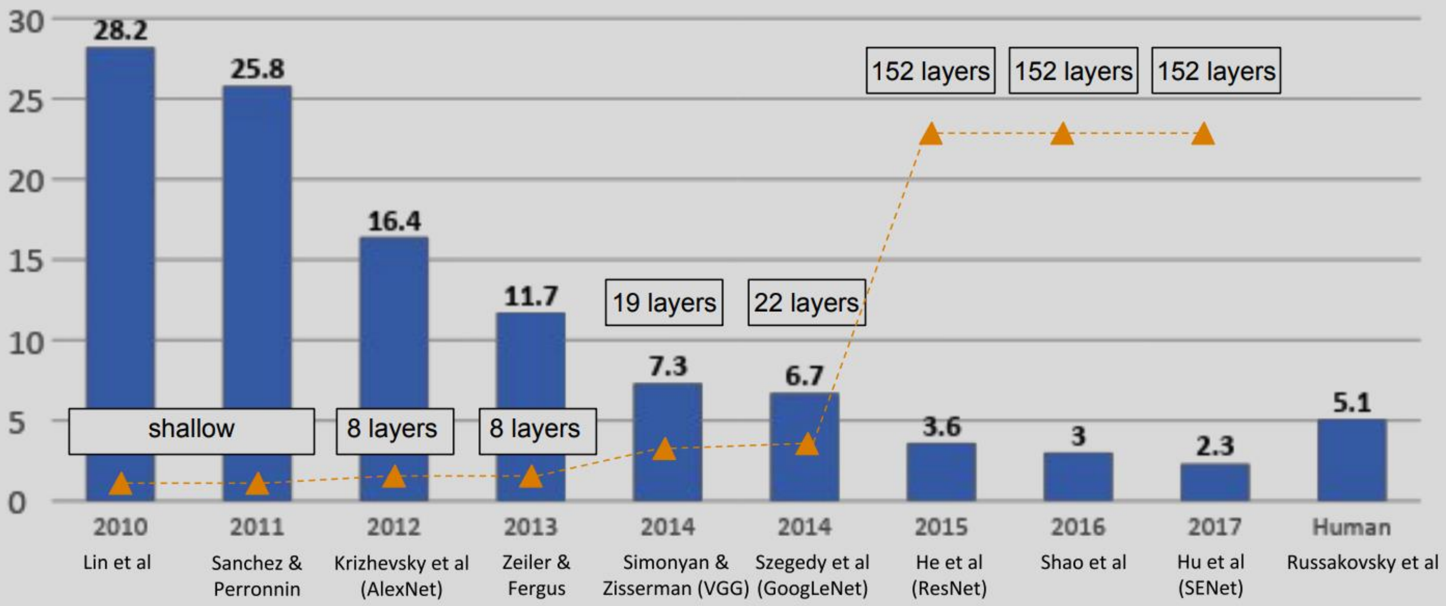
Computer Vision

Lecture 05: Convolutional Neural Network-2

Alex Net

1. Dropout and data augmentation are used.
2. ReLU is first used (Proves faster convergence).
3. Trained with Multi-GPUs.
4. First architecture that recorded as the SOTA in ImageNet challenge using deep learning.



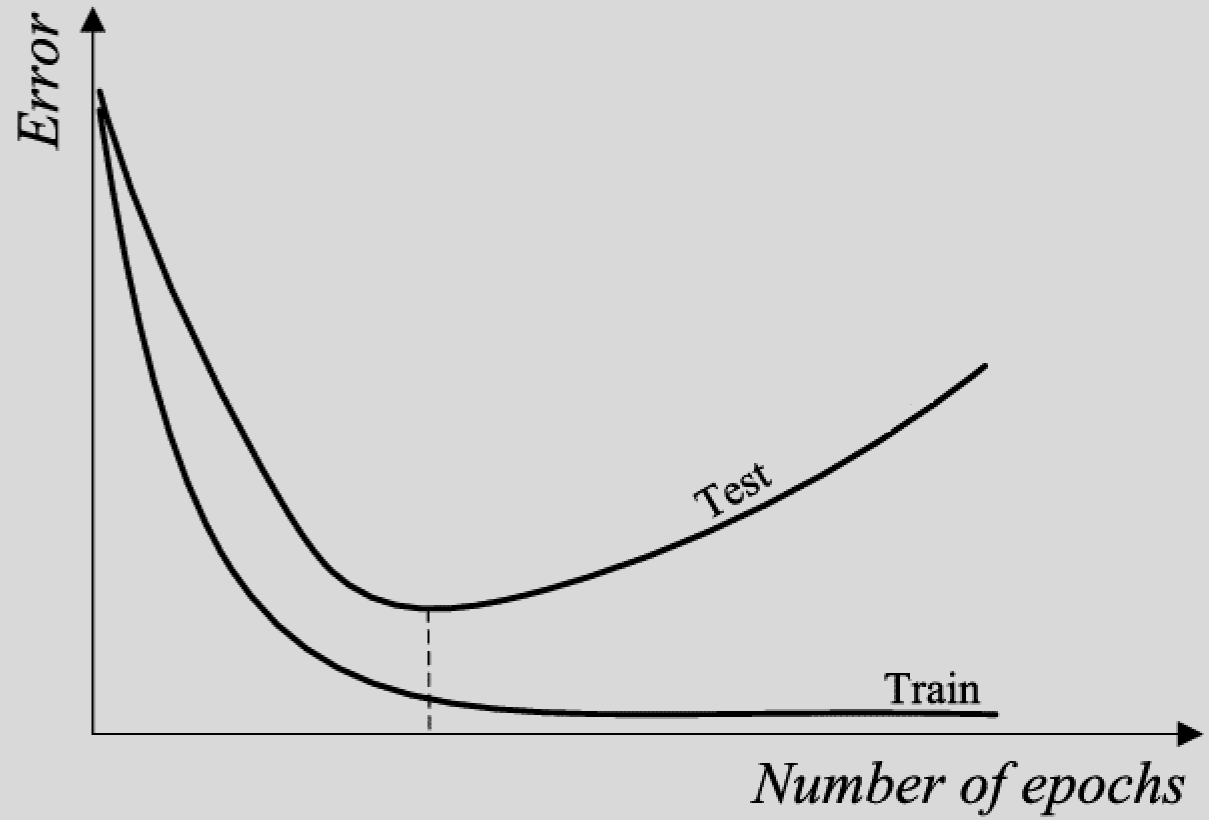
Alex Net

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

    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            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(),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Linear(4096, num_classes),
        )

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

Over-fitting

```
Epoch: 0 Train loss: 2.2817 Test loss: 2.2540 Train accuracy: 14.16 Test accuracy: 34.11
Epoch: 1 Train loss: 2.1948 Test loss: 2.0867 Train accuracy: 50.41 Test accuracy: 56.95
Epoch: 2 Train loss: 1.8551 Test loss: 1.5684 Train accuracy: 56.49 Test accuracy: 62.10
Epoch: 3 Train loss: 1.3052 Test loss: 1.0489 Train accuracy: 67.44 Test accuracy: 74.17
Epoch: 4 Train loss: 0.9017 Test loss: 0.7621 Train accuracy: 77.70 Test accuracy: 81.95
Epoch: 5 Train loss: 0.6930 Test loss: 0.6112 Train accuracy: 82.86 Test accuracy: 85.24
Epoch: 6 Train loss: 0.5783 Test loss: 0.5238 Train accuracy: 85.29 Test accuracy: 86.71
Epoch: 7 Train loss: 0.5090 Test loss: 0.4686 Train accuracy: 86.75 Test accuracy: 87.75
Epoch: 8 Train loss: 0.4633 Test loss: 0.4298 Train accuracy: 87.60 Test accuracy: 88.62
Epoch: 9 Train loss: 0.4306 Test loss: 0.4009 Train accuracy: 88.34 Test accuracy: 89.17
Epoch: 10 Train loss: 0.4056 Test loss: 0.3788 Train accuracy: 88.82 Test accuracy: 89.68
Epoch: 11 Train loss: 0.3855 Test loss: 0.3602 Train accuracy: 89.27 Test accuracy: 90.08
Epoch: 12 Train loss: 0.3686 Test loss: 0.3444 Train accuracy: 89.65 Test accuracy: 90.45
Epoch: 13 Train loss: 0.3540 Test loss: 0.3310 Train accuracy: 89.95 Test accuracy: 90.75
Epoch: 14 Train loss: 0.3407 Test loss: 0.3183 Train accuracy: 90.29 Test accuracy: 91.00
```

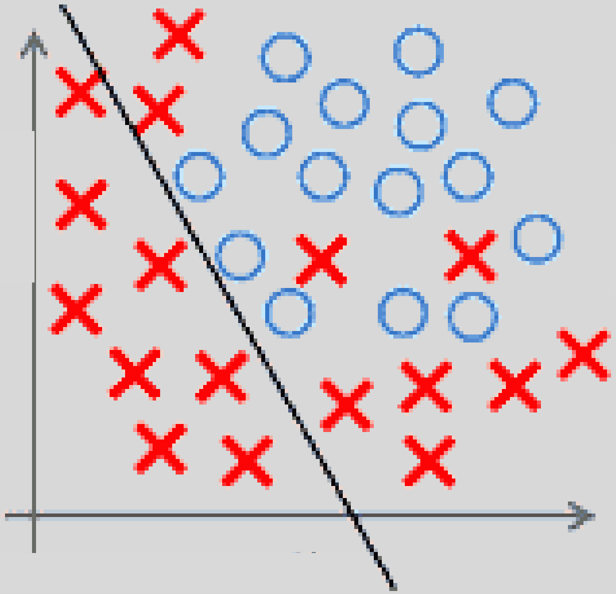
Over-fitting



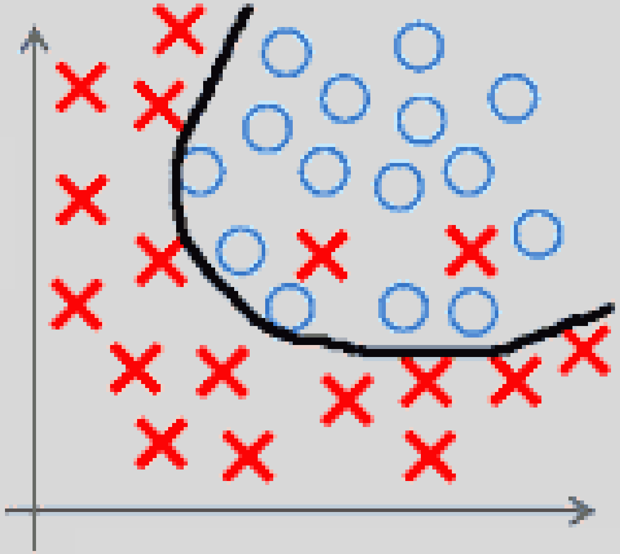
Because train/test datasets are different!

Over-fitting

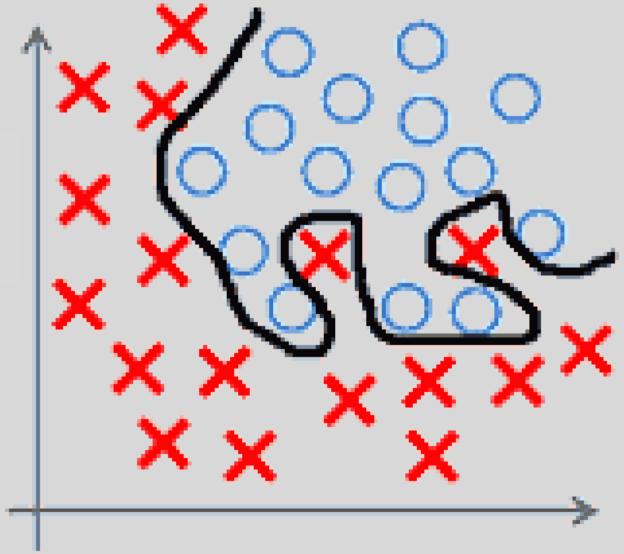
Over-fitting



<Under fit>



<Normal fit>



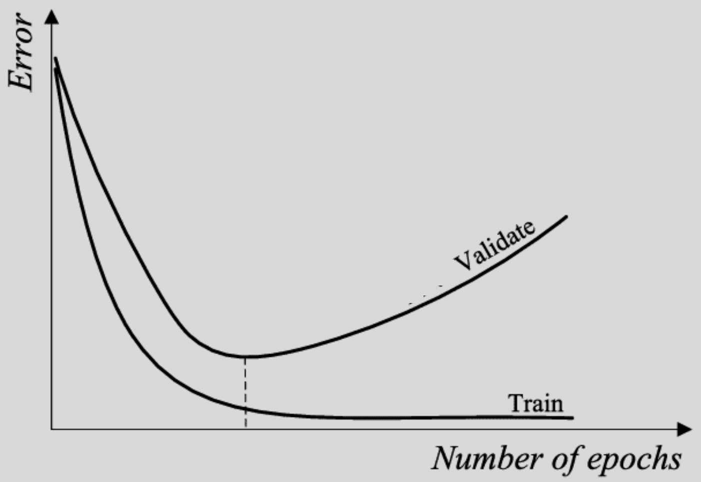
<Over fit>

Over-fitting

There are many ways to prevent Overfitting.

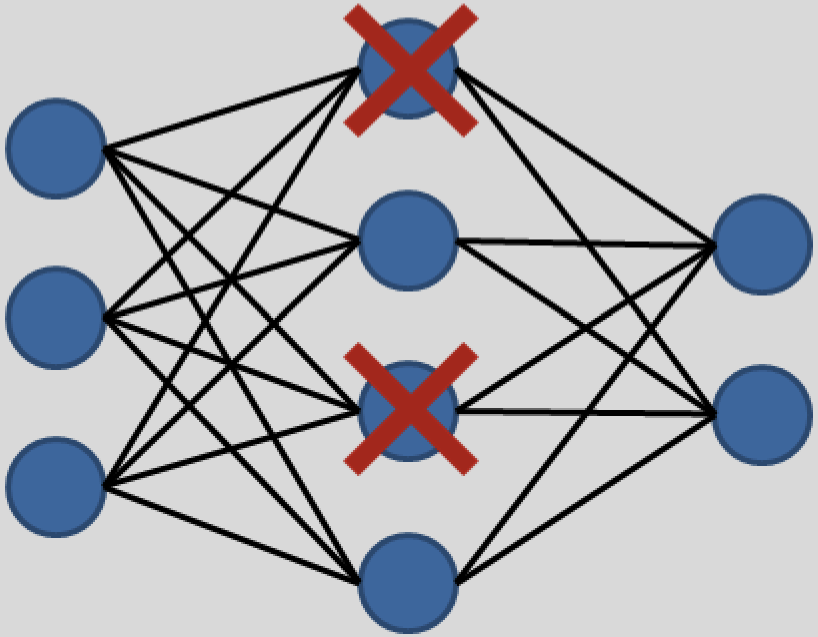
1. **One naïve method is increasing the size of your data.**
 → Make the network overfit to million-scale data.

2. **Having validation dataset.**
 → Find a trained model that operates well on validation dataset.



Dropout

Randomize your network so that it cannot easily overfit to train data.



```
torch.nn.Dropout(p=0.5)
```

p is the parameter for random dropout probability.

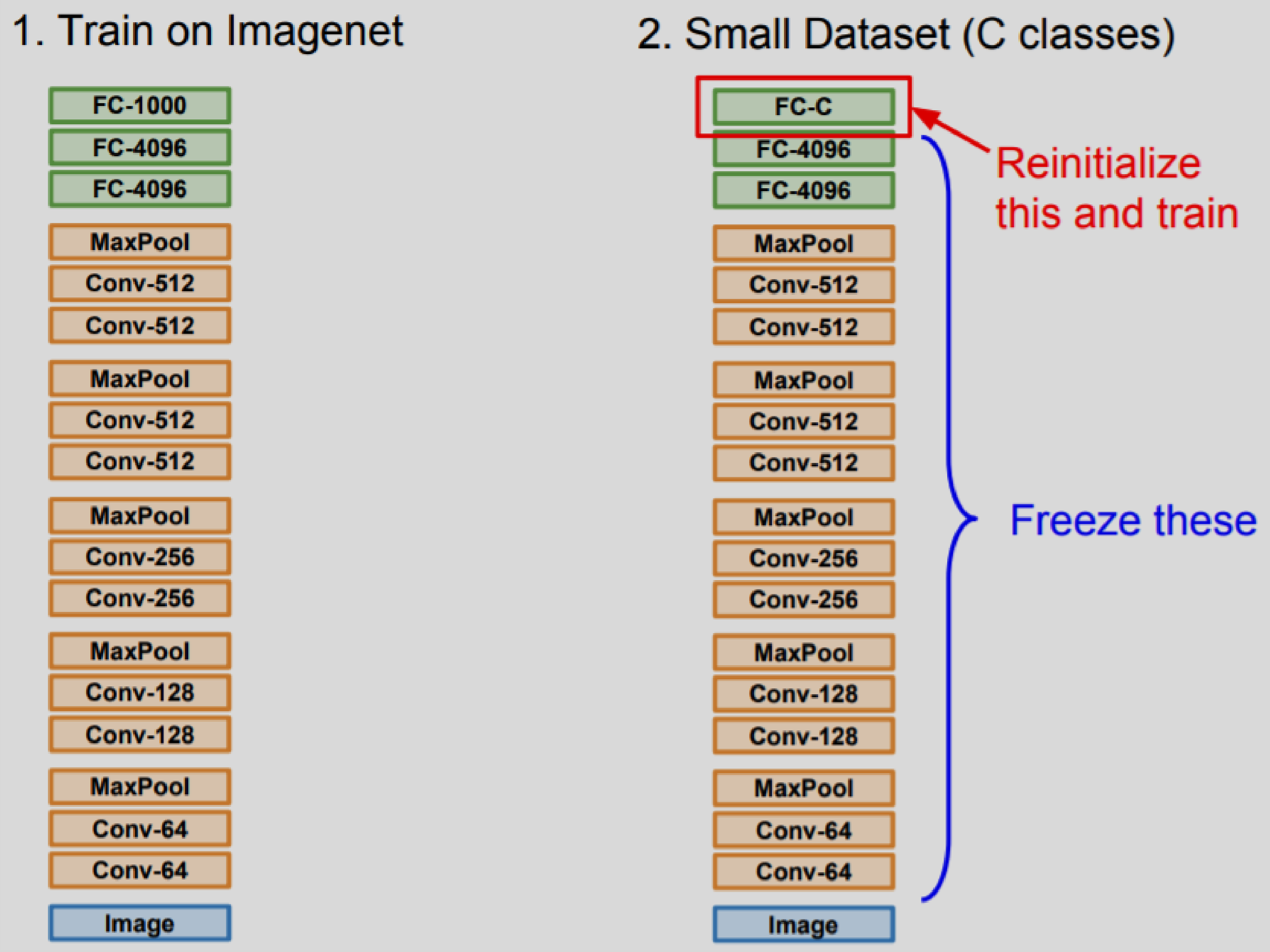
If p=0.5, half will randomly dropped when model is in the ‘train’ mode.

In the ‘eval’ mode, it is not used.

Data augmentation

```
import PIL
import numpy as np
import torch
import torchvision
import torchvision.datasets as datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
import cv2

transforms = torchvision.transforms.Compose([
    torchvision.transforms.Resize((224,224)),
    torchvision.transforms.ColorJitter(hue=.05, saturation=.05),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.RandomRotation(20, resample=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor()
])

for i in range(5):
  train_dataset = datasets.MNIST(root = 'mnist_data', train=True, transform=transforms, download=True)
  train_loader = DataLoader(dataset=train_dataset, batch_size=1, shuffle=False)
  for x, y in train_loader:
    break
  R = np.stack((x[0,0]*255.,x[0,0]*255.,x[0,0]*255.), axis=2)
  cv2_imshow(R)
```

Transfer learning



Transfer learning

https://pytorch.org/docs/stable/model_zoo.html

- AlexNet
- VGG
- ResNet
- SqueezeNet
- DenseNet
- Inception v3
- GoogLeNet
- ShuffleNet v2
- MobileNet v2
- ResNeXt
- Wide ResNet
- MNASNet

You can construct a model with random weights by calling its constructor:

```
import torchvision.models as models
resnet18 = models.resnet18()
alexnet = models.alexnet()
vgg16 = models.vgg16()
squeezenet = models.squeezenet1_0()
densenet = models.densenet161()
inception = models.inception_v3()
googlenet = models.googlenet()
shufflenet = models.shufflenet_v2_x1_0()
mobilenet = models.mobilenet_v2()
resnext50_32x4d = models.resnext50_32x4d()
wide_resnet50_2 = models.wide_resnet50_2()
mnasnet = models.mnasnet1_0()
```

Transfer learning

```python
from torchvision import models

net = models.alexnet(pretrained=True)

print(net)
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Transfer learning

```python
from torchvision import models

net = models.alexnet(pretrained=True)

for p in net.parameters():
  p.requires_grad = False

net.classifier[6] = torch.nn.Linear(in_features=4096, out_features=10,
 bias=True)

print(net)
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Deep learning model save/load

```
torch.save(model.state_dict(), PATH)
```

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

VGG Net

1. VGGNet tried to investigate the relationship between the depth of the network and the overall accuracy.

2. Consisted with only 3x3 conv layer, max pooling and fully connected layers.

3. Experimented with 11-layer-model to 19-layer models.

VGG Net

```
class VGG16(nn.Module):

    def __init__(self, num_classes):
        super(VGG16, self).__init__()

        self.block_1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )

        self.block_2 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )

        self.block_3 = nn.Sequential(
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )
```

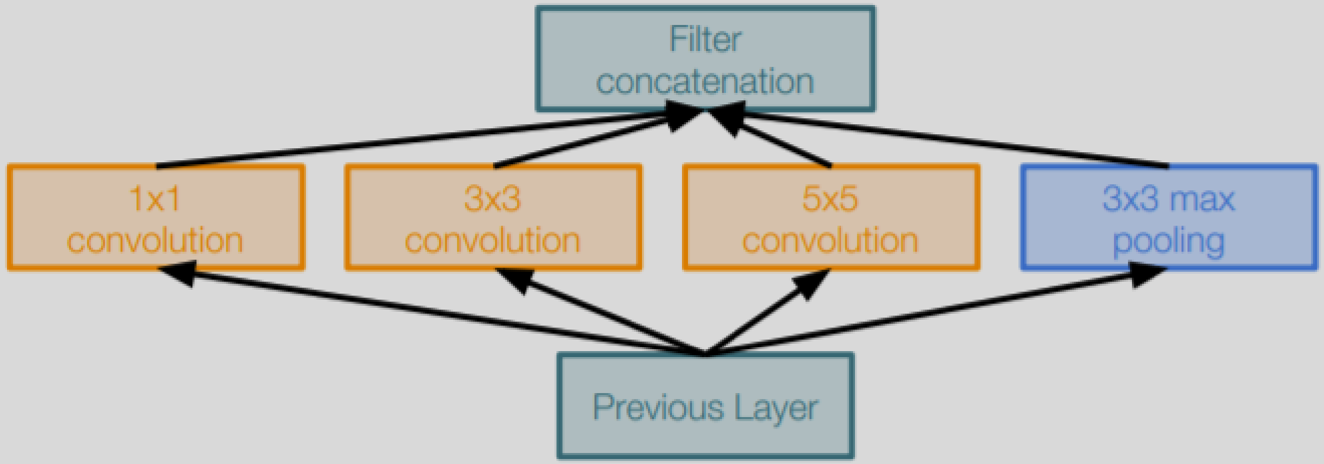
VGG Net

```
class VGG16(nn.Module):

    def __init__(self, num_classes):
        super(VGG16, self).__init__()

        self.block_4 = nn.Sequential(
            nn.Conv2d(in_channels=256, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )

        self.block_5 = nn.Sequential(
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=(3, 3), stride=(1, 1), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
        )
```

VGG Net

```
class VGG16(nn.Module):

    def __init__(self, num_classes):
        super(VGG16, self).__init__()

        self.classifier = nn.Sequential(
            nn.Linear(512, 4096),
            nn.ReLU(True),
            nn.Dropout(p=0.65),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(p=0.65),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):

        x = self.block_1(x)
        x = self.block_2(x)
        x = self.block_3(x)
        x = self.block_4(x)
        x = self.block_5(x)
        x = x.view(x.size(0), -1)
        logits = self.classifier(x)
        probas = F.softmax(logits, dim=1)
        return probas
```

Google LeNet



Naïve "Inception" module:

Apply parallel filter operations on the input from previous layer:
Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
Pooling operation (3x3)
Concatenate all filter outputs together depth-wise

- 22 layers , 9 inception models
- Efficient "Inception" module
- No FC layers
-12x less parameters than AlexNet
- ILSVRC'14 classification winner (6.7% top 5 error)

Google LeNet

```
class inception_module(nn.Module):

    def __init__(self,in_dim,out_dim_1,mid_dim_3,out_dim_3,mid_dim_5,out_dim_5,pool):
        super(inception_module,self).__init__()

        self.conv_1 = conv_1(in_dim,out_dim_1)
        self.conv_1_3 = conv_1_3(in_dim,mid_dim_3,out_dim_3)
        self.conv_1_5 = conv_1_5(in_dim,mid_dim_5,out_dim_5)
        self.max_3_1 = max_3_1(in_dim,pool)

    def forward(self,x):
        out_1 = self.conv_1(x)
        out_2 = self.conv_1_3(x)
        out_3 = self.conv_1_5(x)
        out_4 = self.max_3_1(x)
        output = torch.cat([out_1,out_2,out_3,out_4],1)

        return output
```

Google LeNet

```python
class GoogLeNet(nn.Module):
    def __init__(self, base_dim, num_classes=2):
        super(GoogLeNet, self).__init__()
        self.layer_1 = nn.Sequential(
            nn.Conv2d(3,base_dim,7,2,3),
            nn.MaxPool2d(3,2,1),
            nn.Conv2d(base_dim,base_dim*3,3,1,1),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_2 = nn.Sequential(
            inception_module(base_dim*3,64,96,128,16,32,32),
            inception_module(base_dim*4,128,128,192,32,96,64),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_3 = nn.Sequential(
            inception_module(480,192,96,208,16,48,64),
            inception_module(512,160,112,224,24,64,64),
            inception_module(512,128,128,256,24,64,64),
            inception_module(512,112,144,288,32,64,64),
            inception_module(528,256,160,320,32,128,128),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_4 = nn.Sequential(
            inception_module(832,256,160,320,32,128,128),
            inception_module(832,384,192,384,48,128,128),
            nn.AvgPool2d(7,1),
        )
        self.layer_5 = nn.Dropout2d(0.4)
        self.fc_layer = nn.Linear(1024,1000)
```

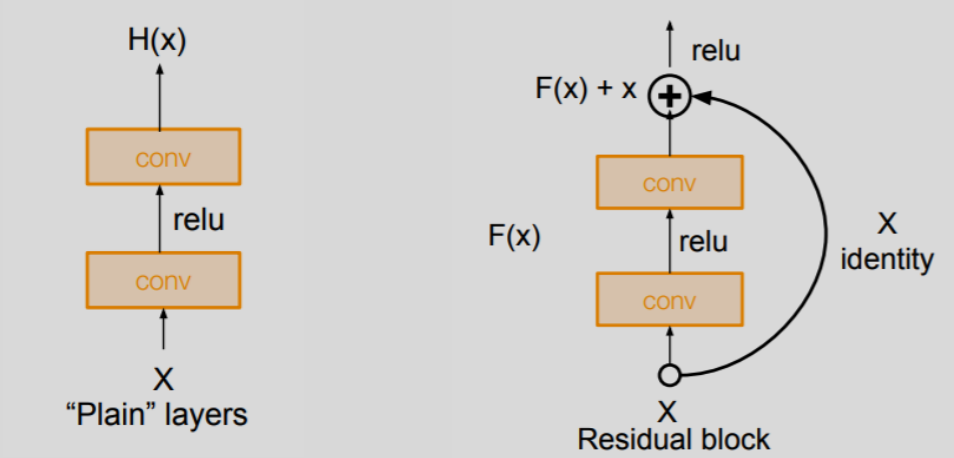
```python
def forward(self, x):
        out = self.layer_1(x)
        out = self.layer_2(out)
        out = self.layer_3(out)
        out = self.layer_4(out)
        out = self.layer_5(out)
        out = out.view(batch_size,-1)
        out = self.fc_layer(out)
        return out
```

ResNet

20 layers vs 56 layers, training error and test error:

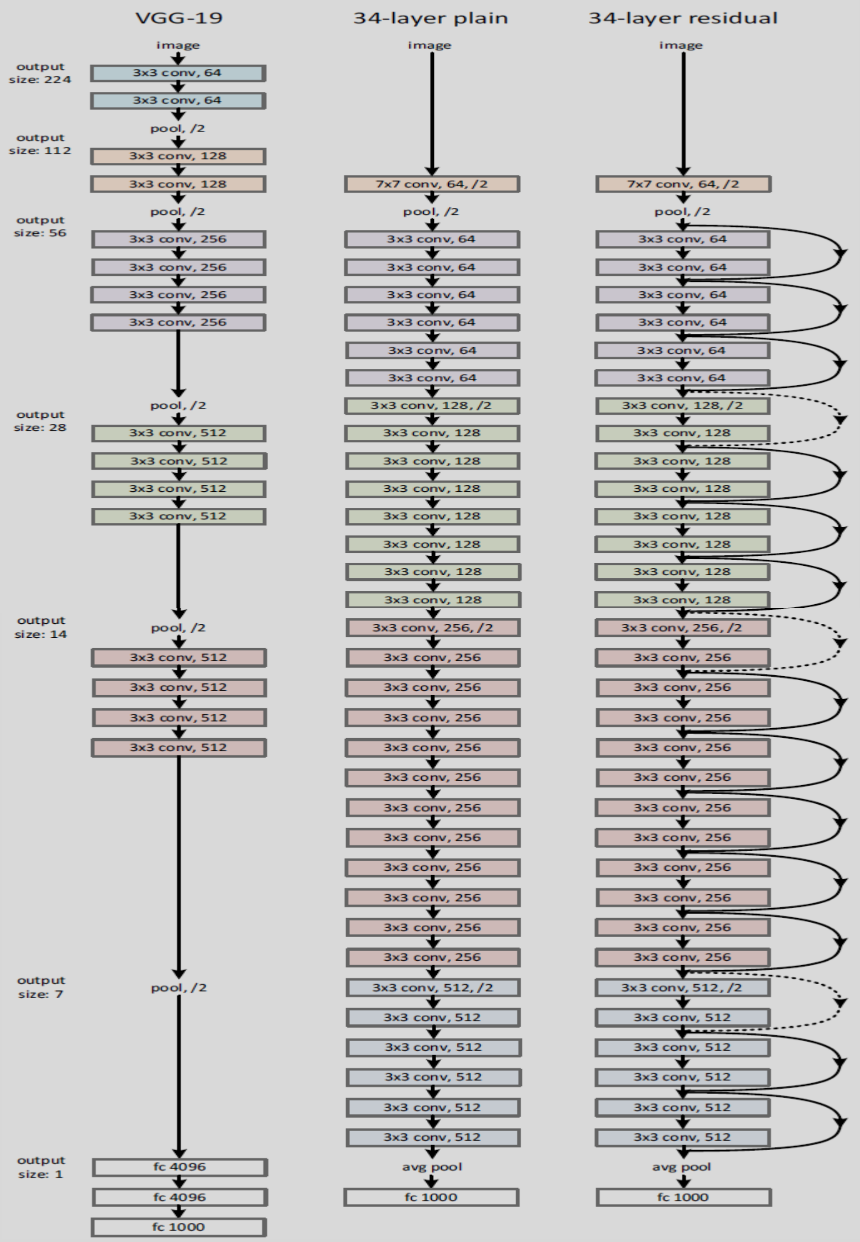
ResNet

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



Instead of learning H(x) directly, we ask what do we need to add/subtract in order to get H(x) ?
$H(x) = F(x) + x$

ResNet



Getting deeper without getting less accuracy.

ResNet

```
def conv_block_1(in_dim,out_dim,act_fn):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=1, stride=1),
        act_fn,
    )
    return model

def conv_block_1_stride_2(in_dim,out_dim,act_fn):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=1, stride=2),
        act_fn,
    )
    return model

def conv_block_1_n(in_dim,out_dim):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=1, stride=1),
    )
    return model

def conv_block_1_stride_2_n(in_dim,out_dim):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=1, stride=2),
    )
    return model

def conv_block_3(in_dim,out_dim,act_fn):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim, kernel_size=3, stride=1, padding=1),
        act_fn,
    )
    return model
```

ResNet

```
class BottleNeck(nn.Module):

    def __init__(self,in_dim,mid_dim,out_dim,act_fn):
        super(BottleNeck,self).__init__()
        self.layer = nn.Sequential(
            conv_block_1(in_dim,mid_dim,act_fn),
            conv_block_3(mid_dim,mid_dim,act_fn),
            conv_block_1_n(mid_dim,out_dim),
        )
        self.downsample = nn.Conv2d(in_dim,out_dim,1,1)

    def forward(self,x):
        downsample = self.downsample(x)
        out = self.layer(x)
        out = out + downsample

        return out


class BottleNeck_no_down(nn.Module):

    def __init__(self,in_dim,mid_dim,out_dim,act_fn):
        super(BottleNeck_no_down,self).__init__()
        self.layer = nn.Sequential(
            conv_block_1(in_dim,mid_dim,act_fn),
            conv_block_3(mid_dim,mid_dim,act_fn),
            conv_block_1_n(mid_dim,out_dim),
        )

    def forward(self,x):
        out = self.layer(x)
        out = out + x

        return out
```

```
class BottleNeck_stride(nn.Module):

    def __init__(self,in_dim,mid_dim,out_dim,act_fn):
        super(BottleNeck_stride,self).__init__()
        self.layer = nn.Sequential(
            conv_block_1_stride_2(in_dim,mid_dim,act_fn),
            conv_block_3(mid_dim,mid_dim,act_fn),
            conv_block_1_n(mid_dim,out_dim),
        )
        self.downsample = nn.Conv2d(in_dim,out_dim,1,2)

    def forward(self,x):
        downsample = self.downsample(x)
        out = self.layer(x)
        out = out + downsample

        return out
```

ResNet

```
class ResNet(nn.Module):

    def __init__(self, base_dim, num_classes=2):
        super(ResNet, self).__init__()
        self.act_fn = nn.ReLU()
        self.layer_1 = nn.Sequential(
            nn.Conv2d(3,base_dim,7,2,3),
            nn.ReLU(),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_2 = nn.Sequential(
            BottleNeck(base_dim,base_dim,base_dim*4,self.act_fn),
            BottleNeck_no_down(base_dim*4,base_dim,base_dim*4,self.act_fn),
            BottleNeck_stride(base_dim*4,base_dim,base_dim*4,self.act_fn),
        )
        self.layer_3 = nn.Sequential(
            BottleNeck(base_dim*4,base_dim*2,base_dim*8,self.act_fn),
            BottleNeck_no_down(base_dim*8,base_dim*2,base_dim*8,self.act_fn),
            BottleNeck_no_down(base_dim*8,base_dim*2,base_dim*8,self.act_fn),
            BottleNeck_stride(base_dim*8,base_dim*2,base_dim*8,self.act_fn),
        )
        self.layer_4 = nn.Sequential(
            BottleNeck(base_dim*8,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_no_down(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_no_down(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_no_down(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_no_down(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
            BottleNeck_stride(base_dim*16,base_dim*4,base_dim*16,self.act_fn),
        )
        self.layer_5 = nn.Sequential(
            BottleNeck(base_dim*16,base_dim*8,base_dim*32,nn.ReLU()),
            BottleNeck_no_down(base_dim*32,base_dim*8,base_dim*32,self.act_fn),
            BottleNeck(base_dim*32,base_dim*8,base_dim*32,self.act_fn),
        )
        self.avgpool = nn.AvgPool2d(7,1)
        self.fc_layer = nn.Linear(base_dim*32,num_classes)
```

```
    def forward(self, x):
        out = self.layer_1(x)
        out = self.layer_2(out)
        out = self.layer_3(out)
        out = self.layer_4(out)
        out = self.layer_5(out)
        out = self.avgpool(out)
        out = out.view(batch_size,-1)
        out = self.fc_layer(out)

        return out
```

Few notice

No mid-term exam.

No classes during the mid-term exam period.

We will resume the class from 10/24.

During the mid-term period, PA2 will be announced (4 weeks duration).

Thank you!

UNIST

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

2007