# Algorithms

Convolutional Neural Network (CNN)

Back-Propagation

# Convolution Neural Network

- CNN more likely act like the way in which the mammals perceive the world around them.
- Extract features from input image.
- Neural Network with convolution layers
  - Share parameters
  - Local connectivity
  - Pooling unit

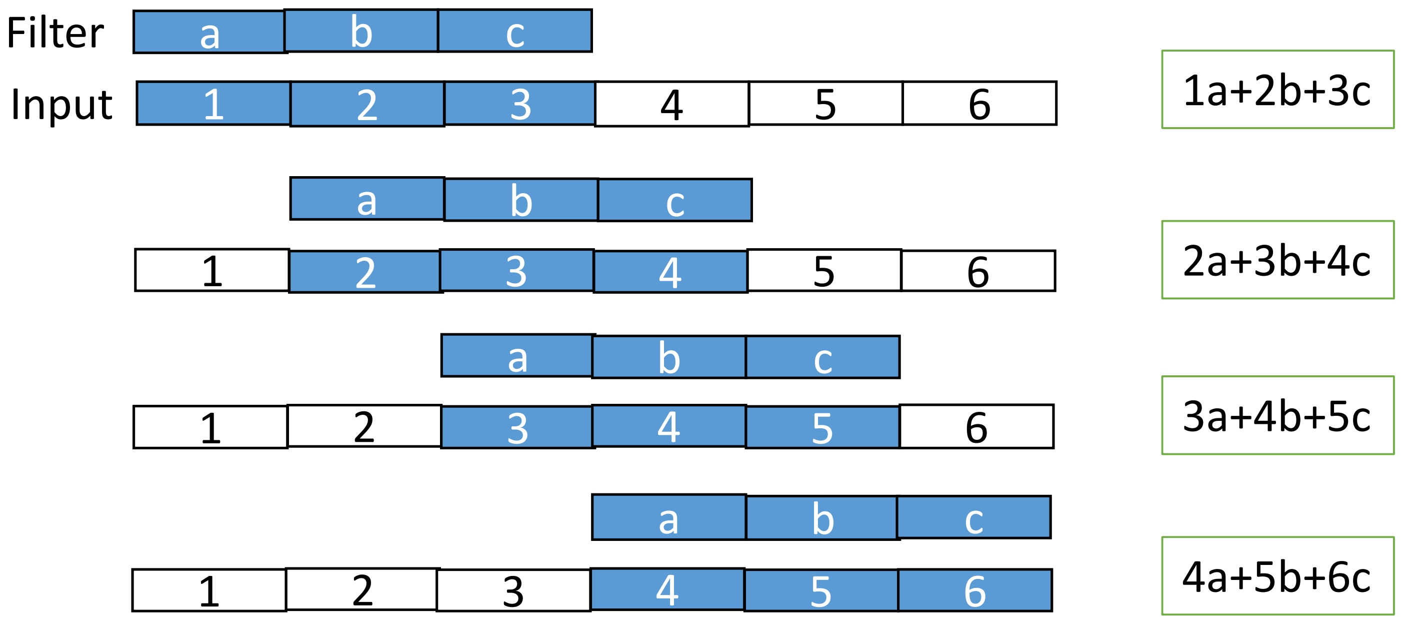
  Convolution
- Deep convolution with more than one convolution layer

# Steps for CNN

- Convolution
  - Input (x) and filter (w) is discrete or continuous
  - s = (x*w)+b                    s →feature map
  - Convolution w=[x, y, z]
  - Different implementations in convolution use cross-correlation w'=[z, y, x]
    - Same as convolution if filter is symmetric

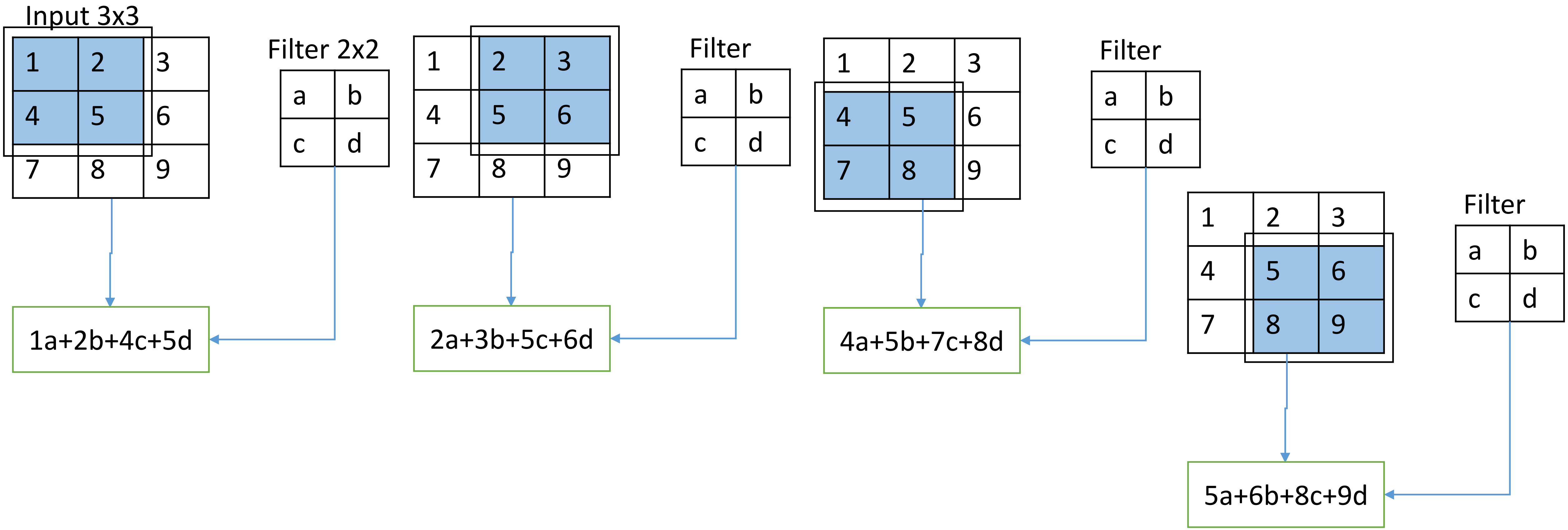
# Steps for CNN

- Convolution 1-D
  - It’s a cross product of two matrixes input and filter:



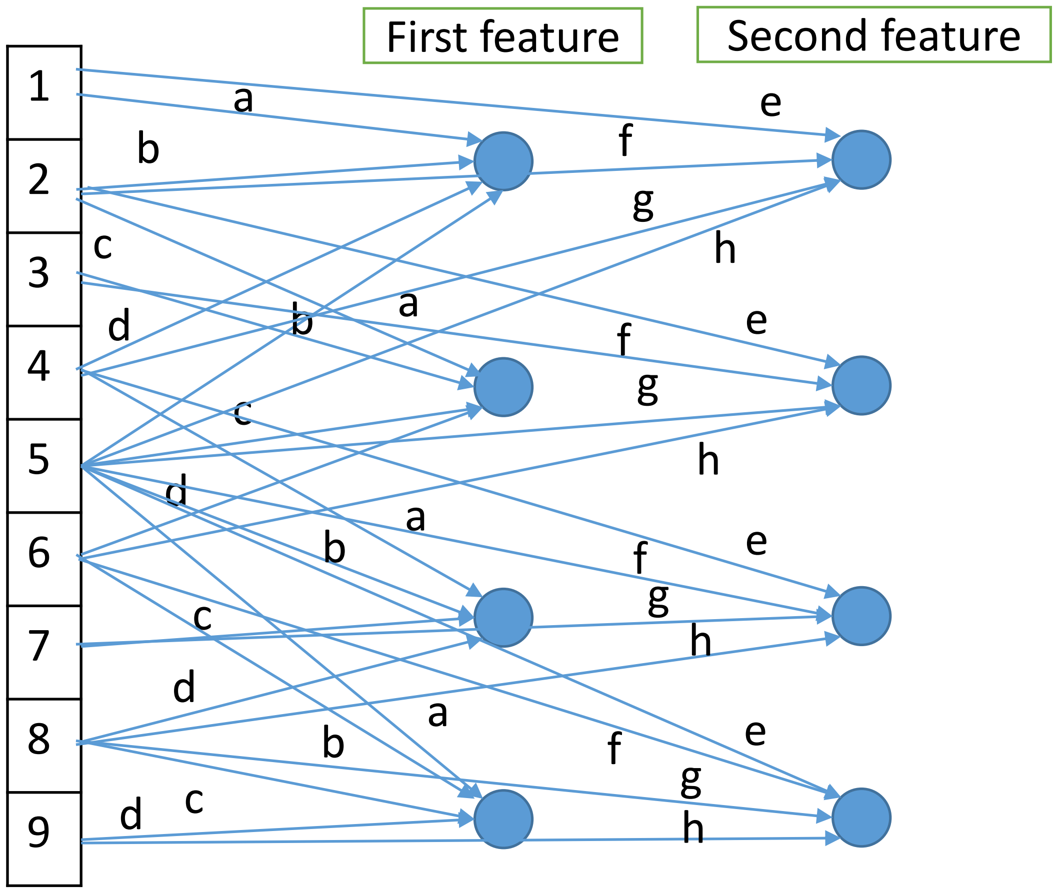
# Cont...

- Convolution 2-D



# Cont…

- In network level the input is connected to neurons like




Use different filter with different parameters to compute multiple feature maps which high light different aspects of the input image.

Example:

Input image: $10^3$ x $10^3$

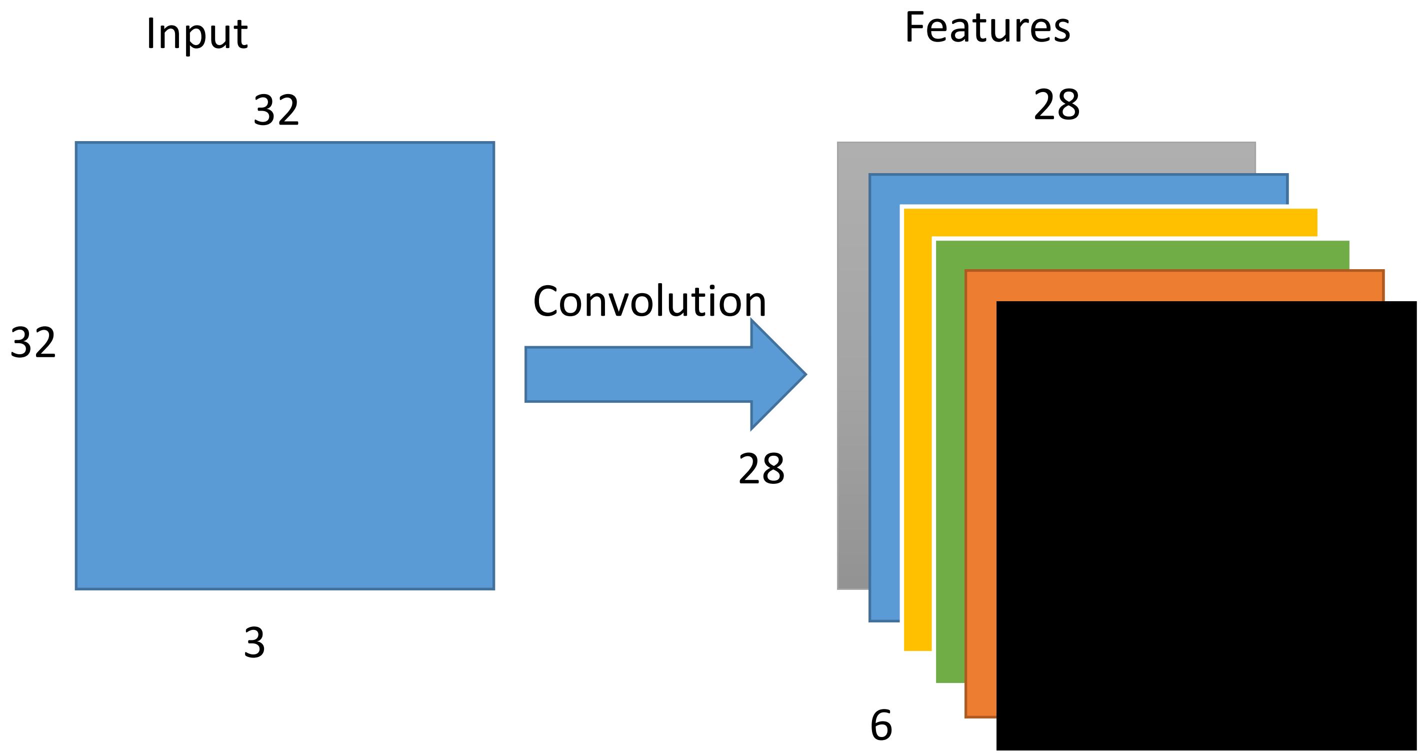
Filter size: 10 x 10

No. of filters: 100

Total number of parameters: $10^4$

# Cont…

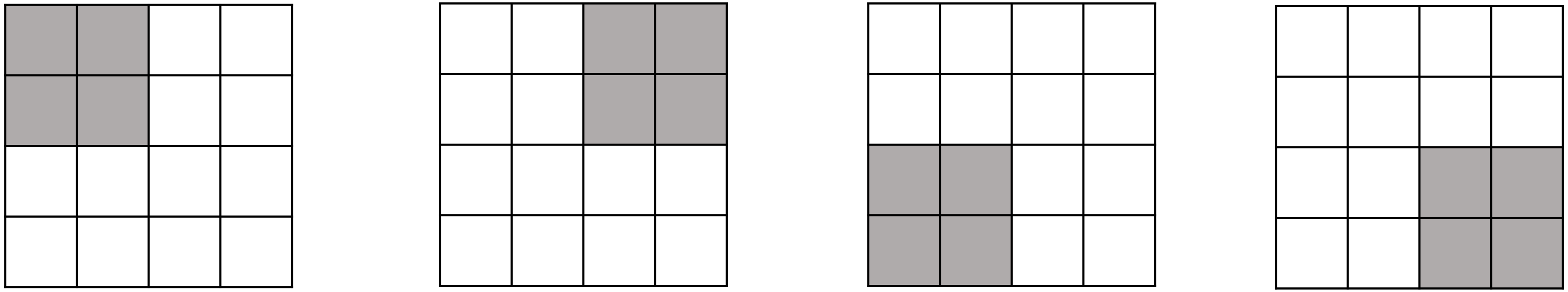
- Feature maps are depend on the number of filters:




Convolution Layer:
Input size = H1 X W1 X D1 = 32 X 32 X 3
Filter size = F = 5
No. of filter = K = 6
Formula for calculating size of output:
H2 = H1 - F + 1 = 32 - 5 + 1 = 28
W2 = W1 – F + 1 = 32 – 5 + 1 = 28
D2 = K = 6

# Cont…

- Convolution with Stride:
  - In this filter is not apply on every location of the image. Introduce stride value to move filter example input(4x3), filter(2x2) and stride = 2



  - Output size = (image size – filter size)/stride + 1

# Cont…

- Convolution with padding:
  - Add padding with zero to the border of whole image.

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | | | | | 0 |
| 0 | | | | | 0 |
| 0 | | | | | 0 |
| 0 | | | | | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Example:
Input: 4 x 4
Filter: 2 x 2
Stride: 2
Padding = 1
Output: (image size – filter size + 2(padding)) / stride + 1
Output: 3 x 3

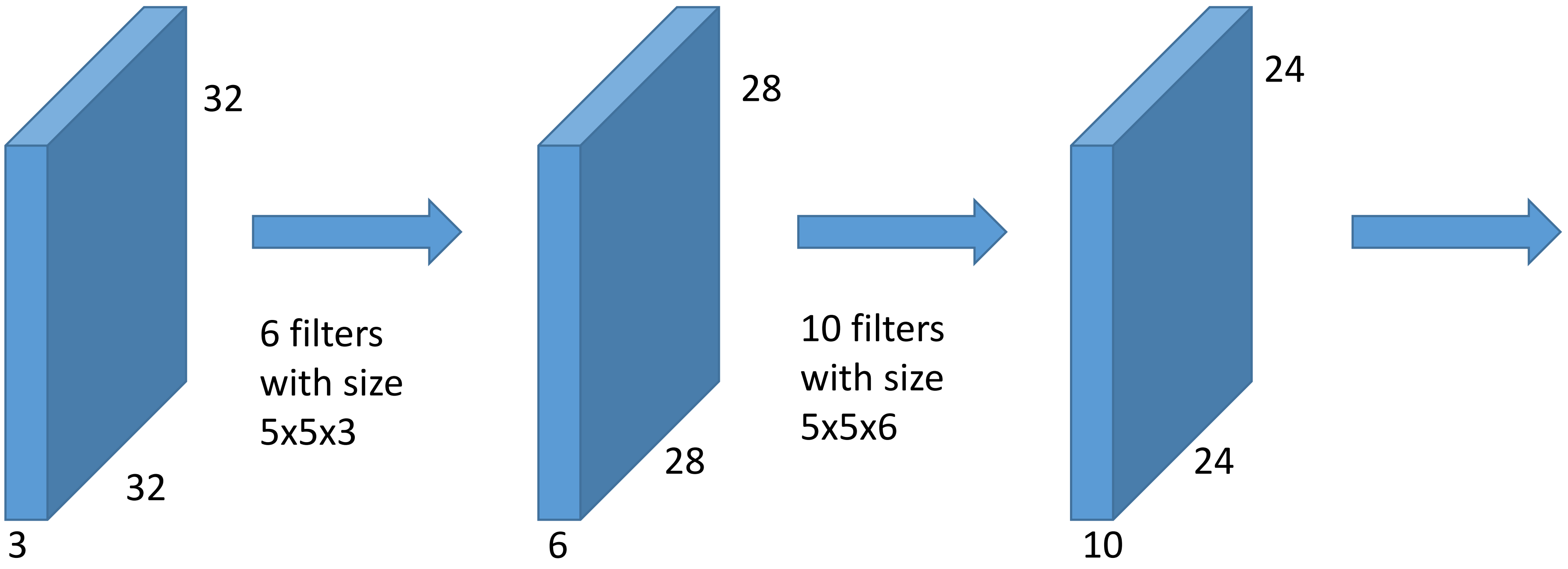
# Cont…

- There are three different convolutions:
  - Valid (no padding and output is smaller than input)
  - Same (padding with half the filter size and if stride = 1 then identical)
  - Full (padding with one less than filter size and output is larger than input)

| Valid: | Same | Full: |
|---|---|---|
| P = 0, S = 1 | P = 1, S = 1 | P = 2, S = 1 |
| Input = 4 x 4 | Input = 5 x 5 | Input = 5 x 5 |
| Filter = 3 x 3 | Filter = 3 x 3 | Filter = 3 x 3 |
| Output = 2 x 2 | Output = 5 x 5 | Output = 7 x 7 |

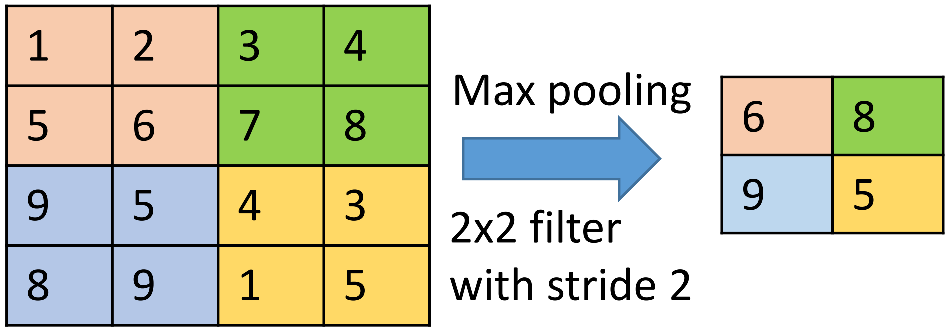
# Cont…

- There is a sequence of convolutional layers to produce more and more receptive fields



# Cont…

- Pooling
  - It makes feature maps smaller or reduce parameters of the receptive field.
  - It operate over each feature map.
  - Slightly translate the input to check if the feature if exist don’t know where.
  - Types of pooling:
    - Max pooling (choose max value)
    - Average pooling (choose average value)



# Cont…

- Relu
  - In this part of convolution we perform Relu function.
  - It perform on the small patches of the input image instead of whole input image.
  - Relu is the most common normalization technique which remove all negative values from the image after convolution layer.
  - After normalization perform pooling on the layer.

Feature Map

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| -8 | -9 | 5 | 8 |
| 6 | -5 | 5 | 2 |
| 7 | 4 | -1 | 5 |

Normalized Feature Map

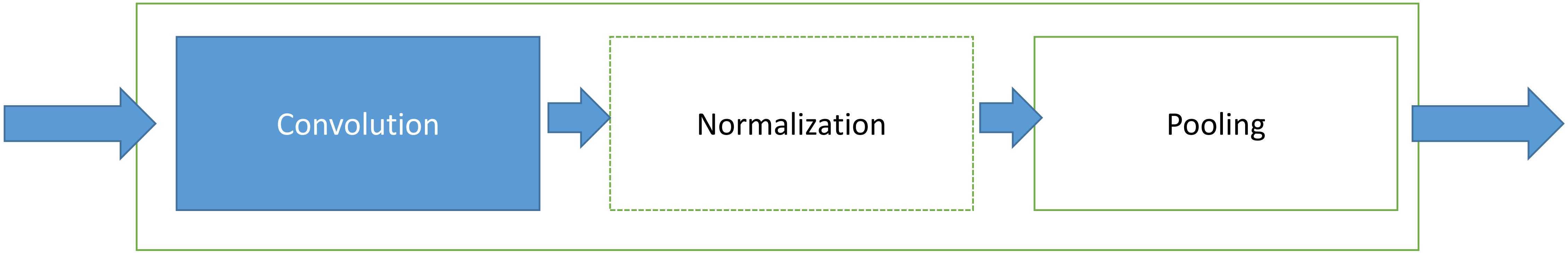
| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 0 | 0 | 5 | 8 |
| 6 | 0 | 5 | 2 |
| 7 | 4 | 0 | 5 |

Receptive Field

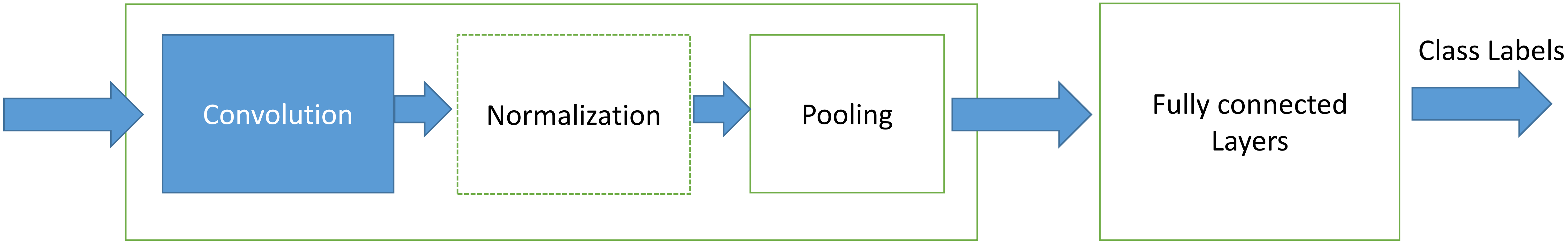
| | |
|---|---|
| 2 | 8 |
| 7 | 5 |

# Cont…

- This whole process is called a convolution block.

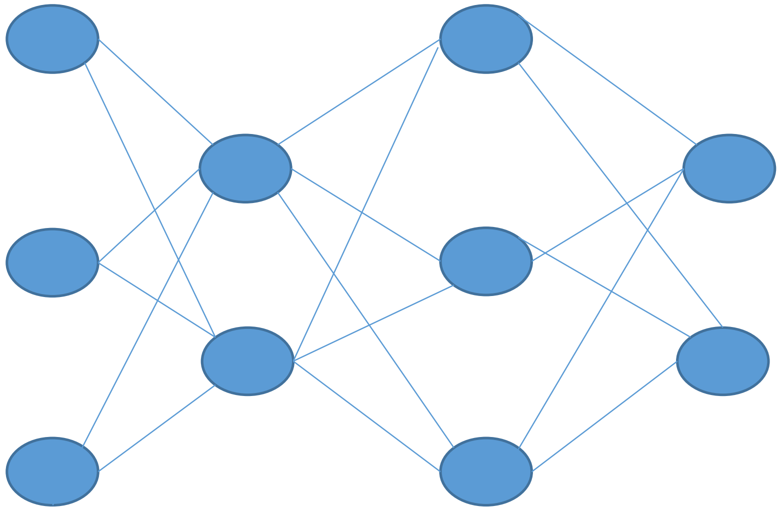



- Multiple convolution bocks are used to create Deep convolution network.
- The final convolution block output is fully connected to the neurons.



# Cont…

- Dropout (Regularization)
  - At the fully connected neuron network regularization technique Dropout.
  - It force the learn network to learn independent features.
  - To perform it simple disable random neurons from the network by setting their value to 0.



# Cont…

- Calculate probability
  - Output layer calculate probability of each class label depend on the input feature map similarity.
  - Choose high probability value label at the end as an output result.
  - Match the network output with the actual/ expected output.
  - If output does not match then perform back propagation to update the filter values.
  - Filter values are shared with different neurons in convolution layers.
  - Calculate each neuron gradient decent and then sum the values to update the filter values.

# Cont…

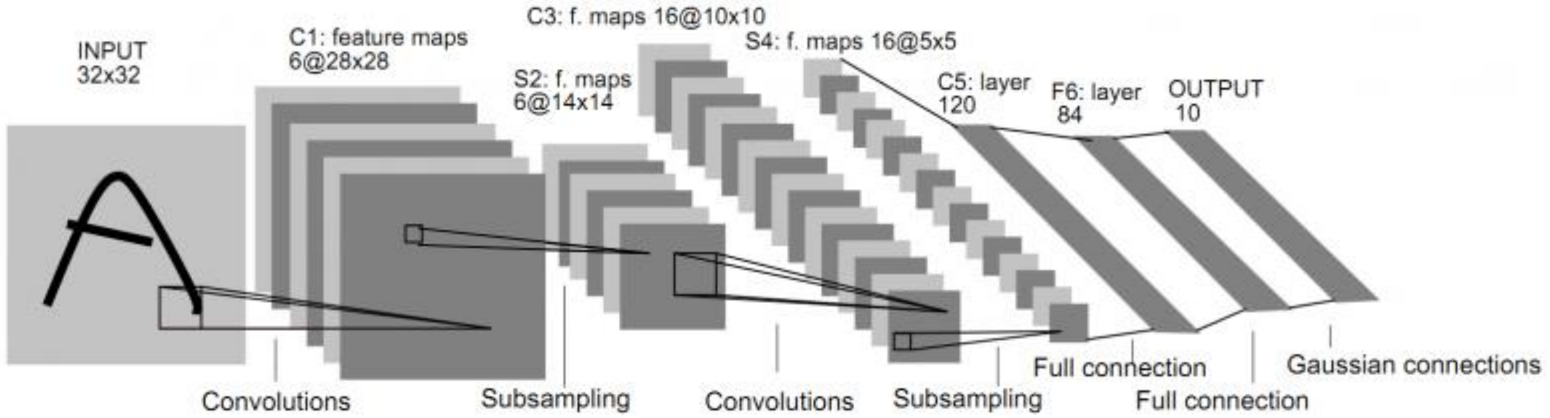
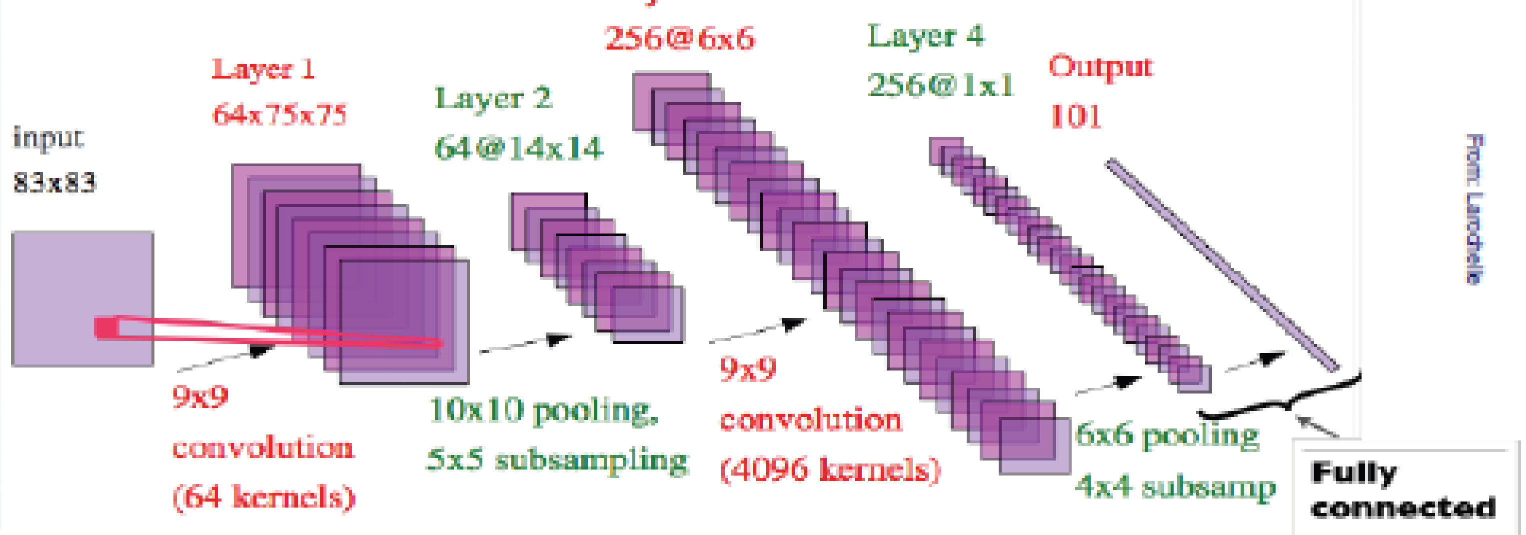



Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

(image from Yann Lecun)




From: Larochelle

# Back Propagation

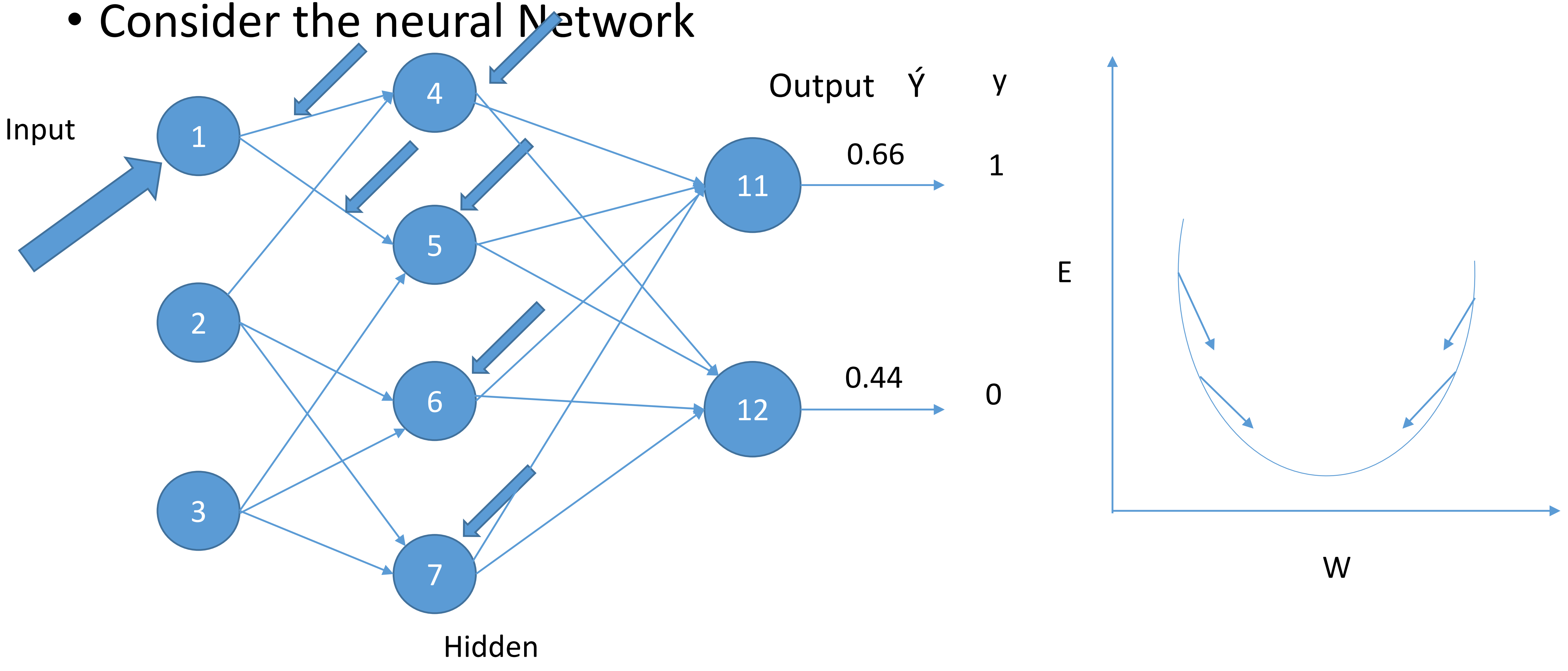
- It's an efficient method of optimization.
- Popular in NN to optimize weights.

| Inputs | | output |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 1 | 1 |

- Not a learning method of NN.
- Good computational trick used in Learning methods.

# How Back-propagation works

- Consider the neural Network



# Cont.

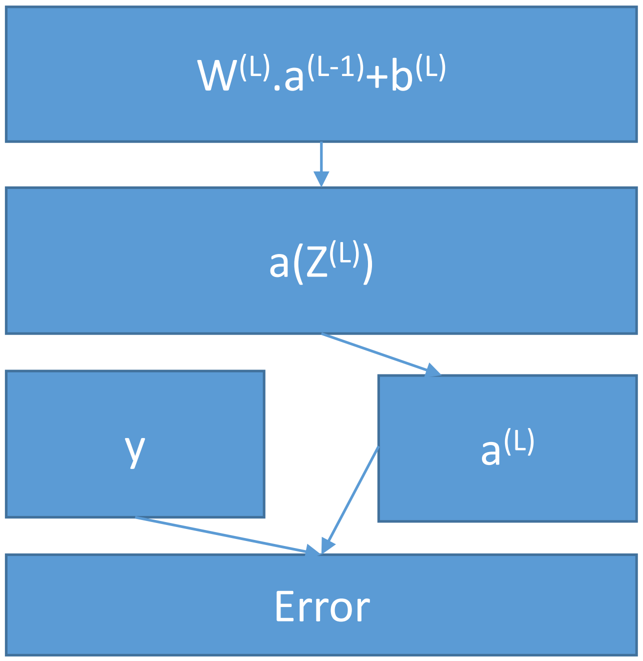
- The process of Forward propagation:

Post Synaptic Potential

Activation Function

Neuron Output
&
Actual Output

Error/ Loss

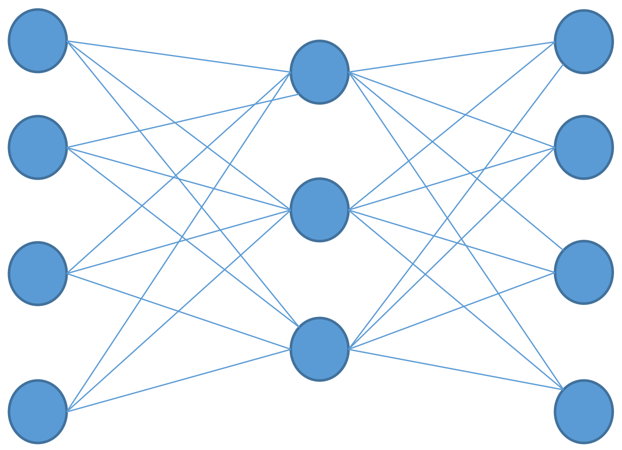


# Cont.

- First we calculate error or loss of the output layer:
  - $\Delta^{L}{}_{j} = \frac{\partial L}{\partial z^{L}{}_{j}} \frac{\partial z^{L}{}_{j}}{\partial a^{L}{}_{j}}$
- Then we calculate the errors of the hidden layers neurons:
  - $\Delta^{l}{}_{k} = \sum^{n}_{j=1} \frac{\partial L}{\partial z^{l+1}{}_{j}} \frac{\partial z^{l+1}{}_{j}}{\partial a^{l}{}_{k}} \frac{\partial a^{l}{}_{j}}{\partial z^{l}{}_{k}}$
- Calculate the partial derivative of loss w.r.t weights:
  - $w_{jk}$_input$_i = \frac{\partial L}{\partial w^{l}{}_{jk}} = \frac{\partial L}{\partial z^{l}{}_{j}} \frac{\partial z^{l}{}_{j}}{\partial a^{l}{}_{j}} \frac{\partial a^{l}{}_{j}}{\partial w^{l}{}_{jk}}$
  - $\Delta wjk$_final = $\sum^{N}_{i=1} \Delta wjk$_input$_i$
  - $w_{jk} = w_{jk}$ - ($\Delta wjk$_final - learning rate)

# Cont.

- Calculate the partial derivative of loss w.r.t bias:
  - $b_j$_input$_i$ = $\frac{\partial L}{\partial b^l{}_j}$ = $\frac{\partial L}{\partial z^l{}_j}\frac{\partial z^l{}_j}{\partial a^l{}_j}\frac{\partial a^l{}_j}{\partial b^l{}_j}$
  - $\Delta b_j$_final = $\sum^N_{i=1} \Delta b_j$_input$_i$
  - $b_j = b_j$ - ($\Delta b_j$_final - learning rate)



Error loss of each output layer

Back propagate the errors to previous layer

For each connection calculate bias and weight for all input values

# Practical Example

- AND Logic Gate implementation using Neural Network:
  - 2-Input AND Gate
  - Find the optimal Weight values for the network
  - Use Back Propagation to find optimal weight
  - Sigmoid Activation function

# Cont.

- Python Code:

```
#main function
def main():
        # take the input pattern as a map. In the binary form to perform AND logic gate operation in 2D environment.
        pattern = [[[0,0], [0]],
                   [[0,1], [0]],
                   [[1,0], [0]],
                   [[1,1], [1]]]

        neuralNetwork = Neural(pattern) #generate the basic structure of the network.
```

| A | B | A AND B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Cont.

```
class Neural:
    def __init__(self, pattern):
        #lets take 3 input nodes, 3 hidden nodes and 1 output node.
        self.inputNode=3 #number of input nodes
        #introduce additional constant input value 1 and weight value -threshold.
        self.hiddenNode=3 #number of hidden nodes
        self.outputNode=1 #number of output nodes
        #initialize two dimensional array for network weights. It generate weight to connect layer node to next layer node
        self.wih = [] #array of weights from input to hidden layers
        for i in range(self.inputNode):
            self.wih.append([0.0]*self.hiddenNode)
        self.who = [] #array of weights from hidden to output layer
        for j in range(self.hiddenNode):
            self.who.append([0.0]*self.outputNode)
        #create activation function matrix for each layer and initialize them with 1
        self.ai, self.ah, self.ao = [],[],[]
        self.ai=[1.0]*self.inputNode
        self.ah=[1.0]*self.hiddenNode
        self.ao=[1.0]*self.outputNode
```

```
wih= [0.0] [0.0] [0.0]
     [0.0] [0.0] [0.0]
     [0.0] [0.0] [0.0]

who= [0.0] [0.0] [0.0]
     [0.0] [0.0] [0.0]
     [0.0] [0.0] [0.0]
```

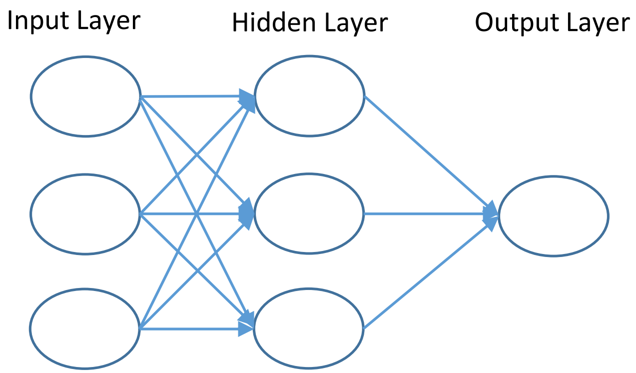
```
ai= [[1.0] [1.0] [1.0]]
ah= [[1.0] [1.0] [1.0]]
ao= [[1.0]]
```

# Cont.

```
#assign random weight values to the connection call randomizeMatrix function some bounds on values
randomizeMatrix(self.wih,-0.2,0.2) #random bound values
randomizeMatrix(self.who,-2.0,2.0)
```

```
#randomizeMatrix function definition. To generate random weight values
def randomizeMatrix ( matrix, a, b):
    for i in range ( len (matrix) ):
        for j in range ( len (matrix[0]) ):
            # For each connection in neural network assign a random weight uniformly between the bound values
            matrix[i][j] = random.uniform(a,b)
```

```
neuralNetwork = Neural(pattern) #generate the basic structure of the network.
#This line generate the simple skeleton of the network with randomly initialized
weight values
```



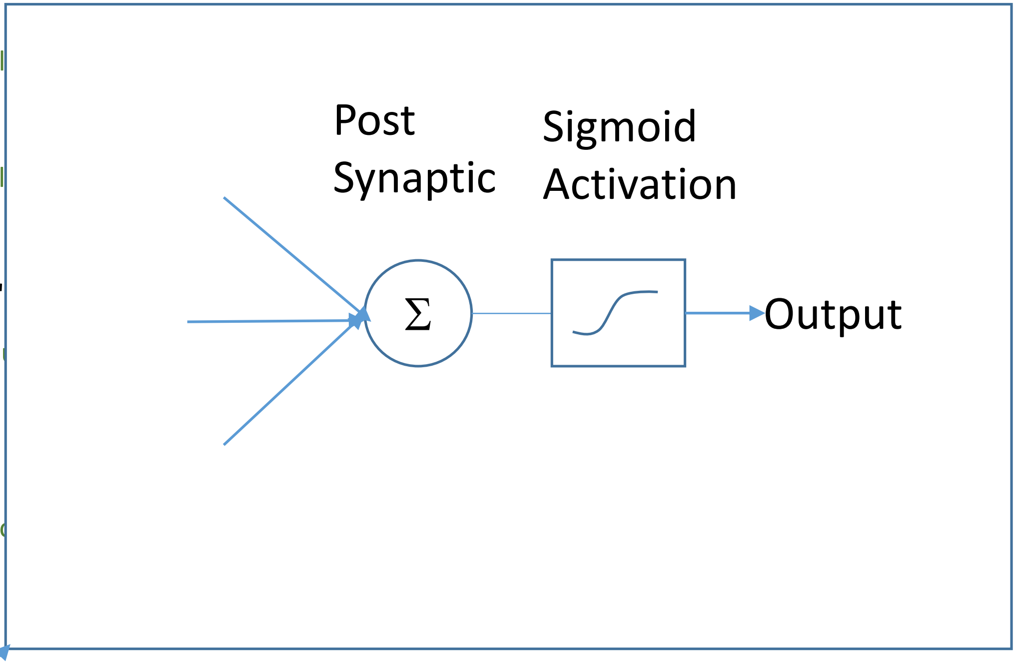
# Cont.

```
neuralNetwork.trainNetwork(pattern) #train the network on the given input pattern

#trainNetwork function definition. To train the neural network
def trainNetwork(self, pattern):
        for i in range(100):
                # Run the network for every set of input values, get the output values and Backpropagate them until satisfy the correct answers
                for p in pattern:
                        # Run the network for every tuple in p.
                        inputs = p[0] #select input values from pattern
                        output = self.runNetwork(inputs) #run network for every input pair value in pattern
                        expectedOutput = p[1] #expected output of the input pattern
                        self.backpropagate(inputs,expectedOutput,output) #call backpropagate to update the weight values
        self.test(pattern) #test the input pattern with updated weights
```

# Cont.

```
#runNetwork function definition. To run the network on specific set of input val
def runNetwork(self, values):
        #check the number of values are equal to the number of input l
        if(len(values)!=self.inputNode-1):
                print ("number of input values are not correct.'
        #activate the inputNodes with input values (inputNode-1) becau
        for i in range(self.inputNode-1):
                self.ai[i]=values[i]
        #calculate Post synaptic potential for all connected layers of eac
        for j in range(self.hiddenNode): #hidden nodes
                sum=0.0
                for i in range(self.inputNode): #input nodes
                        sum+=self.ai[i]*self.wih[i][j] #multiply the input value with their respective weight value and sum up the all values
                #call the sigmoid function for the activation function of hidden layer nodes
                self.ah[j]=sigmoid(sum)
```



# Cont.

```
for k in range(self.outputNode): #output nodes
          sum=0.0
          for l in range(self.hiddenNode): #hidden nodes
                    #multiply the activation value of hidden node with their respective weight value and sum up the all values
                    sum+=self.ah[l]*self.who[l][k]
          #call the sigmoid function for the activation function of output layer node
          self.ao[k]=sigmoid(sum)
#return the activation function value of output node
return self.ao

#sigmoid function definition. To calculate the activation functions. You can change it to other activation functions like Relu etc.
def sigmoid(x):
          return 1 / (1 + math.exp(-x))
#sigmoid function used to handle non-linear situations in neural network.
```

# Cont.

```
neuralNetwork.trainNetwork(pattern) #train the network on the given input pattern


#trainNetwork function definition. To train the neural network
def trainNetwork(self, pattern):
        for i in range(100):
                # Run the network for every set of input values, get the output values and Backpropagate them until satisfy the correct answers
                for p in pattern:
                        # Run the network for every tuple in p.
                        inputs = p[0] #select input values from pattern
                        output = self.runNetwork(inputs) #run network for every input pair value in pattern
                        expectedOutput = p[1] #expected output of the input pattern
                        self.backpropagate(inputs,expectedOutput,output) #call backpropagate to update the weight values
        self.test(pattern) #test the input pattern with updated weights
```

# Cont.

```
#backpropagate function definition it adjusts the weights according the expected output and network output to minimize the error.
def backpropagate(self, inputs, expected, output, N=0.2): #N is the learning rate
        #calculate error on output layer
        #introduce new matrix outputDeltas error for the output layer
        outputDeltas = [0.0]*self.outputNode
        for k in range(self.outputNode): #output nodes
                #error is equal to (Target value - Output value)
                error = expected[k] - output[k] #calculate error
                outputDeltas[k]=error*dsigmoid(self.ao[k]) #multiply error with differentiate sigmoid of output layer activations
        #update hidden to output layer weights
        for j in range(self.hiddenNode): #hidden nodes
                for k in range(self.outputNode): #output nodes
                        #multiply hidden layer node activation with output layer node delta error
                        deltaWeight = self.ah[j] * outputDeltas[k]
                        #multiply weight error (delta weight) and learning rate then add it into previous weight
                        self.who[j][k]+= N*deltaWeight
```

See figure:1 (Section 1)

See figure:1 (Section 2)

# Cont.

```
#calculate error on hidden layer
#introduce new matrix hiddenDeltas error for the hidden layer
hiddenDeltas = [0.0]*self.hiddenNode
for j in range(self.hiddenNode): #hidden nodes
        #error in hidden layer node is the sum of (hidden layer weights times output delta error of output node)
        error=0.0
        for k in range(self.outputNode): #output nodes
                #sum of (each hidden layer node weight times output delta error of output node)
                error+=self.who[j][k] * outputDeltas[k]
        hiddenDeltas[j]= error * dsigmoid(self.ah[j]) #multiply error with differentiate sigmoid of hidden layer activations
#update input to hidden layer weights
for i in range(self.inputNode):
        for j in range(self.hiddenNode):
                deltaWeight = hiddenDeltas[j] * self.ai[i] #multiply input layer node activation with hidden layer node delta error
                self.wih[i][j]+= N*deltaWeight #multiply weight error (delta weight) and learning rate then add it into previous weight
```

See figure:1 (Section 3)

See figure:1 (Section 4)

# Cont.

```
#dsigmoid function definition. To calculate the derivative of the sigmoid function.
def dsigmoid(y):
        return y * (1 - y)
```

```
neuralNetwork.trainNetwork(pattern) #train the network on the given input pattern

        #trainNetwork function definition. To train the neural network
        def trainNetwork(self, pattern):
                for i in range(100):
                        # Run the network for every set of input values, get the output values and Backpropagate them until satisfy the correct answers
                        for p in pattern:
                                # Run the network for every tuple in p.
                                inputs = p[0] #select input values from pattern
                                output = self.runNetwork(inputs) #run network for every input pair value in pattern
                                expectedOutput = p[1] #expected output of the input pattern
                                self.backpropagate(inputs,expectedOutput,output) #call backpropagate to update the weight values
                self.test(pattern) #test the input pattern with updated weights
```

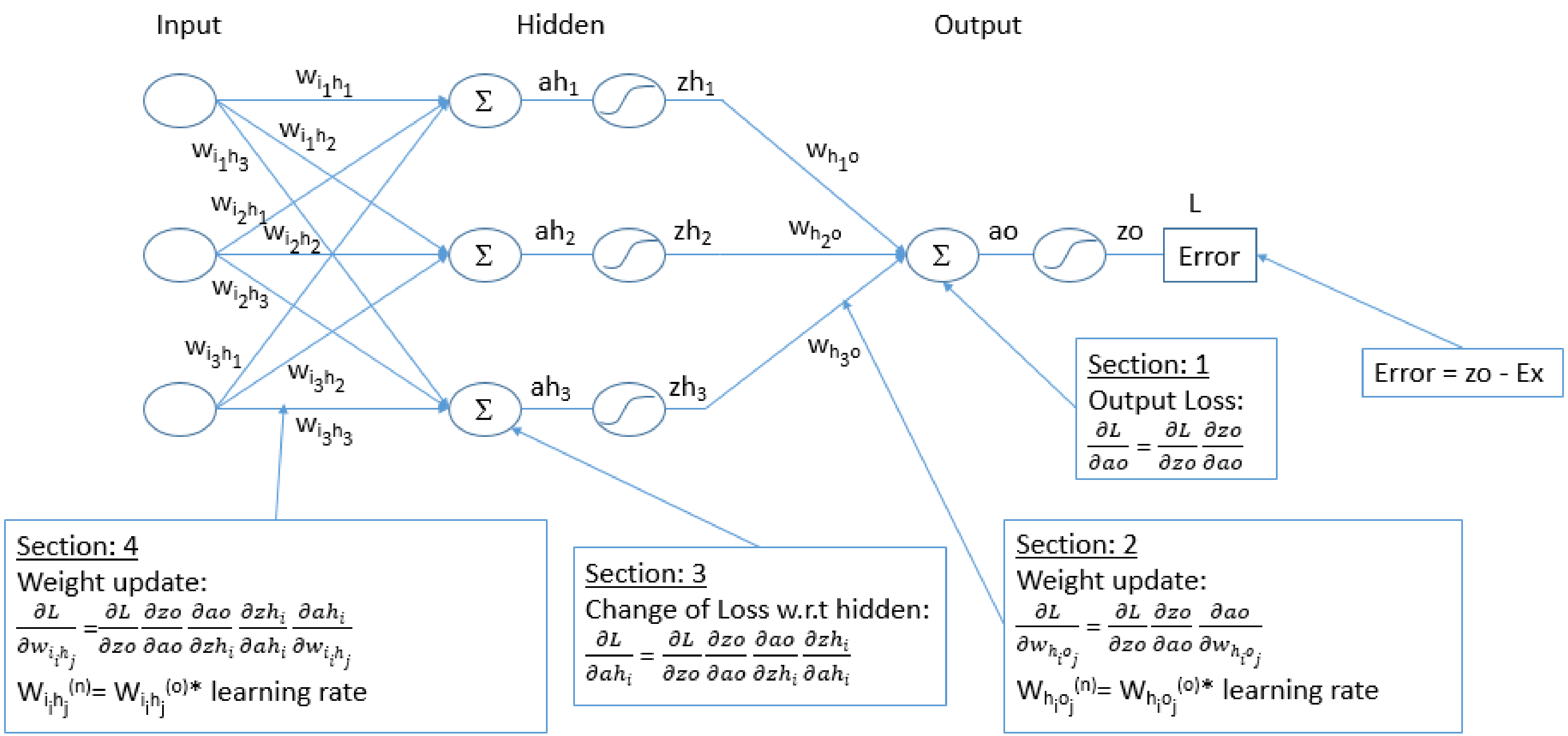
# Cont.

```
#test function definition. To test the network after the training and Backpropagation is completed
    def test(self, patterns):
        for p in patterns:
            inputs = p[0]
            print ("For input:" , p[0] , " Output -->" , self.runNetwork(inputs) , "\tTarget: " , p[1])
```

## Output:

```
For input: [0, 0]  Output --> [0.028644383620873615]    Target:  [0]
For input: [0, 1]  Output --> [0.19861387841743]        Target:  [0]
For input: [1, 0]  Output --> [0.19834250696397693]     Target:  [0]
For input: [1, 1]  Output --> [0.6662174911269693]      Target:  [1]
```

# Figure: 1



Input
Hidden
Output
$w_{i_1h_1}$
$w_{i_1h_2}$
$w_{i_1h_3}$
$w_{i_2h_1}$
$w_{i_2h_2}$
$w_{i_2h_3}$
$w_{i_3h_1}$
$w_{i_3h_2}$
$w_{i_3h_3}$
$ah_1$
$ah_2$
$ah_3$
$zh_1$
$zh_2$
$zh_3$
$w_{h_1o}$
$w_{h_2o}$
$w_{h_3o}$
ao
zo
L
Error
Error = zo - Ex
Section: 1
Output Loss:
$\frac{\partial L}{\partial ao} = \frac{\partial L}{\partial zo}\frac{\partial zo}{\partial ao}$
Section: 2
Weight update:
$\frac{\partial L}{\partial w_{h_io_j}} = \frac{\partial L}{\partial zo}\frac{\partial zo}{\partial ao}\frac{\partial ao}{\partial w_{h_io_j}}$
$W_{h_io_j}^{(n)} = W_{h_io_j}^{(o)} *$ learning rate
Section: 3
Change of Loss w.r.t hidden:
$\frac{\partial L}{\partial ah_i} = \frac{\partial L}{\partial zo}\frac{\partial zo}{\partial ao}\frac{\partial ao}{\partial zh_i}\frac{\partial zh_i}{\partial ah_i}$
Section: 4
Weight update:
$\frac{\partial L}{\partial w_{i_ih_j}} = \frac{\partial L}{\partial zo}\frac{\partial zo}{\partial ao}\frac{\partial ao}{\partial zh_i}\frac{\partial zh_i}{\partial ah_i}\frac{\partial ah_i}{\partial w_{i_ih_j}}$
$W_{i_ih_j}^{(n)} = W_{i_ih_j}^{(o)} *$ learning rate

# References


- Raval, S. (2018). *Generative Adversarial Networks (LIVE)*. [online] YouTube. Available at: https://www.youtube.com/watch?v=0VPQHbMvGzg [Accessed 7 Jun. 2018].
- Raval, S. (2018). *Convolutional Neural Networks - The Math of Intelligence (Week 4)*. [online] YouTube. Available at: https://www.youtube.com/watch?v=FTr3n7uBIuE [Accessed 7 Jun. 2018].
- YouTube. (2018). *Backpropagation calculus | Appendix to deep learning chapter 3*. [online] Available at: https://www.youtube.com/watch?v=tIeHLnjs5U8 [Accessed 7 Jun. 2018].
- Raval, S. (2018). *Generating Pokemon with a Generative Adversarial Network*. [online] YouTube. Available at: https://www.youtube.com/watch?v=yz6dNf7X7SA [Accessed 7 Jun. 2018].
- Raval, S. (2018). *llSourcell/Generative_Adversarial_networks_LIVE*. [online] GitHub. Available at: https://github.com/llSourcell/Generative_Adversarial_networks_LIVE/blob/master/EZGAN.ipynb [Accessed 7 Jun. 2018].
- Raval, S. (2018). *llSourcell/Convolutional_neural_network*. [online] GitHub. Available at: https://github.com/llSourcell/Convolutional_neural_network/blob/master/convolutional_network_tutorial.ipynb [Accessed 7 Jun. 2018].
- Objective function, l. (2018). *Objective function, cost function, loss function: are they the same thing?*. [online] Cross Validated. Available at: https://stats.stackexchange.com/questions/179026/objective-function-cost-function-loss-function-are-they-the-same-thing [Accessed 7 Jun. 2018].
- https://www.youtube.com/watch?v=tIeHLnjs5U8
- https://dev.to/shamdasani/build-a-flexible-neural-network-with-backpropagation-in-python
- http://code.activestate.com/recipes/578148-simple-back-propagation-neural-network-in-python-s/