Pooling in Convolutional Networks

Sargur Srihari
srihari@buffalo.edu

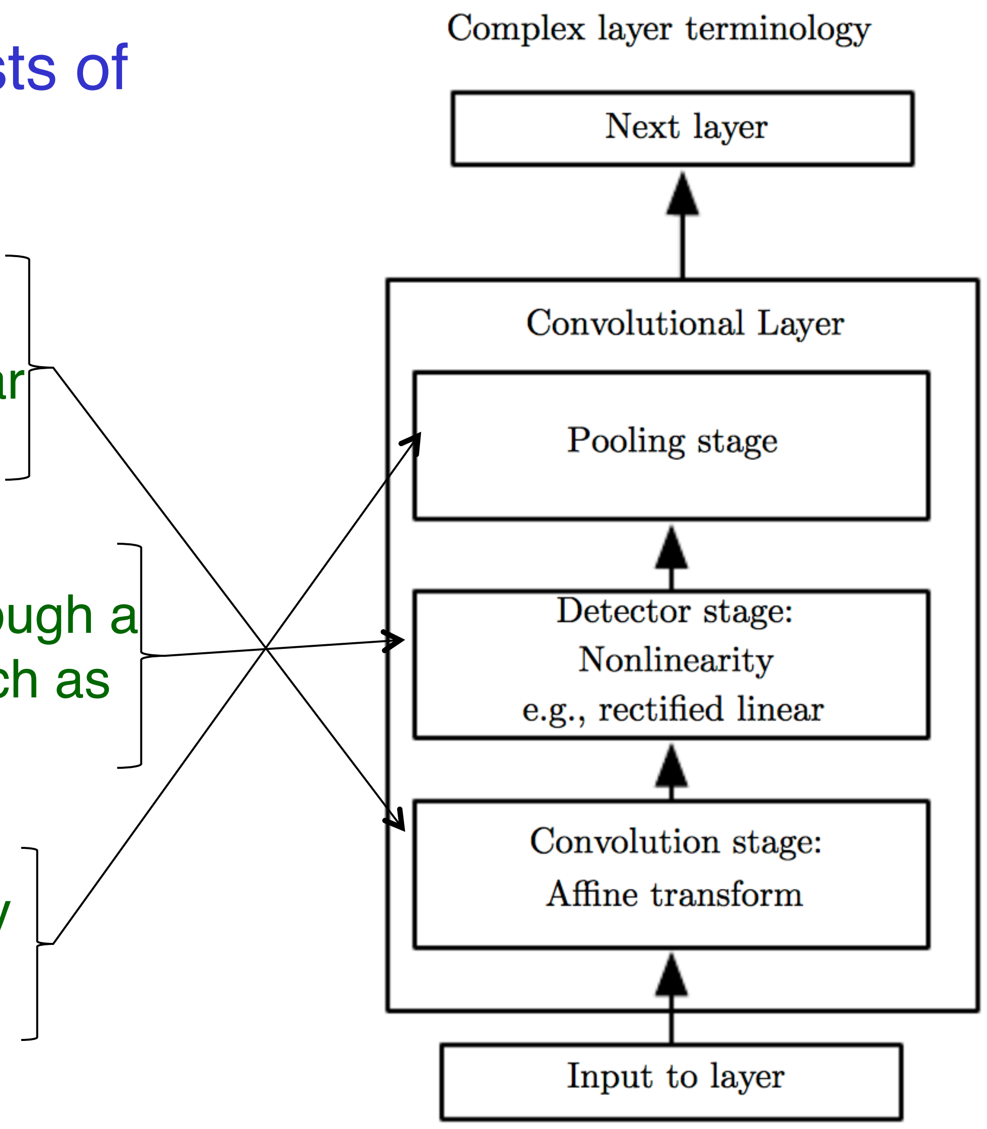
Topics in Convolutional Networks

Topics in Pooling

- Three stages of CNNs
- Two terminologies: simple layers, complex layers
- Types of Pooling functions: Max, Average
- Translation invariance
- Rotation invariance
- Pooling with downsampling
- ConvNet Architectures

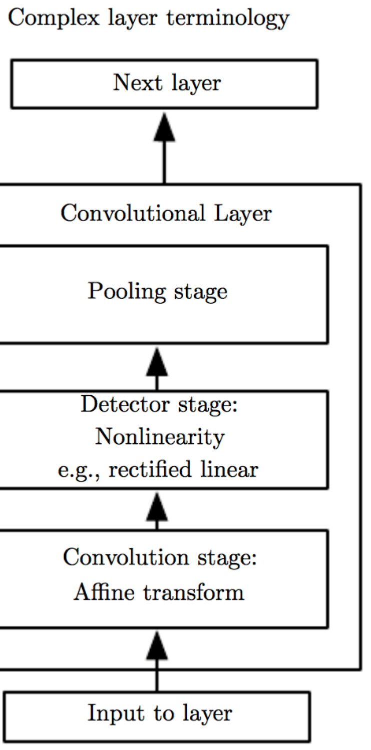
Pooling

- Typical layer of a CNN consists of three stages
- Stage 1:
  - perform several convolutions in parallel to produce a set of linear activations
- Stage 2 (Detector):
  - each linear activation is run through a nonlinear activation function such as ReLU
- Stage 3 (Pooling):
  - Use a pooling function to modify output of the layer further

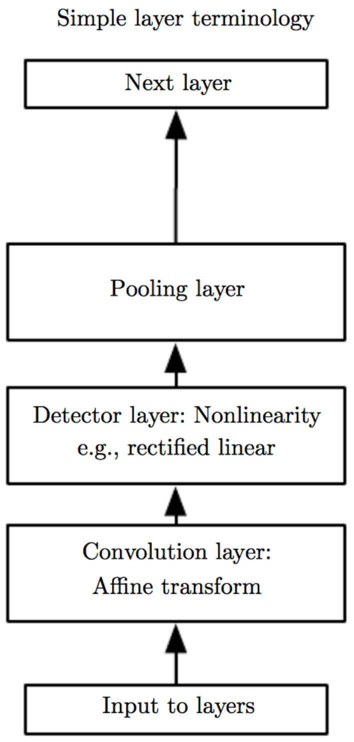


Two terminologies for a typical CNN layer

1. Net is viewed as a small no. of complex layers, each layer having many stages
   - 1-1 mapping between kernel tensors and network layers


Complex layer terminology
Next layer
Convolutional Layer
Pooling stage
Detector stage: Nonlinearity e.g., rectified linear
Convolution stage: Affine transform
Input to layer

2. Net is viewed as a larger no of simple layers
   - Every processing step is a layer in its own right
   - Not every layer has parameters


Simple layer terminology
Next layer
Pooling layer
Detector layer: Nonlinearity e.g., rectified linear
Convolution layer: Affine transform
Input to layers

Types of Pooling functions

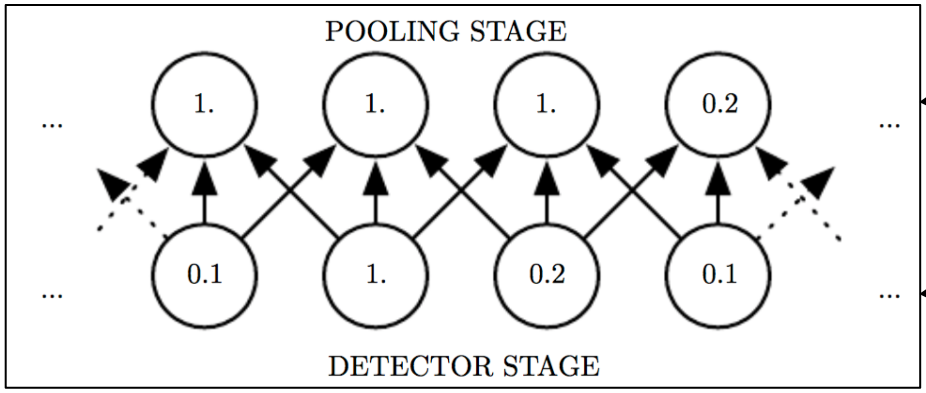
- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby inputs
- Popular pooling functions:
  1. *max pooling* operation reports the maximum output within a rectangular neighborhood
  2. Average of a rectangular neighborhood
  3. L^2 norm of a rectangular neighborhood
  4. Weighted average based on the distance from the central pixel

Pooling causes translation invariance

- In all cases, pooling helps make the representation become approximately *invariant* to small translations of the input
  - If we translate the input by a small amount values of most of the outputs does not change
  - Pooling can be viewed as adding a strong prior that the function the layer learns must be invariant to small translations

Max pooling introduces invariance to translation

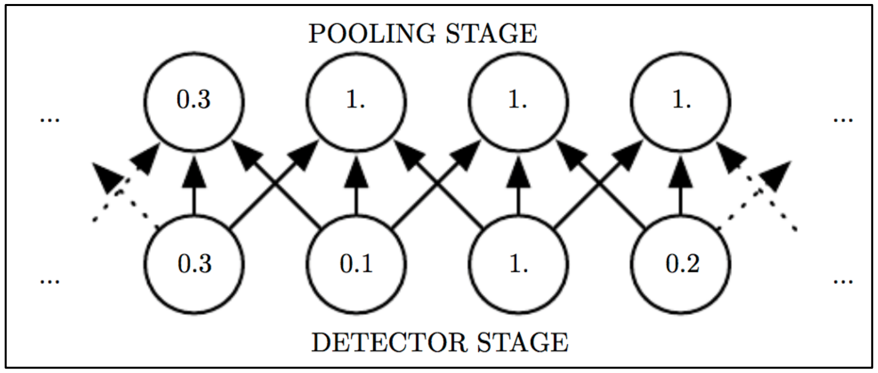
- View of middle of output of a convolutional layer



Outputs of maxpooling

Outputs of nonlinearity

- Same network after the input has been shifted by one pixel



- Every input value has changed, but only half the values of output have changed because maxpooling units are only sensitive to maximum value in neighborhood not exact value

Importance of Translation Invariance

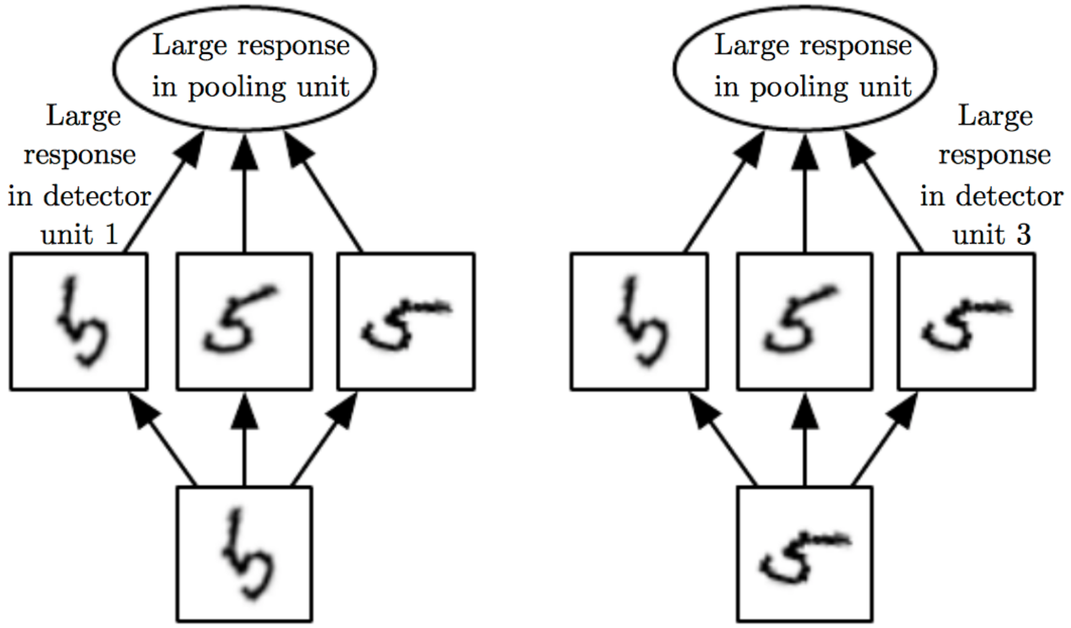
- Invariance to translation is important if we care about whether a feature is present rather than exactly where it is
  - For detecting a face we just need to know that an eye is present in a region, not its exact location
- In other contexts it is more important to preserve location of a feature
  - E.g., to determine a corner we need to know whether two edges are present and test whether they meet

Learning other invariances

- Pooling over spatial regions produces invariance to translation
- But if we pool over the results of separately parameterized convolutions, the features can learn which transformations to become invariant to

Learning Invariance to rotation

- A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input

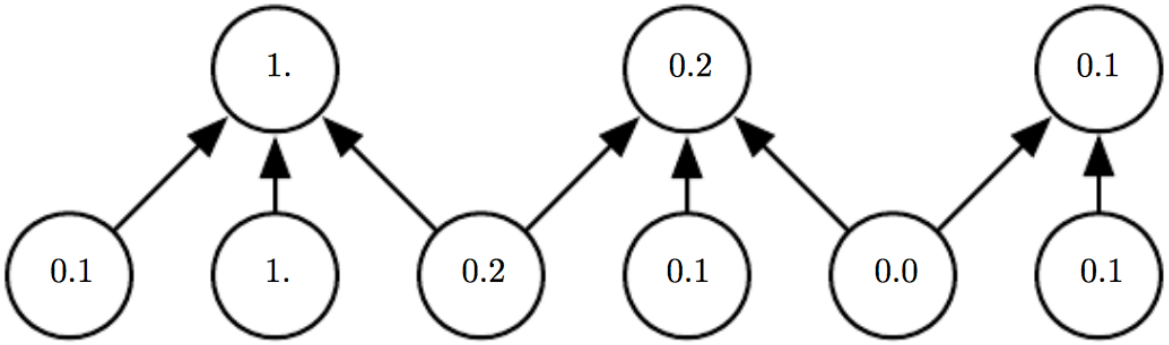

Large response in pooling unit
Large response in detector unit 1
Large response in pooling unit
Large response in detector unit 3

Using fewer pooling units than detector units

- Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units
  - By reporting summary statistics for pooling regions spaced k pixels apart rather than one pixel apart
  - This improves computational efficiency because the next layer has k times fewer inputs to process
- An example is given next

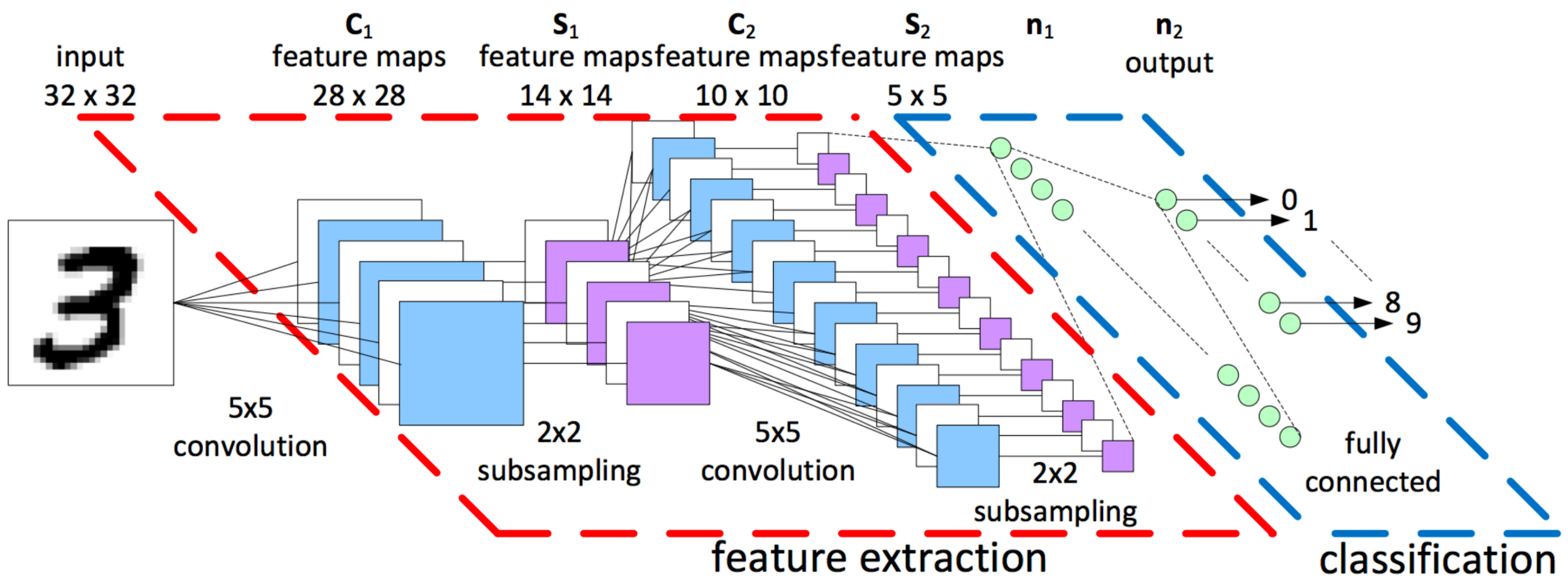
Pooling with down-sampling

- Max-pooling with a pool width of three and a stride between pools of two



- This reduces representation size by a factor of two
  - Which reduces computational burden of next layer
  - Rightmost pooling region has a smaller size but must be included if we don't want to ignore some of the detector units

Subsampling as Average pooling

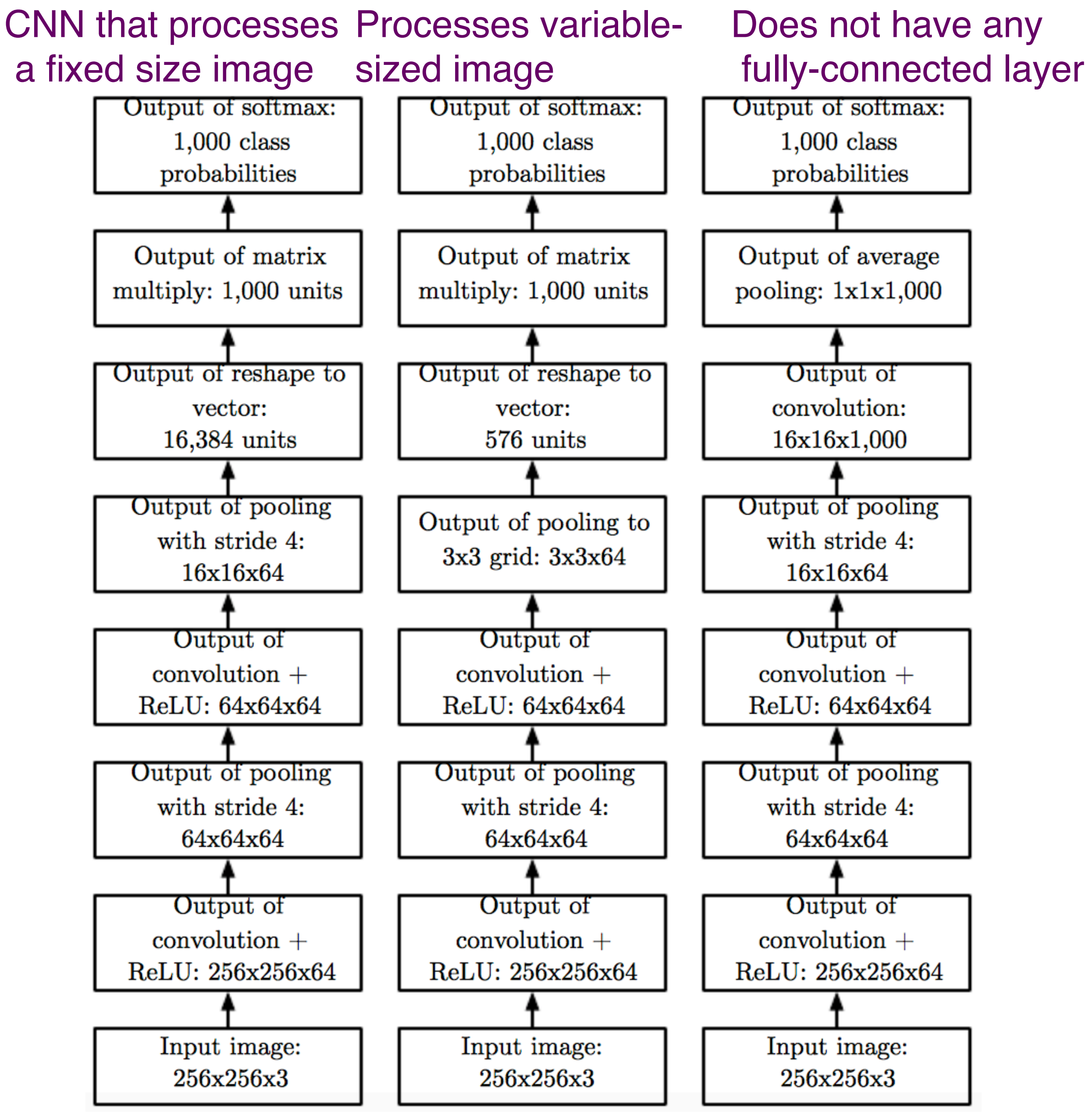


Theoretical Guidance on Pooling

- Which kind of pooling one should use in different situations
- It is possible to dynamically pool features together
  - By running a clustering algorithm on locations of interesting features
    - Yields a different set of pooling regions for each image
  - Another approach: learn a single pooling structure
- Pooling can complicate architectures that use top-down information
  - E.g., Boltzmann machines and autoencoders

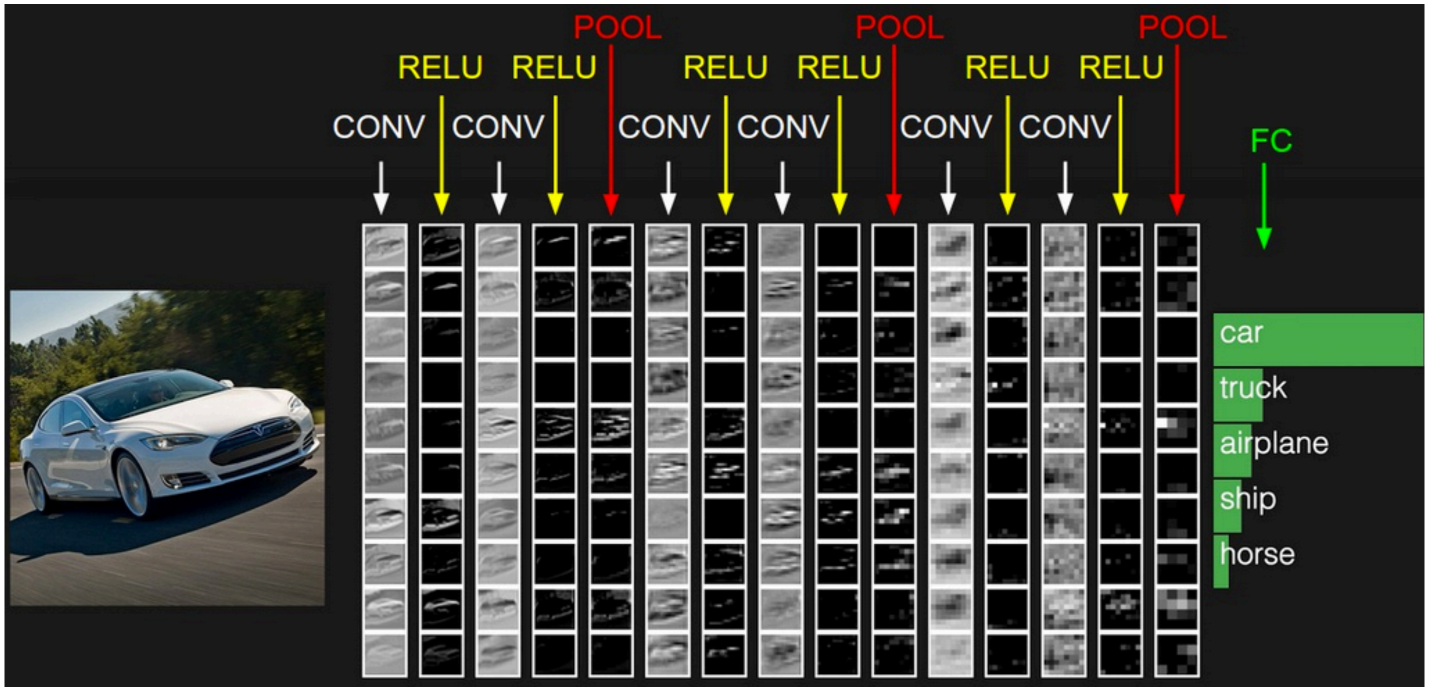
Examples of Architectures for Classification with CNNs

- Real networks have branching structures.
- Chain structures shown for simplicity


CNN that processes a fixed size image
Processes variable-sized image
Does not have any fully-connected layer
Output of softmax: 1,000 class probabilities
Output of matrix multiply: 1,000 units
Output of reshape to vector: 16,384 units
Output of pooling with stride 4: 16x16x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3
Output of softmax: 1,000 class probabilities
Output of matrix multiply: 1,000 units
Output of reshape to vector: 576 units
Output of pooling to 3x3 grid: 3x3x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3
Output of softmax: 1,000 class probabilities
Output of average pooling: 1x1x1,000
Output of convolution: 16x16x1,000
Output of pooling with stride 4: 16x16x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3

A ConvNet architecture

INPUT: 32x32x3 holds raw pixel values: an image of width 32, height 32 and 3 color channels RGB
CONV layer will compute the output of neurons connected to local regions in the input
Each computing a dot product between their weights and a small region they are connected to
In the input volume. This may result in a volume such as 32x32x12 if we used 12 filters
POOL layer will perform a down-sampling operation along spatial dimensions (width, height)
resulting in a volume such as 12x16x12

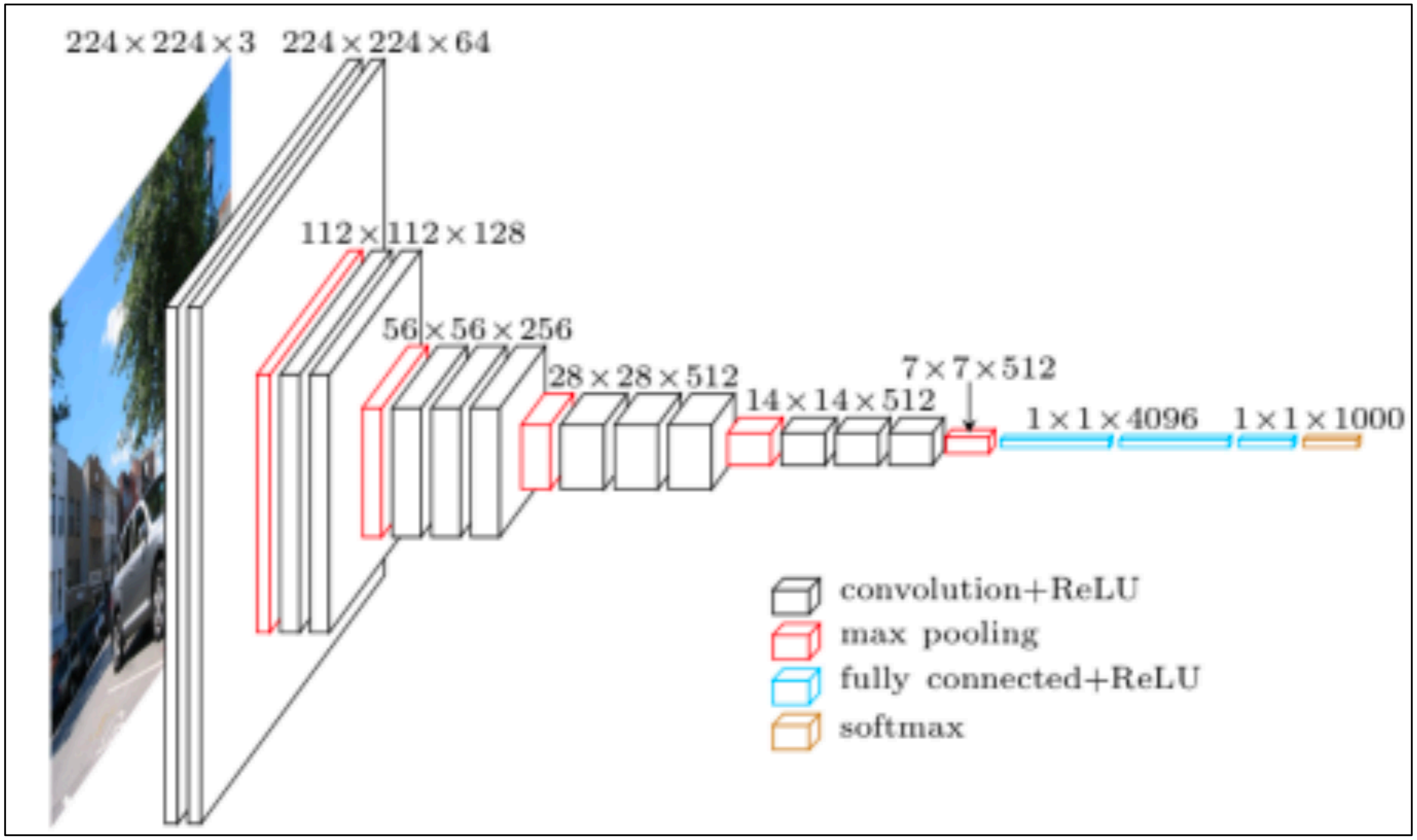


Activations of an example ConvNet architecture.
Initial volume stores raw image pixels (left) and the last volume stores class scores (right)
Each volume of activations along the processing path is shown as a column.
Since it is difficult to visualize 3D volumes, each volume's slices are laid out in rows

VGG Net

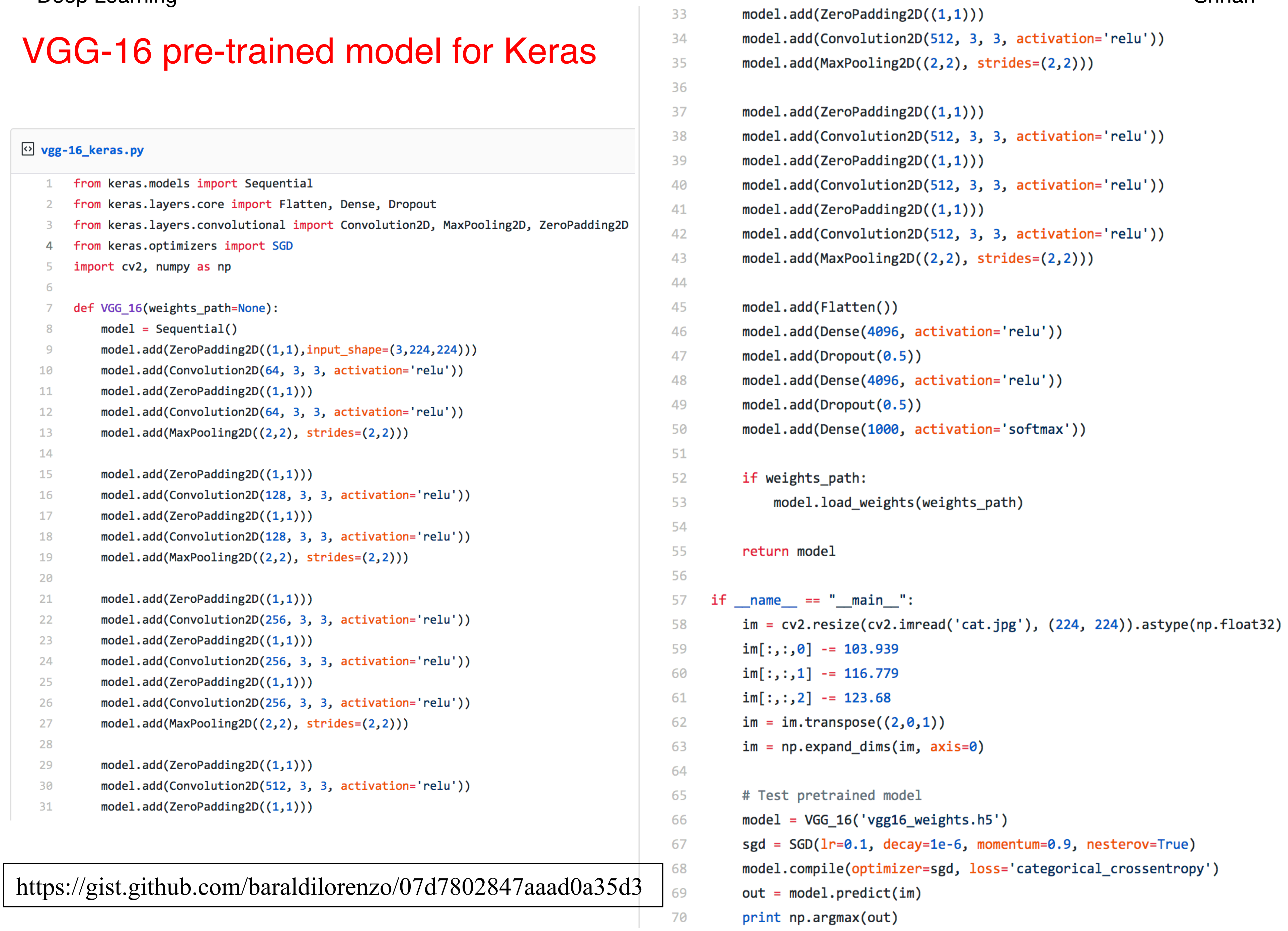
- VGG is a convolutional neural network model
  - K. Simonyan and A. Zisserman, University of Oxford
  - *“Very Deep Convolutional Networks for Large-Scale Image Recognition”*
- The model achieves 93% top-5 test accuracy in ImageNet
  - which is a dataset of over 14 million images belonging to 1000 classes.

VGG 16



Source: https://www.cs.toronto.edu/~frossard/post/vgg16/

VGG-16 pre-trained model for Keras



vgg-16_keras.py

```python
from keras.models import Sequential
from keras.layers.core import Flatten, Dense, Dropout
from keras.layers.convolutional import Convolution2D, MaxPooling2D, ZeroPadding2D
from keras.optimizers import SGD
import cv2, numpy as np

def VGG_16(weights_path=None):
    model = Sequential()
    model.add(ZeroPadding2D((1,1),input_shape=(3,224,224)))
    model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(64, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(128, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(128, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(256, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Convolution2D(512, 3, 3, activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))

    model.add(Flatten())
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(4096, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1000, activation='softmax'))

    if weights_path:
        model.load_weights(weights_path)

    return model

if __name__ == "__main__":
    im = cv2.resize(cv2.imread('cat.jpg'), (224, 224)).astype(np.float32)
    im[:,:,0] -= 103.939
    im[:,:,1] -= 116.779
    im[:,:,2] -= 123.68
    im = im.transpose((2,0,1))
    im = np.expand_dims(im, axis=0)

    # Test pretrained model
    model = VGG_16('vgg16_weights.h5')
    sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(optimizer=sgd, loss='categorical_crossentropy')
    out = model.predict(im)
    print np.argmax(out)
```

https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3