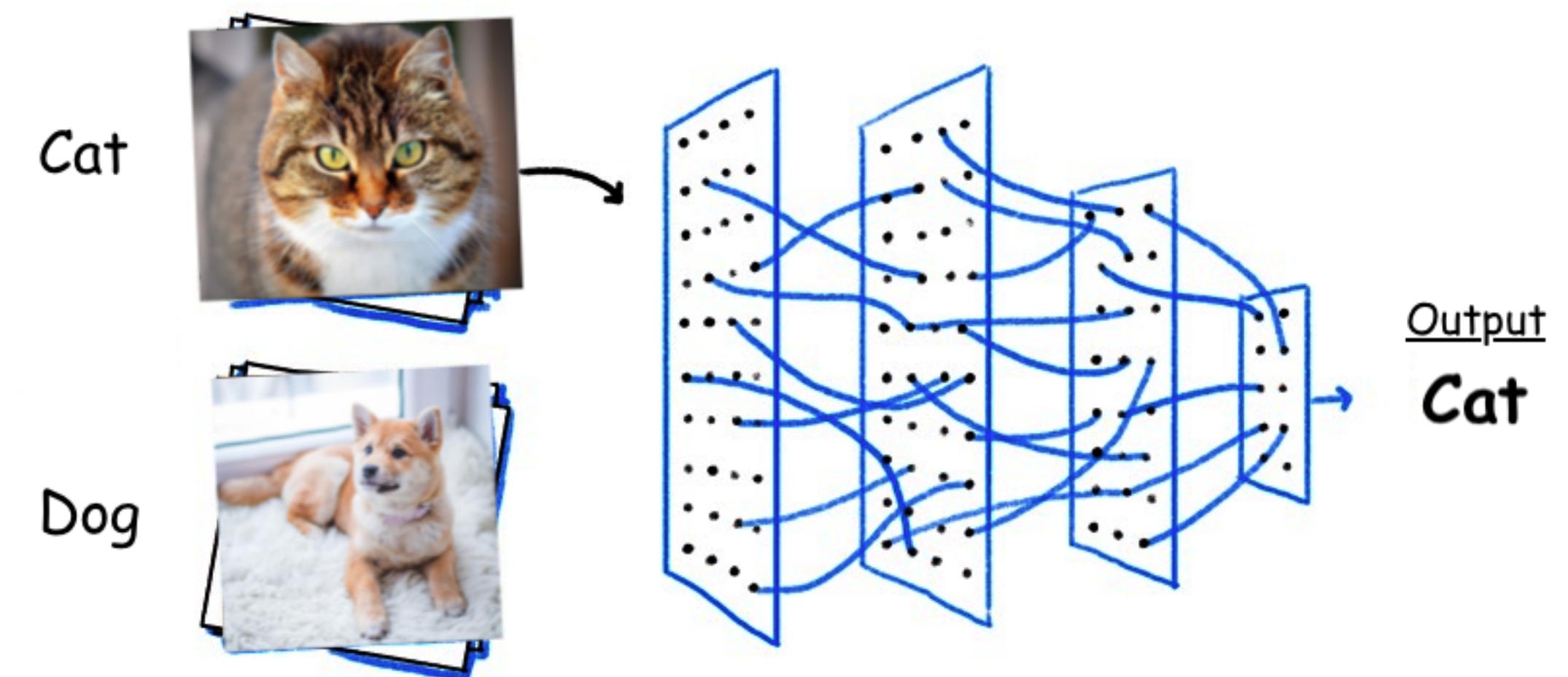
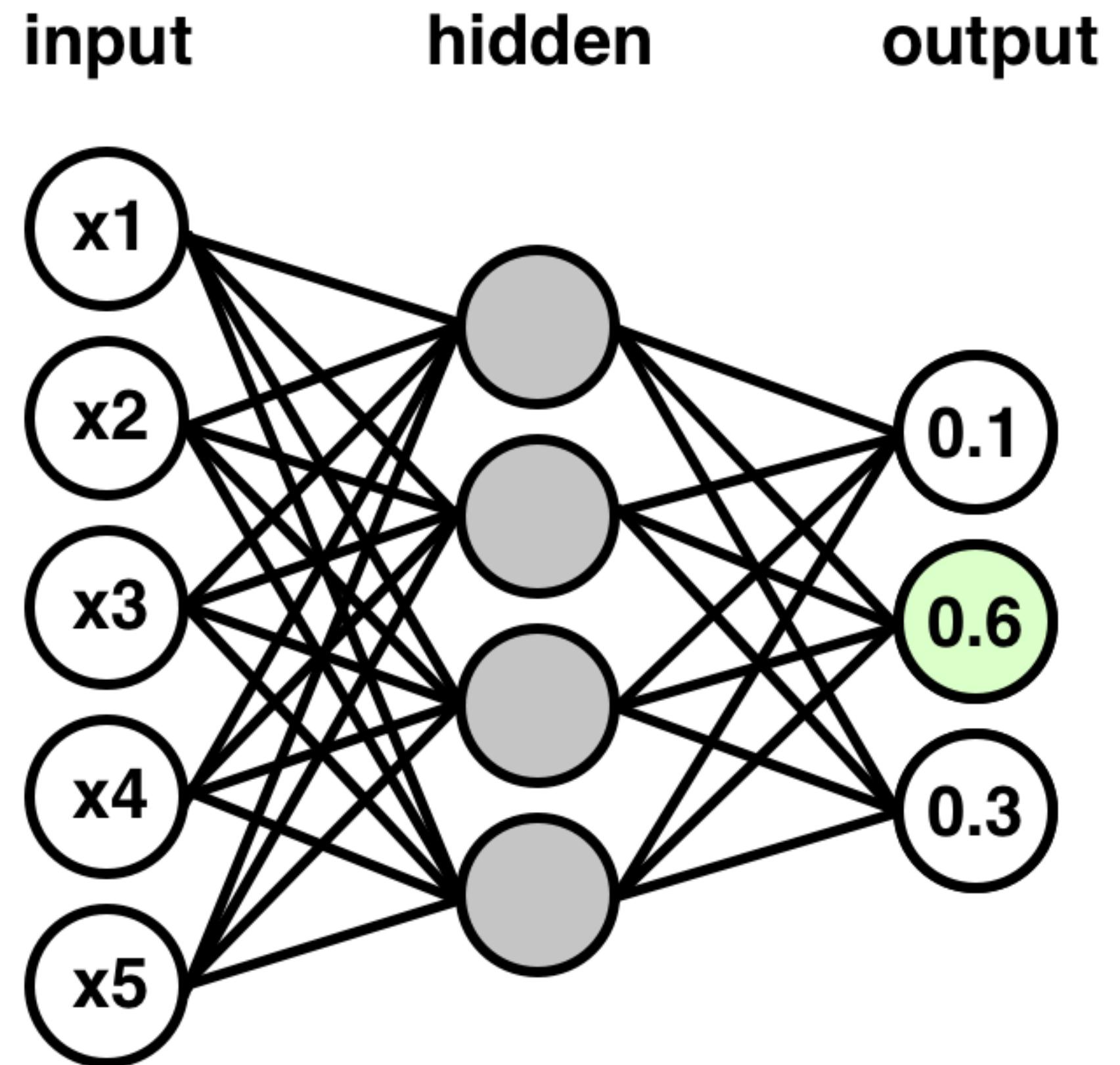




Lesson 10

Introduction to Deep Learning

Neural Network



Common Layers in Deep Learning

Linear layers:

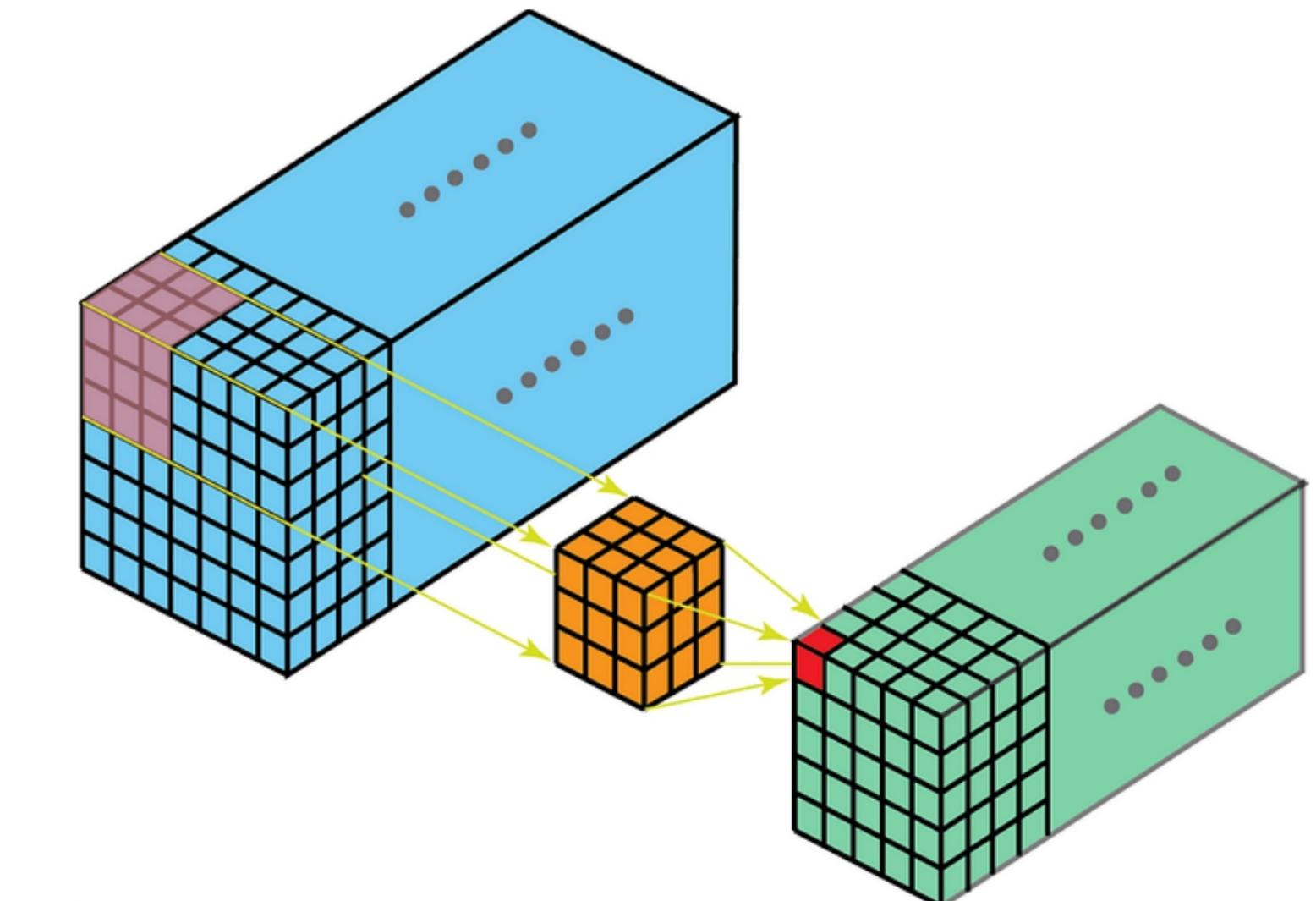
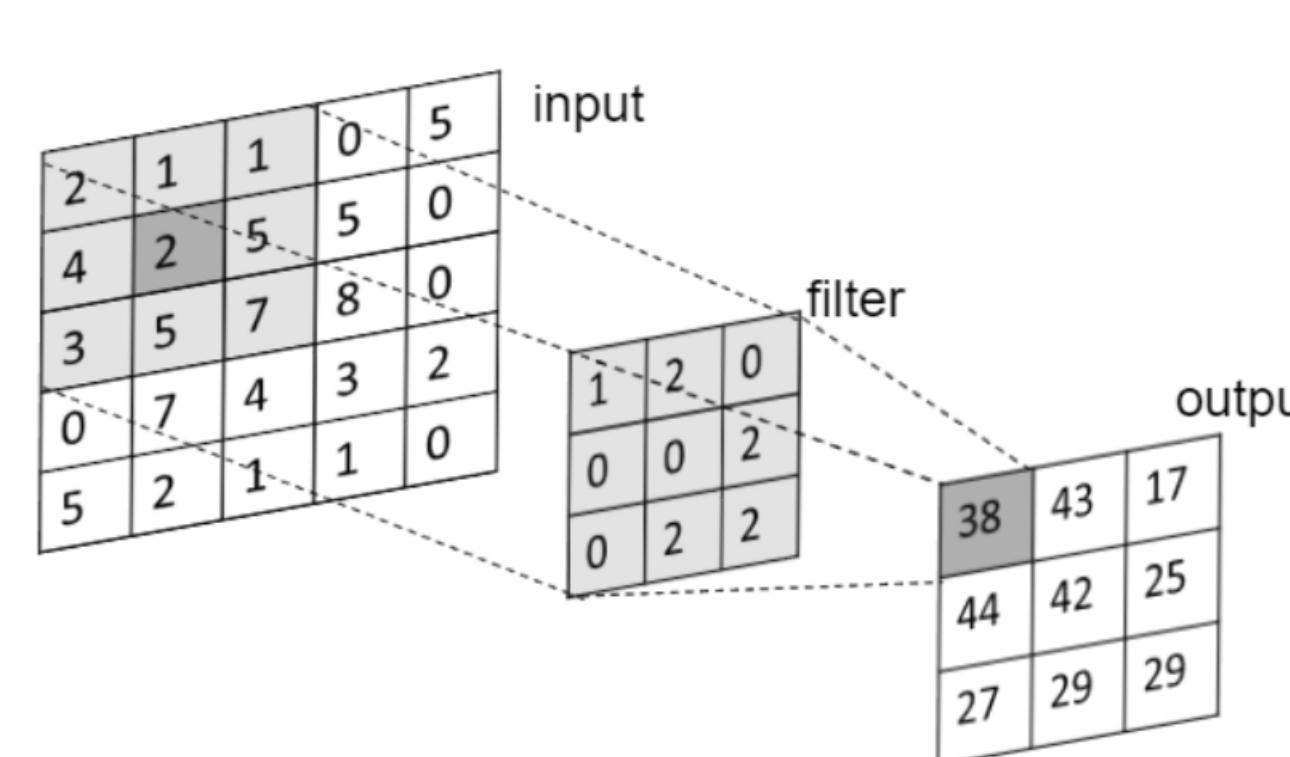
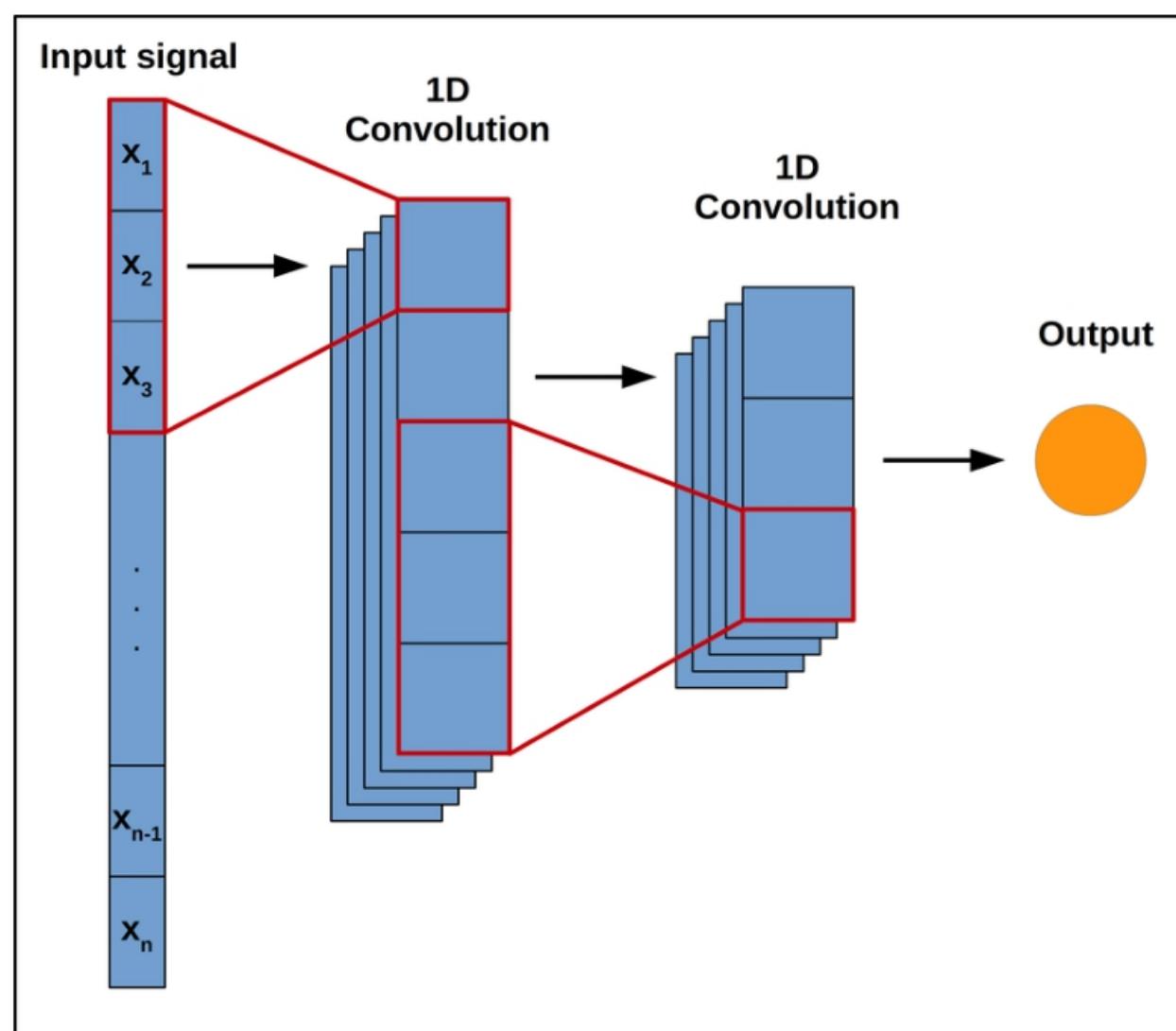
- **nn.Linear** - applies a linear transformation to the incoming data.

$$y = xA^T + b$$

Common Layers in Deep Learning

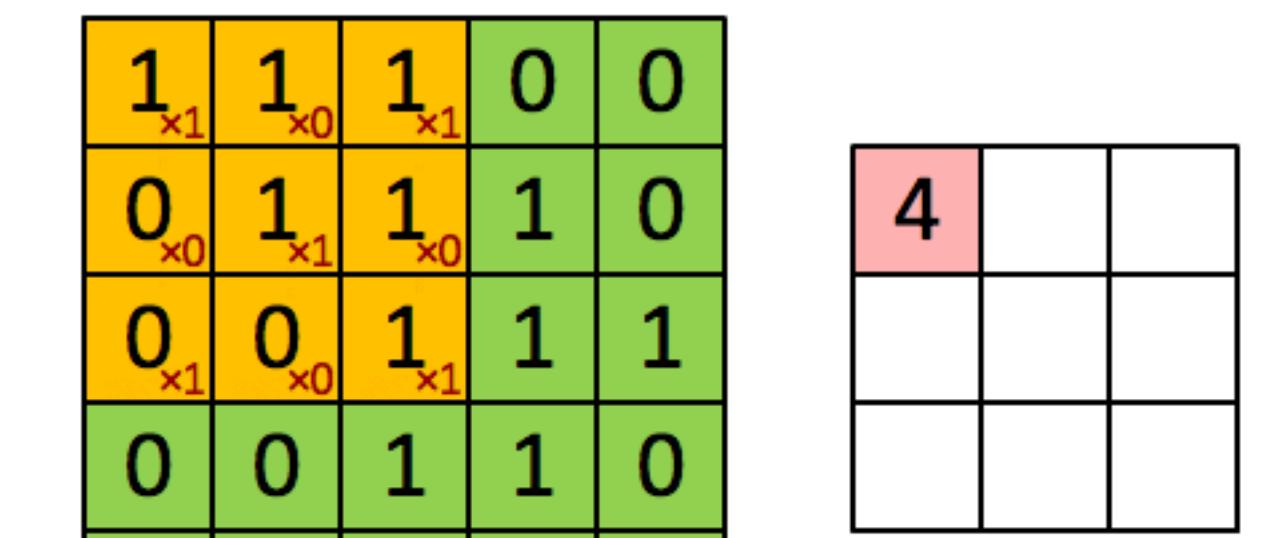
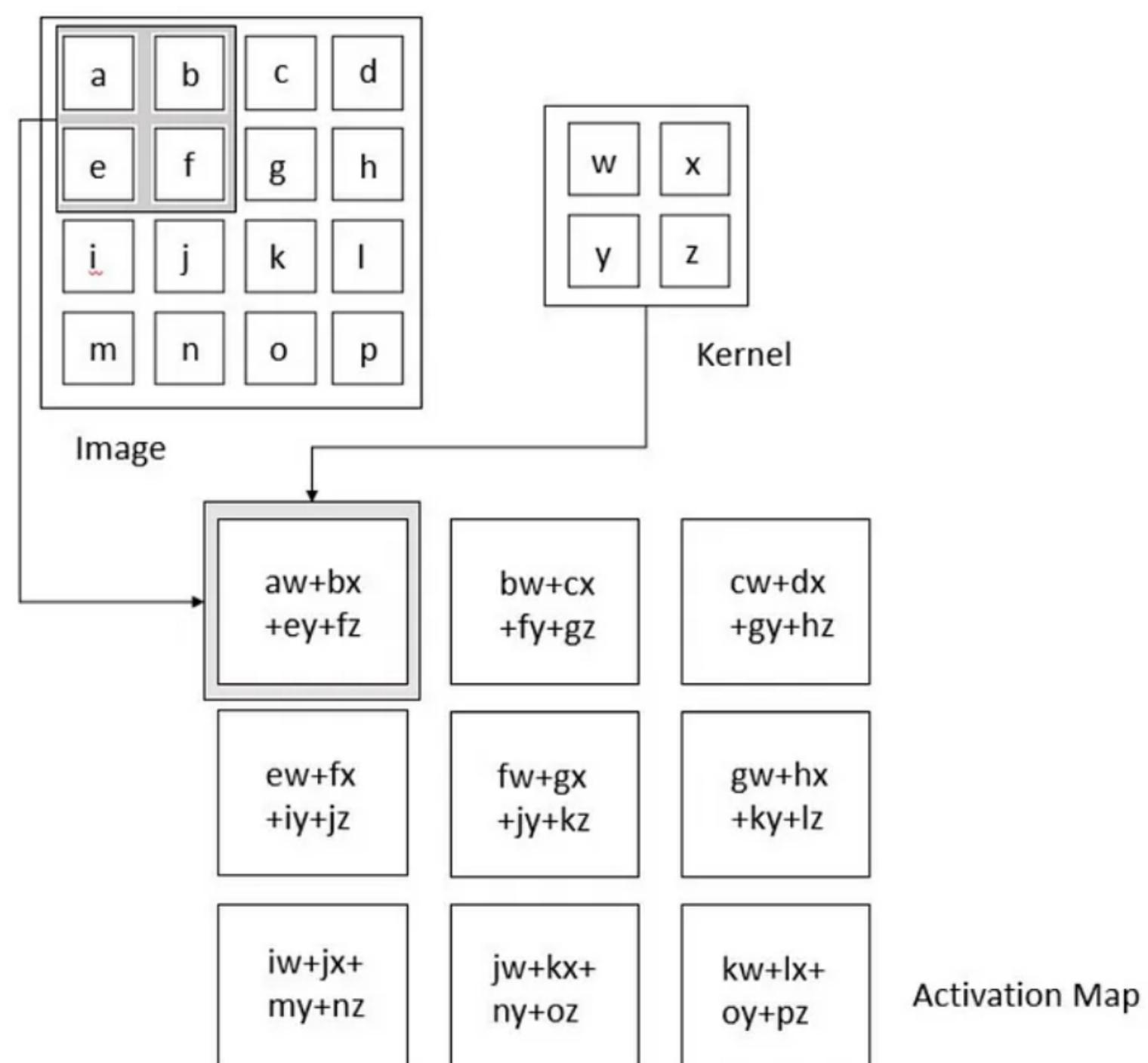
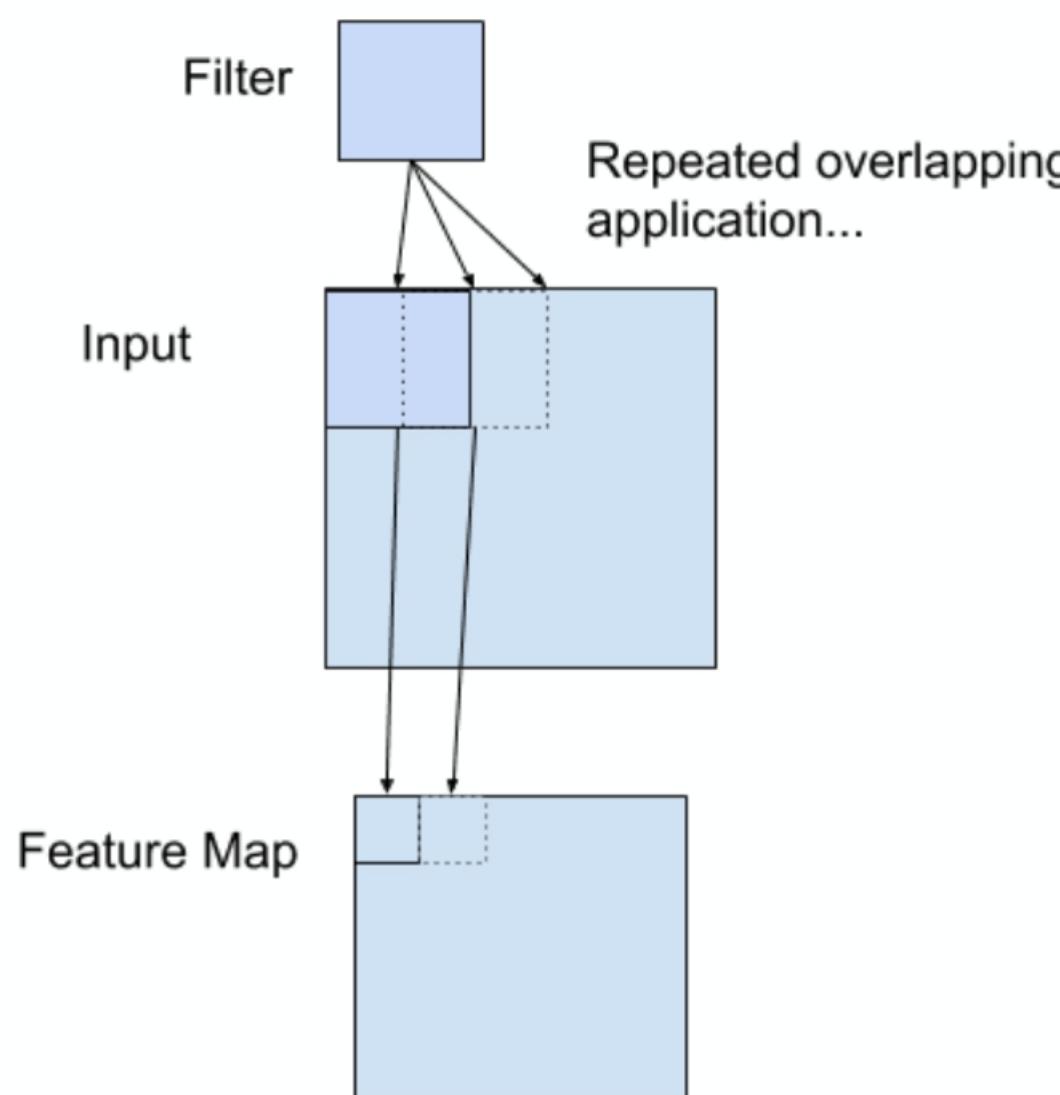
Convolution layers:

- **nn.Conv1d** - applies a 1D convolution over an input signal.
- **nn.Conv2d** - applies a 2D convolution over an input image.
- **nn.Conv3d** - applies a 3D convolution over an input volume.
- **nn.ConvTranspose1d** - applies a 1D transposed convolution.
- **nn.ConvTranspose2d** - applies a 2D transposed convolution.
- **nn.ConvTranspose3d** - applies a 3D transposed convolution.

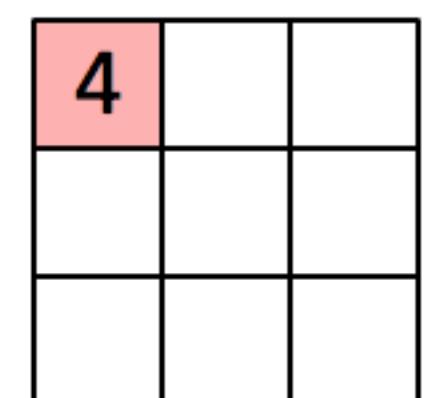


Common Layers in Deep Learning

Math behind convolution layer:



Activation Map



Convolved Feature

Common Layers in Deep Learning

Math behind convolution layer:

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
0	155	153	153	158	168	...
...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...

Input Channel #3 (Blue)

-1	-1	1
0	1	-1
0	1	1

Kernel Channel #1



308

1	0	0
1	-1	-1
1	0	-1

Kernel Channel #2



-498

0	1	1
0	1	0
1	-1	1

Kernel Channel #3



164

$$+ 1 = -25$$

Bias = 1
↑

-25				...
				...
				...
				...
...

Common Layers in Deep Learning

Convolution layer parameters:

- number of input channels
- number of output channels
- kernel size
- step size (stride)
- padding & padding mode
- bias
- dilation*
- groups*

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

n_{in} : number of input features

n_{out} : number of output features

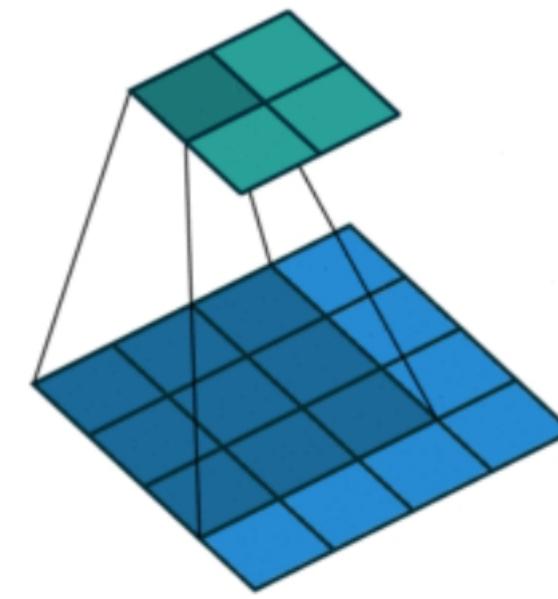
k : convolution kernel size

p : convolution padding size

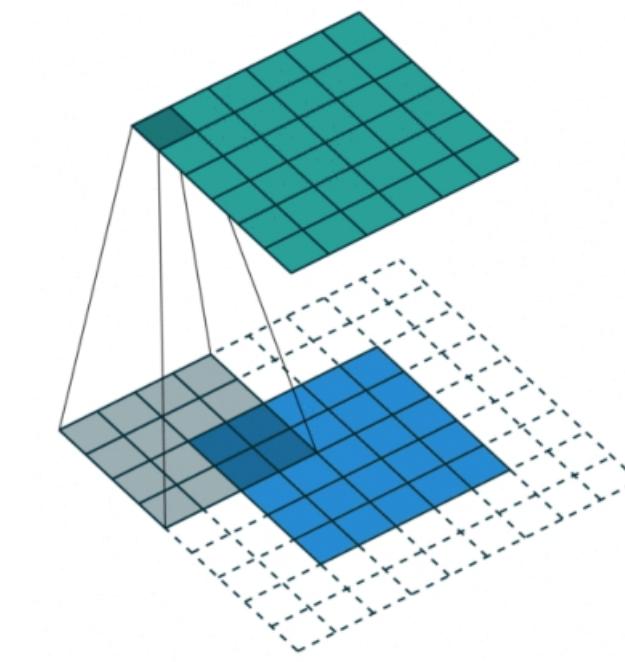
s : convolution stride size

Common Layers in Deep Learning

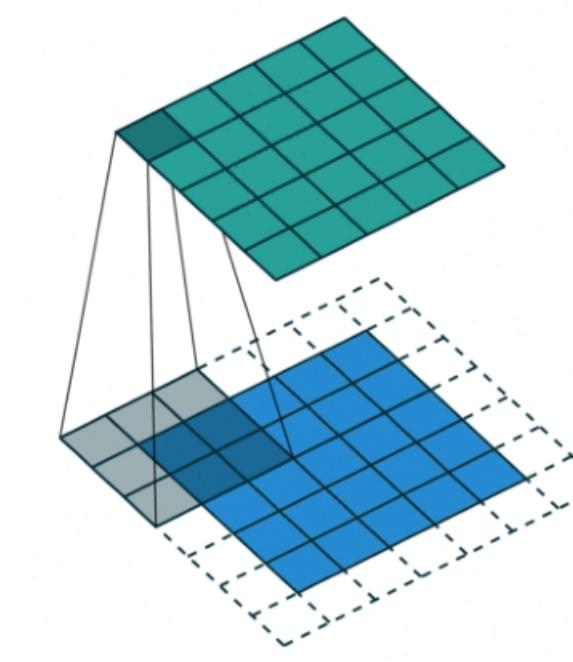
Convolution Arithmetic



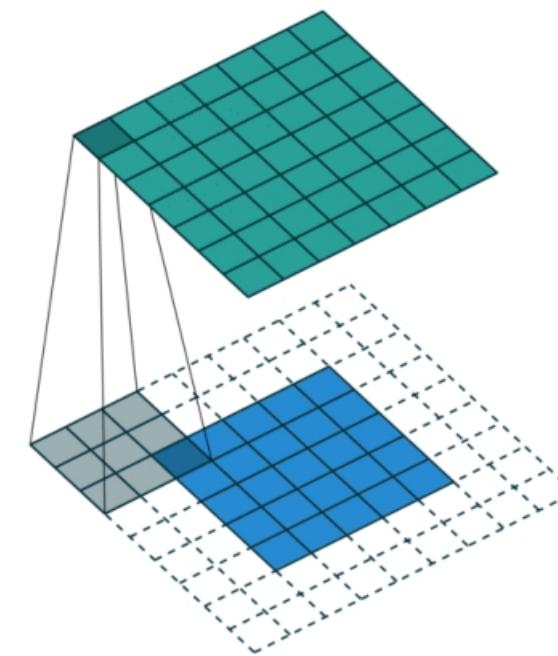
No padding, no strides



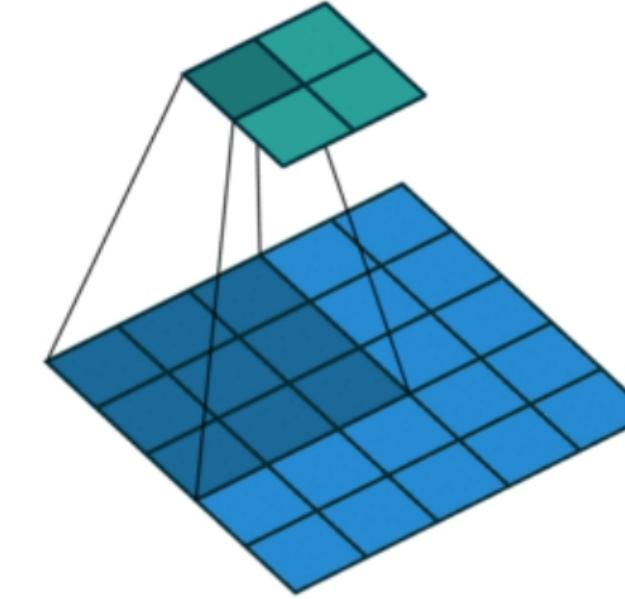
Arbitrary padding, no strides



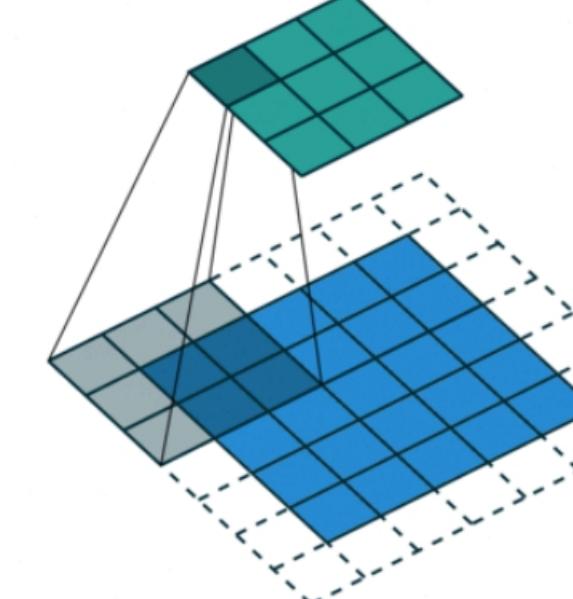
Half padding, no strides



Full padding, no strides



No padding, strides

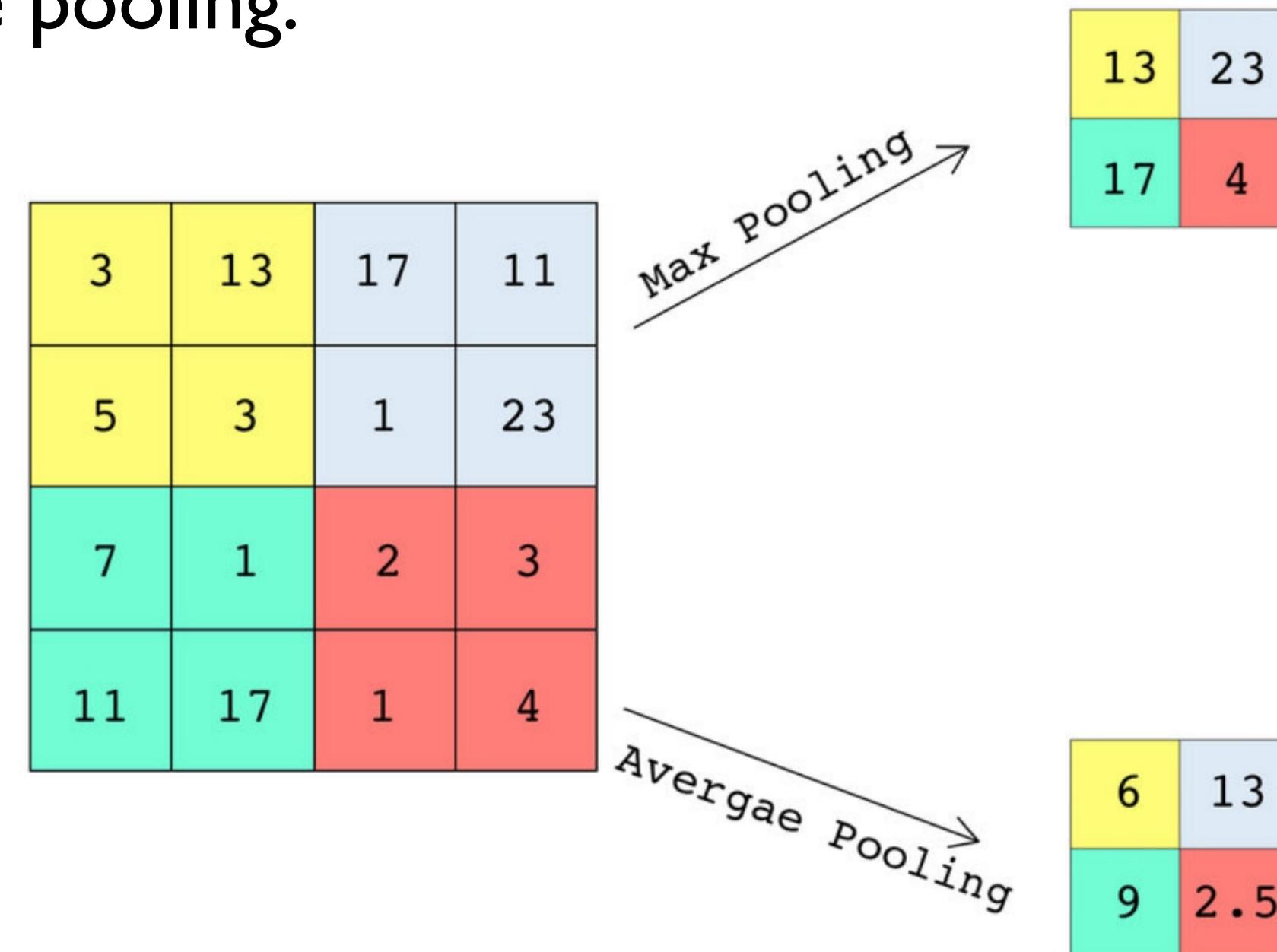


Padding, strides

Common Layers in Deep Learning

Pooling layers:

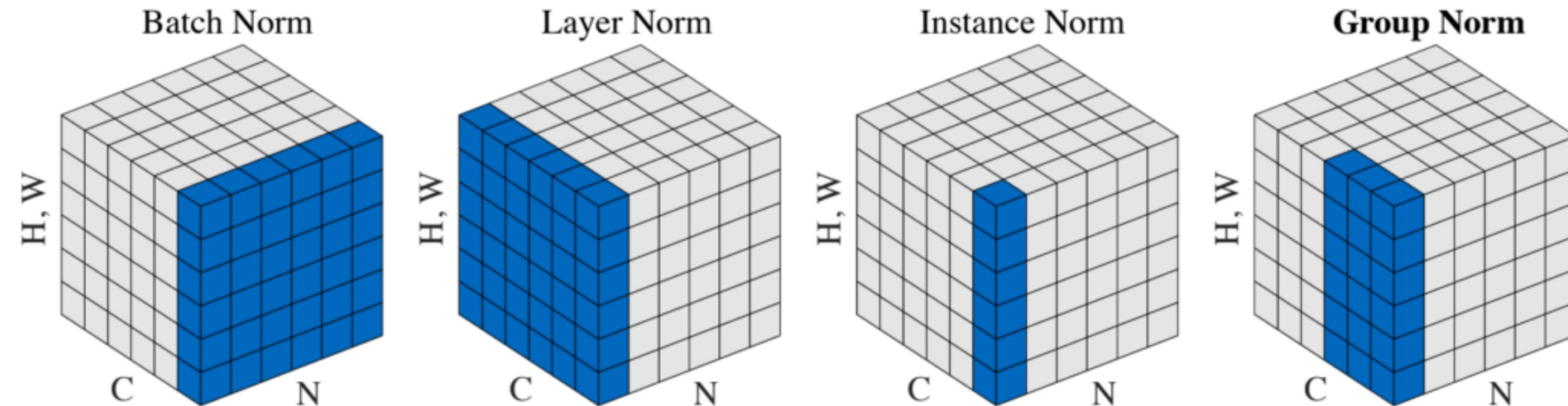
- **nn.MaxPool1d, nn.MaxPool2d, nn.MaxPool3d** - applies 1D/2D/3D max pooling.
- **nn.AvgPool1d, nn.AvgPool2d, nn.AvgPool3d** - applies 1D/2D/3D average pooling.
- **nn.AdaptiveMaxPool1d, nn.AdaptiveMaxPool2d, nn.AdaptiveMaxPool3d** - applies 1D/2D/3D adaptive max pooling.
- **nn.AdaptiveAvgPool1d, nn.AdaptiveAvgPool2d, nn.AdaptiveAvgPool3d** - applies 1D/2D/3D adaptive average pooling.



Common Layers in Deep Learning

Normalization layers:

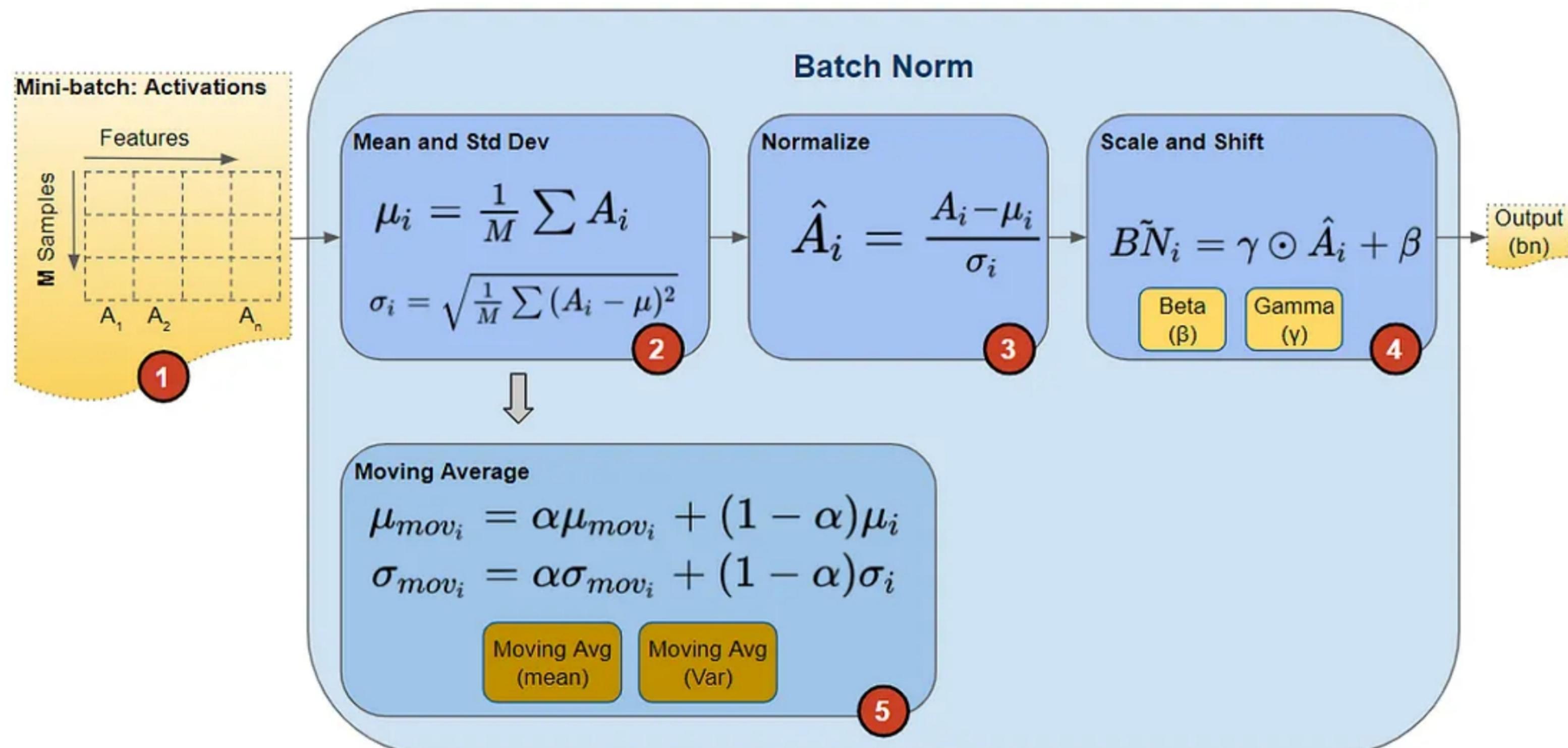
- **nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d** - applies Batch Normalization over 1D/2D/3D input.
- **nn.InstanceNorm1d, nn.InstanceNorm2d, nn.InstanceNorm3d** - applies Instance Normalization over 1D/2D/3D input.
- **nn.LayerNorm** - applies Layer Normalization over an input.
- **nn.GroupNorm** - applies Group Normalization over an input.



Common Layers in Deep Learning

Math behind the layer:

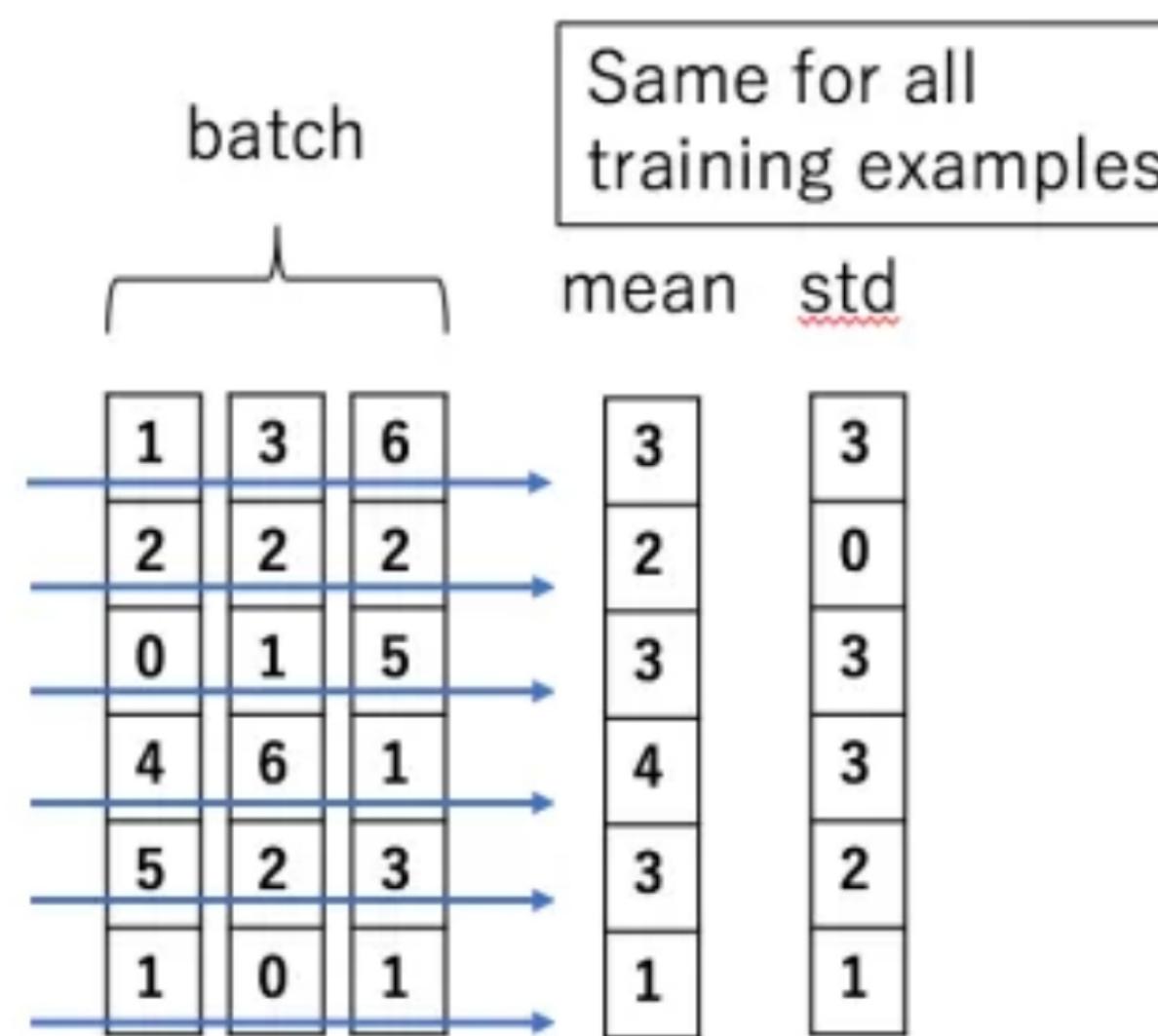
1. Obtain features
2. Calculate Mean and Variance for each feature
3. Normalize features
4. Scale and Shift features
5. Update Exponential Moving Average (used for inference)



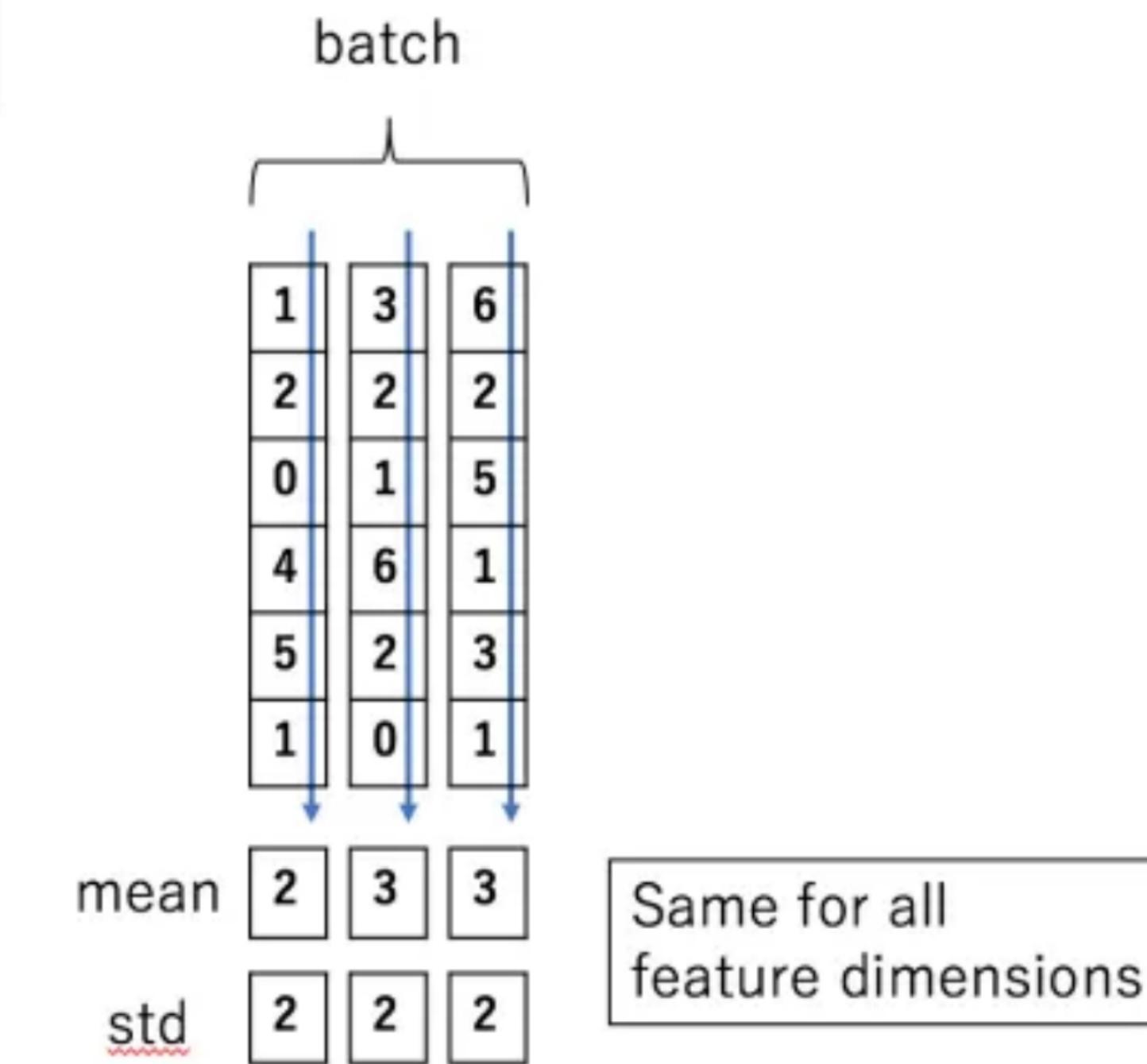
Common Layers in Deep Learning

Normalization layers comparison

Batch Normalization



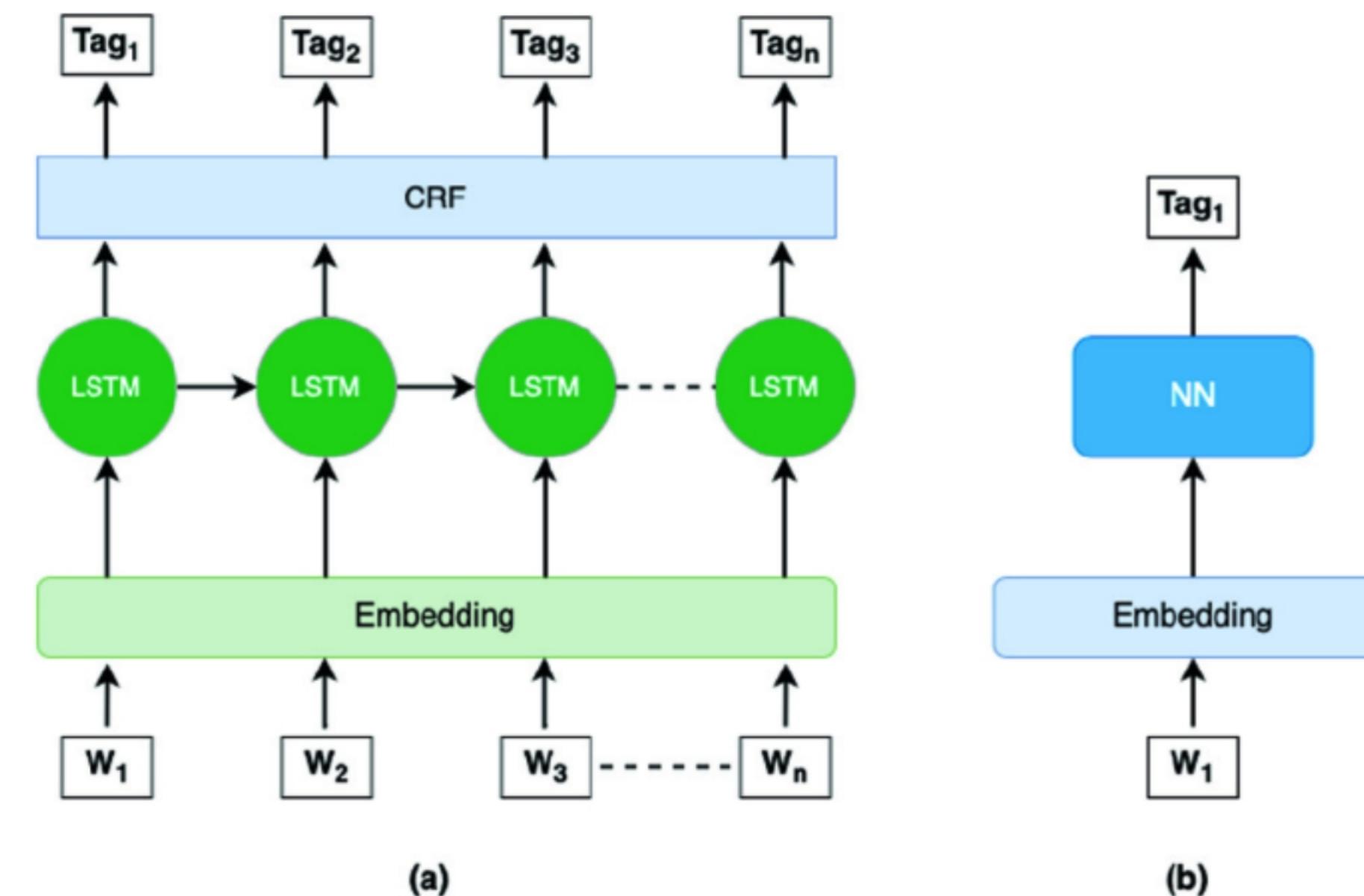
Layer Normalization



Common Layers in Deep Learning

Embedding layers:

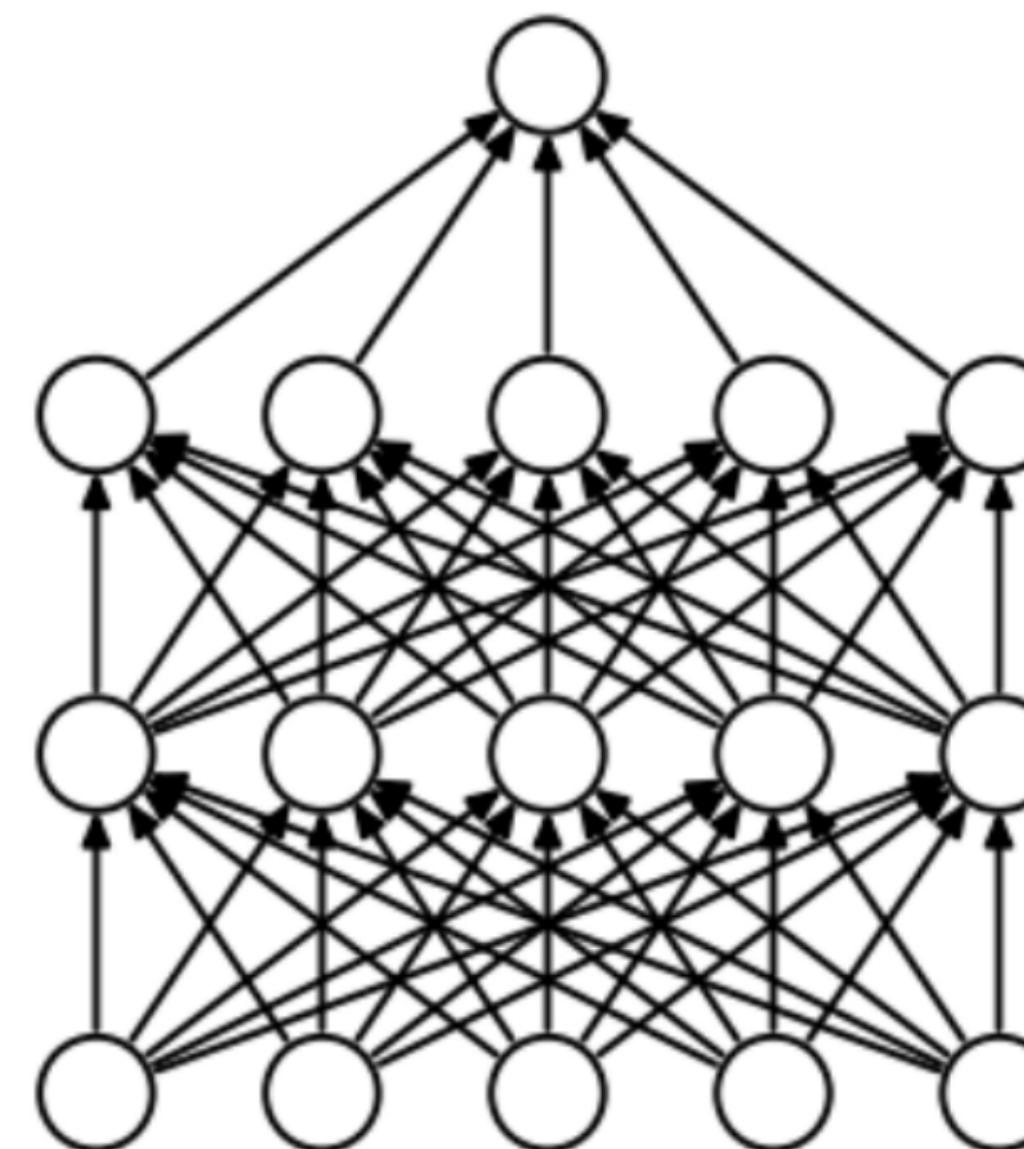
- **nn.Embedding** - a simple lookup table that stores embeddings of a fixed dictionary and size.



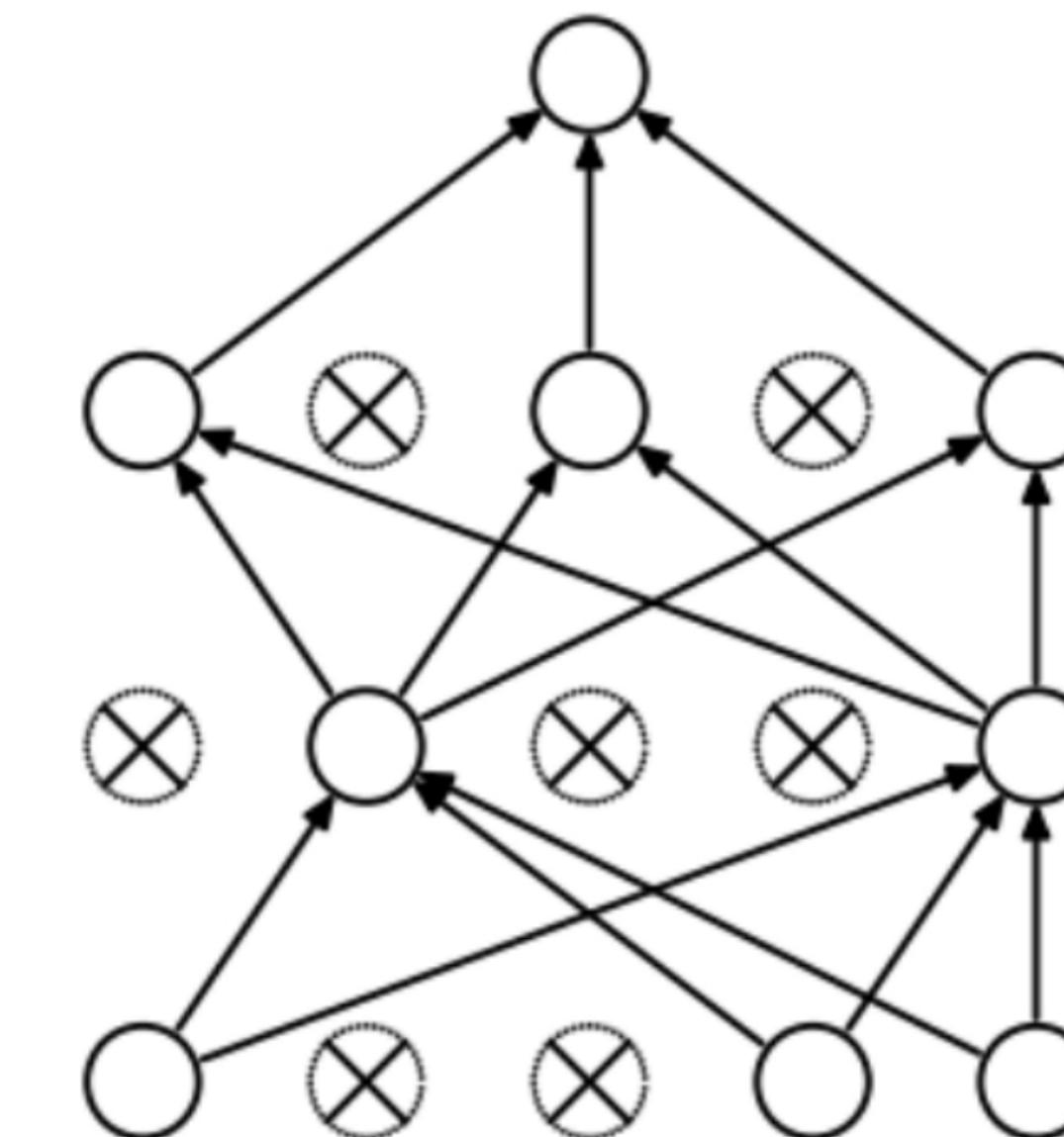
Common Layers in Deep Learning

Dropout layers:

- **nn.Dropout** - randomly zeroes some of the elements with probability p .
- **nn.Dropout2d** - randomly zeroes whole channels of the input tensor.
- **nn.Dropout3d** - randomly zeroes whole 3d features of the input tensor.
- **nn.AlphaDropout** - Alpha Dropout is a type of Dropout that maintains the self-normalizing property.



(a) Standard Neural Net

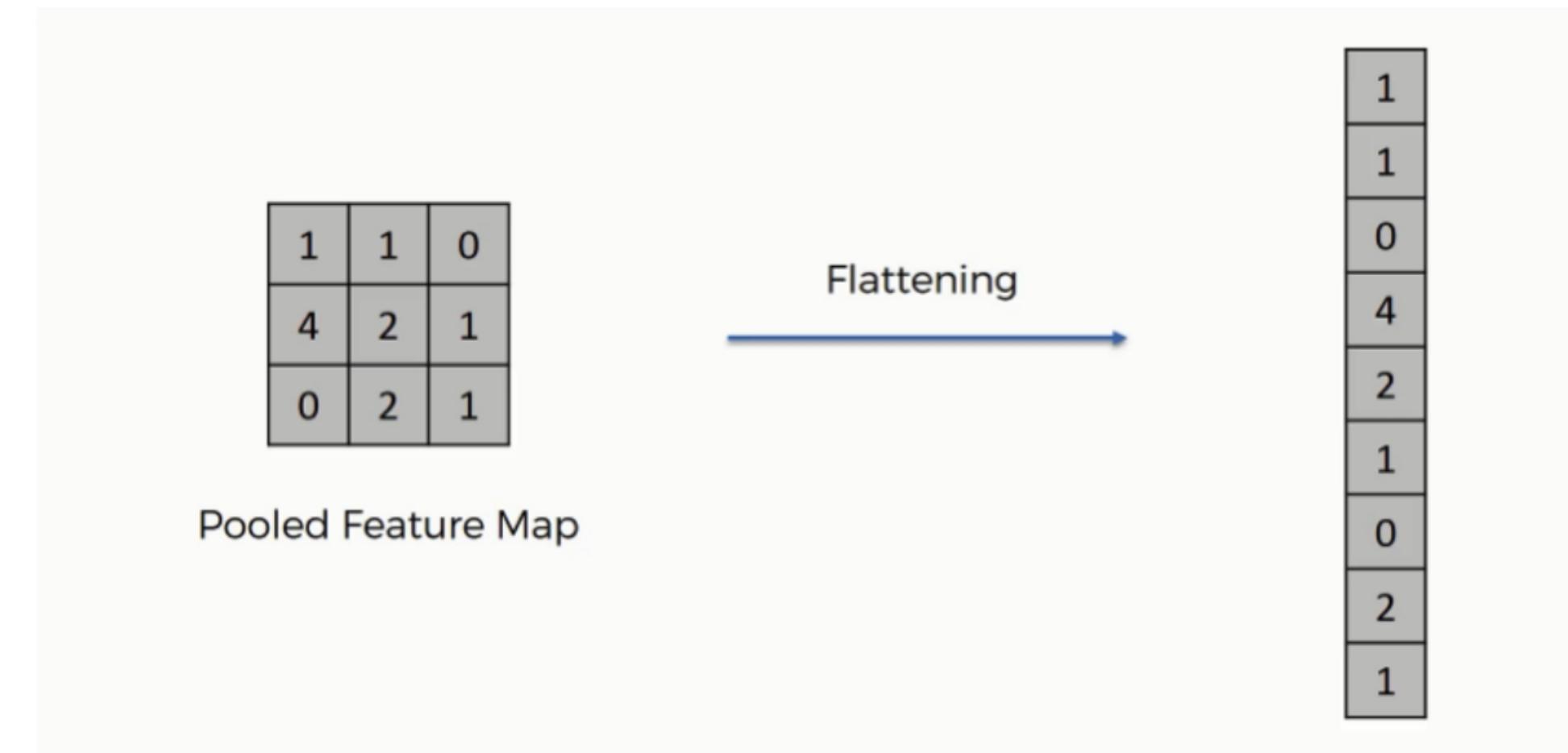


(b) After applying dropout.

Common Layers in Deep Learning

Utility layers:

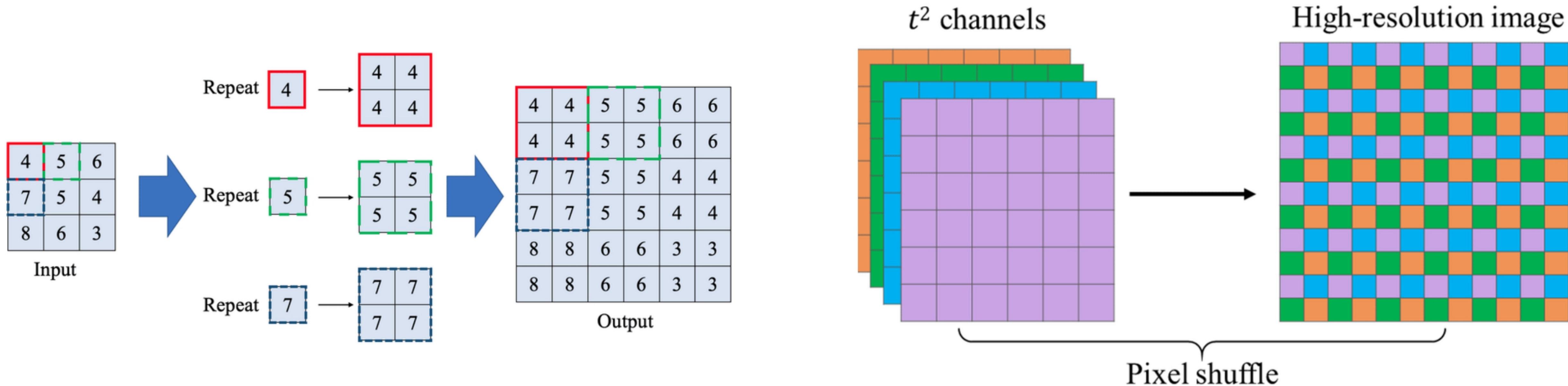
- **nn.Flatten** - flattens a contiguous range of dims into a tensor.
- **nn.Identity** - placeholder identity operator.



Common Layers in Deep Learning

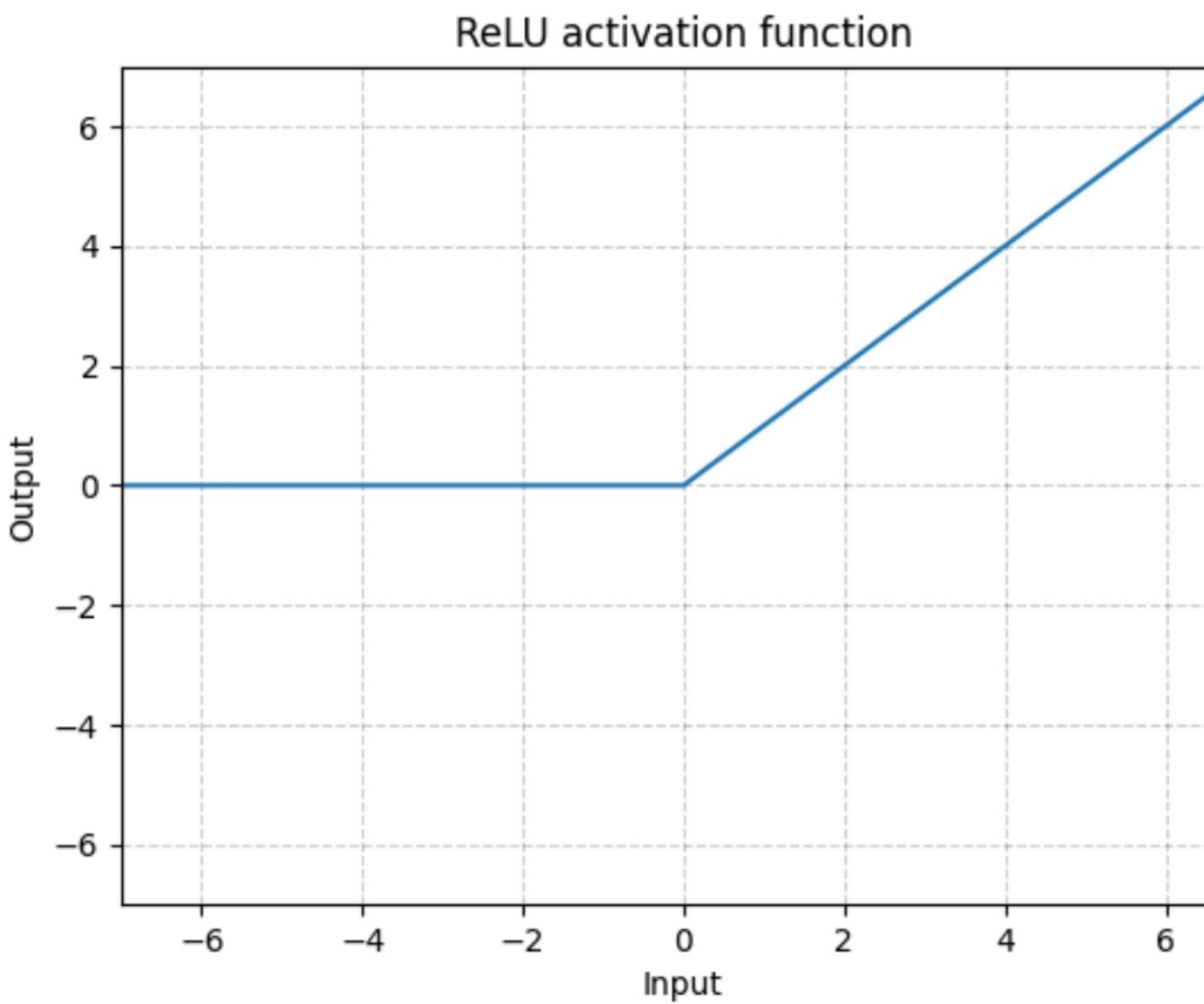
Vision-specificity layers:

- **nn.Upsample** - upsamples a given multi-channels 1D, 2D, or 3D data.
- **nn.PixelShuffle** - used for super-resolution and other tasks that require upsampling.



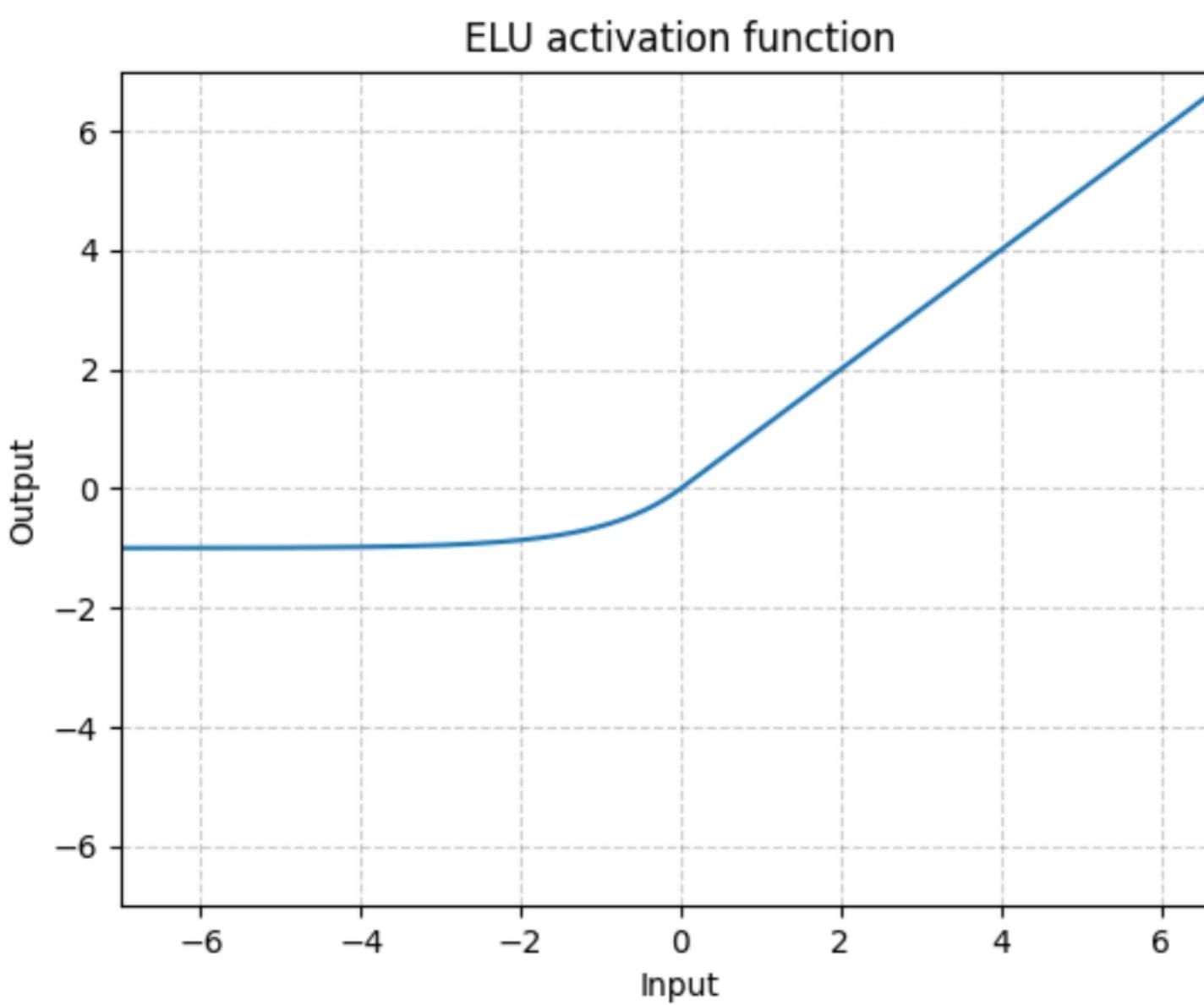
Activation Layers in Deep Learning

$$ReLU(x) = (x)^+ = \max(0, x)$$



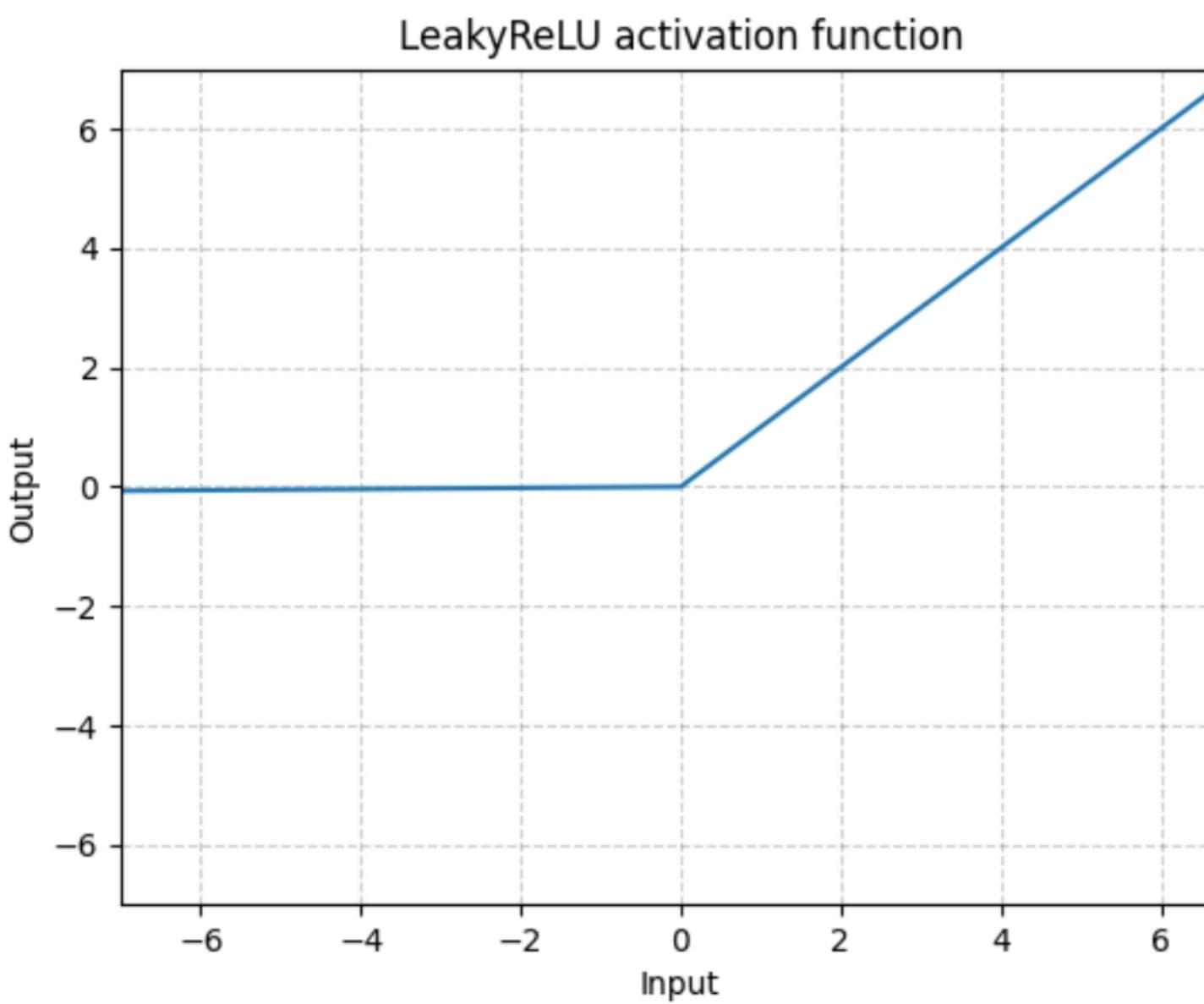
Activation Layers in Deep Learning

$$ELU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$



Activation Layers in Deep Learning

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \text{negative slope} * x & \text{otherwise} \end{cases} = \max(0, x) + \text{negative slope} * \min(0, x)$$

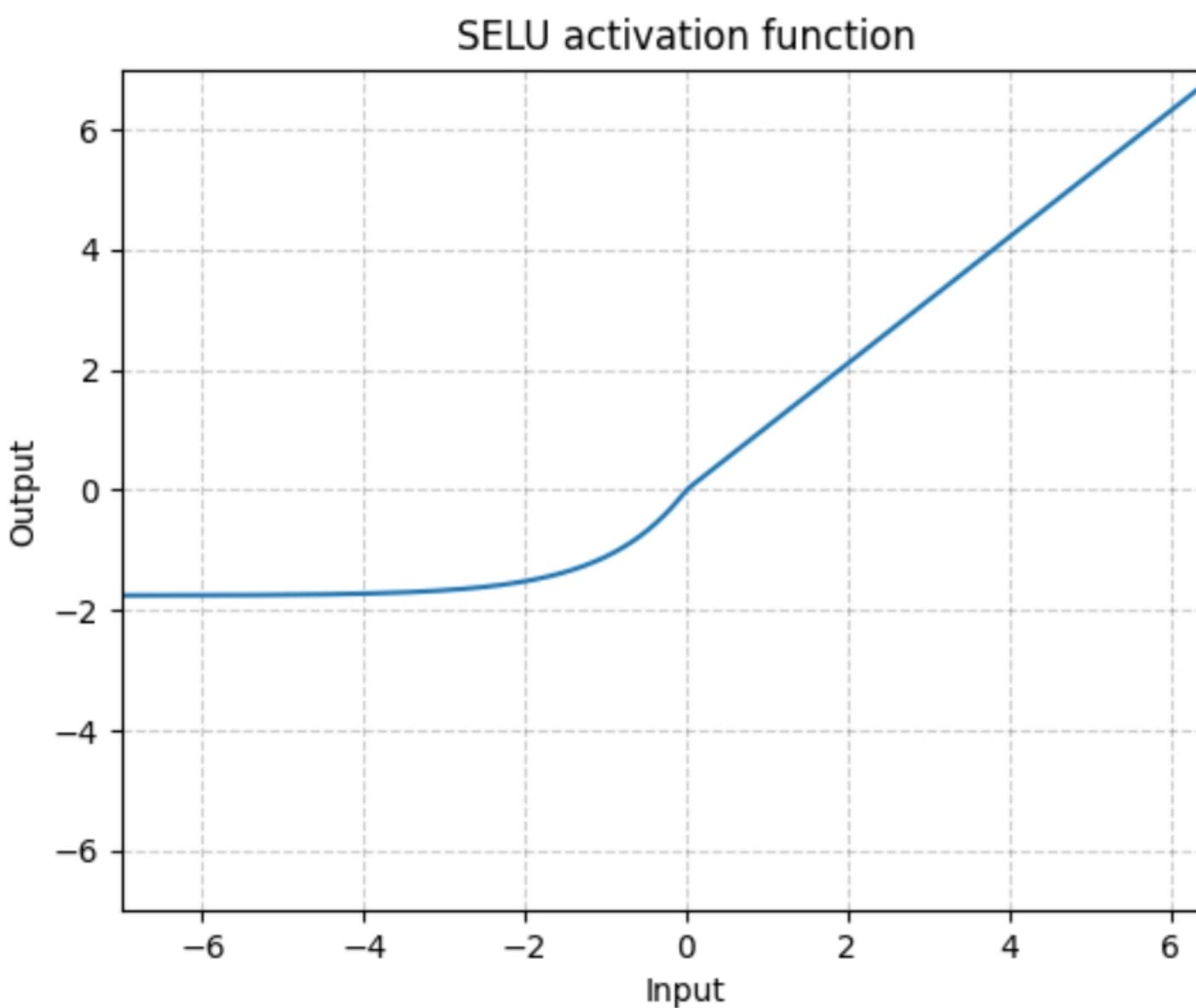


Activation Layers in Deep Learning

$$\text{SELU}(x) = \text{scale} * (\max(0, x) + \min(0, \alpha * e^x - 1))$$

where:

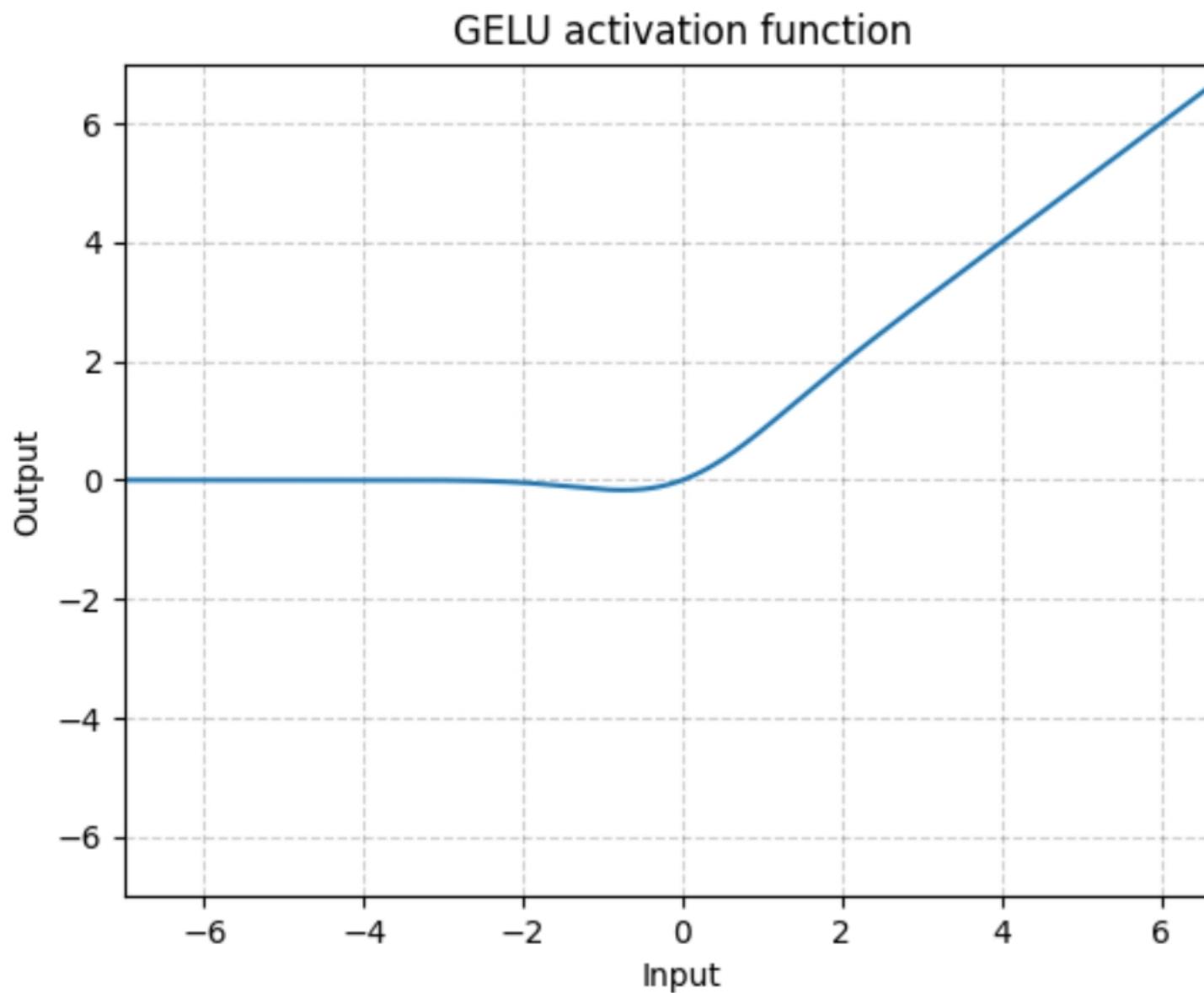
- $\alpha = 1.6732632423543772848170429916717$
- $\text{scale} = 1.0507009873554804934193349852946$



Activation Layers in Deep Learning

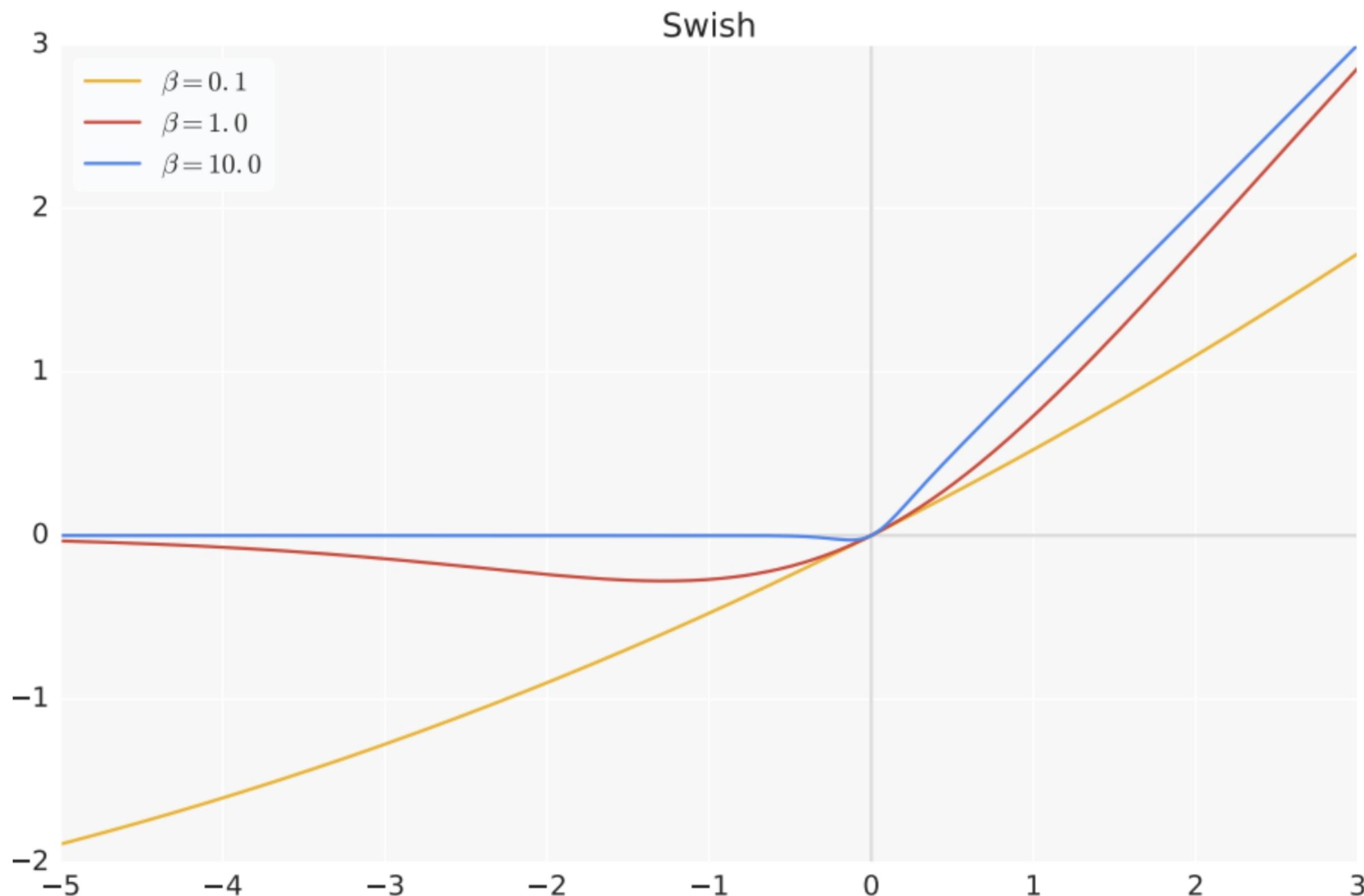
$$\text{GELU}(x) = x * \Phi(x)$$

where $\Phi(x)$ is a Cumulative Distribution Function for Gaussian Distribution.



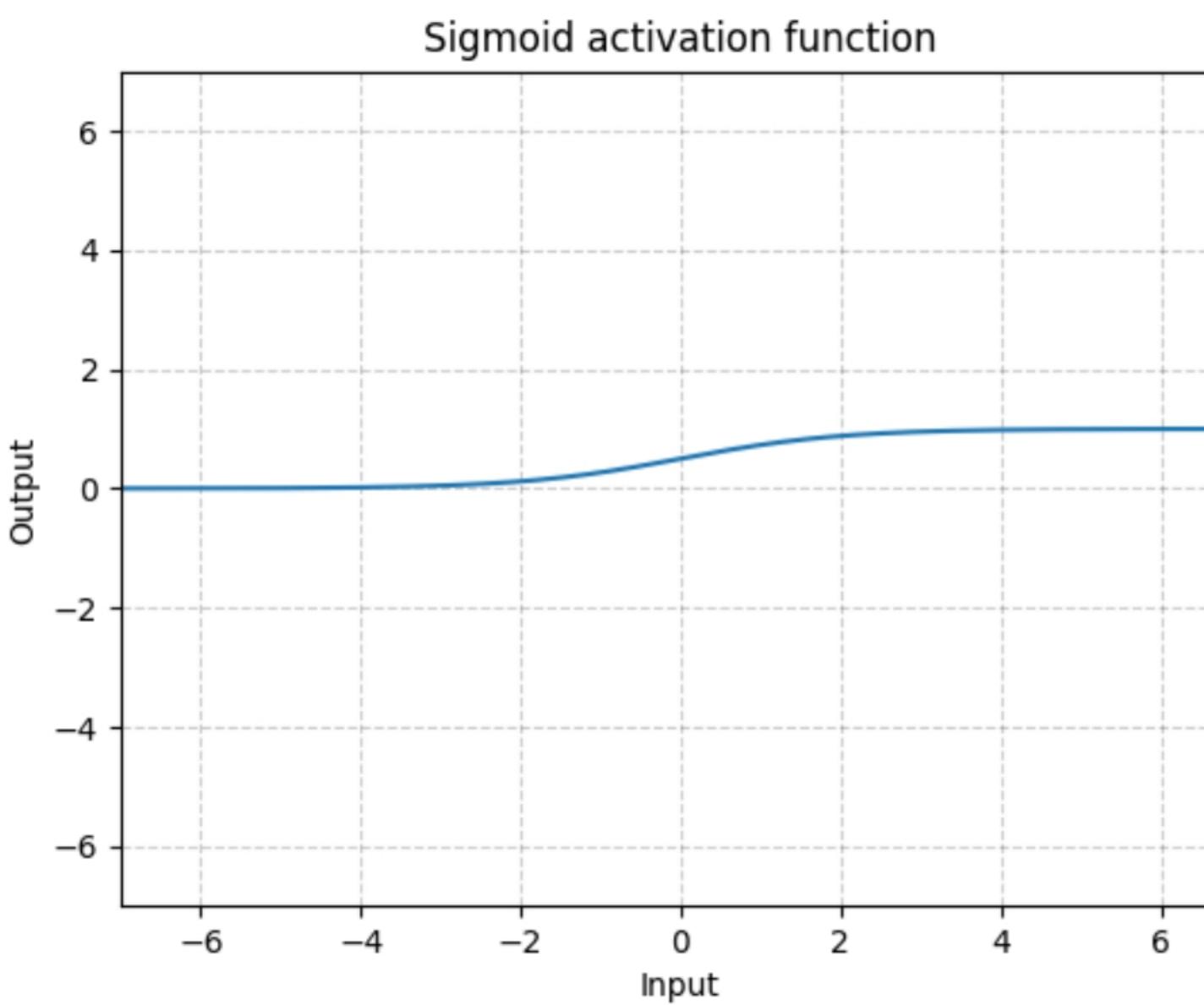
Activation Layers in Deep Learning

$$\text{Swish}(x) = x * \text{Sigmoid}(\beta x) = x * \sigma(\beta x) = \frac{x}{1 + e^{-\beta x}}$$



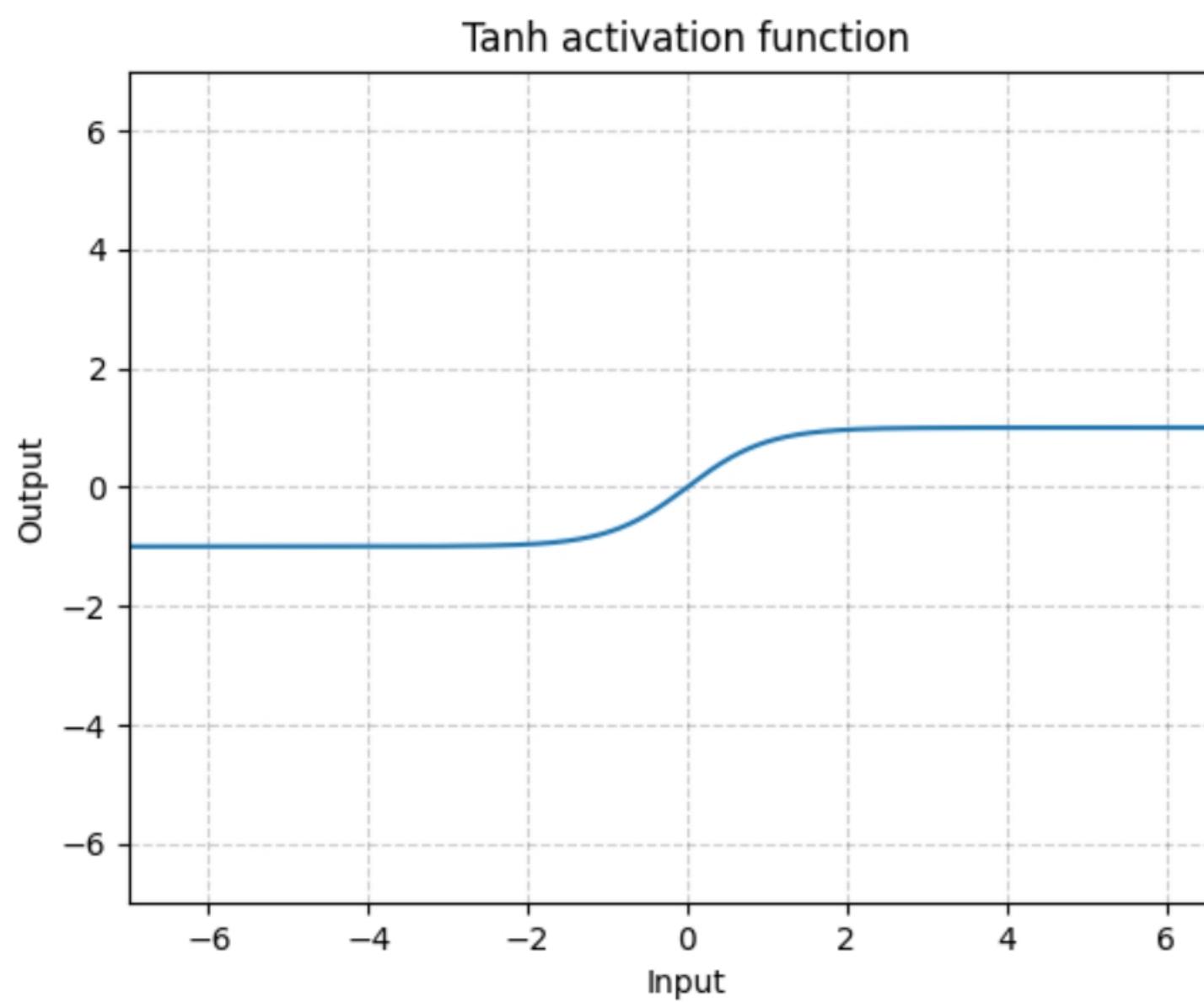
Activation Layers in Deep Learning

$$\text{Sigmoid}(x) = \sigma x = \frac{1}{1 + e^{-x}}$$



Activation Layers in Deep Learning

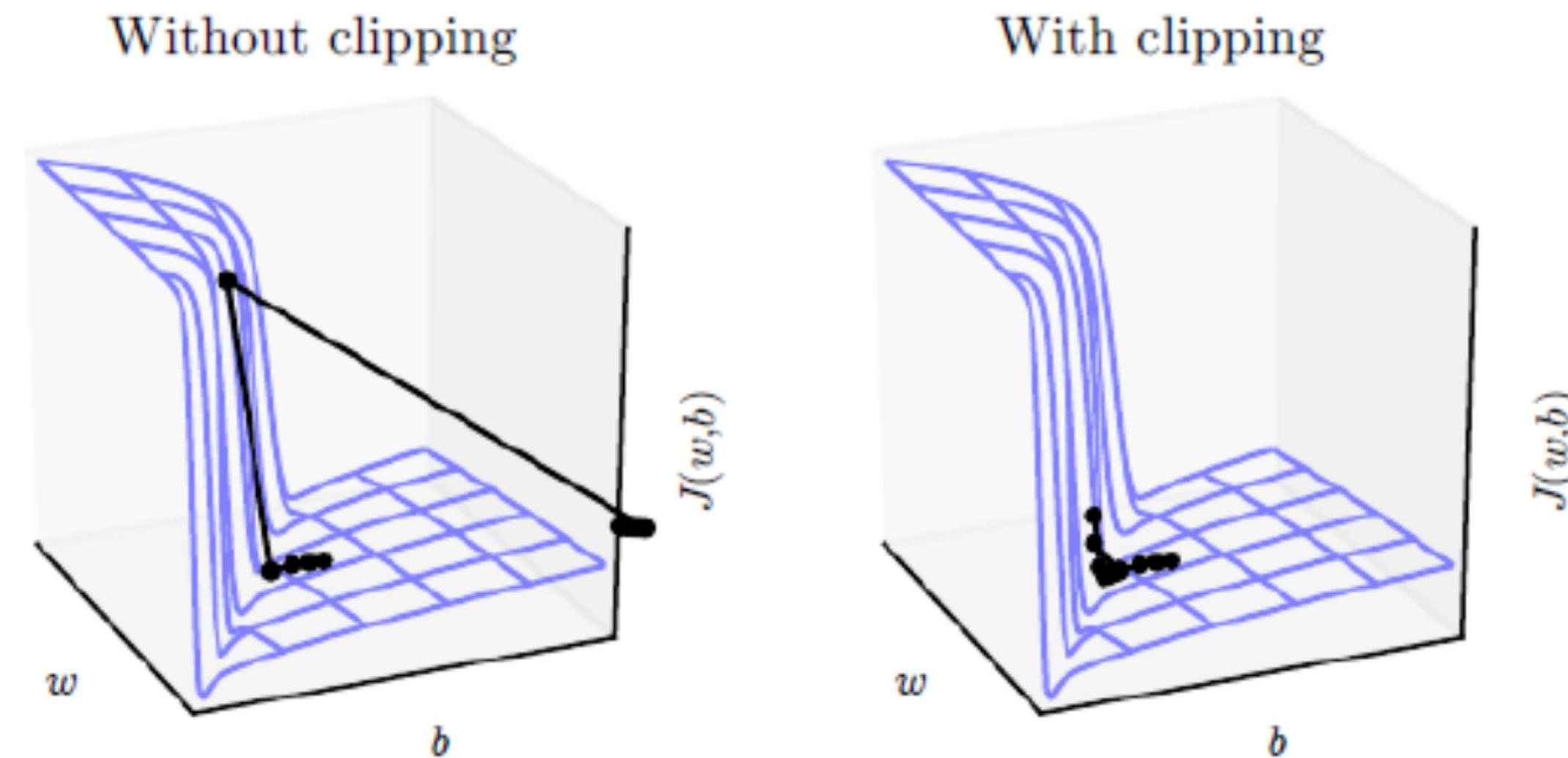
$$\text{Tanh}(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Gradient Explosion

Вибух градієнта є проблемою глибокого навчання, коли градієнти функції втрат щодо параметрів моделі стають занадто великими, що призводить до екстремального оновлення ваг моделі, що може привести до нестабільності моделі.

Щоб побачити приклад, переглянемо `gradient_explosion.ipynb`.



Gradient Explosion: Solutions

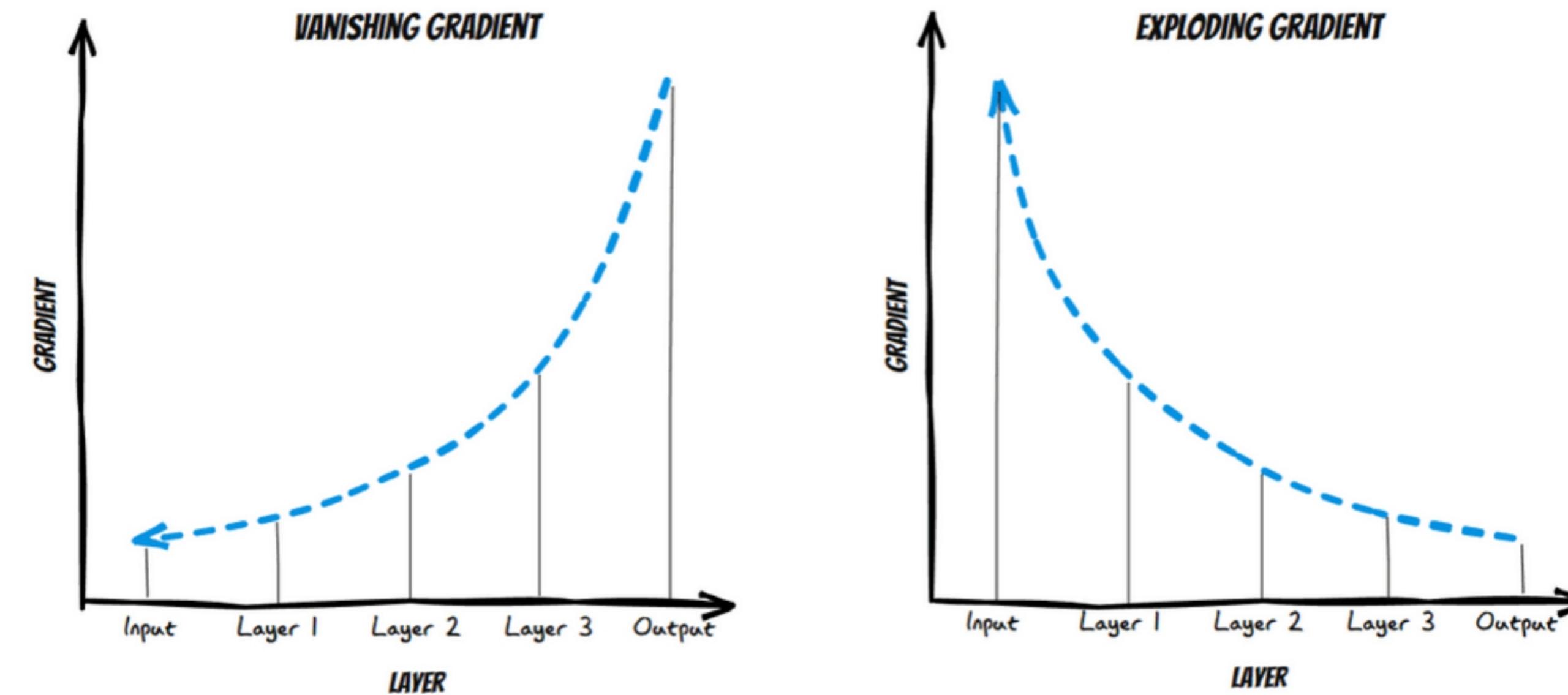
- **Gradient clipping** - після обчислення градієнтів, але перед оновленням ваг, градієнти зменшуються, якщо вони перевищують певний поріг. *In PyTorch: torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)*
- **Lower Learning Rate** - висока швидкість навчання збільшує шанси вибуху градієнта. Зменшення швидкості навчання може допомогти стабілізувати навчання.
- **Weight Regularisation** - регулярізація L1 і L2 може допомогти запобігти тому, щоб ваги стали занадто великими, що, у свою чергу, може допомогти запобігти великим градієнтам.
- **Batch Normalization** - пакетна нормалізація може стабілізувати активації нейронів у мережі, що може допомогти запобігти великим градієнтам.

- **Careful Initialisation** - правильна ініціалізація ваги може запобігти тому, щоб градієнти стали занадто великими або занадто малими на ранніх етапах навчання. Такі методи, як ініціалізація Не або Xavier, можуть бути особливо ефективними для глибоких мереж.
- **Use simpler architectures** - інколи простіші архітектури можуть досягти непоганої продуктивності.
- **Skip Connections / Residual Connections** - вони використовуються в таких архітектурах, як ResNet, і допомагають запобігти вибуху градієнта (або зникненню).
- **Review Loss Function** - іноді вибір функції втрат може привести до великих градієнтів.

Gradient Vanishing

Зникнення градієнта є проблемою глибокого навчання, коли градієнти функції втрат відносно параметрів моделі стають занадто малими. Це може уповільнити навчання або привести до того, що модель повністю припинить навчання.

To see an example please check `gradient_vanishing.ipynb`



Gradient Vanishing: Solutions

- Use activation functions like ReLU, Leaky ReLU, or Swish instead of Sigmoid.
- Proper weight initialisation techniques like He initialisation.
- Using architectures with skip connections, like ResNet.
- Batch normalization.
- Gradient scaling.

Weight Initializations

Ініціалізація Xavier (Glorot) - працює з симетричними функціями активації, такими як \tanh , softmax, sigmoid.

(Ініціалізуйте ваги, використовуючи рівномірний розподіл, і помножте їх на $\sqrt{\frac{6}{fan_{in} + fan_{out}}}$

Ініціалізація Kaiming - працює з функціями активації такими як ReLU.

(Ініціалізуйте ваги, використовуючи нормальний розподіл із середнім значенням 0 і дисперсією 1, і помножте його на $\sqrt{\frac{2}{fan_{in}}}$



Thanks for your attention