TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2018

Convolutional Neural Networks — CNNs

Review: EDF

In EDF we write

$$
\begin{aligned}
y &= F(x) \\
z &= G(y, x) \\
u &= H(z) \\
\ell &= u
\end{aligned}
$$

This is Python code where variables are bound to nodes (objects).

Input Nodes, Parameter Nodes, and Computed Nodes

Inputs and computed nodes have batch values (tensors with a batch element index).

Parameter values have no batch index.

Input nodes are directly assigned a batch of values.

Computed nodes (including the loss node) are assigned a batch of values by thier forward method.

Parameters are updated by an SGD step after each training batch.

MLP in EDF

The following Python code constructs the computation graph of an MLP with one hidden layer.

```
L1 = Relu(Affine(Phi1,x))
Q = Softmax(Sigmoid(Affine(Phi2,L1))
ell = LogLoss(Q,y)
```

Here `Phi1` and `Phi2` are "parameter packages" (a matrix and a bias vector in this case).

The Core of EDF

```
def Forward():
    for c in CompNodes: c.forward()

def Backward(loss):
    for c in CompNodes + Parameters: c.grad = 0
    loss.grad = 1/B #batch size
    for c in CompNodes[::-1]: c.backward()

def SGD():
    for p in Parameters:
        p.SGD()
```

Explicit Index Notation

$$A[x_1, \ldots, x_n] \texttt{ += } e[x_1, \ldots, x_n, y_1, \ldots, y_k]$$

abbreviates

for $x_1, \ldots, x_n, y_1, \ldots, y_k$: $A[x_1, \ldots, x_n] \texttt{ += } e[x_1, \ldots, x_n, y_1, \ldots, y_k]$

$$A[x_1, \ldots, x_n] = e[x_1, \ldots, x_n, y_1, \ldots, y_k]$$

abbreviates

$$\text{for } x_1, \ldots, x_n : \; A[x_1, \ldots, x_n] = 0$$

$$A[x_1, \ldots, x_n] \texttt{ += } e[x_1, \ldots, x_n, y_1, \ldots, y_k]$$

Affine Transformation with Batch Index

$$
\begin{aligned}
y[b, i] &= x[b, j] W[j, i] \\
y[b, i] &\texttt{ += } B[i]
\end{aligned}
$$

The Swap Rule for Tensor Products

For backprop swap an input with the output and add "grad".

$$y[b,i] \;=\; x[b,j]W[j,i]$$

$$x.\text{grad}[b,j] \;\texttt{+=}\; y.\text{grad}[b,i]W[j,i]$$

$$W.\text{grad}[i,j] \;\texttt{+=}\; x[b,j]y.\text{grad}[b,i]$$

$$y[b,i] \;\texttt{+=}\; B[i]$$

$$B.\text{grad}[i] \;\texttt{+=}\; y.\text{grad}[b,i]$$

The General Swap Rule

$$A[x_1, \ldots, x_n] \texttt{ += } f\left(\begin{array}{c} B[x_1, \ldots, x_n, y_1, \ldots, y_k], \\ C[x_1, \ldots, x_n, y_1, \ldots, y_k] \end{array}\right)$$

$$B.\text{grad}\left[\begin{array}{c} x_1, \ldots, x_n, \\ y_1, \ldots, y_k \end{array}\right] \texttt{ += } A.\text{grad}[x_1, \ldots, x_n] \frac{\partial f\left(\begin{array}{c} B[x_1, \ldots, x_n, y_1, \ldots, y_k], \\ C[x_1, \ldots, x_n, y_1, \ldots, y_k] \end{array}\right)}{\partial B\left[\begin{array}{c} x_1, \ldots, x_n, \\ y_1, \ldots, y_k \end{array}\right]}$$

Here f takes scalar arguments and the tensor values are scalar. This pattern covers essentially all of deep learning.

Reshaping Tensors

For an ndarray x (tensor) we have that x.shape is a tuple of dimensions. The product of the dimensions is the number of numbers.

In NumPy an ndarray (tensor) x can be reshaped into any shape with the same number of numbers.

Broadcasting

Shapes can contain dimensions of size 1.

Dimensions of size 1 are treated as "wild card" dimensions in operations on tensors.

$$
\begin{aligned}
x.\text{shape} &= (5, 1) \\
y.\text{shape} &= (1, 10) \\
z &= x * y \\
z.\text{shape} &= (5, 10) \\
z[i, j] &= x[i, 0] * y[0, j]
\end{aligned}
$$

```
class Affine(CompNode):
    ...
    def backward(self):
        ...
        # W.grad[i,j] += y.grad[b,j]x.value[b,i]
        self.Phi.W.addgrad(self.grad[:,np.newaxis,:]
                           *self.x.value[:,:,np.newaxis])

class Parameter:
    ...
    def addgrad(self, delta):
        self.grad += np.sum(delta, axis = 0)
```

Broadcasting

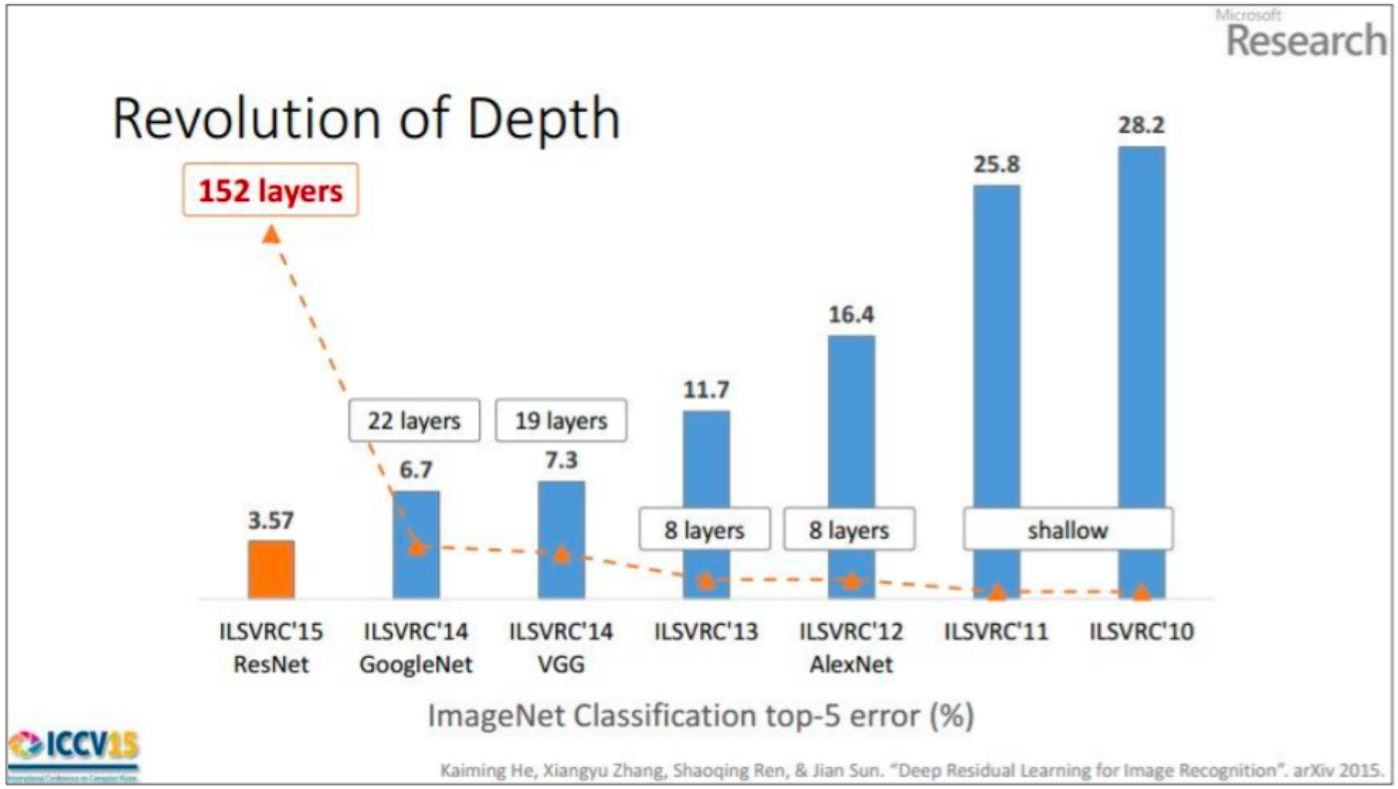
When a scalar is added to a matrix the scalar is reshaped to shape $(1, 1)$ so that it is added to each element of the matrix.

When a vector of shape (k) is added to a matrix the vector is reshaped to $(1, k)$ so that it is added to each row of the matrix.

In general when two tensors of different order (number of dimensions) are added, unit dimensions are prepended to the shape of the tensor of smaller order to make the orders match.

CNNs

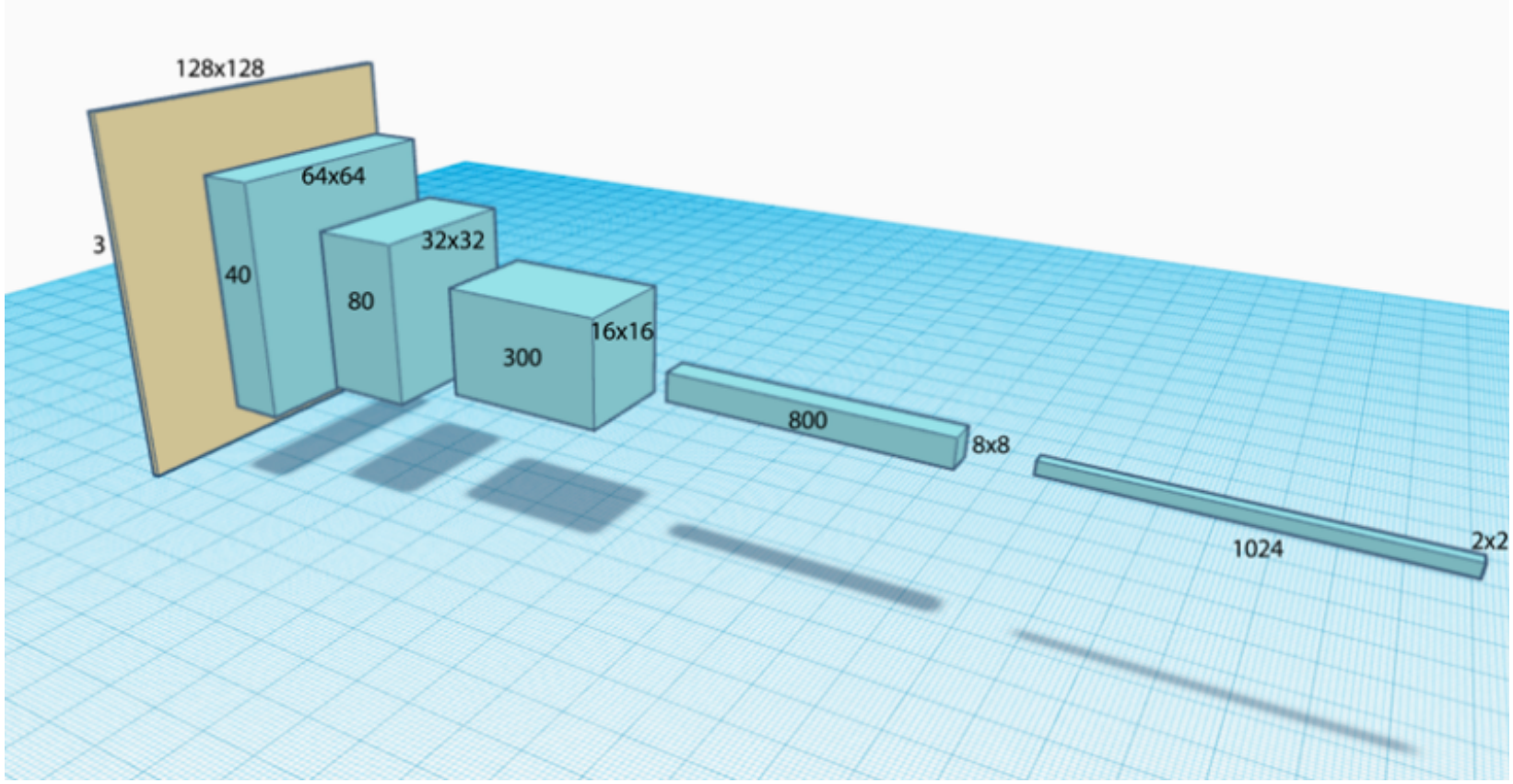
Imagenet Classification. 1000 kinds of objects.



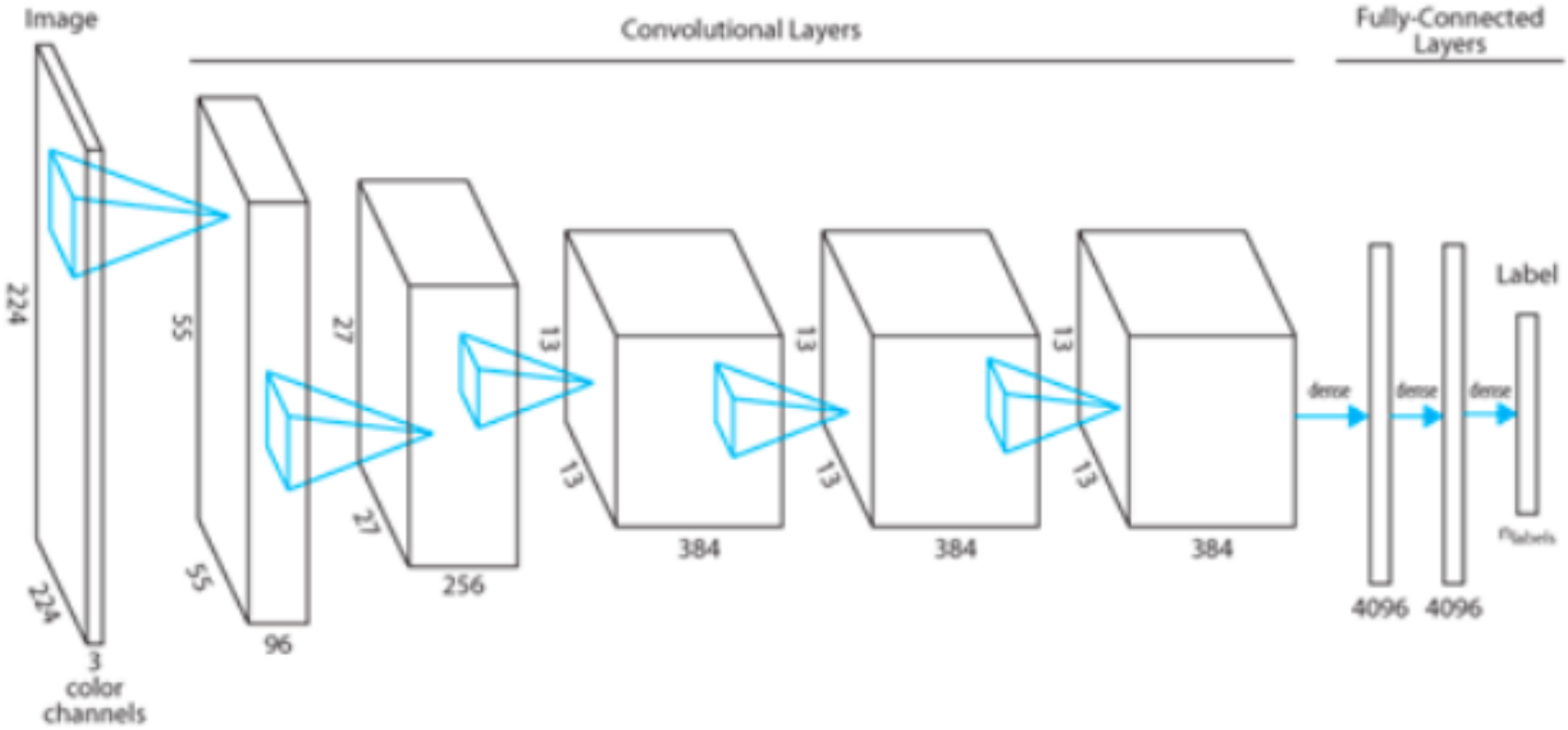
(slide from Kaiming He's recent presentation)

2016 error rate is 3.0% 2017 error rate is 2.25%

A Sequence of "Images"



AlexNet, 2012



Pytorch Conv2d Parameter Package Class

PyTorch:

```
torch.nn.Conv2d(in_channels,
                out_channels,
                kernel_size,
                stride=1,
                padding=0,
                dilation=1,
                groups=1,
                bias=True)
```

An instance of this class is a "parameter package" or "model" rather than a computation node.

A Convolution Class in EDF

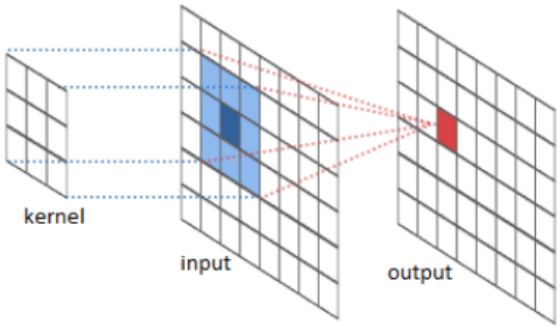
In EDF we would define a class for convolution parameter packages.

We would then construct the computation node for the output using Python code.

```
Y = Relu(Conv(Phi,X)).
```

A 2D CNN

A convolution slides a filter (a kernel) across an image.



$W \qquad x \qquad y$

River Trail Documentation

$$y[b, i, j, c_y] \;=\; W[\Delta i, \Delta j, c_x, c_y]x[b, i + \Delta i, j + \Delta j, c_x]$$

$$y[b, i, j, c_y] \;\texttt{+=}\; B[c_y]$$

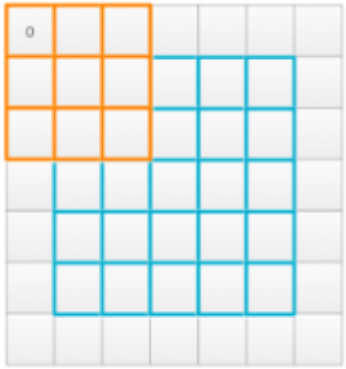
Use Swap Rule to get Backward Method

$$y[b, i, j, c_y] \;=\; W[\Delta i, \Delta j, c_x, c_y] x[b, i + \Delta i, j + \Delta j, c_x]$$

$$W.\text{grad}[\Delta i, \Delta j, c_x, c_y] \;\texttt{+=}\; y.\text{grad}[b, i, j, c_y] x[b, i + \Delta i, j + \Delta_j, c_x]$$

$$x.\text{grad}[b, i{+}\Delta i, j{+}\Delta j, c_x] \;\texttt{+=}\; W[\Delta i, \Delta j, c_x, c_y] y.\text{grad}[b, i, j, c_y]$$

Padding



Jonathan Hui

If we pad the input with zeros then the input and output can have the same spatial dimensions.

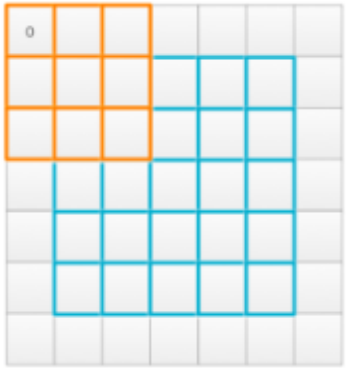
Zero Padding in NumPy

In NumPy we can add a zero padding of width p to an image as follows:

```
padded = np.zeros(W + 2*p,  H + 2*p)

padded[p:W+p, p:H+p] = x
```

Padding



Jonathan Hui

Let x' (full square) be the padding of x (blue square). y also has the blue shape.

$$y[b, i, j, c_y] = W[\Delta i, \Delta j, c_x, c_y]x'[b, i + \Delta i, j + \Delta j, c_x] + B[c_y]$$

For padding p and a filter of width $2p + 1$, we get that y has the same spatial dimensions as x.

Image to Column (Im2C)

Reduce convolution to matrix multiplication — more space but faster.

$$x'[b, i, j, \Delta i, \Delta j, c_x] = x[b, i + \Delta i, j + \Delta j, c_x]$$

$$y[b, i, j, c_y]$$

$$= \left(\sum_{\Delta i, \Delta j, c_x} W[\Delta i, \Delta j, c_x, c_y] * x[b, i + \Delta i,\ j + \Delta j,\ c_x] \right) + B[c_y]$$

$$= \left(\sum_{\Delta i, \Delta j, c_x} x'[b, i, j, \Delta i, \Delta j, c_x] * W[\Delta i, \Delta j, c_x, c_y] \right) + B[c_y]$$

$$= \left(\sum_{(\Delta i, \Delta j, c_x)} x'[(b, i, j), (\Delta i, \Delta j, c_x)] * W[(\Delta i, \Delta j, c_x), c_y] \right) + B[c_y]$$

Strides

We can move the filter by a "stride" s for each spatial step.

$$y[b, i, j, c_y] = W[\Delta i, \Delta j, c_x, c_y]x[b, s * i + \Delta i, s * j + \Delta j, c_x] + B[c_y]$$

Dilation

We can "dilate" the filter by introducing an image step size d for each step in the filter coordinates.

$$y[b, i, j, c_y] = W[\Delta i, \Delta j, c_x, c_y]x[b, i + d * \Delta i, j + d * \Delta j, c_x] + B[c_y]$$

Grouping and Bias Terms

Grouping: Grouping groups the parameters such that different optimizers (learning rates of SGD variants) can be used for different groups.

Bias: The PyTorch parameter package can also specify whether to use a bias term (an affine transformation) or not (a linear transformation).

Max Pooling

```
torch.nn.MaxPool2d(kernel_size,
                   stride=None,
                   padding=0,
                   dilation=1,
                   return_indices=False,
                   ceil_mode=False)
```

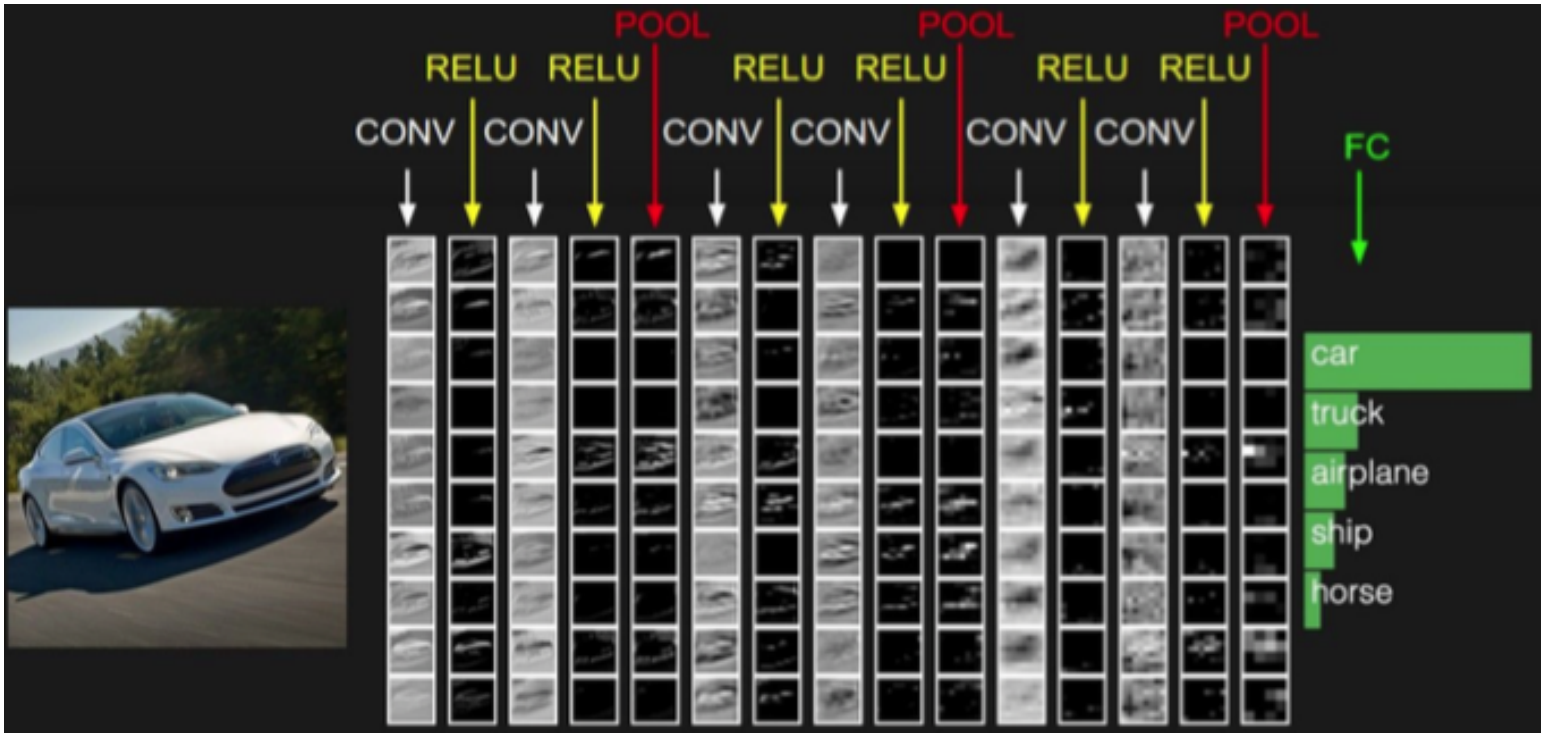
$$y[b, i, j, c] = \max_{\Delta i, \Delta j} \; x[b, s * i + \Delta i, \; s * j + \Delta j, \; c]$$

This is typically done with a stride greater than one so that the image dimension is reduced.

Fully Connected (FC) Layers

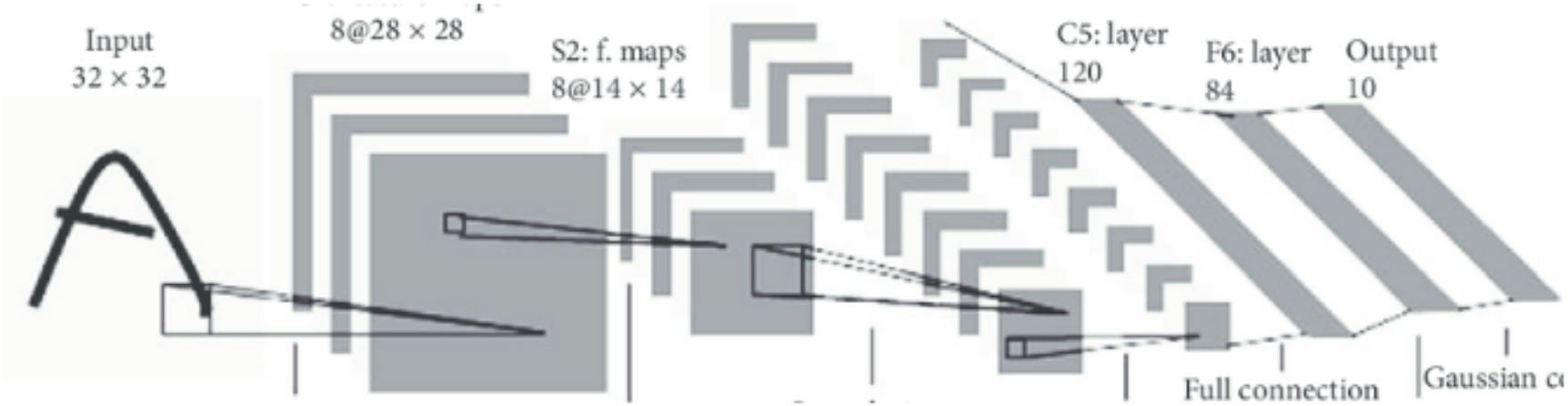
We reshape $x[b, x, y, c]$ to $x[b, (x, y, c)]$ and convert to using an MLP.

Example

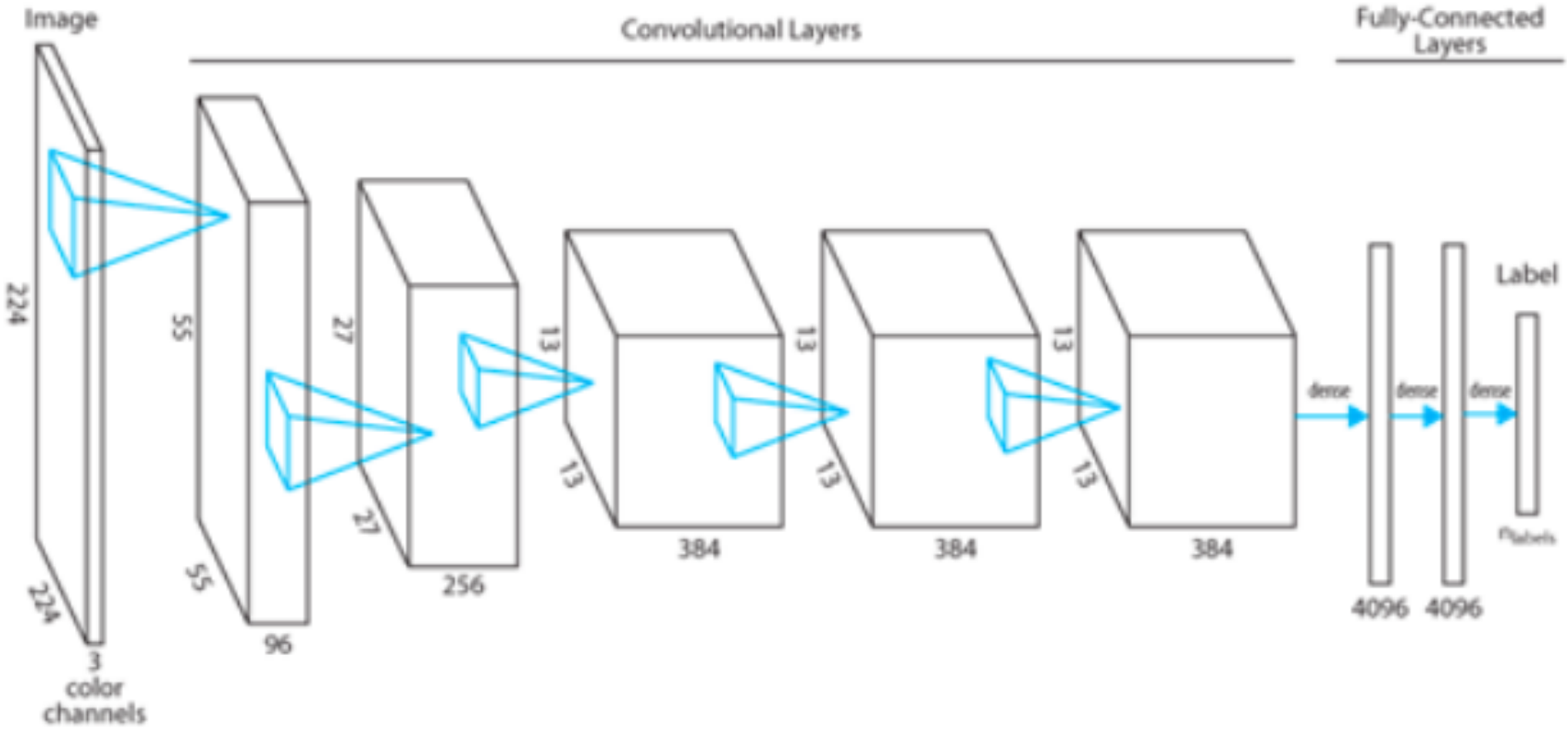


Stanford CS231 Network

LeNet, LeCun, 1998

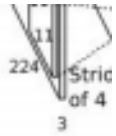


AlexNet, 2012



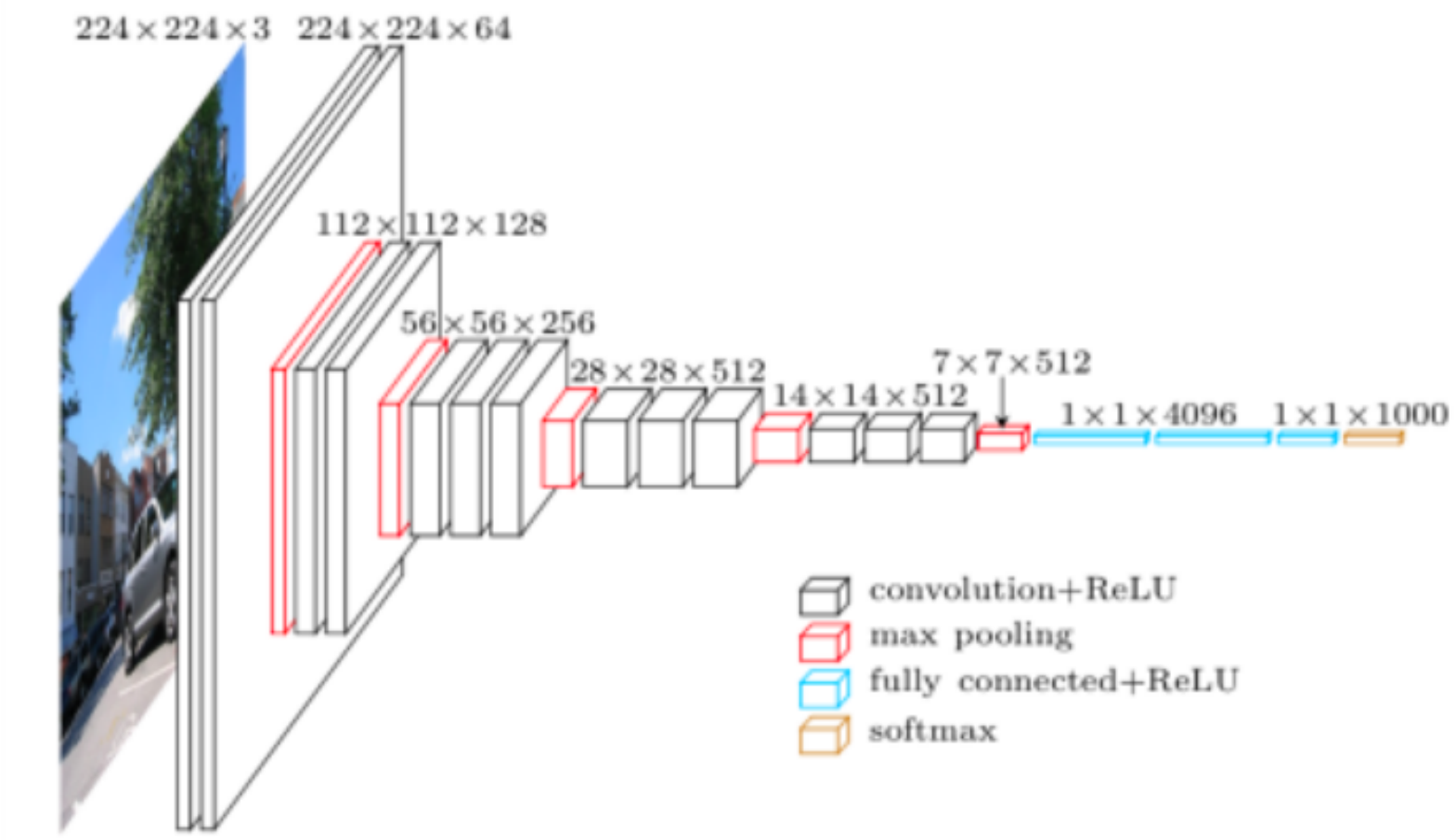
Alexnet

Full (simplified) AlexNet architecture:

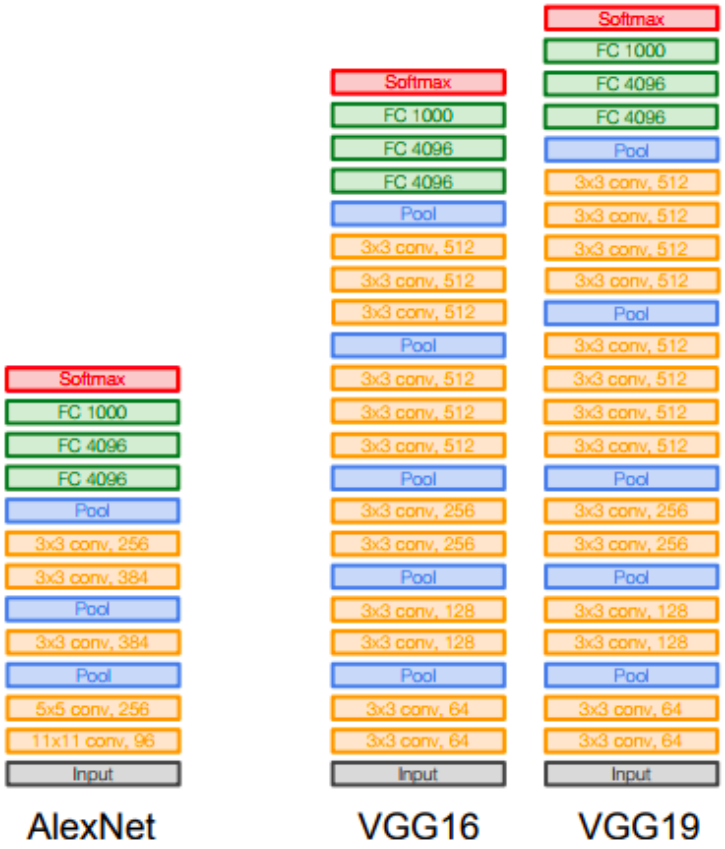


[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

VGG, Zisserman, 2014

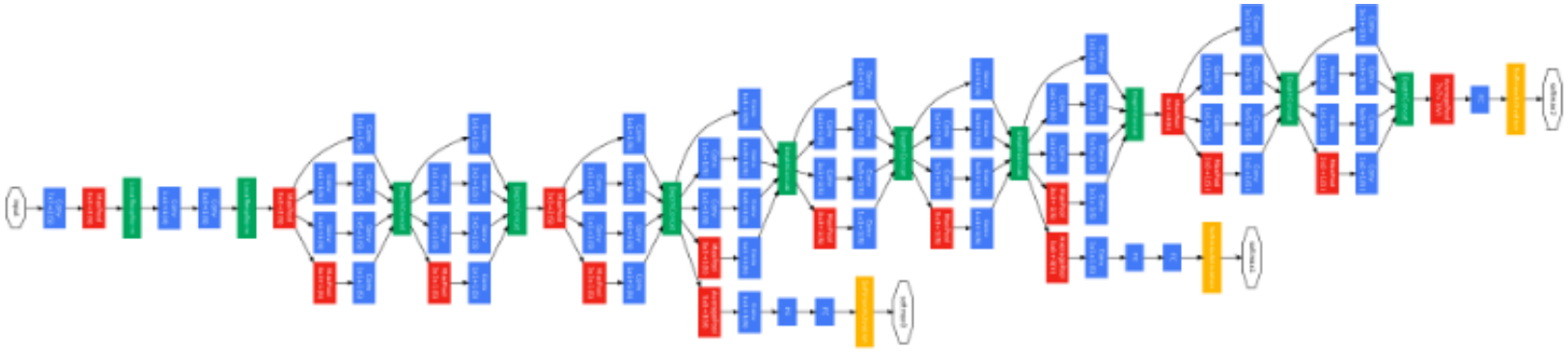


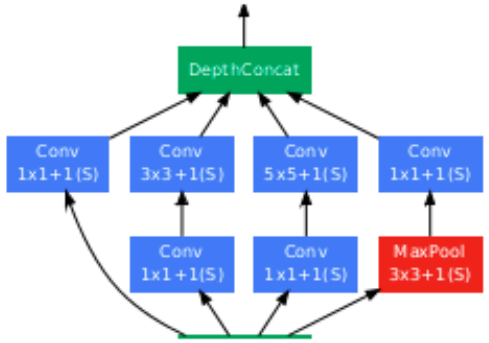
VGG



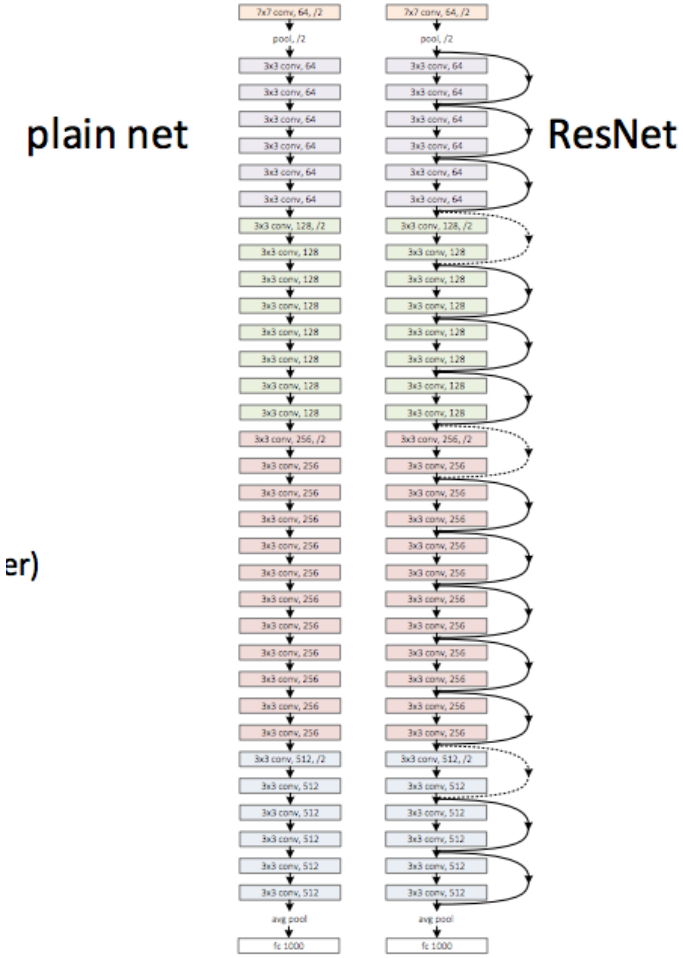
Stanford CS231

Inception, Google, 2014

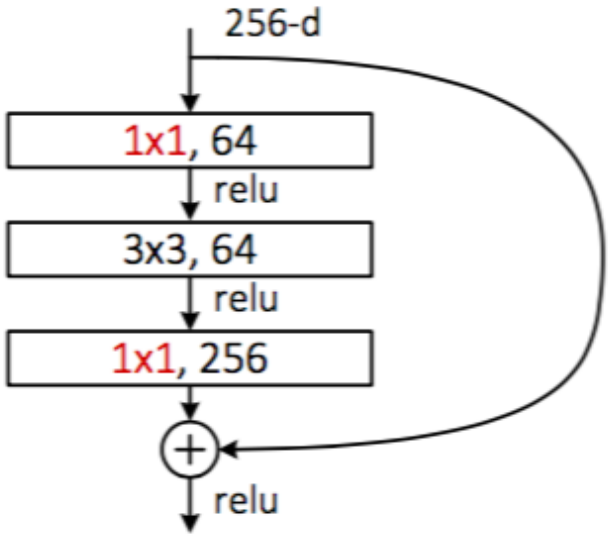




ResNet Kaiming He, 2015



Resnet

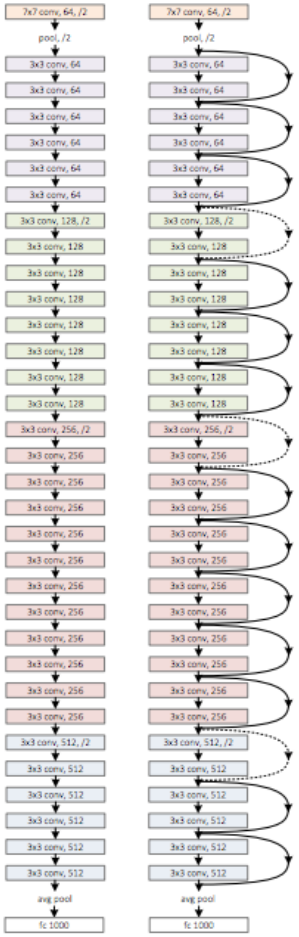


bottleneck

(for ResNet-50/101/152)

plain net

er)

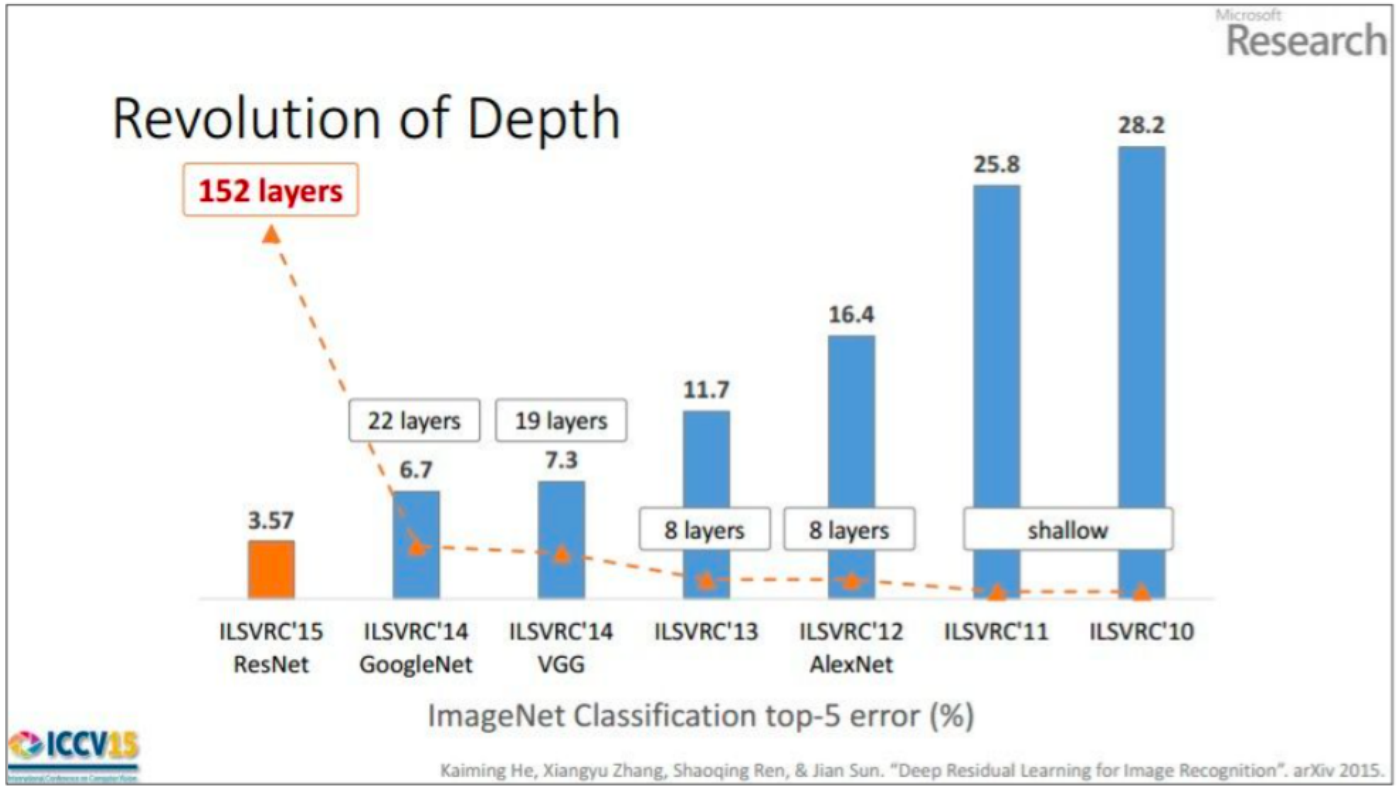


ResNet

[Kaiming He]

Imagenet Classification

1000 kinds of objects.



(slide from Kaiming He's recent presentation)

2016 error rate is 3.0%

2017 error rate is 2.25%

END