# General ConvNet Notes

Sean Wu

October 6, 2019

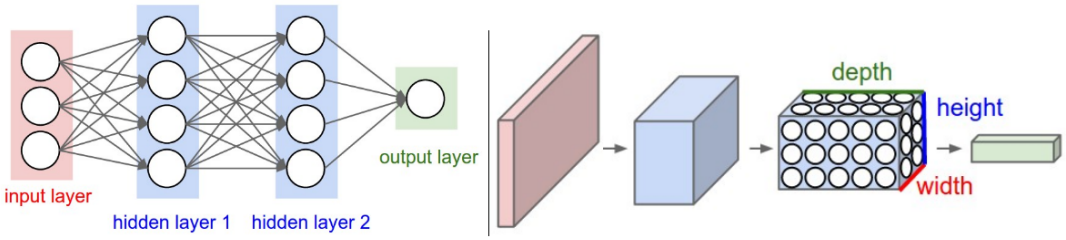
## Contents

# 1 ConvNet Structure

## 1.1 ConvNet Layers

**Convolutional Neural Net (ConvNet):** specialized neural nets

- Explicitly assume that inputs are images (reduces params req)
- Regular neural nets don't scale well w/ images because they maintain full connectivity for all neurons $\implies$ many parameters and overfitting
- ConvNets have layers w/ neurons called **activation volumes** arranged in 3 dimensions (**length**, **height**, **depth**)
- Each layer transforms an input 3D volume to an output 3D volume w/ some differentiable function that may/may not have parameters
- ConvNets transform image layer by layer
- Params are trained w/ gradient descent so class scores that the ConvNet computes are consistent w/ training labels




Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Figure 1: Illustration of ConvNet layers as Volumes

## 1.2 Main ConvNet Layer Types

1. Convolutional Layer (CONV)
2. Pooling Layer (POOL)
3. Fully-Connected Layer (FC)

- CONV/FC/POOL have hyperparameters, RELU doesn't

## 1.3 Example ConvNet Architecture

- Ex. ConvNet for CIFAR-10 classification
- INPUT $\longrightarrow$ CONV $\longrightarrow$ RELU $\longrightarrow$ POOL $\longrightarrow$ FC

1. **INPUT** ($32 \times 32 \times 3$): takes raw pixel values of $32 \times 32$ RGB image
2. **CONV** ($32 \times 32 \times 12$ if using 12 filters): computes output of neurons connected to input w/ dot product betw their weights and the filter
3. **RELU** ($32 \times 32 \times 12$): applies elementwise activation function (ex. $f(x) = max(0, x)$)
4. **POOL** ($16 \times 16 \times 12$): downsamples along width & height (depth unchanged)
5. **FC** ($1 \times 1 \times 10$): computes class scores (1 neuron per class score)

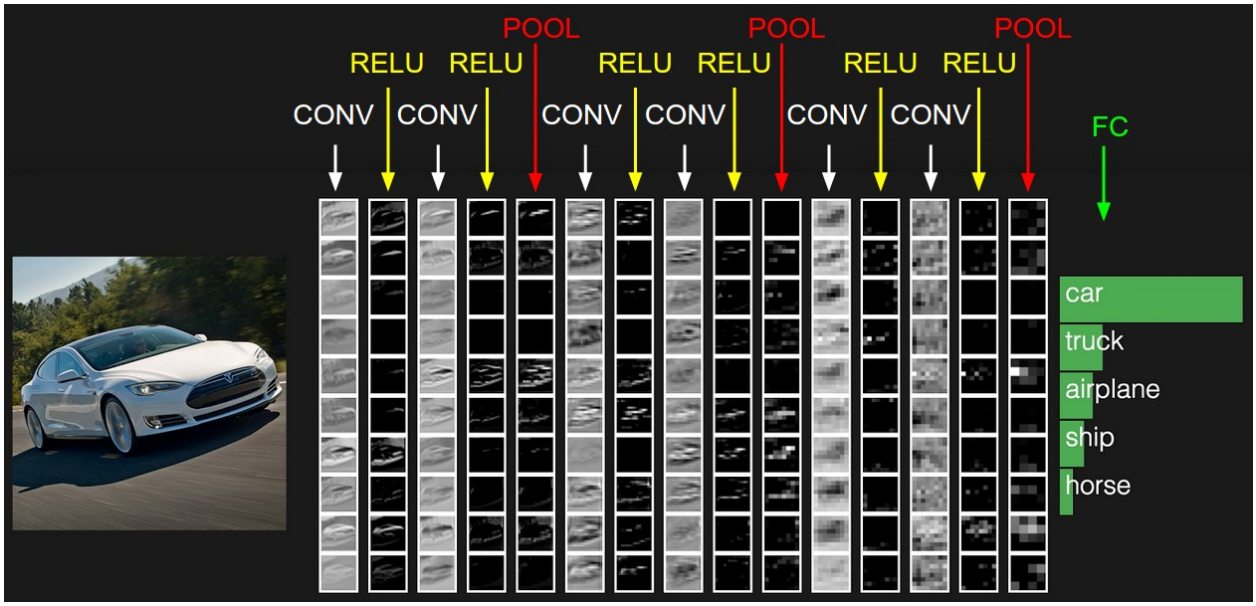



Figure 2: Illustration of ConvNet layers as Volumes

## 1.4 Convolutional Layer (CONV)

- Input Volume: $W_1 \times H_1 \times D_1$
- Consists of filters that are convolved across width and height of input volume during forward pass
  - Computes dot product betw entries of filter and input at any position
  - Produces 2D activation maps that are stacked along depth dimension that give responses of a filter at each any position
- Helps ConvNet learn filters that activate when some type of visual feature is detected (ex. edge, specific colour, etc)
- Requires 4 hyperparams

1. # of filters, **K**
2. Spatial extent (size of filter), **F**
3. Stride (distance betw centres of filters), **S**
4. Zero padding, **P**

- Output Volume: $W_2 \times H_2 \times D_2$

**Output Volume Equations**

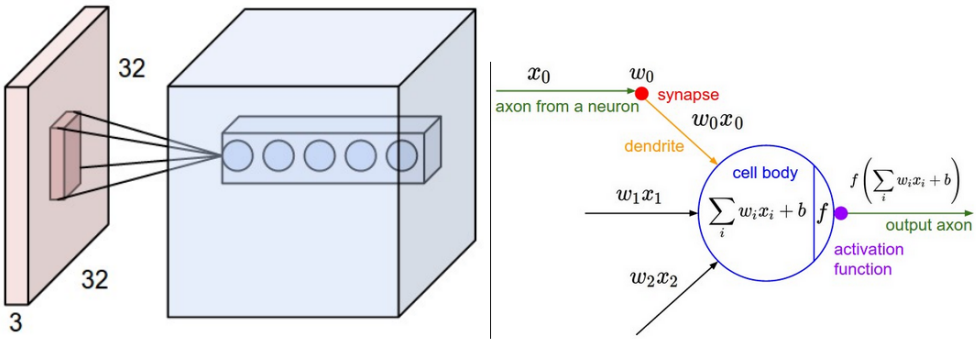
$$W_2 = (W_1 - F + 2P)/S + 1 \tag{1}$$
$$H_2 = (H_1 - F + 2P)/S + 1 \tag{2}$$
$$D_2 = K \tag{3}$$

- With parameter sharing, the CONV introduces $F \cdot F \cdot D_1$ weights per filter for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K baises
- In output volume, the d-th slice $(W_2 \cdot H_2)$ is result of convolution of d-th filter over input volume w/ stride S and then offset by dth biases

**Common Hyperparameter Settings:** $F = 3$, $S = 1$, $P = 1$

**Receptive field (filter):** hyperparameter representing spatial extent of connectivity of each neuron to a local region of input volume

- Extent of connectivity along depth axis equals depth of input volumes




**Left:** An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input - see discussion of depth columns in text below. **Right:** The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

Figure 3: Illustration of ConvNet layers as Volumes

## 1.5 Spatial Arrangement of CONV Neurons

- 3 hyperparameters control size of output volume

1. Depth (equivalent to # of filters, $D$)
2. Stride ($S$)
3. Zero-padding ($P$)

**Depth:** corresponds to # of filters

- Each filter learns to look for something different in the input

**Depth column:** set of neurons that are all looking at the same region of the input
**Stride** Speed at which filter is moved

- Stride of $S = 1$ moves filters 1 pixel at a time
- Larger strides produce smaller output volumes spatially

- Note: stride is constrained by $W$, $F$, $P$ since $(W - F + 2P)/S$ must be an integer
- Otherwise, neurons don't fit symmetrically across input
- Considered invalid setting of hyperparameters and ConvNet library could throw exception, zero pad, or crop to make it fit

- If stride is $S = 1$, usually set zero padding as $P = (F - 1)/2$ so input and output volume have same size

## 1.6 Parameter Sharing

- Parameter sharing is used in CONV to reduce # of params
- Assumes if a feature is useful to computer at some spatial position $(x, y)$ then it should also be useful to compute it at a different position $(x_2, y_2)$
- i.e. denote a single 2D slice of depth as a **depth slice** and set its neurons to use the same weights and biases (1 depth slice per filter)
- Thus CONV layer will only have 1 weight per depth slice
- During backpropagation, the gradient for every neuron's weights are added up across each depth slice but only a single set of weights is updated per slice
- ex. a $55 \times 55 \times 96$ input has 96 depth slices ($55 \times 55$) and uses only 96 unique weights
- Note: if all neurons in a single depth slice use the same weight vector, the forward pass of CONV layer in each depth slice can be computed as a **convolution** of neurons weights w/ input volume

- Usually refer to the sets of weights as a **filter** or **kernel** that is convolved w/ input

- Param sharing does not work when we expect completely diff features on each side of input image
- Use less strict param sharing scheme (called a **locally-connected layer**)

### 1.6.1 Numpy Examples

- Input volume: numpy array `X`
- Depth column at position $(x, y)$: `X[x,y,:]`
- Depth slice/activation map at depth $d$: `X[:,:,d]`

- ex. input volume `X` has shape `X.shape: (11,11,14)` and uses zero padding $P = 0$, filter size $F = 5$, and stride $S = 2$
- Output volume has size: $(11 - 5)/2 + 1 = 4$ giving volume width & height of 4

```
# First activation map in output volume V w/ parameter sharing
# W0 is the neuron's weight vector and b0 is the bias
# Dimensions along width are increasing in steps of 2 (stride)
# Each line rep dot product of 1 channel of image multiplied by filter W0
V[0,0,0] = np.sum(X[:5,:5,:]) * W0) + b0
V[1,0,0] = np.sum(X[2:7,:5,:] * W0) + b0
V[2,0,0] = np.sum(X[4:9,:5,:] * W0) + b0
V[3,0,0] = np.sum(X[6:11,:5,:] * W0) + b0
```

- Note: `*` in numpy means elementwise multiplication

```
# Second activation map in output volume V w/ parameter sharing
# Note: Some steps (including activation functions like ReLU) are not shown
V[0,0,1] = np.sum(X[:5,:5,:] * W1) + b1
V[1,0,1] = np.sum(X[2:7,:5,:] * W1) + b1
V[2,0,1] = np.sum(X[4:9,:5,:] * W1) + b1
V[3,0,1] = np.sum(X[6:11,:5,:] * W1) + b1
V[0,1,1] = np.sum(X[:5,2:7,:] * W1) + b1 #(example of going along y)
V[2,3,1] = np.sum(X[4:9,6:11,:] * W1) + b1 #(or along both)
```

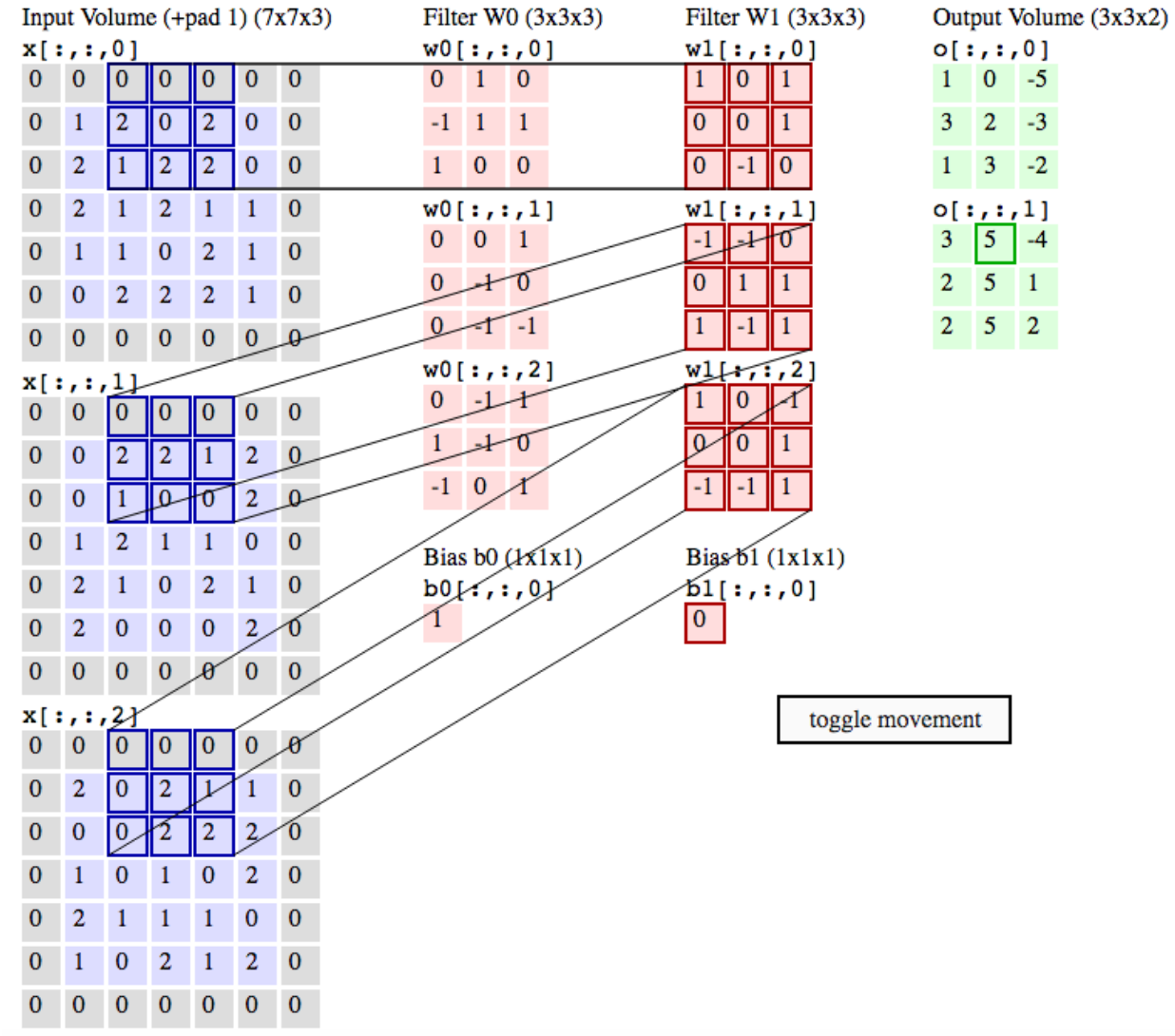



Figure 4: Visualization of above numpy code for convolutions

## 2 Convolutions as Matrix Multiplication

- Can treat convolution operation as a big matrix multiply of filters and local regions of input

1. Local regions in input image stretched into columns using `im2col`
   - ex. for $227 \times 227 \times 3$ input convolved w/ $11 \times 11 \times 3$ filters at stride $S = 4$, take the $11 \times 11 \times 3$ blocks of pixels in inputs and stretch block into a $11 \times 11 \times 3 = 363$ column vector
   - Repeating process w/ stride $S = 4$ gives $(227 - 11)/4 + 1 = 55$ locations along both width and height, leading to output matrix `X_col` of size $363 \times 3025$ (Note: $11 \cdot 11 \cdot 3 = 363$ and $55^2 = 3025$) where every column is a stretched out receptive field/filter
2. Weights of CONV layer are similarly stretched into rows

3. Result of convolution is now equivalent to `np.dot(W/_row, X_col)` which evaluates the dot product betw every filter and receptive field location

- ex. output is $96 \times 3025$ matrix

4. Result must finally be reshaped into proper output dimensions $55 \times 55 \times 96$

**Backpropagation:** backward pass for convolution operation (both data & weights) is also a convolution but w/ spatially-flipped filters

**Dilated Convolutions:** use filters w/ spaces betw each cell (dilation)

- Dilation is a hyperparam
- ex. dilation $= 0 \implies$ `w[0]*x[0] + w[1]*x[1] + w[2]*x[2]` dilation $= 1 \implies$ `w[0]*x[0] + w[1]*x[1] + w[2]*x[2]`
- Can use dilation=1 w/ dilation=0 to merge spatial info across inputs w/ fewer layers
- ex. if you stack two $3 \times 3$, the neurons on 2nd layer are a function of a $5 \times 5$ patch of input (effective receptive field is $5 \times 5$)

# 3 Pooling Layer

- Usually insert Pooling layer betw successive CONV layers
- Reduces spatial size of representation to reduce amount of params and computation in network $\implies$ control overfitting
- Operates indep on every depth slice and resizes it using MAX operation
- Usually use filters of size $2 \times 2$ applied w/ stride $S = 2$, which downsamples depth slice input by 2 along height & width

– Takes max of 4 numbers and reduces size to $frac14$ of original (depth stays constant)

- Input size: $W_1 \times H_1 \times D_1$
- Requires 2 hyperparameters

1. Spatial extent $F$
2. Stride $S$

- produces a volume of size $W_2 \times H_2 \times D_2$ where

$$W_2 = (W_1 - F)/S + 1 \tag{4}$$
$$H_2 = (H_1 - F)/S + 1 \tag{5}$$
$$D_2 = D_1 \tag{6}$$

- Introduces 0 params since it computes a fixed function of inputs
- Usually do not zero pad
- 2 common pooling layers

1. $F = 2$, $S = 2$
2. $F = 3$, $S = 2$

- Pooling sizes w/ larger receptive fields are too destructive (too much info loss)

### 3.1 General Pooling

- Pooling units can be used for max pooling, average pooling, or L2-norm pooling
- Used to use average, but now use max pooling

**Backpropagation:** backward pass for a $max(x, y)$ only routes the gradient it the input that had the highest value in the forward pass

- Usually track index of max activation (switches) in the forward pass of pooling player to keep gradient routing efficient during backpropagation
- Some people avoid using pooling and use repeated CONV layers instead
  - To reduce size of representation, use larger stride in CONV layer once in a while
- Discarding pooling layers important for good generative models (ex. variational autoencoders, generative adversarial networks)

## 4 Fully Connected Layers

- Neurons in FC have full connections to all activation in previous layer
- Can compute activations w/ matrix multiplications w/ bias offset

### 4.1 Converting FC layers to CONV layers

- Only difference is that CONV layer connected only to local region in input and many neurons share params
- Same functional form (dot products)
- For any CONV layer, there is an FC layer w/ same forward function
  - Weight matrix is large matrix that is mostly zero except at certain blocks (bec local connectivity) and where weights in many of the blocks are equal (bec of param sharing)
- Can express any FC as CONV layer
  - ex. for FC layer $K = 4096$ for input volume $7 \times 7 \times 512$, equivalent CONV is $F = 7$, $P = 0$, $S = 1$, $K = 4096$
  - i.e set filter size to be exactly size of input volume $\implies$ output is $1 \times 1 \times 4096$ since only a single depth column fits across the input volume

**FC $\longrightarrow$ CONV conversion:** Useful to convert FC to CONV bec can slide original ConvNet efficiently across many spatial positions in a larger image in a single forward pass

- For better performance, can resize an image to make it bigger, use a converted ConvNet to evaluate class at many spatial positions and then average class scores

# 5 ConvNet Architectures

**ReLU:** activation function which applies elementwise non-linearity

**Most Common ConvNet:** INPUT $\longrightarrow$ [CONV $\longrightarrow$ RELU]*N $\longrightarrow$ POOL?]*M $\longrightarrow$ [FC $\longrightarrow$ RELU]*K $\longrightarrow$ FC where $N \geq 0$ (usually $N \leq 3$), $M \geq 0$, $K < 3$

Examples:

- INPUT $\longrightarrow$ FC: linear classifier ($N = M = K = 0$)
- INPUT $\longrightarrow$ CONV $\longrightarrow$ RELU $\longrightarrow$ FC
- INPUT $\longrightarrow$ [CONV $\longrightarrow$ RELU $\longrightarrow$ POOL]*2 $\longrightarrow$ FC $\longrightarrow$ RELU $\longrightarrow$ FC‘
- Single CONV layer betw every pool
- INPUT $\longrightarrow$ [CONV $\longrightarrow$ RELU $\longrightarrow$ CONV $\longrightarrow$ RELU $\longrightarrow$ POOL]*3 $\longrightarrow$ [FC $\longrightarrow$ RELU]*2 $\longrightarrow$ FC
- 2 CONV layers before every POOL
- Usually good for larger and deeper networks bec multiple stacked CONV layers can develop more complex features of input volume before destructive pooling operation

- Stack of CONV w/ small filters better than one larger receptive CONV layer
- Allows expression of more powerful features of input w/ fewer params but req more memory to hold all intermediate CONV layer results if using backpropagation
- ex. using three $3 \times 3$ CONV on top (w/ non-linearities) vs single CONV w/ $7 \times 7$ receptive field
- Effective receptive field is $7 \times 7$ for both, but neurons in single $7 \times 7$ are computing a linear function over input, while 3 stacks of CONV contain non-linearities
- Also, if all volumes have C channels, then single $7 \times 7$ CONV has $C \cdot (7 \times 7 \times C) = 49C^2$ channels while three $3 \times 3$ CONV have $3 \cdot (\cdot(3 \times 3 \times C)) = 27C^2$ channels

# 6 Layer Sizing Patterns

1. **INPUT layer:** should be divisible by 2 many times
   - ex. 32 (CIFAR-10), 64, 96 (STL-10), 224 (ImageNet), 384, 512
2. **CONV layers:** small filters ($3 \times 3$ or $5 \times 5$ at most) w/ $S = 1$ and padding input volumes w/ zeros s.t. CONV layer does not alter spatial dimensions of input

3. **POOL layers:** usually max-poolign w/ 2x2 receptive fields ($F = 2$) w/ stride $S = 2$
- Discards exactly 755 of activations in input volumes
- Receptive fields ¿ $3 \times 3$ are too lossy and aggressive $\implies$ worse performance

- This pattern only does downsampling in POOL layers
– Easier to track sizes than if downsampling in both CONV and POOL
- Smaller strides work better
- Using padding prevents info at borders from being washed away too quickly
– No padding causes volumes' sizes to reduce by small amounts each time

# 7 Famous Nets

1. **LeNet**
- 1st successful applications of ConvNets (1990s)
- Single CONV layer immediately followed by POOL layer
2. **AlexNet**
- 1st work popularizing ConvNets in Computer Vision (2012)
- Similar to LeNet but was deeper, bigger, and used CONV layers stacked on top of each other
3. **ZF Net**
- Improved AlexNet by tweaking architecture hyperparams
- Expanded size of middle CONV layers and made stride & filter size on 1st layers smaller
4. **GoogLeNet**
- Developed an Inception Module that reduced # of params in network (4M vs AlexNet 60M)
- Uses Average Pooling instead of FC at to of ConvNet which eliminated insignificant params
5. **VGGNet**
- Showed depth of network is critical for good performance
- Used 16 CONV/FC layers
- Homogenous architecture that only performs 3x3 convolutions and 2x3 pooling
- More expensive to evaluate & uses a lot more memory and params (140M)
- Most params in 1st FC, but later found that these FC layers can be removed w/ no performance downgrade
6. **ResNet (Residual Network)**
- Uses special skip connections and lots of batch normalization
- Missing FC layers at end of network

# 8 Computational Considerations

- Largest bottleneck for ConvNet is memory bottleneck
- Modern GPUs have around 3-6 GB memory w/ top having 12 GB

## 8.1 3 Major Sources of Memory Usage

1. Intermediate volume sizes
- Most **activations** in earlier CONV layers of ConvNet
- Kept bec needed for backprop
- But if only running ConvNet at test time can reduce activations by only storing current activations at any layer and discarding the prev activations on lower layers
2. Param sizes
- Stores network **parameters**, gradients during backprop, and a step cache if optimization is using momentum, Adagrad, or RMSProp
- Memory to store param vector alone usually multiplied by a factor $\geq 3$
3. **Miscellaneous** memory
- Image data batches, augmented versions, etc

- Once have rough estimate for total #'s of values (for activations, gradients, and misc), convert size to GB
- Multiple # of values by 4 to get raw bytes (every floating pt is 4 bytes, 8 for double precision) and then divide by 1024 to get amt of memory in KB, MB, and GB
- If network doesn't fit, make it fit by decreasing batch size since most memory consumed by activations