курс «Глубокое обучение»

Архитектуры свёрточных нейронных сетей

часть 1: чемпионы ImageNet и их «родственники»

Александр Дьяконов

5 марта 2022 года

План (чемпионы ImageNet и их «родственники»):

LeNet-5 (1998)

AlexNet (2012)

VGG (2014) + анализ

GoogLeNet / Inception (2014)

ResNet = Residual Network (2015)

Highway Net

Inception v2-v4

Xception

SENet

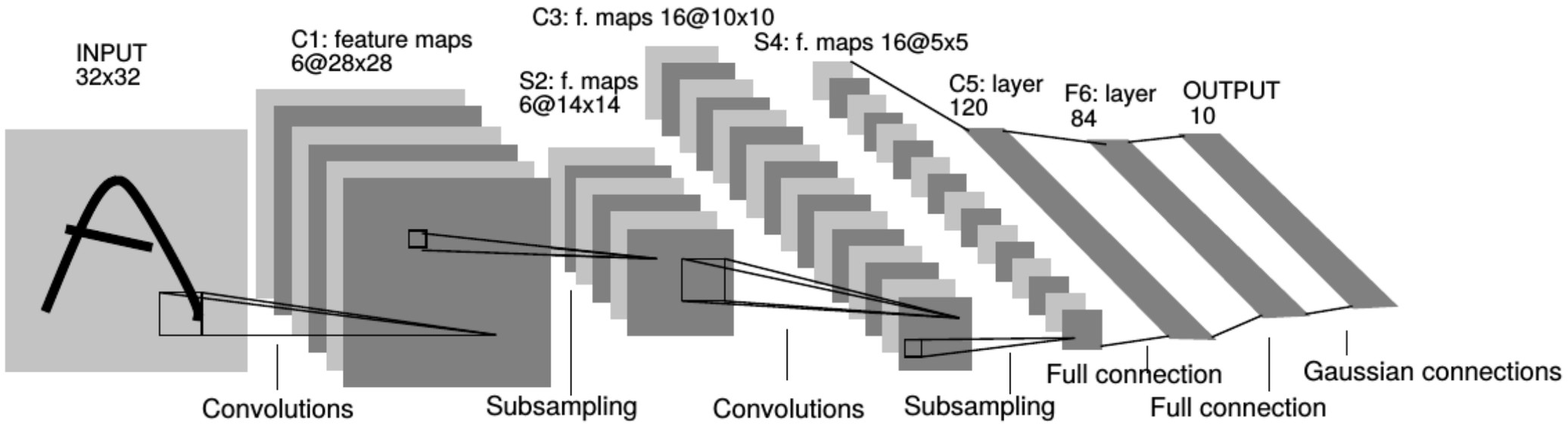
ResNet: почему работает

Классические архитектуры в наши дни

Что не вошло в первую часть

Network in Network (NiN), Deep Networks with Stochastic Depth, FractalNet, Fractal of FractalNet, DenseNets, SqueezeNet, ResNeXt, MultiResNet, PolyNet, HyperNets, EfficientNet, MobileNet, SqueezeNet, ShuffleNet, FBNet (+NAS), WideResNets, RevNet, iRevNet, NFNets

LeNet-5 (1998)



5 = 2 свёрточных слоя + 3 полносвязных

свёртки 5×5

C@H×W

третий слой можно считать свёрточным, а можно полносвязным

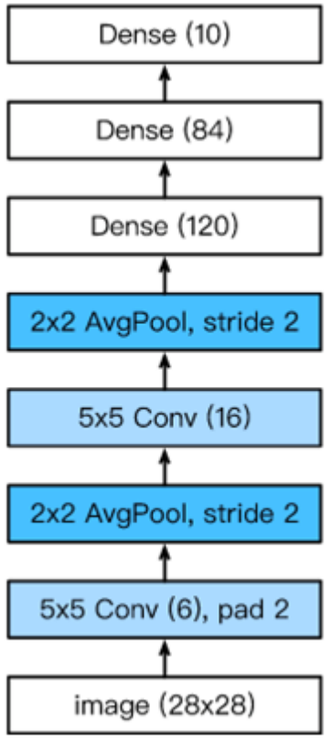
(там 5×5-свёртка)

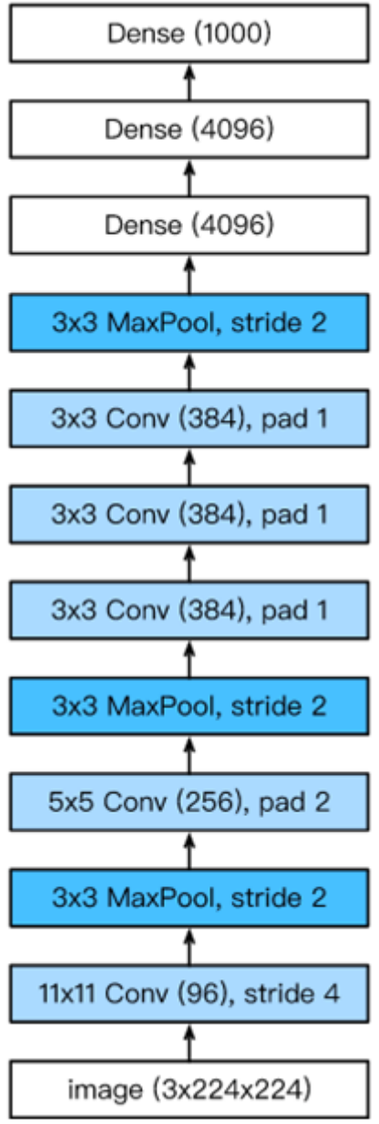
Минутка кода: модификация LeNet-5

```
class LeNet5(nn.Module):
    def __init__(self, n_classes):
        super(LeNet5, self).__init__()

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.Tanh()
        )
        self.classifier = nn.Sequential(
            nn.Linear(in_features=120, out_features=84),
            nn.Tanh(),
            nn.Linear(in_features=84, out_features=n_classes),
        )
    def forward(self, x):
        x = self.feature_extractor(x)
        x = torch.flatten(x, 1)
        logits = self.classifier(x)
        probs = F.softmax(logits, dim=1)
        return logits, probs
```

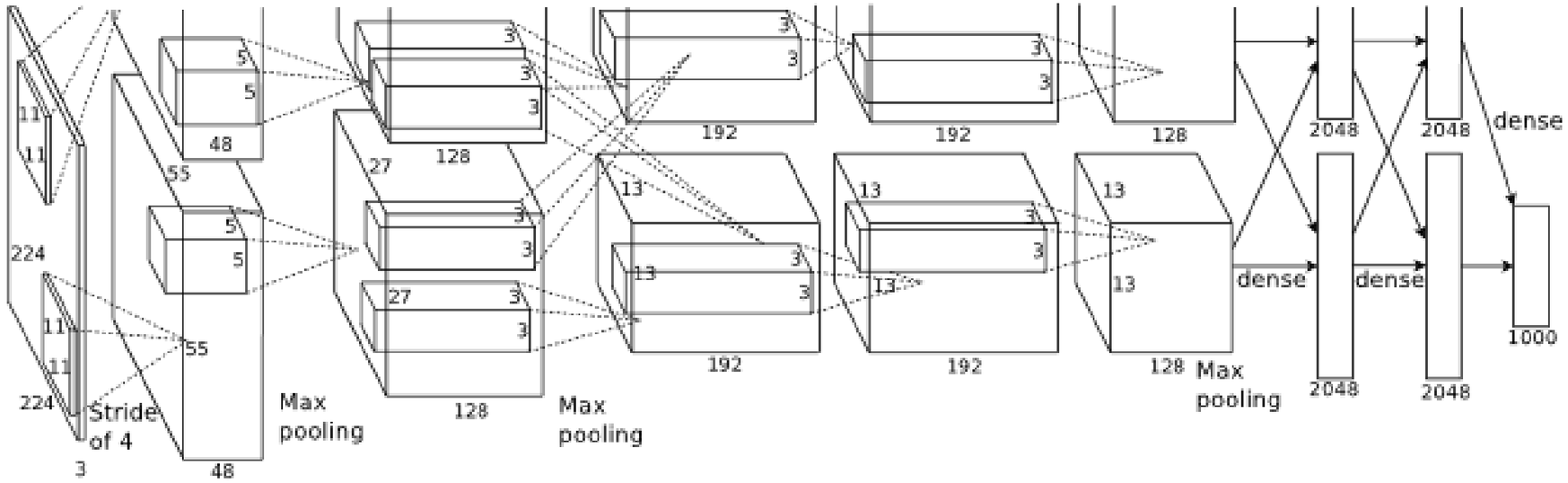
https://github.com/erykml/medium_articles/blob/master/Computer%20Vision/lenet5_pytorch.ipynb

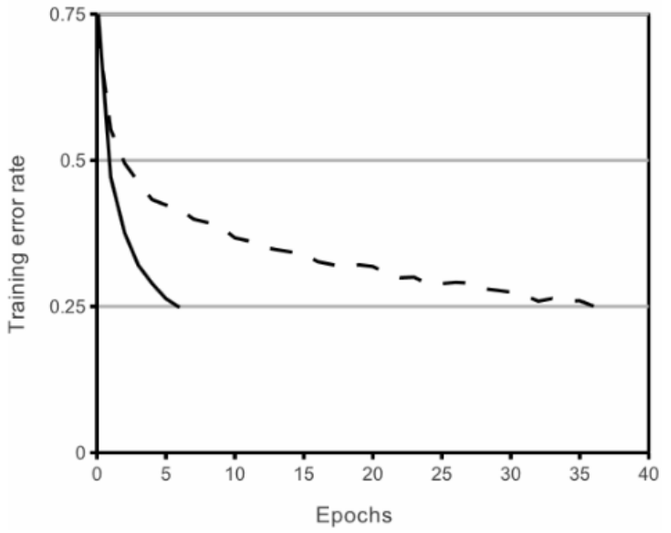
LeNet-5 → AlexNet





AlexNet (2012)





- **ReLU – после ∀ conv и dense слоя (см. рис, скорость 6×)**
  - **MaxPool (вместо AvgPool), полно-связные слои**
    - **Data augmentation**
  - **Dropout 0.5 (но и время обучения 2×)**
  - **Batch size = 128, SGD Momentum = 0.9**
    - **60M параметров / 650K нейронов**
    - **1 неделя на 2 GPU (50x над CPU)**
      - **7 скрытых слоёв**

Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton “ImageNet Classification with Deep Convolutional Neural Networks” // https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

AlexNet (2012)

Dropout – перед 1м и 2м полносвязными слоями

свёртки 3×3, 5×5, 11×11 (большие по сравнению с LeNet, в них stride=4)

несколько свёрточных слоёв подряд (не было в LeNet)

7 CNN ансамбль: 18.2% → 15.4%

Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SIFT + FVs [7]	—	—	*26.2%*
1 CNN	40.7%	18.2%	—
5 CNNs	38.1%	16.4%	**16.4%**
1 CNN*	39.0%	16.6%	—
7 CNNs*	36.7%	15.4%	**15.3%**

Table 2: Comparison of error rates on ILSVRC-2012 validation and test sets. In *italics* are best results achieved by others. Models with an asterisk* were "pre-trained" to classify the entire ImageNet 2011 Fall release. See Section 6 for details.

AlexNet (2012)

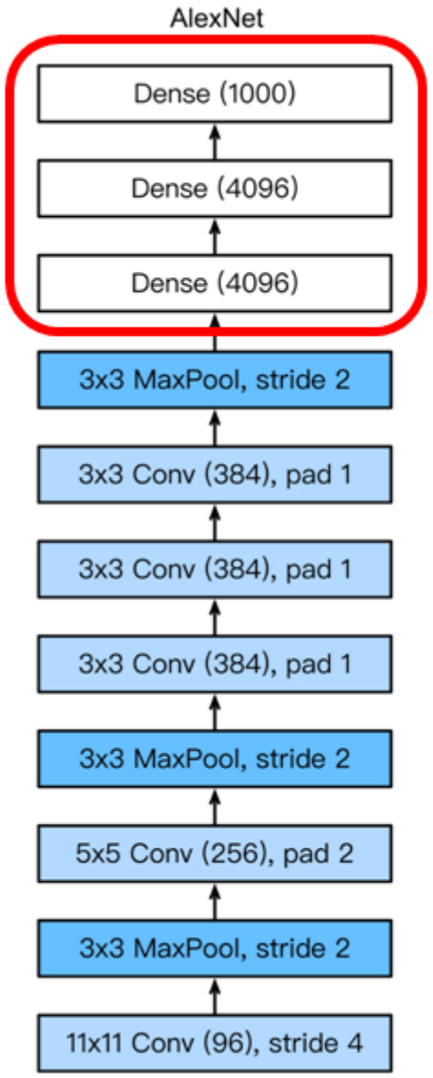
Есть тонкость с нормализацией после RELU:

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^\beta$$

Эксперименты по удалению слоёв

Убрать 16M параметров (последний полносвязный слой) – качество 1.1% ↓
Убрать 50M параметров (2 последних полносвязных слоя) – качество 5.7% ↓
Убрали 1M параметр (3 и 4 слои) – качество 3.0% ↓
Убрать несколько слоёв (3, 4, 6, 7) – качество 33.5% ↓

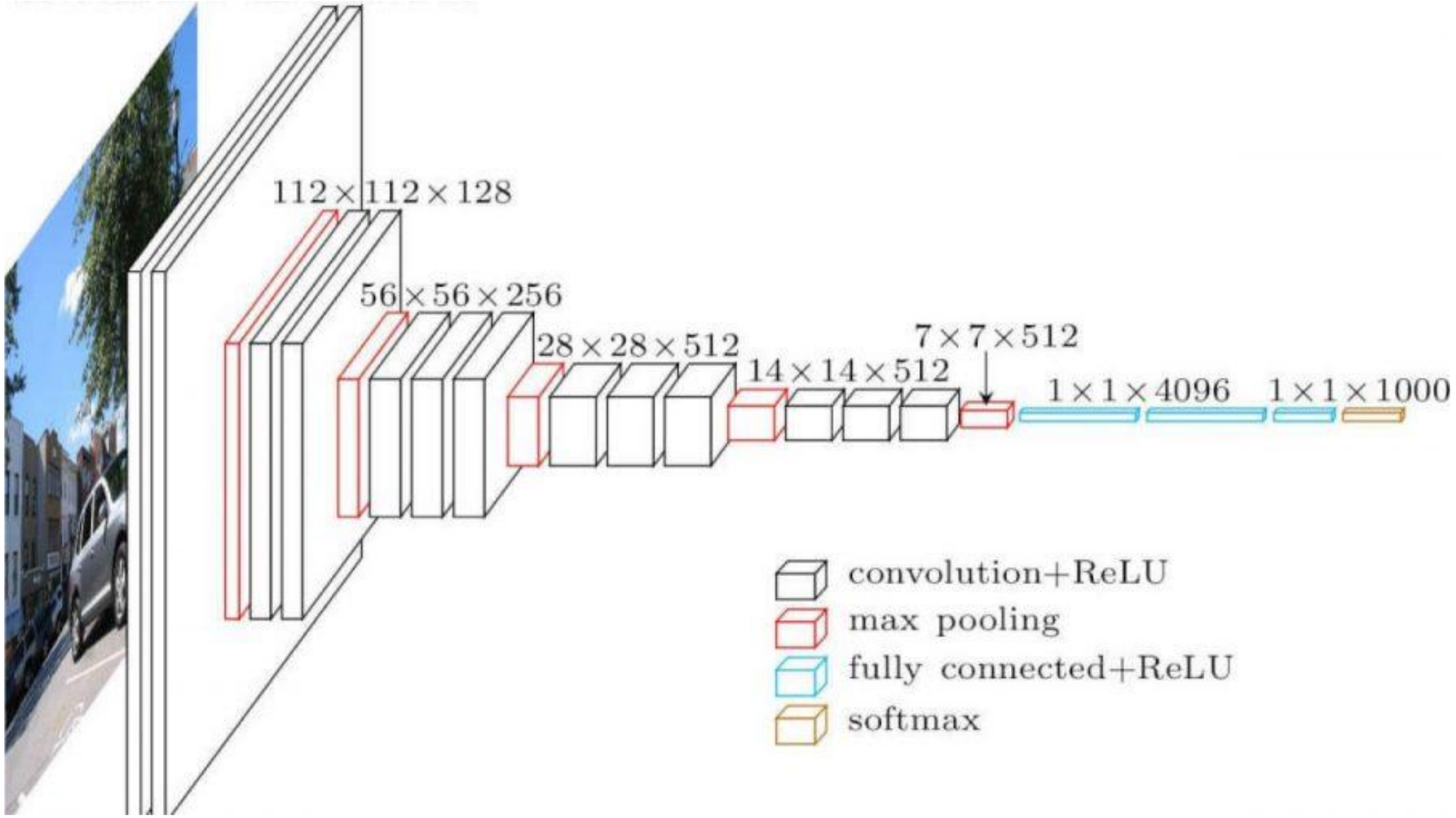
Параметры и сложность сетей



Layer Name	Tensor Size	Weights	Biases	Parameters
Input Image	227x227x3	0	0	0
Conv-1	55x55x96	34,848	96	34,944
MaxPool-1	27x27x96	0	0	0
Conv-2	27x27x256	614,400	256	614,656
MaxPool-2	13x13x256	0	0	0
Conv-3	13x13x384	884,736	384	885,120
Conv-4	13x13x384	1,327,104	384	1,327,488
Conv-5	13x13x256	884,736	256	884,992
MaxPool-3	6x6x256	0	0	0
FC-1	4096×1	37,748,736	4,096	37,752,832
FC-2	4096×1	16,777,216	4,096	16,781,312
FC-3	1000×1	4,096,000	1,000	4,097,000
Output	1000×1	0	0	0
Total				**62,378,344**

https://github.com/aws-samples/aws-machine-learning-university-accelerated-cv/tree/master/slides

VGG (2014, VGG group, Oxford)



K. Simonyan, A. Zisserman «Very Deep Convolutional Networks for Large-Scale Image Recognition» https://arxiv.org/pdf/1409.1556.pdf

тут есть ошибка;)

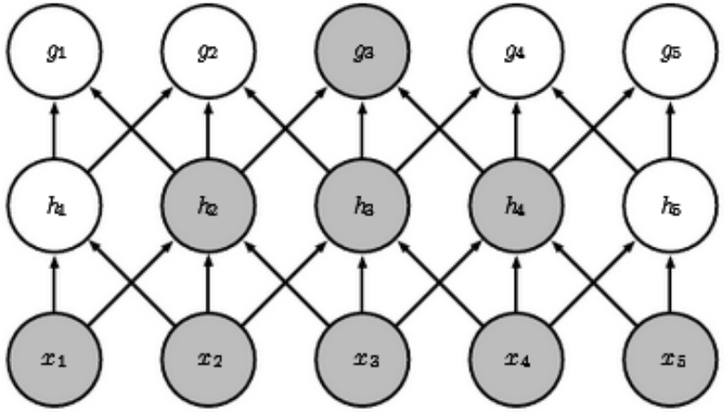
VGG (2014)

- 5 Convolutional blocks + 3 Fully Connected Layers
- вход = 256×256 (здесь 224×224)
- ReLu
- каскад 3×3 свёрток (замена 7×7)
- 138M (133-144) параметров (очень тяжеловесная)
- 3 недели 4 GPU, тоже использовали ансамбль
- большинство вычислений – в первых свёртках, большинство параметров – в конце
- batch size = 256
- momentum = 0.9
- L2-reg ~ $5 \cdot 10^{-4}$
- dropout = 0.5 (2 первых полносвязных слоя)
- LR = 10^{-2} (/ 10 после стабилизации)
- 370K итерация (74 эпох)
- более глубокие сети инициализировались по обученным простым (A-D)

можно просто «хорошую инициализацию»

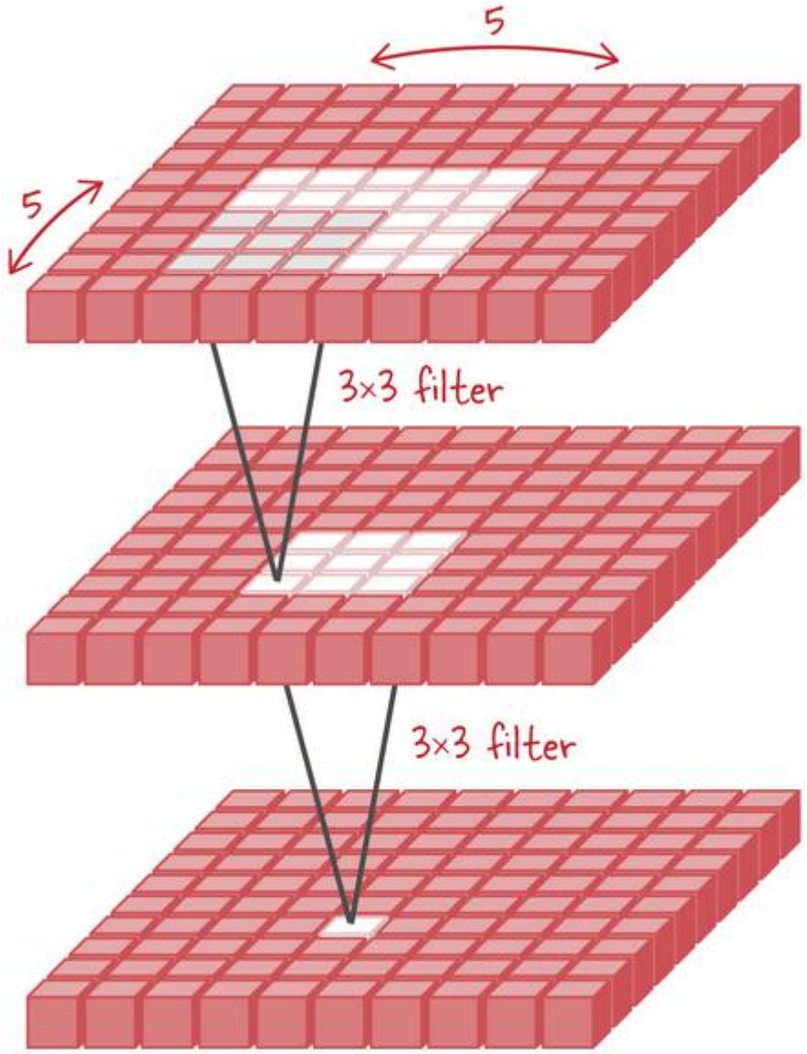
C++ Caffe

Идея каскада свёрток



«Receptive field»

- **меньше параметров**
- **быстрее**
- **дополнительные нелинейности**



[Practical Machine Learning for Computer Vision]

VGG 16 vs 19

см. колонку D и E

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	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**
maxpool					
conv3-512 conv3-512	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**
maxpool					
conv3-512 conv3-512	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**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Минутка кода: VGG

```
import torch
model = torch.hub.load('pytorch/vision:v0.6.0', 'vgg11', pretrained=True)
# or any of these variants
# model = torch.hub.load('pytorch/vision:v0.6.0', 'vgg11_bn', pretrained=True)
# model = torch.hub.load('pytorch/vision:v0.6.0', 'vgg13', pretrained=True)
# model = torch.hub.load('pytorch/vision:v0.6.0', 'vgg13_bn', pretrained=True)
# model = torch.hub.load('pytorch/vision:v0.6.0', 'vgg16', pretrained=True)
# model = torch.hub.load('pytorch/vision:v0.6.0', 'vgg16_bn', pretrained=True)
# model = torch.hub.load('pytorch/vision:v0.6.0', 'vgg19', pretrained=True)
# model = torch.hub.load('pytorch/vision:v0.6.0', 'vgg19_bn', pretrained=True)
model.eval()
```

есть готовые модели:

https://pytorch.org/docs/stable/torchvision/models.html

Минутка кода: VGG

```
class VGG(nn.Module):

    def __init__(self, features, num_classes=1000):
        super(VGG, self).__init__()
        self.features = features
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )
        self._initialize_weights()
```

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

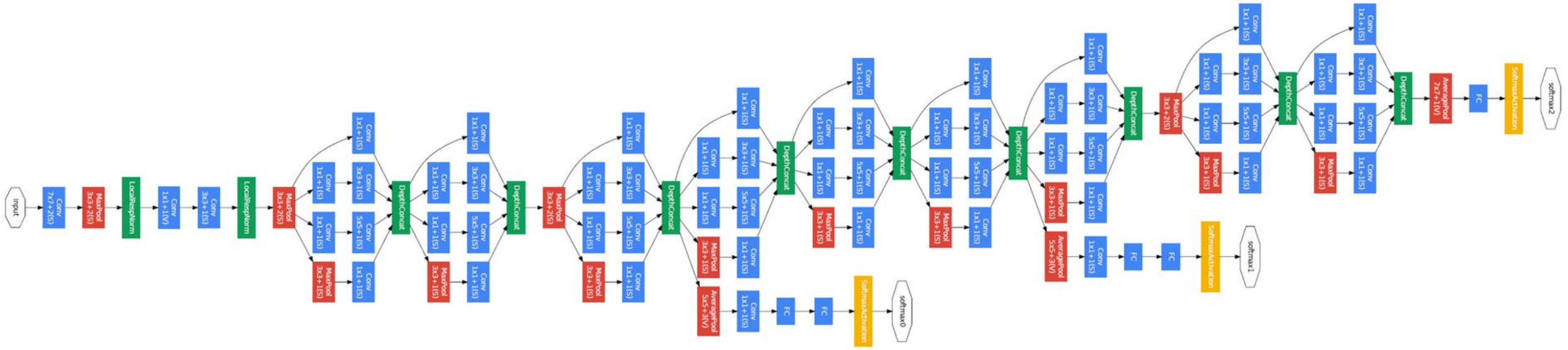
```
cfg = {
    'A': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'B': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    # ... }
```

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

def vgg11(pretrained=False, **kwargs):
    """VGG 11-layer model (configuration "A")
    Args:          pretrained (bool): If True, returns a model pre-trained on ImageNet
    """
    model = VGG(make_layers(cfg['A']), **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['vgg11']))
    return model
```

Минутка кода: VGG

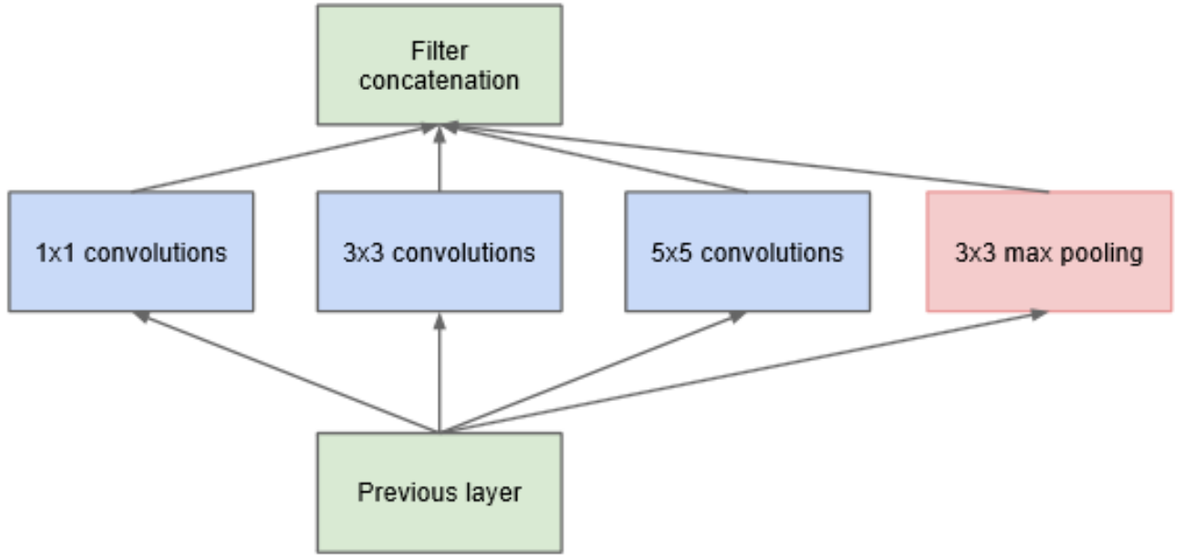
```
def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
            elif isinstance(m, nn.Linear):
                m.weight.data.normal_(0, 0.01)
                m.bias.data.zero_()
```

GoogLeNet / Inception (2014)

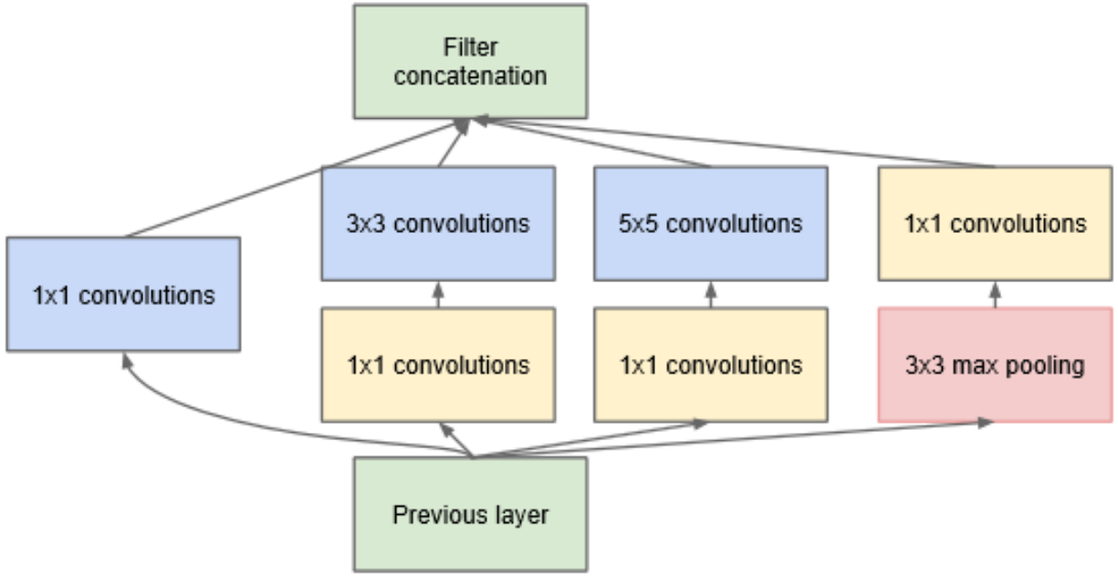


- **«конструктор» НС – «Modular Architecture»**
- **22 слоя – нет полносвязных**
- **Модуль «Inception», 1×1-свёртки,**
- **5M параметров (меньше!)**
- **дополнительные выходы классификации** (с весом 0.3 к общей ошибке)
- **тоже ансамбль (из 7)**
- **Global Average Pooling** (немного улучшает качество)

Модуль «Inception»



(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

Изначальная идея – разные свёртки + пулинг

1×1-свёртки существенно уменьшают число параметров!

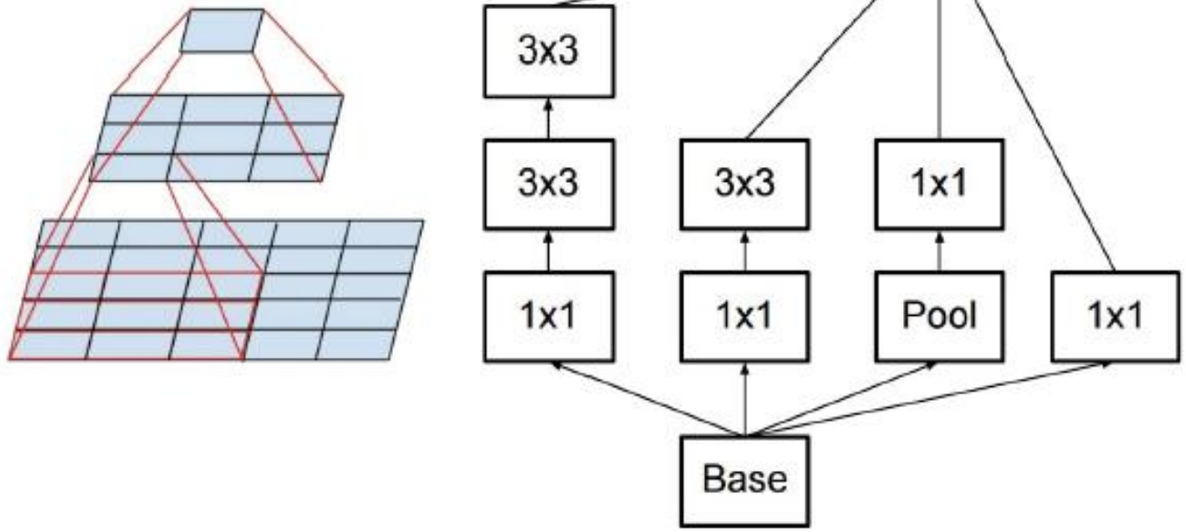
Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich «Going Deeper with Convolutions» // https://arxiv.org/abs/1409.4842

GoogLeNet / Inception (2014)

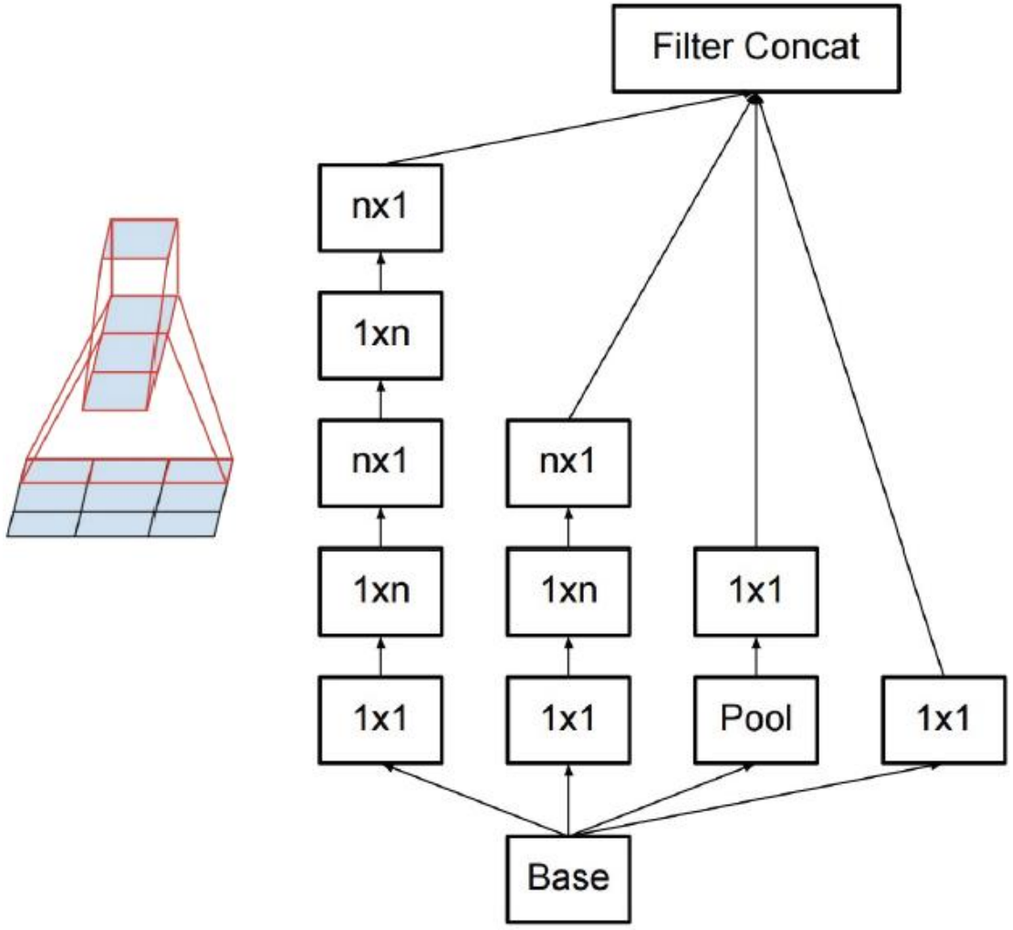
type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	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								

Inception v2, v3

Другое строение модулей (+batchnorm)



5×5 свёртку превратить в две 3×3



дальнейшая факторизация

Минутка кода: модуль Inception

```
class Inception_base(nn.Module):
    def __init__(self, depth_dim, input_size, config):
        super(Inception_base, self).__init__()
        self.depth_dim = depth_dim
        # 1x1
        self.conv1 = nn.Conv2d(input_size, out_channels=config[0][0], kernel_size=1, stride=1, padding=0)
        # 3x3_bottleneck + 3x3
        self.conv3_1 = nn.Conv2d(input_size, out_channels=config[1][0], kernel_size=1, stride=1, padding=0)
        self.conv3_3 = nn.Conv2d(config[1][0], config[1][1], kernel_size=3, stride=1, padding=1)
        # 5x5_bottleneck + 5x5
        self.conv5_1 = nn.Conv2d(input_size, out_channels=config[2][0], kernel_size=1, stride=1, padding=0)
        self.conv5_5 = nn.Conv2d(config[2][0], config[2][1], kernel_size=5, stride=1, padding=2)
        # maxpool + 1x1
        self.max_pool_1 = nn.MaxPool2d(kernel_size=config[3][0], stride=1, padding=1)
        self.conv_max_1 = nn.Conv2d(input_size, out_channels=config[3][1], kernel_size=1, stride=1,
                                    padding=0)
        self.apply(layer_init)

    def forward(self, input):
        output1 = F.relu(self.conv1(input))
        output2 = F.relu(self.conv3_1(input))
        output2 = F.relu(self.conv3_3(output2))
        output3 = F.relu(self.conv5_1(input))
        output3 = F.relu(self.conv5_5(output3))
        output4 = F.relu(self.conv_max_1(self.max_pool_1(input)))
        return torch.cat([output1, output2, output3, output4], dim=self.depth_dim)
```

https://github.com/antspy/inception_v1.pytorch

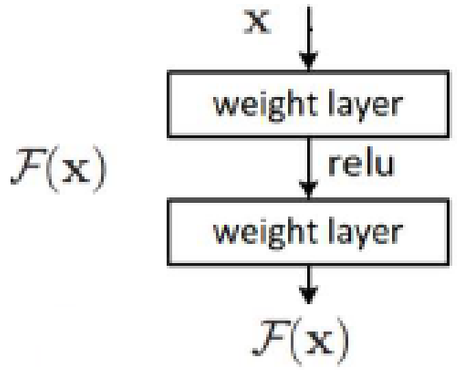
```
class Inception_v1(nn.Module):
    def __init__(self, num_classes=1000):
        super(Inception_v1, self).__init__()
        #conv2d0
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.max_pool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.lrn1 = LRN(local_size=11, alpha=0.00109999999404, beta=0.5, k=2)
        # Local Response Normalization layer - пропустили...
        #conv2d1
        self.conv2 = nn.Conv2d(64, 64, kernel_size=1, stride=1, padding=0)
        #conv2d2
        self.conv3  = nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1)
        self.lrn3 = LRN(local_size=11, alpha=0.00109999999404, beta=0.5, k=2)
        self.max_pool3 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.inception_3a = Inception_base(1, 192, [[64], [96,128], [16, 32], [3, 32]]) #3a
        self.inception_3b = Inception_base(1, 256, [[128], [128,192], [32, 96], [3, 64]]) #3b
        self.max_pool_inc3= nn.MaxPool2d(kernel_size=3, stride=2, padding=0)
        self.inception_4a = Inception_base(1, 480, [[192], [ 96,204], [16, 48], [3, 64]]) #4a
        self.inception_4b = Inception_base(1, 508, [[160], [112,224], [24, 64], [3, 64]]) #4b
        self.inception_4c = Inception_base(1, 512, [[128], [128,256], [24, 64], [3, 64]]) #4c
        self.inception_4d = Inception_base(1, 512, [[112], [144,288], [32, 64], [3, 64]]) #4d
        self.inception_4e = Inception_base(1, 528, [[256], [160,320], [32,128], [3,128]]) #4e
        self.max_pool_inc4 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.inception_5a = Inception_base(1, 832, [[256], [160,320], [48,128], [3,128]]) #5a
        self.inception_5b = Inception_base(1, 832, [[384], [192,384], [48,128], [3,128]]) #5b
        self.avg_pool5 = nn.AvgPool2d(kernel_size=7, stride=1, padding=0)
        self.dropout_layer = nn.Dropout(0.4)
        self.fc = nn.Linear(1024, num_classes)
        self.apply(layer_init)
```

```
    def forward(self, input):
        output = self.max_pool1(F.relu(self.conv1(input)))
        output = self.lrn1(output)

        output = F.relu(self.conv2(output))
        output = F.relu(self.conv3(output))
        output = self.max_pool3(self.lrn3(output))

        output = self.inception_3a(output)
        output = self.inception_3b(output)
        output = self.max_pool_inc3(output)

        output = self.inception_4a(output)
        output = self.inception_4b(output)
        output = self.inception_4c(output)
        output = self.inception_4d(output)
        output = self.inception_4e(output)
        output = self.max_pool_inc4(output)

        output = self.inception_5a(output)
        output = self.inception_5b(output)
        output = self.avg_pool5(output)

        output = output.view(-1, 1024)
        if self.fc is not None:
            output = self.dropout_layer(output)
            output = self.fc(output)
    return output
```

ResNet = Residual Network (2015)

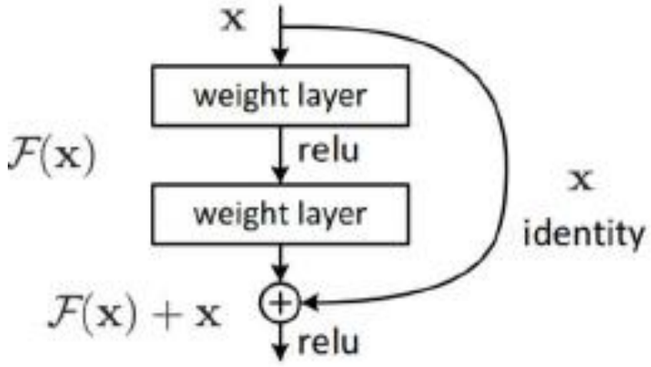
$$y = f(x)$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} f'(x)$$



$$y = f(x) + x$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} f'(x) + \frac{\partial L}{\partial y}$$



skip (shortcut) connections

упрощение реализации тождественной функции,
по крайней мере, через два слоя

Есть ещё «highway networks» – прокидывание связей с настраиваемым «гейтом»

Просто добавление слоёв не помогает!

Добавлять надо по-умному…

He et al. «Deep Residual Learning for Image Recognition» https://arxiv.org/pdf/1512.03385.pdf

ResNet = Residual Network (2015)

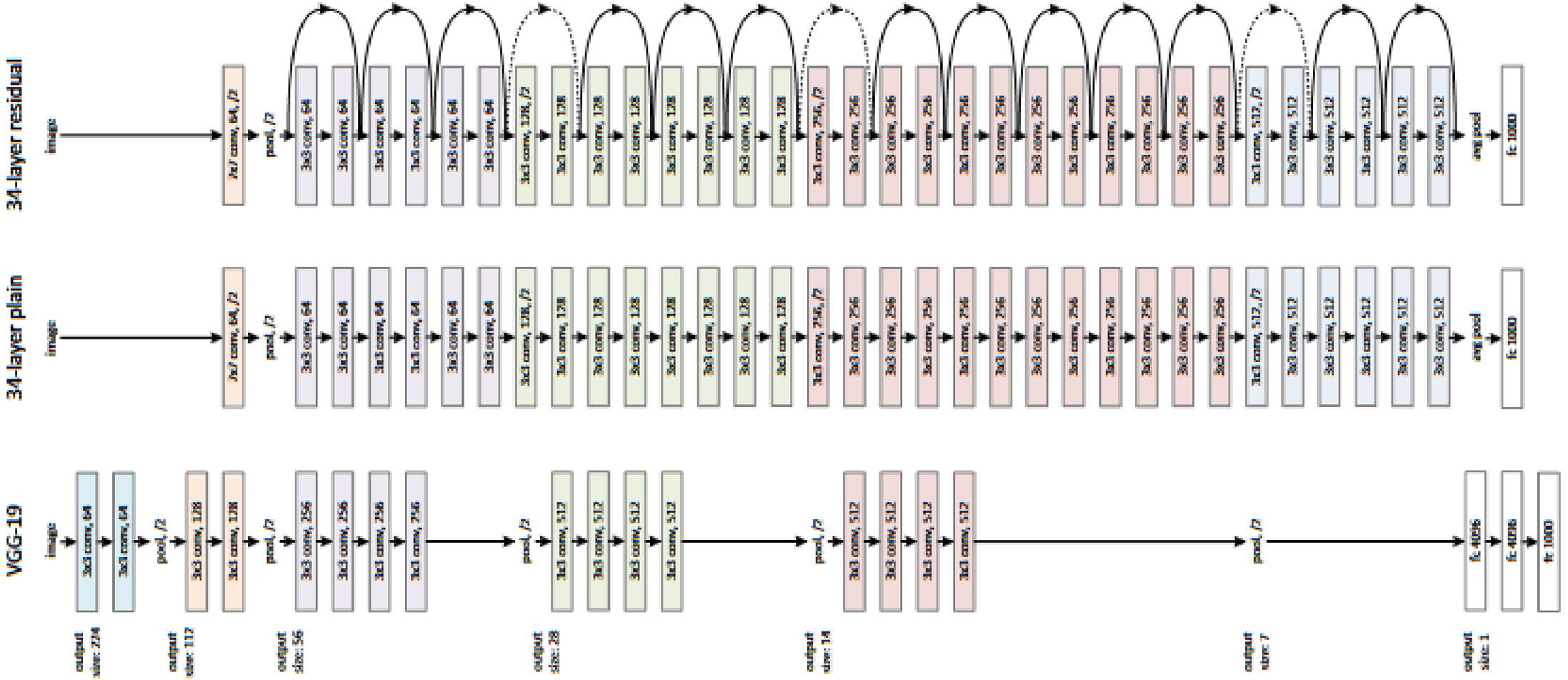


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase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

ResNet = Residual Network (2015)

- 152 слоя
- связи проходят через слои
- Batch Normalization после свёрток перед активациями (впервые)
- Умные инициализации весов
- SGD + Momentum (0.9)
- Mini-batch size = 256
- Нет Dropout!
- Average Global Pooling вместо FC-слоёв

ResNet = Residual Network (2015)

Deeper residual module (bottleneck)

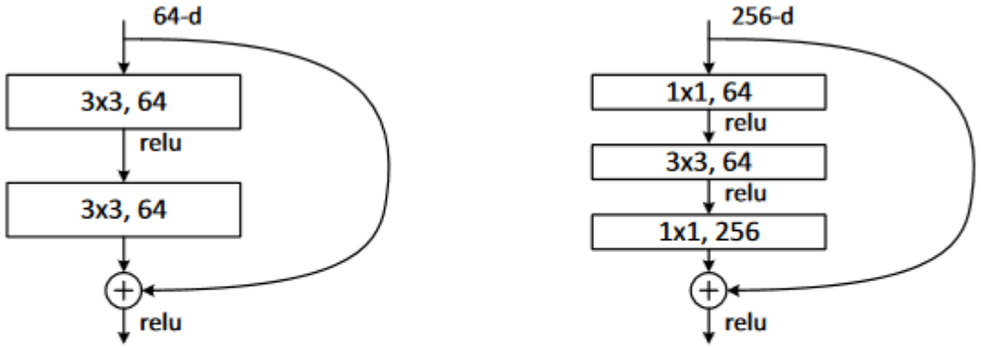
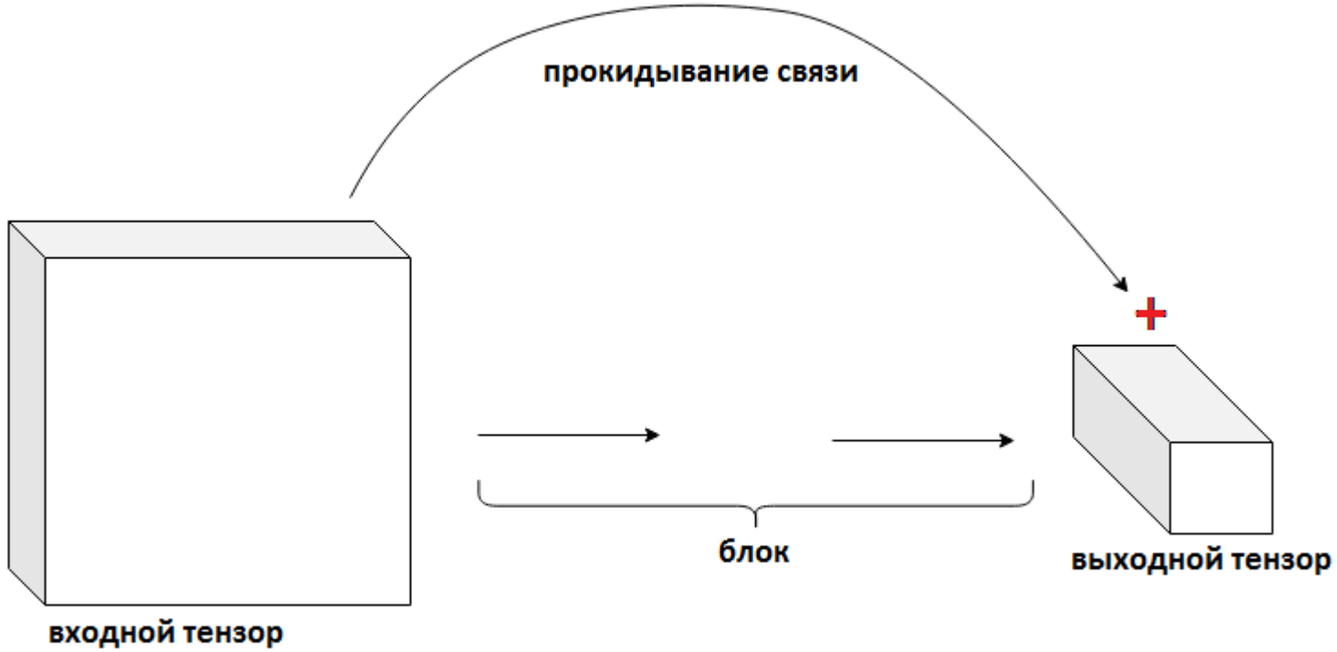


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

layer name	output size	152-layer
conv1	112×112	7×7, 64, stride 2
conv2_x	56×56	3×3 max pool, stride 2 $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax
FLOPs		11.3×10^9

Проблемы с прокидыванием в свёрточных слоях

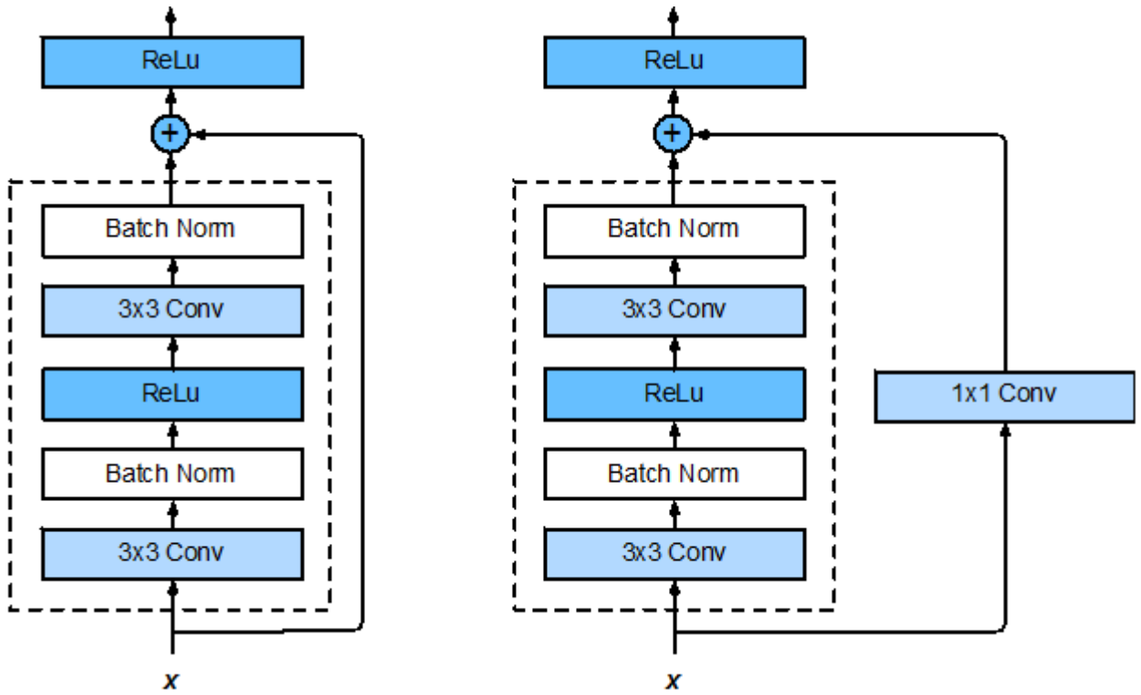


не совсем прямая связь:

1) размеры уменьшаются

2) число каналов может уменьшаться (в некоторых архитектурах)

Стандартное прокидывание и прокидывание с понижением размеров



не / использование свёртки при прокидывании, в коде – свёртка + BN:

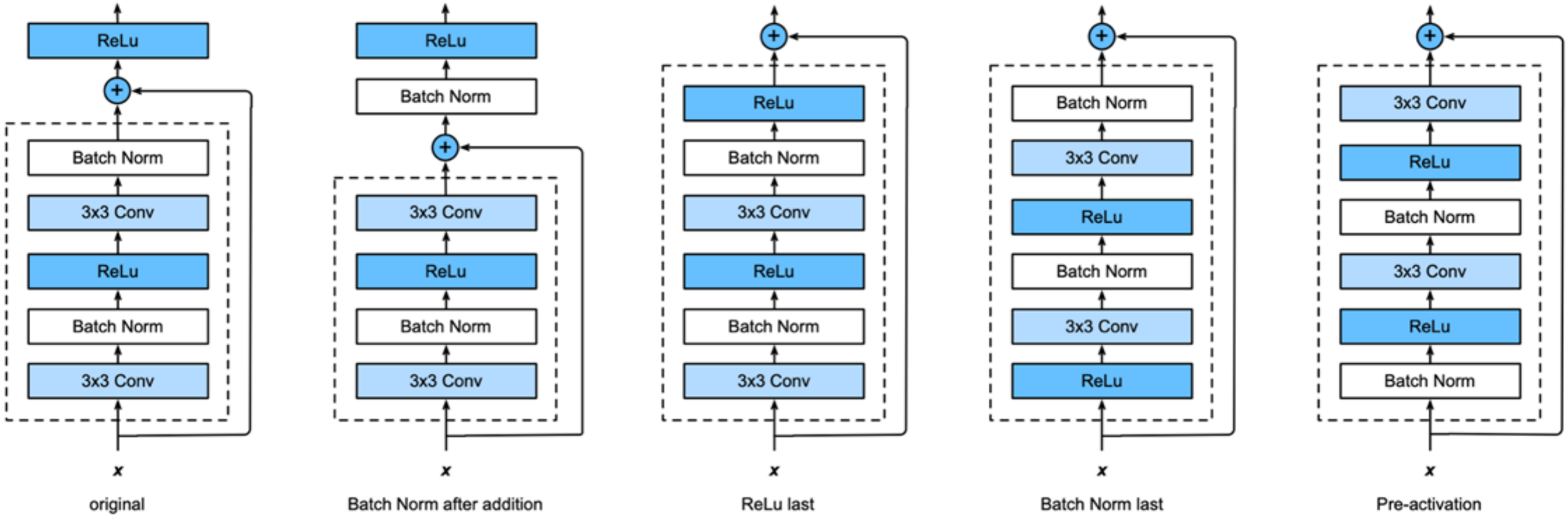
```
if stride != 1 or self.inplanes != planes * block.expansion:
    downsample = nn.Sequential(nn.Conv2d(self.inplanes, planes * block.expansion,
                                         kernel_size=1, stride=stride, bias=False),
                               nn.BatchNorm2d(planes * block.expansion))
```

Минутка кода: ResNet (остальное смотреть по ссылке!)

```
class BasicBlock(nn.Module): # building block ResNet 34 - не bottleneck
    expansion = 1
    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride
    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

https://chsasank.github.io/vision/_modules/torchvision/models/resnet.html

ResNet: рекорды по числу слоёв



Вообще, можно по-разному реализовывать идею прокидывания

https://github.com/aws-samples/aws-machine-learning-university-accelerated-cv/tree/master/slides

ResNet: рекорды по числу слоёв (1001)

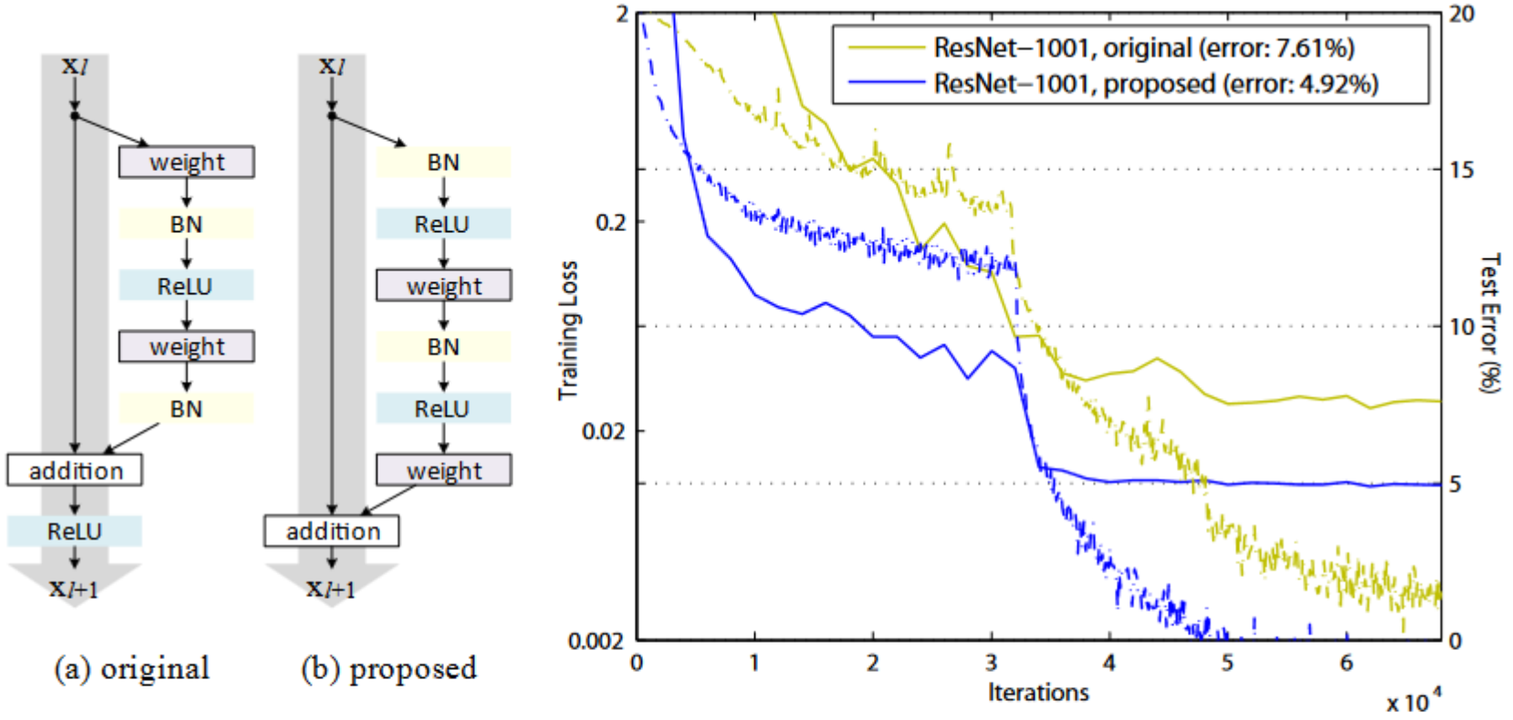
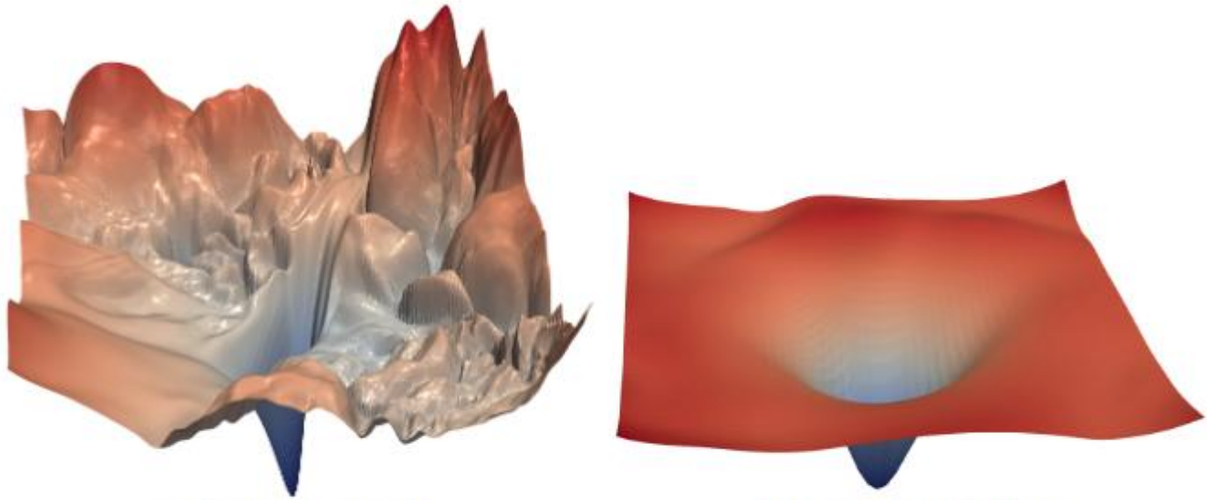


Figure 1. Left: (a) original Residual Unit in [1]; (b) proposed Residual Unit. The grey arrows indicate the easiest paths for the information to propagate, corresponding to the additive term "$\mathbf{x}_l$" in Eqn.(4) (forward propagation) and the additive term "1" in Eqn.(5) (backward propagation). **Right**: training curves on CIFAR-10 of **1001-layer** ResNets. Solid lines denote test error (y-axis on the right), and dashed lines denote training loss (y-axis on the left). The proposed unit makes ResNet-1001 easier to train.

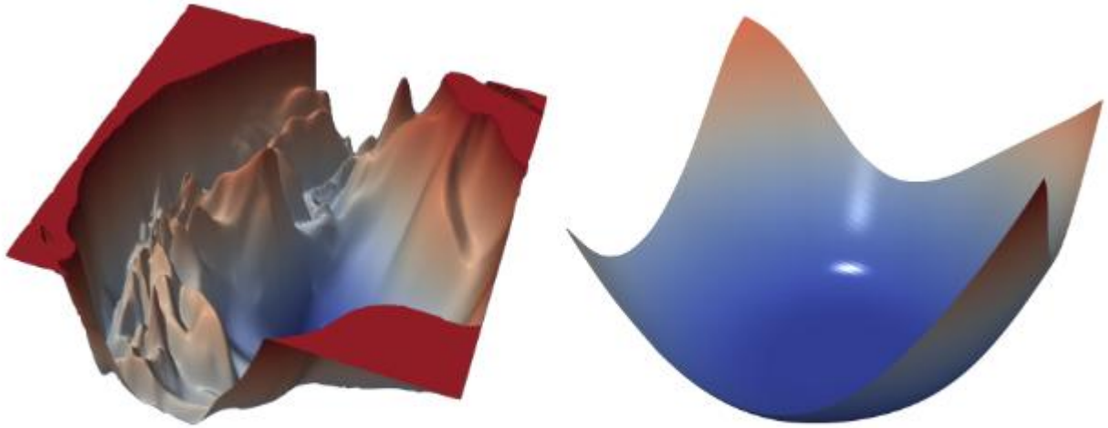
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun «Identity Mappings in Deep Residual Networks» https://arxiv.org/abs/1603.05027

Эффект прокидывания связей



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.



(a) ResNet-110, no skip connections (b) DenseNet, 121 layers

Figure 4: The loss surfaces of ResNet-110-noshort and DenseNet for CIFAR-10.

глубина и ширина «улучшают» поверхность функции ошибки

правильная оптимизация позволяет «правильно» идти по поверхности

«Visualizing the Loss Landscape of Neural Nets» https://arxiv.org/abs/1712.09913

Highway Net

$$y = H(x, W_H) \cdot T(x, W_T) + x \cdot (1 - T(x, W_T))$$

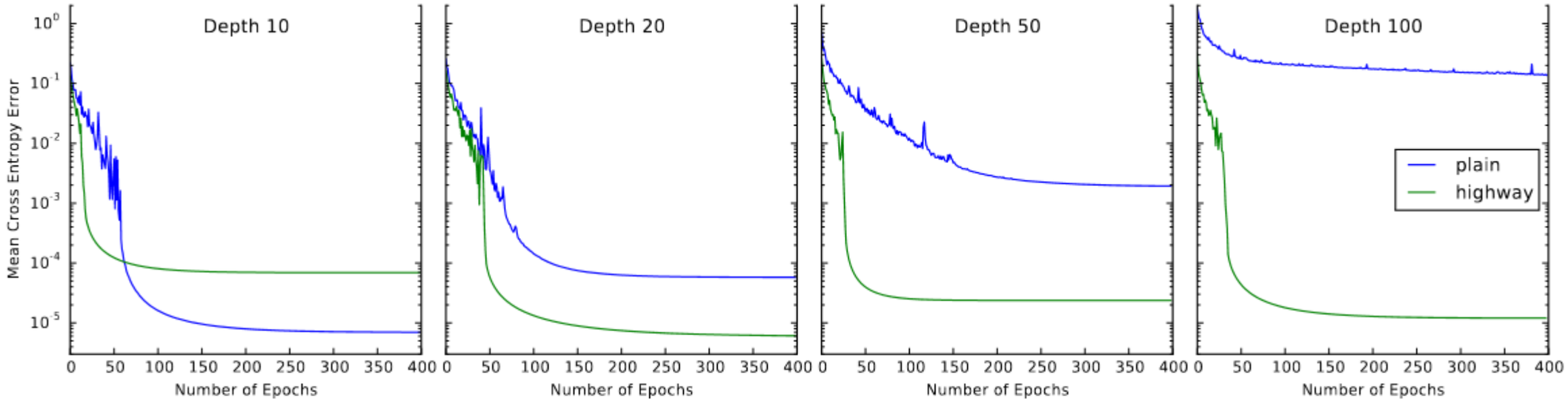
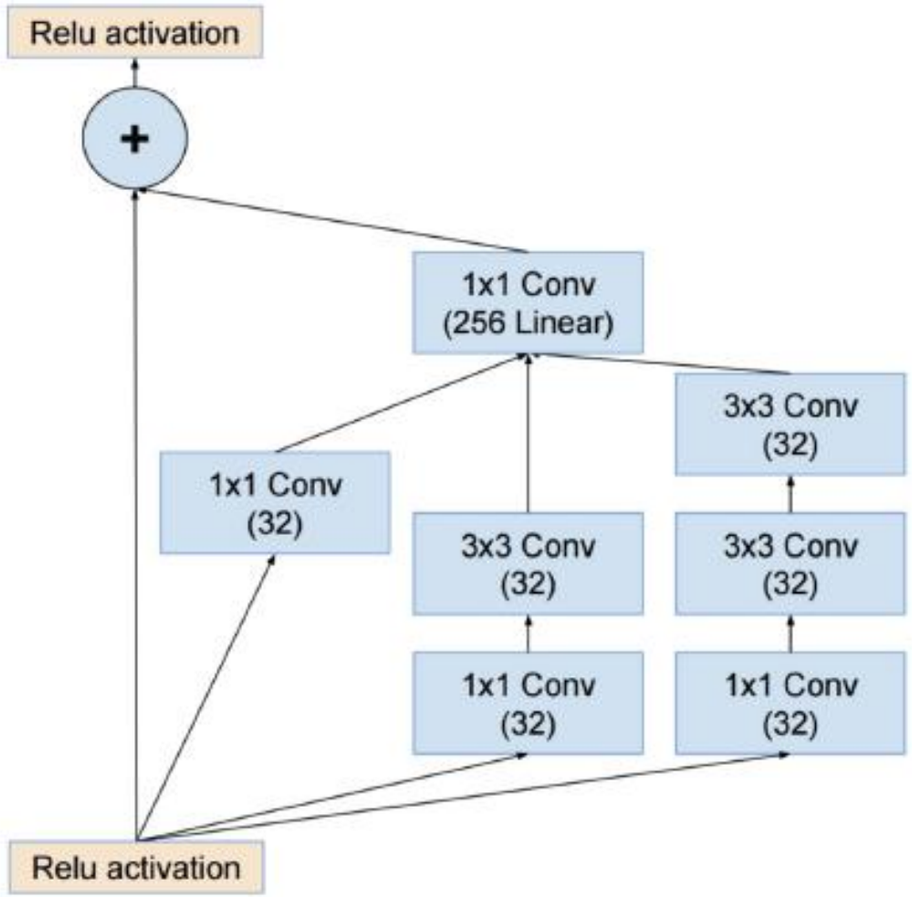


Figure 1. Comparison of optimization of plain networks and highway networks of various depths. All networks were optimized using SGD with momentum. The curves shown are for the best hyperparameter settings obtained for each configuration using a random search. Plain networks become much harder to optimize with increasing depth, while highway networks with up to 100 layers can still be optimized well.

Rupesh Kumar Srivastava, Klaus Greff, Jürgen Schmidhuber «Highway Networks» // https://arxiv.org/abs/1505.00387

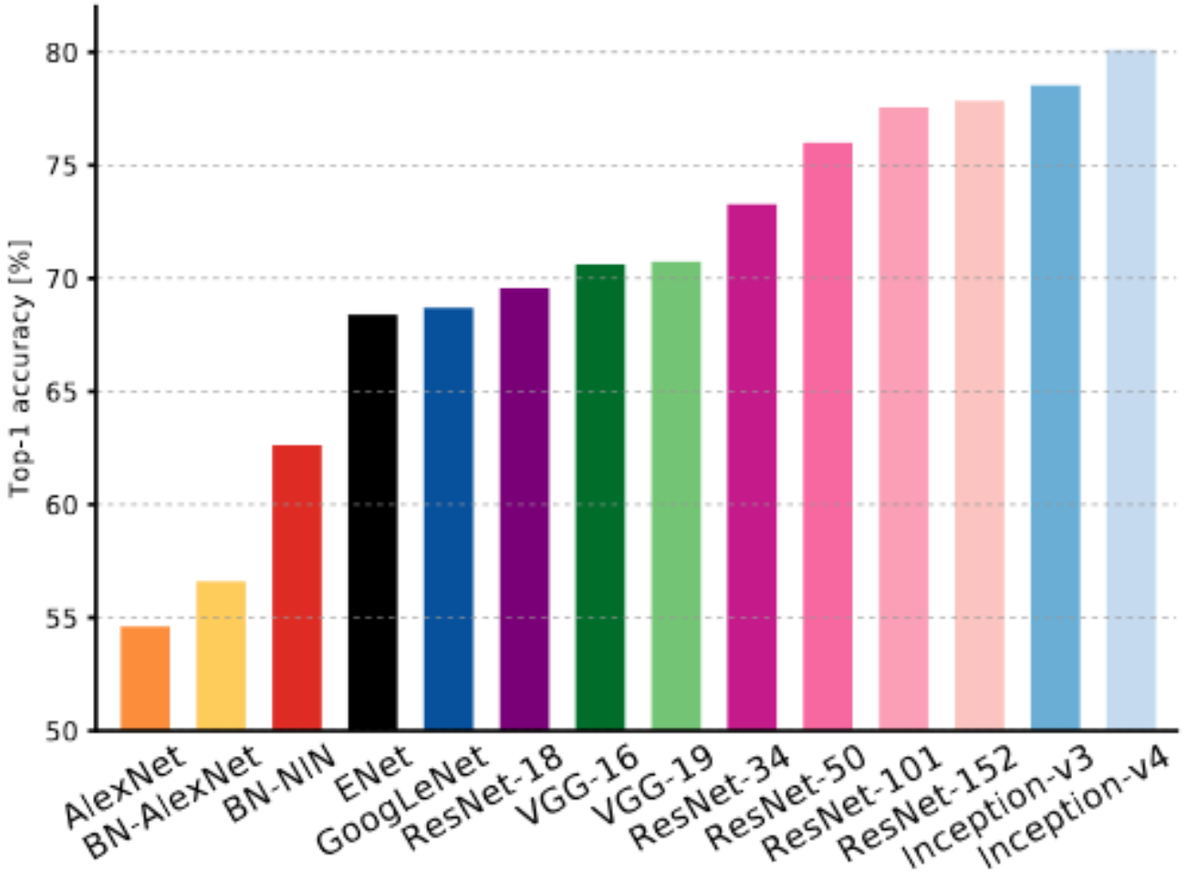
Inception-v4 (Inception-Resnet)

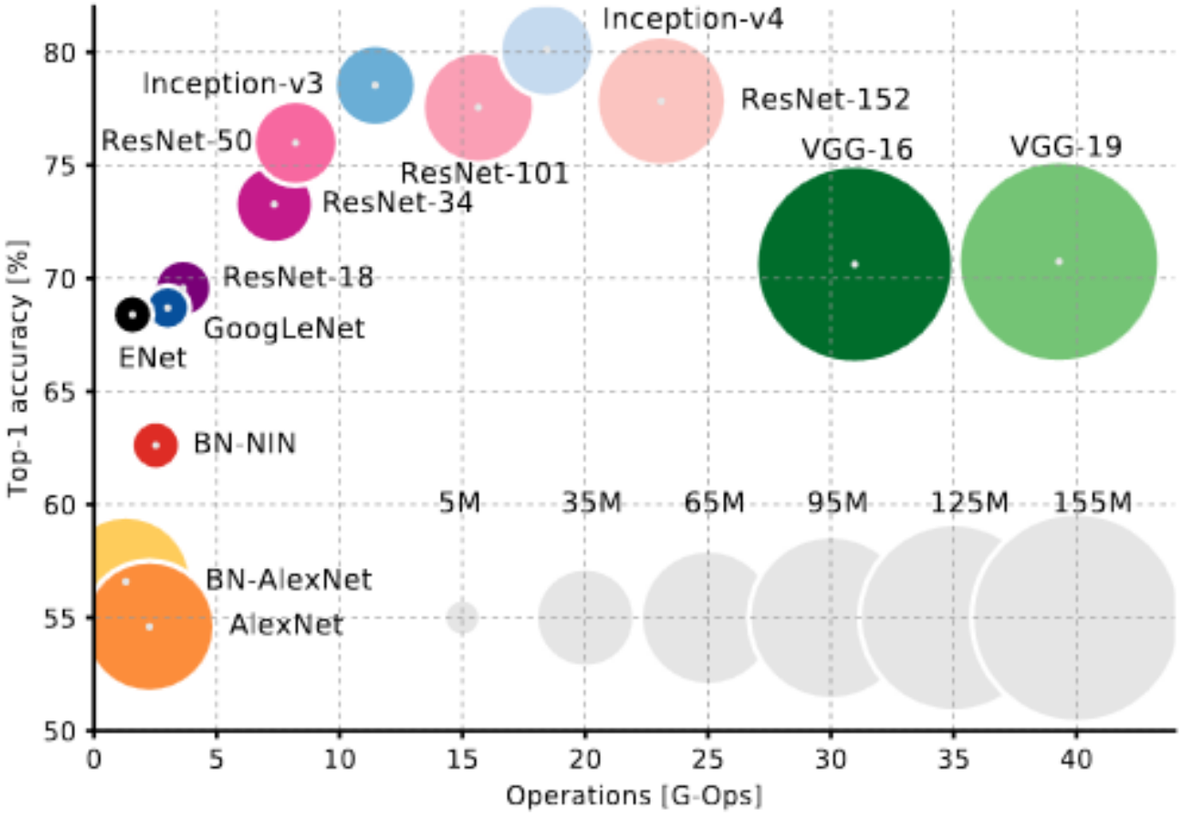
совмещение двух (даже больше) идей:



Szegedy, Christian, et al. "Inception-v4, inception-resnet and the impact of residual connections on learning." arXiv preprint arXiv:1602.07261 (2016).

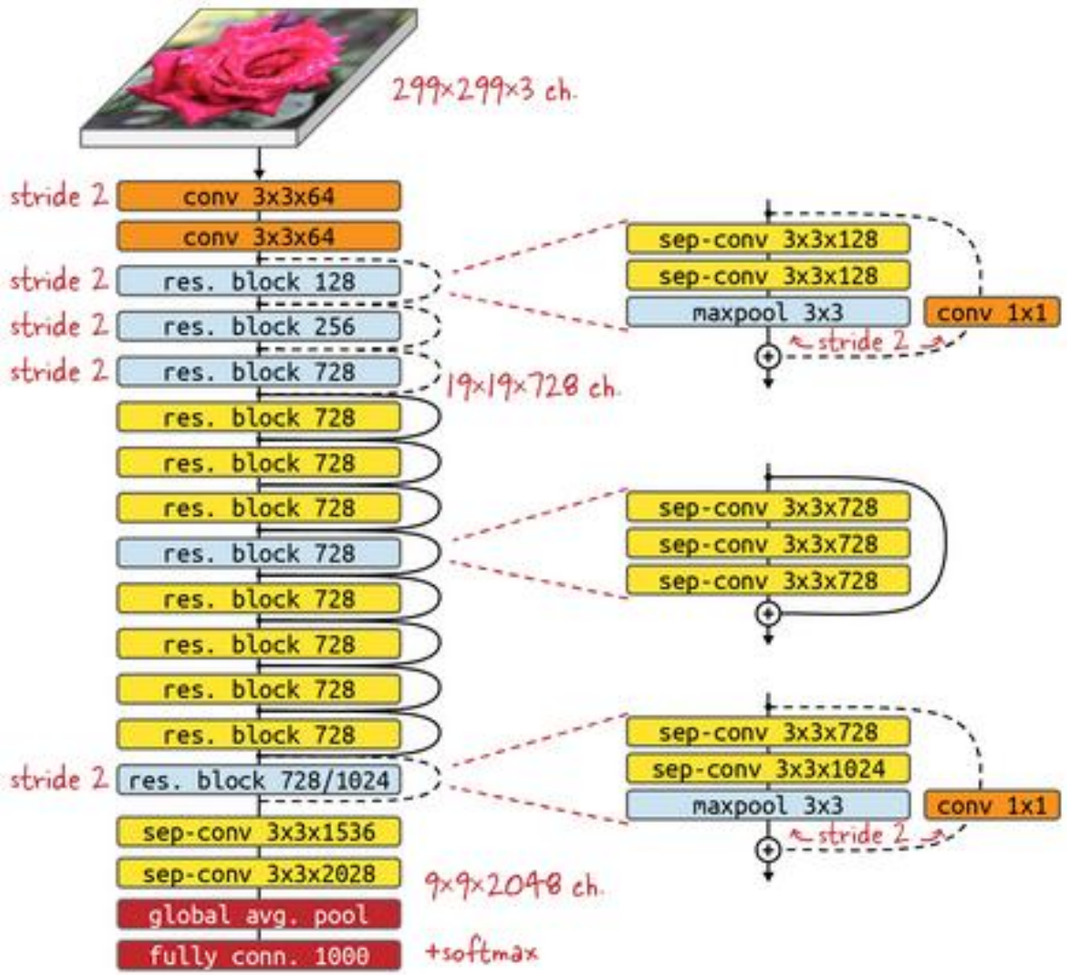
Inception-v4 (Inception-Resnet)





A. Canziani, A. Paszke, E. Culurciello, « An Analysis of Deep Neural Network Models for Practical Applications», 2017 https://arxiv.org/pdf/1605.07678.pdf

Xception = separable convolutions + ResNet

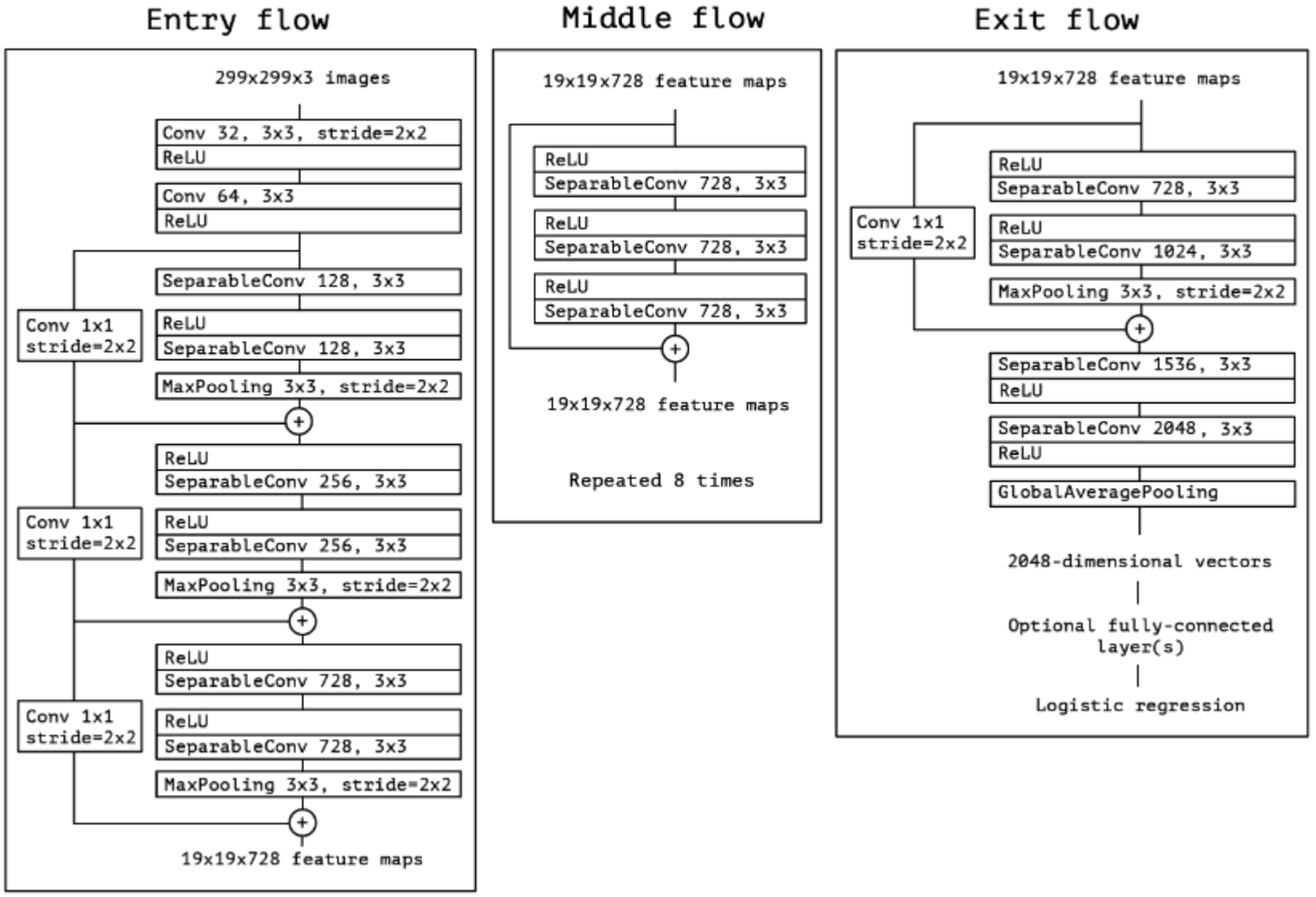


[Practical Machine Learning for Computer Vision]

François Chollet «Xception: Deep Learning with Depthwise Separable Convolutions» // https://arxiv.org/abs/1610.02357

Xception

Figure 5. The Xception architecture: the data first goes through the entry flow, then through the middle flow which is repeated eight times, and finally through the exit flow. Note that all Convolution and SeparableConvolution layers are followed by batch normalization [7] (not included in the diagram). All SeparableConvolution layers use a depth multiplier of 1 (no depth expansion).



Xception

Table 1. Classification performance comparison on ImageNet (single crop, single model). VGG-16 and ResNet-152 numbers are only included as a reminder. The version of Inception V3 being benchmarked does not include the auxiliary tower.

	Top-1 accuracy	Top-5 accuracy
VGG-16	0.715	0.901
ResNet-152	0.770	0.933
Inception V3	0.782	0.941
Xception	**0.790**	**0.945**

Figure 6. Training profile on ImageNet

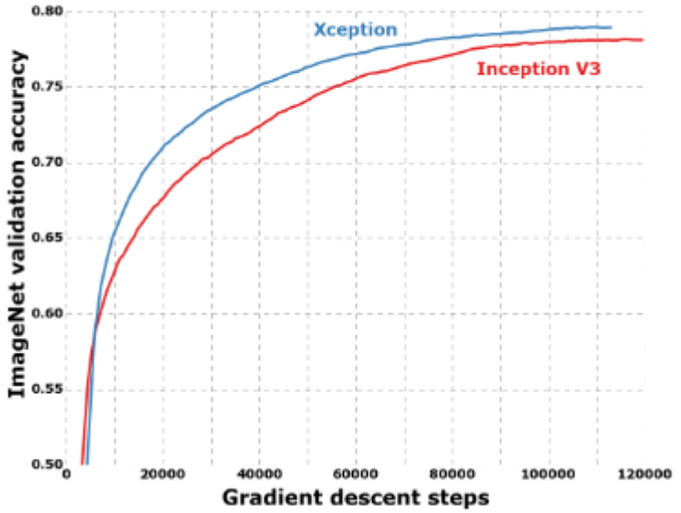
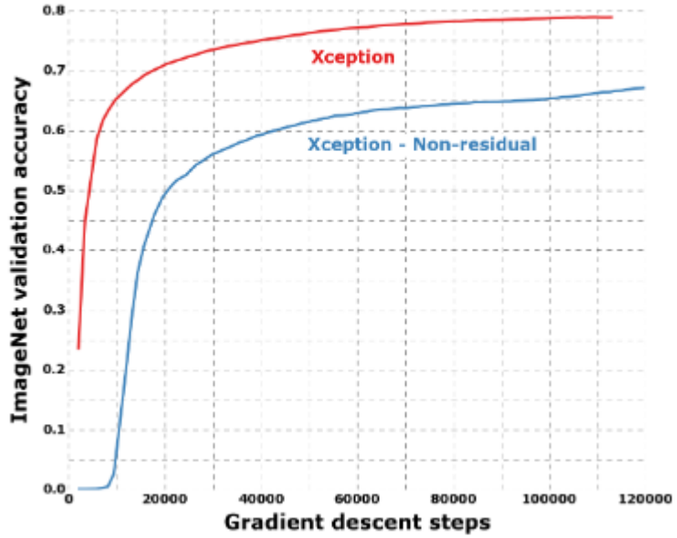


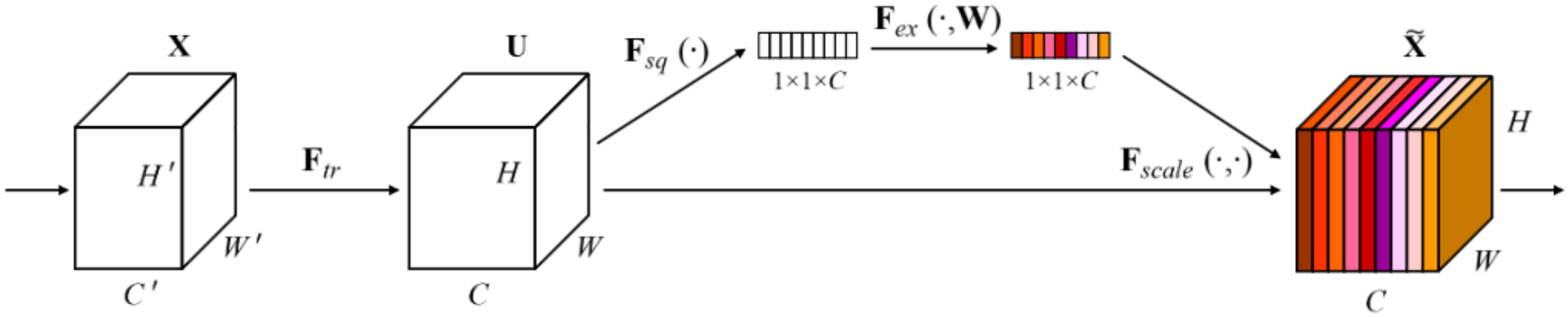
Figure 9. Training profile with and without residual connections.



SENet (Squeeze-and-Excitation Network, 2017)

обычно в CNN трансформация $F_{\text{tr}} : X_{H' \times W' \times C'} \to U_{H \times W \times C}$ **(например, свёртка)**

теперь добавим «Squeeze-and-Excitation» (SE) block $F_{\text{SE}} : U_{H \times W \times C} \to \tilde{X}_{H \times W \times C}$



не меняет размеры тензора,
но проводит адаптивную перекалибровку каналов (= признаков)

J. Hu и др. «Squeeze-and-Excitation Networks», 2018 https://arxiv.org/pdf/1709.01507.pdf

SENet (Squeeze-and-Excitation Network, 2017)

сжатие (squeeze) – агрегация по каналам (Global pooling):

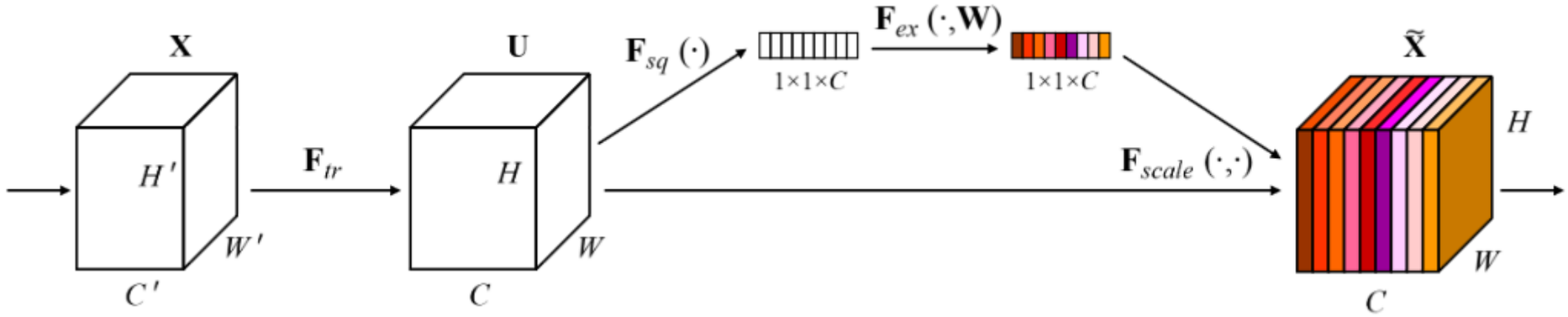
$$F_{\text{sq}} : \| u_{h,w,c} \|_{H\times W\times C} \to \left\| \frac{1}{HW} \sum_{w=1}^{W} \sum_{h=1}^{H} u_{h,w,c} \right\|_C$$

возбуждение (excitation) – подготовка коэффициентов (FC + ReLu + FC + Sigmoid):

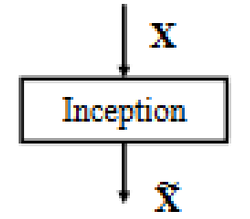
$$F_{\text{ex}} = \sigma(W_{C\times k}\, \text{ReLu}(V_{k\times C} z_C))$$

калибровка (Scale) – умножение коэффициентов на каналы

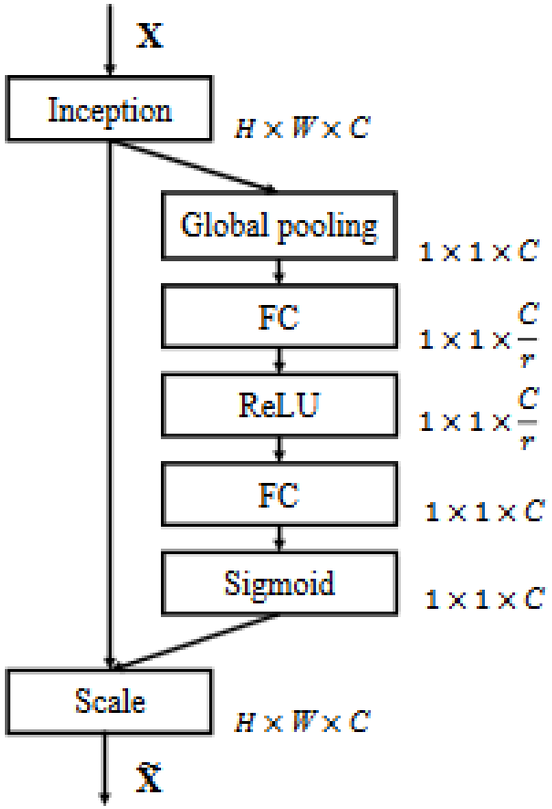
$$F_{\text{scale}} : \| u_{h,w,c} \|_{H\times W\times C} \to \| u_{h,w,c} F_{\text{ex}}(z)_c \|_C$$



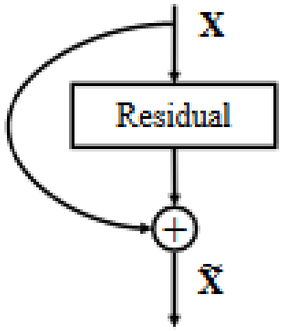
SENet: можно переделать «старые сети»



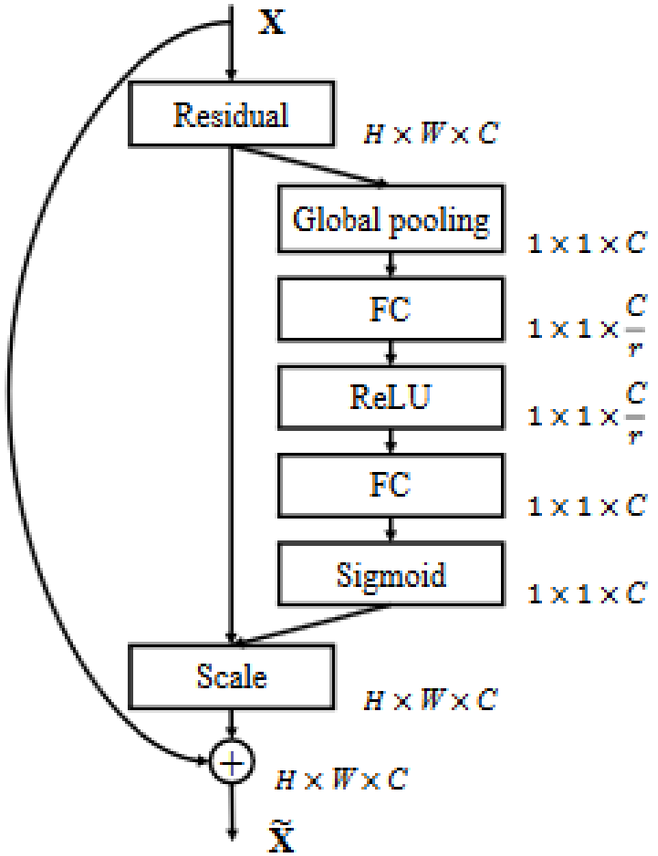
Inception Module



SE-Inception Module



ResNet Module



SE-ResNet Module

Динамическая перекалибровка признаков позволяет «увеличивать» важные признаки и «уменьшать» неважные

SENet (Squeeze-and-Excitation Network, 2017)

SE-блок – пример самовнимания (self-attention)

SE-блок – пример применения «узкого горлышка»

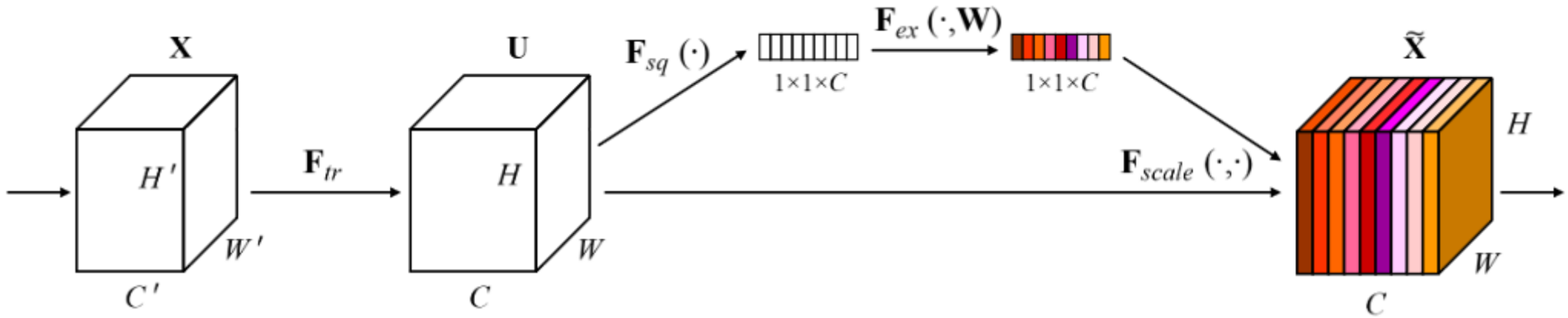
увеличивает функциональную выразимость сети (representational power)

позволяет любой архитектуре CNN использовать «межканальные зависимости»

автоматически получили селекцию каналов (примерно как в LASSO)

Минутка кода: SENet

```
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(nn.Linear(channel, channel // reduction, bias=False),
                                nn.ReLU(inplace=True),
                                nn.Linear(channel // reduction, channel, bias=False),
                                nn.Sigmoid())

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)
```



Сравнение архитектур https://arxiv.org/pdf/1810.00736.pdf

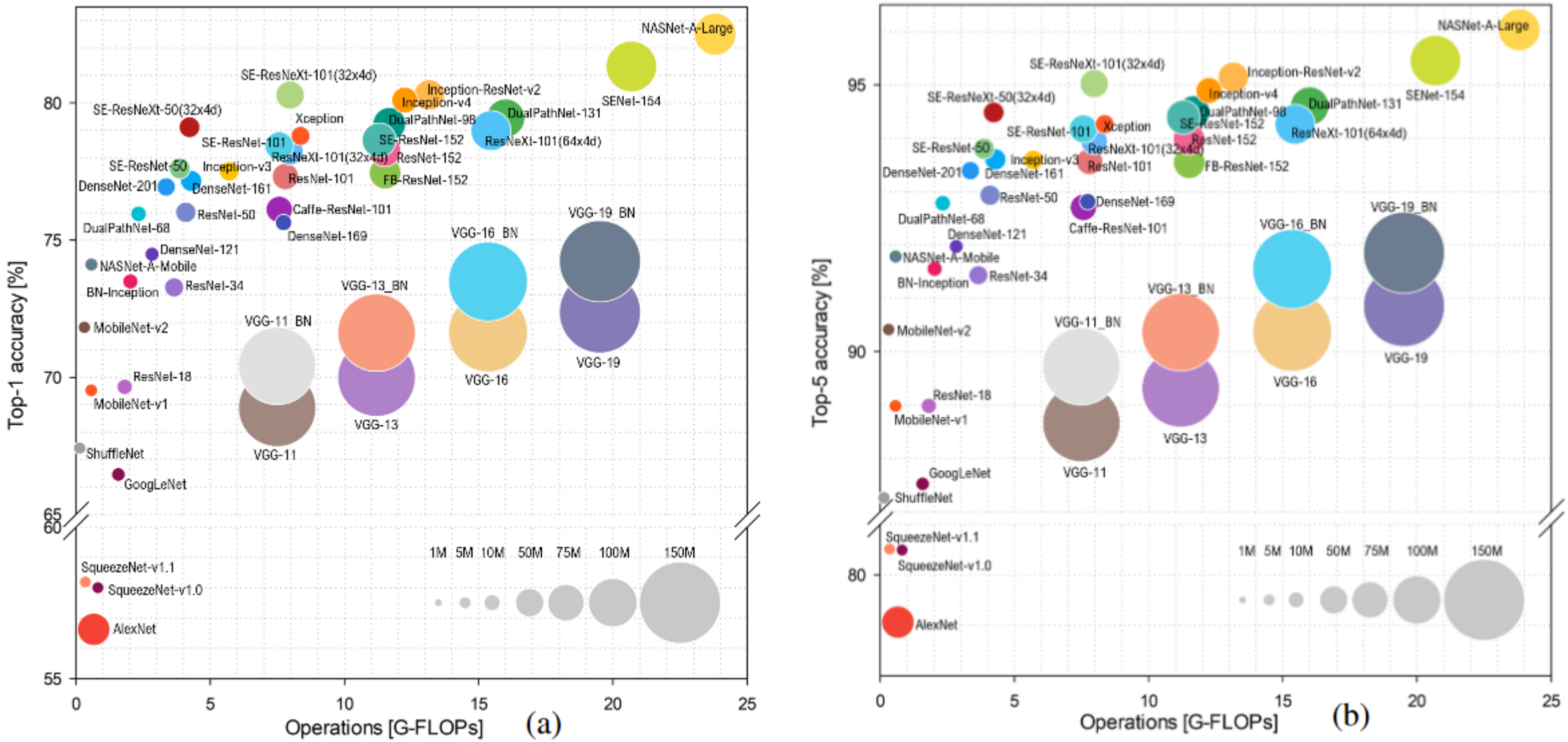
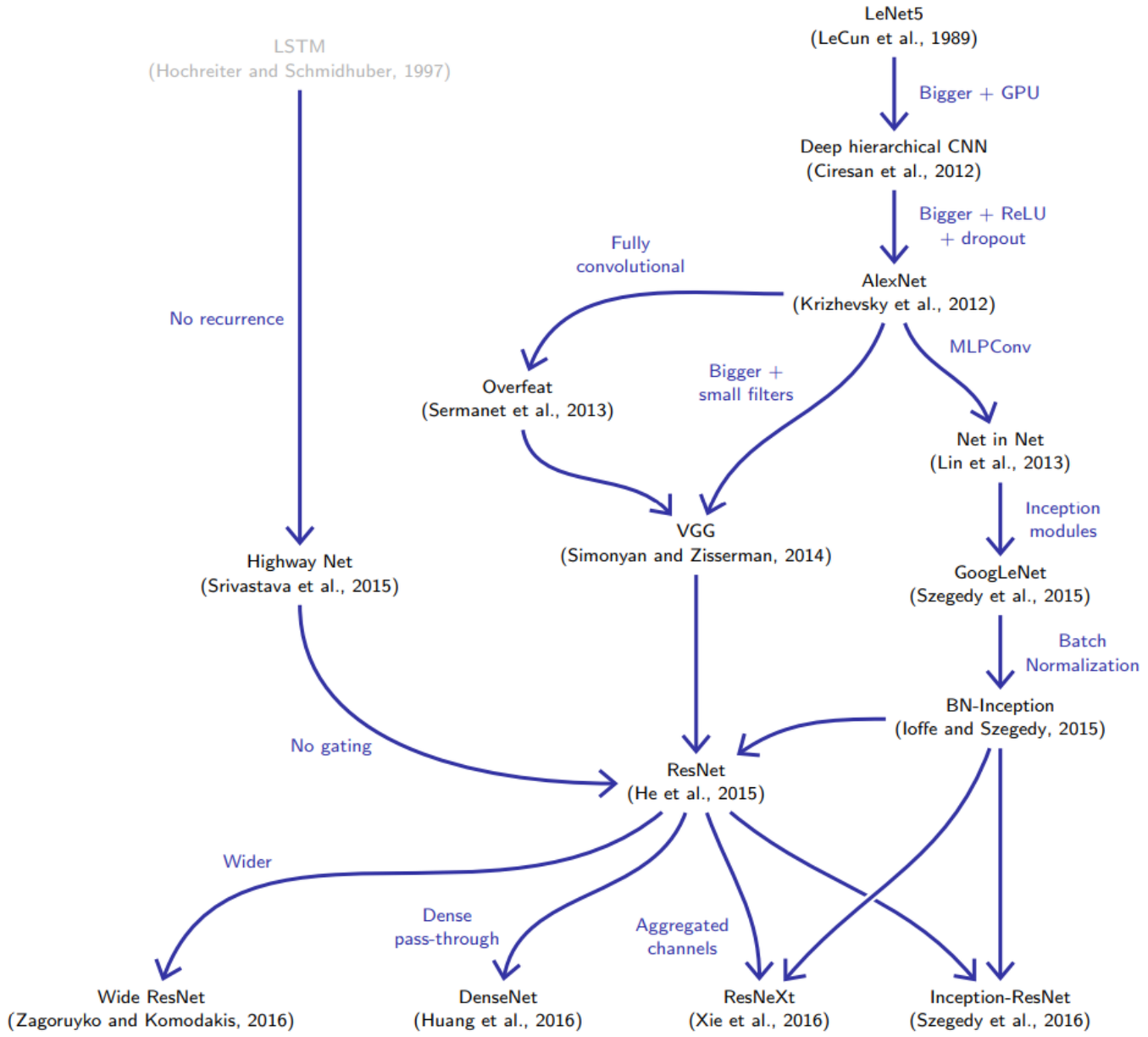
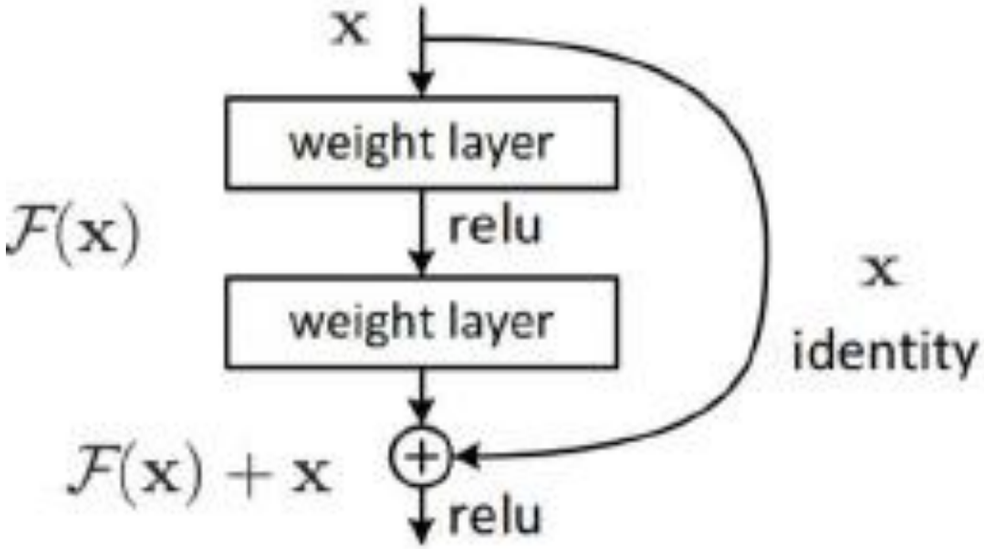


FIGURE 1: Ball chart reporting the Top-1 and Top-5 accuracy *vs.* computational complexity. Top-1 and Top-5 accuracy using only the center crop versus floating-point operations (FLOPs) required for a single forward pass are reported. The size of each ball corresponds to the model complexity. (a) Top-1; (b) Top-5.



https://fleuret.org/ee559/ee559-slides-7-2-image-classification.pdf

ResNet: почему работает



сейчас разберёмся...

ResNet: почему работает

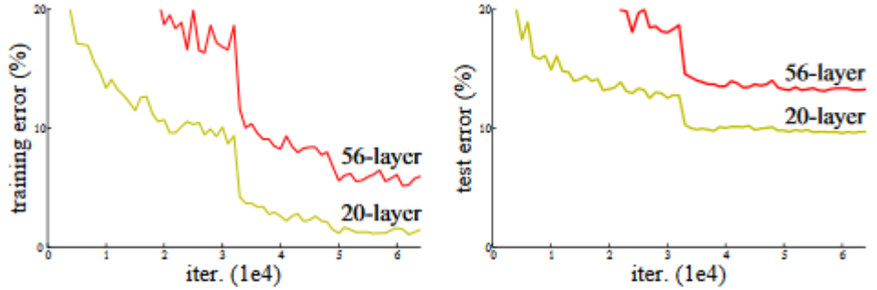
Рассмотрим сети без прокидывания связей, по числу слоёв: plain-18 (лучше!) и plain-34

layer name	output size	18-layer	34-layer
conv1	112×112		
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$

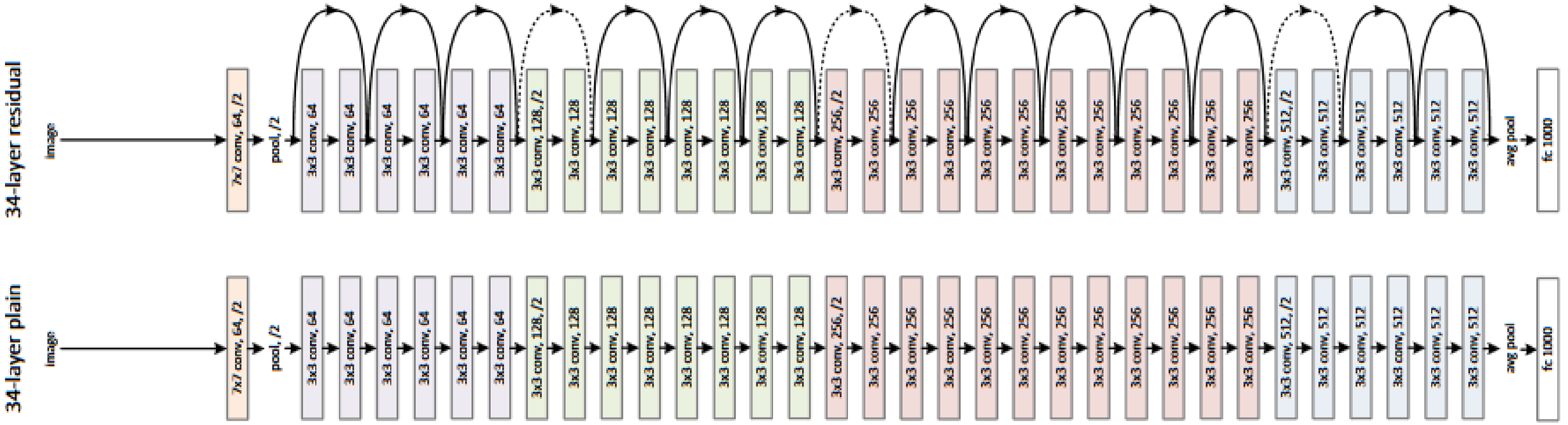
Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun «Deep Residual Learning for Image Recognition» // CVPR 2016, https://arxiv.org/abs/1512.03385

ResNet: почему работает

Причины, почему plain-18 лучше

Исчезающие градиенты (Vanishing Gradients) **Нет**	«We argue that this optimization difficulty is unlikely to be caused by vanishing gradients. These plain networks are trained with BN, which ensures forward propagated signals to have non-zero variances. We also verify that the backward propagated gradients exhibit healthy norms with BN. So neither forward nor backward signals vanish»
Переобучение (Overfitting) **Нет**	 Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.
Функциональная мощность (Representation power) **нет**	

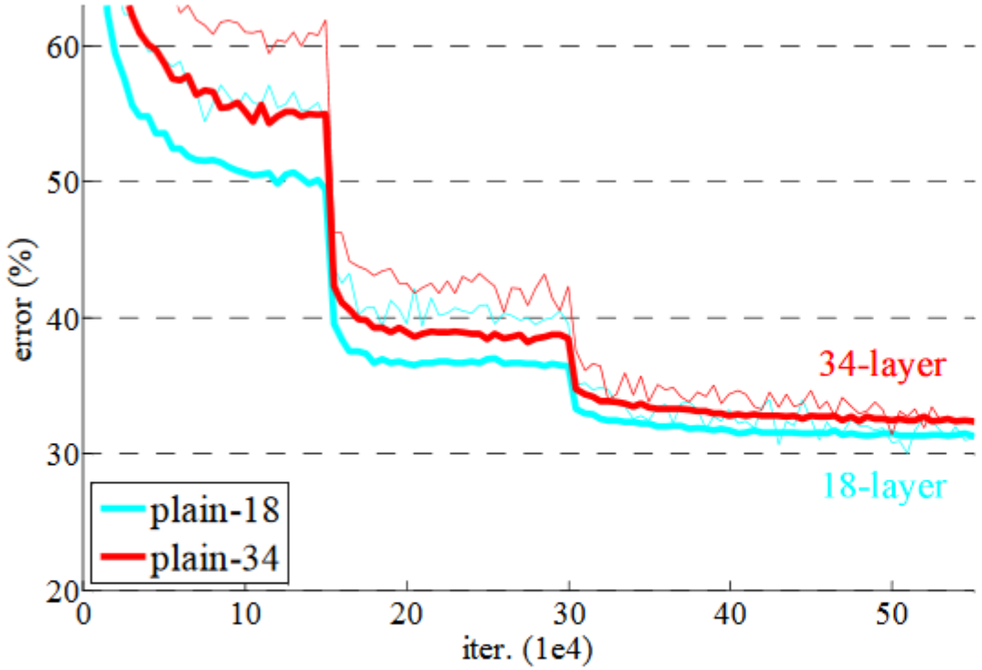
ResNet: почему работает



Сделали прокидывание связей – и ситуация изменилась
(глубокие сети лучше)

посмотрите, как соотносится картинка и предыдущая табличка

ResNet: почему работает



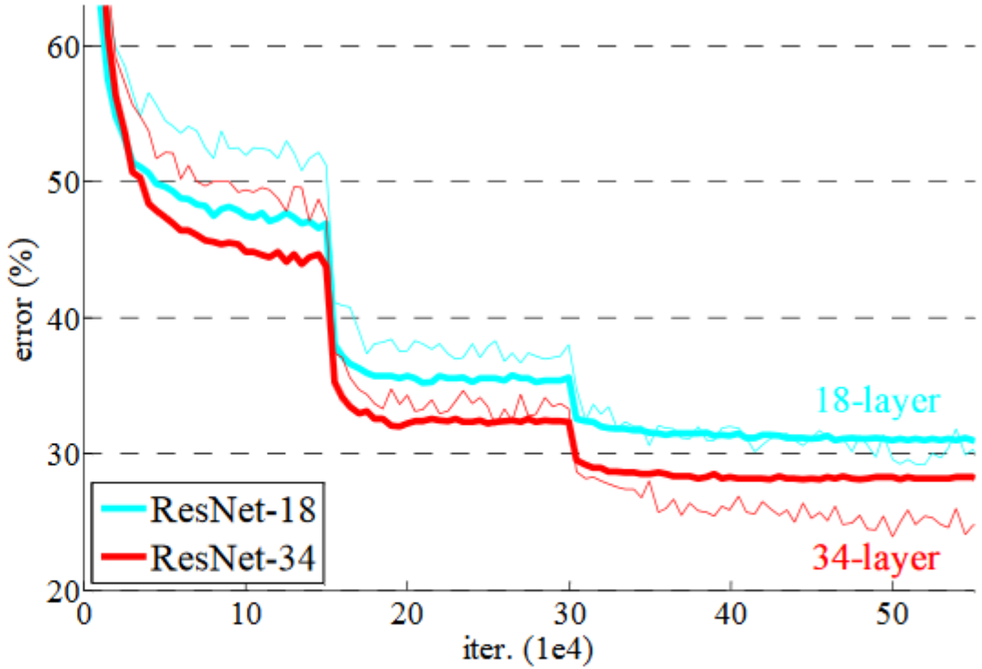


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Сделали прокидывание связей – и ситуация изменилась
(глубокие сети лучше)

	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

Table 2. Top-1 error (%, 10-crop testing) on ImageNet validation. Here the ResNets have no extra parameter compared to their plain counterparts. Fig. 4 shows the training procedures.

ResNet: почему работает

Меньше разброс в активациях

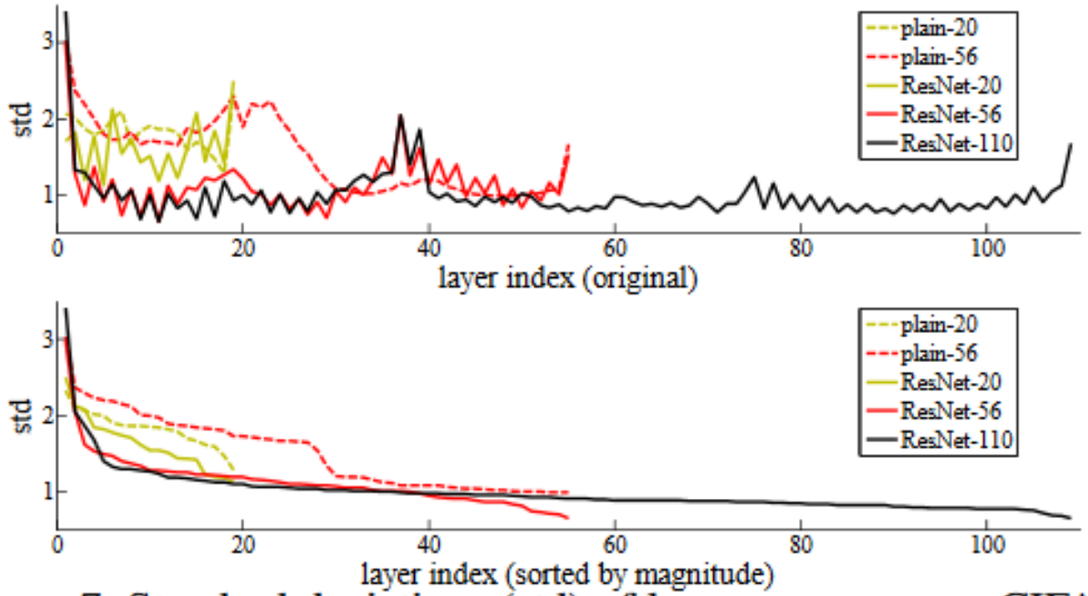
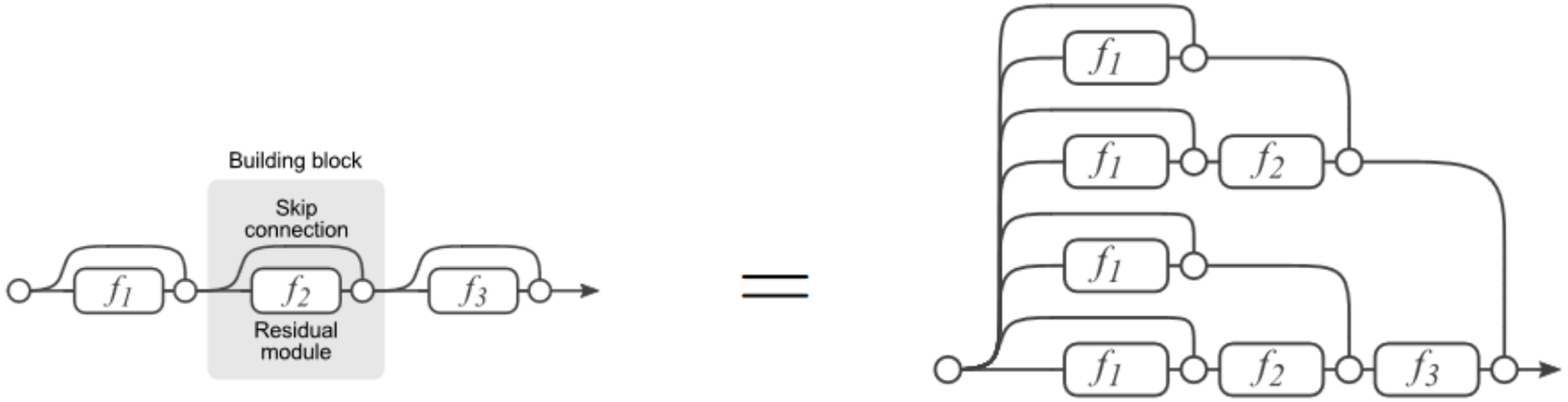


Figure 7. Standard deviations (std) of layer responses on CIFAR-10. The responses are the outputs of each 3×3 layer, after BN and before nonlinearity. **Top**: the layers are shown in their original order. **Bottom**: the responses are ranked in descending order.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, Deep Residual Learning for Image Recognition, CVPR 2016 // https://arxiv.org/abs/1512.03385

ResNet: почему работает

Моделирует ансамбль сетей разной глубины



(a) Conventional 3-block residual network (b) Unraveled view of (a)

Figure 1: Residual Networks are conventionally shown as (a), which is a natural representation of Equation (1). When we expand this formulation to Equation (6), we obtain an *unraveled view* of a 3-block residual network (b). Circular nodes represent additions. From this view, it is apparent that residual networks have $O(2^n)$ implicit paths connecting input and output and that adding a block doubles the number of paths.

вспомним, что на разных уровнях – разные графические примитивы

Andreas Veit, Michael Wilber, Serge Belongie «Residual Networks Behave Like Ensembles of Relatively Shallow Networks» // https://arxiv.org/pdf/1605.06431.pdf

Эксперименты с удалением слоёв

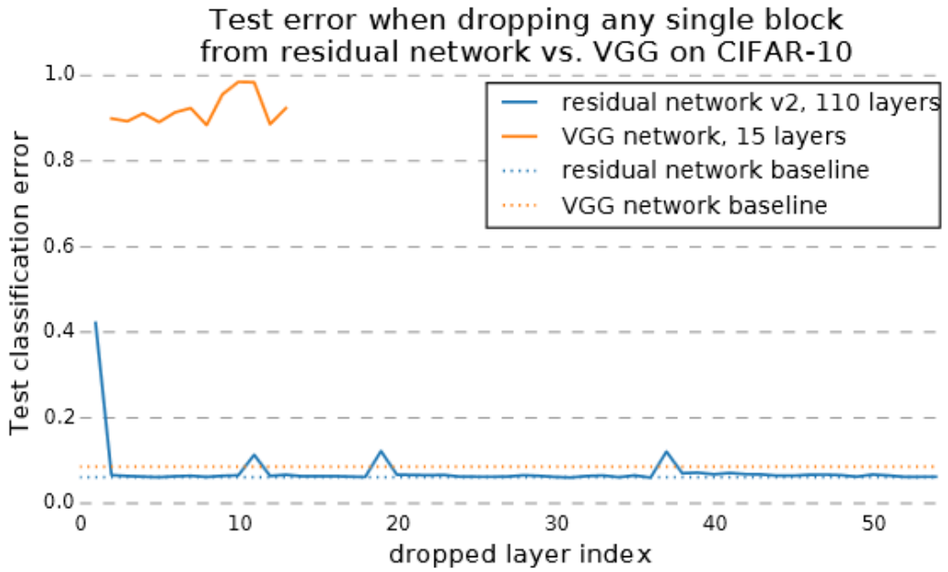


Figure 3: Deleting individual layers from VGG and a residual network on CIFAR-10. VGG performance drops to random chance when any one of its layers is deleted, but deleting individual modules from residual networks has a minimal impact on performance. Removing downsampling modules has a slightly higher impact.

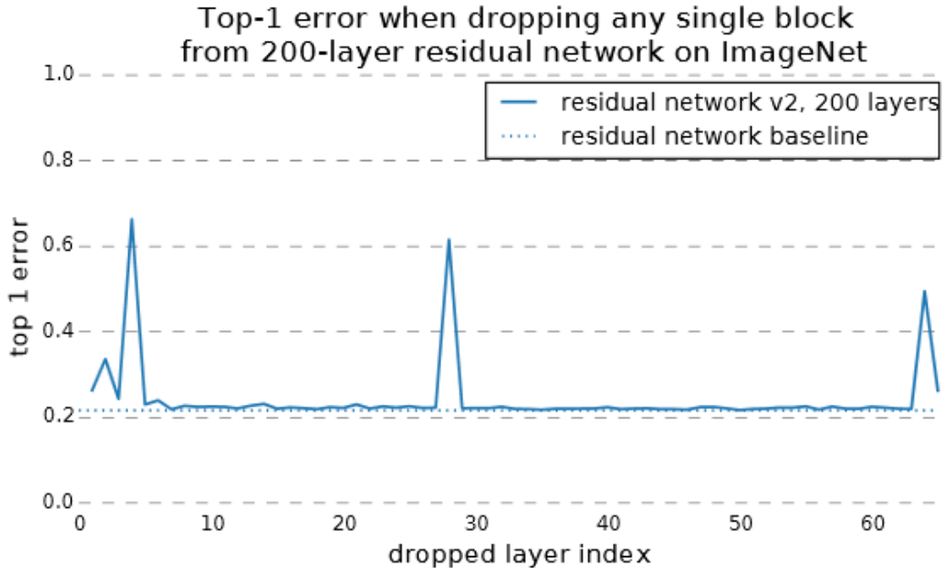
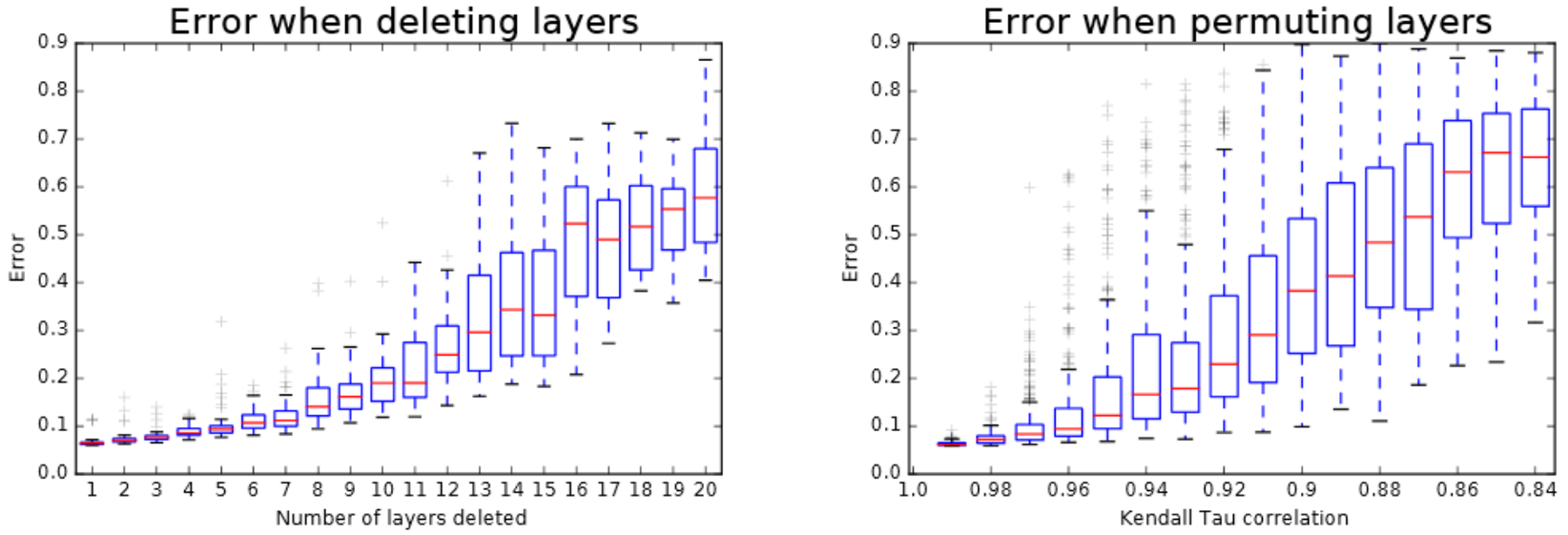


Figure 4: Results when dropping individual blocks from residual networks trained on ImageNet are similar to CIFAR results. However, downsampling layers tend to have more impact on ImageNet.

удаляется residual block, а прямая связь остаётся

пики – понижение размерности

Эксперименты с удалением и перестановкой слоёв



(a) (b)

Figure 5: (a) Error increases smoothly when randomly deleting several modules from a residual network. (b) Error also increases smoothly when re-ordering a residual network by shuffling building blocks. The degree of reordering is measured by the Kendall Tau correlation coefficient. These results are similar to what one would expect from ensembles.

удаление схоже на удаление деревьев в RF – малый эффект

ещё одно подтверждение сходства с ансамблем

см. ещё рис. в статье

ResNet: число слоёв

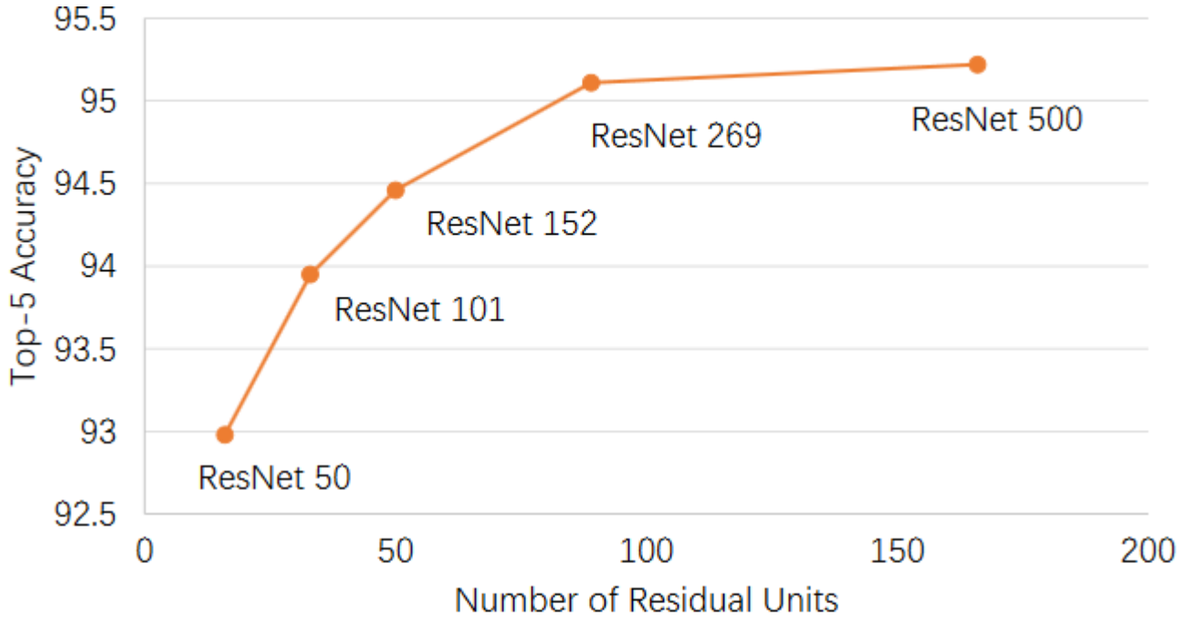


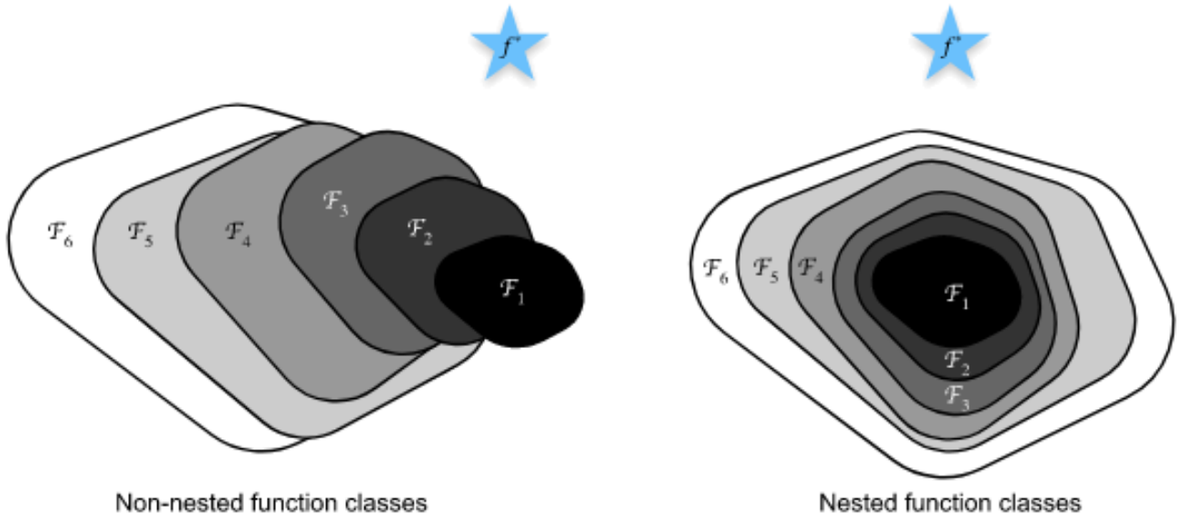
Figure 2: Top-5 single crop accuracies of ResNets [10] with different number of residual units on the ILSVRC 2012 validation set. As the number of residual units increases beyond 100, we can see that the return diminishes.

https://arxiv.org/pdf/1611.05725.pdf

ResNet: функциональное обоснование

При добавлении слоя и прокидывании связи мы получаем нейросейть-функцию, которая не менее функционально мощная (по вложению)

без прокидывания – нет гарантии



https://d2l.ai/chapter_convolutional-modern/resnet.html

ResNet: уменьшение полносвязных слоёв

	# of Parameters	# of Convolutional Layers	# of Dense Layers
LeNet	60 k	2	3
AlexNet	62 M	5	3
VGG16	138 M	13	3
VGG19	144 M	16	3
ResNet50	25.6 M	49	1
ResNet101	44.5 M	100*	1
ResNet152	60.3 M	151*	1

* Not include "1x1 conv skip connection"

https://github.com/aws-samples/aws-machine-learning-university-accelerated-cv/tree/master/slides

«Классика в наши дни»

Классические архитектуры в наши дни: VGG

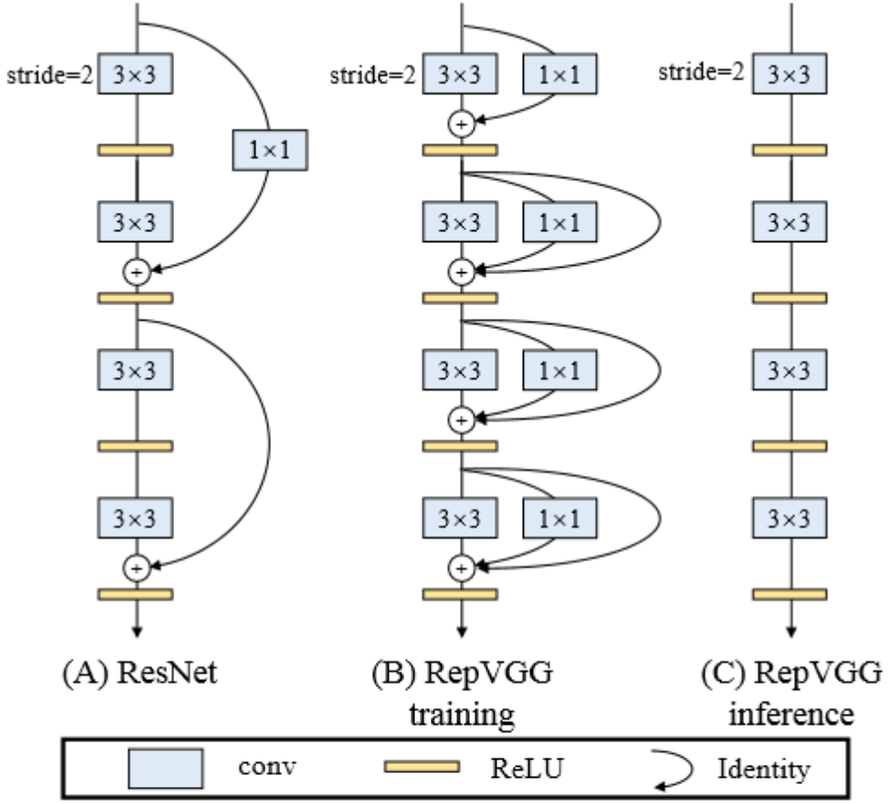


Figure 2: Sketch of RepVGG architecture. RepVGG has 5 stages and conducts down-sampling via stride-2 convolution at the beginning of a stage. Here we only show the first 4 layers of a specific stage. As inspired by ResNet [10], we also use identity and 1×1 branches, but only for training.

Идея: обучить с прокидыванием как в ResNet, а потом его убрать

Свёртки 3×3 имеют наилучшую «вычислительную плотность», поэтому будем использовать их

Table 1: Speed test with varying kernel size and batch size = 32, input channels = output channels = 2048, resolution = 56×56, stride = 1 on NVIDIA 1080Ti. The results of time usage are average of 10 runs after warming up the hardware.

Kernel size	Theoretical FLOPs (B)	Time usage (ms)	Theoretical TFLOPS
1×1	420.9	84.5	9.96
3×3	3788.1	198.8	**38.10**
5×5	10522.6	2092.5	10.57
7×7	20624.4	4394.3	9.38

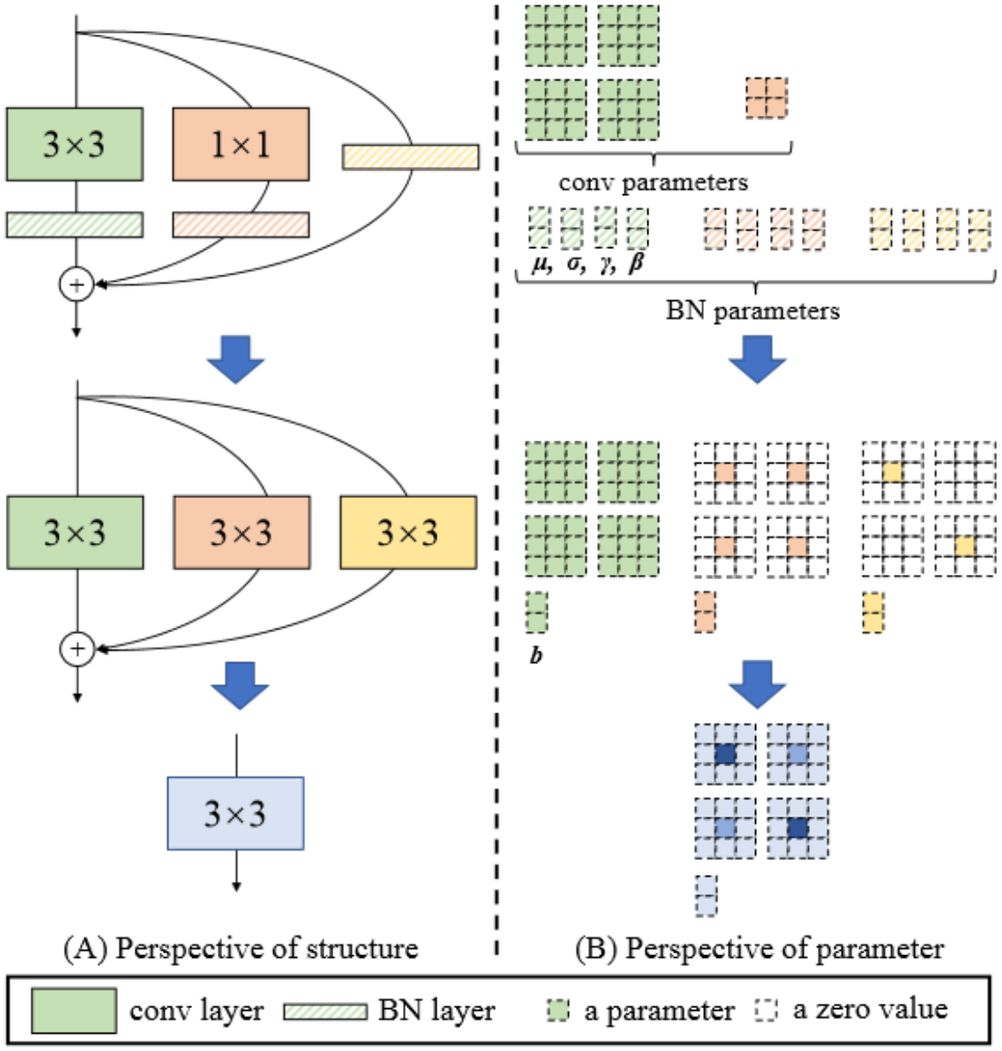


Figure 4: Structural re-parameterization of a RepVGG block. For the ease of visualization, we assume $C_2 = C_1 = 2$, thus the 3×3 layer has four 3×3 matrices and the kernel of 1×1 layer is a 2×2 matrix.

Почему VGG

- **скорость**

есть и более быстрые архитектуры, но скорость VGG близка к теоретической, а у остальных – нет

- **экономия памяти**

если есть прокидывание – нужно дольше хранить вычисления

- **гибкость**

можно, например, удалять ненужные каналы

Model	Top-1 acc	Speed	Params (M)	Theo FLOPs (B)	Wino MULs (B)
RepVGG-A0	72.41	3256	8.30	1.4	0.7
ResNet-18	71.16	2442	11.68	1.8	1.0
RepVGG-A1	74.46	2339	12.78	2.4	1.3
RepVGG-B0	75.14	1817	14.33	3.1	1.6
ResNet-34	74.17	1419	21.78	3.7	1.8
RepVGG-A2	76.48	1322	25.49	5.1	2.7
RepVGG-B1g4	77.58	868	36.12	7.3	3.9
EfficientNet-B0	75.11	829	5.26	0.4	-
RepVGG-B1g2	77.78	792	41.36	8.8	4.6
ResNet-50	76.31	719	25.53	3.9	2.8
RepVGG-B1	78.37	685	51.82	11.8	5.9
RegNetX-3.2GF	77.98	671	15.26	3.2	2.9
RepVGG-B2g4	78.50	581	55.77	11.3	6.0
ResNeXt-50	77.46	484	24.99	4.2	4.1
RepVGG-B2	78.78	460	80.31	18.4	9.1
ResNet-101	77.21	430	44.49	7.6	5.5
VGG-16	72.21	415	138.35	15.5	6.9
ResNet-152	77.78	297	60.11	11.3	8.1
ResNeXt-101	78.42	295	44.10	8.0	7.9

Table 5: RepVGG models and baselines trained in 200 epochs with Autoaugment [4], label smoothing and mixup.

В итоге получается RepVGG =
[3×3 conv + ReLU] × k

Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, Jian Sun «RepVGG: Making VGG-style ConvNets Great Again» // https://arxiv.org/abs/2101.03697

Тренировка ResNet

Training Procedure	Number of epochs	Training resolution	Training time	Peak memory by GPU (MB)	Numbers of GPU	Top-1 accuracy val	real	v2
A1	600	224 × 224	110h	22,095	4	80.4	85.7	68.7
A2	300	224 × 224	55h	22,095	4	79.8	85.4	67.9
A3	100	160 × 160	15h	11,390	4	78.1	84.5	66.1

Table 1: Training resources used for our three training procedures on V100 GPUs and corresponding accuracies at resolution 224×224 on ImageNet1k-val, -V2 and -Real. Note, the top-1 val acc. of pytorch-zoo [1] is 76.1%.

Попытка обучить модель с использованием последних хаков…
аналогично обучили и другие модели

binary cross-entropy (BCE) – вместо CE, лучше согласовывается с ~ Mixup
RRC = Random Resized Crop
LAMB из статьи «Large batch optimization for deep learning: Training BERT in 76 minutes»
ECA-Net – Efficient Channel Attention

Ross Wightman, Hugo Touvron, Hervé Jégou «ResNet strikes back: An improved training procedure in timm» // https://arxiv.org/abs/2110.00476

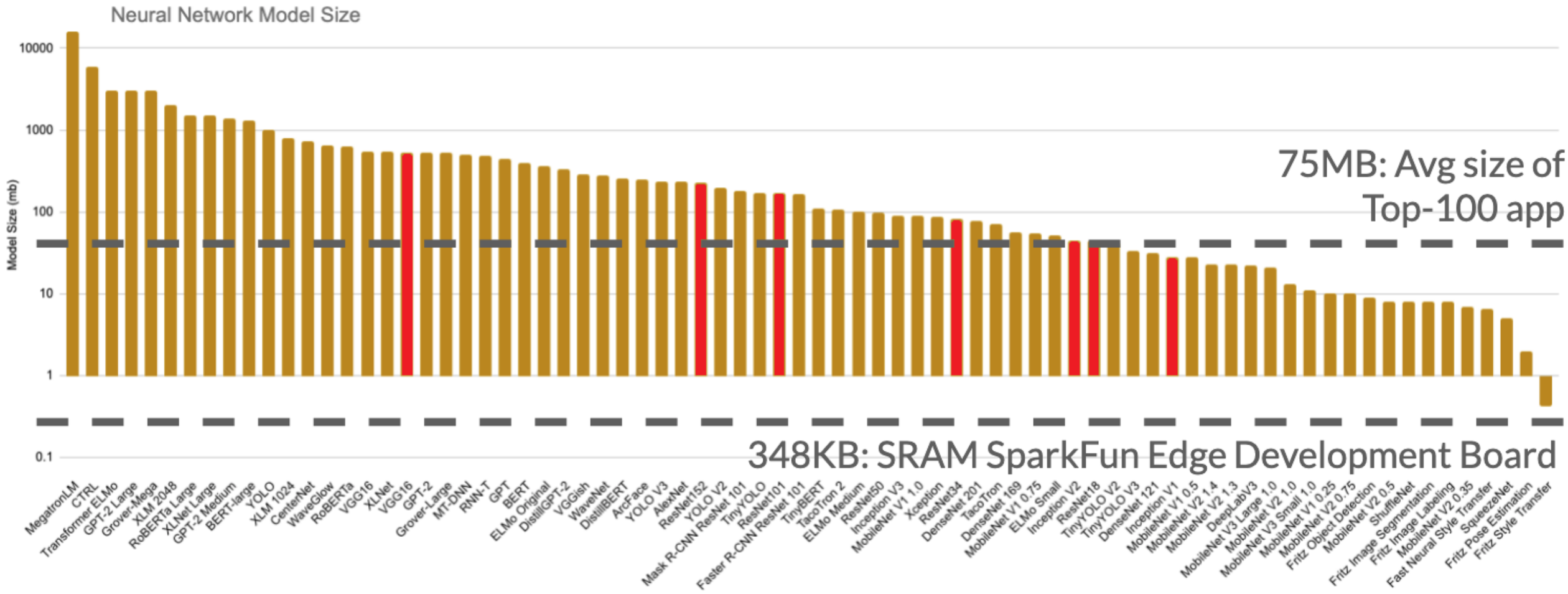
Table 2: Ingredients and hyper-parameters used for ResNet-50 training in different papers. We compare existing training procedures with ours.

	Previous approaches					Ours		
Procedure →	ResNet	PyTorch	FixRes	DeiT	FAMS (×4)	A1	A2	A3
Reference	[13]	[1]	[48]	[45]	[10]			
Train Res	224	224	224	224	224	224	224	160
Test Res	224	224	224	224	224	224	224	224
Epochs	90	90	120	300	400	600	300	100
# of forward pass	450k	450k	300k	375k	500k	375k	188k	63k
Batch size	256	256	512	1024	1024	2048	2048	2048
Optimizer	SGD-M	SGD-M	SGD-M	AdamW	SGD-M	LAMB	LAMB	LAMB
LR	0.1	0.1	0.2	1×10^{-3}	2.0	5×10^{-3}	5×10^{-3}	8×10^{-3}
LR decay	step	step	step	cosine	step	cosine	cosine	cosine
decay rate	0.1	0.1	0.1	-	$0.02^{t/400}$	-	-	-
decay epochs	30	30	30	-	1	-	-	-
Weight decay	10^{-4}	10^{-4}	10^{-4}	0.05	10^{-4}	0.01	0.02	0.02
Warmup epochs	✗	✗	✗	5	5	5	5	5
Label smoothing ε	✗	✗	✗	0.1	0.1	0.1	✗	✗
Dropout	✗	✗	✗	✗	✗	✗	✗	✗
Stoch. Depth	✗	✗	✗	0.1	✗	0.05	0.05	✗
Repeated Aug	✗	✗	✓	✓	✗	✓	✓	✗
Gradient Clip.	✗	✗	✗	✗	✗	✗	✗	✗
H. flip	✓	✓	✓	✓	✓	✓	✓	✓
RRC	✗	✓	✓	✓	✓	✓	✓	✓
Rand Augment	✗	✗	✗	9/0.5	✗	7/0.5	7/0.5	6/0.5
Auto Augment	✗	✗	✗	✗	✓	✗	✗	✗
Mixup alpha	✗	✗	✗	0.8	0.2	0.2	0.1	0.1
Cutmix alpha	✗	✗	✗	1.0	✗	1.0	1.0	1.0
Erasing prob.	✗	✗	✗	0.25	✗	✗	✗	✗
ColorJitter	✗	✓	✓	✗	✗	✗	✗	✗
PCA lighting	✓	✗	✗	✗	✗	✗	✗	✗
SWA	✗	✗	✗	✗	✓	✗	✗	✗
EMA	✗	✗	✗	✗	✗	✗	✗	✗
Test crop ratio	0.875	0.875	0.875	0.875	0.875	0.95	0.95	0.95
CE loss	✓	✓	✓	✓	✓	✗	✗	✗
BCE loss	✗	✗	✗	✗	✗	✓	✓	✓
Mixed precision	✗	✗	✗	✓	✓	✓	✓	✓
Top-1 acc.	75.3%	76.1%	77.0%	78.4%	79.5%	80.4%	79.8%	78.1%

Table 3: Comparison on ImageNet classification between other architectures trained with our ResNet-50 optimized training procedure **without any hyper-parameters adaptation**. In particular, our procedure must be adapted for deeper/larger models, which benefit from more regularization. For the training cost we report the training time (time) in hours, the number of GPU used (#GPU) and the peak memory by GPU (Pmem) in GB. For A1 and A2, we adopt the same training and test resolution as in the original publication introducing the architecture. For A3 we use a smaller training resolution to reduce the compute-time. †: torchvision [1] results. *: DeiT [45] results.

	A1-A2-org.		A3		Cost							ImageNet-1k-val			
	train	test	train	test	A1	A2	A1-A2			A3		A1	A2	A3	org.
↓ Architecture	res.	res.	res.	res.	time (hour)		# GPU	Pmem	time	# GPU	Pmem	Accuracy(%)			
ResNet-18 [13]†	224	224	160	224	186	93	2	12.5	28	2	6.5	71.5	70.6	68.2	69.8
ResNet-34 [13]†	224	224	160	224	186	93	2	17.5	27	2	9.0	76.4	75.5	73.0	73.3
ResNet-50 [13]†	224	224	160	224	110	55	4	22.0	15	4	11.4	80.4	79.8	78.1	76.1
ResNet-101 [13]†	224	224	160	224	74	37	8	16.3	8	8	8.5	81.5	81.3	79.8	77.4
ResNet-152 [13]†	224	224	160	224	92	46	8	22.5	9	8	11.8	82.0	81.8	80.6	78.3
RegNetY-4GF [32]	224	224	160	224	130	65	4	27.1	15	4	13.9	81.5	81.3	79.0	79.4
RegNetY-8GF [32]	224	224	160	224	106	53	8	19.8	10	8	10.3	82.2	82.1	81.1	79.9
RegNetY-16GF [32]	224	224	160	224	150	75	8	25.6	13	8	13.4	82.0	82.2	81.7	80.4
RegNetY-32GF [32]	224	224	160	224	120	60	16	17.6	12	16	9.4	82.5	82.4	82.6	81.0
SE-ResNet-50 [20]	224	224	160	224	102	51	4	27.6	16	4	14.2	80.0	80.1	77.0	76.7
SENet-154 [20]	224	224	160	224	110	55	16	23.3	12	16	12.2	81.7	81.8	81.9	81.3
ResNet-50-D [14]	224	224	160	224	100	50	4	23.9	14	4	12.3	80.7	80.2	78.7	79.3
ResNeXt-50-32x4d [51]†	224	224	160	224	80	40	8	14.3	15	4	14.6	80.5	80.4	79.2	77.6
EfficientNet-B0 [41]	224	224	160	224	110	55	4	22.1	15	4	11.4	77.0	76.8	73.0	77.1
EfficientNet-B1 [41]	240	240	160	224	62	31	8	17.9	8	8	7.9	79.2	79.4	74.9	79.1
EfficientNet-B2 [41]	260	260	192	256	76	38	8	22.8	9	8	11.9	80.4	80.1	77.5	80.1
EfficientNet-B3 [41]	300	300	224	288	62	31	16	19.5	6	16	10.1	81.4	81.4	79.2	81.6
EfficientNet-B4 [41]	380	380	320	380	64	32	32	20.4	8	32	14.3	81.6	82.4	81.2	82.9
ViT-Ti [45]*	224	224	160	224	98	49	4	16.3	14	4	7.0	74.7	74.1	66.7	72.2
ViT-S [45]*	224	224	160	224	68	34	8	16.1	8	8	7.0	80.6	79.6	73.8	79.8
ViT-B [11]*	224	224	160	224	66	33	16	16.4	5	16	7.3	80.4	79.8	76.0	81.8
timm [50] specific architectures															
ECA-ResNet-50-T	224	224	160	224	112	56	4	29.3	15	4	15.0	81.3	80.9	79.6	-
EfficientNetV2-rw-S [42]	288	384	224	288	52	26	16	16.6	7	16	10.1	82.3	82.9	80.9	83.8
EfficientNetV2-rw-M [42]	320	384	256	352	64	32	32	18.5	9	32	12.1	80.6	81.9	82.3	84.8
ECA-ResNet-269-D	320	416	256	320	108	54	32	27.4	11	32	17.8	83.3	83.9	83.3	85.0

Сложность моделей



Итоги

Есть много стандартных приёмов:

- **каскад свёрток**
- **факторизация свёрток / параметров**
- **1×1-свёртки**
- **узкое горло**
- **прокидывание связей**

Если задача как-то связана с изображениями,
часто берут архитектуру проверенную на ImageNet-е
(детектирование, сегментация, определение позы/действия, img → text)

И даже, если не связана с изображениями…
Volumetric Brain Segmentation (VoxResNet)
City-Wide Crowd Flow Prediction (ST-ResNet)
Generating Realistic Voices (WaveNet)