COMP 432 Machine Learning

Neural Networks (Part 2)

Computer Science & Software Engineering
Concordia University, Fall 2024

Summary of the last episode….

What we have seen **last time**:

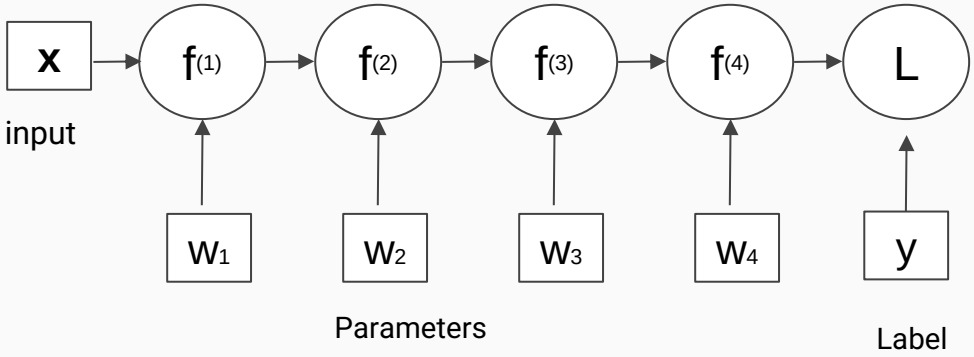
- Linear basis function models
- Introduction to Neural Networks

What we are going to learn **today**:

- Basic Concepts for Neural Networks (vanishing gradient, initialization, normalization)
- Convolutional Neural Networks

Backpropagation

- Last lecture we have seen that we can compute the gradient within a **pipeline of computations** (e.g., those involved in deep neural networks) with the **chain rule**.



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1}\frac{\partial f^{(2)}}{\partial f^{(1)}}\frac{\partial f^{(3)}}{\partial f^{(2)}}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

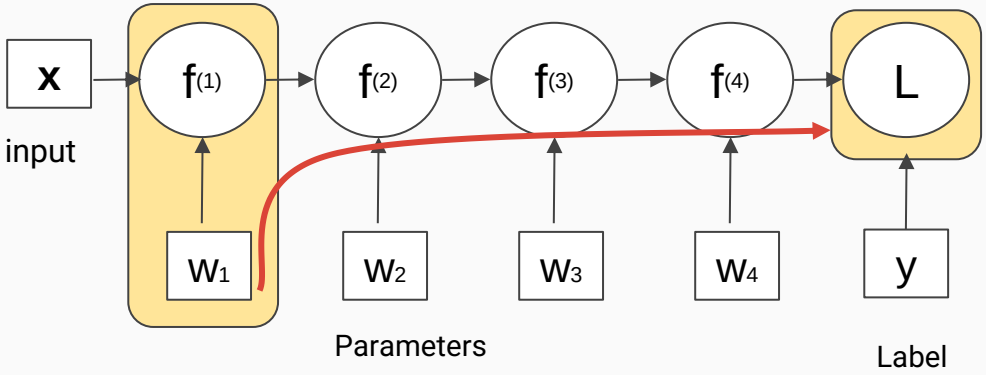
$$\frac{\partial L}{\partial w_2} = \frac{\partial f^{(2)}}{\partial w_2}\frac{\partial f^{(3)}}{\partial f^{(2)}}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial f^{(3)}}{\partial w_3}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

$$\frac{\partial L}{\partial w_4} = \frac{\partial f^{(4)}}{\partial w_4}\frac{\partial L}{\partial f^{(4)}}$$

- **Backpropagation** is an algorithm that computes the chain rule **efficiently**.

Backpropagation



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1}\frac{\partial f^{(2)}}{\partial f^{(1)}}\frac{\partial f^{(3)}}{\partial f^{(2)}}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial f^{(2)}}{\partial w_2}\frac{\partial f^{(3)}}{\partial f^{(2)}}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial f^{(3)}}{\partial w_3}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

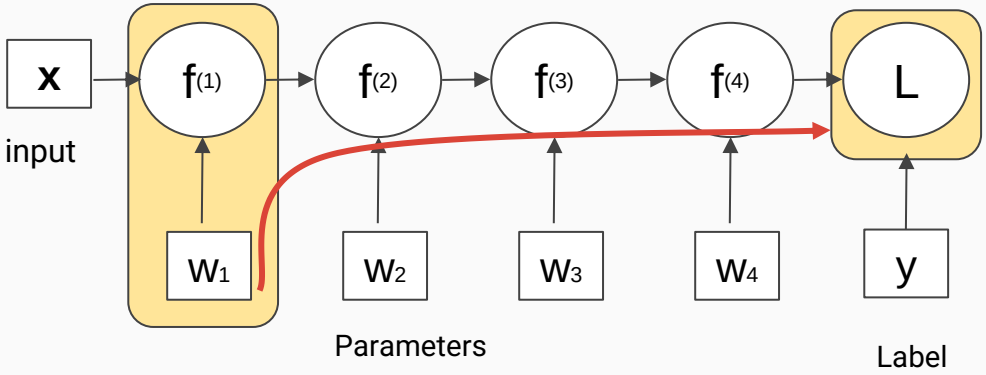
$$\frac{\partial L}{\partial w_4} = \frac{\partial f^{(4)}}{\partial w_4}\frac{\partial L}{\partial f^{(4)}}$$

- **Early layers** are far away from the final loss function.
- The information coming from the early layers has to undergo **many stages** before reaching L.

Is that critical?

- As a result, their **final gradient** requires a **long chain of multiplications** (or dot products in a multi-dimensional space).

Vanishing and Exploding Gradients



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1} \frac{\partial f^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(4)}}{\partial f^{(3)}} \frac{\partial L}{\partial f^{(4)}}$$

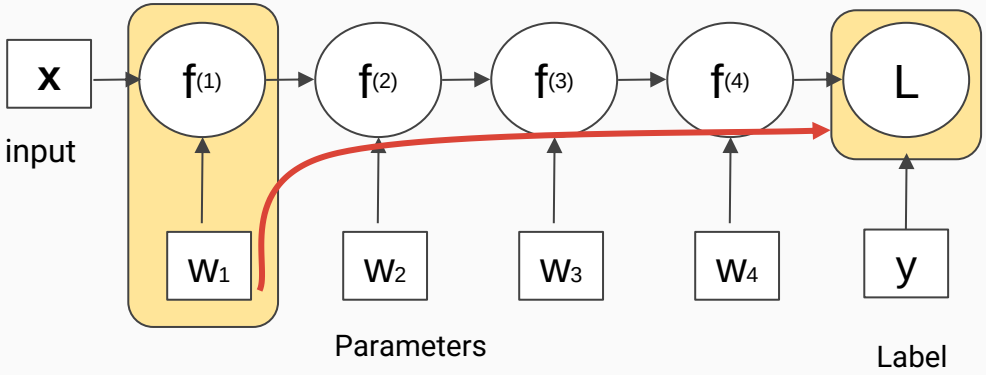
Vanishing: 0.5 x 0.3 x 0.1 x 0.1 x 0.2 = 0.0003

Exploding: 2.5 x 1.3 x 9.9 x 5.4 x 8.2 = 1424

- When backpropagation the gradient through long chains of computations there a two possible issues:
  - **Vanishing Gradient** (often): the gradient gets smaller and smaller.
  - **Exploding Gradient** (rare): the gradient gets bigger and bigger.

In multi-dimensional spaces, the multiplications turn into **dot products**. What causes vanishing and exploding gradients are the **eigenvalues** of the **jacobian matrices**.

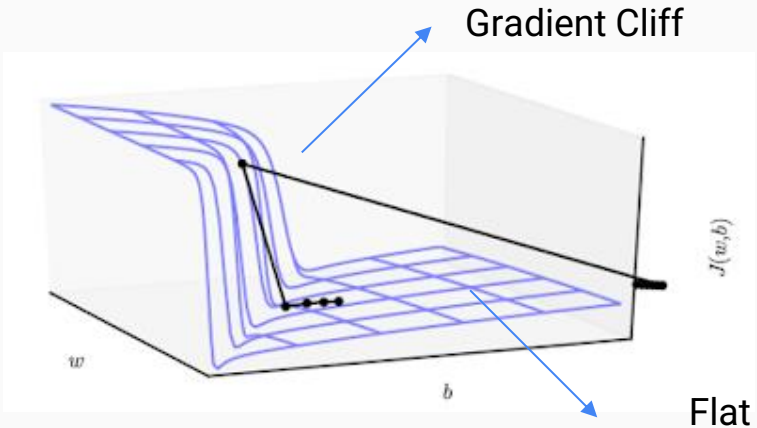
Vanishing and Exploding Gradients



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1}\frac{\partial f^{(2)}}{\partial f^{(1)}}\frac{\partial f^{(3)}}{\partial f^{(2)}}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

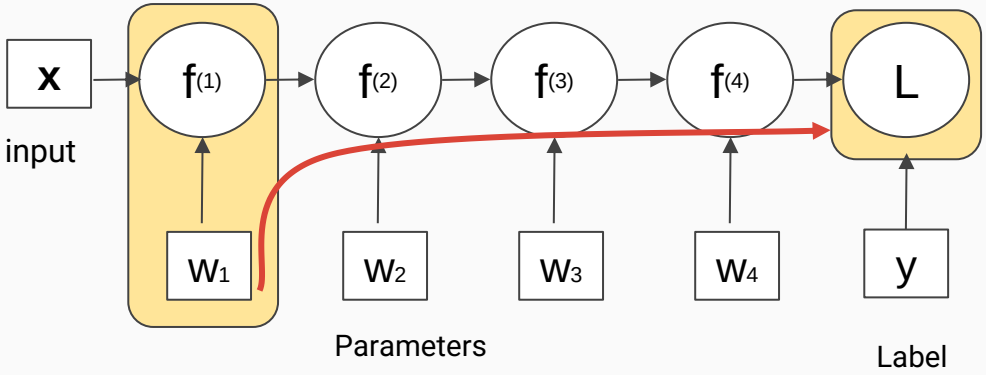
Vanishing: 0.5 x 0.3 x 0.1 x 0.1 x 0.2 = 0.0003

Exploding: 2.5 x 1.3 x 9.9 x 5.4 x 8.2 = 1424



- This causes the parameter space to have very steep areas (**exploding gradients**) and flat areas (**vanishing gradients**).
- These problems get worse when adding **several hidden layers** (deep neural networks).

Exploding Gradients



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1}\frac{\partial f^{(2)}}{\partial f^{(1)}}\frac{\partial f^{(3)}}{\partial f^{(2)}}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

Vanishing: 0.5 x 0.3 x 0.1 x 0.1 x 0.2 = 0.0003

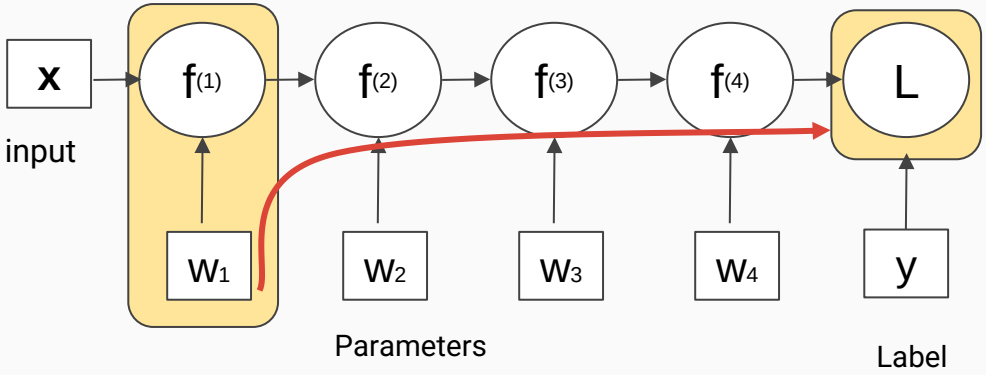
Exploding: 2.5 x 1.3 x 9.9 x 5.4 x 8.2 = 1424

- Exploding gradient causes the **gradient** of the parameters in the early layers to be very **big**.
- *What's the problem with that?*

Can you guess a possible solution?

- Too large updates could throw the parameters very far, into a region where the objective function is larger (thus undoing much of the work that had been done to reach the current solution).
- Moreover, the **magnitude** of the **weights** gets **bigger** and bigger causing (at some point) **numerical issues** (e.g, *inf*).

Exploding Gradients



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1}\frac{\partial f^{(2)}}{\partial f^{(1)}}\frac{\partial f^{(3)}}{\partial f^{(2)}}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

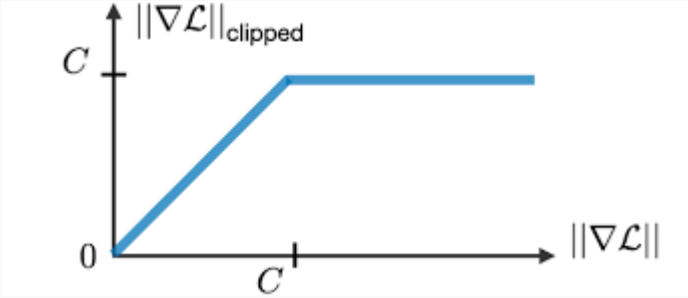
Vanishing: 0.5 x 0.3 x 0.1 x 0.1 x 0.2 = 0.0003

Exploding: 2.5 x 1.3 x 9.9 x 5.4 x 8.2 = 1424

- A solution largely adopted is **gradient clipping**.
- One way to do gradient clipping involves forcing the gradient values (element-wise) to a specific **maximum absolute value** C.

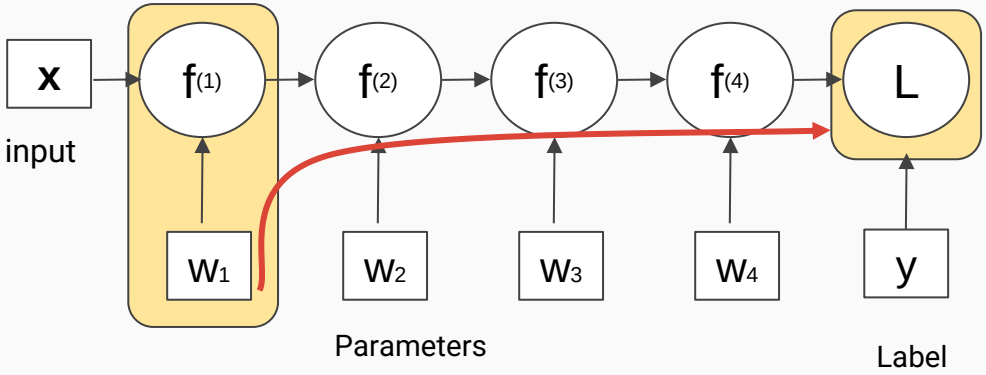
E.g. if the gradient on my parameters is [-0.3, 0.5, -1000, -0.4] and C=5, my gradient will be [-0.3, 0.5, -5, -0.4].

C is a hyperparameter configured by trial and error.



- When gradient descent proposes to make a very large step, the gradient clipping intervenes to reduce the step size. This way it is less likely to go outside the region of interest.

Vanishing Gradient



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1} \frac{\partial f^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(4)}}{\partial f^{(3)}} \frac{\partial L}{\partial f^{(4)}}$$

Vanishing: 0.5 x 0.3 x 0.1 x 0.1 x 0.2 = 0.0003

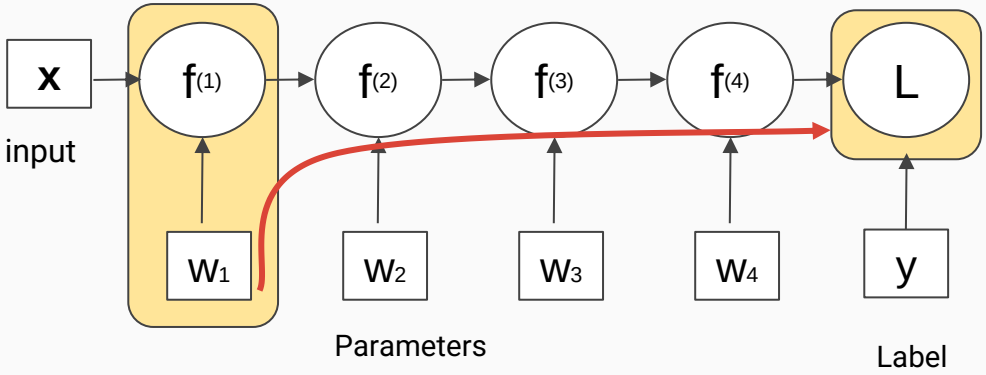
Exploding: 2.5 x 1.3 x 9.9 x 5.4 x 8.2 = 1424

- Vanishing gradient causes the **gradient** of the parameters in the early layers to be very **small**.
- What's the problem with small gradients?

Can you guess a possible solution?

- The parameters **do not change** too much during **training** and they will stay "almost" randomly initialized.

Vanishing Gradient



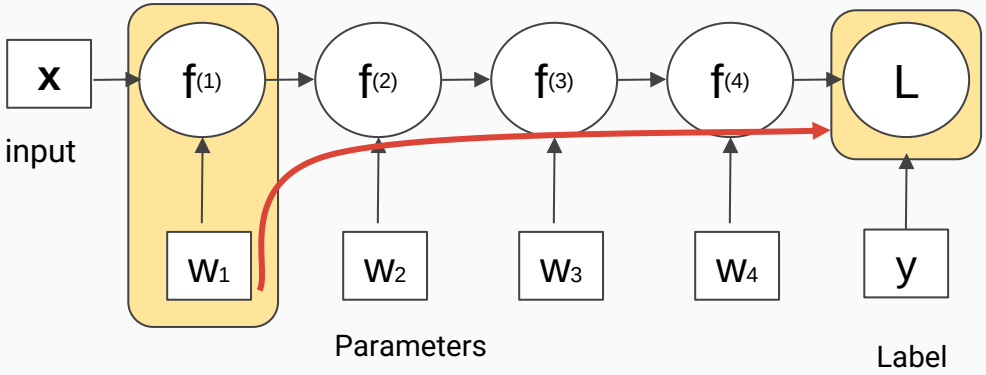
$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1} \frac{\partial f^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(4)}}{\partial f^{(3)}} \frac{\partial L}{\partial f^{(4)}}$$

Vanishing: 0.5 x 0.3 x 0.1 x 0.1 x 0.2 = 0.0003

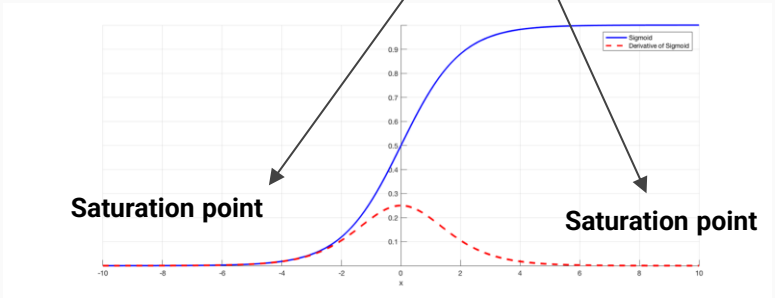
Exploding: 2.5 x 1.3 x 9.9 x 5.4 x 8.2 = 1424

- Several solutions to mitigate this problem have been proposed. Popular approaches are:
  - *Non-saturating activation functions*
  - *Gradient shortcuts*
  - *Initialization*
  - *Normalization*
- Fighting vanishing gradients is a central problem in deep learning. It is an active research area.

Saturating Activation Functions

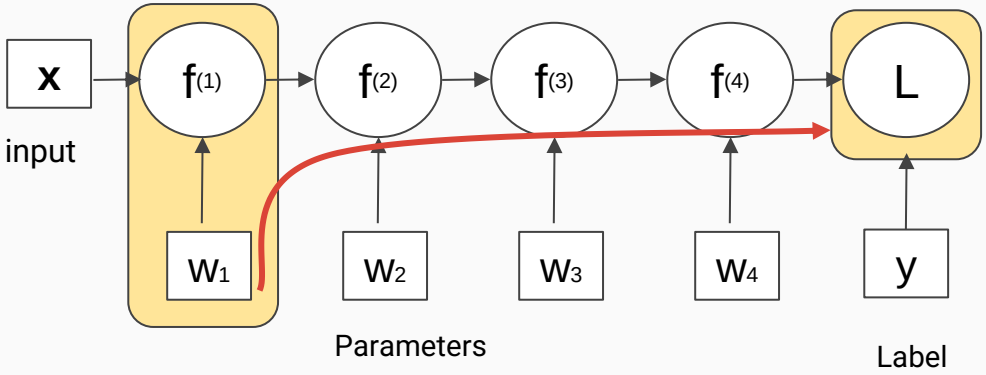


$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1} \frac{\partial f^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(4)}}{\partial f^{(3)}} \frac{\partial L}{\partial f^{(4)}}$$



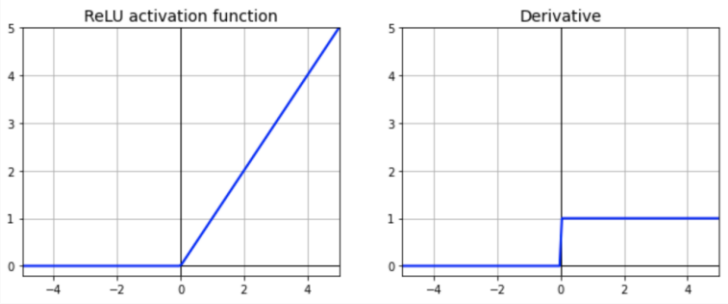
- Sigmoid and Tanh activation functions have **two saturation points** where the gradient is **close to zero**.
- This causes the final gradient to be small as well.
- We have to avoid using these functions as activations of the hidden layers.

Non-saturating Activation Functions



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1} \frac{\partial f^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(4)}}{\partial f^{(3)}} \frac{\partial L}{\partial f^{(4)}}$$

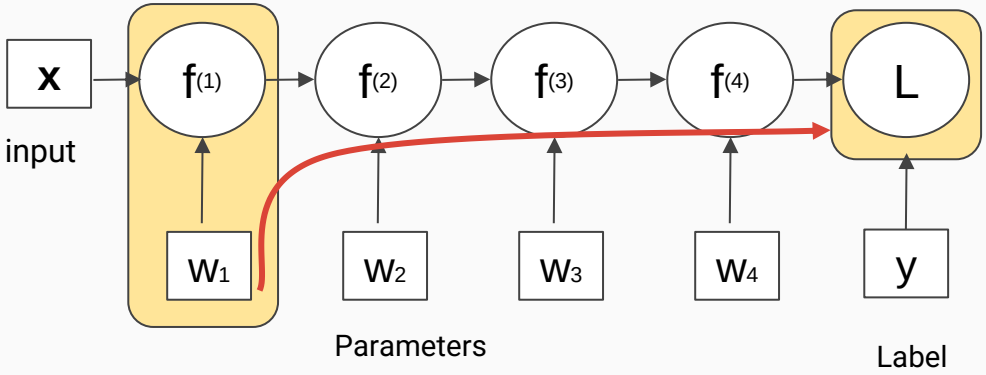
ReLU



Solution 1: Avoid saturating activation functions (e.g., Sigmoid and Tanh)

- Alternatively, we can use **ReLU**s.
- If x > 0, the derivative is one and the gradient can flow **unchanged**.
- If x < 0, the derivative is zero and we have **vanishing problems**.
- However, this happens only on **one side**, and not on both sides as for sigmoids and tanhs (thus alleviating this problem)

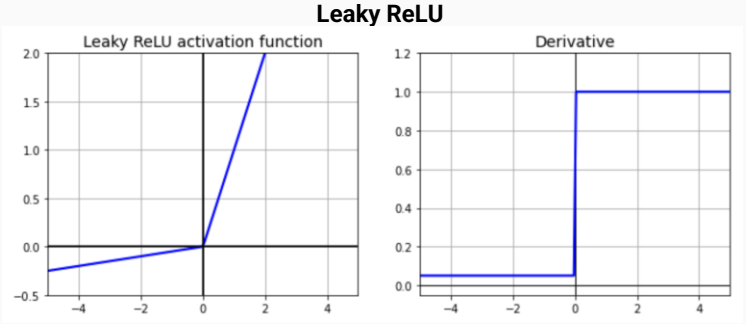
Non-saturating Activation Functions



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1} \frac{\partial f^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(4)}}{\partial f^{(3)}} \frac{\partial L}{\partial f^{(4)}}$$

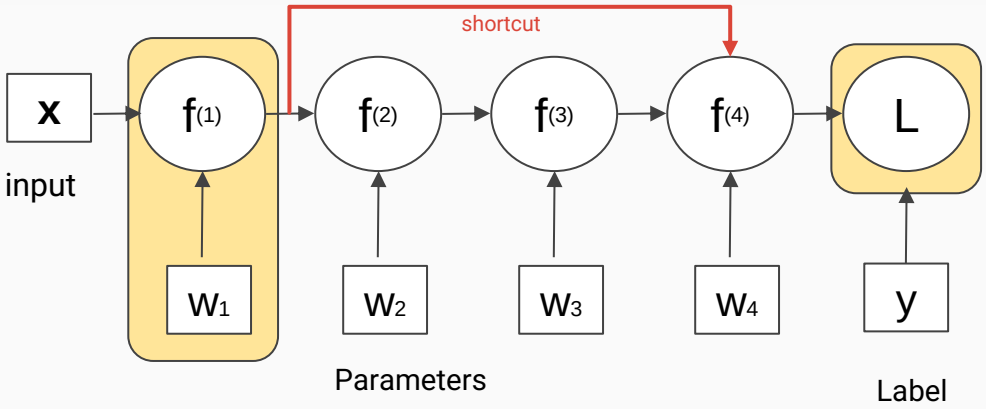
Solution 1: Avoid saturating activation functions (e.g., Sigmoid and Tanh)

- To further mitigate this issue, we can use **Leaky ReLUs**



- In this case, we propagate a gradient > 0 even when x < 0.

Gradient Shortcuts



$$\frac{\partial L}{\partial w_1} = \frac{\partial f^{(1)}}{\partial w_1}\frac{\partial f^{(2)}}{\partial f^{(1)}}\frac{\partial f^{(3)}}{\partial f^{(2)}}\frac{\partial f^{(4)}}{\partial f^{(3)}}\frac{\partial L}{\partial f^{(4)}}$$

$$+$$

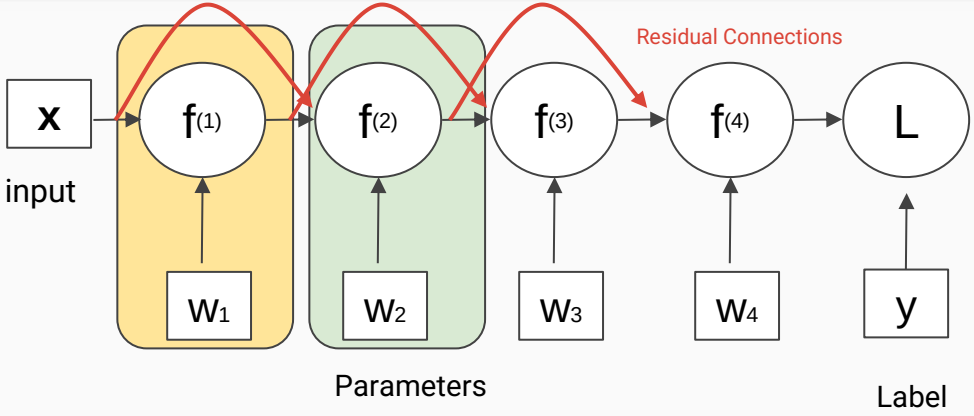
$$\frac{\partial f^{(1)}}{\partial w_1}\frac{\partial f^{(4)}}{\partial f^{(1)}}\frac{\partial L}{\partial f^{(4)}}$$

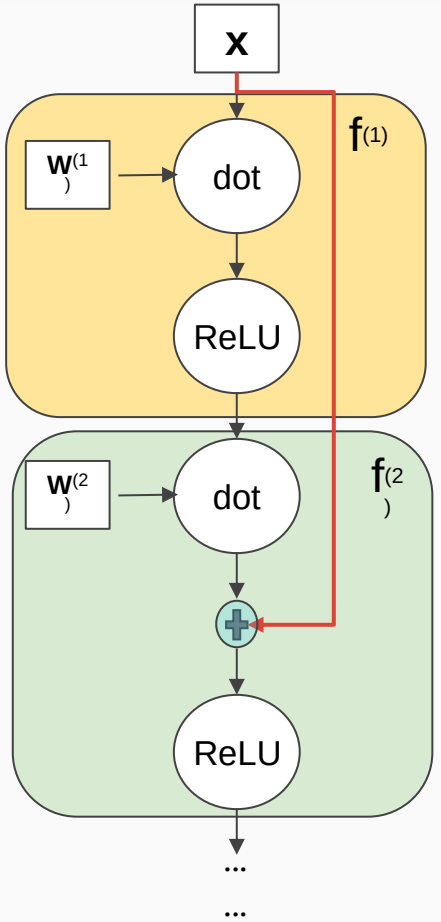
- One way to mitigate vanishing gradients is to add **gradient shortcuts** in the model.
- The idea is to add connections that **shorten the gradient path** from the loss to the early layers.

Shorter Path ➡ *Fewer multiplications* ➡ *Larger Gradient*

- Examples are **residual networks**, **skip connections**, **dense blocks**, **highway connections**, and **attention models**.

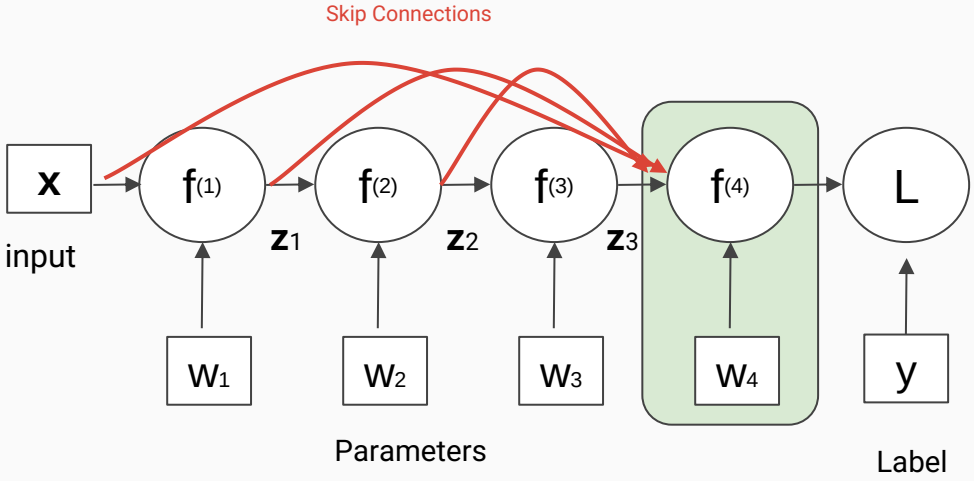
Residual Networks

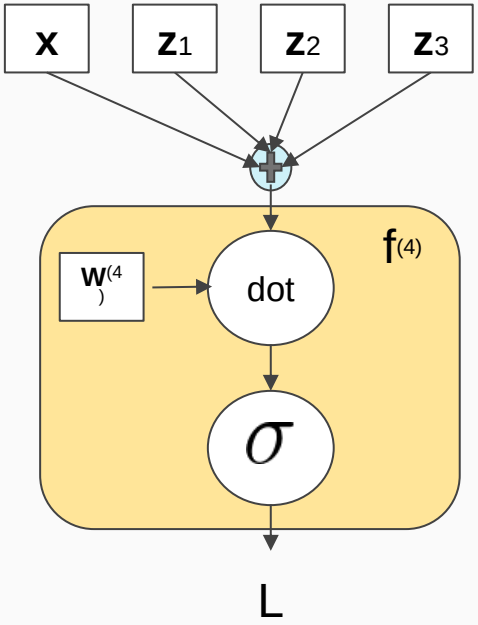




- **Residual networks** add shortcuts across hidden layers.
- These models are called **Residual** Networks because each layer has to model a distribution that is the difference between the current distribution and the one in the previous layer.
- Residual networks work well when **several hidden layers** are employed. In this case, the difference in the distribution learned in two consecutive layers is small and it is easier to learn the residual rather than the full distribution.

Skip Connections

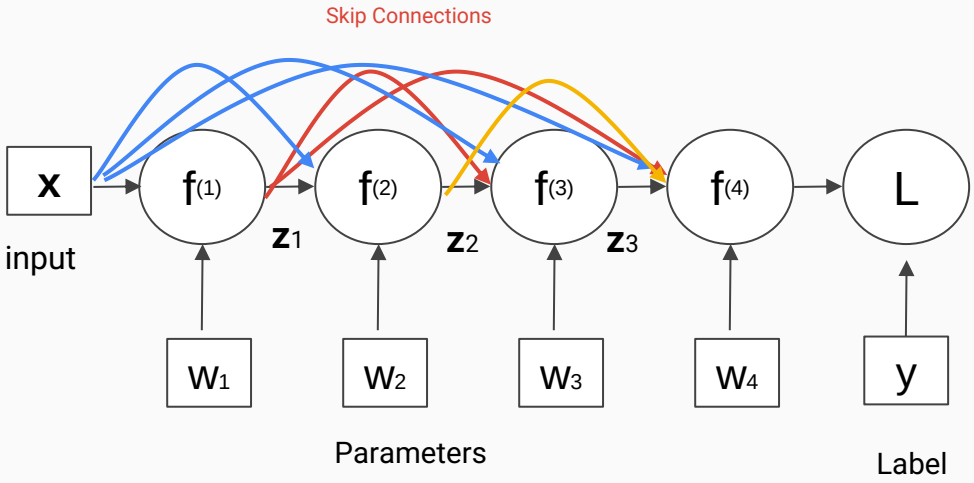




- Skip connections are **direct connections** between each **hidden layer** and the **last** one.

- The last hidden layer sums up the **features coming from all the previous layers**.

DenseNet



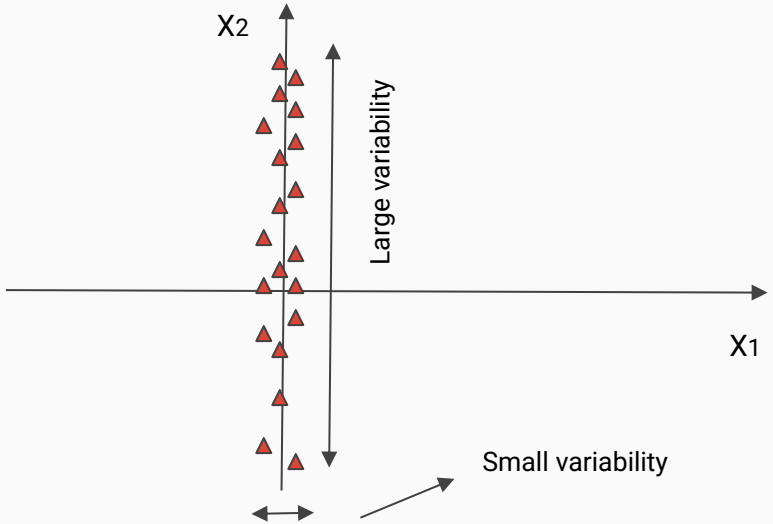
- In **DenseNet** each hidden layer takes in input the **features** from **all the previous layers**.
- This model introduces a lot of gradient shortcuts that help fight vanishing gradients.

Normalization

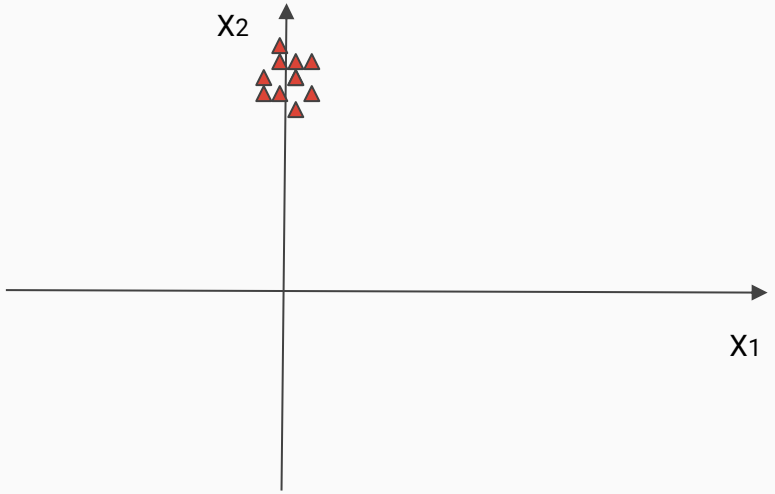
Normalization

- Many learning algorithms struggle when the input features take values on very **different scales** or with very **different biases.**

scales: [-0.001,0.001] vs [-1000,1000]

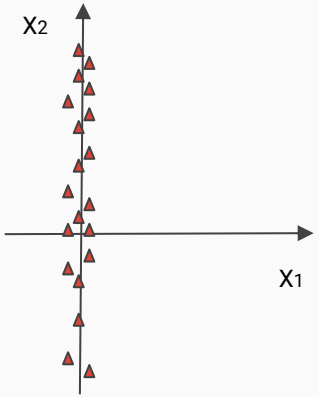


biases: [-1, 1] vs [999, 1001]

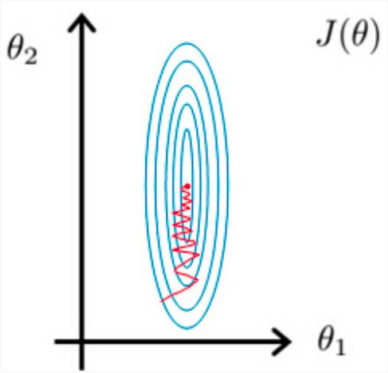


Normalization

Input space

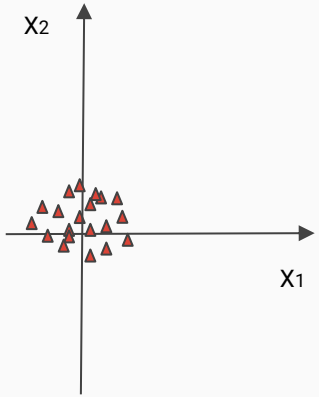


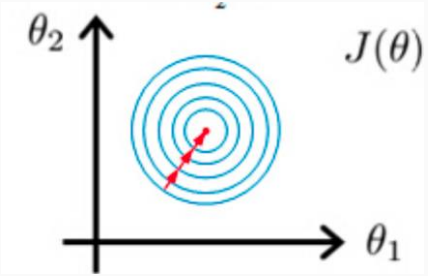
Parameter space



Normalization

Input space

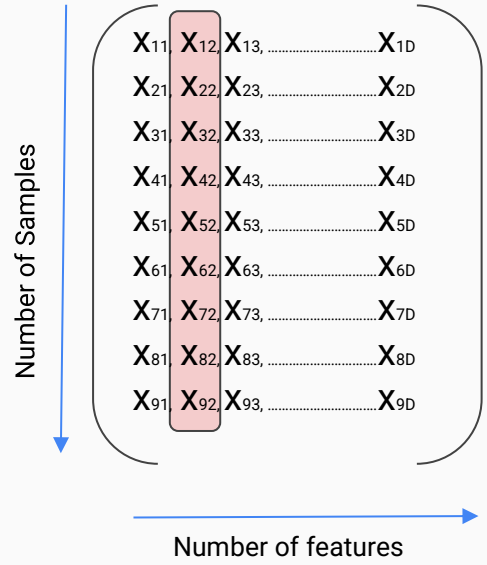




Normalization

- There are different ways to **normalize** the input features.
- One popular approach consists to **shift** and **scale each feature** so that its **mea**n is **zero** and its **variance** is **one**.

Input Features



Mean (for each feature)

$$\boldsymbol{\mu} = [\mu_1, ..., \mu_d, ..., \mu_D]^T$$

$$\mu_d = \frac{1}{N}\sum_{i=1}^{N} x_{id}$$

Standard deviation (for each feature)

$$\boldsymbol{\sigma} = [\sigma_1, ..., \sigma_d, ..., \sigma_D]^T$$

$$\sigma_d = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_{id} - \mu_d)^2}$$

$$\mathbf{x}_i^{norm} = \frac{\mathbf{x}_i - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

Normalization

Input Features

$\mathbf{X}=$

1.5	0.6	7.0
1.4	1.6	0.0
1.3	1.0	-6.0
1.5	1.4	1.6

Mean: [1.425, 1.15 , 0.65]

Std : [0.08291562, 0.38405729, 4.63330336]

Important: We need to **remember** these values, so that **test data** can be shifted and scaled the same way that the training samples!

Normalized Features

$\mathbf{X}^{norm} =$

0.90453403,	-1.43207802,	1.37051246
-0.30151134,	1.1717002 ,	-0.14028868
-1.50755672,	-0.39056673,	-1.43526108
0.90453403,	0.65094455,	0.2050373

```
X = np.array([[1, 2, 3],
              [0, 0, 2],
              [0, 0, 1]])
```

Numpy

```
X_normalized = (X - X.mean(axis=0)) / X.std(axis=0)
X_normalized
```

Scikit Learn

```
sklearn.preprocessing.scale(X)
```

```
array([[ 1.4142,  1.4142,  1.2247],
       [-0.7071, -0.7071,  0.    ],
       [-0.7071, -0.7071, -1.2247]])
```

Batch Normalization

- So far, we applied the normalization to the **input features** only.

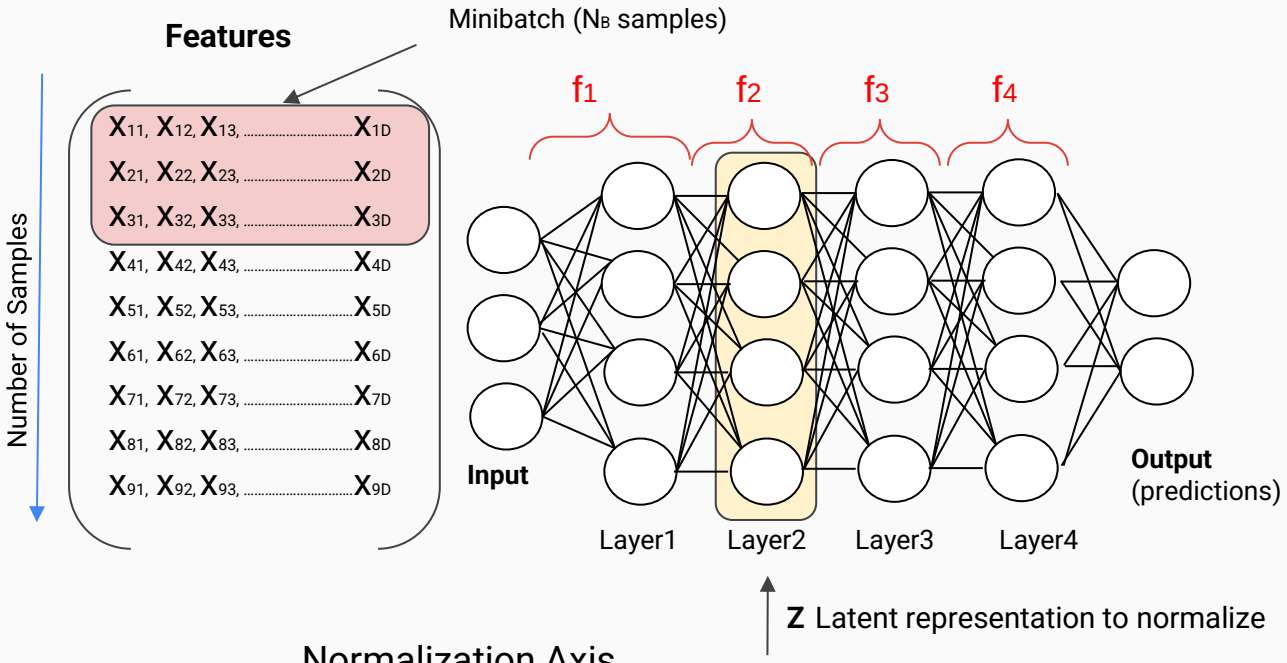
Can we normalize the features learned in the hidden layers as well?

This is a great idea that turned out to **speed up** neural network training significantly

There are a few complications to consider though:

- The internal features **change during training** (while the input ones do not change)
- Normally, we update the parameters using minibatches (SGD) and not using the full training set.

Batch Normalization



Batch Normalization

1. Compute the minibatch mean:

$$\boldsymbol{\mu}_B = \frac{1}{N_B} \sum_{i=1}^{N_B} \mathbf{z}_i$$

2. Compute the minibatch std:

$$\boldsymbol{\sigma}_B = \frac{1}{N_B} \sum_{i=1}^{N_B} (\mathbf{z}_i - \boldsymbol{\mu}_B)^2$$

3. Normalize:

$$\mathbf{z}_i^{norm} = \frac{\mathbf{z}_i - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

Batch Normalization

- Batch normalization is similar to **input normalization**, but it is computed at a **minibatch level** in the **internal layers.**

- Often, we add additional **learnable parameters** that allow the neural network to set a mean different than 0 and a variance different than 1.

4. Learnable scale and shift:

Scale Shift

$$\mathbf{z}_i^{batchnorm} = \boldsymbol{\gamma}\mathbf{z}_i^{norm} + \boldsymbol{\beta}$$

The **scale** and **shift factors** are **learned** from data (just like any other parameter of the network)

These parameters give more flexibility to the network. In practice, however, they do not change too much during training (γ=1, β=0).

Batch Normalization

1. Compute the minibatch mean:

$$\boldsymbol{\mu}_B = \frac{1}{N_B}\sum_{i=1}^{N_B}\mathbf{z}_i$$

2. Compute the minibatch std:

$$\boldsymbol{\sigma}_B = \frac{1}{N_B}\sum_{i=1}^{N_B}(\mathbf{z}_i - \boldsymbol{\mu}_B)^2$$

3. Normalize:

$$\mathbf{z}_i^{norm} = \frac{\mathbf{z}_i - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

Batch Normalization

- What do we do at **inference/test time**?

- At inference/test time, we might even have no minibatches (e.g., think about a user feeding a single image into the model and asking a prediction for that single image).

- To avoid this issue, we compute **running averages** during training (and we used them at inference time):

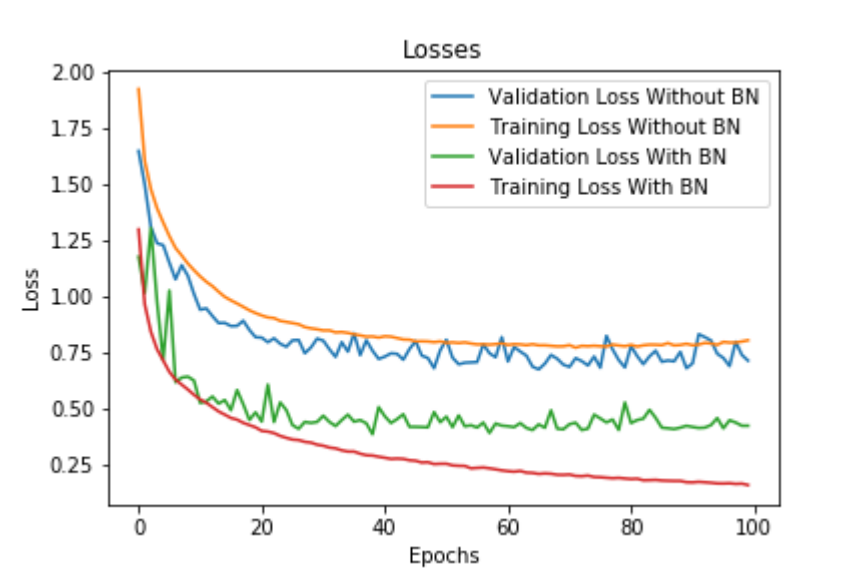
$$\boldsymbol{\mu}_B^{mov} = \alpha \boldsymbol{\mu}_B^{mov} + (1-\alpha)\boldsymbol{\mu}_B$$

$$\boldsymbol{\sigma}_B^{mov} = \alpha \boldsymbol{\sigma}_B^{mov} + (1-\alpha)\boldsymbol{\sigma}_B$$

$$\mathbf{z}_i^{norm-test} = \frac{\mathbf{z}_i - \boldsymbol{\mu}_B^{mov}}{\sqrt{\sigma_B^{mov^2} + \epsilon}} \qquad \mathbf{z}_i^{batchnorm-test} = \boldsymbol{\gamma}\mathbf{z}_i^{norm-test} + \boldsymbol{\beta}$$

Batch Normalization

- Batchnorm significantly **speeds up training** and generally leads to better convergence:



The effect of batch normalization is to make the **parameter space** more “friendly” and **easier to optimize**.

Training DNNs is complicated by the fact that the **distribution of each layer’s inputs changes during training**, as the parameters of the previous layers change (**internal covariate shift**).

This slows down the training because all layers are changed during an update and the update procedure is forever chasing a **moving target**.

Batchnorm offers a solution to this problem. Even if the distribution of the previous layer changes, at least we can expect it to have a **certain mean** and **variance**.

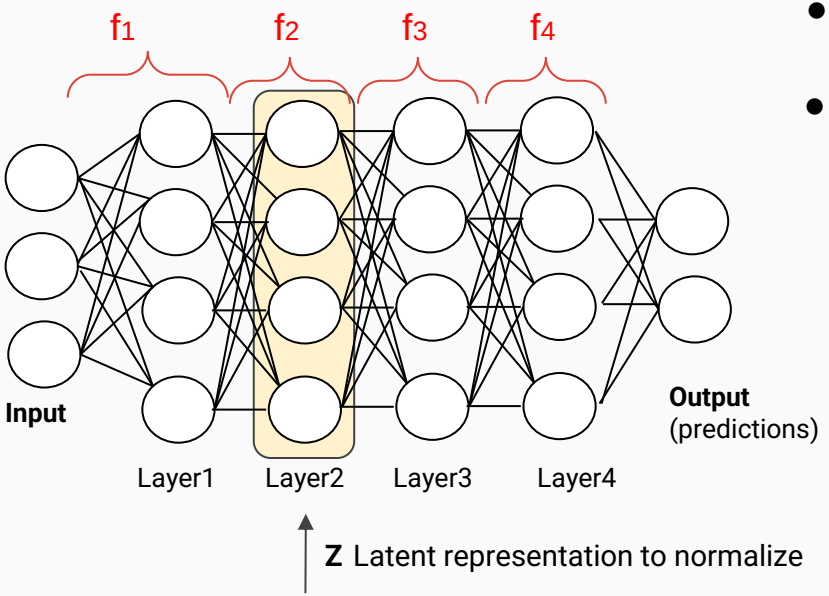
Batch Normalization

Few **tips** and remarks on batch normalization:

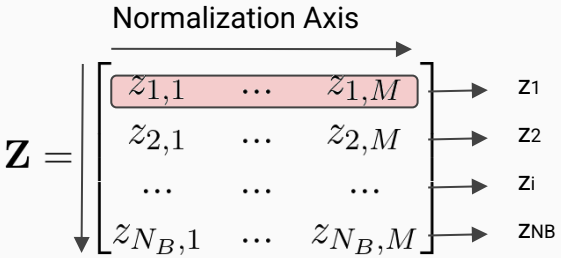
- You can use batch norm either **before** or **after** the non-linearity. In most cases, it is applied before.
- You can use **larger learning rates**. This is because batchnorm makes the network **more stable** during training.
- Batchnorm makes training **less sensitive** to **weight Initialization**.
- You can do it for the **input features** as well (as a replacement for the input normalization).

Layer Normalization

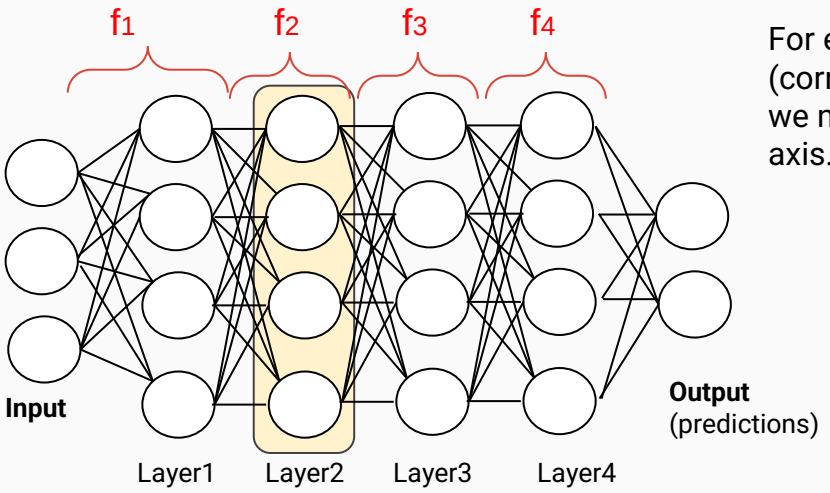
- There are different ways to perform normalization.
- One other method that has been proven effective is **layer normalization**.



- **Layernorm** is similar to **batchnorm**. T
- The main difference is that the normalization axis is swapped: we normalize over the **neuron axis** rather than the minibatch .



Layer Normalization



For each vector $\mathbf{z}_i$ (corresponding to an input $\mathbf{x}_i$), we normalize over the neuron axis.

Z Latent representation to normalize

M is the number of neurons

Normalization Axis (batch axis)

$$\mathbf{Z} = \begin{bmatrix} z_{1,1} & \cdots & z_{1,M} \\ z_{2,1} & \cdots & z_{2,M} \\ \cdots & \cdots & \cdots \\ z_{N_B,1} & \cdots & z_{N_B,M} \end{bmatrix}$$

z_1 z_2 z_i z_{NB}

Layer Normalization

1. Compute the minibatch mean:

$$\mu_i = \frac{1}{M}\sum_{m=1}^{M} z_{im}$$

2. Compute the minibatch std:

$$\sigma_i = \frac{1}{M}\sum_{i=m}^{M}(z_{im} - \mu_i)^2$$

3. Normalize:

$$\mathbf{z}_i^{norm} = \frac{\mathbf{z}_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

Layer Normalization

- Similar to batchnorm, layernorm often adds additional **learnable parameters** that allow the neural network to set a mean different than 0 and a variance different than 1.

4. Learnable scale and shift:

Scale Shift

$$\mathbf{z}_i^{layernorm} = \gamma \mathbf{z}_i^{norm} + \beta$$

The **scale** and **shift factors** are **learned** from data (just like any other parameter of the network).

These parameters give more flexibility to the network. In practice, however, they do not change too much during training (γ=1, β=0).

Layer Normalization

1. Compute the minibatch mean:

$$\mu_i = \frac{1}{M}\sum_{m=1}^{M} z_{im}$$

2. Compute the minibatch std:

$$\sigma_i = \frac{1}{M}\sum_{i=m}^{M}(z_{im} - \mu_i)^2$$

3. Normalize:

$$\mathbf{z}_i^{norm} = \frac{\mathbf{z}_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

Layer Normalization

- What can we do at **test/inference time**?
- Due to the choice of the normalization axis, we can do exactly the **same computations** for both training and test/inference.
- We do not need to keep track of the running average and running std as for batchnorm.
- *Is layernorm better than batchnorm?*
- This is not clear in the community as the best technique depends on the task and data used to train the machine learning model.

Layer Normalization

1. Compute the minibatch mean:

$$\mu_i = \frac{1}{M}\sum_{m=1}^{M} z_{im}$$

2. Compute the minibatch std:

$$\sigma_i = \frac{1}{M}\sum_{i=m}^{M}(z_{im} - \mu_i)^2$$

3. Normalize:

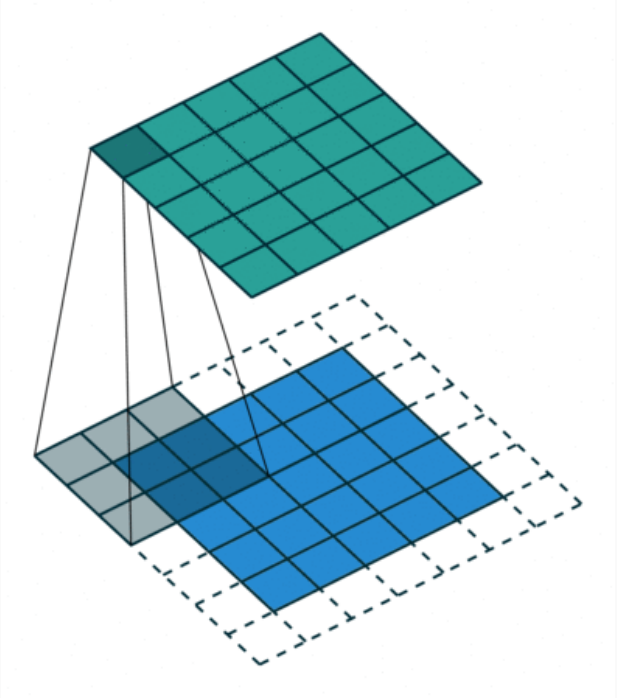
$$\mathbf{z}_i^{norm} = \frac{\mathbf{z}_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

Convolutional Neural Networks

Motivation

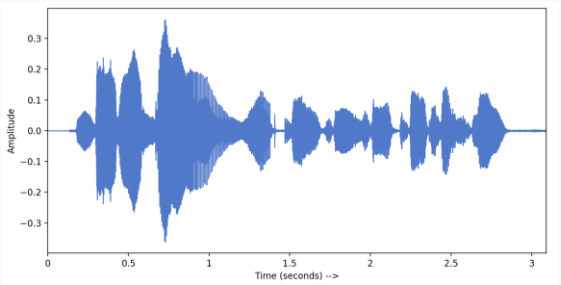
- **Convolutional neural networks** (CNNs) are simply neural networks that use **convolution** in place of general **linear transformation**.

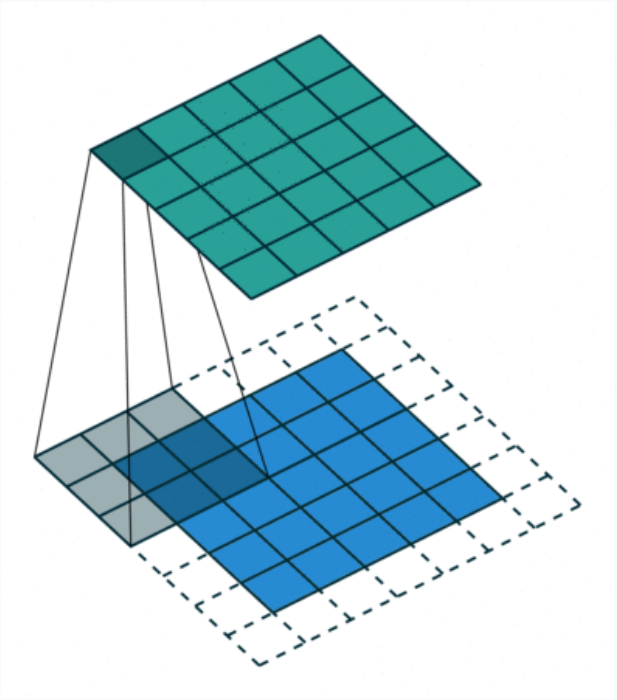
- The main difference with MLPs are the following:
  - **Local connections** (differently to MLPs that are “fully-connected” models).
  - **Weight Sharing** (differently to MLPs that employ different weights for different neurons).



Motivation

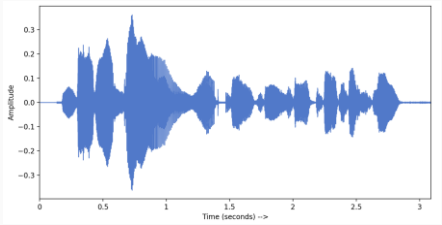
- Several signals naturally have **spatial** or **temporal correlations** (images, audio, etc.).
- **Local connections** can capture **local patterns** better than fully-connected models.

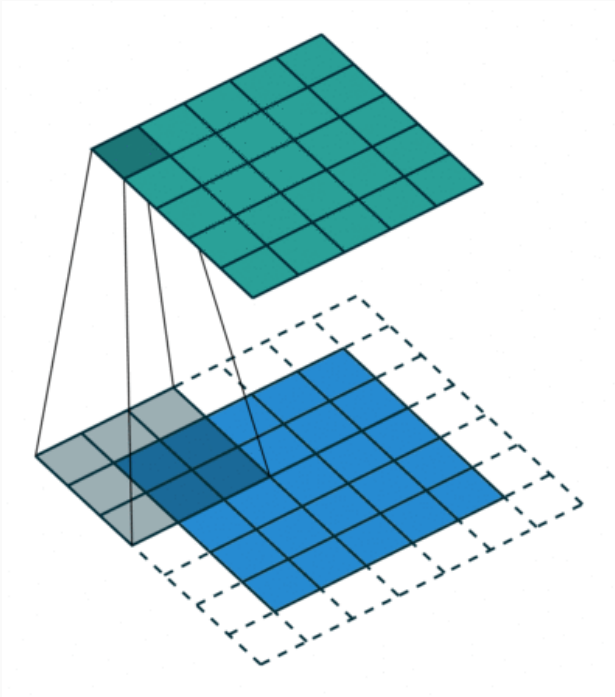




Motivation

- **Local patterns** can be everywhere in the image.
- Thus we can search for all the local patterns by **sliding the same kernel** (i.e., with the same weights) **over** all the **input.**
- This way, we have the chance to react to patterns that are in **different positions**.

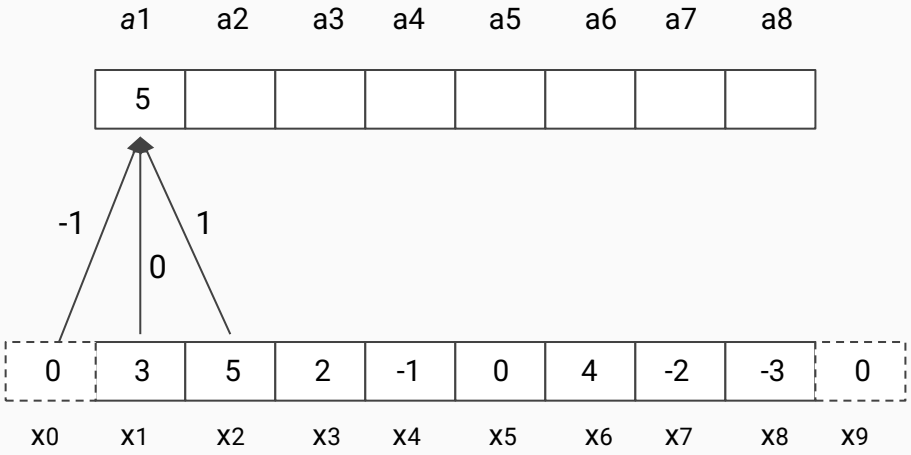




What is a convolution?

- **Convolution** is a standard operation, long used in **compression** and **signal processing** (*DFT*), **computer vision** ("*edge detection*") and **image processing** tools like Photoshop ("*custom filter*").

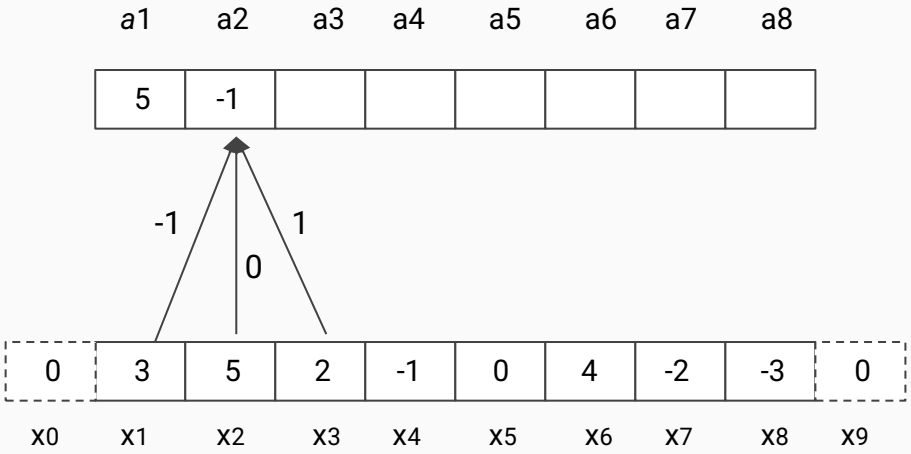
- Let's start with a **1D Convolution**:



What is a convolution?

- **Convolution** is a standard operation, long used in **compression** and **signal processing** (*DFT*), **computer vision** (“*edge detection*”) and **image processing** tools like Photoshop (“*custom filter*”).

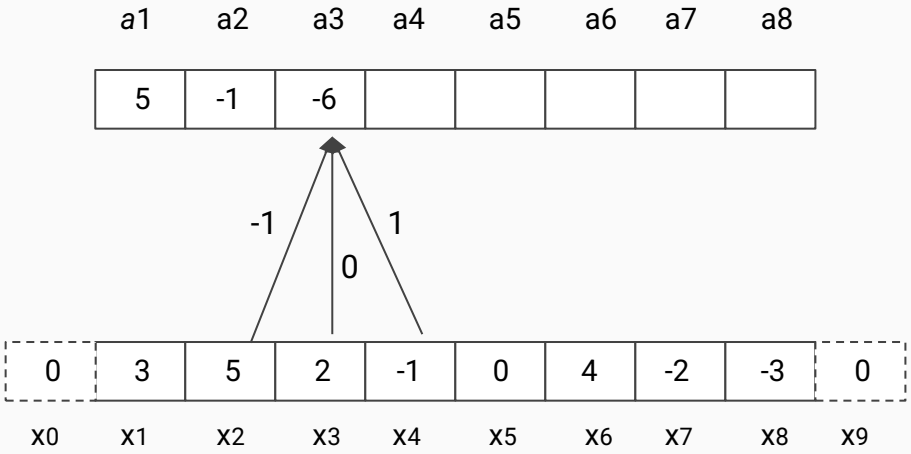
1D Convolution



What is a convolution?

- **Convolution** is a standard operation, long used in **compression** and **signal processing** (*DFT*), **computer vision** (“*edge detection*”) and **image processing** tools like Photoshop (“*custom filter*”).

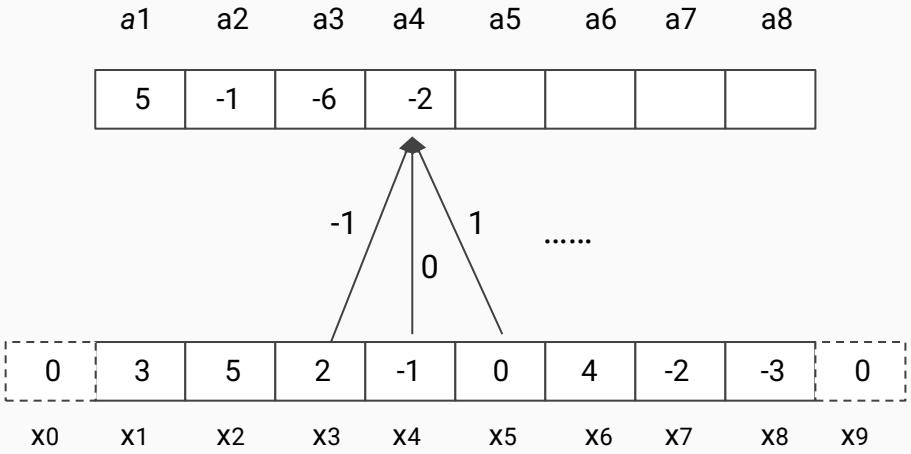
1D Convolution



What is a convolution?

- **Convolution** is a standard operation, long used in **compression** and **signal processing** (*DFT*), **computer vision** (“*edge detection*”) and **image processing** tools like Photoshop (“*custom filter*”).

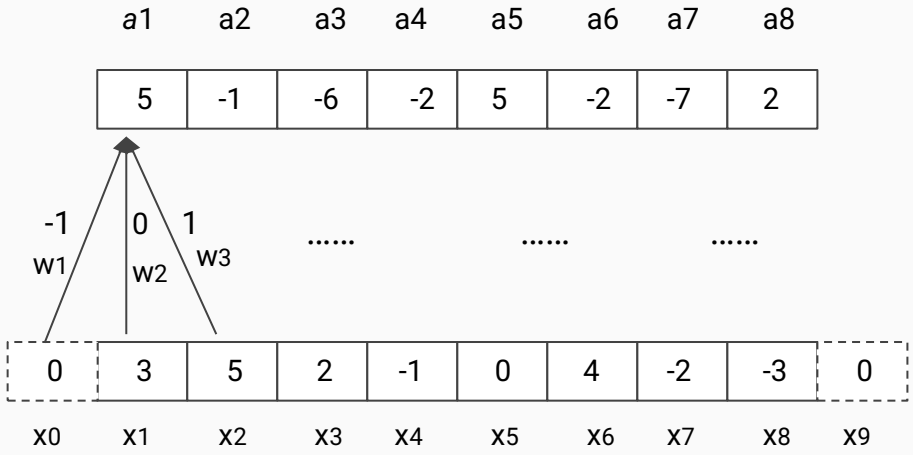
1D Convolution



What is a convolution?

- **Convolution** is a standard operation, long used in **compression** and **signal processing** (*DFT*), **computer vision** ("*edge detection*") and **image processing** tools like Photoshop ("*custom filter*").

1D Convolution



To make convolution work, we need to do **padding** at the edges of our input.

Usually, we just add **zeros.**

The amount of padding depends on the **filter (kernel) size**.

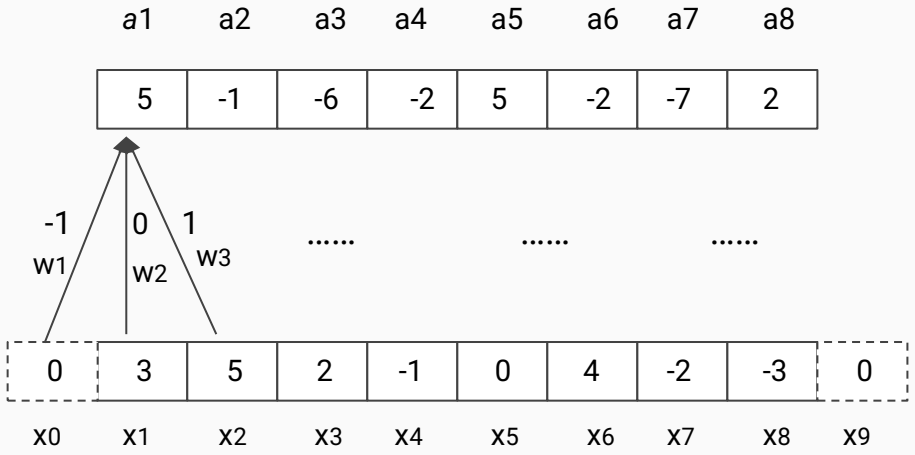
Can you guess the <u>total number of zeros</u> to add? Let's assume an odd kernel size Ks

Total	On each side
$2p = K_s - 1$	$p = (K_s - 1)/2$

What is a convolution?

- **Convolution** is a standard operation, long used in **compression** and **signal processing** (*DFT*), **computer vision** ("*edge detection*") and **image processing** tools like Photoshop ("*custom filter*").

1D Convolution



The generic output $a_{i'}$ can be written as:

$$a_{i'} = \sum_{i=1}^{K_s} w_i x_{i+i'-p-1}$$

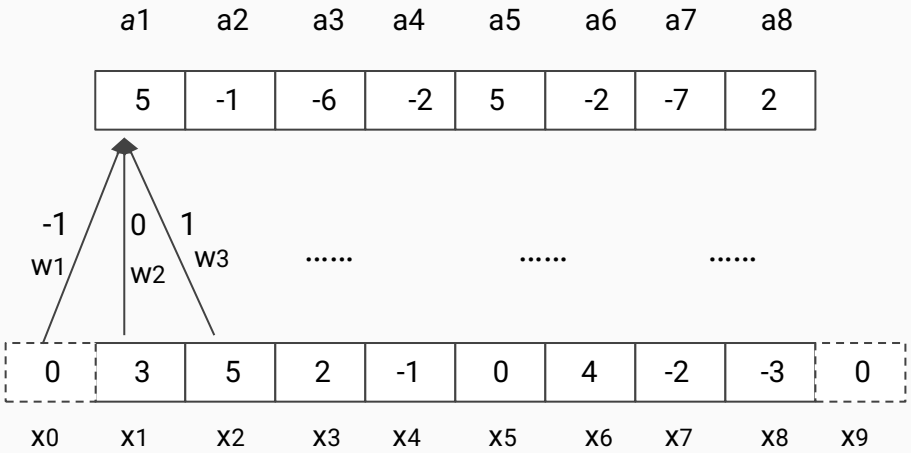
K_s → Kernel size

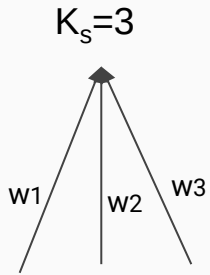
Padding

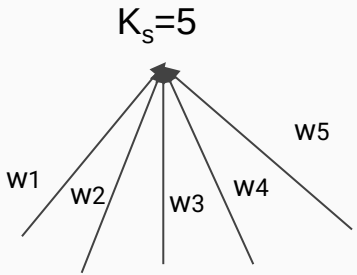
This operation is called **correlation** in math and signal processing. In the context of neural networks they called it (erroneously) **convolution**.

Kernel Size

- The **weights** of the filter w are **learned from data**.
- The **length** of the convolving **filter** (which corresponds to the number of learnable parameters) is called **kernel Size** (K_s).
- This is an architectural **hyperparameter** to set. Depending on the kernel size we can embed more or less local information.

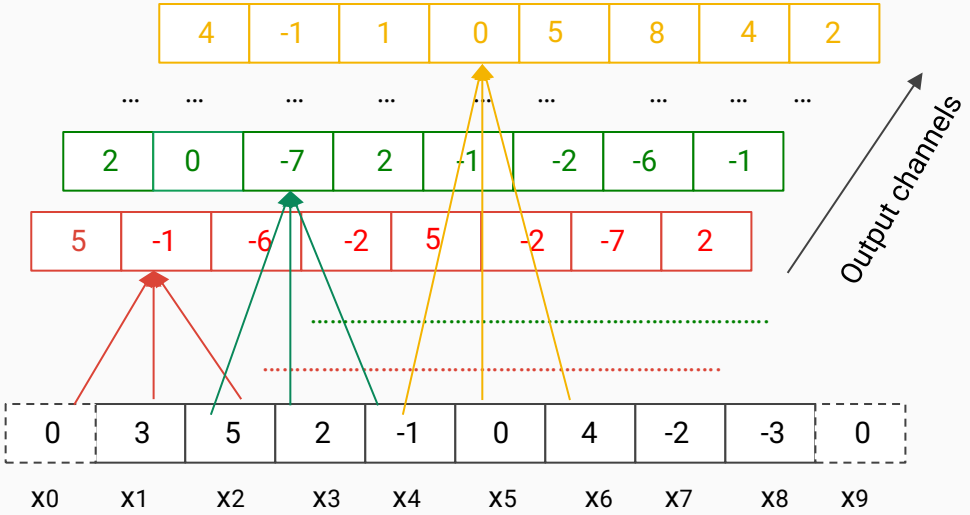






Output Channels

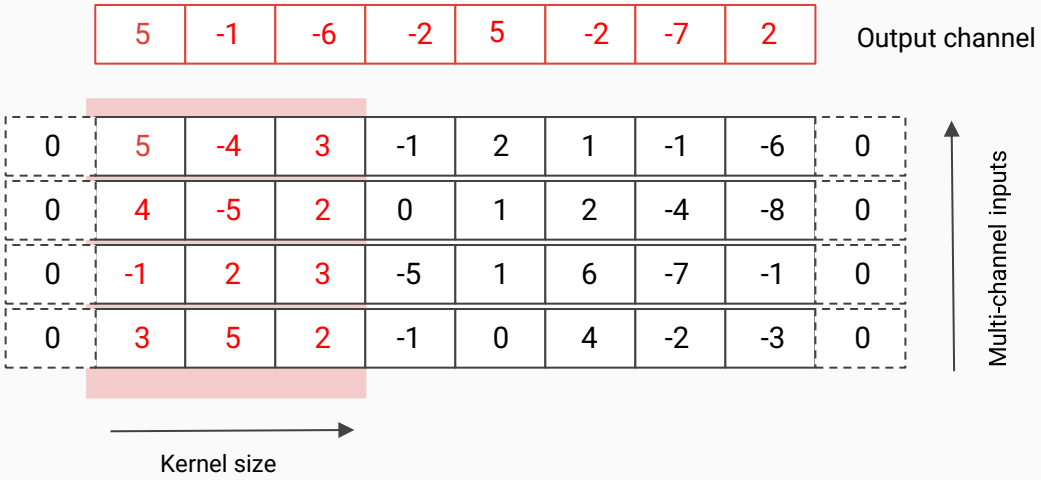
- In convolutional neural networks, we want **each filter** to react to **different local patterns**.
- To capture **multiple patterns** we have to process the input with **many different filters**.
- For each filter, the convolution produces a “filtered” version of the input that reacts to a different local pattern.



- All these outputs produced by the convolution are gathered in a single matrix with dimensionality (signal length **L** x **number of output channels**).
- The number of output channels corresponds to the number of filters.

Input channels

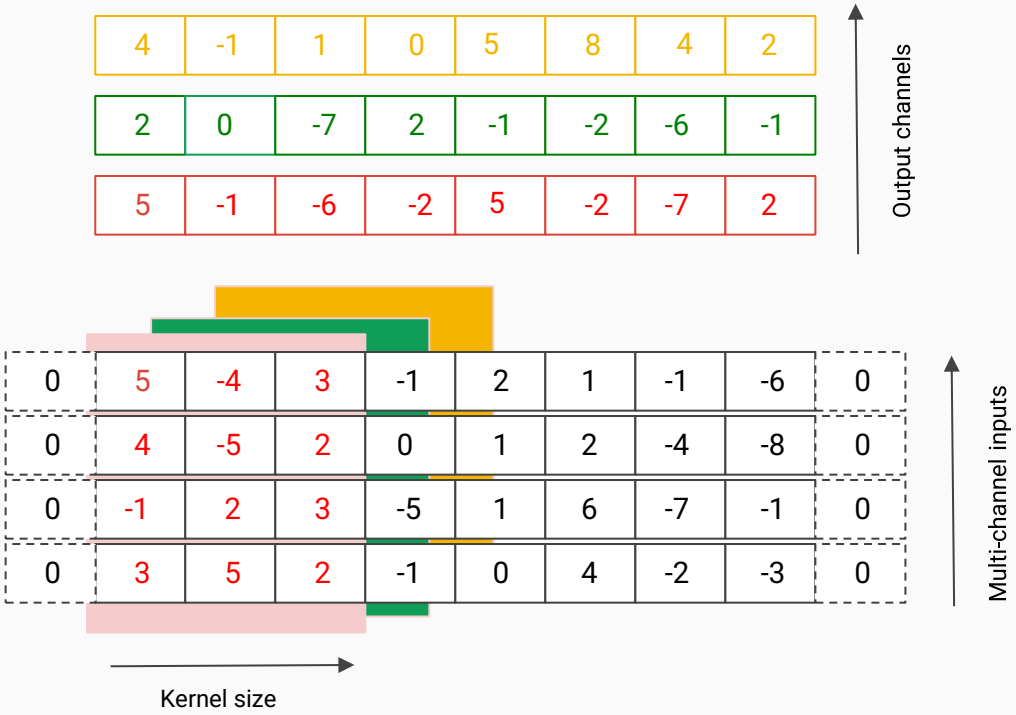
- So far, we have assumed **single-channel inputs**.
- The convolutional framework is flexible enough to manage **multi-channel inputs** as well.
- In general, the convolution takes a **multi-channel input**, filters it with some filters, and returns a **multi-channel output.**



- The multi-channel input is processed by a filter of dimensionality (**kernel_size x input channels**).
- This operation creates a single **output channel.**

Input channels

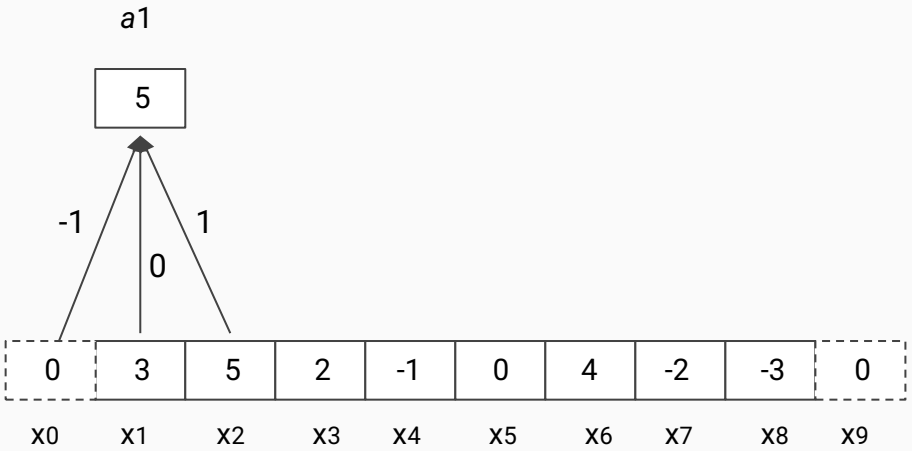
- We can process the same multi-channel input with many filters to get multiple outputs



Stride Factor

- The **stride factor** is another **hyperparamete**r of the convolutional layer.
- It quantifies the amount of movement (step size) by which we slide a filter over an input.
- So far, we have assumed a stride factor = 1. If stride factor > 1, the effect is to **compress** (downsample) the input.

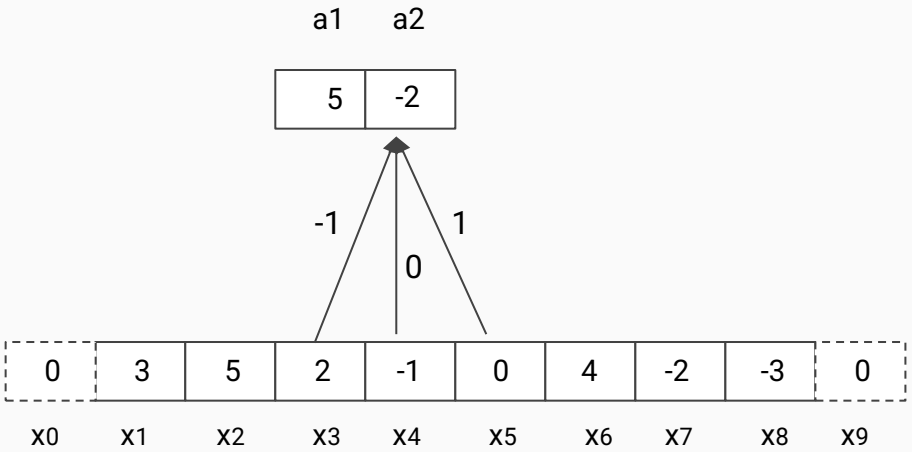
Stride factor = 3



Stride Factor

- The **stride factor** is another **hyperparamete**r of the convolutional layer.
- It quantifies the amount of movement (step size) by which we slide a filter over an input.
- So far, we have assumed a stride factor = 1. If stride factor > 1, the effect is to **compress** (downsample) the input.

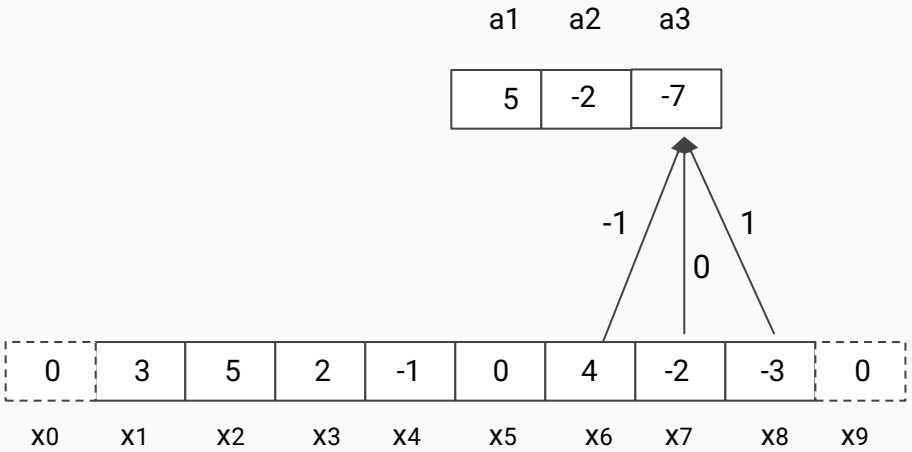
Stride factor = 3



Stride Factor

- The **stride factor** is another **hyperparamete**r of the convolutional layer.
- It quantifies the amount of movement (step size) by which we slide a filter over an input.
- So far, we have assumed a stride factor = 1. If stride factor > 1, the effect is to **compress** (downsample) the input.

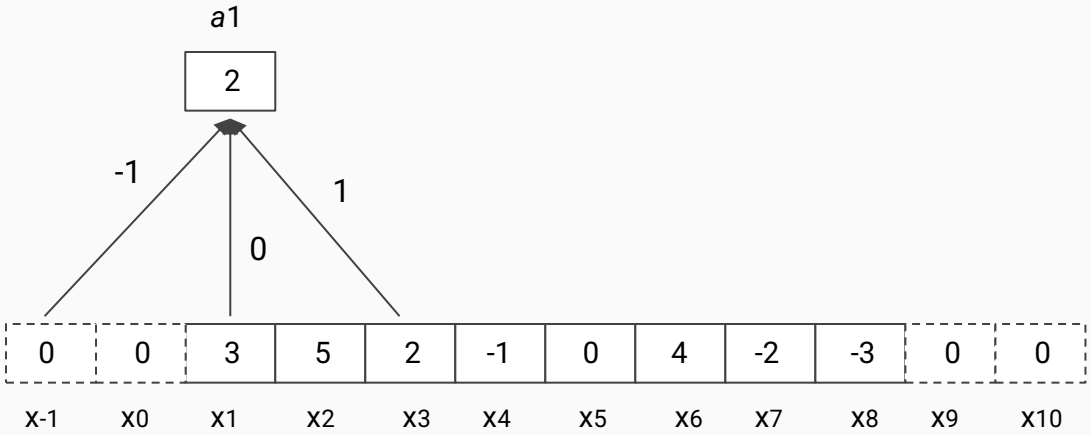
Stride factor = 3



Dilated Convolution

- Dilated Convolution is a technique that expands the filter by **inserting holes** between the its **consecutive elements**.
- This can be done to cover a **larger area** of the input.
- So far, we have assumed a dilation = 1. If dilation > 1, the effect is the following

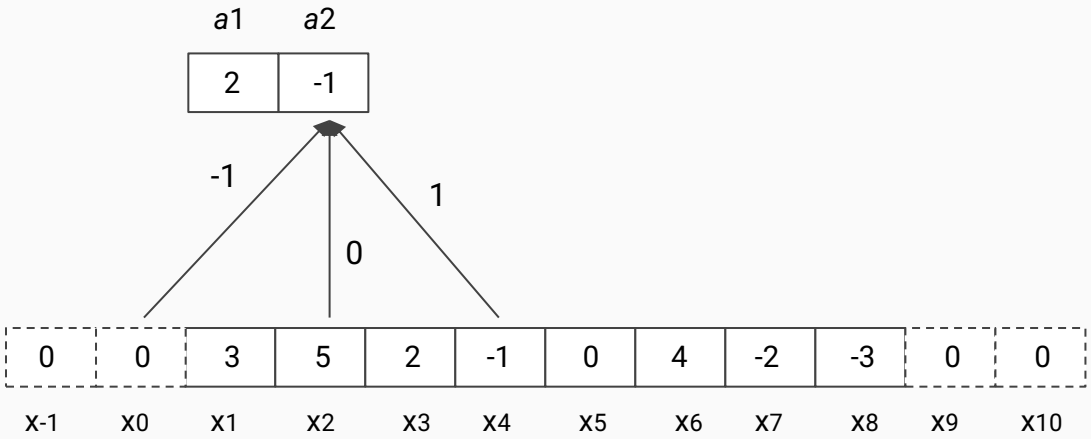
Dilation = 2



Dilated Convolution

- Dilated Convolution is a technique that expands the filter by **inserting holes** between the its **consecutive elements**.
- This can be done to cover a **larger area** of the input.
- So far, we have assumed a dilation = 1. If dilation > 1, the effect is the following

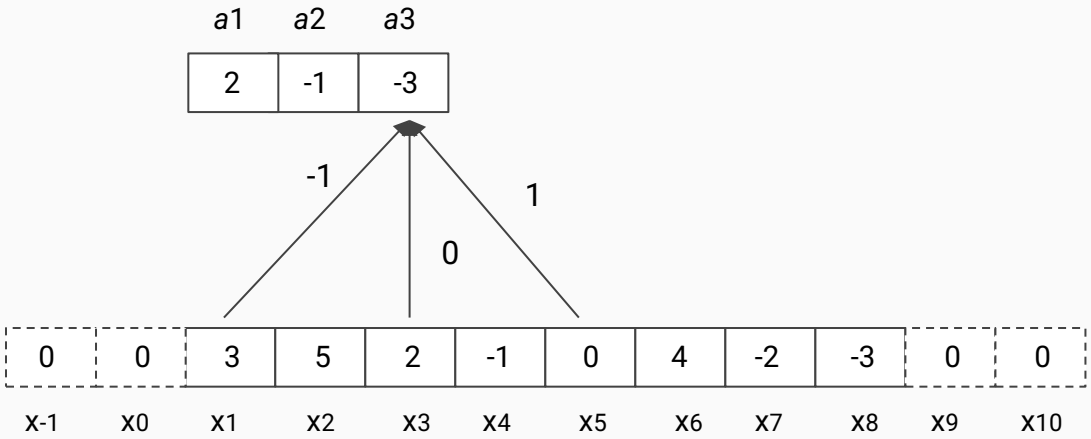
Dilation = 2



Dilated Convolution

- Dilated Convolution is a technique that expands the filter by **inserting holes** between the its **consecutive elements**.
- This can be done to cover a **larger area** of the input.
- So far, we have assumed a dilation = 1. If dilation > 1, the effect is the following

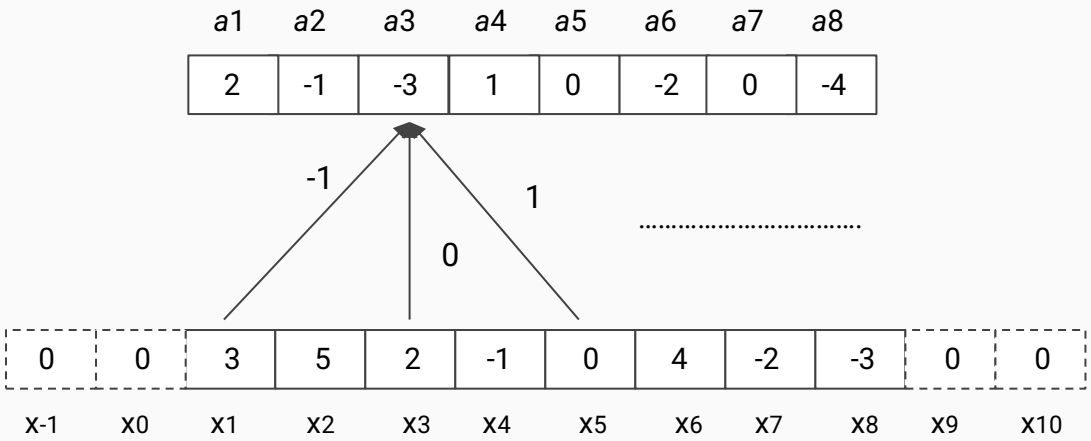
Dilation = 2



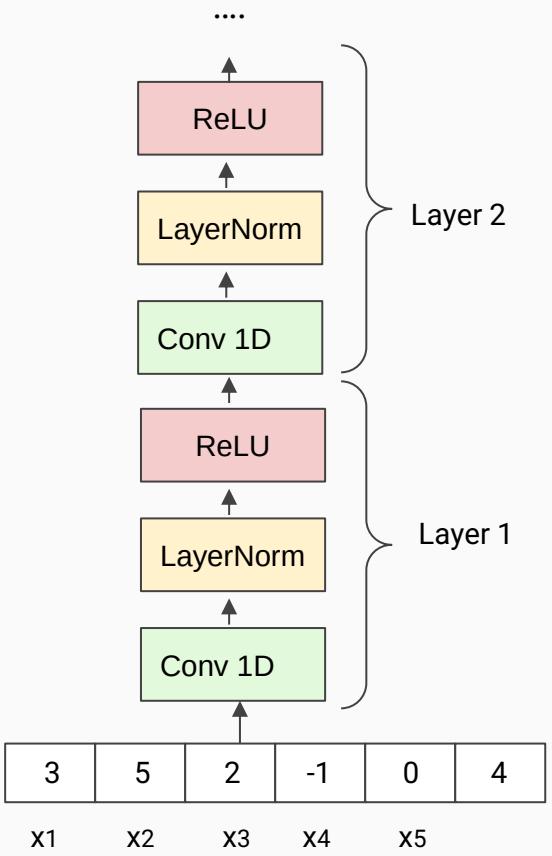
Dilated Convolution

- Dilated Convolution is a technique that expands the filter by **inserting holes** between the its **consecutive elements**.
- This can be done to cover a **larger area** of the input.
- So far, we have assumed a dilation = 1. If dilation > 1, the effect is the following

Dilation = 2

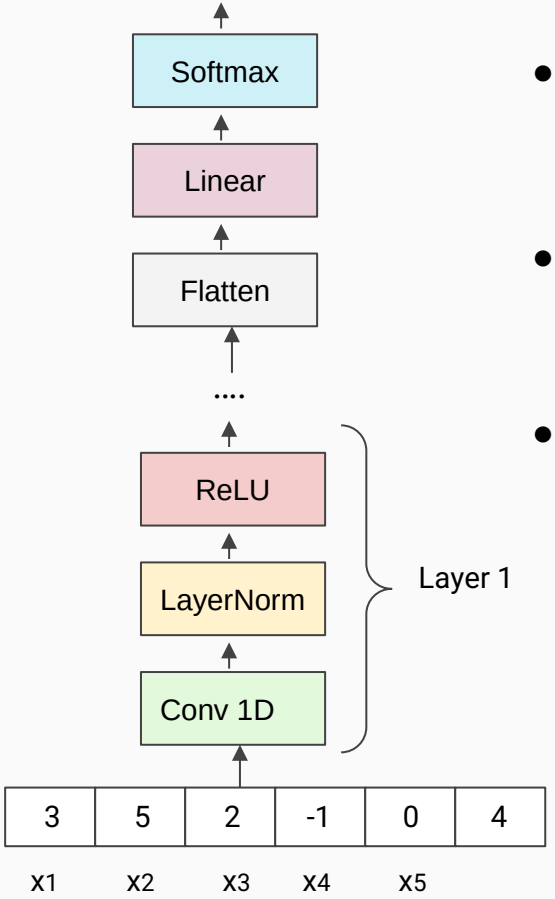


Stacking Convolutional Layers



- We can stack **multiple convolutional layers** to form a deep convolutional network.
- Optionally, **normalization** is applied right after the convolution (e.g., layernorm or batchnorm).
- Then, a **non-linearity** is applied (e.g., ReLU or LeakyReLU).
- The set of features after the non-linearity is called **feature maps.**
- We have one **feature map** for each **output channel** of the convolution.

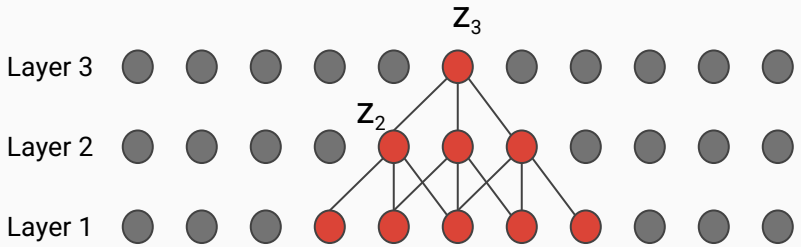
Stacking Convolutional Layers



- After stacking multiple convolutional layers, we can apply a final **linear transformation** (i.e., the standard "*fully-connected*" one implemented with dot product).
- Note that the linear layer expects in input a **vector**, while convolution is providing in output a **matrix** (with dimension input length x output channels)**.**
- We thus apply a flatten operation, that stacks in a single big vector all the output channels.
- Then we can apply a **softmax** (needed if we are solving a **multiclass classification** problem).

Receptive Field

- The **receptive field** in Convolutional Neural Networks (CNN) is the **region of the input space** that affects a particular unit of the network.



- The receptive field depends on different factors, including:

1. Kernel Size
2. Number of Layers
3. Stride Factor
4. Dilation factor

- The receptive field for the unit z_3 at Layer 3 is 5 because this output depends on 5 inputs.
- What is the receptive field for unit z_2? It is 3

Training

- *How can we train convolutional neural networks?*

  As usual, we use **gradient descend** (and all the variants introduces in lecture 2).

- *How can we compute the gradient?*

  As for MLP, we use the **backpropagation algorithm**.

  The expression for the gradient will be different from that derived for an MLP.

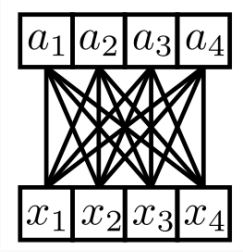
  Fortunately, **PyTorch** is going to compute all the gradients of interest for us using its **automatic differentiation** engine.

  All the methods mentioned for MLPs (e.g., initialization, normalization, residual/skip/dense connections) can be used for CNNs as well.

From MLPs to CNNs

"Fully-Connected" Neural Network

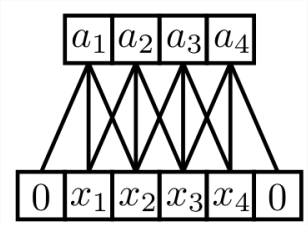
$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$



CNN are a special case of MLPs

1D "locally-connected" layer

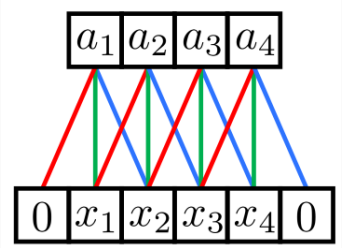
$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & 0 & 0 & 0 \\ 0 & w_{2,1} & w_{2,2} & w_{2,3} & 0 & 0 \\ 0 & 0 & w_{3,1} & w_{3,2} & w_{3,3} & 0 \\ 0 & 0 & 0 & w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}$$



1D Convolutional Layer (kernel_size=3, input_ch=1, output_ch_1, stride=1 ,dilation=1)

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}$$

MUCH FEWER PARAMETERS!!



Parameters

- *How many parameters do we have in 1D convolutional layers?*
- The number of parameters depends on the kernel size (K_s), input channels C_{in}, and output channels C_{out}.

- If $C_{in}=1$ and $C_{out}=1$ ⇒ $N_p = K_s$
- If $C_{in}=1$ and $C_{out}>1$ ⇒ $N_p = K_s \cdot C_{out}$
- If $C_{in}>1$ and $C_{out}>1$ ⇒ $N_p = K_s \cdot C_{in} \cdot C_{out}$

2D Convolution

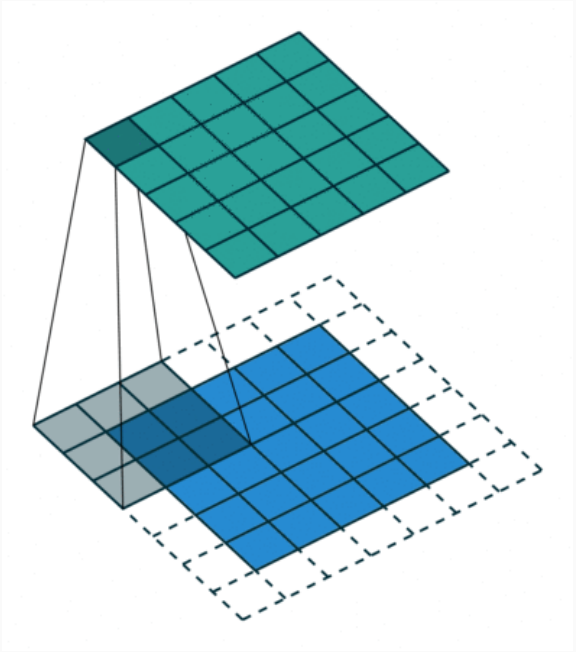
- The same mechanism seen for 1D convolutions can be extended to 2D inputs (e.g., images).
- In this case, the filters are **sliding matrices** that process the input:

0	0	0	0	0	0
0	1	5	3	2	0
0	-1	-3	-3	0	0
0	0	1	2	-1	0
0	-2	0	0	6	0
0	0	0	0	0	0

Input

0	5	3
-1	-3	-3
0	1	2

2D Kernel (filter)



2D Convolution

- The same mechanism seen for 1D convolutions can be extended to 2D inputs (e.g., images).
- In this case, the filters are **sliding matrices** that process the input:

0	0	0	0	0	0
0	1	5	3	2	0
0	-1	-3	-3	0	0
0	0	1	2	-1	0
0	-2	0	0	6	0
0	0	0	0	0	0

Input

0	0	1
0	-1	1
1	1	0

2D Kernel (filter)

3	-6	-7	-5
3	4	..	..
..	..	..	..
..	..	..	..

Output

Complete it as an exercise

2D Convolution

- The 2D convolution is defined by the same hyperparameters seen for the 1D case:

  in_channels: Number of channels in the input image (C_{in}).

  out_channels: Number of channels produced by the convolution (C_{out}).

  kernel_size: Size of the convolving kernel (K_{sx}x K_{sy}).

  stride: Stride of the convolution (this time 2D).

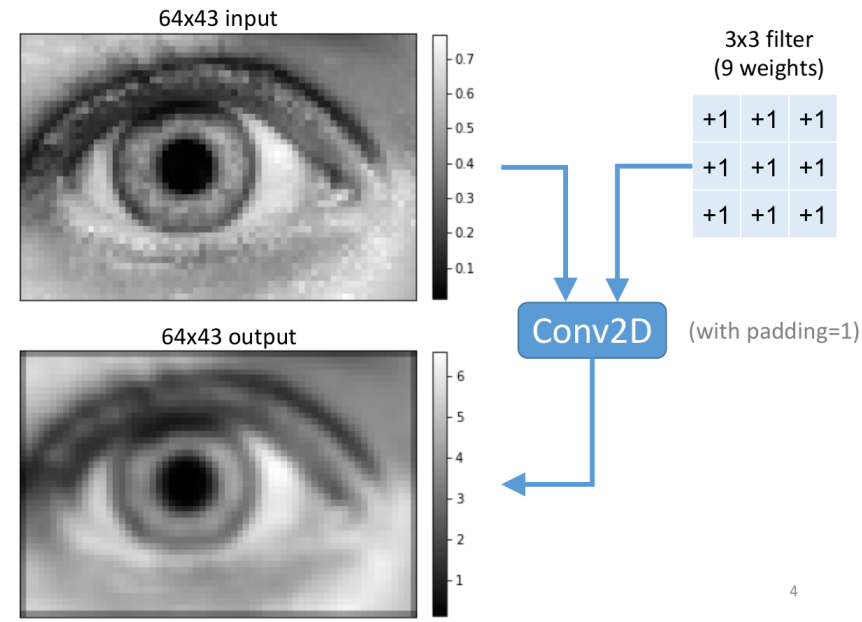
  dilation: Spacing between kernel elements (this time 2D).

- *How many parameters do we have in 2D convolutional layers?*

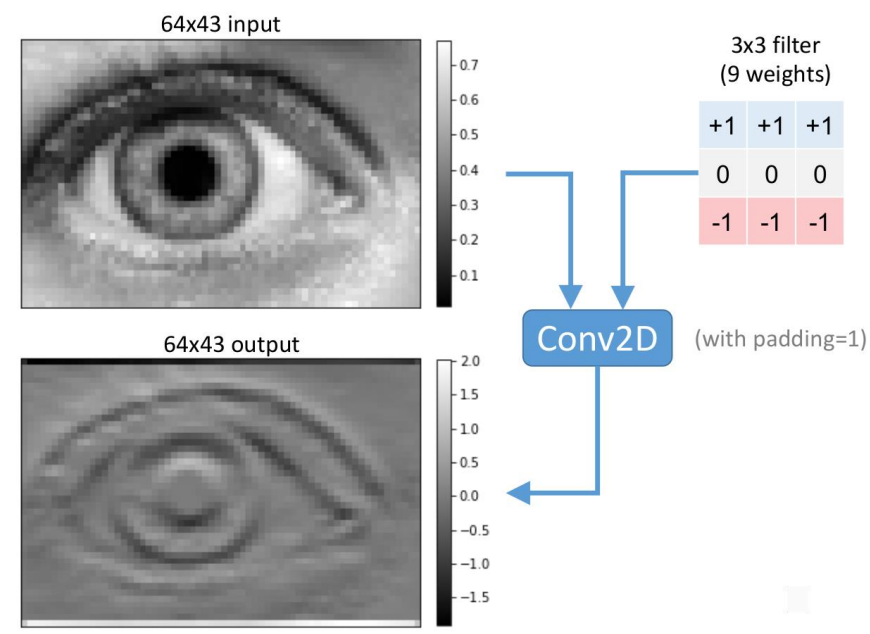
  $N_p = K_{sx} \cdot K_{sy} \cdot C_{in} \cdot C_{out}$

2D Convolution

- With 2D convolution, we can visualize more clearly the effect of the convolution.
- The effect depends on the weights of the 2D kernel used for the convolution:



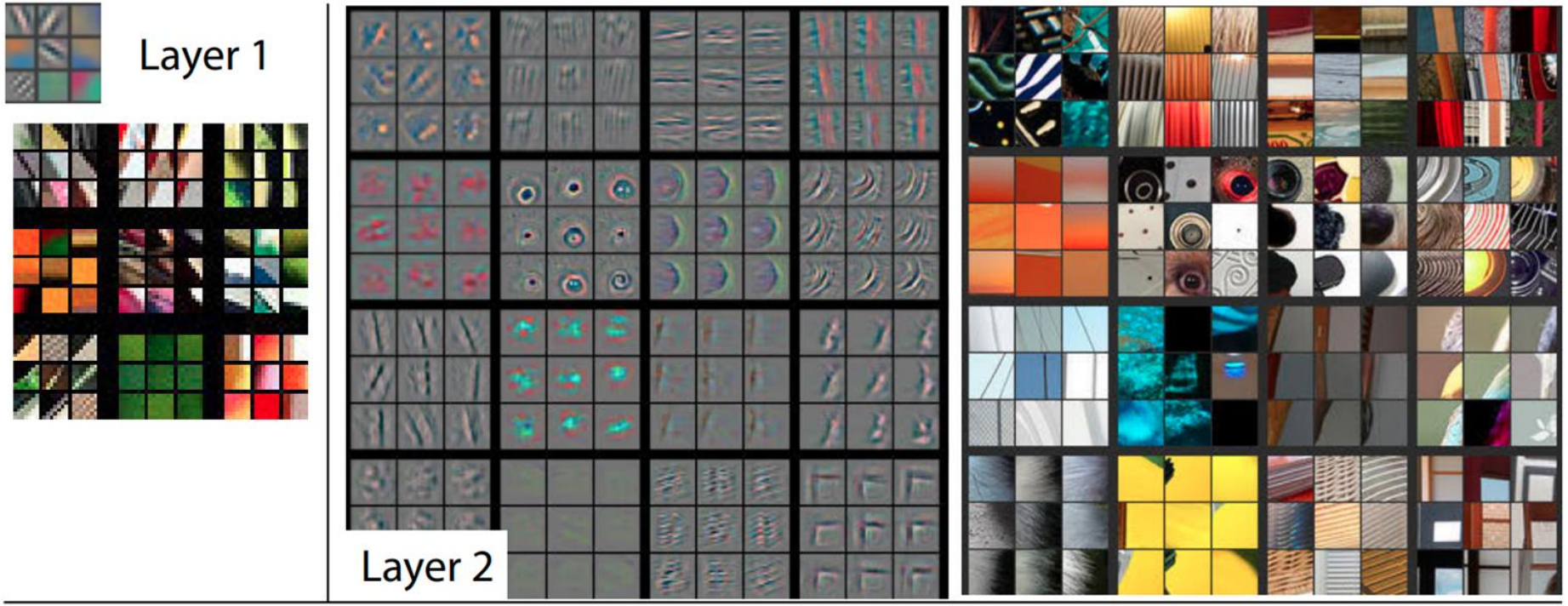
This filter makes an image more blurry



This filter extracts boundaries

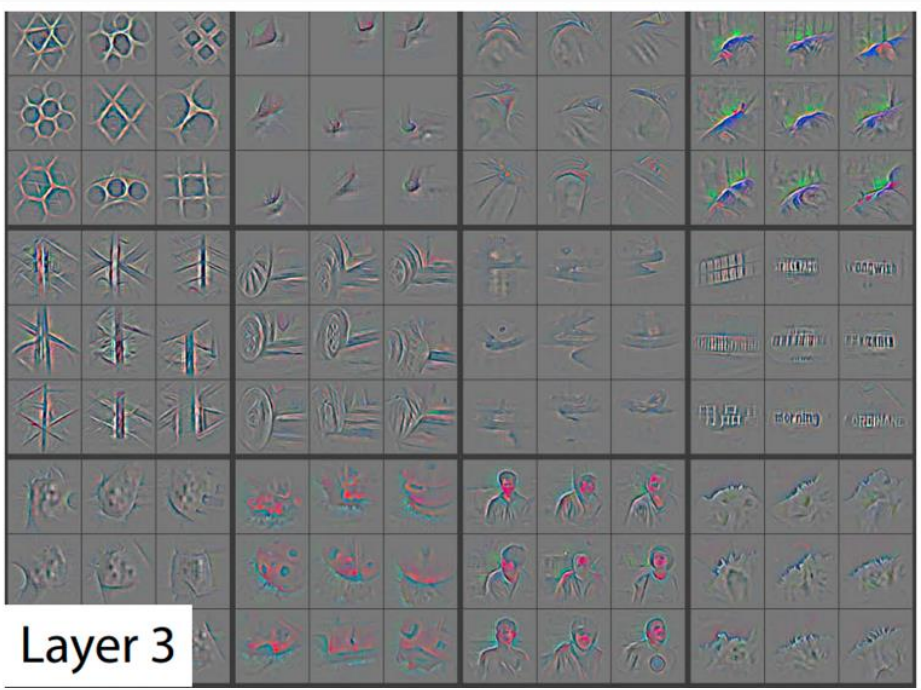
2D Convolution

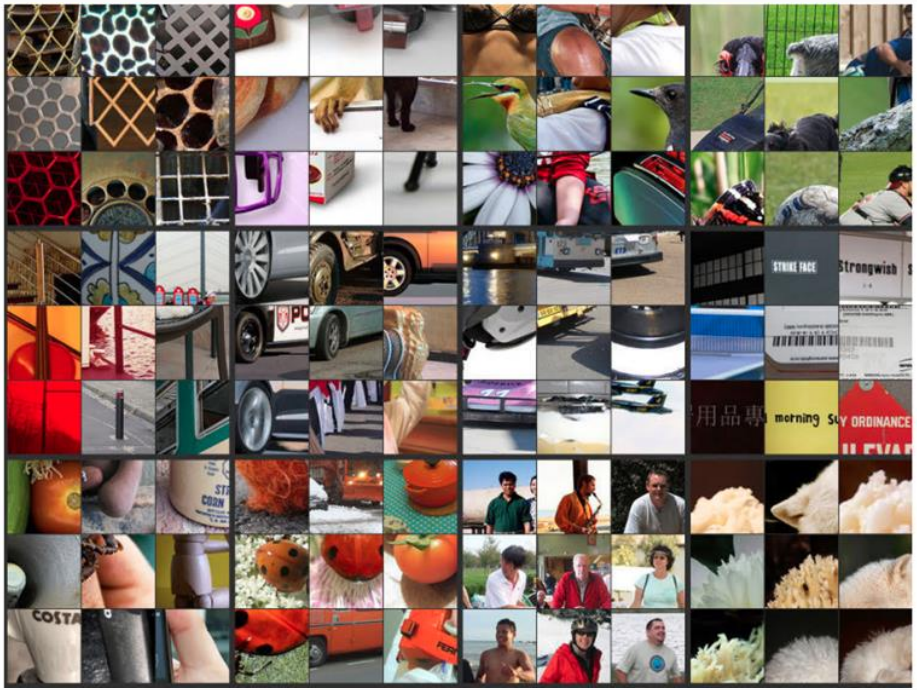
- With CNNs we can effectively learn **hierarchical features**:



2D Convolution

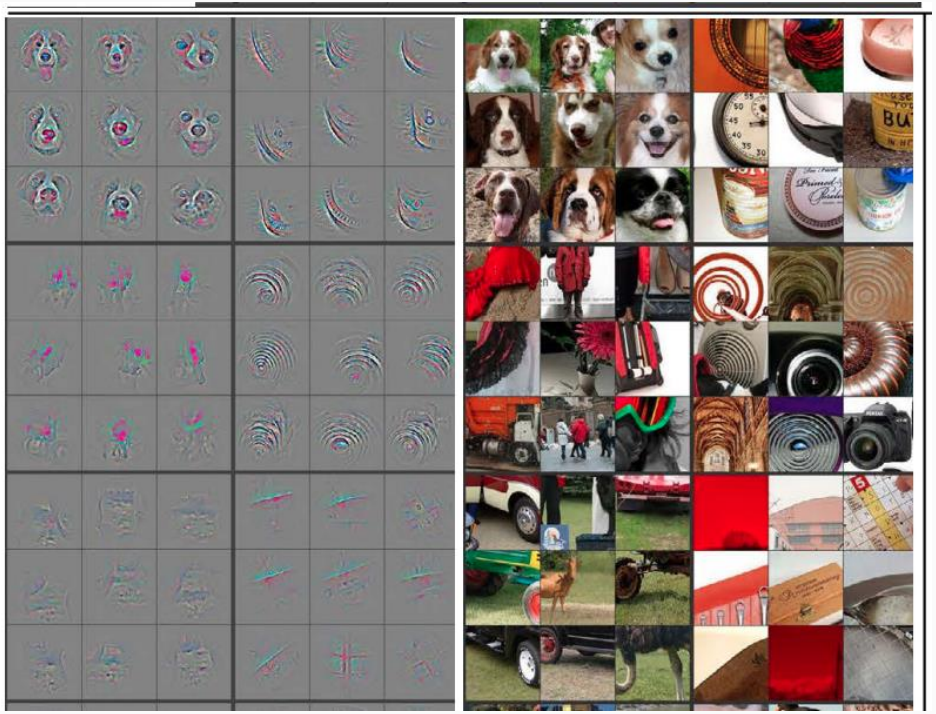
- With CNNs we can effectively learn **hierarchical features**:



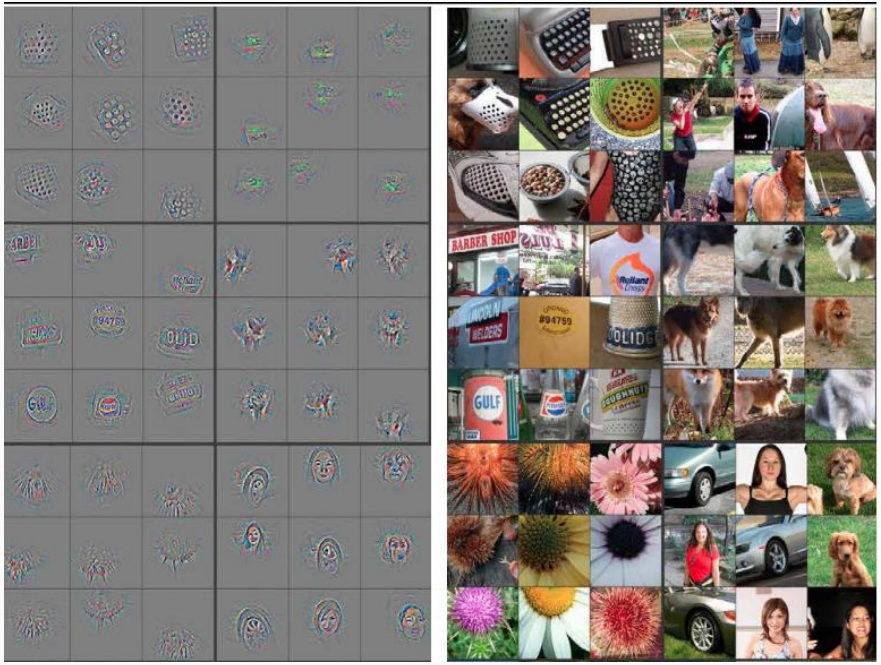


2D Convolution

- With CNNs we can effectively learn **hierarchical features**:



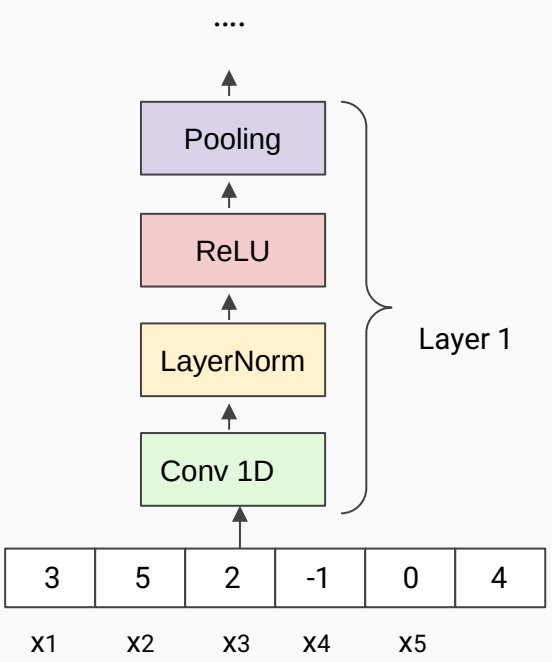
Layer 4



Layer 5

Pooling

Pooling



- In CNN architectures, we often apply **pooling** after the non-linearity.
- Pooling helps to make the feature maps **approximately invariant** to **small translations** of the **input**.
- This is a useful property if we care whether a pattern is **present** than exactly **where** it is.
- For instance, think about determining if an image contains a face. We don't need the exact location of the eyes, but rather if they are present

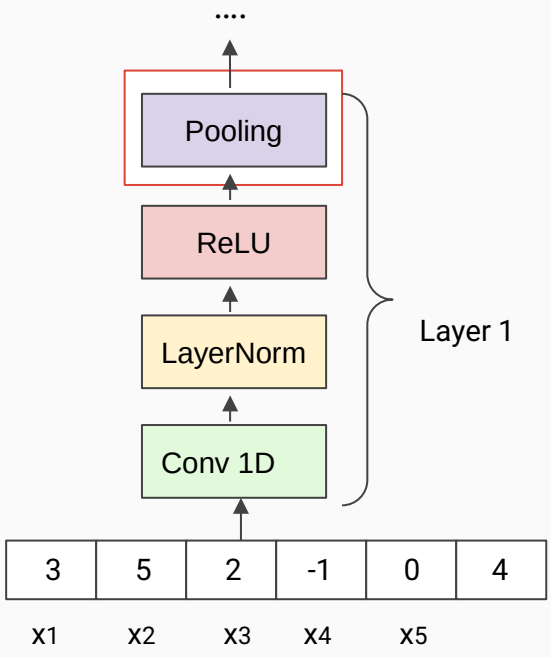
Pooling

- Similar to convolution, pooling is implemented using **sliding windows**.
- The **size** of the sliding window and its **stride factor** are **hyperparameters** of the pooling operation.
- Within the window, the pooling outputs a summary of some **statistics** of its inputs at that given location.
- Examples are:

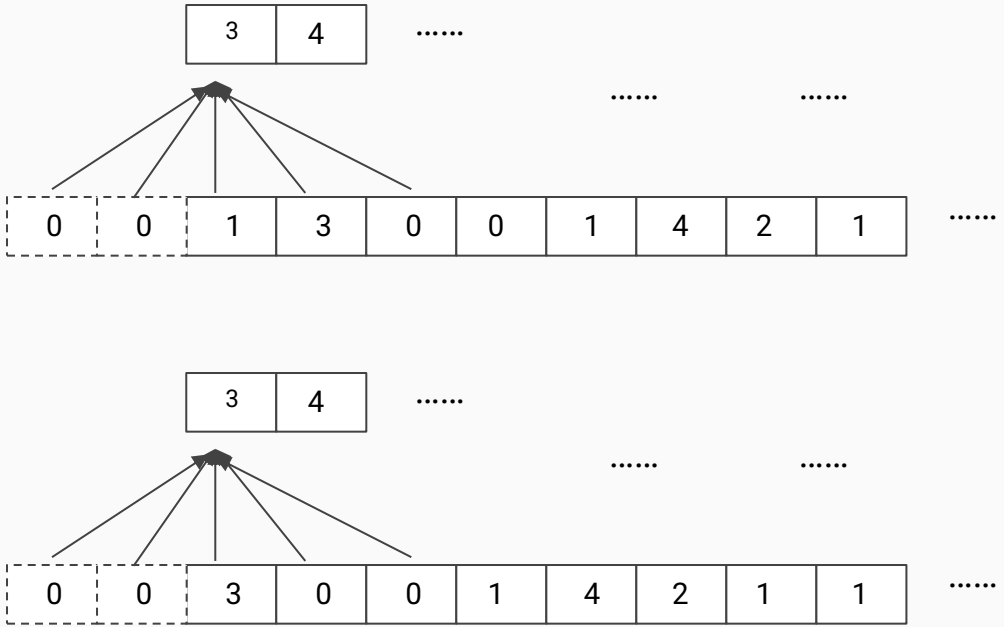
- **Max pooling**, which outputs the maximum element within the window.
- **Average pooling**, which outputs the average of the elements within the window.

Pooling

- Let's see an example

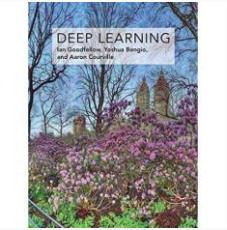


Max pooling (kernel size = 5, stride=5)

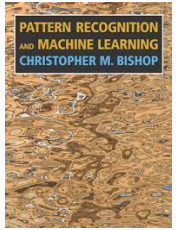


The output of max pooling is the same even if the input is shifted.

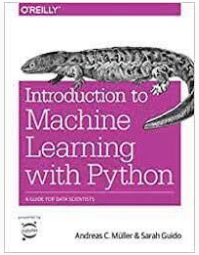
Additional Material



Chapter 2: Linear Algebra
Chapter 3: Probability and information theory
Chapter 5: Machine Learning Basics
Chapter 6: **Deep Feedforward Networks**
Chapter 9: **Convolutional Networks**



1.1.0 Example: Polynomial Curve Fitting
1.2.0 Probability Theory
3.1.0 Linear Basis Function Models
3.1.1 Maximum likelihood and least squares
4.3.2 Logistic regression
5.0 - 5.4 **Neural Networks**
5.5.6: Convolutional Networks



Introduction (page 1-27)
Linear Models (page 47-70)
Neural Network (page 106-121)

Lab Session

- During the weekly lab session, we will do:

Tutorial on Debugging Neural Networks

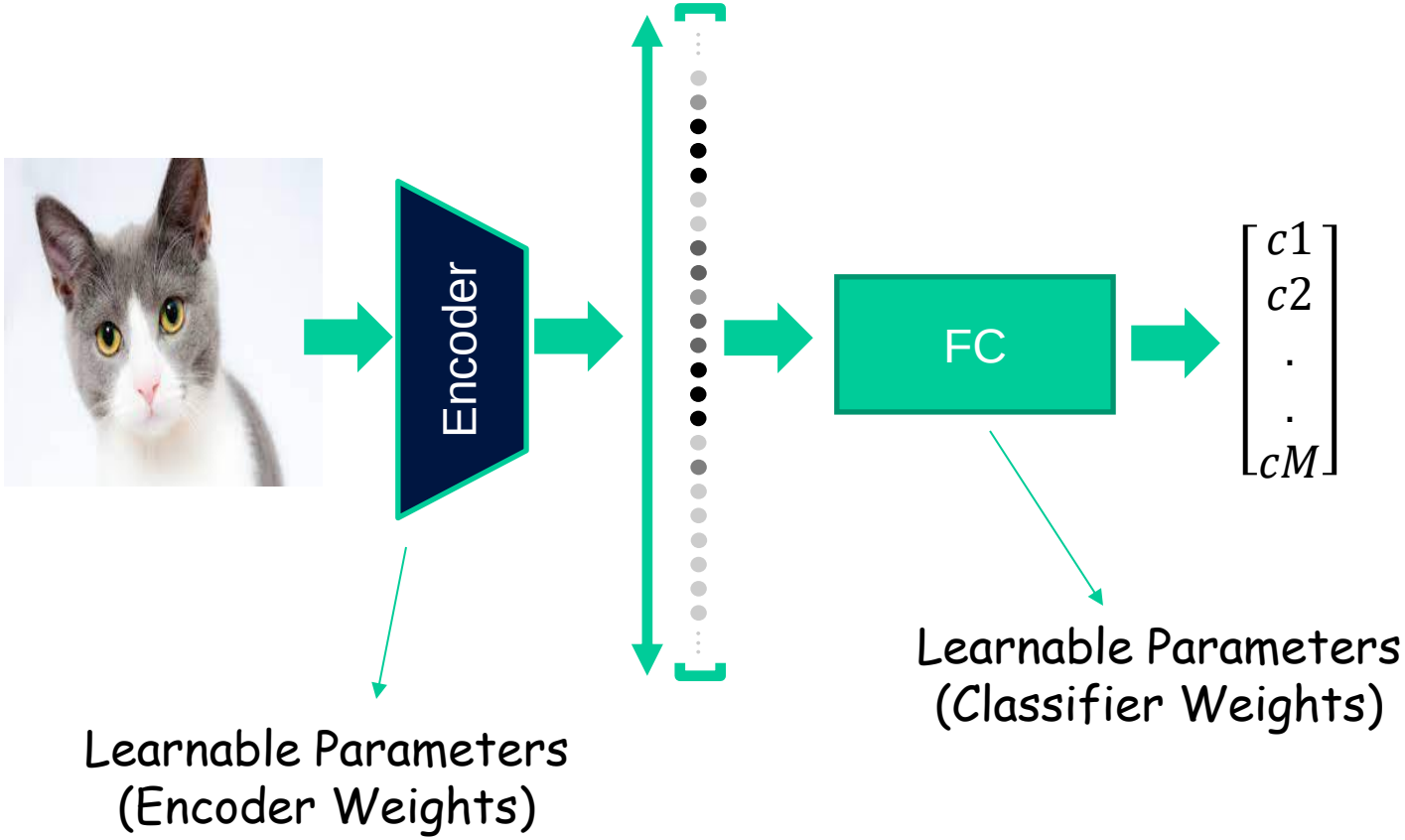
Tutorial on Convolution

Convolutional Neural Networks

Lab Assignment 4

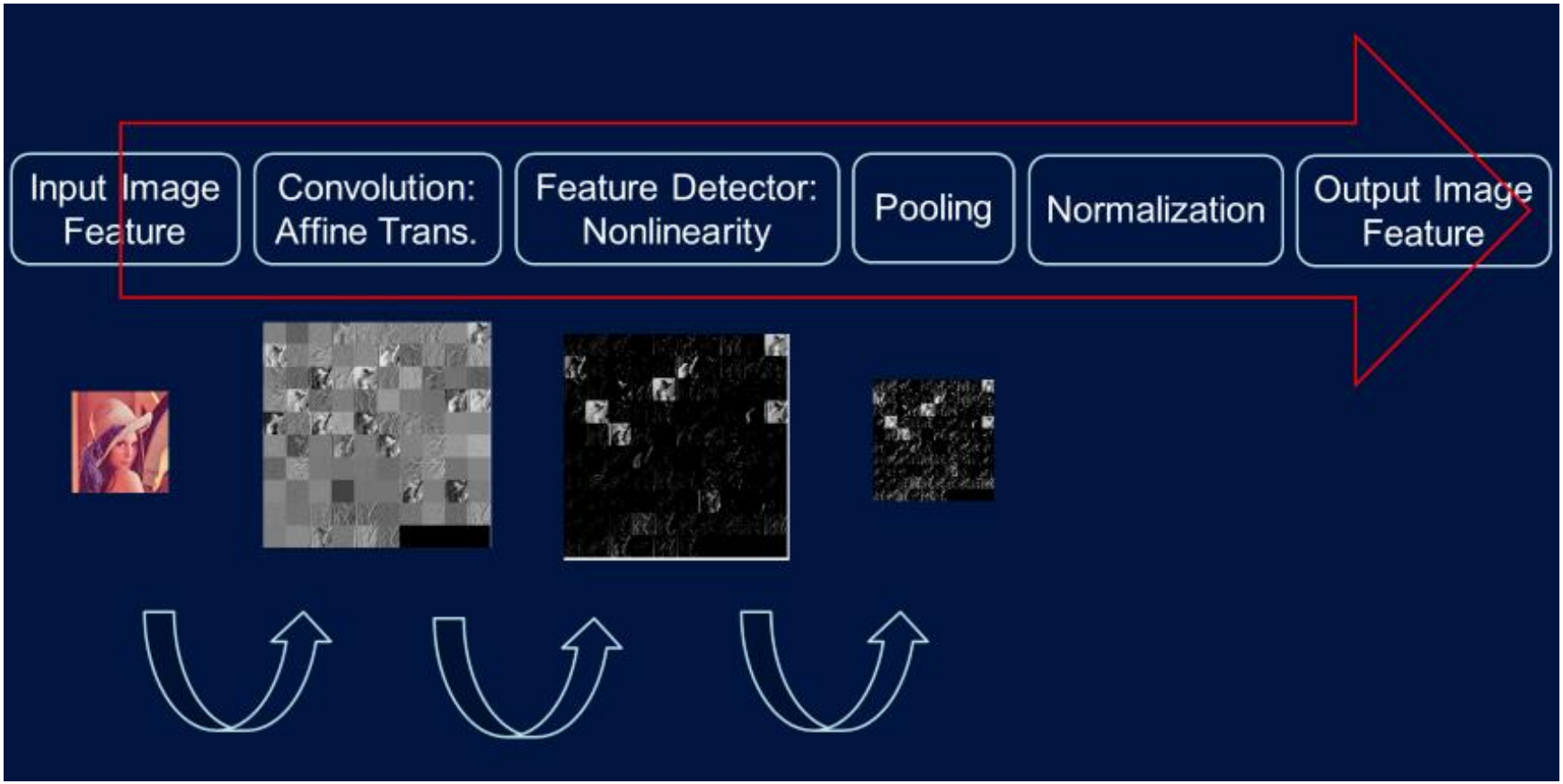
Convolutional Neural Network: Extra Slides-1

Feature (Encoder) Learning



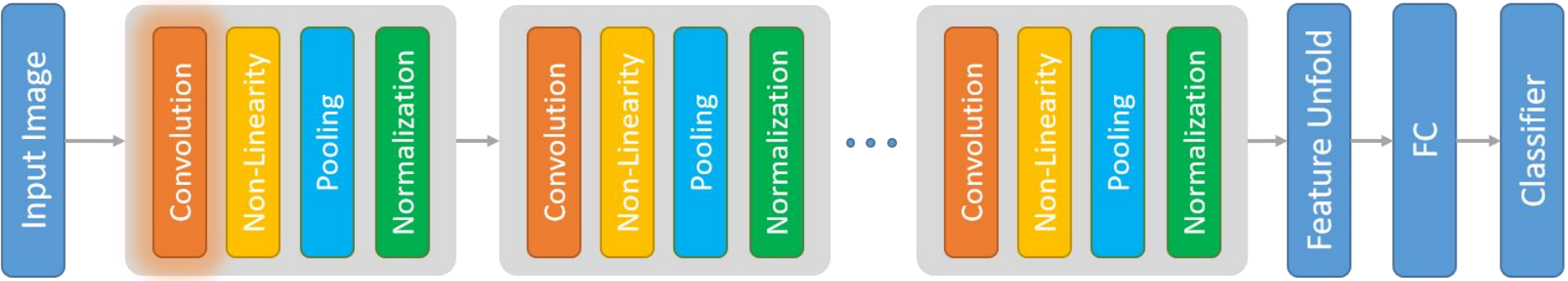
Encoder: Convolution Layer

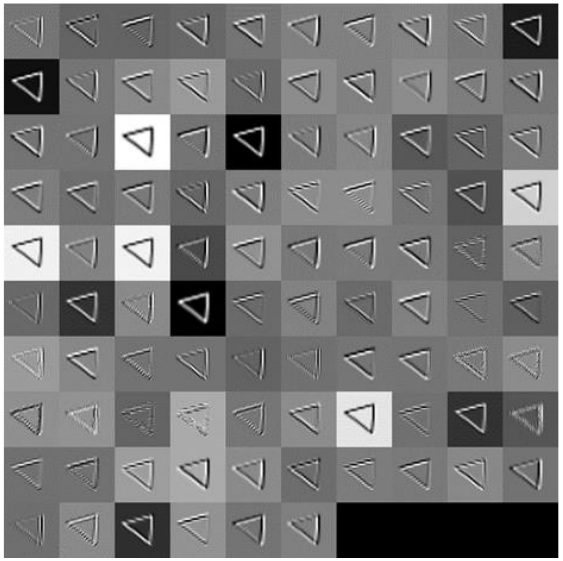
Higher level of feature abstraction



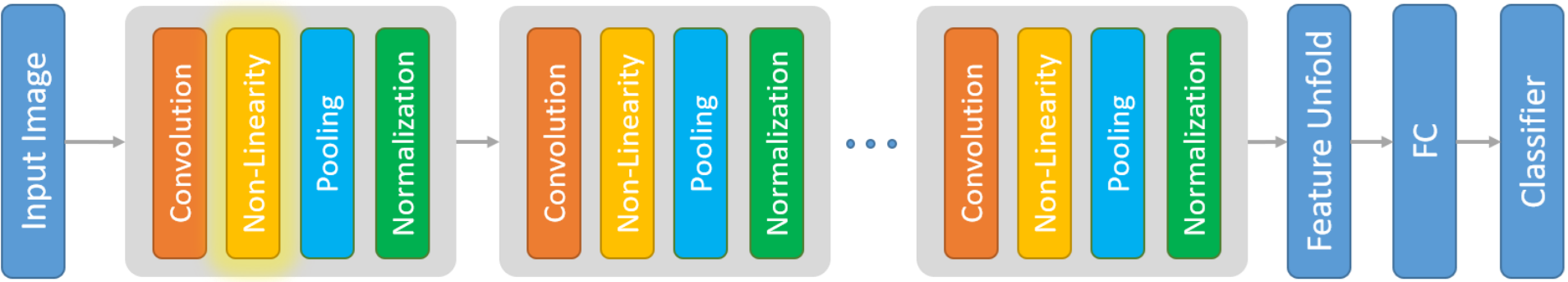
Encoder: Cascade Layer Design

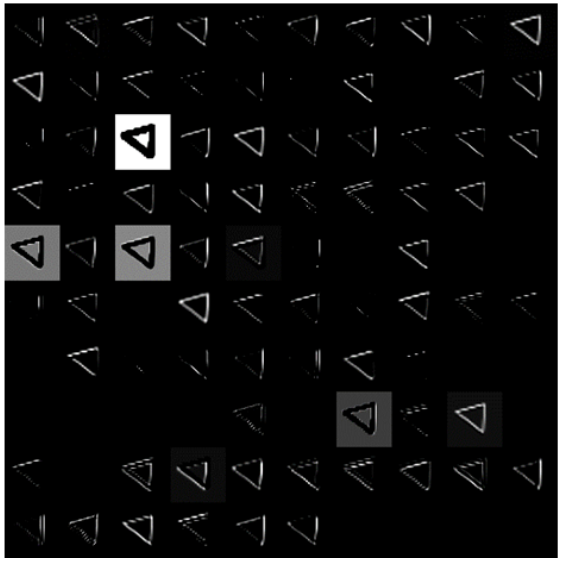
Encoder: Cascade Layer Design



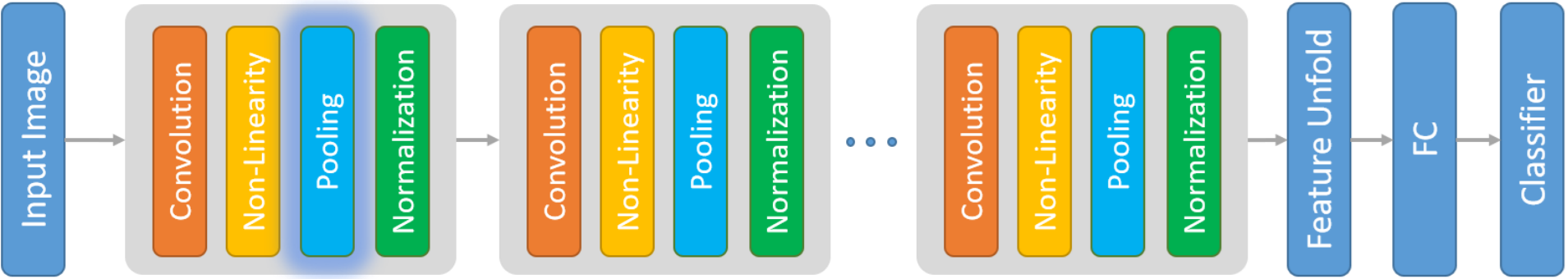


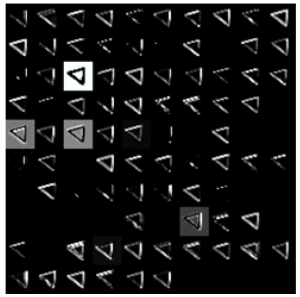
Encoder: Cascade Layer Design



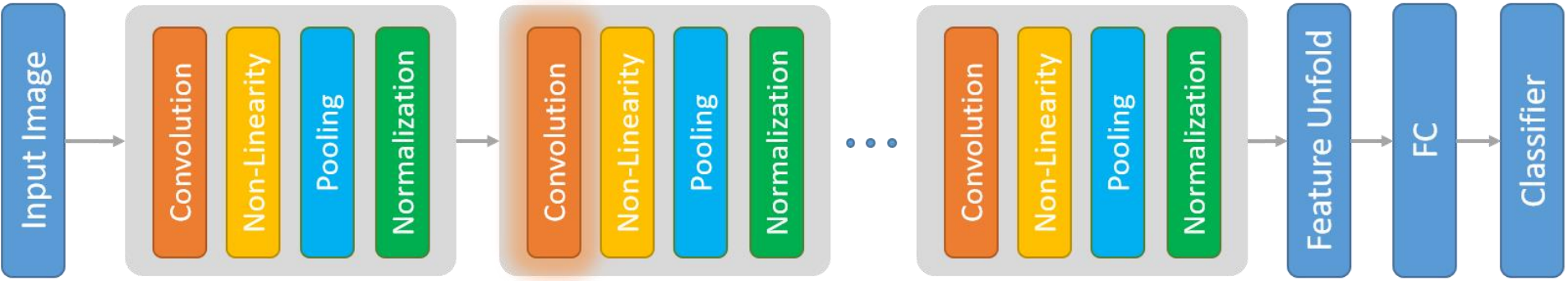


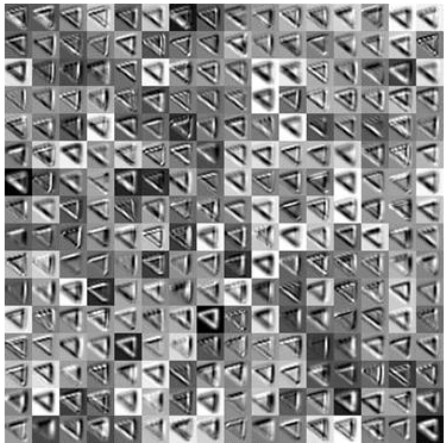
Encoder: Cascade Layer Design



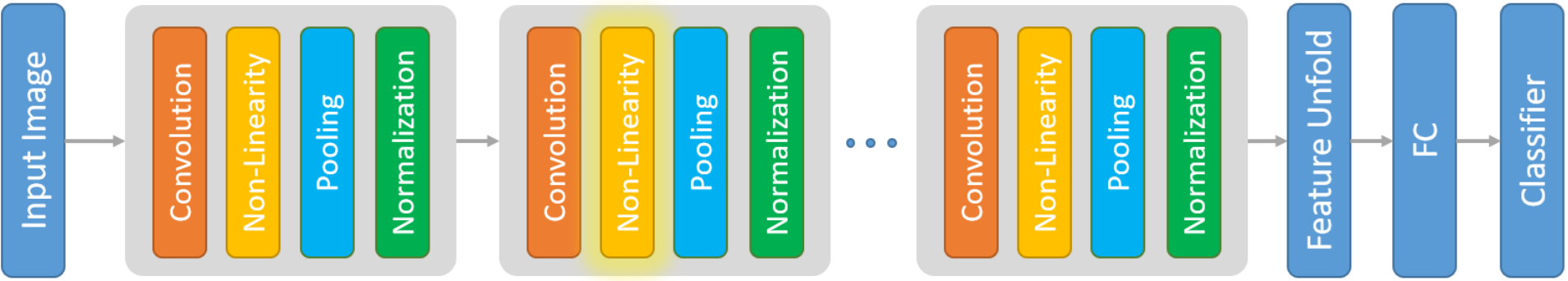


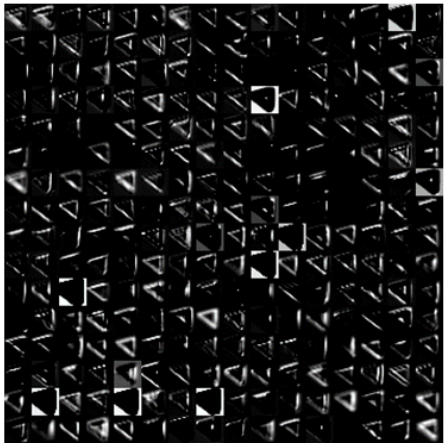
Encoder: Cascade Layer Design



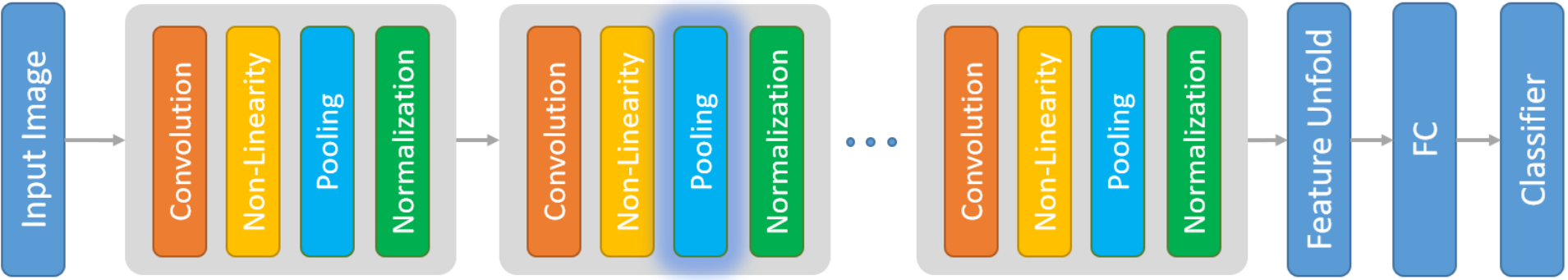


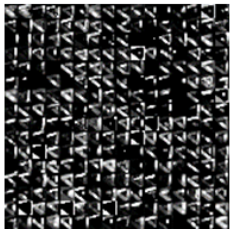
Encoder: Cascade Layer Design



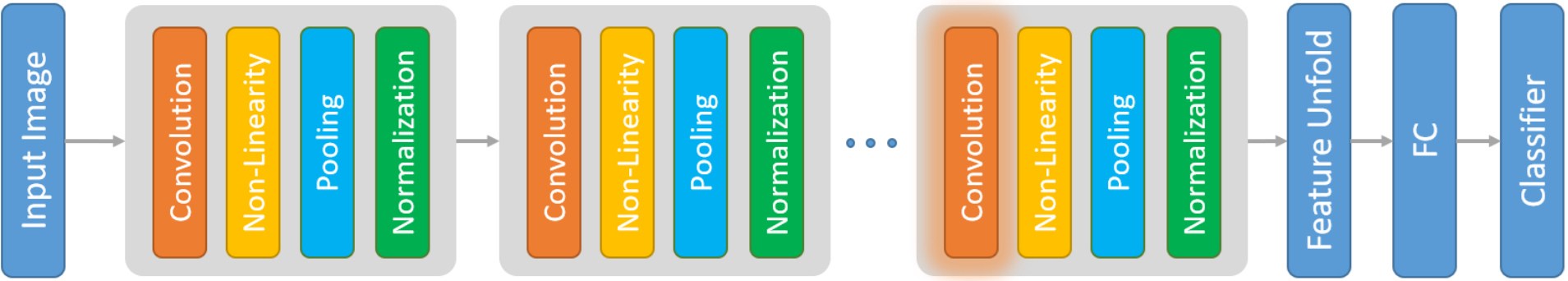


Encoder: Cascade Layer Design

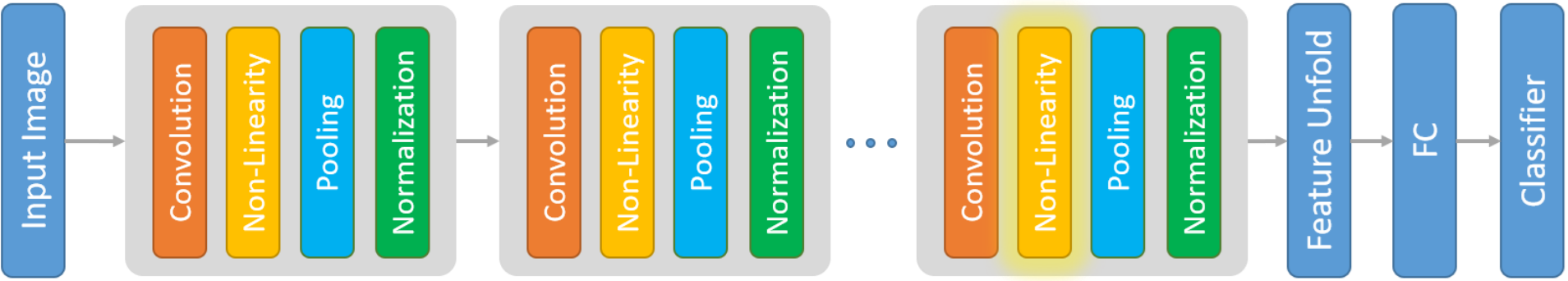


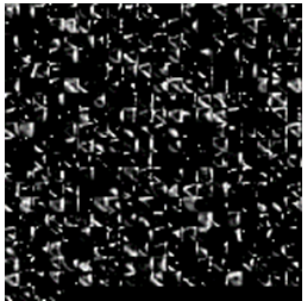


Encoder: Cascade Layer Design

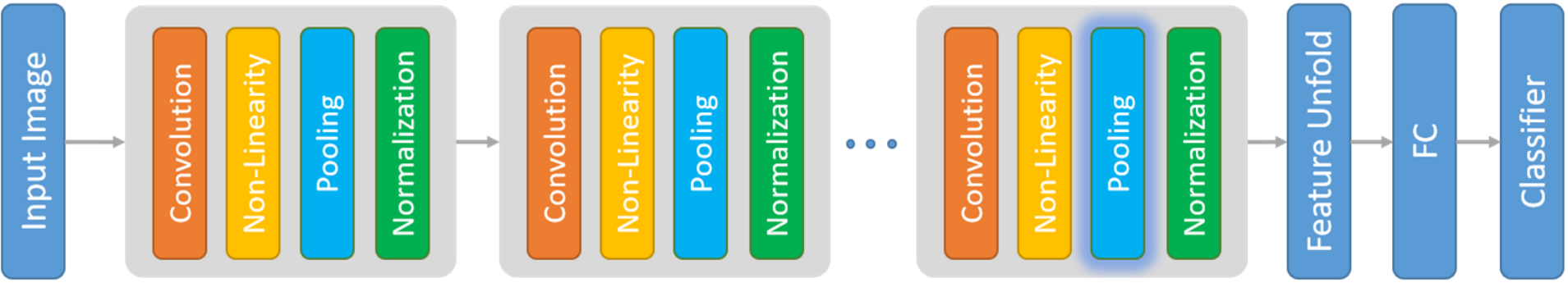


Encoder: Cascade Layer Design

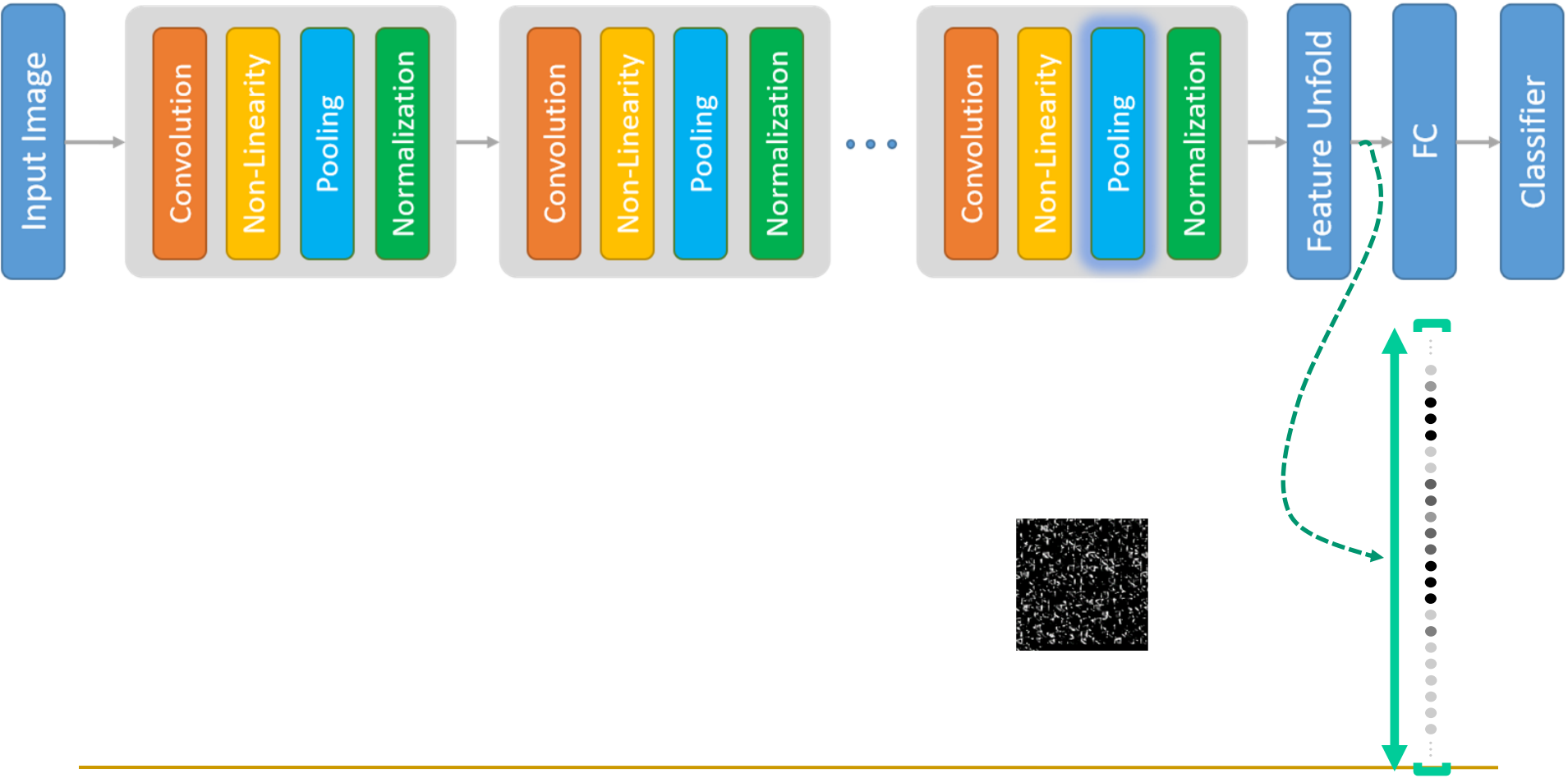




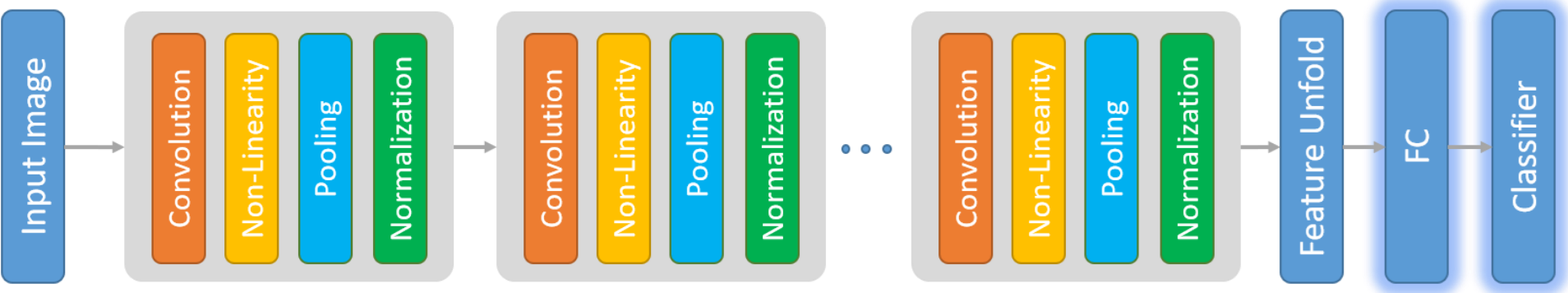
Encoder: Cascade Layer Design

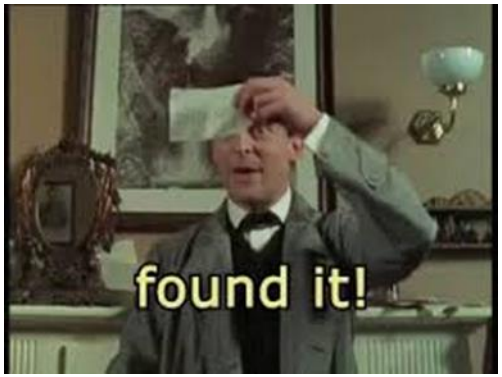


Encoder: Cascade Layer Design



Encoder: Cascade Layer Design

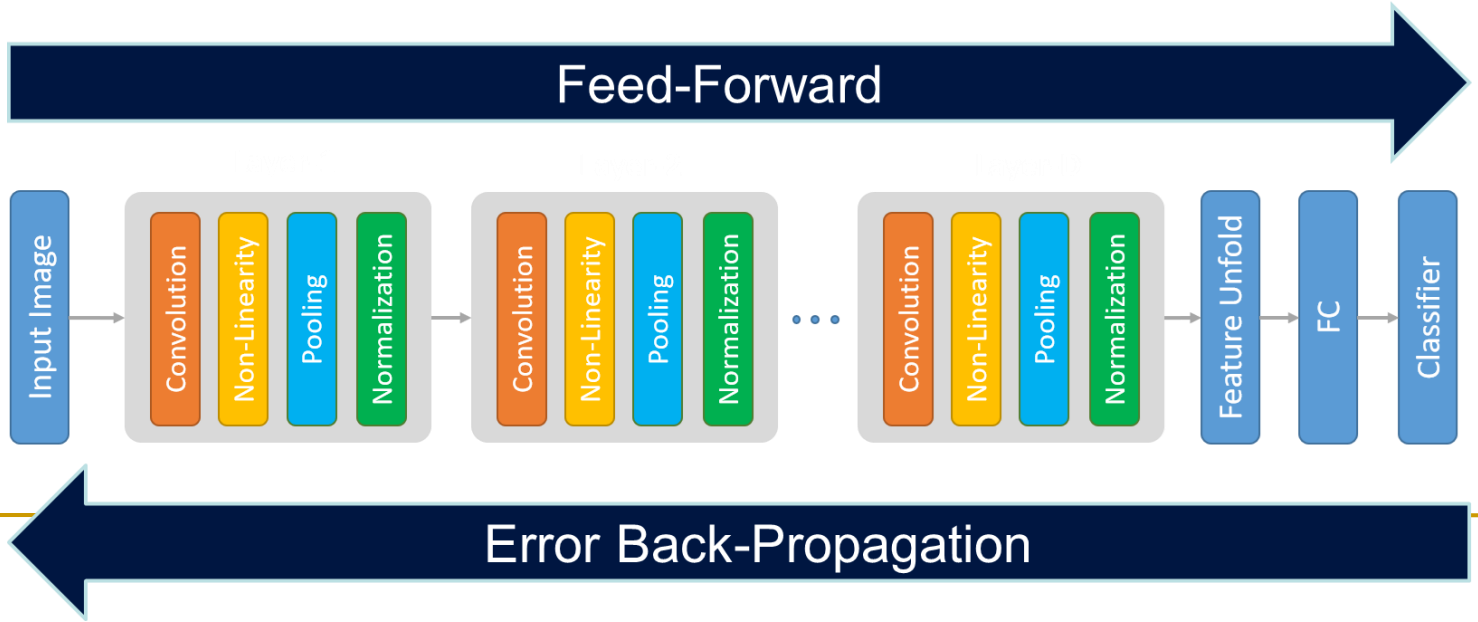




Y1: Square
Y2: Triangle
Y3: Circle
.
.
.
yN: Dimond

Encoder: Overall Training

1. Initialize all learnable parameters (e.g. random sampling)
2. Preprocess train image dataset and randomly shuffle them
3. Feed-forward images in mini-batches and compute "Target Predictions"
4. Calculate Error = Abs("Target Labels" – "Target Predictions")
5. Propagate back error gradients and adjust weights using an optimization method (e.g. stochastic gradient descent)
6. Repeat (1)-(4) to converge to a minimal error



Initial Drawbacks

1. Standard backpropagation with sigmoid activation function does not scale well with multiple layers
   - Weight of early layers change too slowly (no learning)
2. Overfitting
   - Large network -> lots of parameters -> increased capacity to "learn by heart"
3. Multilayered ANNs need lots of labeled data
   - Most data is not labeled 💻☎

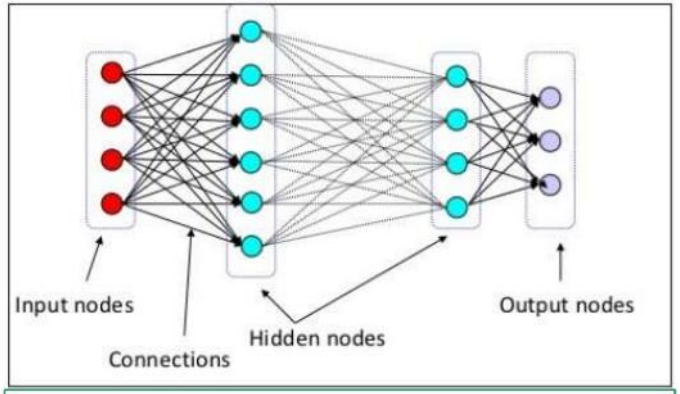
Initial Drawbacks (1)

1. Standard gradient-based backpropagation does not scale well with multiple layers…

When we multiply the gradients many times (for each layer), it can lead to …

a) Vanishing gradient problem:
   - gradients shrink exponentially with the number of layers
   - so weight updates get smaller and smaller
   - and weights of early layers change very slowly and network learns very very slowly

b) Exploding gradient problem:
   - multiplying gradients could also make them grow exponentially.
   - so weight updates get larger and larger
   - and the weights can become so large as to overflow and result in NaN values



$$\delta_h = g'(x_h) \times Err_h$$
$$= O_h\,(1 - O_h) \times \sum_k (w_{hk}\,\delta_k)$$

$$\delta_6 = O_6\,(1 - O_6) \times \sum\ (w_{6,7}\,\delta_7)$$

$$\delta_5 = O_5\,(1 - O_5) \times \sum\ (w_{5,6}\,\delta_6)$$

$$\delta_4 = O_4\,(1 - O_4) \times \sum\ (w_{4,5}\,\delta_5)$$

$$\delta_3 = O_3\,(1 - O_3) \times \sum\ (w_{3,4}\,\delta_4)$$

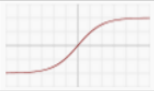
…

Initial Drawbacks (1)

To help, we can :

a) Use other activation functions…

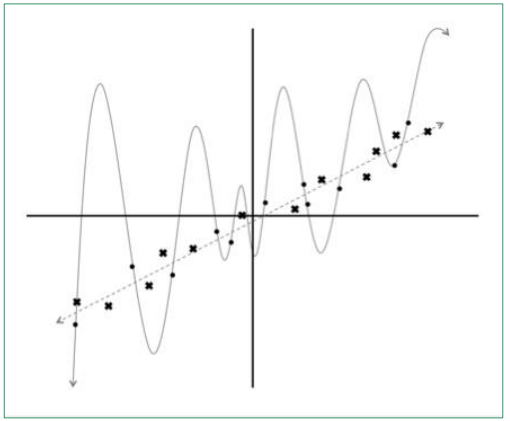
a) Do "gradient clipping" (i.e. set bounds on the gradients)

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

https://medium.com/towards-data-science/activation-functions-and-its-types-which-is-better-a9a5310cc8f

https://medium.com/towards-data-science/activation-functions-neural-networks-1cbd9f8d91d6

Initial Drawbacks (2)

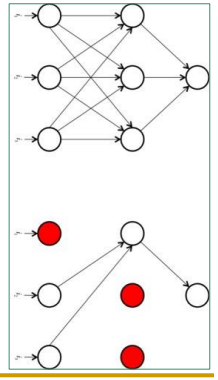


2. Overfitting
   - Large network -> lots of parameters -> increased capacity to "learn by heart"
   - *Solutions:*
     - *Regularization:*
       - modify the error function that we minimize to penalize large weights.

$$\frac{1}{2}\sum_{i=0}^{n}\frac{(T_i - O_i)^2}{n} + \lambda f(w)$$

       - where *f(w)* grows larger as the weights grow larger and λ is the regularization strength

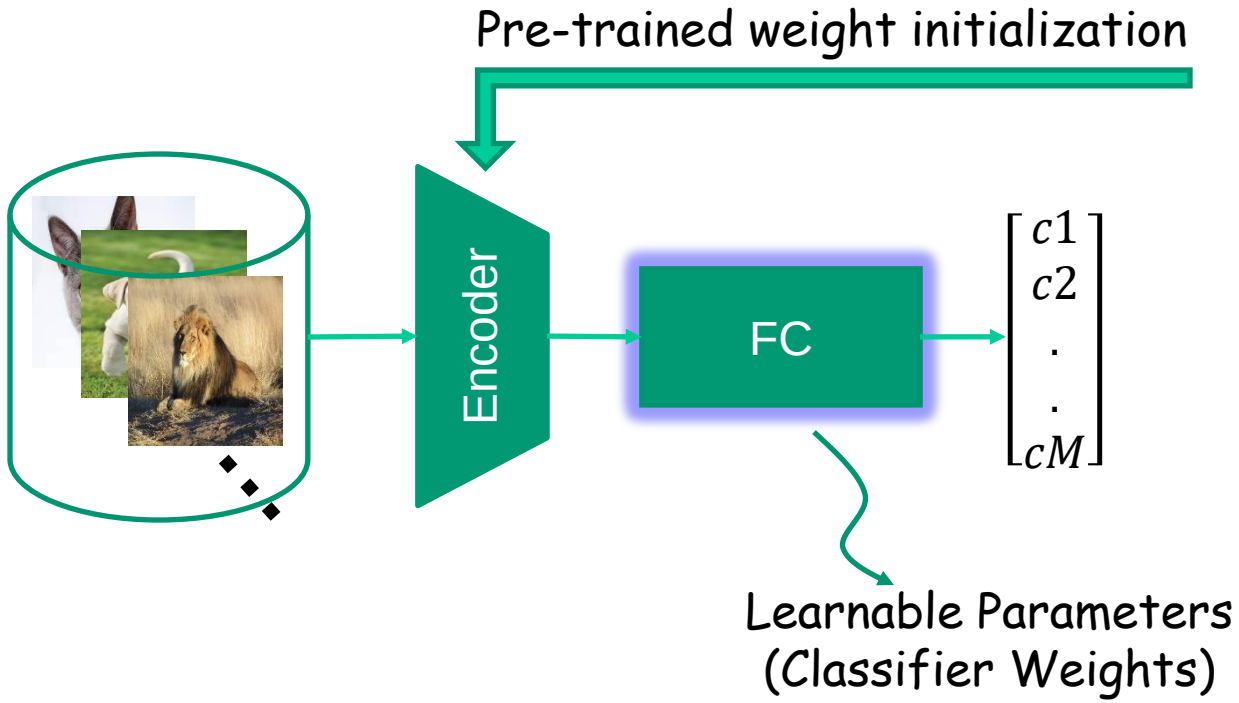
     - *Dropout:*
       - keep a neuron active with some probability *p* or setting it to zero otherwise.
       - prevents the network from becoming too dependent on any one neuron.



https://www.kdnuggets.com/2015/04/preventing-overfitting-neural-networks.html

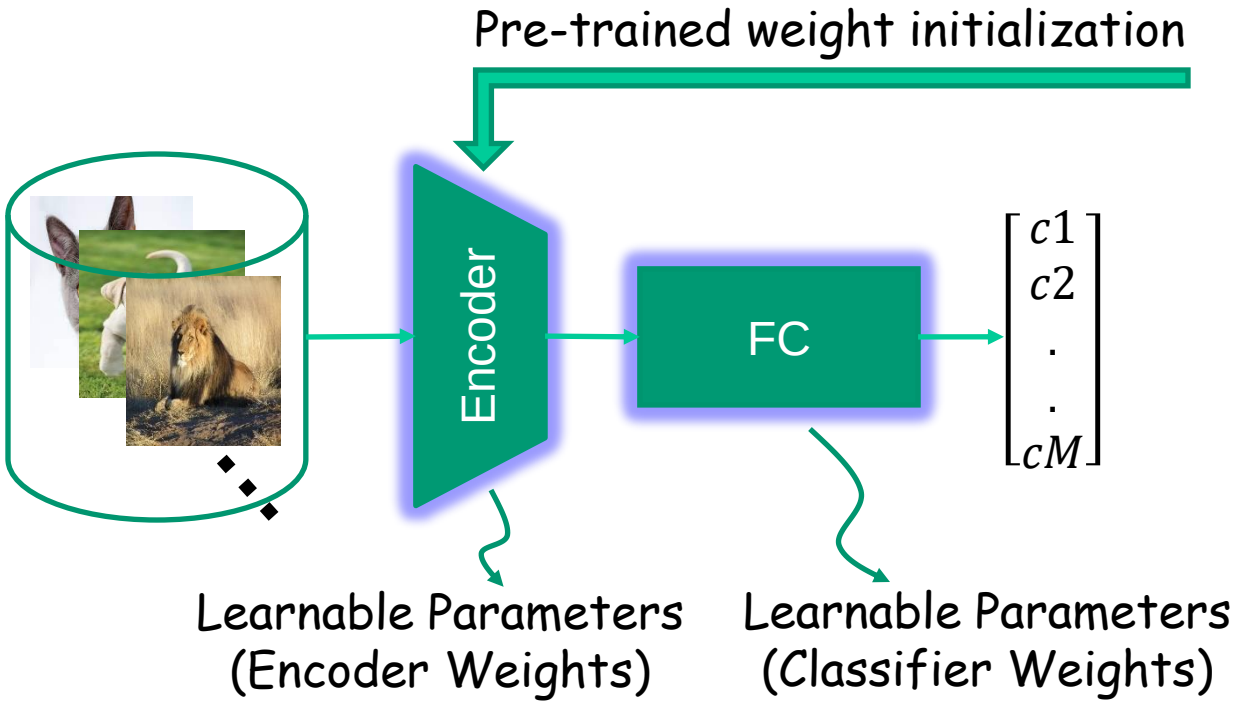
Initial Drawbacks (3)

3. Multilayered ANNs need lots of labeled data for training. To solve the problem:
   - **Transfer Learning:** use pre-trained encoder weights
     - Fine-Tuning



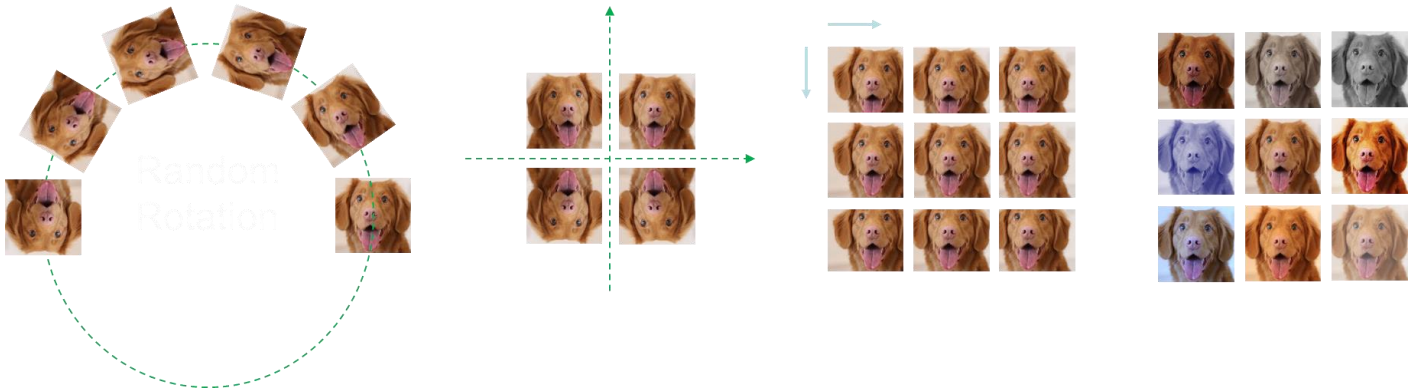
Initial Drawbacks (3)

3. Multilayered ANNs need lots of labeled data for training. To solve the problem:
   - **Transfer Learning:** use pre-trained encoder weights
     - Deep-Tuning

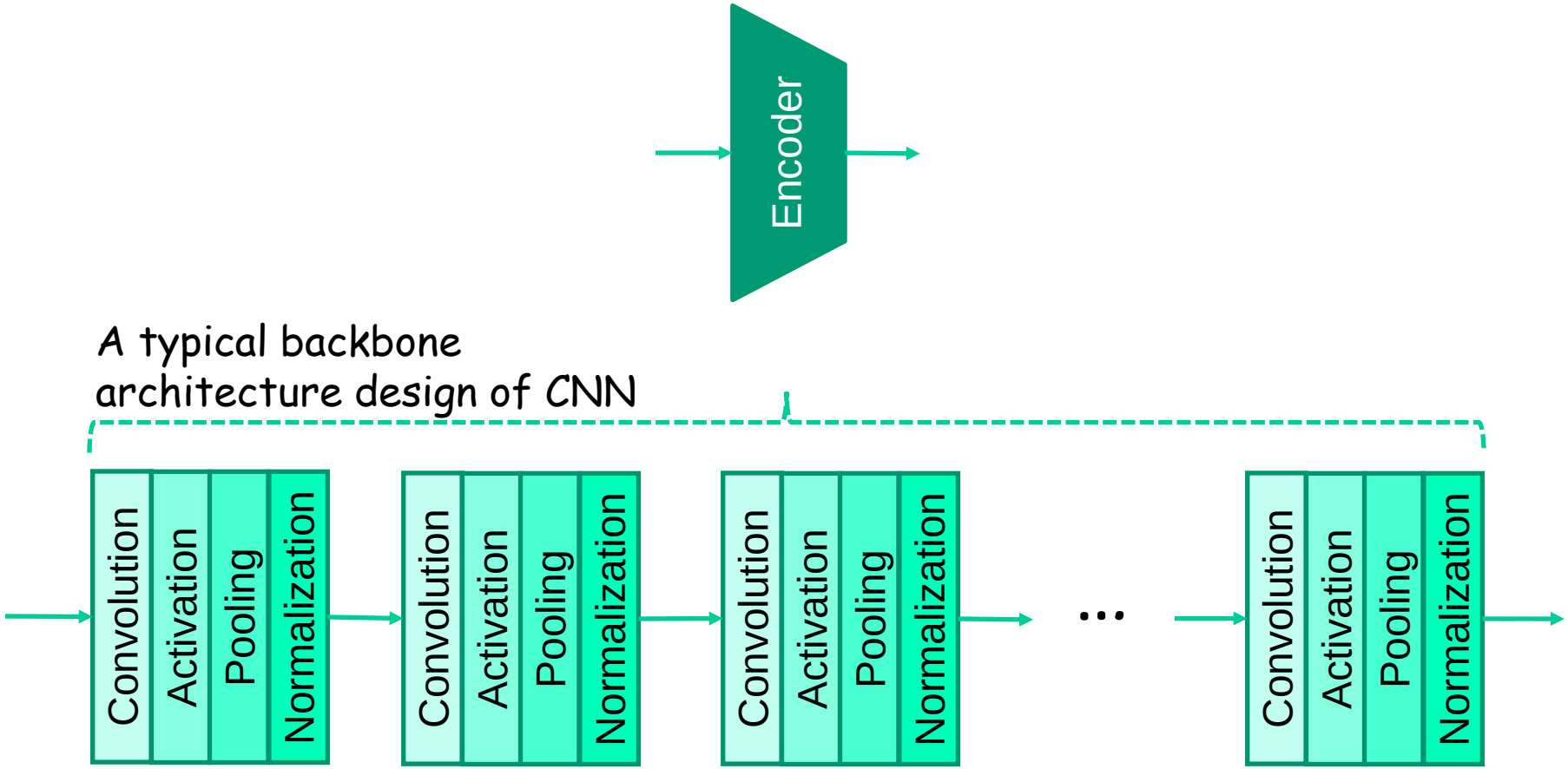


Initial Drawbacks (3)

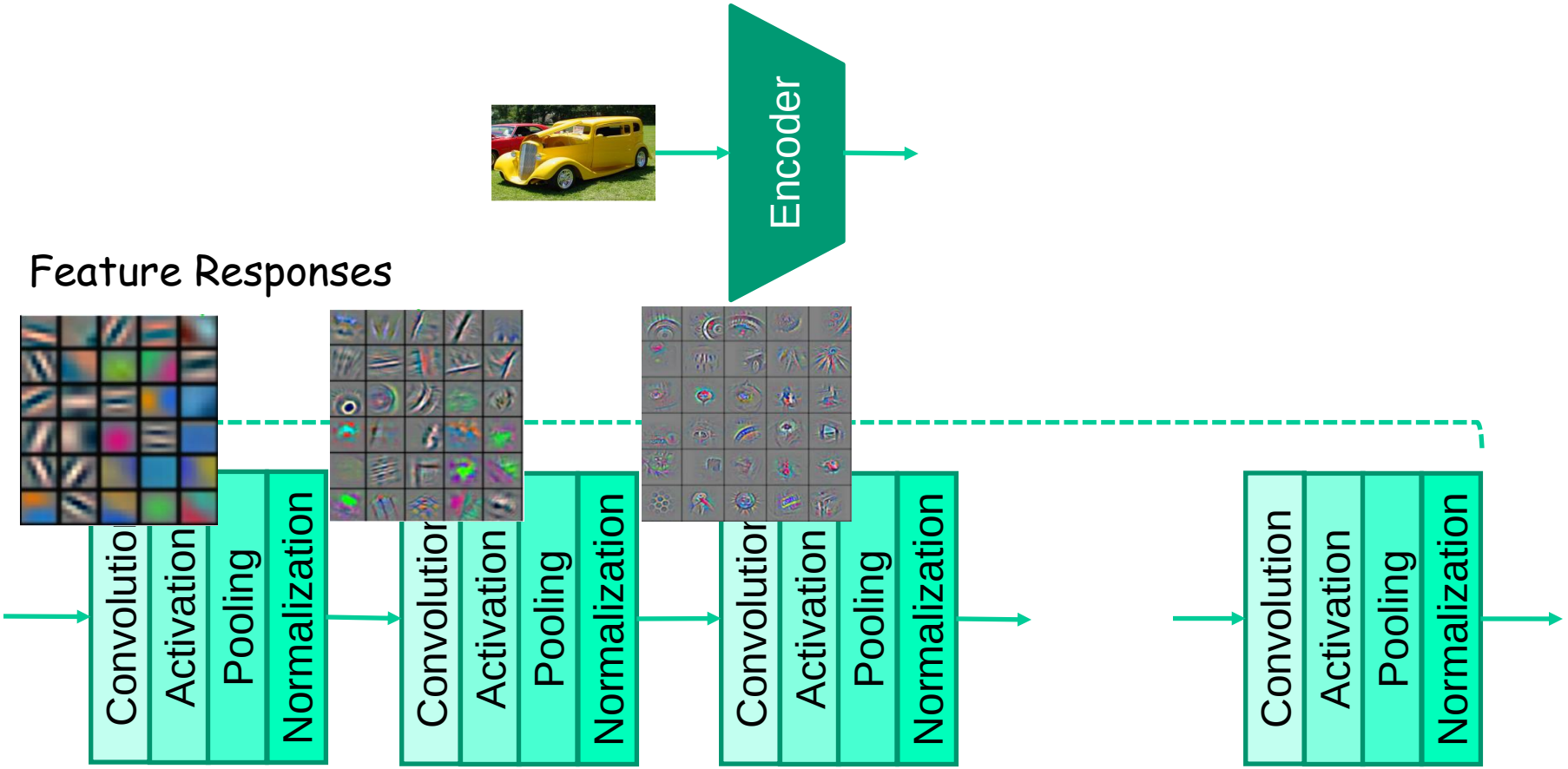
3. Multilayered ANNs need lots of labeled data for training. To solve the problem:
   - **Image Augmentation:** randomly perturb representation
     - Shifting
     - Flipping
     - Rotation
     - Skewing
     - Color/illumination perturbation



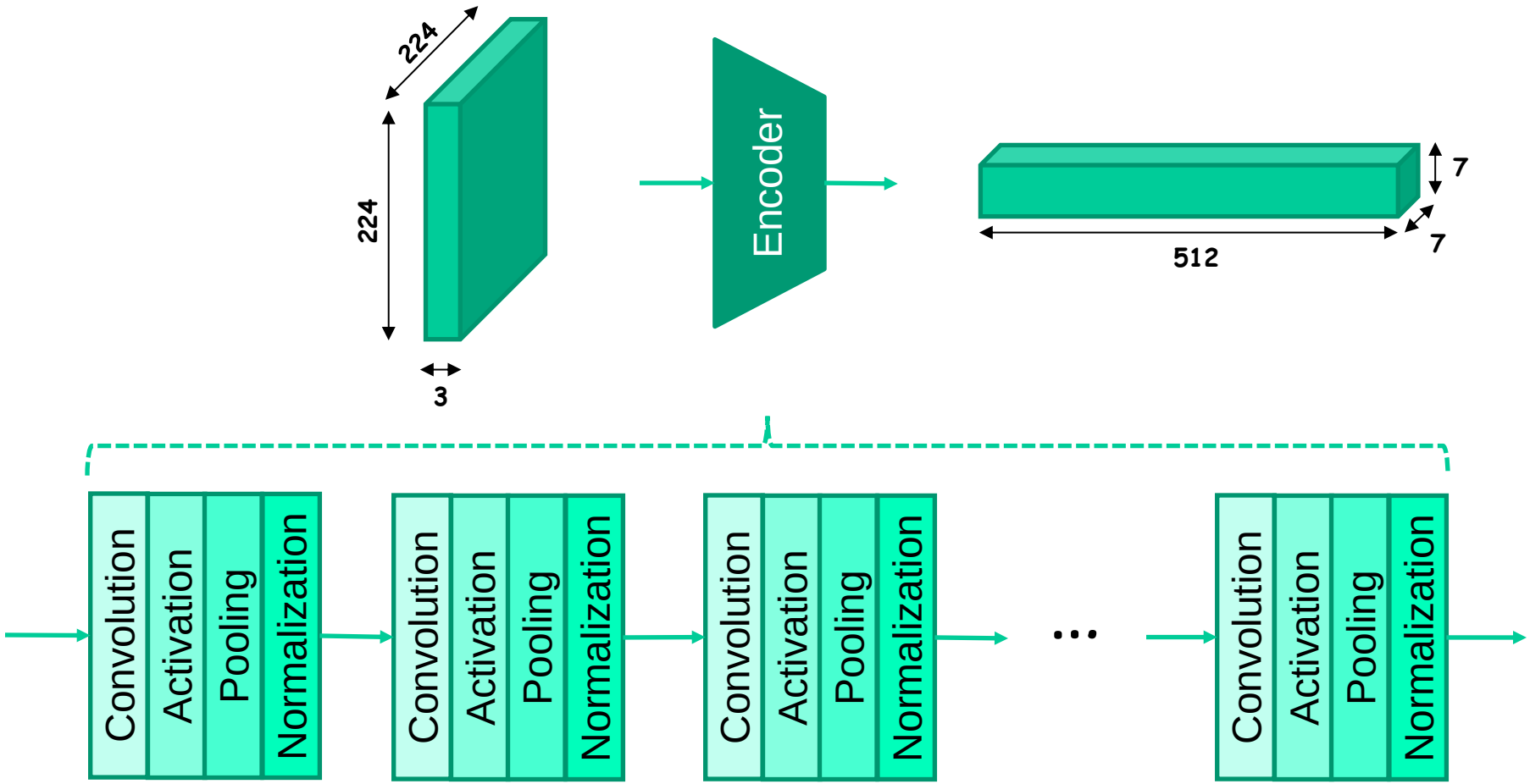
Building Block Design of CNN



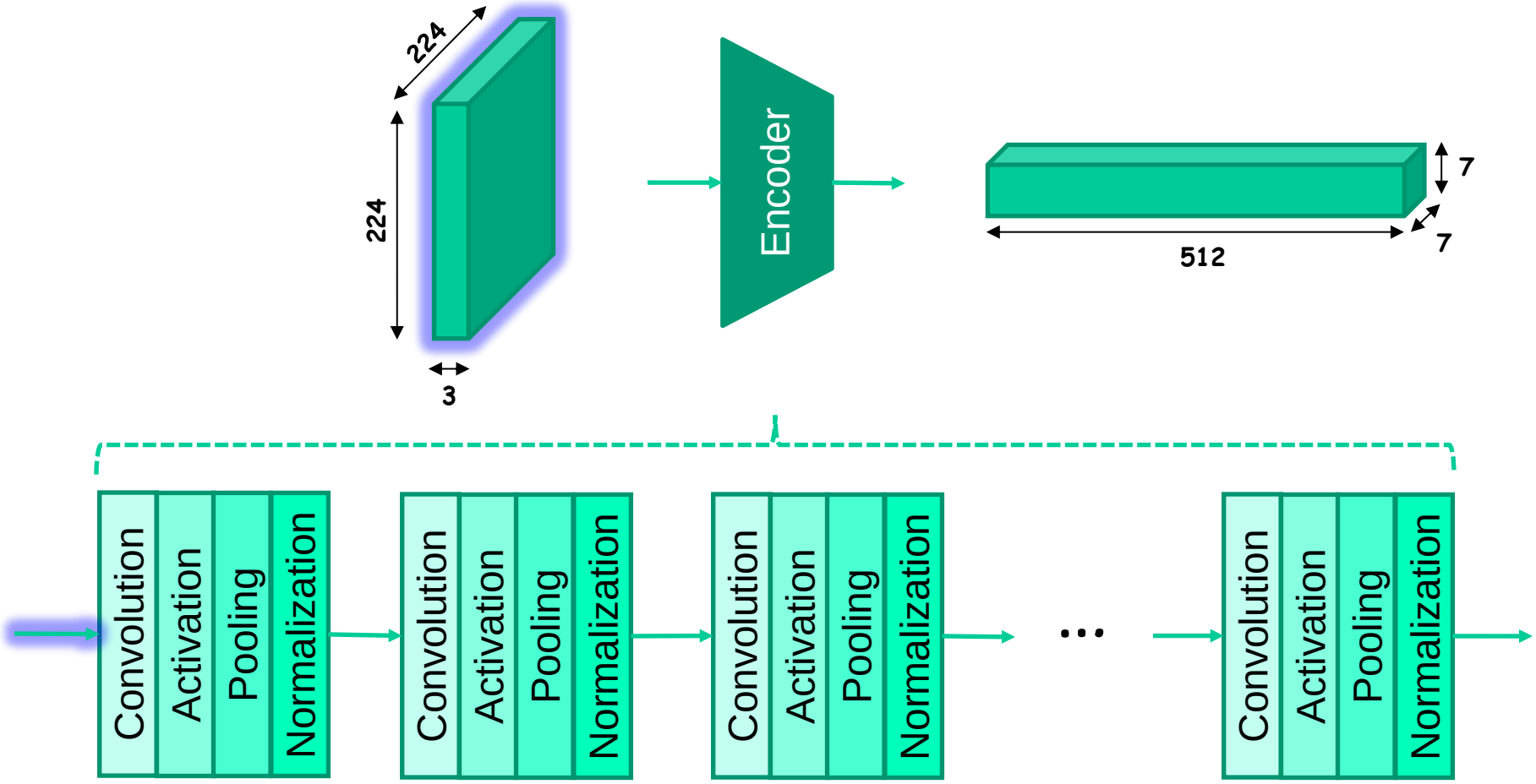
Building Block Design of CNN



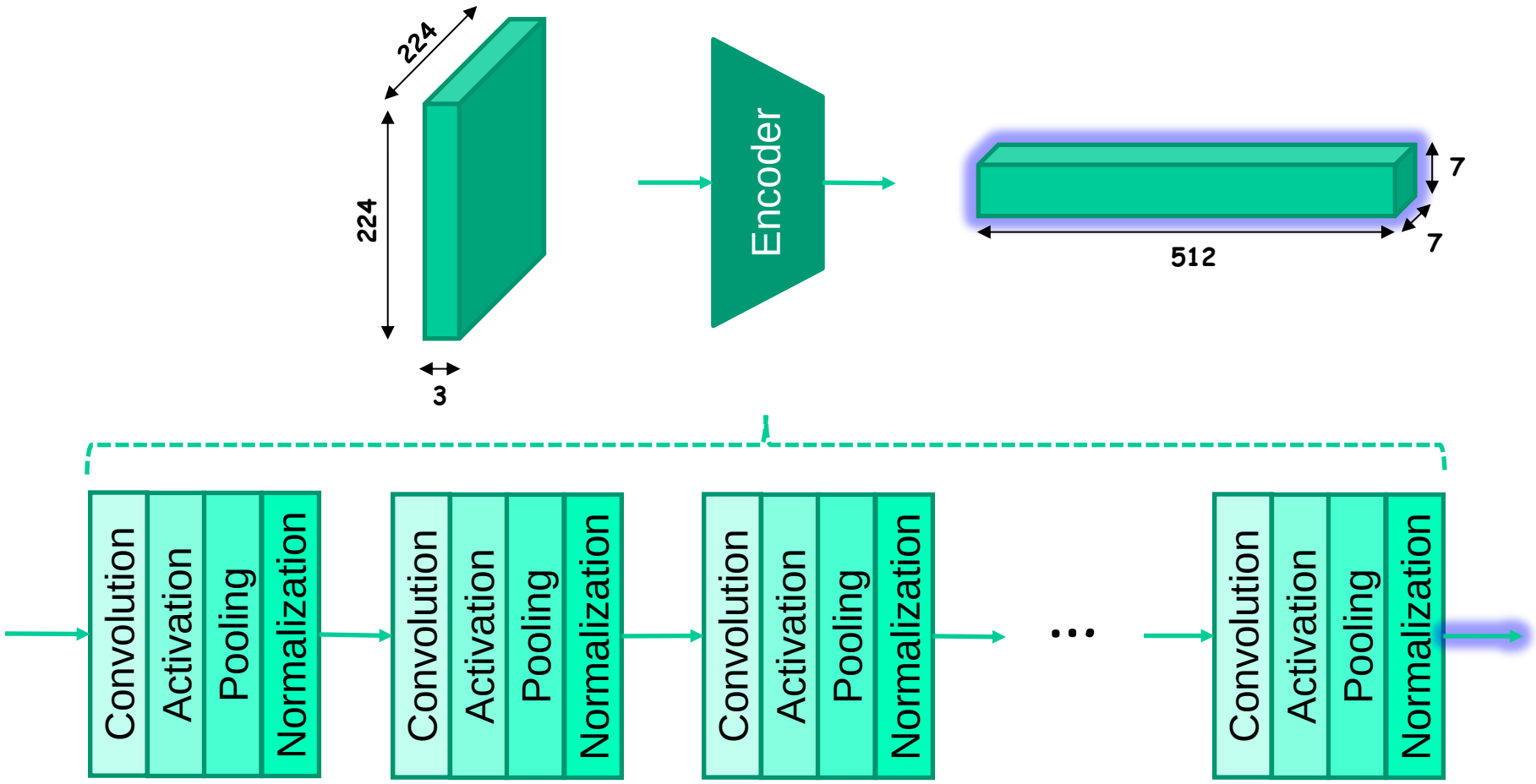
Building Block Design of CNN



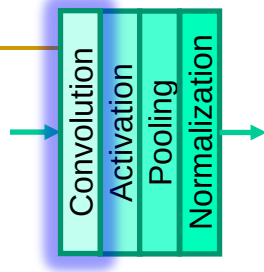
Building Block Design of CNN



Building Block Design of CNN

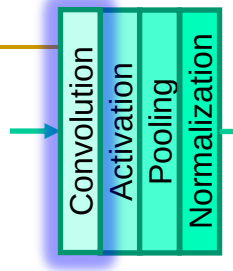


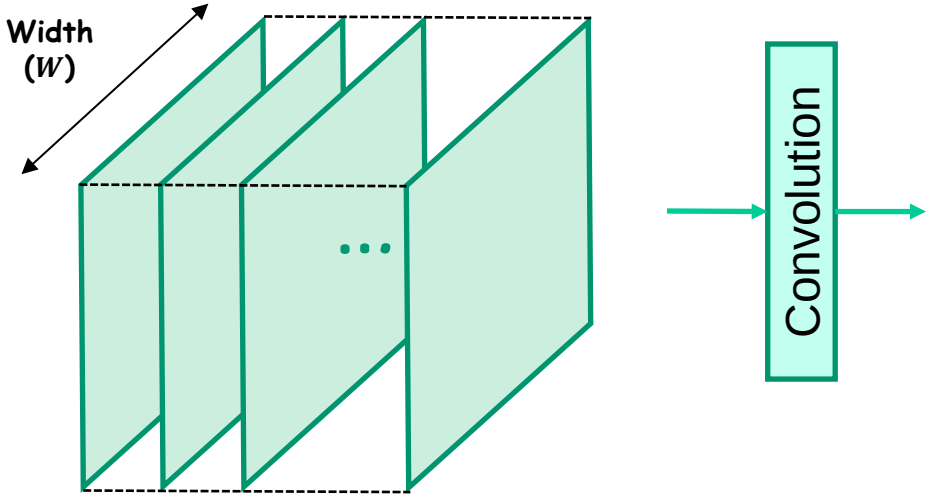
Conv Layer of CNN



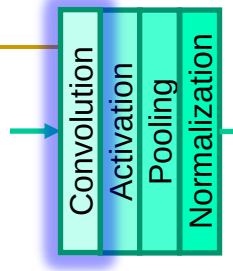
https://cs231n.github.io/convolutional-networks/

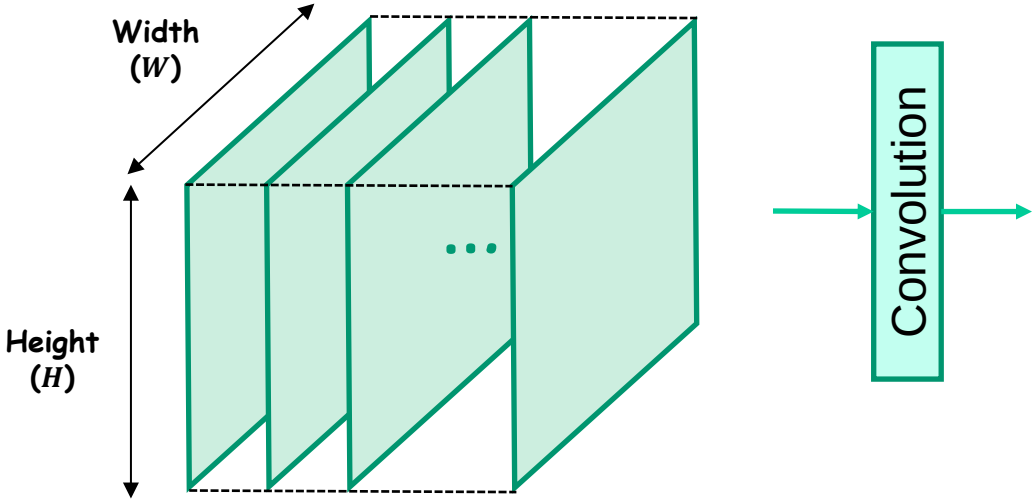
Convolutional Feature Mapping



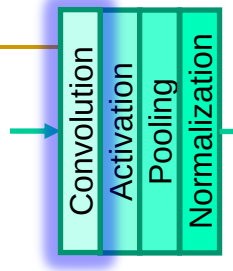


Convolutional Feature Mapping

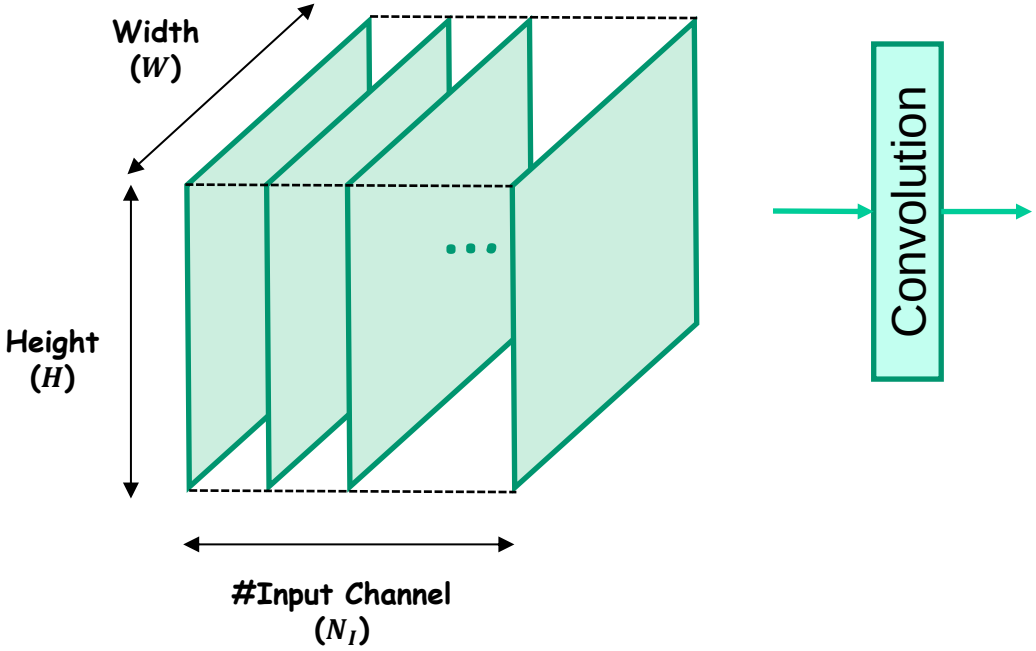




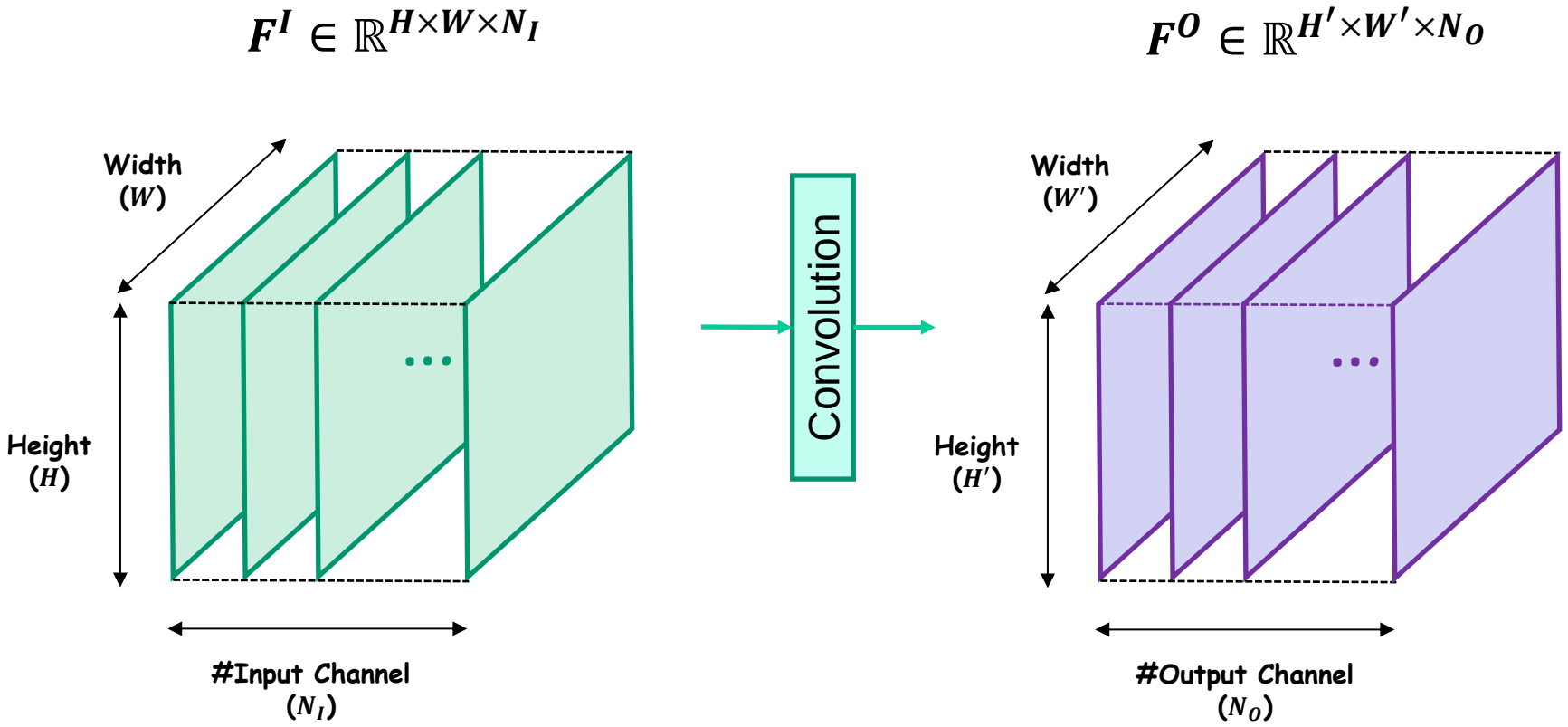
Convolutional Feature Mapping



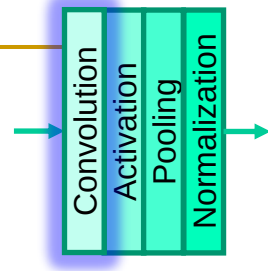
$$F^I \in \mathbb{R}^{H \times W \times N_I}$$

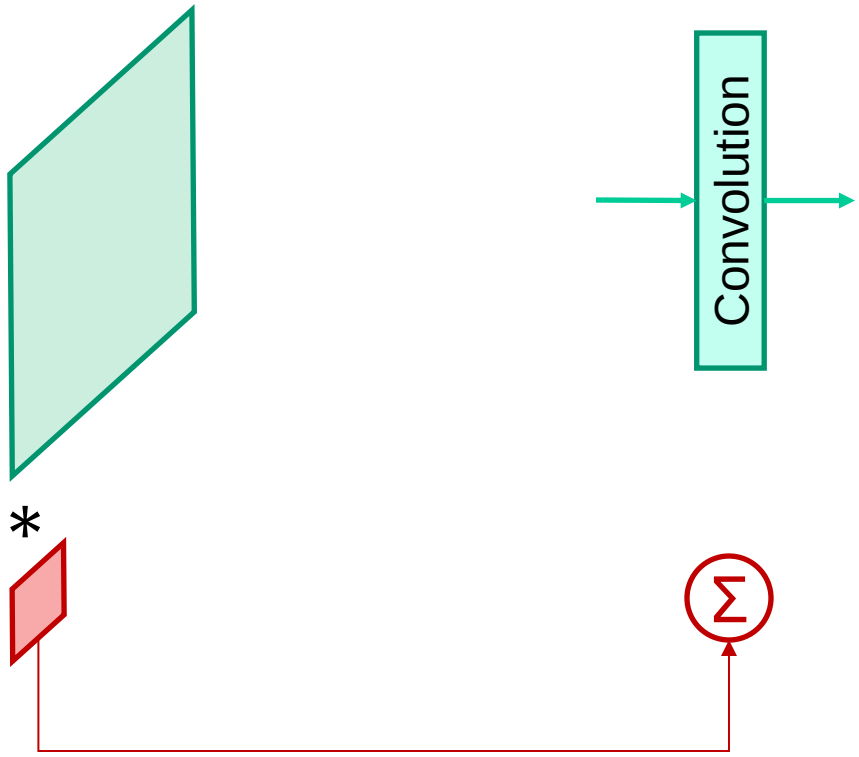


Convolutional Feature Mapping

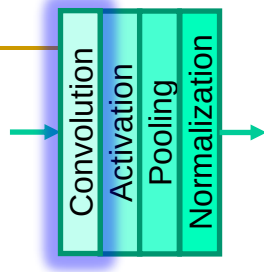


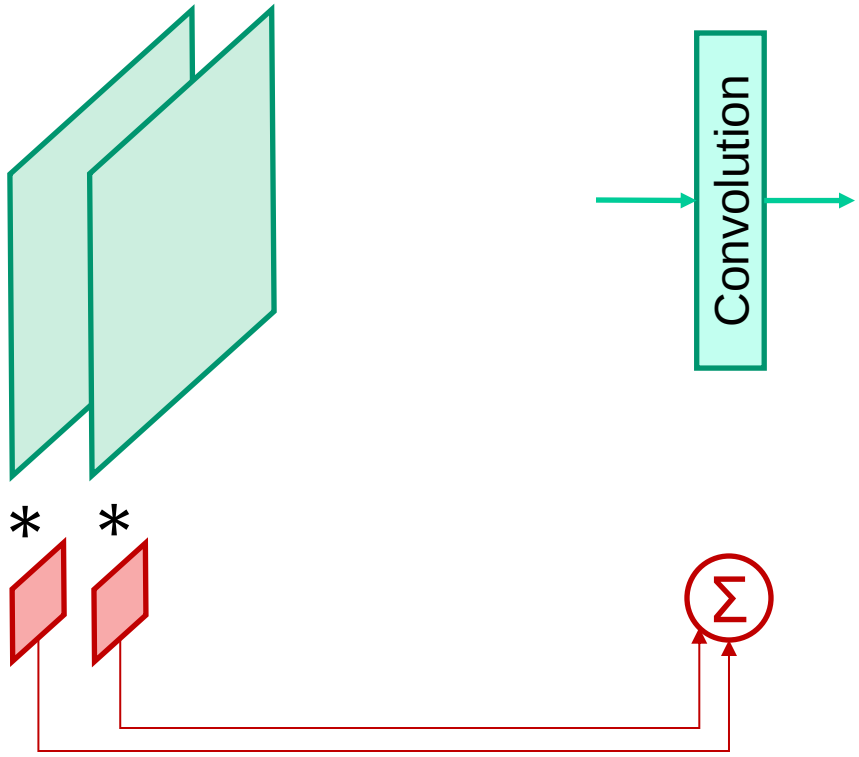
Convolutional Feature Mapping



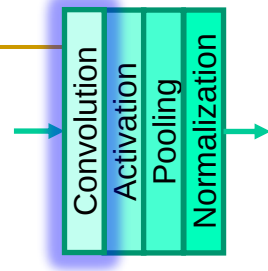


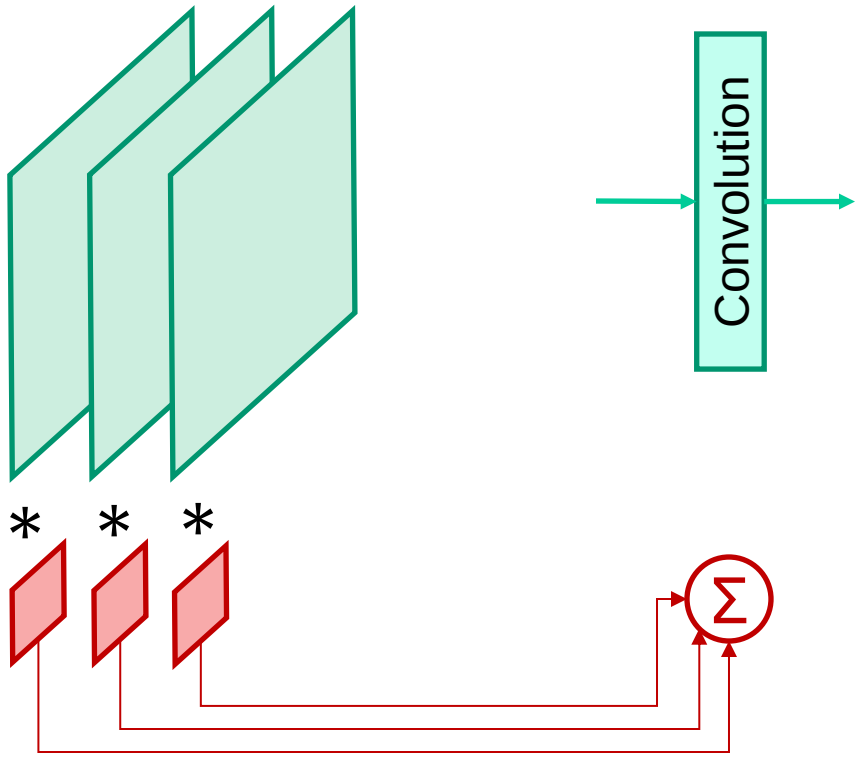
Convolutional Feature Mapping



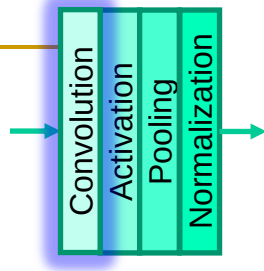


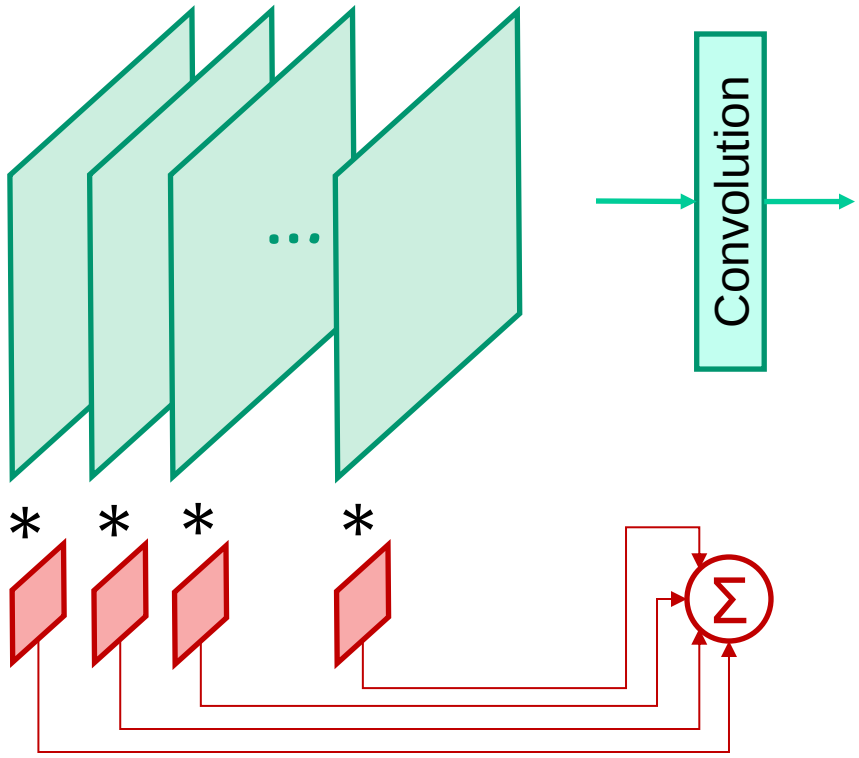
Convolutional Feature Mapping



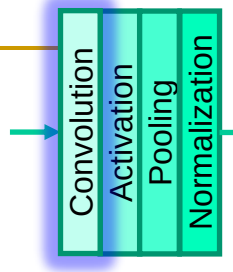


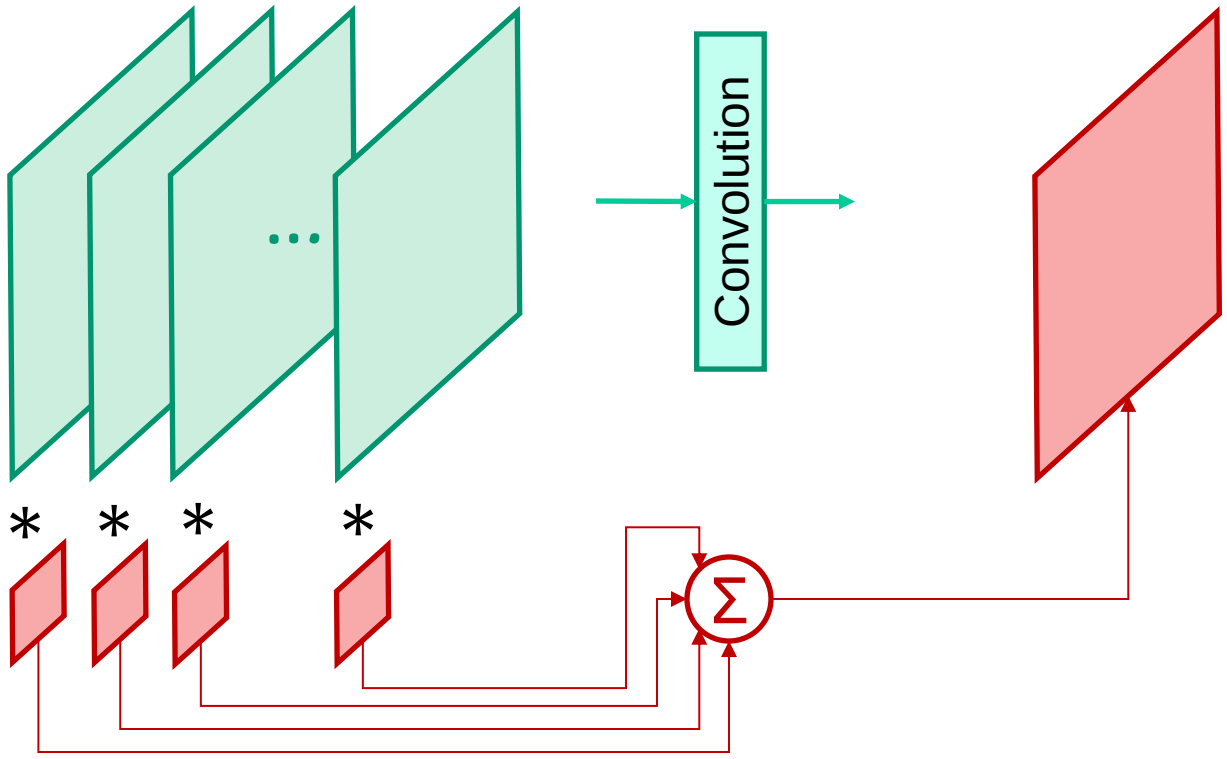
Convolutional Feature Mapping



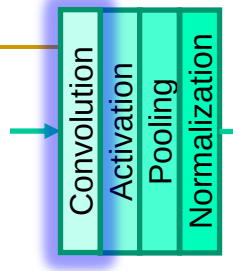


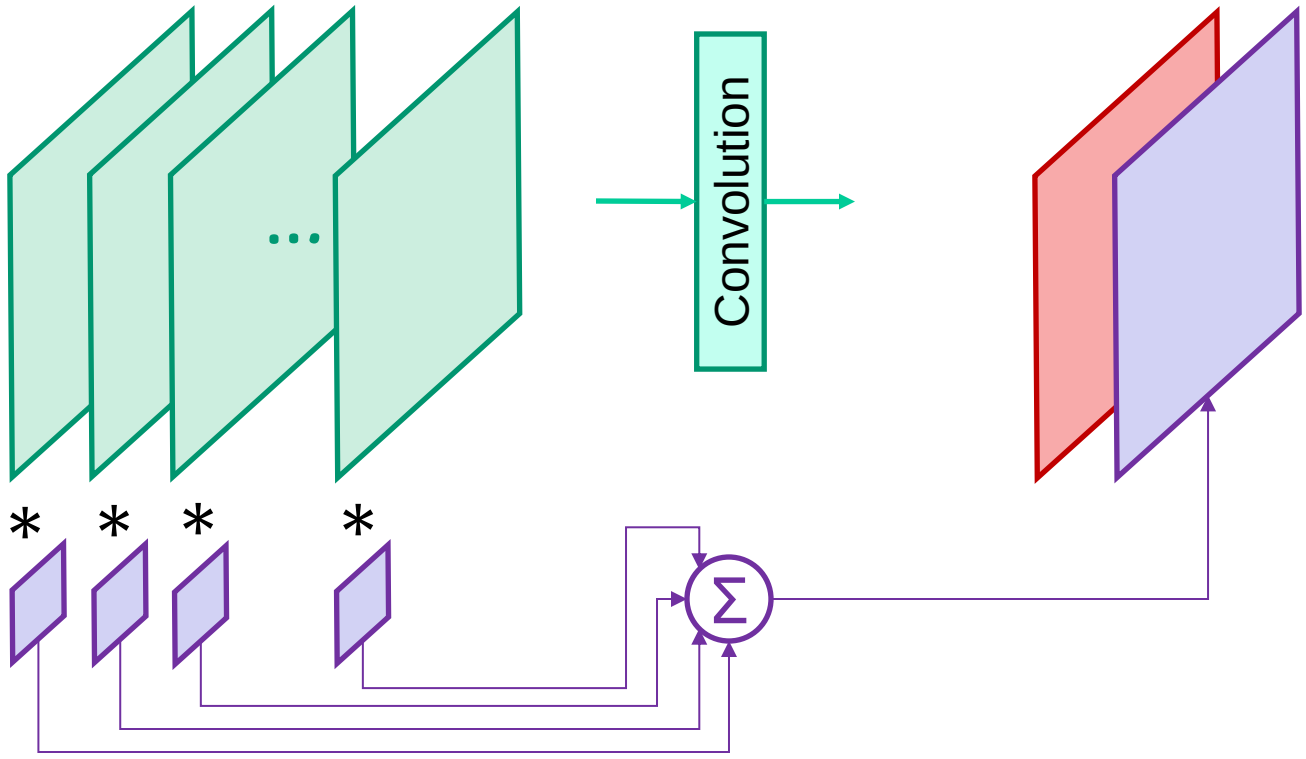
Convolutional Feature Mapping



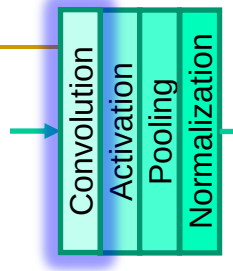


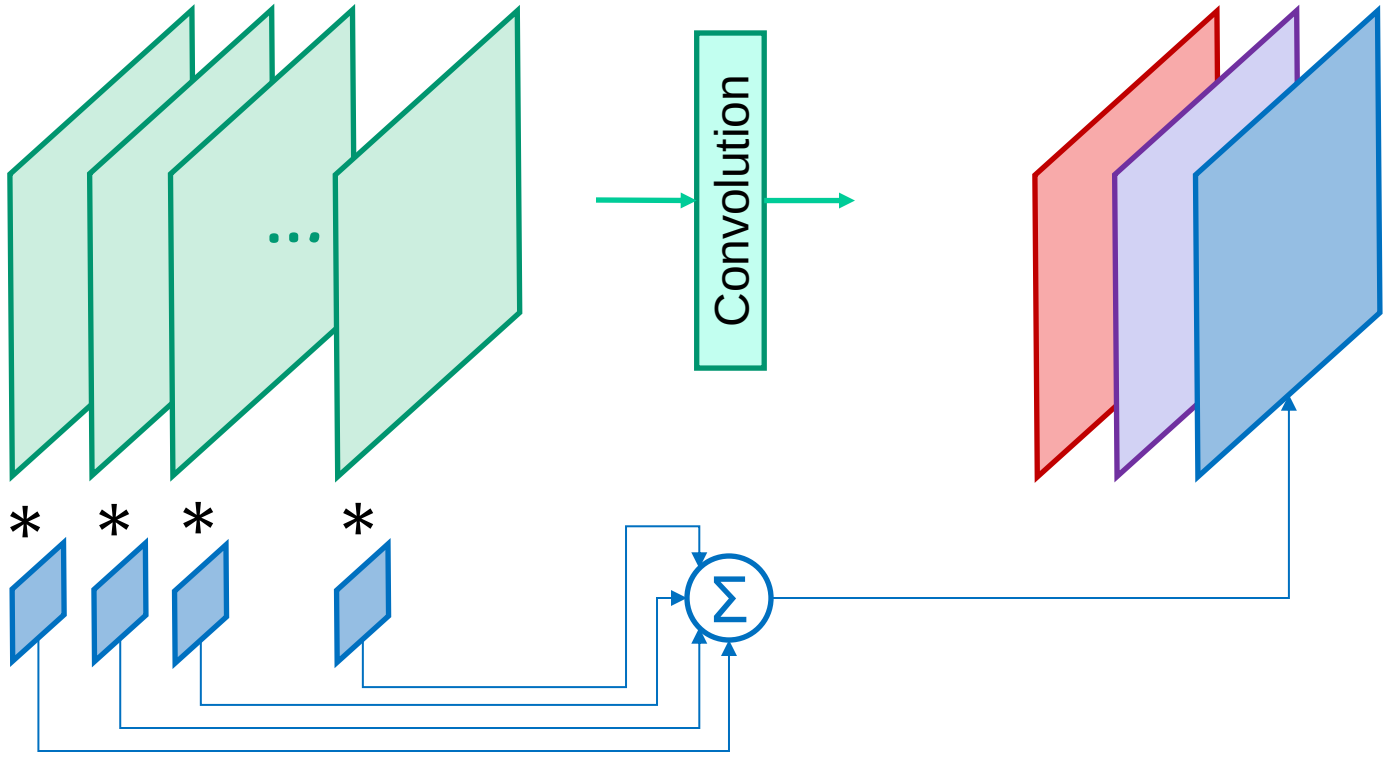
Convolutional Feature Mapping



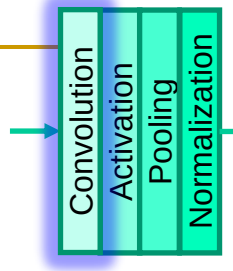


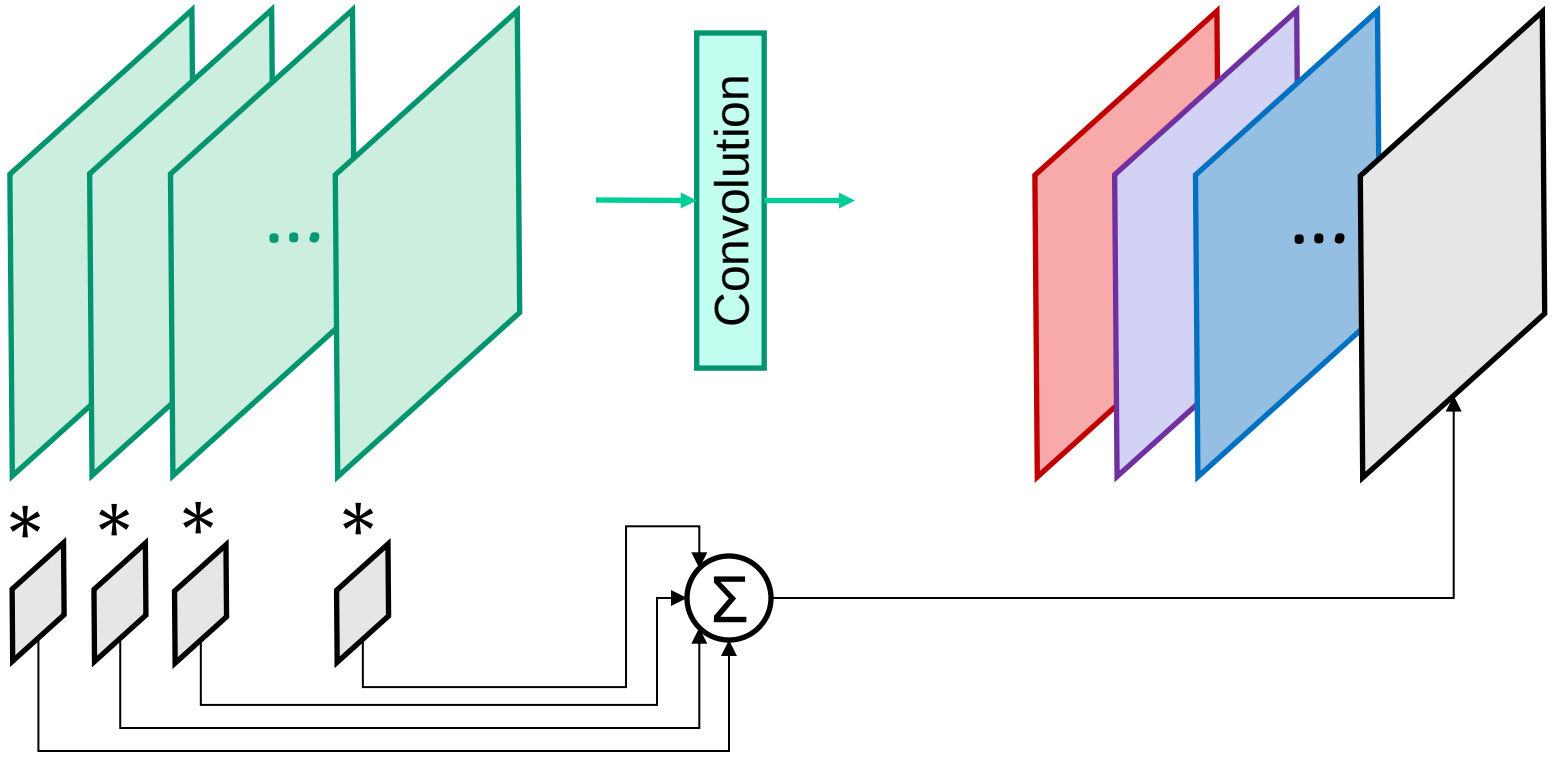
Convolutional Feature Mapping



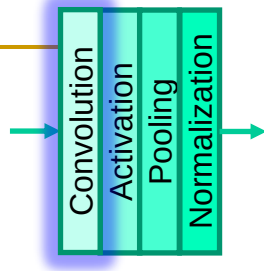


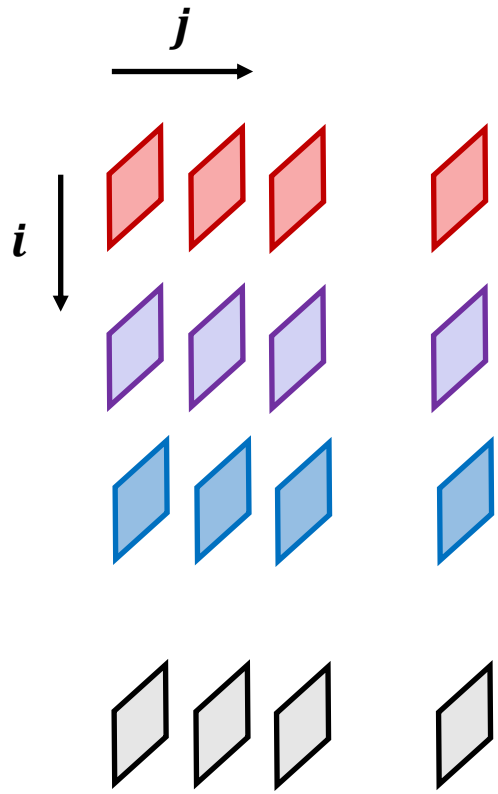
Convolutional Feature Mapping





Convolutional Feature Mapping

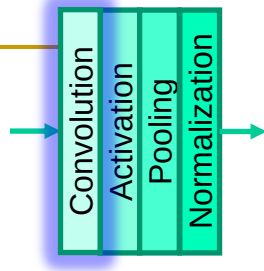


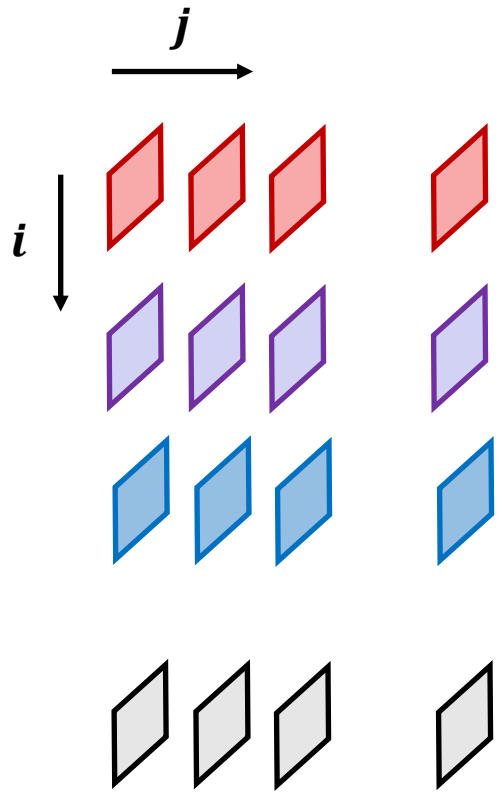


Convolutional filters (aka kernels) can be represented in Tensor format

$$\boldsymbol{K} \in \mathbb{R}^{\boldsymbol{h}\times\boldsymbol{w}\times\boldsymbol{N_I}\times\boldsymbol{N_O}}$$

Convolutional Feature Mapping



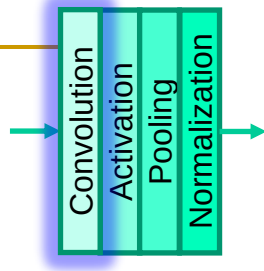


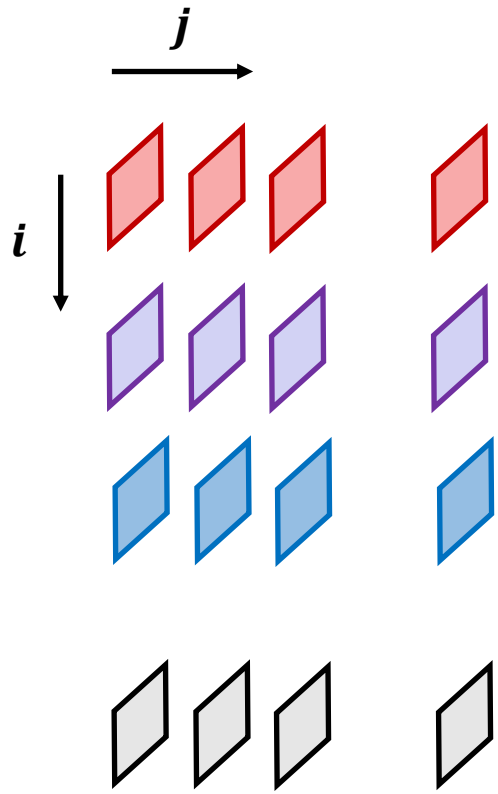
Convolutional filters (aka kernels) can be represented in Tensor format

$$\boldsymbol{K} \in \mathbb{R}^{\boldsymbol{h}\times\boldsymbol{w}\times\boldsymbol{N_I}\times\boldsymbol{N_O}}$$

Ex1. 3x3x64x128

Convolutional Feature Mapping





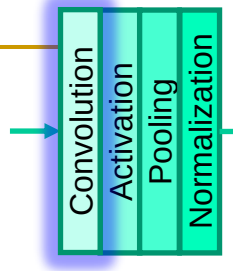
Convolutional filters (aka kernels) can be represented in Tensor format

$$\boldsymbol{K} \in \mathbb{R}^{\boldsymbol{h}\times\boldsymbol{w}\times\boldsymbol{N_I}\times\boldsymbol{N_O}}$$

Ex1. 3x3x64x128

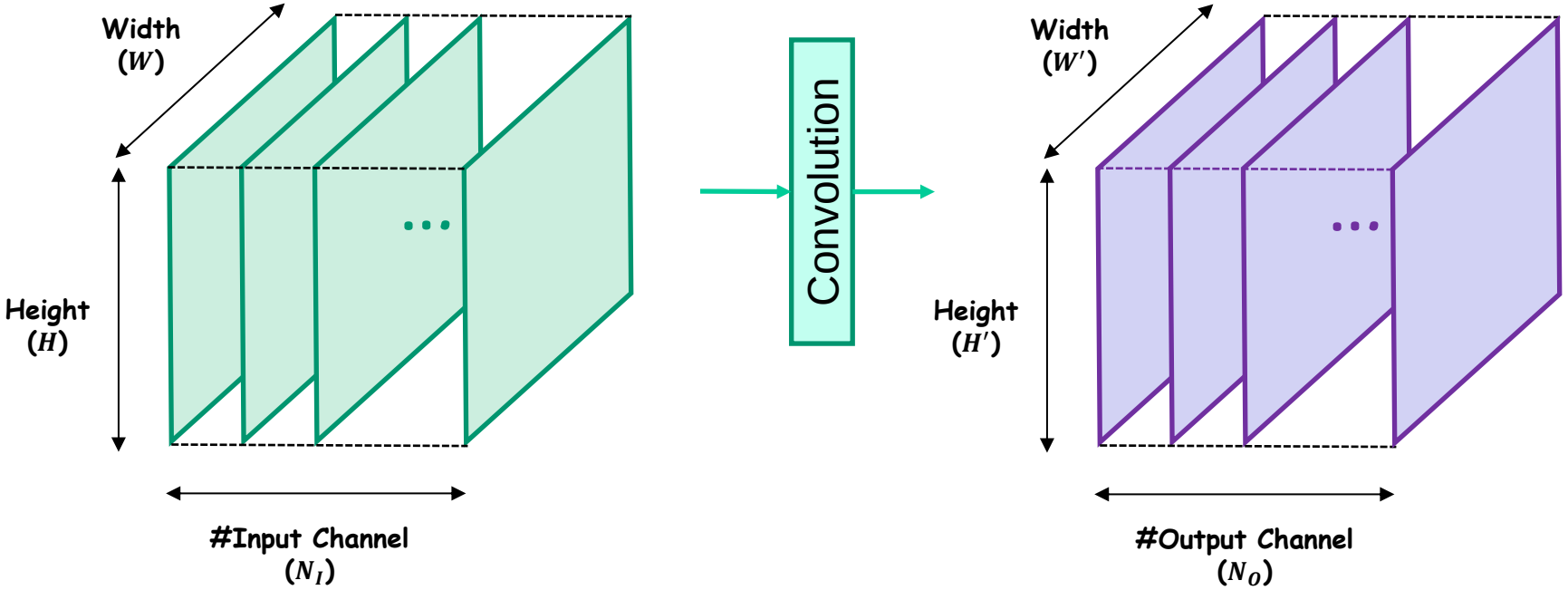
Ex2. 5x5x64x128

Convolutional Feature Mapping



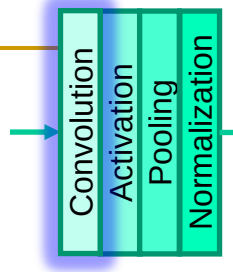
$$F^I \in \mathbb{R}^{H \times W \times N_I}$$

$$F^O \in \mathbb{R}^{H' \times W' \times N_O}$$



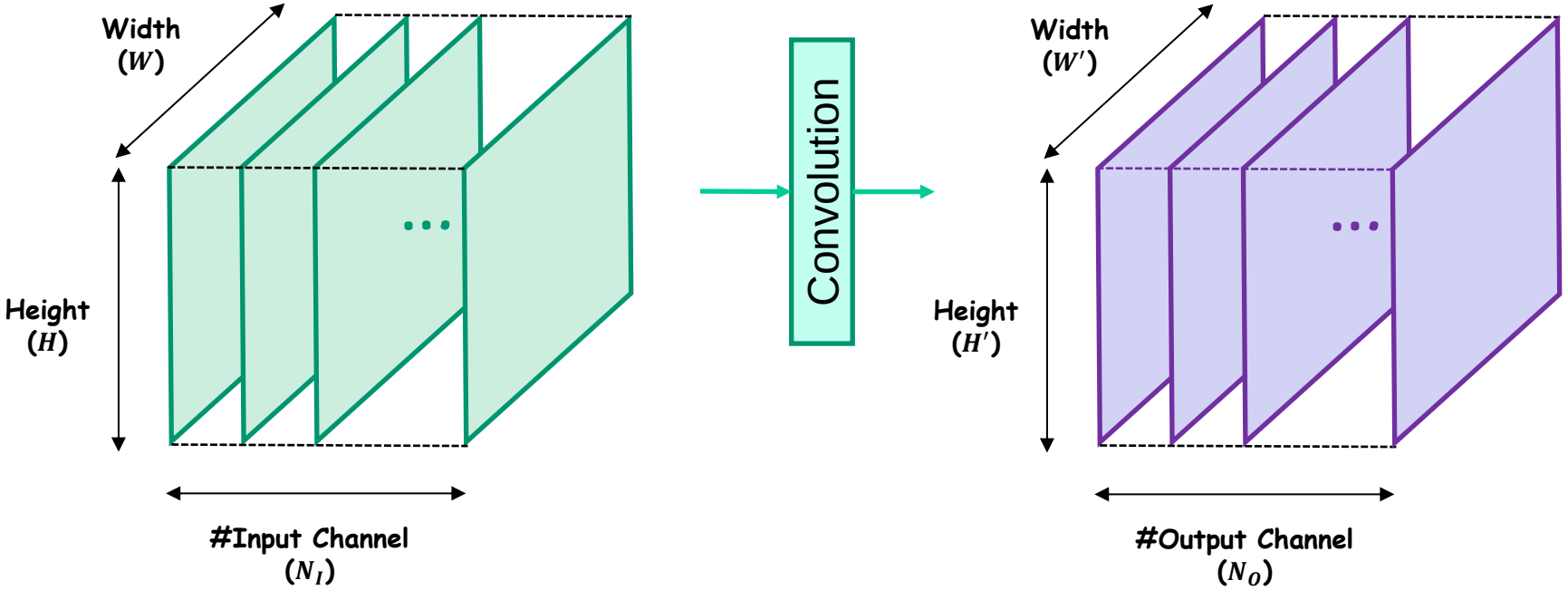
$$F^O(:,:,j) = \left[\sum_{i=1}^{N_I} F^I(:,:,i) * K(:,:,i,j)\right] + b_j$$

Convolutional Feature Mapping

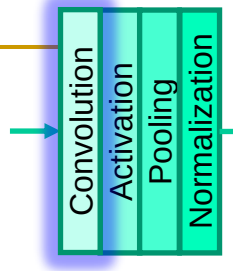


Ex1. Input Feature Map: 56x56x64 -->

Output Feature Map: 56x56x128 (Stride=1)

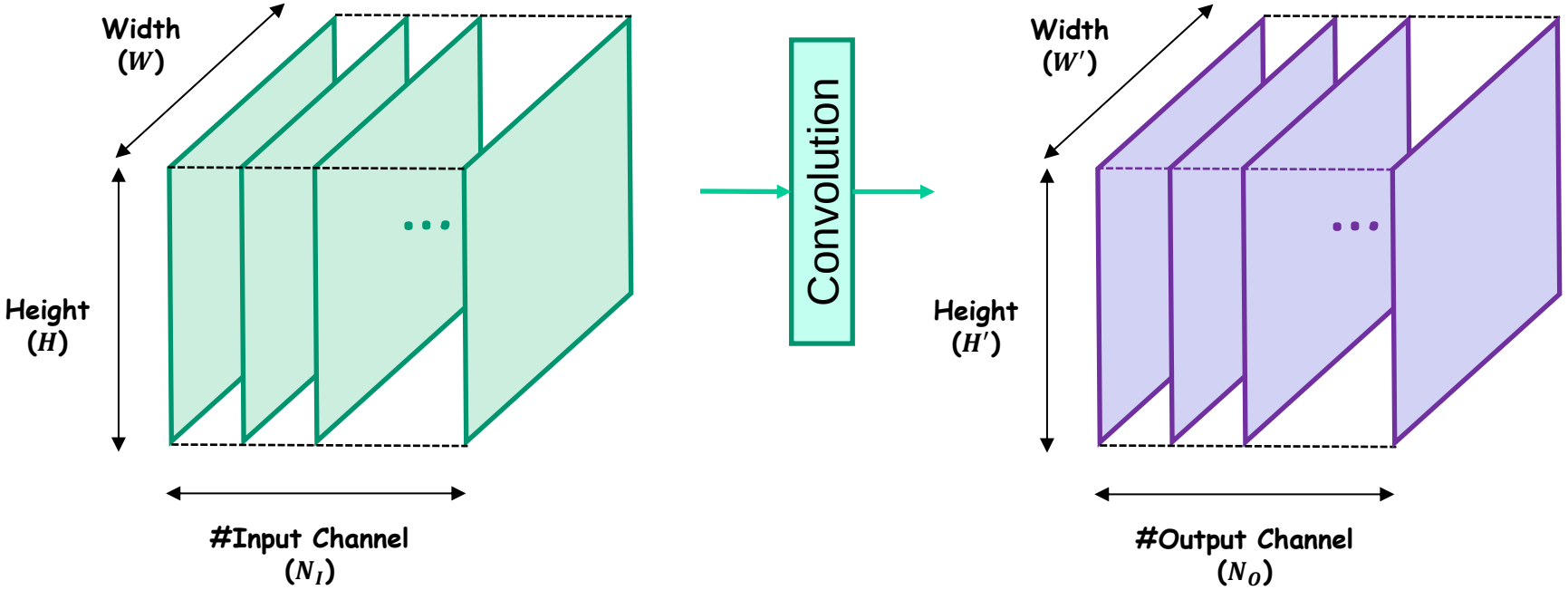


Convolutional Feature Mapping



Ex1. Input Feature Map: 56x56x64 -->

Output Feature Map: 28x28x128 (Stride=2)



Convolutional Neural Network: Extra Slides-2

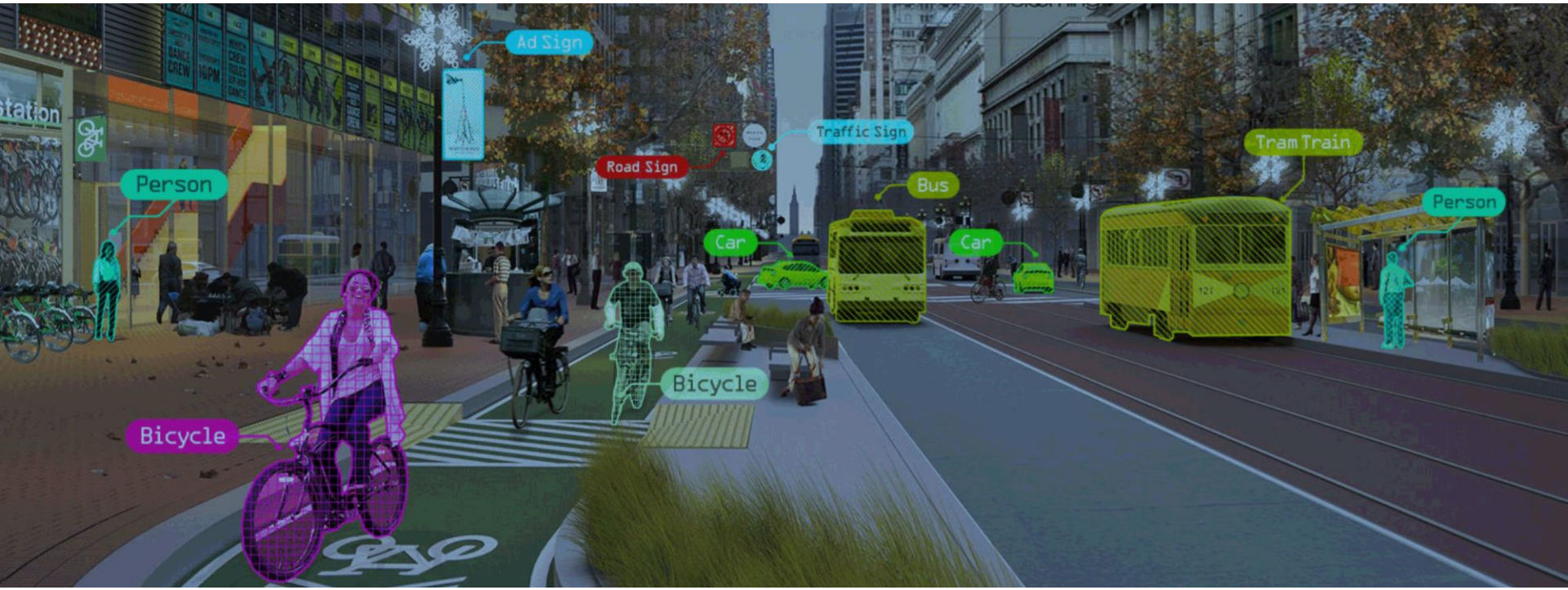
Outline

1. Applications
2. Backbone CNN Models
    1. Serial Cascading
    2. Serial/Parallel Cascading
    3. Residual Connection
    4. Depthwise/Separable Convolutions
    5. Squeeze-Excitation

How Computers Recognize Objects?

Question: Objects are anywhere in the scene (in any orientation, color hue, perspectives, illumination, etc), so how can we recognize them?

Answer: Learn a ton of features (millions) from the bottom up, by learning convolutional filters rather than pre-computing them



Feature Invariance to Perturbation is Hard

Viewpoint Variation
(Perspective Geometry)

Scale Variation

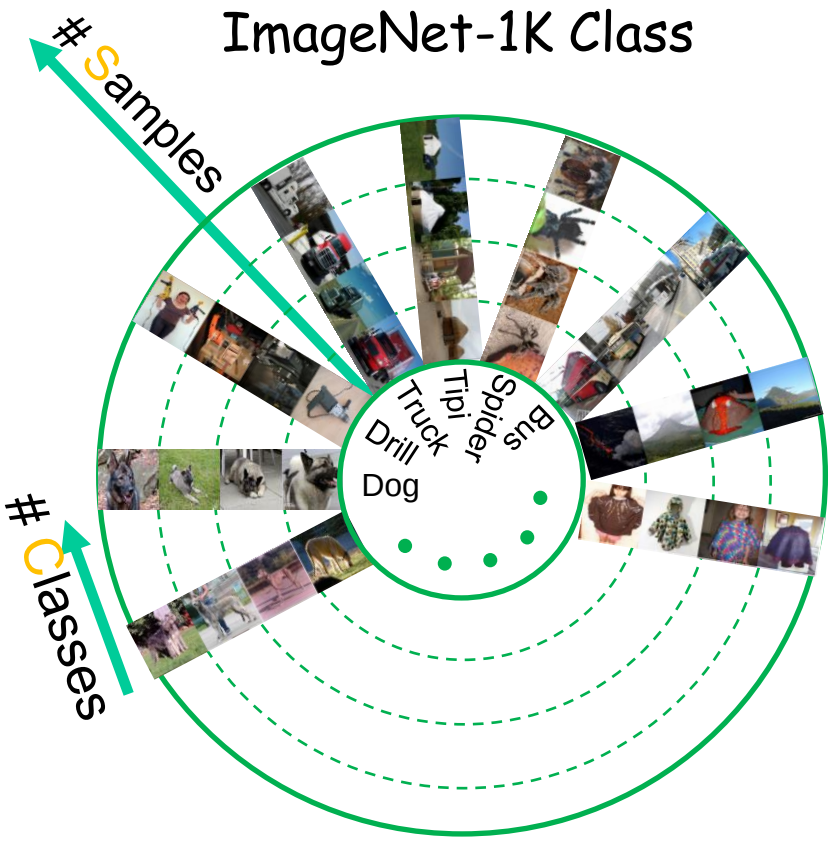
Deformation

Illumination Conditions

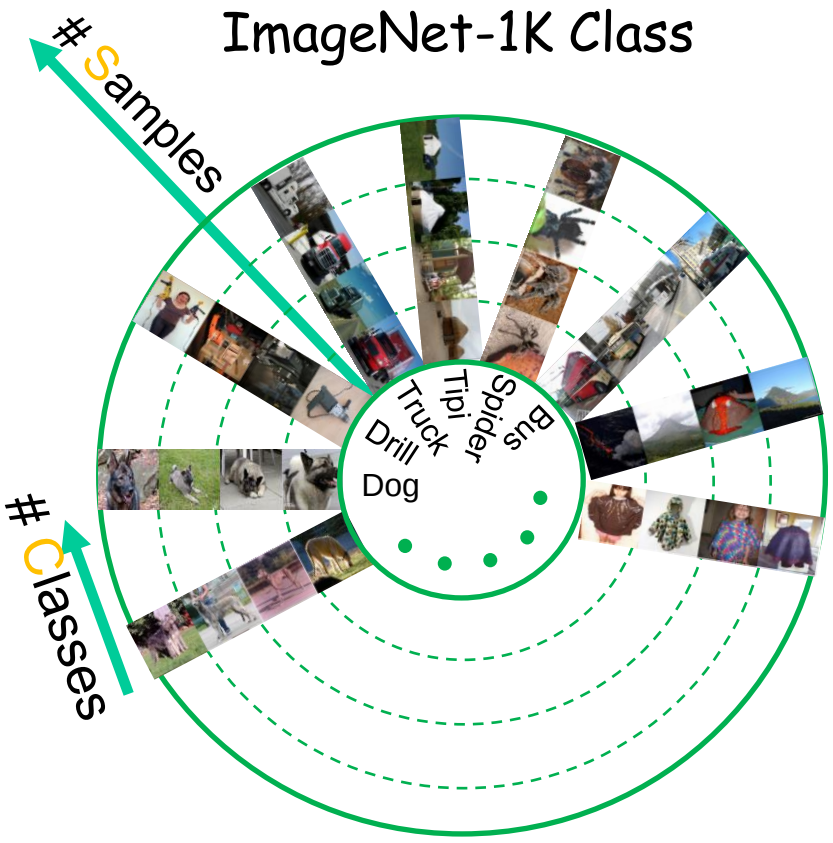
Background Clutter

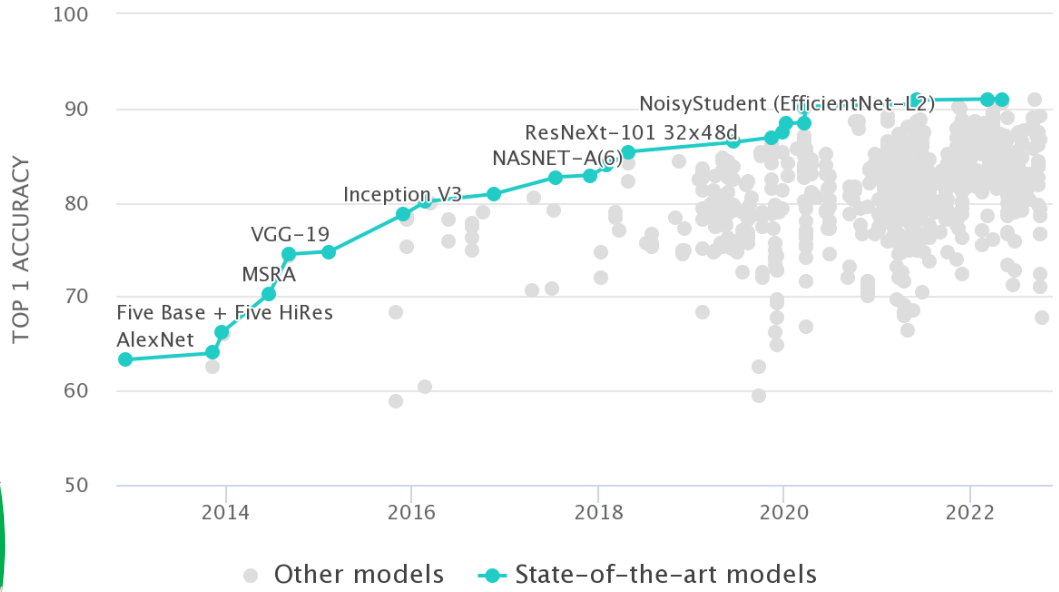
Intra-Class Variation

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

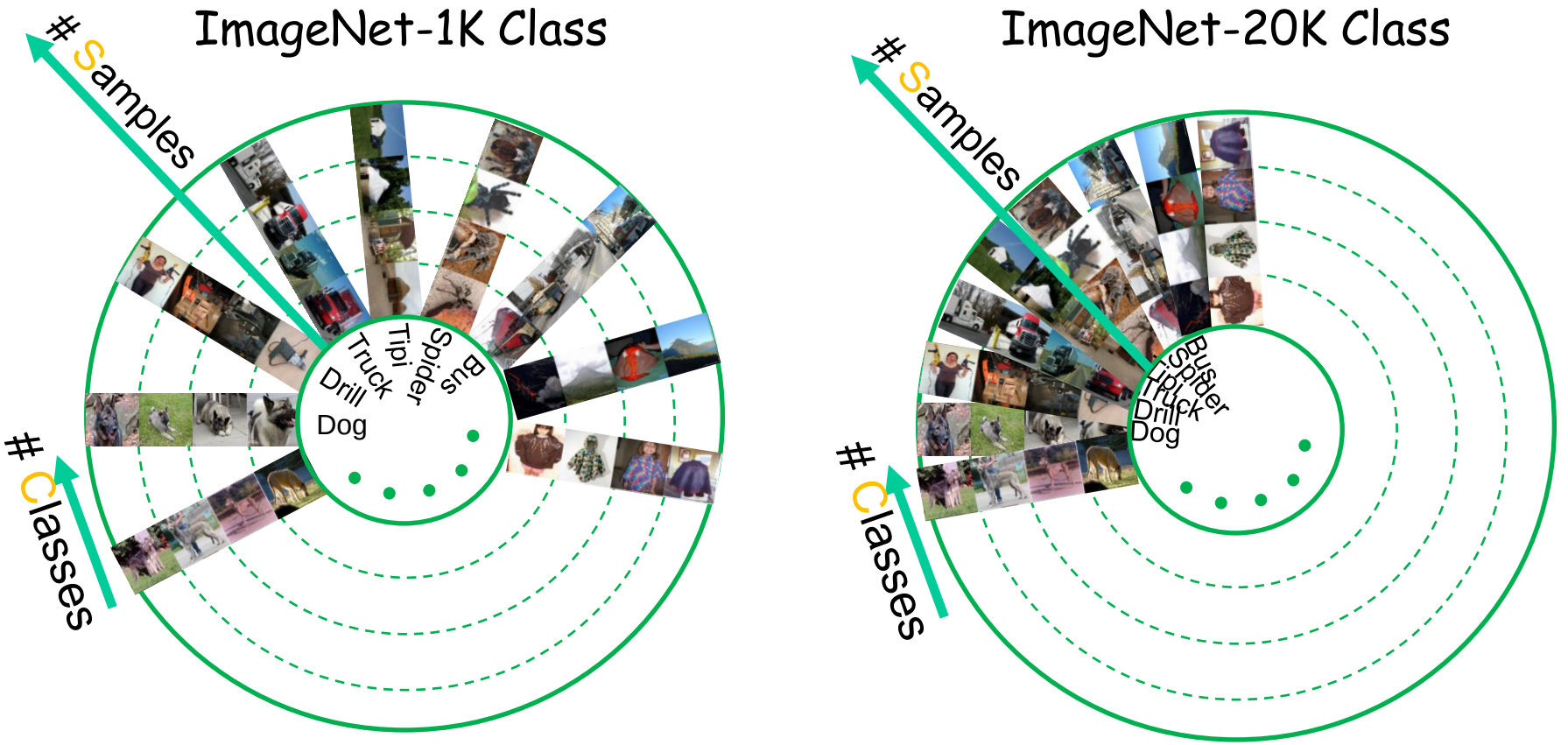


ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



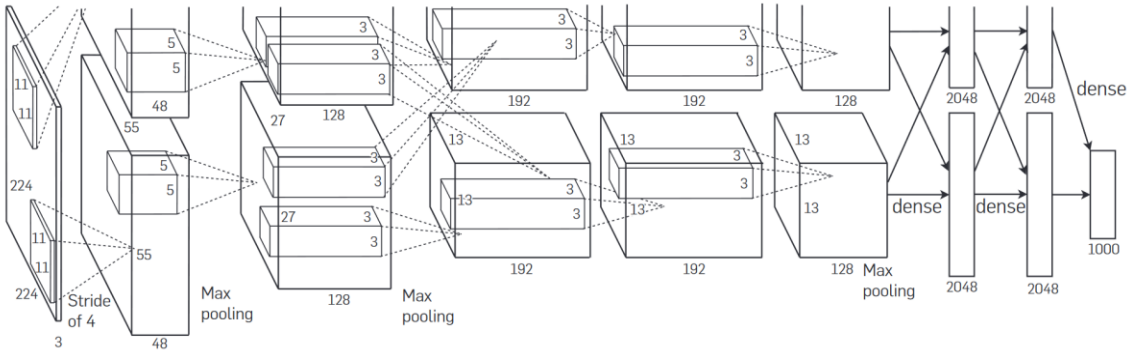


ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



Serial Cascade: AlexNet

- 5 Conv and 3 FC layers
- ReLU Activation
- Training on Multiple GPUs (GTX 580 with 3GB memory)
- Response Normalization
- Overlapping Pooling
- Heavy data augmentation
  - Image translation/horizontal reflection
  - Altering intensities of RGB channels using PCA



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

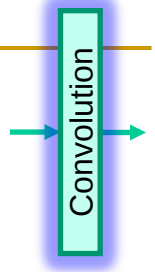
ImageNet Classification with Deep Convolutional Neural Networks

Part of Advances in Neural Information Processing Systems 25 (NIPS 2012)

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

Cited by 117842

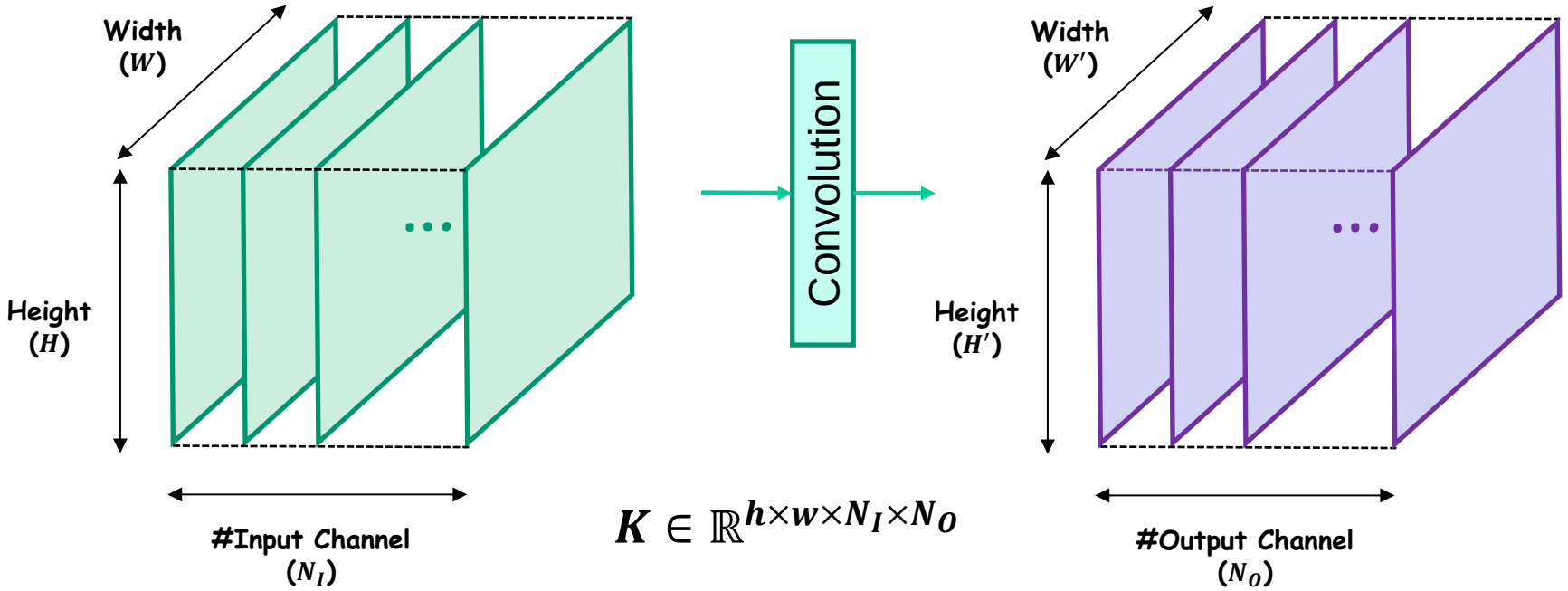
Serial Cascade: AlexNet



Reminder from Convolution Feature mapping

$$F^I \in \mathbb{R}^{H \times W \times N_I}$$

$$F^O \in \mathbb{R}^{H' \times W' \times N_O}$$

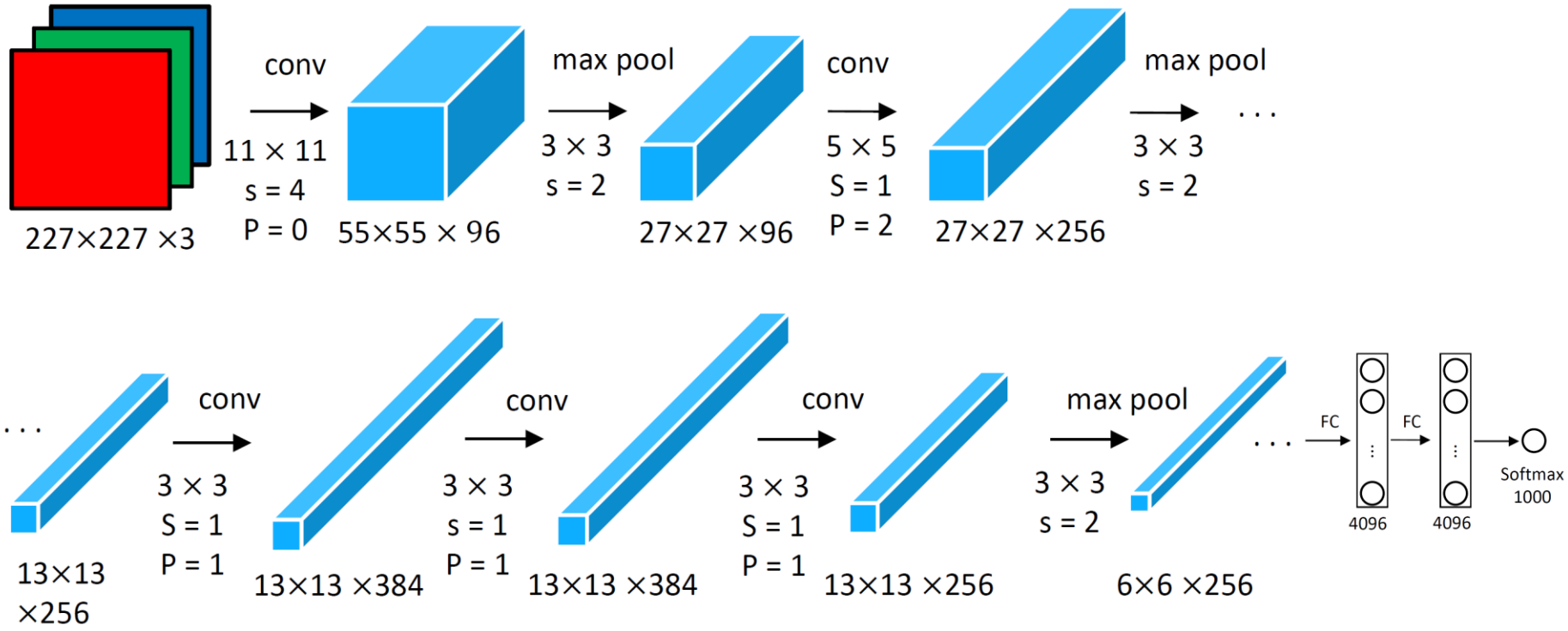


$$K \in \mathbb{R}^{h \times w \times N_I \times N_O}$$

$$F^O(:,:,j) = \left[\sum_{i=1}^{N_I} F^I(:,:,i) * K(:,:,i,j) \right] + b_j$$

Serial Cascade: AlexNet

- Feature mapping via cascaded convolutional layers



ImageNet Classification with Deep Convolutional Neural Networks

Part of Advances in Neural Information Processing Systems 25 (NIPS 2012)

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

Cited by 117842

Serial Cascade: AlexNet

- AlexNet was the coming out party for CNNs in the computer vision community. This was the first time a model performed so well on a historically difficult ImageNet dataset.

Table 1. Comparison of results on ILSVRC-2010 test set.

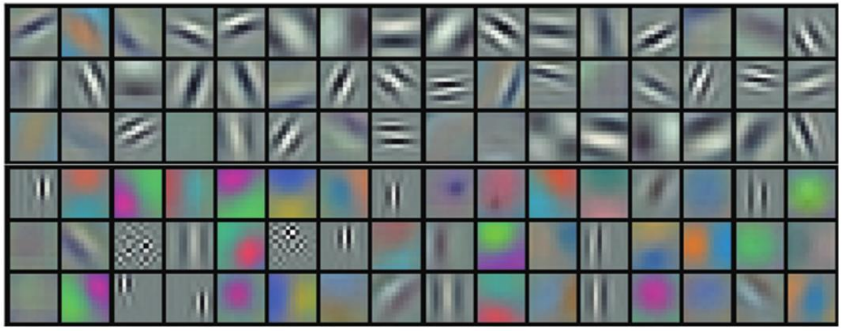
Model	Top-1 (%)	Top-5 (%)
Sparse coding[2]	*47.1*	*28.2*
SIFT + FVs[29]	*45.7*	*25.7*
CNN	**37.5**	**17.0**

Table 2. Comparison of error rates on ILSVRC-2012 validation and test sets.

Model	Top-1 (val, %)	Top-5 (val, %)	Top-5 (test, %)
SIFT + FVs[6]	–	–	*26.2*
1 CNN	40.7	18.2	–
5 CNNs	38.1	16.4	**16.4**
1 CNN*	39.0	16.6	–
7 CNNs*	36.7	15.4	**15.3**

- How did the learned kernels responses look like?

Figure 3. Ninety-six convolutional kernels of size 11 × 11 × 3 learned by the first convolutional layer on the 224 × 224 × 3 input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2 (see Section 7.1 for details).



ImageNet Classification with Deep Convolutional Neural Networks

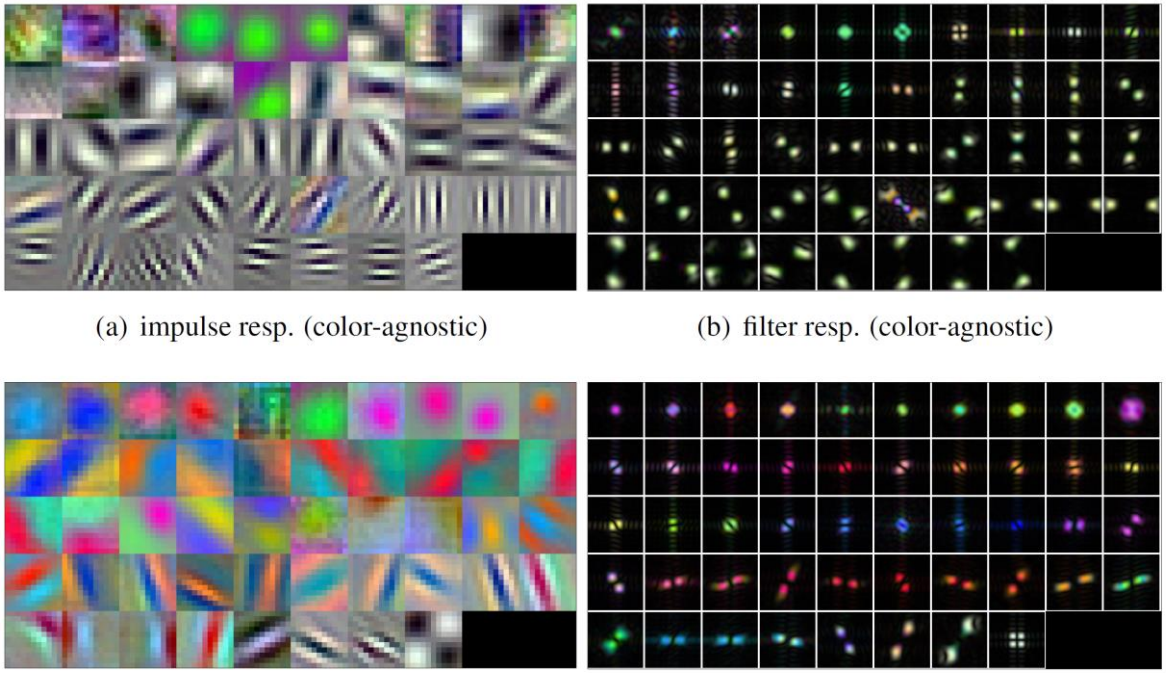
Part of Advances in Neural Information Processing Systems 25 (NIPS 2012)

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

Cited by 117842

Serial Cascade: AlexNet

- Further analysis on AlexNet pretrained kernels (e.g. in 1st layer) reveals that convolution kernels encode features in different orientations, frequencies, and colors



(a) impulse resp. (color-agnostic) (b) filter resp. (color-agnostic)

(c) impulse resp. (color-specific) (d) filter resp. (color-specific)

Figure 1. AlexNet first layer convolution kernels for color-agnostic and color-specific sets.

ImageNet Classification with Deep Convolutional Neural Networks

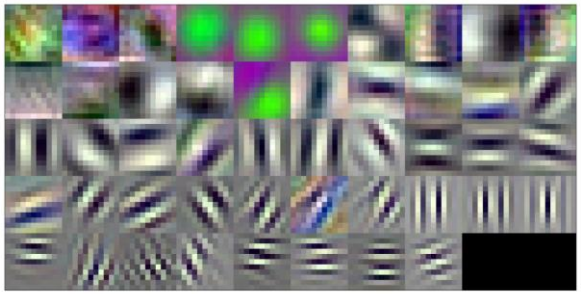
Part of Advances in Neural Information Processing Systems 25 (NIPS 2012)

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

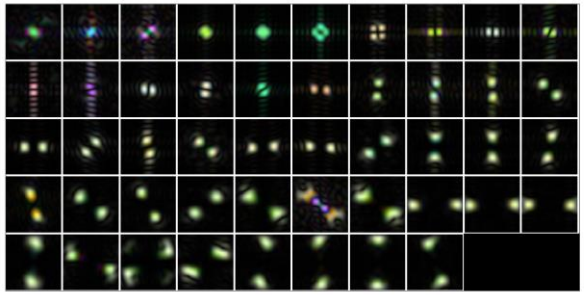
Cited by 117842

Serial Cascade: AlexNet

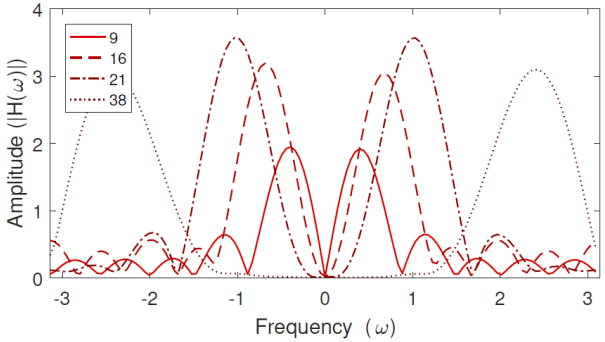
- Further analysis on AlexNet pretrained kernels (e.g. in 1st layer) reveals that convolution kernels encode features in different orientations, frequencies, and colors



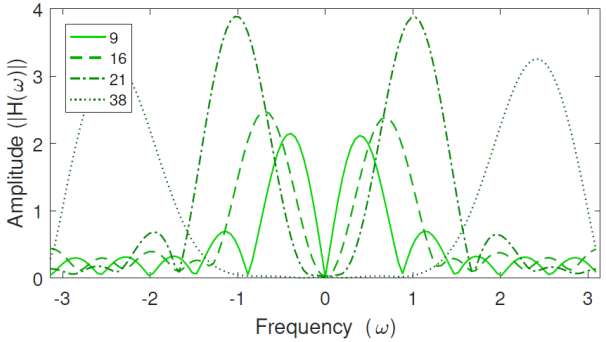
(a) impulse resp. (color-agnostic)



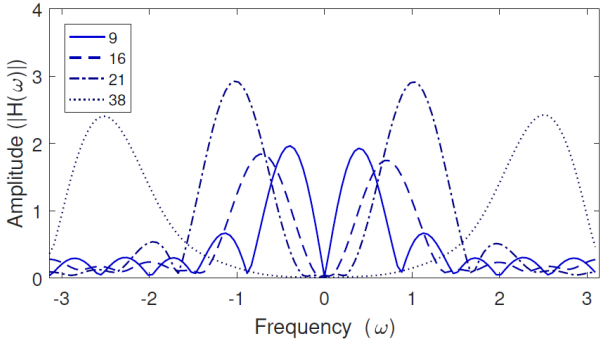
(b) filter resp. (color-agnostic)



(e) Red channel



(f) Green channel



(g) Blue channel

ImageNet Classification with Deep Convolutional Neural Networks

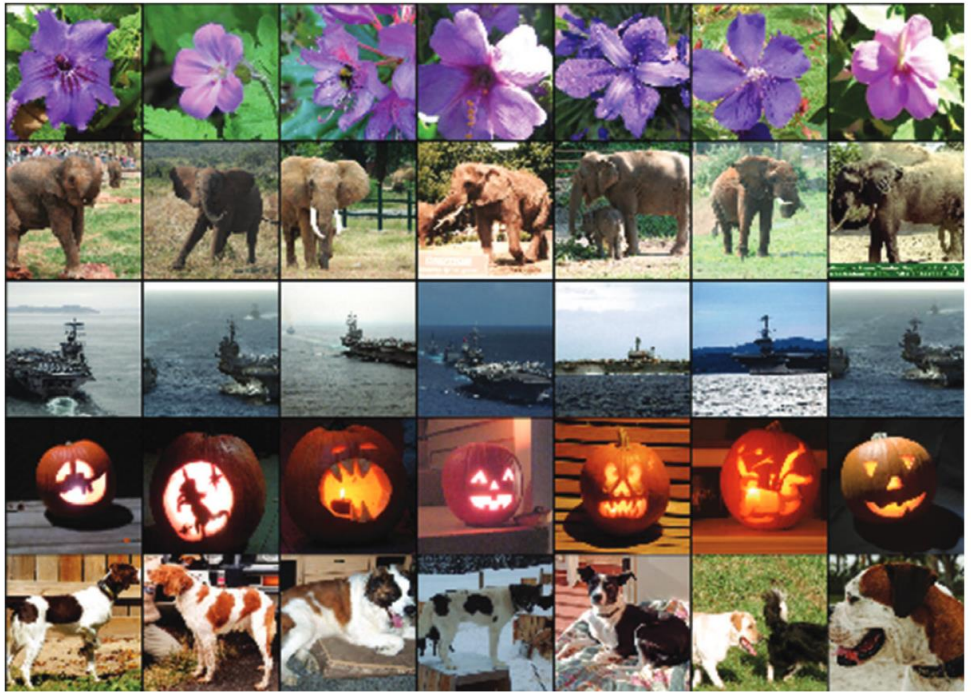
Part of Advances in Neural Information Processing Systems 25 (NIPS 2012)

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

Cited by 117842

Serial Cascade: AlexNet

Figure 4. (Left) Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5). (Right) Five ILSVRC-2010 test images in the first column. The remaining columns show the six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.



ImageNet Classification with Deep Convolutional Neural Networks

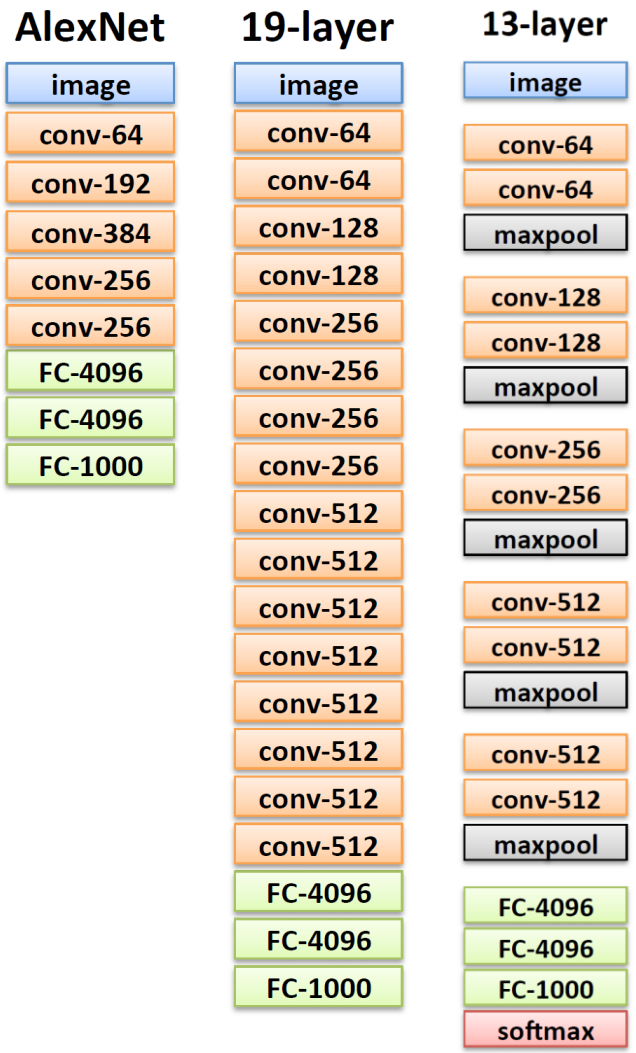
Part of Advances in Neural Information Processing Systems 25 (NIPS 2012)

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton

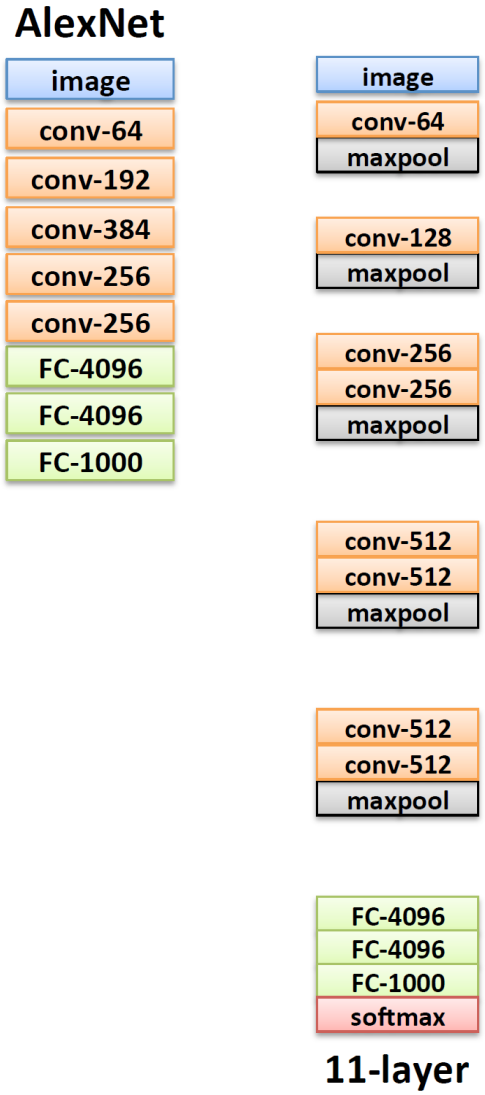
Cited by 117842

Serial Cascade: Going Deeper with VGG

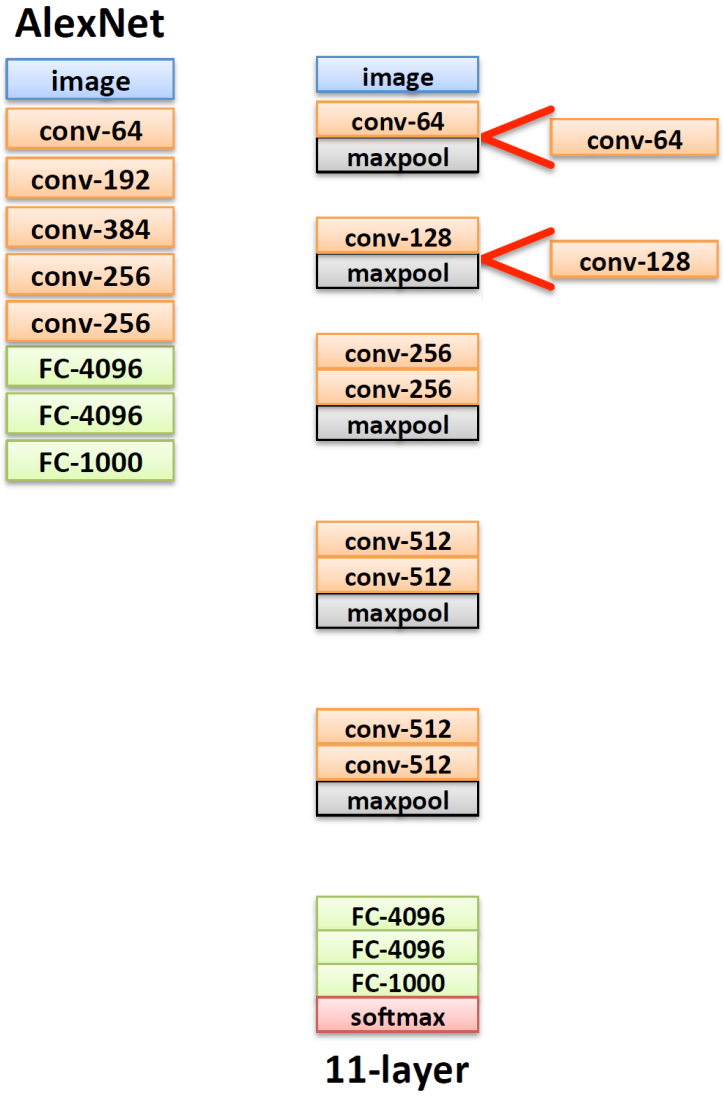
- Investigate the effect of the convolutional network depth
- Great boost is achieved by increasing #layers to 16-19
- Won ILSVRC2014 challenge
- Key design choice
  - 3x3 kernel size
  - Stack of conv layers w/o pooling
  - Conv stride=1 (no skipping)
  - ReLU activation
  - 5 Max-pooling (x2 downsampling)
  - 3 FC layers
- Later designs added Batch Normalization



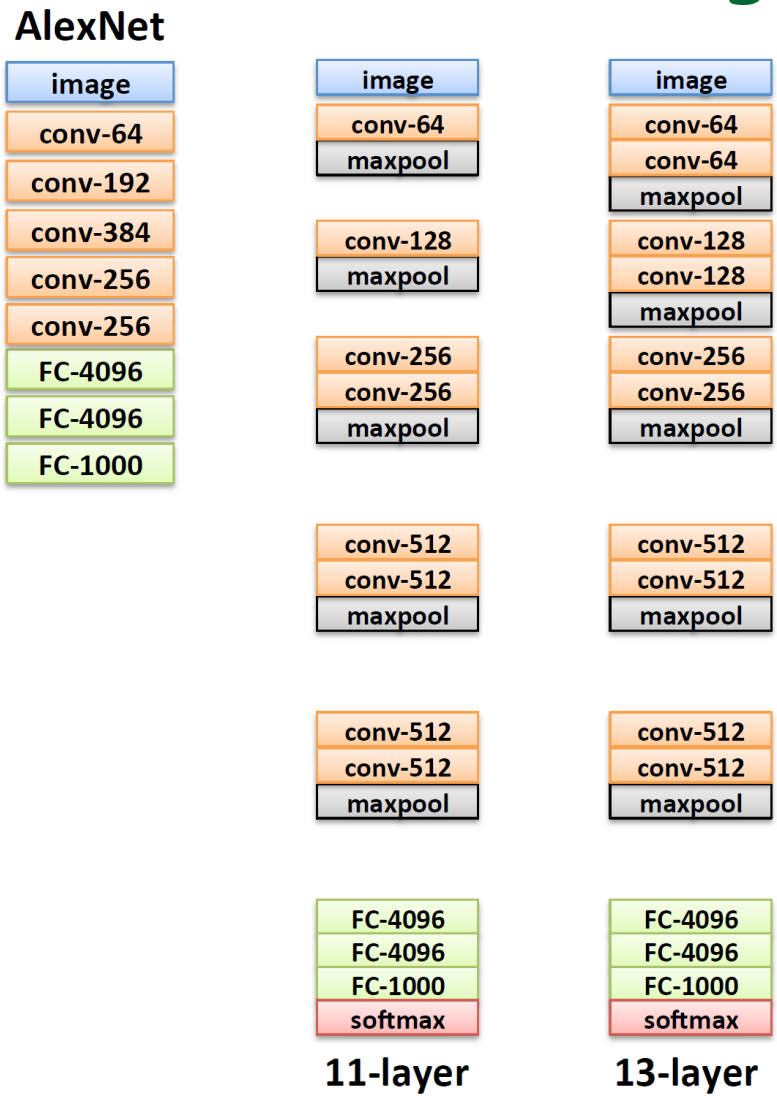
Serial Cascade: Going Deeper with VGG



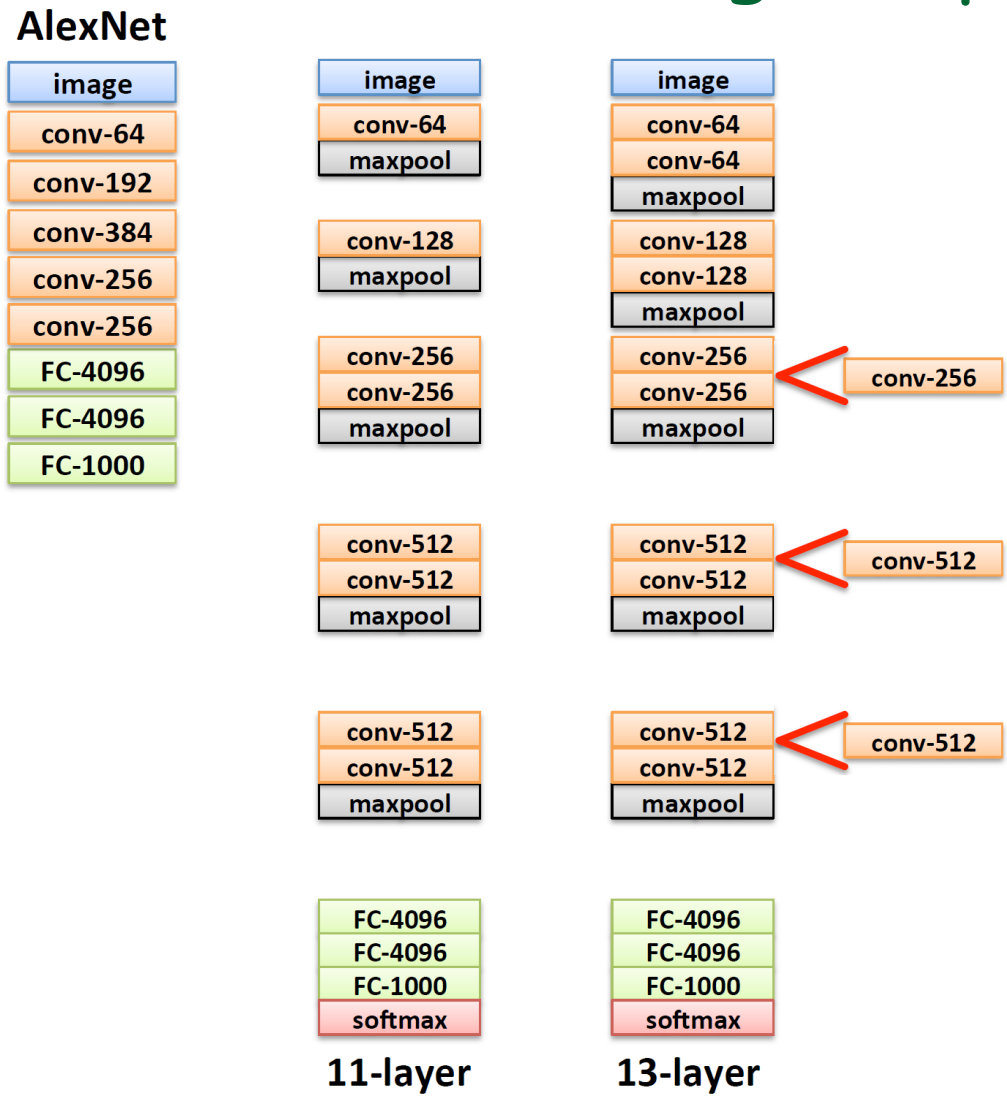
Serial Cascade: Going Deeper with VGG



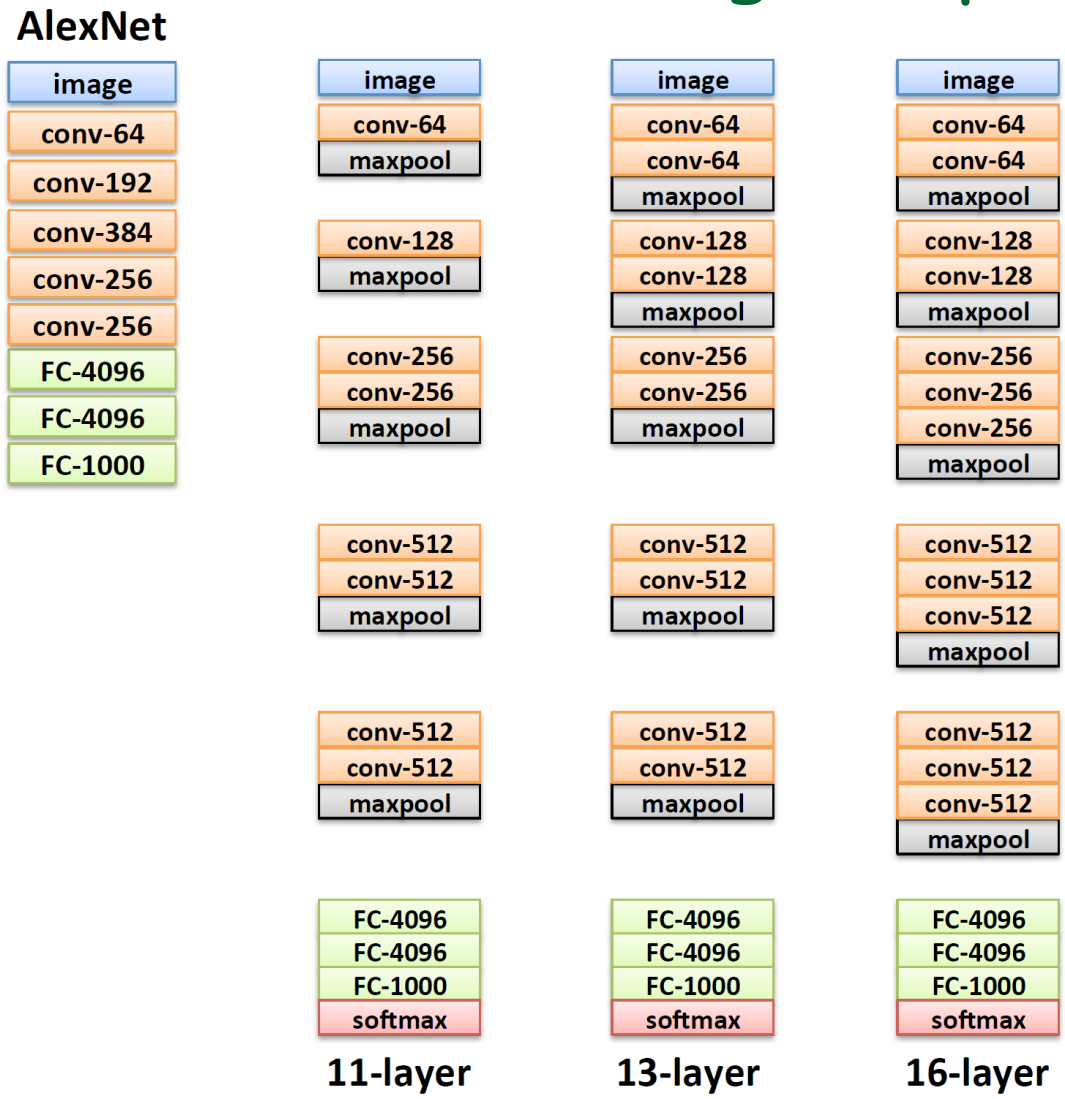
Serial Cascade: Going Deeper with VGG



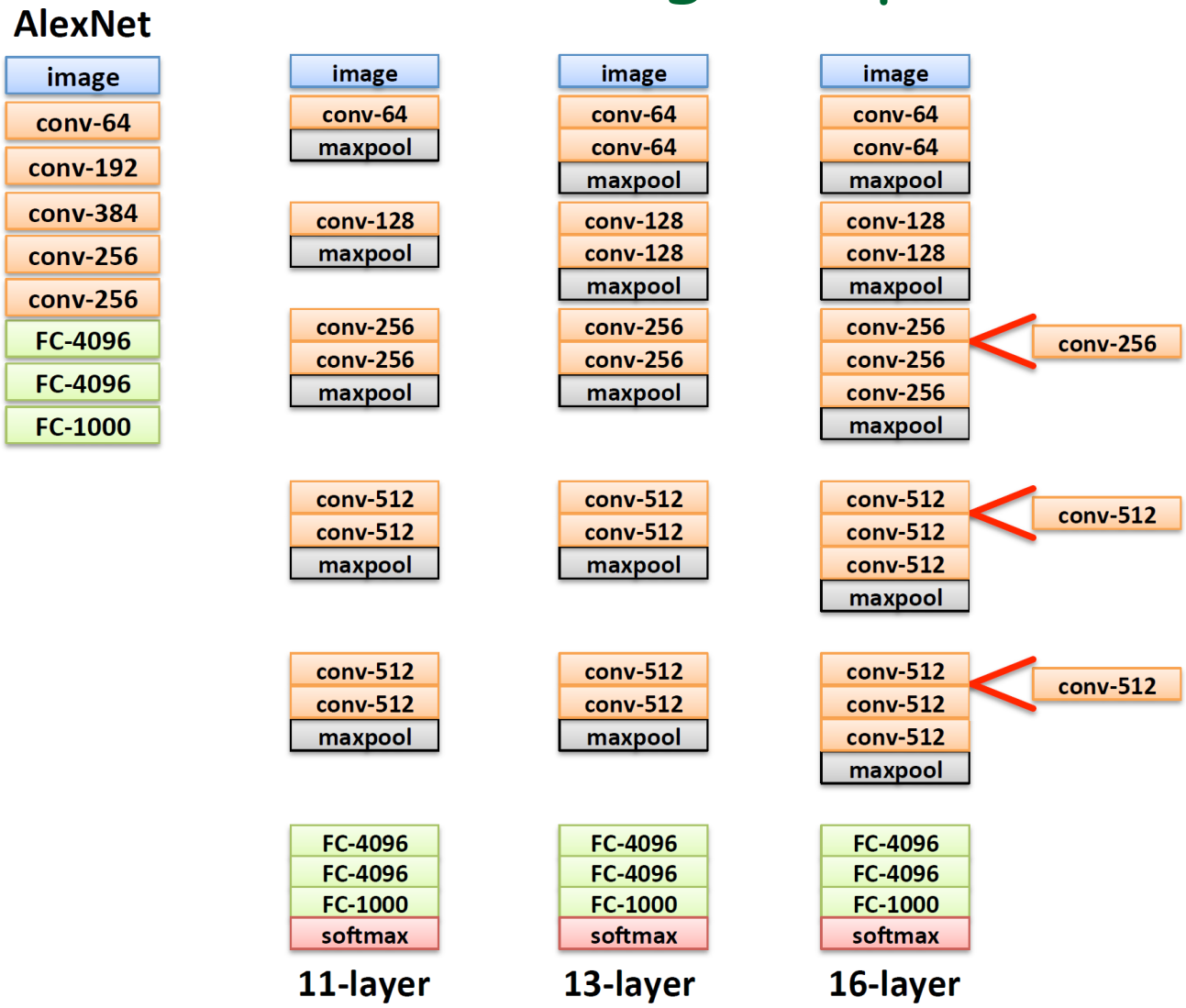
Serial Cascade: Going Deeper with VGG



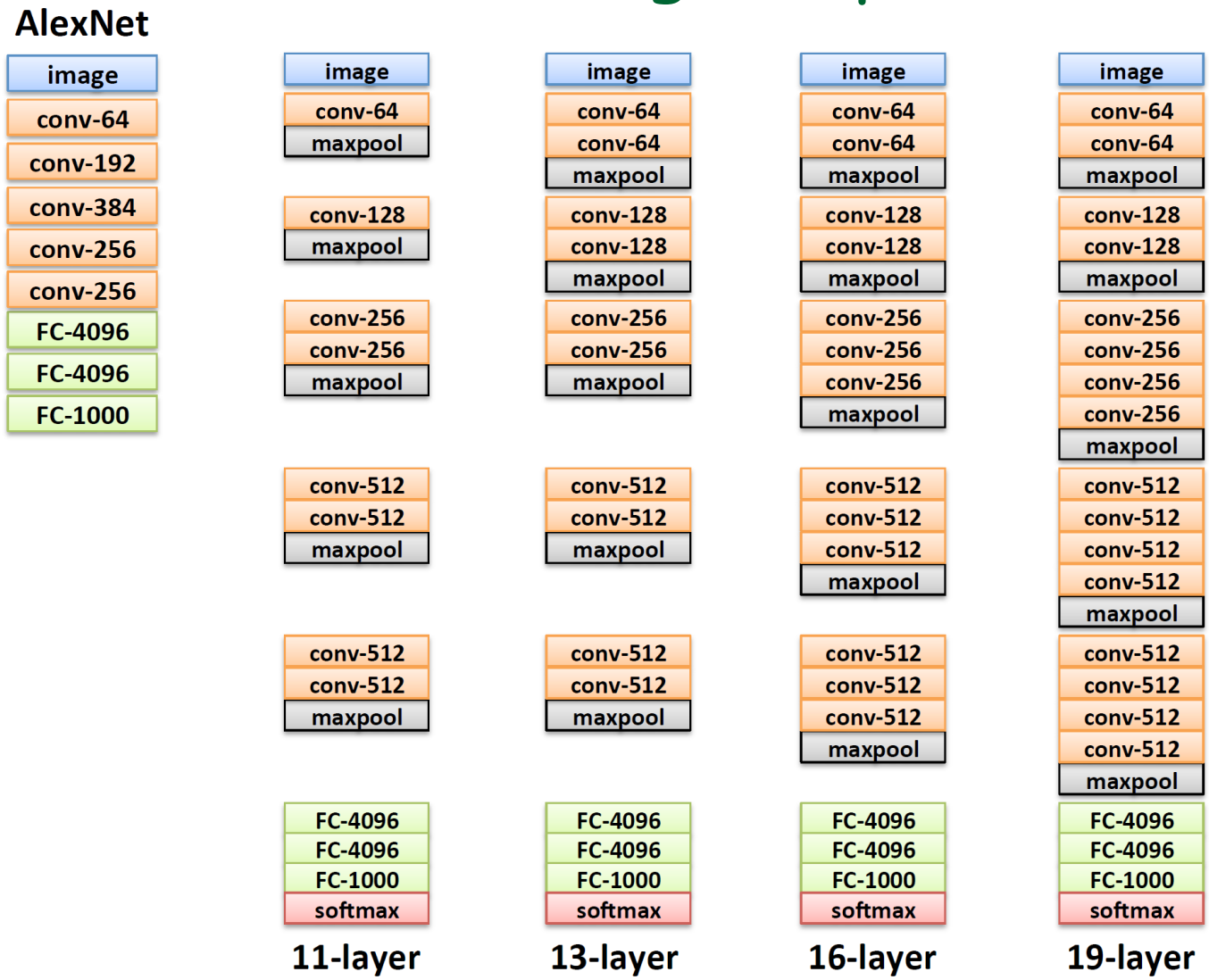
Serial Cascade: Going Deeper with VGG



Serial Cascade: Going Deeper with VGG



Serial Cascade: Going Deeper with VGG

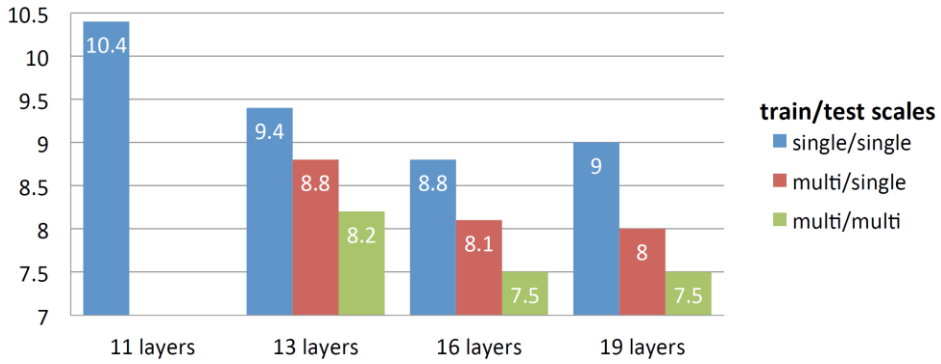


Serial Cascade: Going Deeper with VGG—Training Phase

- Input training image: fixed size of 224x224 crop
- Images have varying size, so upscale to e.g. 384x(N>384)
- Random crop 224x224
- Standard augmentation: random flip and RGB shift
- SGD-Momentum (next lecture)
- Regularization: dropout and weight decay
- Fast convergence (74 training epochs)
- Initialization (some sort of transfer-learning)
  - Deeper networks are prone to vanishing-gradients
  - 11-layer net: random initialization from N(0;0.01)
  - Deeper nets: Top & bottom layers initialized with 11-layer. Other layers: random initialization

Serial Cascade: Going Deeper with VGG—Testing Phase (ImageNet-1k)

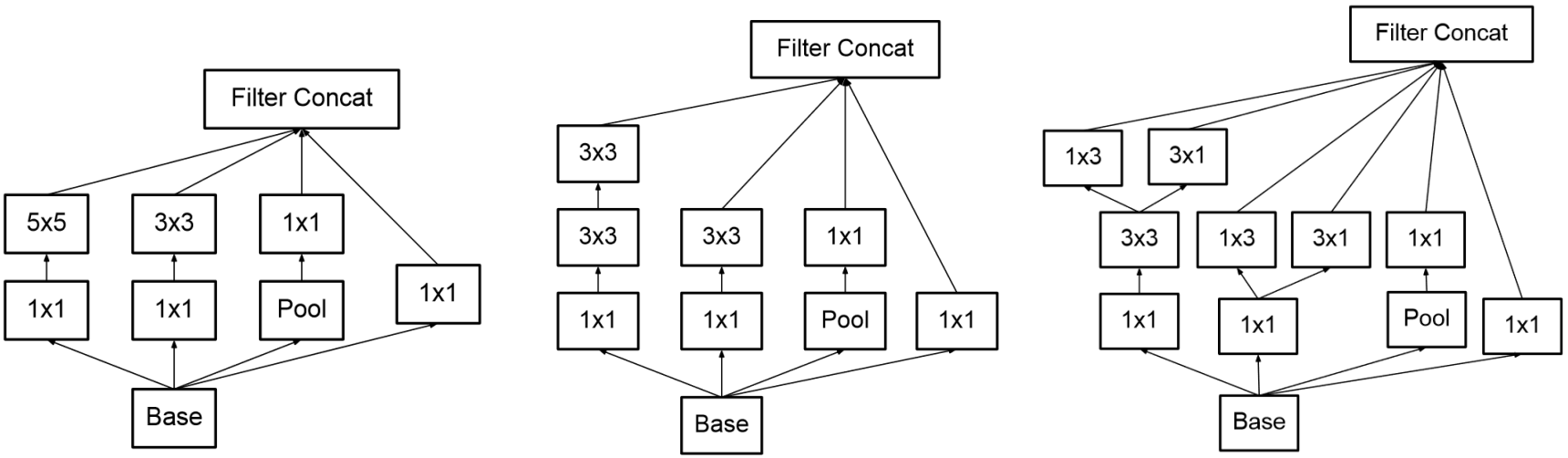
- Evaluation on variable size images
- Testing on multiple 224x224 crops [AlexNet]
- Multiple scales are tested (256xN, 384xN, 512xN) and class score averaged



- Error decreases with depth
- Using multiple scales is important
  - Multi-scale training outperforms single scale training =
  - Multi-scale testing further improves the results

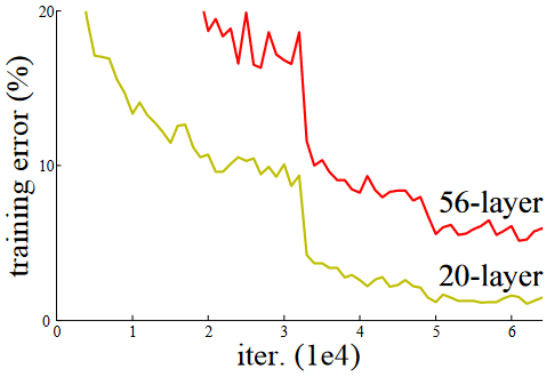
Serial/Parallel Cascade: Inception Net

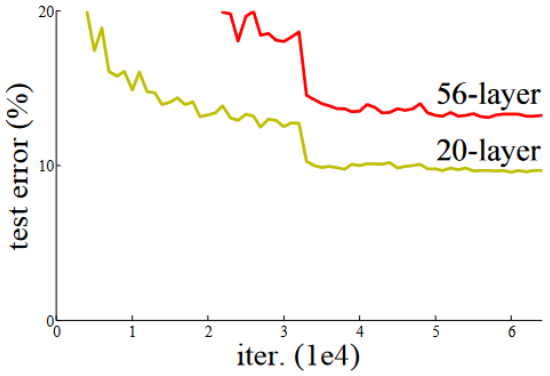
- Multiple scales are encoded in parallel and cascaded to next layers



Residual Connection: ResNet Architecture

- Is learning better networks as easy as stacking more layers?
- **Obstacle:** deeper networks are difficult to train because of the notorious problem of vanishing/exploding gradients
- Early solutions were proposed by introducing
  - normalized initialization
  - intermediate normalization layer
- Still accuracy gets saturated with increasing depth and then degrades rapidly (this is not caused by overfitting!)
- Adding more layers leads to higher training error





Residual Connection: ResNet Architecture

- **Residual learning:** instead of hoping each few stacked layers directly fit a desired underling mapping, we explicitly let these layers fit a residual mapping
- Define underlying forward mapping by H(x) := F(x) + x

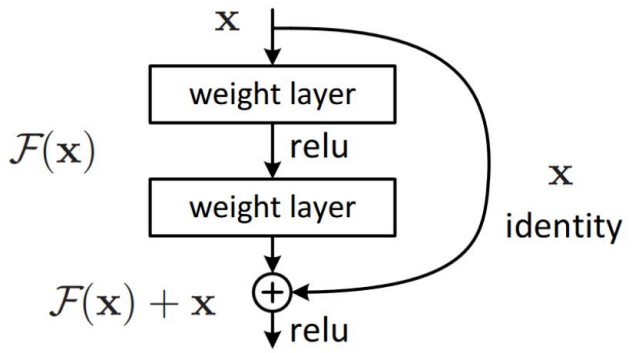


Figure 2. Residual learning: a building block.

- Why this should help?

Residual Connection: ResNet Architecture

- **Residual learning:** instead of hoping each few stacked layers directly fit a desired underling mapping, we explicitly let these layers fit a residual mapping
- Define underlying forward mapping by H(x) := F(x) + x

- **Residual learning:** instead of hoping each few stacked layers directly fit a desired underling mapping, we explicitly let these layers fit a residual mapping
- Define underlying forward mapping by H(x) := F(x) + x

- Corresponding gradient back-propagation:

$$\frac{\partial \epsilon}{\partial x} = \frac{\partial \epsilon}{\partial H(x)} \cdot \frac{\partial H(x)}{\partial x} = \frac{\partial \epsilon}{\partial H(x)} \cdot \left[\frac{\partial F}{\partial x} + 1\right] = \frac{\partial \epsilon}{\partial H(x)} \cdot \frac{\partial F}{\partial x} + \frac{\partial \epsilon}{\partial H(x)}$$

- Gradient from output layer transfers directly to the input layer and avoids vanishing

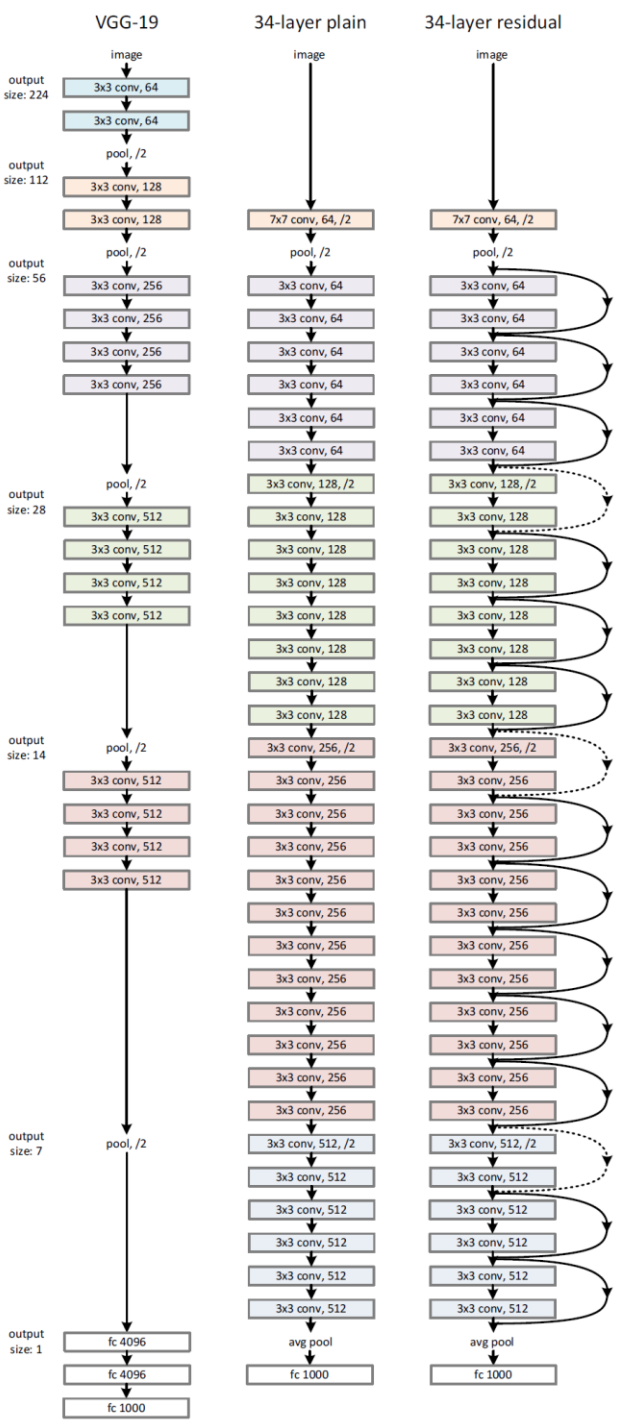
- Corresponding gradient back-propagation:

$$\frac{\partial \epsilon}{\partial x} = \frac{\partial \epsilon}{\partial H(x)} \cdot \frac{\partial H(x)}{\partial x} = \frac{\partial \epsilon}{\partial H(x)} \cdot \left[\frac{\partial F}{\partial x} + 1\right] = \frac{\partial \epsilon}{\partial H(x)} \cdot \frac{\partial F}{\partial x} + \frac{\partial \epsilon}{\partial H(x)}$$

- Gradient from output layer transfers directly to the input layer and avoids vanishing

Residual Connection: ResNet

- Plain baseline is inspired by VGG nets
- Shortcuts introduced on plain baseline
- Same number of filters are used for the same output feature map size
- If the feature map size is halved, the number of filters is doubled
- Down-sampling by stride=2
- Network ends with global-average-pooling
- 1000-way FC layer with softmax
- ResNet34 has 3.6 BFLOPS (18% of VGG19 i.e. 19.6 BFLOPS)



Residual Connection: ResNet Architecture

- Different ResNet architectures of ResNet18, ResNet34, ResNet50, ResNet101, ResNet152

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

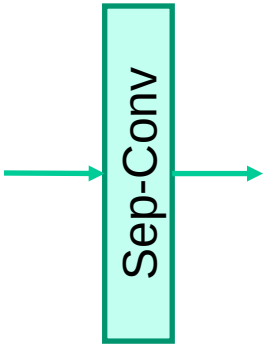
Residual Connection: ResNet Architecture

- ResNet performance on ImageNet

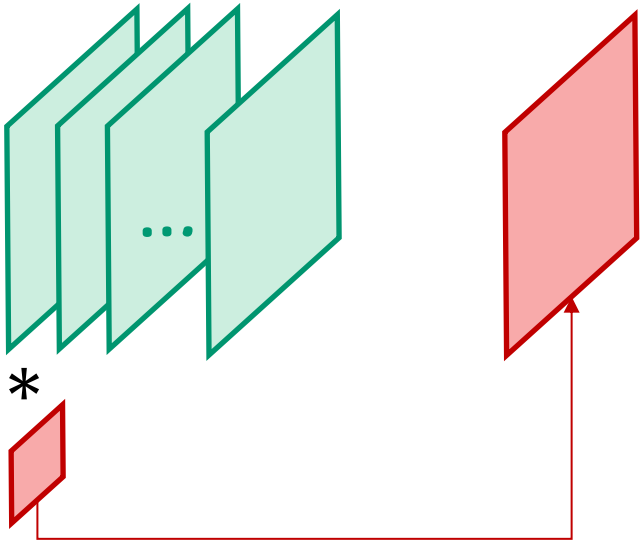
model	top-1 err.	top-5 err.
VGG-16 [40]	28.07	9.33
GoogLeNet [43]	-	9.15
PReLU-net [12]	24.27	7.38
plain-34	28.54	10.02
ResNet-34 A	25.03	7.76
ResNet-34 B	24.52	7.46
ResNet-34 C	24.19	7.40
ResNet-50	22.85	6.71
ResNet-101	21.75	6.05
ResNet-152	**21.43**	**5.71**

Table 3. Error rates (%, **10-crop** testing) on ImageNet validation. VGG-16 is based on our test. ResNet-50/101/152 are of option B that only uses projections for increasing dimensions.

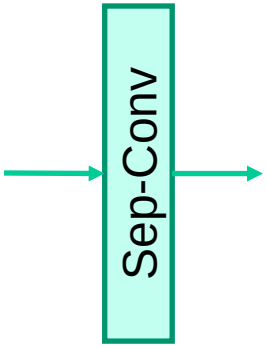
Depthwise/Separable Convolutions



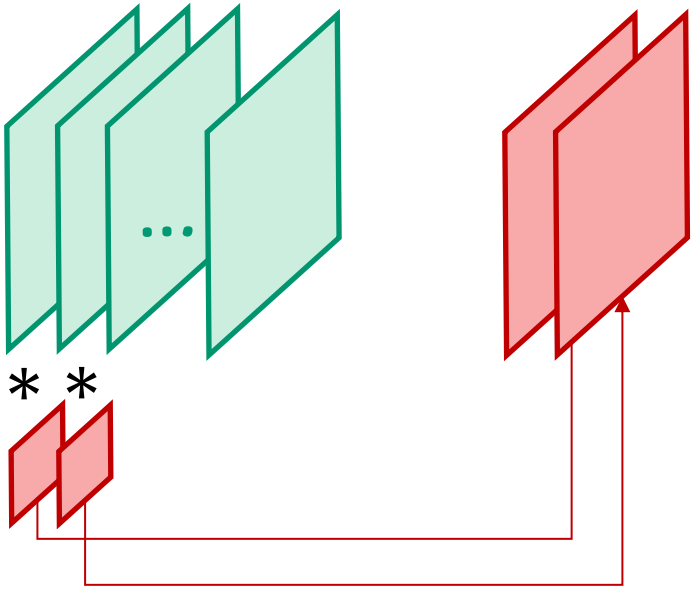
- Keep input channel convolution separate from mapping to output feature map



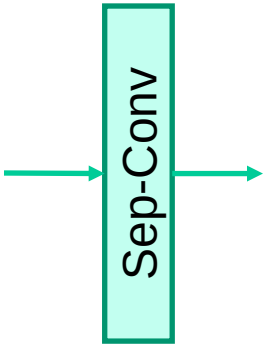
Depthwise/Separable Convolutions



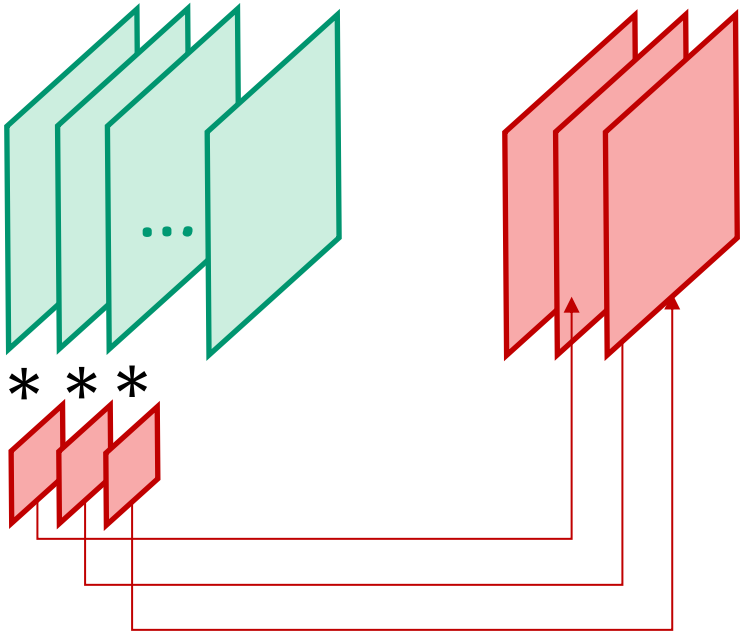
- Keep input channel convolution separate from mapping to output feature map



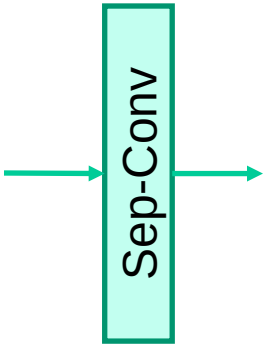
Depthwise/Separable Convolutions



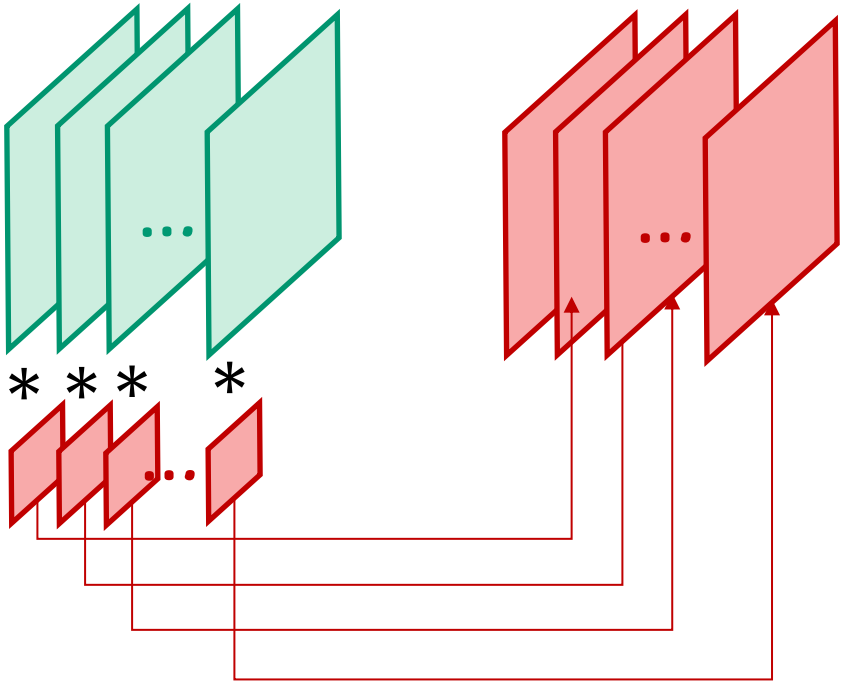
- Keep input channel convolution separate from mapping to output feature map



Depthwise/Separable Convolutions

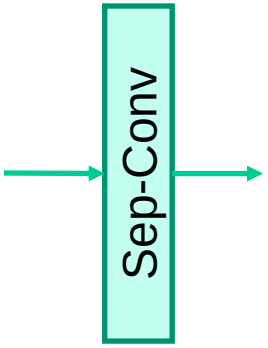


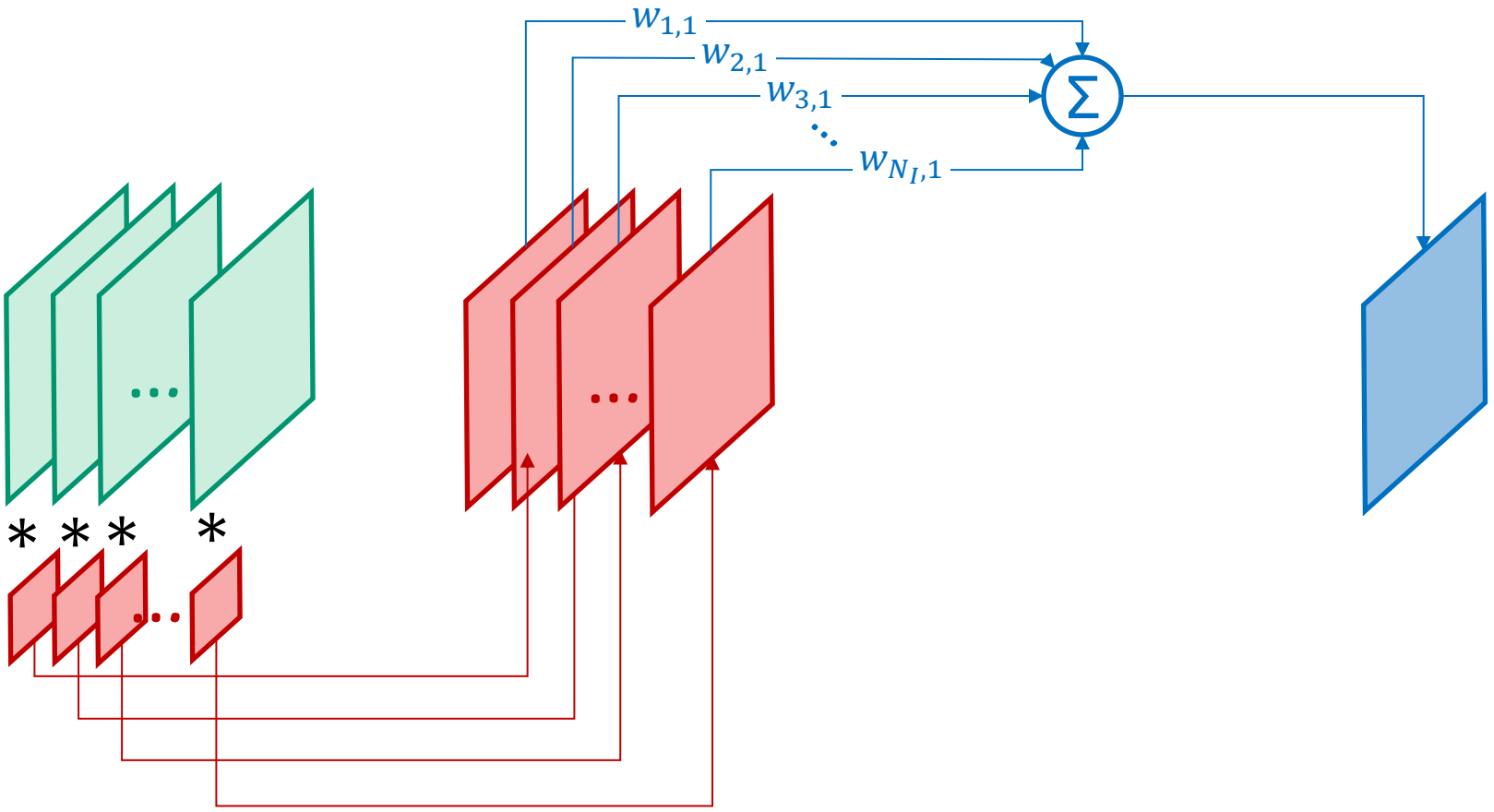
- Keep input channel convolution separate from mapping to output feature map



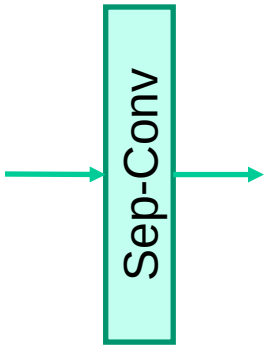
Depthwise/Separable Convolutions

- Keep input channel convolution separate from mapping to output feature map

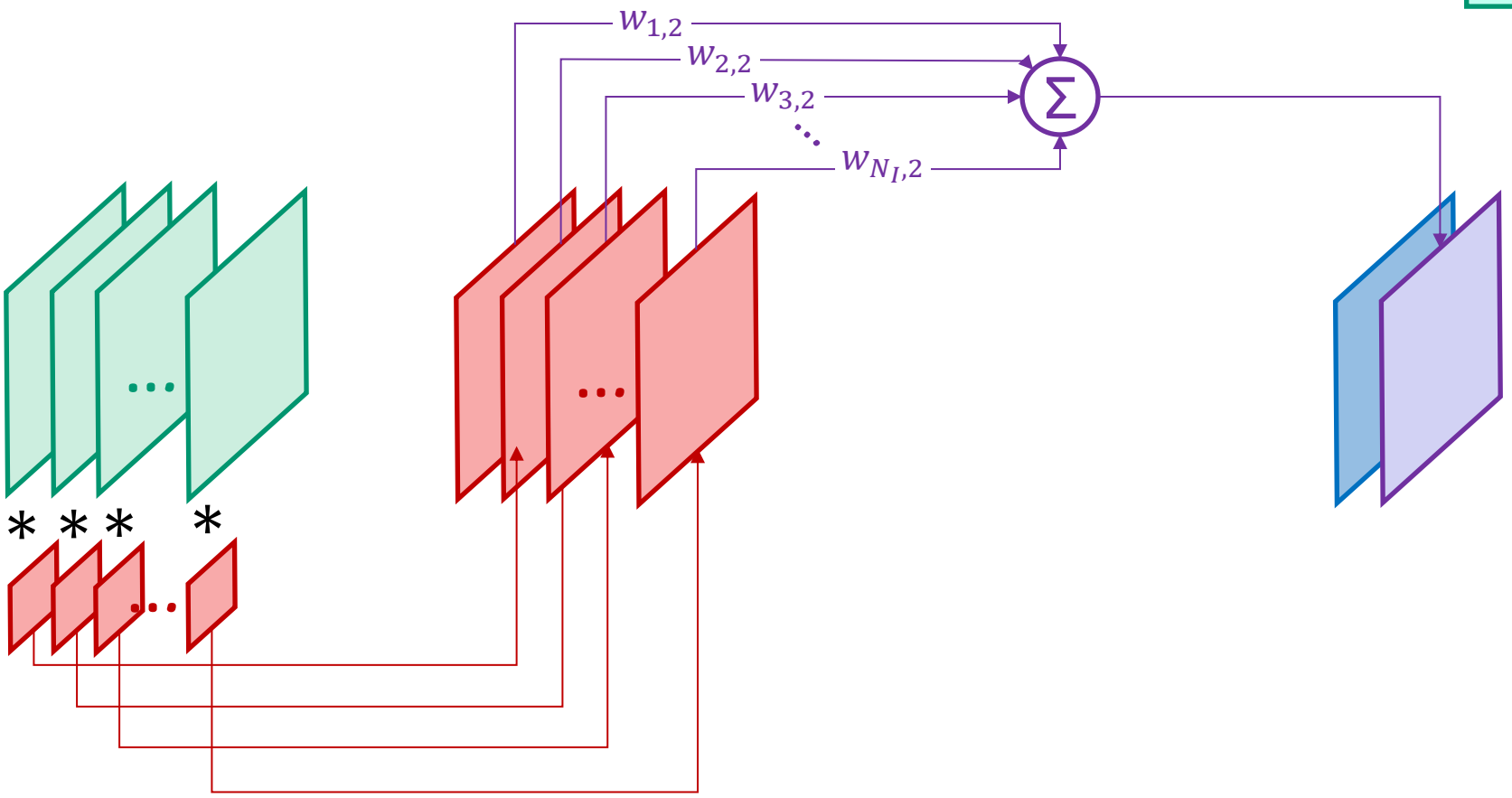




Depthwise/Separable Convolutions

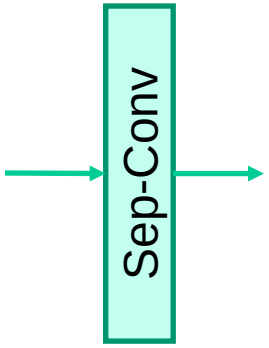


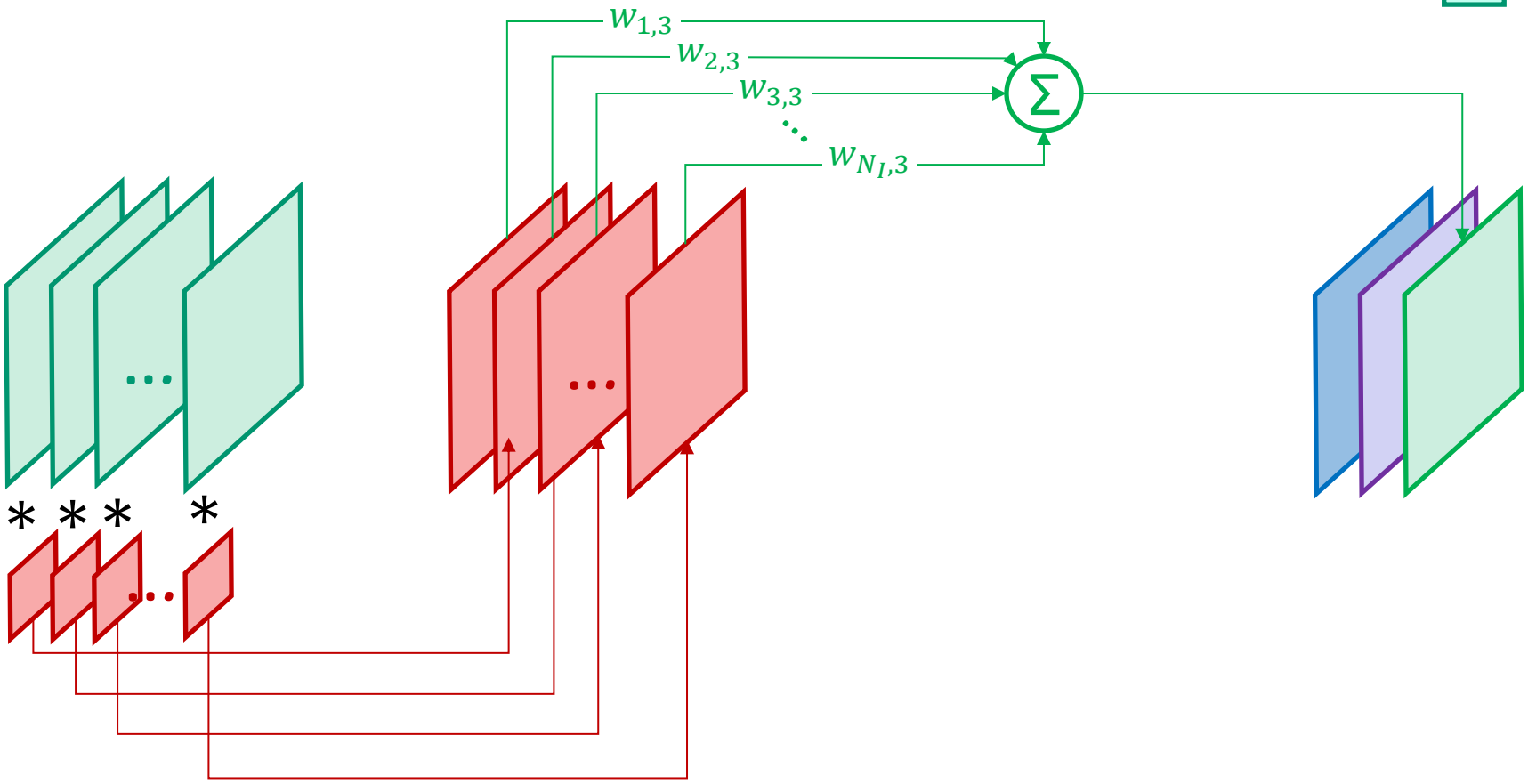
- Keep input channel convolution separate from mapping to output feature map



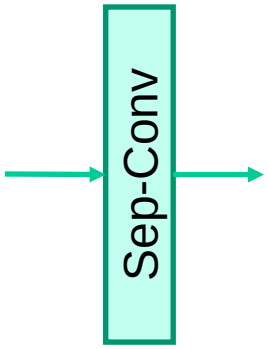
Depthwise/Separable Convolutions

- Keep input channel convolution separate from mapping to output feature map

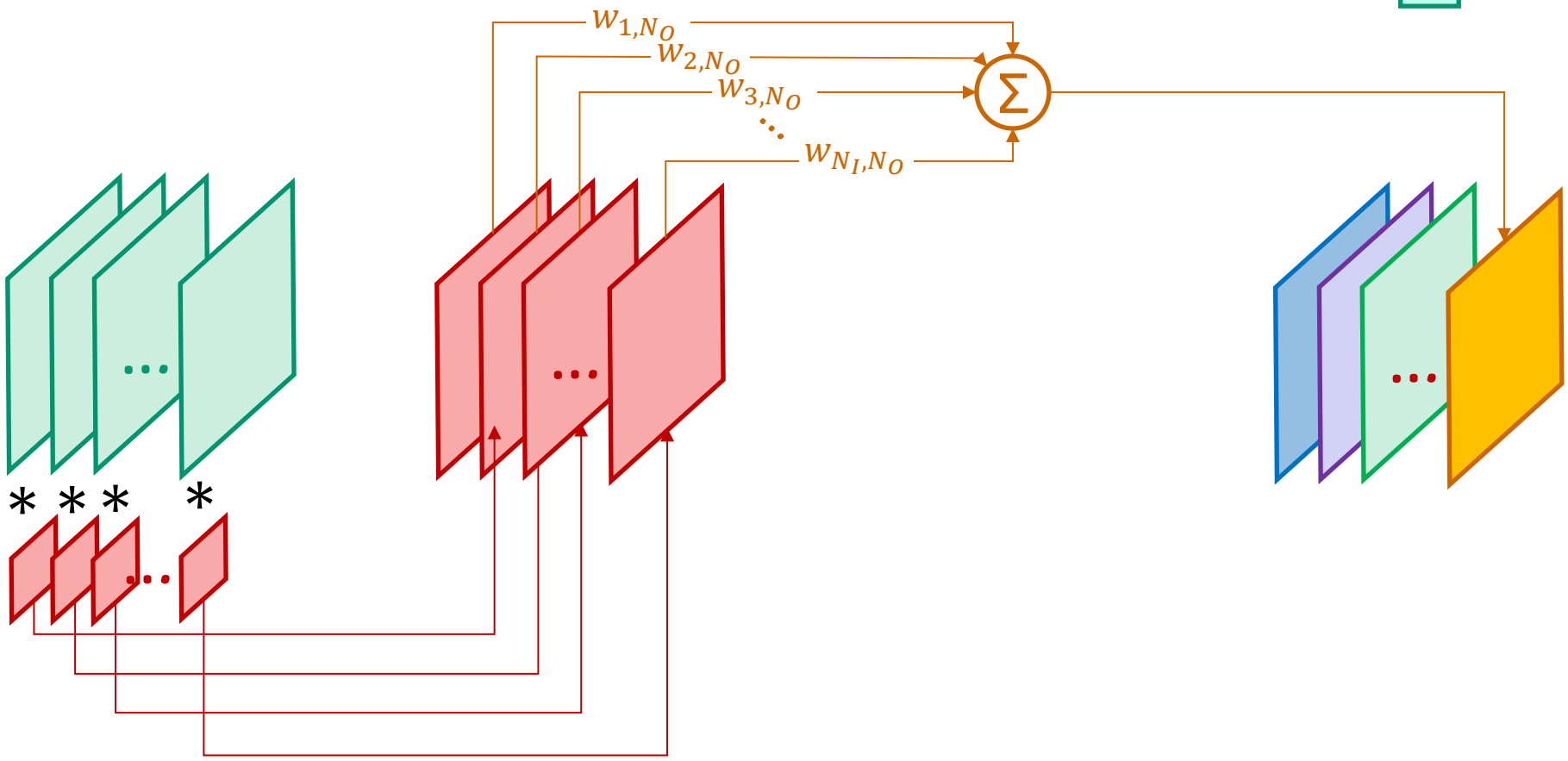




Depthwise/Separable Convolutions

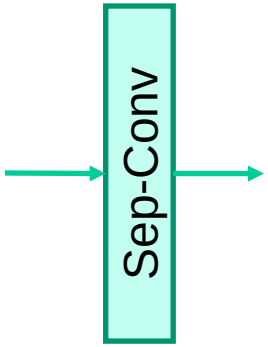


- Keep input channel convolution separate from mapping to output feature map

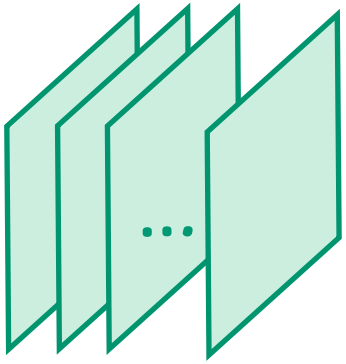


Depthwise/Separable Convolutions

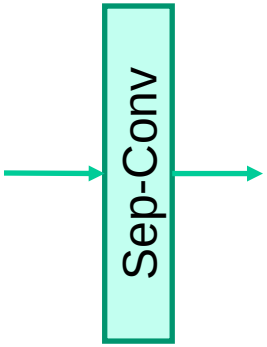
- Keep input channel convolution separate from mapping to output feature map



$\boldsymbol{F^I} \in \mathbb{R}^{\boldsymbol{H \times W \times N_I}}$



Depthwise/Separable Convolutions

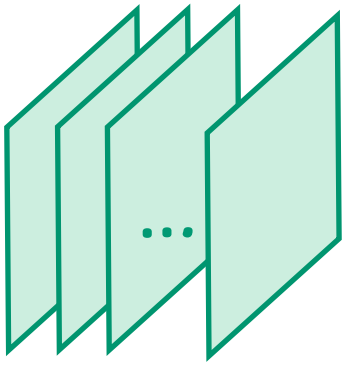


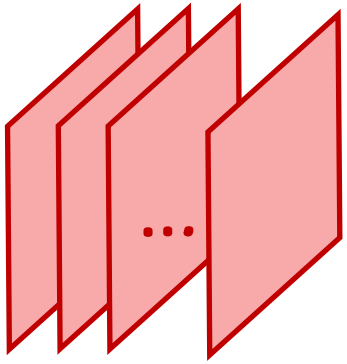
- Keep input channel convolution separate from mapping to output feature map

$\overline{\boldsymbol{K}} \in \mathbb{R}^{\boldsymbol{h}\times\boldsymbol{w}\times\boldsymbol{1}\times\boldsymbol{N_I}}$

$\boldsymbol{F^I} \in \mathbb{R}^{\boldsymbol{H}\times\boldsymbol{W}\times\boldsymbol{N_I}}$

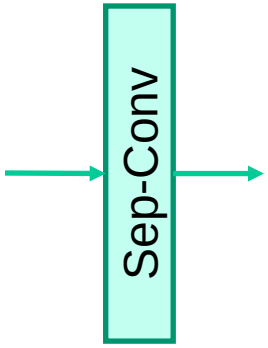
$\overline{\boldsymbol{F}}^{\boldsymbol{I}} \in \mathbb{R}^{\boldsymbol{H'}\times\boldsymbol{W'}\times\boldsymbol{N_I}}$

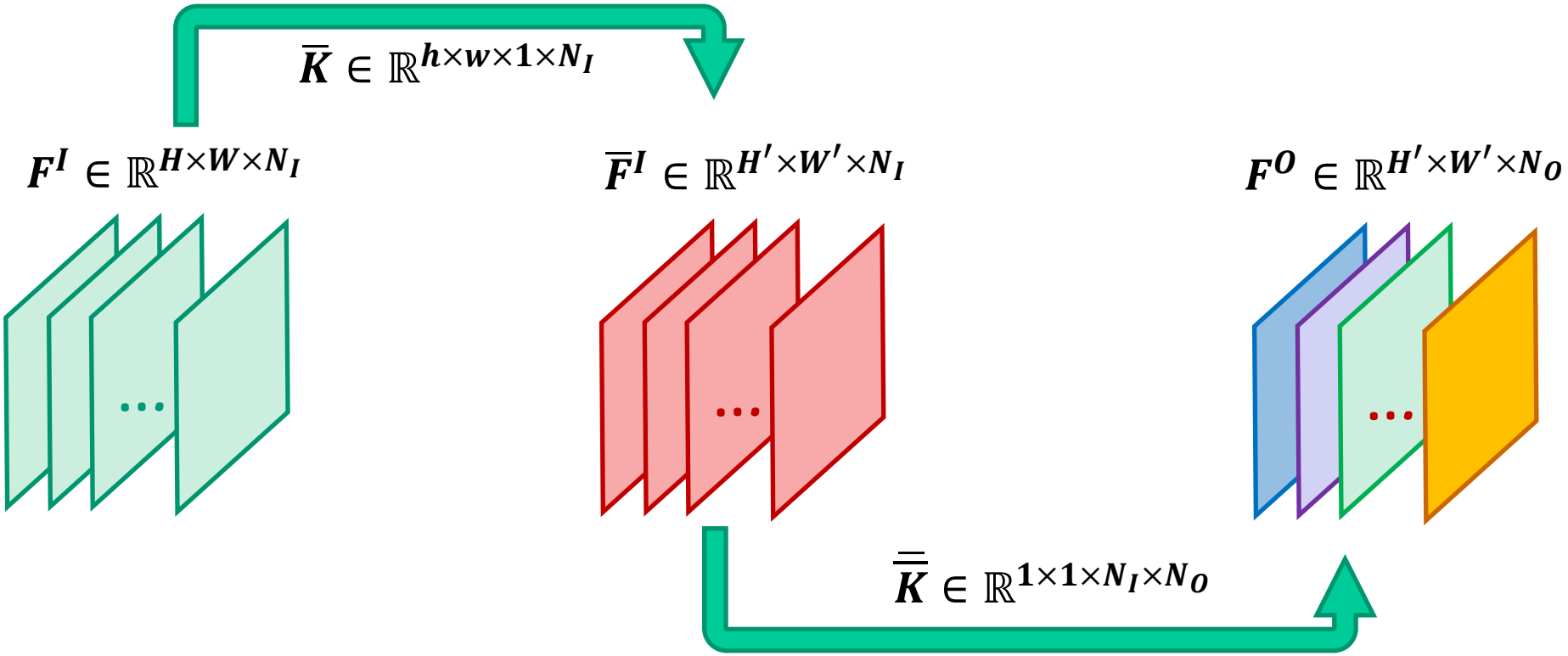




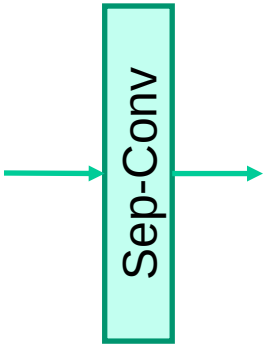
Depthwise/Separable Convolutions

- Keep input channel convolution separate from mapping to output feature map





Depthwise/Separable Convolutions



- What are the benefits?
- Standard convolution has the computation cost of

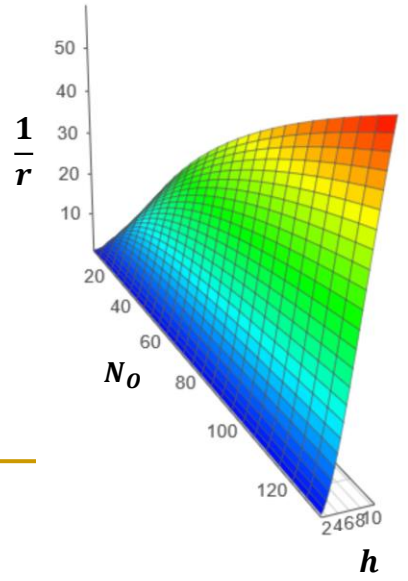
$$\underbrace{O(h \cdot w \cdot N_I \cdot N_O}_{\text{Convolution Kernel}} \cdot \underbrace{H \cdot W)}_{\text{Input Feature Map}}$$

- Depthwise-separable convolution has the computation cost of

$$\underbrace{O(h \cdot w \cdot N_I}_{\text{Conv Kernel-1}} \cdot \underbrace{H \cdot W}_{\text{Input Feature}} + \underbrace{N_I \cdot N_O}_{\text{Conv Kernel-2}} \cdot \underbrace{H \cdot W)}_{\text{Input Feature}}$$

- Reduction in computation ratio

$$r = \frac{hwN_IHW + N_IN_OHW}{hwN_IN_OHW} = \frac{1}{N_O} + \frac{1}{hw}$$



Depthwise/Separable Convolutions

Sep-Conv

- Activation and Batch-Normalization are used in between
- 1x1 convolution also referred by *Pointwise Convolution*

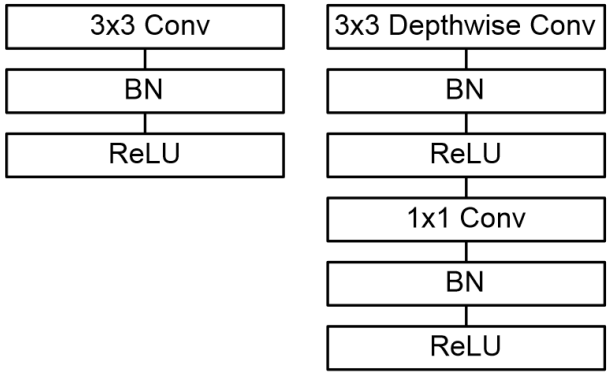


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

MobileNet by Depthwise Separable Convolutions

- All layers are followed by BN and ReLU
- MobileNet has 28 separate layers

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

MobileNet by Depthwise Separable Convolutions

- Significant parameter reduction while maintaining performance accuracy

Table 12. Face attribute classification using the MobileNet architecture. Each row corresponds to a different hyper-parameter setting (width multiplier α and image resolution).

Width Multiplier / Resolution	Mean AP	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	88.7%	568	3.2
0.5 MobileNet-224	88.1%	149	0.8
0.25 MobileNet-224	87.2%	45	0.2
1.0 MobileNet-128	88.1%	185	3.2
0.5 MobileNet-128	87.7%	48	0.8
0.25 MobileNet-128	86.4%	15	0.2
Baseline	86.9%	1600	7.5

Squeeze-Excitation Block

- The idea is to boost the representation power of the network
- **SE:** channel relationships are adaptively recalibrated in channel-wise feature response by explicit modelling of interdependencies between channels

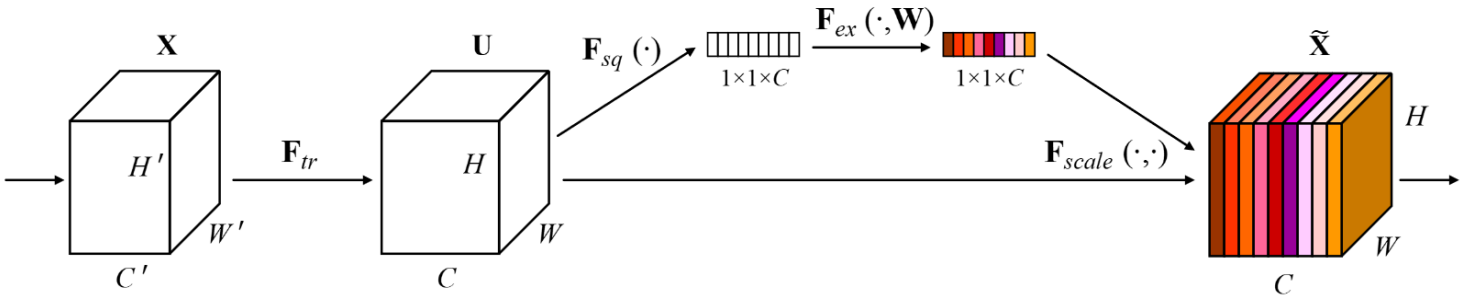


Figure 1: A Squeeze-and-Excitation block.

Squeeze-Excitation Block

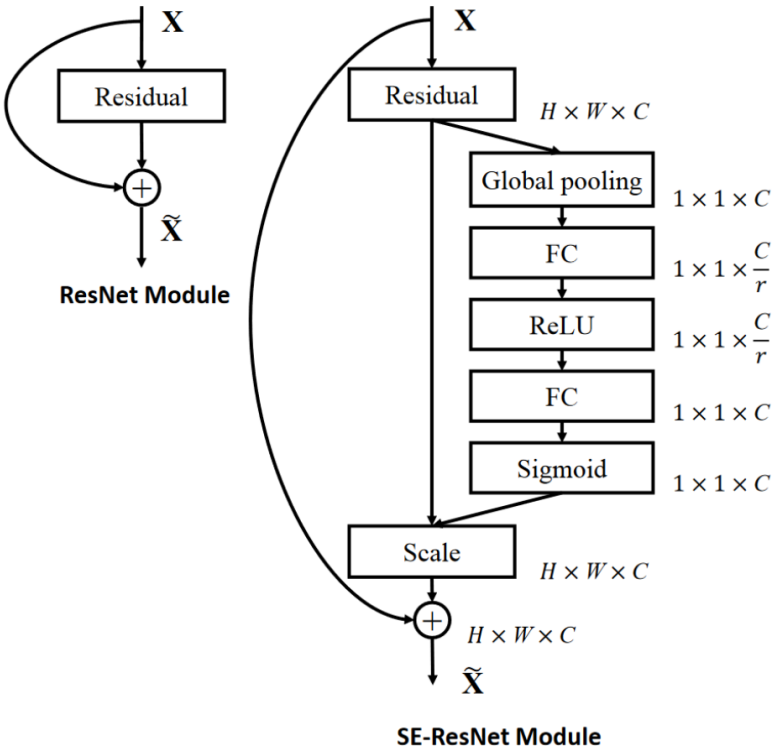


Figure 3: The schema of the original Residual module (left) and the SE-ResNet module (right).