-long
    self.root = nn.Sequential(OrderedDict([
        ('conv', StdConv2d(3, 64*wf, kernel_size=7, stride=2, padding=3, bias=False)),
        ('pad', nn.ConstantPad2d(1, 0)),
        ('pool', nn.MaxPool2d(kernel_size=3, stride=2, padding=0)),
        # The following is subtly not the same!
        # ('pool', nn.MaxPool2d(kernel_size=3, stride=2, padding=1)),
    ]))
```

6 Big Transfer (BiT): General Visual Representation Learning (2020)

❖ **BiT Pytorch : https://github.com/google-research/big_transfer, big_transfer_pytorch.ipynb**

```
self.body = nn.Sequential(OrderedDict([
    ('block1', nn.Sequential(OrderedDict(
        [('unit01', PreActBottleneck(cin=64*wf, cout=256*wf, cmid=64*wf))] +
        [(f'unit{i:02d}', PreActBottleneck(cin=256*wf, cout=256*wf, cmid=64*wf)) for i in range(2, block_units[0] + 1)],
    ))),
    ('block2', nn.Sequential(OrderedDict(
        [('unit01', PreActBottleneck(cin=256*wf, cout=512*wf, cmid=128*wf, stride=2))] +
        [(f'unit{i:02d}', PreActBottleneck(cin=512*wf, cout=512*wf, cmid=128*wf)) for i in range(2, block_units[1] + 1)],
    ))),
    ('block3', nn.Sequential(OrderedDict(
        [('unit01', PreActBottleneck(cin=512*wf, cout=1024*wf, cmid=256*wf, stride=2))] +
        [(f'unit{i:02d}', PreActBottleneck(cin=1024*wf, cout=1024*wf, cmid=256*wf)) for i in range(2, block_units[2] + 1)],
    ))),
    ('block4', nn.Sequential(OrderedDict(
        [('unit01', PreActBottleneck(cin=1024*wf, cout=2048*wf, cmid=512*wf, stride=2))] +
        [(f'unit{i:02d}', PreActBottleneck(cin=2048*wf, cout=2048*wf, cmid=512*wf)) for i in range(2, block_units[3] + 1)],
    ))),
]))
# pylint: enable=line-too-long

self.zero_head = zero_head
self.head = nn.Sequential(OrderedDict([
    ('gn', nn.GroupNorm(32, 2048*wf)),
    ('relu', nn.ReLU(inplace=True)),
    ('avg', nn.AdaptiveAvgPool2d(output_size=1)),
    ('conv', nn.Conv2d(2048*wf, head_size, kernel_size=1, bias=True)),
]))
```

6 Big Transfer (BiT): General Visual Representation Learning (2020)

❖ **BiT Pytorch : https://github.com/google-research/big_transfer, big_transfer_pytorch.ipynb**

```
def forward(self, x):
  x = self.head(self.body(self.root(x)))
  assert x.shape[-2:] == (1, 1)  # We should have no spatial shape left.
  return x[...,0,0]

def load_from(self, weights, prefix='resnet/'):
  with torch.no_grad():
    self.root.conv.weight.copy_(tf2th(weights[f'{prefix}root_block/standardized_conv2d/kernel']))
    # pylint: disable=line-too-long
    self.head.gn.weight.copy_(tf2th(weights[f'{prefix}group_norm/gamma']))
    self.head.gn.bias.copy_(tf2th(weights[f'{prefix}group_norm/beta']))
    if self.zero_head:
      nn.init.zeros_(self.head.conv.weight)
      nn.init.zeros_(self.head.conv.bias)
    else:
      self.head.conv.weight.copy_(tf2th(weights[f'{prefix}head/conv2d/kernel']))
      # pylint: disable=line-too-long
      self.head.conv.bias.copy_(tf2th(weights[f'{prefix}head/conv2d/bias']))

    for bname, block in self.body.named_children():
      for uname, unit in block.named_children():
        unit.load_from(weights, prefix=f'{prefix}{bname}/{uname}/')
```

```
KNOWN_MODELS = OrderedDict([
    ('BiT-M-R50x1', lambda *a, **kw: ResNetV2([3, 4, 6, 3], 1, *a, **kw)),
    ('BiT-M-R50x3', lambda *a, **kw: ResNetV2([3, 4, 6, 3], 3, *a, **kw)),
    ('BiT-M-R101x1', lambda *a, **kw: ResNetV2([3, 4, 23, 3], 1, *a, **kw)),
    ('BiT-M-R101x3', lambda *a, **kw: ResNetV2([3, 4, 23, 3], 3, *a, **kw)),
    ('BiT-M-R152x2', lambda *a, **kw: ResNetV2([3, 8, 36, 3], 2, *a, **kw)),
    ('BiT-M-R152x4', lambda *a, **kw: ResNetV2([3, 8, 36, 3], 4, *a, **kw)),
    ('BiT-S-R50x1', lambda *a, **kw: ResNetV2([3, 4, 6, 3], 1, *a, **kw)),
    ('BiT-S-R50x3', lambda *a, **kw: ResNetV2([3, 4, 6, 3], 3, *a, **kw)),
    ('BiT-S-R101x1', lambda *a, **kw: ResNetV2([3, 4, 23, 3], 1, *a, **kw)),
    ('BiT-S-R101x3', lambda *a, **kw: ResNetV2([3, 4, 23, 3], 3, *a, **kw)),
    ('BiT-S-R152x2', lambda *a, **kw: ResNetV2([3, 8, 36, 3], 2, *a, **kw)),
    ('BiT-S-R152x4', lambda *a, **kw: ResNetV2([3, 8, 36, 3], 4, *a, **kw)),
])
```

6 Big Transfer (BiT): General Visual Representation Learning (2020)

❖ **BiT Pytorch : https://github.com/google-research/big_transfer, big_transfer_pytorch.ipynb**

Boilerplate

```
from IPython.display import HTML, display

def progress(value, max=100):
    return HTML("""
        <progress
            value='{value}'
            max='{max}',
            style='width: 100%'
        >
            {value}
        </progress>
    """.format(value=value, max=max))

def stairs(s, v, *svs):
    """ Implements a typical "stairs" schedule for learning-rates.
    Best explained by example:
    stairs(s, 0.1, 10, 0.01, 20, 0.001)
    will return 0.1 if s<10, 0.01 if 10<=s<20, and 0.001 if 20<=s
    """
    for s0, v0 in zip(svs[::2], svs[1::2]):
        if s < s0:
            break
        v = v0
    return v

def rampup(s, peak_s, peak_lr):
  if s < peak_s:  # Warmup
    return s/peak_s * peak_lr
  else:
    return peak_lr

def schedule(s):
  step_lr = stairs(s, 3e-3, 200, 3e-4, 300, 3e-5, 400, 3e-6, 500, None)
  return rampup(s, 100, step_lr)
```

CIFAR-10 Example

```
import PIL

preprocess_train = tv.transforms.Compose([
    tv.transforms.Resize((160, 160), interpolation=PIL.Image.BILINEAR),  # It's the default,
just being explicit for the reader.
    tv.transforms.RandomCrop((128, 128)),
    tv.transforms.RandomHorizontalFlip(),
    tv.transforms.ToTensor(),
    tv.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Get data into [-1, 1]
])

preprocess_eval = tv.transforms.Compose([
    tv.transforms.Resize((128, 128), interpolation=PIL.Image.BILINEAR),
    tv.transforms.ToTensor(),
    tv.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = tv.datasets.CIFAR10(root='./data', train=True, download=True,
transform=preprocess_train)
testset = tv.datasets.CIFAR10(root='./data', train=False, download=True,
transform=preprocess_eval)
```

6 Big Transfer (BiT): General Visual Representation Learning (2020)

❖ **BiT Pytorch : https://github.com/google-research/big_transfer, big_transfer_pytorch.ipynb**

Eval pre-trained model (verify conversion)

```
weights_cifar10 = get_weights('BiT-M-R50x1-CIFAR10')

model = ResNetV2(ResNetV2.BLOCK_UNITS['r50'], width_factor=1, head_size=10)  # NOTE: No new head.
model.load_from(weights_cifar10)
model.to(device);

def eval_cifar10(model, bs=100, progressbar=True):
  loader_test = torch.utils.data.DataLoader(testset, batch_size=bs, shuffle=False, num_workers=2)

  model.eval()

  if progressbar is True:
    progressbar = display(progress(0, len(loader_test)), display_id=True)

  preds = []
  with torch.no_grad():
    for i, (x, t) in enumerate(loader_test):
      x, t = x.to(device), t.numpy()
      logits = model(x)
      _, y = torch.max(logits.data, 1)
      preds.extend(y.cpu().numpy() == t)
      progressbar.update(progress(i+1, len(loader_test)))

  return np.mean(preds)

print("Expected: 97.61%")
print(f"Accuracy: {eval_cifar10(model):.2%}")
```

Find indices to create a 5-shot CIFAR10 variant

```
preprocess_tiny = tv.transforms.Compose([tv.transforms.CenterCrop((2, 2)),
tv.transforms.ToTensor()])
trainset_tiny = tv.datasets.CIFAR10(root='./data', train=True, download=True,
transform=preprocess_tiny)
loader = torch.utils.data.DataLoader(trainset_tiny, batch_size=50000, shuffle=False,
num_workers=2)
images, labels = iter(loader).next()

indices = {cls: np.random.choice(np.where(labels.numpy() == cls)[0], 5, replace=False)
for cls in range(10)}

print(indices)

fig = plt.figure(figsize=(10, 4))
ig = ImageGrid(fig, 111, (5, 10))
for c, cls in enumerate(indices):
  for r, i in enumerate(indices[cls]):
    img, _ = trainset[i]
    ax = ig.axes_column[c][r]
    ax.imshow((img.numpy().transpose([1, 2, 0]) * 127.5 + 127.5).astype(np.uint8))
    ax.set_axis_off()
fig.suptitle('The whole 5-shot CIFAR10 dataset');

train_5shot = torch.utils.data.Subset(trainset, indices=[i for v in indices.values() for i in v])
len(train_5shot)
```

6 Big Transfer (BiT): General Visual Representation Learning (2020)

❖ **BiT Pytorch : https://github.com/google-research/big_transfer, big_transfer_pytorch.ipynb**

Fine-tune BiT-M on this 5-shot CIFAR10 variant

```
model = ResNetV2(ResNetV2.BLOCK_UNITS['r50'], width_factor=1, head_size=10,
zero_head=True)
model.load_from(weights)
model.to(device);

# Yes, we still use 512 batch-size! Maybe something else is even better, who knows.
# loader_train = torch.utils.data.DataLoader(train_5shot, batch_size=512, shuffle=True,
num_workers=2)

# NOTE: This is necessary when the batch-size is larger than the dataset.
sampler = torch.utils.data.RandomSampler(train_5shot, replacement=True,
num_samples=256)
loader_train = torch.utils.data.DataLoader(train_5shot, batch_size=256, num_workers=2,
sampler=sampler)

crit = nn.CrossEntropyLoss()
opti = torch.optim.SGD(model.parameters(), lr=0.003, momentum=0.9)
model.train();

S = 500
def schedule(s):
  step_lr = stairs(s, 3e-3, 200, 3e-4, 300, 3e-5, 400, 3e-6, S, None)
  return rampup(s, 100, step_lr)

pb_train = display(progress(0, S), display_id=True)
pb_test = display(progress(0, 100), display_id=True)
losses = [[]]
accus_train = [[]]
accus_test = []
```

```
steps_per_iter = 512 // loader_train.batch_size

while len(losses) < S:
  for x, t in loader_train:
    x, t = x.to(device), t.to(device)

    logits = model(x)
    loss = crit(logits, t) / steps_per_iter
    loss.backward()
    losses[-1].append(loss.item())

    with torch.no_grad():
      accus_train[-1].extend(torch.max(logits, dim=1)[1].cpu().numpy() == t.cpu().numpy())

    if len(losses[-1]) == steps_per_iter:
      losses[-1] = sum(losses[-1])
      losses.append([])
      accus_train[-1] = np.mean(accus_train[-1])
      accus_train.append([])

      # Update learning-rate according to schedule, and stop if necessary
      lr = schedule(len(losses) - 1)
      for param_group in opti.param_groups:
        param_group['lr'] = lr

      opti.step()
      opti.zero_grad()

      pb_train.update(progress(len(losses) - 1, S))
      print(f'\r[Step {len(losses) - 1}] loss={losses[-2]:.2e} '
            f'train accu={accus_train[-2]:.2%} '
            f'test accu={accus_test[-1] if accus_test else 0:.2%} '
            f'(lr={lr:g})', end='', flush=True)

      if len(losses) % 25 == 0:
        accus_test.append(eval_cifar10(model, progressbar=pb_test))
        model.train()
```