Machine Learning L+Pr

Béla J. Szekeres, PhD

Lecture 11

ELTE Faculty of Informatics, Szombathely, Hungary

Convolutional Neural Networks

They were proposed in 1998, but only became practical in 2012.

Why?

- Computation Power (Matrix Multiplication on GPUs)
- Arise of several datasets

Proposed by Yann LeCun

Yann LeCun
Director of AI Research
facebook

First, Why not just flatten the image and use Feed Forward Networks?

- Imagine we have an image with simply 50x50 pixels
- Still OK, the size is pretty small.

50x50
= 2500 Pixels

Flatten

2500
Input
Features

HOW MANY CONNECTIONS (WEIGHTS) WILL BE HERE?

Hidden Layer

Since we are working with fully-connected layers, meaning each neuron is connected to all other neurons

In the **Hidden Layer**, each neuron receives 2500 weights. The more number of neurons in the hidden layer, the more connections!

First, Why not just flatten the image and use Feed Forward Networks?

- Imagine we have an image with 250x250
- Well, now there is a problem

250x250
= 62,500 pixels

Flatten

62500
Input
Features

HOW MANY CONNECTIONS (WEIGHTS) WILL BE HERE?

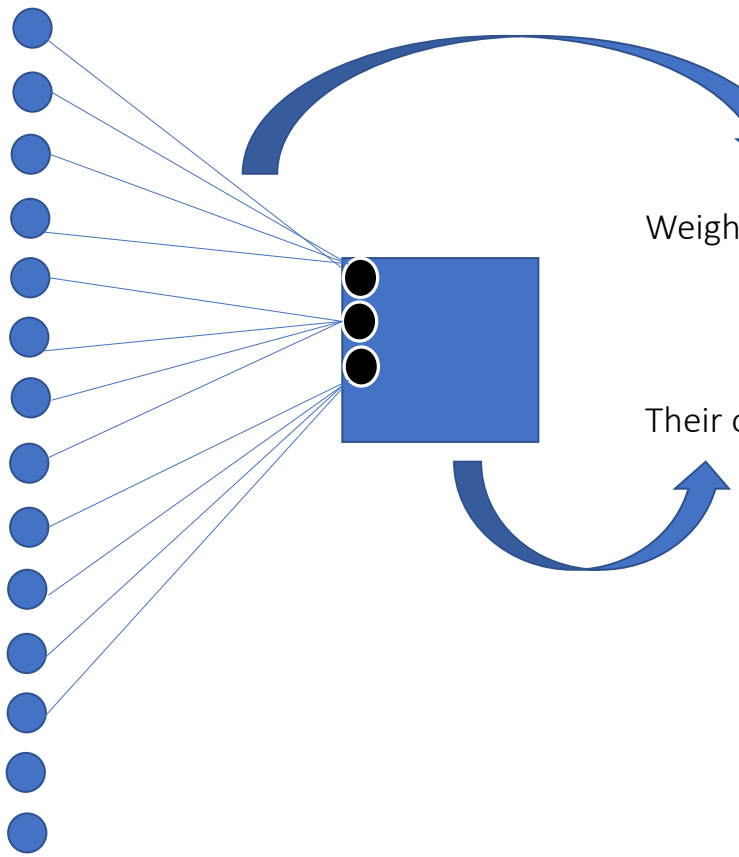
Hidden Layer

Since we are working with fully-connected layers, meaning each neuron is connected to all other neurons

In the **Hidden Layer**, each neuron receives 62,500 weights!. The more number of neurons in the hidden layer, the more connections!

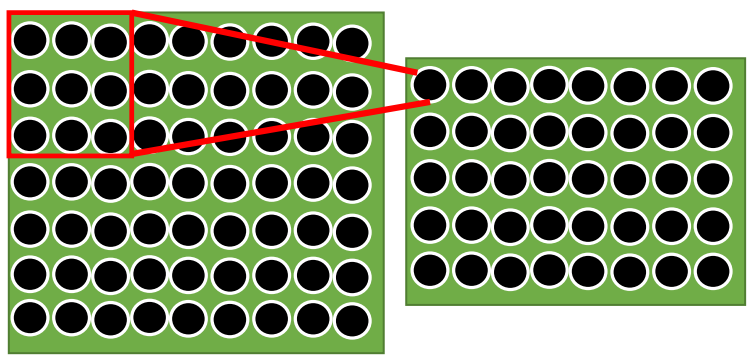
TOO MANY PARAMETERS TO LEARN! THE NETWORK CAN'T HANDLE THAT MUCH AND WILL OVERFIT

We need to think of a way to reduce this....



Weights are the **Filters**

Their output is a feature map



How are features extracted?

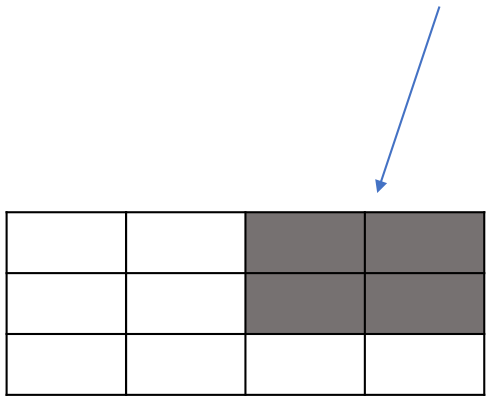
Image

3	4	5	8	7	4
2	7	4	6	0	6
6	6	7	2	9	7
7	3	8	4	0	6
0	9	1	0	6	7

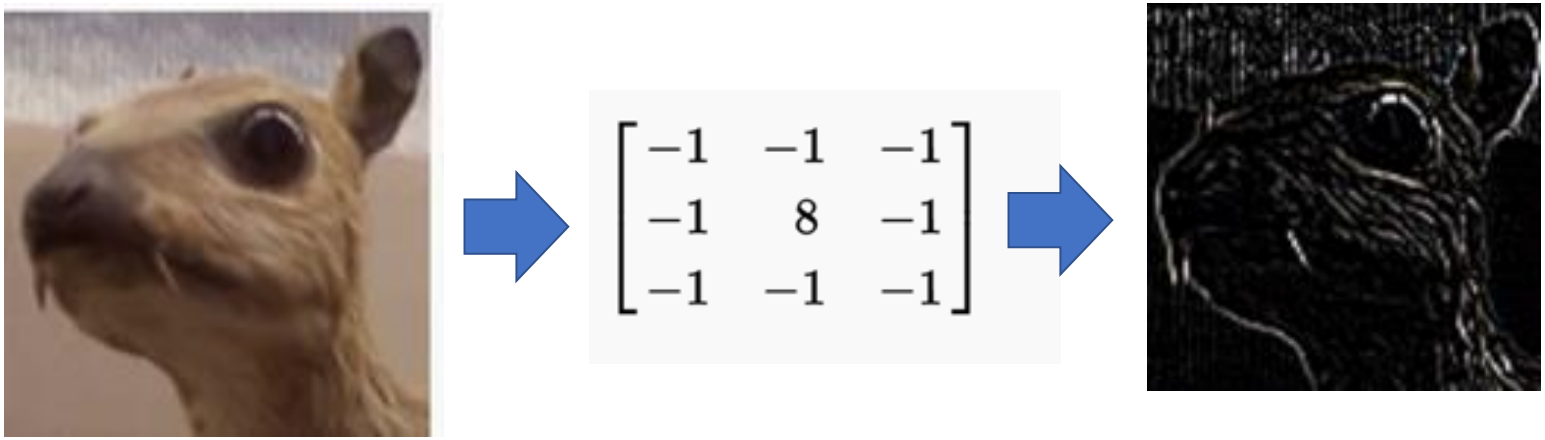
Filter

8	7	4
6	0	6
2	9	7

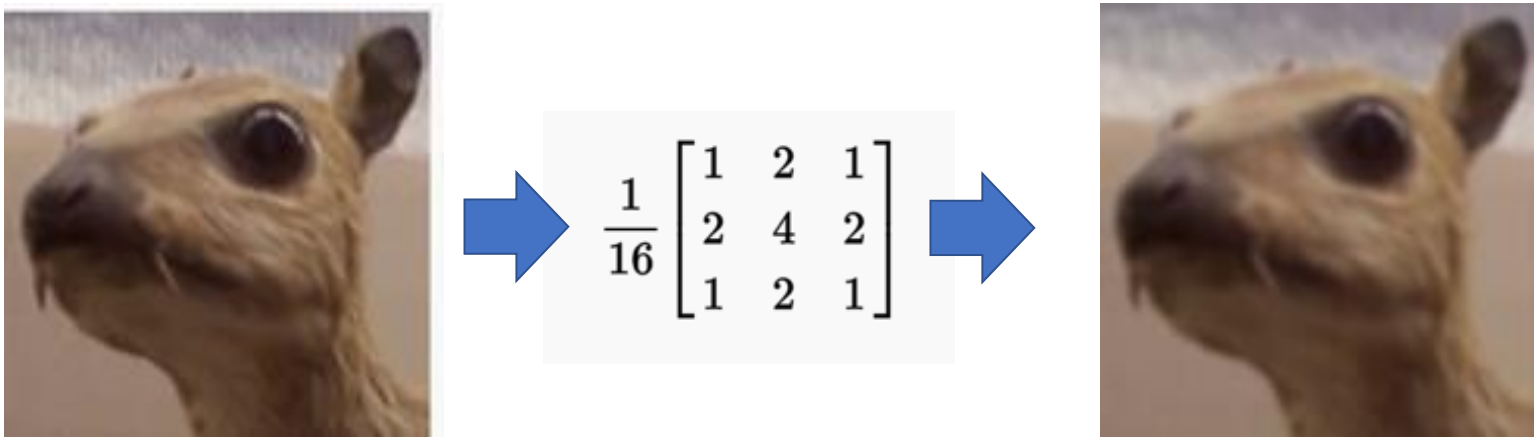
The Area which matches the filter gives high values



Edge Detection Filters



Gaussian Blurring Filters



In Convolutional Networks....

- We don't set these filters → NO HANDCRAFTED FILTERS
- The networks ***learns*** these filters using the backpropagation.
- The weights are the filters

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

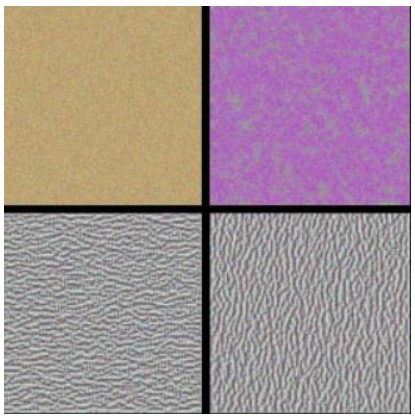
So….Meaning?

- Large Value → some sort of curve in the input
- 0 or Small Value → no curve
- We applied only a curve filter, but the more filters we apply, the more information about the image content we get!

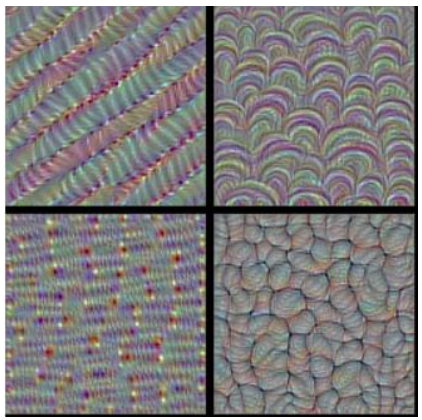
Take Note:

- We don’ t choose the filters, we let the network learn these filters and figure out on it’ s own what filters it needs
- The filter depth should be the same as the previous depth
- We apply the convolution operation across the dimension

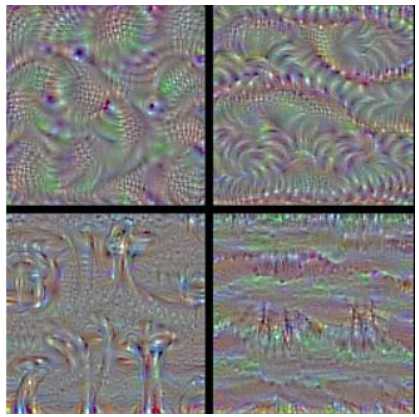
Visualizing Filters Learned from the Image



First Layer



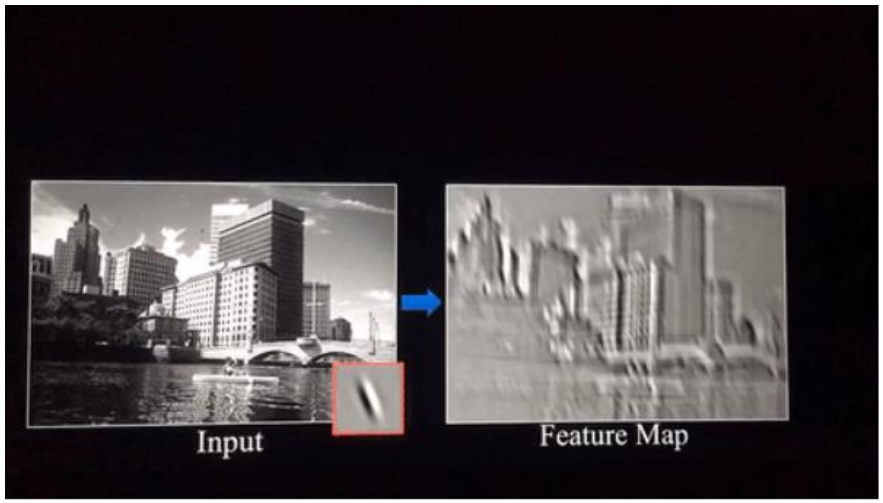
Before Last Layer

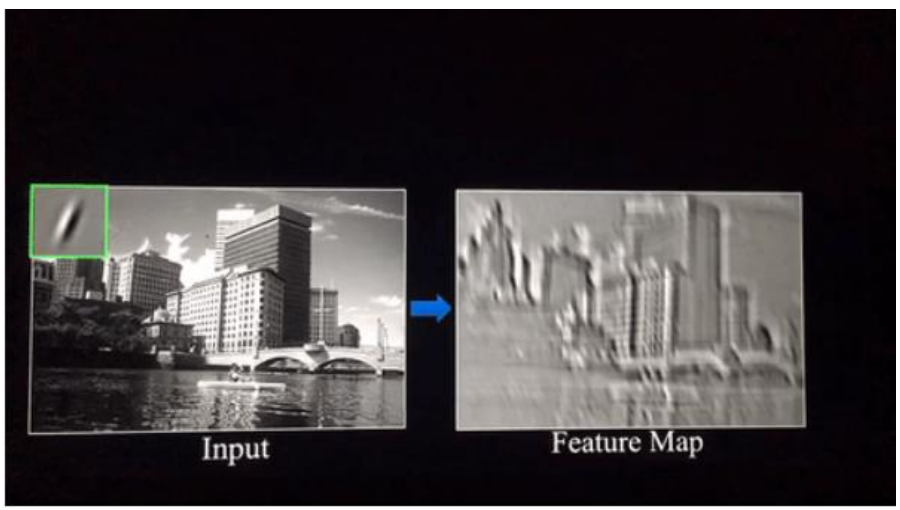


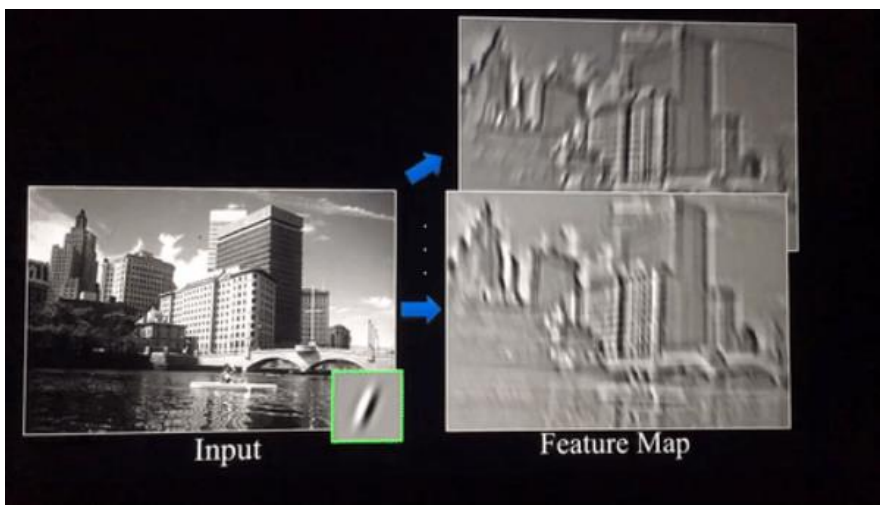
Last Layer

Let's Apply Some of these filters and see what happens to the image

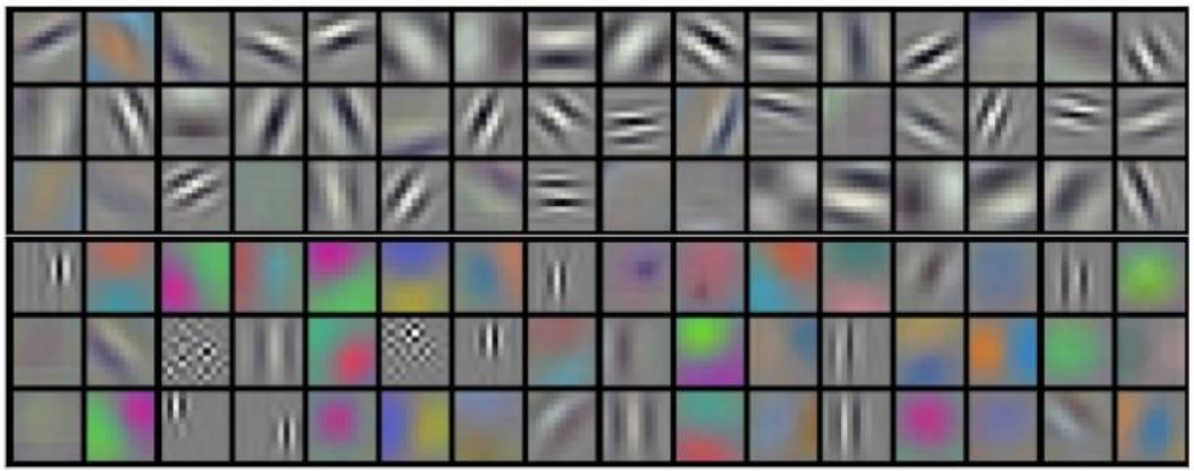

Input


Input
Feature Map


Input
Feature Map


Input
Feature Map

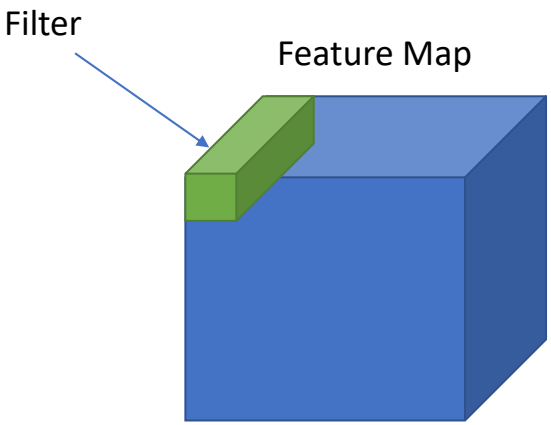
Filters learned from an image



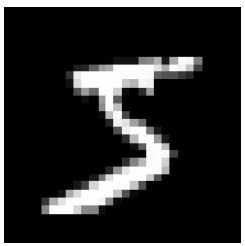
Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

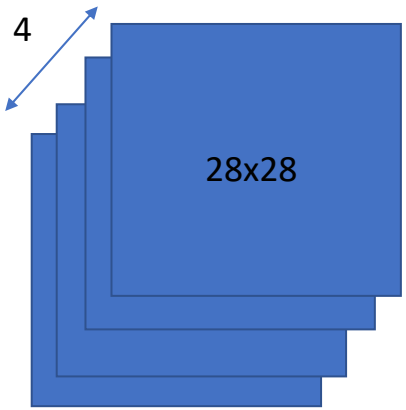
http://cs231n.github.io/convolutional-networks/

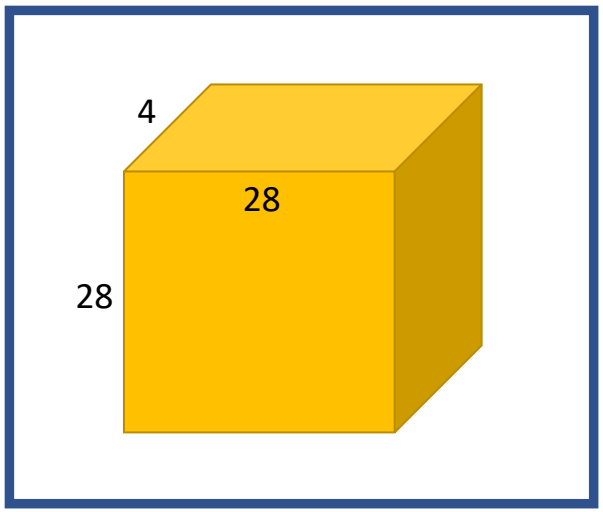
Convolution Over Volume

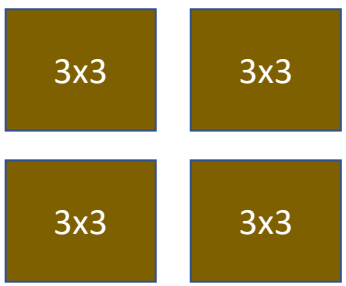


Filter Depth should always be the same as the depth of the feature map it is getting convolved with. For example, if the feature map has 3 channels, then each filter should also have 3 channels. The result is the addition of all convolved channels

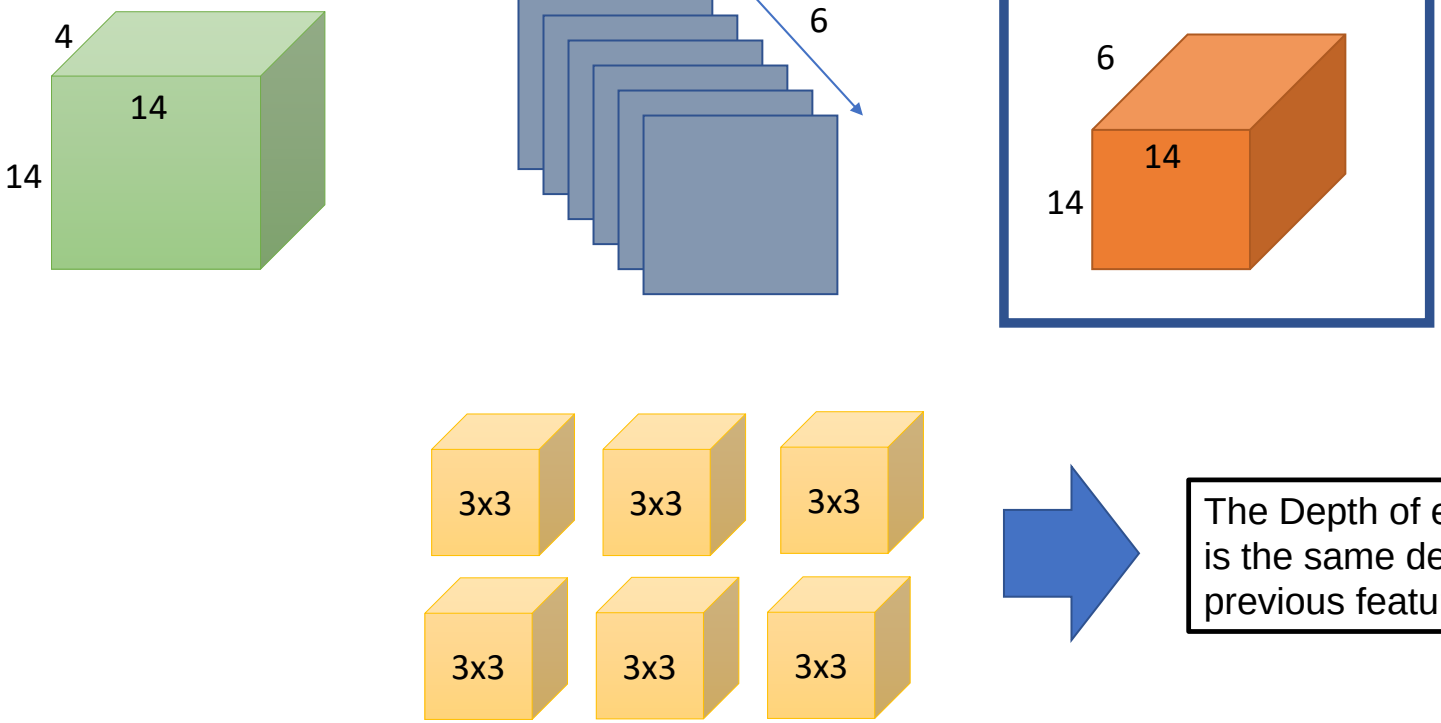








The Depth of each filter is the same depth of the previous feature map. In this case, the previous feature map is just the grayscale image (one channel image). So the depth of the filter is 1.


4
14
14
6
6
14
14
3x3
3x3
3x3
3x3
3x3
3x3
The Depth of each filter is the same depth of the previous feature map: 4

Test Your Understanding

If 16 filters are applied to the image in the first convolution layer, how many resulting feature maps would you have?

- 32
- 1
- 16
- 8

The number of feature maps generated = number of filters applied

If 16 filters are applied to the image at the first convolutional layer, what is the depth of each filter?

- 1
- 16
- 3
- 8

The depth of the filter must follow the depth of the previous feature maps (image with 3 channels in the first layer)

Test Your Understanding

If 16 filters are applied to the image in the first convolution layer, how many resulting feature maps would you have?

- 32
- 1
- 16
- 8

The number of feature maps generated = number of filters applied

If 16 filters are applied to the image at the first convolutional layer, what is the depth of each filter?

- 1
- 16
- 3
- 8

The depth of the filter must follow the depth of the previous feature maps (image with 3 channels in the first layer)

WRAP IT: Consider a CNN with 5 layers. At the fourth layer, the depth of the feature maps is 256. You wish to generate 512 feature maps at the fifth layer. How many filters should you apply, and what is the depth of each filter?

- ○ 512,3
- ○ 512, 256
- ○ 256,512
- ○ 256,3

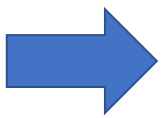
WRAP IT: Consider a CNN with 5 layers. At the fourth layer, the depth of the feature maps is 256. You wish to generate 512 feature maps at the fifth layer. How many filters should you apply, and what is the depth of each filter?

- ○ 512,3
- ● 512, 256
- ○ 256,512
- ○ 256,3

Activation
(Non-Linearity)

ReLU or it's Variants

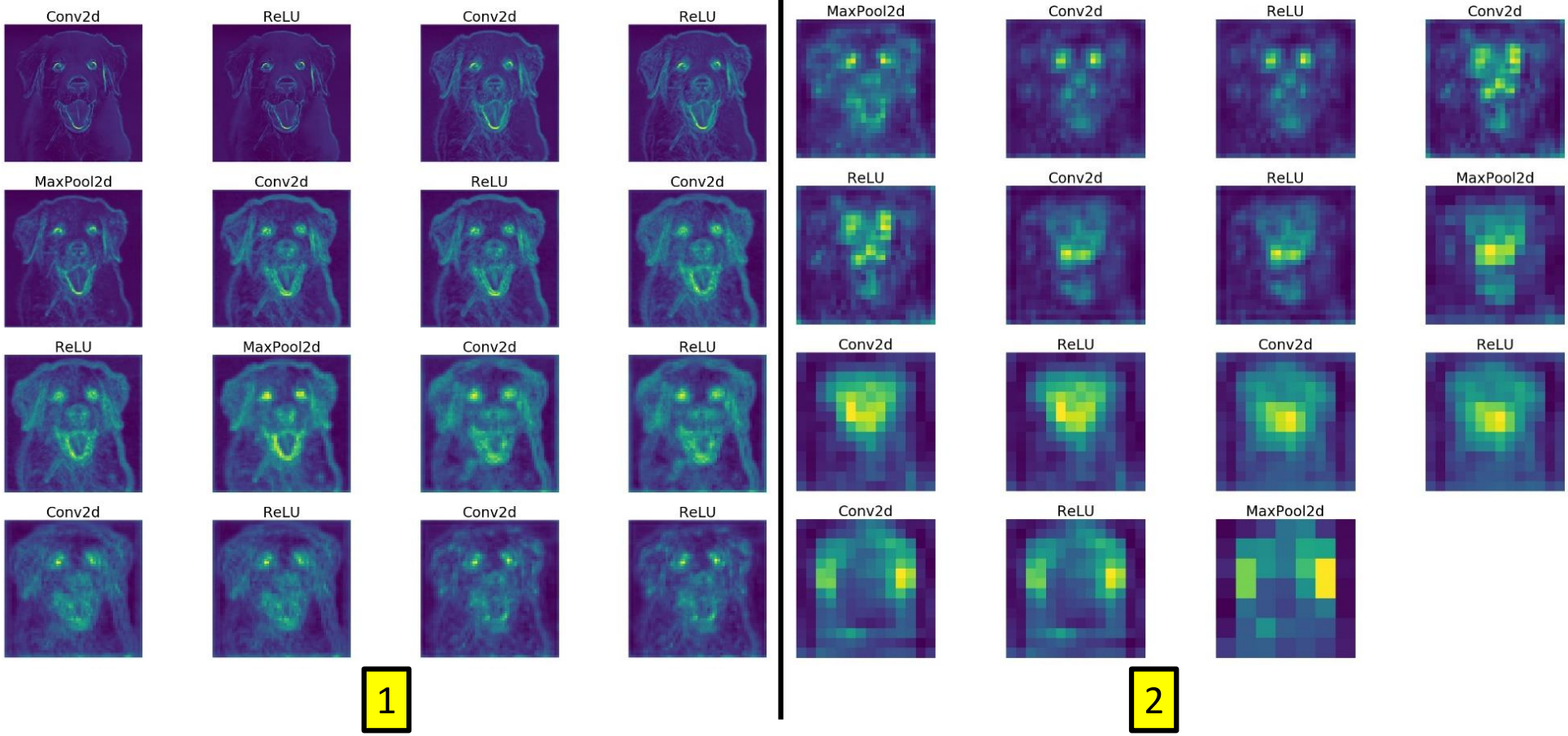
32	4	-1	0
23	1	5	-4
-3	5	-4	5



32	4	0	0
23	1	5	0
0	5	0	5

Feature Map Visualization

The model activates in the features that correspond to a dog



Pooling

Pooling and FC

- Pooling is used to reduce the size of the feature map while still maintaining the important features.
- Remember, the important features are characterized by having the highest values.
- After the CNN process, we can switch back to using Feed Forward layers (Fully-Connected/FC) for processing or classification.

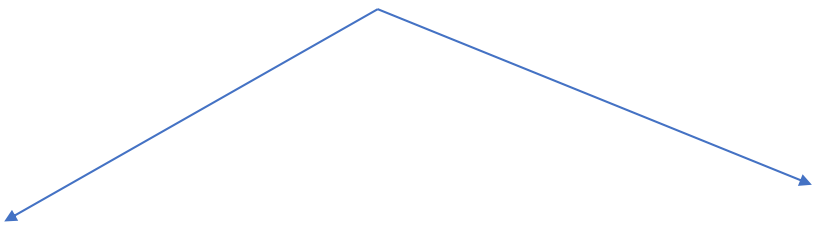
We divide the feature map into blocks (the size of the block is the pooling size), and take the maximum value for each block and hence reducing the size of the feature map but at the same time preserving the important information.

Pooling

If we use 2x2 Max-Pooling, the size of the output =

$$\frac{size\ of\ input}{2}$$

1	0	2	3
9	4	6	8
2	7	1	2
1	2	4	6

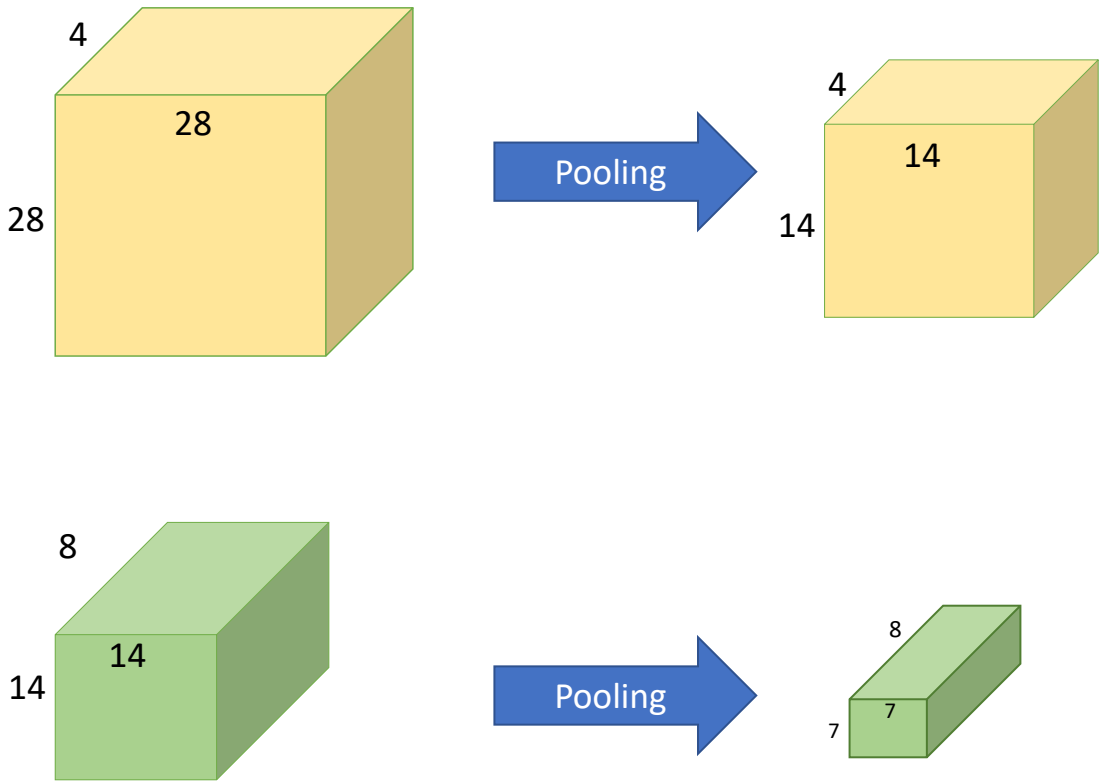


9	8
7	6

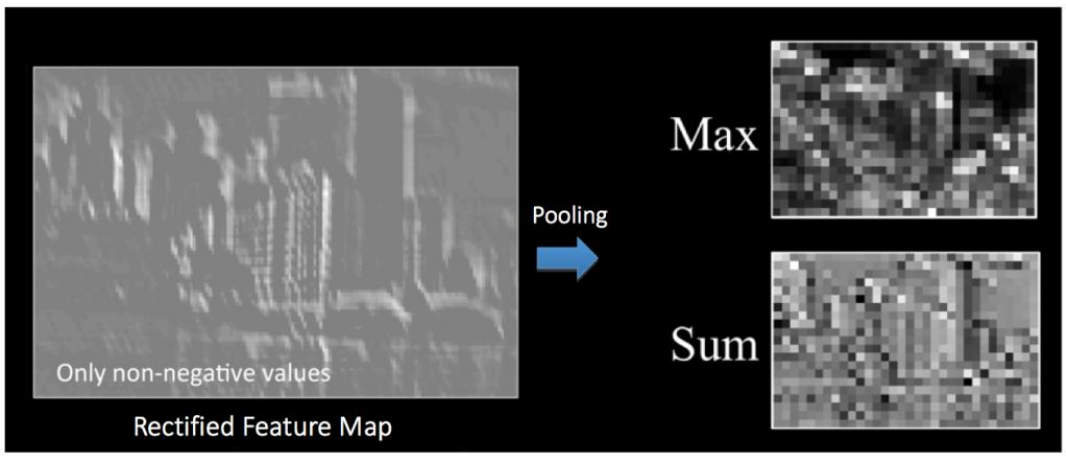
Max Pooling: Take the maximum of the block

3.5	4.75
3	3.25

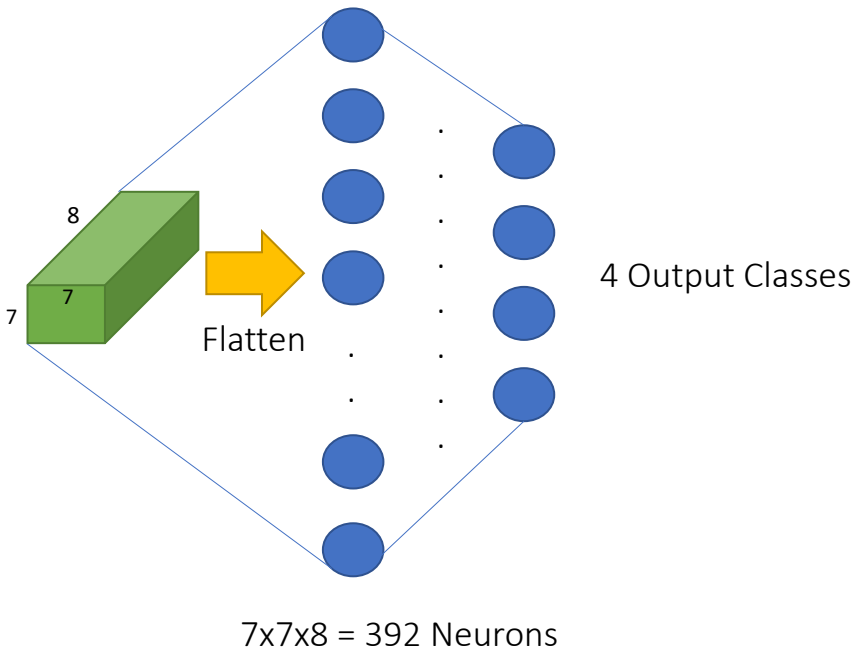
Average Pooling: Take the average of the block


4
28
28
Pooling
4
14
14
8
14
14
Pooling
8
7
7

Result of Pooling

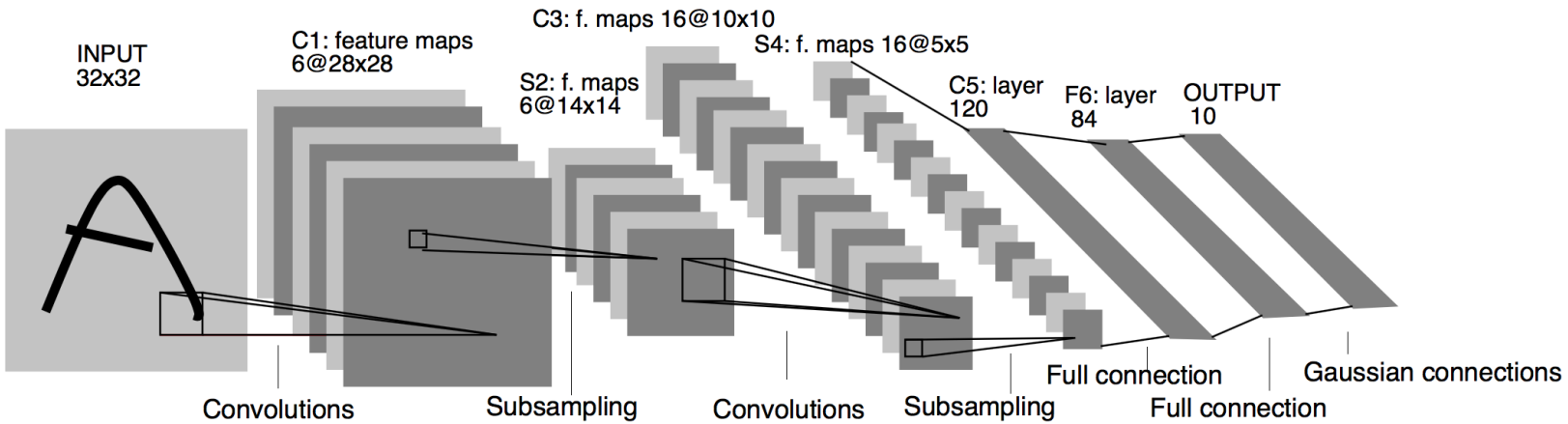


FC (Fully-Connected Layer)



7x7x8 = 392 Neurons

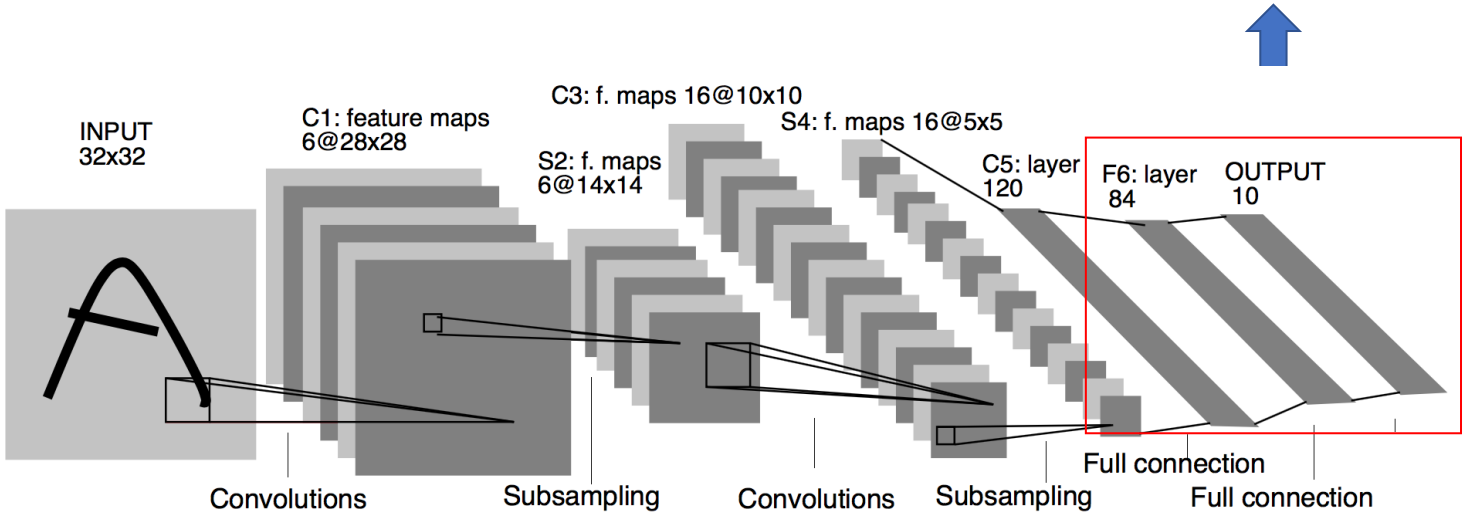
The Complete Picture



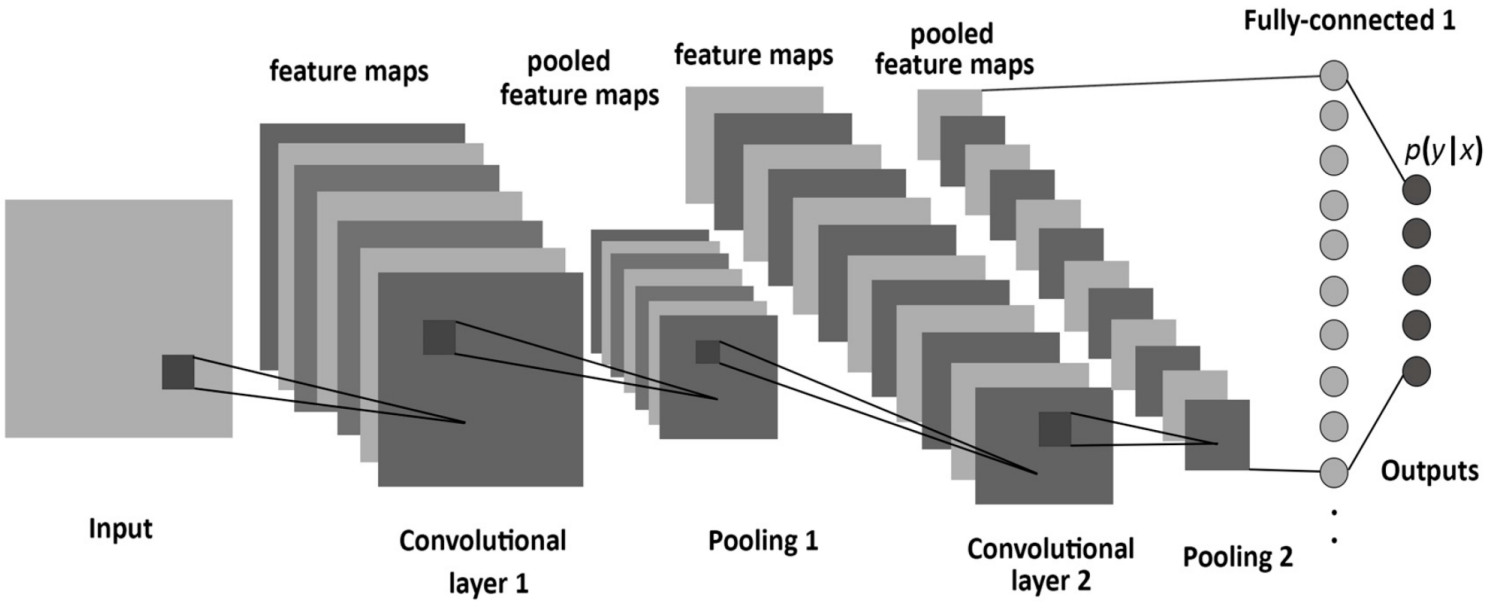
Reference: LeCun Paper, Gradient-Based Learning Applied to Document Recognition

The Complete Picture

Our Feed Forward Network Comes Back Again Here!

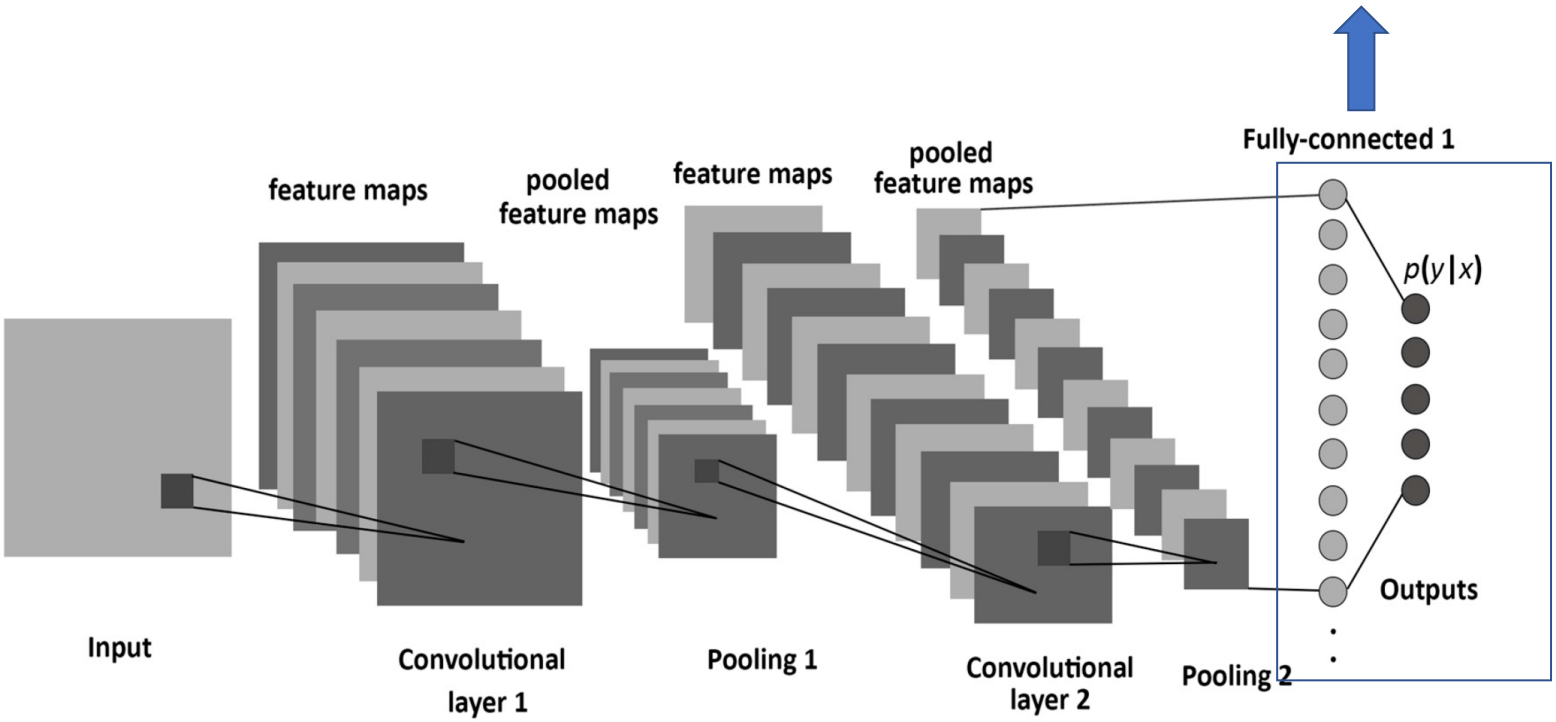


The Complete Picture



The Complete Picture

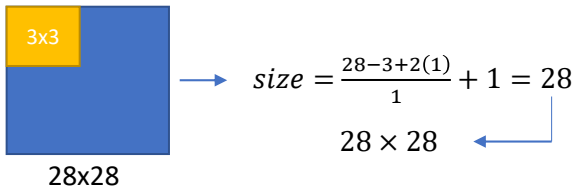
Our Feed Forward Network Comes Back Again Here!



Important Formulas

Output size of the feature map:

$$size = \frac{input\ size\ - filter\ size + 2(padding)}{stride} + 1$$

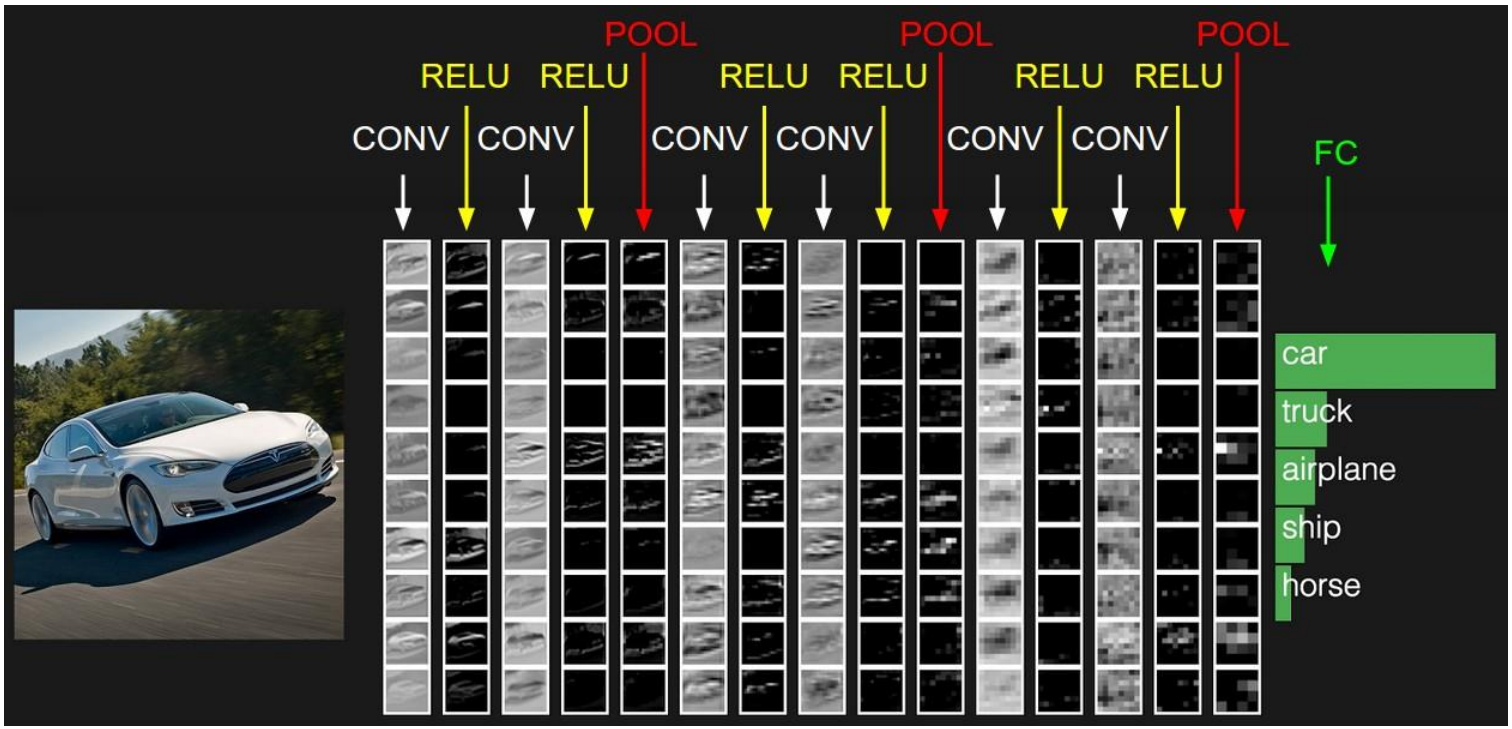


Same Padding: You want your feature map size = to your input size
Valid Padding: Your Feature Map size is less than your input size (You don't apply any padding)

Same Padding:

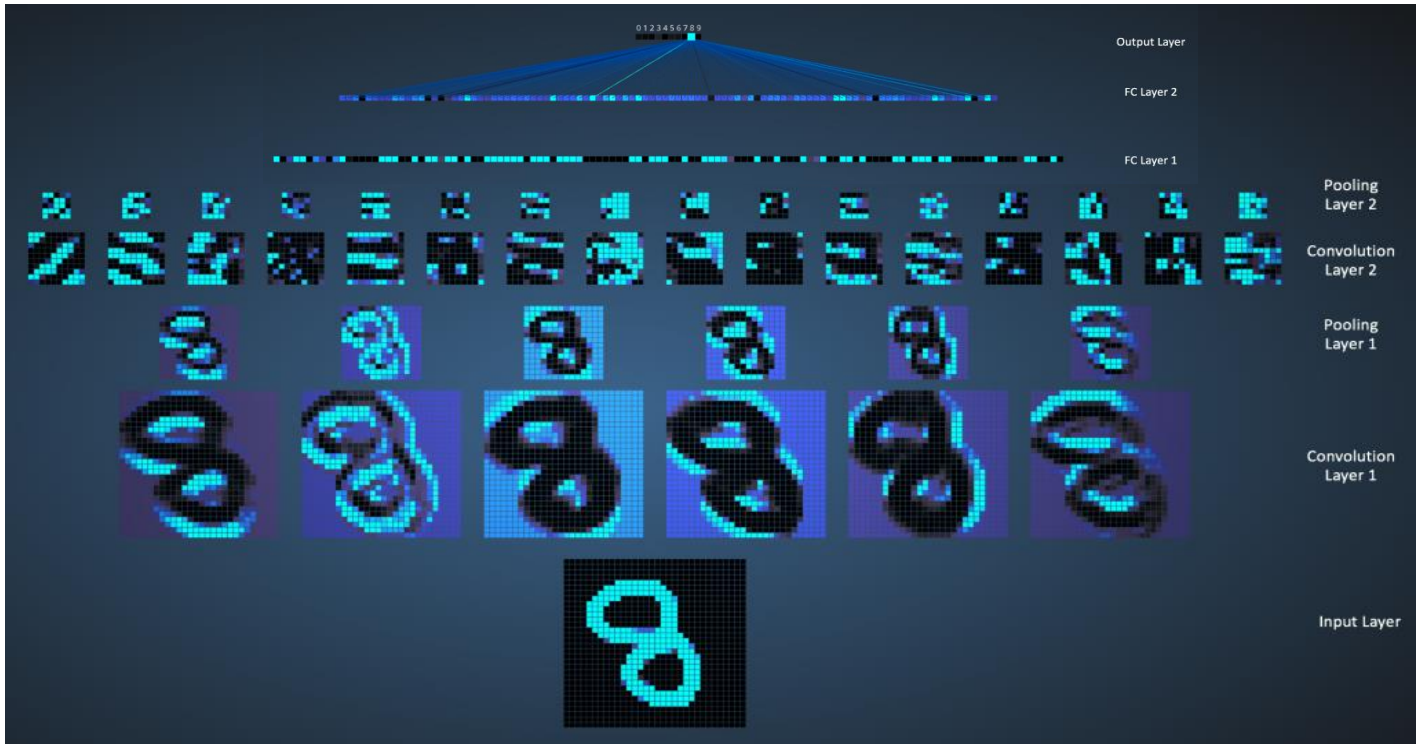
$$padding\ amount = \frac{filter\ size - 1}{2}$$

Visualization



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

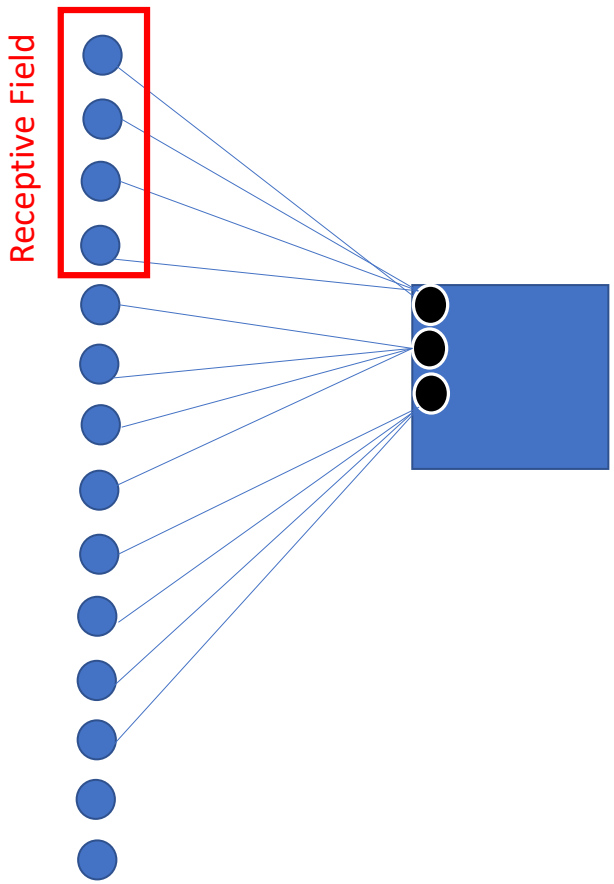
Visualization

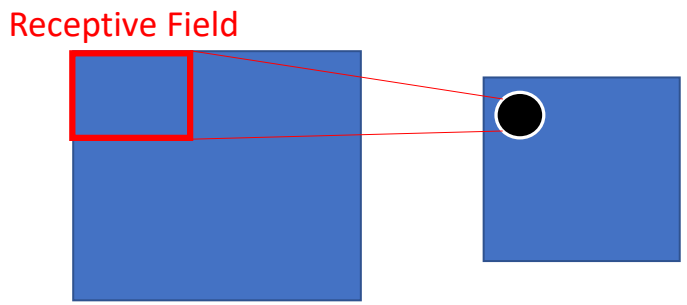


http://scs.ryerson.ca/~aharley/vis/conv/

CNN Main Characteristics

1. Receptive Field





Each unit in a hidden layer is connected to a specific number of units in the previous layer. For instance a node in the first hidden layer will only be connected to a small patch of region of the input image. This region is called the **receptive field.** Note that the **receptive field** is also the **filter size.**

2. Shift Invariance

In simple terms, CNNs are invariant to shifting. Example, if the network if trained on images of a dog at a particular displacement, and at testing time we encounter an image of a dog, but shifted from that of the training (a different displacement), the CNN is stable able to generalize and would still perform as expected. → movements shifts of a couple of pixels do not affect the prediction

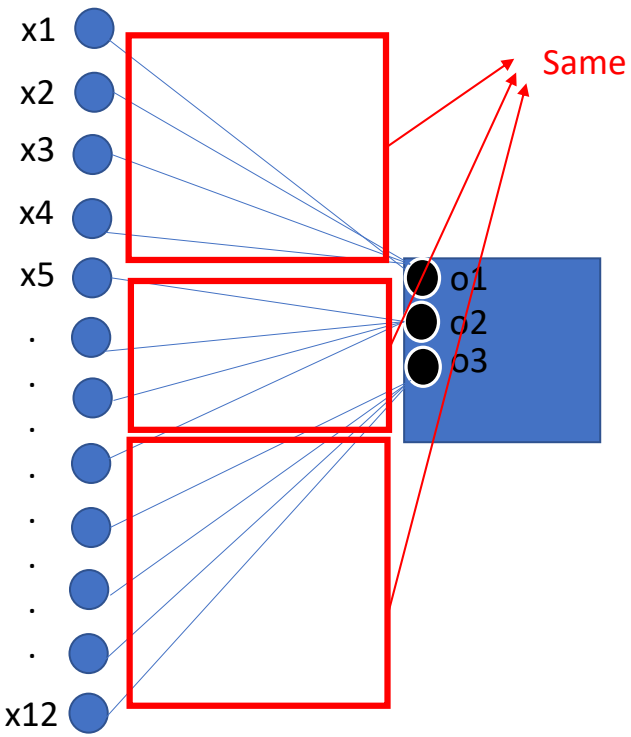
Cat on the Right Side

Cat on the Left Side

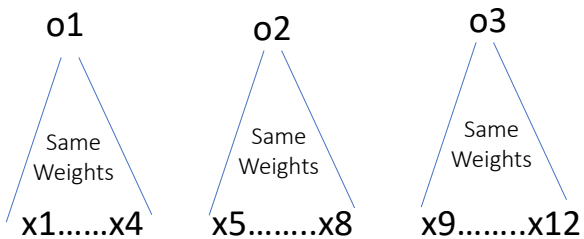
Why?
Filters have learned to extract relevant anywhere in the image, no matter here the object appears.

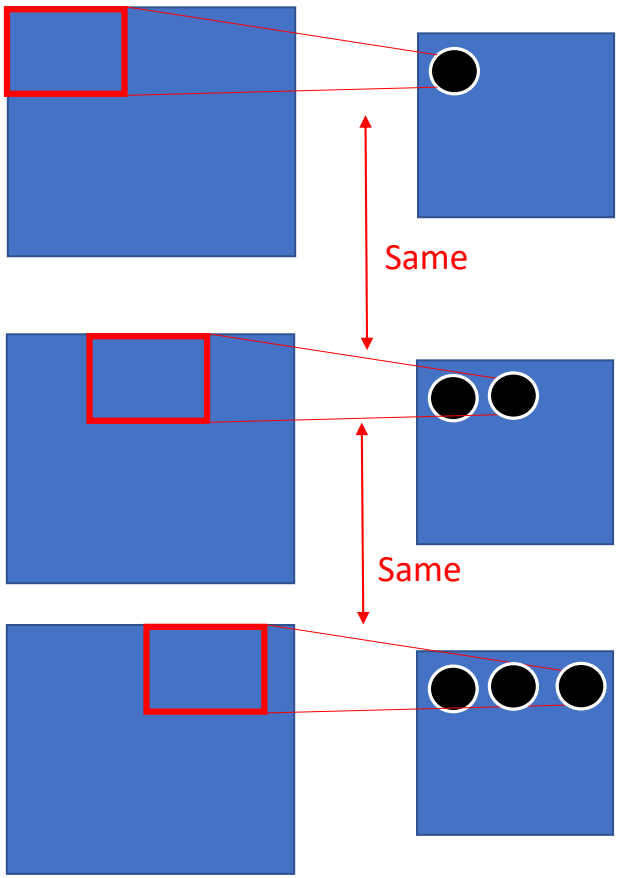
3. Parameter Sharing

- It simply means using the same weight vector to do the convolution operation.



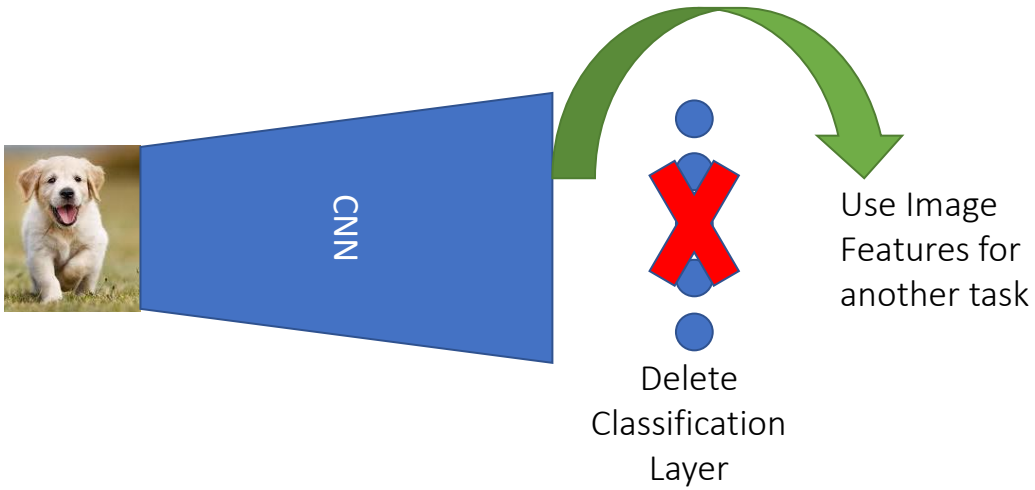
All these weights are one filter convolved with the whole image




Same
Same

4. Feature extraction

Since a CNN is able to learn the filters it needs for the specific application, it extracts image features very well, better than any other feature extraction technique based on image processing. We can also use these features for further applications!



Regularization in CNNs

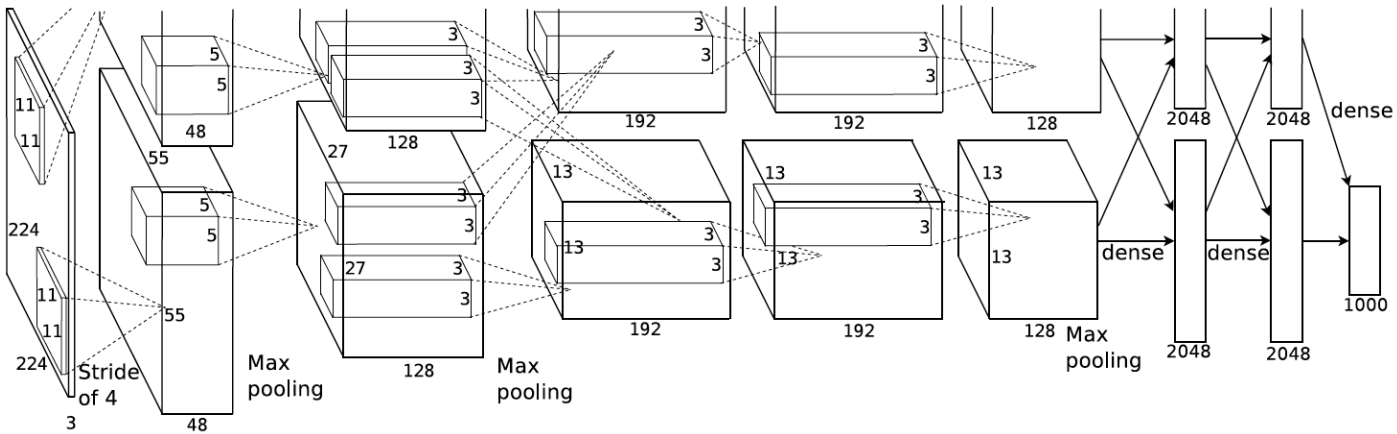
Solution: Batch Normalization

- If the inputs have zero mean and unit variances, the network converges faster.
- Remember, batch normalization normalizes the outputs of a layer according to the batch of samples (which will be input to the next layer). Therefore, there is no shifting around the mean, and the variance doesn't shift → No Covariance Shift

CNN Architectures

AlexNet

ImageNet Classification with Deep Convolutional Neural Networks



- The one that started it all
- 2012 marked the first year where a CNN was used to achieve a top 5 test error rate of 15.4%

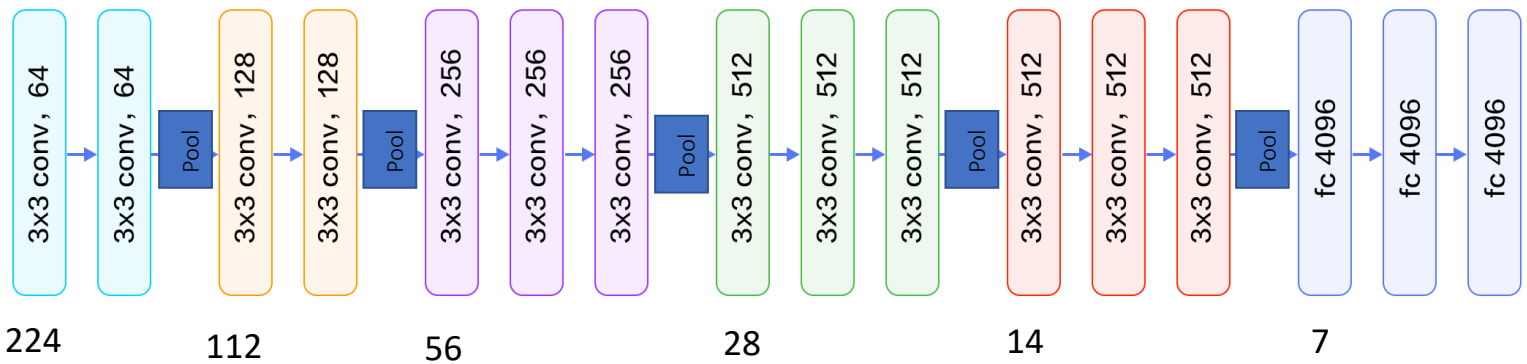
Main Points

- Trained the network on ImageNet data, which contains 1 million images from with 1,000 classes
- Used ReLU for the nonlinearity functions
- Used data augmentation techniques
- Implemented dropout layers in order to combat the problem of overfitting to the training data.
- Trained the model using batch stochastic gradient descent, with specific values for momentum and weight decay.
- Trained on **two** GTX 580 GPUs for **five to six days**.

VGGNet

Very deep convolutional networks for large-scale image recognition

- Simplicity and depth.
- 7.3% error rate
- The use of only 3x3 sized filters
- As the spatial size of the input volumes at each layer decreases, the depth of the volumes increase due to the increased number of filters as you go down the network.
- Worked well on both image classification and localization tasks
- Trained on 4 Nvidia Titan Black GPUs for **two to three weeks**



Motivation of using 3x3 Filters

- Using 3x3 Filters, we can allow the CNN to learn diverse filters and better features than that of using larger filters.
- For example two 3x3 convolution blocks has similar receptive field as a single 5x5 convolution.
- If we have an input of 6x6:

Case 1: Using 2 3x3 Filters

- Convolving it with the first 3x3 filter → Output is of size 4x4
- Convolving it with the second 3x3 filter → Output is of size 2x2

Case 2: Using a single 5x5 Filter

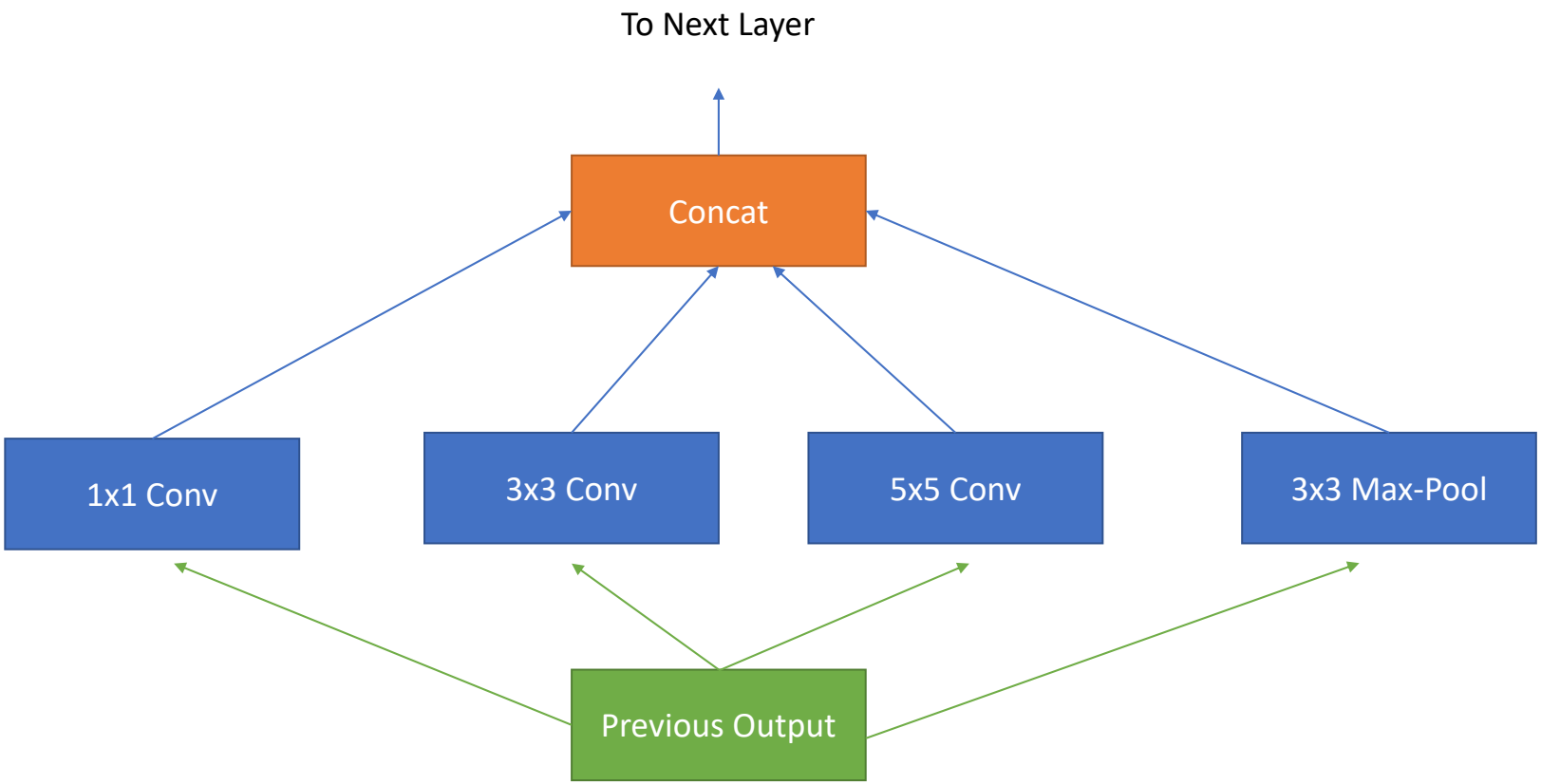
- Convolving it with the 5x5 filter → Output is of size 2x2

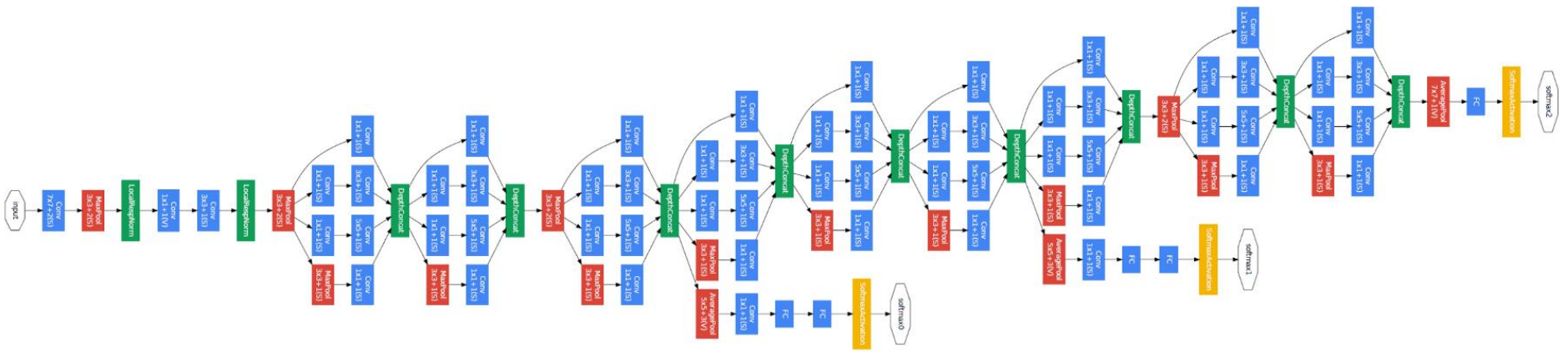
Same Result, but in case 1, we learned more filters and can extract diverse features

InceptionNet (GoogleNet)

Going Deeper with Convolutions

- top 5 error rate of 6.7%
- Key Point: parallel convolutions with output concatenation
- 100 layers in total
- No use of fully connected layers! They use an average pool instead, to go from a 7x7x1024 volume to a 1x1x1024 volume. This saves a huge number of parameters.
- Uses 12x fewer parameters than AlexNet.
- Trained on “a few high-end GPUs **within a week**”.


To Next Layer
Concat
1x1 Conv
3x3 Conv
5x5 Conv
3x3 Max-Pool
Previous Output



Reference: *Going Deeper with Convolutions*

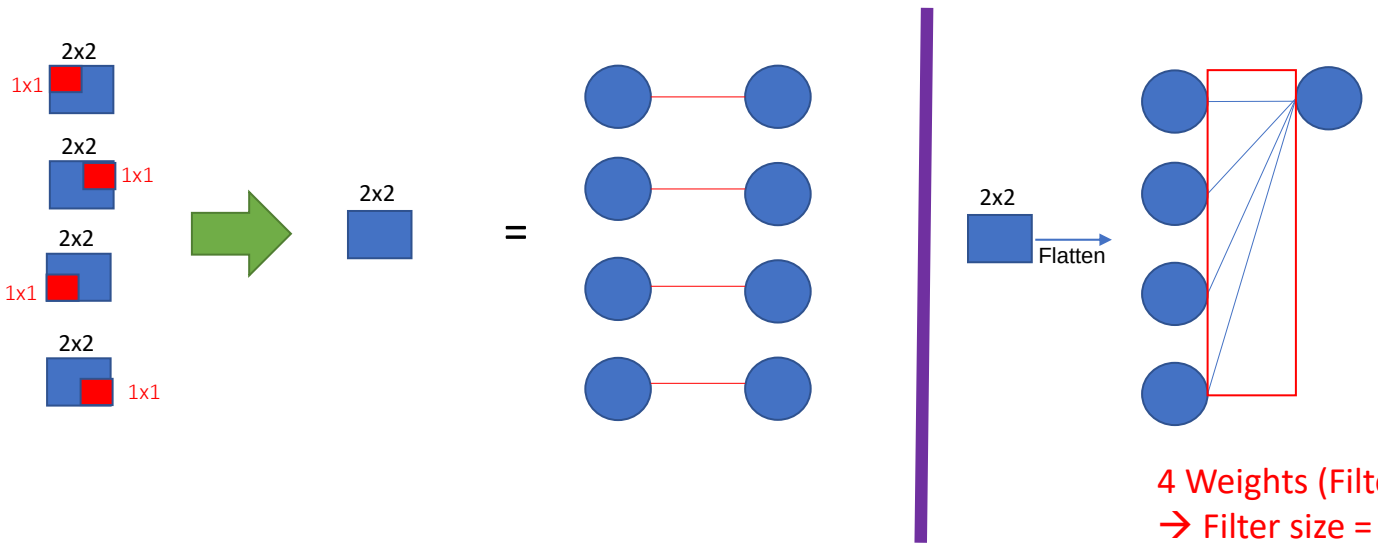
Question:

Is a fully connected neural network same as a 1x1 convolutional neural network?

The community thinks about it in a different way, so some people say yes and some people say no.

For those who say no……

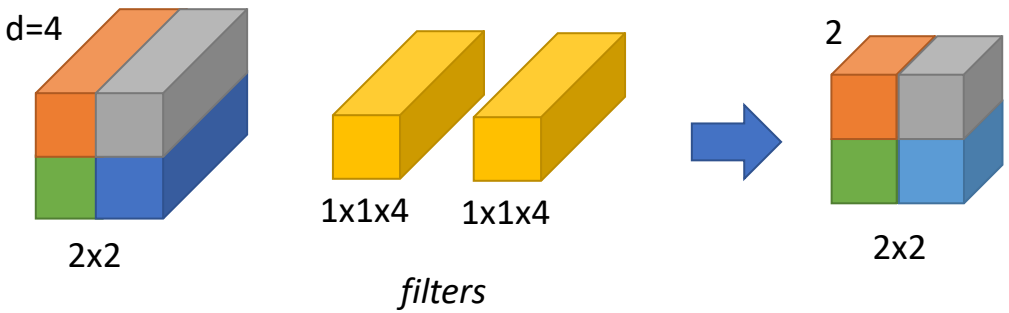
The equivalent to a fully connected network is a CNN with the kernel size the same as input size. This way, every "unit" of the convolutional layer is connected to every element of the previous layer.



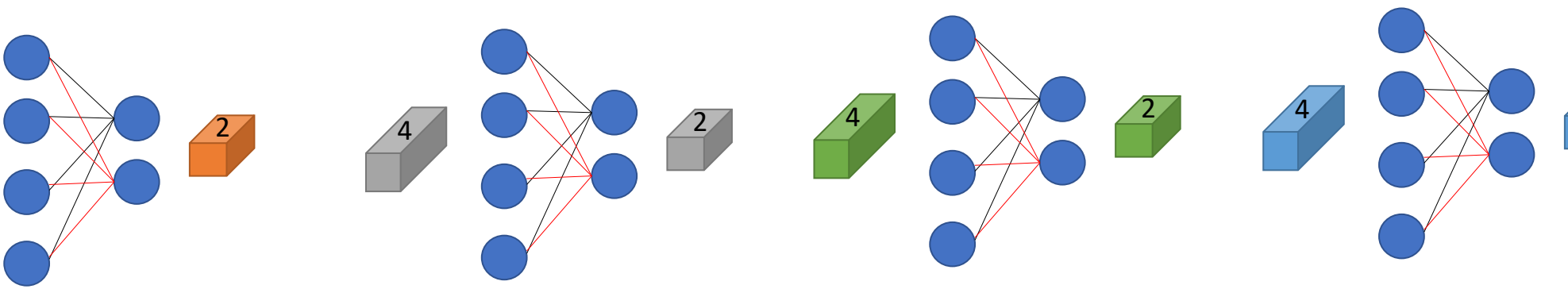
4 Weights (Filter of shape 2x2)
→ Filter size = input size

And for those who say yes.....

Yes is the right answer by the way ☺



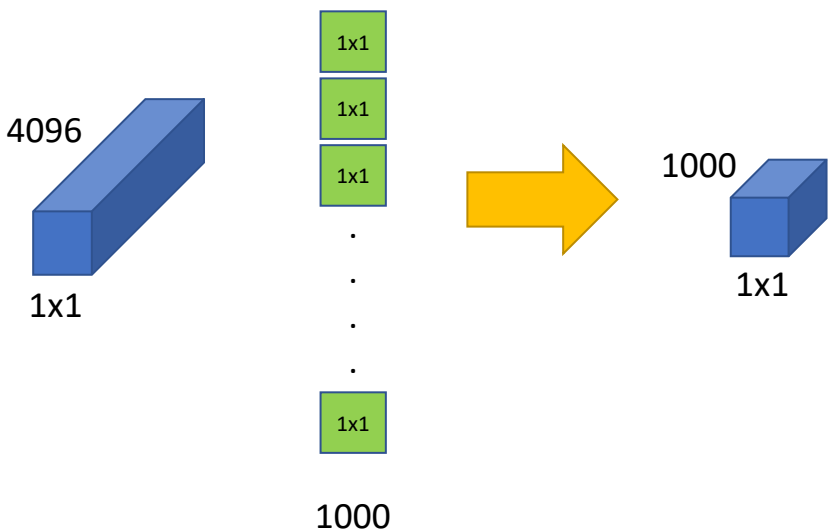
*This is equivalent to applying the **same** FC layer independently on each pixel*



So for those saying no....

You would be right only if your input is 1x1xD

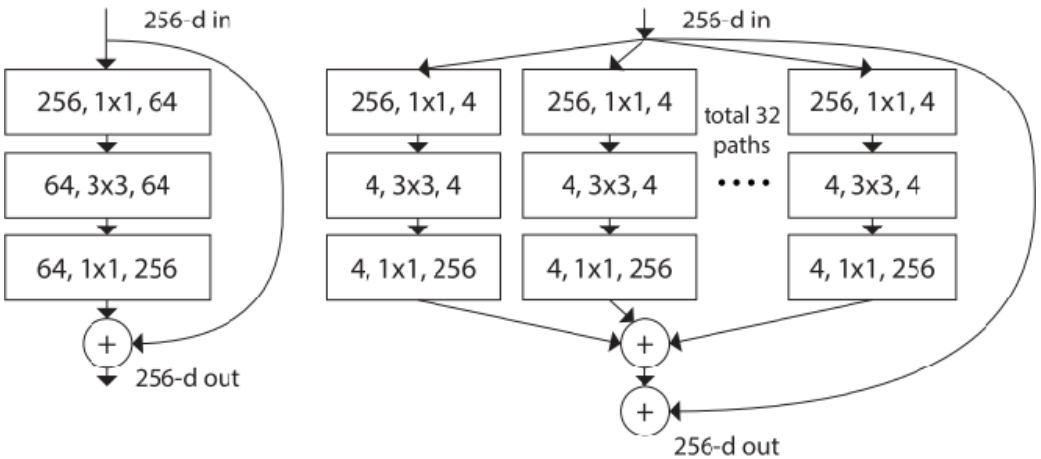
For example, taking the last fc layer of AlexNet which has 4096 features as input (which also means 1x1x4096) and has a layer with 1000 outputs (number of classes)→ This is equivalent to a 1000 different 1x1 conv filters of depth 4096 because the filter size = input size



ResNext

Aggregated Residual Transformations for Deep Neural Networks

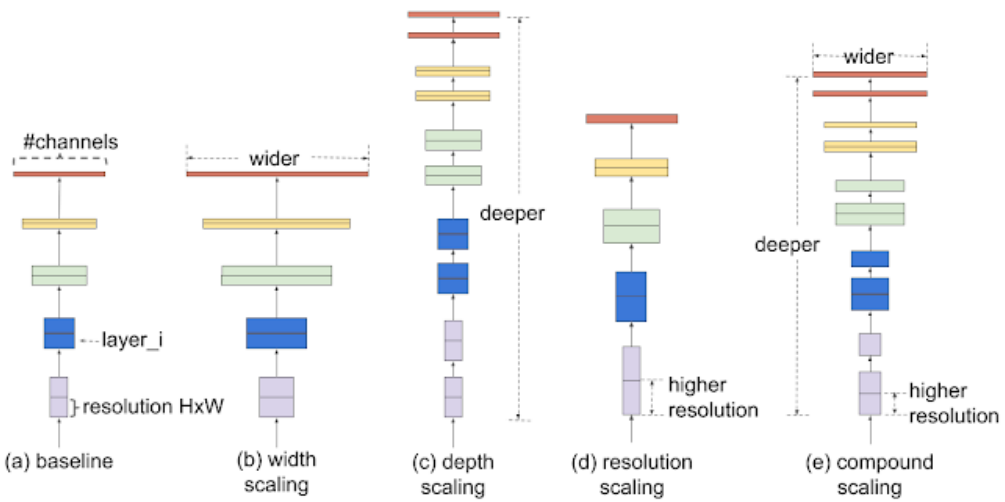
- Combined InceptionNet with ResNet

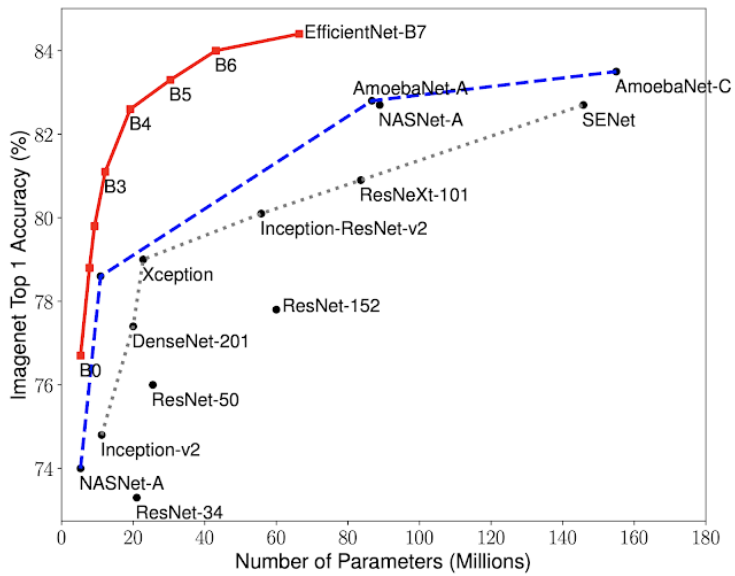


Reference: *Aggregated Residual Transformations for Deep Neural Networks*

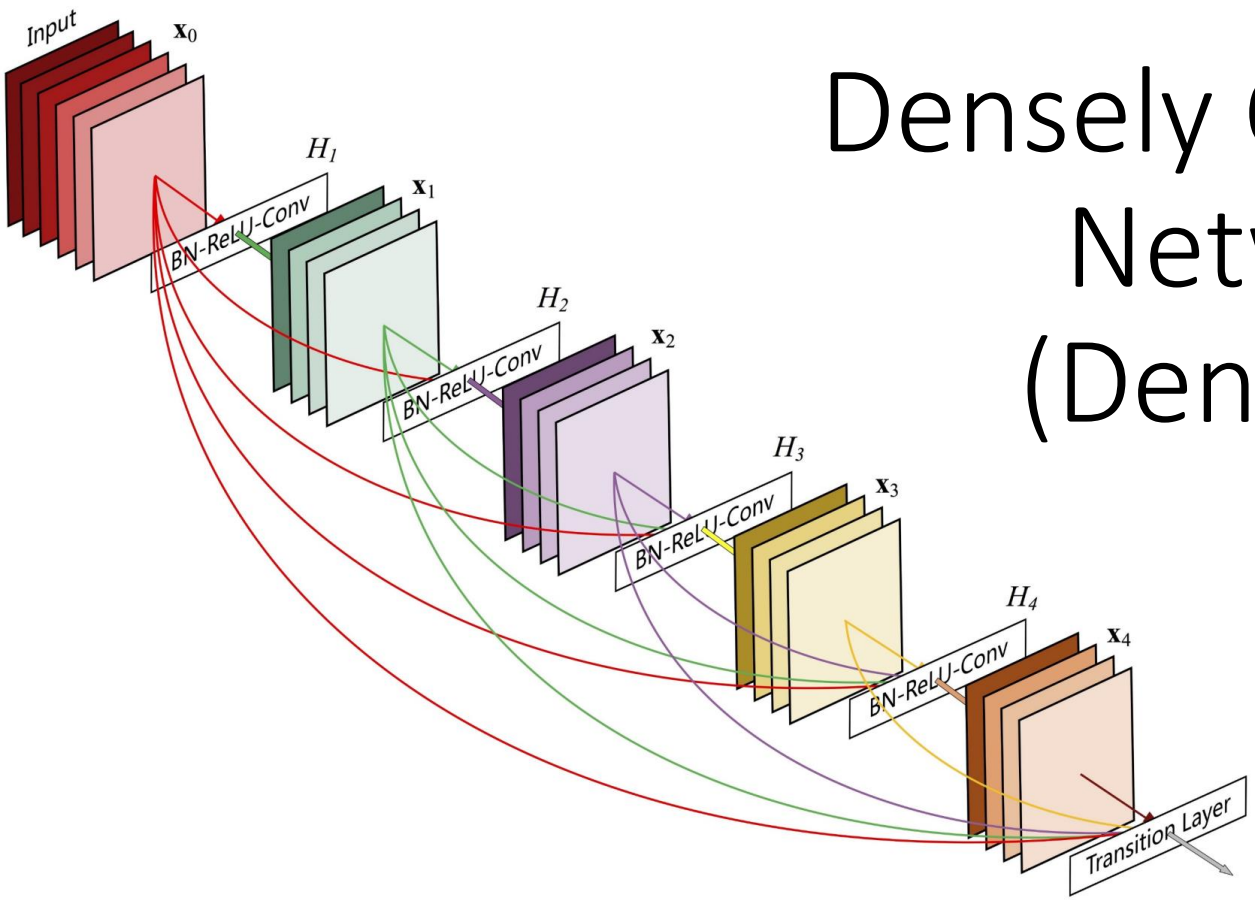
EfficientNet

- ***Key Idea***: Scaling different dimensions (depth, width and resolution). together (balancing them).



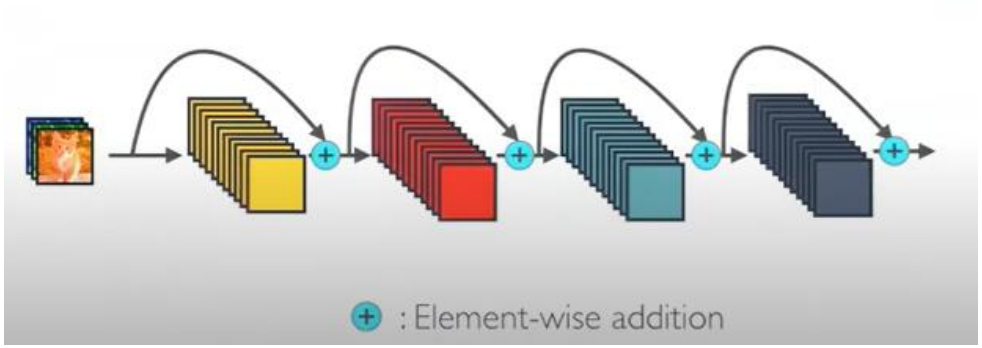


Reference: EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling

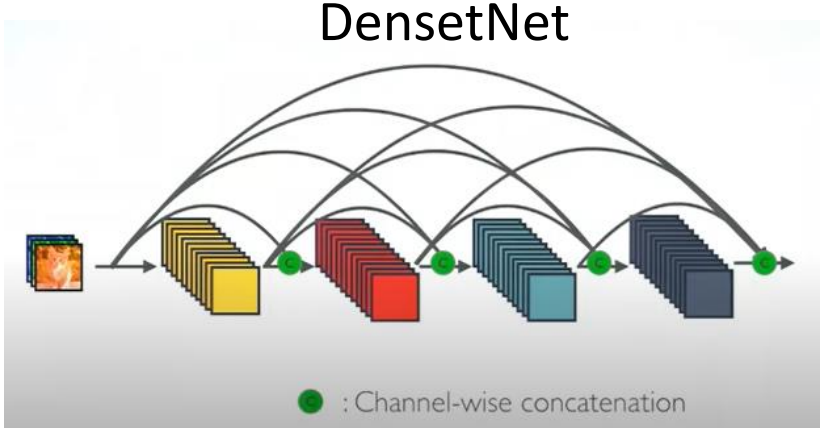


Densely Connected Networks (DenseNet)

ResNet

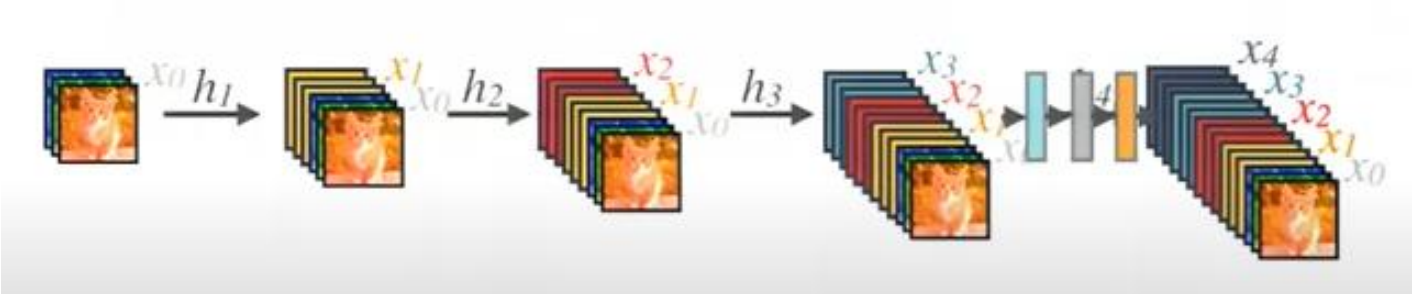


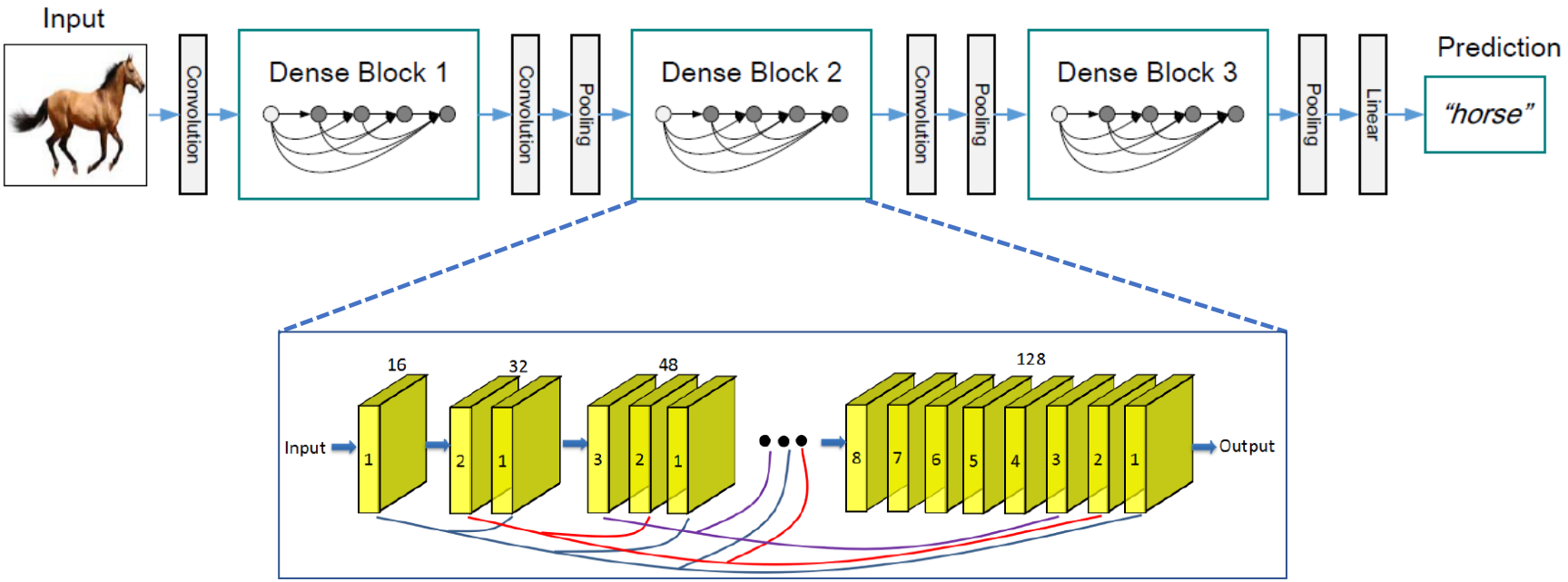
DensetNet



The *i*-th layer receives the feature maps of all preceding layers as input

Each layer receives the feature maps of all previous layers.

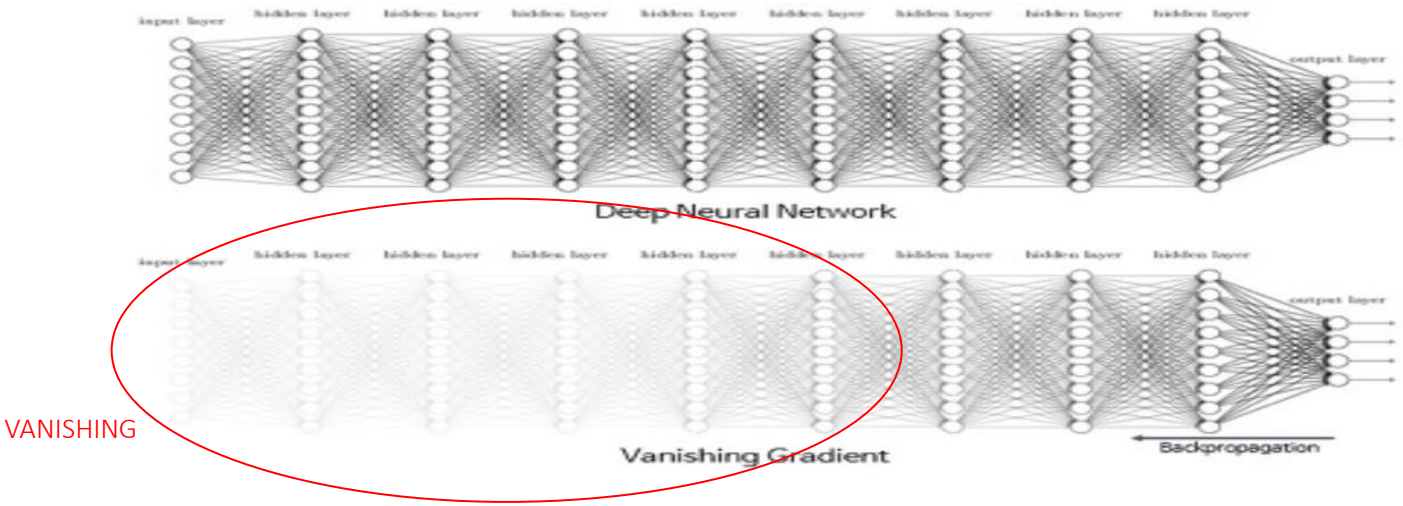




Example: Consider 8 convolution layers in one dense block

When each convolution layer in dense block produce k feature maps as output, the total number of feature maps generated by one block is $k \times 8$, where k is referred to as growth rate.

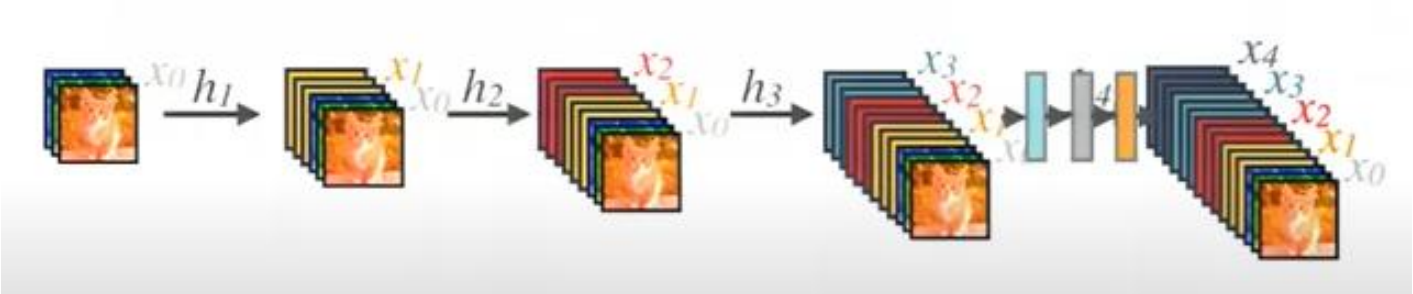
Advantage 1: Solving the Vanishing Gradient Problem



- **The structure of DenseNet alleviates vanishing-gradient problem.**

As networks get deeper, gradients aren't back-propagated sufficiently to the initial layers of the network. The gradients keep getting smaller as they move backwards into the network and as a result, the initial layers lose their capacity to learn the basic low-level features. Several architectures have been developed to solve this problem, including ResNet. They all try to create channels for information to flow between the initial layers and the final layers. DenseNet does the same objective by concatenating previous layers to the next layers and thus allows better gradient flow.

Advantage 2: Feature Preserving and Avoiding Redundancy



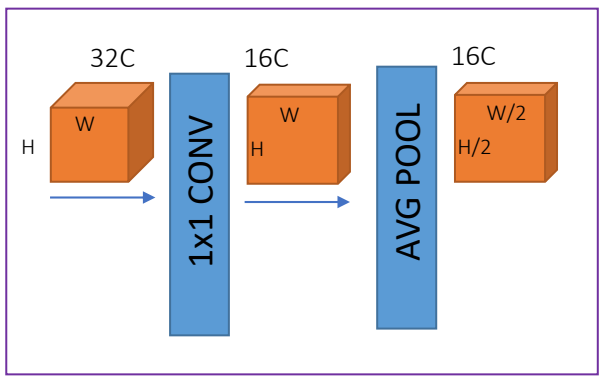
- Stronger feature propagation and feature reuse
- Reusing feature maps that are already learnt forces the current layer to learn complementary information, thus avoiding the learning of redundant features. Because of the dense connections in DenseNet, the model requires fewer layers, as there is no need to learn redundant feature maps, allowing the collective knowledge (features learnt collectively by the network) to be reused

Transition Layer

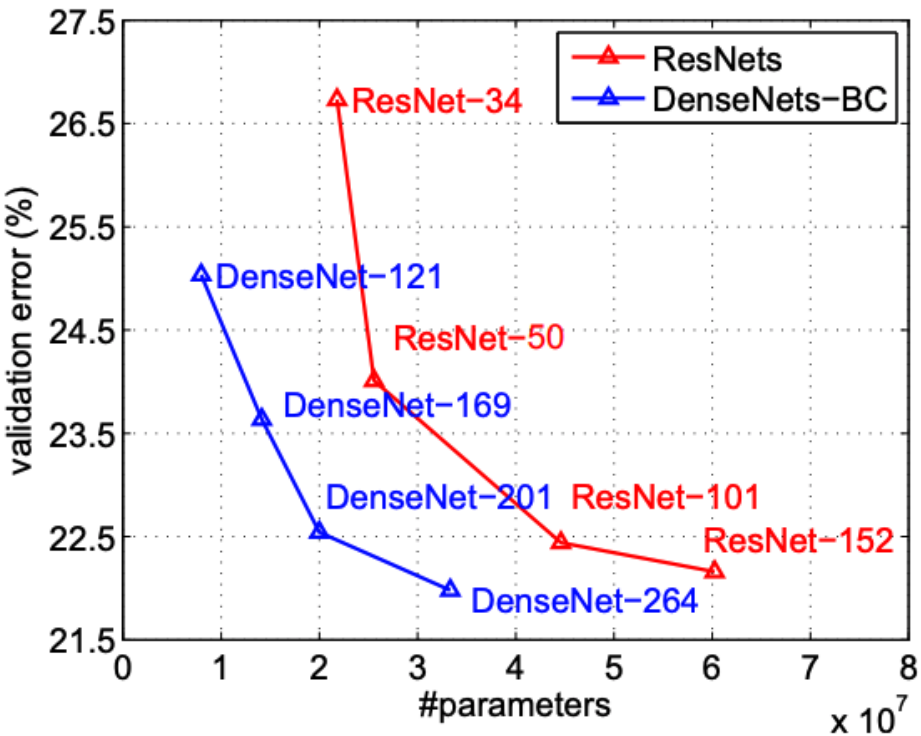
- At the end of each dense block, the number of feature maps gets larger. To bring down the number of feature maps, a **transition layer** is added between two dense blocks, which consists of:

(a) **1 X 1 CONV** operation that reduces the feature maps to half

(b) **2 X 2 AVG POOL** layer which is responsible for downsampling the features in terms of the width and height

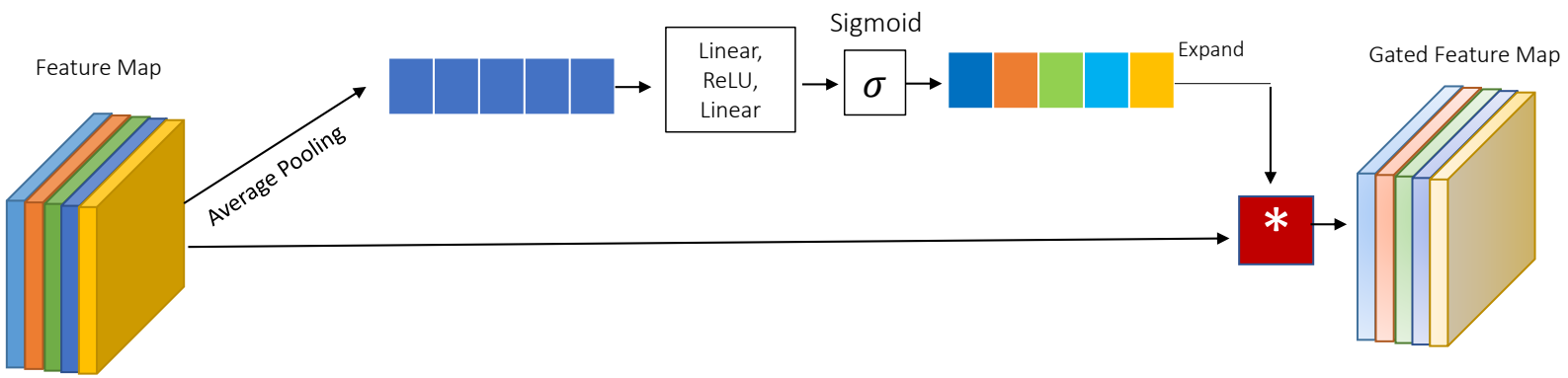


Performance

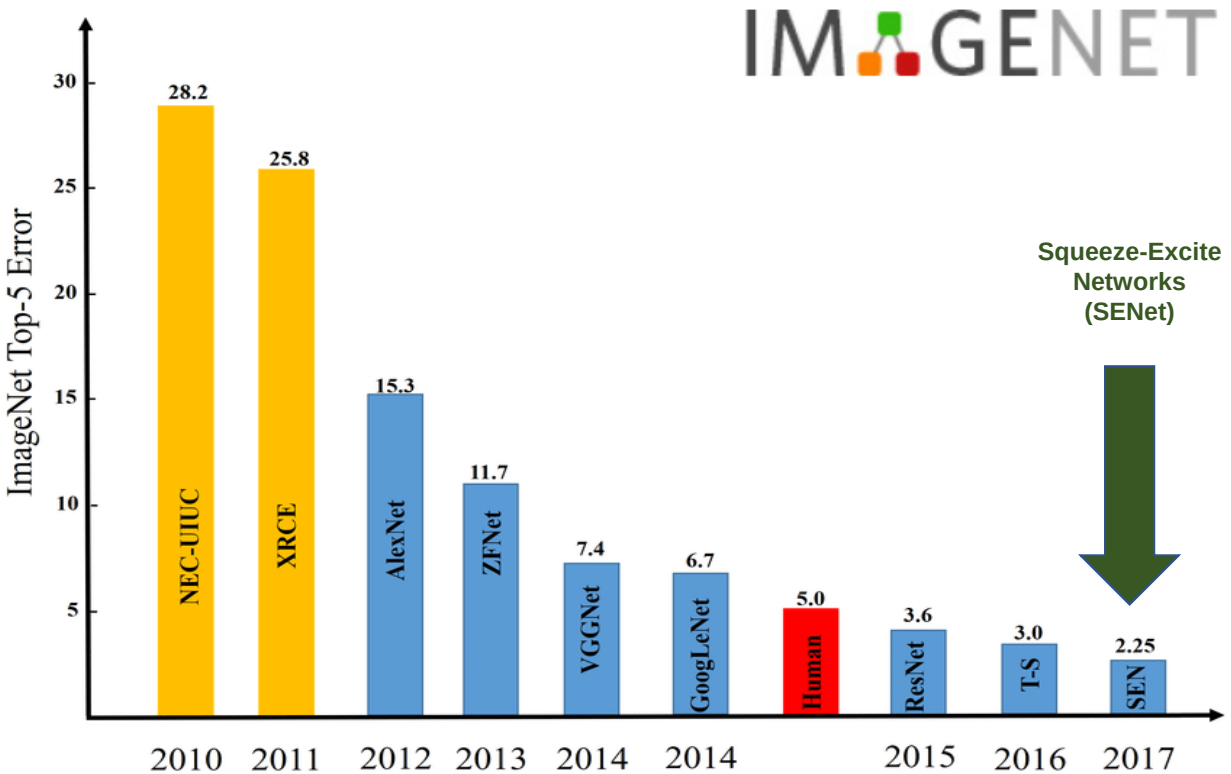


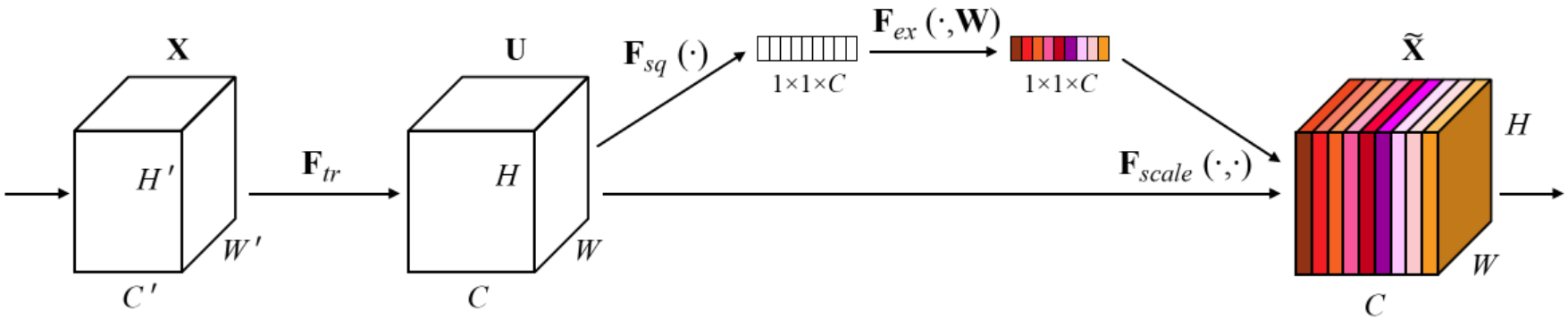
- *Reduced number of parameters*
- *DenseNet model has a significantly lower validation error than the ResNet model with the same number of parameters.*

Squeeze-Excite Networks (SENet)

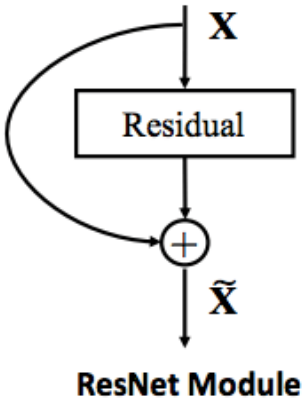


Winner of 2017 ImageNet Challenge

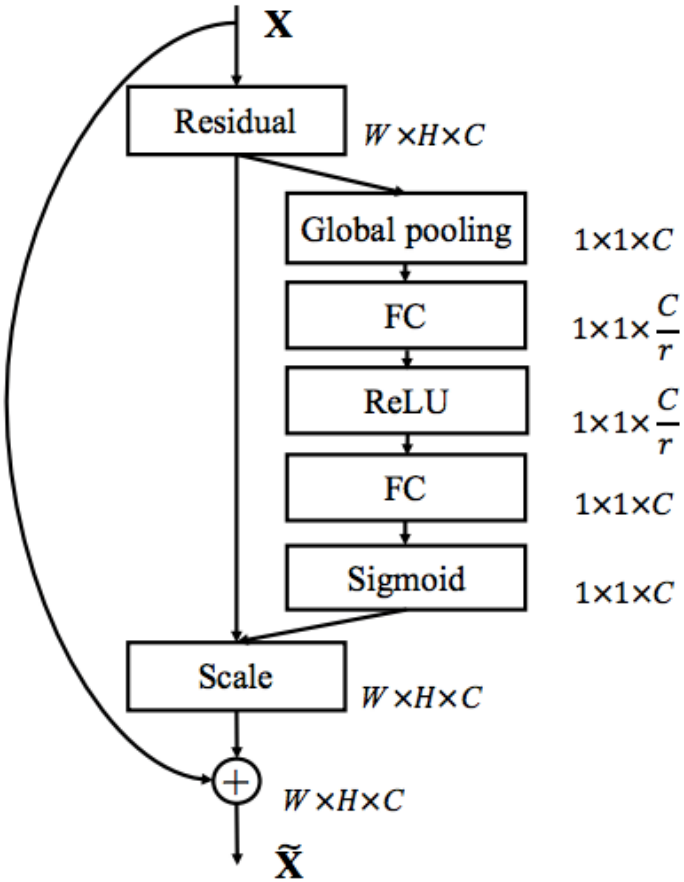




- Usually, a network weights each of its channels equally when creating the output feature maps. SENets change this by adding a content aware mechanism to weight each channel adaptively.
- First, a global understanding of each channel is acquired by applying average pooling, and thus squeezing each feature map to a single numeric value. This results in a vector of size *c*, where *c* is the number of feature maps. Afterwards, it is fed through a two-layer neural network for further processing, and the final layer is activated by a logistic function and outputs a vector of the same size (*c*). This vector is expanded to the same shape as the input feature map (each value is expanded to HxW), and can now be used as a gate on the original features maps, weighting each channel based on its importance.



ResNet Module



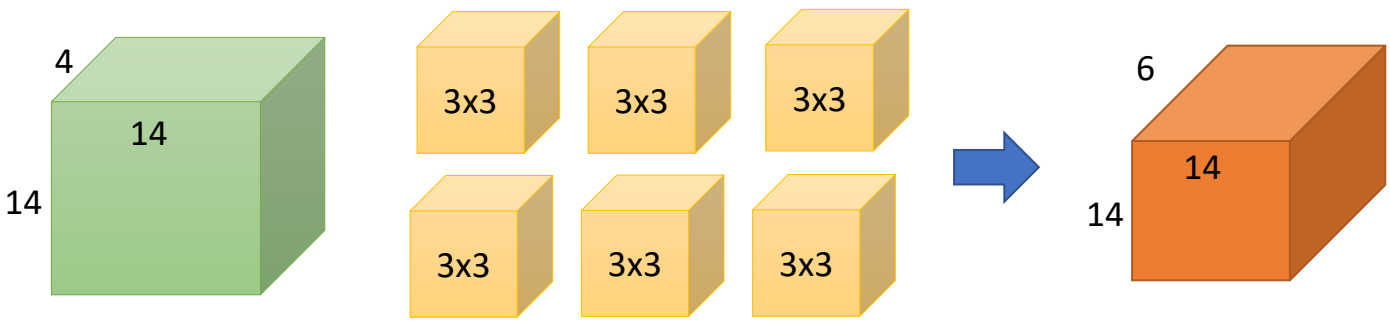
SE-ResNet Module

```python
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel // reduction, channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)
```

Recap on Conventional Convolutions

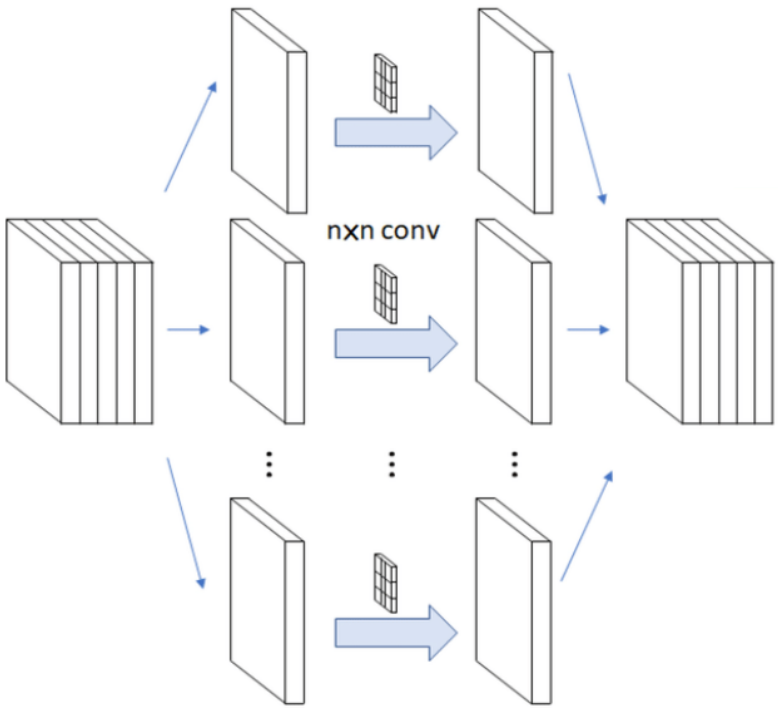
- In conventional convolutions , we learn k filters, each operating on the depth of the feature maps
- Therefore, each filter has the same depth as the input (the number of input feature maps)



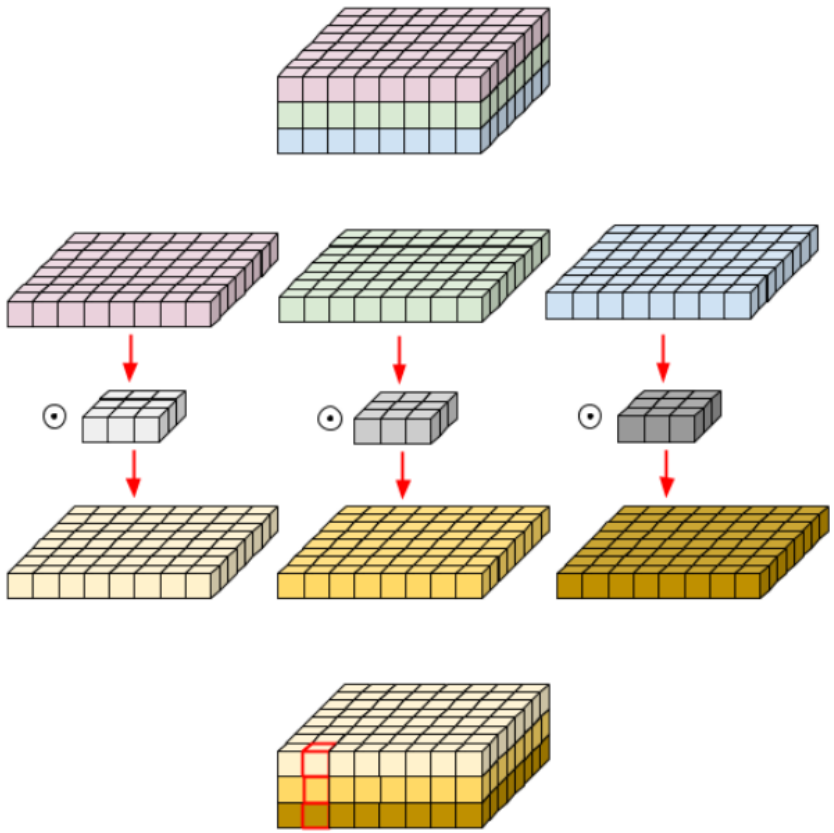
The Depth of each filter is the same depth of the previous feature map: 4

Separable Convolutions

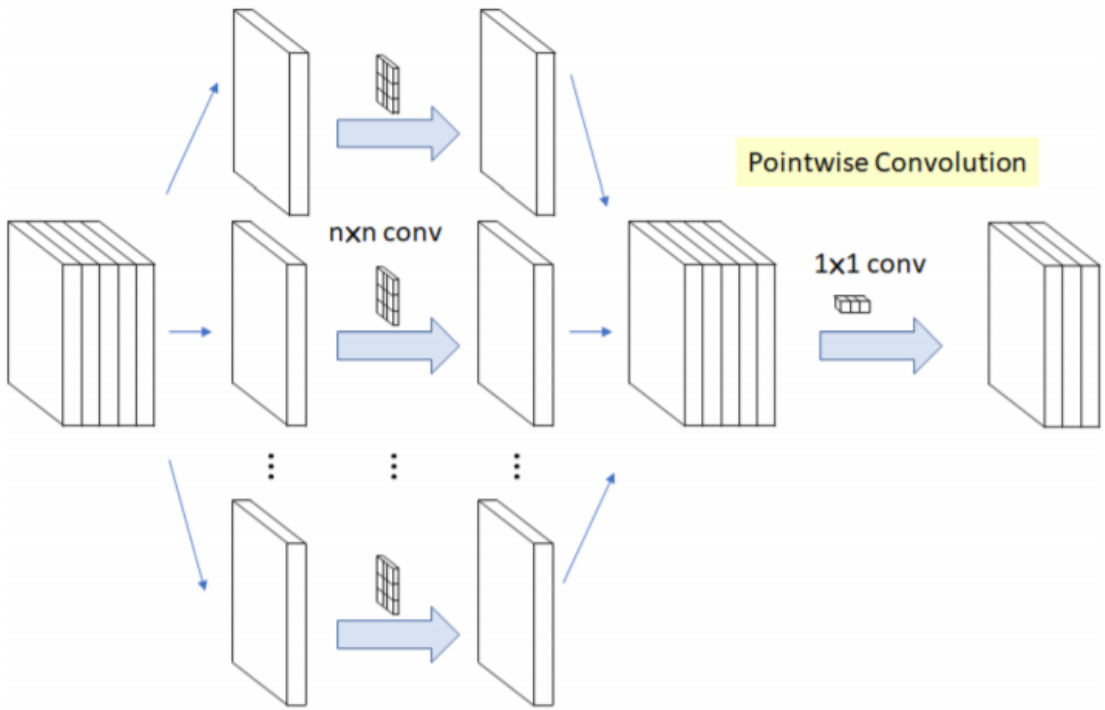
Learn a separate filter for each feature map, and combine the outputs of all



A better visualization



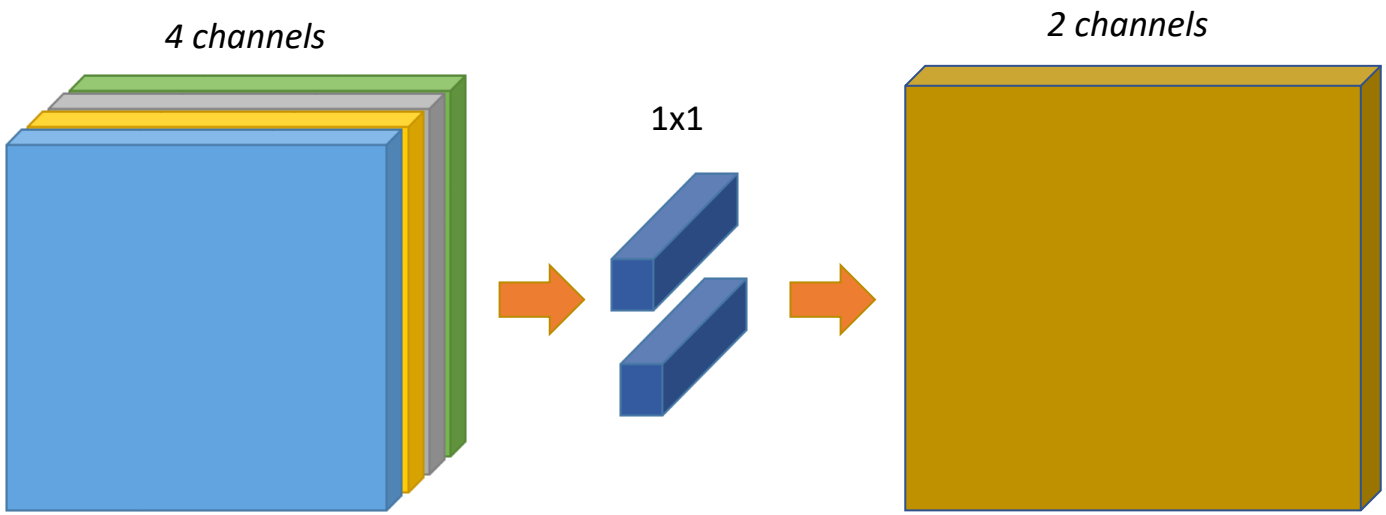
Usually, separable convolutions are combined with pointwise convolutions



Recap on Pointwise Convolution

Using n convolutional filters with a size of 1x1

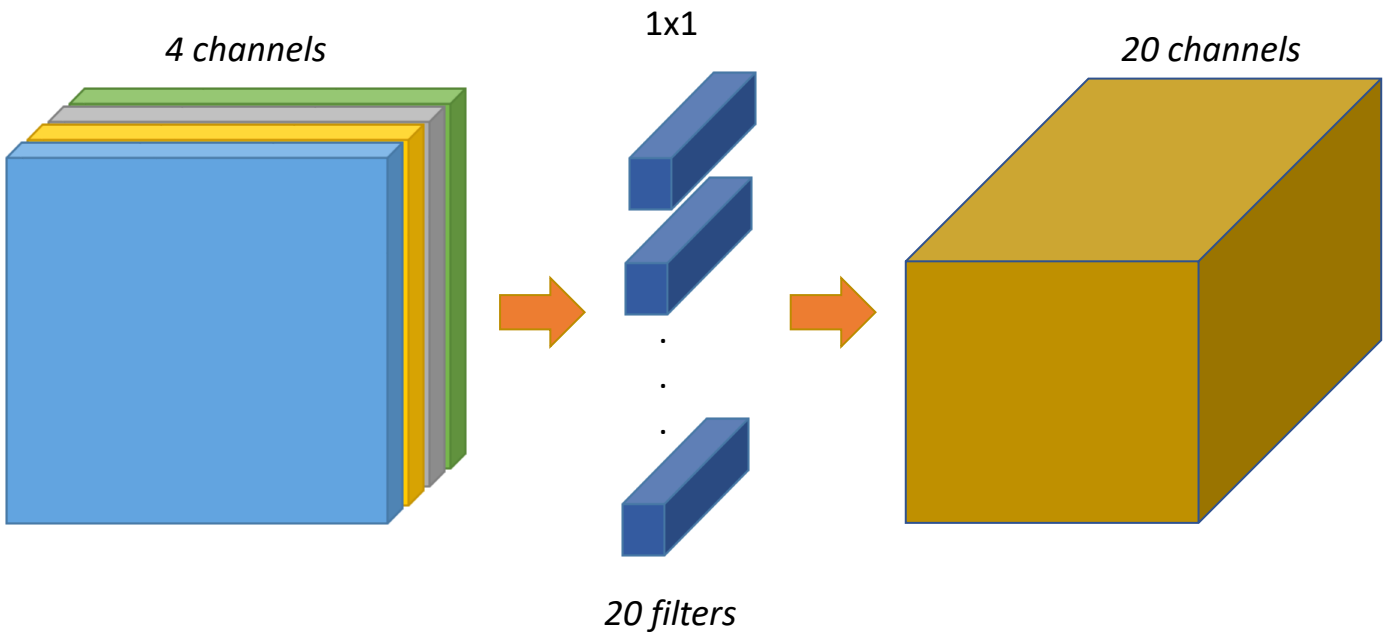
Reducing the dimension



Recap on Pointwise Convolution

Using n convolutional filters with a size of 1x1

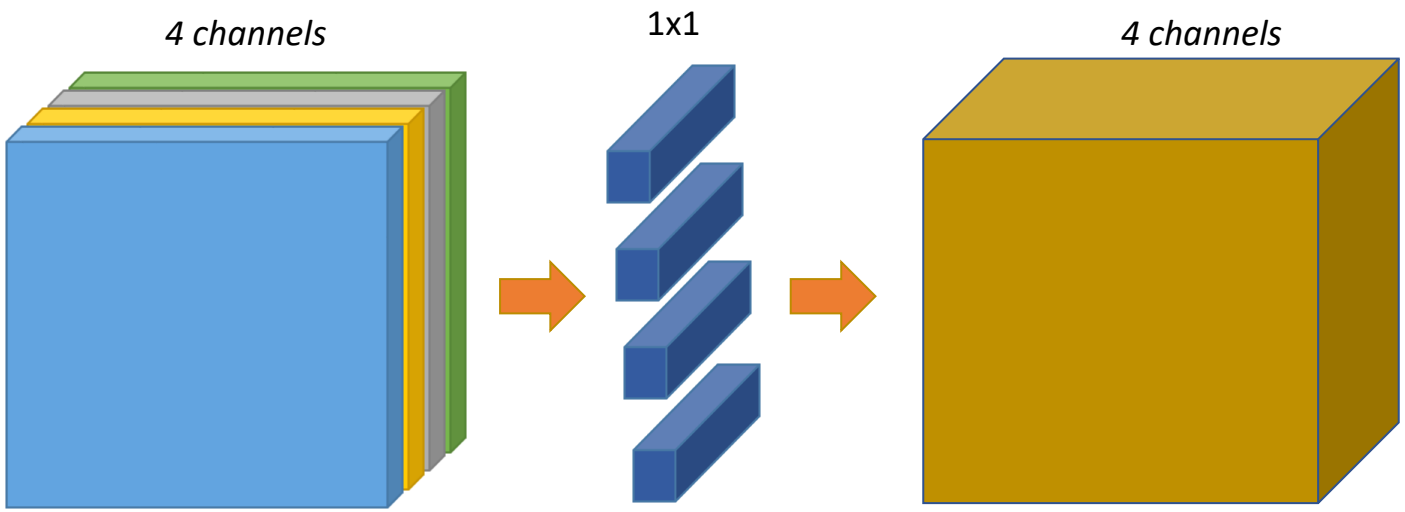
Increasing the dimension



Recap on Pointwise Convolution

Using n convolutional filters with a size of 1x1

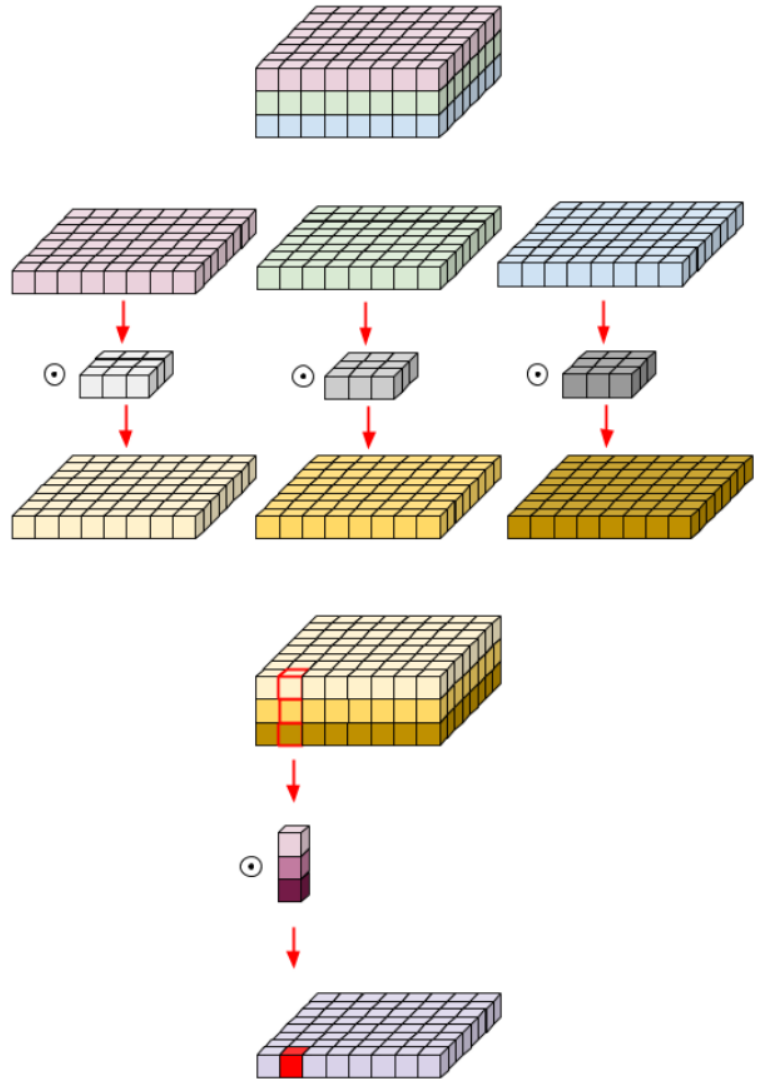
Keeping the dimension



Why do we usually include pointwise convolution after the separable convolution?

1- To play with the dimensionality (increase, decrease)

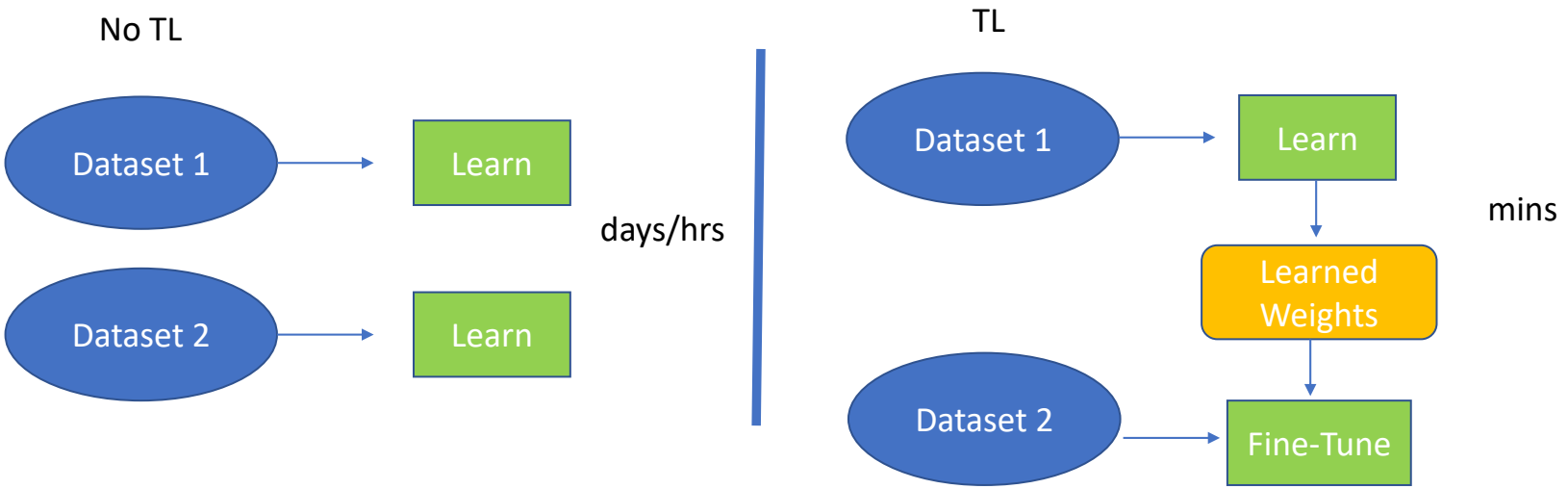
2- To interact between the feature map (depth-wise) which separable convolution does not perform



Transfer Learning

- Transfer Learning is referred to the task of utilizing a pre-trained model, and fine-tuning that model for a related task of the **same nature.**

Example: We may utilize a CNN trained on Image-Net, and fine-tune it to recognize other images not included in the Image-Net Dataset. Training from scratch is eliminated. We don't have to train it from scratch, but rather fine-tune it. This saves a lot of time.



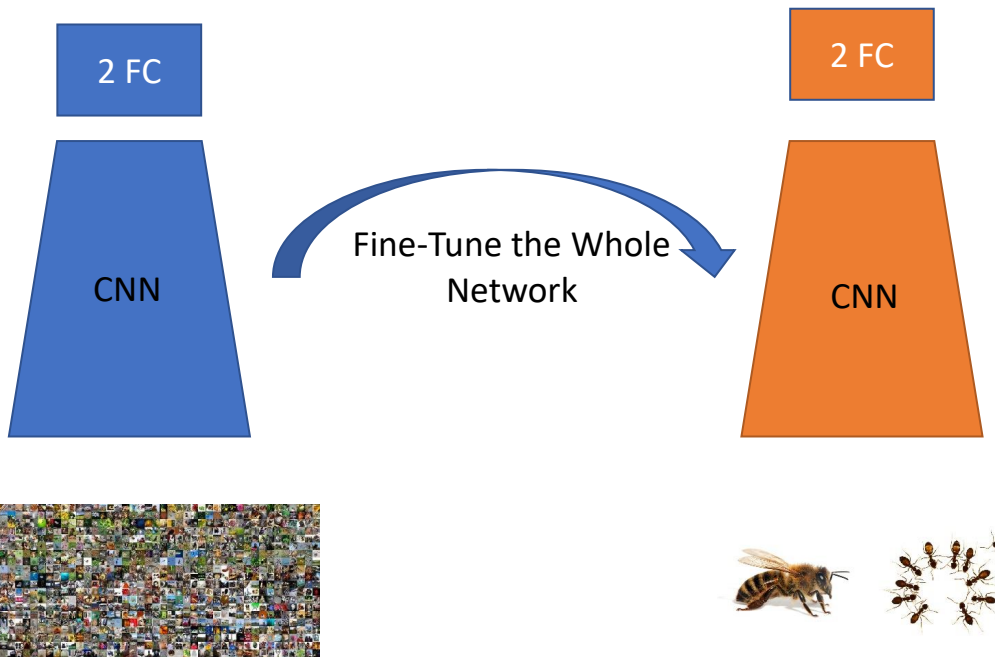
Three Ways of Transfer Learning

1- Fine-Tuning the Whole Network

2- Freeze First Part of the Network and Fine-Tuning the Last Layers of the Network

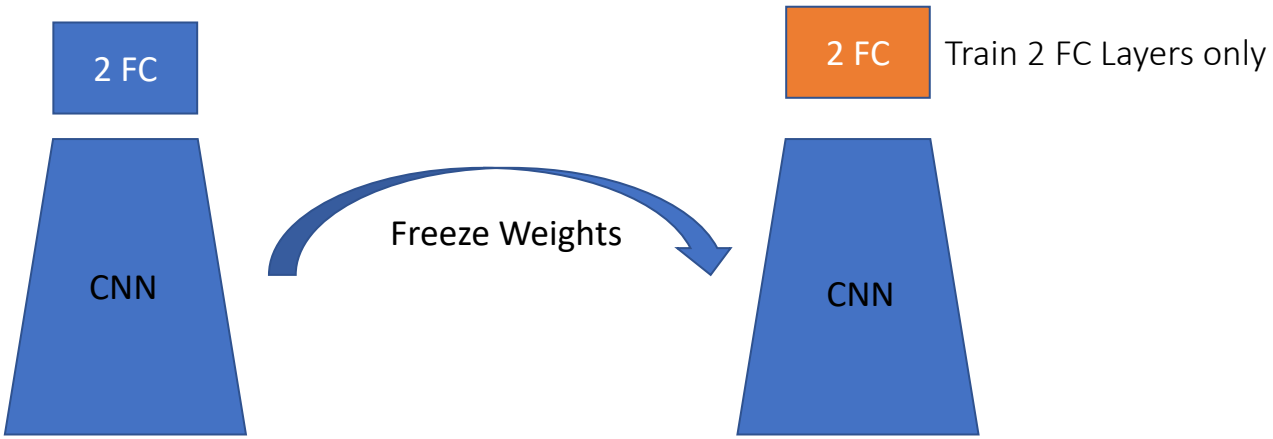
3- Freeze First Part of the Network and Train the Last Layers from scratch

Fine-Tuning the Whole Network



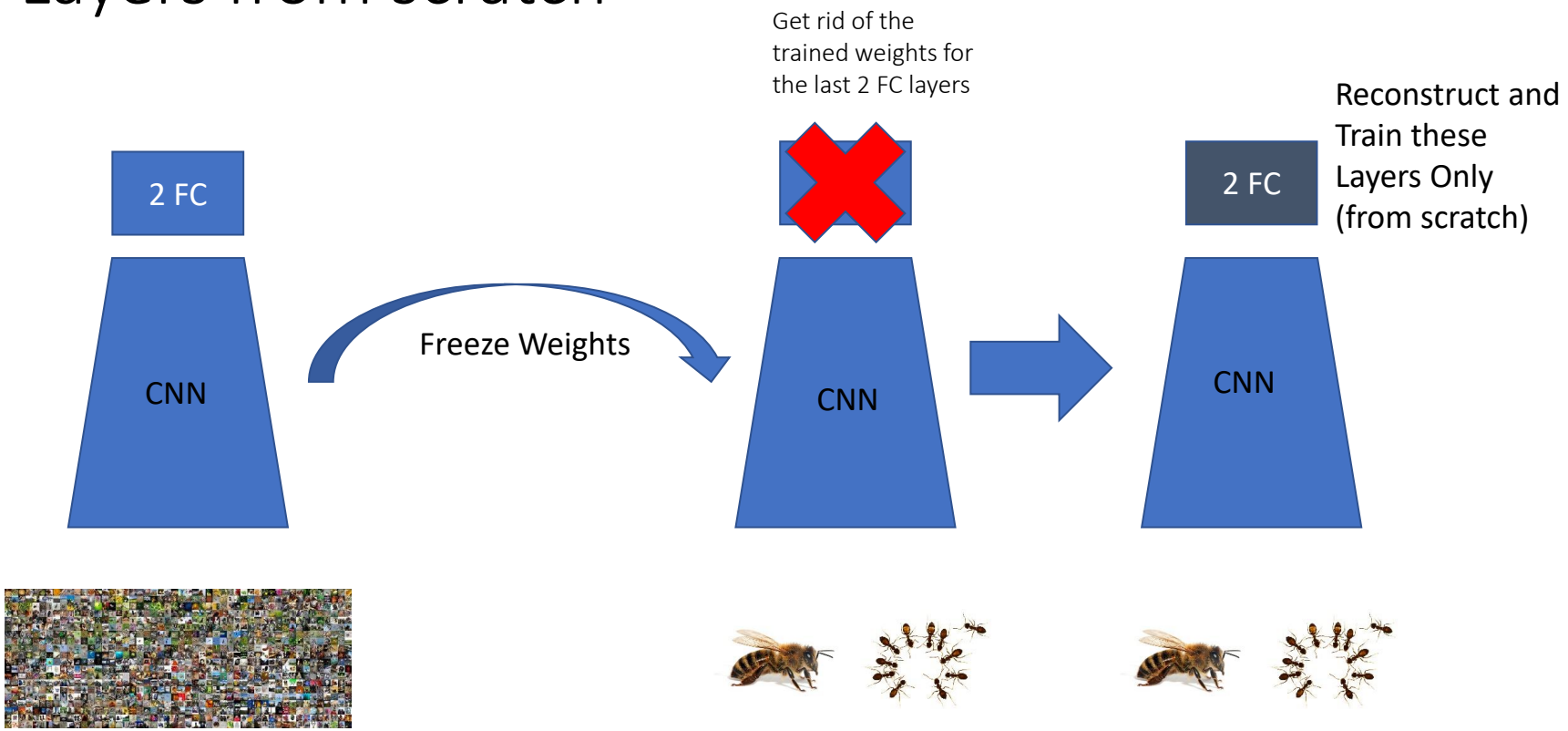
Generally, the low-level features are trained and don't require learning again. We just train the high-level features

Freeze First Part of the Network and Fine-Tuning the Last Layers of the Network



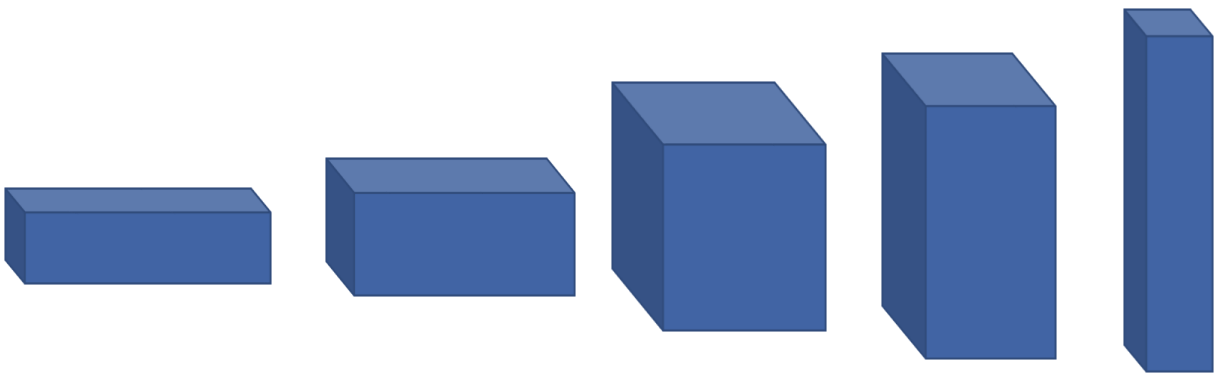
Generally, the low-level features are trained and don't require learning again. We just train the high-level features

Freeze First Part of the Network and Train the Last Layers from scratch



Transposed Convolutions

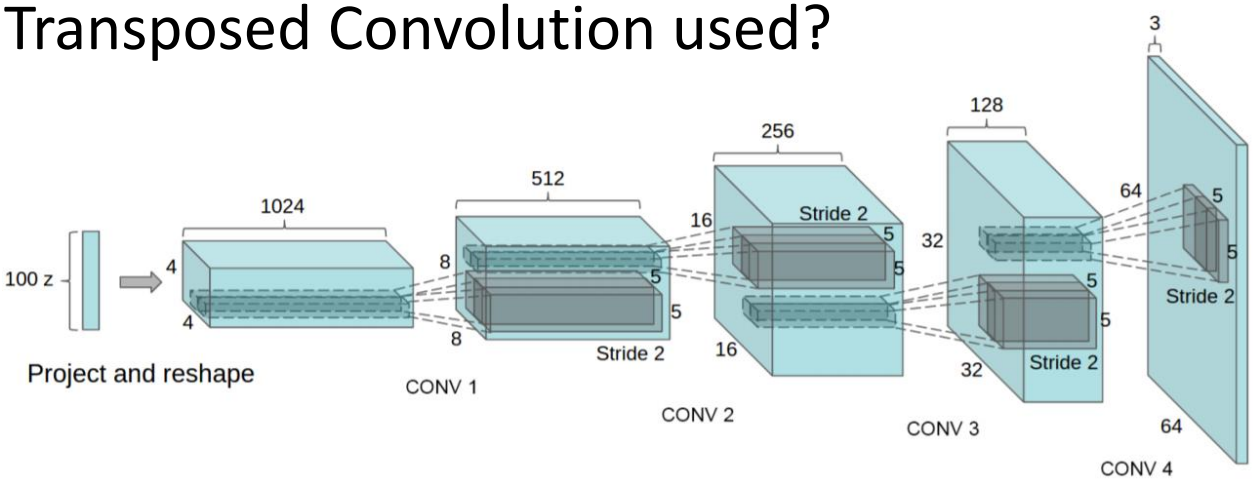
Learning to *Upsample*



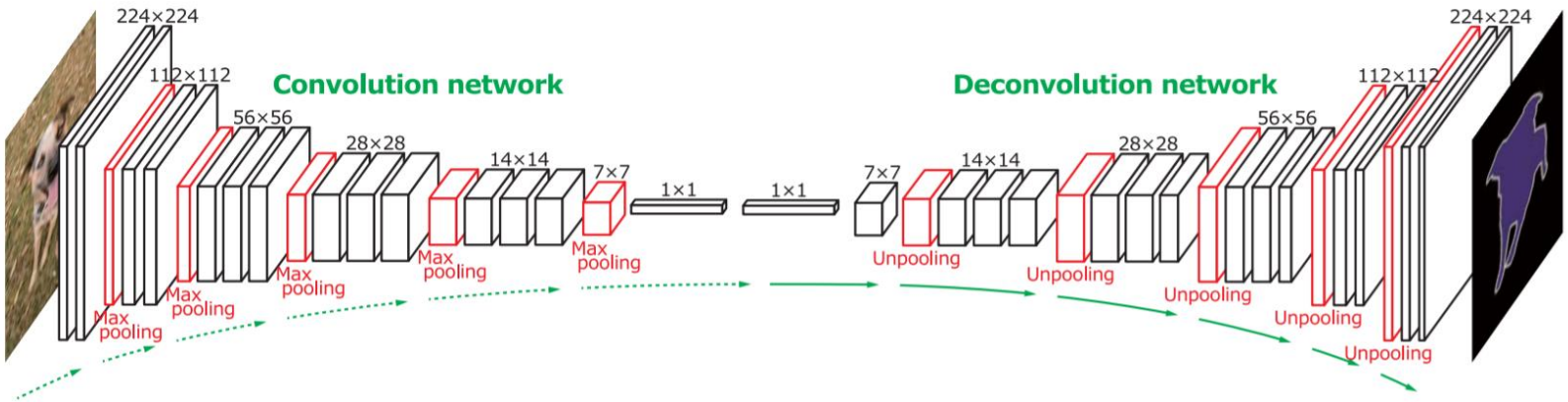
Transposed Convolutions

When we want to up-sample an image, we usually use some image processing techniques such as Nearest neighbor interpolation, Bi-linear or Bi-cubic interpolation. These are *hand-crafted engineering techniques* and **not learnable techniques**. Transposed Convolutions allow us to ***learn the up-sampling operation***. Hence, we let the network itself learn the proper transformation automatically.

Where is Transposed Convolution used?



***DCGAN*: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks**



***Semantic Segmentation*: Learning Deconvolution Network for Semantic Segmentation**

Convolutional Auto-encoder

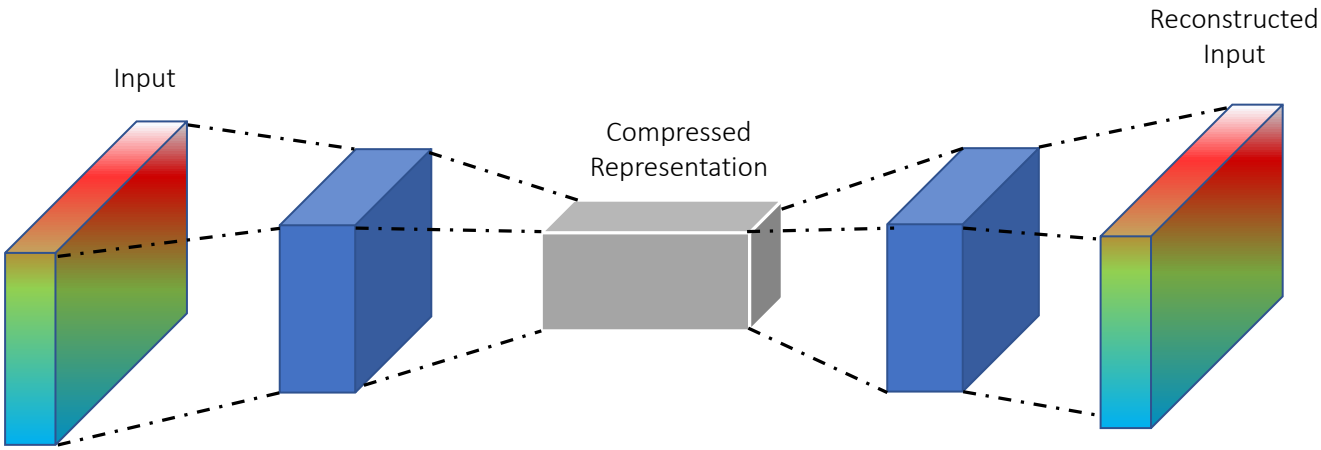
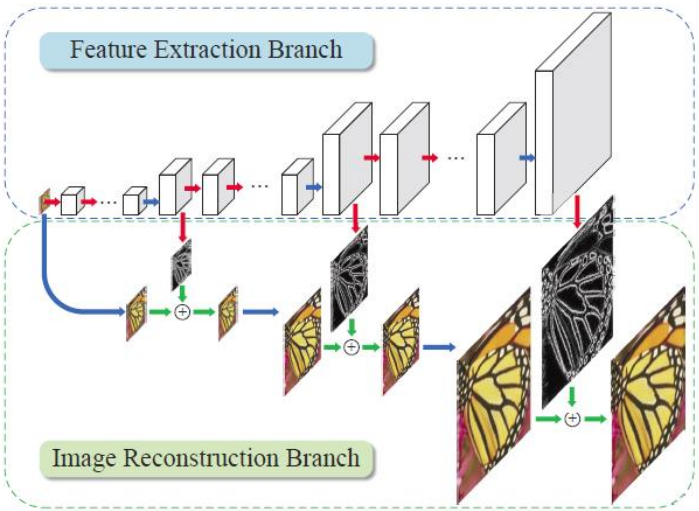


Image Super-Resolution



Re-Visiting the Convolution Operation

Input

x_1	x_2	x_3	x_4
x_5	x_6	x_7	x_8
x_9	x_{10}	x_{11}	x_{12}
x_{13}	x_{14}	x_{15}	x_{16}

Kernel

k_1	k_2	k_3
k_4	k_5	k_6
k_7	k_8	k_9

Convolution

o_1	o_2
o_3	o_4

k_1	k_2	k_3	x_4
k_4	k_5	k_6	x_8
k_7	k_8	k_9	x_{12}
x_{13}	x_{14}	x_{15}	x_{16}

x_1	k_1	k_2	k_3
x_5	k_4	k_5	k_6
x_9	k_7	k_8	k_9
x_{13}	x_{14}	x_{15}	x_{16}

x_1	x_2	x_3	x_4
k_1	k_2	k_3	x_8
k_4	k_5	k_6	x_{12}
k_7	k_8	k_9	x_{16}

x_1	x_2	x_3	x_4
x_5	k_1	k_2	k_3
x_9	k_4	k_5	k_6
x_{13}	k_7	k_8	k_9

o_1	

o_1	o_2

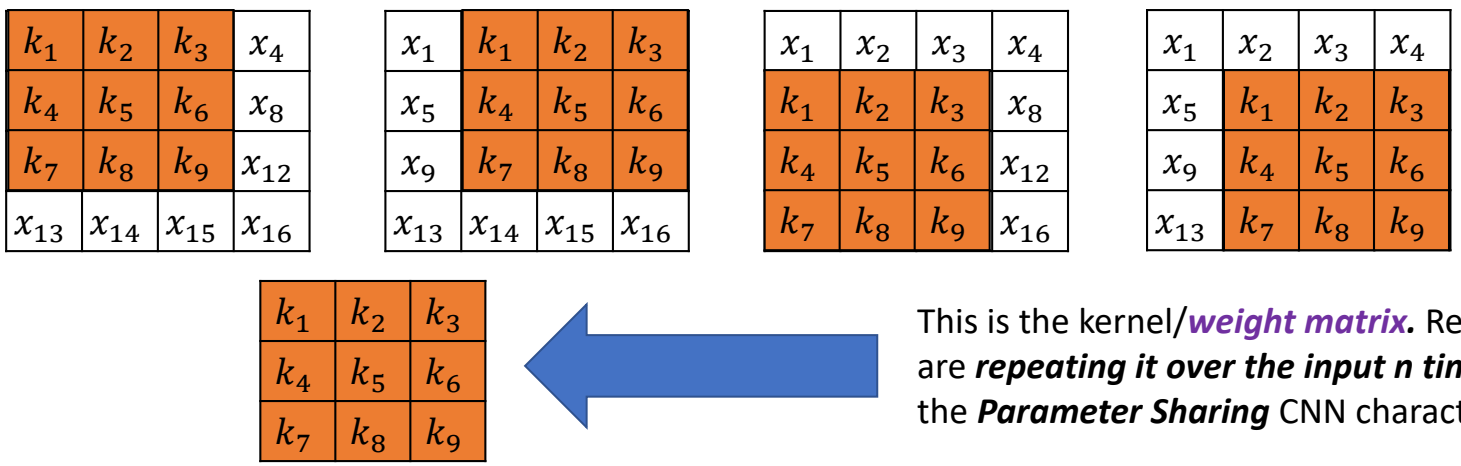
o_1	o_2
o_3	

o_1	o_2
o_3	o_4

Remember....as we learned in the initial sections

All what we are doing is multiplying the input matrix with a learnable weight matrix and getting the output. So at the end, a neural network is just a series of ***matrix multiplications***.

Let's think of the convolution operation as a *matrix multiplication*



This is the kernel/***weight matrix.*** Realize that we are ***repeating it over the input n times*** (this is the ***Parameter Sharing*** CNN characteristic)

If we want to perform the convolution operation as a matrix multiplication....

We need to repeat the kernel ***n*** times and re-arrange it (***n*** is the number of times we slide the kernel through the input). We can then obtain the weight matrix equivalent to performing the convolution operation. This weight matrix is called the *Convolution Matrix*.

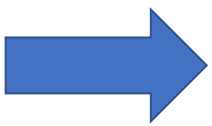
Summary: We can express a convolution operation using a matrix. This matrix is the kernel matrix repeated and rearranged so that we can use a matrix multiplication to conduct the convolution operation. ***In other words, we unroll the convolution operation into a matrix***

Since we slide the kernel n times through our input, the unrolled convolution matrix shape depends on the input

Usually, it is: $size \times (n \times size)$

3x3

k_1	k_2	k_3
k_4	k_5	k_6
k_7	k_8	k_9



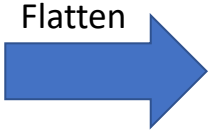
4x16

k_1	k_2	k_3	0	k_4	k_5	k_6	0	k_7	k_8	k_9	0	0	0	0	0
0	k_1	k_2	k_3	0	k_4	k_5	k_6	0	k_7	k_8	k_9	0	0	0	0
0	0	0	0	k_1	k_2	k_3	0	k_4	k_5	k_6	0	k_7	k_8	k_9	0
0	0	0	0	0	k_1	k_2	k_3	0	k_4	k_5	k_6	0	k_7	k_8	k_9

x_1	x_2	x_3	x_4
x_5	x_6	x_7	x_8
x_9	x_{10}	x_{11}	x_{12}
x_{13}	x_{14}	x_{15}	x_{16}

Input

Flatten



16x1

x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
x_9
x_{10}
x_{11}
x_{12}
x_{13}
x_{14}
x_{15}
x_{16}

4x16

k_1	k_2	k_3	0	k_4	k_5	k_6	0	k_7	k_8	k_9	0	0	0	0	0
0	k_1	k_2	k_3	0	k_4	k_5	k_6	0	k_7	k_8	k_9	0	0	0	0
0	0	0	0	k_1	k_2	k_3	0	k_4	k_5	k_6	0	k_7	k_8	k_9	0
0	0	0	0	0	k_1	k_2	k_3	0	k_4	k_5	k_6	0	k_7	k_8	k_9

⊗ Matrix Multiply

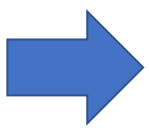
16x1

x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
x_9
x_{10}
x_{11}
x_{12}
x_{13}
x_{14}
x_{15}
x_{16}

We flatten the input for the purpose of performing matrix multiplication

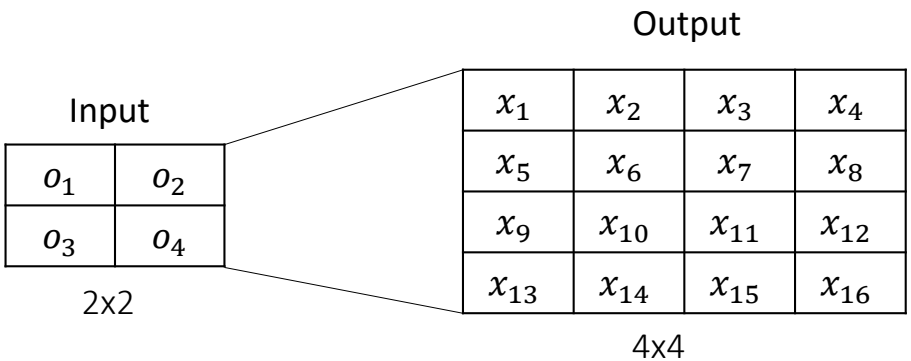
$4x16 \otimes 16x1 = 4x1$

o_1
o_2
o_3
o_4

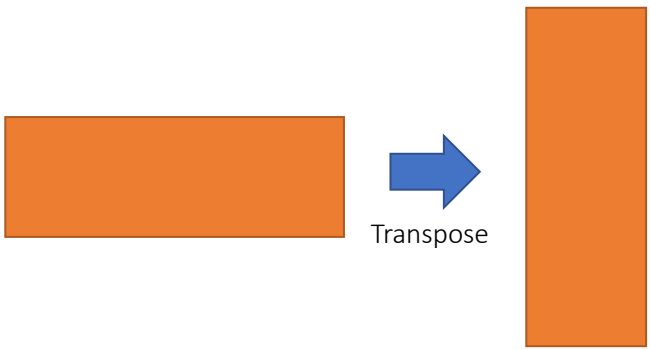


o_1	o_2
o_3	o_4

How can we go the other way around, from *downsampling* to *upsampling*?



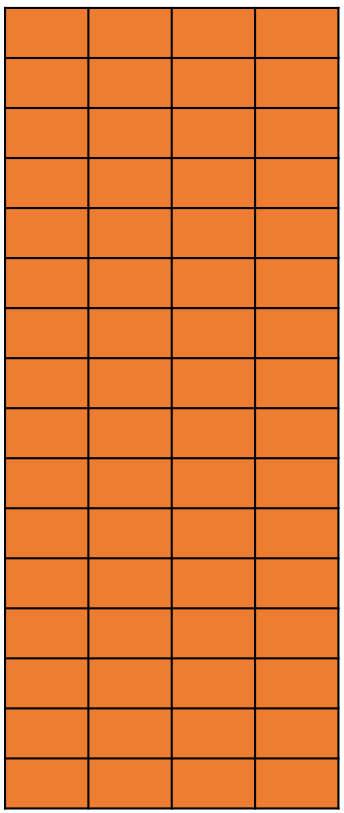
We can *transpose* the convolution matrix



In this case, the convolution matrix shape depends on the size we want to upsample to

$(n \times size) \times size$

16x4



$\otimes$

Matrix Multiply

4x1

x_1
x_2
x_3
x_4

Input (Flattened)

$=$

16x1

x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
x_9
x_{10}
x_{11}
x_{12}
x_{13}
x_{14}
x_{15}
x_{16}

Reshape

4x4

x_1	x_2	x_3	x_4
x_5	x_6	x_7	x_8
x_9	x_{10}	x_{11}	x_{12}
x_{13}	x_{14}	x_{15}	x_{16}

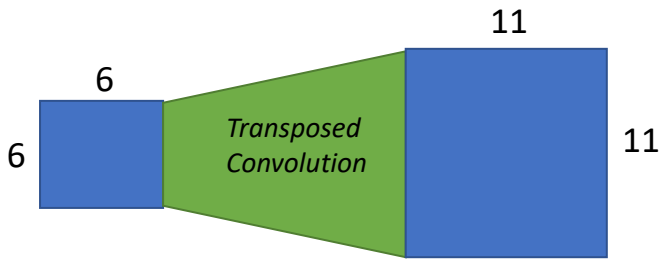
$16x4 \otimes 4x1 = 16x1$

Formula

$$output\ size = (input\ size\ - 1)stride - 2(padding) + (kernel\ size - 1) + 1$$

Example

Kernel size = 3
Stride=2
Padding=1



$$output\ size = (6 - 1)2 - 2(1) + (3 - 1) + 1 = 11$$

```
input = torch.randn(1,16,6,6)
upsample = nn.ConvTranspose2d(16, 16, 3, stride=2, padding=1)
output = upsample(input)
print(output.shape)
```

```
torch.Size([1, 16, 11, 11])
```

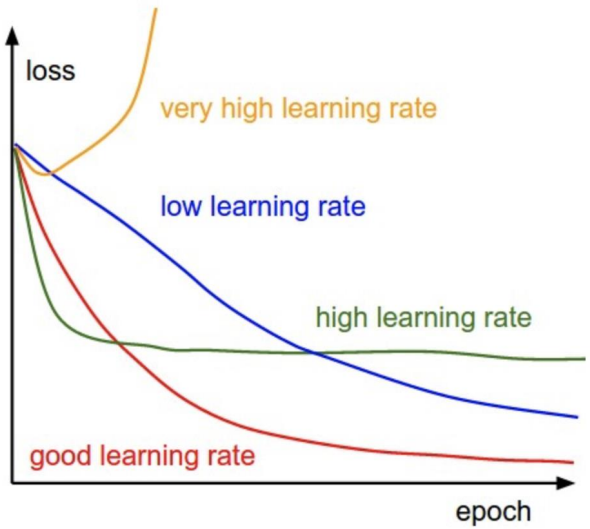
Hyperparameter Tuning and Learning Rate Scheduling

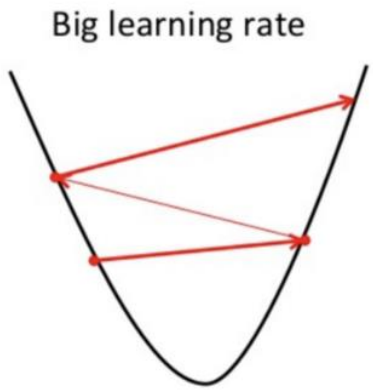
What is Hyperparameter Tuning? And Why do we need it?

- Usually, it refers to choosing the optimization settings (such as the batch size, learning rate, scheduling the learning rate, the optimizer ….etc.). In general, they are the training variables set manually with a pre-determined value before starting the training.
- **Hyperparameter Optimization means to find the model hyperparameters that yield the best score on the validation set metric**
- These are the main components that a neural network with in order to minimize the error. In order to reach the minimum error with the highest accuracy, we need to carefully choose these hyperparameters.

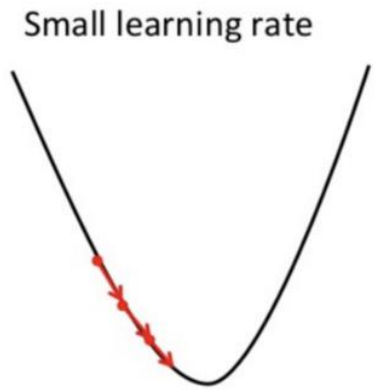
Learning Rate

- The learning rate is very high → Network will never learn
- The learning rate is very low → Network will take too long to learn
- Usually, a learning rate in between is good (0.001, 0.0001, 5e-4)



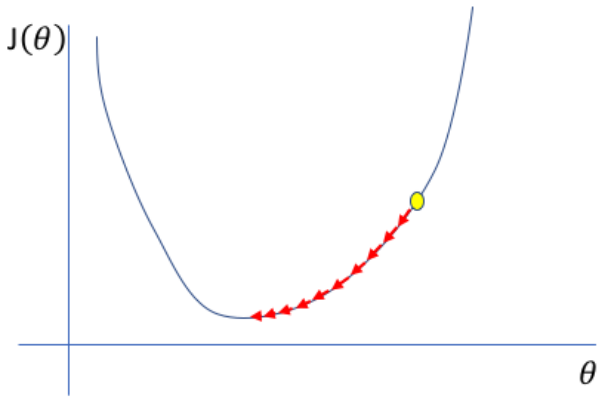


High Learning rate leads to overshooting during gradient descent and might never reach the minima



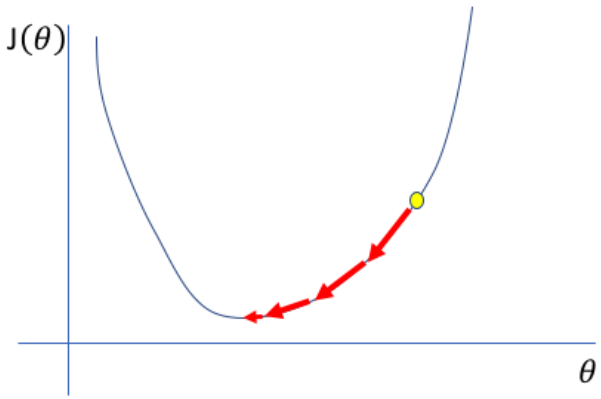
Small and optimal learning rate leads to gradual descent towards the minima

Too low



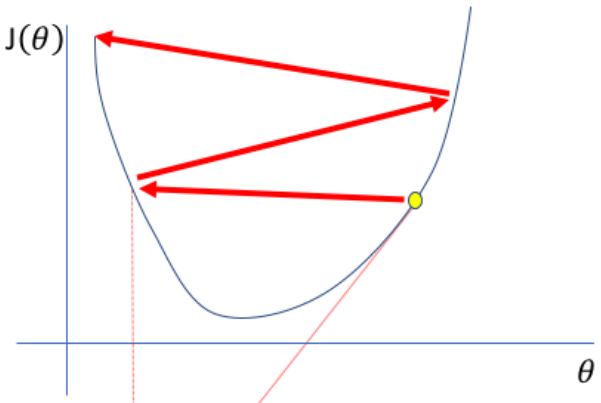
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

Learning Rate Scheduling

Tuning the Learning Rate

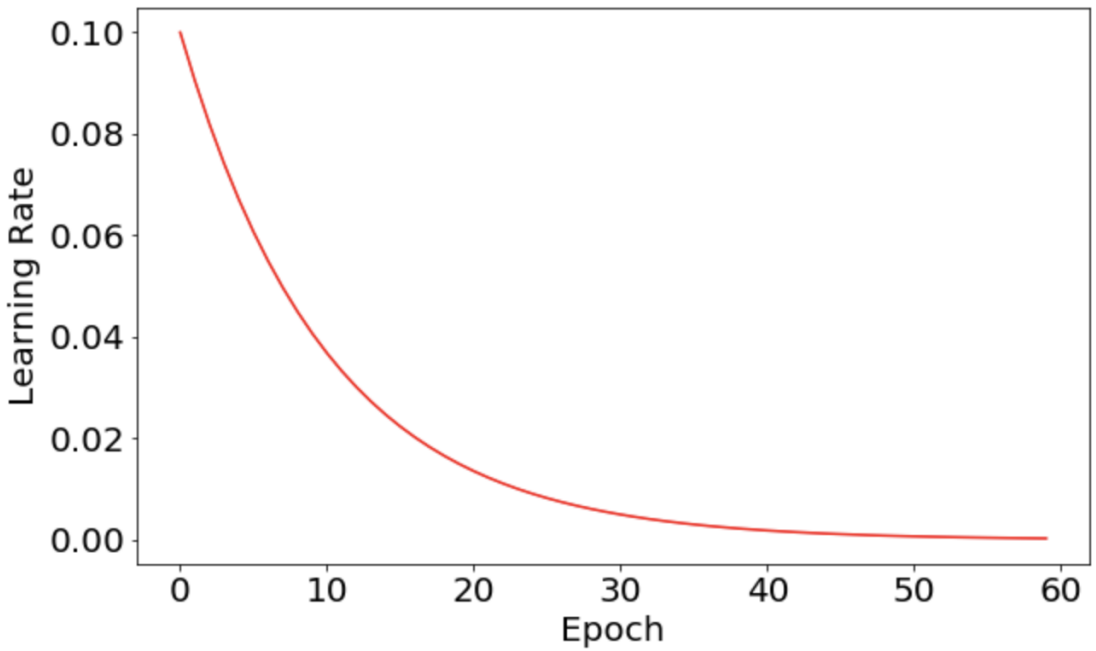
- It is very important that the learning rate is decayed each number of epochs. For example, decay the learning rate by a factor of 0.8 every 3 epochs.

Why?

After some epochs of training, a neural network would have learnt something that is much better than the previous epochs. Therefore, we want to reserve these weights as much as possible. If we use a constant learning rate, these good weights will keep on getting overwritten. Instead we don't want to change them "too" much. Some of them are already optimized.

LR Scheduling

- Reducing the learning rate during training



Constant Learning Rate

- The learning rate is set constant for all the training epochs and not decayed.
- The model might never converge

Why?

After some epochs of training, a neural network would have learnt something that is much better than the previous epochs. Therefore, we want to reserve these weights as much as possible. If we use a constant learning rate, these good weights will keep on getting overwritten. Instead we don't want to change them "too" much. Some of them are already optimized.

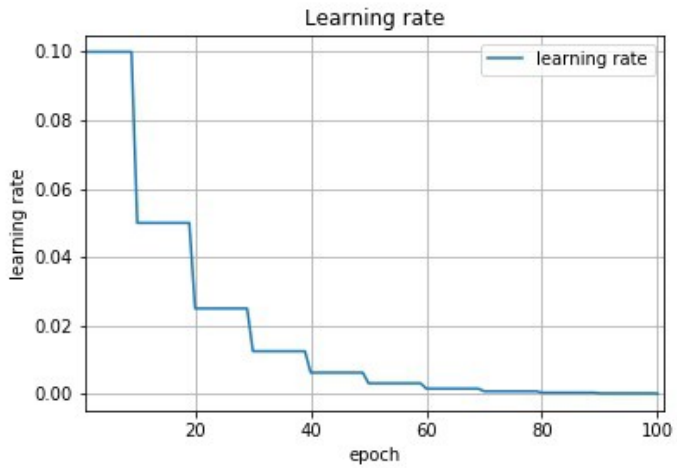
// is division without remainder

Step Decay

The new learning rate is calculated as: $initial\ lr\ * decay\ factor$

$$decay\ factor = decay\ rate^{fraction}$$

$$fraction = epoch - start\_decay\_epoch//decay\_every$$



Example

initial_lr = 0.001, decay rate = 0.8, start decay at epoch 0, decay every 3

- **Epoch 0**: 0//3 = 0, fraction = 0, decay factor = 1, new lr = 0.001 * 1 = 0.001
- **Epoch 1**: 1//3 = 0, fraction = 0, decay factor = 1, new lr = 0.001 * 1 = 0.001
- **Epoch 2**: 2//3 = 0, fraction = 0, decay factor = 1, new lr = 0.001 * 1 = 0.001
- **Epoch 3**: 3//3 = 1, fraction = 1, decay factor = 0.8, new lr = 0.001 * 0.8 = 0.0008
- **Epoch 4**: 4//3 = 1, fraction = 1, decay factor = 0.8, new lr = 0.001 * 0.8 = 0.0008
- **Epoch 5**: 5//3 = 1, fraction = 1, decay factor = 0.8, new lr = 0.001 * 0.8 = 0.0008
- **Epoch 6**: 6//3 = 2, fraction = 2, decay factor = 0.64, new lr = 0.001 * 0.64 = 0.00064
- **Epoch 7**: 7//3 = 2, fraction = 2, decay factor = 0.64, new lr = 0.001 * 0.64 = 0.00064
- **Epoch 8**: 8//3 = 2, fraction = 2, decay factor = 0.64, new lr = 0.001 * 0.64 = 0.00064
- **Epoch 9**: 9//3 = 3, fraction = 3, decay factor = 0.512, new lr = 0.001 * 0.512 = 0.000512
- **Epoch 10**: 10//3 = 3, fraction = 3, decay factor = 0.512, new lr = 0.001 * 0.512 = 0.000512

Simpler Way to implement

```
# Decay the learning rate every 3 epochs by a factor of 0.8
decay = 0
for epoch in range(num_epochs):
    if (epoch+1) % 3 == 0:
        decay+=1
        new_lr = initial_lr * (0.8**decay)
```

CLASS `torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma=0.1, last_epoch=-1)` [SOURCE]

Decays the learning rate of each parameter group by gamma every step_size epochs. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler. When last_epoch=-1, sets initial lr as lr.

Parameters

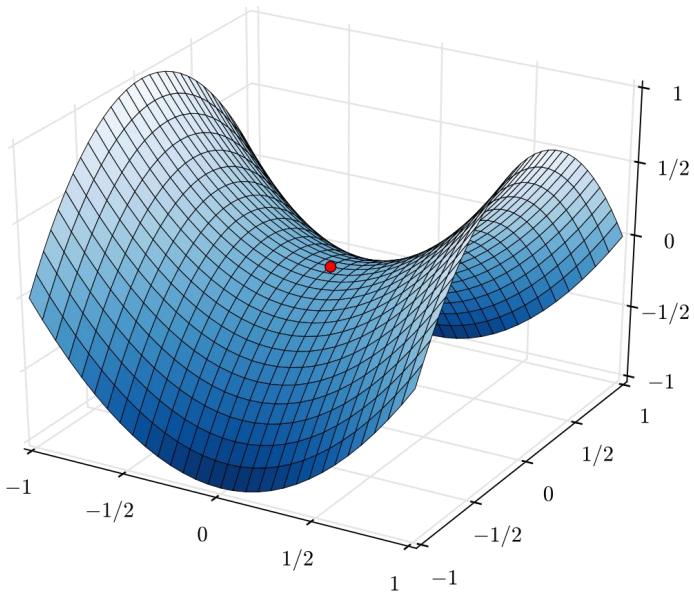
- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **step_size** (*python:int*) – Period of learning rate decay.
- **gamma** (*python:float*) – Multiplicative factor of learning rate decay. Default: 0.1.
- **last_epoch** (*python:int*) – The index of last epoch. Default: -1.

Example

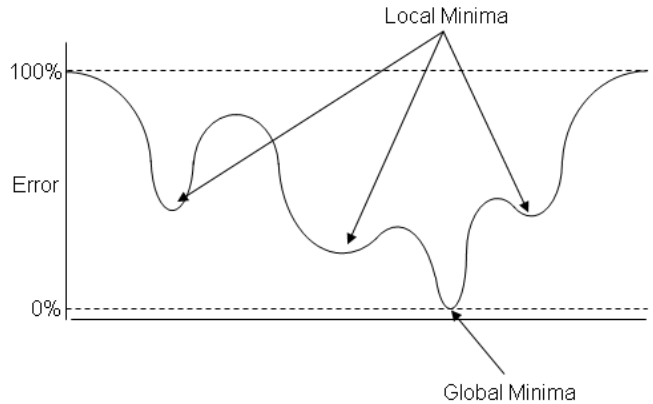
```
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 30
>>> # lr = 0.005    if 30 <= epoch < 60
>>> # lr = 0.0005   if 60 <= epoch < 90
>>> # ...
>>> scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

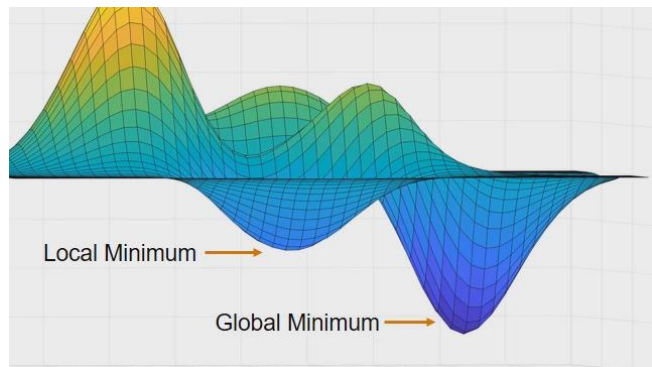
Cyclic Learning Rate and Restarts

The problem of Saddle Points and Local Minima

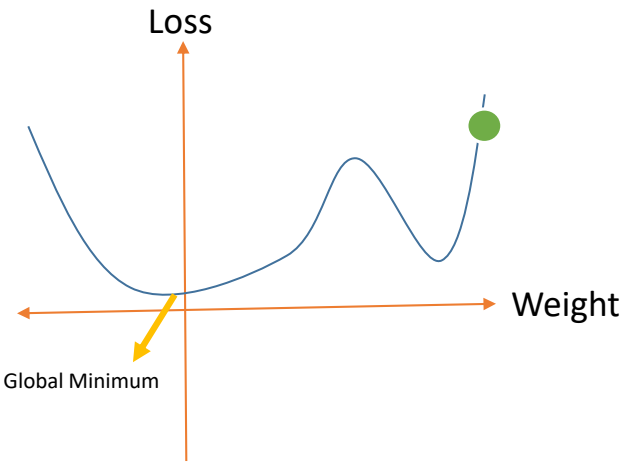


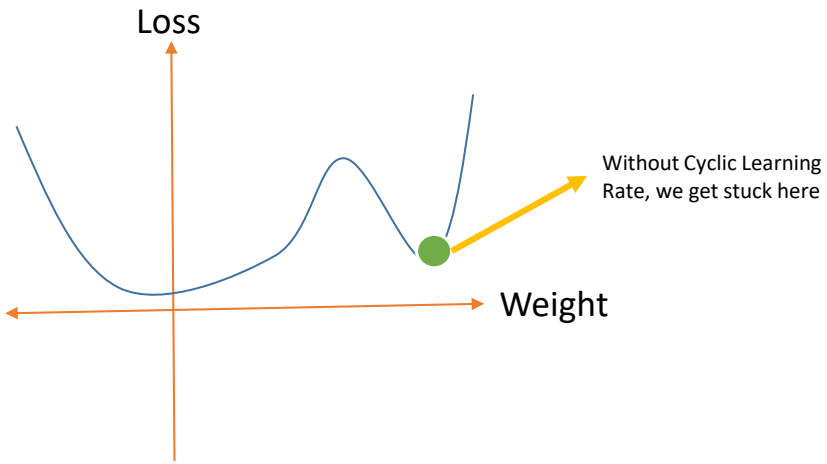
Saddle Point

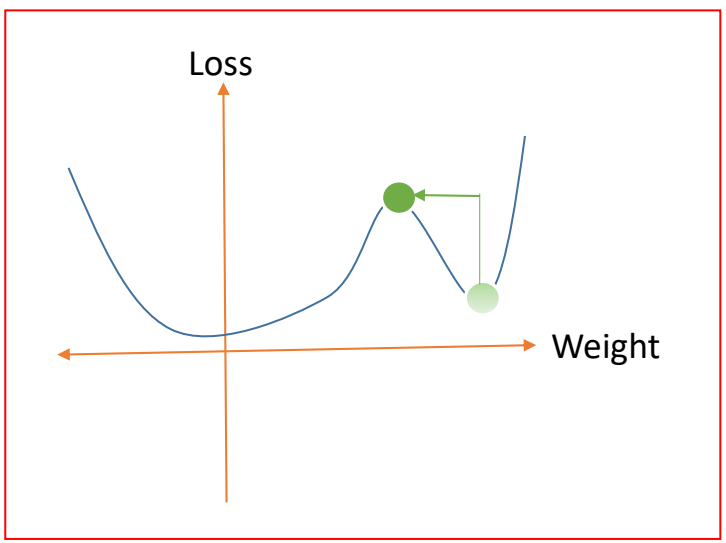


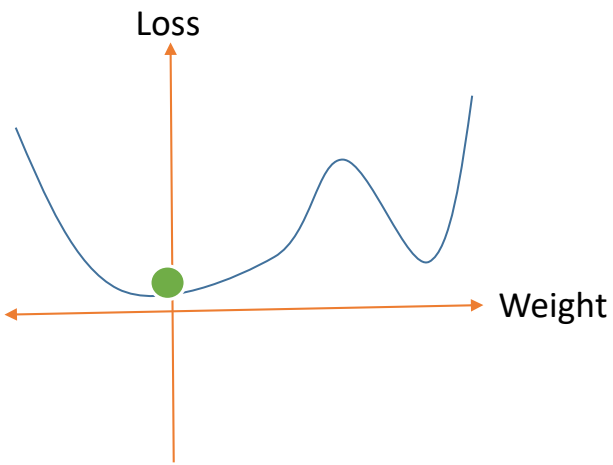


Local Minima


Loss
Weight
Global Minimum


Loss
Weight
Without Cyclic Learning Rate, we get stuck here


Loss
Weight

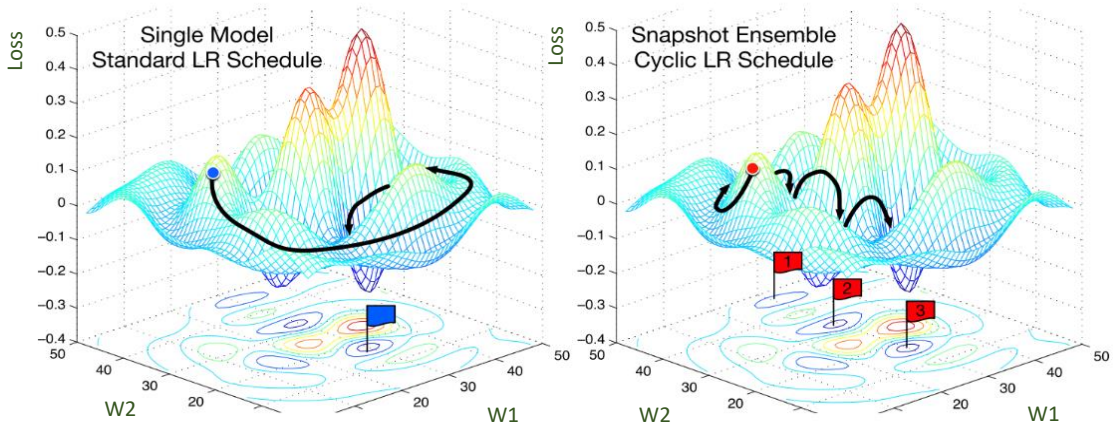

Loss
Weight

Cyclic Learning Rate

The learning rate changes in a **cyclical way** (increases and decreases back within two bounds, periodically).

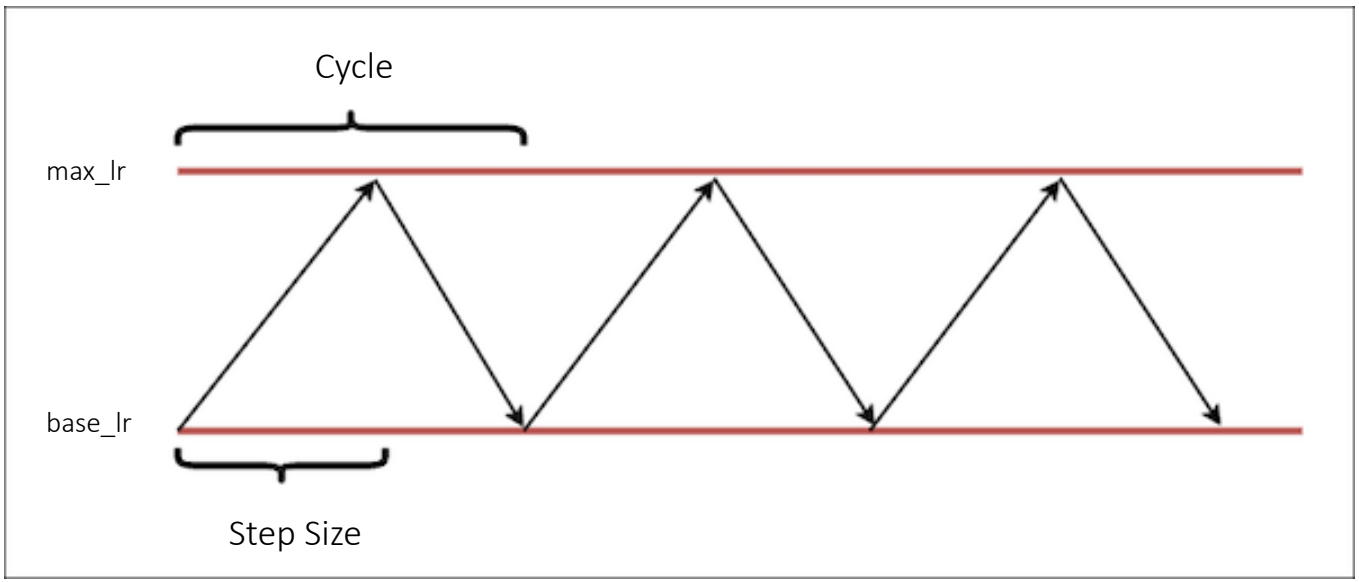
- We can come out of any saddle points or local minima if we encounter one. If saddle point happens to be an elaborated plateau, lower learning rates will probably never generate enough gradient to come out of it, resulting in difficulty in minimizing the loss. This restart in the learning rate will be able to more easily pop out of a saddle points and local minimum if stuck.

- If we make a poor initial choice in learning rate, our model may be stuck from the very start.



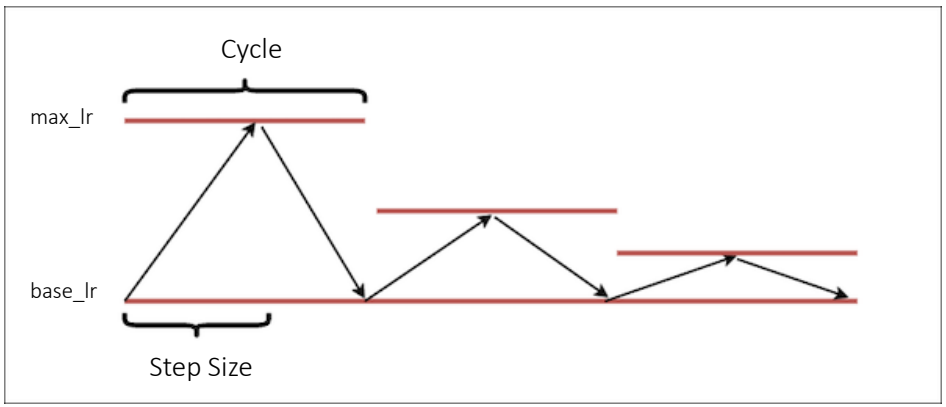
Example of loss curve for 2 parameters W1 and W2.

- We need to set the bounds between which the learning rate will vary: **base learning rate** and **maximum learning rate**.
- **Cycle:** Number of iterations we want for our learning rate to go from lower bound to upper bound, and then back to lower bound.
- **Step size:** Number of iterations to complete half of a cycle.
- Once the maximum learning rate is hit, we then decrease the learning rate back to the base learning rate, which takes half a cycle

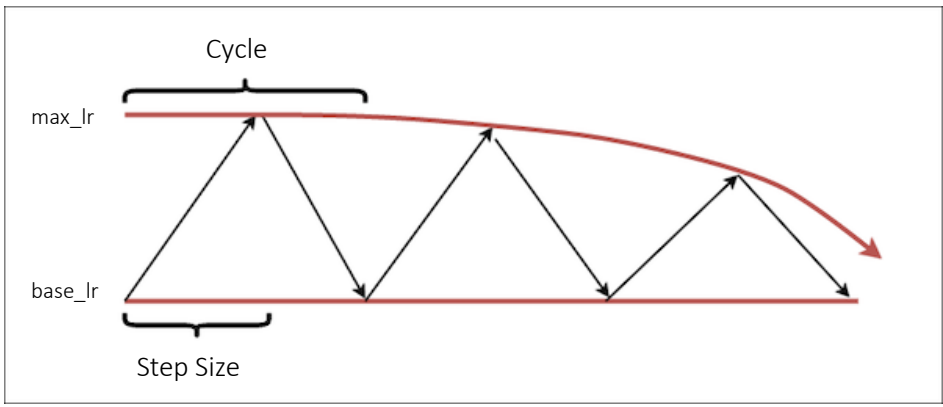


Other Variations

Triangular2 : Here the *max_lr* is halved after every cycle.

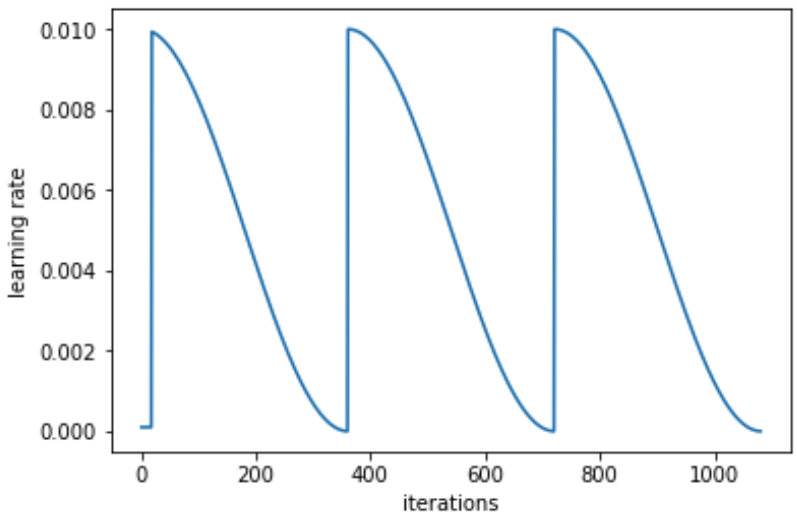


Exponential Range : Here *max_lr* is reduced exponentially



Cosine Annealing with Warm Restarts

Follows the same concept as cyclic learning rate, but anneals the learning rate using a cosine function.



T_{cur} is updated at each batch iteration, so it can take discredited values such as 0.1, 0.2…etc

$\cos(\pi) = -1$
$\cos(0) = 1$

η_{max} Initial/maximum lr

η_{min} minimum lr

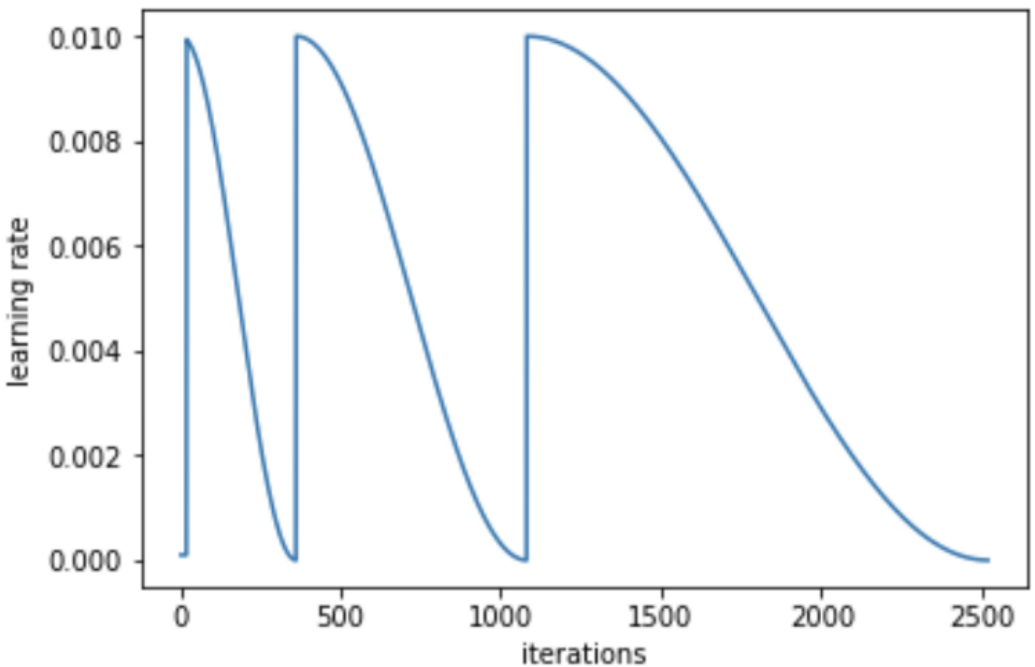
T_{cur} number of epochs since the last restart

T_i number of epochs between two warm restarts in SGDR

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_i}\pi\right)\right)$$

When $T_{cur} = T_i \Rightarrow \eta_t = \eta_{min}$. When $T_{cur} = 0$ after restart $\Rightarrow \eta_t = \eta_{max}$

- We will most likely get closer to a global minimum the more iterations we do through our dataset. Therefore, instead of restarting every epoch, we can lengthen the number of epochs until restart.
- Example: first cycle will decrease over 1 epoch, the second 2 epochs, and the third over 4 epochs, etc.



PyTorch

`torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, T_0, T_mult=1, eta_min=0, last_epoch=-1)`

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **T_0** (*python:int*) – Number of iterations for the first restart.
- **T_mult** (*python:int, optional*) – A factor increases T_i after a restart. Default: 1.
- **eta_min** (*python:float, optional*) – Minimum learning rate. Default: 0.
- **last_epoch** (*python:int, optional*) – The index of last epoch. Default: -1.

`step(epoch=None)`

Step could be called after every batch update

Example

```
>>> scheduler = CosineAnnealingWarmRestarts(optimizer, T_0, T_mult)
>>> iters = len(dataloader)
>>> for epoch in range(20):
>>>     for i, sample in enumerate(dataloader):
>>>         inputs, labels = sample['inputs'], sample['labels']
>>>         scheduler.step(epoch + i / iters)
>>>         optimizer.zero_grad()
>>>         outputs = net(inputs)
>>>         loss = criterion(outputs, labels)
>>>         loss.backward()
>>>         optimizer.step()
```

Effect of Batch Size

The larger the batch size, in general, the better the performance

Why?

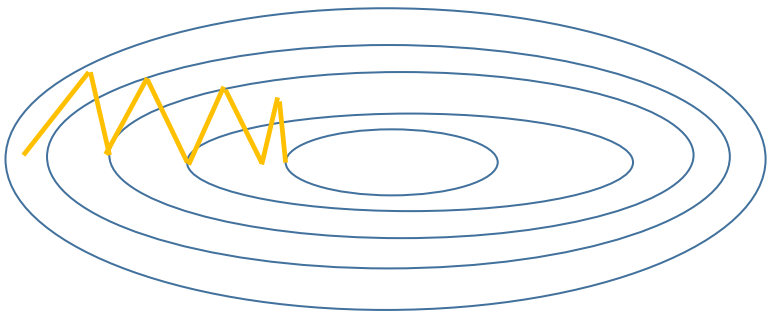
The bigger the batch size is, the less nosier are your updates. Some samples are noisy. By using a large batch size, they can be ignored since other samples will be more dominant. A bigger batch size means we are tending towards *learning the real distribution of the data* and more confident in the direction of the descent.

Therefore, if we increase the batch size we should also increase the learning rate

(example: **when multiplying the batch size by k, one should multiply the learning rate by $\sqrt{k}$**)

$$y = f(x) + w$$

Small Batch Size will lead to noise and less confidence in updates



Large Batch Size will be less noiser and more confident in updates