Amity School of Engineering and Technology

B.Tech. Computer Science and Engineering

Semester VI

Generative Artificial Intelligence (Generative AI)

[AIML 303]

Faculty: Dr Rinki Gupta

Module II:
Model Parameters Optimization and Convolution Neural Networks

Module II (20%)

Methods to deal with overfitting issues, Optimizers (Momentum, Nesterov Accelerated Gradient, Adagrad, RMSprop, Adam), Convolution neural networks, Understanding the architectural characteristics of deep CNNs, Transfer Learning and Fine-tuning

Problem of Overfitting

- Large neural nets trained on relatively small datasets can overfit the training data

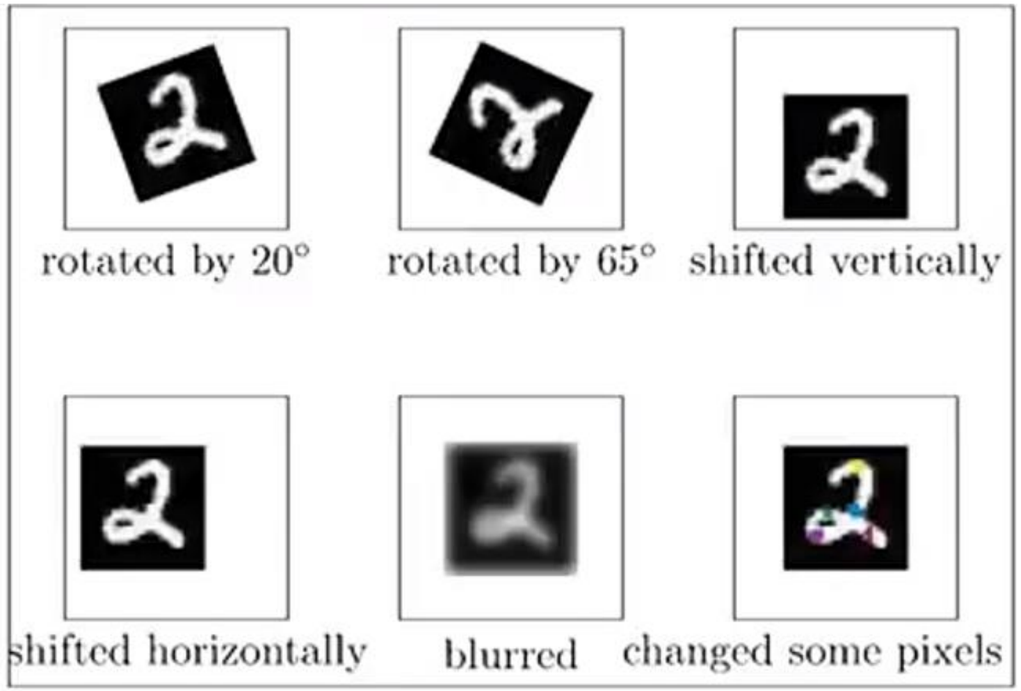
- Techniques to Avoid Overfitting
  - Data Augmentation
  - Regularization
  - Drop-out
  - Early-stopping

Data Augmentation

- Train with more data to avoid overfitting, regularize the model
- Capturing and labeling of data is usually expensive
- New data is generated from existing data, with the help of
  - image rotations,
  - Translation
  - Blur, include noise
  - Change brightness
  - scaling
  - flips (up down, left right)
  - and so on

label = 2

[given training data]



Regularization

- Large weights are a sign of a more complex network -> may overfit
- Regularization is required to avoid Overfitting

$$Cost = Loss(\underline{w}) + \lambda\, Complexity(\underline{w}) = Loss(\underline{w}) + \lambda \sum_{i=1}^{n} |w_i|^q$$

Note: λ is a hyperparameter. As λ increase, $Complexity(\underline{w})$ decreases, and vice versa.

https://srdas.github.io/DLBook/ImprovingModelGeneralization.html#Regularization

Regularization

- Large weights are a sign of a more complex network -> may overfit
- Regularization is required to avoid Overfitting

$$Cost = Loss(\underline{w}) + \lambda\, Complexity(\underline{w}) = Loss(\underline{w}) + \lambda \sum_{i=1}^{n} |w_i|^q$$

L_1 regularization or LASSO Regularization

- minimize sum of abs value of weights ($q = 1$)
- $Cost = Loss(\underline{w}) + \lambda \sum_{i=1}^{n} |w_i|$
- Produces sparse model as wts->zero

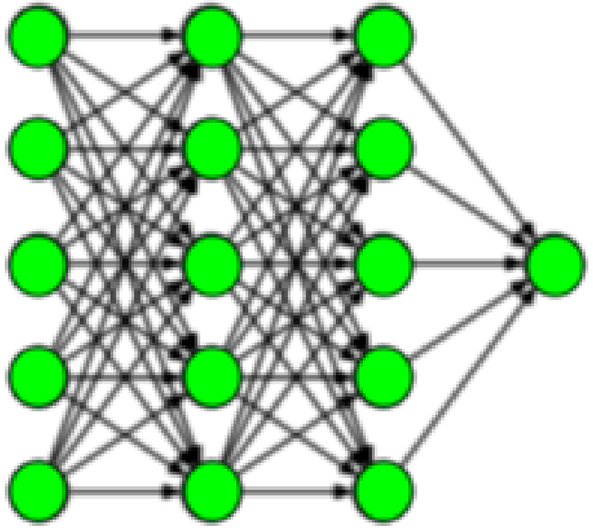
L_2 regularization or RIDGE Regularization

- minimize sum of square of weights ($q = 2$)
- $Cost = Loss(\underline{w}) + \lambda \sum_{i=1}^{n} |w_i|^2$
- Pushes wts towards zero, but doesn't make them exactly zero

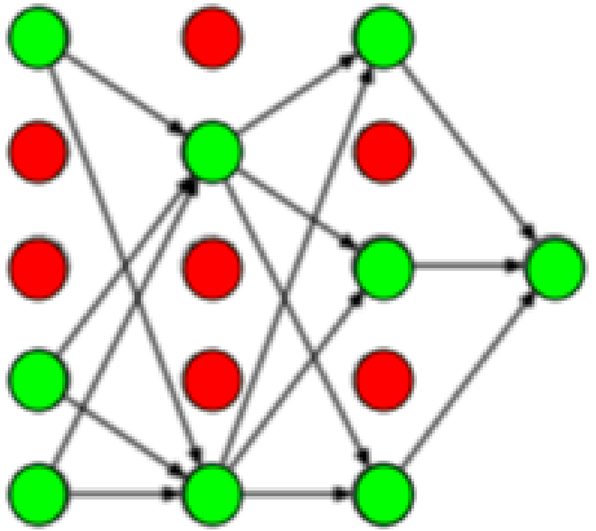
Note: λ is a hyperparameter. As λ increase, $Complexity(\underline{w})$ decreases, and vice versa.

Drop-out

- During training, some number of layer outputs are randomly ignored or “dropped out”
- During weight updation, the layer configuration appears “new”
- Provides Regularization by avoiding co-adaption between network layers to correct mistakes from prior layers
- Improves generalization of the model
- Useful in Wider Networks to avoid overfitting



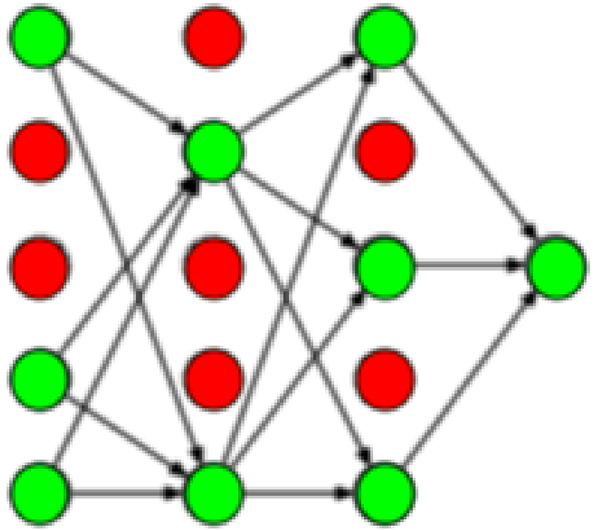
Standard Neural Network



Neural Network with Dropout

Drop-out

- Implemented layer-wise in a neural network
- Implemented on any or all hidden layers in the network, as well as the input layer, but not on the output layer
- A hyperparameter specifies the probability at which outputs of a layer are dropped out, typically
  - 0.5 for hidden layers
  - 0.2 or less for input layer
- Can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers
- During testing, there is no drop-out of units

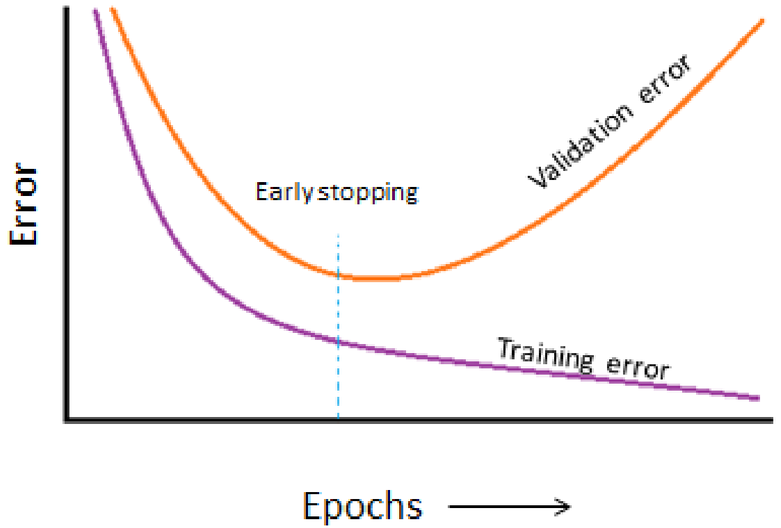


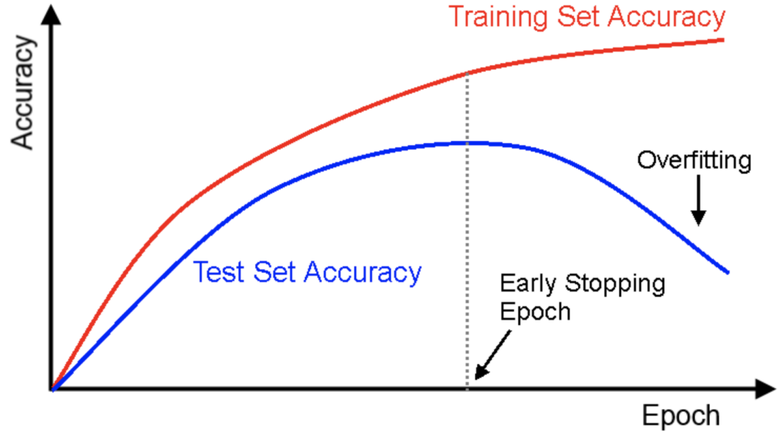
Neural Network with Dropout

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R..“Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, 2014
https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

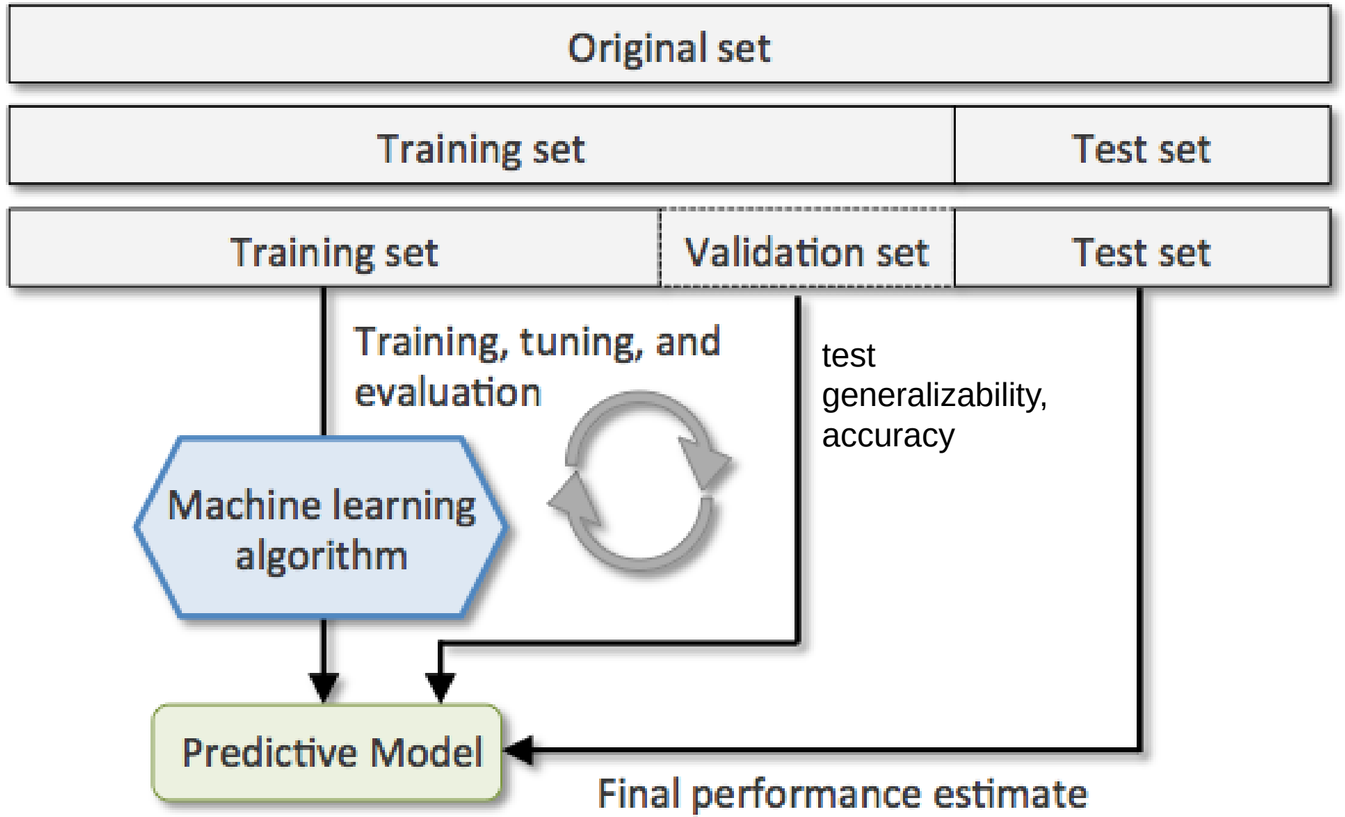
Early Stopping

- Number of Iterations (epochs) is a hyperparameter
  - Less epochs=> Suboptimal solution (Underfit)
  - Too many epochs=> Overfitting



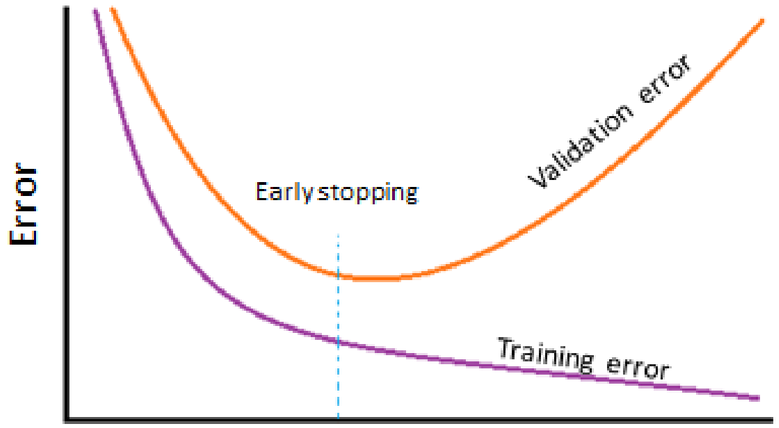


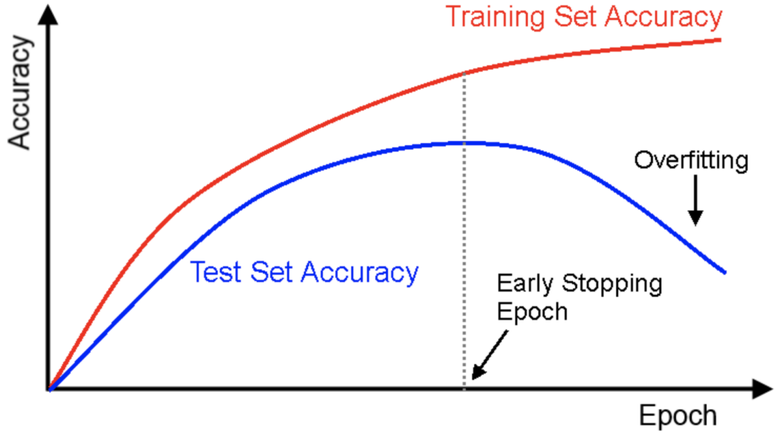
Validation Data for Model Tuning and Model Selection



Early Stopping

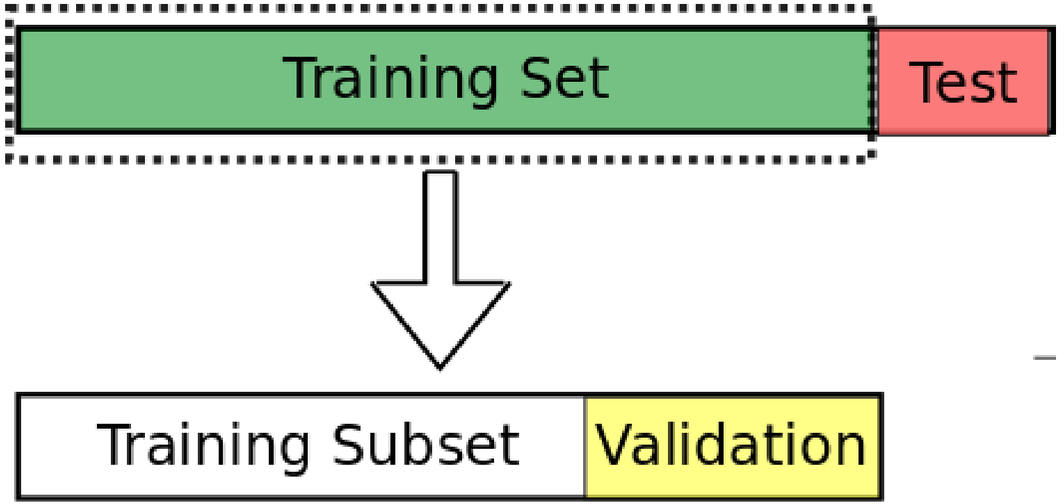
- Number of Iterations (epochs) is a hyperparameter
  - Less epochs=> Suboptimal solution (Underfit)
  - Too many epochs=> Overfitting
- Inspect the validation loss/accuracy:
  - When val loss goes up again (or val acc ↓), while training loss remains constant or decreases (or training acc ↑ or remains same), model may be overfitting
  - Hence, stop iterations. Load previously saved optimal model weights.





Hold-out cross validation

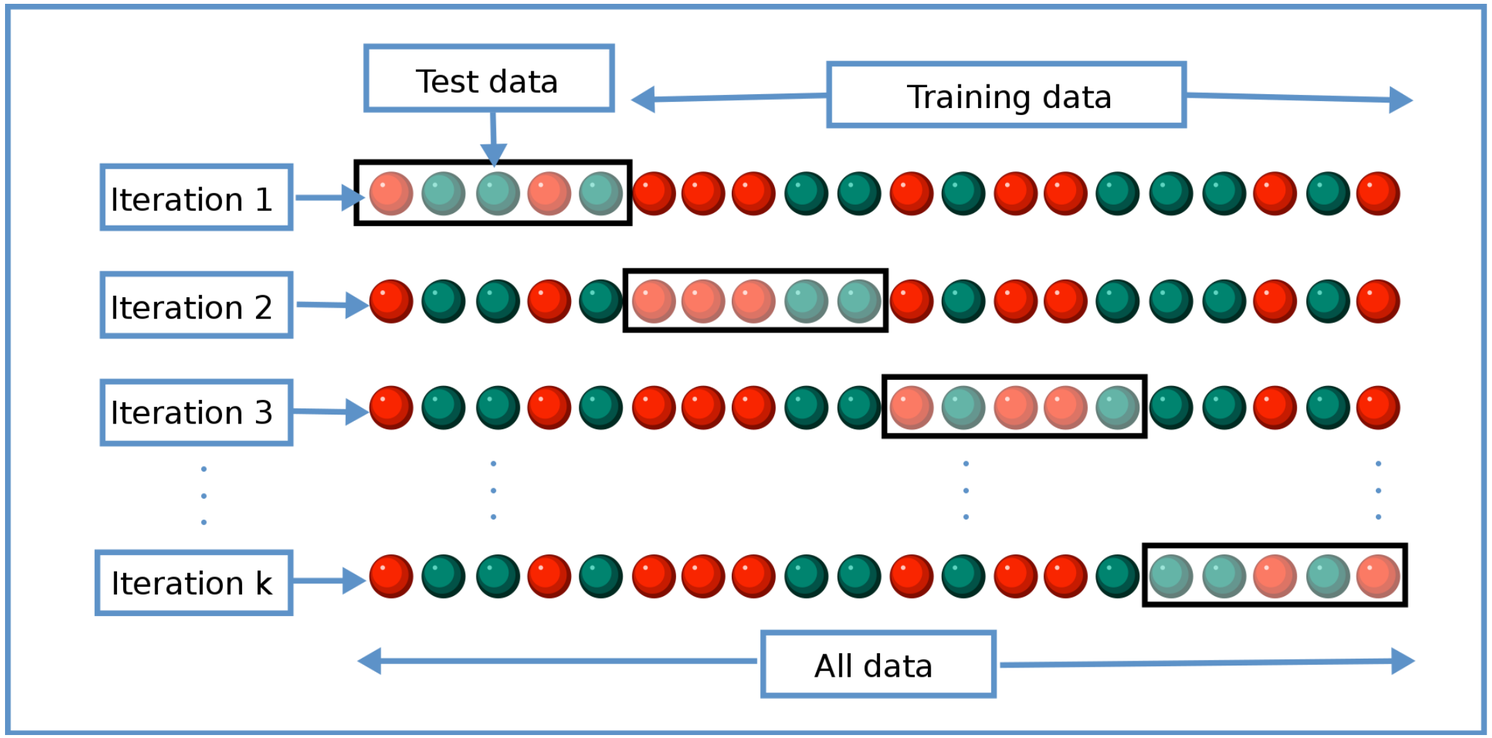
- When data is plentiful, set aside a part of training data as Validation Data-> Perform Model Selection
- Declare final result on Test Data
- Typical ratio for splitting into training, validation, test data: 60:20:20



k-fold Cross validation

- K-fold cross-validation
- When data is not sufficient, split data in k segments,

  train with (k-1) segments, validate with 1 segment and iterate



Training Tips

- Avoid Overfitting
  - Data Augmentation
  - Regularization
  - Drop-out
  - Early-stopping
- Improve Training
  - Optimizers
  - Normalization

Module II:
Model Parameters Optimization and Convolution Neural Networks

Module II (20%)

Methods to deal with overfitting issues, Optimizers (Momentum, Nesterov Accelerated Gradient, Adagrad, RMSprop, Adam), Convolution neural networks, Understanding the architectural characteristics of deep CNNs, Transfer Learning and Fine-tuning

Optimizers in Neural Networks: https://www.youtube.com/watch?v=7m8f0hP8Fzo

Training of Neural Networks: Optimizers with

Fixed Learning Rate

- Gradient descent (GD)
- Stochastic Gradient Descent (SGD)
- Mini Batch Gradient Descent

- Momentum
- Nesterov Accelerated Gradient (NAG)

Adaptive Learning Rates

- Adagrad (Adaptive gradient)
- RMSProp (Root mean square propogation)
- Adam (Adaptive moment)

Gradient Descent based Optmization

- Batch Gradient Descent (BGD): $W \leftarrow W - \eta \sum_{\forall X_i} \nabla_W \mathrm{E}$

It uses the *entire dataset* at every step, making it slow for large datasets. However, it is computationally efficient, since it produces a *stable* error gradient and a stable convergence

- Stochastic Gradient Descent (SGD): $W \leftarrow W - \eta \nabla_W \mathrm{E}$

It is on the other extreme of the idea, using a *single example* (batch of 1) per each learning step. Much faster, may return *noisy gradients* which can cause the error rate to jump around

- Mini Batch Gradient Descent: $W \leftarrow W - \eta \sum_{X_i \in Minibatch} \nabla_W \mathrm{E}$

  Computes the gradients on small random sets of instances called *mini batches*. Reduce noise from SGD and still more efficient than BGD

Momentum Optimizer/ SGD with Momentum

- SGD: $W_{t+1} \leftarrow W_t - \eta \nabla_W \mathrm{E}$
- Momentum Optimizer: $W_{t+1} \leftarrow W_t + v_t$

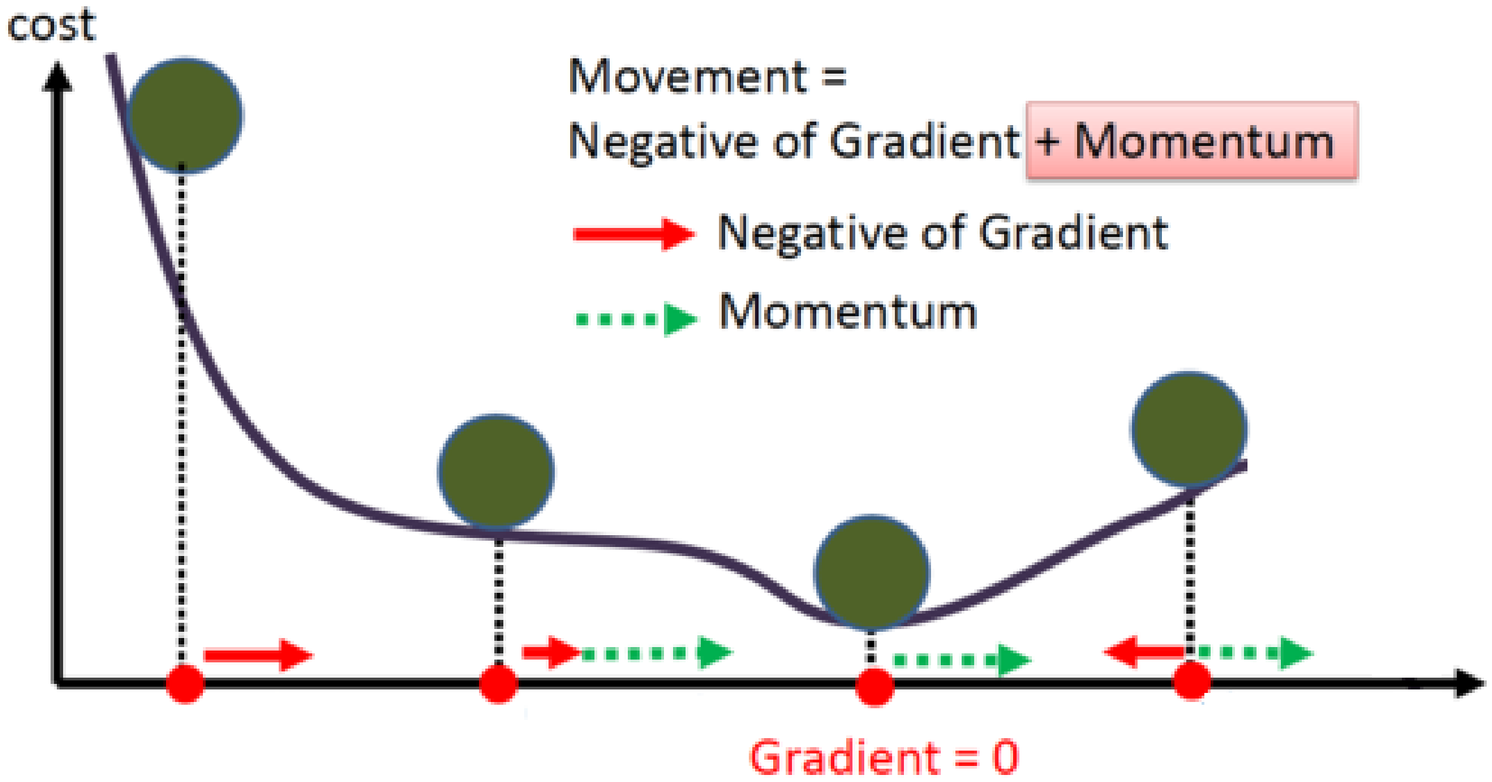
  Velocity $v_t \leftarrow \alpha v_{t-1} - \eta g_t$, $g_t = \nabla_W \mathrm{E}$
- Momentum (v) accumulates an exponentially decaying moving average of past gradients and continues to move in their direction
- $\alpha \epsilon [0,1)$ determines how quickly the contributions of previous gradients exponentially decay

In Physics,
Momentum = mass x velocity
Taking unit mass,
Momentum = velocity

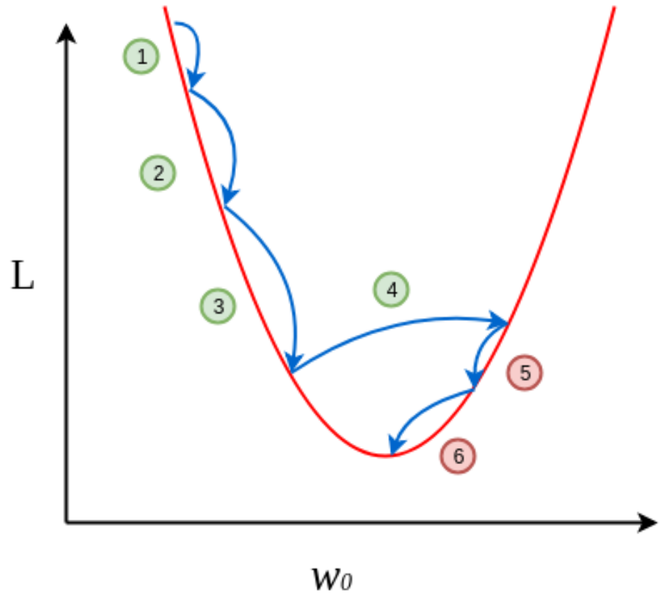
consider past gradients to smooth out the update

Momentum Optimizer

- As slope (grad) increases, step increases (ball gains momentum)
- Near local minimum (slope decreases, momentum continues to move the ball in same direction)
- Solution may overshoot the minimum, oscillate around it before converging to min
- Avoids Local Minimum



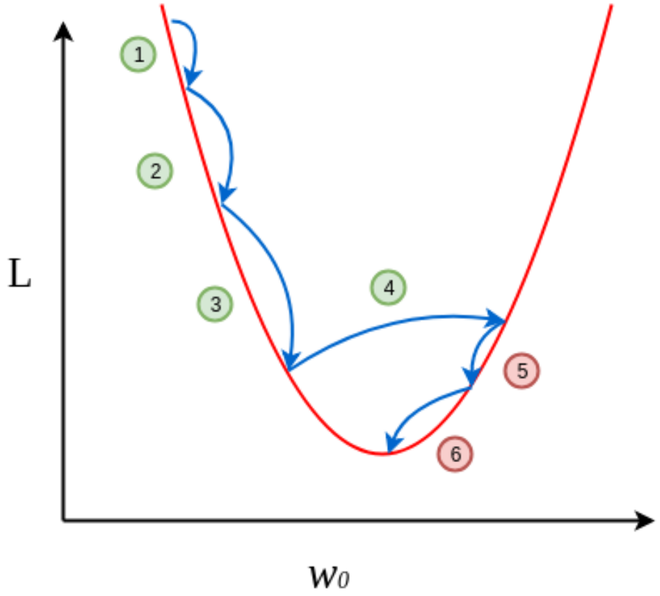
Momentum Optimizer



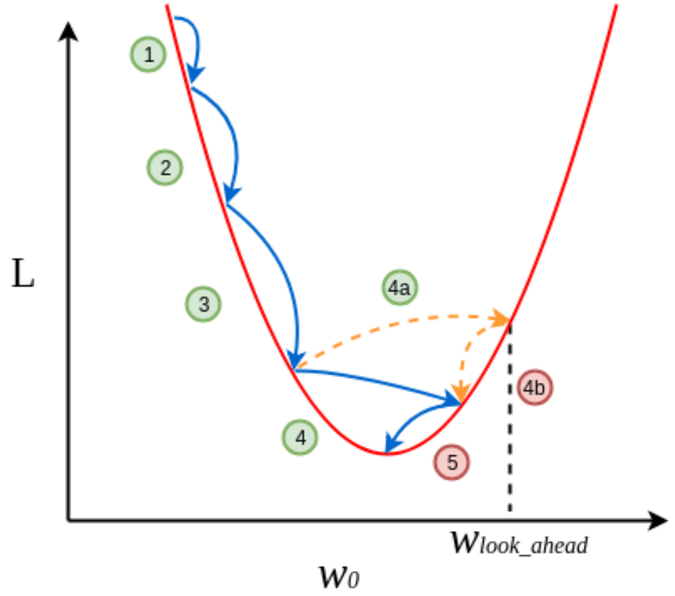
(a) Momentum-Based Gradient Descent

- Momentum may be a good method but if the momentum is too high the algorithm may miss the local minima and may continue to rise up.
- To resolve this issue the Nesterov Accelerated Gradient algorithm was developed

Nesterov Accelerated Gradient



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

- NAG is a momentum-based SGD optimizer that "looks ahead" to where the parameters will be to calculate the gradient ex post rather than ex ante:

$$W_{t+1} \leftarrow W_t + v_t$$

$$v_t \leftarrow \alpha v_{t-1} + \eta \nabla_W \mathrm{E}(W - \alpha v_{t-1})$$

Training of Neural Networks: Optimizers with

Fixed Learning Rate

- Gradient descent (GD)
- Stochastic Gradient Descent (SGD)
- Mini Batch Gradient Descent

- Momentum
- Nesterov Accelerated Gradient (NAG)

Adaptive Learning Rates

- Adagrad (Adaptive gradient)
- RMSProp (Root mean square propogation)
- Adam (Adaptive moment)

Adaptive Learning Rates

- SGD: $W \leftarrow W - \eta \nabla_W \mathrm{E}$,

  η =constant and same for all weights in W
- **AdaGrad algorithm**:
  - Individually adapts the learning rates of all model parameters by scaling them inversely proportional to square root of the sum of all the historical squared values of the gradient
  - Learning rate reduces faster for parameters showing large slope

$$g_t = \sum_{X_i \in Minibatch} \nabla_W \mathrm{E}$$

$$r_t = \sum_{\tau=1}^{t} g_\tau \cdot g_\tau$$

$$W_t \leftarrow W_t - \frac{\eta}{\sqrt{\varepsilon I + r_t}} \cdot g_t$$

Small value to avoid divide by zero

Adaptive Learning Rates

- SGD: $W \leftarrow W - \eta \nabla_W \mathrm{E}$,

  η =constant and same for all weights in W
- **RMSProp algorithm**:
  - Individually adapts the learning rates of all model parameters by scaling them inversely proportional to square root of exponentially decaying average of squares of gradient
  - Discards history from extreme past

$$g_t = \sum_{X_i \in Minibatch} \nabla_W \mathrm{E}$$

$$r_t = \beta r_{t-1} + (1-\beta)\, g_t \cdot g_t$$

$$W_t \leftarrow W_t - \frac{\eta}{\sqrt{\varepsilon I + r_t}} \cdot g_t$$

Adaptive Moment (Adam) Optimizer

- Adam = RMSProp + SGD with Momentum
- computes adaptive learning rates for each parameter, using
  - exponentially decaying average of past squared gradients RMSprop, and
  - exponentially decaying average of past gradients
- Bias Correction, $\hat{r}_t = \frac{r_t}{1-\beta_1}$, $\hat{s}_t = \frac{s_t}{1-\beta_2}$

$$g_t = \sum_{X_i \in Minibatch} \nabla_W \mathrm{E}$$

$$r_t = \beta_1 \, r_{t-1} + (1-\beta_1) \, g_t \cdot g_t$$

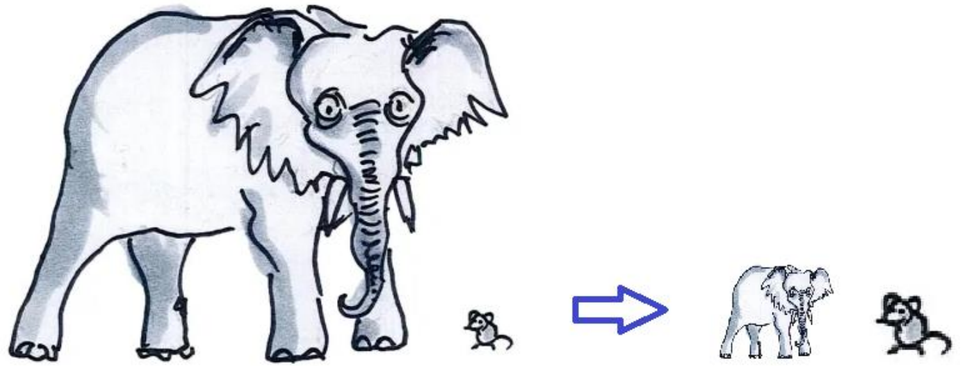
$$s_t = \beta_2 \, s_{t-1} + (1-\beta_2) \, g_t$$

$$W_t \leftarrow W_t - \frac{\eta}{\sqrt{\varepsilon I + \hat{r}_t}} \cdot \hat{s}_t$$

Training Tips

- Avoid Overfitting
  - Data Augmentation
  - Regularization
  - Drop-out
  - Early-stopping
- Improve Training
  - Optimizers
  - Normalization

Normalization/ Feature Scaling



- Different attributes may have different range, hence normalization of values is required

- Standardization or Z-score normalization:
  - Rescale the data so that the mean is zero and the standard deviation from the mean (standard scores) is one

$$x_{norm} = \frac{x - \mu}{\sigma}$$

μ is mean, σ is standard deviation from the mean

- Min-Max normalization:
  - Scale the data to a fixed range – between 0 and 1

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- Standardization or Z-score normalization:

$$x_{norm} = \frac{x - \mu}{\sigma}$$

Similar range

	Sqft	Standardized Sqft		# Beds	Standardized #Beds
	1346	-1.2643		2	-1.2136
	2765	1.4697		6	1.4832
	2120	0.2270		4	0.1348
	1776	-0.4358		3	-0.5394
	2004	0.0035		4	0.1348
Avg	2002.20		Avg	3.80	
Std	464.22		Std	1.48	

Goal: Predict House Price.

Sqft	# Beds	# Baths	Acres	Price
1,346	2	1	0.23	$346,000
2,765	6	5	0.46	$987,000
2,120	4	2	0.29	$620,000
1,776	3	1	0.36	$513,000
2,004	4	4	5.87	$1,390,000

- Min-Max Normalization:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Similar range

	Sqft	Normalized Sqft		# Beds	Normalized #Beds
	1346	0.0000		2	0.0000
	2765	1.0000		6	1.0000
	2120	0.5455		4	0.5000
	1776	0.3030		3	0.2500
	2004	0.4637		4	0.5000
Min	1346		Min	2.00	
Max	2765		Max	6.00	

Goal: Predict House Price.

Sqft	# Beds	# Baths	Acres	Price
1,346	2	1	0.23	$346,000
2,765	6	5	0.46	$987,000
2,120	4	2	0.29	$620,000
1,776	3	1	0.36	$513,000
2,004	4	4	5.87	$1,390,000

Batch Normalization

- In Mini-Batch Gradient Descent
  - Data may be very different from batch to batch
  - Features learned in one batch may not be appropriate for other batch
  - Feature distribution may vary => Decision boundary may vary
- Internal Covariate Shift

Sergey Ioffe and Szegedy Christian. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In *International conference on machine learning*, pp. 448-456. PMLR, 2015.

Batch Normalization

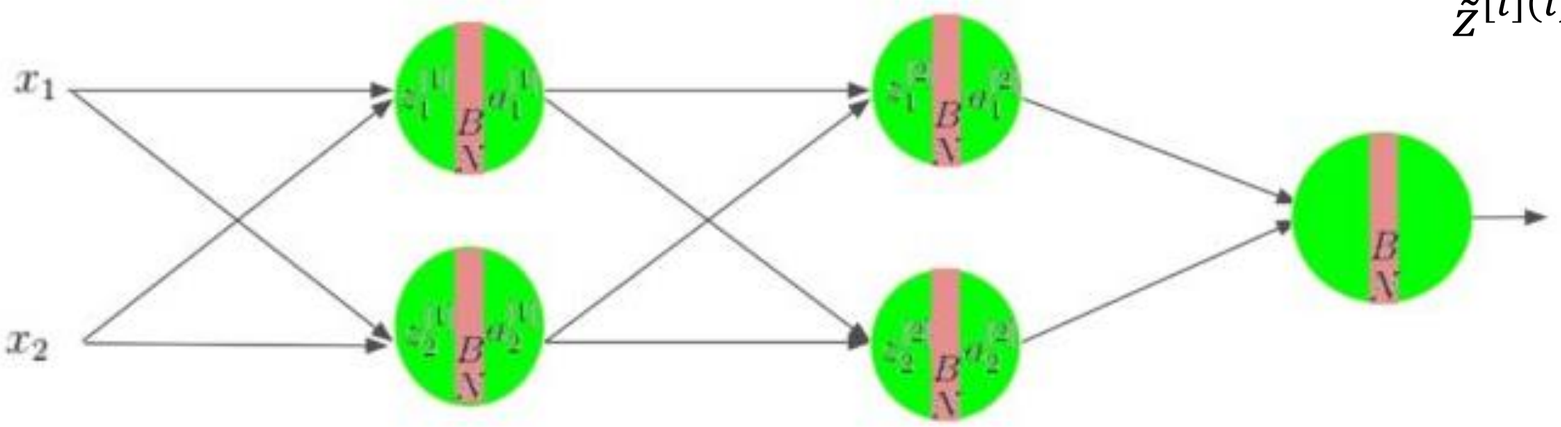
- Feature normalization is required not only in Input layer, but also in Hidden layers

$$z^{[l]} = W^{[l]} \cdot a^{[l-1]} \rightarrow \mu^{[l]} = \frac{1}{n}\sum_i z^{[l](i)}$$

$$\sigma^{[l]2} = \frac{1}{n}\sum_i (z^{[l](i)} - \mu^{[l]})^2$$

$$z_{norm}^{[l](i)} = \frac{z^{[l](i)} - \mu^{[l]}}{\sqrt{\sigma^{[l]2} + \varepsilon}}$$

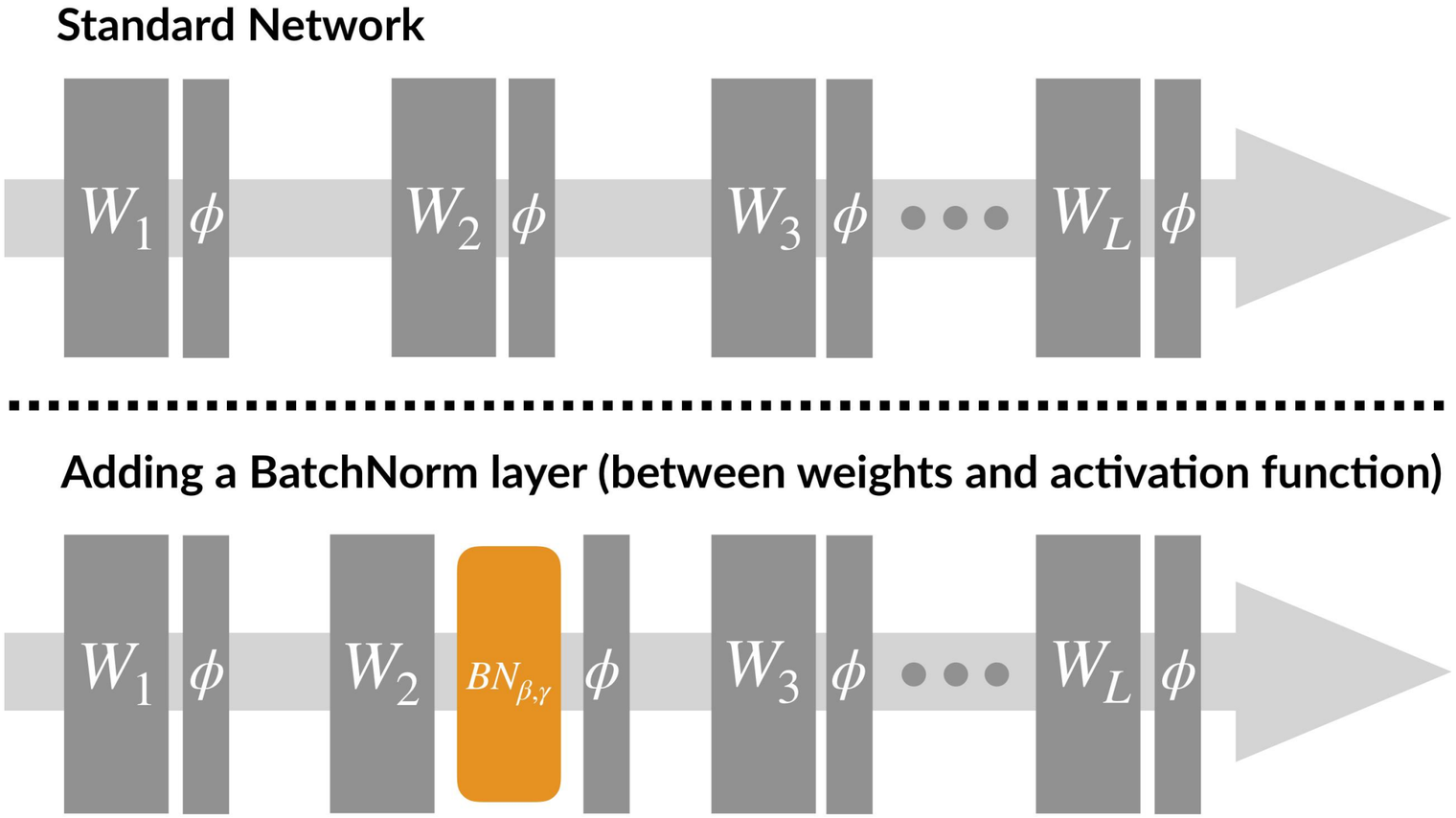
$$\tilde{z}^{[l](i)} = \gamma^{[l]} z_{norm}^{[l](i)} + \beta^{[l]} \rightarrow a^{[l]} = g^{[l]}(\tilde{z}^{[l]})$$



$\mu^{[l]}, \sigma^{[l]2}$ are min-batch mean and variance,
$\gamma^{[l]}, \beta^{[l]}$ are learned along with weights

Batch Normalization

- Neural Networks with BatchNorm tend to train faster, and are less sensitive to the choice of hyperparameters



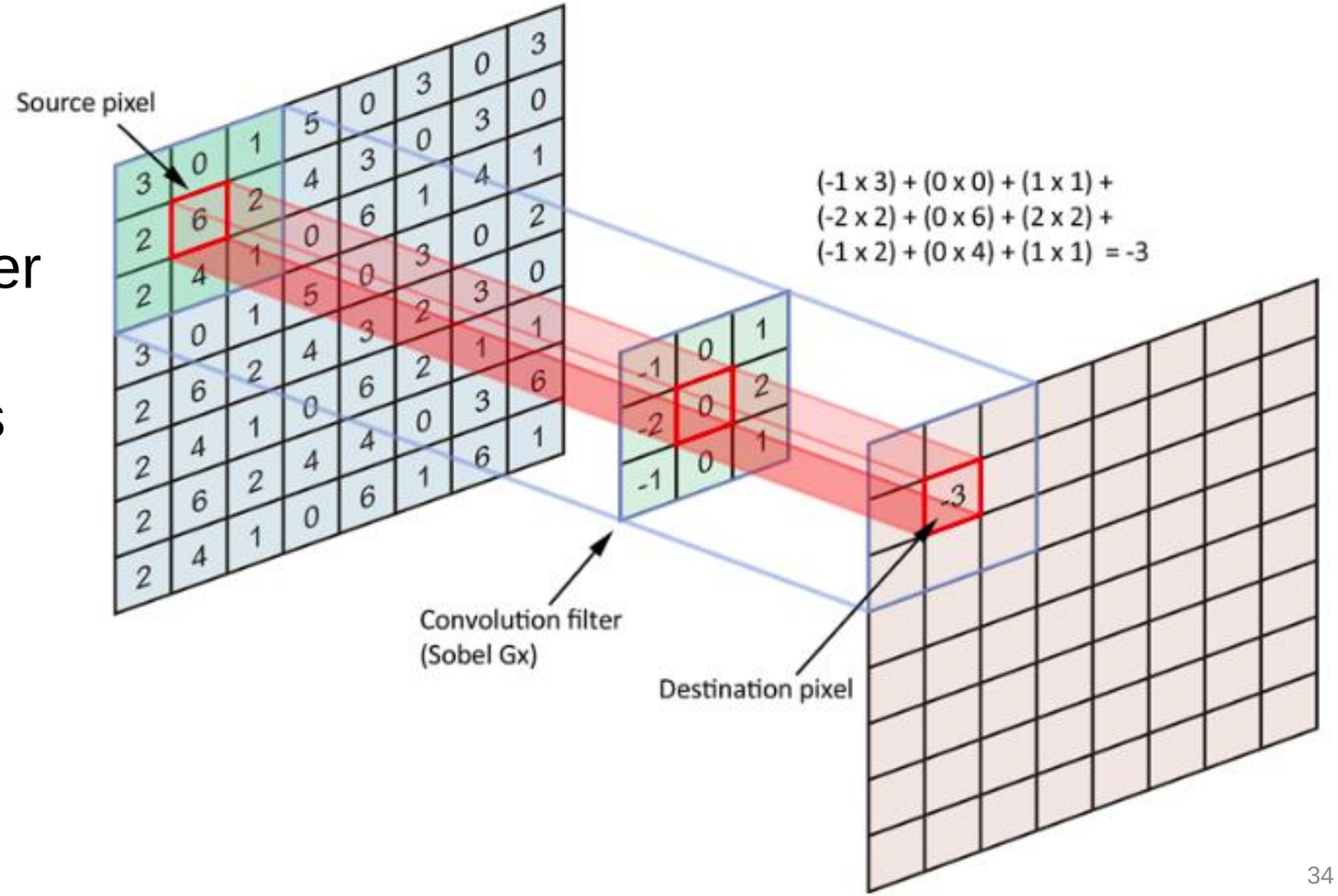
Module II:
Model Parameters Optimization and Convolution Neural Networks

Module II (20%)

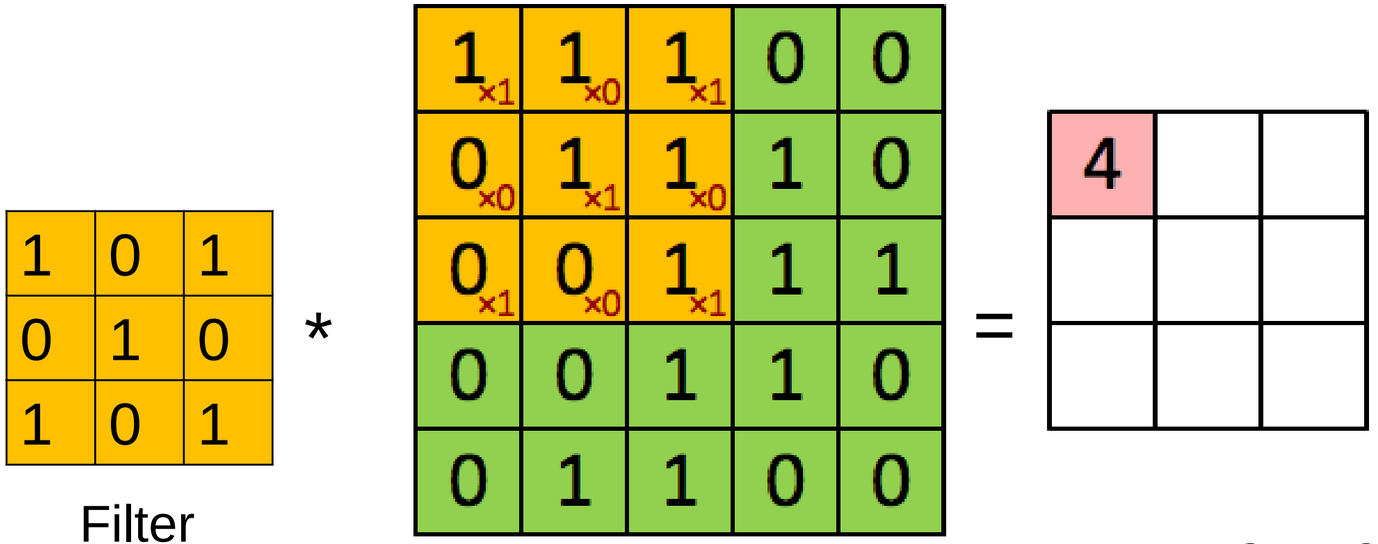
Methods to deal with overfitting issues, Optimizers (Momentum, Nesterov Accelerated Gradient, Adagrad, RMSprop, Adam), Convolution neural networks, Understanding the architectural characteristics of deep CNNs, Transfer Learning and Fine-tuning

"Convolution" in Deep Learning

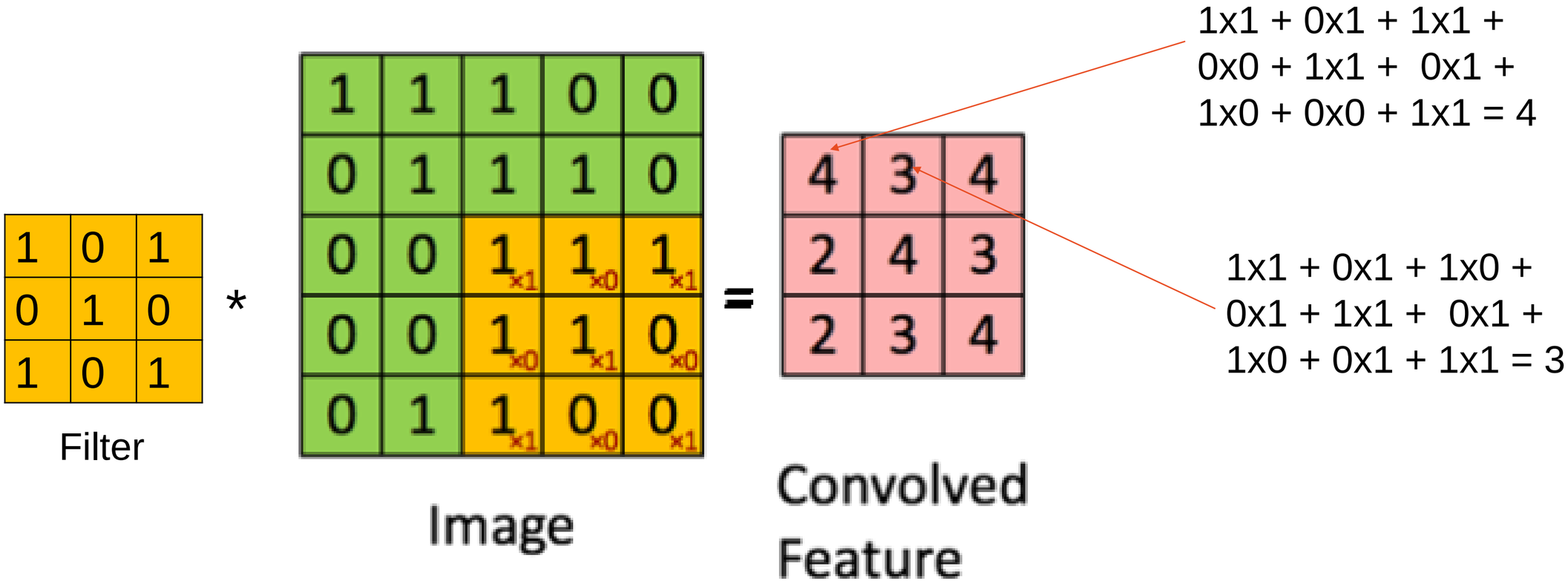
- Weights are defined in the filter
- Output feature is obtained by taking weighted sum of inputs and applying activation function
- Filter is smaller in size than input
- Filter slides across the input



Convolution Operation

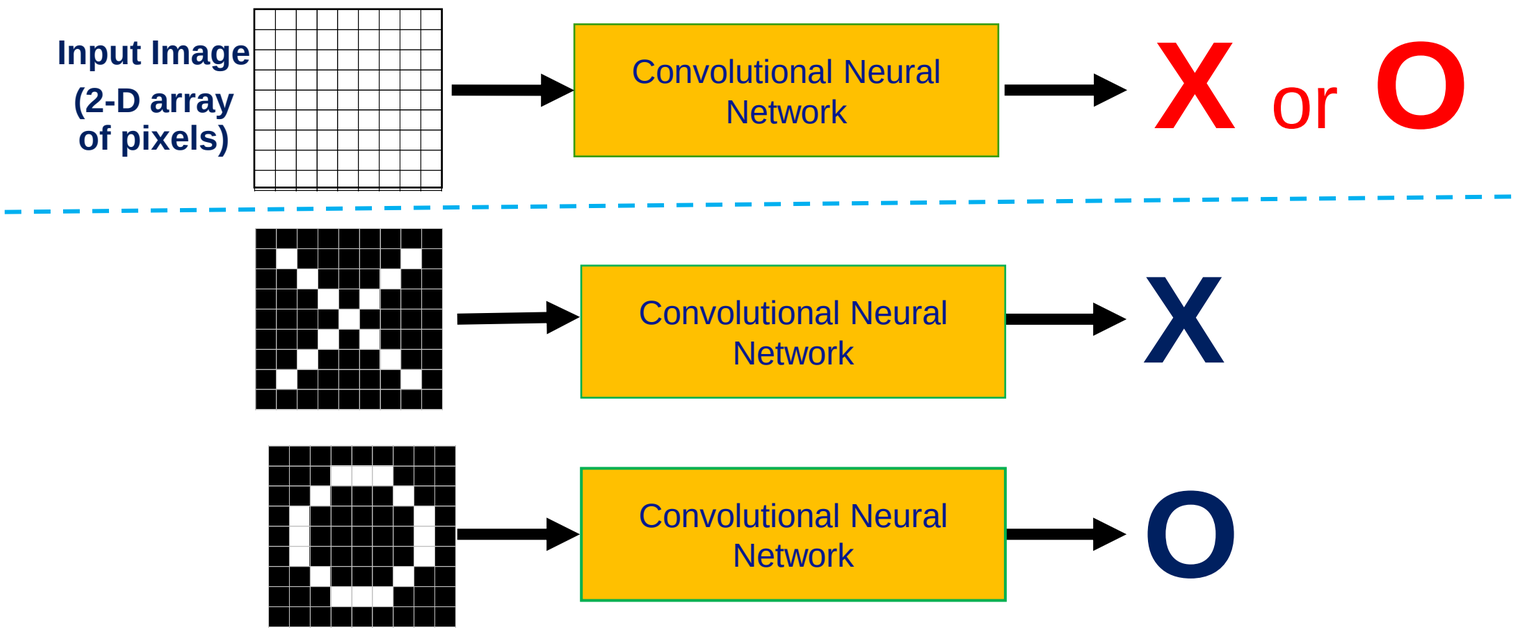


Convolution Operation



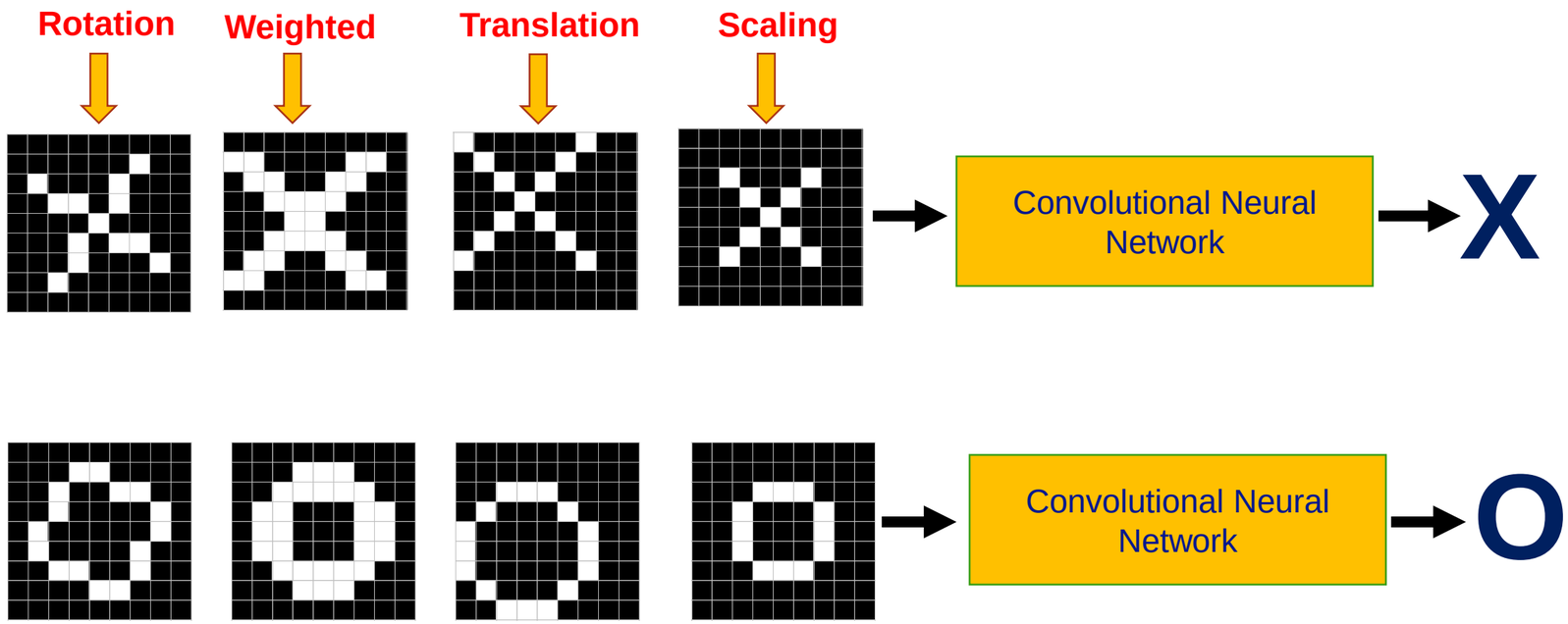
Why Convolution ?

Convolutional Neural Network (Feature Extraction and Convolution)

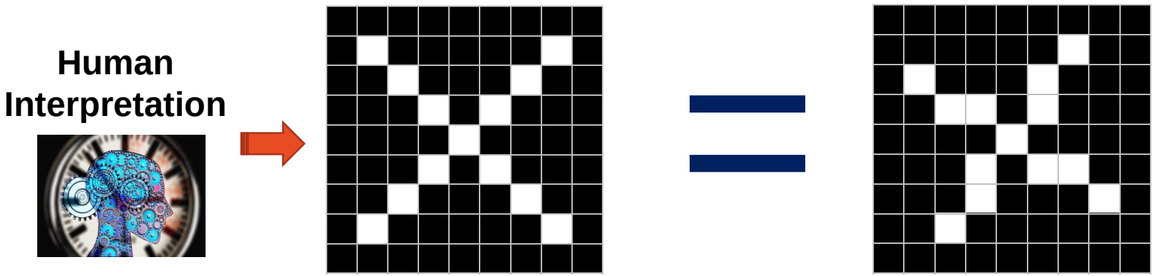


We need to design a Neural Network for classification of x an o

Challenging Cases

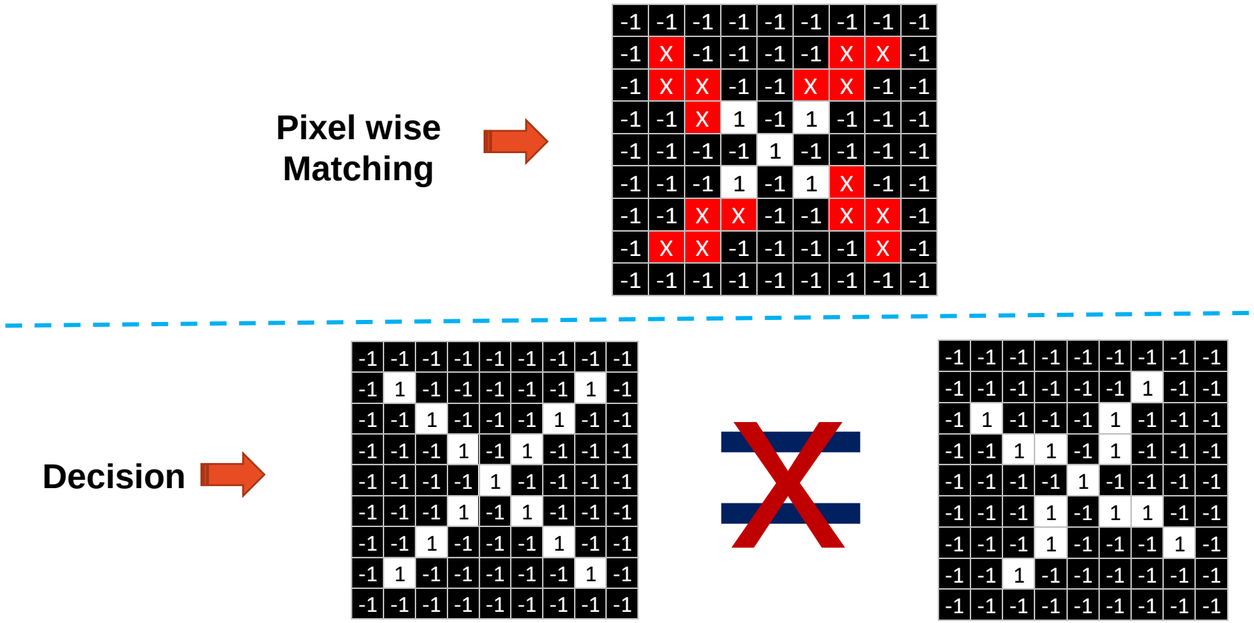


Human Interpretation



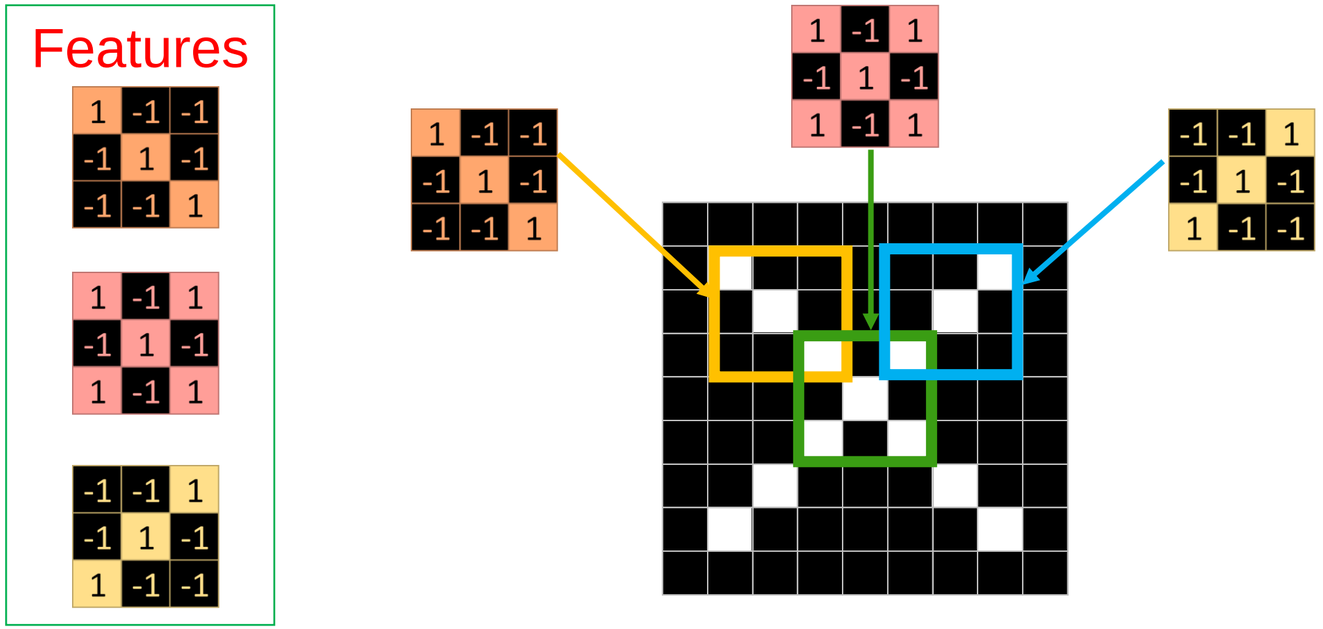
When you look at the two images, they seem as x and not o

Computer Interpretation



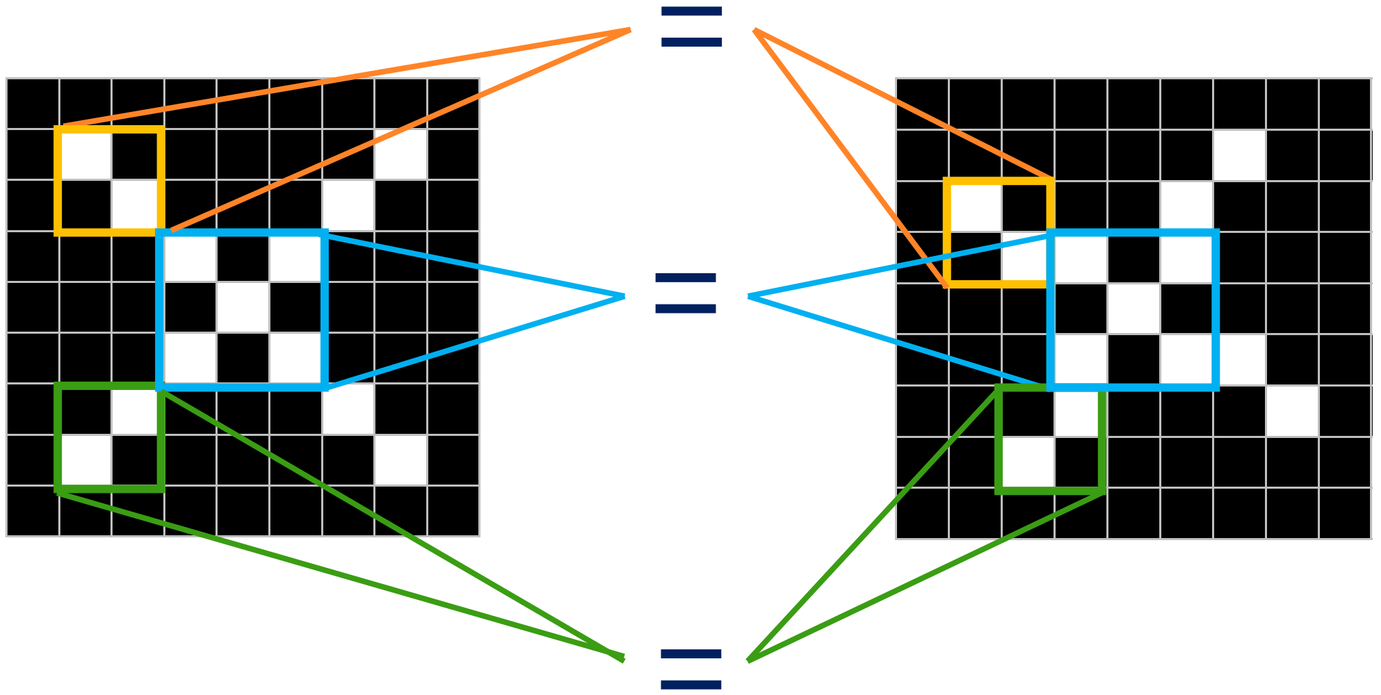
When a computer looks at the two images, it matches them pixel by pixel

Piece Matching of Features in Convolution Layer



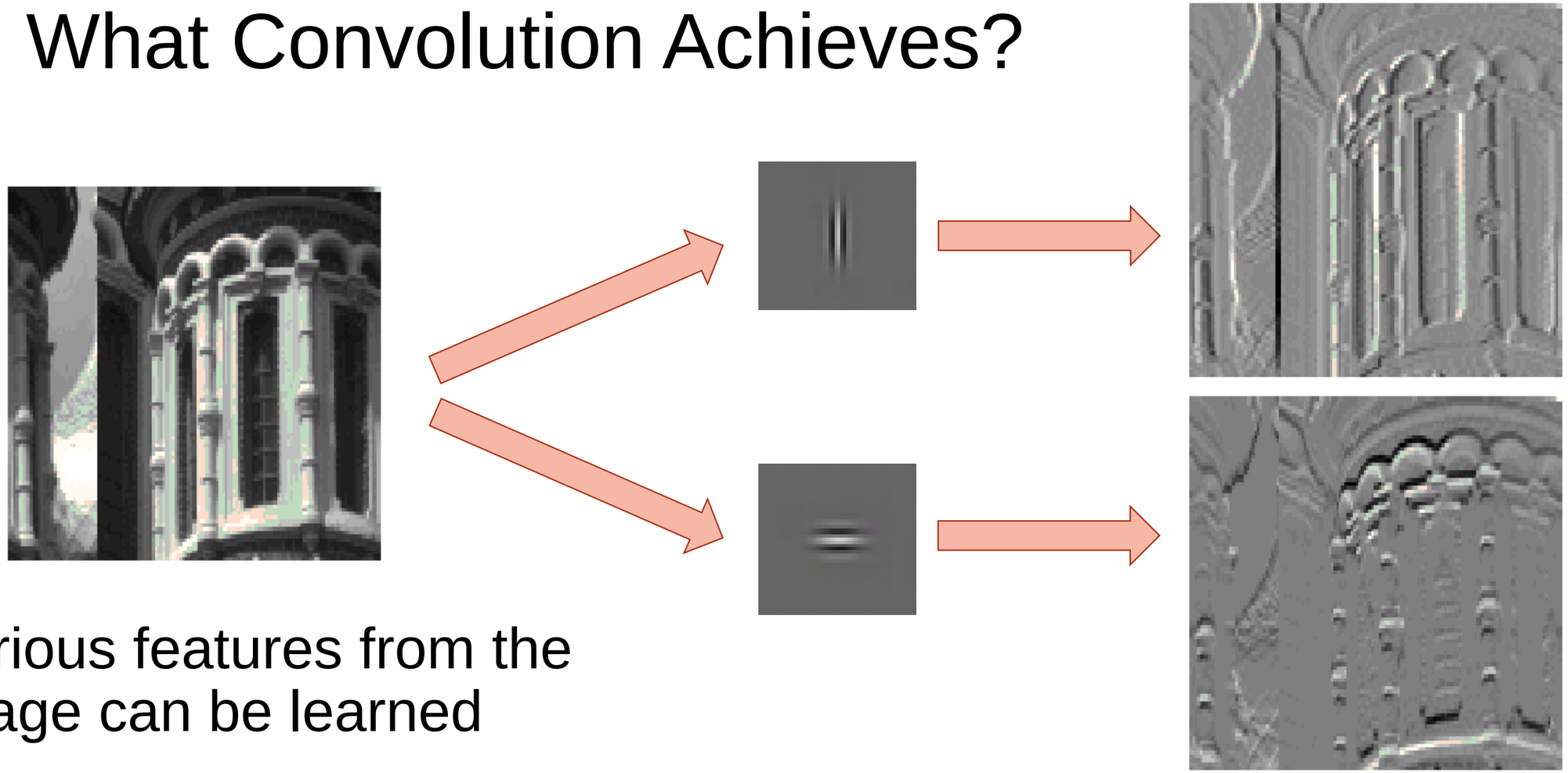
In convolutional layers, feature extraction is carried out on segments of images

Feature matching for symbol ‘X’ Using Convolution Operation



Over segments, similar features may be observed in the two circuits

What Convolution Achieves?



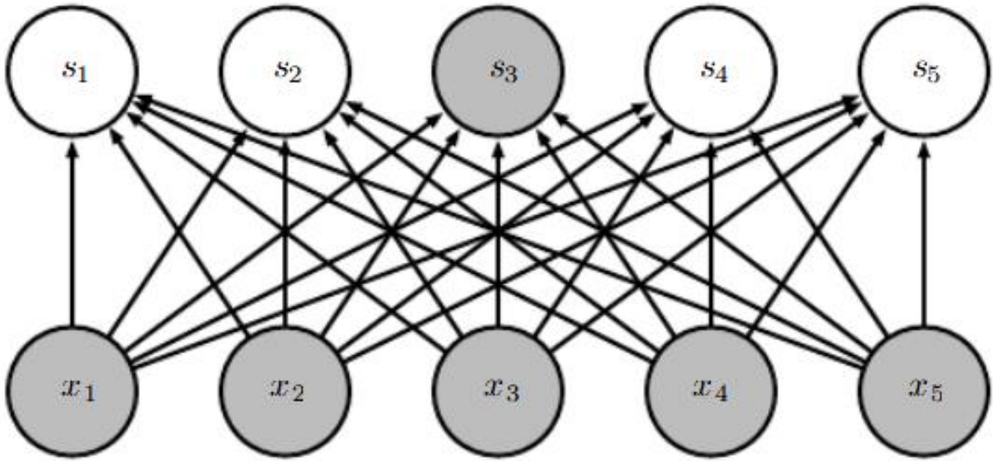
- Various features from the image can be learned

From Dense to Convolutional Architecture

- MNIST image 28x28=784 pixels in image
- 12 megapixel image => extremely large number of hidden nodes required to learn good hidden representations of images
- Three important improvements provided by Convolution to improve learning:
  - Sparse Interactions (Receptive Field)
  - Parameter sharing
  - Equivariant Representations

Dense Layer

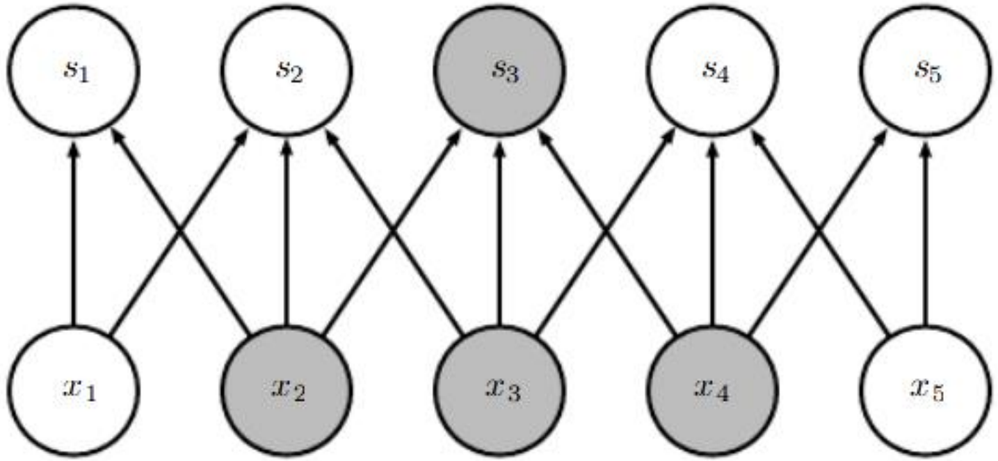
- Every output unit interacts with every input unit, with separate parameter describing the interaction between each input unit and each output unit



Convolutional Layer

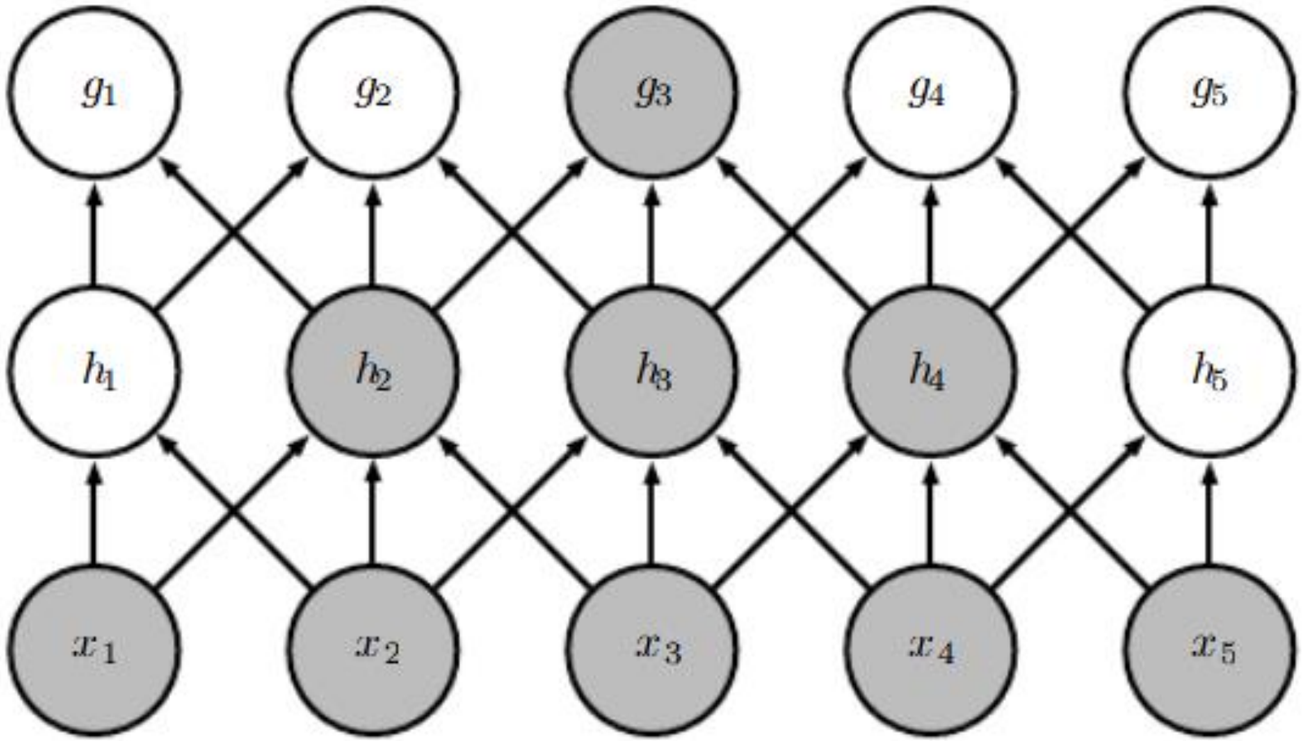
- Kernel smaller than the input
- Have sparse interactions (also referred to as sparse connectivity or sparse weights)

Eg: Kernel width=3 => only three inputs affect *s*



Receptive Field

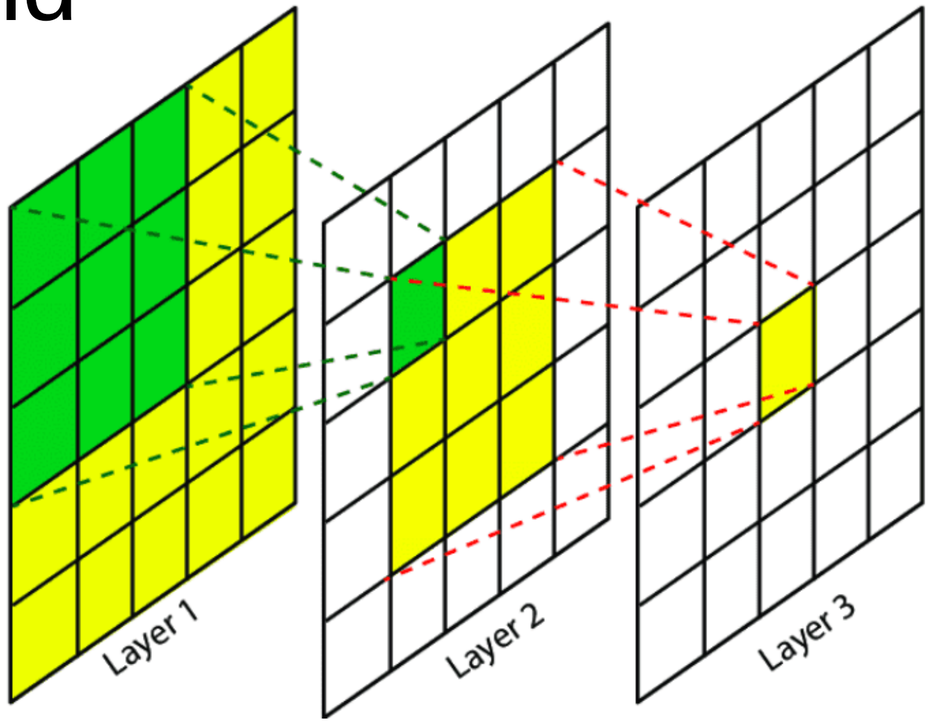
- Range of input samples over which convolution takes place
- Kernel Smaller than input (Eg. 3 here) => sparse weights/connections
- All of input is not required to calculate an output=> Reduce storage and computation



Receptive field of o/p is highlighted

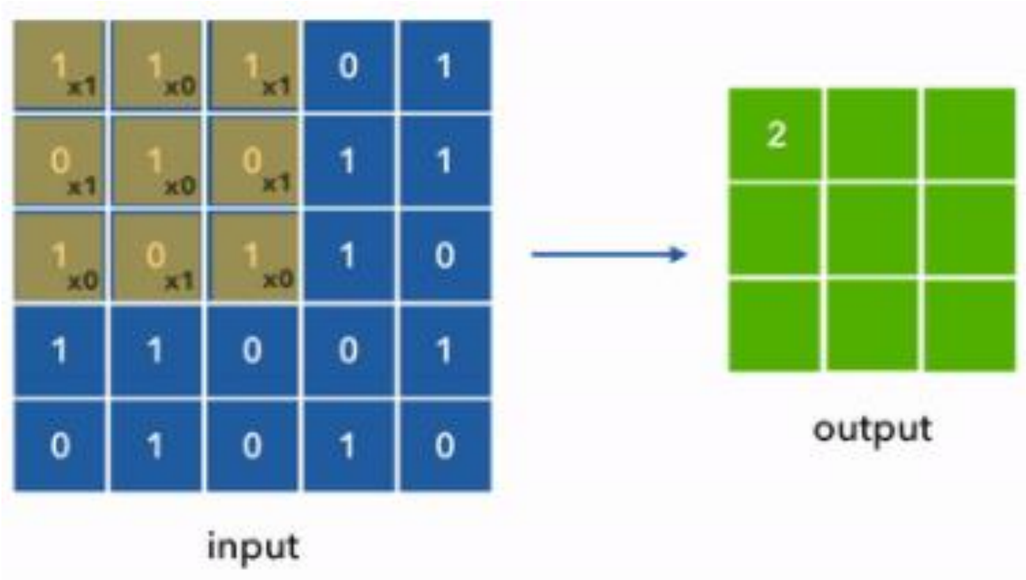
Receptive Field

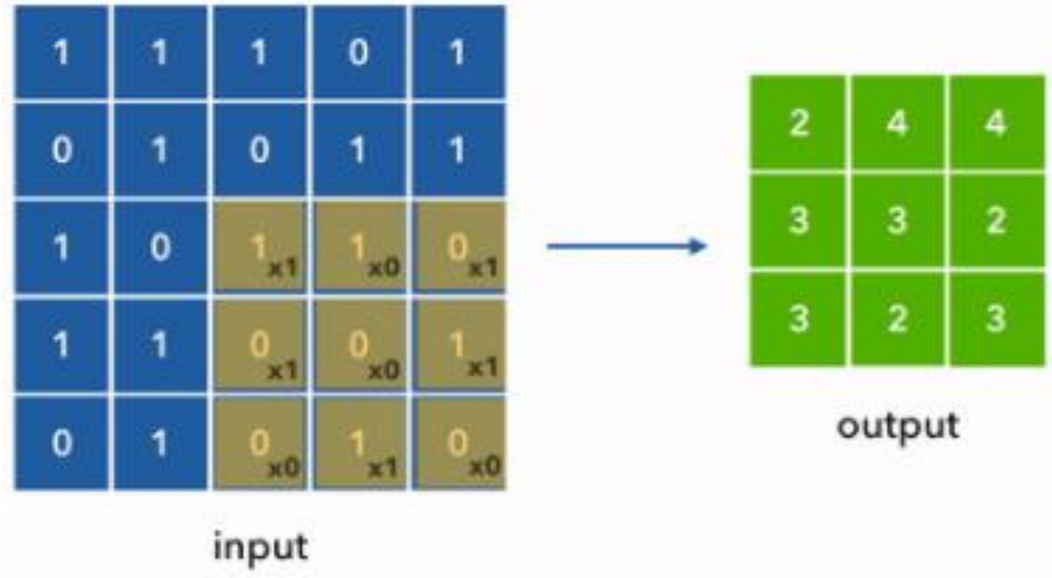
- Deeper layers may indirectly interact with a larger portion of the input to efficiently describe complicated interactions between many variables
- Multiple layers lead to Wide Receptive Field
- Effective receptive field increases if there is pooling/strided convolution



Parameter Sharing

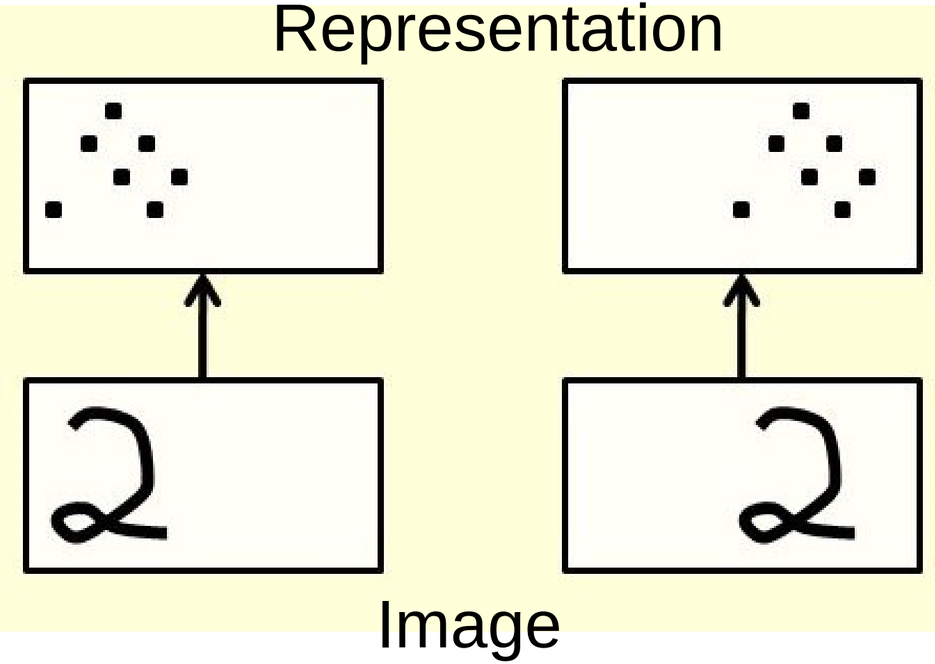
- Parameter sharing refers to using the same parameter for more than one function in a model
- Kernel is reused (by sliding) when calculating the layer o/p
- Less weights to store & train





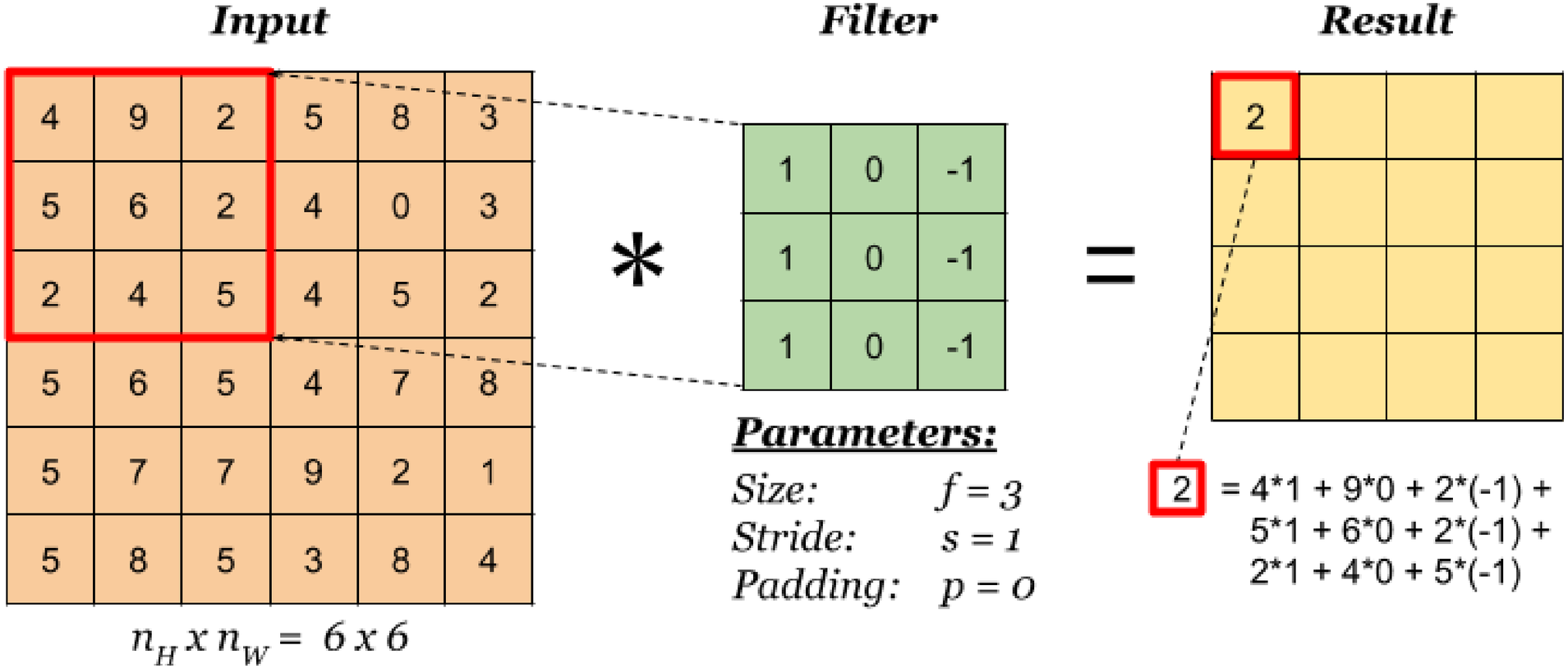
Equivalent Representation

- Parameter sharing causes the layer to have a property called equivariance to translation
- Convolution creates 2-D map of where certain features appear in the input
- If we move the object in the input, its representation will move the same amount in the output
- Eg: Same kernel for Edge Detection wherever the edge occurs in the image



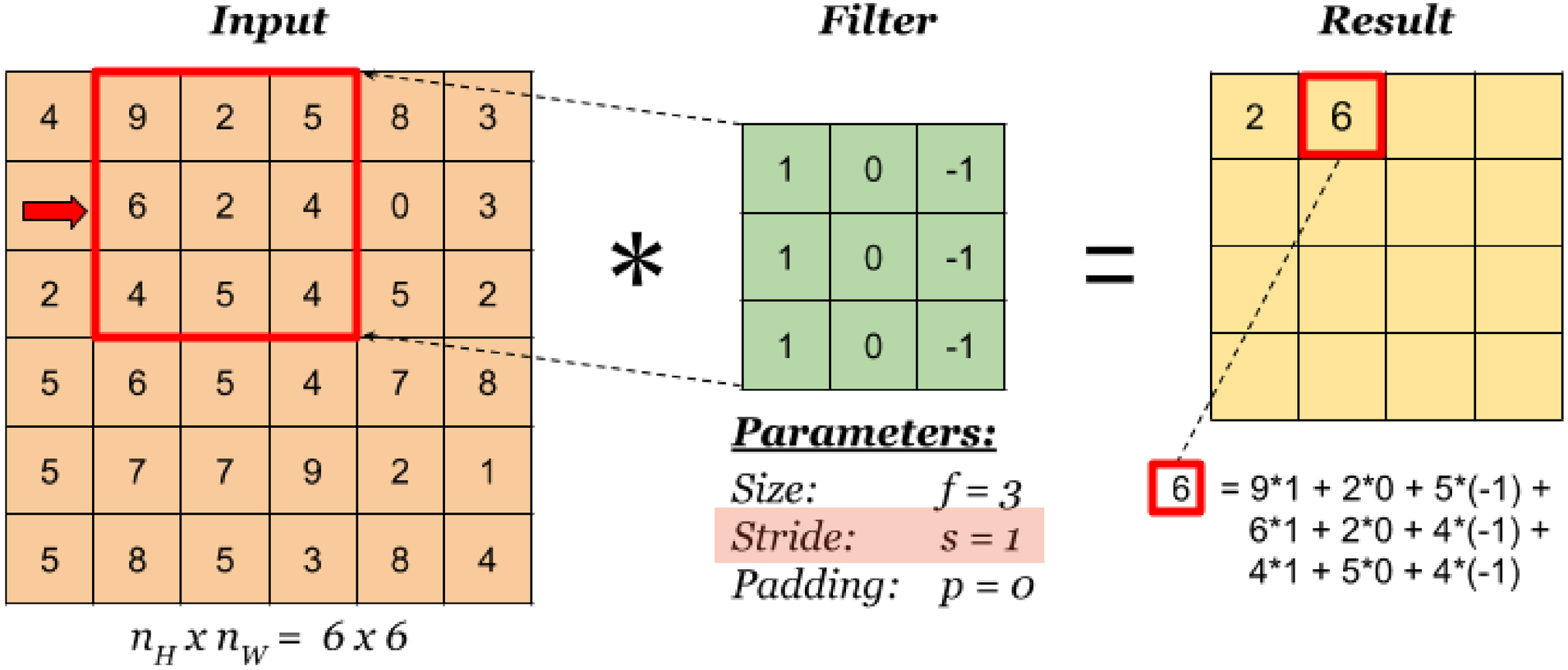
Stride

- number of cells the filter is moved to calculate the next output
- sample only every s pixels in each direction in the output



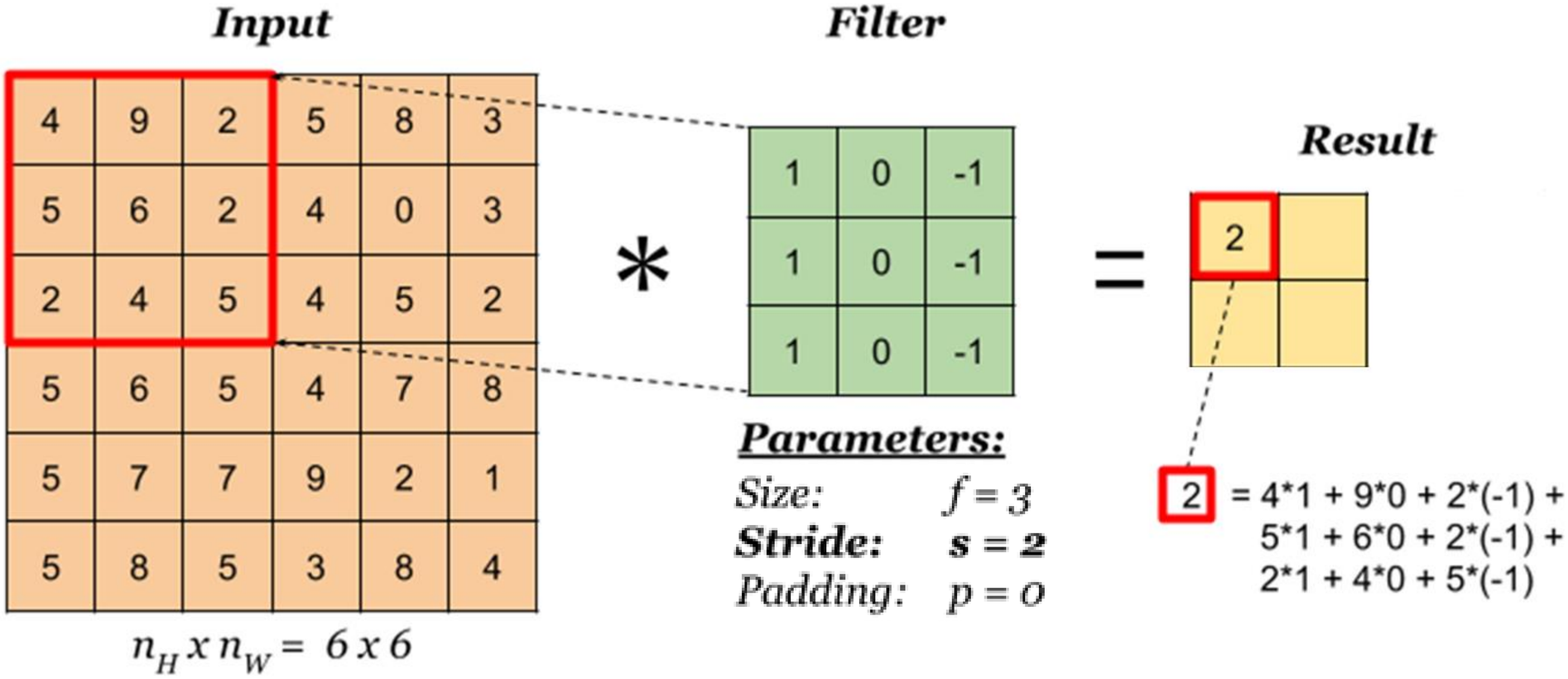
Stride

- number of cells the filter is moved to calculate the next output
- sample only every s pixels in each direction in the output



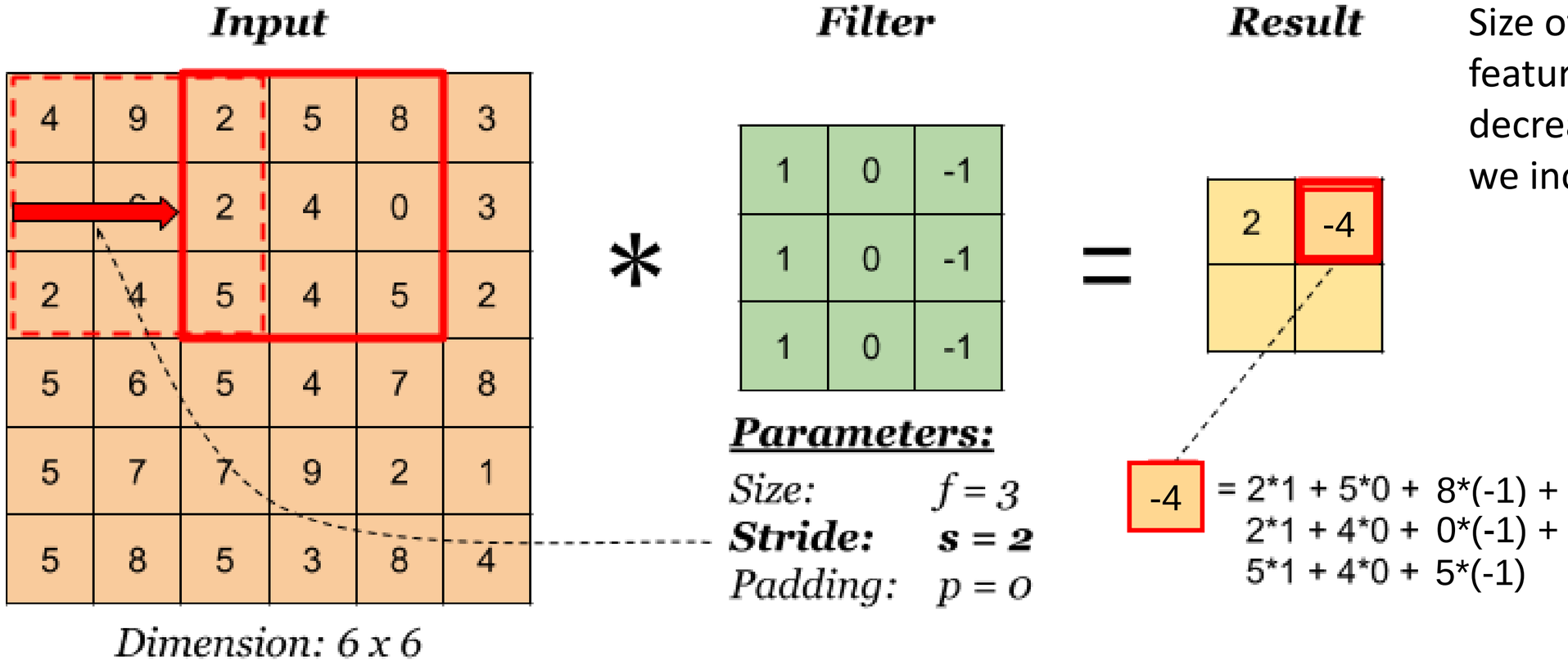
Stride

- Stride = 2
- First Value:



Stride

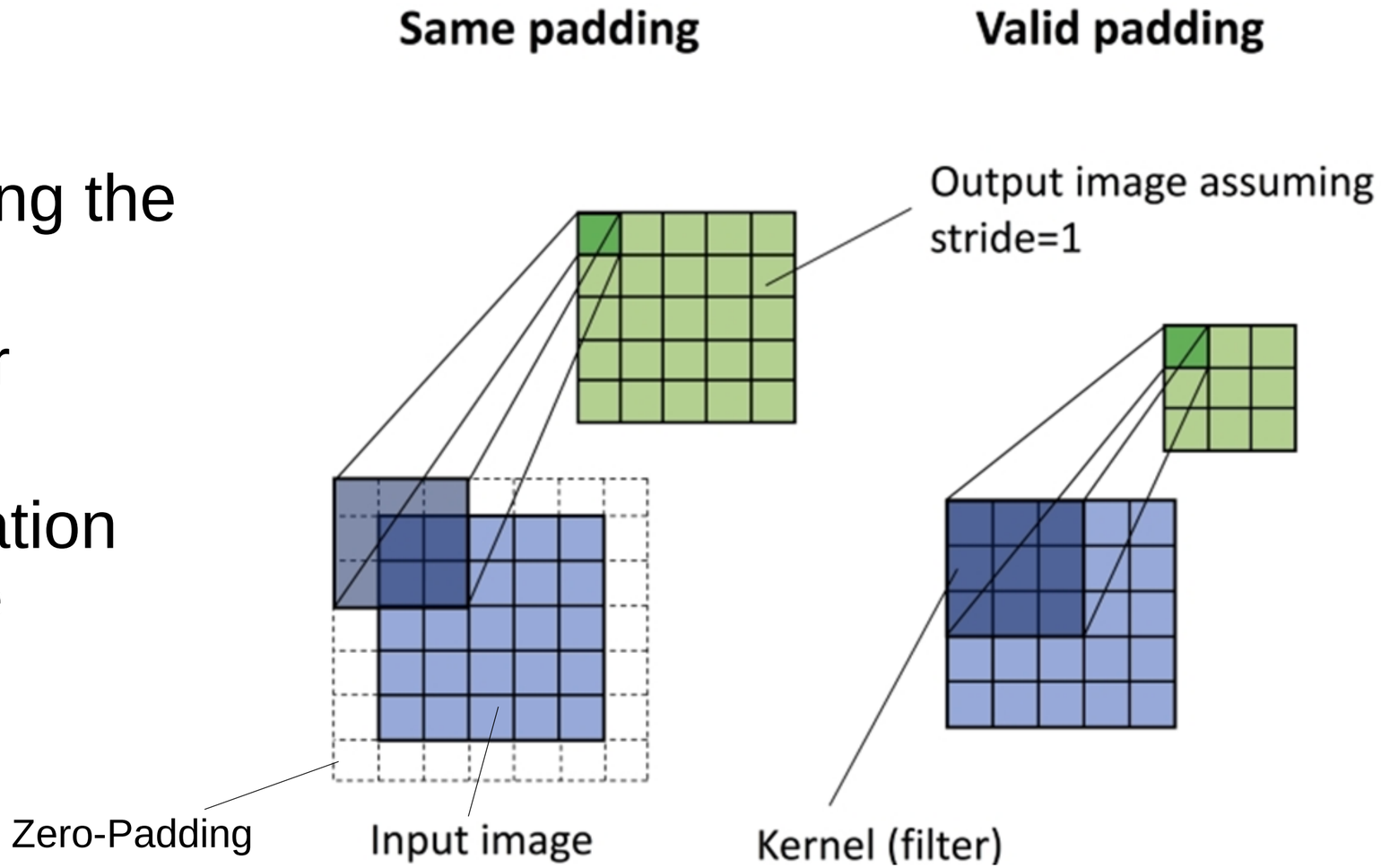
- Stride = 2
- Next Value:



Size of output feature map may decrease when we increase stride

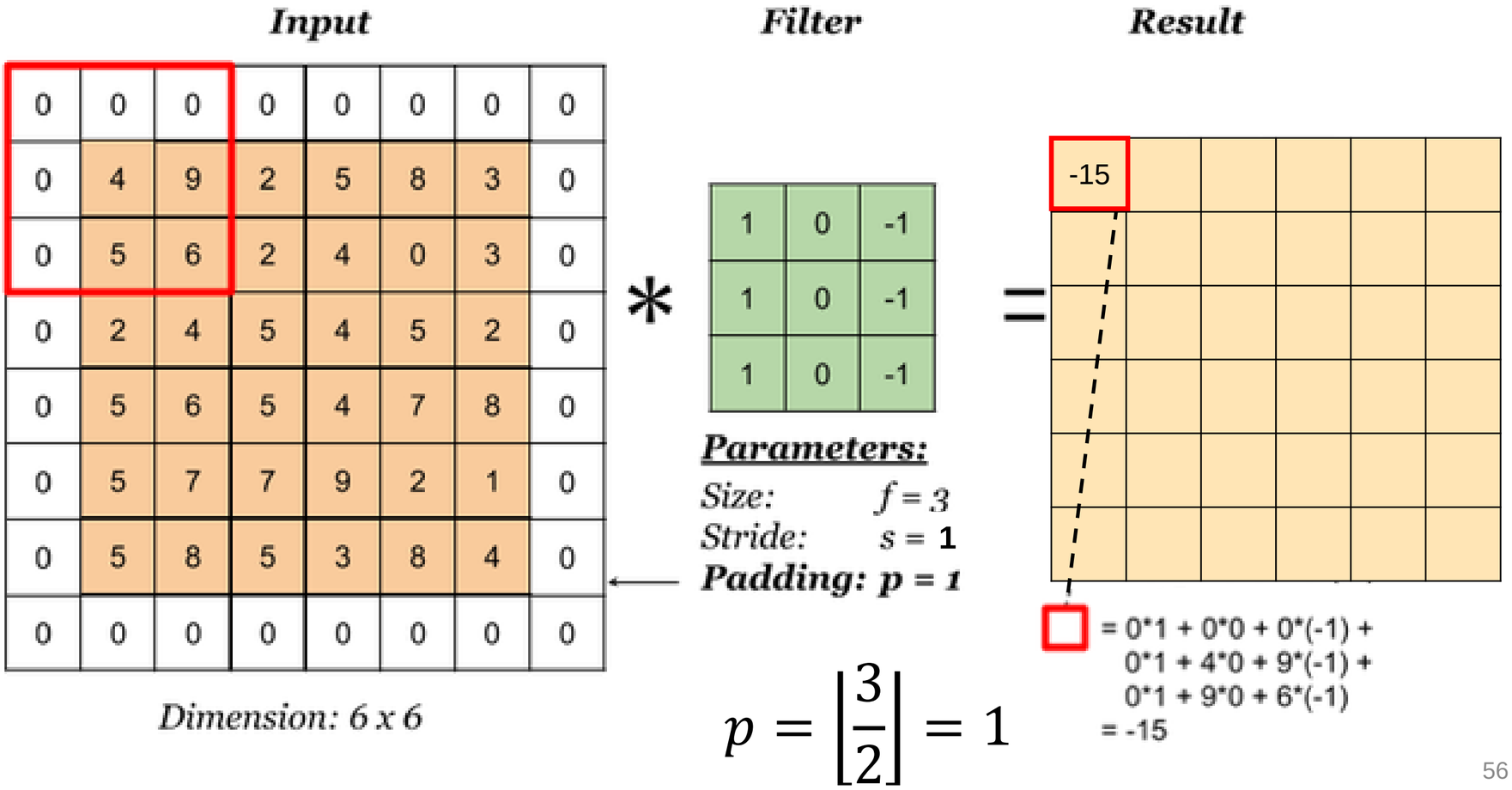
Padding

- Use Conv without shrinking the height and width
- Helpful in building deeper networks
- Keep more of the information at the border of an image



Same Padding

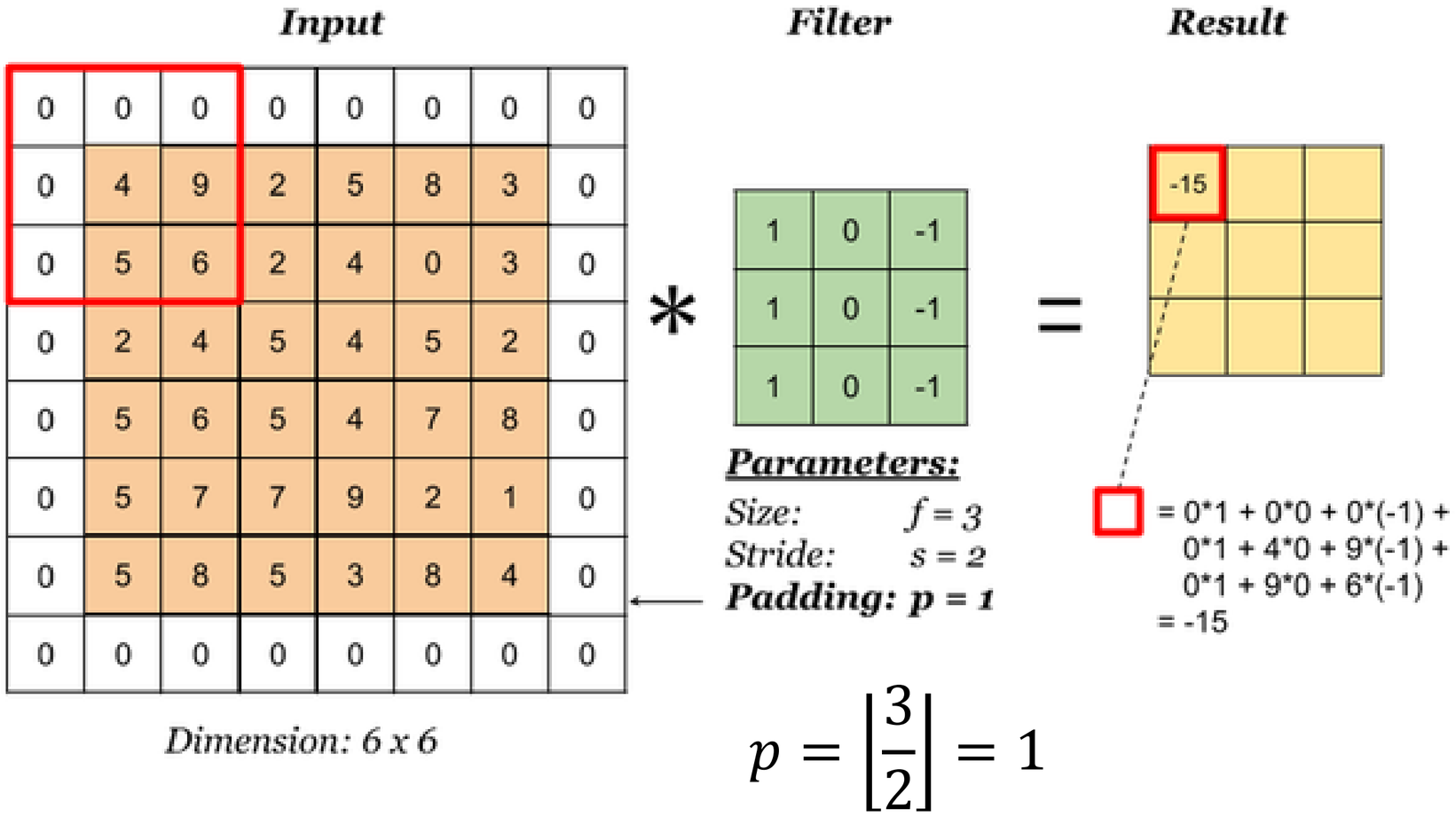
- Buffers the edge of the input with [filter_size/2] zeros (integer division)
- Output dimension is the same as the input for s=1



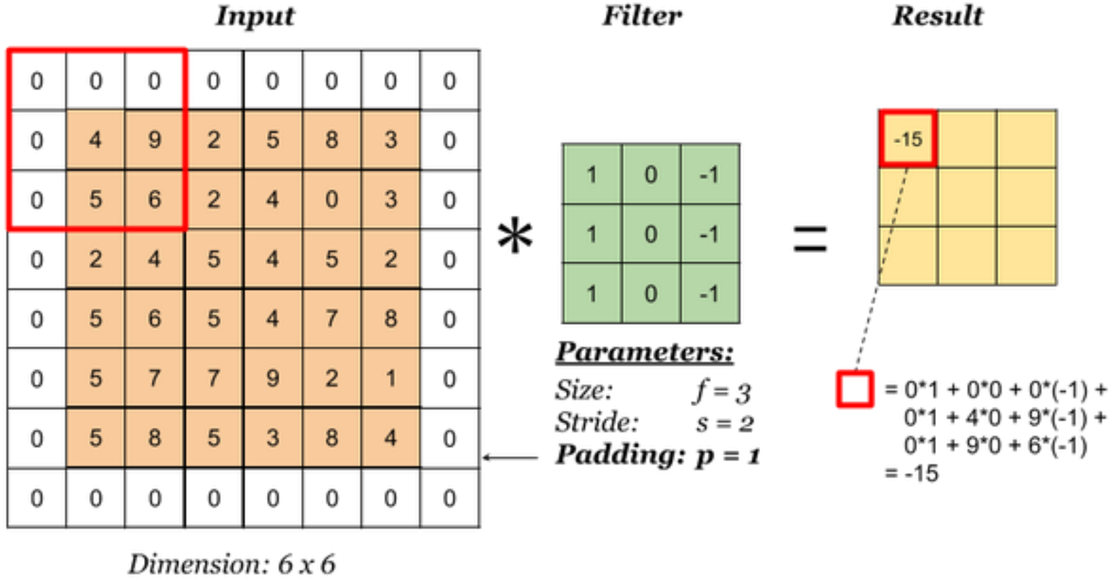
$$p = \left\lfloor \frac{3}{2} \right\rfloor = 1$$

Same Padding

- Buffers the edge of the input with [filter_size/2] zeros (integer division)
- Output dimension is the same as the input for s=1
- Output dimension reduces less for s>1



Output Dimension



$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}: number of input features

n_{out}: number of output features

k: convolution kernel size

p: convolution padding size

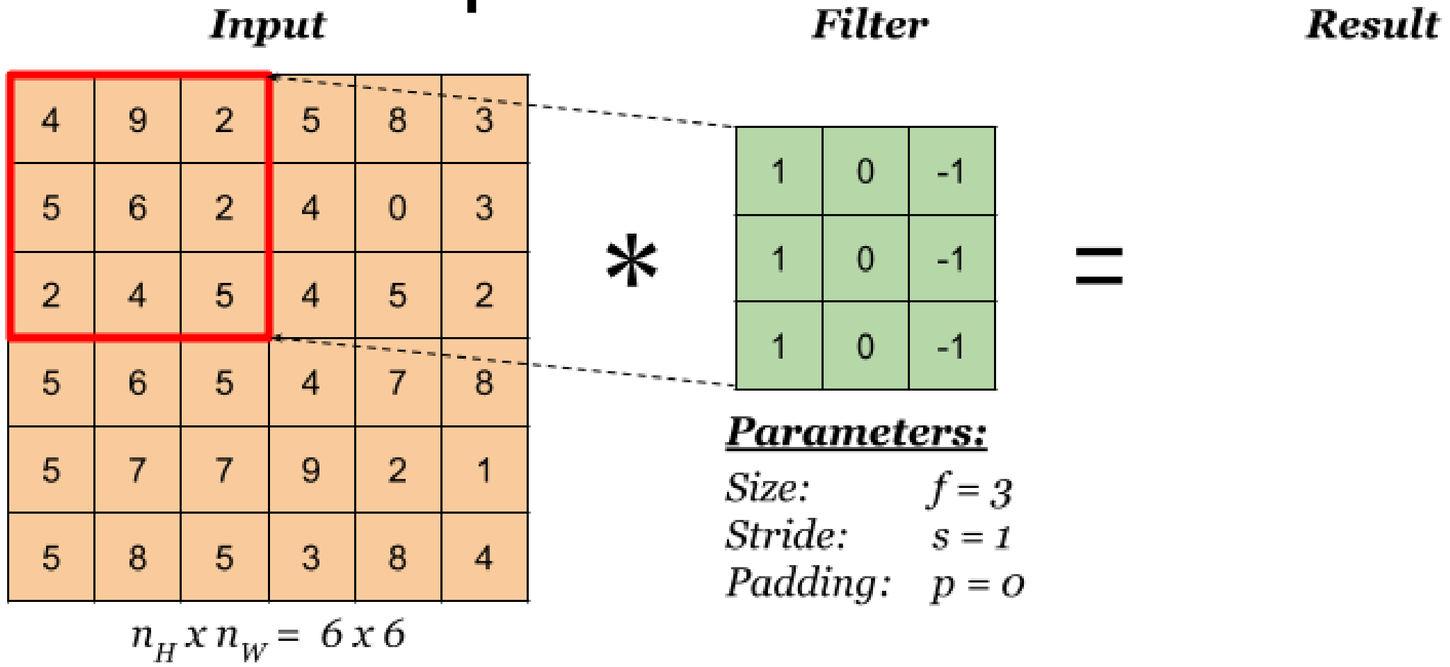
s: convolution stride size

- $n_{out} = \left\lfloor \frac{6+2*1-3}{2} \right\rfloor + 1 = \left\lfloor \frac{5}{2} \right\rfloor + 1$

$= 2 + 1 = 3$

Determine the Output Dimension

- $n_{out} = ?$

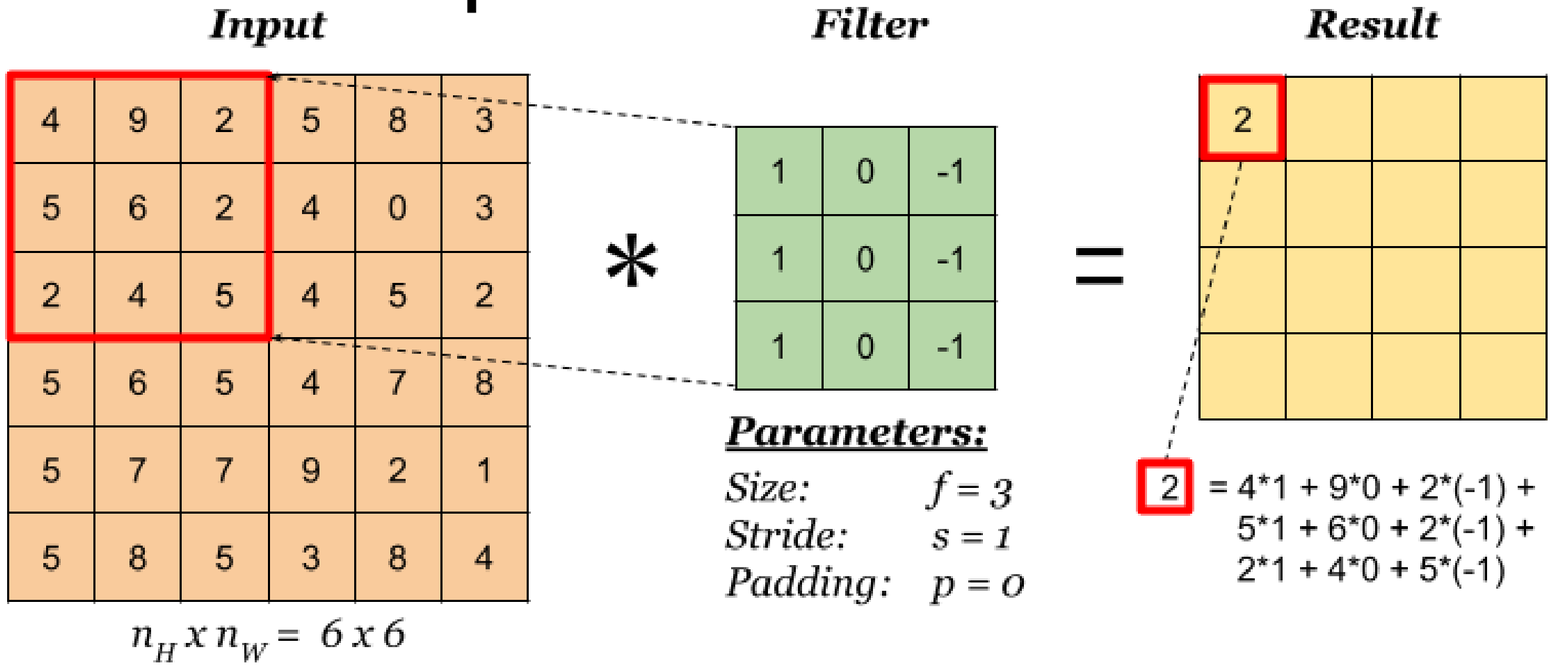


Determine the Output Dimension

- $n_{out} = \left\lfloor \frac{6-3+2*0}{1} \right\rfloor + 1$

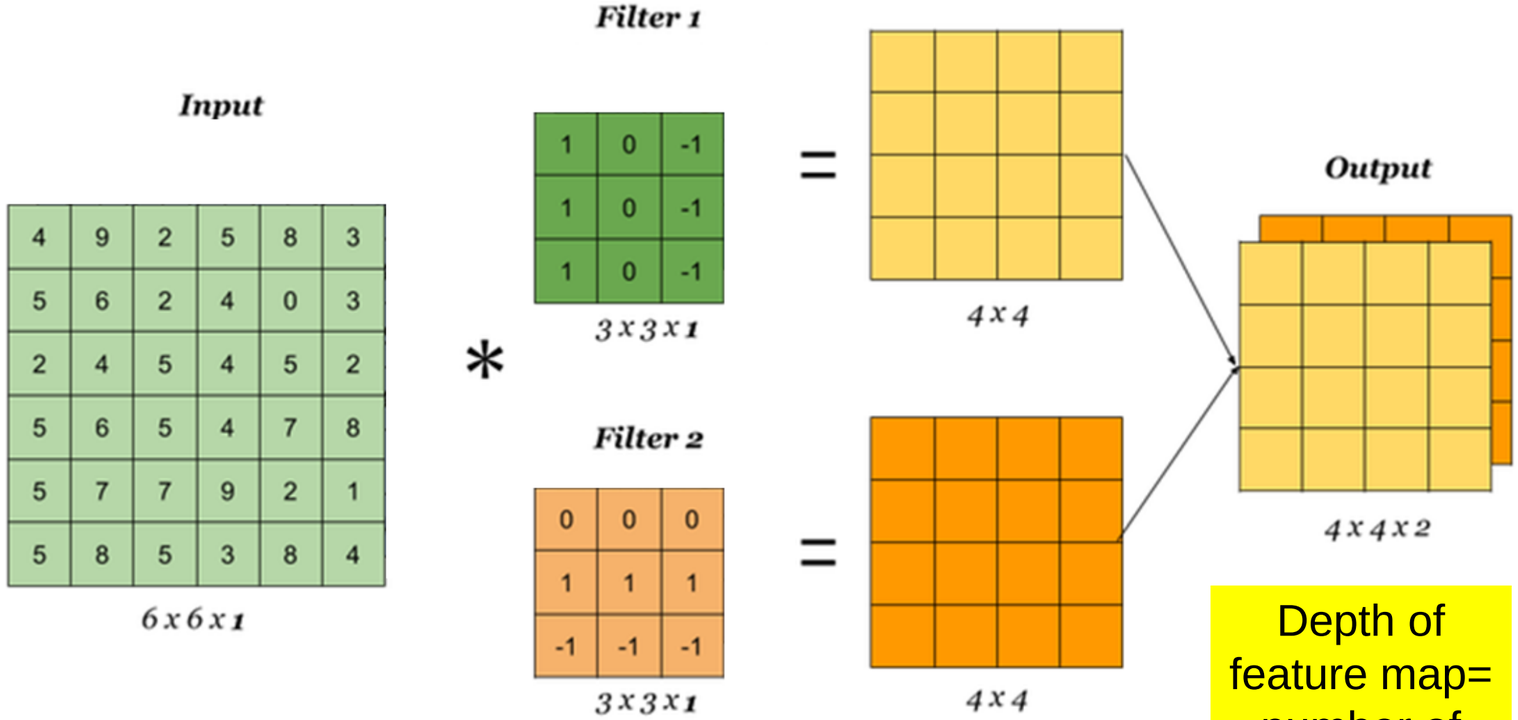
$= \left\lfloor \frac{3}{1} \right\rfloor + 1$

$= 4$



Convolution with Single Channel and Multiple Filters

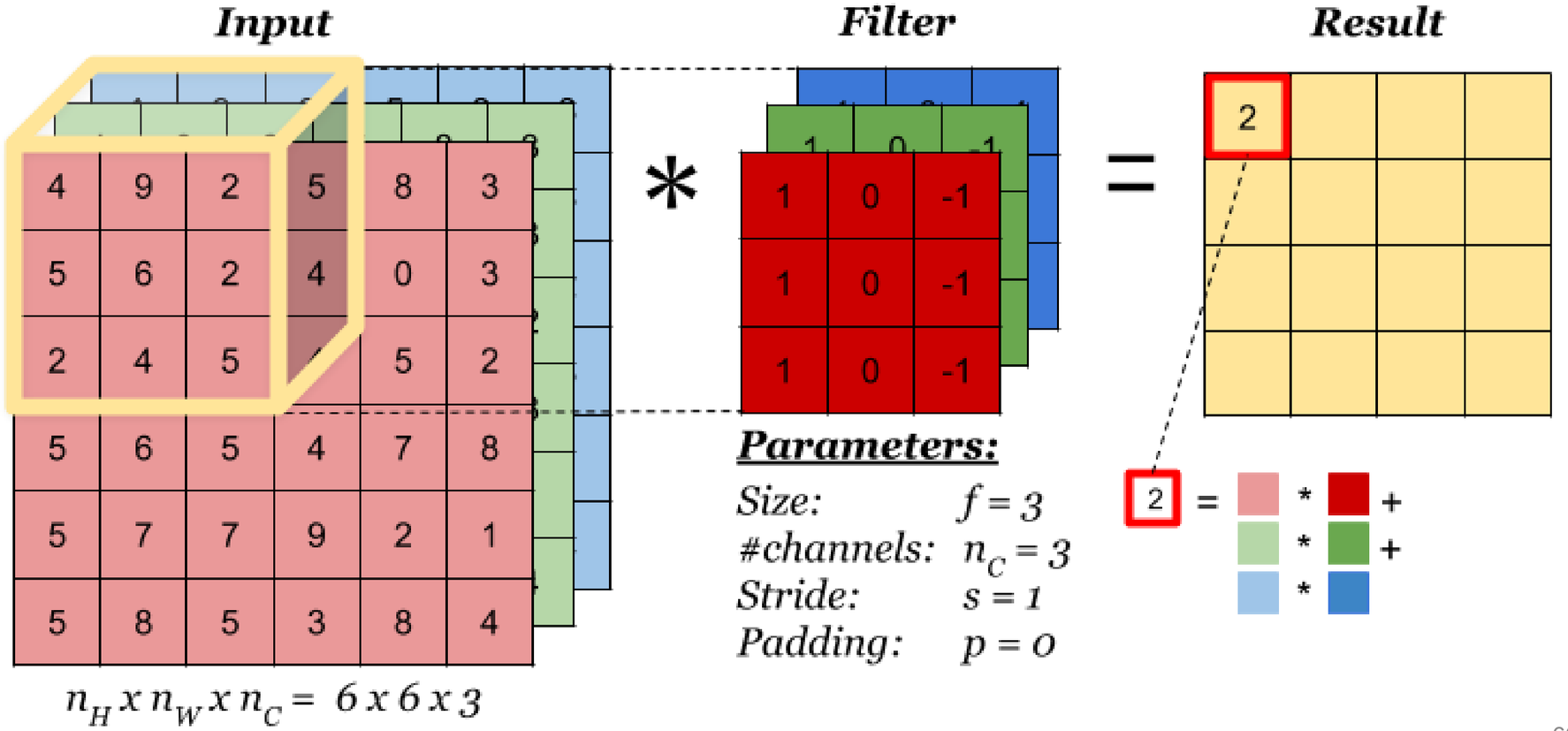
- Input with 1 channel (eg. grayscale image) then a 3×3 filter will be applied in 3x3x1 blocks



Convolution over Volume (Multiple Channels)

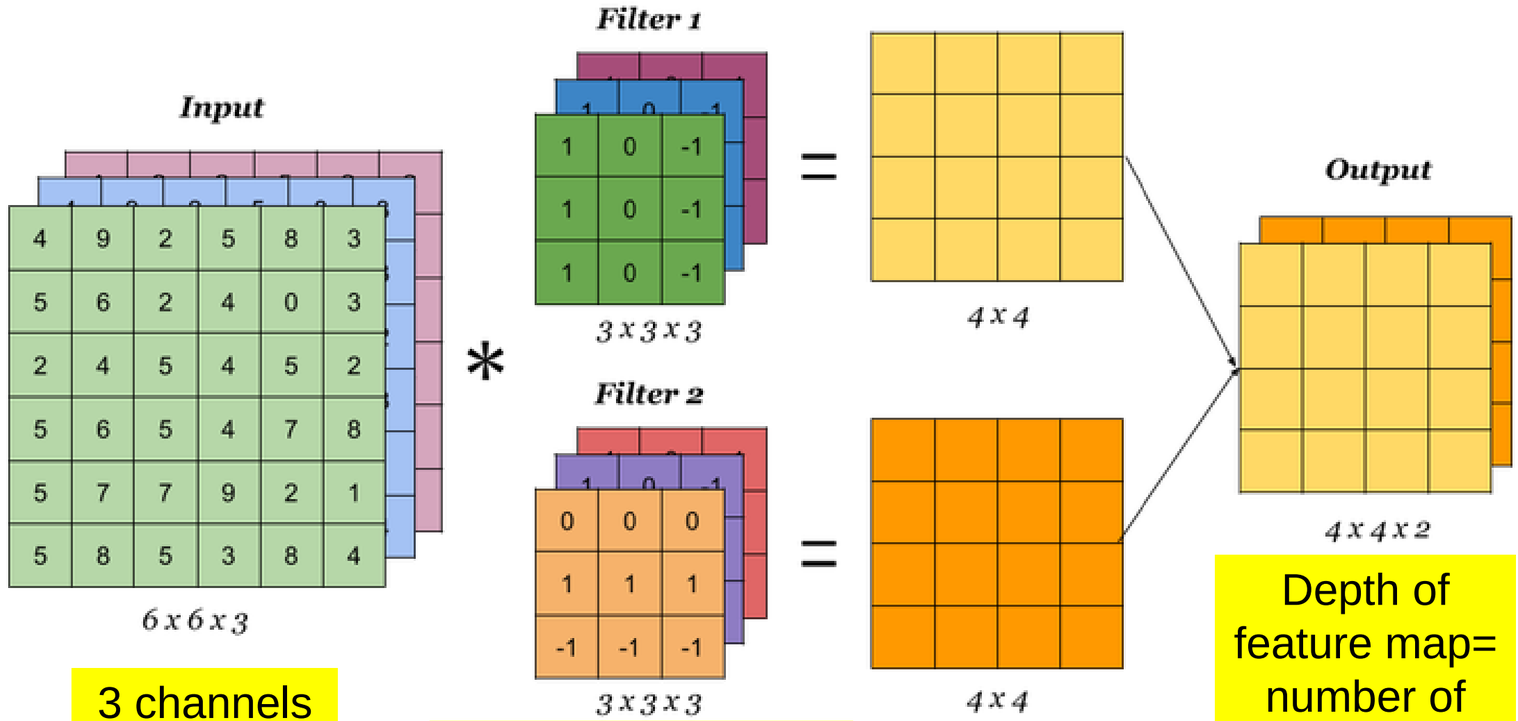
- RGB images has 3 channels:

  Red, Green, Blue
- One kernel for every input channel to the layer (each kernel is unique)
- Each filter = a *collection of kernels*



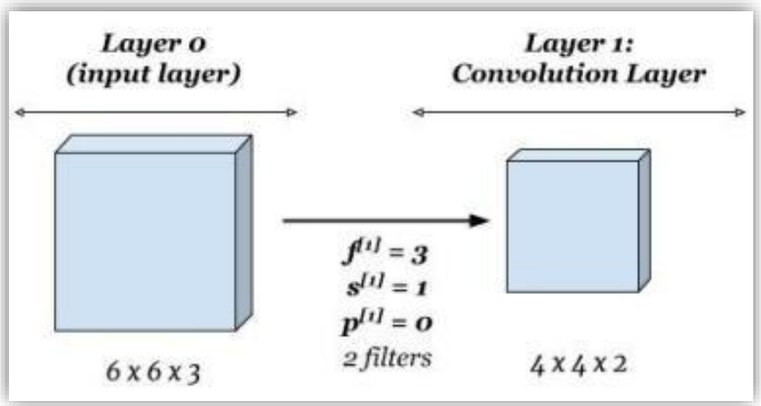
Convolution with Multiple Channels and Multiple Filters

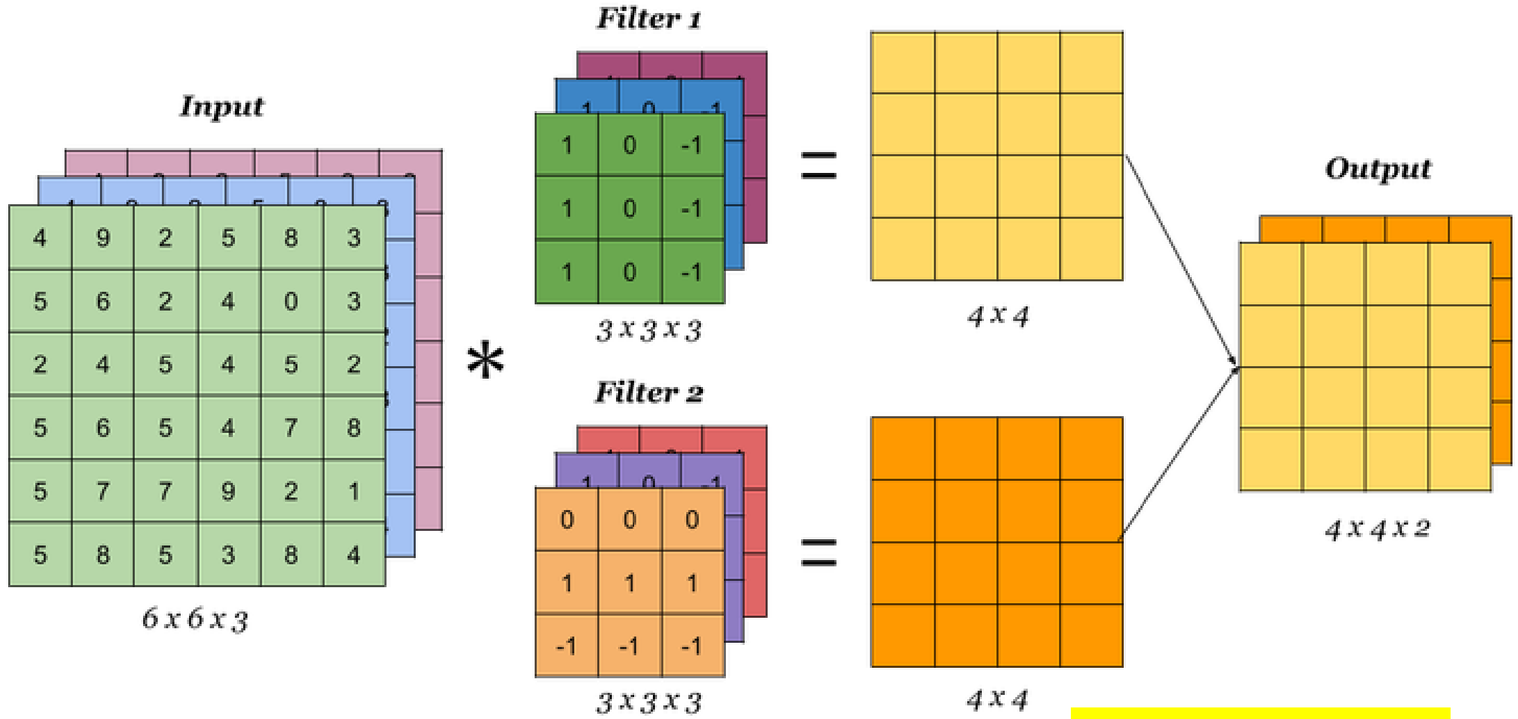
- Input with 1 channel (eg. grayscale image) then a 3×3 filter will be applied in 3x3x1 blocks
- Input with 3 channels (eg red, green, and blue for colour image), then a 3×3 filter will be applied in 3x3x3 blocks
- Input is block of feature maps from another convolutional layer with depth say 64, then the 3×3 filter will be applied in 3x3x64 blocks to create a single output feature map

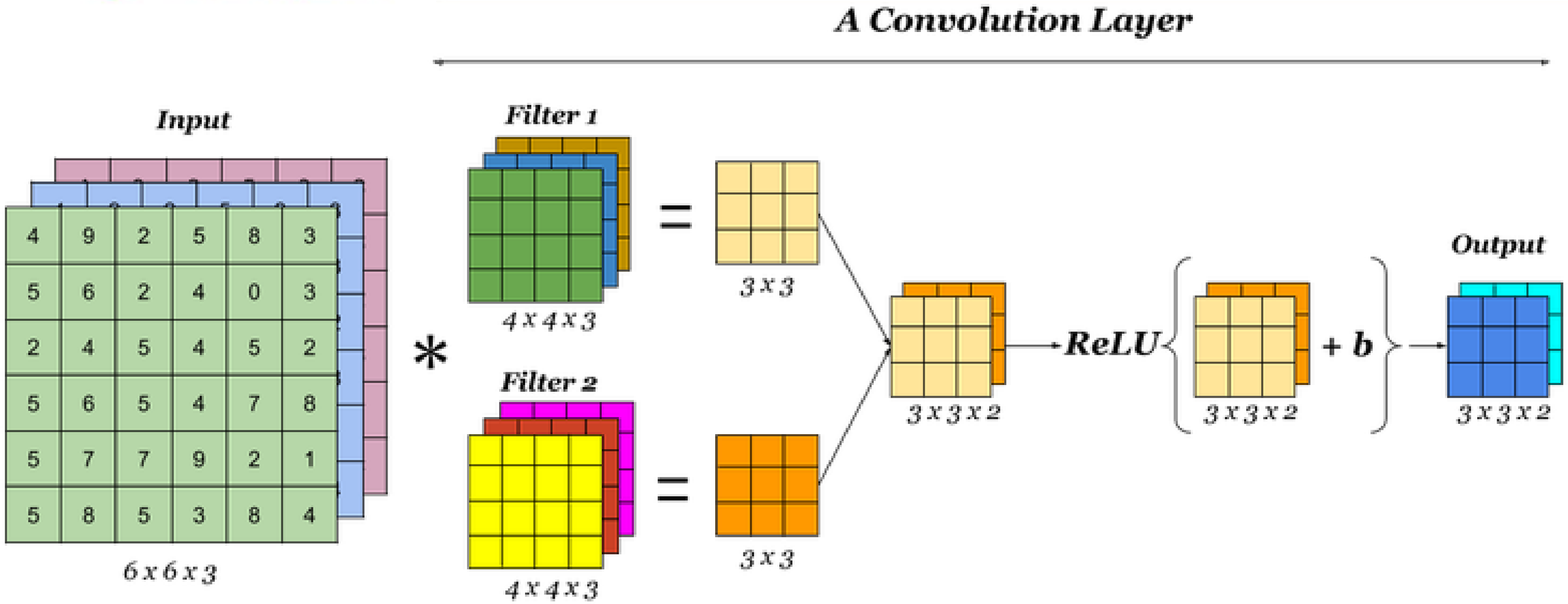


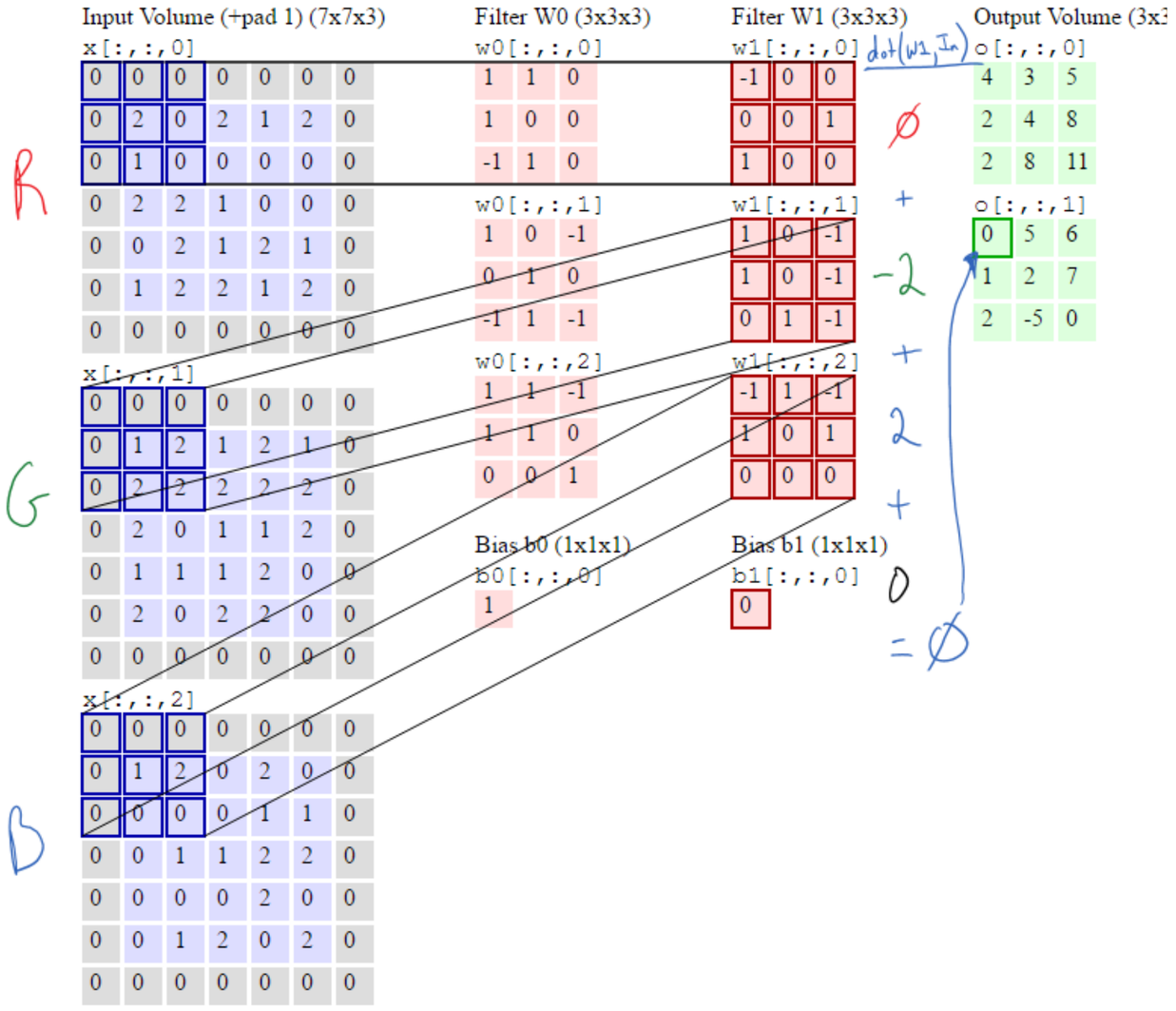
Convolution with Multiple Channels and Multiple Filters

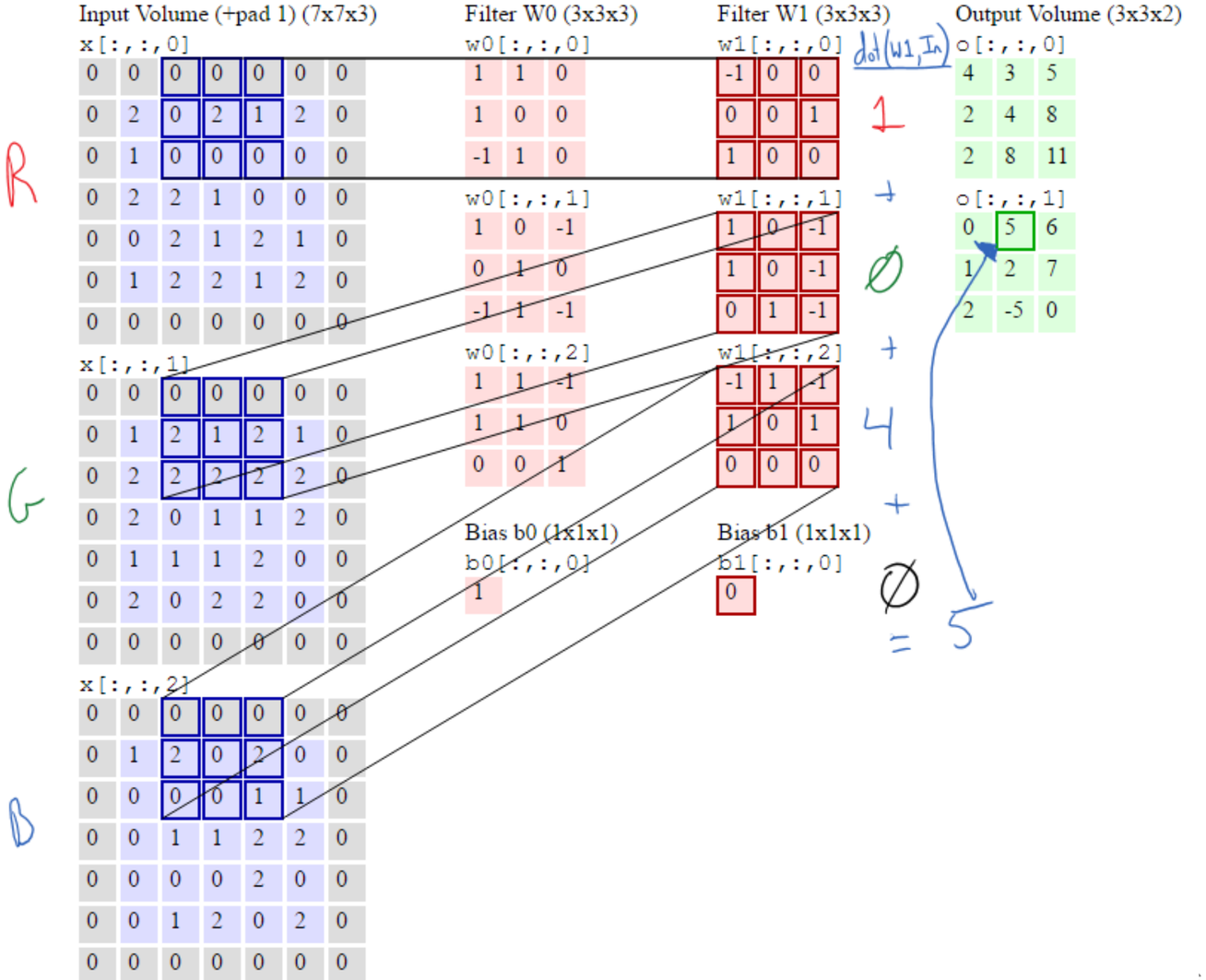
- Input feature maps (*axaxb*), on applying *d* filters,
- output feature map is (*cxcxd*)





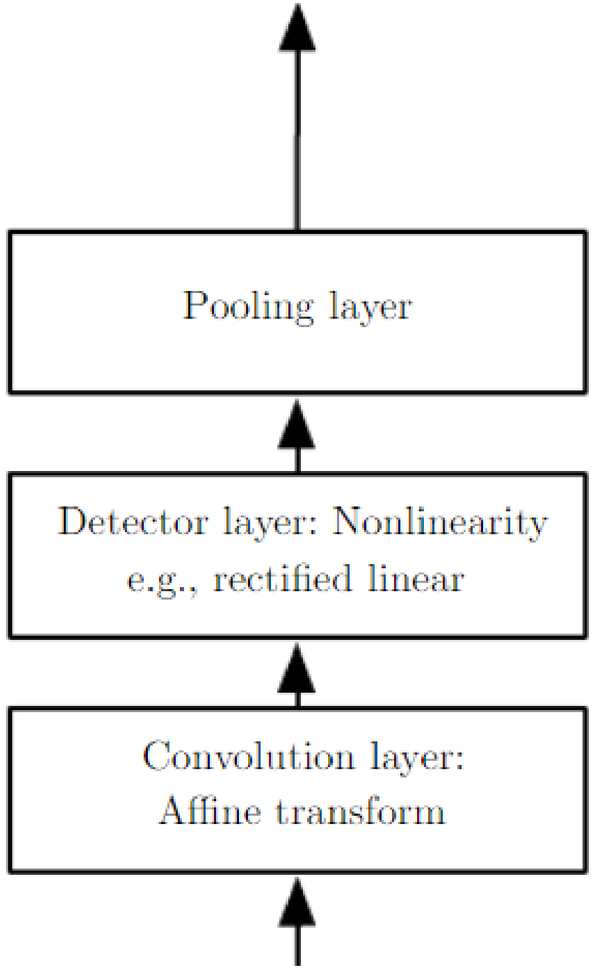

A Convolution Layer
Input
4 9 2 5 8 3
5 6 2 4 0 3
2 4 5 4 5 2
5 6 5 4 7 8
5 7 7 9 2 1
5 8 5 3 8 4
6 x 6 x 3
*
Filter 1
4 x 4 x 3
=
3 x 3
Filter 2
4 x 4 x 3
=
3 x 3
3 x 3 x 2
ReLU
3 x 3 x 2
+ b
Output
3 x 3 x 2


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
x[:,:,1]
x[:,:,2]
Filter W0 (3x3x3)
w0[:,:,0]
w0[:,:,1]
w0[:,:,2]
Filter W1 (3x3x3)
w1[:,:,0]
w1[:,:,1]
w1[:,:,2]
dot(W1, In)
Output Volume (3x3
o[:,:,0]
o[:,:,1]
Bias b0 (1x1x1)
b0[:,:,0]
Bias b1 (1x1x1)
b1[:,:,0]
R
G
B
∅
+
-2
+
2
+
0
=∅


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
x[:,:,1]
x[:,:,2]
Filter W0 (3x3x3)
w0[:,:,0]
w0[:,:,1]
w0[:,:,2]
Filter W1 (3x3x3)
w1[:,:,0]
w1[:,:,1]
w1[:,:,2]
dot(W1, In)
Output Volume (3x3x2)
o[:,:,0]
o[:,:,1]
Bias b0 (1x1x1)
b0[:,:,0]
Bias b1 (1x1x1)
b1[:,:,0]
R
G
B
1
+
∅
+
4
+
∅
= 5

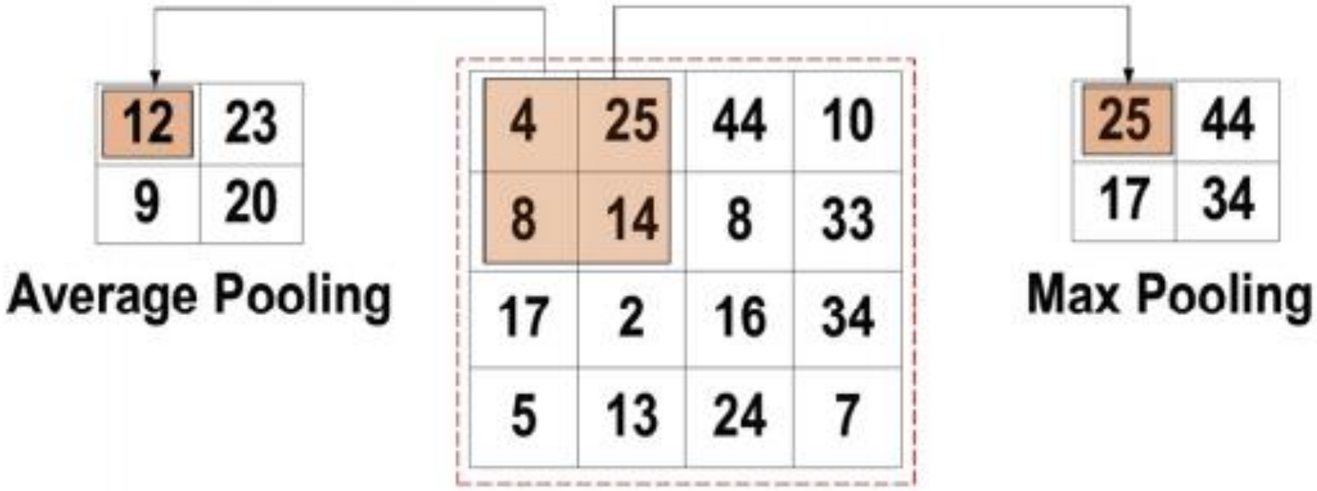
Typical Convolutional Layer

- Layer performs several convolutions in parallel to produce a set of linear activations
- each linear activation is run through a nonlinear activation function, such as the ReLU (detector stage)
- Use a pooling function to modify the output of the layer



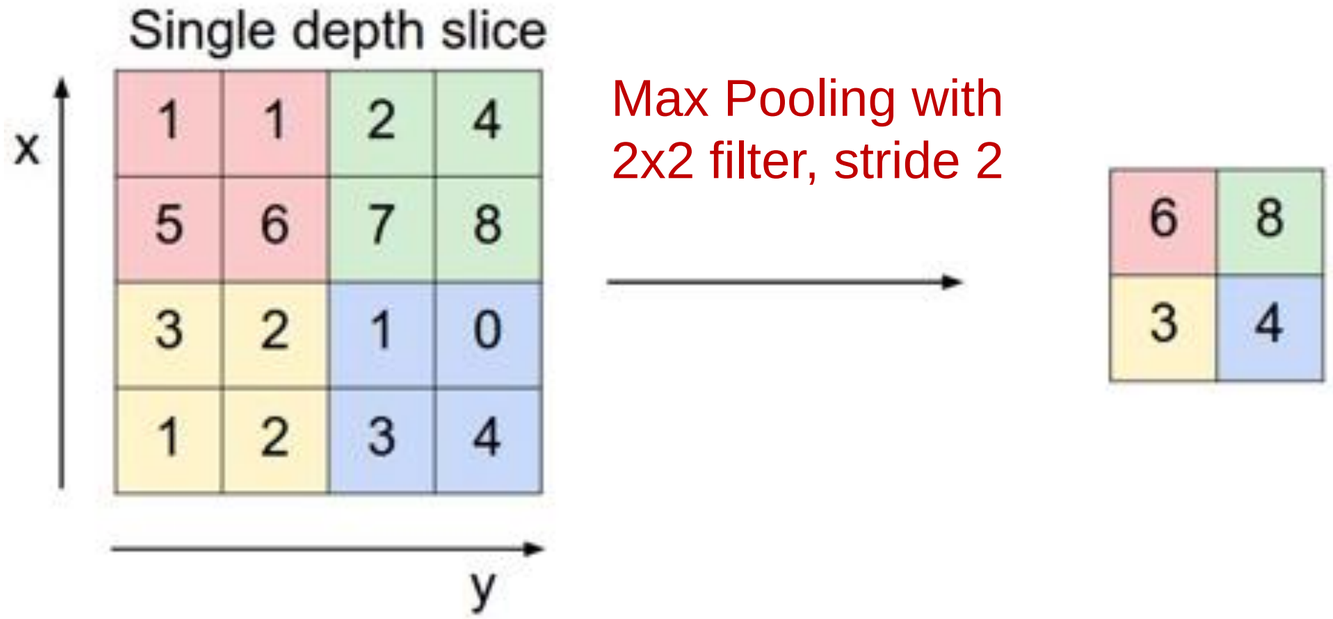
Pooling

- Replaces the output of at a certain location in the layer with a summary statistic of the nearby outputs
- Makes the representation approximately invariant to small translations of the input



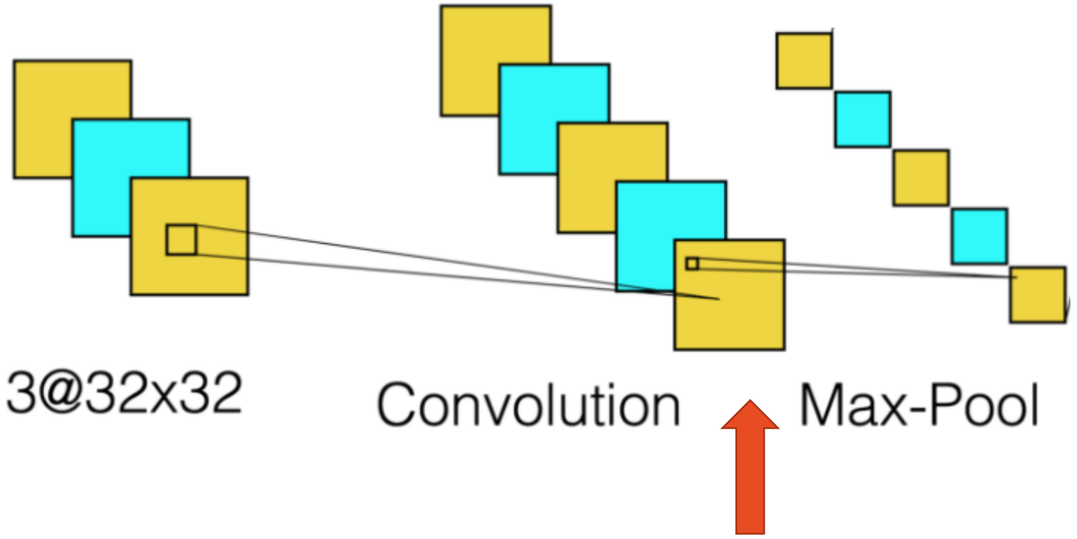
Pooling

- Pooling with stride *k* improves the computational efficiency of the network because the next layer has roughly *k* times fewer inputs to process



Determine the Output Dimension

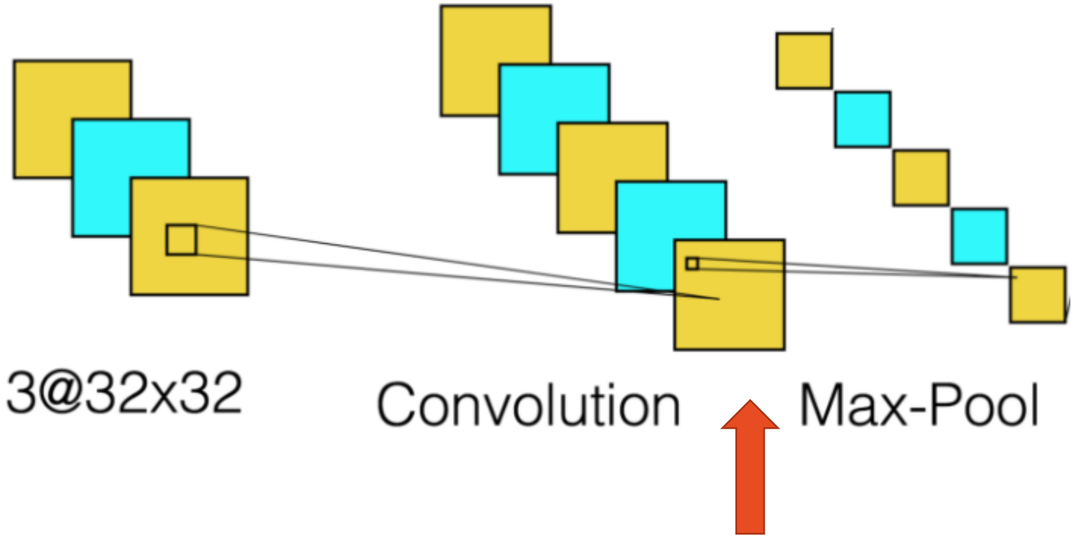
- Input image, I with dimensions (32x32x3)
- Convolution Layer
  - A filter size 3x3
  - Stride is 1
  - Valid padding, and
  - Depth/feature maps are 5 (D =5)
- **Output dimensions = ?**



$$n_{out} =$$

Determine the Output Dimension

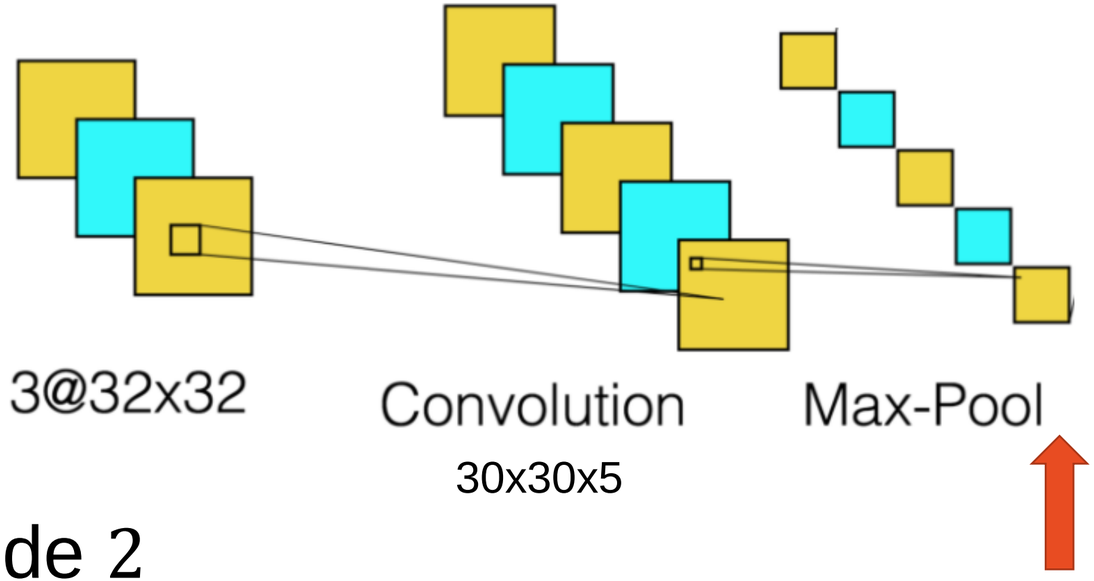
- Input image, I with dimensions (32x32x3)
- Convolution Layer
  - A filter size 3x3
  - Stride is 1 (s=1)
  - Valid padding (p=0), and
  - Depth/feature maps are 5 (D =5)
- **Output dimensions =** 30x30x5
- **After Pooling**?



$$n_{out} = \left\lfloor \frac{32 - 3 + 2 * 0}{1} \right\rfloor + 1 = \left\lfloor \frac{29}{1} \right\rfloor + 1 = 30$$

Determine the Output Dimension

- Input to Pooling Layer (30x30x5)
- After Pooling with
  - Filter size $k \times k$
  - Stride s
- $n_{out} = \left\lfloor \frac{n_{in} - k}{s} \right\rfloor + 1$

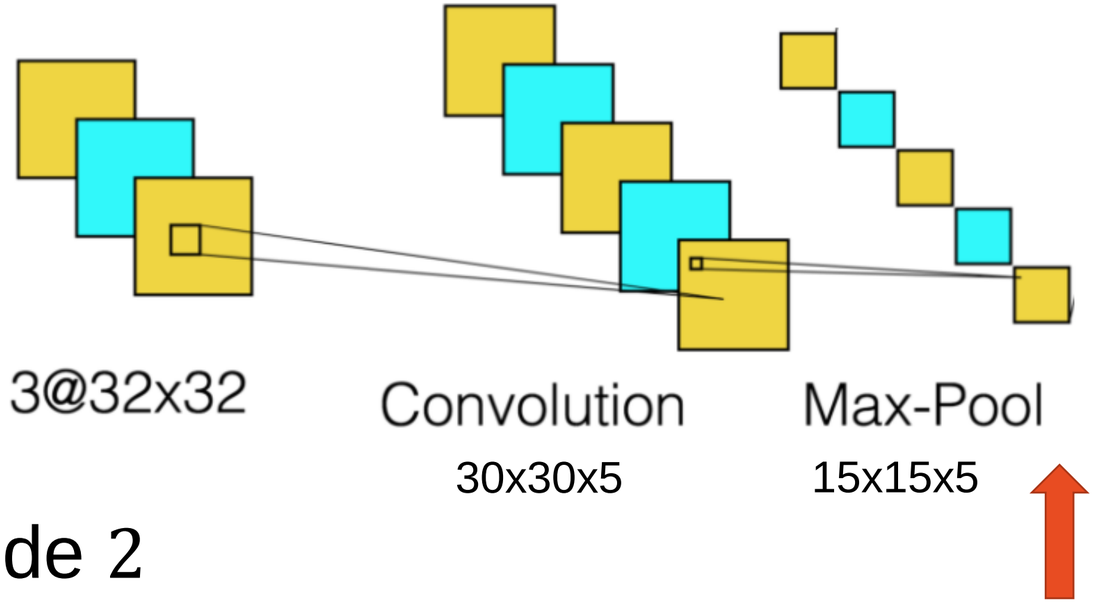


- **Eg,** Pooling with, Filter size 2×2, Stride 2
- Output dimensions =

$n_{out} =$

Determine the Output Dimension

- Input to Pooling Layer (30x30x5)
- After Pooling with
  - Filter size $k \times k$
  - Stride s
- $n_{out} = \left\lfloor \frac{n_{in} - k}{s} \right\rfloor + 1$



- **Eg,** Pooling with, Filter size 2×2, Stride 2
- Output dimensions =15x15x5

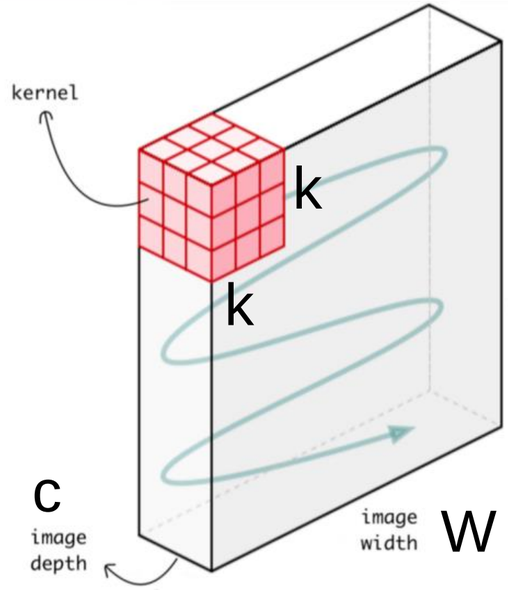
$$n_{out} = \left\lfloor \frac{30 - 2}{2} \right\rfloor + 1 = \left\lfloor \frac{28}{2} \right\rfloor + 1 = 15$$

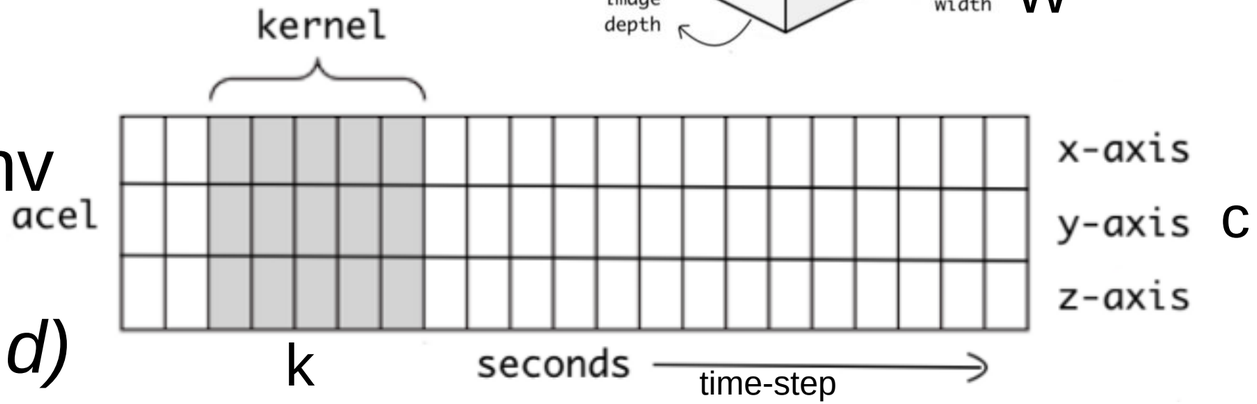
1D, 2D Convolution

- 2D Convolution
  - 2-directions (x,y) to calculate conv
  - input = (WxHxc), *d* filters (kxkxc)
    output = (W1xH1x*d*)
  - Eg: Image data (gray or color)
- 1D Convolution
  - 1-direction (time) to calculate conv
  - input = (time-step x c),

d filters (k x c) , output (time-step1 x *d)*

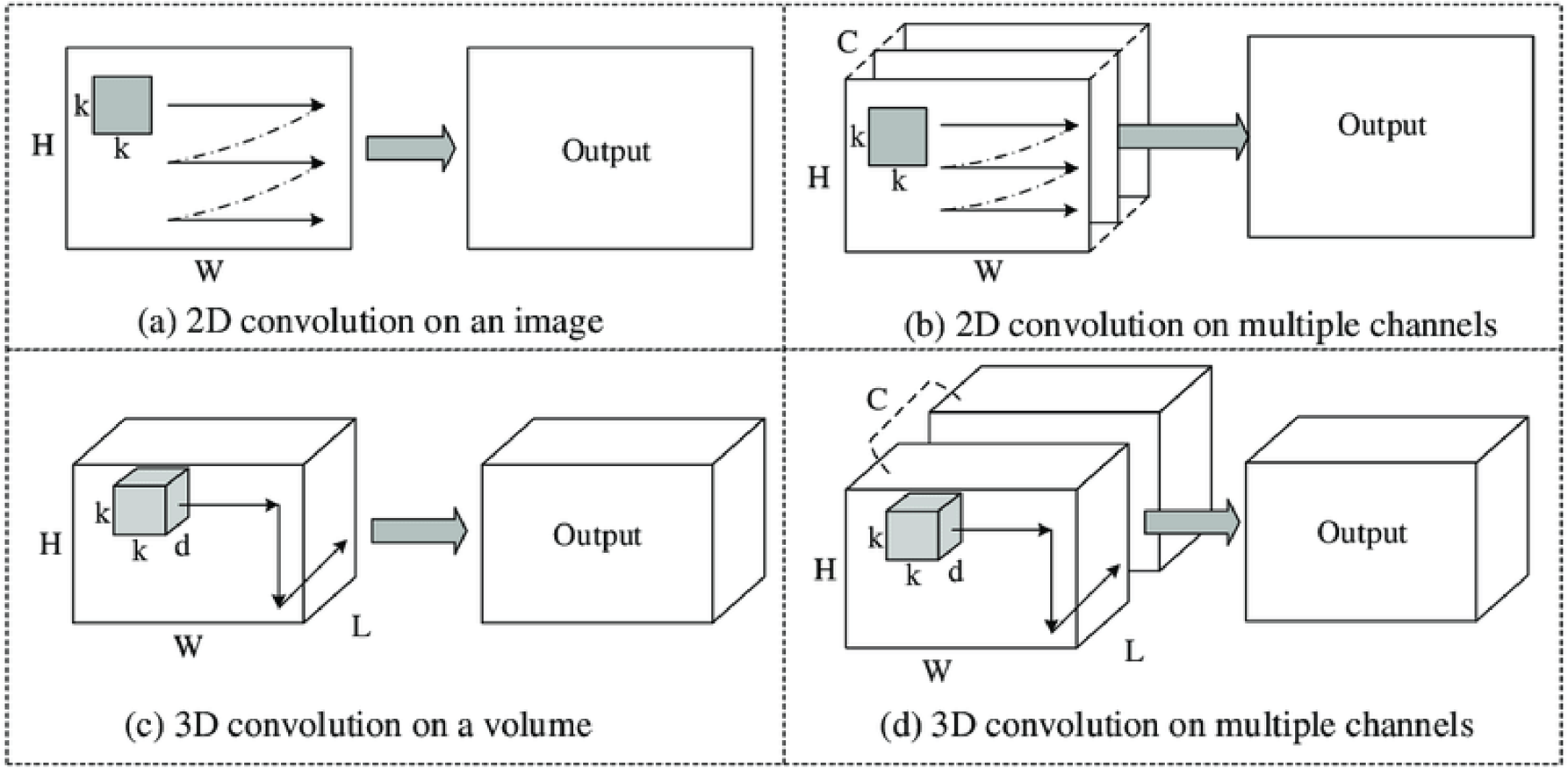
Eg: Time-series data, text analysis





2D, 3D Convolution

- 3D Convolution
  - 3-directions (x,y,z) to calculate conv
  - input (WxHxLxC), *m* filters (kxkxd) output (W1xH1xL1xm)
  - Eg: MRI data, Videos

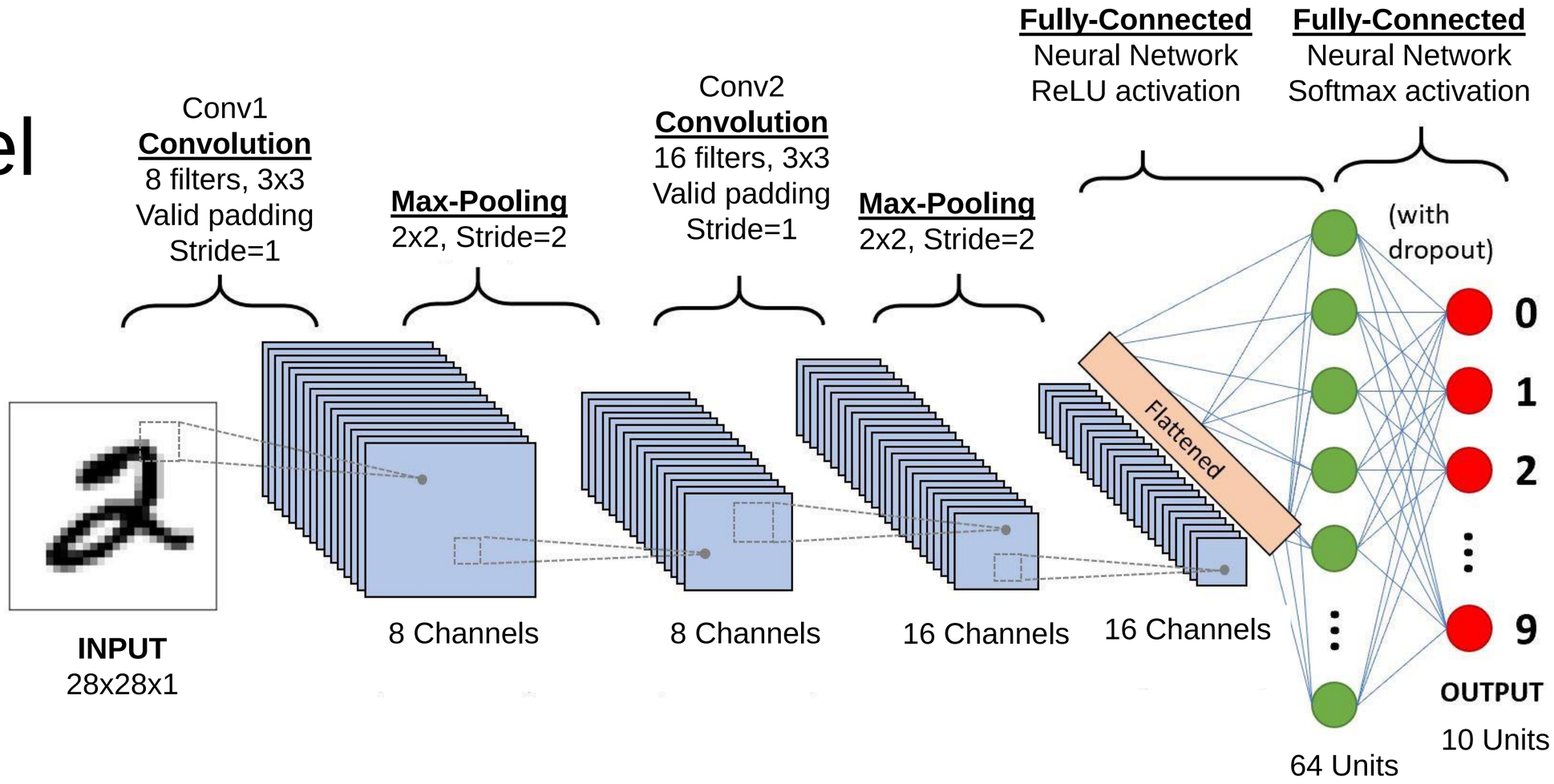


Typical CNN Model

- Conv1

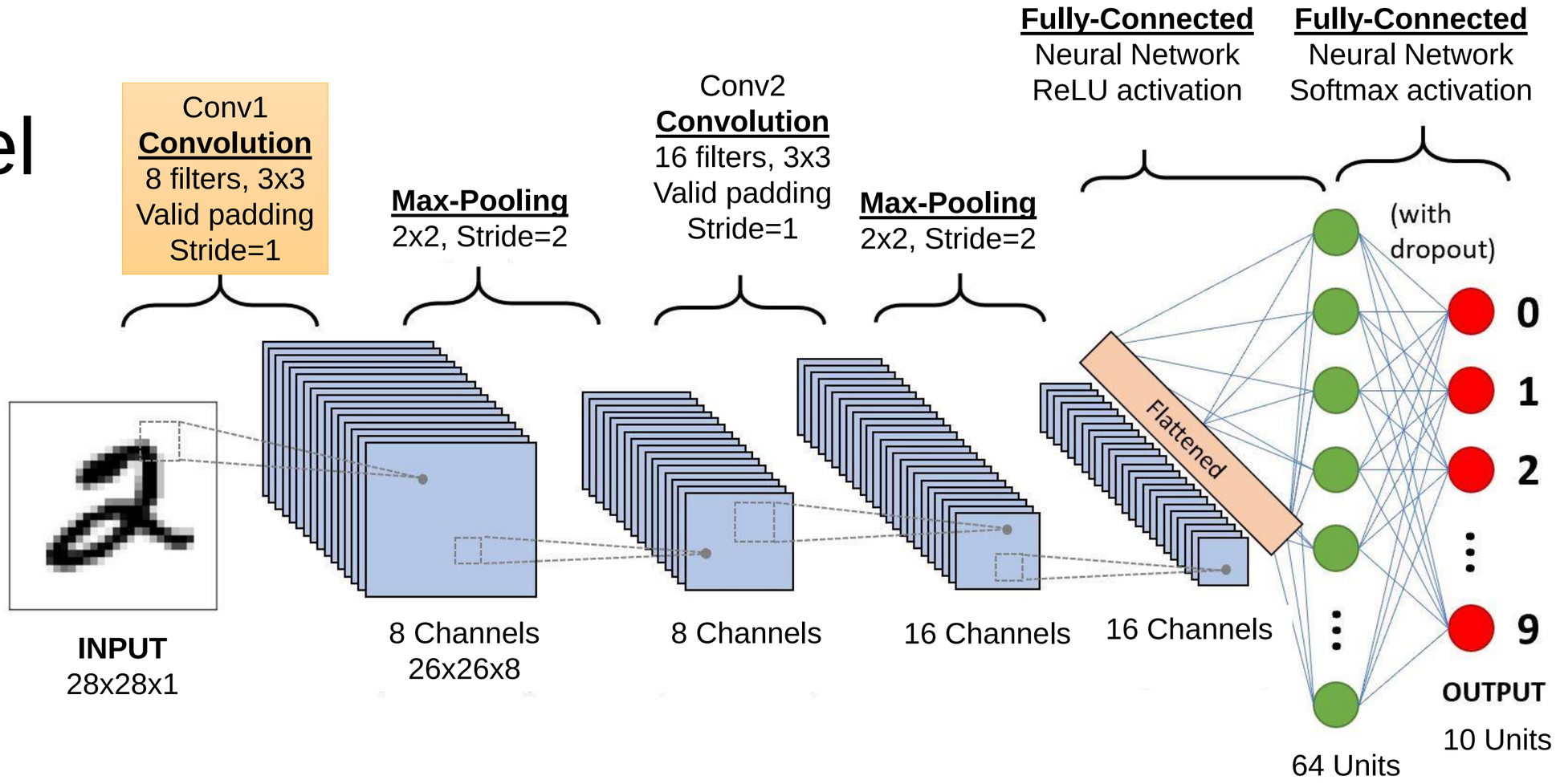
$$n_{out} = \left\lfloor \frac{n_{in} - k + 2*p}{s} \right\rfloor + 1 = ?$$

- o/p:



Typical CNN Model

- Conv1 $\left\lfloor \frac{28-3+2*0}{1} \right\rfloor + 1 = 26$
- o/p: $\mathbf{26 \times 26 \times 8}$
- Param: 3x3x1x8+8 = 80
- 3x3 filter for 1 channel, 8 such filters and 8 biases



Typical CNN Model

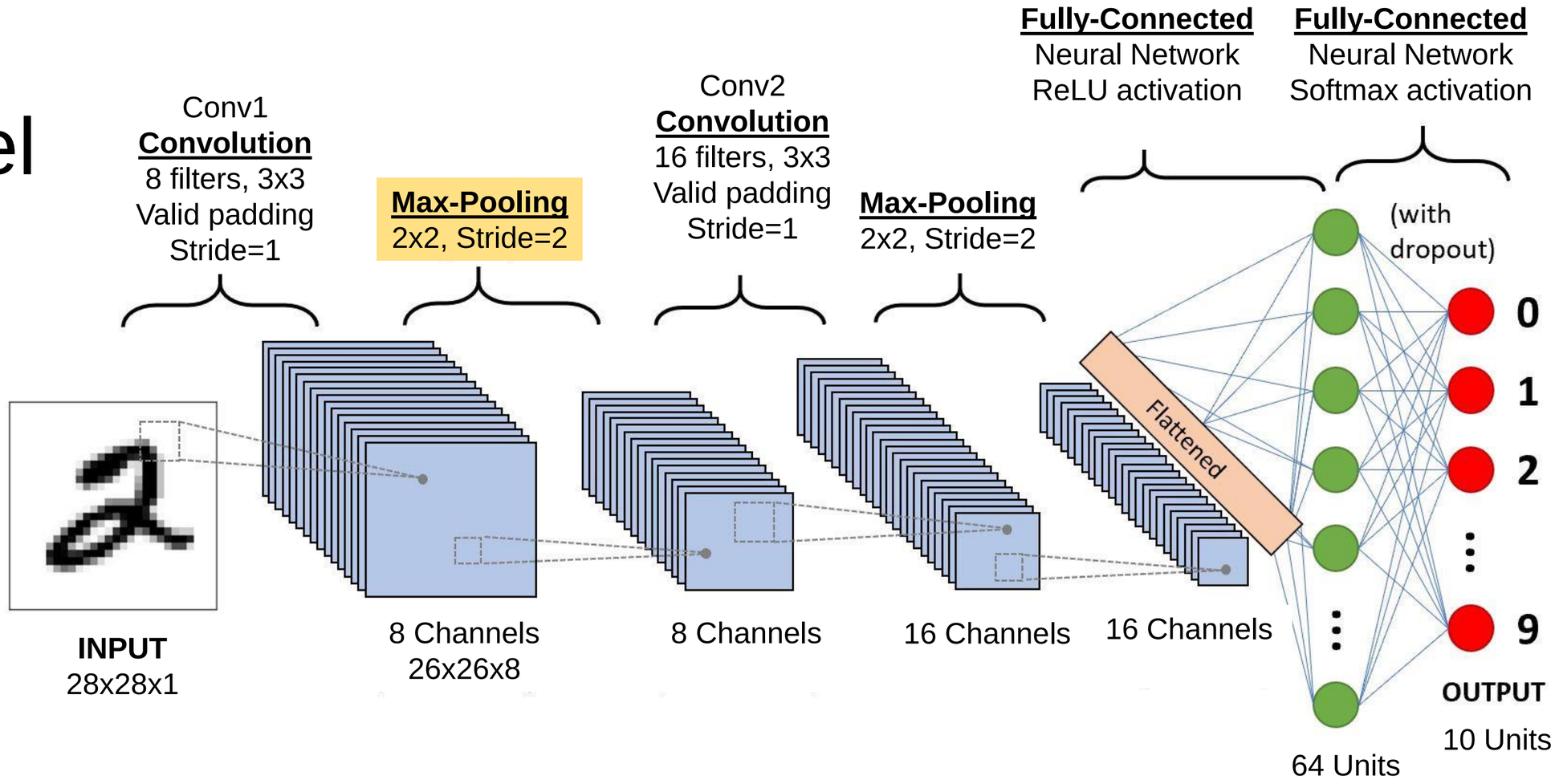
- Conv1:

  $26 \times 26 \times 8$

- Max-Pool

$$\left\lfloor \frac{n_{in} - k}{s} \right\rfloor + 1 \; =?$$

o/p: ?



Typical CNN Model

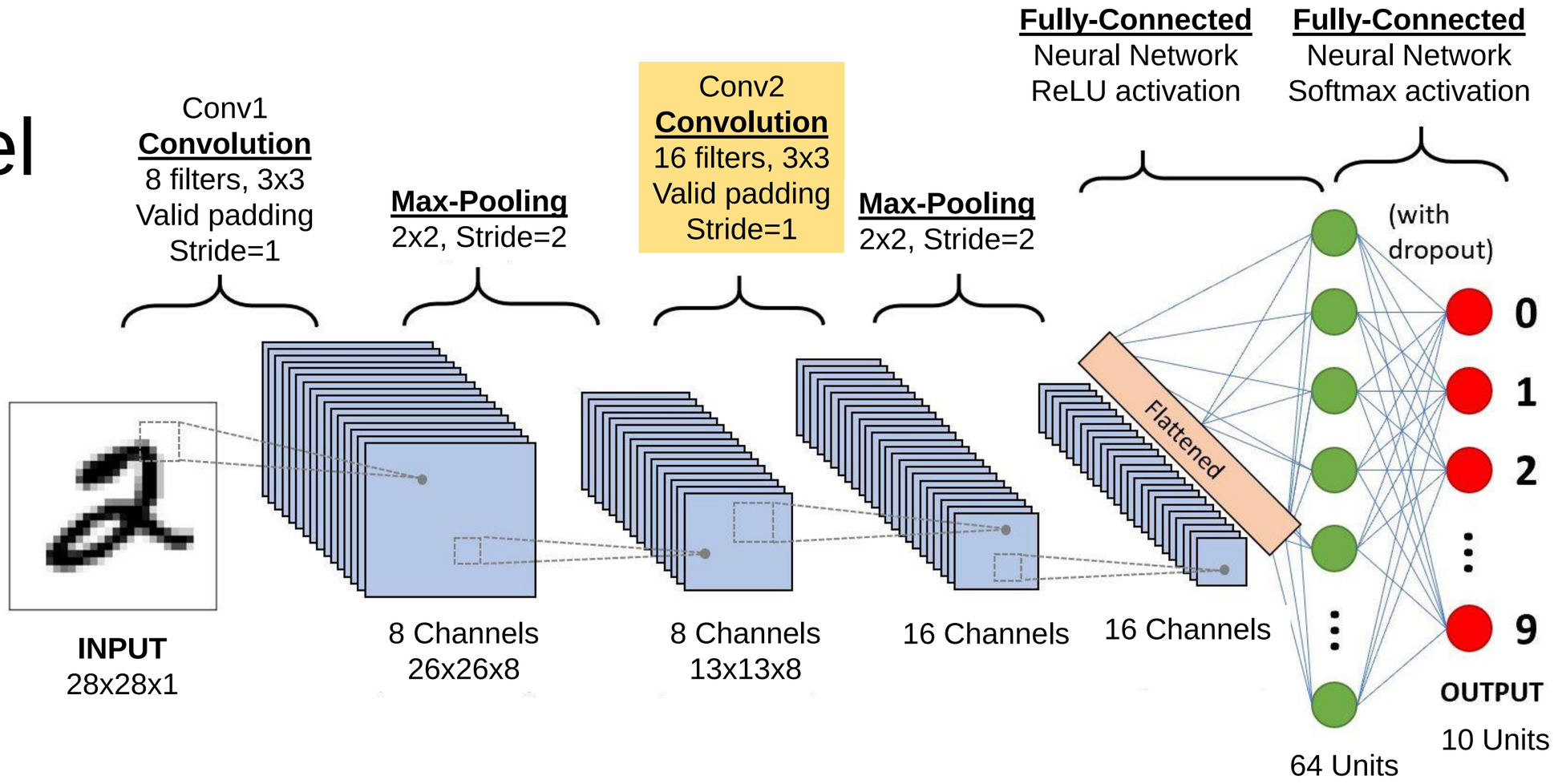
- Conv1:

  $26 \times 26 \times 8$

- Max-Pool
- $\left\lfloor \frac{26-2}{2} \right\rfloor + 1 = 13$

  o/p: $13 \times 13 \times 8$

- Conv2, $n_{out} = \left\lfloor \frac{n_{in}-k+2*p}{s} \right\rfloor + 1$



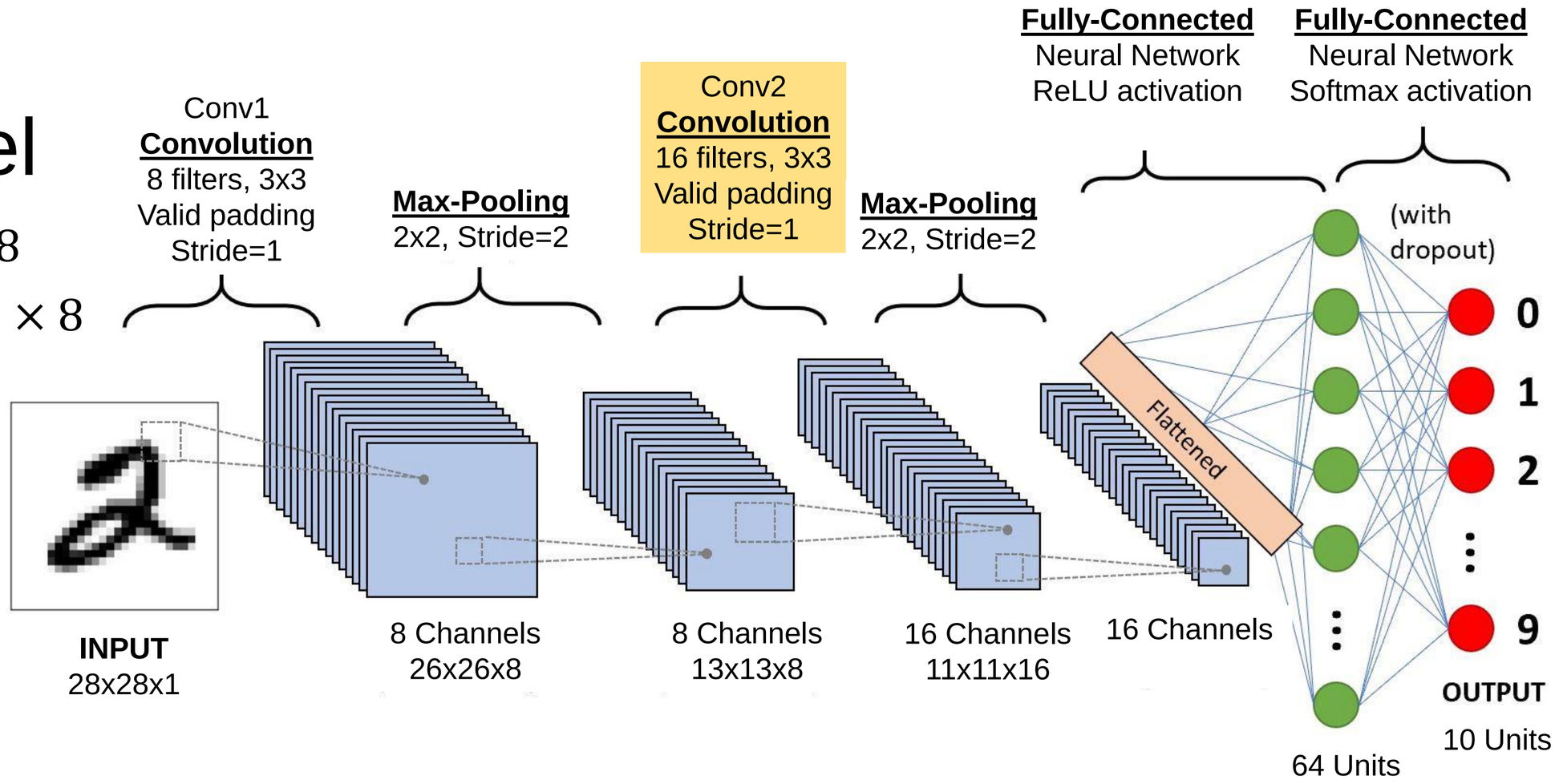
Typical CNN Model

- Conv1: $26 \times 26 \times 8$
- Max-Pool: $13 \times 13 \times 8$
- Conv2:

$$\left\lfloor \frac{13-3+2*0}{1} \right\rfloor + 1 = 11$$

o/p: $11 \times 11 \times 16$

- Param?



Typical CNN Model

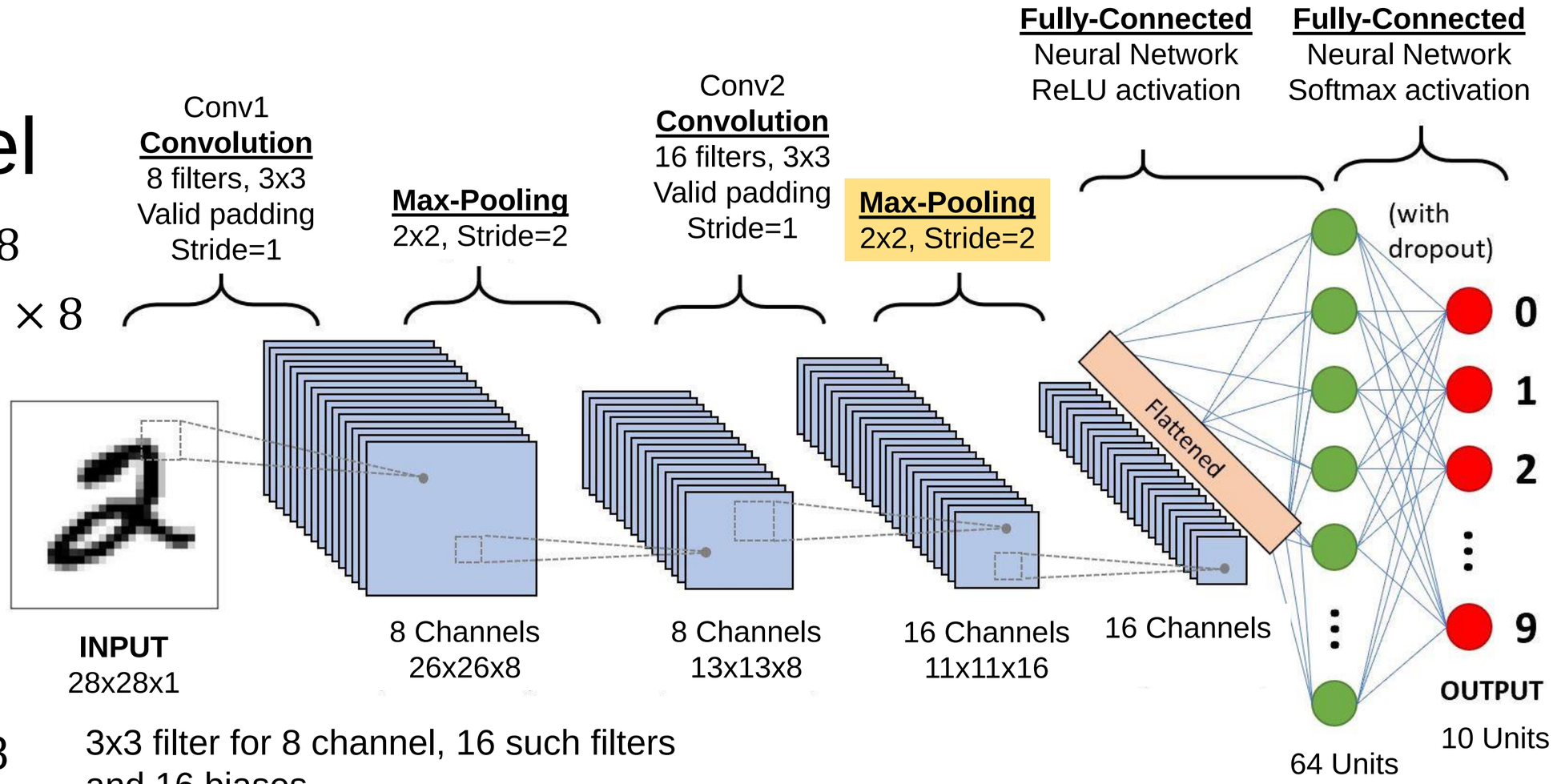
- Conv1: $26 \times 26 \times 8$
- Max-Pool: $13 \times 13 \times 8$
- Conv2:

$$\left\lfloor \frac{13-3+2*0}{1} \right\rfloor + 1 = 11$$

o/p: $11 \times 11 \times 16$

- Param=

3x3x8x 16+16=1168



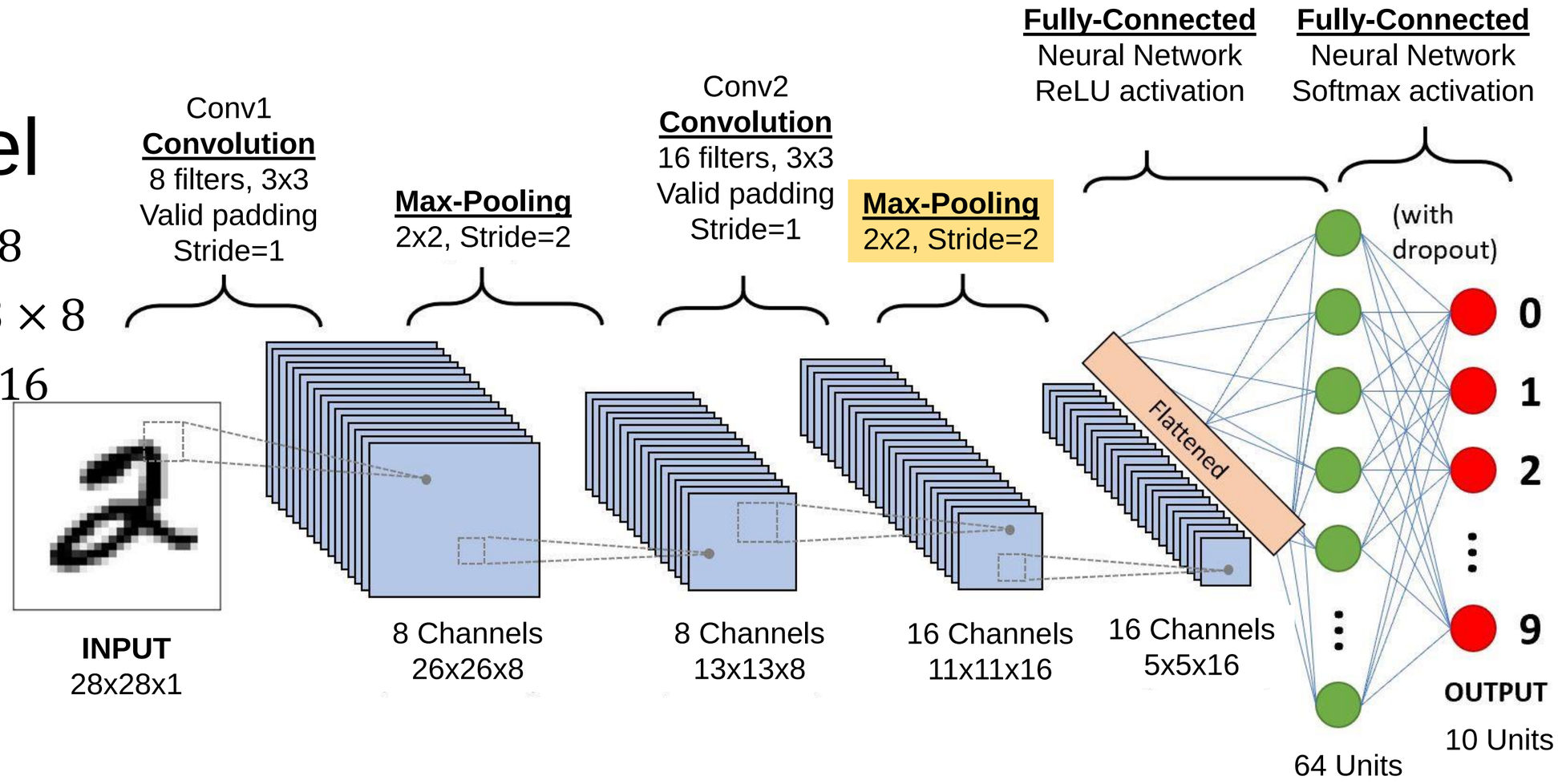
3x3 filter for 8 channel, 16 such filters and 16 biases

Typical CNN Model

- Conv1: $26 \times 26 \times 8$
- Max-Pool: $13 \times 13 \times 8$
- Conv2: $11 \times 11 \times 16$
- Max-Pool:

$5 \times 5 \times 16$

- $\left\lfloor \frac{11-2}{2} \right\rfloor + 1 = 5$

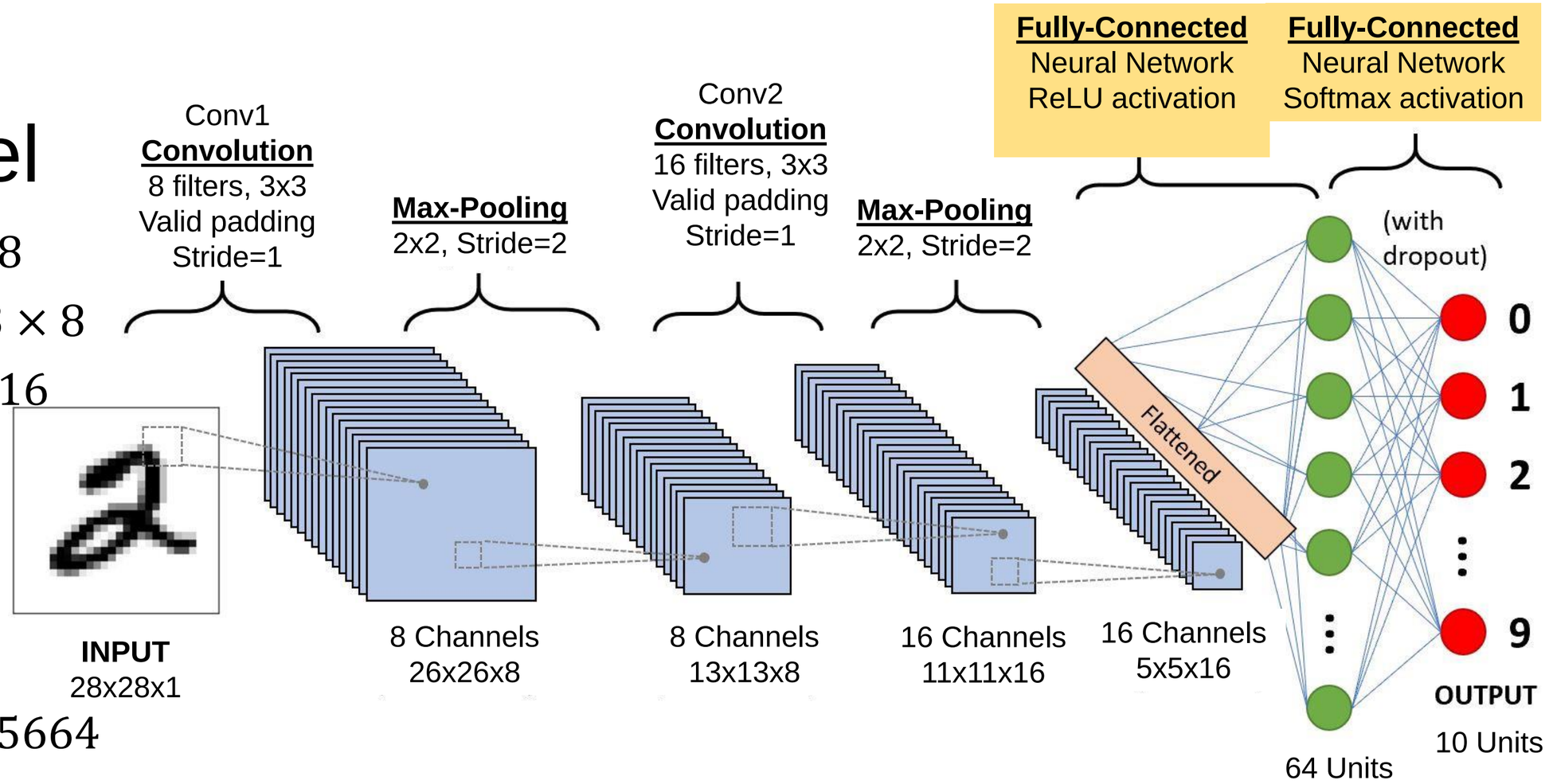


Typical CNN Model

- Conv1: $26 \times 26 \times 8$
- Max-Pool: $13 \times 13 \times 8$
- Conv2: $11 \times 11 \times 16$
- Max-Pool:

  $5 \times 5 \times 16$
- $5 \times 5 \times 16 = 400$
- FC1:

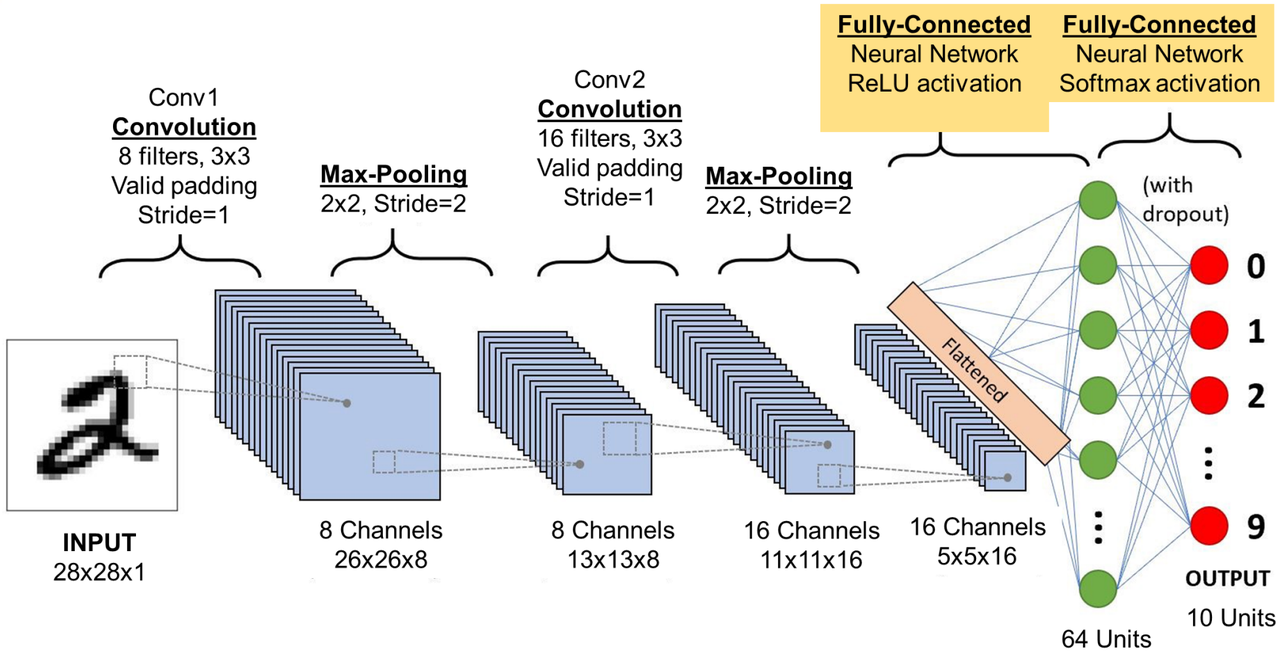
  $(400 + 1) \times 64 = 25664$
- FC2: (64+1)×10=650



Model Summary

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 8)	80
max_pooling2d (MaxPooling2D)	(None, 13, 13, 8)	0
conv2d_1 (Conv2D)	(None, 11, 11, 16)	1168
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 16)	0
flatten (Flatten)	(None, 400)	0
dense (Dense)	(None, 64)	25664
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 10)	650

Total params: 27,562
Trainable params: 27,562
Non-trainable params: 0



- The model summary displays the Shape of output feature maps and the number of trainable parameters in each layer, which is explained for this model in the previous slides

Parameters and Hyperparameters

	Parameters	Hyperparameters
Convolution layer	Kernels	Kernel size, number of kernels, stride, padding, activation function
Pooling layer	None	Pooling method, filter size, stride, padding
Fully connected layer	Weights	Number of weights, activation function
Others		Model architecture, optimizer, learning rate, loss function, mini-batch size, epochs, regularization, weight initialization, dataset splitting

Note that a parameter is a variable that is automatically optimized during the training process and a hyperparameter is a variable that needs to be set beforehand

Challenges with Deep Learning Models

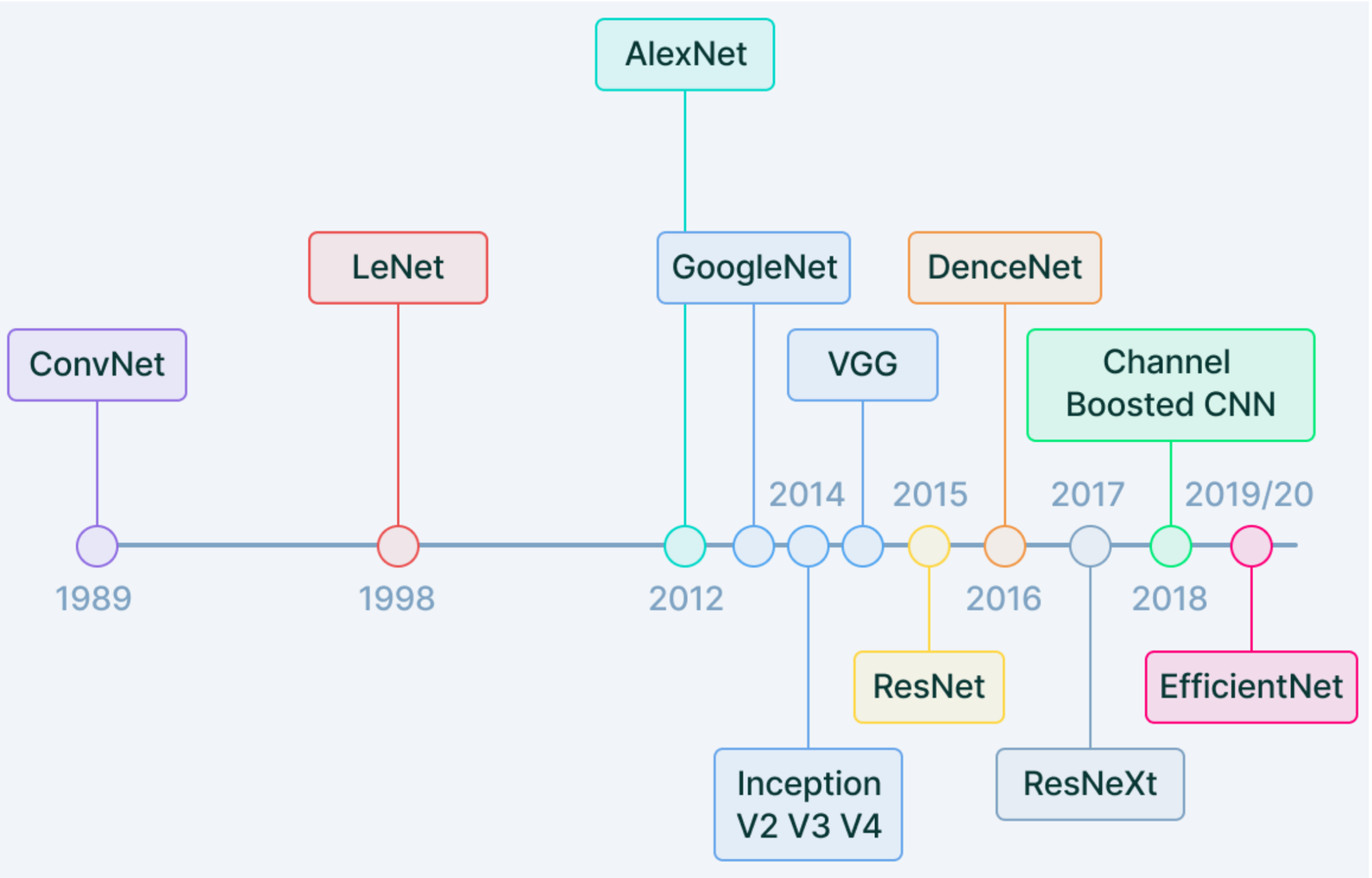
- Requirement of large amount of data
  - Data Augmentation
- Overfitting and Lack of Generalization
  - L1, L2 Regularization
  - Drop-out Regularization
  - Early-stopping
- Vanishing and Exploding Gradients
  - Choice of Activation Functions
  - Eg. CNN Architectures
- Effective Training
  - Optimizer
  - Normalization (Problem of Covariate Shift)
  - Early-stopping

Module II:
Model Parameters Optimization and Convolution Neural Networks

Module II (20%)

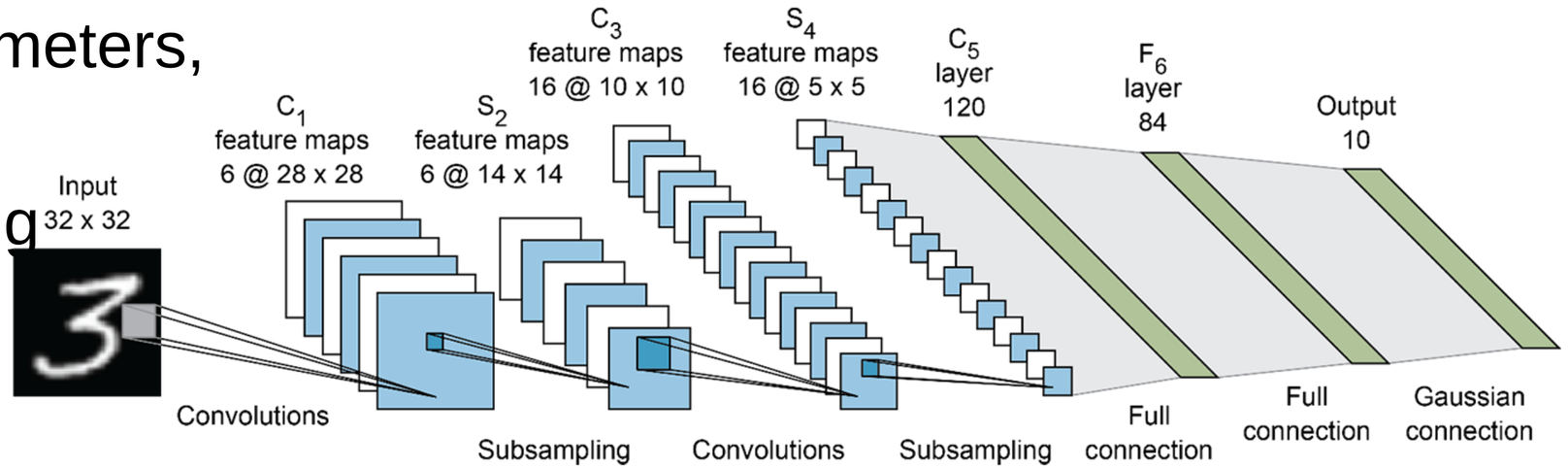
Methods to deal with overfitting issues, Optimizers (Momentum, Nesterov Accelerated Gradient, Adagrad, RMSprop, Adam), Convolution neural networks, Understanding the architectural characteristics of deep CNNs, Transfer Learning and Fine-tuning

Popular CNN Architectures



Popular CNN Architectures: **LeNet-5**

- Proposed by LeCun et al in 1998
- Applied by several banks to recognise hand-written characters on cheques digitized to 32x32 pixel greyscale input images
- 5 layers with learnable parameters,
  7 layer in total
  - 2 set of Conv-Subsampling
  - 1 Conv, 1 FC
  - 1 Output (10 units)

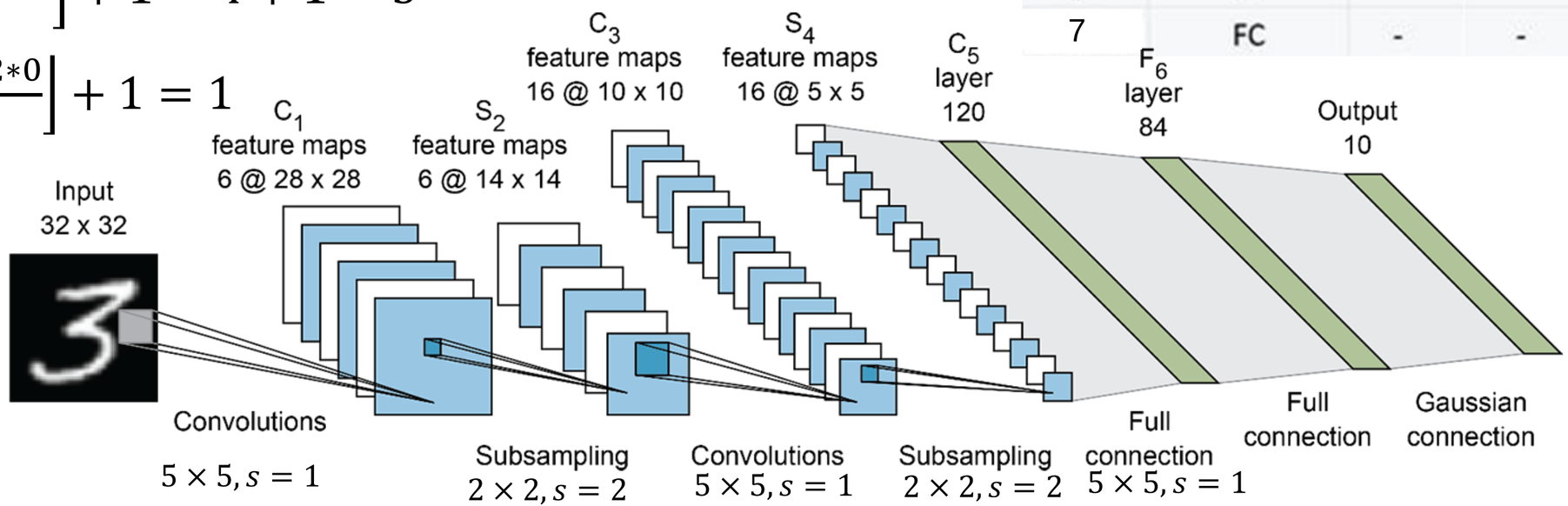


LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." *Proc. IEEE* 86, no. 11 (1998): 2278-2324.

LeNet-5

- Input: 32x32x1 greyscale images
- C1: $\left\lfloor \frac{32-5+2*0}{1} \right\rfloor + 1 = 27 + 1 = 28$
- S1: $\left\lfloor \frac{28-2+2*0}{2} \right\rfloor + 1 = 13 + 1 = 14$
- C3: $\left\lfloor \frac{14-5+2*0}{1} \right\rfloor + 1 = 9 + 1 = 10$
- S4: $\left\lfloor \frac{10-2+2*0}{2} \right\rfloor + 1 = 4 + 1 = 5$
- C5: $\left\lfloor \frac{5-5+2*0}{1} \right\rfloor + 1 = 1$

Layer		Kernel Size	Feature Map	Stride	Size	Activation
Input	Image	-	1	-	32x32	-
1	Convolution	5x5	6	1	28x28	tanh
2	Average Pooling	2x2	6	2	14x14	tanh
3	Convolution	5x5	16	1	10x10	tanh
4	Average Pooling	2x2	16	2	5x5	tanh
5	Convolution	5x5	120	1	1x1	tanh
6	FC	-	-	-	84	tanh
7	FC	-	-	-	10	softmax



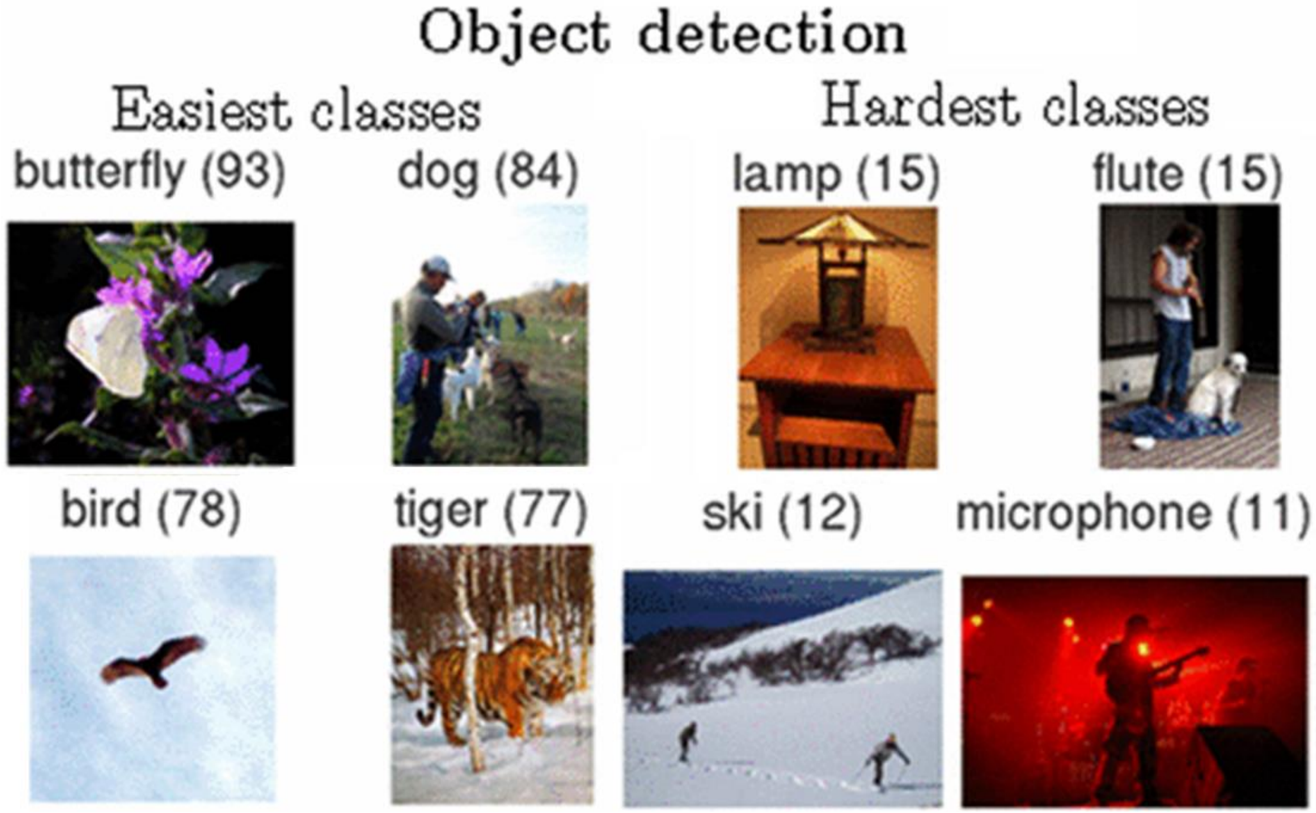
What's Novel in LeNet-5?

- LeNet-5: Architecture has become the standard 'template': stacking convolutions and pooling layers, and ending the network with one or more fully-connected layers

- It became popular due to its good performance for recognition of hand-written digits on bank cheques

ImageNet Dataset

- ImageNet is a dataset of
  - over 15 million labelled high-resolution images
  - from ~22,000 categories
- ImageNet Large Scale Visual Recognition Challenge (ILSVRC):
  - Between 2010 -2017
  - Uses ~1000 categories, each with ~1000 images



https://image-net.org/index.php

Popular CNN Architectures

- ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners:
  - AlexNet
  - VGGNet
  - GoogLeNet/Inception
  - ResNet

Year	CNN (Layers)	Developed by	Place In ILSVRC	Top-5 error rate	No. of parameters
1998	LeNet (7)	Yann LeCun et al			60 thousand
2012	AlexNet (8)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2014	GoogLeNet (22)	Google	1st	6.67%	4 million
2014	VGG Net (16 or 19)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet (152)	Kaiming He	1st	3.6%	25 million (ResNet 50)

Popular CNN Architectures

Top-5 Error Rate, eg.

- Input: cat image
- Neural network outputs:
  - Tiger: 0.4
  - Dog: 0.3
  - Cat: 0.1
  - Lynx: 0.09
  - Lion: 0.08
  - Bird: 0.02
  - Bear: 0.01
- Top-1 o/p contains Tiger ✗
- Top-5 o/p contains Cat ✓

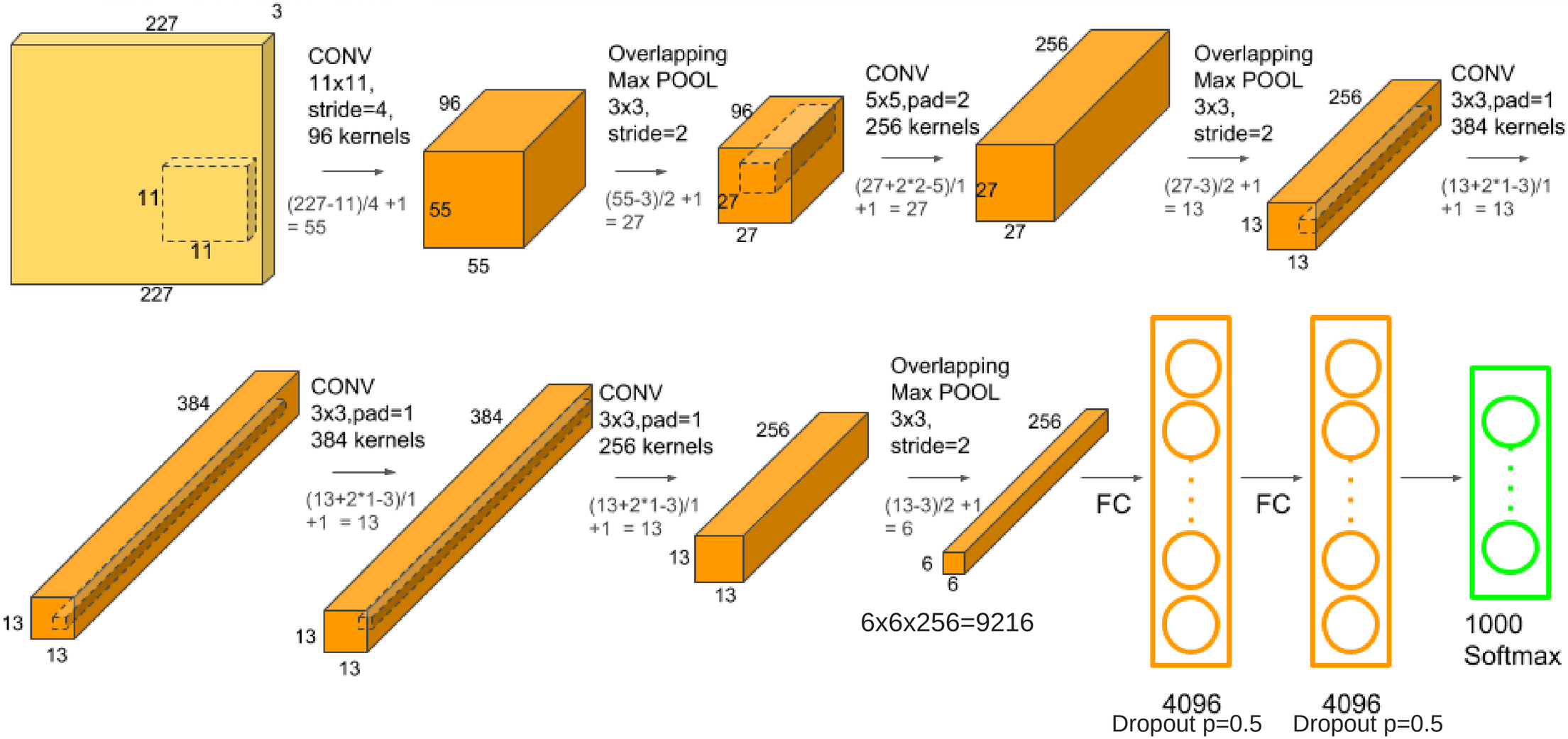
Year	CNN (Layers)	Developed by	Place In ILSVRC	Top-5 error rate	No. of parameters
1998	LeNet (7)	Yann LeCun et al			60 thousand
2012	AlexNet (8)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2014	GoogLeNet (22)	Google	1st	6.67%	4 million
2014	VGG Net (16 or 19)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet (152)	Kaiming He	1st	3.6%	25 million (ResNet 50)

AlexNet

- Developed by Alex Krizhevsky under the guidance of Sutskever Ilya and Geoffrey E. Hinton (PhD supervisors) in 2012
- Winner of ILSVRC challenge 2012 (top-5 error of 15.3%)
- 8-layer CNN (5 Conv+MaxPool and 3 FC),
- More than 60 million parameters

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012): 1097-1105.

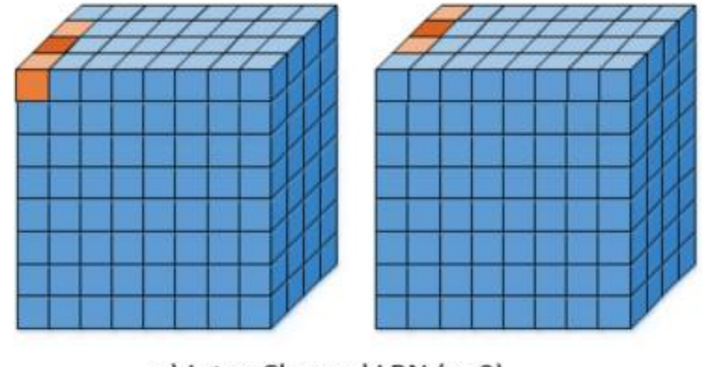
AlexNet



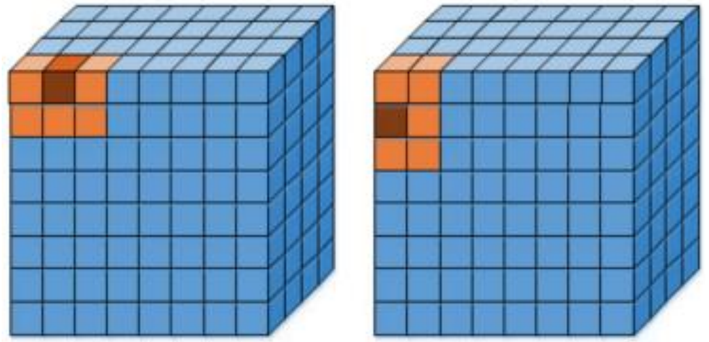
- Input: 227x227x3
- Conv 11x11, 5x5, three 3x3
- 3 MaxPool
- ReLu for hidden units
- Softmax for output

AlexNet

- ReLU activation (avoid vanishing gradient),
- Data Augmentation (avoid overfitting),
- Dropout regularization (avoid co-adaptation)
- Introduced Local Response Normalization (LRN)
  - LRN is a non-trainable layer that square-normalizes the pixel values in a feature map within a local neighbourhood (Inter-channel, Intra-channel)
  - does lateral inhibition: refers to the capacity of a neuron to reduce the activity of its neighbours



a) Inter-Channel LRN (n=2)



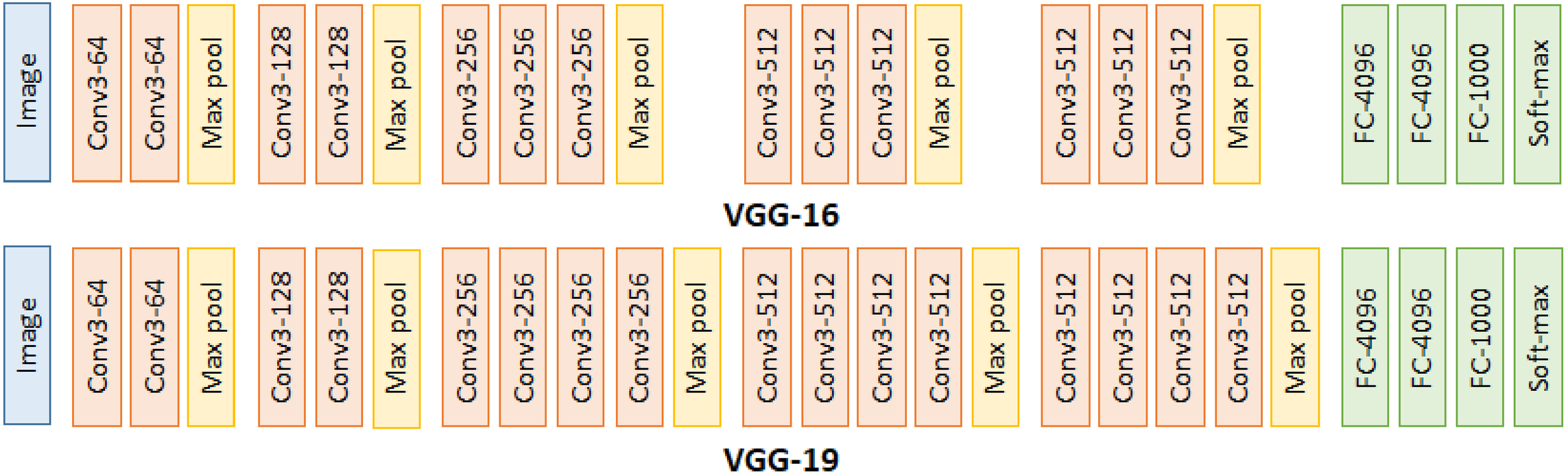
b) Intra-Channel LRN (n=2)

VGGNet

- Developed by Visual Geometry Group in 2014
- VGG16 was 2nd in ILSVRC challenge 2014 (top-5 classification error of 7.32%)
- Characterized by Simplicity and Depth
  - All Conv layers with 3x3 filters and stride 1, SAME padding
  - All max polling layers 2x2 filters, stride 2
- VGG16: 16-layer CNN (16 layers with trainable parameters, over 134 million parameters); VGG19: 19-layer CNN (more than 144 million parameters)

Karen Simonyan, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

VGGNet

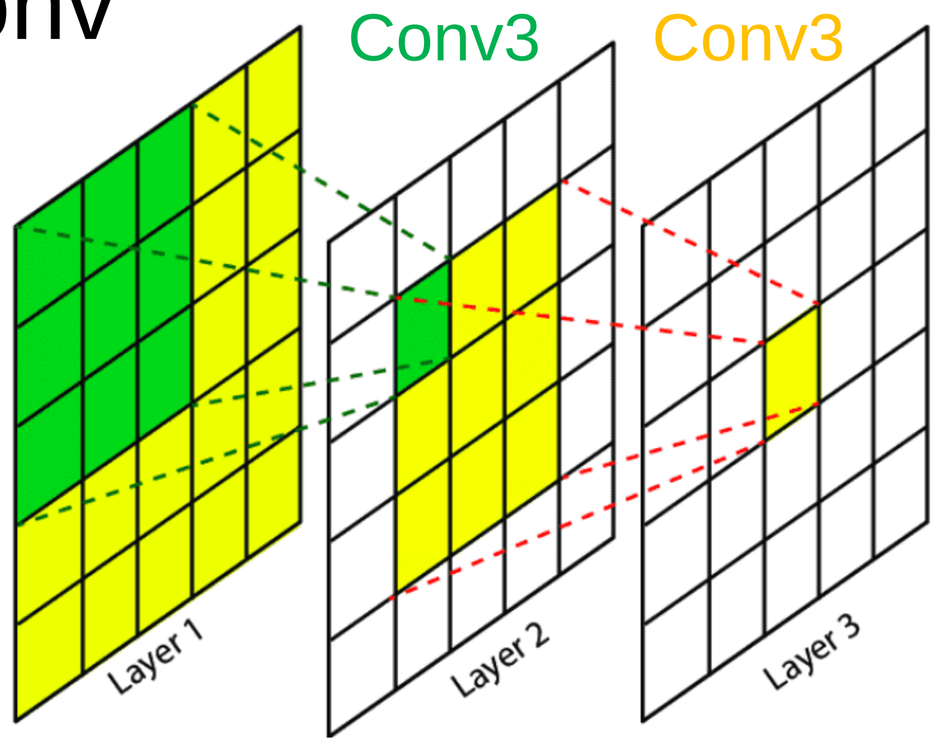


Conv= 3x3 filter, s=1, same
Max pool= 2x2, s=2 (5 Max pooling layers)

ReLU activation in all hidden units
Softmax activation in output units

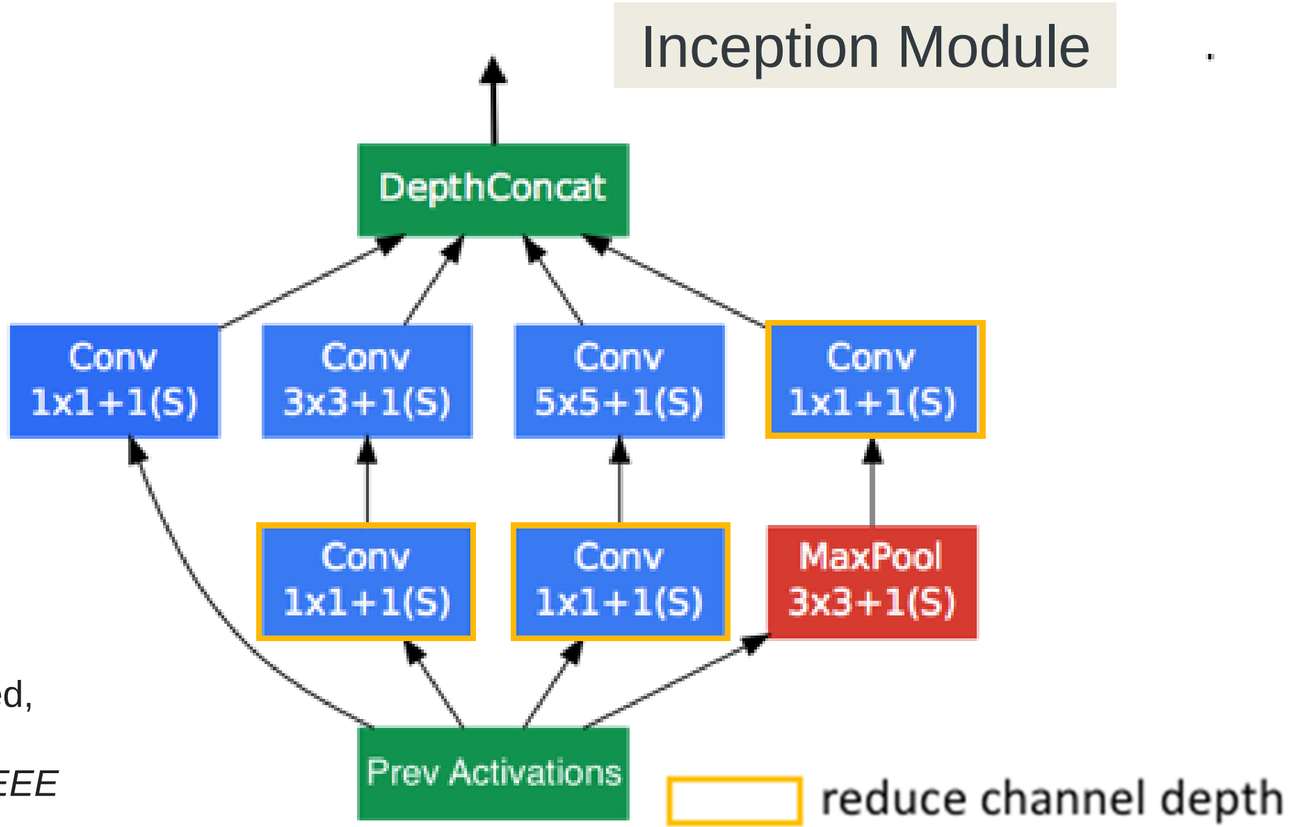
Stacking of Conv

- Multiple Stacked Conv layers lead to Wide Receptive Field
- In VGG, varying filter sizes are implemented by stacking Conv layers with fixed filter sizes
- Limitation: Training is very slow



GoogLeNet (Inception v1)

- Developed by Google in 2014
- 1st position in ILSVRC challenge 2014 (top-5 classification error of 6.66%)
- 22-layers with trainable parameters (27 layers including Max Pool layers)
- Parameters: 5 million (V1), 23 million (V3)
- Contains Inception Modules

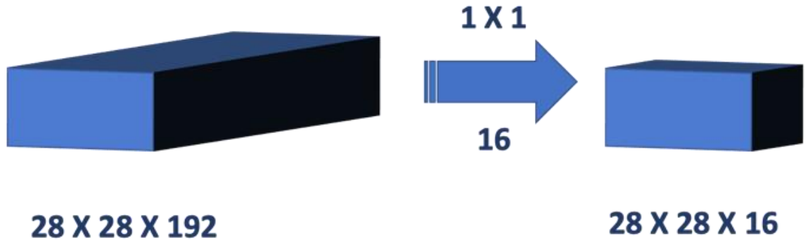


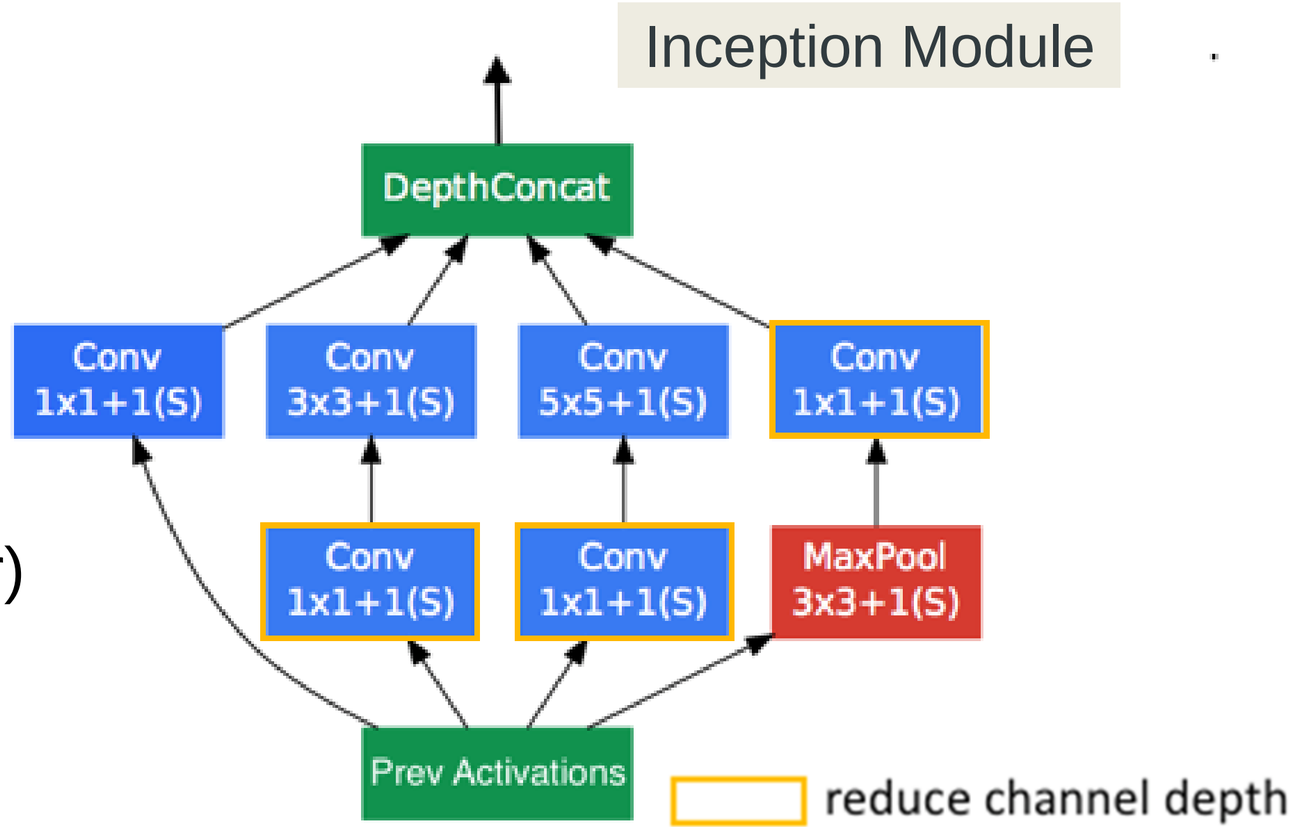
Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9. 2015.

GoogLeNet (Inception v1)

Inception Module/ Cell

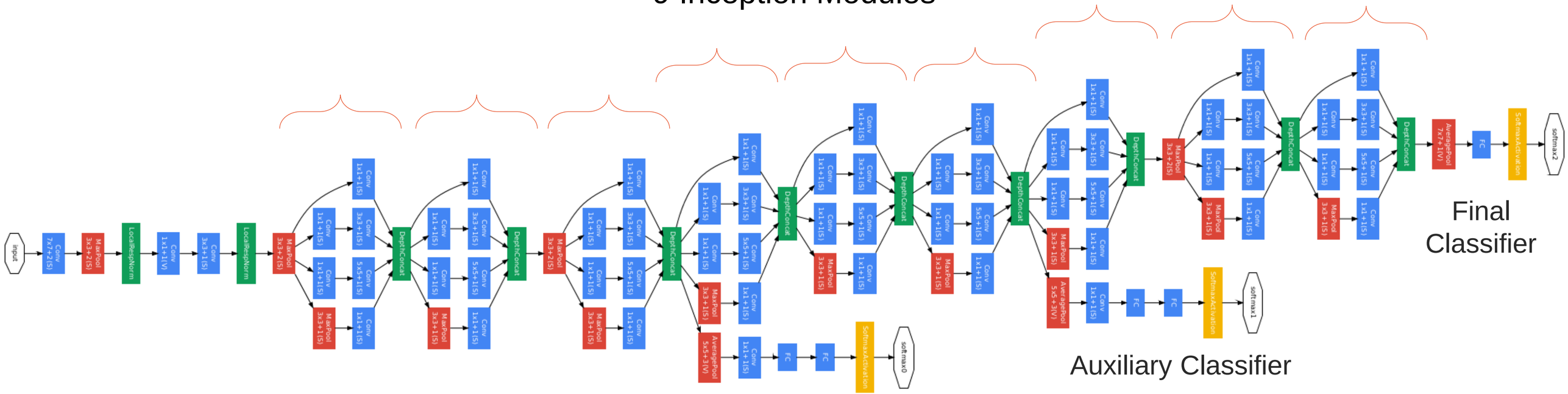
- Extract features at different scales from the input (1x1, 3x3, 5x5)
- Max pooling with "same" padding to preserve dimensions
- 1x1 Conv to decrease the number of feature maps (feature-map pooling layer)



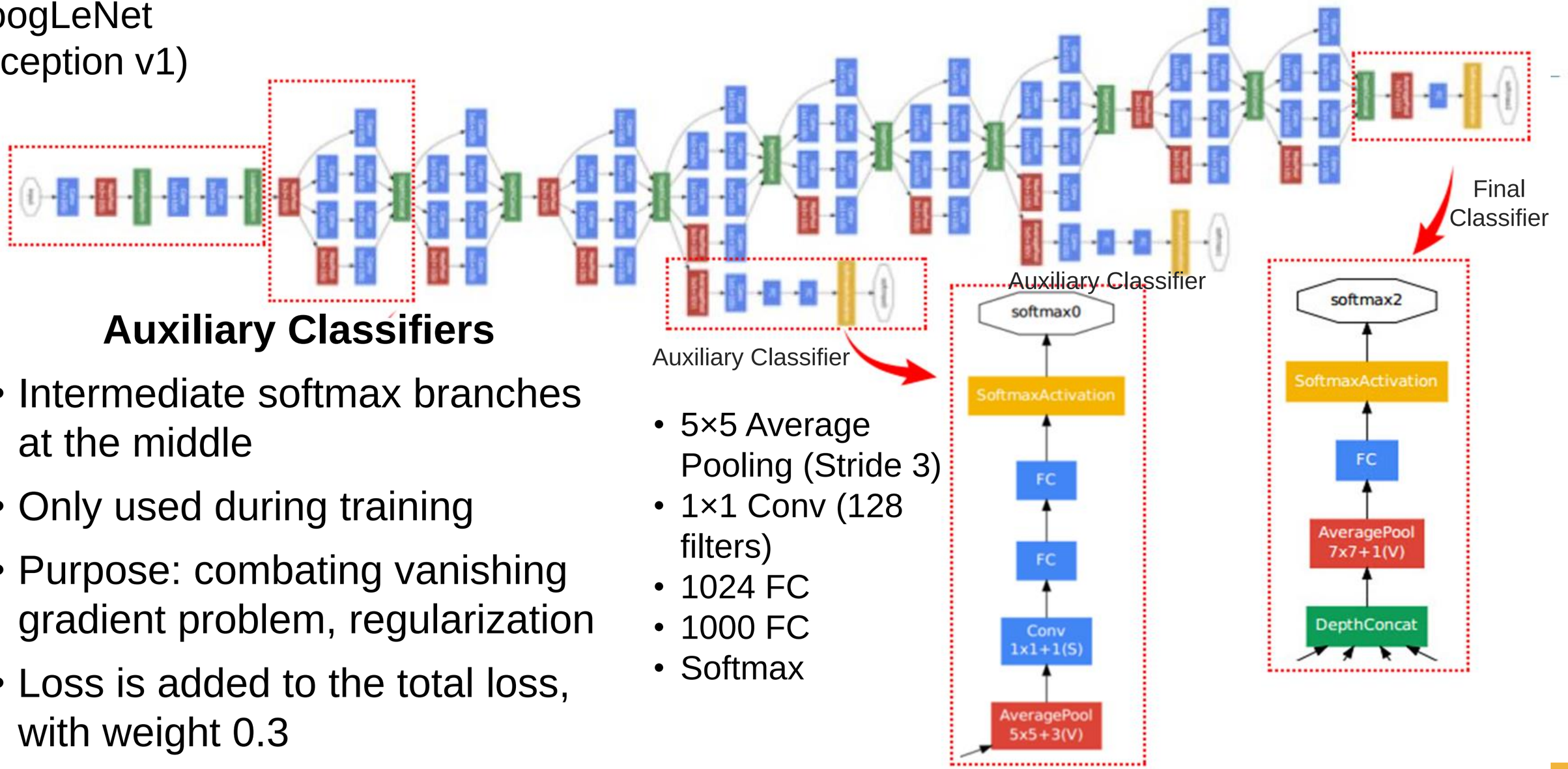


GoogLeNet (Inception v1)

9 Inception Modules



GoogLeNet
(Inception v1)

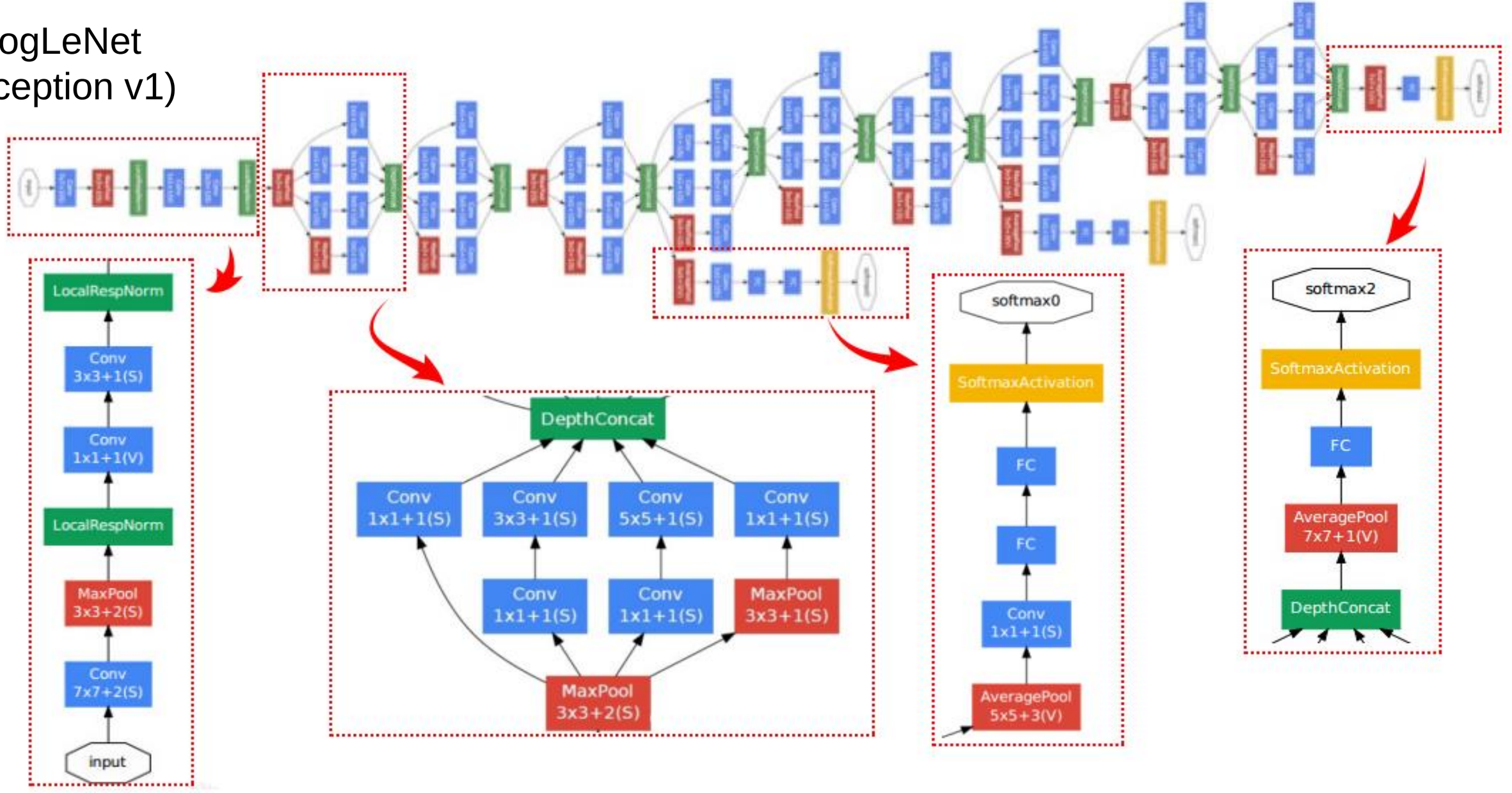


Auxiliary Classifiers

- Intermediate softmax branches at the middle
- Only used during training
- Purpose: combating vanishing gradient problem, regularization
- Loss is added to the total loss, with weight 0.3

- 5×5 Average Pooling (Stride 3)
- 1×1 Conv (128 filters)
- 1024 FC
- 1000 FC
- Softmax

GoogLeNet (Inception v1)

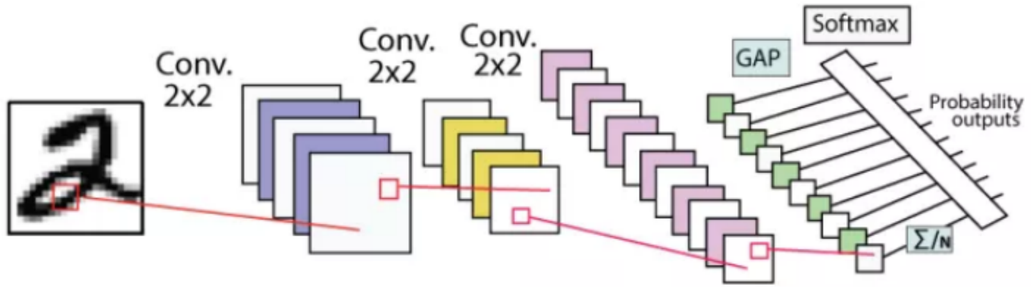


Global Average Pooling

- Typically, CNNs use convolutions in lower layers, followed by FCs and softmax logistic regression for classification
- FC are prone to overfitting and computationally expensive
- Global average pooling can replace the FC layers in CNNs

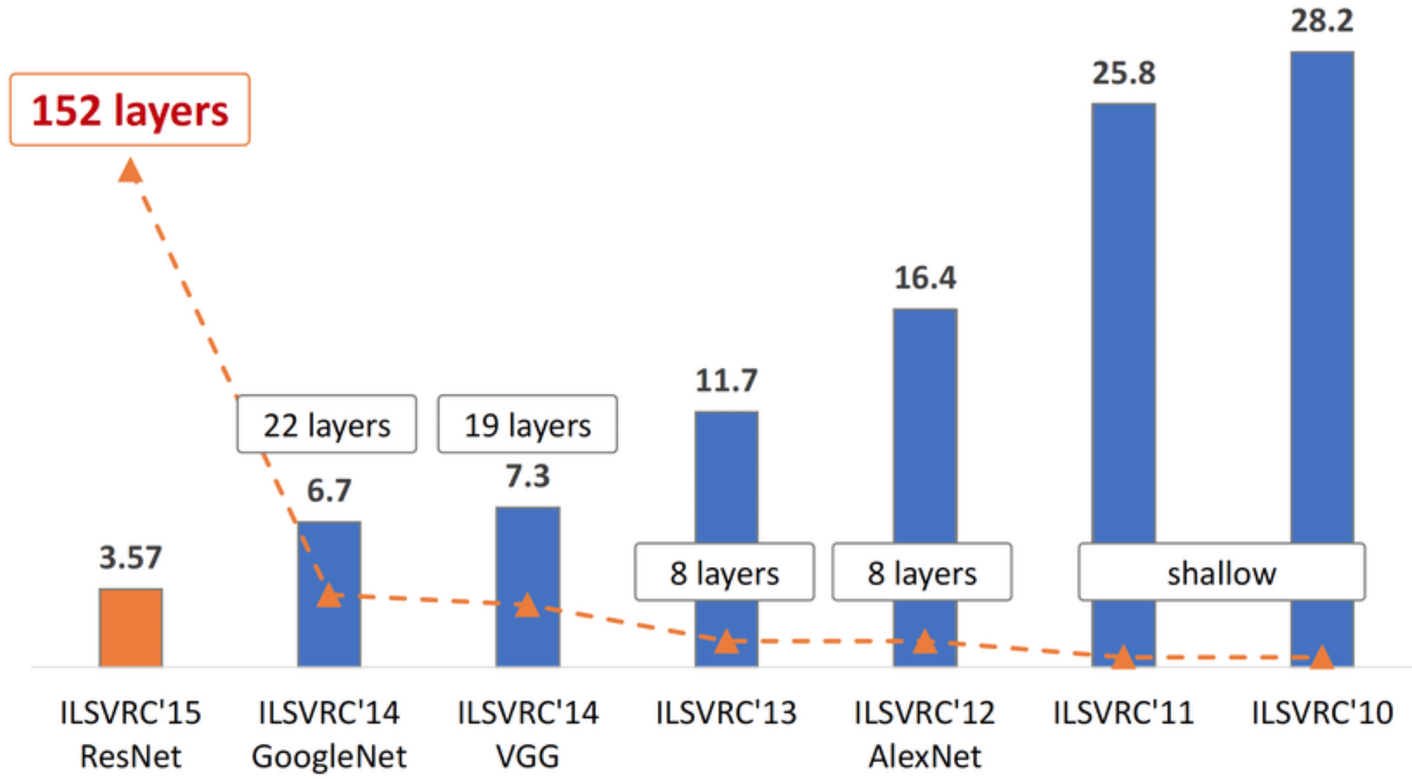
GAP idea:

1. Generate 1 activation map per class
2. Take the average of each map and feed this vector to softmax



Popular CNN Architectures

- ImageNet Large Scale Visual Recognition Challenge (ILSVRC) Winners
  - VGGNet (2nd, 2014)
  - GoogLeNet (1st, 2014)
  - ResNet (1st, 2015)



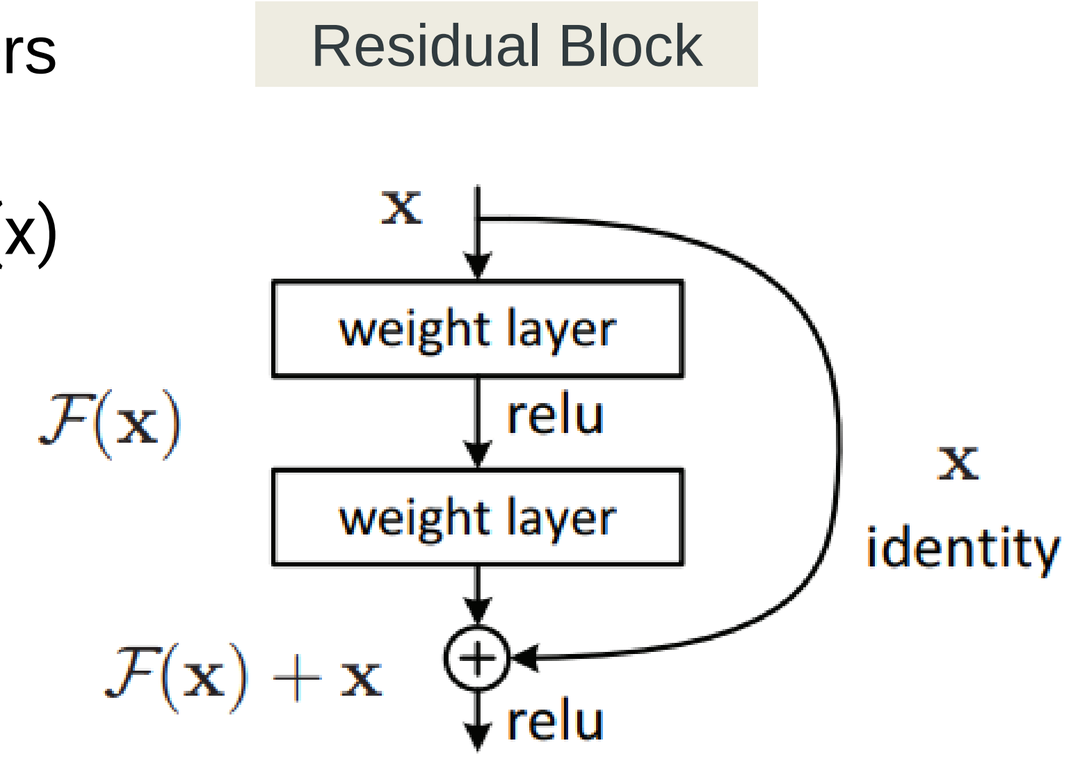
ResNet

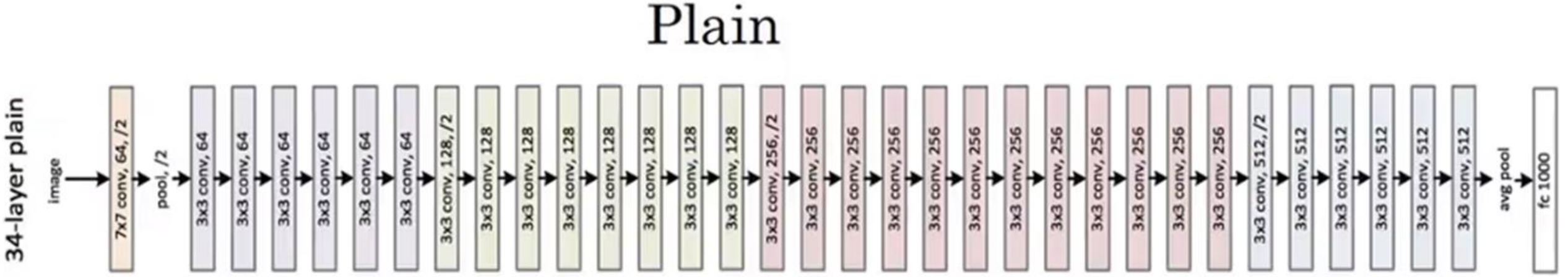
- Proposed by Shaoqing Ren, Kaiming He, Jian Sun, and Xiangyu Zhang
- Degradation problem of Deep NN: with increase in depth, the accuracy rate drops (*w* optimized to local minima)
- 1st position in ILSVRC challenge 2015 (top-5 classification error of 3.57%)
- 152-layers with trainable parameters Parameters: 25 million (ResNet 50)

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016, DOI: https://arxiv.org/abs/1512.03385.

Residual Block

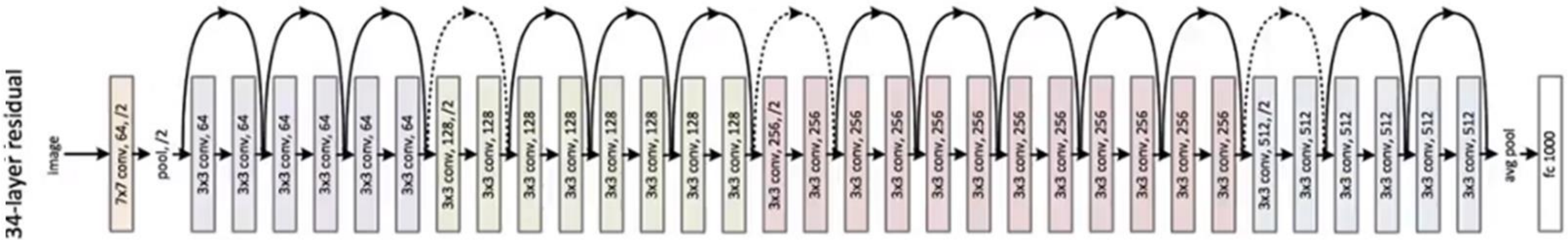
- Skip connection skips training from a few layers and connects directly to the output
- Instead of learning the underlying mapping H(x) from stacked layers, let network learn the residual F(x) = H(x)-x
- Hence, after adding identity, F(x)+x = H(x)
- Speeds learning by reducing the impact of vanishing gradients, avoid degradation
- Enable development of Deeper Networks



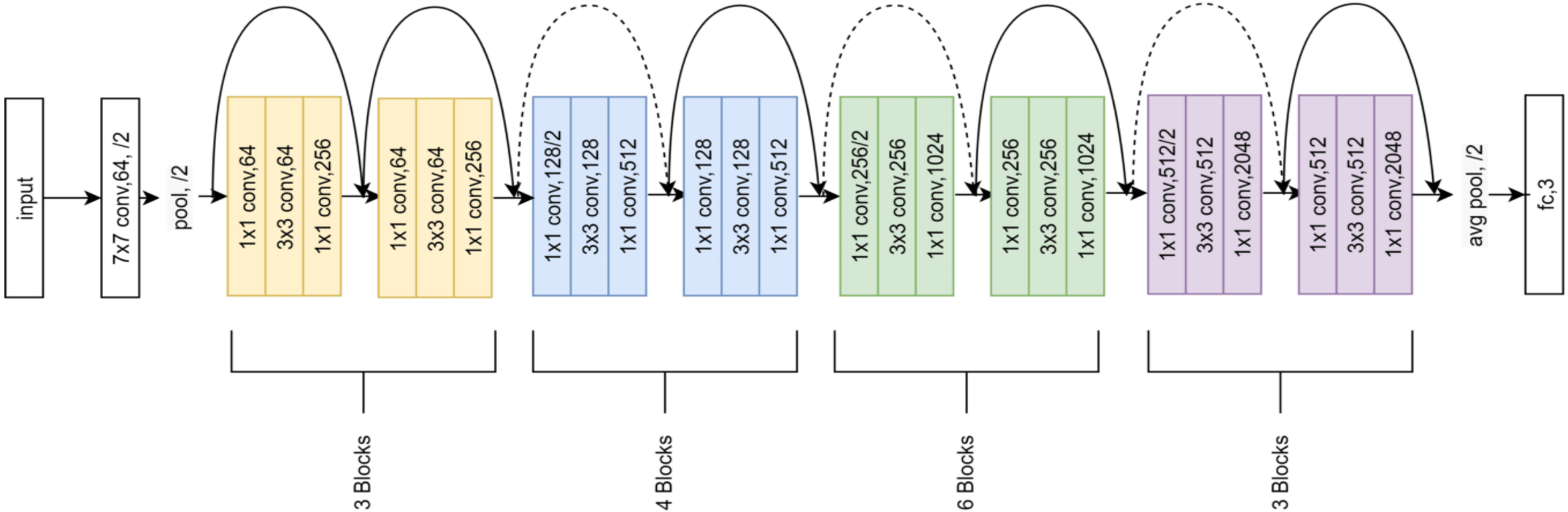


ResNet-34, which comprised 34 weighted layers.
A shortcut connection "skips over" some layers, converting a regular network to a residual network

It provided a novel way to add more convolutional layers to a CNN, without running into the vanishing gradient problem, using the concept of shortcut connections.



ResNet-50



ResNet-50: The Basics and a Quick Tutorial (datagen.tech)

Summary

Comparison					
Network	Year	Salient Feature	top5 accuracy	Parameters	FLOP
AlexNet	2012	Deeper	84.70%	62M	1.5B
VGGNet	2014	Fixed-size kernels	92.30%	138M	19.6B
Inception	2014	Wider - Parallel kernels	93.30%	6.4M	2B
ResNet-152	2015	Shortcut connections	95.51%	60.3M	11B

Module II:
Model Parameters Optimization and Convolution Neural Networks

Module II (20%)

Methods to deal with overfitting issues, Optimizers (Momentum, Nesterov Accelerated Gradient, Adagrad, RMSprop, Adam), Convolution neural networks, Understanding the architectural characteristics of deep CNNs, Transfer Learning and Fine-tuning

Transfer Learning

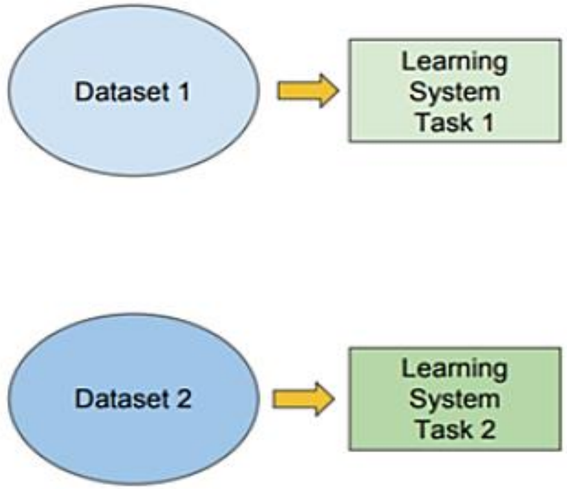
.

a) Transferring Learned knowledge from Java to Python.

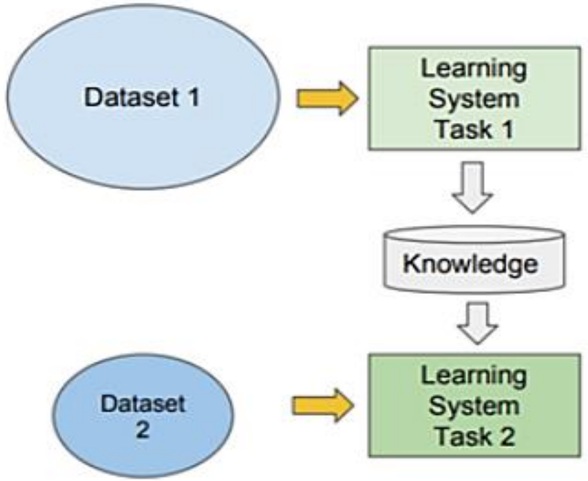
Knowing one programming language makes learning of another programming language simpler

Traditional ML vs Transfer Learning

- Isolated, single task learning:
  - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks



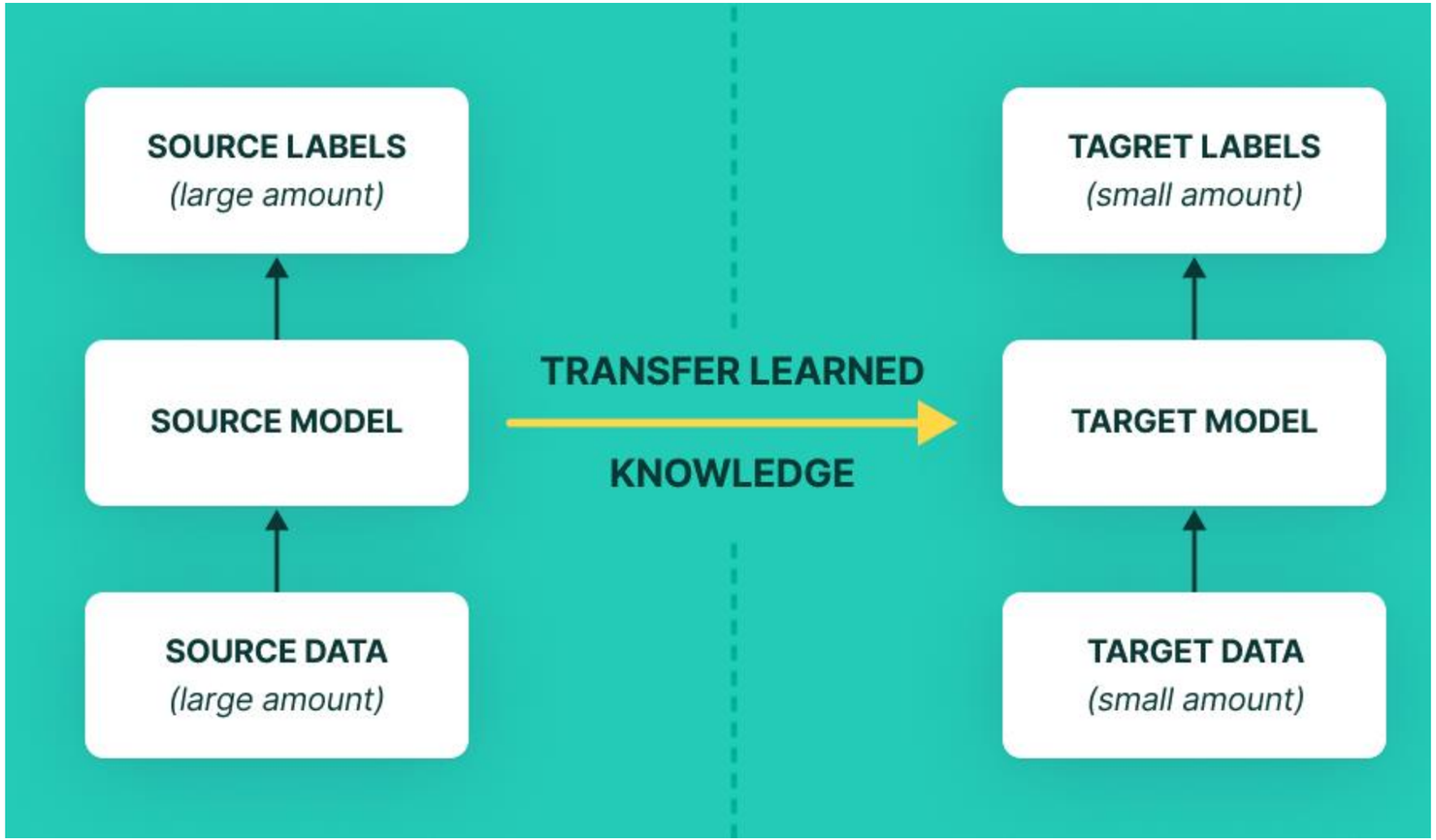
- Learning of a new tasks relies on the previous learned tasks:
  - Learning process can be faster, more accurate and/or need less training data



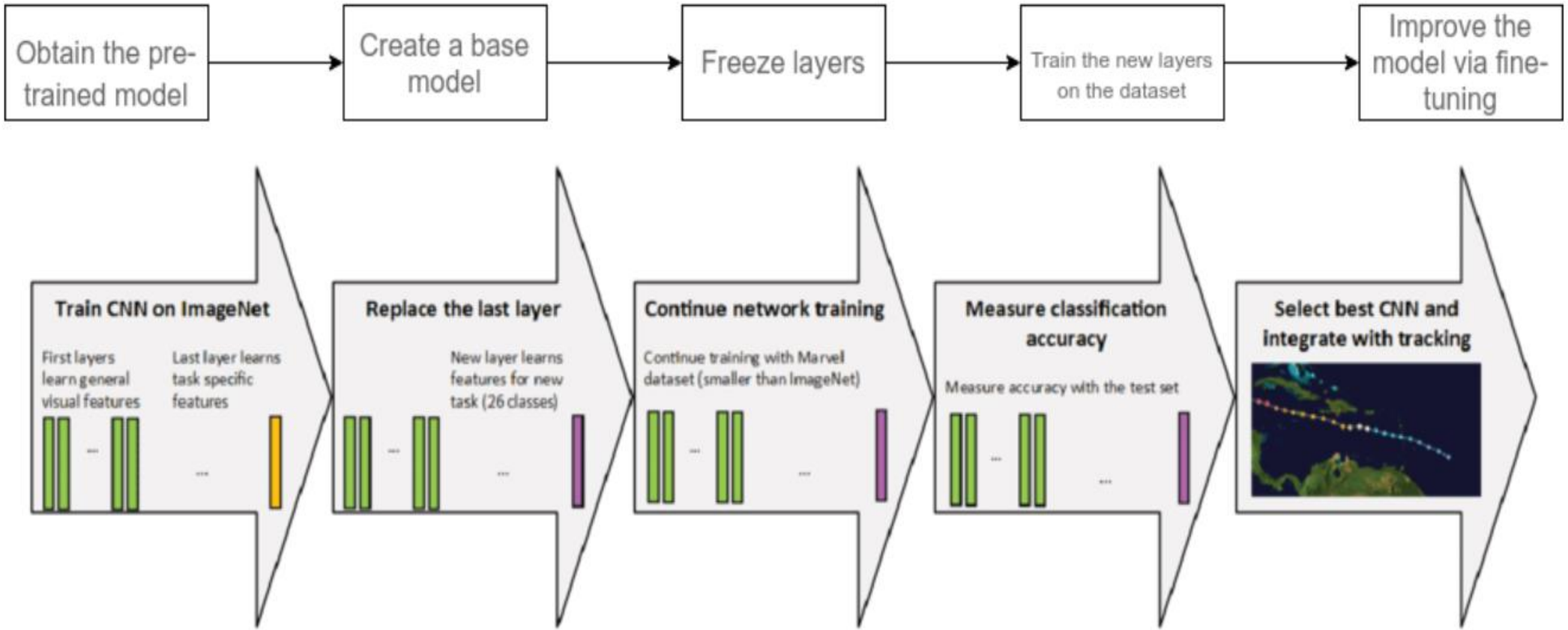
Traditional Learning vs Transfer Learning

How TL works in case of Deep Learning Models?

- The machine exploits the knowledge gained from a Source task to improve generalization about Target task
- Improvement of learning in a **new** task through the *transfer of knowledge* from a **related** task that has already been learned
- When to use TL? New dataset is small and similar to the original dataset



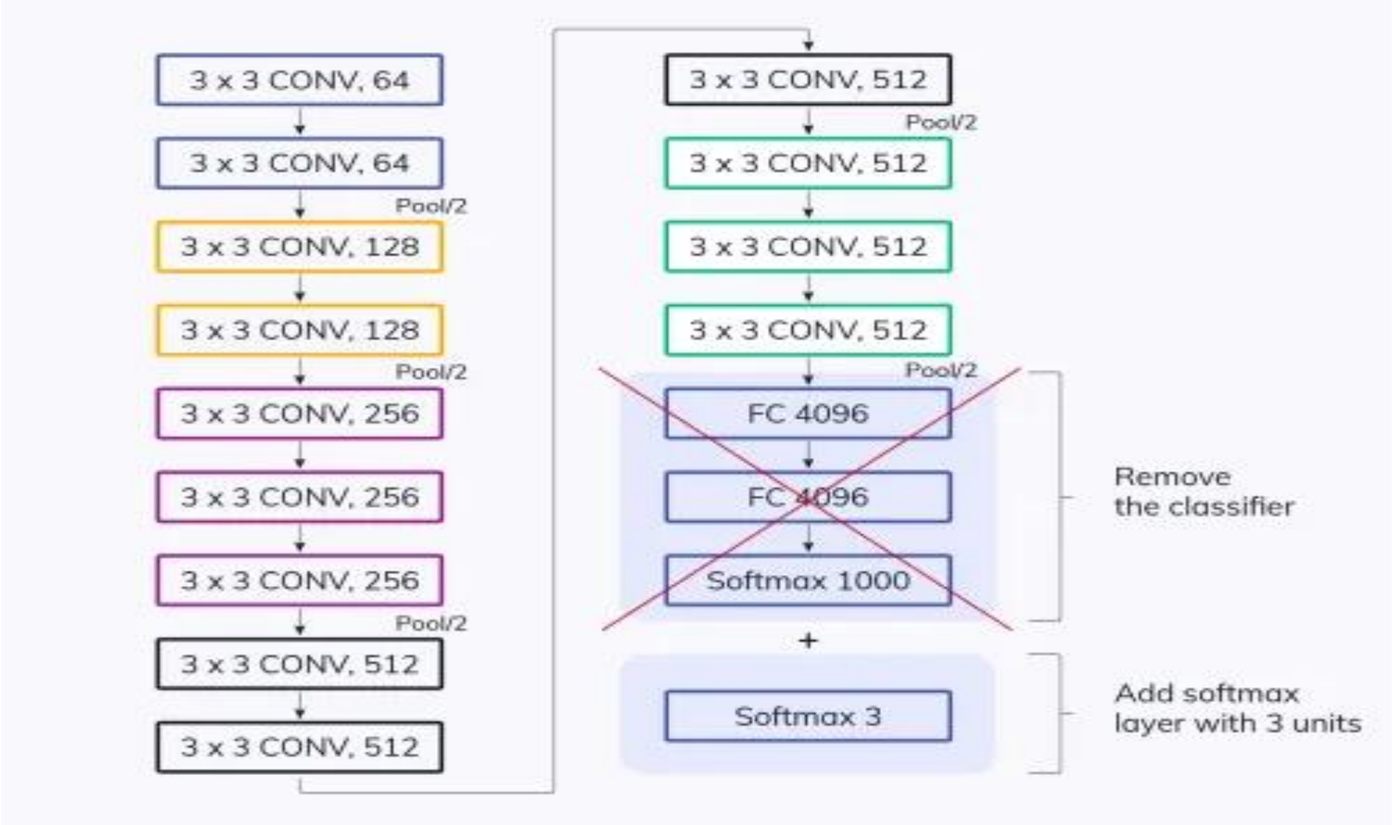
Steps in Transfer Learning



1-Obtain pre-trained model

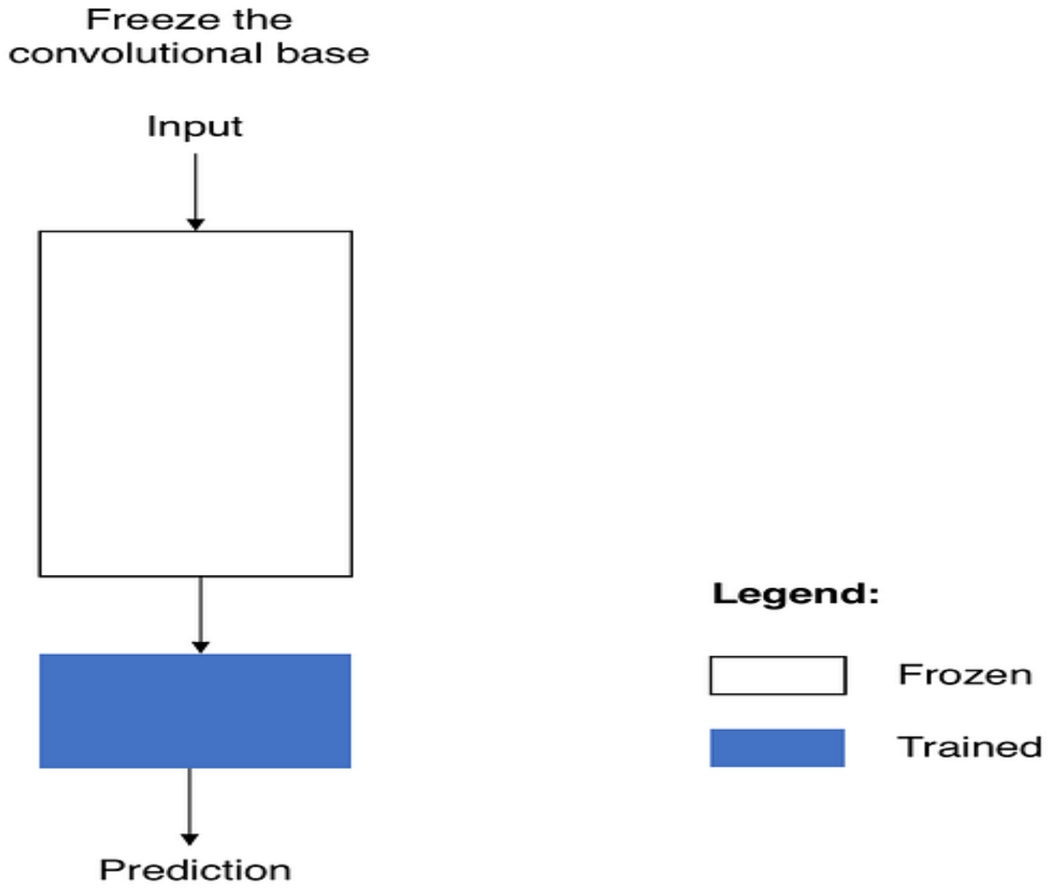
- VGG-16
- VGG-19
- Inception V3
- XCeption
- ResNet-50

2. Create a base model

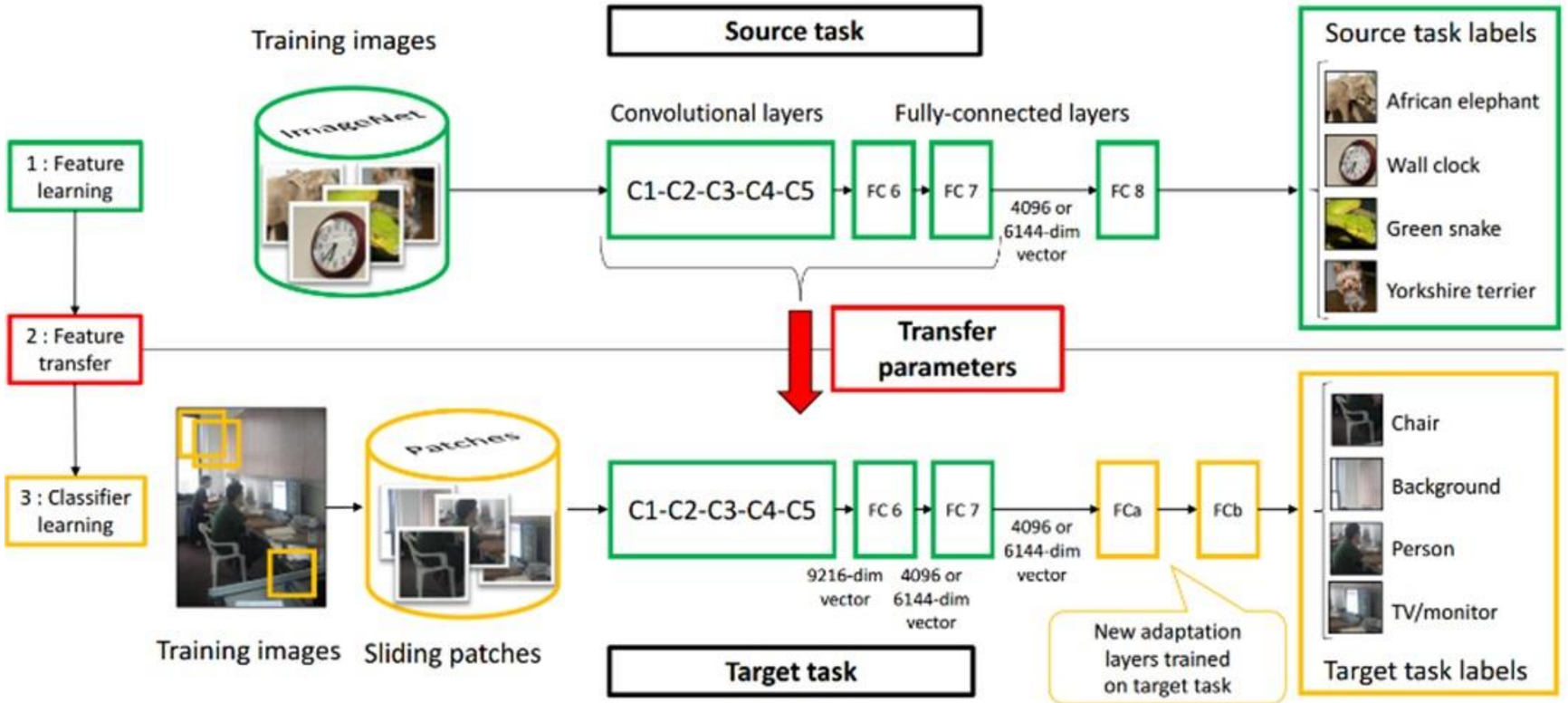


3. Freeze layers

- Freezing the starting layers from the pre-trained model is essential to avoid the additional work of making the model learn the basic features.
- If we do not freeze the initial layers, we will lose all the learning that has already taken place. This will be no different from training the model from scratch and will be a loss of time, resources, etc.



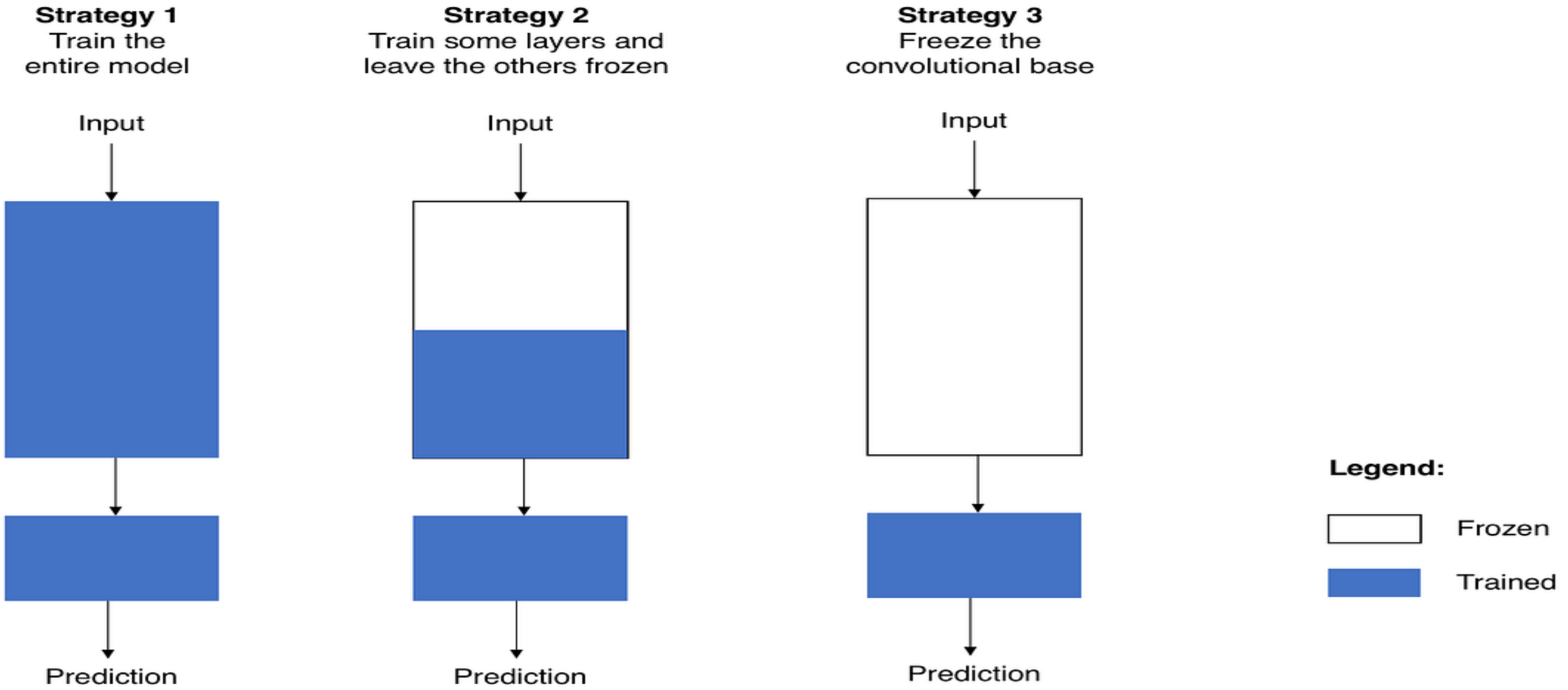
4. Add new trainable layers



- The only knowledge we are reusing from the base model is the feature extraction layers. We need to add additional layers on top of them to predict the specialized tasks of the model. These are generally the final output layers.

5. Train the new layers

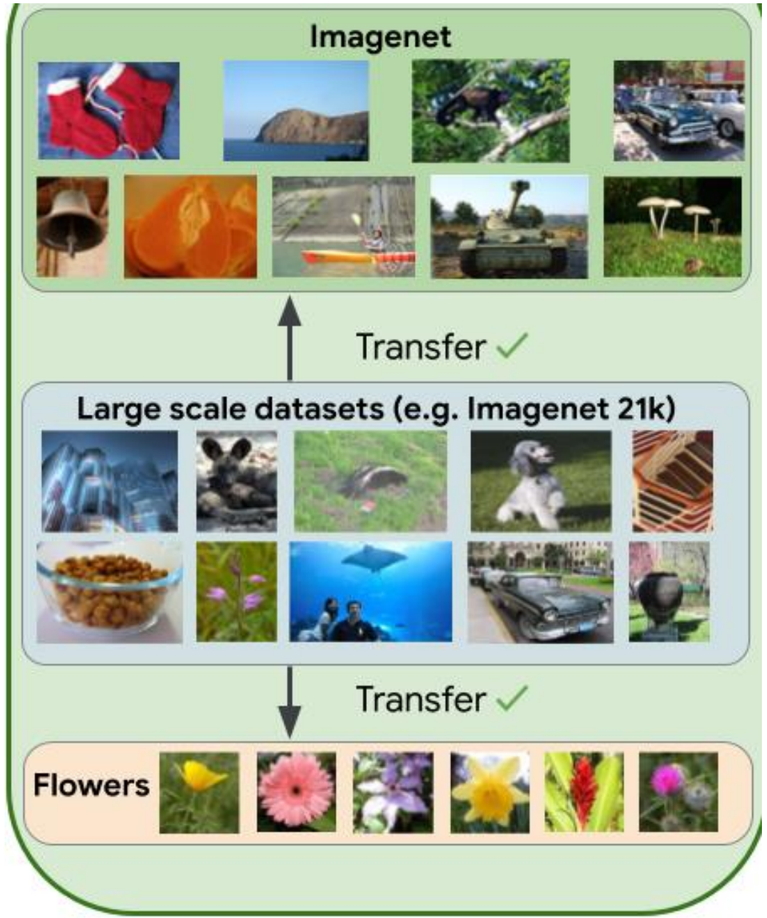
- The pre-trained model's final output will most likely differ from the output we want for our model. For example, pre-trained models trained on the ImageNet dataset will output 1000 classes.
- However, if we need our model to work for two classes, we have to train the model with a new output layer in place


Strategy 1
Train the entire model
Input
Prediction
Strategy 2
Train some layers and leave the others frozen
Input
Prediction
Strategy 3
Freeze the convolutional base
Input
Prediction
Legend:
Frozen
Trained

Fine-tune the model

- One method of improving the performance is fine-tuning.
- Fine-tuning involves unfreezing some part of the base model and training the entire model again on the whole dataset at a very low learning rate.
- A low learning rate will increase the performance of the model on the new dataset while preventing overfitting.

Applications of Transfer Learning



One of the most popular and successful applications of transfer learning in neural networks is image recognition.
Image recognition is the task of identifying and classifying objects, faces, scenes, or emotions in images.
There are many pre-trained neural networks that have been trained on large and diverse image datasets, such as ImageNet or COCO, that can be used as feature extractors or fine-tuned for new image recognition tasks.
For example, you can use a pre-trained network like ResNet or VGG to extract features from your own images, and then add a new classifier layer on top to train on your specific image recognition problem, such as flower type identification or medical image diagnosis