CAP 4453 Robot Vision

Dr. Gonzalo Vaca-Castaño

gonzalo.vacacastano@ucf.edu

Administrative details

- 1 homework
- 1 project

Credits

- Some slides comes directly from:
  - Yosesh Rawat
  - Andrew Ng

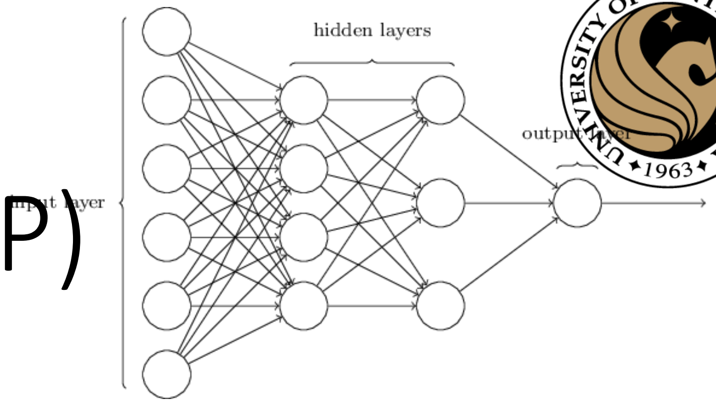
Robot Vision

18. Convolutional Neural Networks I

Fully connected networks: Review

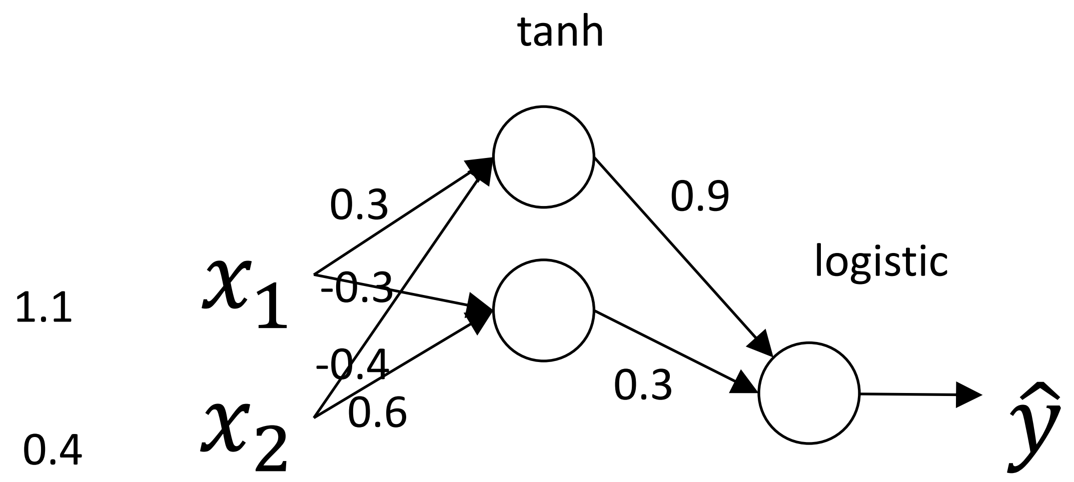
A REVIEW

Fully connected Neural network A.K.A Multi-Layer Perceptron (MLP)



- A deep network is a neural network with many layers
- A neuron in a linear function followed for an activation function
- Activation function must be non-linear
- A loss function measures how close is the created function (network) from a desired output
- The “training” is the process of find parameters (‘weights’) that reduces the loss functions
- Updating the weights as $w_{new} = w_{prev} - \alpha \frac{dJ}{dW}$ reduces the loss
- An algorithm named back-propagation allows to compute $\frac{dJ}{dW}$ for all the weights of the network in 2 steps: 1 forward, 1 backward

Exercise



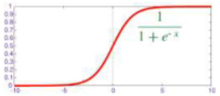
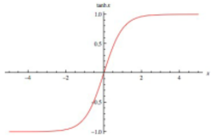
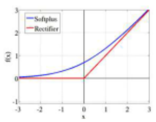
Expected output: 1

$$Div = \frac{(\hat{y}_i - y_i)^2}{2}$$

$$-(\hat{y}_i - y_i^{[N]})$$

?

Activations and their derivatives

	$f(z) = \frac{1}{1+\exp(-z)}$	$f'(z) = f(z)(1-f(z))$
	$f(z) = \tanh(z)$	$f'(z) = (1-f^2(z))$
	$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$	This space left intentionally (kind of) blank
	$f(z) = \log(1+\exp(z))$	$f'(z) = \frac{1}{1+\exp(-z)}$

Gradients: Backward Computation

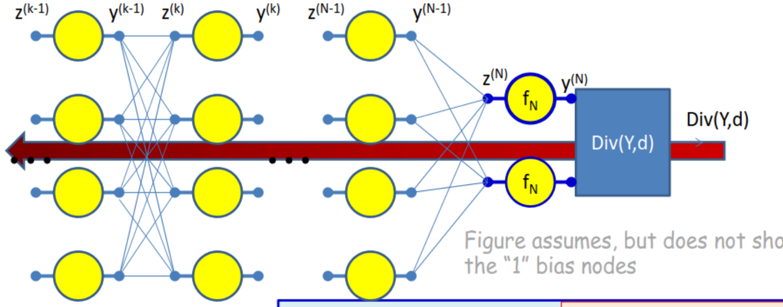


Figure assumes, but does not show the "1" bias nodes

Initialize: Gradient w.r.t network output

$$\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

For $k = N..1$

For $i = 1:layer-width$

$$\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$$

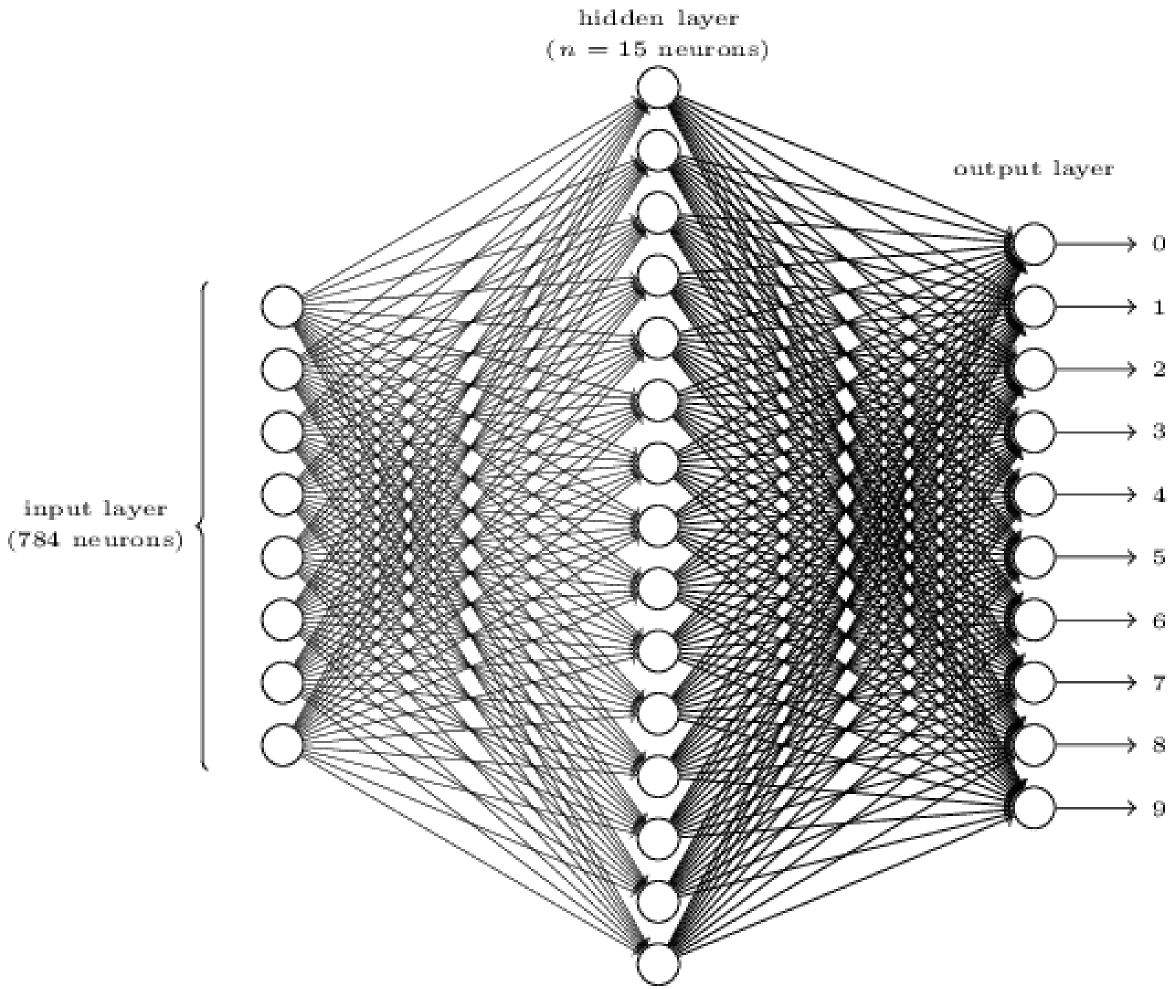
$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

$$\boxed{\frac{\partial Div}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial Div}{\partial z_j^{(k)}}}$$

Digit classification

28

28

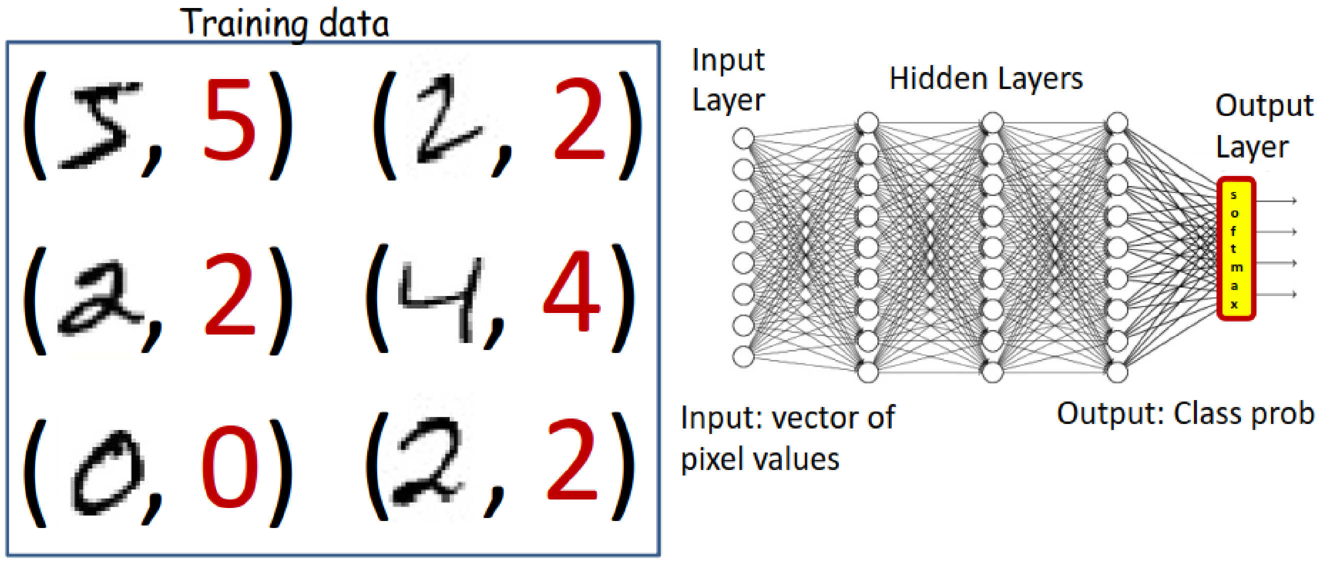


- MNIST dataset:
  - 70000 grayscale images of digits scanned.
  - 60000 for training
  - 10000 for testing
- Loss function

$$J_2(w) = \frac{1}{m} \sum_{train} (\,\hat{y}_i - y_i)^2$$

Digit classification

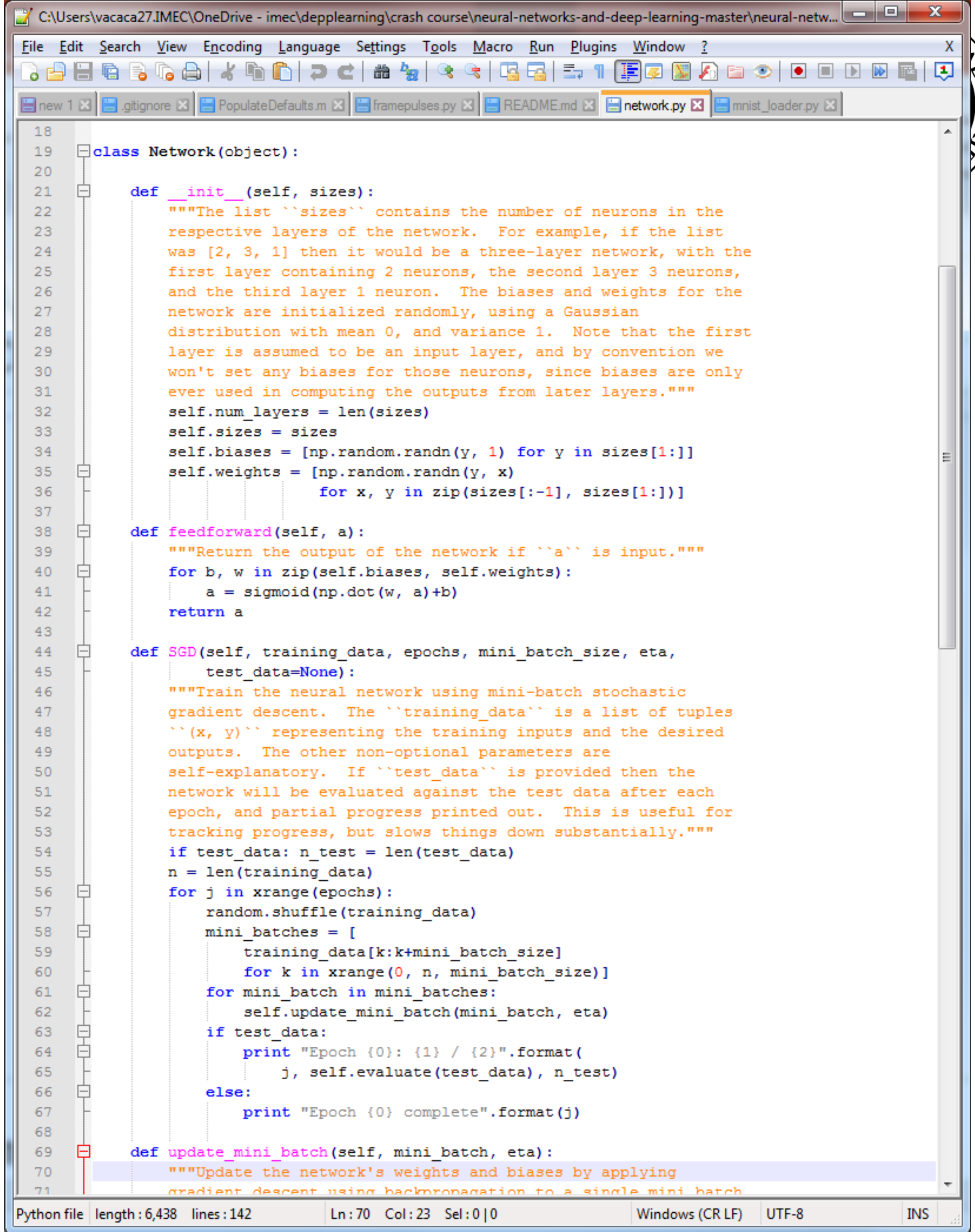
Typical Problem statement: multiclass classification



- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job

A look in the code

- To run this code do:
  - import network
  - net = network.Network([784, 30, 10])
  - net.SGD(training_data, 30, 10, 3.0, test_data=test_data)




```python

class Network(object):

    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.  The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1.  Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch
```

A look in code

Initialize: Gradient w.r.t network output

$$\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

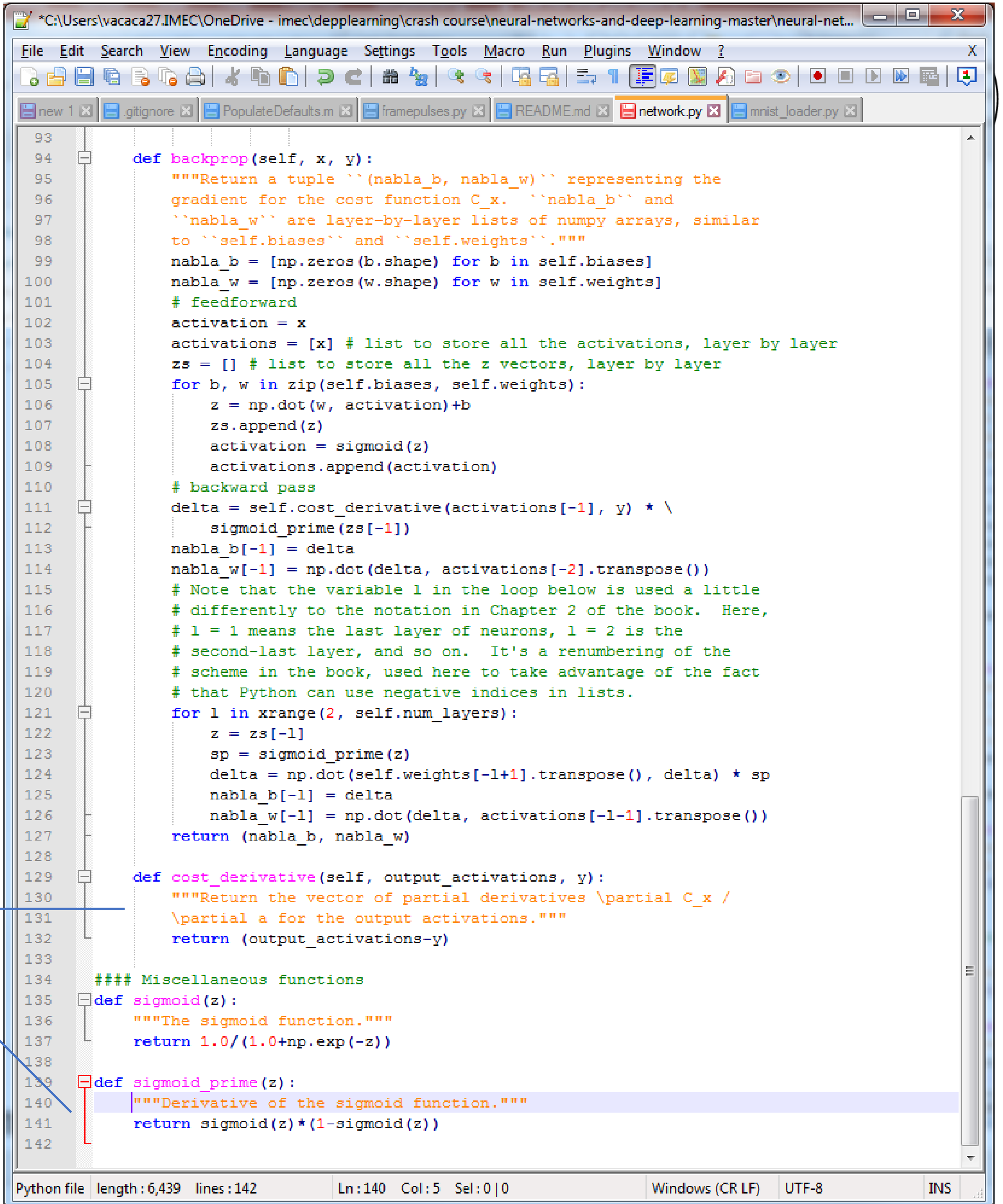
$For\ k = N..1$

$For\ i = 1: layer - width$

$$\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)}\frac{\partial Div}{\partial z_j^{(k)}}$$

$$\boxed{\frac{\partial Div}{\partial w_{ij}^{(k)}} = y_i^{(k-1)}\frac{\partial Div}{\partial z_j^{(k)}}}$$




```python
    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

A look in code

Initialize: Gradient w.r.t network output

$$\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

For $k = N..1$

For $i = 1 : layer - width$

$$\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

$$\boxed{\frac{\partial Div}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial Div}{\partial z_j^{(k)}}}$$

*C:\Users\vacaca27.IMEC\OneDrive - imec\depplearning\crash course\neural-networks-and-deep-learning-master\neural-net...

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

new 1 | .gitignore | PopulateDefaults.m | framepulses.py | README.md | network.py | mnist_loader.py

```python

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

```

Python file | length : 6,439 | lines : 142 | Ln : 140 | Col : 5 | Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

A look in the code

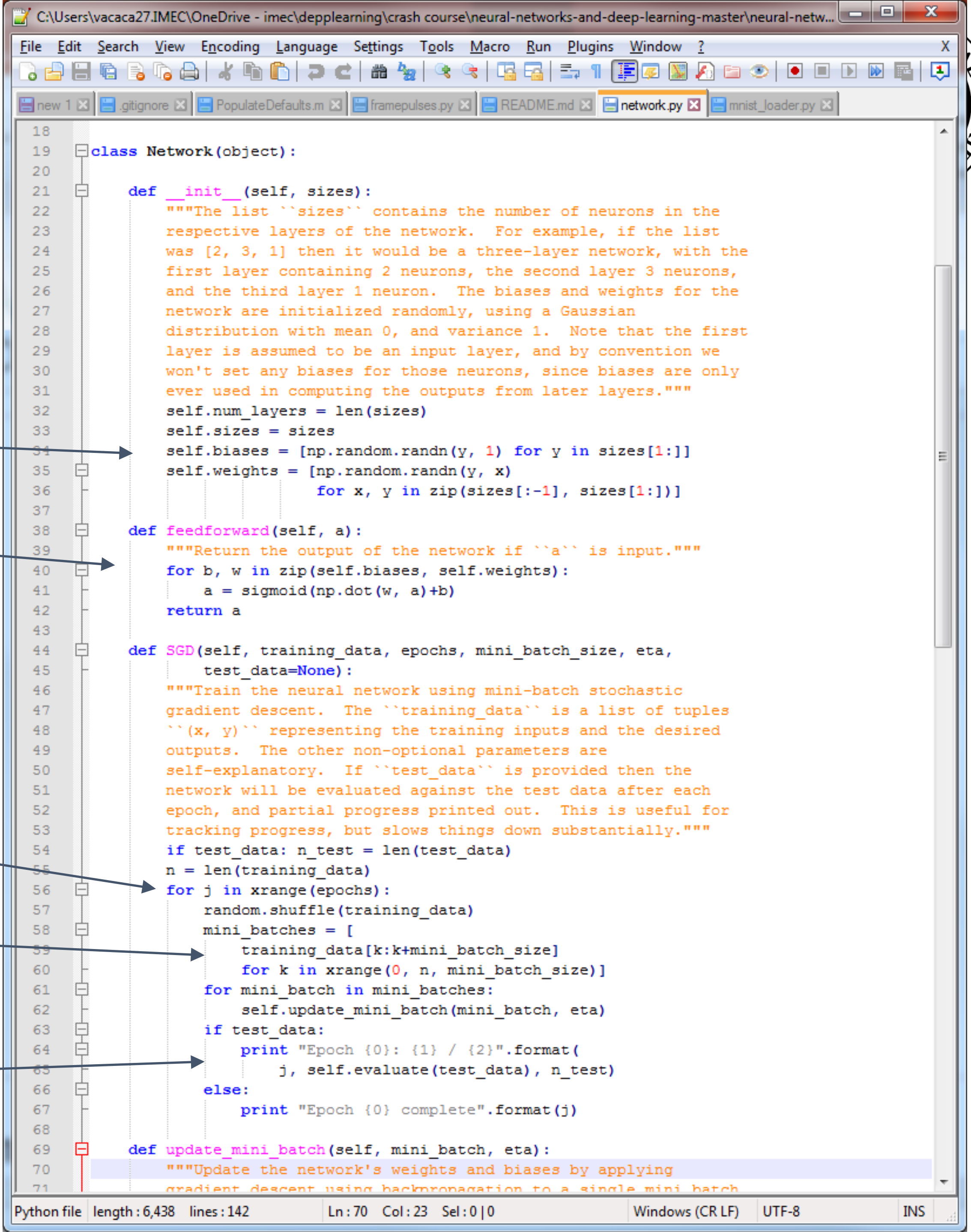
random Initialization

Feed forward ‘a’ thru all the layers

A Epoch is when all the training data has been used to update weights

A minibatch is a subset of all the data used to obtain a ‘quick’ weight updates

If there is test data perform evaluation




```python

class Network(object):

    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.  The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1.  Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch
```

A look in the code

Add errors from all the training data from the mini-batch

Update the weights

```
    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def evaluate(self, test_data):
        """Return the number of test inputs for which the neural
        network outputs the correct result. Note that the neural
        network's output is assumed to be the index of whichever
        neuron in the final layer has the highest activation."""
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)
```

Story so far

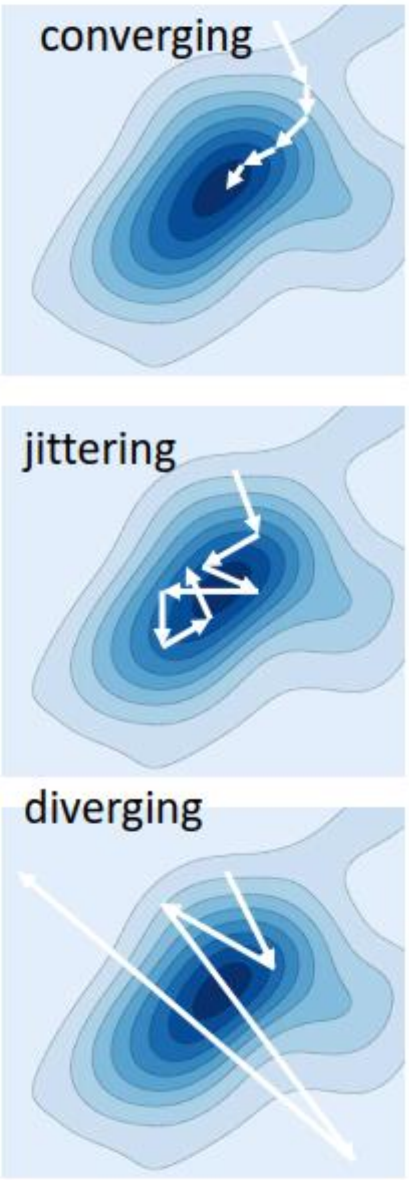
- Neural nets can be trained via gradient descent that minimizes a loss function
- Backpropagation can be used to derive the derivatives of the loss
- Backprop *is not guaranteed* to find a "true" solution, even if it exists, and lies within the capacity of the network to model
  - The optimum for the loss function may not be the "true" solution
- For large networks, the loss function may have a large number of unpleasant saddle points
  - Which backpropagation may find

Convergence of gradient descent

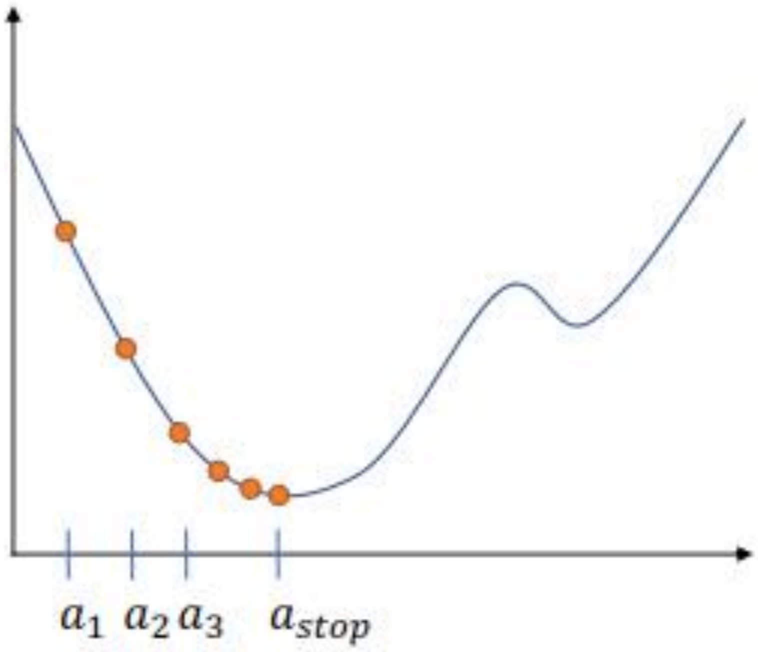
- An iterative algorithm is said to *converge* to a solution if the value updates arrive at a fixed point
  - Where the gradient is 0 and further updates do not change the estimate

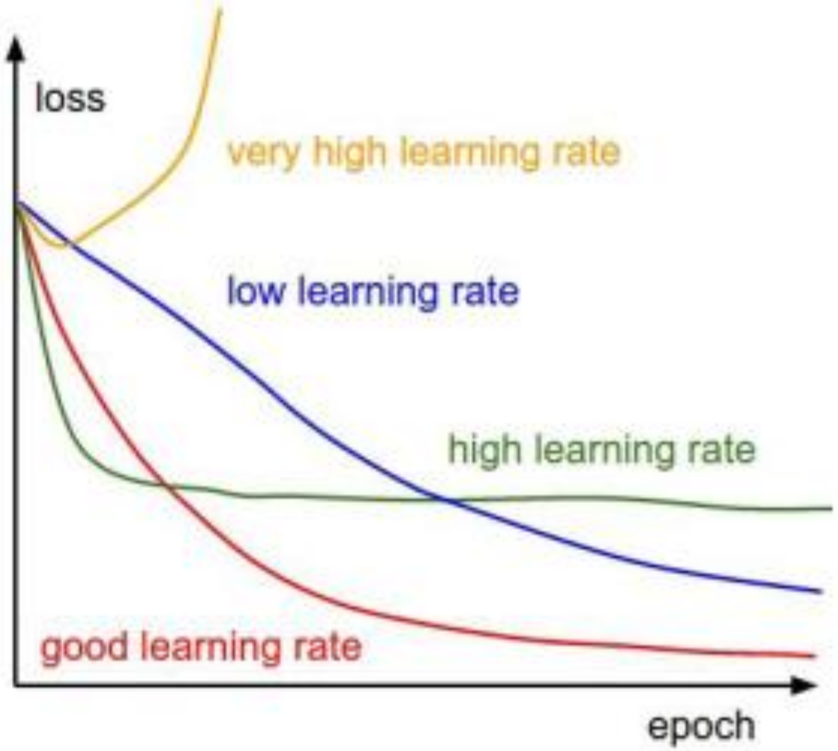
- The algorithm may not actually converge
  - It may jitter around the local minimum
  - It may even diverge

- Conditions for convergence?

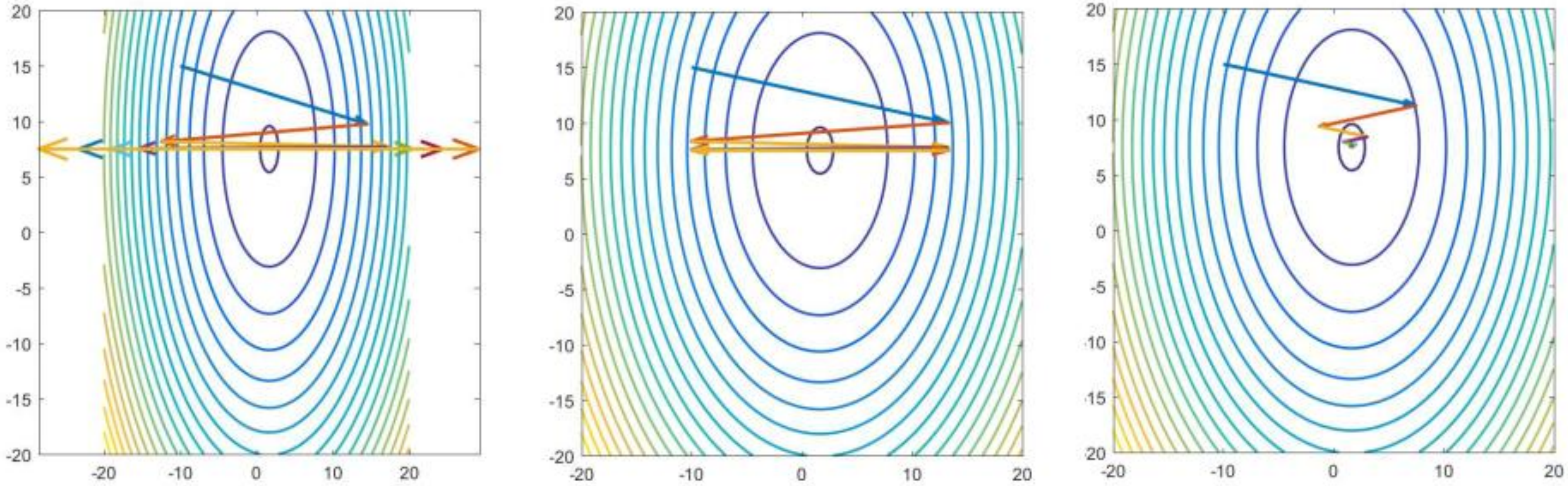


Learning rate





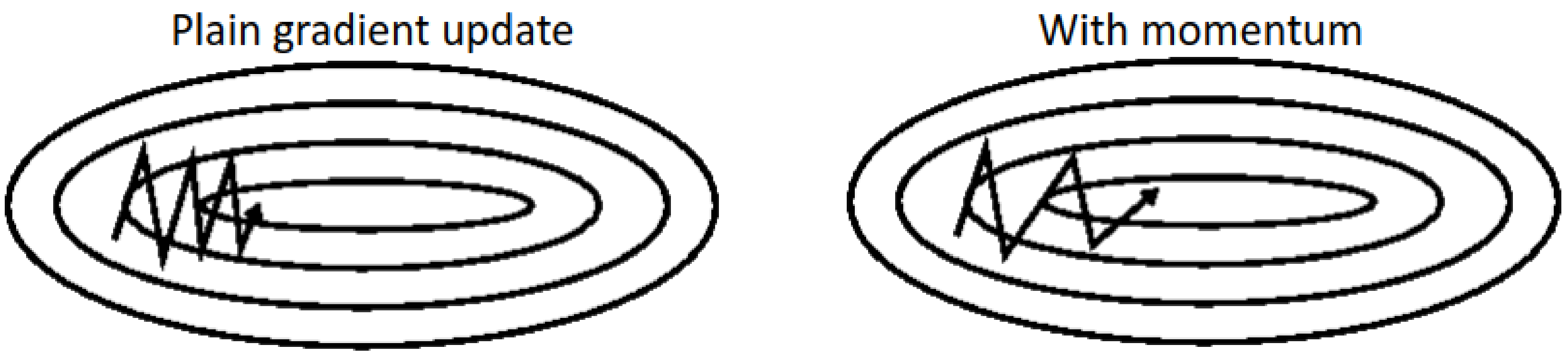
A closer look at the convergence problem



- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others

- **Proposal:**
  - Keep track of oscillations
  - Emphasize steps in directions that converge smoothly
  - Shrink steps in directions that bounce around..

Momentum Update



- The momentum method maintains a running average of all gradients until the *current* step

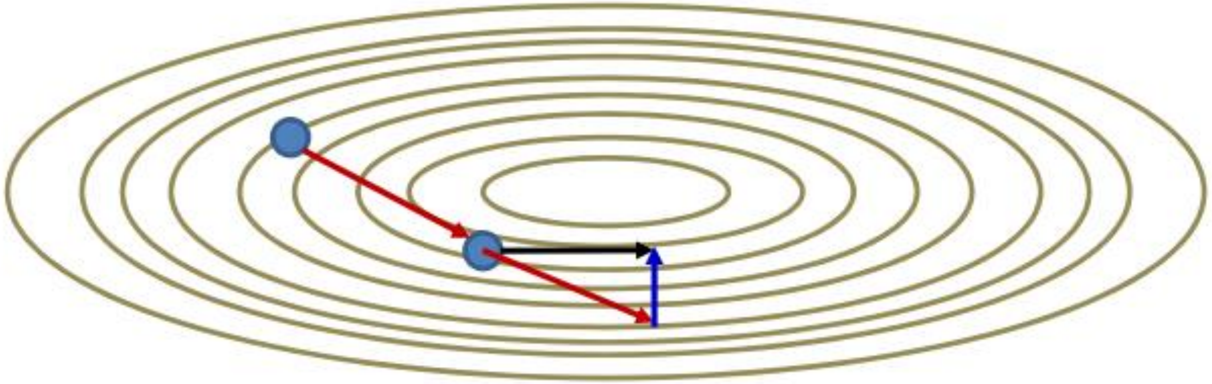
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k-1)}\right)$$

$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

  – Typical β value is 0.9

- The running average steps
  – Get longer in directions where gradient stays in the same sign
  – Become shorter in directions where the sign keeps flipping

Nestorov’s Accelerated Gradient



- Nestorov’s method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k)} + \beta \Delta W^{(k-1)}\right)$$

$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

Momentum methods emphasize directions of steady improvement are demonstrably superior to other methods

Other popular optimizers

RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

Adam

Adaptive Moment Estimation (Adam) [14] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface [15]. We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

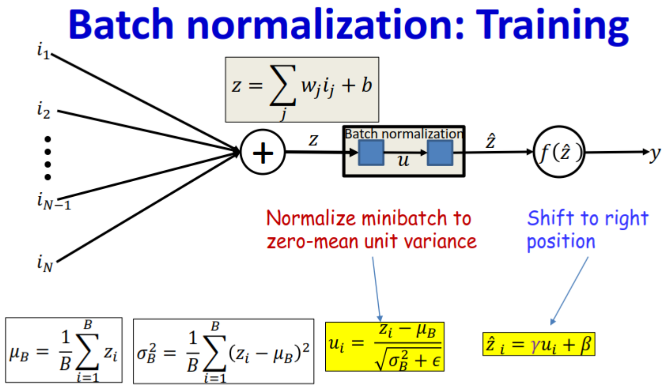
They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other

Other omitted tricks

REGULARIZATION



- Batch normalization

[Batch Normalization | What is Batch Normalization in Deep Learning (analyticsvidhya.com)]
[Batch normalization - Wikipedia]

- Regularization

$$L(W_1, W_2, \dots, W_K) = \frac{1}{T}\sum_t Div(Y_t, d_t) + \frac{1}{2}\lambda \sum_k \|W_k\|_2^2$$

  - Batch mode:

$$\Delta W_k = \frac{1}{T}\sum_t \nabla_{W_k} Div(Y_t, d_t)^T + \lambda W_k$$

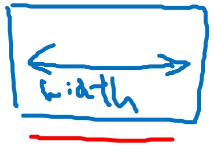
- Dropout: During training, for each input, at each iteration turn off" each neuron with a probability 1-a
- Data augmentation

Chain of assumptions in ML

→ Fit training set well on cost function

(≈ human-level performance)

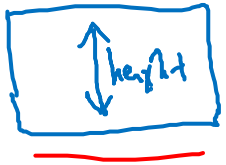
bigger network
Adam
...



early stopping

→ Fit dev set well on cost function

Regularization
Bigger train set



→ Fit test set well on cost function

Bigger dev set

→ Performs well in real world

(Happy cat pic app users.)

Change dev set or cost function

Train/dev/test sets

Data | | dev | test

training set

- Hold-out cross validation
- Development set "dev"

test

train test

Previous era: 70/30 %

100 - 1000 - 10000

60/20/20 %

Big data: 1,000,000 10,000 10,000

98/1/1 %

99.5 / ~~.25~~ .4 / ~~.25~~ .1 %

Outline

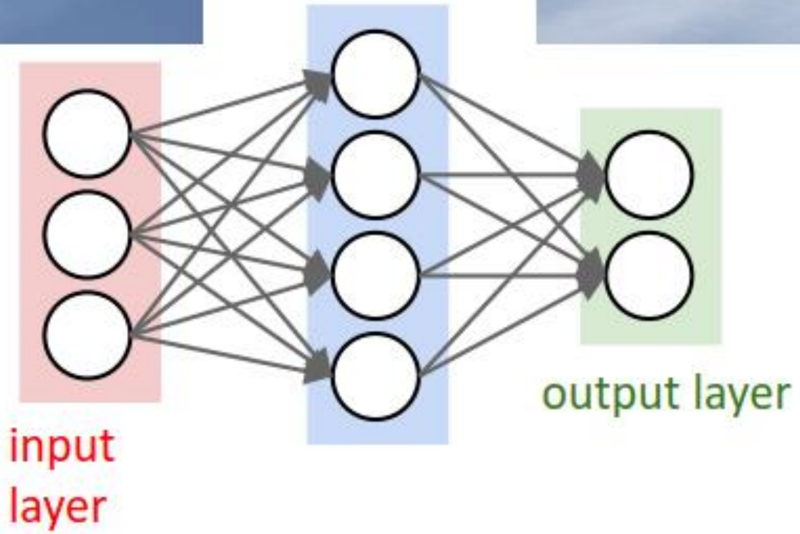
- What is a CNN (convolutional Neural Network)
- Image Classification
  - AlexNet: Network structure
    - Dropout, RELU
  - NN as feature vector
  - More recent networks:
    - VGG
    - ResNet
- Domain adaptation
  - Transfer learning, fine-tuning
  - Example: Python detection

References

- http://neuralnetworksanddeeplearning.com/chap1.html
- https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2015/
- Coursera (Deep learning specialization)

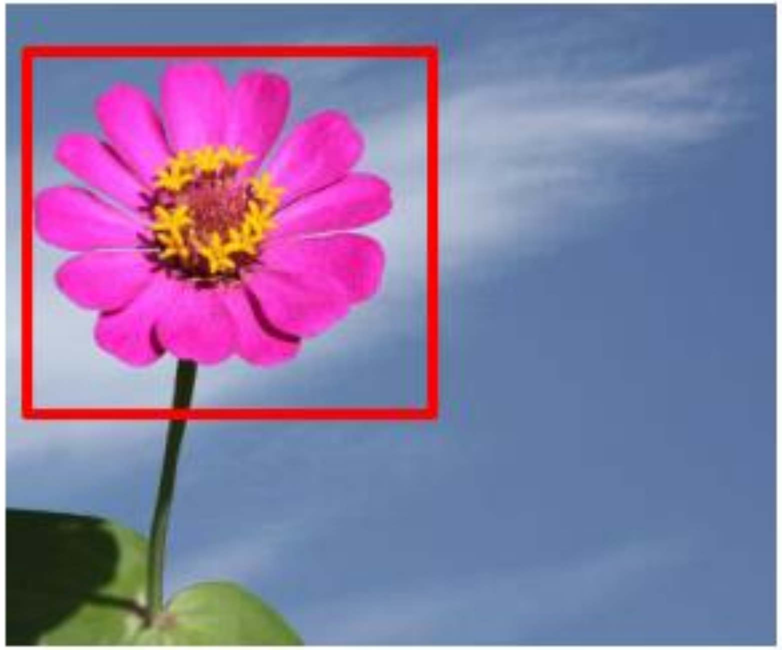
Convolutional Neural Networks

A problem



- Will an MLP that recognizes the left image as a flower also recognize the one on the right as a flower?

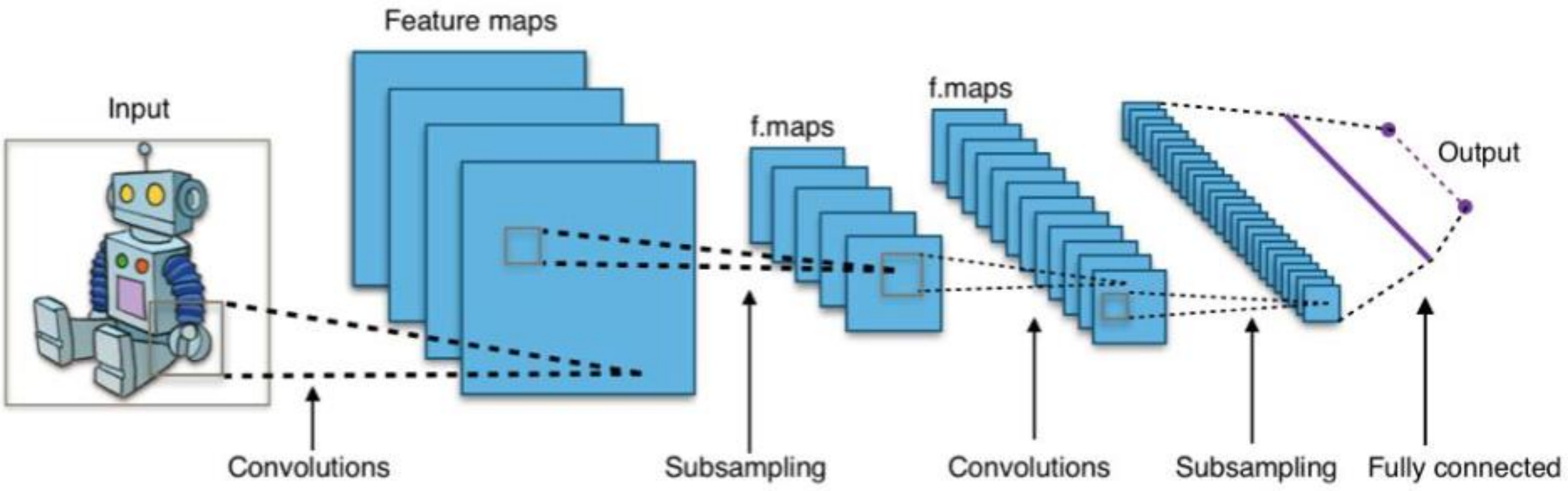
The need for *shift invariance*



=

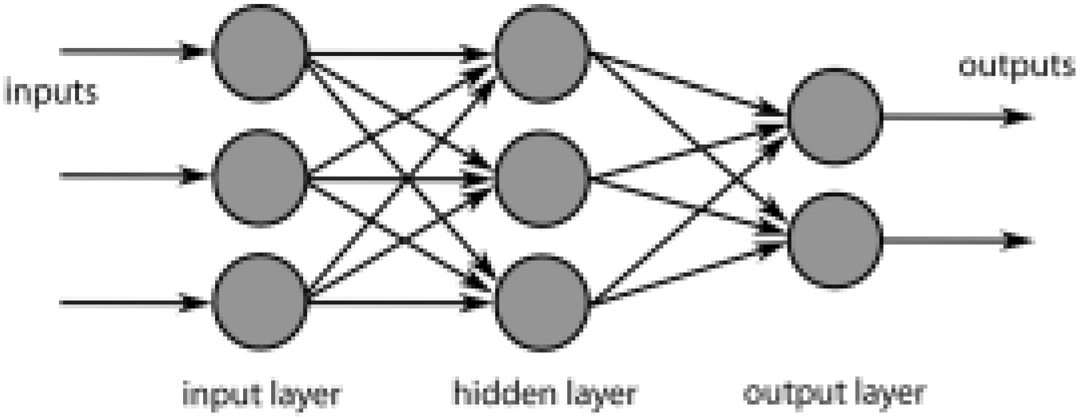
Convolutional Neural Network (CNN)

- A class of Neural Networks
  - Takes image as input (mostly)
  - Make predictions about the input image

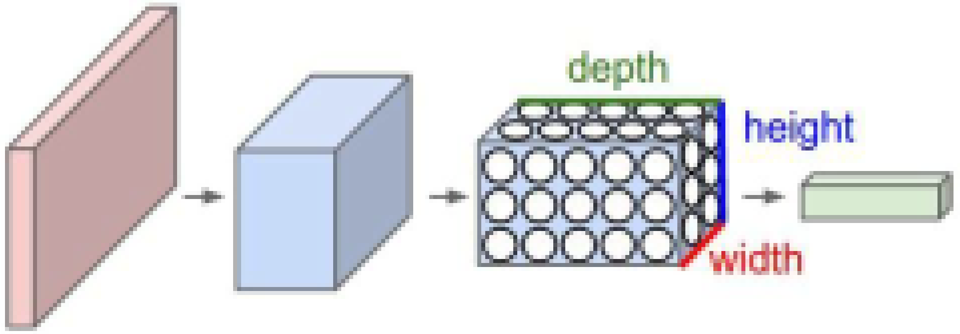


Neural Network vs CNN

- Image as input in neural network
  - Size of feature vector = HxWxC
  - For 256x256 RGB image
    - 196608 dimensions



- CNN - Special type of neural network
  - Operate with volume of data
  - Weight sharing in form of kernels



Source: http://cs231n.github.io

What is a convolution

Example 5x5 image with binary pixels

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

Example 3x3 filter

1	0	1
0	1	0
1	0	1

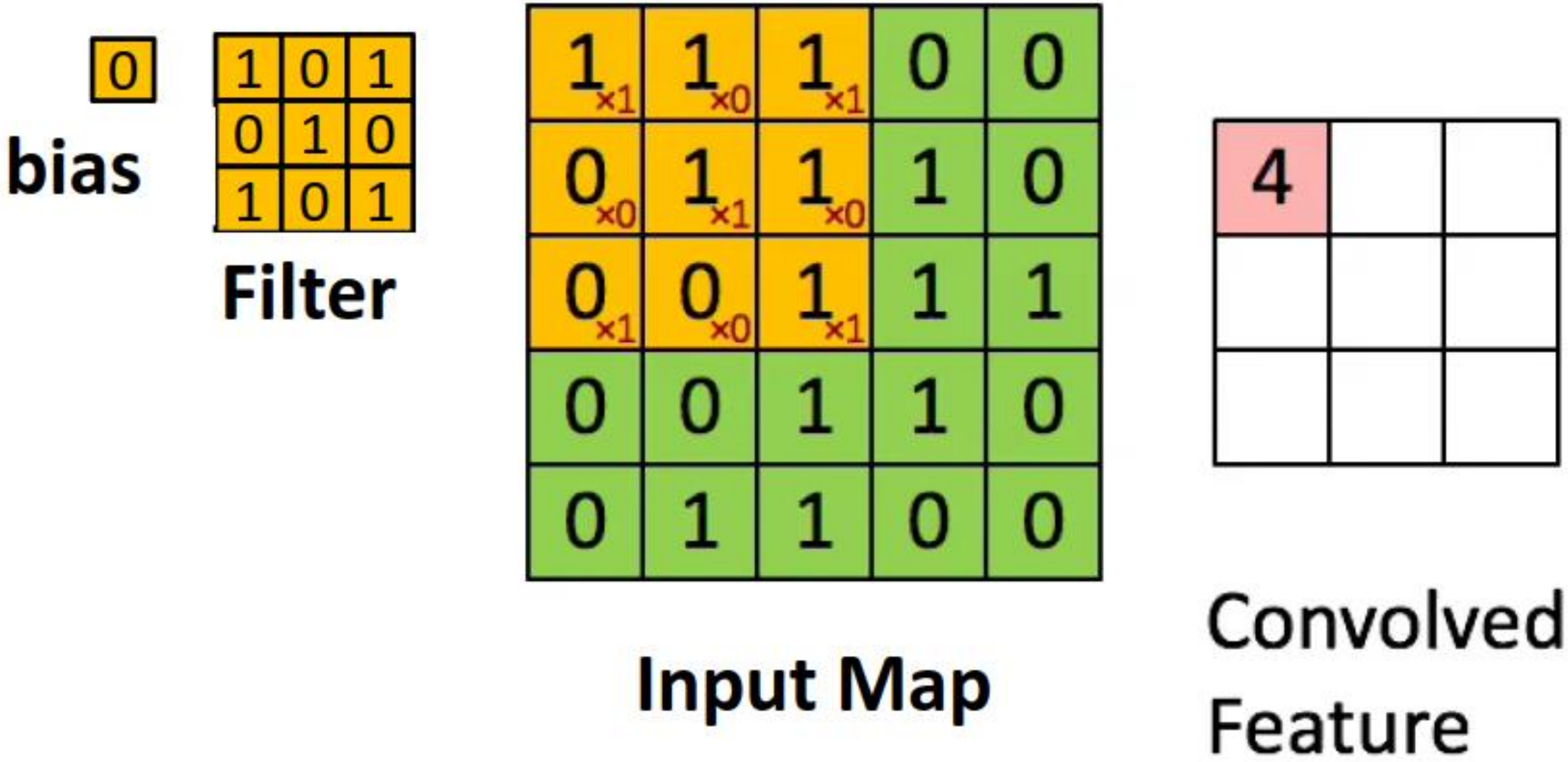
bias

0

$$z(i,j) = \sum_{k=1}^{3} \sum_{l=1}^{3} f(k,l) I(i+l, j+k) + b$$

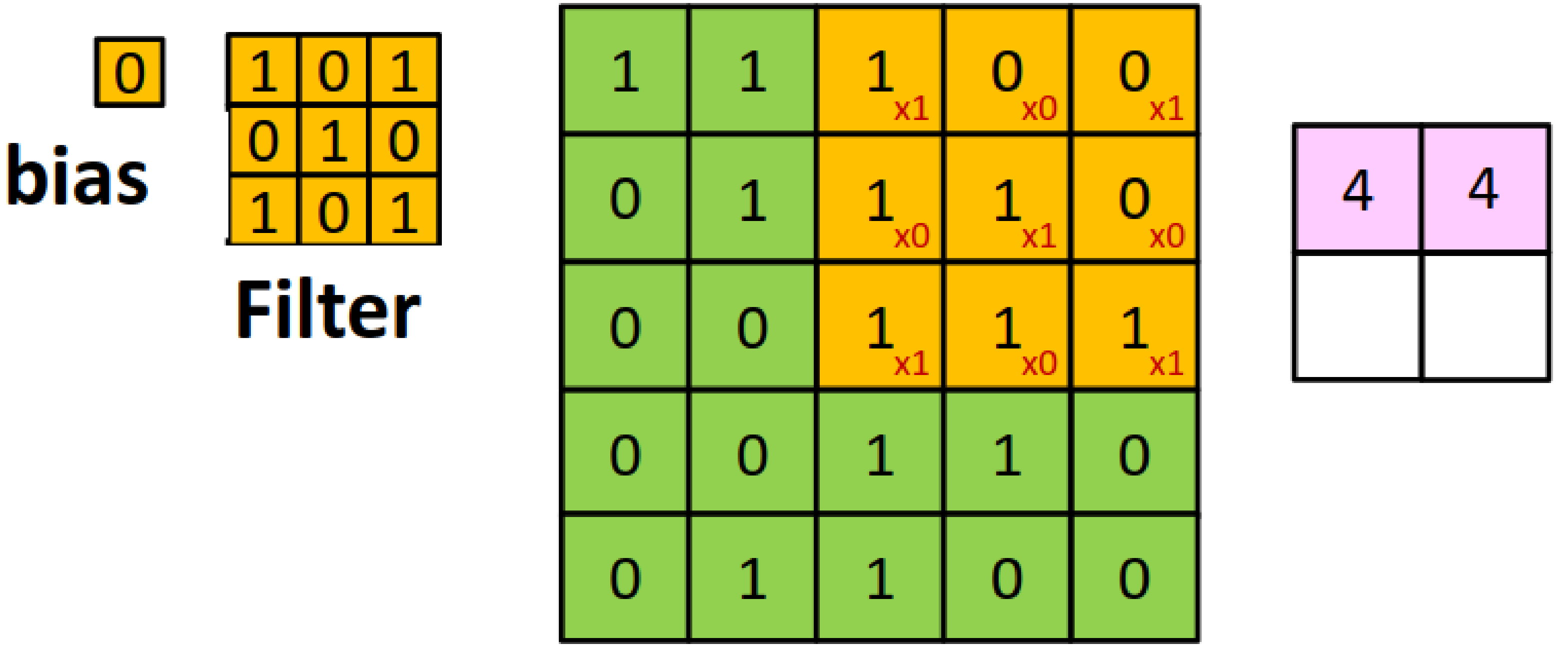
- Scanning an image with a “filter”
  - Note: a filter is really just a perceptron, with weights and a bias

What is a convolution



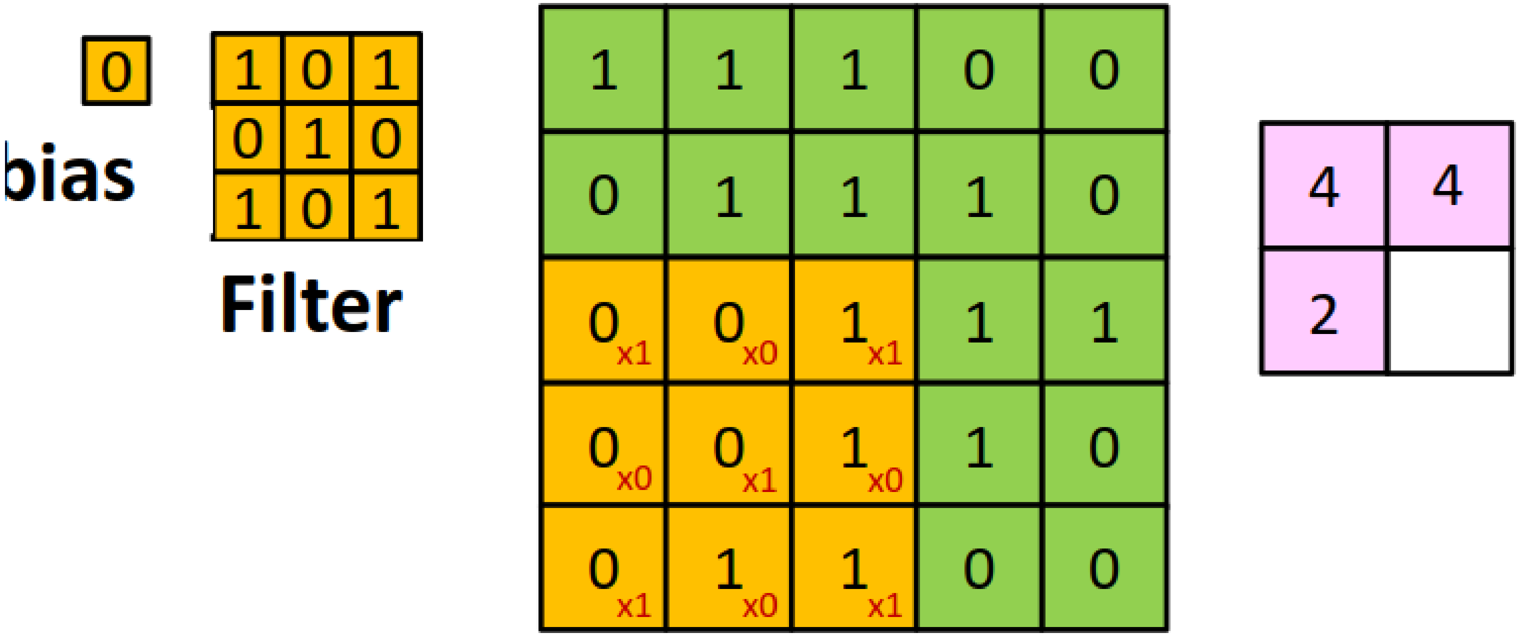
- Scanning an image with a "filter"
  - At each location, the "filter and the underlying map values are multiplied component wise, and the products are added along with the bias

The "Stride" between adjacent scanned locations need not be 1



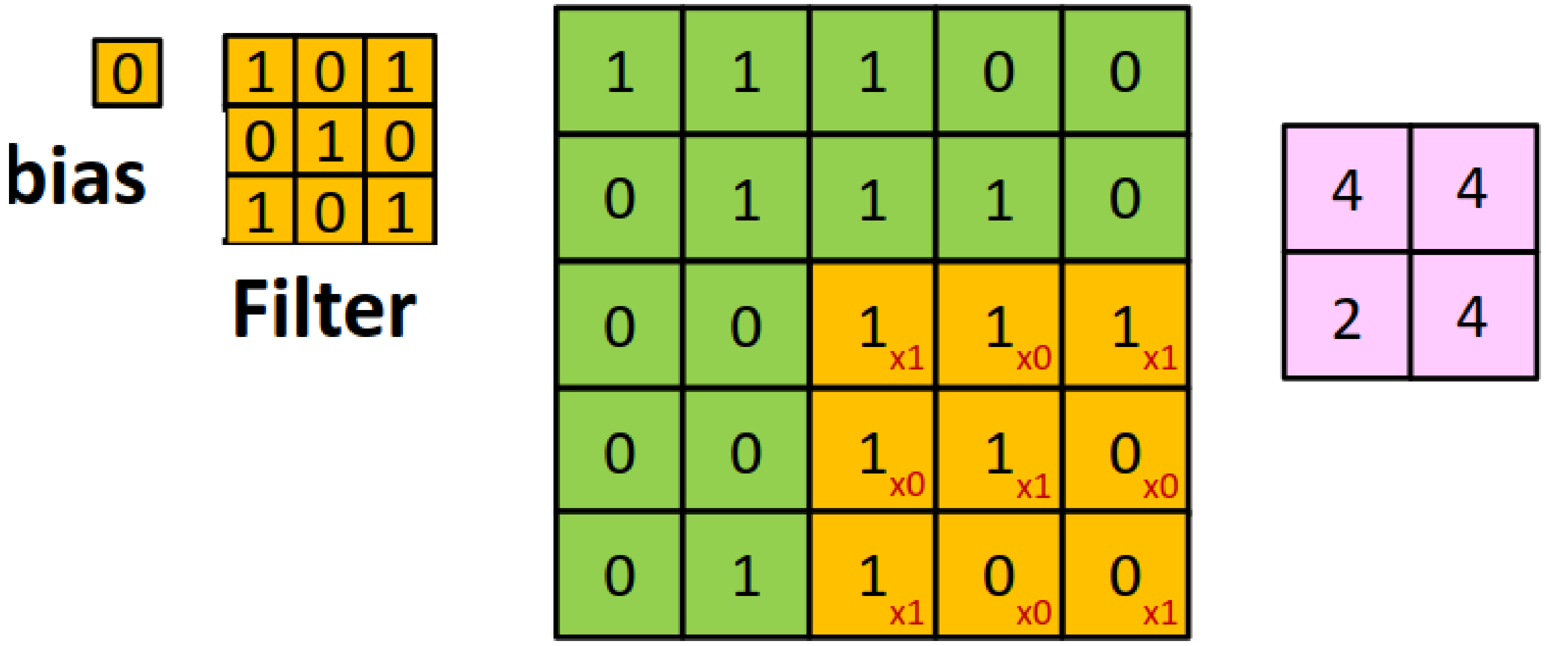
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

The “Stride” between adjacent scanned locations need not be 1



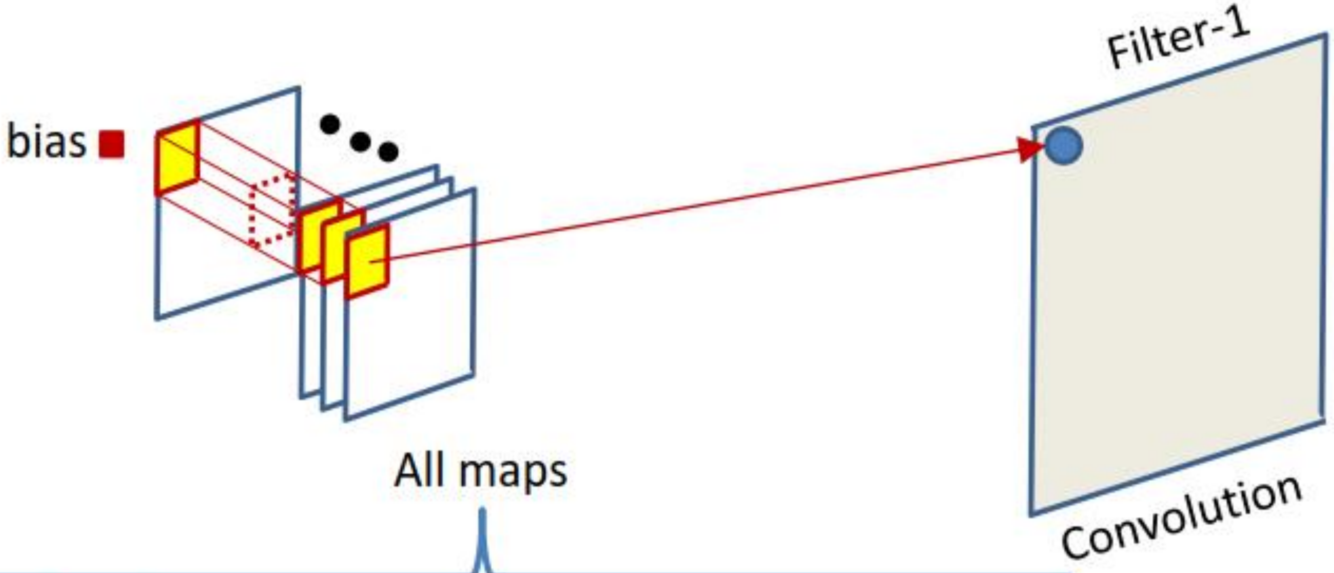
- Scanning an image with a “filter”
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a “hop” of *two* pixels per shift

The "Stride" between adjacent scanned locations need not be 1



- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

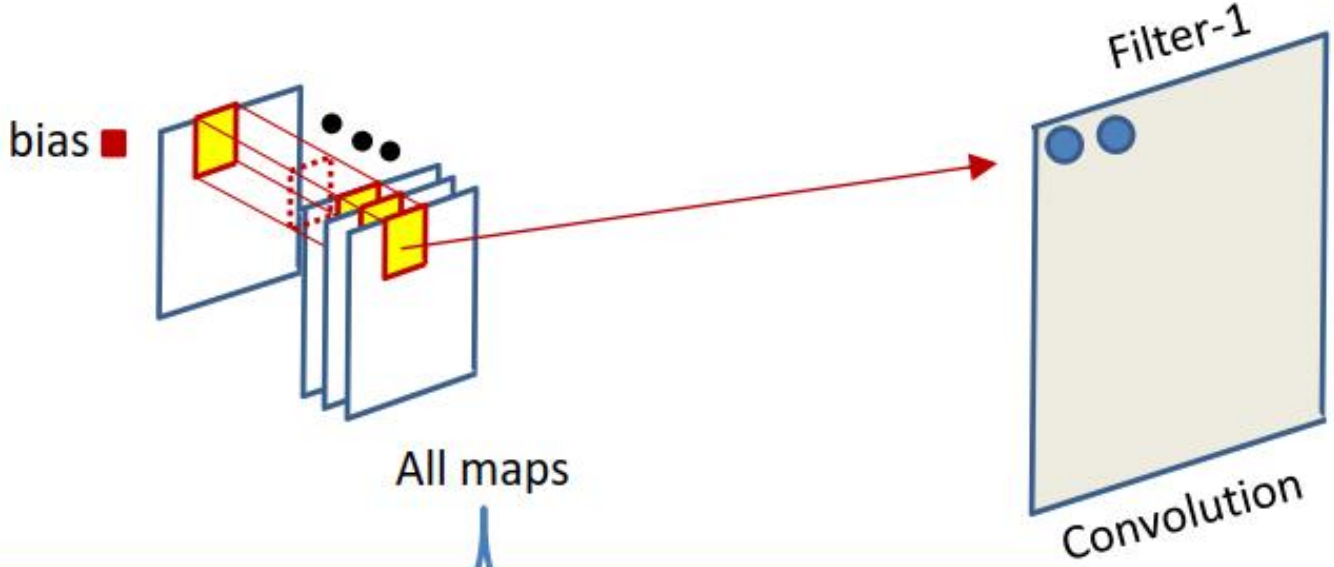
Extending to multiple input maps



$$z(i,j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b$$

- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

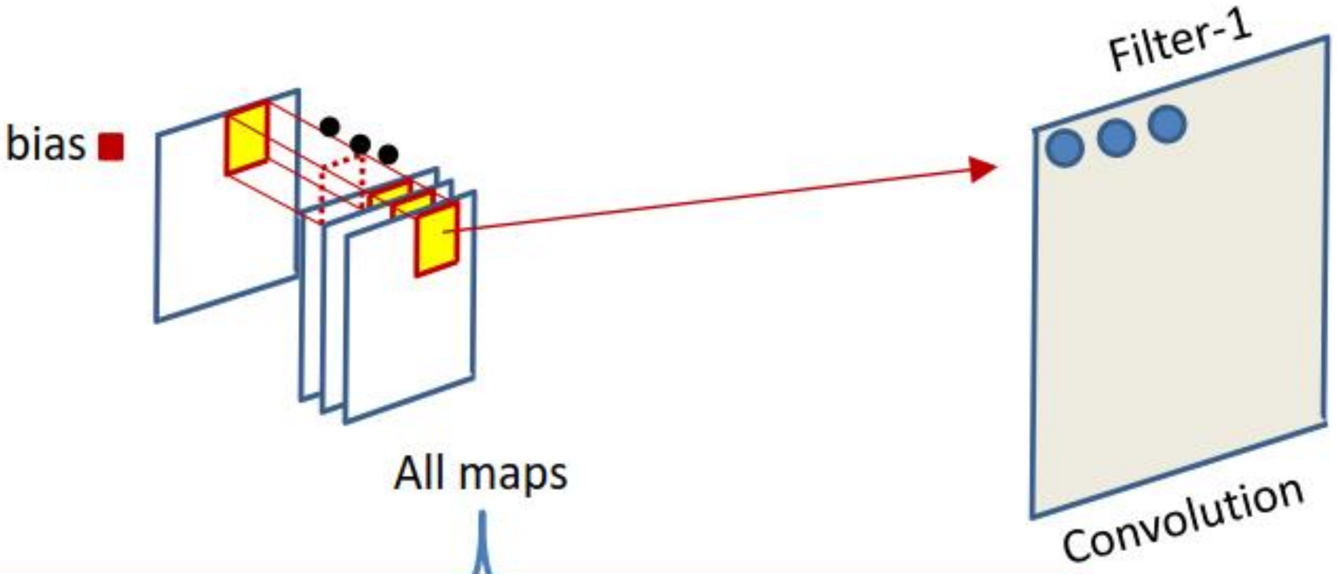
Extending to multiple input maps



$$z(i,j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b$$

- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

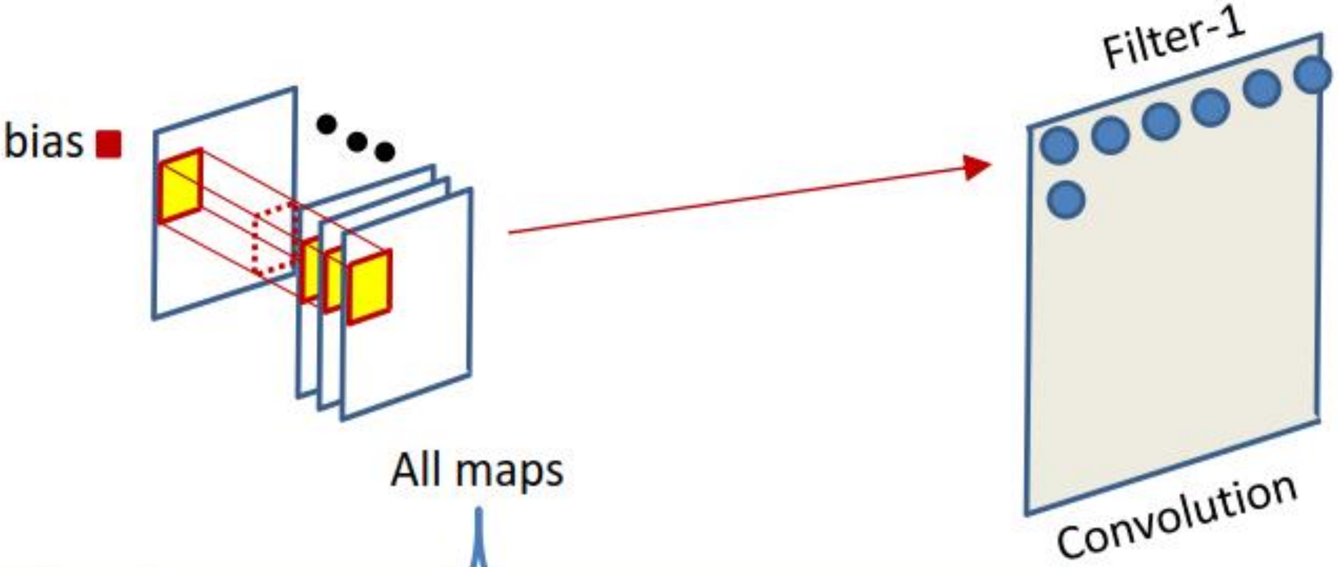
Extending to multiple input maps



$$z(i,j) = \overbrace{\sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b}^{\text{All maps}}$$

- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

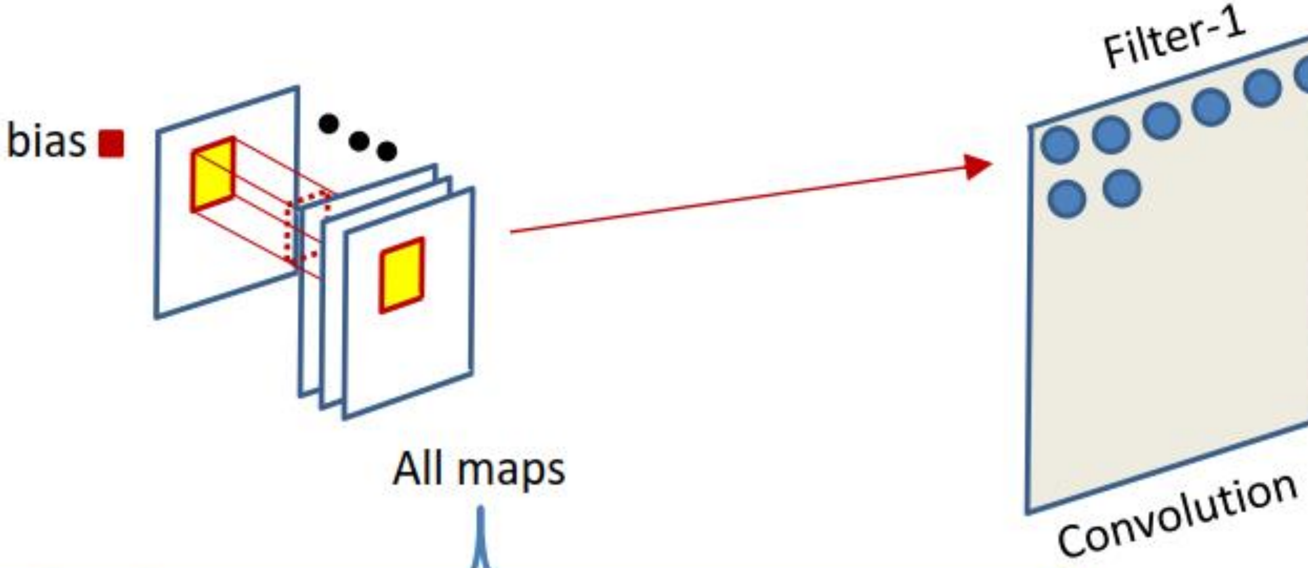
Extending to multiple input maps



$$z(i,j) = \underbrace{\sum_{p}\sum_{k=1}^{L}\sum_{l=1}^{L} w_{S,l,n}(p,k,l)Y_p(i+l,j+k)}_{\text{All maps}} + b$$

- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

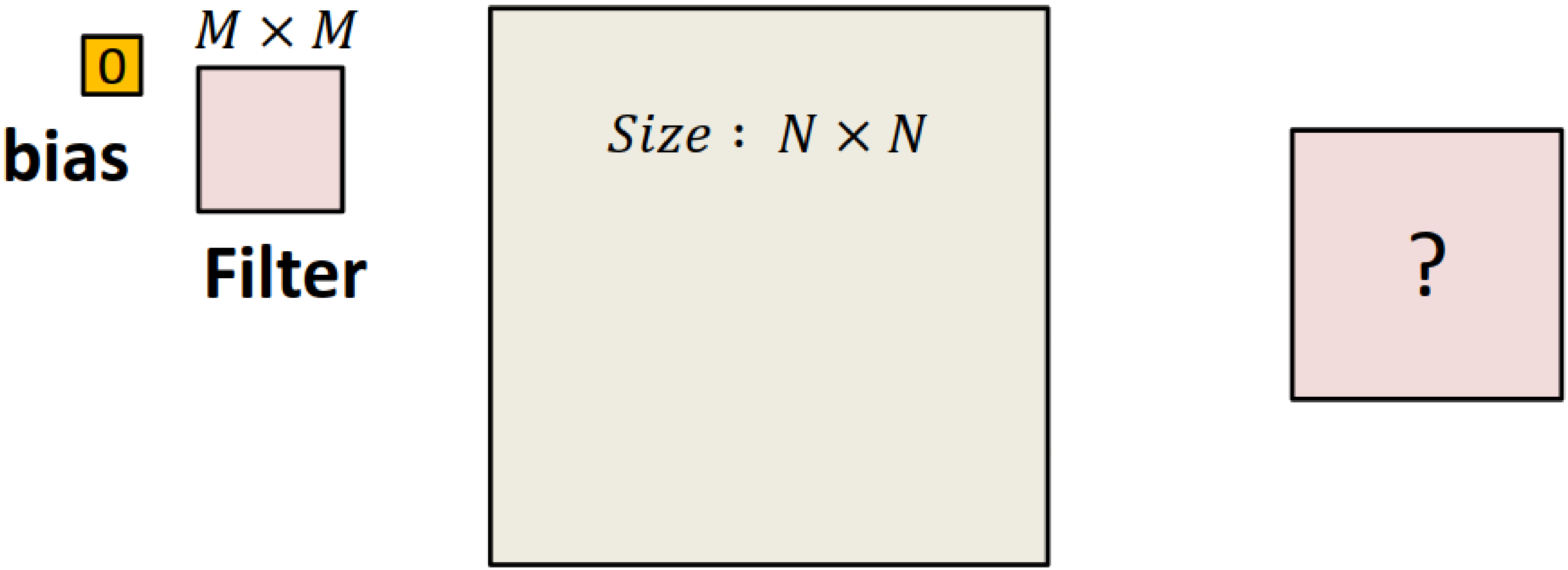
Extending to multiple input maps



$$z(i,j) = \underbrace{\sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b}_{\text{All maps}}$$

- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

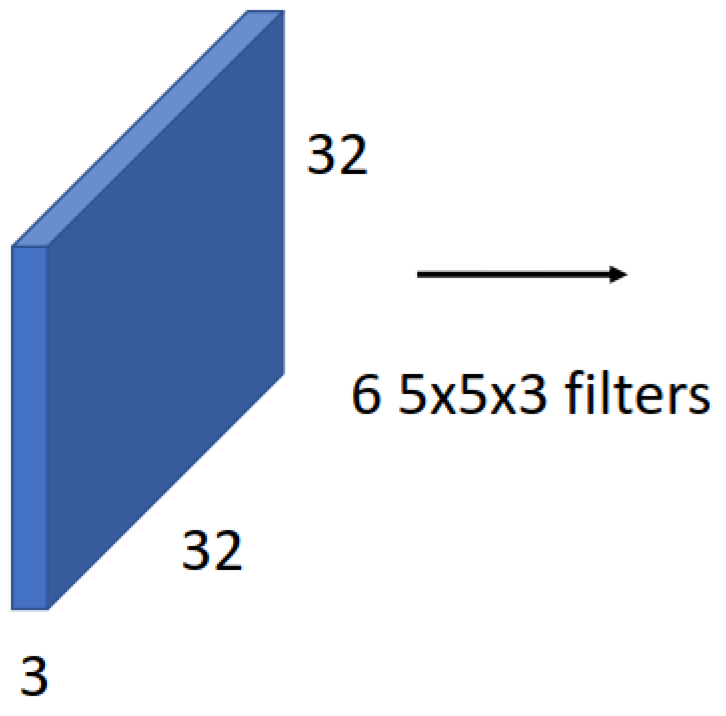
The size of the convolution



- Image size: $N \times N$
- Filter: $M \times M$
- Stride: S
- Output size (each side) = $\lfloor (N - M)/S \rfloor + 1$
  - Assuming you're not allowed to go beyond the edge of the input

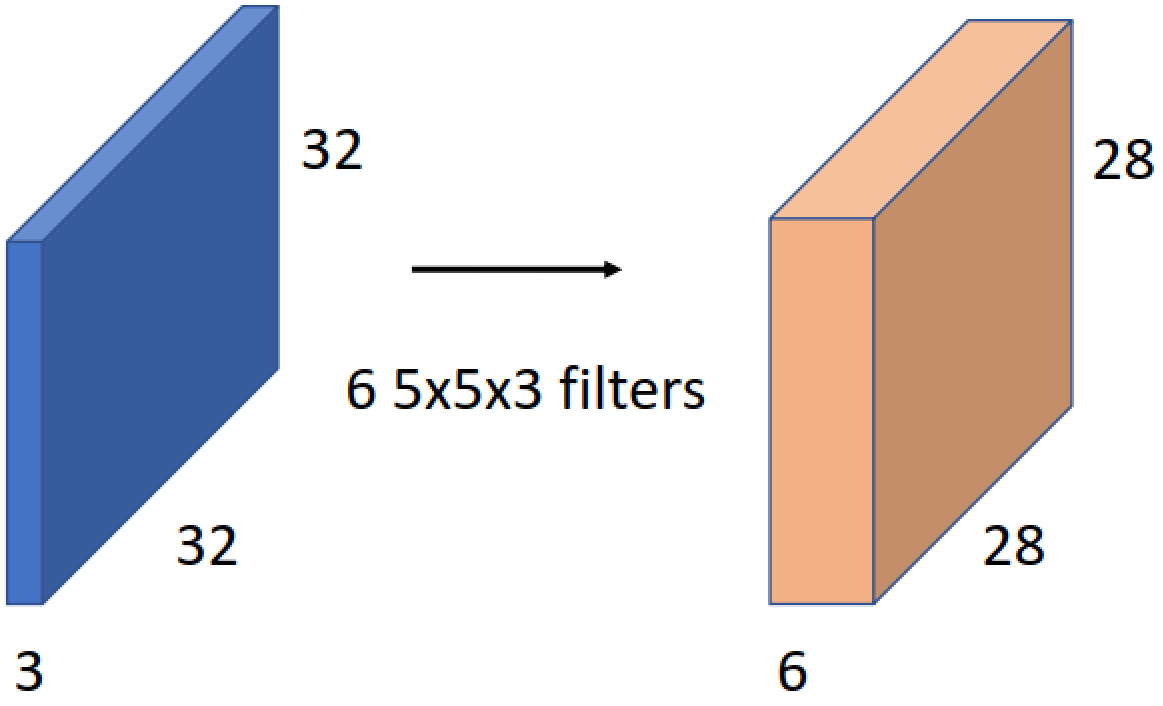
Convolutional Network

- Convolution network is a sequence of these layers

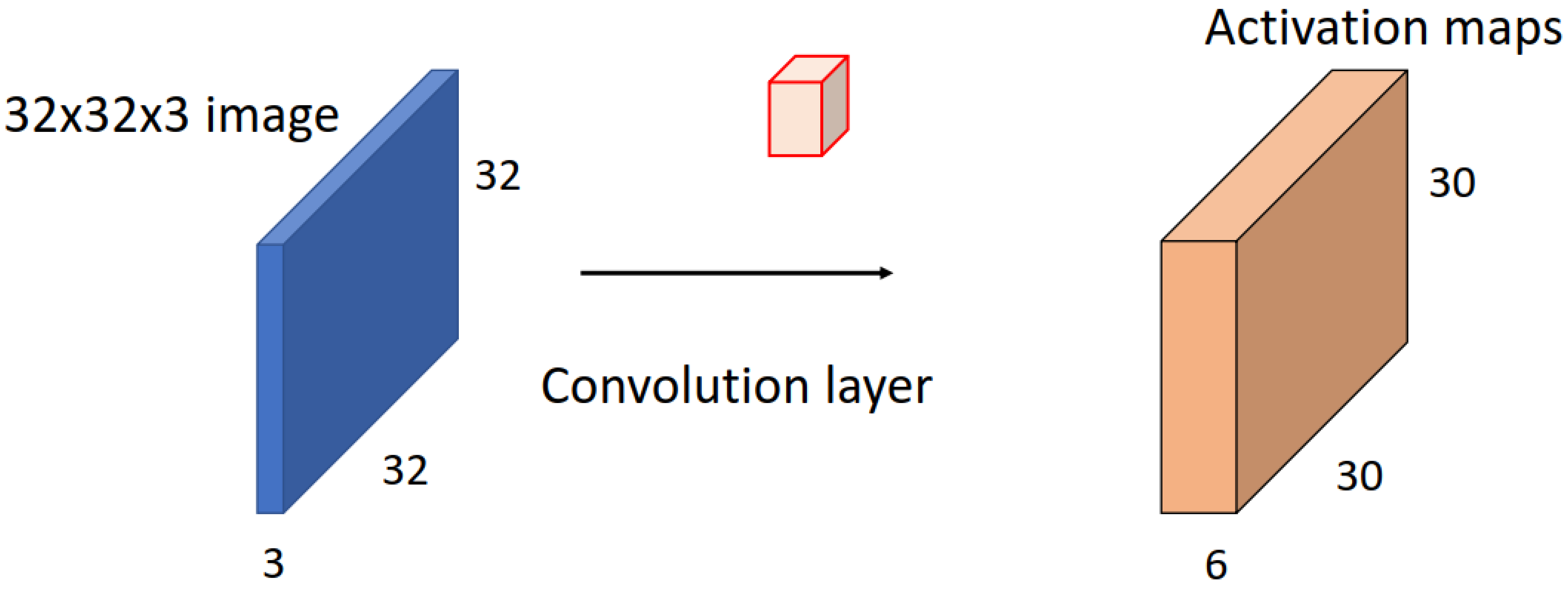


Convolutional Network

- Convolution network is a sequence of these layers



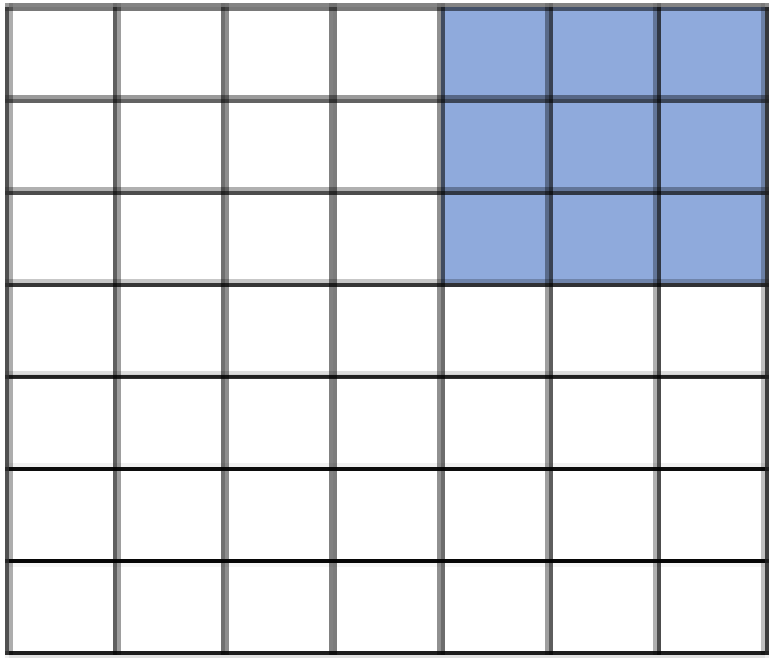
Parameters



6 3x3x3 kernels – 6x3x3x3 parameters = 162

2D Convolution - dimensions

7x7 map



3x3 filter

Output activation map 5x5

Output size

N-F+1

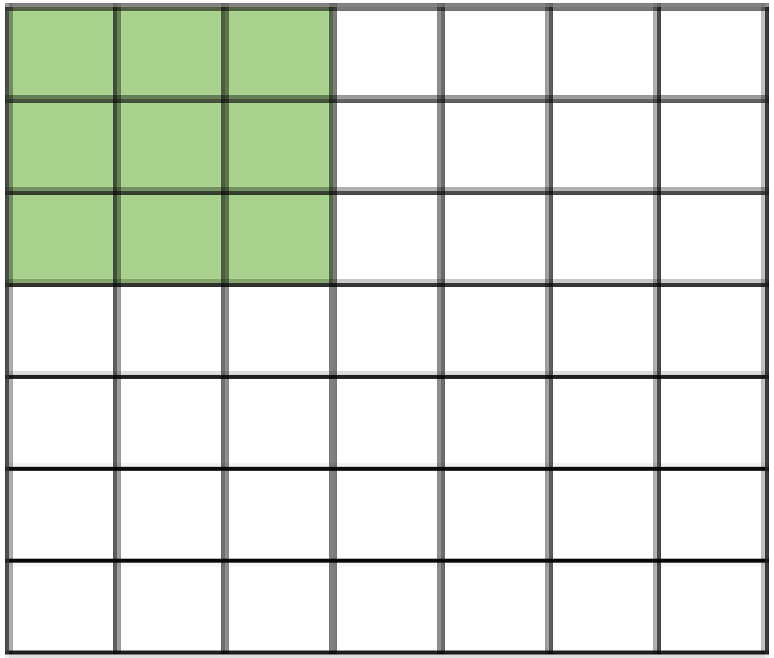
$(7 - 3 + 1) = 5$

N – input size

F – filter size

Stride

7x7 map

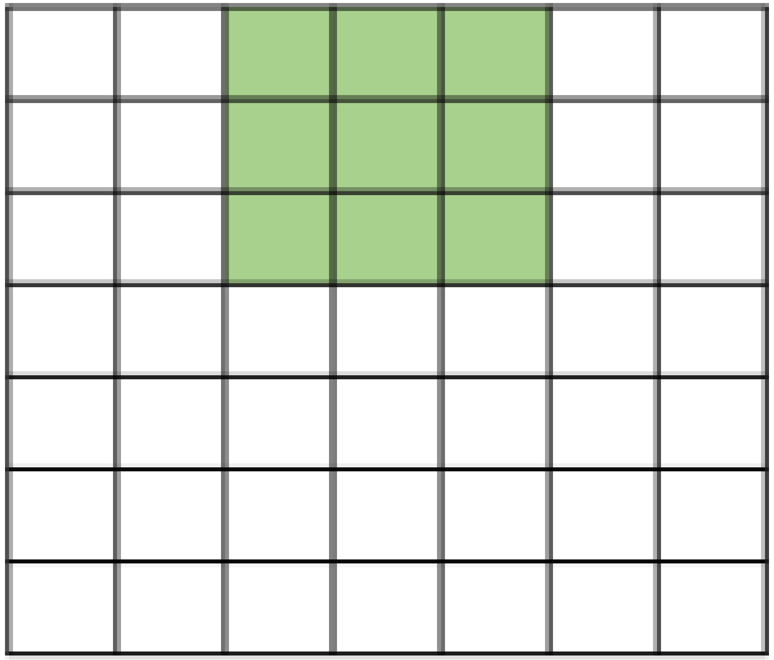


3x3 filter

Filter applied with stride 2

Stride

7x7 map

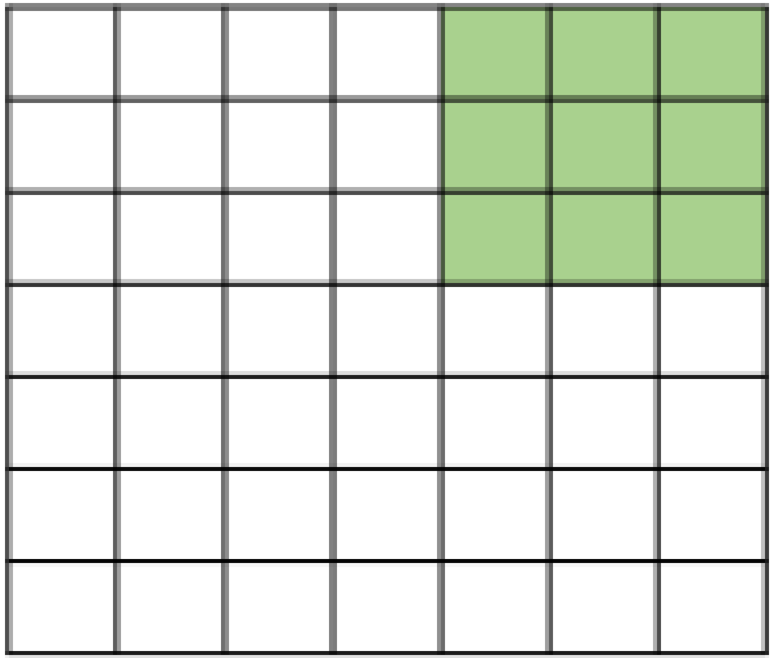


3x3 filter

Filter applied with stride 2

Stride

7x7 map



3x3 filter

Filter applied with stride 2

Activation map size 3x3

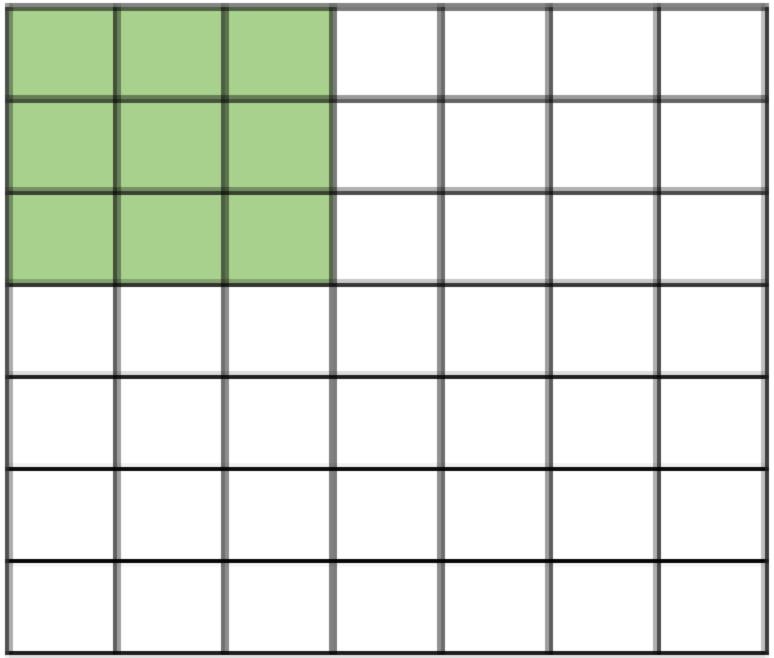
Output size

(7-3)/2 + 1 = 3

(N-F)/S + 1

Stride

7x7 map

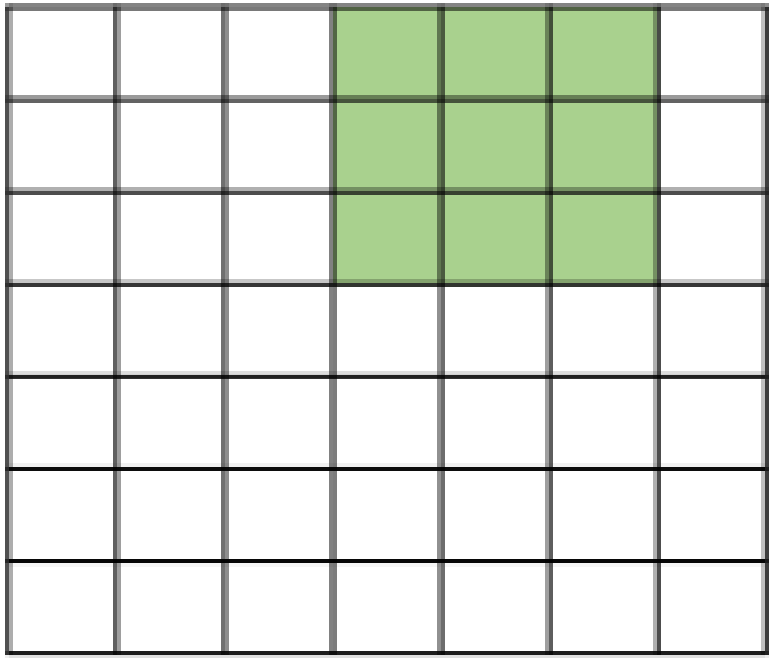


3x3 filter

Filter applied with stride 3

Stride

7x7 map



3x3 filter

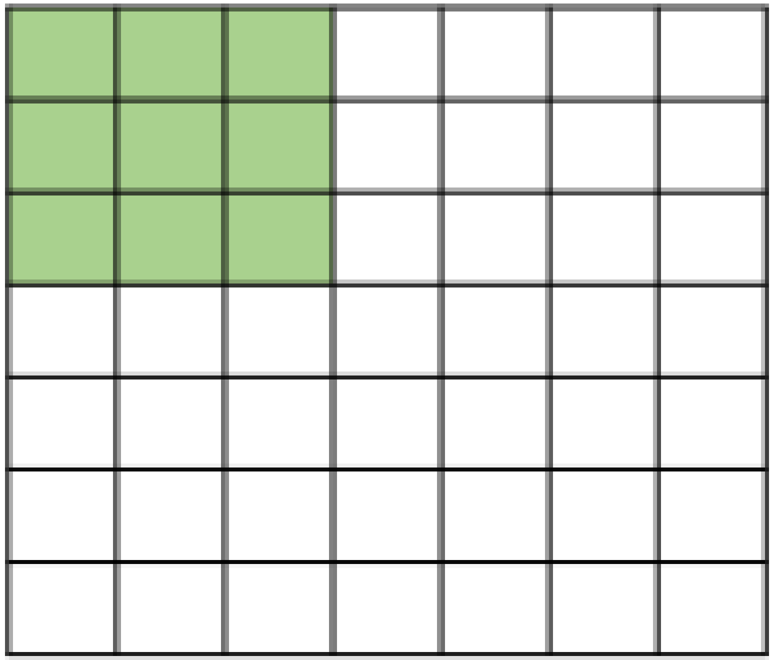
Filter applied with stride 3

Cannot cover perfectly

Not all parameters will fit

Stride

7x7 map



3x3 filter

Output size (N-F)/S + 1

N = 7, F = 3

Stride 1

(7-3)/1 + 1 => 5

Stride 2

(7-3)/2 + 1 => 3

Stride 3

(7-3)/3 + 1 => 2.33

Solution

0

bias

1	0	1
0	1	0
1	0	1

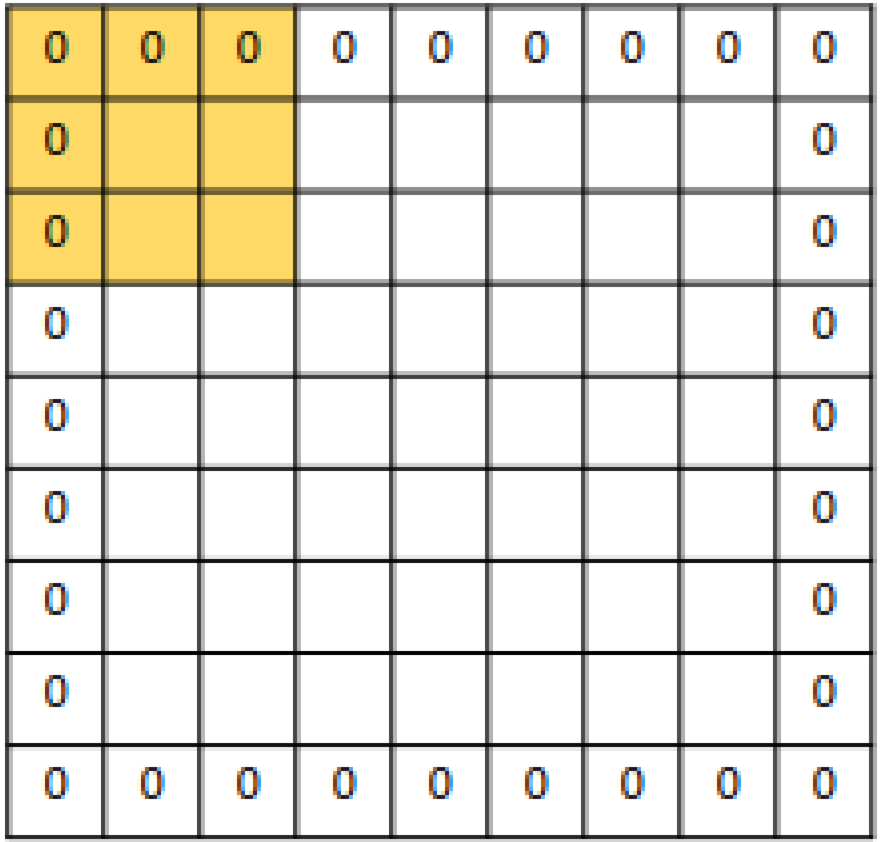
Filter

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

- Zero-pad the input
  - Pad the input image/map all around
  - Pad as symmetrically as possible, such that..
  - **For stride 1, the result of the convolution is the same size as the original image**

Padding

- Zero padding in the input



For 7x7 input and 3x3 filter

If we have padding of one pixel

Output
7x7

Size (recall (N-F)/S+1)
(N-F+2P)/S + 1

Padding

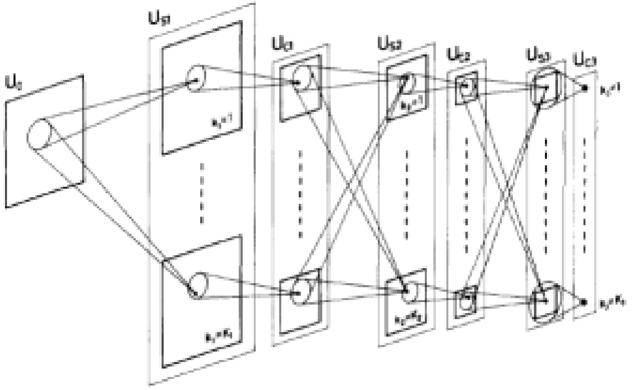
- Zero padding in the input

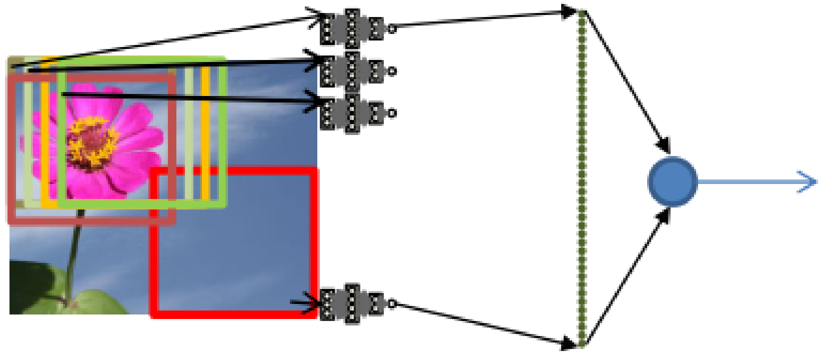
0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Common to see,
(F-1)/2 padding with stride 1 to preserve the map size

$$N = (N-F+2P)/S + 1$$
$$\Rightarrow (N-1)S = N-F+2P$$
$$\Rightarrow P = (F-1)/2$$

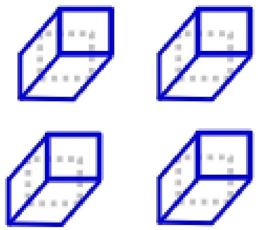
Why convolution?





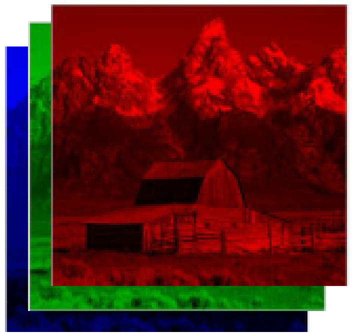
- Convolutional neural networks are, in fact, equivalent to *scanning* with an MLP
  - Just run the entire MLP on each block separately, and combine results
    - As opposed to scanning (convolving) the picture with individual neurons/filters
  - Even computationally, the number of operations in both computations is identical
    - The neocognitron in fact views it equivalently to a scan
- So why convolutions?

Convolutional Neural Networks



K_1 total filters
Filter size: $L \times L \times 3$

Small enough to capture fine features
(particularly important for scaled-down images)

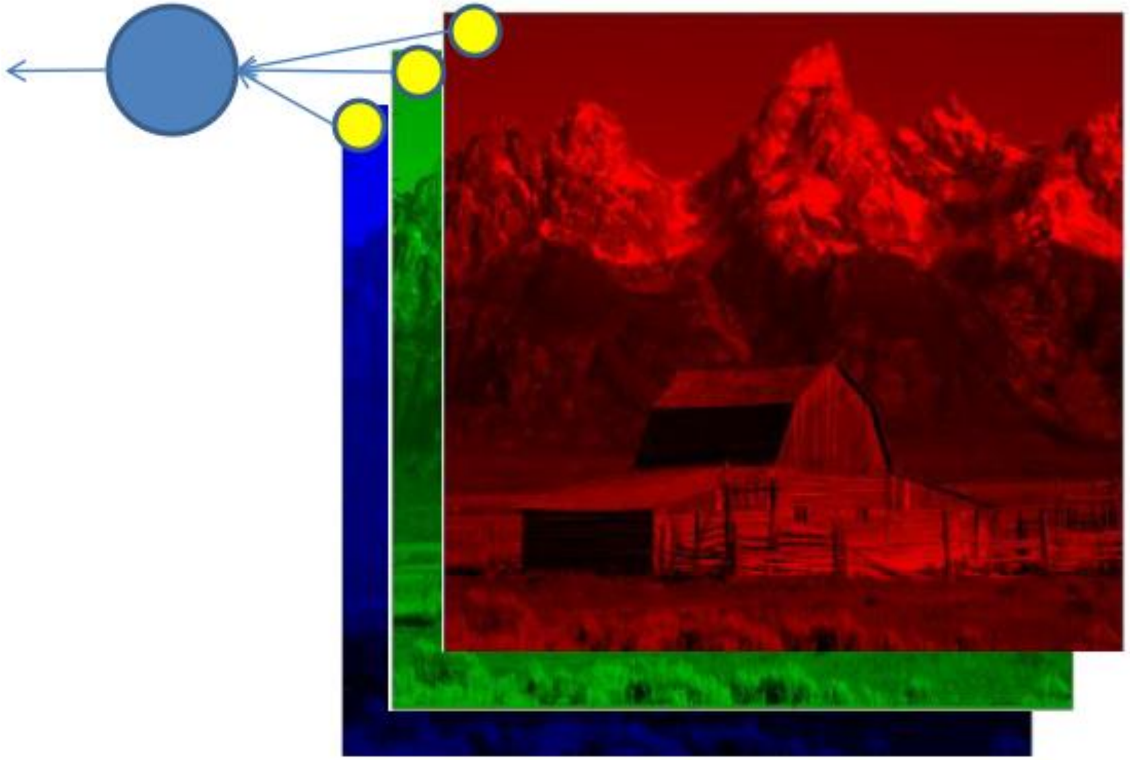


$I \times I$ *image*

What on earth is this?

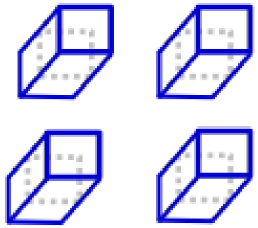
- Input is convolved with a set of K_1 filters
  – Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  – Filters are typically 5x5, 3x3, or even 1x1

The 1x1 filter

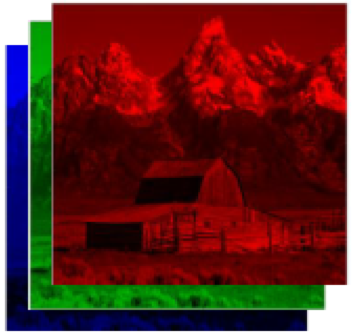


- A 1x1 filter is simply a perceptron that operates over the *depth* of the map, but has no spatial extent
  - Takes one pixel from each of the maps (at a given location) as input

Convolutional Neural Networks



K_1 total filters
Filter size: $L \times L \times 3$



$I \times I$ *image*

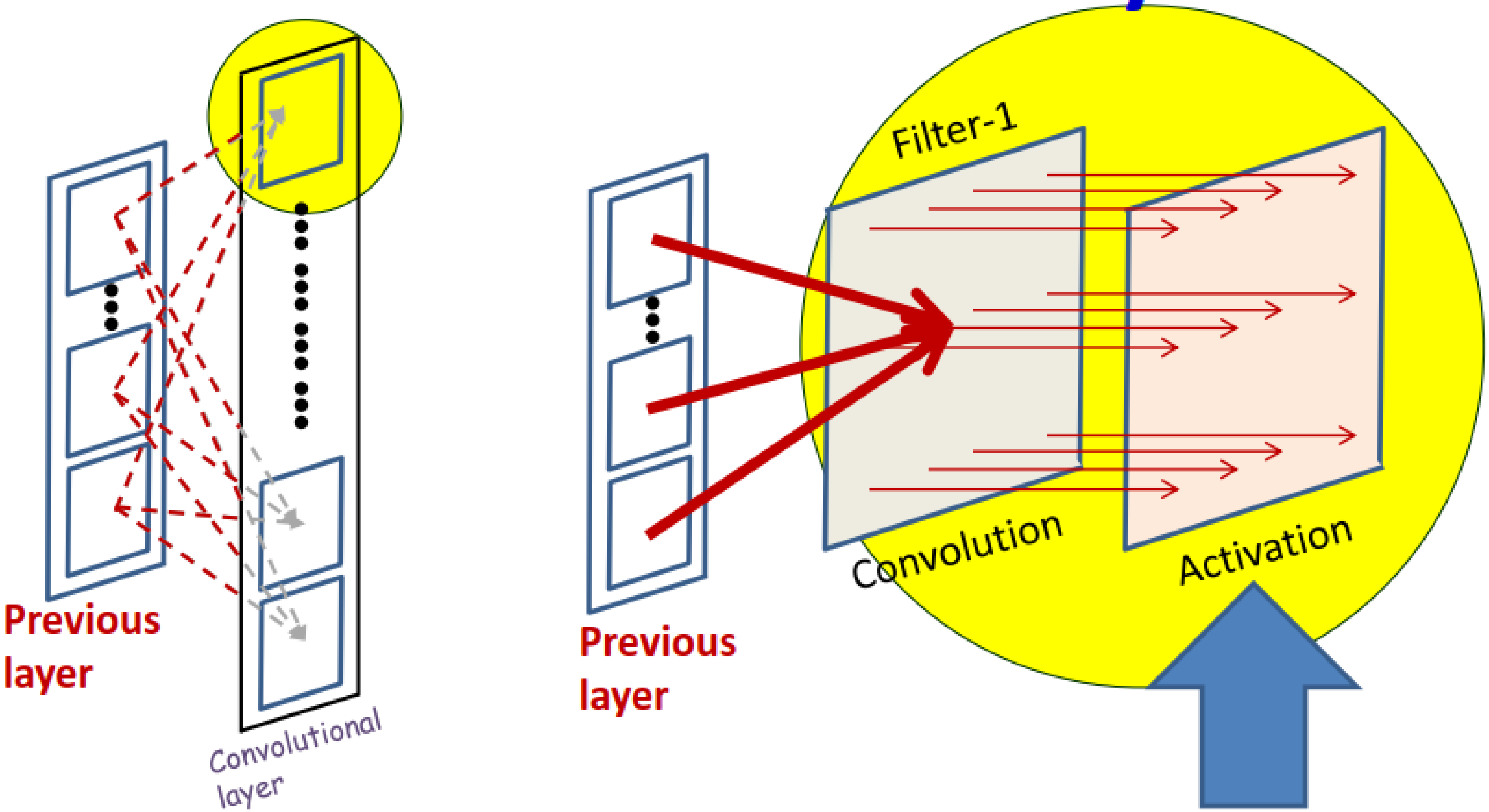
Parameters to choose: K_1, L and S

1. Number of filters K_1
2. Size of filters $L \times L \times 3 + bias$
3. *Stride* of convolution S

Total number of parameters: $K_1(3L^2 + 1)$

- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - **Better notation:** Filters are typically 5x5(x3), 3x3(x3), or even 1x1(x3)
  - ***Typical stride***: 1 or 2

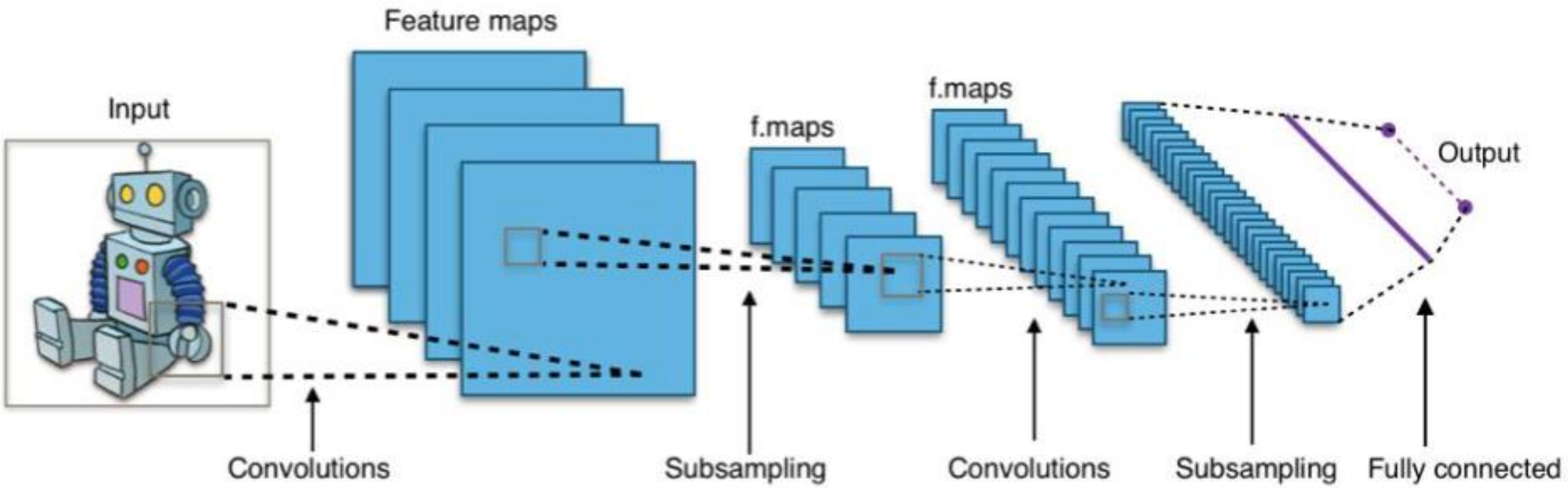
A convolutional layer



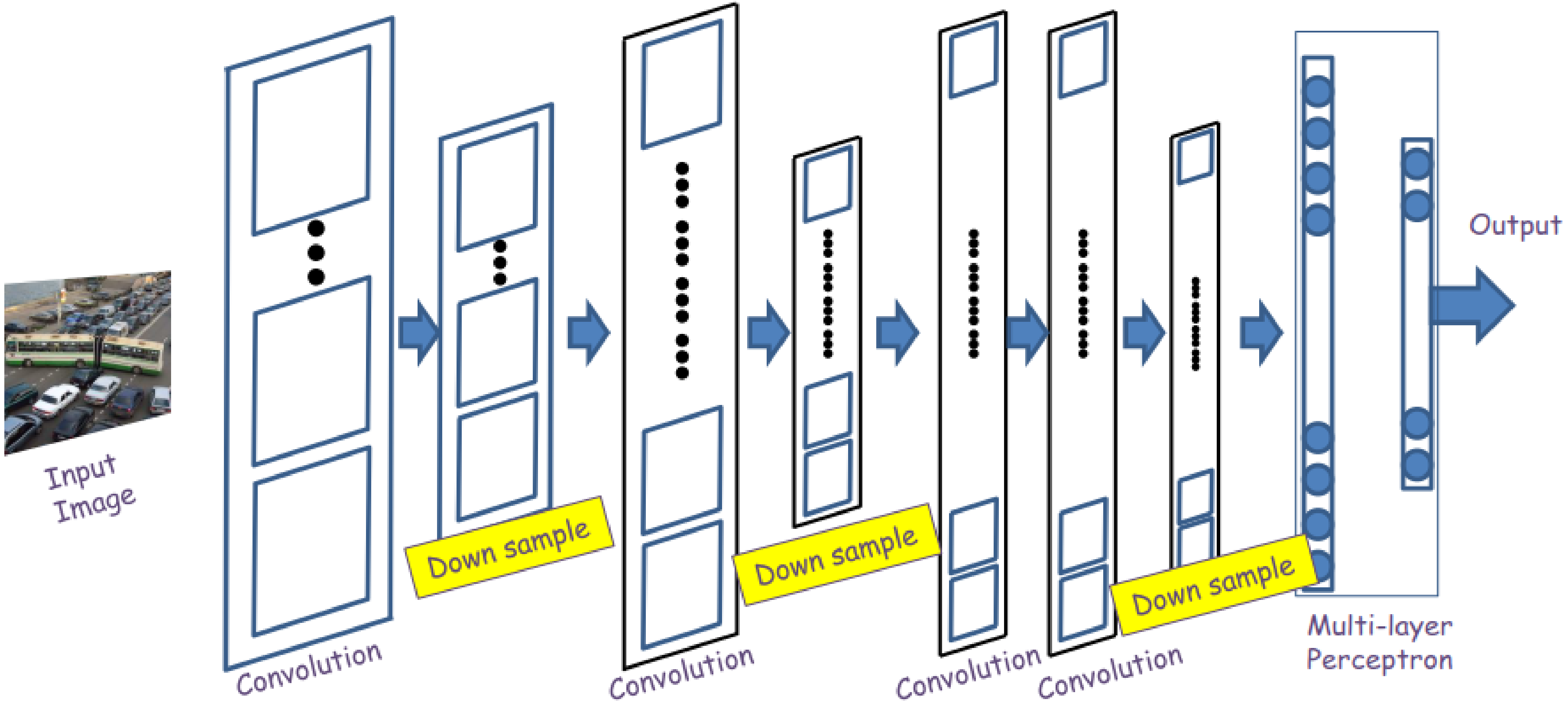
- The convolution operation results in a convolution map
- An *Activation* is finally applied to every entry in the map

Convolutional Neural Network (CNN)

- A class of Neural Networks
  - Takes image as input (mostly)
  - Make predictions about the input image



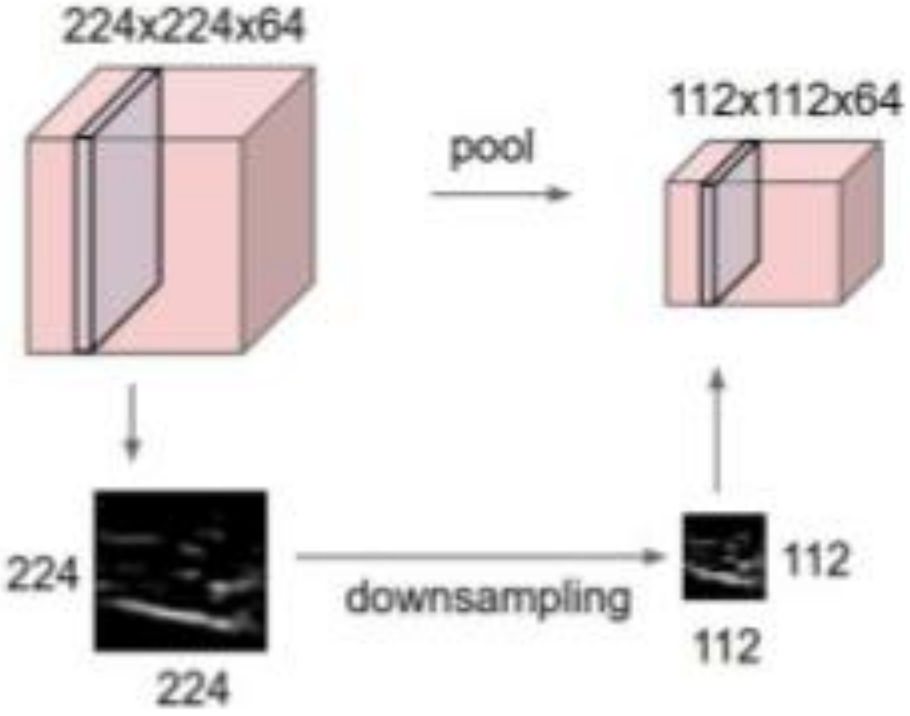
The other component Downsampling/Pooling



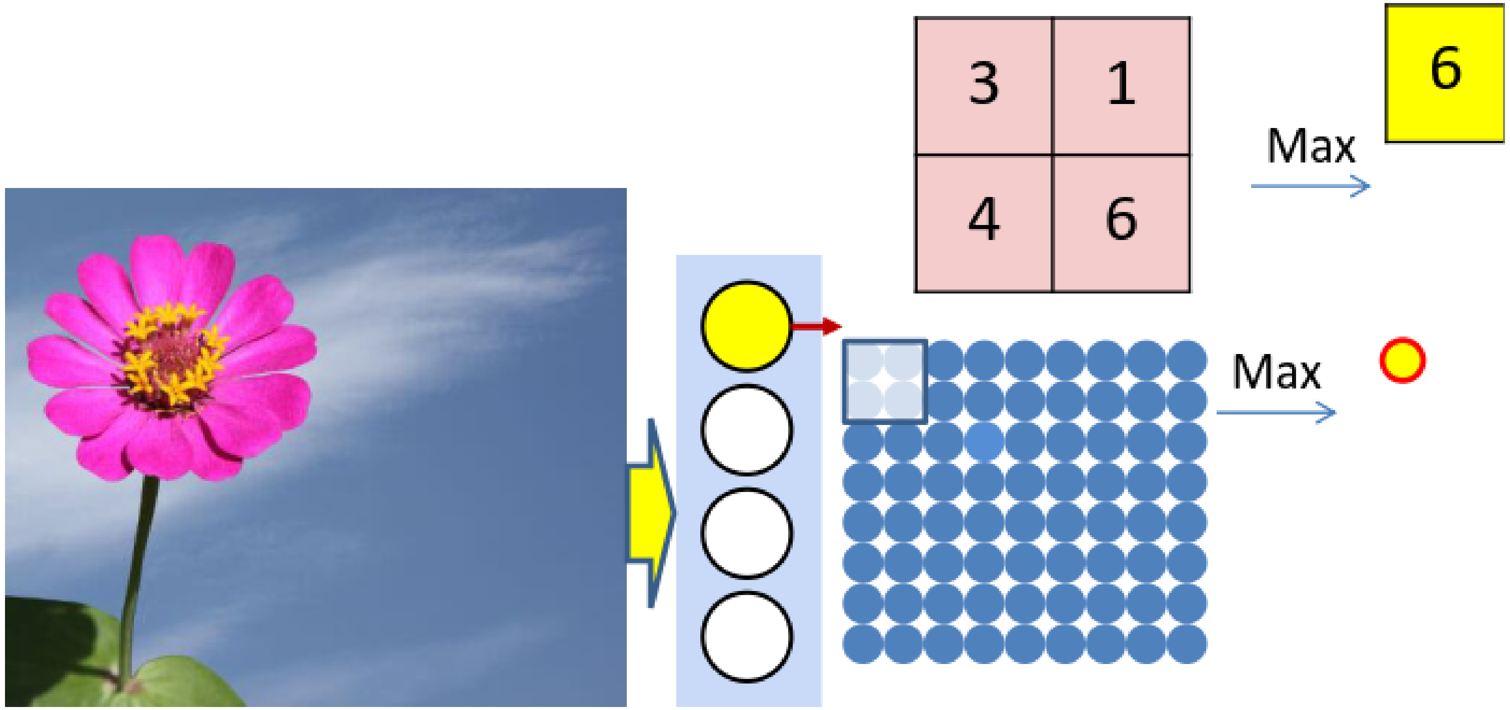
- Convolution (and activation) layers are followed intermittently by “downsampling” (or “pooling”) layers
  - Often, they alternate with convolution, though this is not necessary

Pooling

- Makes the representations smaller
- Operates over each activation map independently



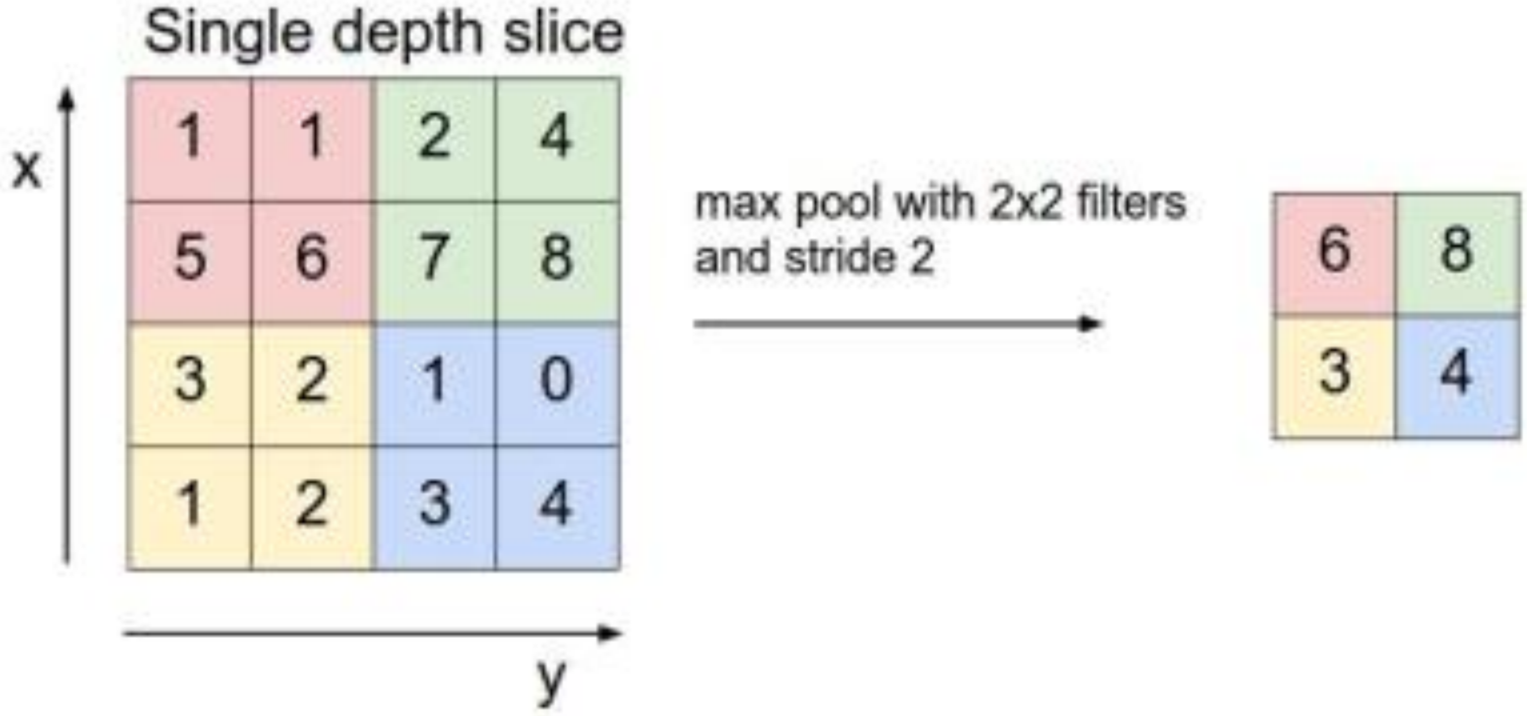
Recall: Max pooling



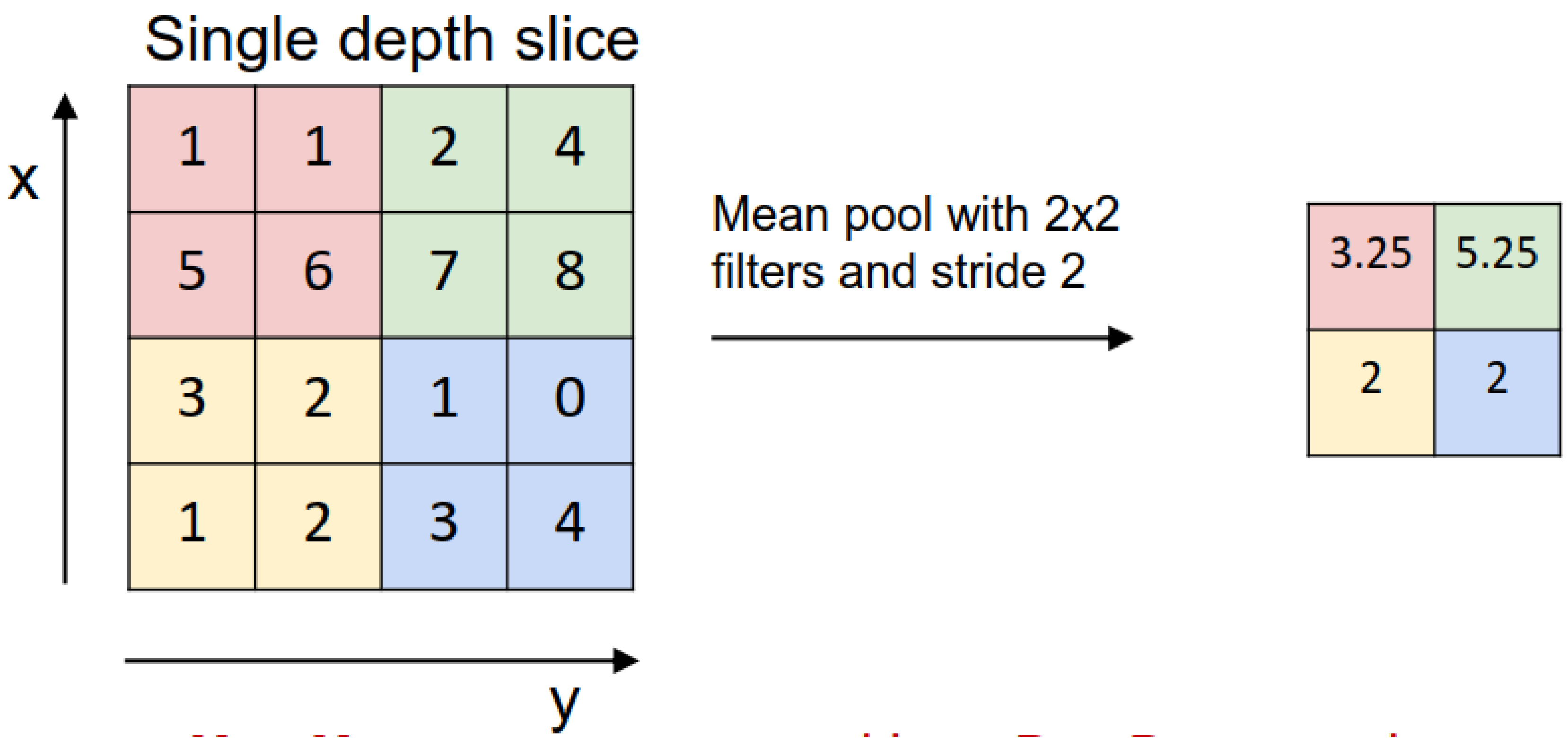
- Max pooling selects the largest from a pool of elements
- Pooling is performed by "scanning" the input

Pooling

- Kernel size
- Stride

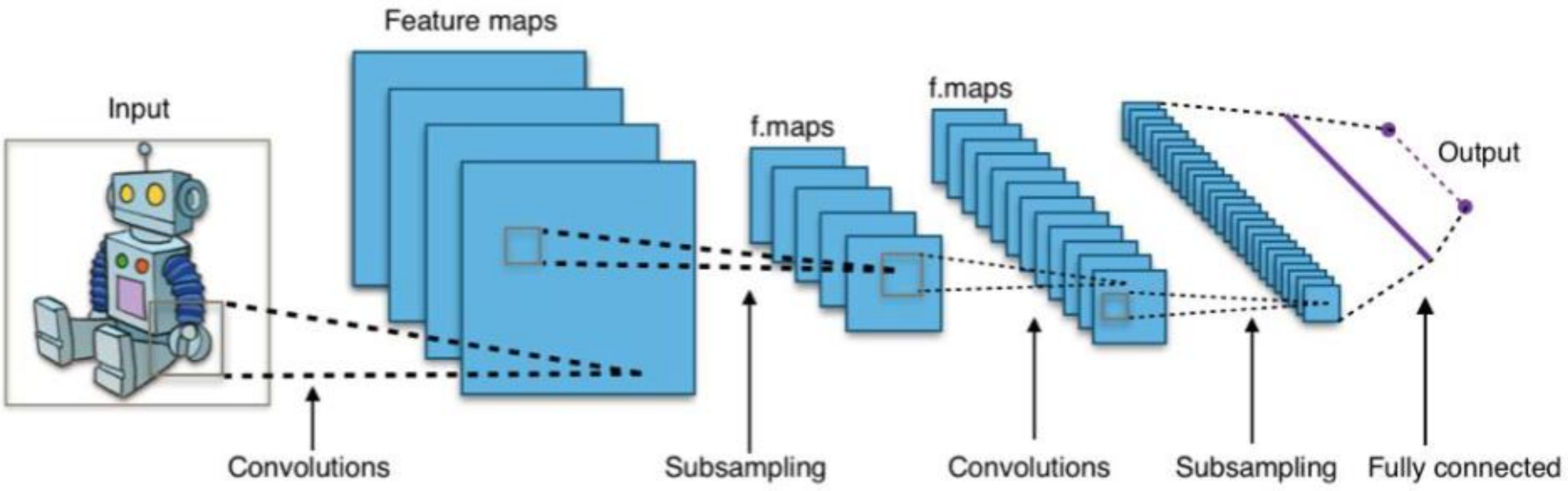


Alternative to Max pooling: Mean Pooling

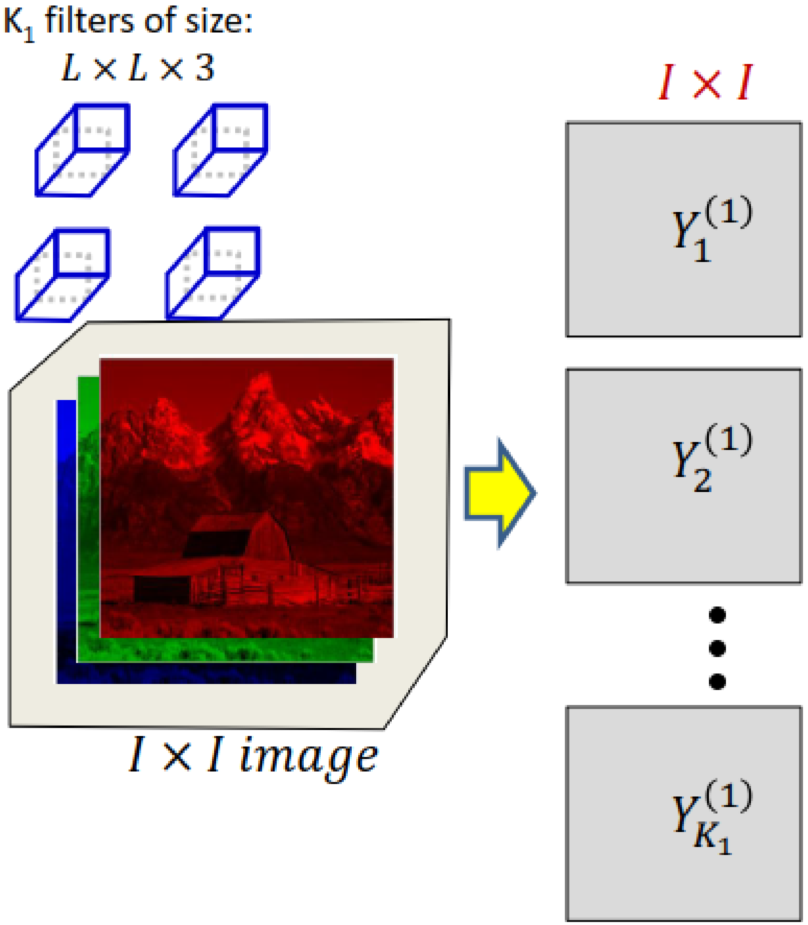


Convolutional Neural Network (CNN)

- A class of Neural Networks
  - Takes image as input (mostly)
  - Make predictions about the input image



Convolutional Neural Networks



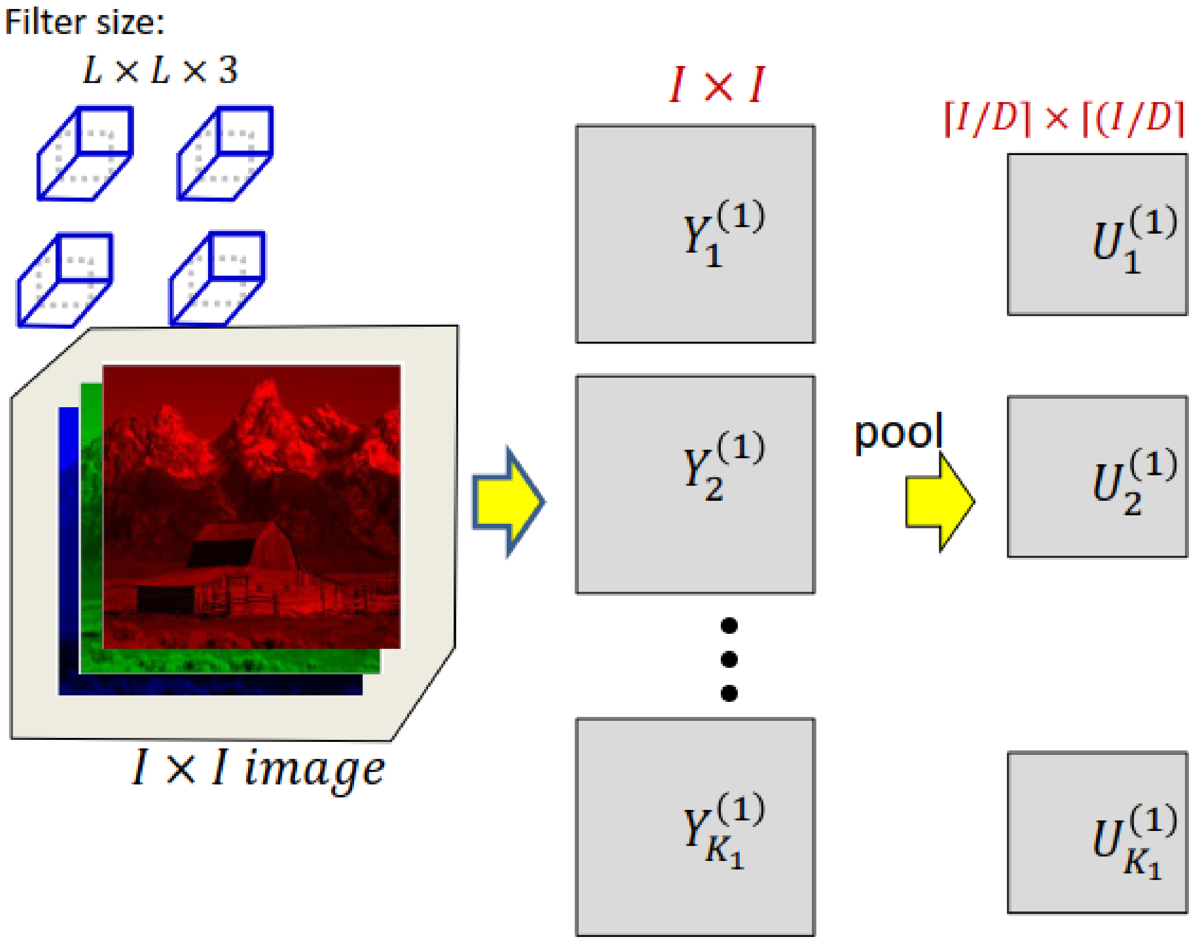
The layer includes a convolution operation followed by an activation (typically RELU)

$$z_m^{(1)}(i,j) = \sum_{c \in \{R,G,B\}} \sum_{k=1}^{L} \sum_{l=1}^{L} w_m^{(1)}(c,k,l) I_c(i+k, j+l) + b_m^{(1)}$$

$$Y_m^{(1)}(i,j) = f\left(z_m^{(1)}(i,j)\right)$$

- **First convolutional layer:** Several convolutional filters
  - Filters are “3-D” (third dimension is color)
  - Convolution followed typically by a RELU activation
- Each filter creates a single 2-D output map

Convolutional Neural Networks

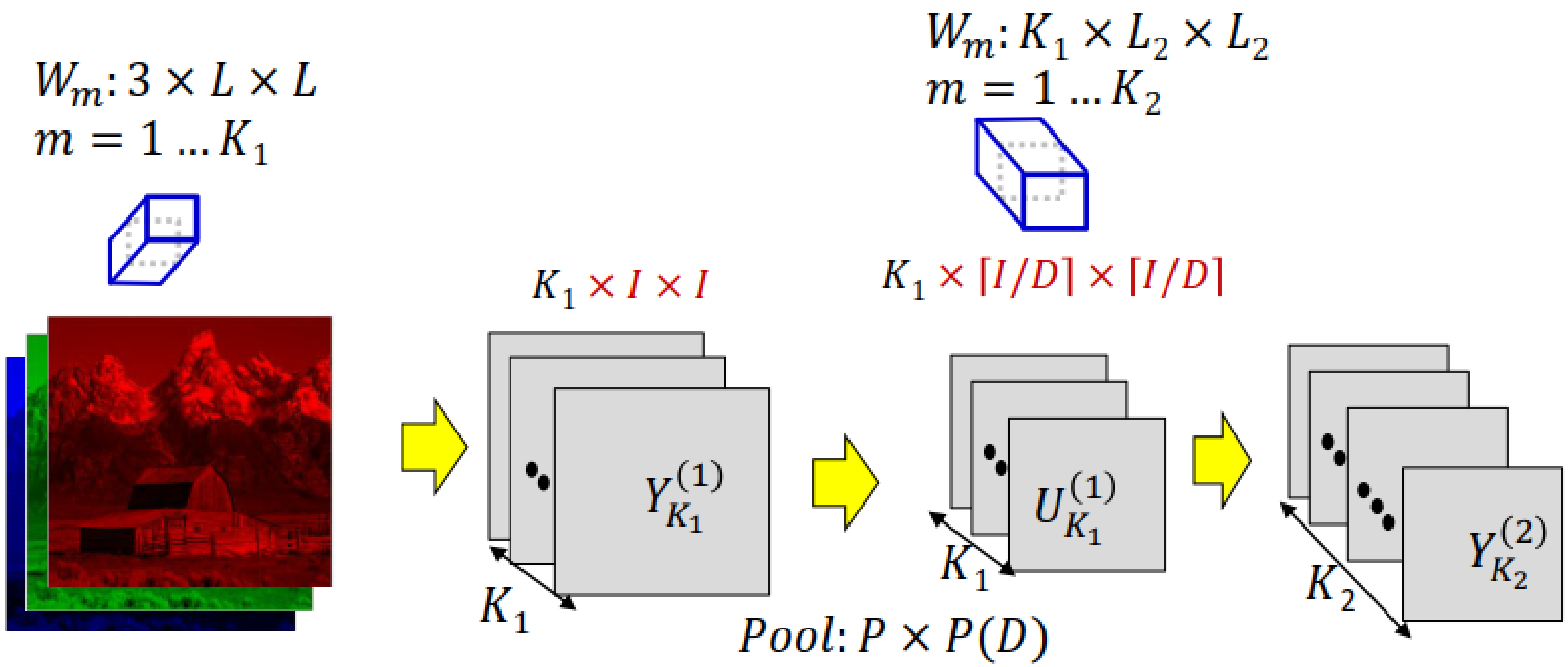


The layer pools PxP blocks of Y into a single value
It employs a stride D between adjacent blocks

$$U_m^{(1)}(i,j) = \max_{\substack{k \in \{(i-1)D+1,\ iD\},\\ l \in \{(j-1)D+1,\ jD\}}} Y_m^{(1)}(k,l)$$

- **First downsampling layer:** From each $P \times P$ block of each map, *pool* down to a single value
  - For max pooling, during training keep track of which position had the highest value

Convolutional Neural Networks

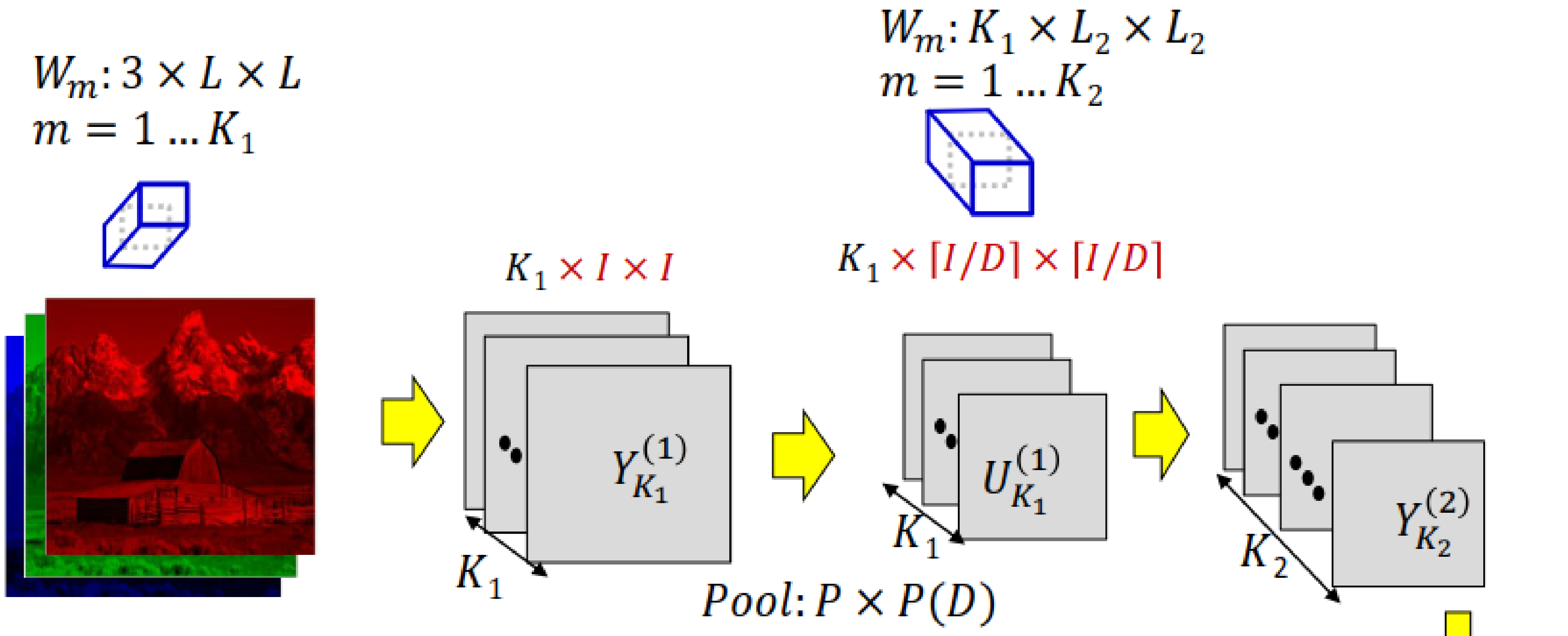


$$z_m^{(n)}(i,j) = \sum_{r=1}^{K_{n-1}} \sum_{k=1}^{L_n} \sum_{l=1}^{L_n} w_m^{(n)}(r,k,l) U_r^{(n-1)}(i+k, j+l) + b_m^{(n)}$$

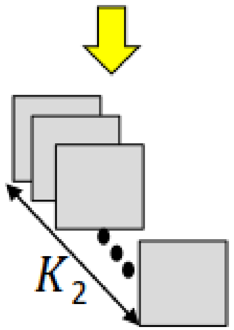
$$Y_m^{(n)}(i,j) = f\left(z_m^{(n)}(i,j)\right)$$

- **Second convolutional layer:** K_2 3-D filters resulting in K_2 2-D maps

Convolutional Neural Networks



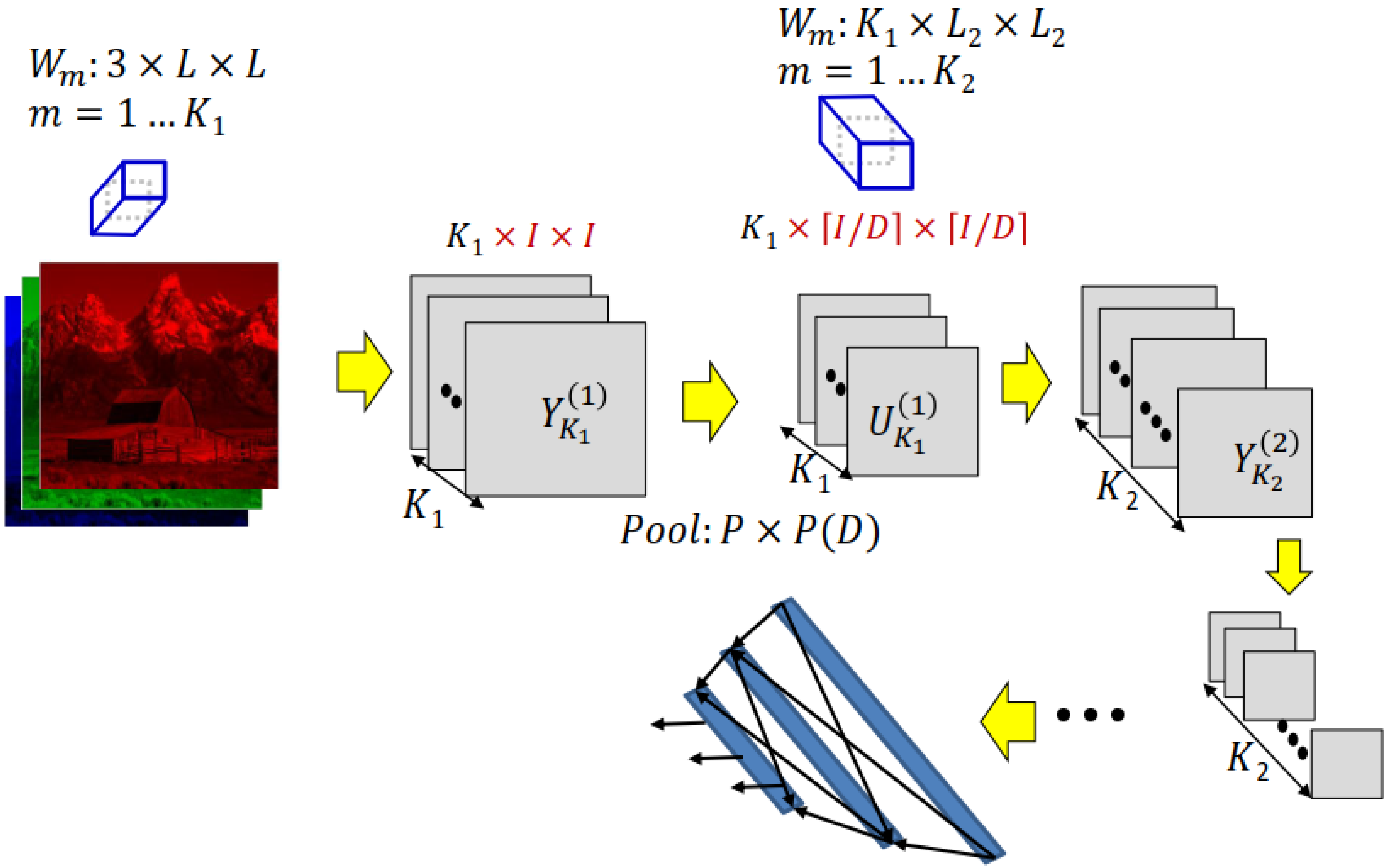
$$P_m^{(n)}(i,j) = \underset{\substack{k \in \{(i-1)d+1,\, id\}, \\ l \in \{(j-1)d+1,\, jd\}}}{\operatorname{argmax}} Y_m^{(n)}(k,l)$$



$$U_m^{(n)}(i,j) = Y_m^{(n)}(P_m^{(n)}(i,j))$$

- **Second convolutional layer:** K_2 3-D filters resulting in K_2 2-D maps
- **Second pooling layer**: K_2 Pooling operations: outcome K_2 reduced 2D maps

Convolutional Neural Networks



- This continues for several layers until the final convolved output is fed to an MLP

Parameters to choose (design choices)

- Number of convolutional and downsampling layers
  - And arrangement (order in which they follow one another)

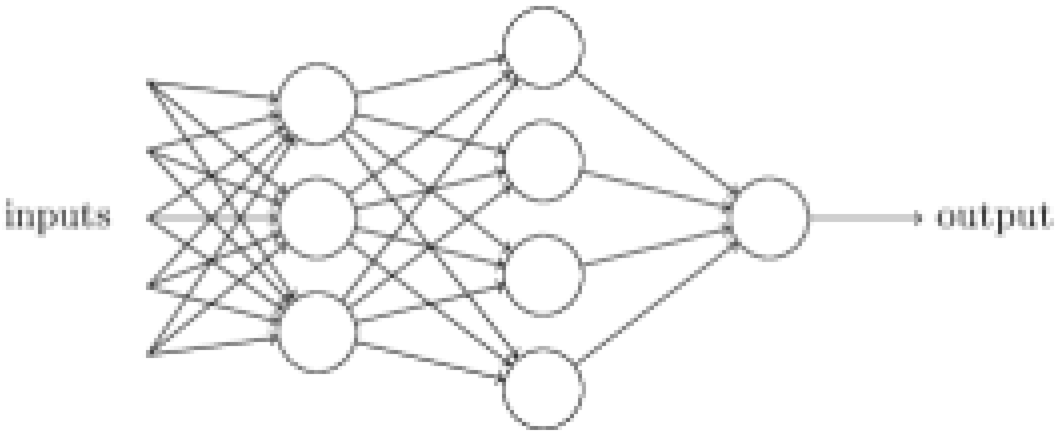
- For each convolution layer:
  - Number of filters K_i
  - Spatial extent of filter $L_i \times L_i$
    - The “depth” of the filter is fixed by the number of filters in the previous layer K_{i-1}
  - The stride S_i

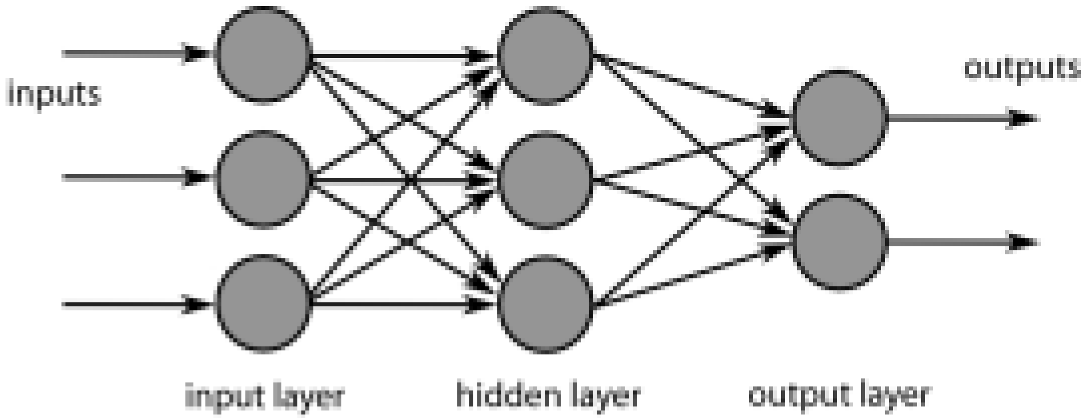
- For each downsampling/pooling layer:
  - Spatial extent of filter $P_i \times P_i$
  - The stride D_i

- For the final MLP:
  - Number of layers, and number of neurons in each layer

Binary classification

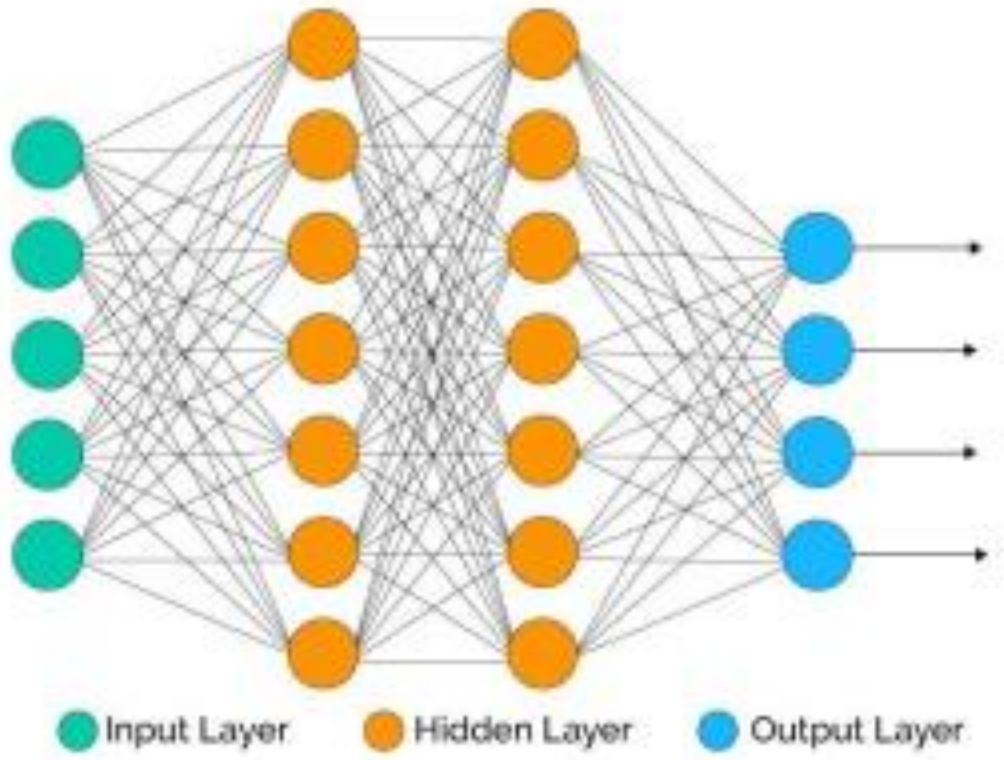
- Target class present or not?
  - Single output
  - Two outputs





Multi-class

- One prediction for each class



Softmax activation

scores = unnormalized log probabilities of the classes.

$$\boxed{P(Y=k|X=x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}} \quad \text{where} \quad \boxed{s = f(x_i; W)}$$

cat	**3.2**
car	5.1
frog	-1.7

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Softmax activation

scores = unnormalized log probabilities of the classes.

$$\boxed{P(Y=k|X=x_i)=\frac{e^{s_k}}{\sum_j e^{s_j}}} \quad \text{where} \quad \boxed{s=f(x_i;W)}$$

cat	**3.2**
car	5.1
frog	-1.7

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Softmax activation

scores = unnormalized log probabilities of the classes.

$$\boxed{P(Y=k|X=x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}} \quad \text{where} \quad \boxed{s = f(x_i; W)}$$

cat	**3.2**		**24.5**
car	5.1	$\xrightarrow{\text{exp}}$	164.0
frog	-1.7		0.18

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Softmax activation

scores = unnormalized log probabilities of the classes.

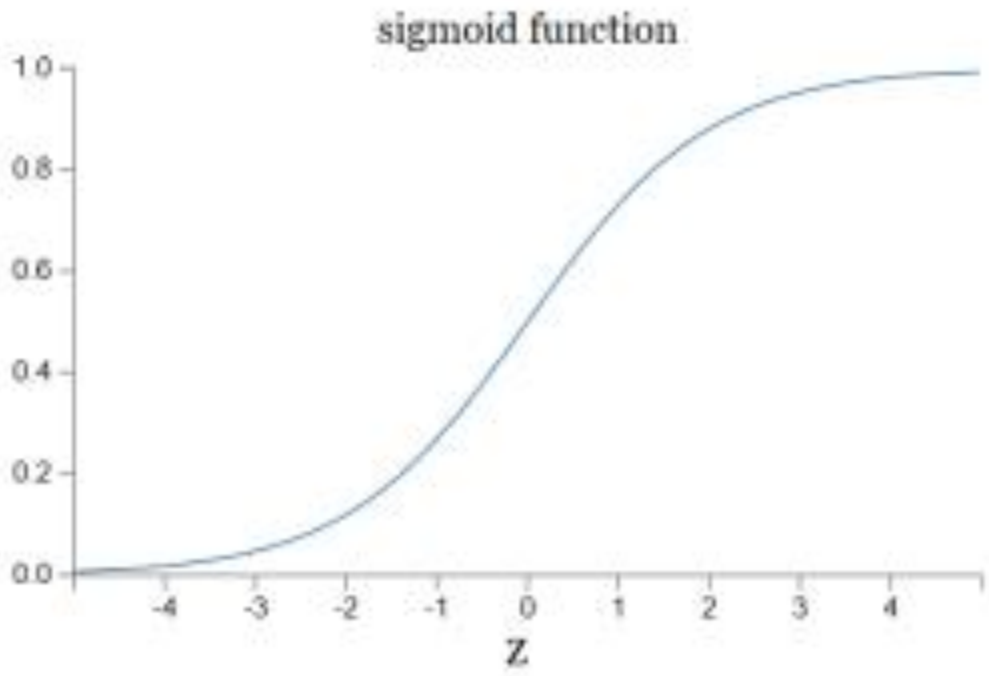
$$\boxed{P(Y=k|X=x_i)=\frac{e^{s_k}}{\sum_j e^{s_j}}} \quad \text{where} \quad \boxed{s=f(x_i;W)}$$

		exp		normalize	
cat	**3.2**		**24.5**		**0.13**
car	5.1	→	164.0	→	0.87
frog	-1.7		0.18		0.00
					probabilities

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Multi-label

- Multiple classes can be active
- Softmax will not work
- Use sigmoid activation

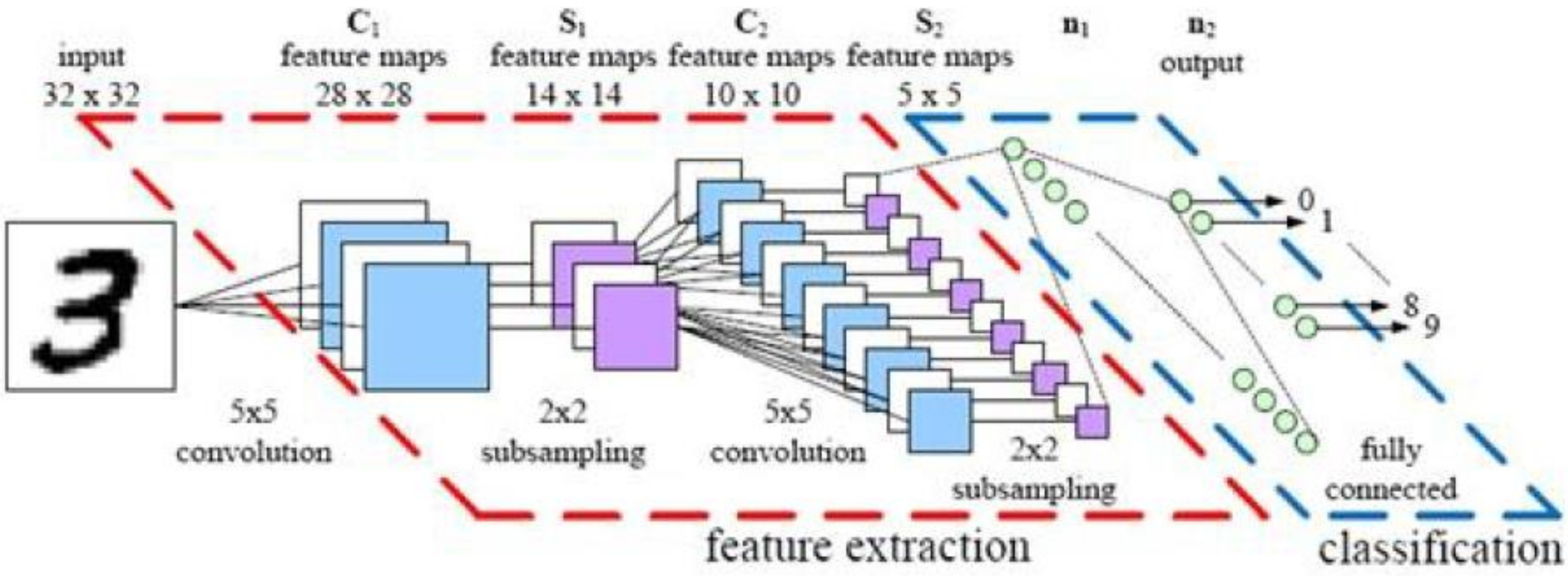


Why not correlation neural network?

- It could be
  - Deep learning libraries actually implement correlation

- Correlation relates to convolution via a 180deg rotation
  - When we *learn* kernels, we could easily learn them flipped

Digit classification



Learning phases

Training

Images

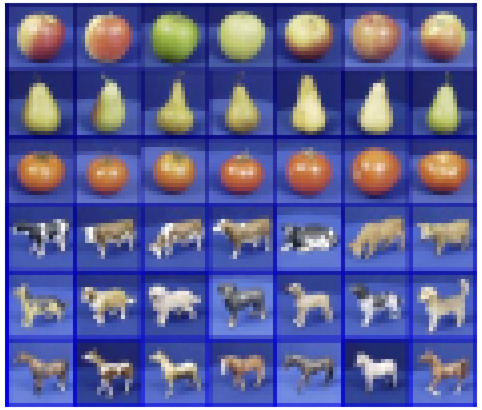


Image Features → Training → Trained classifier

Labels → Training

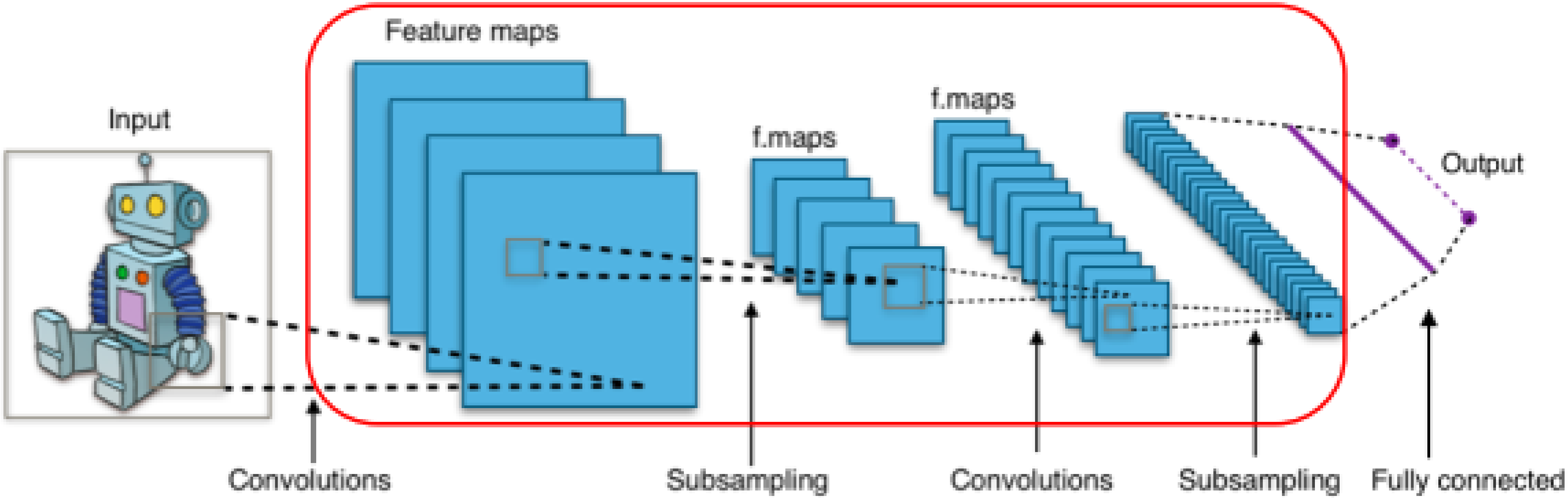
Testing

Image *not in training set*

Image Features → Apply classifier → Prediction

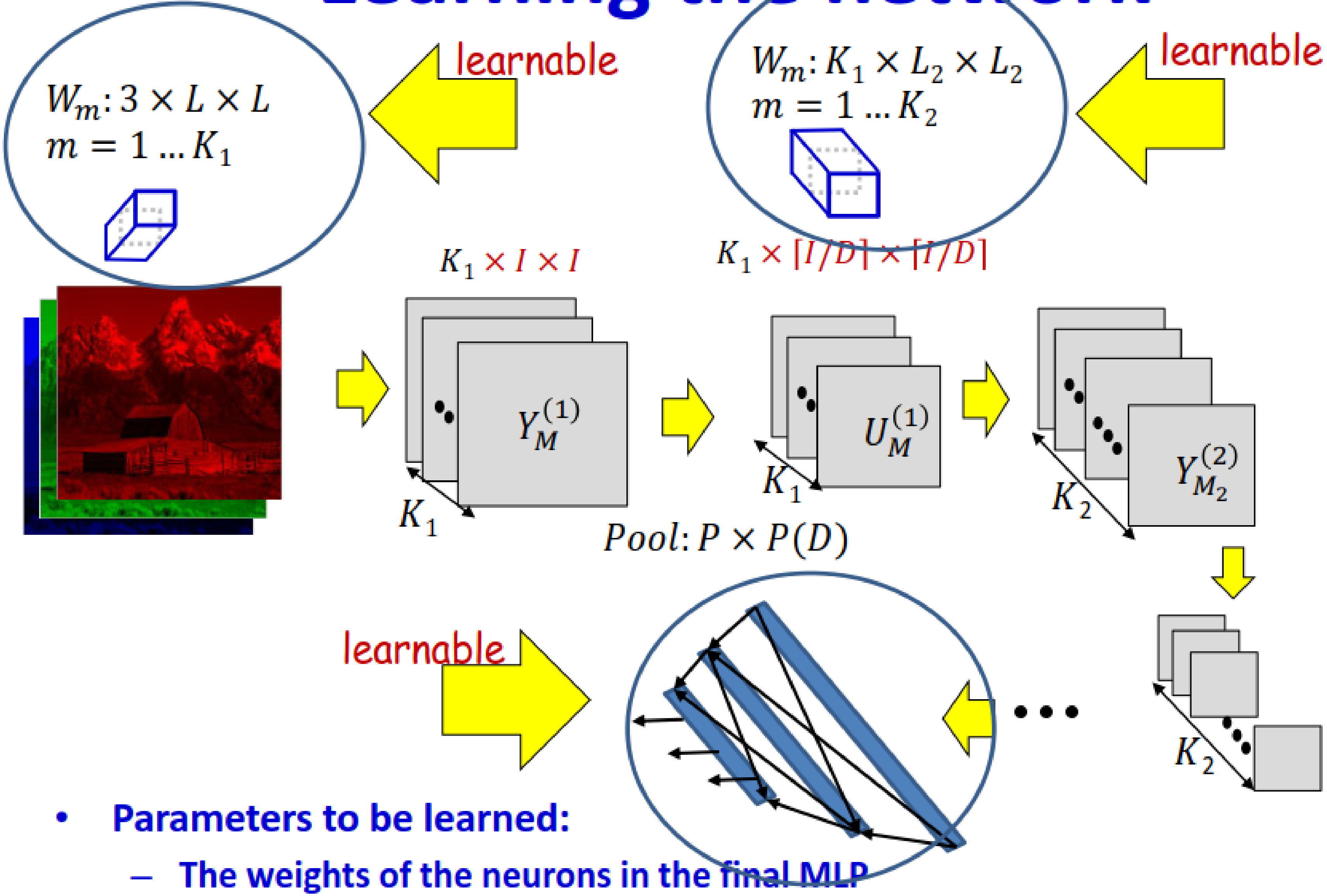
Slide credit: D. Hoiem and L. Lazebnik

General CNN architecture



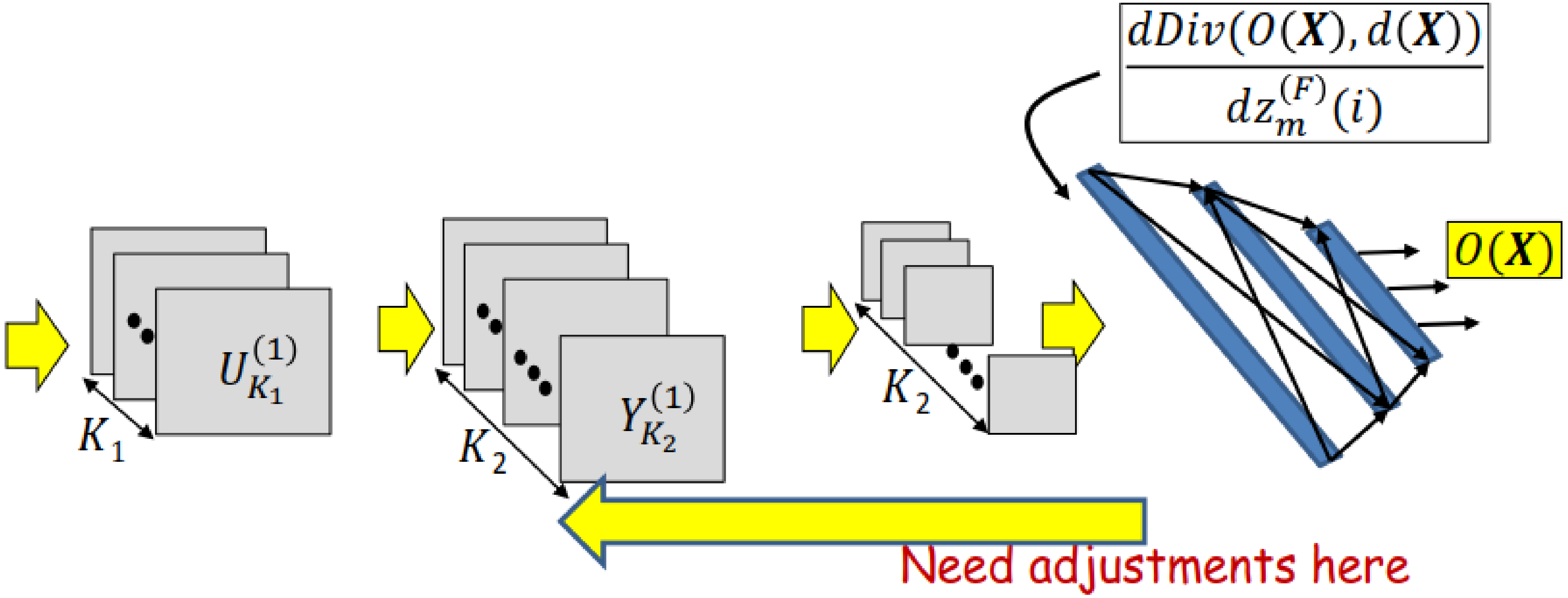
End to end learning!

Learning the network



- **Parameters to be learned:**
  - The weights of the neurons in the final MLP
  - The (weights and biases of the) filters for every *convolutional* layer

Backpropagation: Final flat layers



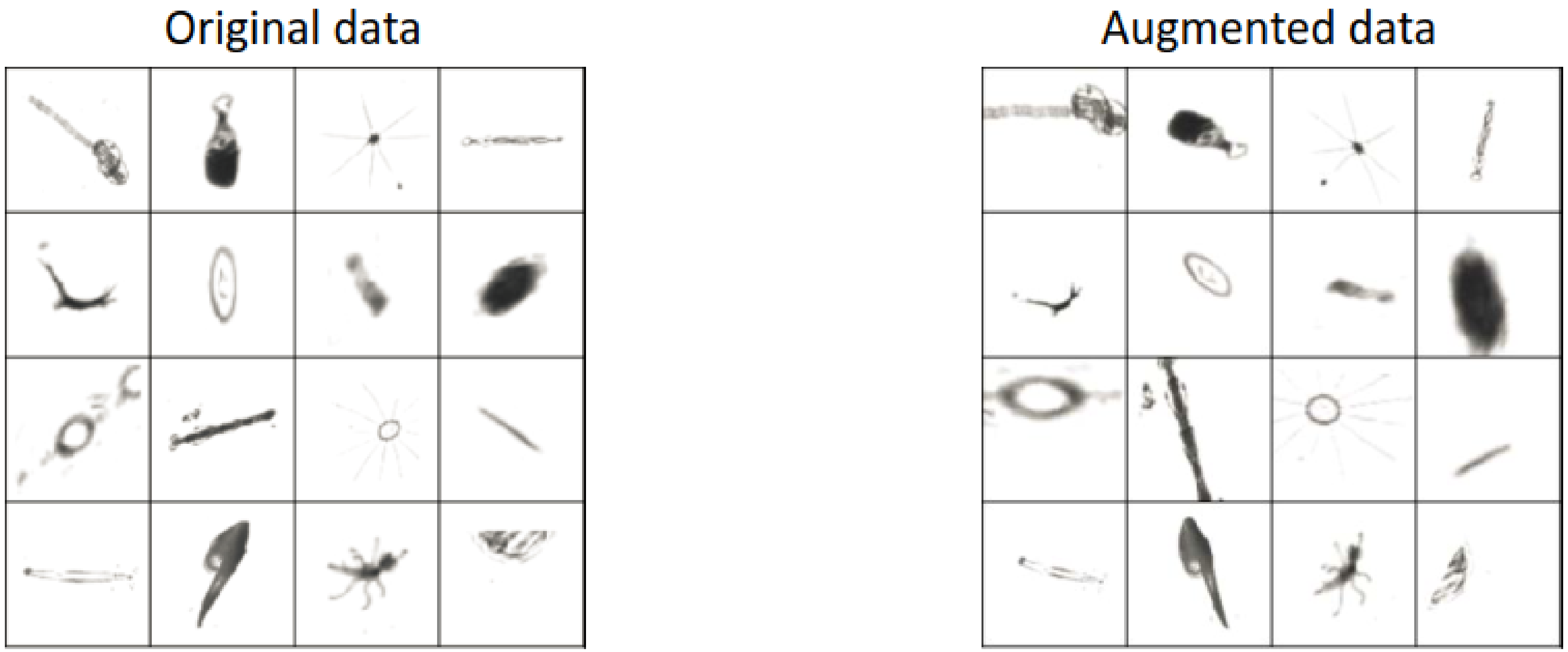
- Backpropagation from the flat MLP requires special consideration of
  - The pooling layers (particularly Maxout)
  - The shared computation in the convolution layers

Training Issues

- Standard convergence issues
  - Solution: RMS prop or other momentum-style algorithms
  - Other tricks such as batch normalization

- The number of parameters can quickly become very large
- Insufficient training data to train well
  - Solution: Data augmentation

Data Augmentation

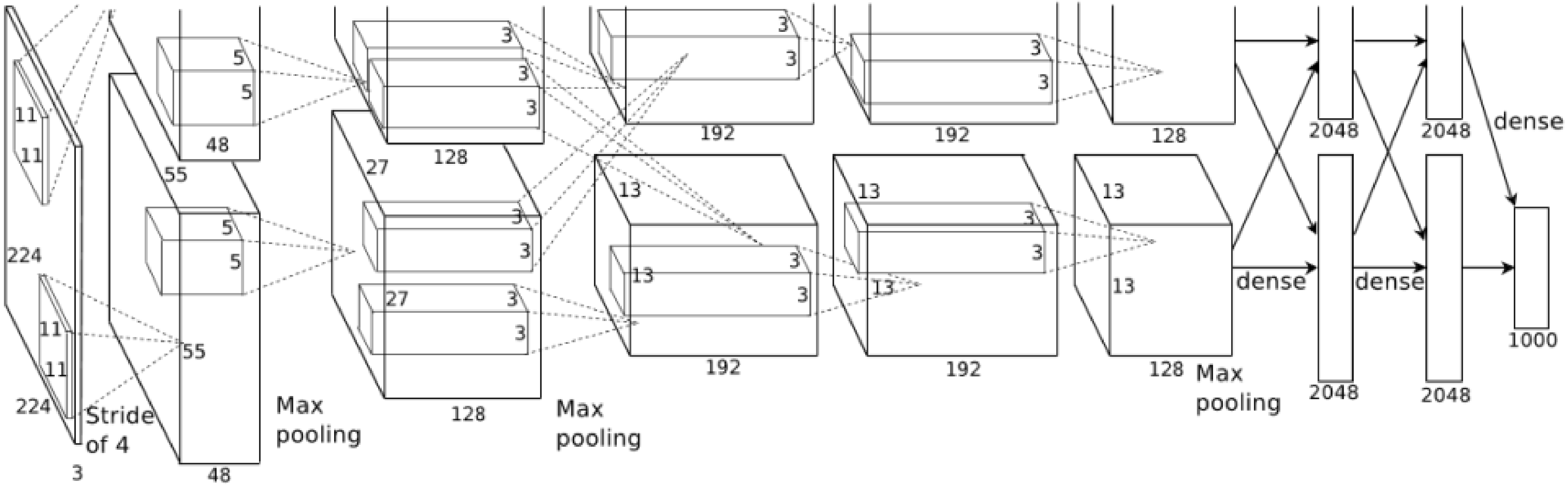


- rotation: uniformly chosen random angle between 0° and 360°
- translation: random translation between -10 and 10 pixels
- rescaling: random scaling with scale factor between 1/1.6 and 1.6 (log-uniform)
- flipping: yes or no (bernoulli)
- shearing: random shearing with angle between -20° and 20°
- stretching: random stretching with stretch factor between 1/1.3 and 1.3 (log-uniform)

https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

AlexNet

- 1.2 million high-resolution images from ImageNet LSVRC-2010 contest
- 1000 different classes (softmax layer)
- NN configuration
  - NN contains 60 million parameters and 650,000 neurons,
  - 5 convolutional layers, some of which are followed by max-pooling layers
  - 3 fully-connected layers

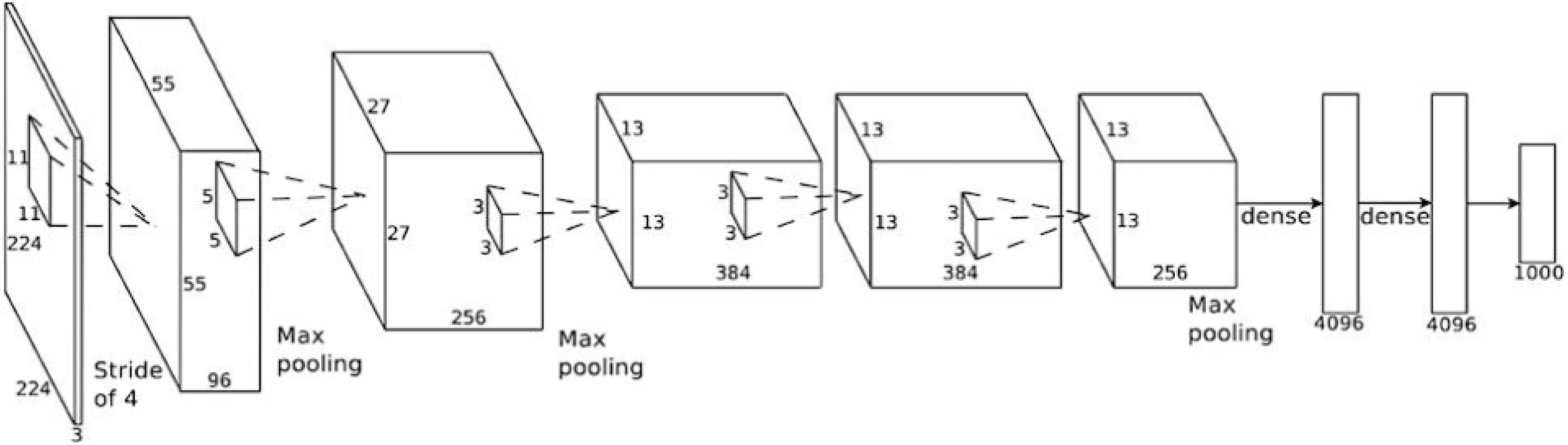


Krizhevsky, A., Sutskever, I. and Hinton, G. E. “ImageNet Classification with Deep Convolutional Neural Networks” NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

Krizhevsky et. al.

- Input: 227x227x3 images
- Conv1: 96 11x11 filters, stride 4, no zeropad
- Pool1: 3x3 filters, stride 2
- “Normalization” layer [Unnecessary]
- Conv2: 256 5x5 filters, stride 2, zero pad
- Pool2: 3x3, stride 2
- Normalization layer [Unnecessary]
- Conv3: 384 3x3, stride 1, zeropad
- Conv4: 384 3x3, stride 1, zeropad
- Conv5: 256 3x3, stride 1, zeropad
- Pool3: 3x3, stride 2
- FC: 3 layers,
  - 4096 neurons, 4096 neurons, 1000 output neurons

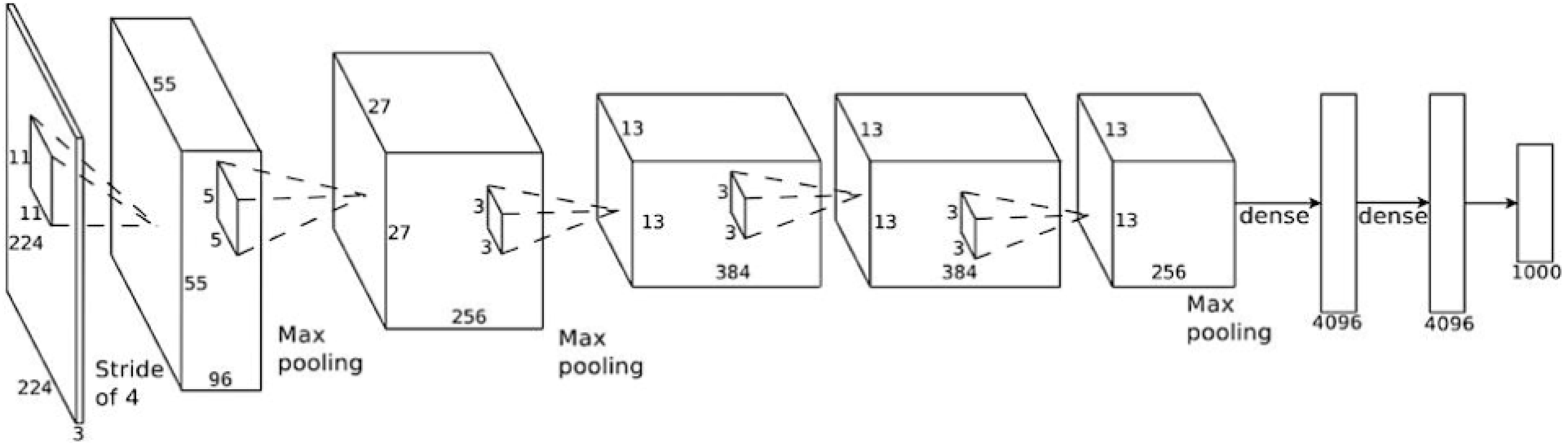
AlexNet : Network Size



CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
MAX POOL3
FC6
FC7
FC8

- Input: 227x227x3 images
- First layer (CONV1): 96 11x11 filters applied at stride 4
- What is the output volume size? (227-11)/4+1 = 55
- What is the number of parameters? 11x11x3x96 = 35K

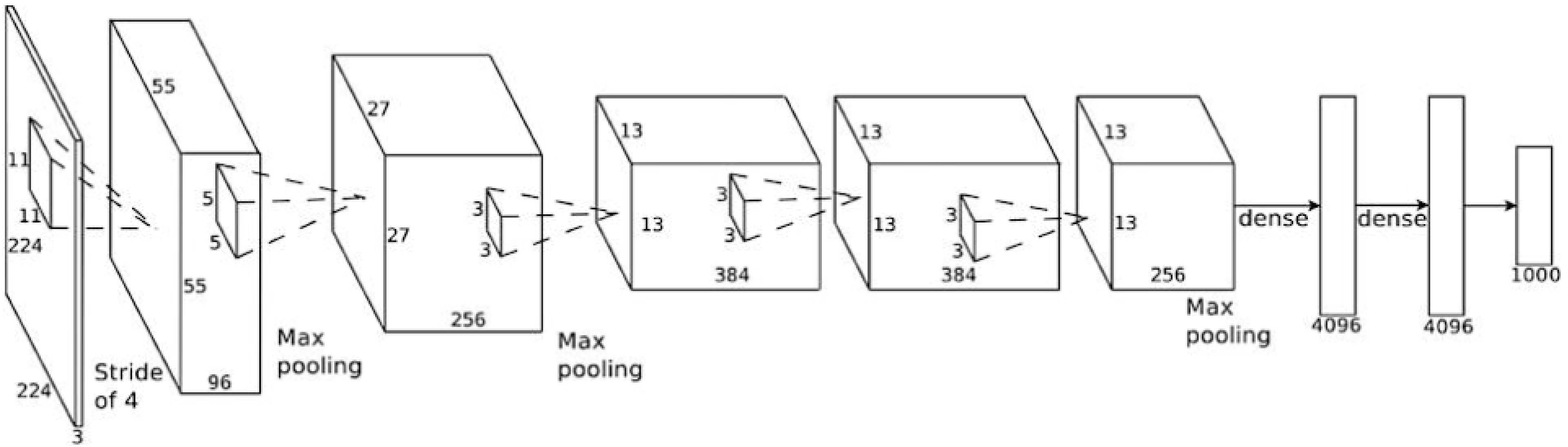
AlexNet : Network Size



CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
MAX POOL3
FC6
FC7
FC8

- After CONV1: 55x55x96
- Second layer (POOL1): 3x3 filters applied at stride 2
- What is the output volume size? (55-3)/2+1 = 27
- What is the number of parameters in this layer? 0

AlexNet : Network Size



CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
MAX POOL3
FC6
FC7
FC8

- After POOL1: 27x27x96
- Third layer (NORM1): Normalization
- What is the output volume size? 27x27x96

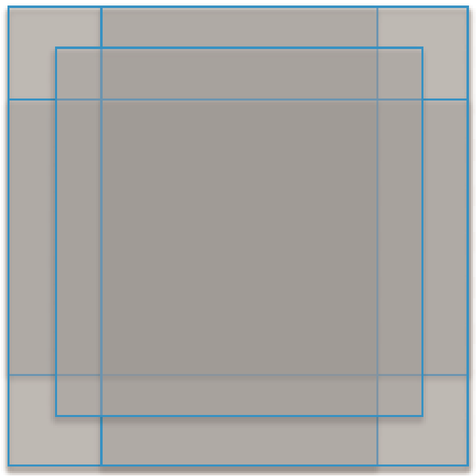
AlexNet : Network Size

1. [227x227x3] INPUT
2. [55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
3. [27x27x96] MAX POOL1: 3x3 filters at stride 2
4. [27x27x96] NORM1: Normalization layer
5. [27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
6. [13x13x256] MAX POOL2: 3x3 filters at stride 2
7. [13x13x256] NORM2: Normalization layer
8. [13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
9. [13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
10. [13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
11. [6x6x256] MAX POOL3: 3x3 filters at stride 2
12. [4096] FC6: 4096 neurons
13. [4096] FC7: 4096 neurons
14. [1000] FC8: 1000 neurons (class scores)

CONV1	35K
MAX POOL1	
NORM1	
CONV2	307K
MAX POOL2	
NORM2	
CONV3	884K
CONV4	1.3M
CONV5	442K
MAX POOL3	
FC6	37M
FC7	16M
FC8	4M

Alexnet: Total parameters

- 650K neurons
- 60M parameters
- 630M connections



- Testing: Multi-crop
  - Classify different shifts of the image and vote over the lot!

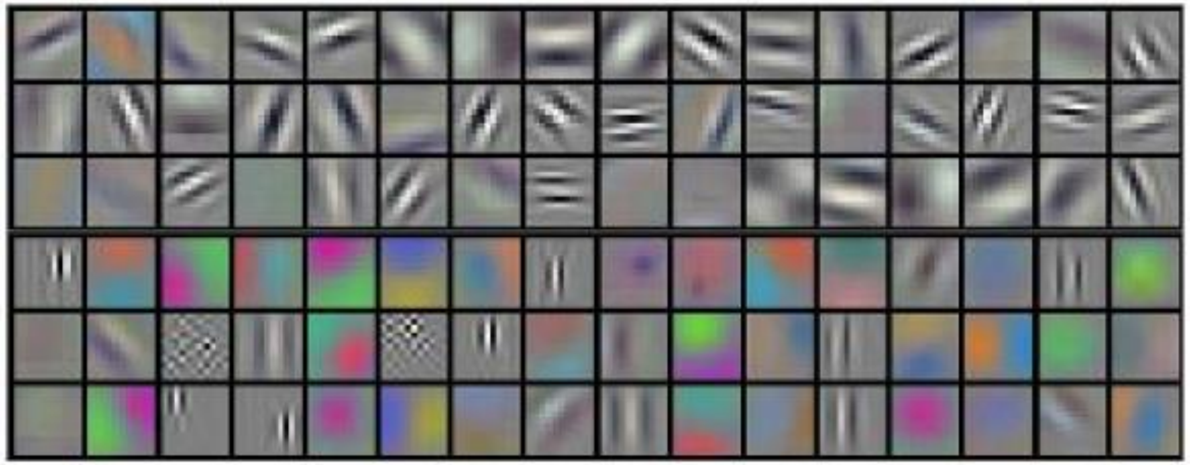
Learning magic in Alexnet

- **Activations were RELU**
  - Made a large difference in convergence
- "Dropout" – 0.5 (in FC layers only)
- *Large amount of data augmentation*
- SGD with mini batch size 128
- Momentum, with momentum factor 0.9
- L2 weight decay 5e-4
- Learning rate: 0.01, decreased by 10 every time validation accuracy plateaus
- Evaluated using: Validation accuracy

- **Final top-5 error: 18.2% with a single net, 15.4% using an ensemble of 7 networks**
  - **Lowest prior error using conventional classifiers: > 25%**

ImageNet

Figure 3: 96 convolutional kernels of size 11×11×3 learned by the first convolutional layer on the 224×224×3 input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.



Krizhevsky, A., Sutskever, I. and Hinton, G. E. “ImageNet Classification with Deep Convolutional Neural Networks” NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

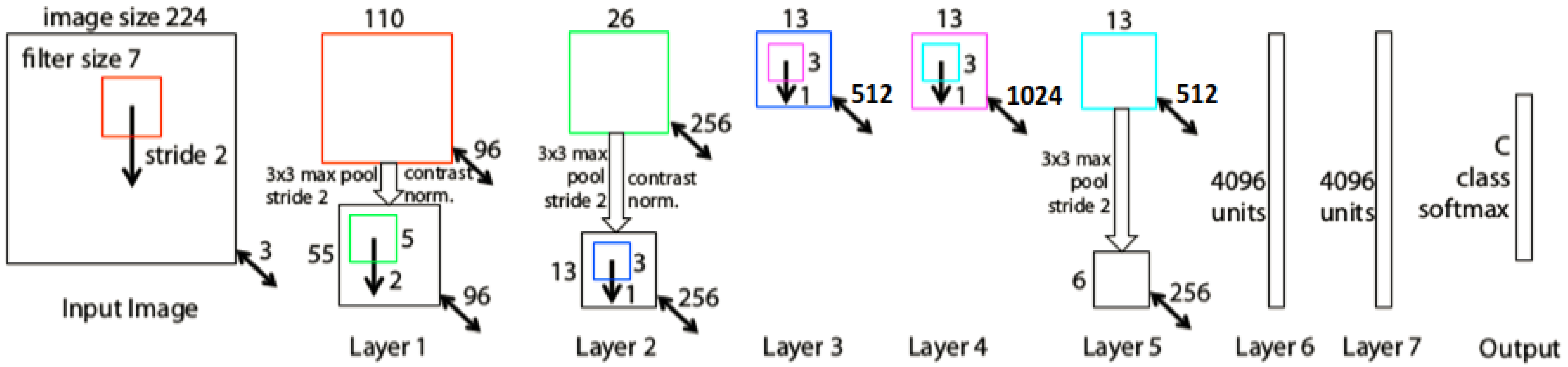
The net actually *learns* features!

Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5).

Five ILSVRC-2010 test images in the first column. The remaining columns show the six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. “ImageNet Classification with Deep Convolutional Neural Networks” NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

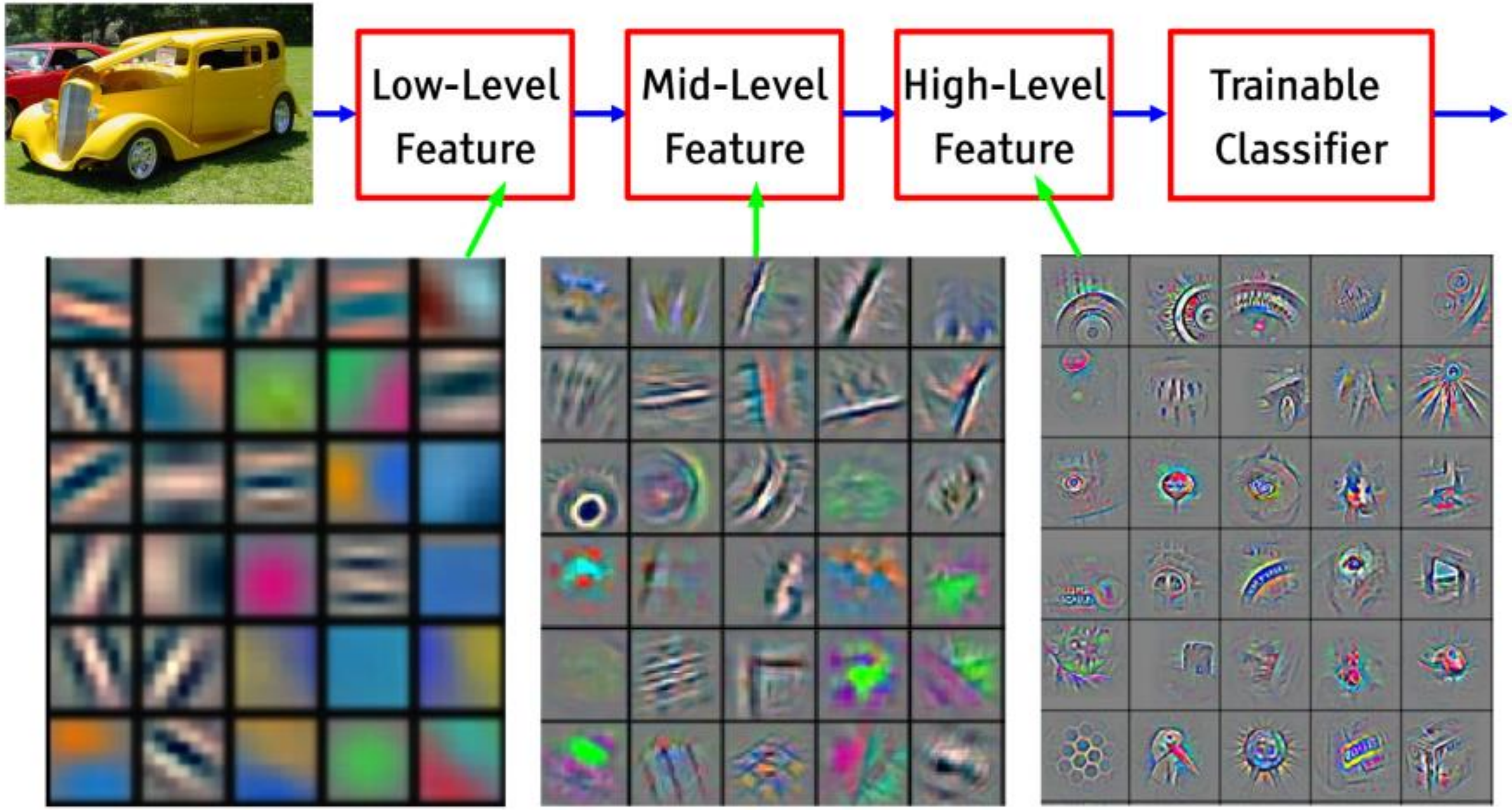
ZFNet



ZF Net Architecture

- Zeiler and Fergus 2013
- Same as Alexnet except:
  - 7x7 input-layer filters with stride 2
  - 3 conv layers are 512, 1024, 512
  - Error went down from 15.4% → 14.8%
    - Combining multiple models as before

Visualizing Convolution



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

VGGNet

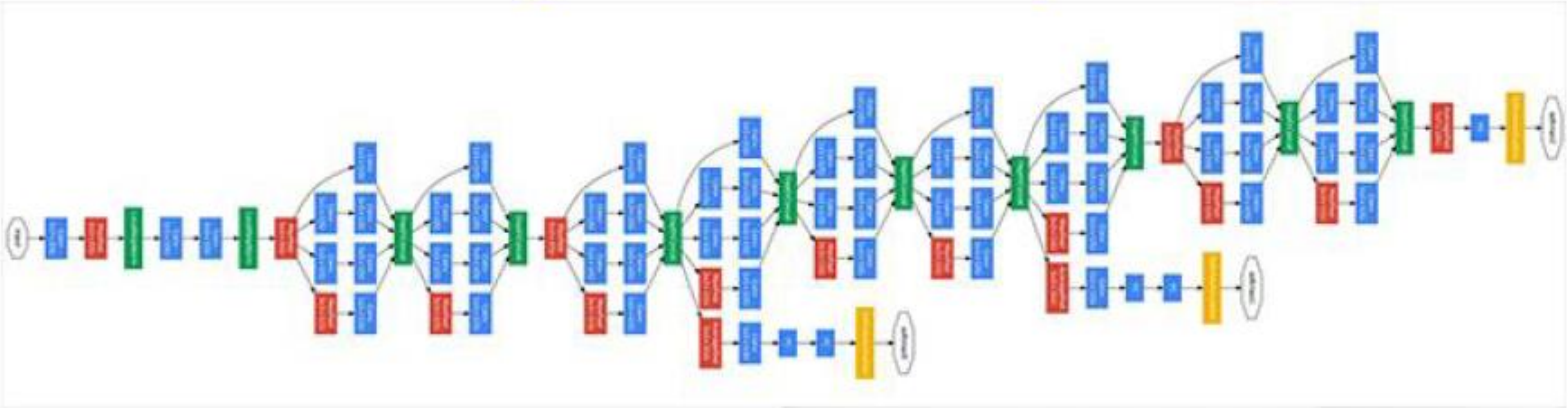
- Simonyan and Zisserman, 2014
- *Only* used 3x3 filters, stride 1, pad 1
- *Only* used 2x2 pooling filters, stride 2

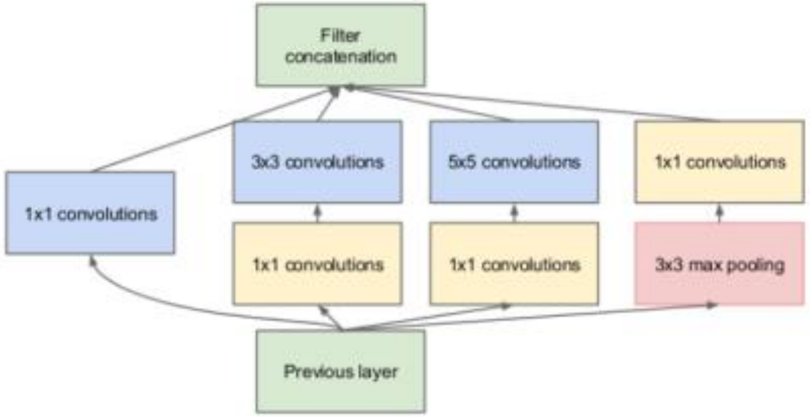
- Tried a large number of architectures.
- Finally obtained 7.3% top-5 error using 13 conv layers and 3 FC layers
  - Combining 7 classifiers
  - Subsequent to paper, reduced error to 6.8% using only two classifiers
- Final arch: 64 conv, 64 conv,
  64 pool,
  128 conv, 128 conv,
  128 pool,
  256 conv, 256 conv, 256 conv,
  256 pool,
  512 conv, 512 conv, 512 conv,
  512 pool,
  512 conv, 512 conv, 512 conv,
  512 pool,
  FC with 4096, 4096, 1000
- ~140 million parameters in all!

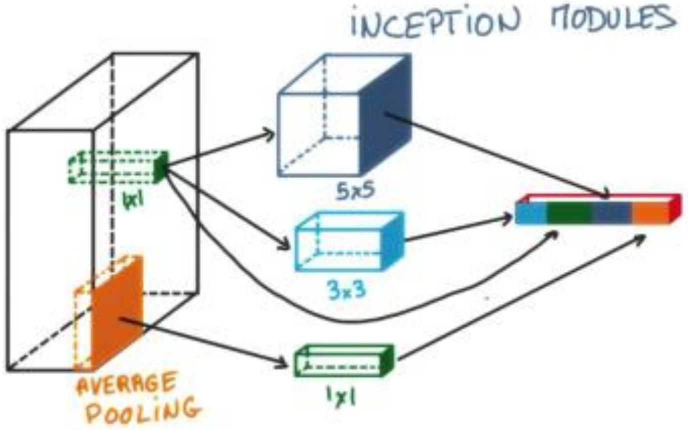
Madness!

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Googlenet: Inception







- Multiple filter sizes simultaneously
- Details irrelevant; error → 6.7%
  - Using only 5 million parameters, thanks to average pooling

Imagenet

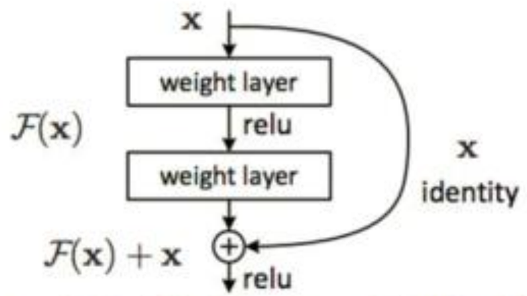
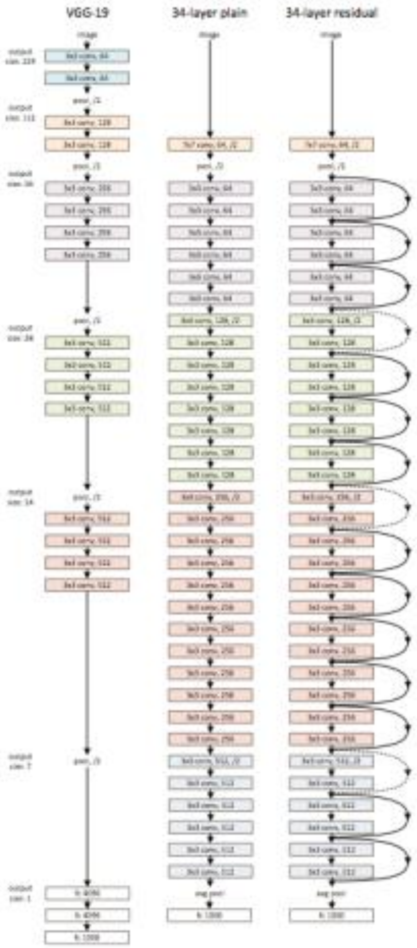


Figure 2. Residual learning: a building block.



- Resnet: 2015
  - Current top-5 error: < 3.5%
  - Over 150 layers, with “skip” connections..

Resnet details for the curious..

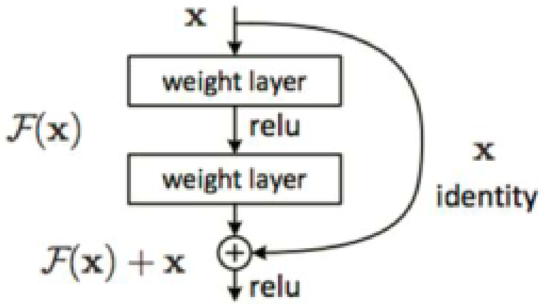


Figure 2. Residual learning: a building block.

- Last layer before addition must have the same number of filters as the input to the module
- Batch normalization after each convolution
- SGD + momentum (0.9)
- Learning rate 0.1, divide by 10 (batch norm lets you use larger learning rate)
- Mini batch 256
- Weight decay 1e-5
- ***No pooling in Resnet***

Questions?