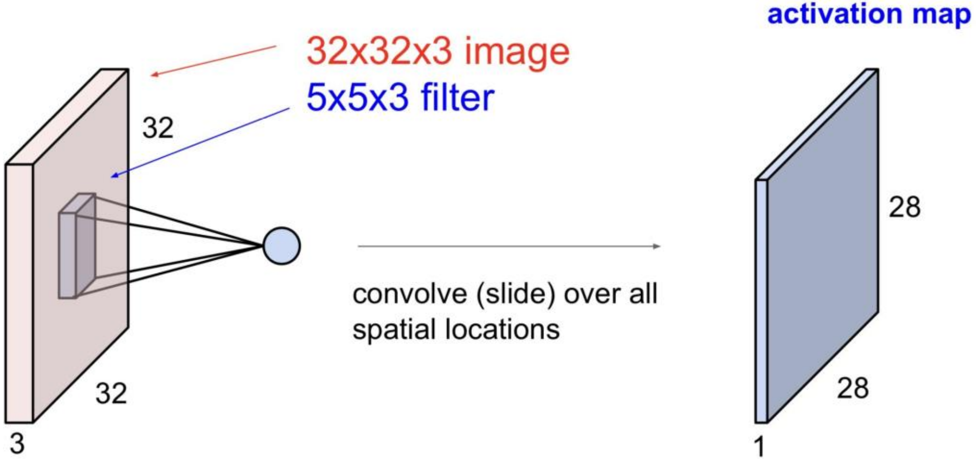
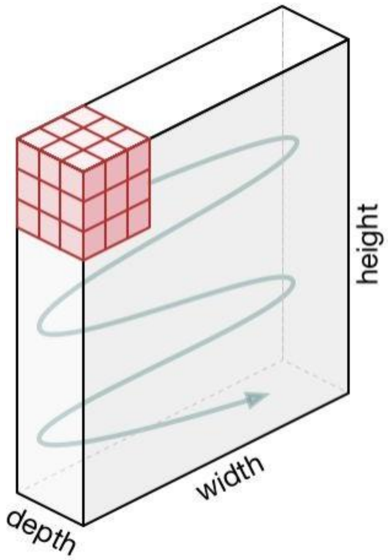
CNN

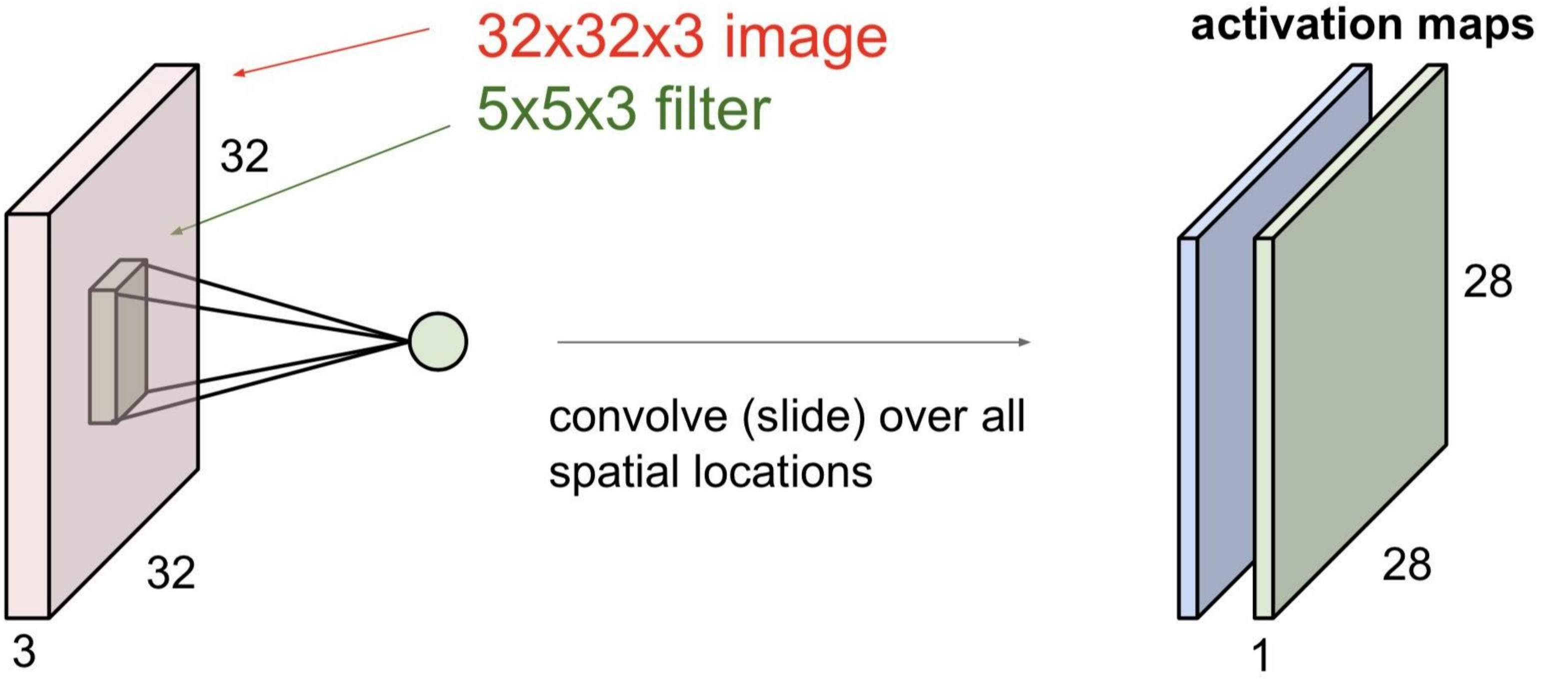
Deep Learning - Fall 2020

Convolution

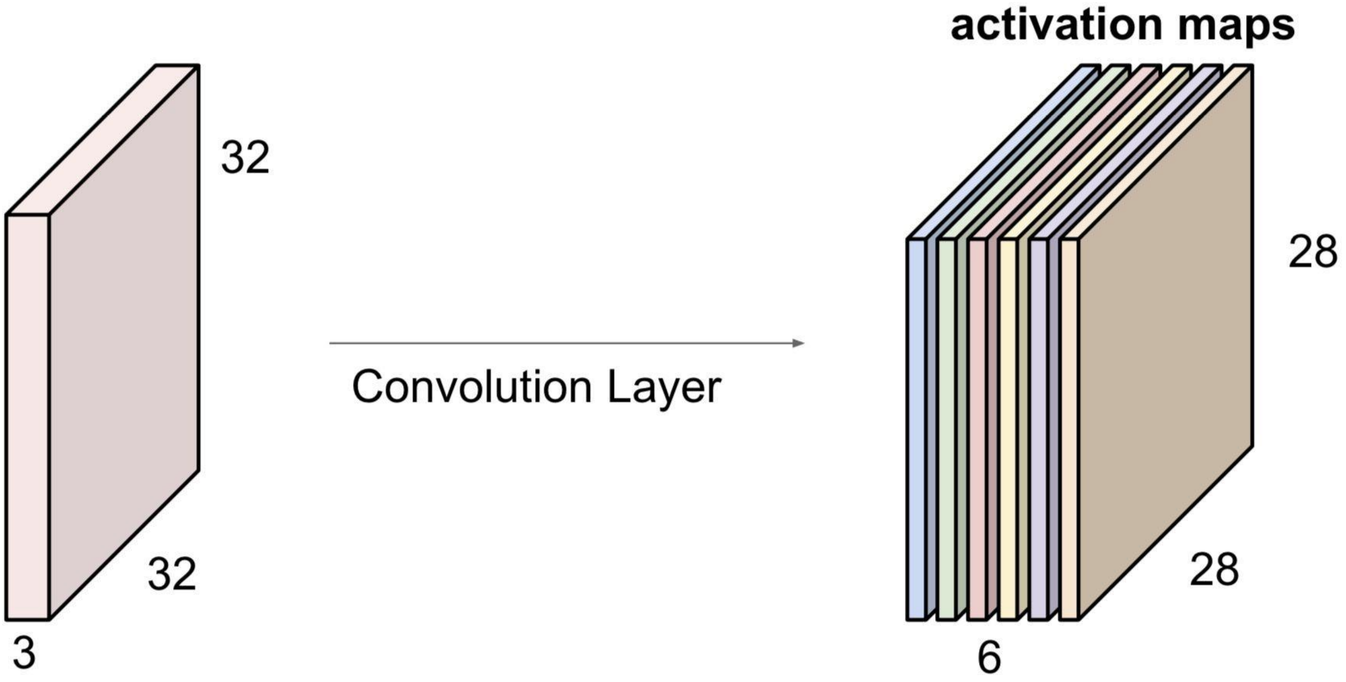




Convolution



Convolution



Convolution

- N: Input size (height & width), F:Filter size, S: Stride, P: Padding size
- Output size: O = (N + 2 * P - F) / S + 1
- K: number of filters
- Depth = K = Number of output channels (C_out)
- Transforms (N * N * C_in) into (O * O * C_out)
- K biases
- # of parameters: F * F * C_in * K + K
- Common settings: K = powers of 2 (32, 64, 128, ...)
- [F=3, S=1, P=1], [F=5, S=1, P=2], [F=1, S=1, P=0]
- Pooling (Common settings: [F=2, S=2], [F=3, S=2])

Convolution

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

Convolution

- Ex.

```
>>> layer = Conv2d(in_channels=64, out_channels=128, kernel_size=3)
>>> layer.weight.shape
torch.Size([128, 64, 3, 3])
>>> layer.bias.shape
torch.Size([128])
```

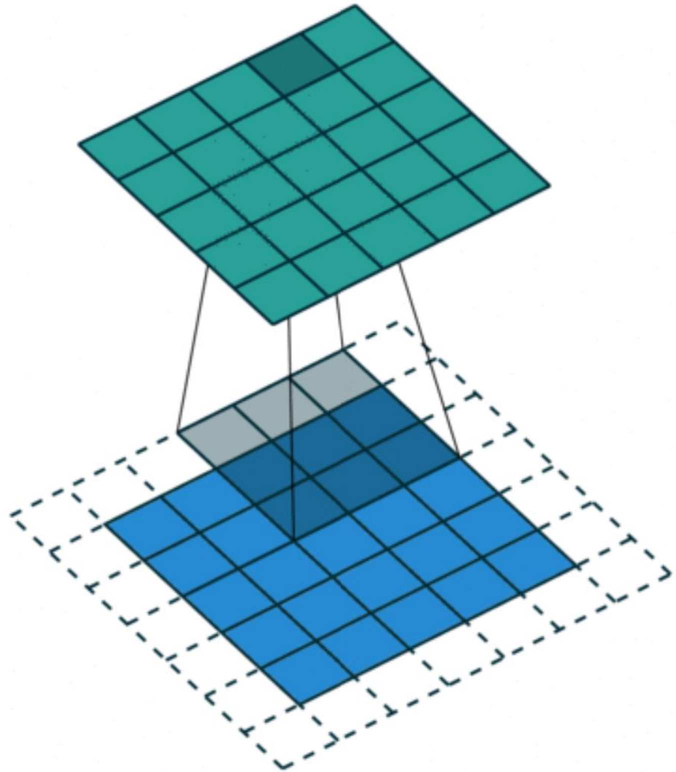
Types of Convolution

Types of Convolution

- Dilated Convolutions
- Transposed Convolutions
- Separable Convolutions

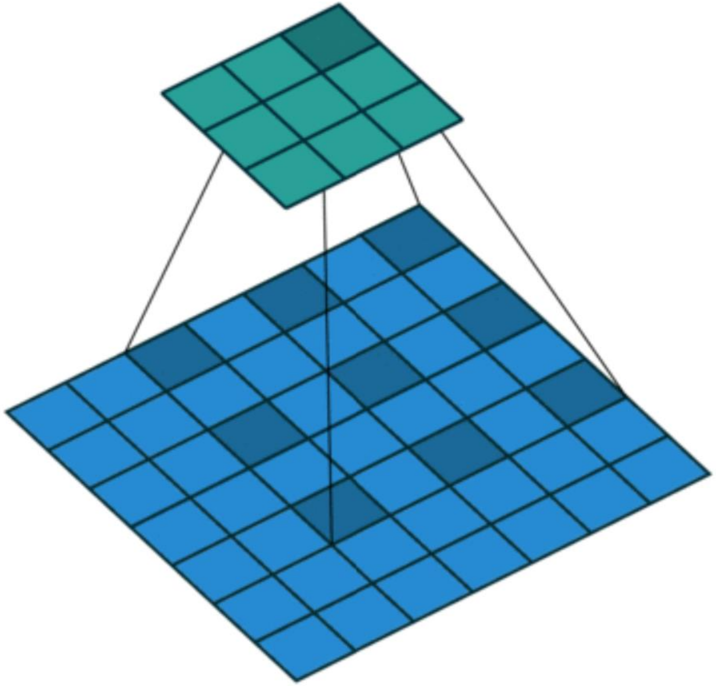
Example:

- Kernel Size = 3
- Stride = 1
- Padding = 1



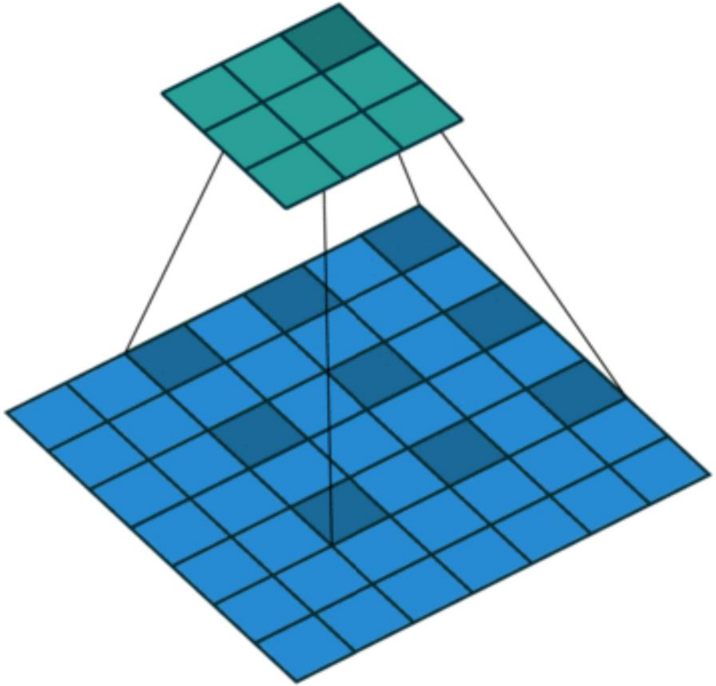
Dilated Convolutions

- Dilated convolutions introduce another parameter to convolutional layers called the **dilation rate**.
- This defines a spacing between the values in a kernel.
- Ex: A 3x3 kernel with a dilation rate of 2 will have the same field of view as a 5x5 kernel, while only using 9 parameters.



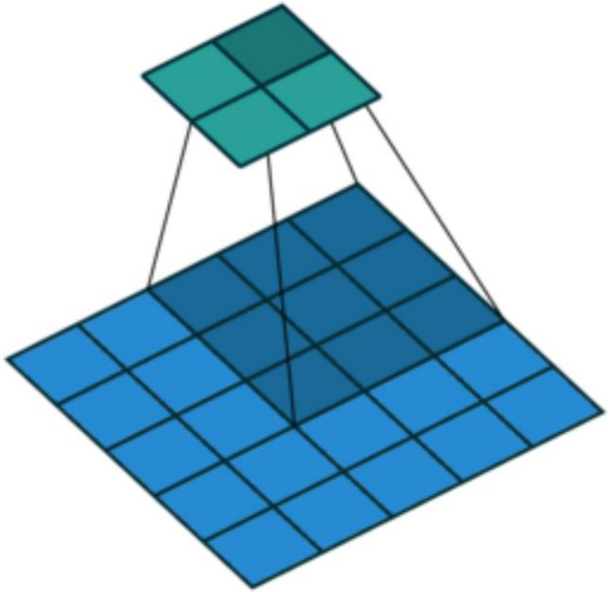
Dilated Convolutions (cont.)

- This delivers a wider field of view at the same computational cost.
- Dilated convolutions are particularly popular in the field of real-time segmentation.
- Use them if you need a wide field of view and cannot afford multiple convolutions or larger kernels.



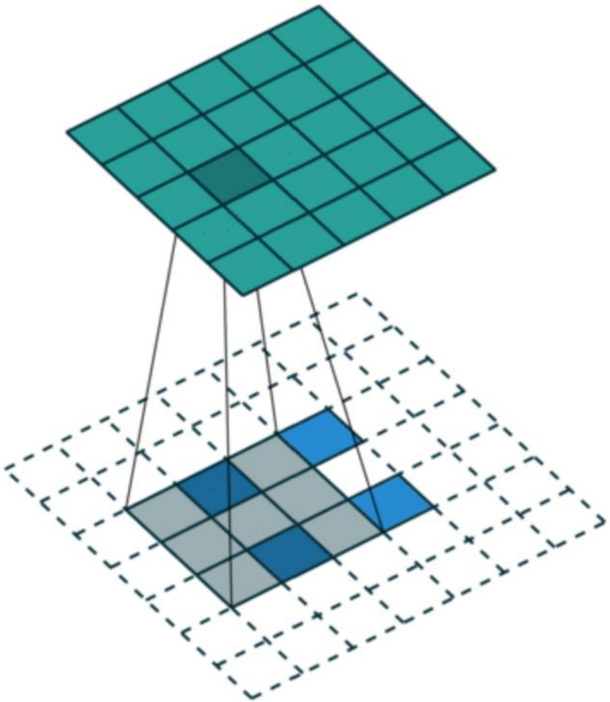
Transposed Convolutions

- a.k.a. deconvolutions or fractionally strided convolutions
- Some sources use the name deconvolution, which is inappropriate. An actual deconvolution reverts the process of a convolution. (mathematically inverse process)



Transposed Convolutions (cont).

- This way we can combine the upscaling of an image with a convolution, instead of doing two separate processes.
- This may not be the mathematical inverse, but for Encoder-Decoder architectures, it's still very helpful.
- It merely reconstructs the spatial resolution from before and performs a convolution.



Separable Convolutions

- In a separable convolution, we can split the kernel operation into multiple steps.
- Let’s express a convolution as **y = conv(x, k)** where **y** is the output image, **x** is the input image, and **k** is the kernel. Easy. Next, let’s assume k can be calculated by: **k = k1.dot(k2)**. This would make it a separable convolution because instead of doing a 2D convolution with k, we could get to the same result by doing 2 1D convolutions with k1 and k2.

-1	0	+1
-2	0	+2
-1	0	+1

x filter

+1	+2	+1
0	0	0
-1	-2	-1

y filter

Sobel X and Y filters

Custom DataLoader

Custom Dataset

- A lot of effort in solving any machine learning problem goes in to preparing the data.
- PyTorch provides many tools to make data loading easy and hopefully, to make your code more readable.

```python
import torch

class Dataset(torch.utils.data.Dataset):
  'Characterizes a dataset for PyTorch'
  def __init__(self, list_IDs, labels):
        'Initialization'
        self.labels = labels
        self.list_IDs = list_IDs

  def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

  def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDs[index]

        # Load data and get label
        X = torch.load('data/' + ID + '.pt')
        y = self.labels[ID]

        return X, y
```

Dataset class

- A lot of effort in solving any machine learning problem goes in to preparing the data.
- torch.utils.data.Dataset is an abstract class representing a dataset.
- Your custom dataset should inherit Dataset and override the following methods:
    - __len__ so that len(dataset) returns the size of the dataset.
    - __getitem__ to support the indexing such that dataset[i] can be used to get ith sample
    - We will read the csv in __init__ but leave the reading of images to __getitem__
    - This is memory efficient because all the images are not stored in the memory at once but read as required.
    - Our dataset will take an optional argument transform so that any required processing can be applied on the sample.

Loading data

```
# Parameters
params = {'batch_size': 64,
          'shuffle': True,
          'num_workers': 6}
max_epochs = 100

# Datasets
partition = # IDs
labels = # Labels

# Generators
training_set = Dataset(partition['train'], labels)
training_generator = torch.utils.data.DataLoader(training_set, **params)

validation_set = Dataset(partition['validation'], labels)
validation_generator = torch.utils.data.DataLoader(validation_set, **params)

# Loop over epochs
for epoch in range(max_epochs):
    # Training
    for local_batch, local_labels in training_generator:
        # Transfer to GPU
        local_batch, local_labels = local_batch.to(device), local_labels.to(device)

        # Model computations
        [...]
```

Transformers, Data Augmentation

Why Transforms?

- One issue with most image data is that the samples are not of the same size.
- Moreover, we may need to crop or resize our images according to our application
- Most neural networks expect the images of a fixed size.
- Therefore, we will need to write some prepocessing code.
- You can write your own custom transforms or use TorchVision predefined transforms.

  - `Rescale`: to scale the image
  - `RandomCrop`: to crop from image randomly. This is data augmentation.
  - `ToTensor`: to convert the numpy images to torch images (we need to swap axes).

TorchVision

- You might not even have to write custom classes.
- torchvision package provides some common datasets and transforms.
- E.g. One of the more generic datasets available in torchvision is ImageFolder.
- It assumes that images are organized in the following way:

```
root/ants/xxx.png
root/ants/xxy.jpeg
root/ants/xxz.png
.
.
.
root/bees/123.jpg
root/bees/nsdf3.png
root/bees/asd932_.png
```

```python
import torch
from torchvision import transforms, datasets

data_transform = transforms.Compose([
        transforms.RandomSizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])
    ])
hymenoptera_dataset = datasets.ImageFolder(root='hymenoptera_data/train',
                                           transform=data_transform)
dataset_loader = torch.utils.data.DataLoader(hymenoptera_dataset,
                                             batch_size=4, shuffle=True,
                                             num_workers=4)
```

Normalization

Why Normalization?

- It normalizes each feature so that they maintains the contribution of every feature, as some feature has higher numerical value than others. This way our network can be unbiased (to higher value features).
- It makes the Optimization faster because normalization doesn't allow weights to explode all over the place and restricts them to a certain range.
- An unintended benefit of Normalization is that it helps network in Regularization
- Batch Norm makes loss surface smoother
- It reduces **Internal Covariate Shift**. It is the change in the distribution of network activations due to the change in network parameters during training. To improve the training, we seek to reduce the internal covariate shift.

Batch Normalization

- Batch normalization is a method that normalizes activations in a network across the mini-batch of definite size. For each feature, batch normalization computes the mean and variance of that feature in the mini-batch. It then subtracts the mean and divides the feature by its mini-batch standard deviation.

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

Batch Norm (cont.)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Weight Normalization

- Weight Normalization **normalize weights of a layer** instead of normalizing the activations directly.
- It reparameterizes the weights w as :

$$\boldsymbol{w} = \frac{g}{\|\boldsymbol{v}\|}\boldsymbol{v}$$

Layer Normalization

- Layer normalization normalizes input across the features instead of normalizing input features across the batch dimension in batch normalization.
- performs better than batch norm in case of **RNN**s

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_{ij}$$
$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_{ij} - \mu_i)^2$$
$$\hat{x_{ij}} = \frac{x_{ij}-\mu_i}{\sqrt{\sigma_i^2+\epsilon}}$$

Switchable Normalization

- This paper proposed switchable normalization, a method that uses a weighted average of different mean and variance statistics from batch normalization, instance normalization, and layer normalization.
- The authors showed that switch normalization could potentially outperform batch normalization on tasks such as image classification and object detection.
- The paper showed that the instance normalization were used more often in earlier layers, batch normalization was preferred in the middle and layer normalization being used in the last more often. Smaller batch sizes lead to a preference towards layer normalization and instance normalization.

Transfer Learning

Transfer learning

- Using a network pretrained on a large dataset (e.g. ImageNet)
- Fine-tune the pre-trained network on our own dataset
- If the new dataset is very small, it’s better to train only the final layers of the network to avoid overfitting
- Add new layers and retrain them only
- The earlier features of a ConvNet contain more generic features (e.g. edge detectors or color blob detectors), but later layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset

Transfer learning

- Implementation

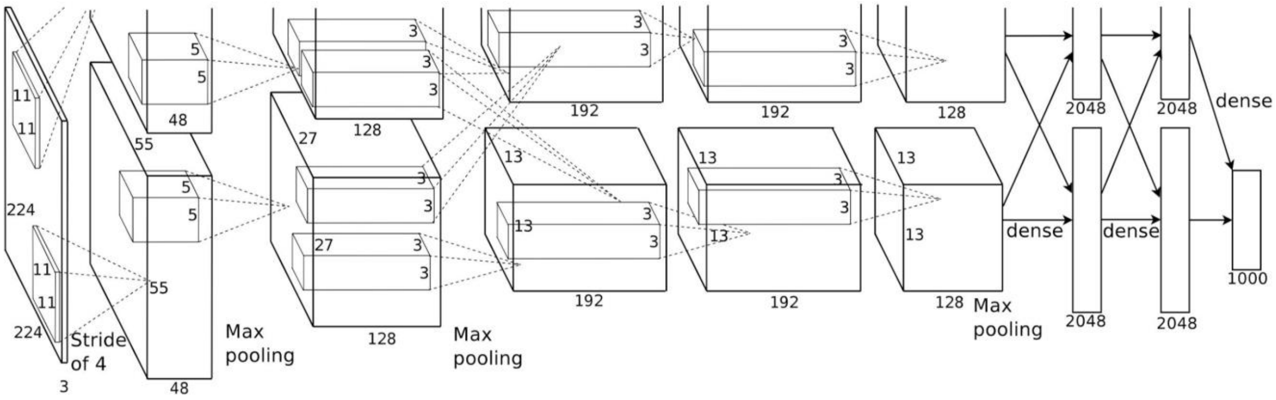
```
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
```

CNN Architectures

ImageNet

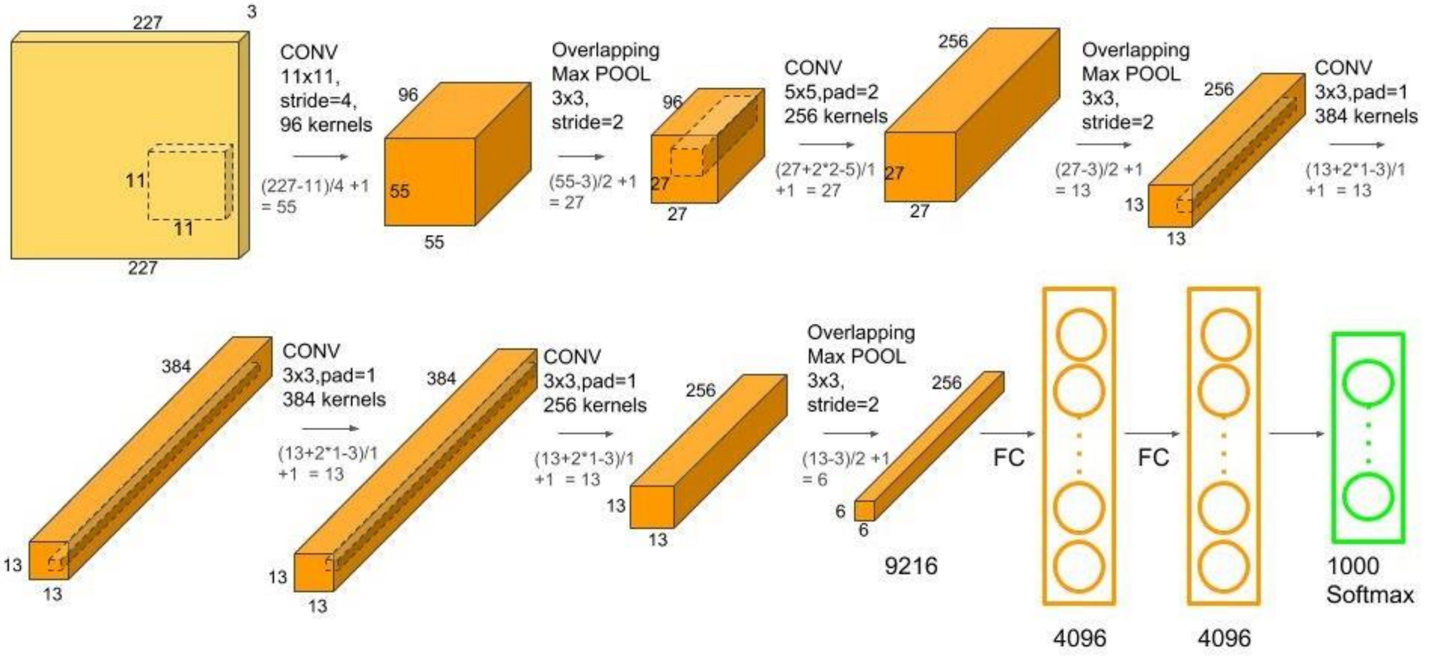
- Near 15 million images containing more than 20,000 categories
- Labeled by Amazon Mechanical Turk
- **I**mageNet **L**arge **S**cale **V**isual **R**ecognition **C**hallenge (ILSVRC)
- ILSVRC is on a subset of ImageNet containing more than 1 million images in 1000 classes

AlexNet



- Train on 2 GPUs
- GTX-580, 3GB memory
- Cross-GPU parallelization
- Local response normalization and Dropout
- Data augmentation and preprocessing
- 60 million parameters
- 17% top-5 error rate

AlexNet

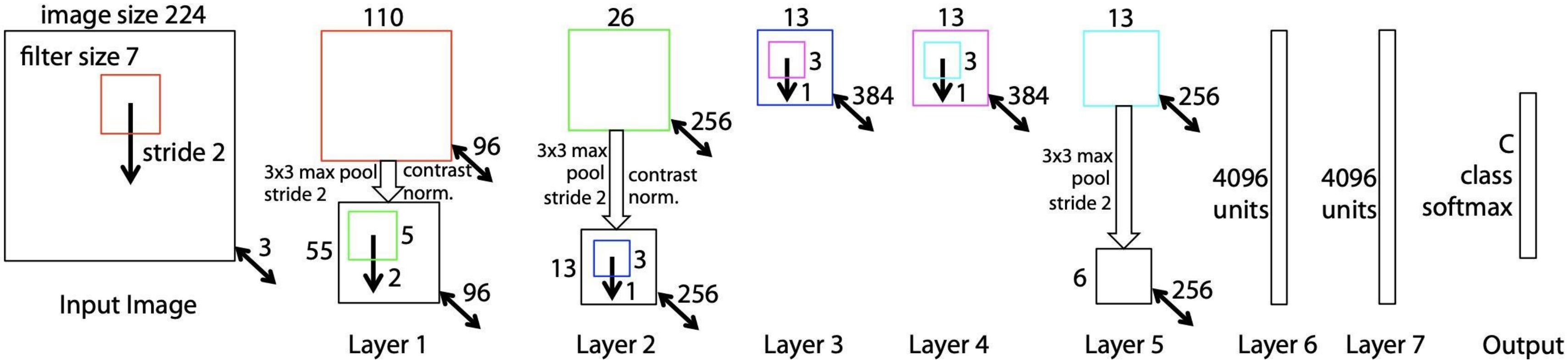

227
3
CONV
11x11,
stride=4,
96 kernels
(227-11)/4 +1
= 55
11
11
227
96
55
55
Overlapping
Max POOL
3x3,
stride=2
(55-3)/2 +1
= 27
96
27
27
CONV
5x5,pad=2
256 kernels
(27+2*2-5)/1
+1 = 27
256
27
27
Overlapping
Max POOL
3x3,
stride=2
(27-3)/2 +1
= 13
256
13
13
CONV
3x3,pad=1
384 kernels
(13+2*1-3)/1
+1 = 13
384
13
13
CONV
3x3,pad=1
384 kernels
(13+2*1-3)/1
+1 = 13
384
13
13
CONV
3x3,pad=1
256 kernels
(13+2*1-3)/1
+1 = 13
256
13
13
Overlapping
Max POOL
3x3,
stride=2
(13-3)/2 +1
= 6
256
6
6
9216
FC
FC
4096
4096
1000
Softmax

ZFNet

Alexnet but:

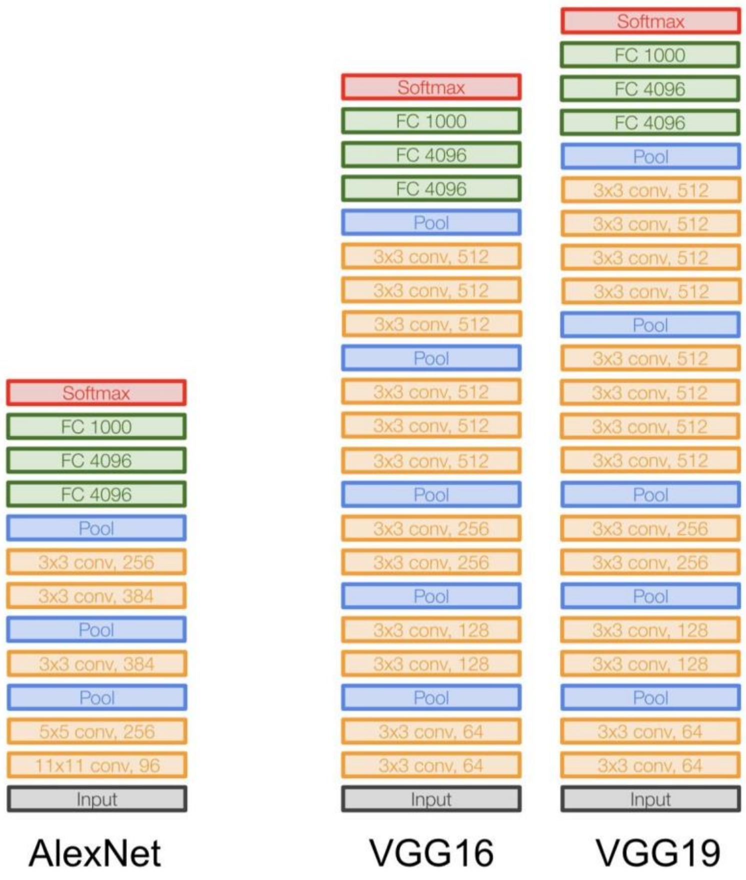
Conv1: change from (11 * 11 stride 4) to (7 * 7 stride 2)

Conv3, 4, 5: instead of 384, 384, 256 filters use 512, 1024, 512



VGG

- 3*3 Conv, stride 1, pad 1
- 2*2 max pool stride 2
- Stack of 3*3 filters has the Same receptive field as 7*7
- More non-linearities
- Fewer parameters
- 138M parameters



VGG

Implementation

```python
class VGG(nn.Module):

    def __init__(self, features, num_classes=1000, init_weights=True):
        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):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

➤Source code available at https://pytorch.org/docs/stable/_modules/torchvision/models/vgg.html

VGG

Implementation

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


cfgs = {
    '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'],
}
```

➤Source code available at https://pytorch.org/docs/stable/_modules/torchvision/models/vgg.html

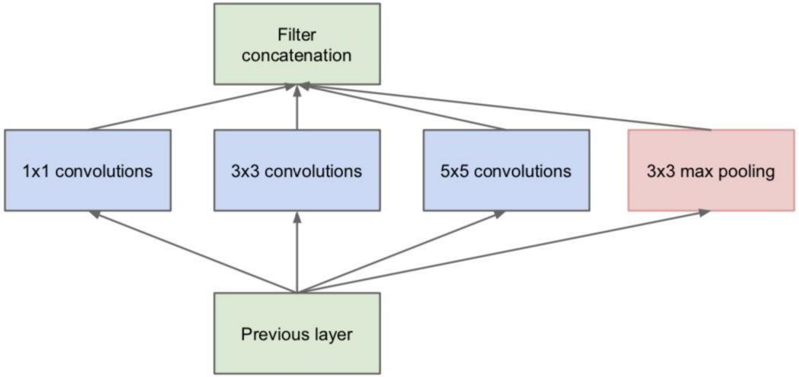
VGG

Implementation

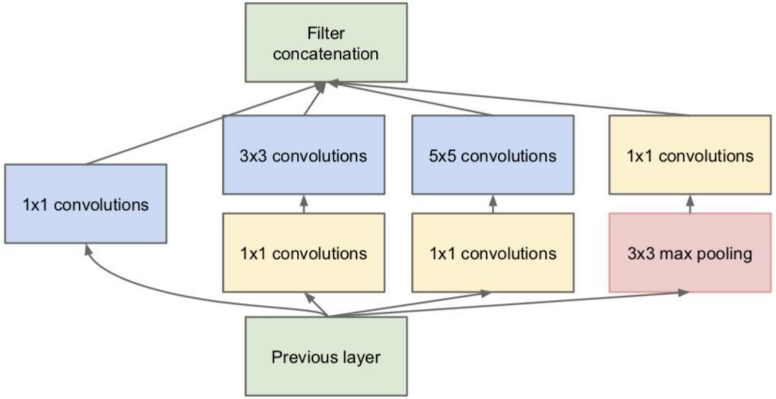
```
    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', 
nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
           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

➤Source code available at https://pytorch.org/docs/stable/_modules/torchvision/models/vgg.html

GoogLeNet

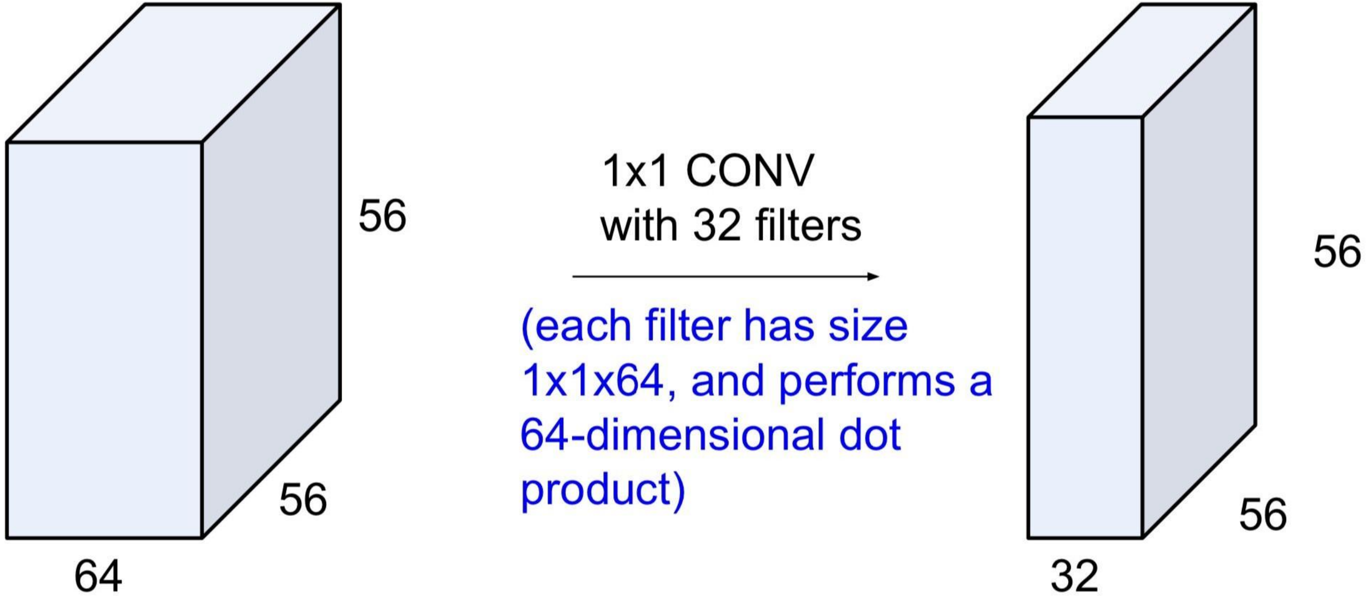


(a) Inception module, naïve version

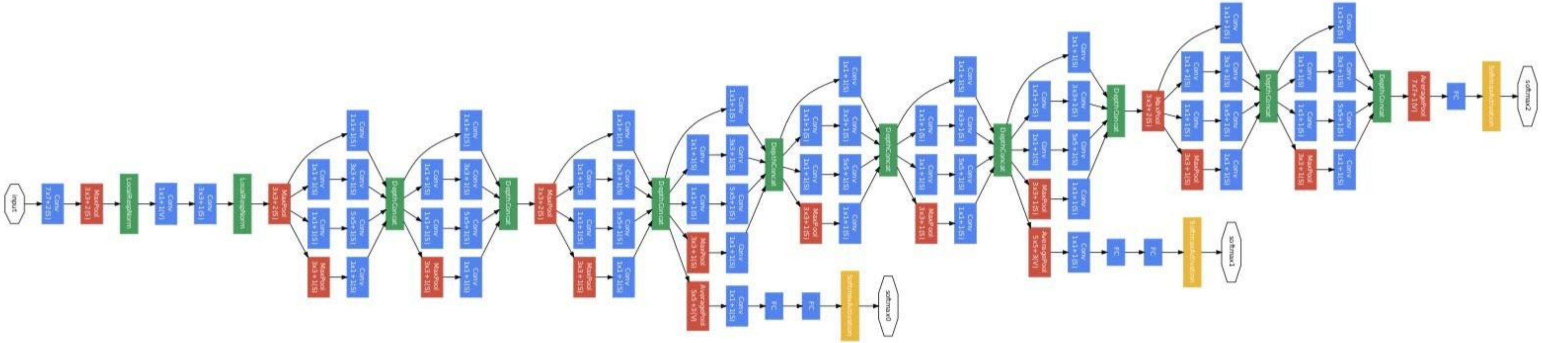


(b) Inception module with dimension reductions

GoogLeNet

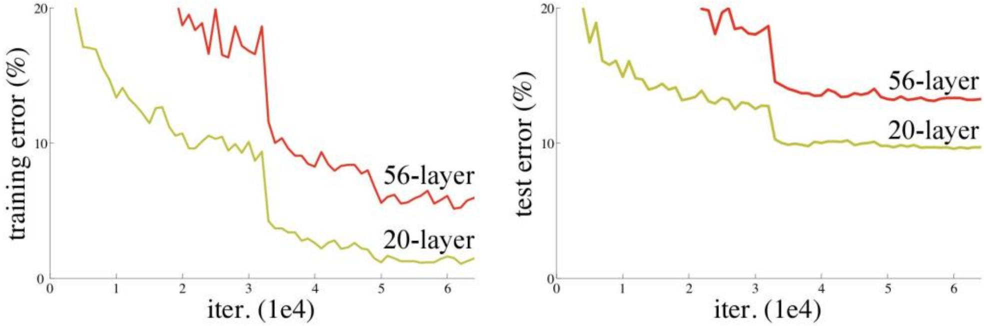

56
56
64
1x1 CONV
with 32 filters
(each filter has size
1x1x64, and performs a
64-dimensional dot
product)
56
56
32

GoogLeNet

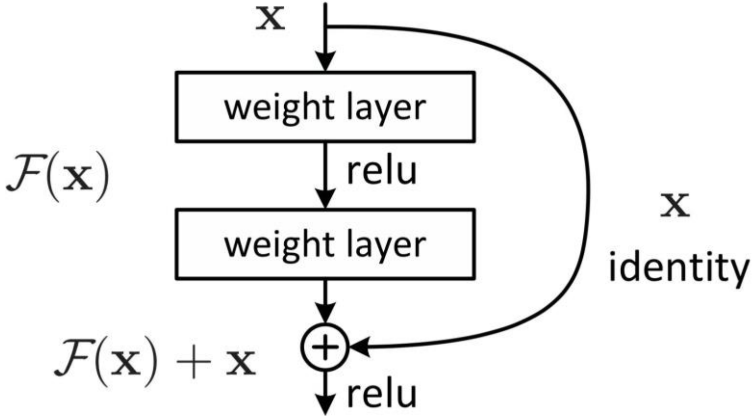


ResNet

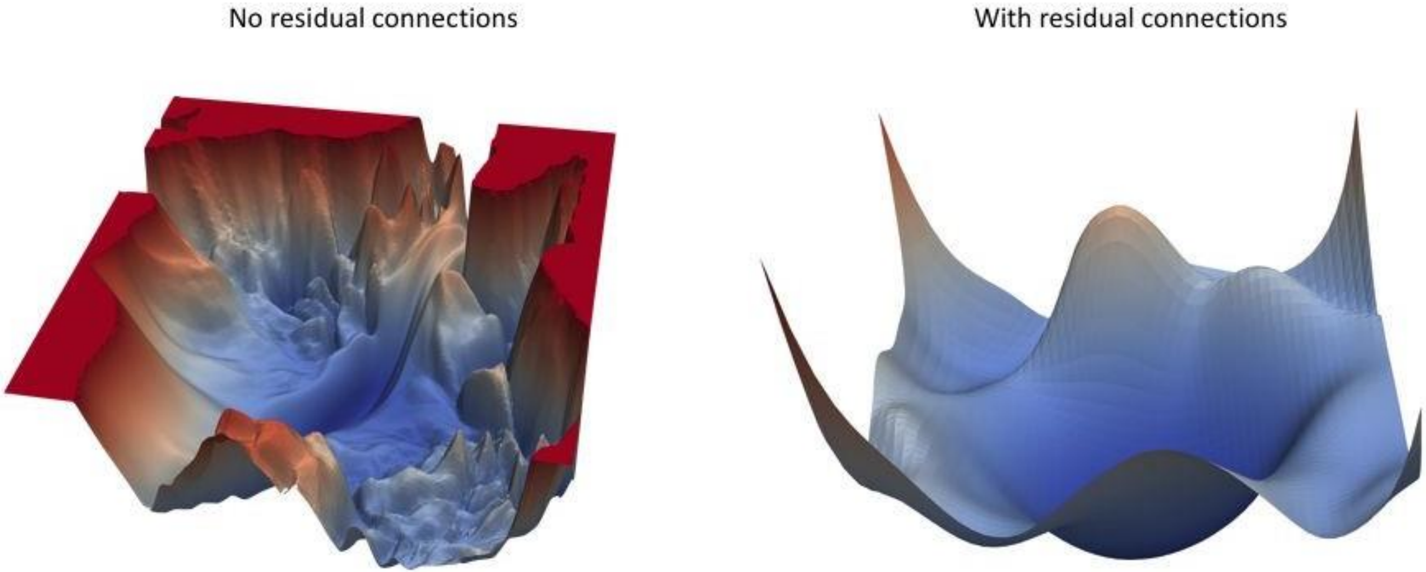
-



training error (%)
20
10
0
0
1
2
3
4
5
6
iter. (1e4)
56-layer
20-layer
test error (%)
20
10
0
0
1
2
3
4
5
6
iter. (1e4)
56-layer
20-layer



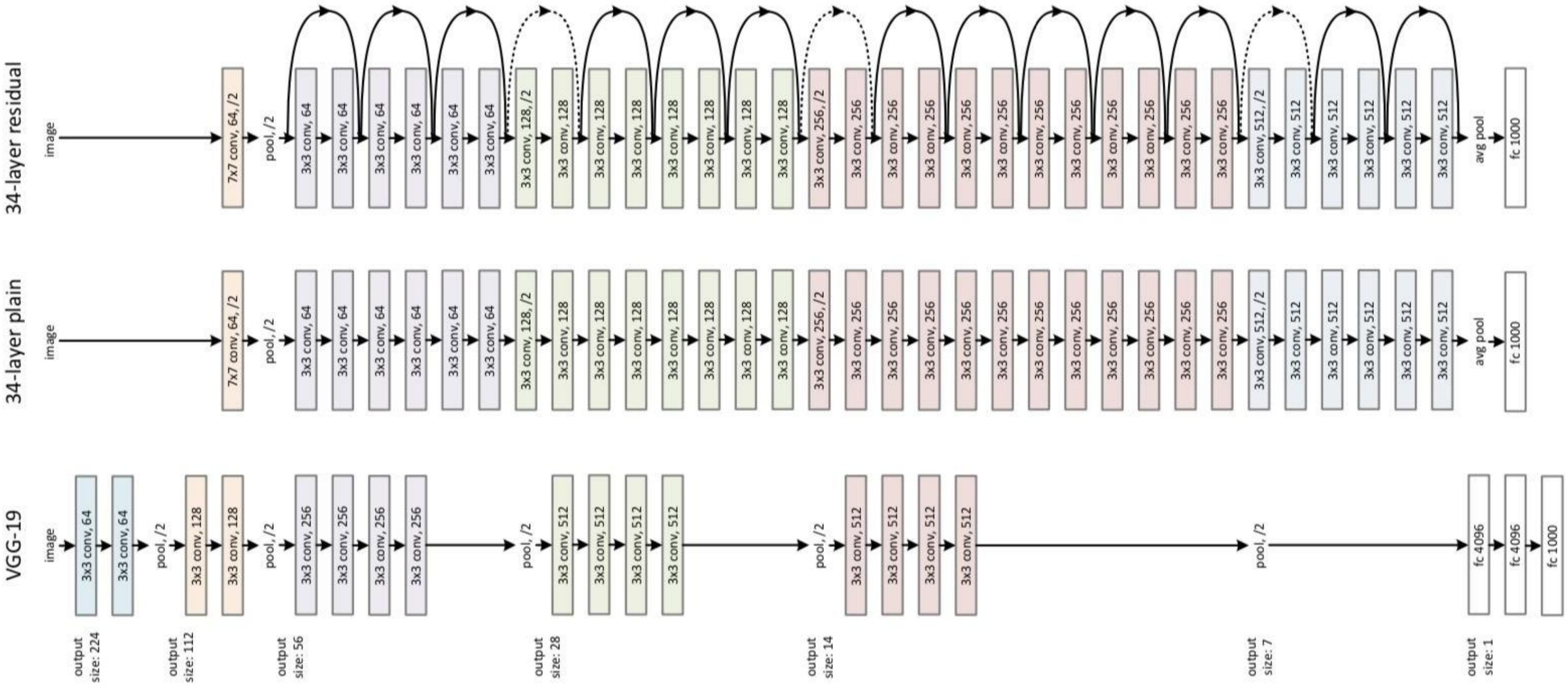
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu



Same general network architecture

Image Credit

ResNet


34-layer residual
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000
34-layer plain
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
avg pool
fc 1000
VGG-19
image
3x3 conv, 64
3x3 conv, 64
pool, /2
3x3 conv, 128
3x3 conv, 128
pool, /2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
fc 4096
fc 4096
fc 1000
output size: 224
output size: 112
output size: 56
output size: 28
output size: 14
output size: 7
output size: 1

ResNet

Implementation

- Conv1*1 and Conv3*3

```
def conv3x3(in_planes, out_planes, stride=1, groups=1, dilation=1):
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                     padding=dilation, groups=groups, bias=False, dilation=dilation)


def conv1x1(in_planes, out_planes, stride=1):
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride, bias=False)
```

➤Source code available at https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

ResNet

Implementation

- Residual blocks

```
class BasicBlock(nn.Module):
    expansion = 1
    __constants__ = ['downsample']

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in BasicBlock")
        # Both self.conv1 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride
```

➤Source code available at https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

ResNet

Implementation

- Residual blocks

```python
def forward(self, x):
    identity = x

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

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

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)

    return out
```

➤Source code available at https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

ResNet

Implementation

- Whole model

```
self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
                       bias=False)
self.bn1 = norm_layer(self.inplanes)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2,
                               dilate=replace_stride_with_dilation[0])
self.layer3 = self._make_layer(block, 256, layers[2], stride=2,
                               dilate=replace_stride_with_dilation[1])
self.layer4 = self._make_layer(block, 512, layers[3], stride=2,
                               dilate=replace_stride_with_dilation[2])
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion, 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.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)
```

➤Source code available at https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

References & Acknowledgements

- https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel
- https://pytorch.org/tutorials/beginner/data_loading_tutorial.html
- https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
- Stanford CS231 Course material
- https://medium.com/techspace-usict/normalization-techniques-in-deep-neural-networks-9121bf100d8
- https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d