Chapter 9

Classical Architectures of CNNs

In the previous chapter, we introduced the basics of CNNs and learned how to implement and train a simple CNN (e.g. LeNet-5) for image classification using PyTorch framework. In this chapter, we will present a few important CNN architectures. Each of these architectures has served as a milestone in deep learning history and was the base model upon which many research projects and deployed systems were built. These models include AlexNet, VGG, ResNet, GoogLeNet and NiN (Network in Network).

In general, a deep neural network is formed simply by stacking many layers (e.g., convolution layers, max/average pooling layers, full-connected layers). However, the performance can vary dramatically with different architectures and hyperparameter choices. The models presented in this chapter are the results of some mathematical insights, human intuitions, and efforts of trial-error. Through this chapter, readers can get some sense of how the research communities made the deep learning neural networks outperform human beings. A practitioner can adopt one of these powerful CNNs for a particular application or develop a new CNN architecture inspired by these classical architectures.

This chapter covers:

- Some popular datasets for computer vision.
- AlexNet
- VGG networks
- Network-in-network (NiN)
- GoogLeNet
- ResNets
- Fine tune pretrained models using PyTorch

9.1 Datasets

The advances of deep learning architectures have been driven by two key factors since 2010: large amounts of data and powerful computing hardware. Large amounts of data are required to train complex models. However, given the limited source of data and limited storage capacity of computers in 1990s, most computer vision research projects relied on small or middle-sized datasets. Since the internet social medium and mobile electronic devices (e.g. cell phone) entered into human's daily life, huge amounts of data in various formats have been generated. There are many datasets publicly available for computer vision projects. While a deep learning model in

specific application handles a particular type of data, it is usually a common practice to train and test the developed model using a standard dataset for an experimental validation purpose.

Before we dive into classical CNN architectures, we introduce some popular datasets, which are considered as typical benchmark datasets for testing a model. These datasets include MNIST, Fashion-MNIST, CIFAR10, ImageNet, COCO datasets, and Cityscapes.

MNIST

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is widely used for training and testing in the field of machine learning. It was created by re-mixing the samples from NIST's original datasets in 1998. The MNIST database contains 60,000 training images and 10,000 testing images. The black and white images were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels.

Extended MNIST (EMNIST) is a newer dataset developed and released by NIST to be the successor to MNIST. While MNIST included images only of handwritten digits, EMNIST includes all the images from NIST Special Database 19, which is a large database of handwritten uppercase and lower-case letters as well as digits. The images in EMNIST were converted into the same 28x28 pixel format, by the same process used for the MNIST images. Accordingly, tools which work with the older, smaller, MNIST dataset will likely work with EMNIST without modification.

PyTorch provides the built-in class for us to download many popular datasets. An example of downloading MNIST train dataset by PyTorch is given below. As the result of the following code, sample images are shown in Fig.9.1.

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

batch_size = 128
mnist_trainset = torchvision.datasets.MNIST(root='C:/Users/weido/ch11/data',
     train=False, download=True, transform=transforms.ToTensor())
Mnist = torch.utils.data.DataLoader(mnist_trainset, batch_size=batch_size,
                                    shuffle=True, num_workers=1)
def imshow_grey(img):
    plt.imshow(img.permute(1,2,0),cmap="Greys")
    plt.show()

# get some random training images
dataiter= iter(Mnist) # return an iterator assigned to dataiter
images, labels = dataiter.next() # get the current iteration on dataiter: [batc
h_size, C, H, W]

# show images in one batch, "255-images"
imshow_grey(torchvision.utils.make_grid(255-images, nrow=16, normalize=True))
```

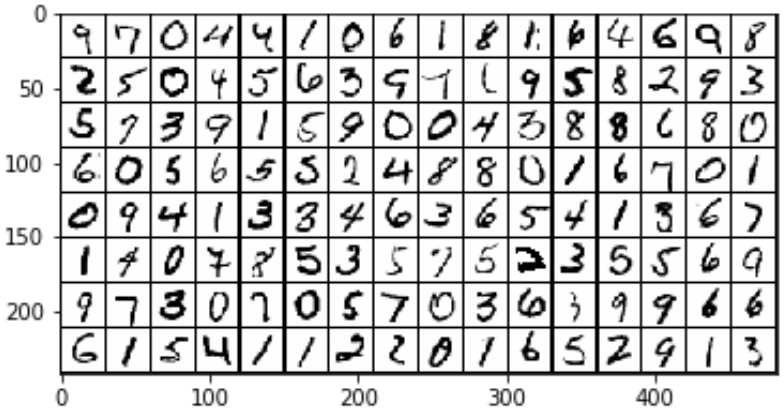


Fig.9.1 Examples of MNIST images

Fashion-MNIST

The Fashion-MNIST dataset is proposed as a more challenging replacement dataset for the MNIST dataset. It is a dataset comprised of 60,000 grayscale images with the same format as MNIST, but with different image contents. In Fashion-MNIST dataset, one image corresponds to one of ten fashion categories. The mapping of the digit label to the category is listed as 0: T-shirt/top, 1: Trouser, 2: Pullover, 3: Dress, 4: Coat, 5: Sandal, 6: Shirt, 7: Sneaker, 8: Bag, 9: Ankle boot. The dataset can be downloaded in the same way as MNIST, but by `torchvision.datasets.FashionMNIST()`. Sample images of Fashion-MNIST are shown in Fig.9.2.

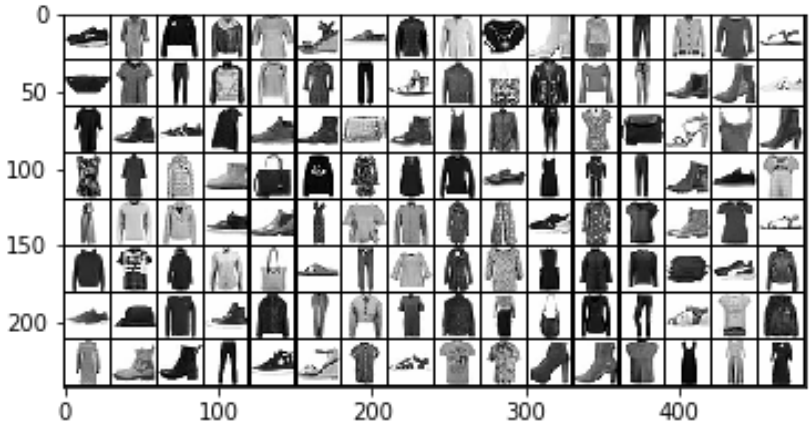


Fig.9.2 Examples of Fashion-MNIST images (generated in the same way as Fig.9.1)

CIFAR10

CIFAR (Canadian Institute For Advanced Research)-10 is a popular computer-vision dataset collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. This dataset is used for object recognition, and it consists of 60,000 32×32 color images in 10 classes, with 6,000 images per class. It is divided into five training batches and one test batch, each batch with 10,000 images. The test batch contains exactly 1000 randomly selected images from each class. The training batches

contain the remaining images in random order, totally with exact 5000 images from each class, but some training batches may contain more images from one class than another. Samples of CIFAR10 is shown in Fig.9.3. The dataset can be downloaded in the similar way of MNIST, except by `torchvision.datasets.CIFAR10()`.

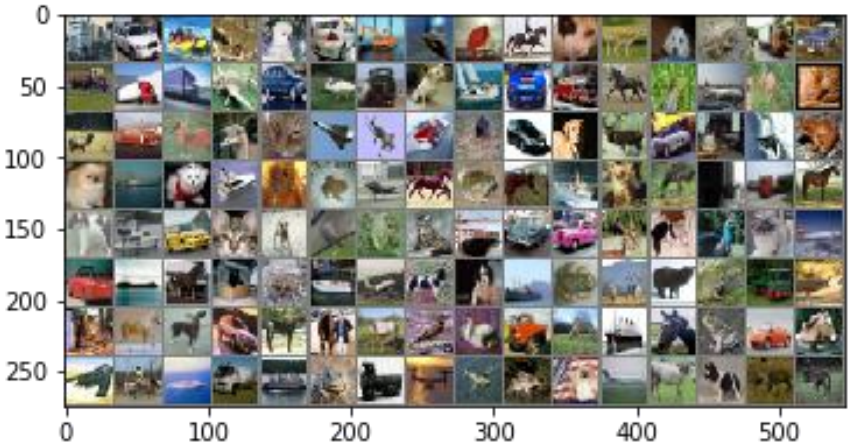


Fig.9.3 Examples of CIFAR10 images (generated in the same way as Fig.9.1)

CIFAR-100

This dataset is just like the CIFAR-10, except that it has 100 classes containing 600 images per class. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 super classes. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the super class to which it belongs).

Here is the list of classes in the CIFAR-100:

Superclass	**Classes**
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel

trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

The dataset can be downloaded by `torchvision.datasets.CIFAR100()`.

ImageNet (https://image-net.org/index.php)

The ImageNet dataset contains 14,197,122 annotated images according to the WordNet hierarchy. The most highly used subset of ImageNet is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012-2017 image classification and localization dataset. This dataset spans 1000 object classes and contains 1,281,167 training images, 50,000 validation images and 100,000 test images. Since 2010 the dataset is used in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a benchmark in image classification and object detection. The publicly released dataset contains a set of manually annotated training images. A set of test images is also released, with the manual annotations withheld. ILSVRC annotations fall into one of two categories: (1) image-level annotation of a binary label for the presence or absence of an object class in the image, and (2) object-level annotation of a tight bounding box and class label around an object instance in the image. Sample images are shown in Fig.9.4.

Fig.9.4 Samples of ImageNet [source: https://cs.stanford.edu/people/karpathy/cnnembed/]

COCO Dataset (https://cocodataset.org/#home)

The COCO dataset stands for Common Objects in Context, and is designed to represent a vast array of objects that we regularly encounter in everyday life. It provides large-scale datasets for object detection, segmentation, keypoint detection, and image captioning. It contains 80 object categories with over 1.5 million object instances for context recognition, object detection, and segmentation.

Cityscapes (https://www.cityscapes-dataset.com/)

Cityscapes is an open-sourced large-scale dataset for computer vision projects, which contains a diverse set of stereo video sequences recorded in street scenes from 50 different cities. It includes high-quality pixel-level annotations of 5,000 frames in addition to a larger set of 20,000 weakly annotated frames. This dataset is mainly used for training deep neural networks and assessing the performance of vision algorithms for major tasks of semantic urban scene understanding. Some samples are shown in Fig.9.5.

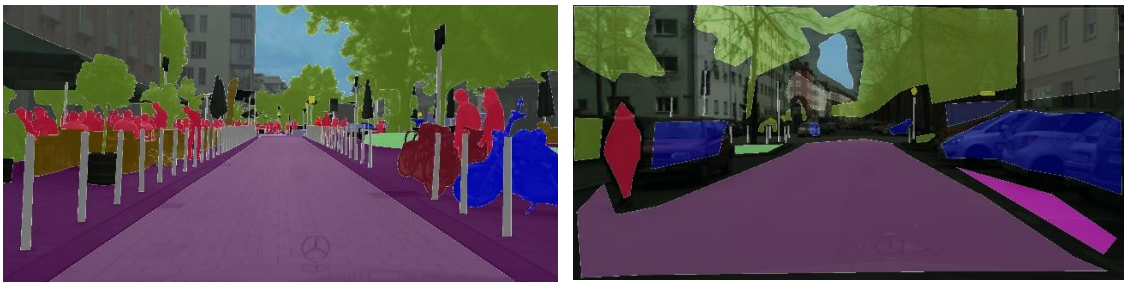


Fig.9.5 Examples of frames with fine annotations (left) and coarser annotations (right)
(Source: https://www.cityscapes-dataset.com/examples/#fine-annotations)

9.2 AlexNet

AlexNet won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 by achieving top-1 and top-5 error rates of 36.7% and 15.3% which is considerably better than its previous state-of-the-art. AlexNet is considered as one of breakthroughs in deep learning.

Similar to LeNet-5, AlexNet stacks a few convolution layers, pool layers and fully-connected layers. The significant improvements in AlexNet lie in 1) the larger input image (color 224x224 pixels); 2) more convolution layers; 3) ReLU activation in all layers; 4) dropout regularization; and 5) two GPUs for computation. For comparison, Fig.9.6 shows the architectures of both LeNet-5 and AlexNet. Note that the original architecture of AlexNet is slightly different than the version presented here. In the original AlexNet, each convolution layer was divided into two parts, each of which fits one GPU. The data in two parts merged only at the outputs of some layers. Now it is not necessary to do so since the current GPU can easily handle the entire convolution layer.

As shown in Fig.9.6, AlexNet consists of 5 convolution layers, 3 Max-pooling layers and 3 fully connected layers. The default stride and zero-padding are 1 and 0, respectively, unless otherwise specified as s and p. To reduce overfitting, dropout regularization is applied to the first two FC layers during the training process. The implementation of AlexNet in PyTorch is shown below.

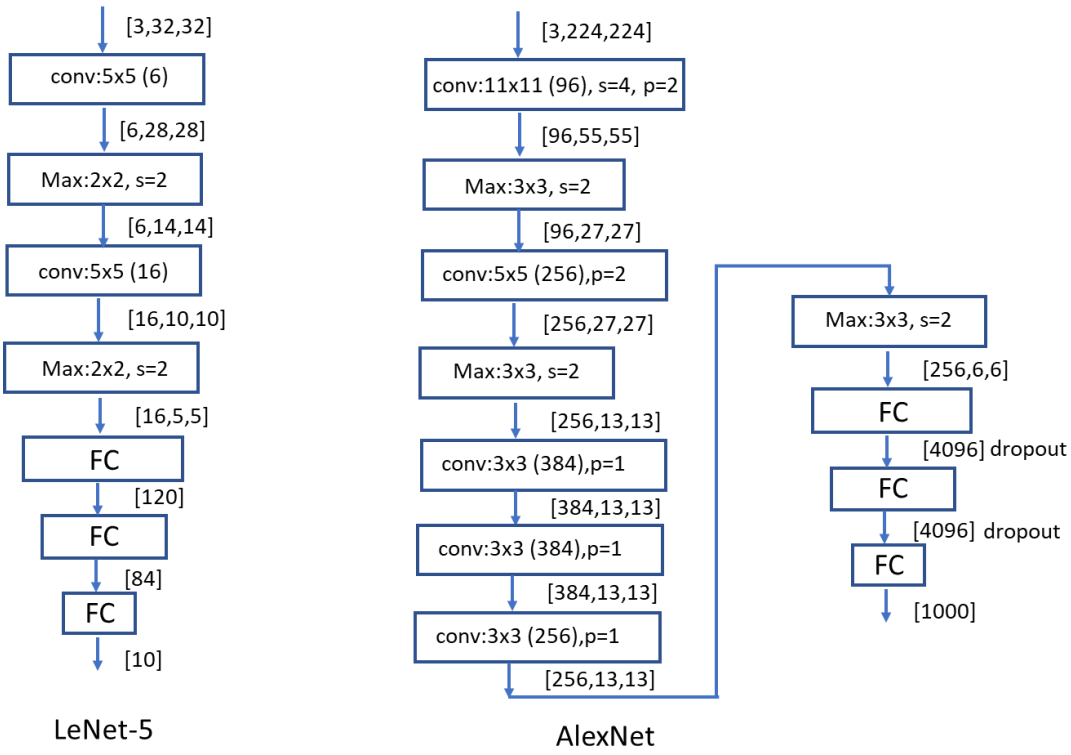


Fig.9.6 Comparison between LeNet-5 (left) and AlexNet (right)

Define AlexNet in PyTorch:

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

class AlexNet(nn.Module):
# define self functions for all layers with learning parameters
    def __init__(self):
        super(AlexNet, self).__init__()

        # define function sefl.conv1 for Conv layer1
        self.conv1 = nn.Conv2d(3,96,11, stride=4, padding=2)
        self.conv2 = nn.Conv2d(96,256,5, padding=2)
        self.conv3 = nn.Conv2d(256,384,3, padding=1)
        self.conv4 = nn.Conv2d(384,384,3, padding=1)
        self.conv5 = nn.Conv2d(384,256,3, padding=1)
        self.fc1 = nn.Linear(256*6*6, 4096)
        self.fc2 = nn.Linear(4096,4096)
        self.fc3 = nn.Linear(4096,1000)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x,3,2)

        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x,3,2)

        x = self.conv3(x)
        x = F.relu(x)
```

```
        x = self.conv4(x)
        x = F.relu(x)

        x = self.conv5(x)
        x = F.relu(x)
        x = F.max_pool2d(x,3,2)

        x = x.view(-1, 256*6*6)
        x = self.fc1(x)
        x = F.relu(x)
        x = F.dropout(x,0.5)

        x = self.fc2(x)
        x = F.relu(x)
        x = F.dropout(x,0.5)

        x = self.fc3(x)
        return x

net = AlexNet()
```

Run a forward pass:

```
X = torch.randn(4, 3, 224, 224)
out=net(X)
```

```
out.shape
torch.Size([4, 1000])
```

9.3 VGG: Networks using Blocks

The neural network architecture in this section is named after the Visual Geometry Group (VGG) at Oxford University, where the researchers proposed this very deep convolution network architecture for large-scale image recognition.

Unlike LeNet-5 and AlexNet CNNs, VGG networks are created in terms of ***convolutional blocks***, instead of convolutional layers. A convolutional block consists of a few convolution layers. The VGG network can be partitioned into two parts: the first part consisting of a sequence of convolutional blocks, with the blocks separated by maximum pooling layers; and the second part consisting of a few fully connected layers (like LeNet-5 or AlexNet). According to the different depths of neural networks, there are five VGG architectures proposed by the original paper: VGG11, VGG13, VGG16-A, VGG16-B, and VGG19. The numbers in the names indicate the total number of weight layers (i.e., convolution layers and FC layers). For example, VGG11 has 8 convolution layers and 3 FC layers. Different VGG architectures are summarized in Table 9.1.

A VGG architecture consists of 5 convolution blocks with each followed by a max pool layer, and 3 fully connected layers. All layers (convolution layers and FC layers) use ReLU activation, except the last FC layer using softmax for classification. The convolution blocks at different locations may have different depth (one layer, two layers or three layers). All convolution layers have the same filter size 3x3 with padding =1, stride=1 so that the output have the same feature size as the input, except one layer in VGG16-A using filter size 1x1 with padding=0 and stride=1. All the max pool layers use the window size 2x2 with stride =2. The number of channels in each layer is specified

in the parenthesis. Using VGG16-B as an example, Fig.9.7 illustrates the shape of feature map after each layer.

Table 9.1 VGG architectures

VGG11	**VGG13**	**VGG16-A**	**VGG16-B**	**VGG19**
Input image [3,224,224]				
Conv3 (64)	Conv3 (64) **Conv3 (64)**	Conv3 (64) Conv3 (64)	Conv3 (64) Conv3 (64)	Conv3 (64) Conv3 (64)
Max pool (2x2, stride=2) → [64, 112, 112]				
Conv3 (128)	Conv3 (128) **Conv3 (128)**	Conv3 (128) Conv3 (128)	Conv3 (128) Conv3 (128)	Conv3 (128) Conv3 (128)
Max pool (2x2, stride=2) → [128, 56, 56]				
Conv3 (256) Conv3 (256)	Conv3 (256) Conv3 (256)	Conv3 (256) Conv3 (256) **Conv1 (256)**	Conv3 (256) Conv3 (256) **Conv3 (256)**	Conv3 (256) Conv3 (256) Conv3 (256) **Conv3 (256)**
Max pool (2x2, stride=2) → [256, 28, 28]				
Conv3 (512) Conv3 (512)	Conv3 (512) Conv3 (512)	Conv3 (512) Conv3 (512) **Conv1 (512)**	Conv3 (512) Conv3 (512) **Conv3 (512)**	Conv3 (512) Conv3 (512) Conv3 (512) **Conv3 (512)**
Max pool (2x2, stride=2) → [512, 14, 14]				
Conv3 (512) Conv3 (512)	Conv3 (512) Conv3 (512)	Conv3 (512) Conv3 (512) **Conv1 (512)**	Conv3 (512) Conv3 (512) **Conv3 (512)**	Conv3 (512) Conv3 (512) Conv3 (512) Conv3 (512)
Max pool (2x2, stride=2) → [512, 7, 7]				
FC-4096 (with optional dropout)				
FC-4096 (with optional dropout)				
FC-1000				
Soft-max				

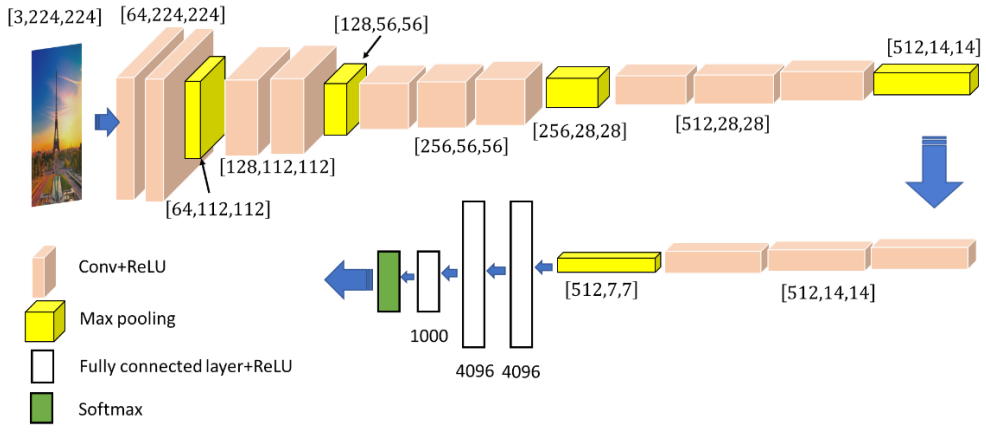


Fig.9.7 VGG16-B architecture

VGG achieved a top-5 error rate of 7.3% in the ILSVRC-2014 for classification and won the second place while the winner GoogLeNet achieved a 6.7% top-5 error rate. The implementation of VGG16 in PyTorch is shown below.

```
class VGG16(nn.Module):
    def __init__(self):
        super(VGG16, self).__init__()
        self.conv1_1 = nn.Conv2d(in_channels=3,out_channels=64,kernel_size=3,pa
dding=1)
        self.conv1_2 = nn.Conv2d(64, 64, 3, padding=1)

        self.conv2_1 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv2_2 = nn.Conv2d(128, 128, 3, padding=1)

        self.conv3_1 = nn.Conv2d(128, 256, 3, padding=1)
        self.conv3_2 = nn.Conv2d(256, 256, 3, padding=1)
        self.conv3_3 = nn.Conv2d(256, 256, 3, padding=1)

        self.conv4_1 = nn.Conv2d(256, 512, 3, padding=1)
        self.conv4_2 = nn.Conv2d(512,512, 3, padding=1)
        self.conv4_3 = nn.Conv2d(512, 512, 3, padding=1)

        self.conv5_1 = nn.Conv2d(512, 512, 3, padding=1)
        self.conv5_2 = nn.Conv2d(512, 512, 3, padding=1)
        self.conv5_3 = nn.Conv2d(512, 512, 3, padding=1)

        self.fc1 = nn.Linear(25088, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 1000)

    def forward(self, x):
        x = F.relu(self.conv1_1(x))
        x = F.relu(self.conv1_2(x))
        x = F.max_pool2d(x,2)
        #--------
        x = F.relu(self.conv2_1(x))
        x = F.relu(self.conv2_2(x))
        x = F.max_pool2d(x,2)
        #--------
        x = F.relu(self.conv3_1(x))
        x = F.relu(self.conv3_2(x))
        x = F.relu(self.conv3_3(x))
        x = F.max_pool2d(x,2)
        #---------
        x = F.relu(self.conv4_1(x))
        x = F.relu(self.conv4_2(x))
        x = F.relu(self.conv4_3(x))
        x = F.max_pool2d(x,2)
        #------
        x = F.relu(self.conv5_1(x))
        x = F.relu(self.conv5_2(x))
        x = F.relu(self.conv5_3(x))
        x = F.max_pool2d(x,2)
        #------
        x = x.reshape(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, 0.5) #dropout was included to combat overfitting
        x = F.relu(self.fc2(x))
        x = F.dropout(x, 0.5)
        x = self.fc3(x)
        return x
```

9.4 Network in Network (NiN)

The original NiN model was proposed shortly after AlexNet. As we know, the conventional convolutional layers use linear filters followed by a nonlinear activation function to scan the input. Instead, in an NiN model, micro neural networks are built to abstract the data within the receptive field. The micro neural network is instantiated with a multilayer perceptron. The multilayer perceptron is equivalent to a sequence of conventional convolution layer with filter size 1x1. Thus, the idea behind NiN is to apply a fully-connected layer at each pixel location. With enhanced local modeling via the micro network, we are able to utilize global average pooling over feature maps in the classification layer, which is easier to interpret and less prone to overfitting than traditional fully connected layers.

9.4.1 NiN blocks

In NiN model, the conventional CNN layers are replaced with NiN blocks, called mlpconv layers. The mlpconv layer is the key to understanding NiN models. Fig.9.8 shows the mlpconv layer, compared with conventional CNN layer.

In conventional convolution layer, a linear filter (e.g., 3x3) followed by a nonlinear activation function (e.g. ReLU) is applied to a receptive field of the input feature maps, and generates one pixel in the output feature maps. By scanning the receptive field across the entire input, the output feature maps can be obtained. In mlpconv layer, a fully connected micro neural network is applied to one pixel (cross channels) of the output feature map of the convolution layer, which is an abstract of a receptive field in the input. Note that the output feature maps of the mlpconv layer can be obtained by independently applying the same FC micro neural network (i.e., with the same weights) to each of pixels (cross channels) of the output feature map of the convolution layer. This cross channel micro neural network on the whole feature map is equivalent to a convolution layer with filter 1x1. Thus, the mlpconv layer is equivalent to a convolution layer followed by 1x1 convolution layers (two layers as shown). Note that all convolution layers include a ReLU activation.

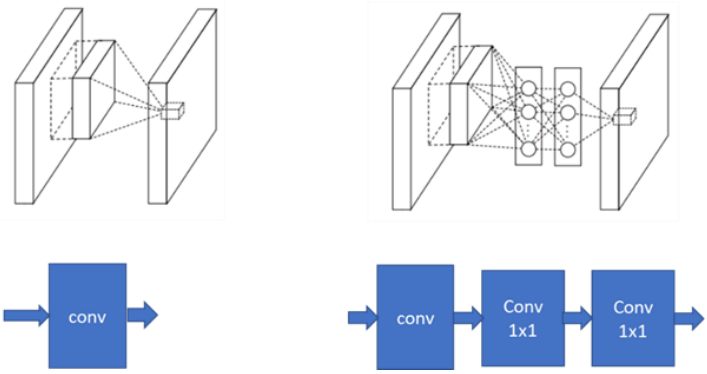


Fig.9.8 Convolution layer (left) and mlpconv layer (right)

9.4.2 Global average pooling

Conventional convolutional neural networks perform convolution to extract features in the lower layers of the network. For classification, the feature maps of the last convolutional layer are vectorized and fed into fully connected layers followed by a softmax logistic regression layer. In

NiN models, we use global average pooling to replace the traditional fully connected layers for classification. The idea is to generate one feature map for each corresponding category of the classification task in the last mlpconv layer. Instead of adding fully connected layers on top of the feature maps, we take the average of each feature map, and the resulting vector is fed directly into the softmax layer. One advantage of global average pooling over the fully connected layers is that it is more native to the convolution structure by enforcing correspondences between feature maps and categories. Thus, the feature maps can be easily interpreted as categories confidence maps. Another advantage is that there is no parameter to optimize in the global average pooling thus overfitting is avoided at this layer. Furthermore, global average pooling sums out the spatial information, thus it is more robust to spatial translations of the input.

9.4.3 Network-in-network architecture

The overall structure of NiN is a stack of mlpconv layers, on top of which lie the global average pooling. Down-sampling layers can be added in between the mlpconv layers. Fig.9.9 shows an NiN architecture with four mlpconv layers. Each mlpconv layer consists of three convolutional layers. The number of layers in both NiN and the micro networks is flexible and can be tuned for specific tasks.

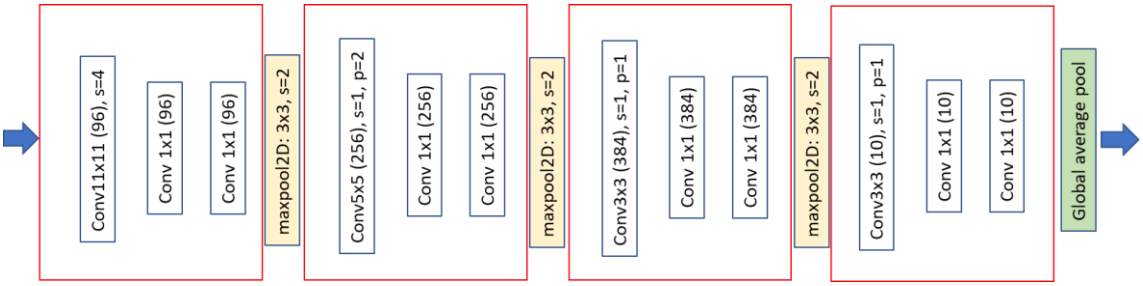


Fig.9.9 An example of NiN model

To make it concrete and specific, we develop an NiN model with AlexNet as a reference. The model consists of 4 NiN blocks (each followed by a max pool layer) and one global average pool layer, as shown in Fig.9. The number of channels and filter size in each layer are specified. The input to the model is a 3-channel image with a shape of [3,224, 224] while the output of the model is a 10-element vector for 10-category classification. The implementation of the model in PyTorch is shown below.

```
class NiN(nn.Module):
    def __init__(self, num_labels):
        super(NiN, self).__init__()
        self.net = nn.Sequential(
     #nin_block(self, in_channels, out_channels, kernel_size, stride, padding):
            self.nin_block(3, 96, 11, stride=4, padding=0),
            nn.Dropout(p=0.5),
            nn.MaxPool2d(kernel_size=3, stride=2),
            self.nin_block(96, 256, 5, stride=1, padding=2),
            nn.Dropout(p=0.5),
            nn.MaxPool2d(kernel_size=3, stride=2),
            self.nin_block(256, 384, 3, stride=1, padding=1),
            nn.Dropout(p=0.5),
            nn.MaxPool2d(kernel_size=3, stride=2),
            self.nin_block(384, num_labels, 3, stride=1, padding=1),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten()
```

```
        )
        self.init_weight()

    def forward(self,x):
        return self.net(x)

    def init_weight(self):
        for layer in self.net:
            if isinstance(layer, nn.Conv2d):
                nn.init.kaiming_normal(layer.weight,mode='fan_out',nonlinearity
='relu')
                nn.init.constant_(layer.bias, 0)

    def nin_block(self,in_channels,out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.Conv2d(in_channels,out_channels,kernel_size,stride=stride,paddin
g=padding),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels, kernel_size=1),
            nn.ReLU()
        )

    def test_output_shape(self):
        test_img = torch.rand(size=(1, 3, 224, 224), dtype=torch.float32)
        for layer in self.net:
            test_img = layer(test_img)
            print(layer.__class__.__name__, 'output shape: \t', test_img.shape)
```

We test the model on the forward pass as follows.

```
nin = NiN(num_labels=1000)
nin.test_output_shape()
```

```
Sequential output shape:     torch.Size([1, 96, 54, 54])
Dropout output shape:        torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Sequential output shape:     torch.Size([1, 256, 26, 26])
Dropout output shape:        torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Sequential output shape:     torch.Size([1, 384, 12, 12])
Dropout output shape:        torch.Size([1, 384, 12, 12])
MaxPool2d output shape:      torch.Size([1, 384, 5, 5])
Sequential output shape:     torch.Size([1, 1000, 5, 5])
AdaptiveAvgPool2d output shape:      torch.Size([1, 1000, 1, 1])
Flatten output shape:        torch.Size([1, 1000])
```

9.5 GoogLeNet

GoogLeNet was proposed in 2014 after VGG and NiN networks, and won ILSVRC 2014 competition with a top-5 error rate 6.7% for ImageNet classification.

9.5.1 Inception blocks

The most straightforward way of improving the performance of deep neural networks is by increasing their size in depth (i.e., the number of network layers) and width (i.e., the number of

units at each layer). However, in general this will lead to two issues: 1) the larger the network, the more prone it to overfitting especially if the training examples are limited; and 2) the dramatically increased use of computational resources or wasting the computational resources.

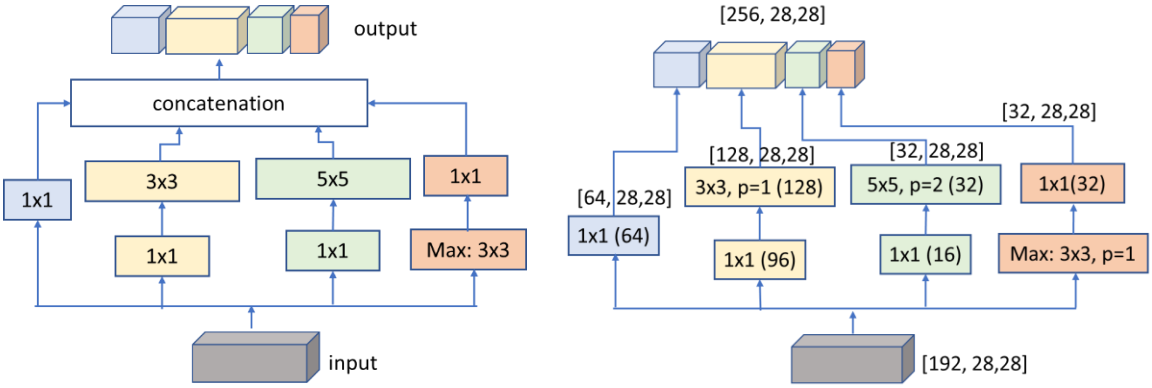


Fig.9.10 Inception block diagram (left) and example (right)

The basic convolution blocks in GoogLeNet, called inception blocks, are introduced to improve the learning efficiency as the network is becoming deeper. The architecture of a suggested inception block is shown in Fig.9.10. The inception block combines convolution filters with different sizes: 1x1, 3x3, 5x5, and 3x3 max pooling for efficiently extracting features and also use 1x1 filters to reduce computation. The incept block consists of four parallel paths. Each path corresponds to a type of filter (1x1, 3x3, 5x5, or max pool). However, before filters of 3x3 and 5x5, or after max pool 3x3, we add a 1x1 filter to reduce the network complexity (i.e., the number of channels). The outputs of four paths are concatenated as the output. Note that all convolution layers in the block use ReLU activation, and stride of 1, and that appropriate zero-paddings are needed to maintain the data size (H,W). One can assign different number of channels for each path. An example in Fig.9.10 (right) shows an instance of the inception block with the detailed information on the zero-paddings, channel numbers and data dimensions [channel, height, width] for all paths.

9.5.2 GoogLeNet architecture

The architecture of GoogLeNet used for ILSVRC 2014 competition can be described by Table 9.2. GoogLeNet uses 9 inception blocks and global average pooling to generate its estimates. Maximum pooling between inception blocks reduces the dimensionality. The network is 22-layer deep when counting only layers with parameters (or 27-layer if we also count pooling). All the convolutions, including those inside the Inception modules, use ReLU activation. Appropriate zero-paddings may be needed to keep the data dimension shape.

The size of the input to the network is the RGB color image [3, 224, 224]. The column "*depth*" specifies the number of layers with learning parameters in depth. The columns *"#1x1*", "*#3x3*", and "*#5x5*" specify the number of channels for 1x1, 3x3, 5x5 filters, respectively. The columns "*#3×3 reduce*" and "*#5×5 reduce*" specify the number of 1×1 filter channels in the reduction layer used before the 3×3 and 5×5 convolutions, respectively. The column "*pool_proj*" specifies the number of 1×1 filter channels following the inception built-in max-pooling. All the convolutional layers are followed by a batch normalization and ReLU activation. Please verify that Fig.9.10 (right) is the implementation of *inception(3a)* in the table.

Table 9.2 GoogLeNet architecture [from Szegedy 2015]

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

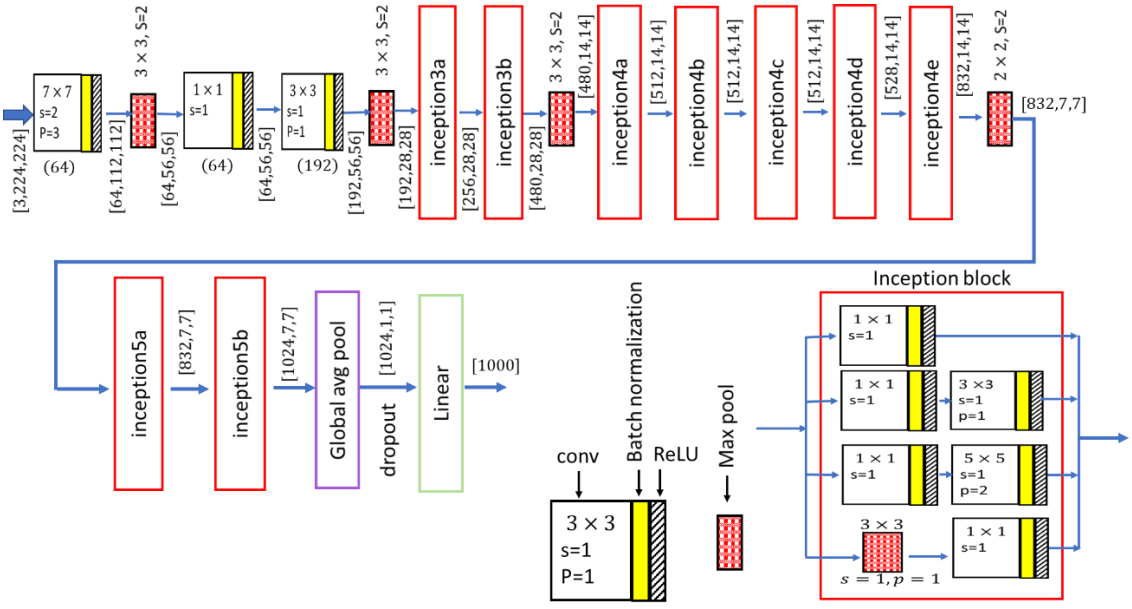


Fig.9.11 Diagram of GoogLeNet architecture

Following the table, we can draw the diagram of GoogLeNet, as shown in Fig.9.11. The first stack of inceptions includes 2 inception blocks, the second stack consists of 5 inception blocks, and the third stack consists of 2 inception blocks. Note that, in general, each inception block in any stack is different in terms of the number of channels, as specified in Table 9.2. In Fig.9.11 the data shape after each block is indicated by [Channel, Height, Width] (e.g., [64, 56,56]).

9.6 ResNet

After AlexNet, the state-of-the-art CNN architectures are going deeper and deeper to increase the expressive capacity of the networks. While AlexNet has only 5 convolutional layers, the VGG network and GoogLeNet have 19 and 22 parameterized layers respectively. However, increasing network depth does not always work by simply stacking layers together. Deep networks are hard to train because of the notorious gradient vanishing or exploding problem.

Residual neural network (ResNet), much deeper than previous neural networks, is the first working deep feedforward neural network with hundreds (or even thousands) of layers. The core idea of ResNet is introducing batch normalization and a so-called "identity shortcut connection" in the network. ResNet won the first place on the ILSVRC 2015 classification task with top-5 error rate of 3.57%, while still having lower complexity than VGG nets. Because of its compelling results, ResNet quickly became one of the most popular architectures in various computer vision tasks.

9.6.1 Residual block

Residual block is a basic building block in ResNet. The theoretical foundation of residual block is deep residual learning framework. Instead of hoping each few stacked layers directly fit a desired underlying mapping, we explicitly let these layers fit a residual mapping. Specifically, let us consider $\mathcal{H}(x)$ as an underlying mapping to be fit by a few stacked layers (not necessarily the entire net), called a *residual block*, with x denoting the inputs to the first of these layers. If one hypothesizes that multiple nonlinear layers can asymptotically approximate complicated functions, then it is equivalent to hypothesize that they can asymptotically approximate the residual functions, i.e., $\mathcal{H}(x) - x$ (assuming that the input and output are of the same dimensions). So rather than expect stacked layers to approximate $\mathcal{H}(x)$ directly, we explicitly let these layers approximate a residual function $\mathcal{F}(x) \coloneqq \mathcal{H}(x) - x$. The original function thus becomes $\mathcal{F}(x) + x$.

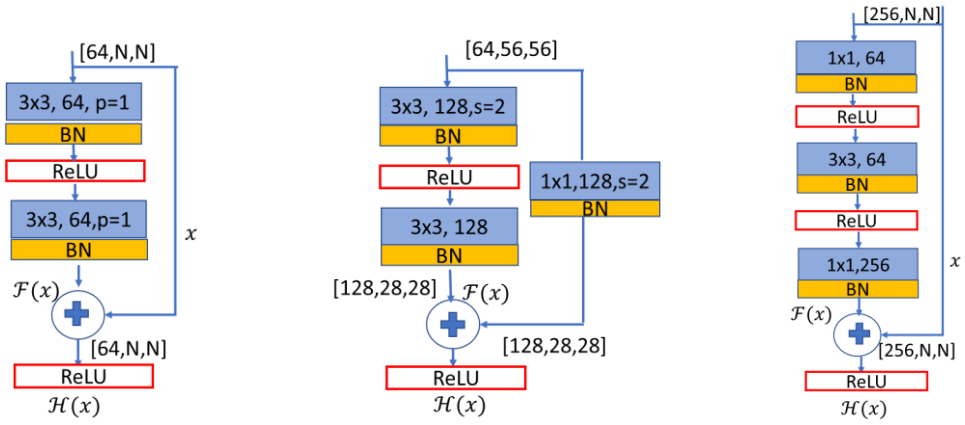


Fig.9.12 Residual block: 2-convolution layer with identity shortcut (left), 2-convolution layer with 1x1 shortcut for dimension match (middle), and 3-convolution layer with identity shortcut (right). Batch normalization is applied after each convolution layer and before activation.

The form of the residual function $\mathcal{F}$ is flexible. Experiments in the original paper involve a function $\mathcal{F}$ that has two or three layers, as shown in Fig.9.12, while more layers are possible. But if $\mathcal{F}$ has only a single layer (without activation), the block is equivalent to a linear layer, for which we have not observed advantages. Note that the addition in Fig.9.12 is performed in element-wise on

features for FC layers, or channel by channel for convolution layers. The 2-layer residual block is used for 18-layer ResNet and 34-layer ResNet while deeper ResNets (e.g. 50-layer, 101-layer, and 152-layer) replace the 2-layer residual blocks with 3-layer residual blocks (you will see later). The shortcut path should be implemented by a 1x1 convolution layer if the input map size is not the same as the output size (middle in Fig.9.12).

9.6.2 ResNet architectures

The original study of ResNet was performed on a few ResNet architectures with different sizes, detailed in Table 9.3, where residual blocks are shown in brackets (see also Fig.9.12), with the numbers of blocks stacked. Downsampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

Table 8.3 Architectures of ResNet

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

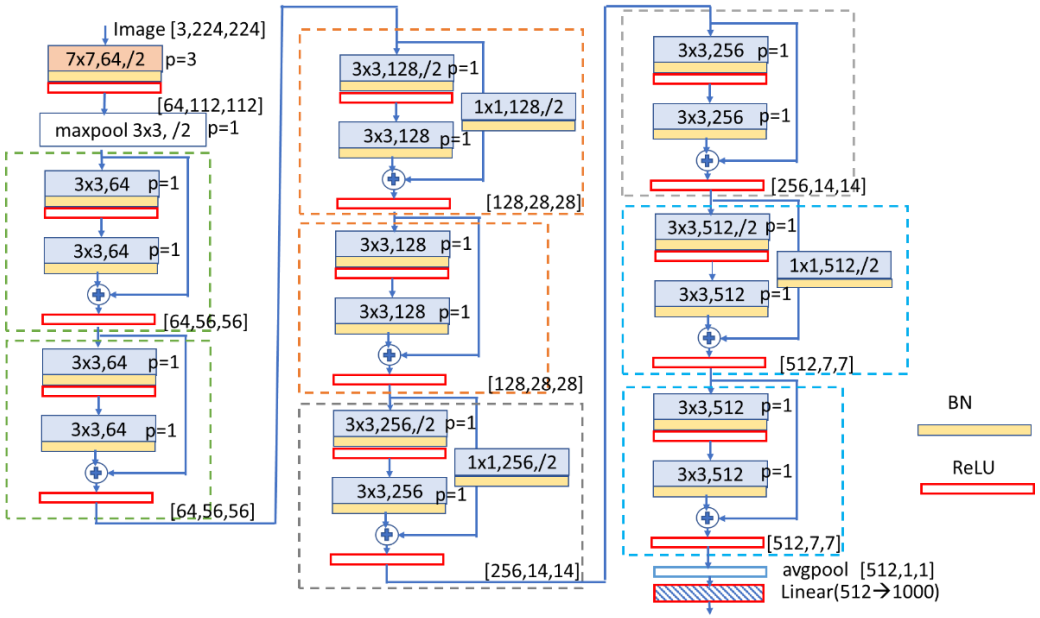


Fig.9.13 18-layer ResNet

To understand the above table and thus ResNet, we draw the detailed diagram of one of architectures: 18-layer ResNet. The ResNet is built upon its plain baseline counterpart by adding shortcut paths. The plain baseline counterpart was inspired by VGG-19. The convolutional layers mostly have 3×3 filters and follow two simple design rules: (i) for the same output feature map size, the layers have the same number of filters; and (ii) if the feature map size is halved, the number of filters is doubled so as to preserve the time complexity per layer. We perform downsampling directly by convolutional layers that have a stride of 2. The network ends with a global average pooling layer and a 1000-way fully-connected layer with softmax. The total number of weighted layers (not including the shortcut paths) is 18. Now you should be able to draw similar diagrams for other deeper ResNets.

9.7 Pretrained Models

So far we have described the architectures of some popular modern deep learning models, such as LeNet-5, AlexNet, NiN, GoogLeNet, and ResNet. We should be able to build the models using class layer models (e.g. nn.Conv2d, nn.MaxPool2d, nn.BatchNorm2d, and etc.) based on torch.nn package. However, this process is time consuming and prone to errors for many practitioners, and it usually takes long time to train them. If we want to adopt one of (or one similar to) these modern deep neural networks for a machine learning task, it is of a common practice to use the corresponding neural network model defined in *torchvision.models* package. These models can be either untrained or pretrained. A pretrained model is defined as a neural network model trained on standard datasets (e.g. ImageNet). We can modify the instance of the model by removing or adding layers, and fine tune it by training on your customized dataset. In this section, we will introduce how to use pretrained models.

9.7.1 Load Pretrained Model using torchvision

In PyTorch framework, many classical models are available in the package *torchvision.models*.

```
from torchvision import models     # import package models
dir(models)  # display different models
```

```
['AlexNet',
 'DenseNet',
 'GoogLeNet',
 'GoogLeNetOutputs',
 'Inception3',
 'InceptionOutputs',
 'MNASNet',
 'MobileNetV2',
 'ResNet',
 'ShuffleNetV2',
 'SqueezeNet',
 'VGG',
 '_GoogLeNetOutputs',
 '_InceptionOutputs',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
```

```
'__path__',
'__spec__',
'_utils',
'alexnet',
'densenet',
'densenet121',
'densenet161',
'densenet169',
'densenet201',
'detection',
'googlenet',
'inception',
'inception_v3',
'mnasnet',
'mnasnet0_5',
'mnasnet0_75',
'mnasnet1_0',
'mnasnet1_3',
'mobilenet',
'mobilenet_v2',
'quantization',
'resnet',
'resnet101',
'resnet152',
'resnet18',
'resnet34',
'resnet50',
'resnext101_32x8d',
'resnext50_32x4d',
'segmentation',
'shufflenet_v2_x0_5',
'shufflenet_v2_x1_0',
'shufflenet_v2_x1_5',
'shufflenet_v2_x2_0',
'shufflenetv2',
'squeezenet',
'squeezenet1_0',
'squeezenet1_1',
'utils',
'vgg',
'vgg11',
'vgg11_bn',
'vgg13',
'vgg13_bn',
'vgg16',
'vgg16_bn',
'vgg19',
'vgg19_bn',
'video',
'wide_resnet101_2',
'wide_resnet50_2']
```

Note that there are upper-case entries (e.g. AlexNet) and lower-case entries (e.g. alexnet) for the same model. The upper-case name refers to the Python class (e.g. AlexNet) whereas the lower-case name (alexnet) is a convenience function that returns the model instantiated from the class (e.g. AlexNet). It's also possible for these convenience functions to have different parameter sets. For example, resnet18, resnet34, resnet50, resnet101, resnet152, are instances of ResNet class but with a different number of layers – 18,34,50, 101 and 152, respectively.

Now we instantiate a pretrained AlexNet and print the detailed information on each layer.

```
# instantiate a pretrained model alexnet
alexnet = models.alexnet(pretrained=True)
```

To see the details on all layers, we can use ***print*** command.

```
print(alexnet) # print the architecture of Alexnet
```

```
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)
  )
)
```

To see the data shape after each layer and the number of parameters, we can use ***summary*** command.

```
from torchsummary import summary

summary (alexnet, (3,224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 55, 55]          23,296
              ReLU-2           [-1, 64, 55, 55]               0
         MaxPool2d-3           [-1, 64, 27, 27]               0
            Conv2d-4          [-1, 192, 27, 27]         307,392
              ReLU-5          [-1, 192, 27, 27]               0
         MaxPool2d-6          [-1, 192, 13, 13]               0
            Conv2d-7          [-1, 384, 13, 13]         663,936
              ReLU-8          [-1, 384, 13, 13]               0
            Conv2d-9          [-1, 256, 13, 13]         884,992
             ReLU-10          [-1, 256, 13, 13]               0
           Conv2d-11          [-1, 256, 13, 13]         590,080
             ReLU-12          [-1, 256, 13, 13]               0
        MaxPool2d-13            [-1, 256, 6, 6]               0
```

```
AdaptiveAvgPool2d-14            [-1, 256, 6, 6]               0
          Dropout-15                  [-1, 9216]               0
           Linear-16                  [-1, 4096]      37,752,832
             ReLU-17                  [-1, 4096]               0
          Dropout-18                  [-1, 4096]               0
           Linear-19                  [-1, 4096]      16,781,312
             ReLU-20                  [-1, 4096]               0
           Linear-21                  [-1, 1000]       4,097,000
================================================================
Total params: 61,100,840
Trainable params: 61,100,840
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 8.38
Params size (MB): 233.08
Estimated Total Size (MB): 242.03
----------------------------------------------------------------
```

(Note: in the inception block of GoogLeNet (or googlenet), the kernel 5×5 is implemented as a 3×3 kernel by PyTorch models. This is considered as a bug. Or it is ok if it works well. One can verify this by commands ***print*** and ***summary***).

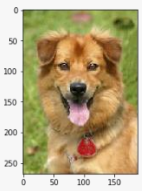
9.7.2 Image Classification using Pretrained AlexNet

Next, we will use the pretrained AlexNet to predict the class of an image. let's load the input image and carry out the appropriate transformations. Note that the input to the model should be a batch with the shape of [batch,channel,height,width], and the model predicts a batch of images at a time. To classify a single image, we need to convert the image to a batch that includes only one image.

To print the predicted result (category and confidence), we create a list "*classes*" by reading the file *imagenet_classes.txt*, which includes the name of one class in one line for 1000 classes and is available for downloading. The following code shows top-1 prediction and top-5 prediction.

■ Prepare image data

```
from PIL import Image
img = Image.open("C:/machine_learning/NN_dnn/dog.jpg")
plt.imshow(img)
```




```
transform = transforms.Compose([            #[1]
 transforms.Resize(256),                    #[2]
 transforms.CenterCrop(224),                #[3]
 transforms.ToTensor(),                     #[4]
 transforms.Normalize(                      #[5]
 mean=[0.485, 0.456, 0.406],                #[6]
 std=[0.229, 0.224, 0.225]                  #[7]
 )])
```

```
img_t = transform(img)
batch_t = torch.unsqueeze(img_t, 0)
```

- Run inference for the image.

```
alexnet.eval()
out = alexnet(batch_t)
print(out.shape)
```

```
torch.Size([1, 1000]
```

- Find the class corresponding to the maximal output.

```
with open('imagenet_classes.txt') as f:
  classes = [line.strip() for line in f.readlines()]
_, index = torch.max(out, 1)
percentage = torch.nn.functional.softmax(out, dim=1)[0] * 100
print(classes[index[0]], percentage[index[0]].item())
```

```
chow 54.78016662597656
```

- Find the top-5 classes.

```
_, indices = torch.sort(out, descending=True)
[(classes[idx], percentage[idx].item()) for idx in indices[0][:5]]
```

```
[('chow', 54.78016662597656),
 ('dingo', 19.381908416748047),
 ('Pembroke', 6.166537284851074),
 ('dhole', 5.396079063415527),
 ('tennis ball', 2.9083800315856934)]
```

9.7.3 Fine Tune Pretrained Model

In many applications, the datasets are usually not the same as the standard datasets which were used to train the pretrained models. For example, popular pretrained models, such as AlexNet, ResNet, and GoogleNet, are trained on ImageNet dataset with 1000 classes. However, for example, our computer vision task may involve only 10 classes, and these 10 classes may not be included in the ImageNet dataset. To rapidly develop such a neural network, we can load a pretrained model, and then modify the model by adding or removing some layers, and finally retrain the modified neural network using the customized dataset. This process is called *fine tune*.

Since the lower layers (i.e. layers near the input) in a model extract the features of the input while the deeper layers (i.e. layers close to the output) are responsible for classification, we usually keep the lower layers unchanged, and modify the last layer(s) for a specific application. To fine tune the model, we have two options: 1) update all parameters initialized with pretrained parameter values; and 2) update the parameters only for the last layer(s) while fixing pretrained model parameters for the rest layers.

In this section, we will present a simple example for fine tuning a pretrained model in PyTorch. Specifically, we will instantiate a pretrained model AlexNet for a 10-class classification task and change the last FC layer from 1000 nodes to 10 nodes. To fine tune the model based on our own dataset (e.g., CIFAR-10), we freeze all layer parameters except the last FC layer.

Load and Explore Pretrained Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models

# Instantiate a pretrained model alexnet
alexnet = models.alexnet(pretrained=True)
```

There are multiple ways we can investigate the model to see its modules and layers. In addition to the commands ***print*** () and ***summary*** (), we can use the function **.named_module* (), or **.named_children* (), which returns in iterator containing all the member objects of the model.

```
# Print all modules
for (name, module) in alexnet.named_modules():
    print(name, module)
```

(Note: the outputs are omitted here)

```
# print the modules at children level
for (name, module) in alexnet.named_children():
    print(name, module)
```

```
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)
)
```

We can print the information on some specific layers, say layers for classification, like this:

```
for (name, module) in alexnet.named_children():
    if name == 'classifier':
        for layer in module.children():
            print(layer)
```

```
    Dropout(p=0.5, inplace=False)
    Linear(in_features=9216, out_features=4096, bias=True)
    ReLU(inplace=True)
    Dropout(p=0.5, inplace=False)
    Linear(in_features=4096, out_features=4096, bias=True)
    ReLU(inplace=True)
    Linear(in_features=4096, out_features=1000, bias=True)
```

The following codes print the layers' name and parameters values for "classifier".

```
for (name, module) in alexnet.named_children():
    if name == 'classifier':
        for layer in module:
            print(layer)
            for param in layer.parameters():
                print(param)
```

(Note: the output is omitted here)

Modify Pretrained Model

We change the out_features of the last layer in 'classifier' from 1000 to 10. Then the parameters for this layer are randomly initialized while other layers maintain the pretrained parameters. We can also delete or add some layers (you are encouraged to find the details from internet resources).

```
in_features = alexnet._modules['classifier'][-1].in_features
out_features = 10
alexnet._modules['classifier'][-1] = nn.Linear(in_features, out_features, bias=
True)
print(alexnet._modules['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)
)
```

We can see that the `out_features` in the last layer has changed from 1000 to 10.

Freeze Parameters

Now we freeze (or fix) some parameters during the training process, by setting their `param.requires_grad = False`, and allow other parameters to be updated in training by setting `param.requires_grad = True`.

```
for (name, module) in alexnet.named_children():
    print(name)
    for (layer_name,layer) in module.named_children():
        if name == 'classifier' and layer_name =="6":
            for param in layer.parameters():
                param.requires_grad = True
            print('{} {} {} was NOT frozen!'.format(name, layer_name, layer))
        else:
            for param in layer.parameters():
                param.requires_grad = False

            print('{} {} {} was frozen!'.format(name, layer_name, layer))
```

```
features
features 0 Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2)) w
as frozen!
features 1 ReLU(inplace=True) was frozen!
features 2 MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=
False) was frozen!
features 3 Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2)) w
as frozen!
features 4 ReLU(inplace=True) was frozen!
features 5 MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=
False) was frozen!
features 6 Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
was frozen!
features 7 ReLU(inplace=True) was frozen!
features 8 Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
was frozen!
features 9 ReLU(inplace=True) was frozen!
features 10 Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
was frozen!
features 11 ReLU(inplace=True) was frozen!
features 12 MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode
=False) was frozen!
avgpool
classifier
classifier 0 Dropout(p=0.5, inplace=False) was frozen!
classifier 1 Linear(in_features=9216, out_features=4096, bias=True) was frozen!
classifier 2 ReLU(inplace=True) was frozen!
classifier 3 Dropout(p=0.5, inplace=False) was frozen!
classifier 4 Linear(in_features=4096, out_features=4096, bias=True) was frozen!
classifier 5 ReLU(inplace=True) was frozen!
classifier 6 Linear(in_features=4096, out_features=10, bias=True) was NOT froze
n!
```

Prepare Datasets

Assuming that we use the pretrained model on the dataset CIFAR-10. To fine tune the model, we prepare the datasets as follows. We need to transform the images to [3,224, 224] for the AlexNet input requirement.

```
batch_size=32
#transform = transforms.Compose([transforms.ToTensor(),
#                                 transforms.Normalize((0.5,0.5,0.5), (0.5, 0.5,
0.5))])
transform = transforms.Compose([            #[1]
 transforms.Resize(256),                    #[2]
 transforms.CenterCrop(224),                #[3]
```

```
 transforms.ToTensor(),                          #[4]
 transforms.Normalize(                           #[5]
 mean=[0.485, 0.456, 0.406],                     #[6]
 std=[0.229, 0.224, 0.225]                       #[7]
 )])
```

```
trainset = torchvision.datasets.CIFAR10(root='C:/Users/weido/ch11/data', train=
True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='C:/Users/weido/ch11/data', train=F
alse, download=False, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

Train the model and Test the Model

First, we test the pretrained model before fine tune. Since the last linear layer were modified and its parameters are randomly initialized. Thus, the network will randomly classify input images. As a result, the overall classification accuracy is about 10%.

```
correct = 0
total = 0
counter=0
with torch.no_grad():
    for data in testloader:
        counter=counter+1
        print(counter)
        images, labels = data
        outputs = alexnet(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 8 %
```

Now we retrain the modified pretrained model. Note that the training only runs 2 epochs in the following code to get the result in a reasonable time on CUP. During the training process, only the parameters of the last layer are updated.

```
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(filter(lambda p: p.requires_grad, alexnet.parameters()),
lr=0.001)

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader,0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()
```

```
        # forward + backward + optimize
        outputs = alexnet(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()

        if i % 100 == 99:    # print every 100 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0

print('Finished Training')
```

```
Accuracy of the network on the 10000 test images: 80 %
```

After two-epoch training, we test the model again on the test dataset and get an accuracy of 80%. Note that the accuracy may depend on the training run due to the weight random initialization in the last layer. You can verify that the parameters in the last layer are updated but those in other layers not, after fine tuning.

Summary

In this chapter, first we described a few standard datasets: MNIST, Fashion-MNIST, CIFAR10 and CIFAR100, ImageNet, COCO dataset, and Cityscapes. All these datasets can be downloaded through *torchvision.datasets* (https://pytorch.org/vision/stable/datasets.html). Note that there are many other datasets available in *torchvision.datasets*. Some (e.g. ImageNet) are for image classification while others (e.g. COCO dataset, Cityscapes) for object detection and segmentation.

Then, we present the architectures of some milestone deep neural networks: AlexNet, VGG, NiN, GoogLeNet, and ResNet. The implementations of AlexNet, VGG, NiN in PyTorch are given. The detailed PyTorch implementation of GoogLeNet and ResNet can be found in "*Dive into Deep Learing (PyTorch)*". The purpose of this chapter is not only to introduce the successful neural networks, more importantly, but also to inspire the reader by explaining why they are successful when they are going deeper. The reason is a combination of mathematical logic and some intuitions. Batch normalization is usually an important layer for deep neural networks for stable and faster training.

Finally, instead of developing a deep neural network from scratch in practice, we demonstrate how to utilize pretrained models available in *torchvision.models*. We can either retrain the entire model or update only the partial model.

File: ch7_practical_cnn/ch7_cnn.ipynb.

Further reading

Original papers:

AlexNet
Krizhevsky, A., Sutskever, I., and Hinton, G. (2012), *ImageNet classification with deep convolutional neural networks*. In Proc. Advances in Neural Information Processing Systems (NIPS) 25 1090–1098 (2012).

VGG
Karen Simonyan, and Andrew Zisserman (2015), *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv:1409.1556 **[cs.CV].**

NiN
Min Lin, Qiang Chen, and Shuicheng Yan (2014), *Network In Network*. arXiv:1312.4400 **[cs.NE].**

GoogLeNet
Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich (2015), *Going Deeper with Convolutions*. arXiv:1409.4842 **[cs.CV]**

ResNet
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2016), *Deep Residual Learning for Image Recognition*. arXiv:1512.03385 **[cs.CV]**

Exercises

1. Download the following datasets using *torchvision.datasets*, and explore some sample images and their labels.
   1) MNIST
   2) Fashion-MNIST
   3) EMINST.
   4) CIFAR-10
   5) CIFAR-100
   6) CelebA
   7) Coco detection
   8) Cityscapes

2. Calculate the number of learnable parameters in AlexNet,VGG16, GooglNet, and ResNet50, respectively. Verify your results with Python codes.

   (hints:

   ```
   numel_list = [p.numel() for p in net_VGG16.parameters() if p.requires_gra
   d == True]
   ```

)

3. Fine tune the following models from *torchvision.models* on CIFAR-10 dataset.

   1) vgg13; 2) googlenet; 3) resnet18.

   Compare them, in terms of classification accuracy and the number of parameters.

4. Resnets are designed for input images [3,224, 224]. Thus, they may be too complex for CIFAR-10 dataset due to the low resolution of the images [3, 32. 32]. Design a simplified resnet using inception blocks for CIFAR-10, with a good accuracy.

5. Fix the bug in PyTorch googlenet (if the bug is still there).
   1) Load the pretrained model googlenet from *torchvision.models*, and verify that the kernel in the branch 3 of each inception block is (3,3), instead of (5,5).
   2) Fine tune the googlenet model CIFAR-10 dataset. Test the performance.
   3) Load the pretrained model googlenet from *torchvision.models* again, and change the kernel in the branch 3 of each inception block from (3,3) to (5,5), and pad appropriate zeros. Fine tune the modified model in the same way in 2), and test the performance.
   4) Compare the results between 2) and 3).

6. Compare the model parameter sizes and computation complexities of AlexNet, VGG, NiN, GoogLeNet, and ResNet.