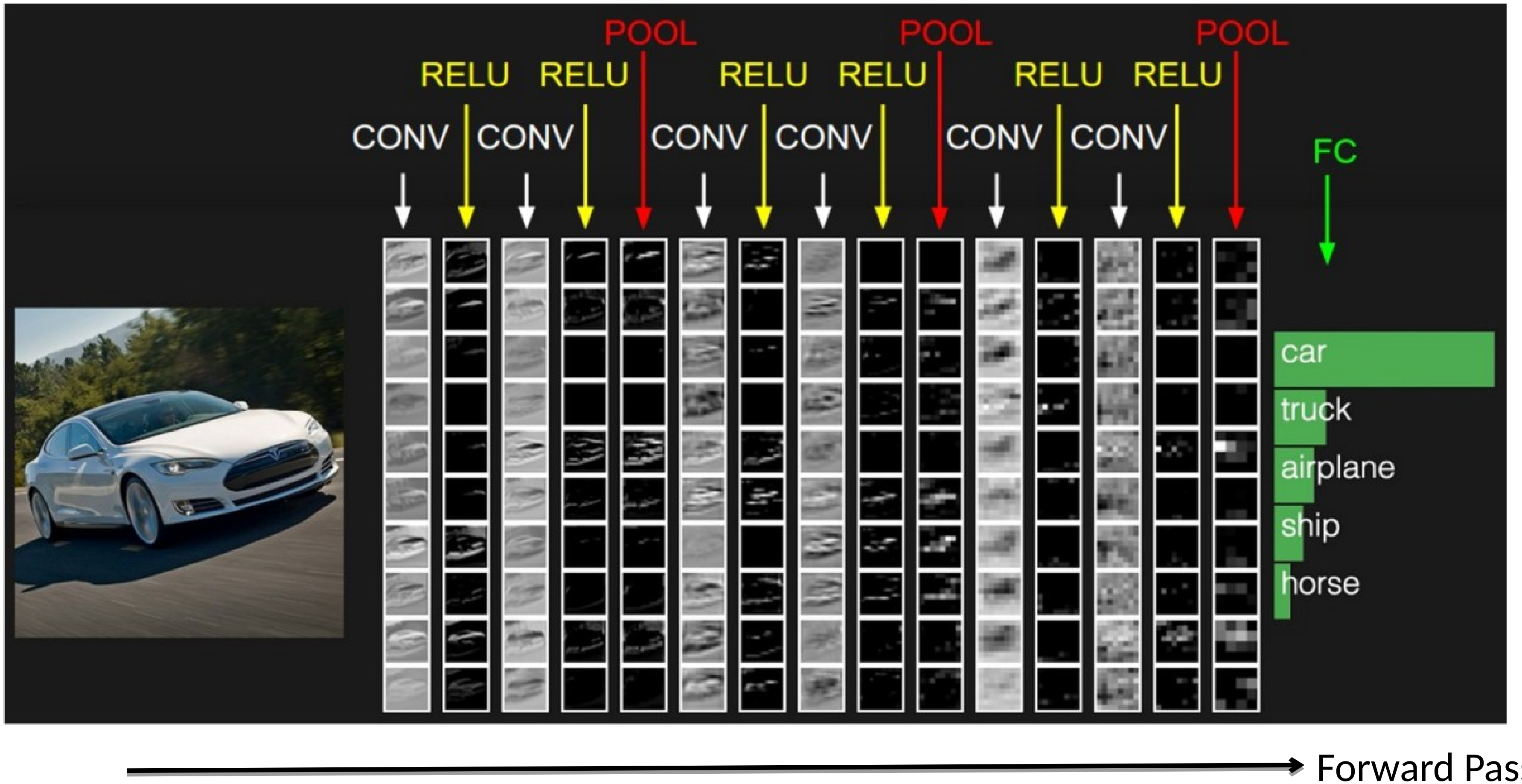
Basic operations in a CNN in typical computer vision inference task (forward pass)

Convolutional Neural Networks (CNN)



From http://cs231n.github.io/convolutional-networks/

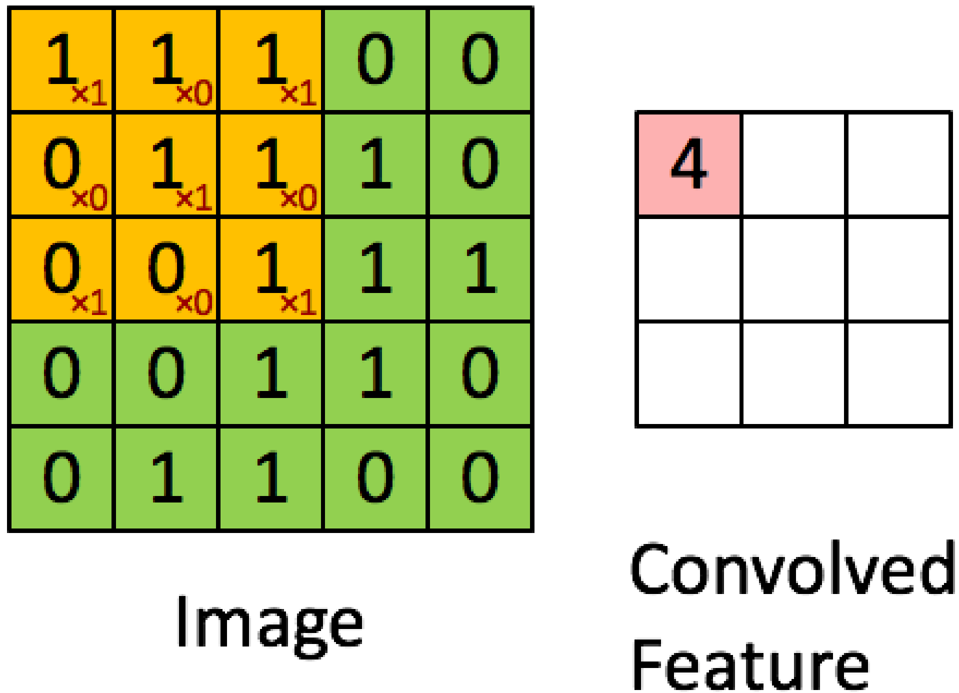
Given a trained network (architecture, hyper-parameters, parameters fixed)

- What mathematical operations are done during inference
- Sample c++ code for the operation (sequential implementation)
  - from https://github.com/JC1DA/DeepSense
- Some intuition on why that operation is useful

Operations

- Convolution
- Rectified Linear Unit (ReLU)
- Max-pooling
- Fully connected

Convolution without padding



1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

5x5 input.

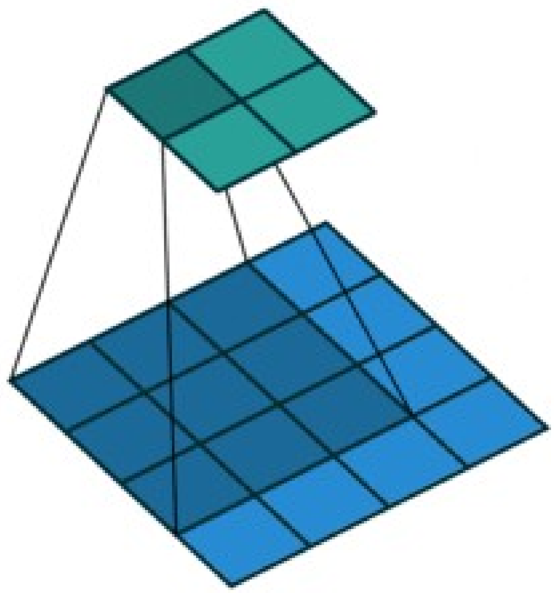
1	0	1
0	1	0
1	0	1

3x3 filter/kernel/feature detector.

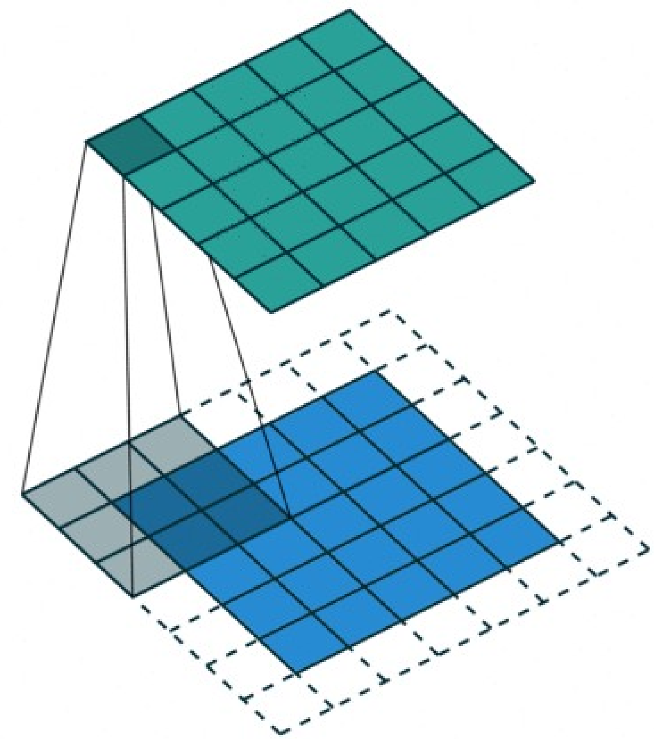
4	3	4
2	4	3
2	3	4

3x3 convolved feature/ activation map/feature map

Convolution with padding



4x4 input. 3x3 filter. Stride = 1.
2x2 output.

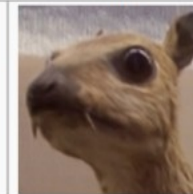
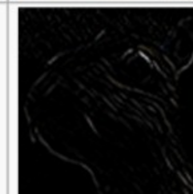
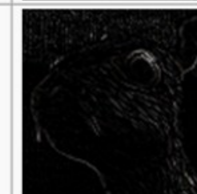
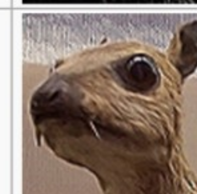
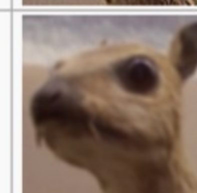


5x5 input. 3x3 filter. Stride = 1.
5x5 output.

Animation source: https://github.com/vdumoulin/conv_arithmetic

Multiple filters

Original image

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

Computations: #Multiplications and additions

```
int i, j, k, x, y, z;
for(i = 0 ; i < output->c; i++) {
    for(j = 0 ; j < output->h ; j++) {
        for(k = 0 ; k < output->w ; k++) {
            float result = 0.0f;
            for(x = 0 ; x < conv_layer->c; x++) {
                for(y = 0 ; y < conv_layer->h; y++) {
                    for(z = 0 ; z < conv_layer->w ; z++) {
                        int w = k * conv_layer->stride[0] - conv_layer->pad[0] + z;
                        int h = j * conv_layer->stride[1] - conv_layer->pad[2] + y;
                        if(w < 0 || w >= frame->w)
                            continue;
                        if(h < 0 || h >= frame->h)
                            continue;

                        float tmp1 = getDataFrom3D(frame->data, frame->h, frame->w, frame->c, h, w, x);
                        float tmp2 = getDataFrom4D(conv_layer->W, conv_layer->n, conv_layer->h, conv_layer->w, conv_layer->c, i, y, z, x);
                        result += tmp1 * tmp2;
                    }
                }
            }

            result += conv_layer->bias[i];
            output->data[getIndexFrom3D(output->c, output->h, output->w, i, j, k)] = result;
        }
    }
}
```

For each filter

For each row in output

For each column in output

Over filter depth

For each row in filter

For each column in filter

Multiplications, additions

Additions

4	3	4
2	4	3
2	3	4

3x3 output

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

5x5 input.

1	0	1
0	1	0
1	0	1

3x3 filter

Storage: #Parameters

```
int i, j, k, x, y, z;
for(i = 0 ; i < output->c; i++) {
    for(j = 0 ; j < output->h ; j++) {
        for(k = 0 ; k < output->w ; k++) {
            float result = 0.0f;
            for(x = 0 ; x < conv_layer->c; x++) {
                for(y = 0 ; y < conv_layer->h; y++) {
                    for(z = 0 ; z < conv_layer->w ; z++) {
                        int w = k * conv_layer->stride[0] - conv_layer->pad[0] + z;
                        int h = j * conv_layer->stride[1] - conv_layer->pad[2] + y;
                        if(w < 0 || w >= frame->w)
                            continue;
                        if(h < 0 || h >= frame->h)
                            continue;

                        float tmp1 = getDataFrom3D(frame->data, frame->h, frame->w, frame->c, h, w, x);
                        float tmp2 = getDataFrom4D(conv_layer->W, conv_layer->n, conv_layer->h, conv_layer->w, conv_layer->c, i, y, z, x);
                        result += tmp1 * tmp2;
                    }
                }
            }

            result += conv_layer->bias[i];
            output->data[getIndexFrom3D(output->c, output->h, output->w, i, j, k)] = result;
        }
    }
}
```

For each filter

Over filter depth

For each row in filter

For each column in filter

Weight parameters

Bias parameters

Hyper-parameters

- Number of filters
- Size of each filter (width, height)
- Padding
- Stride

Affects output size (which is next layer's inputs):

output width = (input width + 2 * padding – filter width)/stride + 1

output height = (input height + 2 * padding – filter height)/stride + 1

output depth = number of filters

Affects #parameters network has to learn during training

Weight terms = number of filters * filter width * filter height * filter depth

Bias terms = number of filters

Knobs to trade-off train and inference latency (more computations and memory reads), model size (storage) vs. accuracy.....

Projective field of an input

How many outputs are affected by that input?

```
int i, j, k, x, y, z;
for(i = 0 ; i < output->c; i++) {
    for(j = 0 ; j < output->h ; j++) {
        for(k = 0 ; k < output->w ; k++) {
            float result = 0.0f;
            for(x = 0 ; x < conv_layer->c; x++) {
                for(y = 0 ; y < conv_layer->h; y++) {
                    for(z = 0 ; z < conv_layer->w ; z++) {
                        int w = k * conv_layer->stride[0] - conv_layer->pad[0] + z;
                        int h = j * conv_layer->stride[1] - conv_layer->pad[2] + y;
                        if(w < 0 || w >= frame->w)
                            continue;
                        if(h < 0 || h >= frame->h)
                            continue;

                        float tmp1 = getDataFrom3D(frame->data, frame->h, frame->w, frame->c, h, w, x);
                        float tmp2 = getDataFrom4D(conv_layer->W, conv_layer->n, conv_layer->h, conv_layer->w, conv_layer->c, i, y, z, x);
                        result += tmp1 * tmp2;
                    }
                }
            }

            result += conv_layer->bias[i];
            output->data[getIndexFrom3D(output->c, output->h, output->w, i, j, k)] = result;
        }
    }
}
```

input neuron column

input neuron row

w=0 (k=0,z=0)

w=1 (k=1,z=0), (k=0,z=1)

w=2 (k=2,z=0),(k=1,z=1),(k=0,z=2)

(Effective) Receptive field of an output

How many (original) inputs affect that output?

```
int i, j, k, x, y, z;
for(i = 0 ; i < output->c; i++) {
    for(j = 0 ; j < output->h ; j++) {
        for(k = 0 ; k < output->w ; k++) {
            float result = 0.0f;
            for(x = 0 ; x < conv_layer->c; x++) {
                for(y = 0 ; y < conv_layer->h; y++) {
                    for(z = 0 ; z < conv_layer->w ; z++) {
                        int w = k * conv_layer->stride[0] - conv_layer->pad[0] + z;
                        int h = j * conv_layer->stride[1] - conv_layer->pad[2] + y;
                        if(w < 0 || w >= frame->w)
                            continue;
                        if(h < 0 || h >= frame->h)
                            continue;

                        float tmp1 = getDataFrom3D(frame->data, frame->h, frame->w, frame->c, h, w, x);
                        float tmp2 = getDataFrom4D(conv_layer->W, conv_layer->n, conv_layer->h, conv_layer->w, conv_layer->c, i, y, z, x);
                        result += tmp1 * tmp2;
                    }
                }
            }

            result += conv_layer->bias[i];
            output->data[getIndexFrom3D(output->c, output->h, output->w, i, j, k)] = result;
        }
    }
}
```

filter width * filter height * filter depth

More knobs to improve efficiency at same accuracy

- More number of smaller filters (VGG vs. Alexnet)
- Different order of looping (dataflow)
- Split computations (mobilenet)

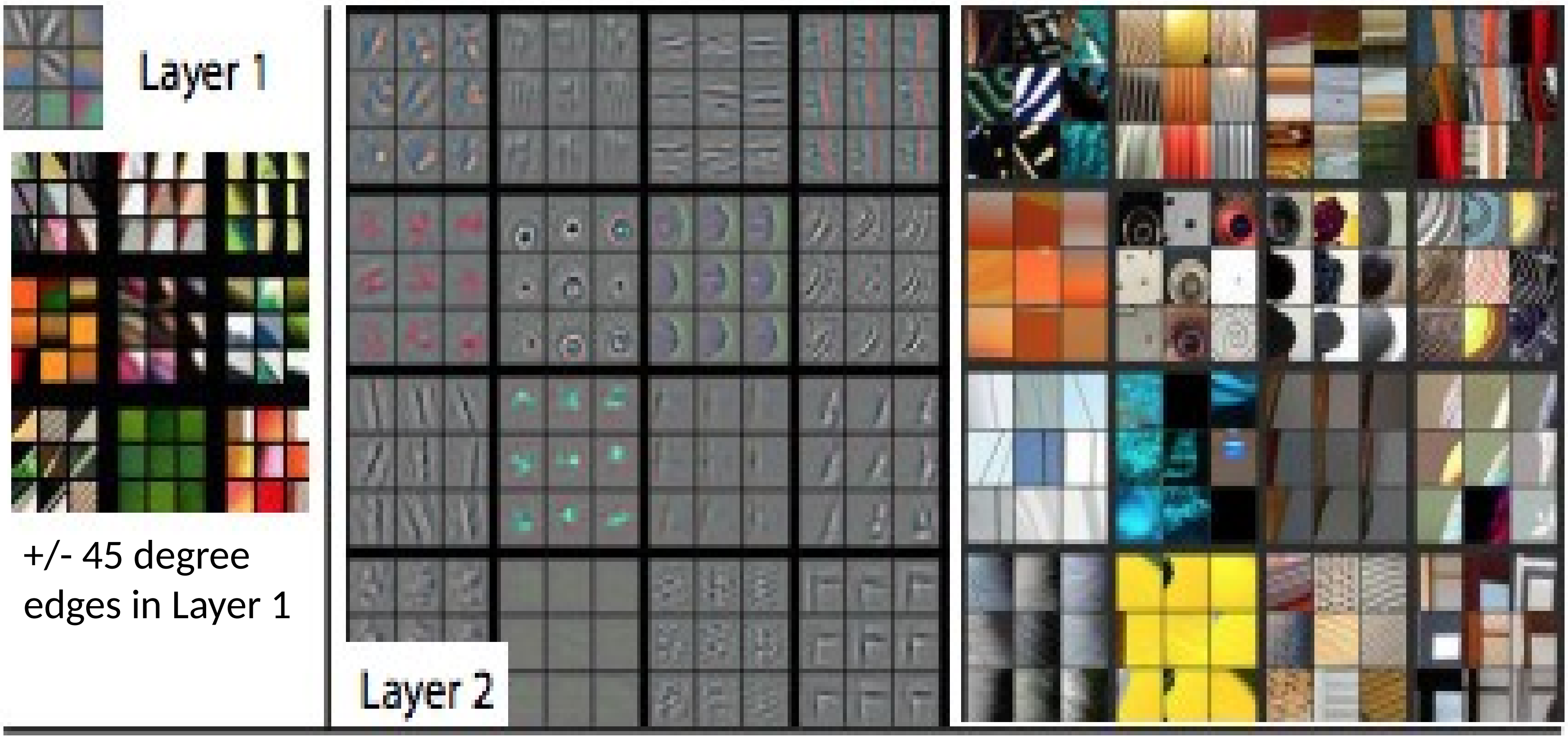
```c
int i, j, k, x, y, z;
for(i = 0 ; i < output->c; i++) {
    for(j = 0 ; j < output->h ; j++) {
        for(k = 0 ; k < output->w ; k++) {
            float result = 0.0f;
            for(x = 0 ; x < conv_layer->c; x++) {
                for(y = 0 ; y < conv_layer->h; y++) {
                    for(z = 0 ; z < conv_layer->w ; z++) {
                        int w = k * conv_layer->stride[0] - conv_layer->pad[0] + z;
                        int h = j * conv_layer->stride[1] - conv_layer->pad[2] + y;
                        if(w < 0 || w >= frame->w)
                            continue;
                        if(h < 0 || h >= frame->h)
                            continue;

                        float tmp1 = getDataFrom3D(frame->data, frame->h, frame->w, frame->c, h, w, x);
                        float tmp2 = getDataFrom4D(conv_layer->W, conv_layer->n, conv_layer->h, conv_layer->w, conv_layer->c, i, y, z, x);
                        result += tmp1 * tmp2;
                    }
                }
            }

            result += conv_layer->bias[i];
            output->data[getIndexFrom3D(output->c, output->h, output->w, i, j, k)] = result;
        }
    }
}
```

Each filter searches for a particular feature at different image locations (translation invariance)

Features at successive convolutional layers



+/- 45 degree edges in Layer 1

Corners and other edge color conjunctions in Layer 2

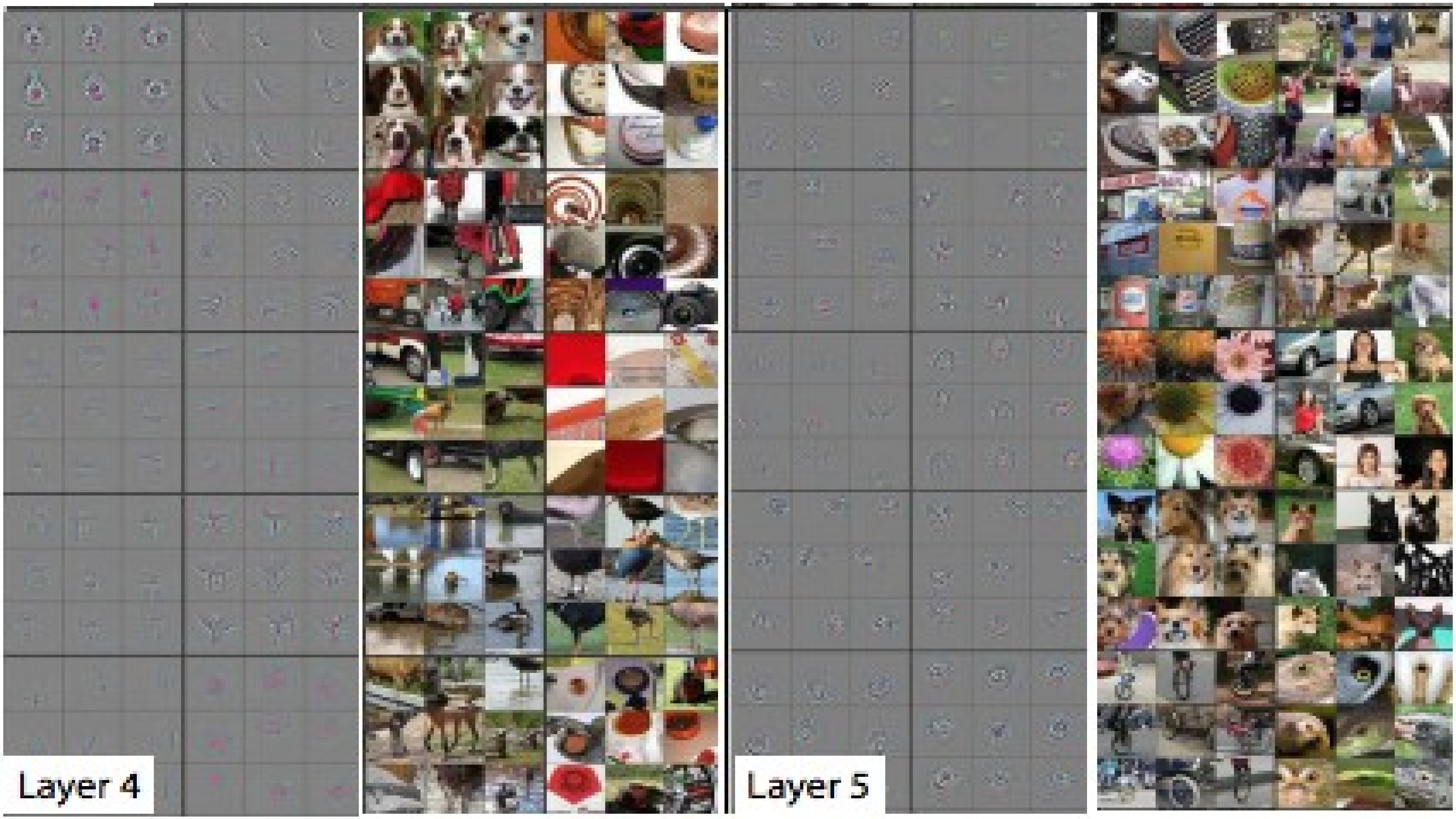
Visualizing and Understanding Convolutional Networks,
Matthew D. Zeiler and Rob Fergus, ECCV 2014

Features at successive convolutional layers

More complex invariances than Layer 2. Similar textures e.g. mesh patterns (R1C1); Text (R2C4).

Visualizing and Understanding Convolutional Networks,
Matthew D. Zeiler and Rob Fergus, ECCV 2014

Features at successive convolutional layers



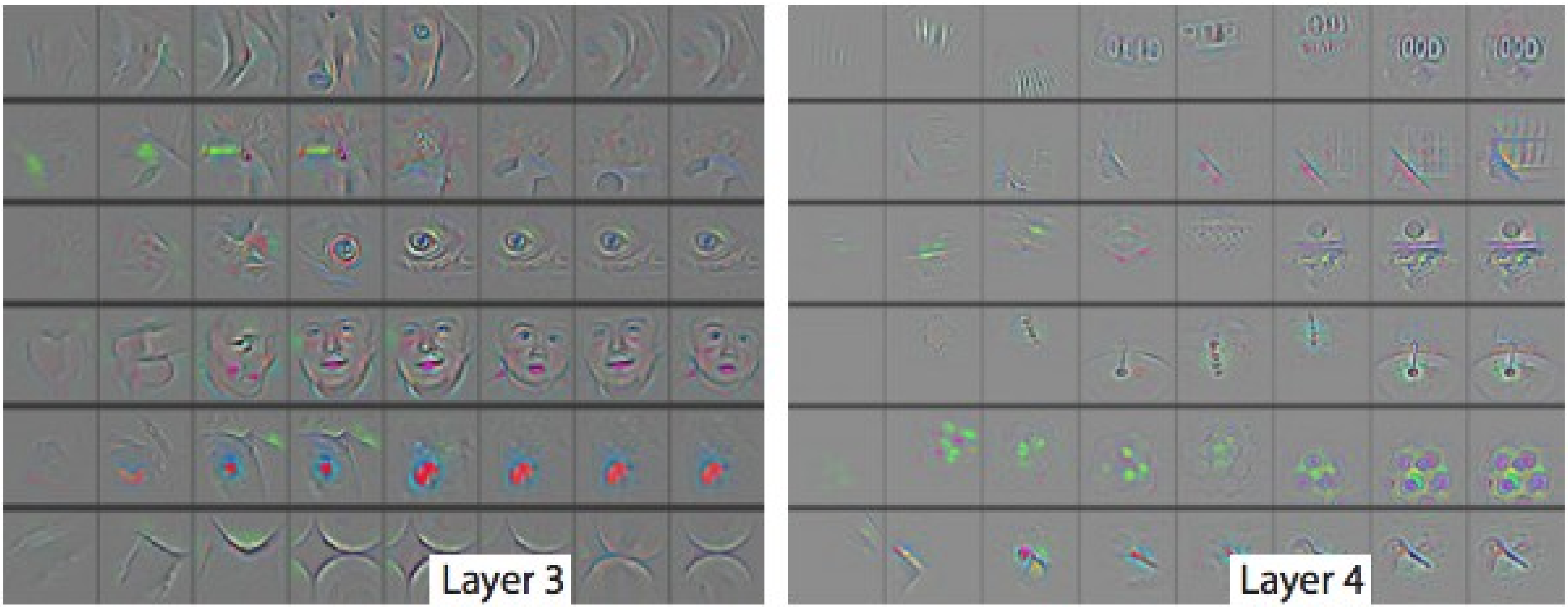
Significant variation, more class specific.
Dog faces (R1C1); Bird legs (R4C2).

Entire objects with significant pose variation.
Keyboards (R1C1); dogs (R4).

Visualizing and Understanding Convolutional Networks,
Matthew D. Zeiler and Rob Fergus, ECCV 2014

Who decides these features?

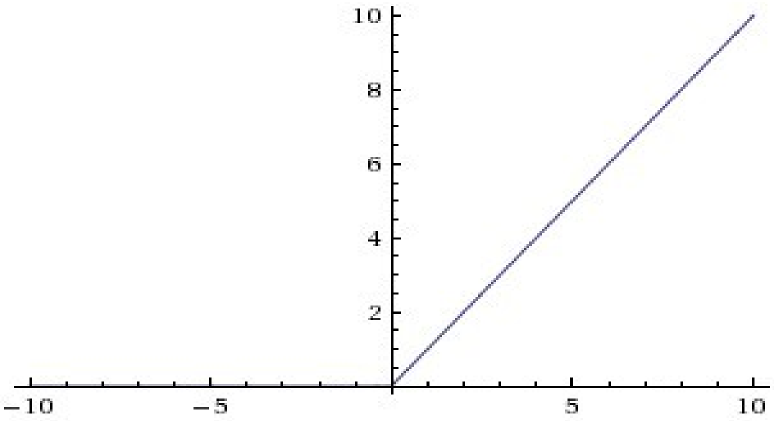
The network itself while **training** learns the filter weights and bias terms.



Evolution of randomly chosen subset of model features at **training epochs** 1,2,5,10,20,30,40,64.

Visualizing and Understanding Convolutional Networks,
Matthew D. Zeiler and Rob Fergus, ECCV 2014

Rectified Linear Unit (ReLU)



For all inputs

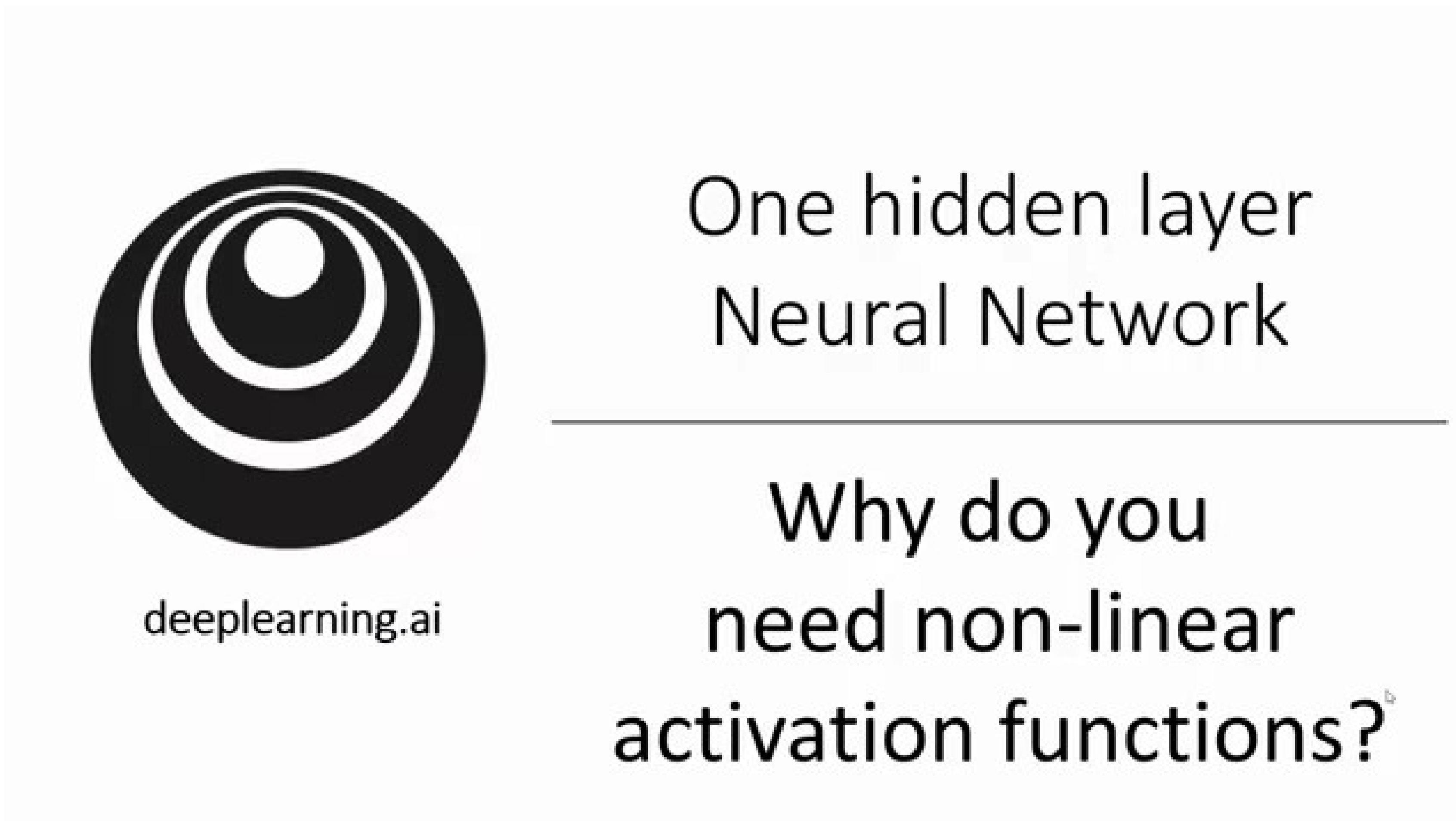
```
cnn_frame *activate_RELU(cnn_frame *frame) {
    int i;
    for(i = 0 ; i < frame->c * frame->h * frame->w ; i++) {
        float x = frame->data[i];
        frame->data[i] =(x > 0) ? x : 0;
    }
    return frame;
}
```

Rectified Linear Unit (ReLU)

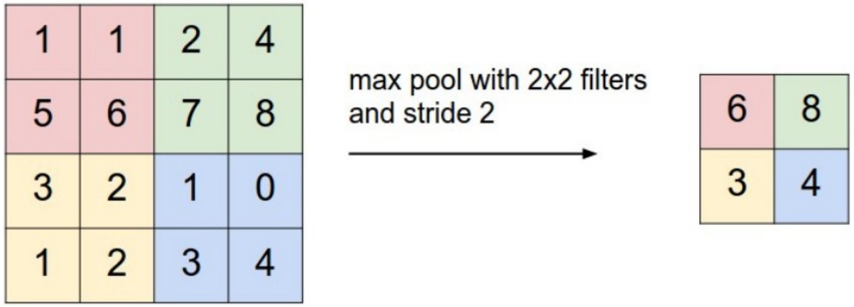
- Simple function -> Fast to compute, no hyper-parameter choice, no parameter learning
- Introduces sparsity when x <= 0. We will see the benefits of sparsity in reducing model size and increasing computation speed later.
- Faster to train, due to constant gradient of ReLUs when x>0 (what has gradient got to do with training speed?)

ReLU is a non-linear activation, following each linear convolution filter operation

Why is non-linearity needed?



Max pooling




```
for(int k = 0 ; k < output->c ; k++) {
    for(int h = 0; h < output->h; h++) {
        for(int w = 0; w < output->w ; w++) {
            float max = -999999.9f;
            for(int x = 0 ; x < maxpool_layer->size ; x++) {
                for(int y = 0 ; y < maxpool_layer->size ; y++) {
                    int x_ = w * maxpool_layer->stride[0] + x - maxpool_layer->pad[0];
                    int y_ = h * maxpool_layer->stride[1] + y - maxpool_layer->pad[2];
                    int valid = (x_ >= 0 && x_ < frame->w && y_ >= 0 && y_ < frame->h);
                    float val = (valid != 0) ? frame->data[getIndexFrom3D(frame->h, frame->w, frame->c, y_, x_, k)] : -999999.9f;
                    max   = (val > max) ? val   : max;
                }
            }
            output->data[getIndexFrom3D(output->h, output->w, output->c, h, w, k)] = max;
        }
    }
}
```

For each output depth channel

For each output row

For each output column

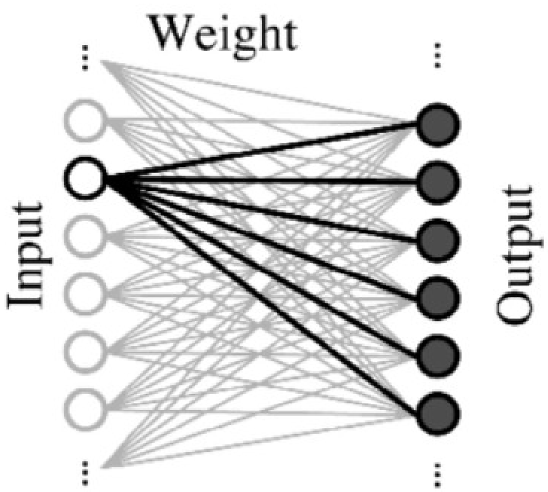
For each maxpool layer column

For each maxpool layer row

Max pooling

- Reduces dimensionality of each feature map, but retains the most important information
- Reduced number of parameters reduces computation, memory reads, storage requirements and over-fitting to training data
- Makes the network invariant to small transformations in input image, as max pooled value over local neighborhood won't change on small distortions

Fully Connected



weight parameters =
number of outputs * number of inputs

```
for(int n = 0 ; n < connected_layer->outputSize ; n++) {
    output->data[n] = 0;
    for(int i = 0 ; i < frame->c ; i++) {
        for(int j = 0 ; j < frame->h ; j++) {
            for(int k = 0 ; k < frame->w ; k++) {
                int index = getIndexFrom3D(frame->c, frame->h, frame->w, i , j , k);
                output->data[n] += frame->data[index] * connected_layer->W[index * connected_layer->outputSize + n];
            }
        }
    }

    output->data[n] += connected_layer->bias[n];
}
```

For each output

For each input depth channel

For each input row

For each input column

Multiplications, additions

#bias parameters = number of outputs

Additions

Let’s compute #parameters

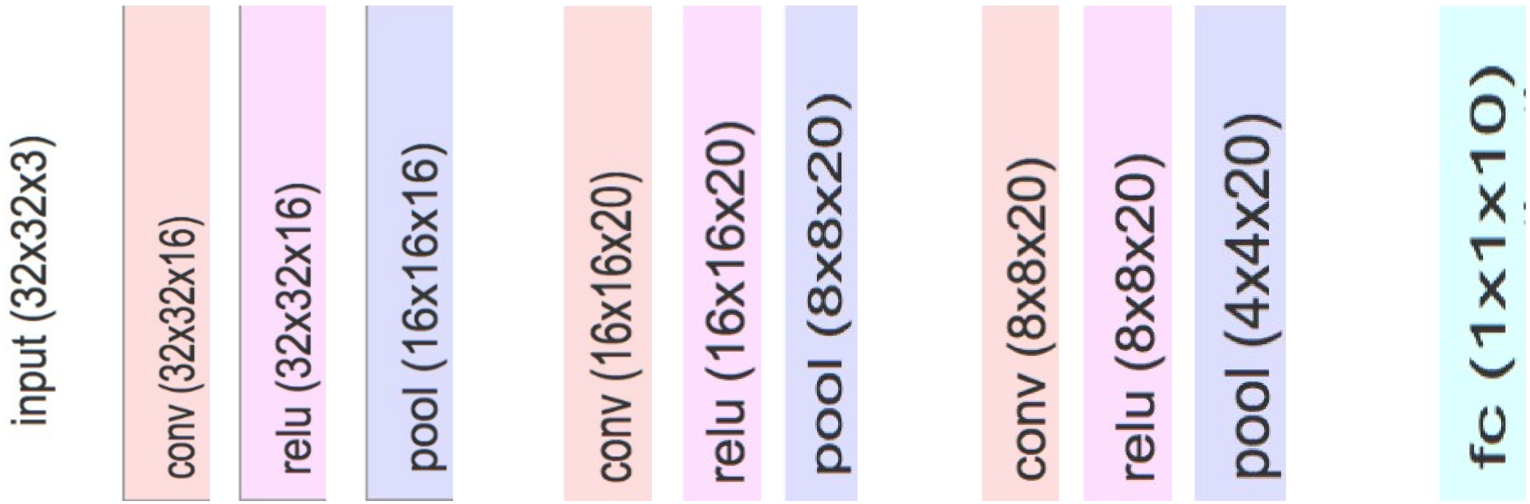
Example from http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

```
layer_defs.push({type:'input', out_sx:32, out_sy:32, out_depth:3});
layer_defs.push({type:'conv', sx:5, filters:16, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:20, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:20, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'softmax', num_classes:10});
```

Let's compute #parameters

Example from http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

```
layer_defs.push({type:'input', out_sx:32, out_sy:32, out_depth:3});
layer_defs.push({type:'conv', sx:5, filters:16, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:20, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'conv', sx:5, filters:20, stride:1, pad:2, activation:'relu'});
layer_defs.push({type:'pool', sx:2, stride:2});
layer_defs.push({type:'softmax', num_classes:10});
```



5x5x3x16+16 5x5x16x20+20 5x5x20x20+20 4x4x20x10+10