Deep Learning: Introduction to CNN

Sanjiv R. Das

http://srdas.github.io/DLBook/ConvNets.html
(http://srdas.github.io/DLBook/ConvNets.html)

In [1]:
```
%pylab inline
import pandas as pd
```

```
Populating the interactive namespace from numpy and matplotlib
```

Why CNNs?

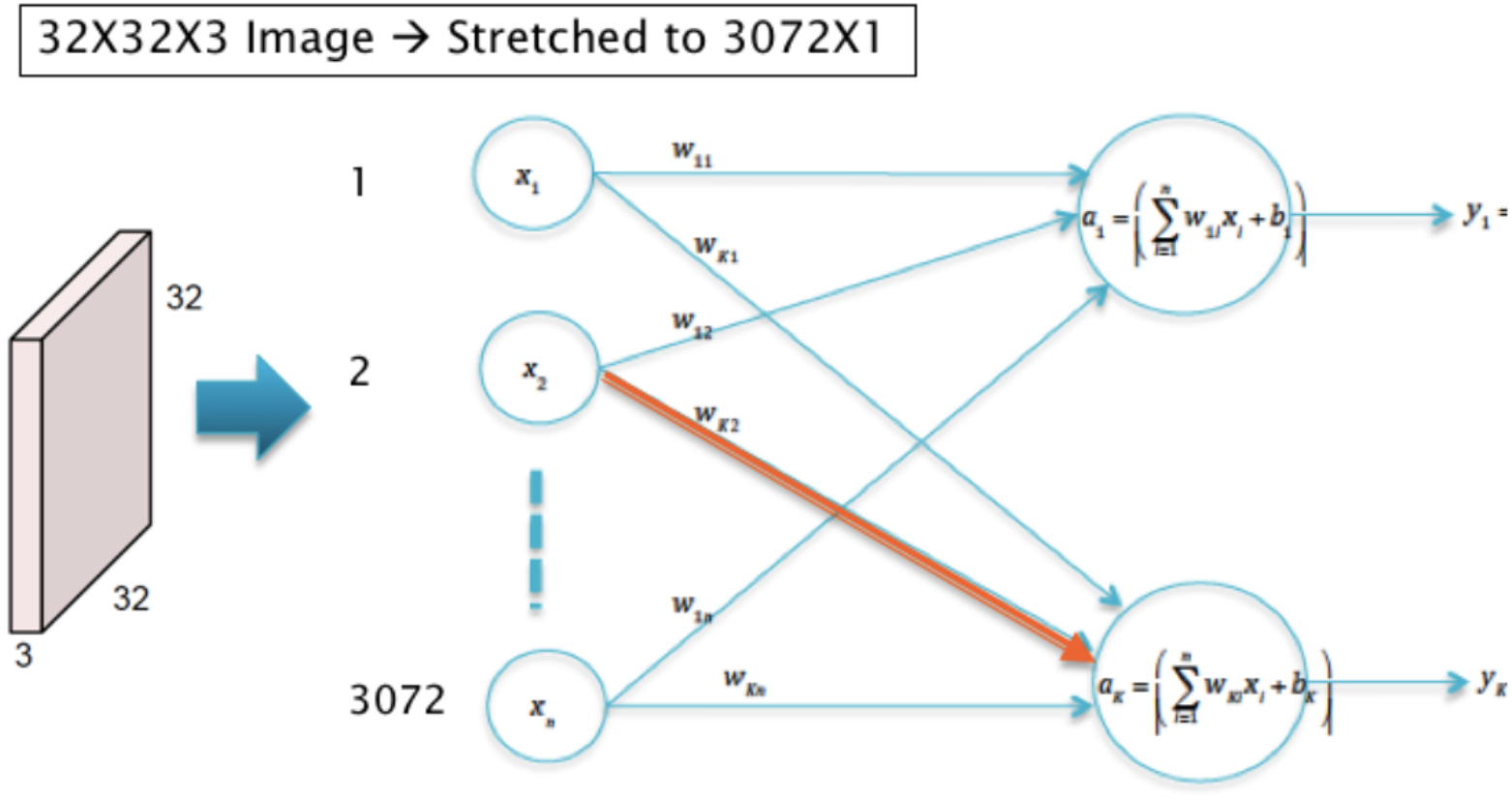
- Consider a typical image consisting of $200 \times 200 \times 3$ pixels, which corresponds to 3 layers of 200×200 numbers, one for each of the colors Red, Green and Blue. Since the input consists of $120,000$ numbers, these many weights are needed for each node in the first Hidden Layer of a fully connected DLN. Given a typical fully connected Deep Feed Forward Network with say 100 nodes in the first layer, this corresponds to 12 million weight parameters needed to describe just this layer.

- More parameters mean that more training data is required to prevent overfitting, which also leads to more time needed to train the model. On the other hand, when ConvNets are used to process images, it reduces the number of parameters by more than two orders of magnitude, thus improving accuracy and reducing training times.

- It has also been observed in practice that the accuracy of Fully Connected Deep Feed Forward Networks does not keep improving as more hidden layers are added, usually max-ing out at 4 to 5 layers. ConvNets however improve their accuracy with more hidden layers, indeed the most advanced ConvNet architecture features 150 layers!

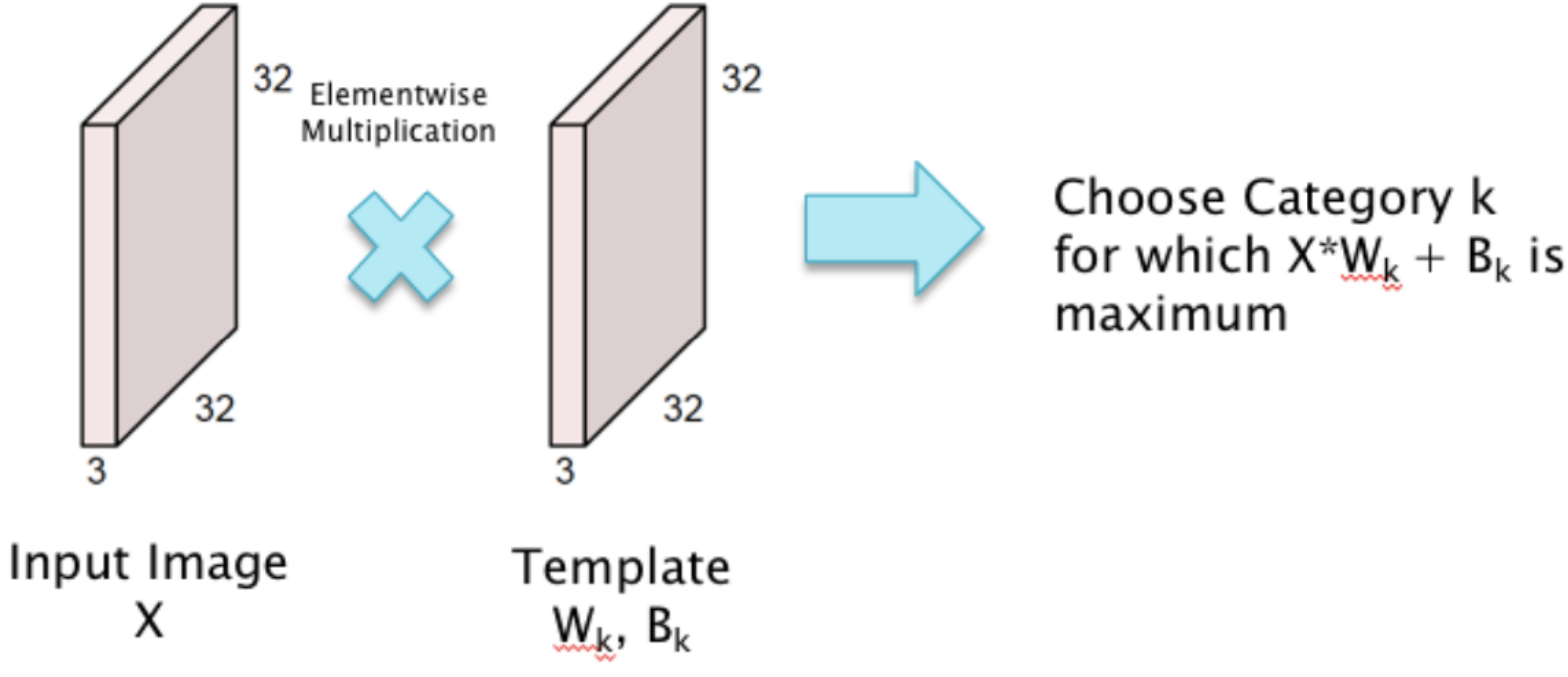
- Processing by Fully Connected Deep Feed Forward Networks requires that the image data be transformed into a linear 1-D vector. This results in a loss of structural information, such as correlation between pixel values that are in proximity of each other in 2-D. ConvNets on the other hand are able to process the original 2-D image data, which makes it easier for them to recognize shapes using template matching.

Fully Connected NN for images

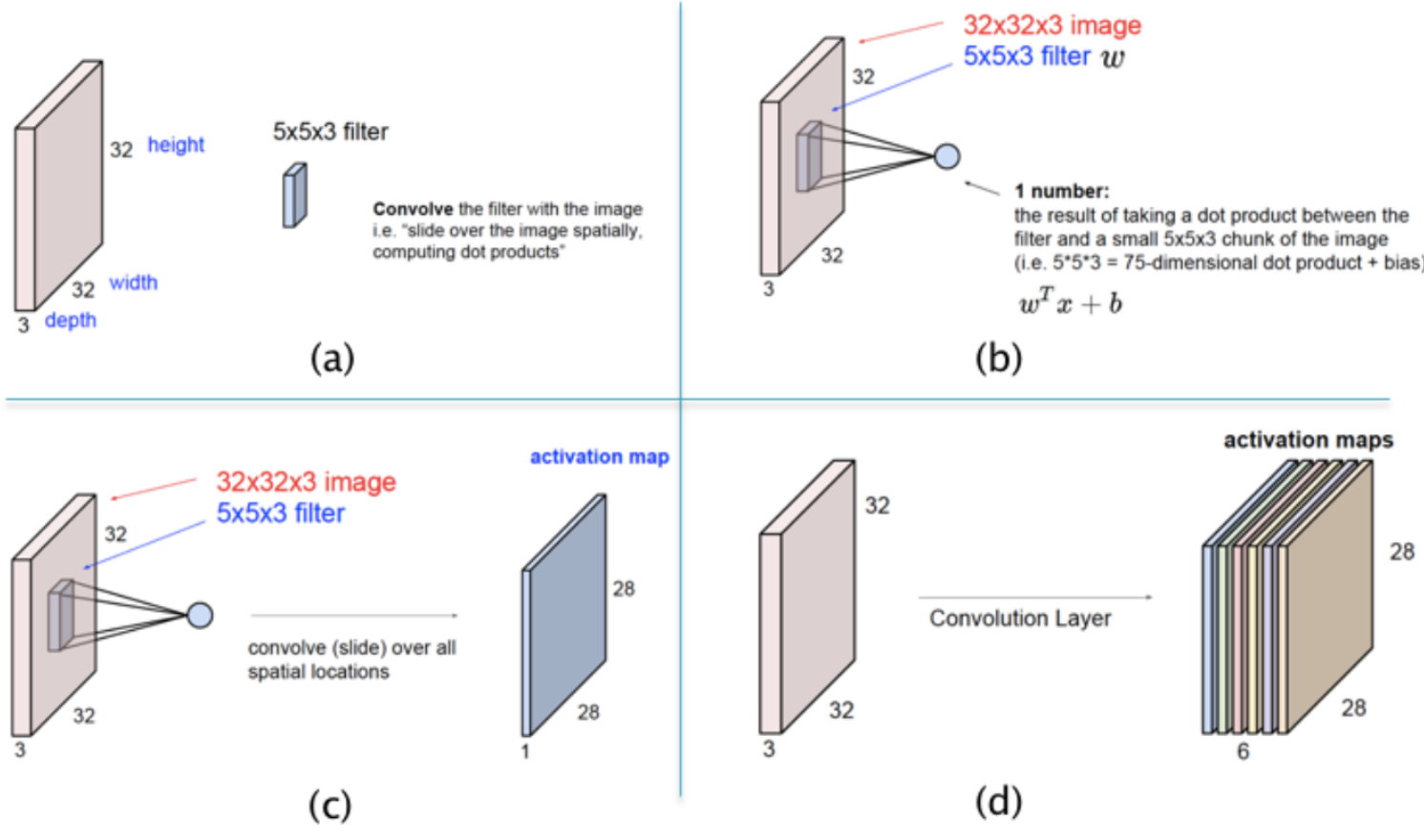


The pre-activations $a_k, i \leq k \leq 10$ in this model are given by

$$a_k = \sum_{i=1}^{3072} w_{ki} x_i + b_k, \quad 1 \leq k \leq 10$$



Local Filters and Feature Maps



1. Smaller templates need a smaller filter size and thus fewer parameters.
2. Even if the object being detected moves around the image, the same template or filter can still be used, i.e., translational invariance.

- Part (a) illustrates the difference between template matching in ConvNets vs Feed Forward Networks. ConvNets use a template (or filter) that is smaller than the size of the image in height and width, while the depths match. In the example shown in (a), a filter of size $5 \times 5 \times 3$ is used for an image of size $32 \times 32 \times 3$.

- Part (b) shows the template matching operation in ConvNets. As shown here, the matching is done locally, for possibly overlapping patches of the image. At each position of the filter, the template matching is done using the following equation to compute the pre-activation a and activation z:

$$a = \sum_{i=1}^{75} w_i x_i + b,$$

$$z = f(a)$$

- In this equation the pixel values $(x_1, \ldots, x_{75})$, known as the Local Receptive Field, correspond to the local image patch of size $5 \times 5 \times 3$ and changs as the filter is moved around (while the filter values w_i and b remain unchanged). Note that the filter now only has $5 \times 5 \times 3 + 1 = 76$ parameters, as opposed to the $32 \times 32 \times 3 + 1 = 3073$ parameters that were needed for the filter.

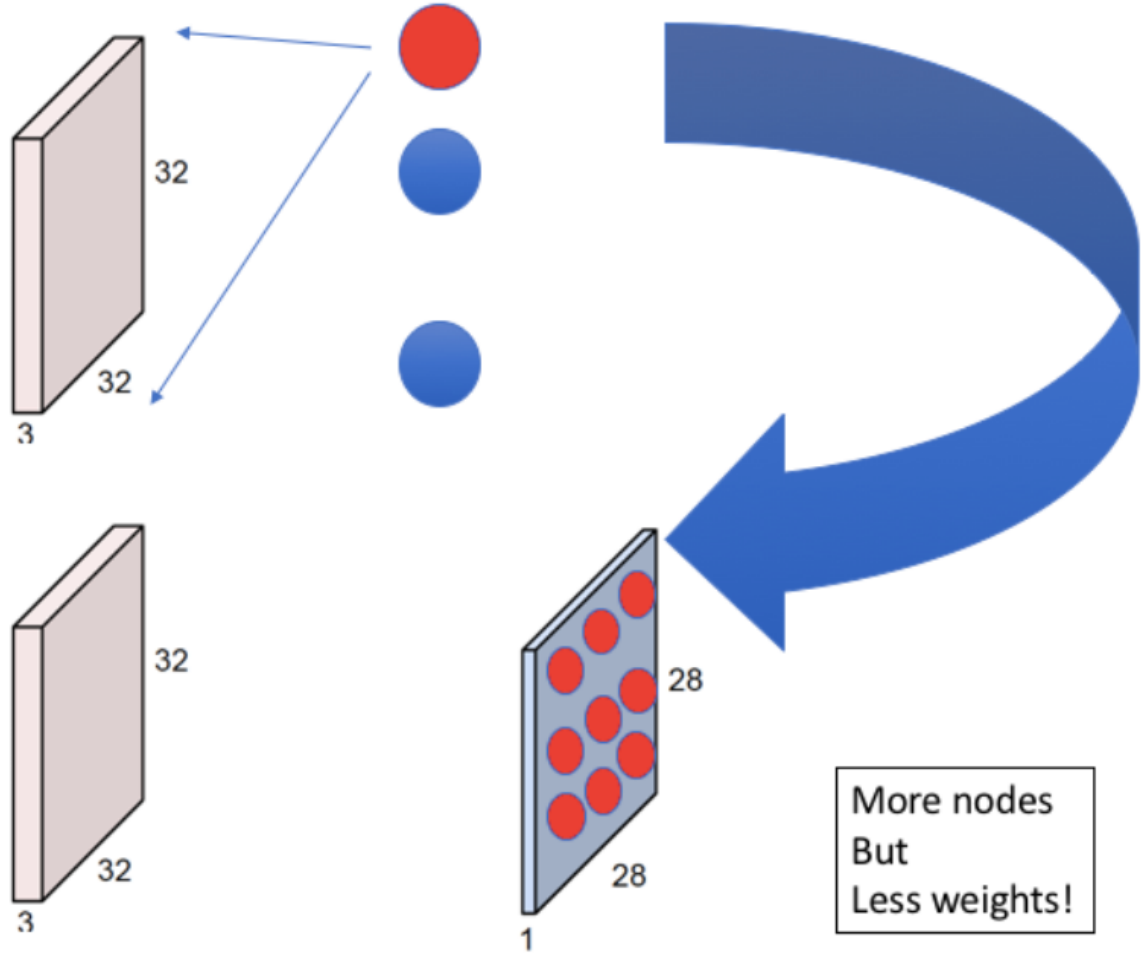
- Part (c) of the figure shows the filter being moved across the image, and at each position we compute a new value of z, and this generates a matrix of size 28×28. This matrix is known as an Activation Map (also called a Feature Map). This operation of sliding the filter across the image, while computing the dot product at each position, is called a **convolution**.

- Using the same Filter for all the nodes in the Activation Map implies that all nodes in the Map are tuned to detect the same feature in the Input Layer, *only at different positions in the image*. This leads to the conclusion that ConvNets possess the property of Translational Invariance, i.e., they are able to detect objects irrespective of their location in the image.

- Note that so far we have used a single filter which is only capable of detecting a single pattern in the input image. If we wish to detect multiple patterns, then we need multiple filters, each of which results in its own Activation Map, as shown in Part (d). For example, Activation Map 1 may detect horizonal edges while Activation Map 2 detects vertical edges etc. As shown, a Hidden Layer in ConvNets consists of a stack of Activation Maps.

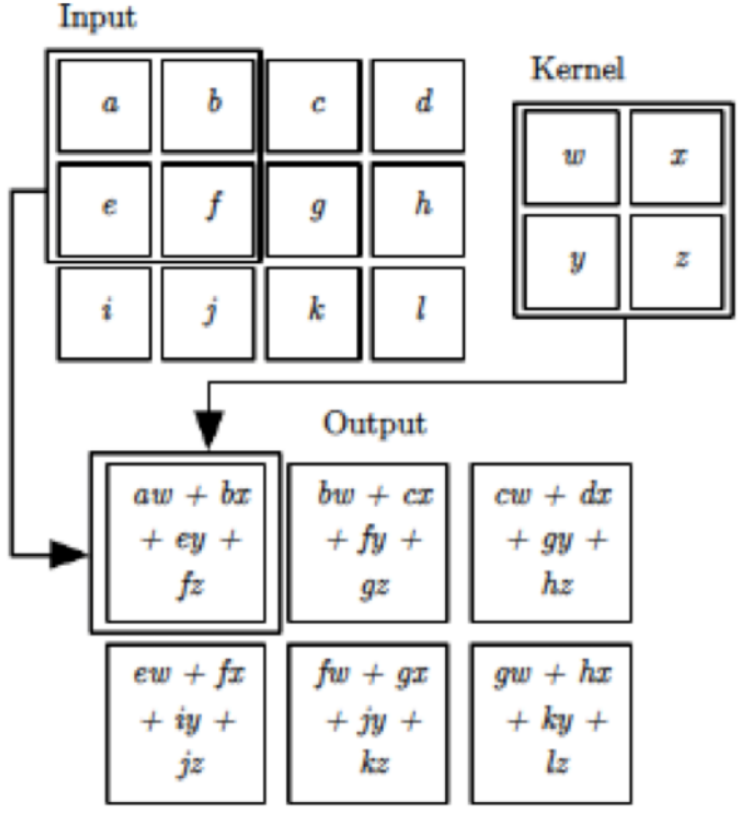
FF 2 CNN

A Node in the Old Architecture turns into an Activation Map in the New Architecture



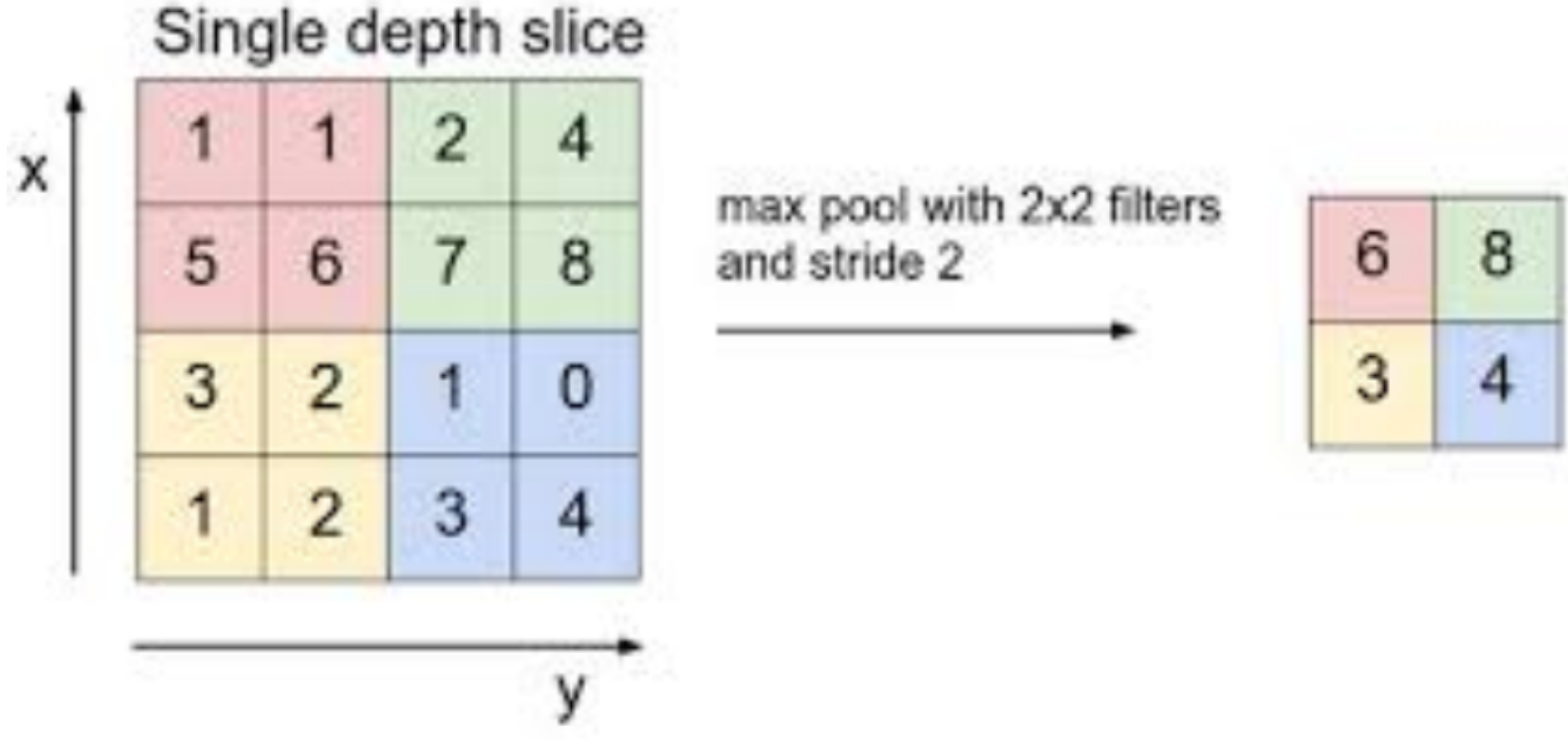
- Convnets are better positioned to detect smaller shapes, which are then built hierarchically into bigger shapes and objects as we go deeper into the network.
- The Translational Invariance property ensures that the shape is detected irrespective of its location in the image plane.
- ConvNets reduce the number of weights in the model at the expense of increasing the number of nodes. The increase in nodes causes the cost of computation to go up, but this is a worthwhile tradeoff to make since the reduction in the parameters makes the model easier to train with a smaller number of training examples.

Convolution



Stride (S), defined as the number of pixels by which the filter is moved after each computation, either horizontally or vertically. Note that the example in the Figure corresponds to $S = 1$.

Pooling



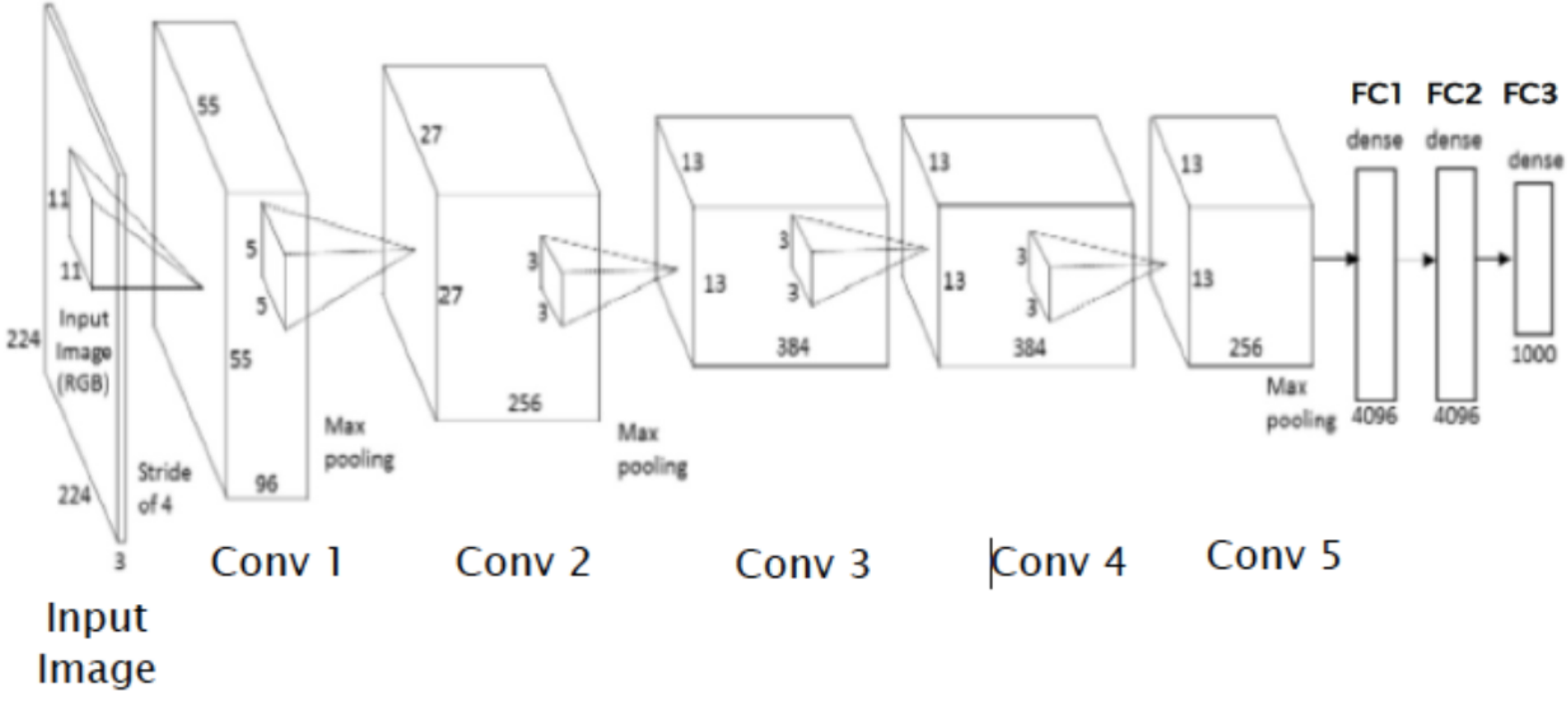
- Pooling usually occurs after a Convolutional Layer, and can be described as condensing the information from that layer into a smaller number of activations. As shown in the figure, Pooling involves replacing a set of activations within a region of the Activation Map (which is just like a Local Receptive Field), by a single number.
- Usually the maximum of the activation values in the Local Receptive Field is used for pooling (called max-pooling), but other functions can also be used, such as the L_1 or L_2 norm.
- Unlike in Local Receptive Fields used in Convolutions, the corresponding fields used for the Pooling operation do not overlap.
- Note that the addition of Pooling does not introduce any new parameters to the ConvNet and the total number of parameters in the model are reduced considerably due to this operation.

Pooling vs Convolution

- In order to understand the Pooling operation, note that the numbers in an Activation Map that results from the Convolution operation, correspond to the likelihood of whether a particular shape or pattern is present in various locations in the previous layer.
- By doing Pooling right after Convolution, we throw away some of that information, which is the same as saying that the network does not care about the exact location of a pattern, it only needs to know whether the pattern is present or not.
- It should also be mentioned that as more processing power becomes available, some modern ConvNets, such as ResNet and the Google Inception Network, no longer incorporate Pooling in their design.

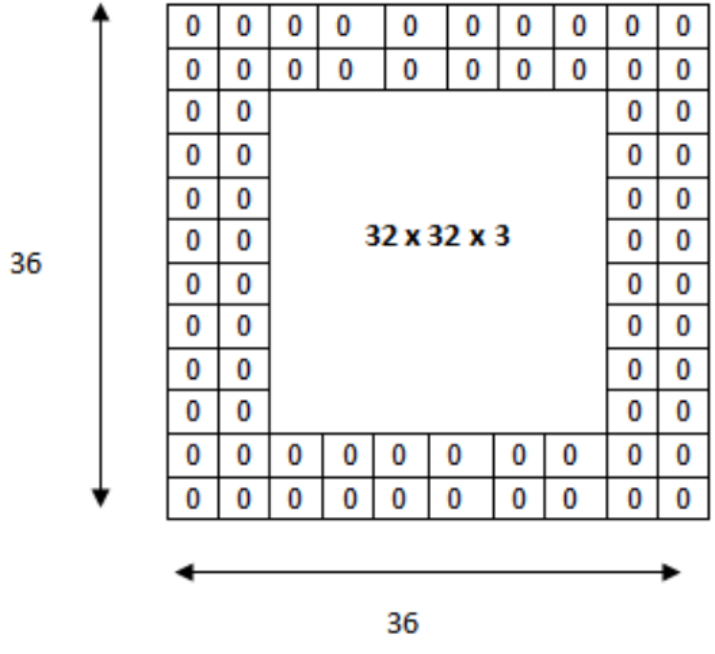
AlexNet

Won the ILSVRC 2012 challenge.



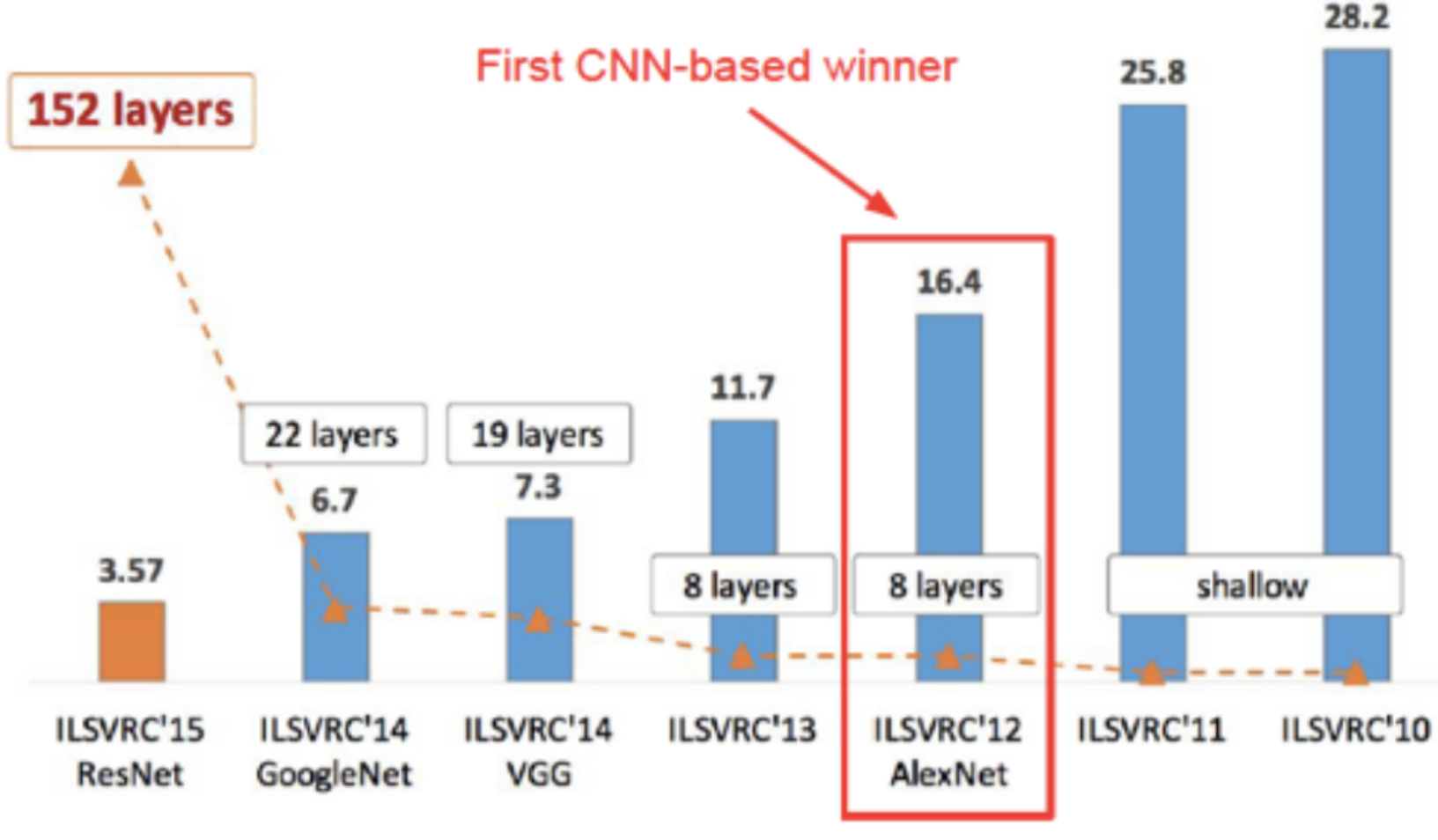
Zero-padding

To ensure that volume sizes remain the same as we progress through the network.

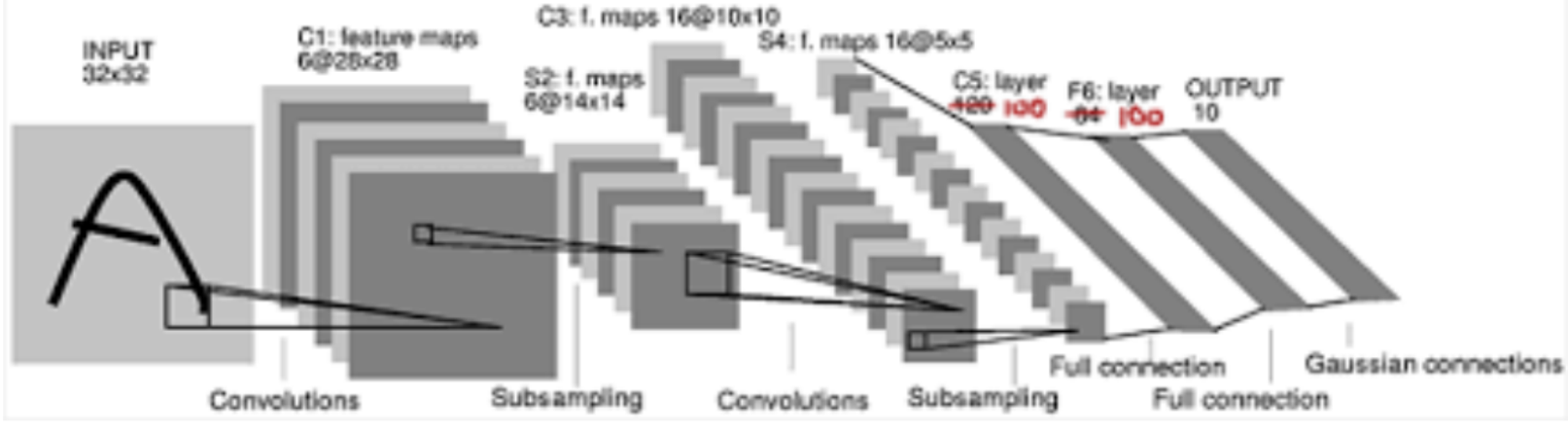


The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x3 output volume.

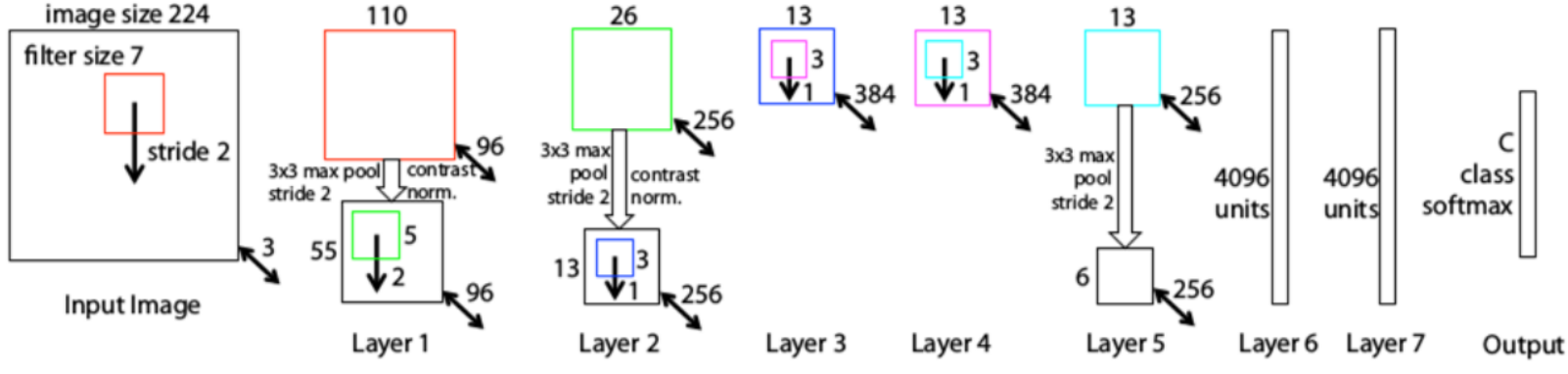
Network size and error rate in ILSVRC



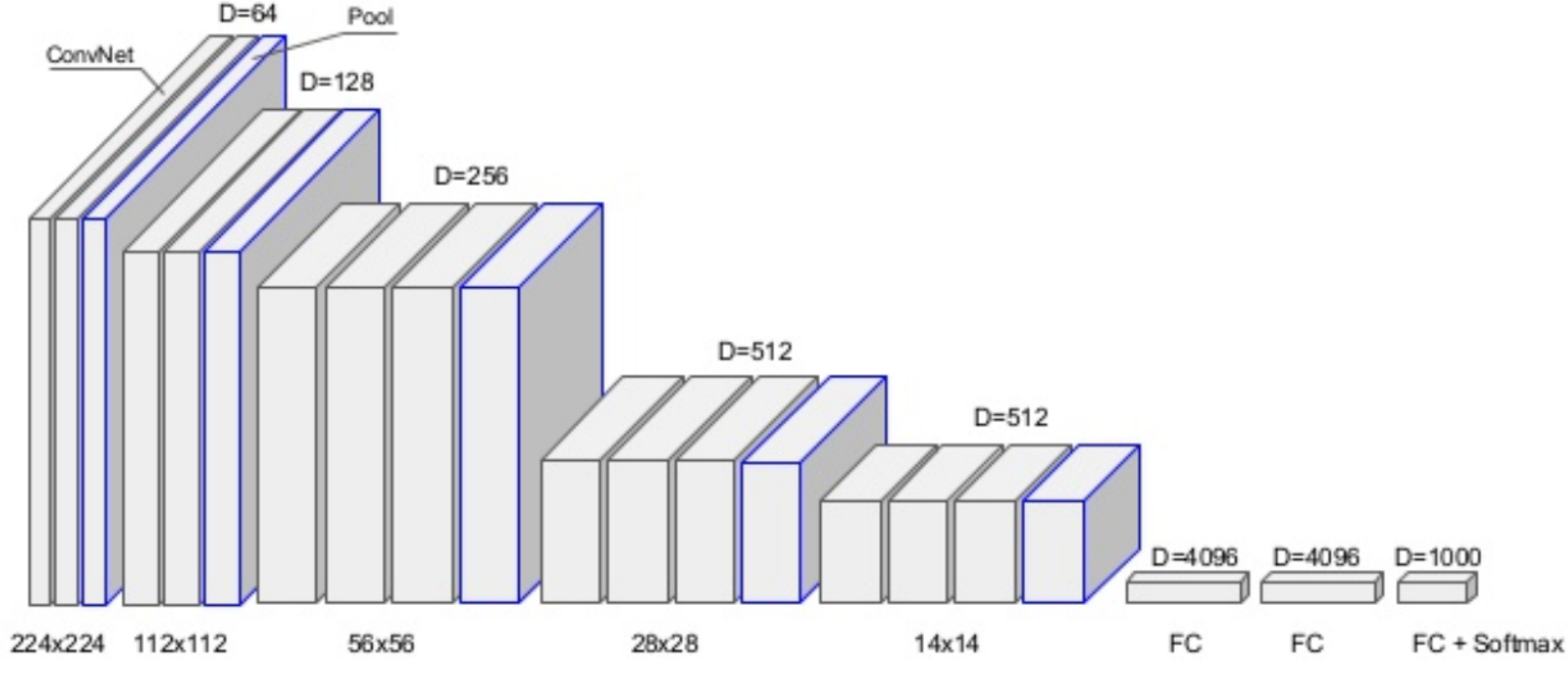
LeNet (1998)



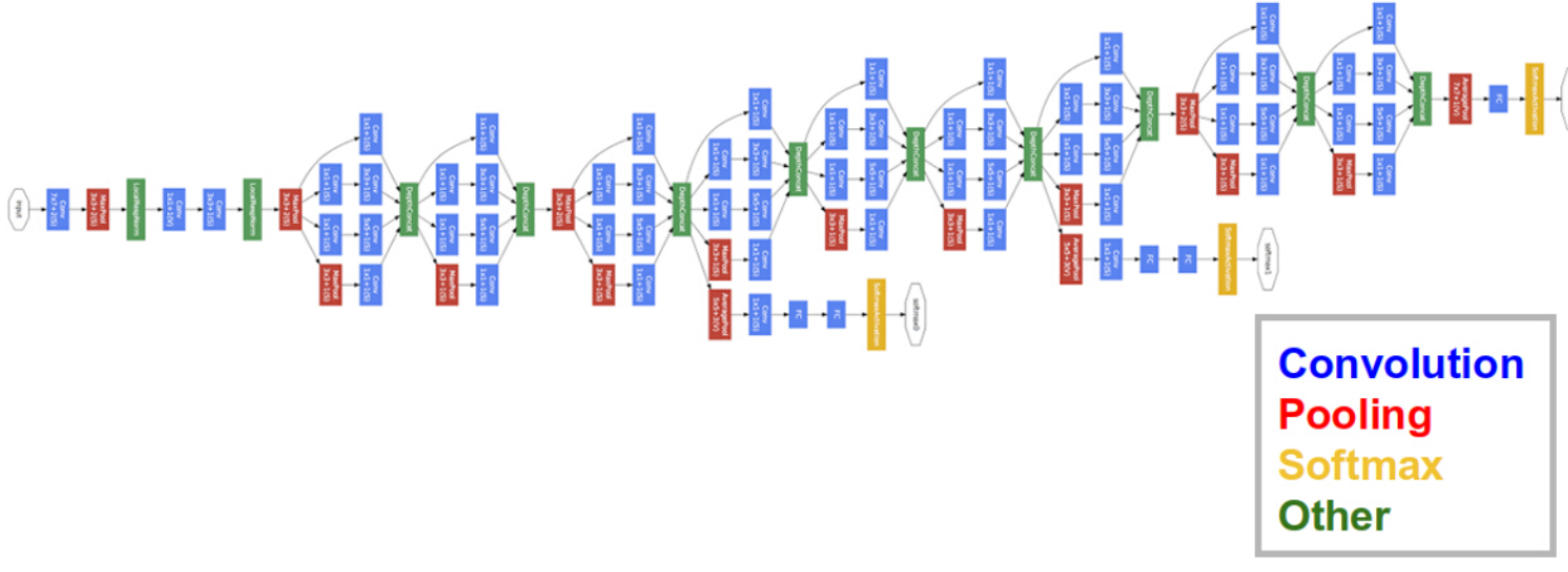
ZFNet (2013)



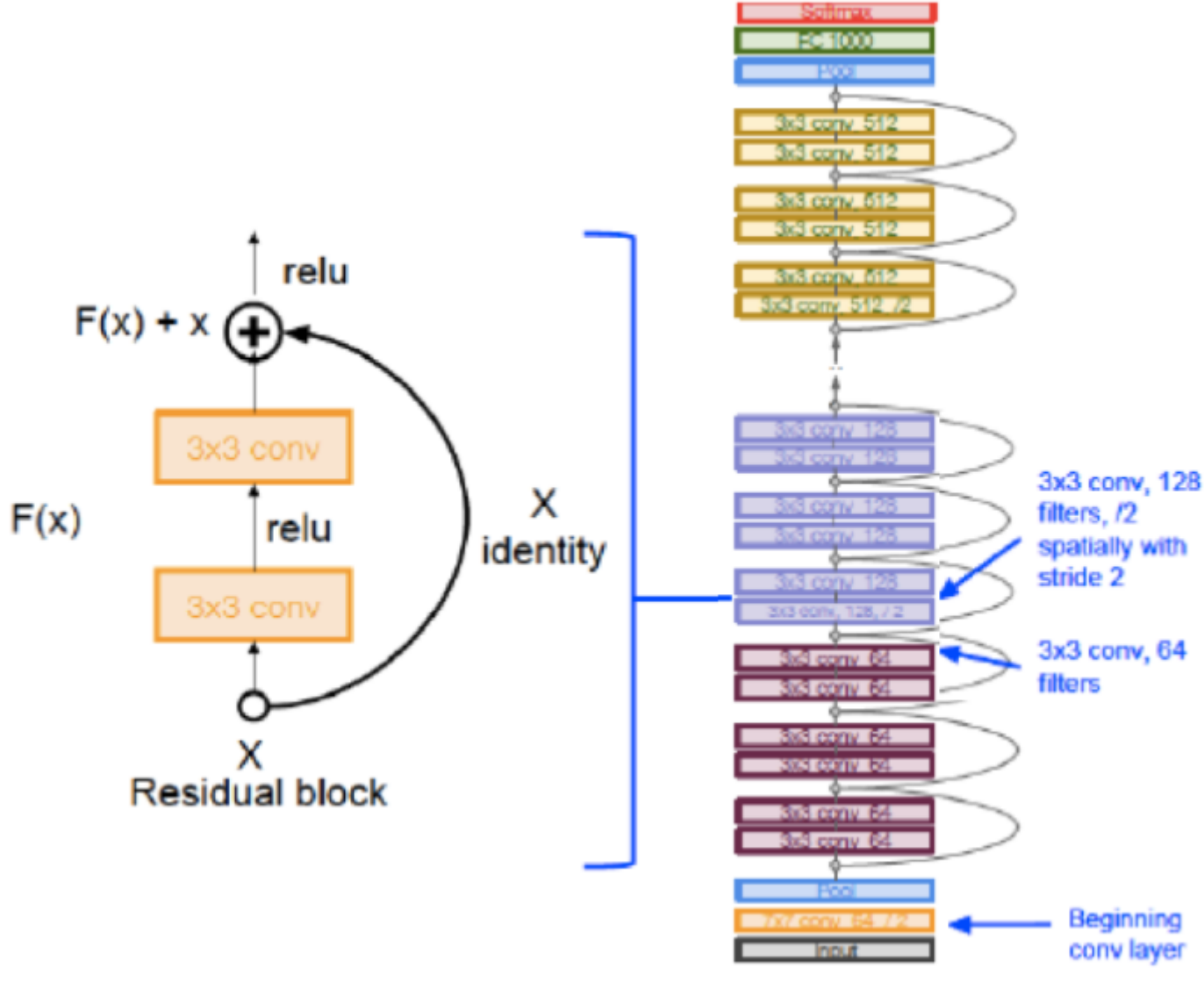
Classical CNN topology - VGGNet (2013)



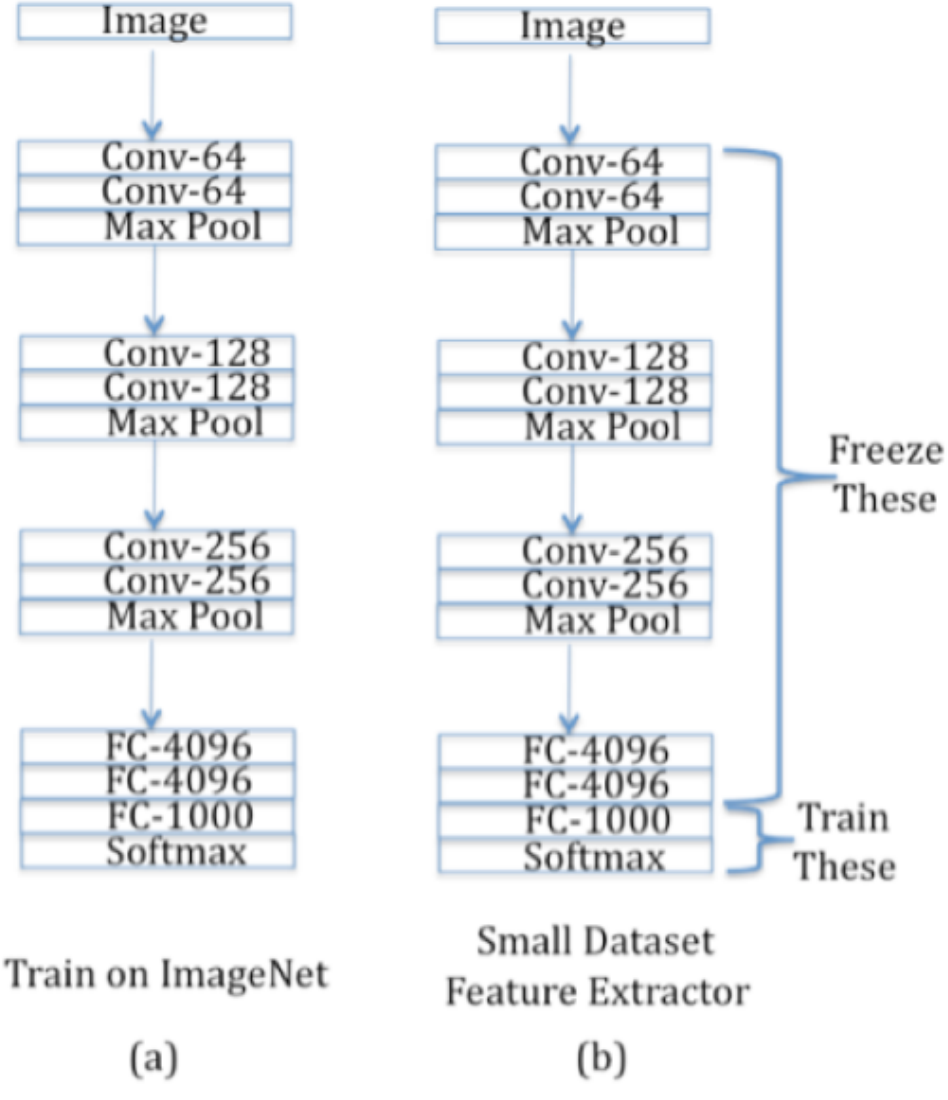
Google Inception Net (2014)

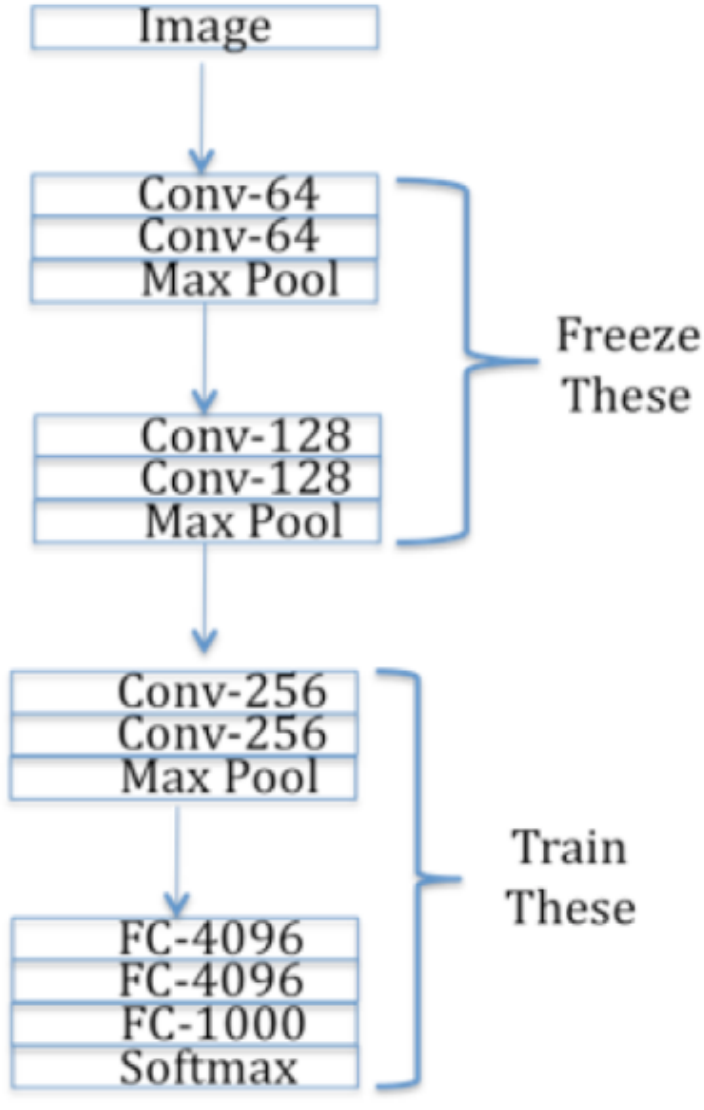


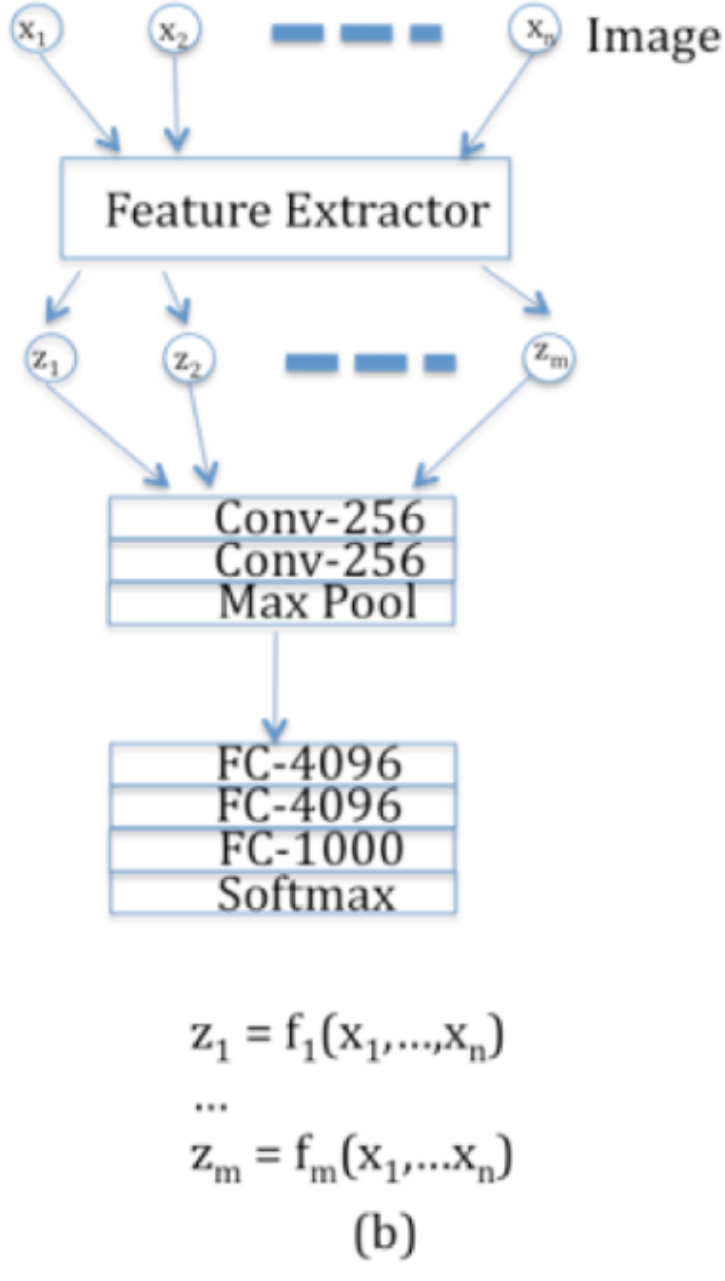
ResNet (2015)



Transfer Learning







In [2]:
```
## Implementation

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
```

```
/Users/srdas/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: Future
Warning: Conversion of the second argument of issubdtype from `float` to `np.f
loating` is deprecated. In future, it will be treated as `np.float64 == np.dty
pe(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

In [3]:
```
data = mnist.load_data()
x_train = data[0][0]
y_train = data[0][1]
x_test = data[1][0]
y_test = data[1][1]
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

In [4]:
```
# image dimensions
img_rows, img_cols = x_train.shape[1:3]
print(img_rows)
print(img_cols)
```

```
28
28
```

In [5]:
```
#Reshape the data

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

print(input_shape)
```

```
(28, 28, 1)
```

In [6]:
```
#Normalization

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train/255.0
x_test = x_test/255.0
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

In [7]:
```
#One-hot encoding

num_categories = 10
y_train = keras.utils.to_categorical(y_train, num_categories)
y_test = keras.utils.to_categorical(y_test, num_categories)
print(y_train.shape)
```

```
(60000, 10)
```

In [8]:
```
# CNN

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_categories, activation='softmax'))
```

In [15]:
```
#Compile

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

In [16]:
```
#Fit the model

epochs = 5
batch_size = 128
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 209s 3ms/step - loss: 0.2594 -
acc: 0.9199 - val_loss: 0.0536 - val_acc: 0.9821
Epoch 2/5
60000/60000 [==============================] - 230s 4ms/step - loss: 0.0848 -
acc: 0.9755 - val_loss: 0.0384 - val_acc: 0.9862
Epoch 3/5
60000/60000 [==============================] - 255s 4ms/step - loss: 0.0641 -
acc: 0.9808 - val_loss: 0.0334 - val_acc: 0.9891
Epoch 4/5
60000/60000 [==============================] - 284s 5ms/step - loss: 0.0529 -
acc: 0.9841 - val_loss: 0.0301 - val_acc: 0.9899
Epoch 5/5
60000/60000 [==============================] - 297s 5ms/step - loss: 0.0456 -
acc: 0.9863 - val_loss: 0.0312 - val_acc: 0.9893
```

Out[16]:
```
<keras.callbacks.History at 0xb23c77908>
```

In [17]:
```
#Evaluate the model

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.031190014394620085
Test accuracy: 0.9893
```