Mạng phần dư (ResNet)

Khi thiết kế các mạng ngày càng sâu, ta cần hiểu việc thêm các tầng sẽ tăng độ phức tạp và khả năng biểu diễn của mạng như thế nào. Quan trọng hơn là khả năng thiết kế các mạng trong đó việc thêm các tầng vào mạng chắc chắn sẽ làm tăng tính biểu diễn thay vì chỉ tạo ra một chút khác biệt.

Để làm được điều này, chúng ta cần một chút lý thuyết.

Function Classes

Coi $\mathcal{F}$ là một lớp các hàm mà một kiến trúc mạng cụ thể(Cùng với tốc độ học và các siêu tham số khác) có thể đạt được. Nói cách khác, với mọi hàm số $f \in \mathcal{F}$ luôn tồn tại một số tập tham số (như weights và biases) có thể tìm được bằng việc huấn luyện trên một tập dữ liệu phù hợp.

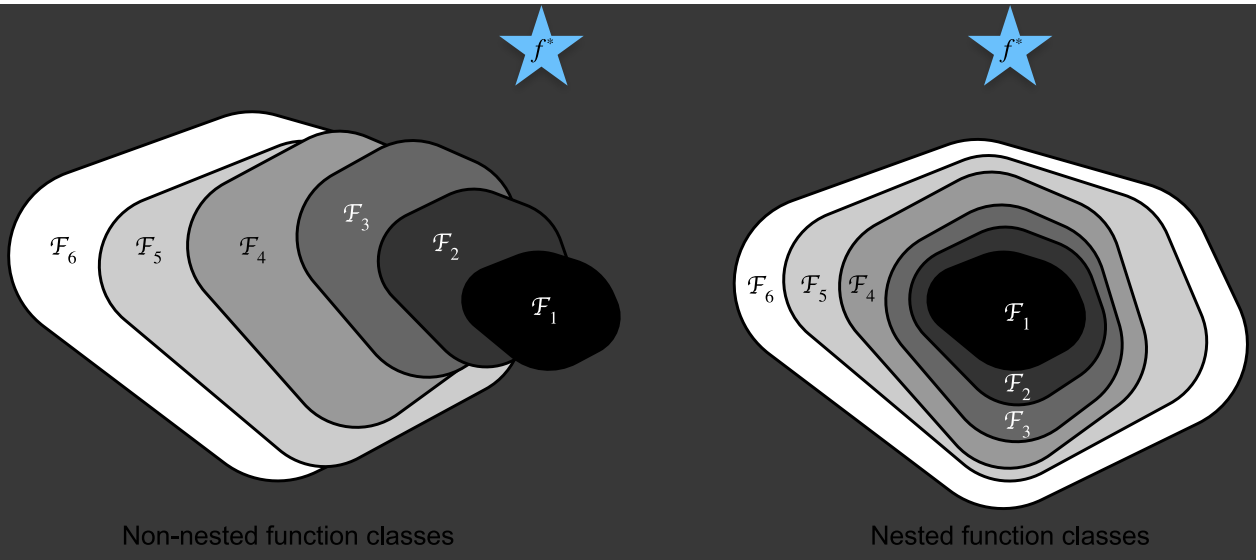
Giả sử f^* là hàm chúng ta cần tìm. Nếu nó thuộc tập $\mathcal{F}$ thì sẽ tốt, nhưng thương không may mắn như vậy :)) Thay vào đó, Chúng ta sẽ cố gắng tìm các hàm số $f^*_\mathcal{F}$ tố nhất có thể trong tập $\mathcal{F}$.

Ví dụ, chúng ta thử giải bài toán tối ưu sau, với tập dataset cho trước, (Features $\mathbf{X}$ và labels $\mathbf{y}$):

$$f^*_\mathcal{F} \stackrel{\mathrm{def}}{=} \mathop{\mathrm{argmin}}_f L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$$

Khá hợp lý khi giả sử rằng nếu thiết kế một kiến trúc khác $\mathcal{F}'$ chúng ta có thể đạt được kết quả tốt hơn. Nói cách khác, chúng ta kỳ vọng $f^*_{\mathcal{F}'}$ sẽ "tốt hơn" $f^*_{\mathcal{F}}$. Tuy nhiên, nếu $\mathcal{F} \not\subseteq \mathcal{F}'$ thì không thể khẳng định là $f^*_{\mathcal{F}'}$ tốt hơn $f^*_{\mathcal{F}}$. Thực tế, $f^*_{\mathcal{F}'}$ thậm chí có thể xấu hơn.

Việc thêm các tầng không phải lúc nào cũng làm tăng tính biểu diễn của mạng. Xem ví dụ ở hình sau, chúng ta thấy rằng ở hình bên trái(các lớp hàm số tổng quát), $\mathcal{F}_3$ gần f^* hơn là $\mathcal{F}_1$, $\mathcal{F}_6$, không thể đảm bảo rằng tăng độ phực tạp có thể giảm khoảng cách với f^*. Trong khi đó, với ở hình bên phải(các lớp với hàm số lống nhau) với $\mathcal{F}_1 \subseteq \ldots \subseteq \mathcal{F}_6$ chúng ta có thể tránh được vấn đề ở trường hợp các lớp hàm số tổng quát, khi mà khoảng cách đến hàm cần tìm f^*, trên thực tế có thể tăng khi độ phức tạp tăng lên



Chỉ khi các lớp hàm lớn hơn chứa các lớp nhỏ hơn, thì mới đảm bảo rằng việc tăng thêm các tầng sẽ tăng khả năng biểu diễn của mạng. Nếu ta huấn luyện tầng mới được thêm vào thành một ánh xạ đồng nhất $f(\mathbf{x}) = \mathbf{x}$, thì mô hình mới sẽ hiệu quả ít nhất bằng mô hình ban đầu. Vì tầng được thêm vào có thể khớp dữ liệu huấn luyện tốt hơn, dẫn đến sai số huấn luyện cũng nhỏ hơn.

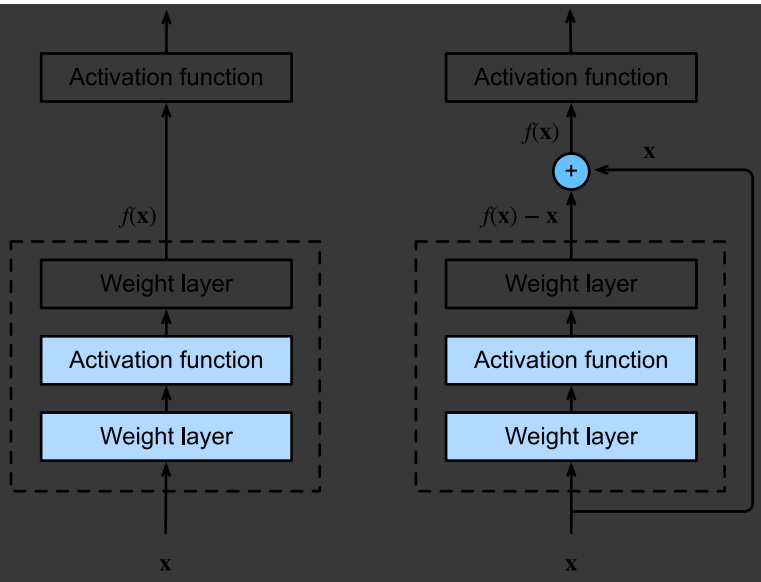
Đây là câu hỏi mà He và các cộng sự đã suy nghĩ khi nghiên cứu các mô hình thị giác sâu năm 2016. Ý tưởng trọng tâm của ResNet là mỗi tầng được thêm vào nên có một thành phần là hàm số đồng nhất

Cách suy nghĩ này khá trừu tượng nhưng lại dẫn đến một lời giải đơn giản khá ngạc nhiên, một khối phần dư (residual block). Với ý tưởng này, ResNet đã chiến thắng cuộc thi Nhận dạng Ảnh ImageNet năm 2015. Thiết kế này có ảnh hưởng sâu sắc tới việc xây dựng các mạng nơ-ron sâu.

(Residual Blocks)

Chúng ta sẽ tập trung vào mạng sau đây. Đầu vào là $\mathbf{x}$, hàm ánh xạ cần học được là $f(\mathbf{x})$, và được dùng làm đầu vào của hàm kích hoạt(on top). Phần nằm trong viền nét đứt bên trái phải khớp trực tiếp với ánh xạ $f(\mathbf{x})$. Ở hình bên phải, phần trong viền nét đứt cần biểu diễn được *residual mapping* $f(\mathbf{x}) - \mathbf{x}$, giống với cái tên của nó :)).

Trên thực tế, ánh xạ phần dư thường dễ tối ưu hơn, chúng ta chỉ cần để weights và biases (e.g., fully-connected layer and convolutional layer) nằm trong phần nét đứt về 0. Trong ảnh bên phải, chúng ta thấy có một shortcut connection nối ngay tử input. Với resudual blocks, quá trình lan truyền thuận nhanh hơn thông qua kết nối residual.



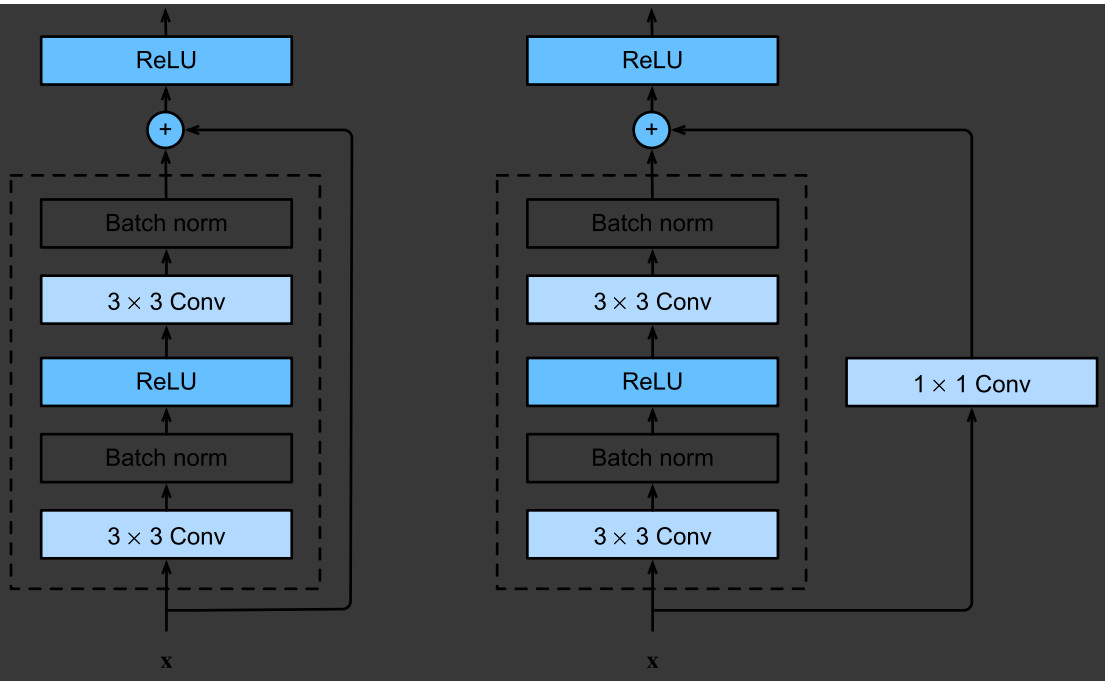
ResNet có thiết kế tầng tích chập 3×3 giống VGG. Khối phần dư có hai tầng tích chập 3×3 với cùng số kênh đầu ra. Mỗi tầng tích chập được theo sau bởi một tầng chuẩn hóa theo batch và một hàm kích hoạt ReLU. Ta đưa đầu vào qua khối phần dư rồi cộng với chính nó trước hàm kích hoạt ReLU cuối cùng. Thiết kế này đòi hỏi đầu ra của hai tầng tích chập phải có cùng kích thước với đầu vào, để có thể cộng lại với nhau. Nếu muốn thay đổi số lượng kênh hoặc sải bước trong khối phần dư, cần thêm một tầng tích chập 1×1 để thay đổi kích thước đầu vào tương ứng ở nhánh ngoài. Hãy cùng xem đoạn mã bên dưới.

```
import torch
import torch.nn as nn

class Residual(nn.Module):
    def __init__(self, in_channels, num_channels, use_1x1conv=False, strides=1):
        super(Residual, self).__init__()
        self.conv1 = nn.Conv2d(None,None, kernel_size=None, padding=None,stride=No
        self.conv2 = nn.Conv2d(None,None, kernel_size=None, padding=None)
        self.conv3 = None # Đừng sửa None này nhé :!
        if use_1x1conv:
            self.conv3 = nn.Conv2d(None,None,kernel_size=None,stride=None)
        self.bn1 = nn.BatchNorm2d(None)
        self.bn2 = nn.BatchNorm2d(None)

    def forward(self, X):
        Y = nn.ReLU()(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return nn.ReLU()(Y + X)
```

Đoạn mã này tạo ra hai loại mạng: một loại cộng đầu vào vào đầu ra trước khi áp dụng hàm phi tuyến ReLU (khi use_1x1conv=True), còn ở loại thứ hai chúng ta thay đổi số kênh và độ phân giải bằng một tầng tích chập 1×1 trước khi thực hiện phép cộng




```
X = torch.randn((4, 3, 6, 6))
blk = Residual(in_channels = None, num_channels = 3)
assert blk(X).shape  == (4,3,6,6)
```

Chúng ta cũng có thể giảm một nửa kích thước chiều cao và chiều rộng của đầu ra trong khi tăng số kênh.

```
blk = Residual(in_channels = None,num_channels = 6, use_1x1conv=True, strides=2)
assert blk(X).shape == (4,6,3,3)
```

[ResNet Model]

Hai tầng đầu tiên của ResNet giống hai tầng đầu tiên của GoogLeNet: tầng tích chập $7\times 7$$7\times 7$ với 64 kênh đầu ra và sải bước 2, theo sau bởi tầng maxpool $3\times 3$$3\times 3$ với sải bước 2. Sự khác biệt là trong ResNet, mỗi tầng tích chập theo sau bởi tầng chuẩn hóa theo batch.

```
net = nn.Sequential()
net.add_module("conv",nn.Conv2d(1,None, kernel_size=None, stride=None, padding=3))
net.add_module("batchnorm",nn.BatchNorm2d(None))
net.add_module("Relu",nn.ReLU())
net.add_module("maxpool",nn.MaxPool2d(None, stride = None, padding = 1))
```

GoogLeNet sử dụng bốn mô-đun được tạo thành từ các khối Inception. ResNet sử dụng bốn mô-đun được tạo thành từ các khối phần dư có cùng số kênh đầu ra. Mô-đun đầu tiên có số kênh bằng số kênh đầu vào. Vì trước đó đã sử dụng tầng gộp cực đại với sải bước 2, nên không cần

phải giảm chiều cao và chiều rộng ở mô-đun này. Trong các mô-đun sau, khối phần dư đầu tiên nhân đôi số kênh, đồng thời giảm một nửa chiều cao và chiều rộng.

Bây giờ ta sẽ lập trình mô-đun này. Chú ý rằng mô-đun đầu tiên được xử lý khác một chút.

```
def resnet_block(in_channels ,num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add_module('residual_{}'.format(i),Residual(None , None, use_1x1co
        else:
            blk.add_module('residual_{}'.format(i),Residual(None, None))
    return blk
```

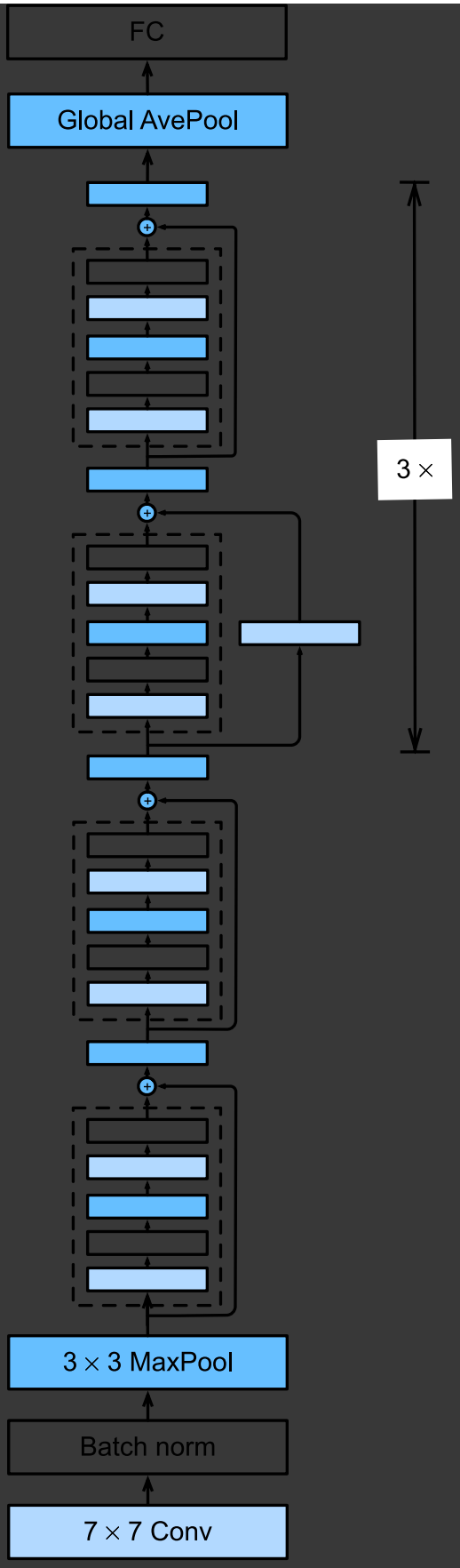
Sau đó, chúng ta thêm các khối phần dư vào ResNet. Ở đây, mỗi mô-đun có hai khối phần dư.

```
net.add_module('resnet_block1',resnet_block(64, 64, 2, first_block=True))
net.add_module('resnet_block2',resnet_block(64,128, 2))
net.add_module('resnet_block3',resnet_block(128,256, 2))
net.add_module('resnet_block4',resnet_block(256,512, 2))
```

Cuối cùng, giống như GoogLeNet, ta thêm một tầng GlobalAvgPool2D và một tầng Dense.

```
net.add_module('GlobalAvr',nn.AdaptiveAvgPool2d((1, 1)))
net.add_module('Flatten',nn.Flatten())
net.add_module('FC',nn.Linear(None,None))
```

Có 4 tầng tích chập trong mỗi mô-đun (không tính tầng tích chập 1×1). Cộng thêm tầng tích chập đầu tiên và tầng kết nối đầy đủ cuối cùng, mô hình có tổng cộng 18 tầng. Do đó, mô hình này thường được gọi là ResNet-18. Có thể thay đổi số kênh và các khối phần dư trong mô-đun để tạo ra các mô hình ResNet khác nhau, ví dụ mô hình 152 tầng của ResNet-152. Mặc dù có kiến trúc lõi tương tự như GoogLeNet, cấu trúc của ResNet đơn giản và dễ sửa đổi hơn. Tất cả các yếu tố này dẫn đến sự phổ cập nhanh chóng và rộng rãi của ResNet



Trước khi huấn luyện, hãy quan sát thay đổi của kích thước đầu vào qua các mô-đun khác nhau trong ResNet. Như trong tất cả các kiến trúc trước, độ phân giải giảm trong khi số lượng kênh tăng đến khi tầng gộp trung bình toàn cục tổng hợp tất cả các đặc trưng.

```
X = torch.randn((1, 1, 224, 224))
for layer in net:
```

```
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__, 'output shape:\t', X.shape)
```

[Training]

Chúng ta sẽ train với tập MNIST, các bước như các bài notebook trước nhé bạn.

```
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt
import random
```

```
epochs = None

# Các tham sô´cần thiết cho quá trình traning.
learning_rate = None
batch_size = None
display_step = None

# Path lưu best model
checkpoint = 'model.pth' # có thể để dạng *.pth

# device chúng ta dùng cuda
device = 'cuda' if torch.cuda.is_available() else 'cpu'
assert device == 'cuda'
```

```
# Transform image
transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
    ])

# load dataset từ torchvision.datasets
train_dataset = datasets.MNIST('../data', train=True, download=True,transform=None
test_dataset = datasets.MNIST('../data', train=False,transform=None)
train_loader = torch.utils.data.DataLoader(None,batch_size=None
test_loader = torch.utils.data.DataLoader(None,batch_size=None)
```

```
# call model, set deivce
None
# load lại pretrained model (nếú có)
try:
  None
except:
  print("!!! Hãy train để có checkpoint file")
```

```
    !!! Hãy train để có checkpoint file
```

```
criterion = None
optimizer = None
best_val_loss = 999

for epoch in range(1,epochs):
    # Quá trình training
    model.train()
    for batch_idx, (data, target) in enumerate(None):
        data, target = None, None # device?
        None # Zero_grad
        output = None
        loss = None
        None # backward
        None # update weights
        if batch_idx % display_step == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tTrain Loss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
    # Quá trình testing
    model.eval()
    test_loss = 0
    correct = 0
    # set no grad cho quá trình testing
    with torch.no_grad():
        for data, target in None:
            data, target = None, None
            output = None
            output = None # log softmax using F, chu y dim nhe
            test_loss += None
            pred = None # argmax để lấy predicted label, chú ý keepdim = True
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    if test_loss < best_val_loss:
      best_val_loss = test_loss
      None # lưu model
      print("***********    TEST_ACC = {}%    ***********".format(correct))
```

Tóm tắt

- Ánh xạ phần dư có thể học hàm nhận dạng dễ dàng hơn, chẳng hạn như đẩy các tham số trong lớp trọng số về không.
- Chúng ta có thể huấn luyện hiệu quả mạng nơ-ron sâu nhờ khối phần dư chuyển dữ liệu liên tầng.
- ResNet có ảnh hưởng lớn đến thiết kế sau này của các mạng nơ-ron sâu, cả tích chập và tuần tự.