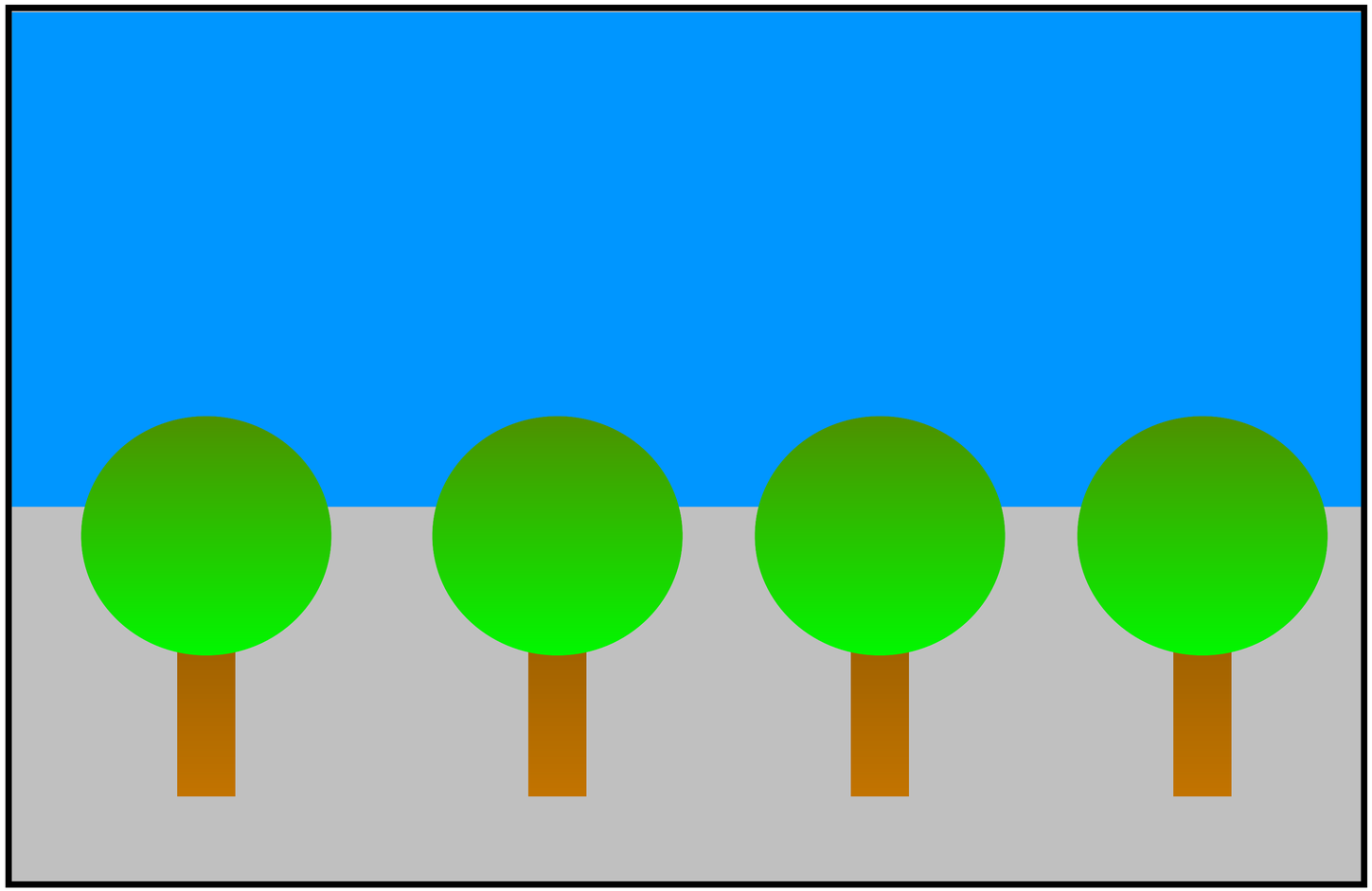
B. Data Structures, Symmetries, and Neural Network Architectures

B.1 Tabular data: MLPs, residual connections, and autoencoders
B.2 Grid-structured data: CNNs
B.3 Sequential data: RNNs, LSTMs, and GRUs
B.4 Graph-structured data: GNNs and message passing
B.5 Set-structured data: Deep Sets

Tiam Heydari, PhD
Postdoctoral Fellow, UBC

Convolutional Neural Netwrks (CNNs): Basics and introduction

- How we (human) understand images?



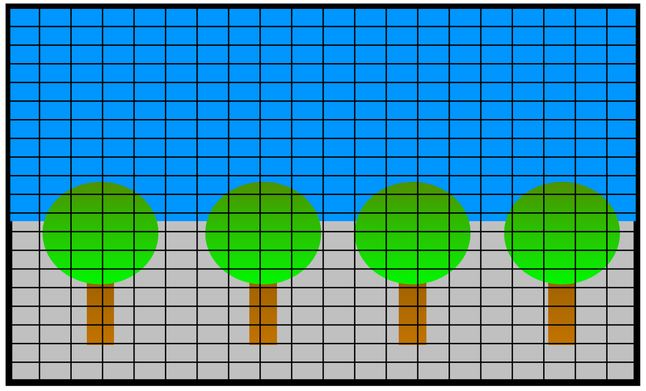
- We see trees
- We see sky
- We see ground?

- Probably is scene of ?
- A park?
- A forest?

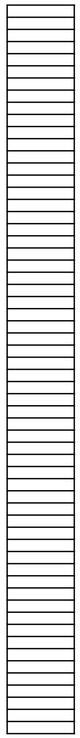
https://writers.com/repetition-definition

Convolutional Neural Netwrks (CNNs): Basics and introduction

- How **MLPs** would try to understand images?
  - e.g. an image is a 20X20 matrix
  - Maybe 3 matrices, one for each color channel (RGB)

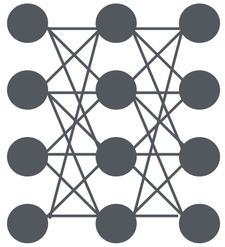


Convert it to row vector



...

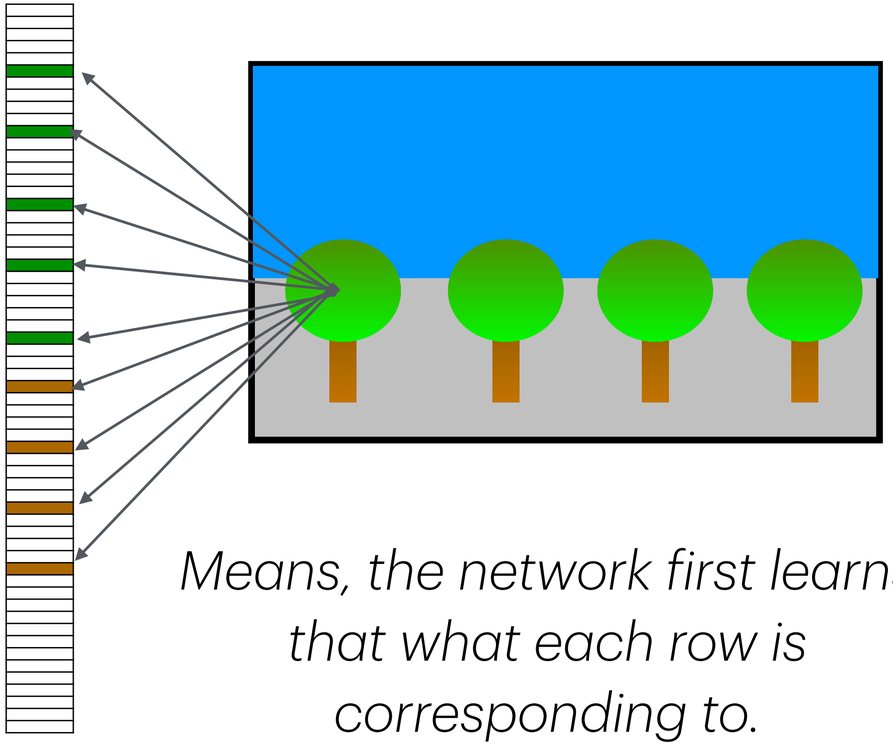
400 rows



...

400 neurons

Probably should learn each tree as something like:

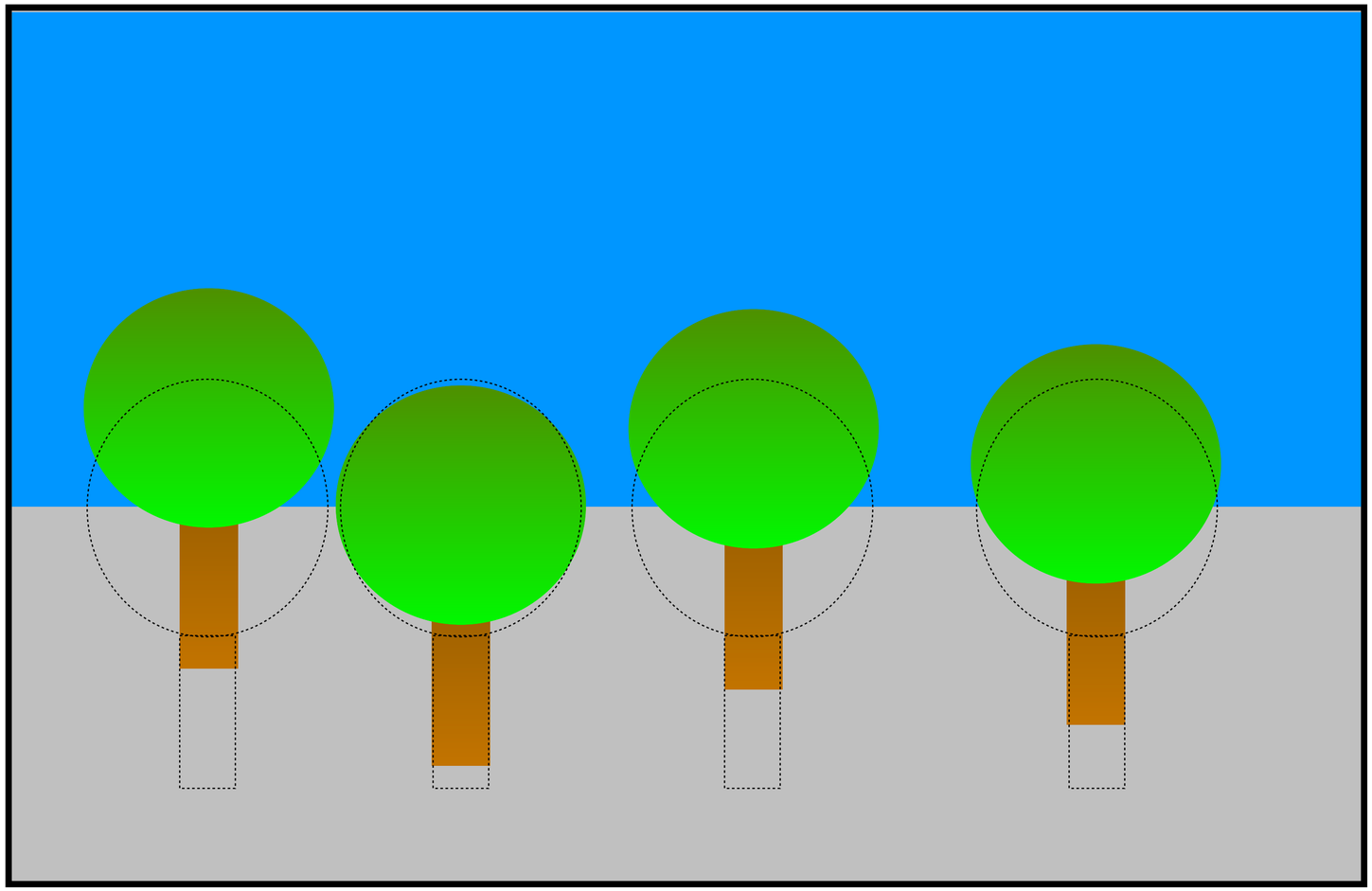


Means, the network first learns that what each row is corresponding to.

Convolutional Neural Netwrks (CNNs): Basics and introduction

- How we (human) understand images?

- **What if the position of trees changes a bit?**



- **Does any of bellow changes?**

- We see trees
- We see sky
- We see ground?

- Probably is scene of ?
- A park?
- A forest?

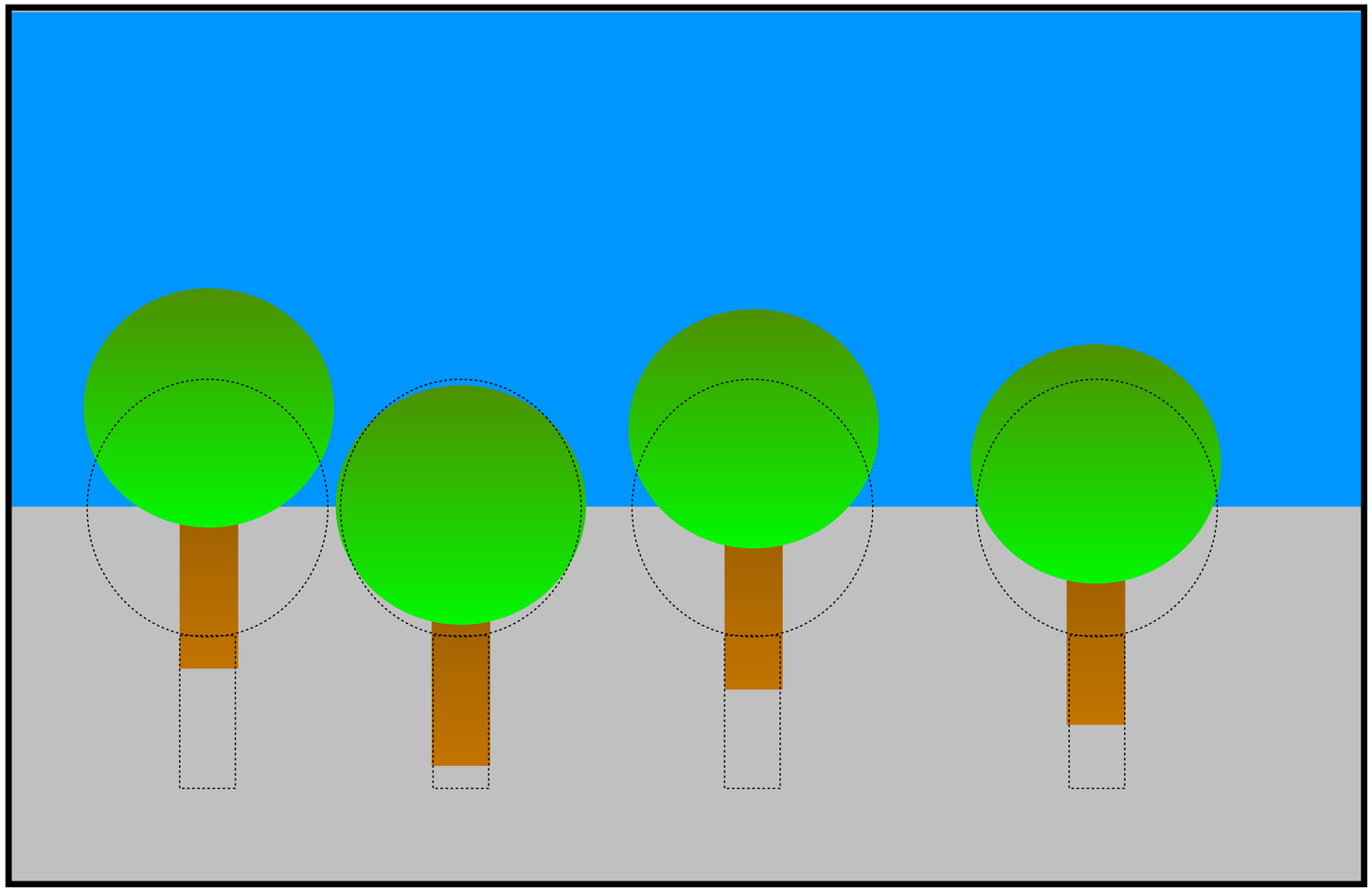
No!

https://writers.com/repetition-definition

Convolutional Neural Netwrks (CNNs): Basics and introduction

- How we (human) understand images?

- **What if the position of trees changes a bit?**



- **Does any of bellow changes?**

- We see trees
- We see sky
- We see ground?

- Probably is scene of ?
- A park?
- A forest?

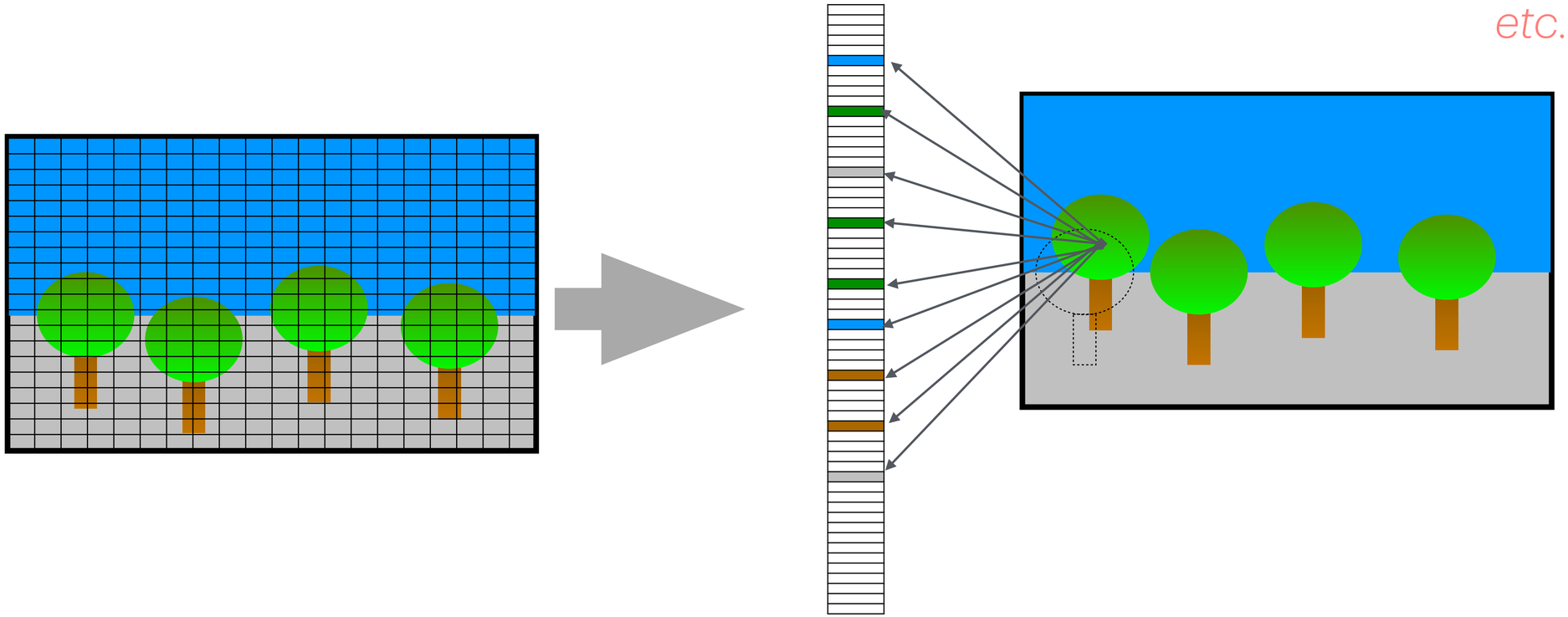
No!

https://writers.com/repetition-definition

Convolutional Neural Netwrks (CNNs): Basics and introduction

- How **MLPs** would try to understand images?
  - **What if the position of trees changes a bit?**

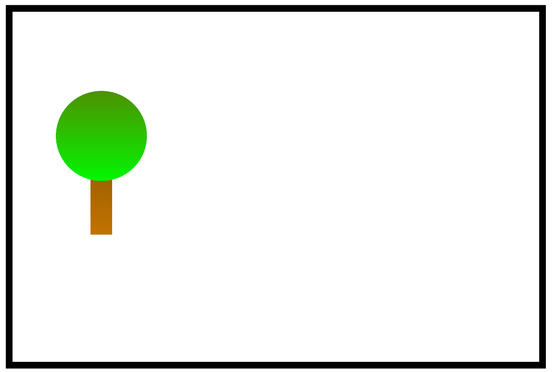
Everything will be messed up, some neurons that would have recognized leaves now are looking at sky or ground etc...



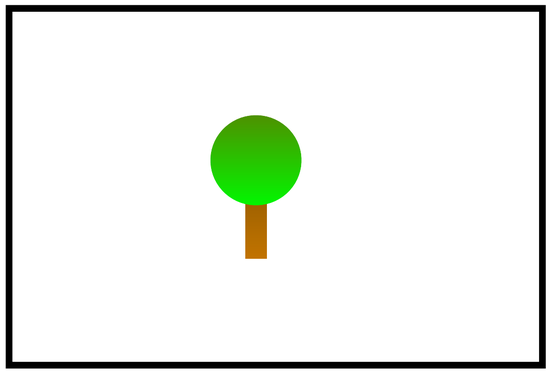
Convolutional Neural Netwrks (CNNs): Basics and introduction

- Lets analyze what happened, and why MPLs get into trouble
- In an abstract view, images have one deep characteristics: **Translational (shift) symmetry**

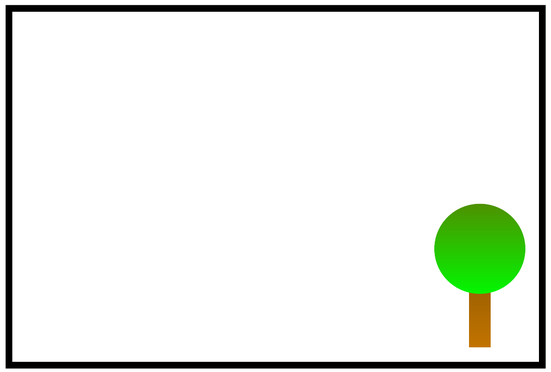
This is also a tree



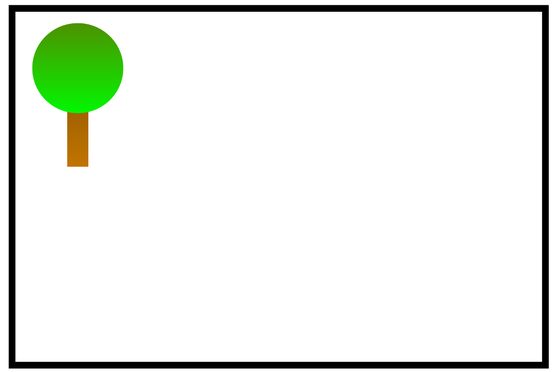
This is also a tree



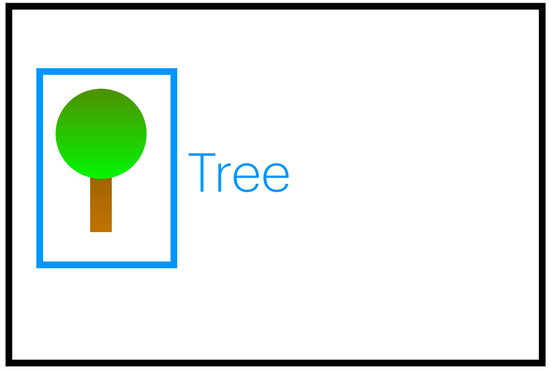
This is also a tree

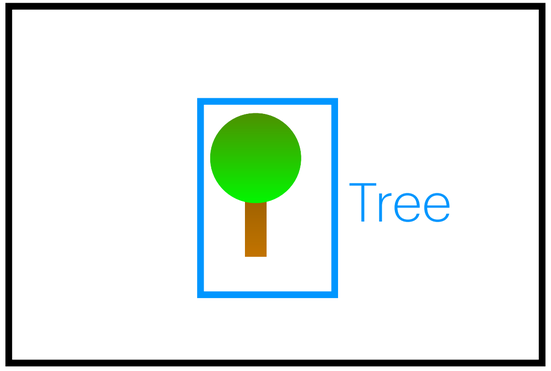


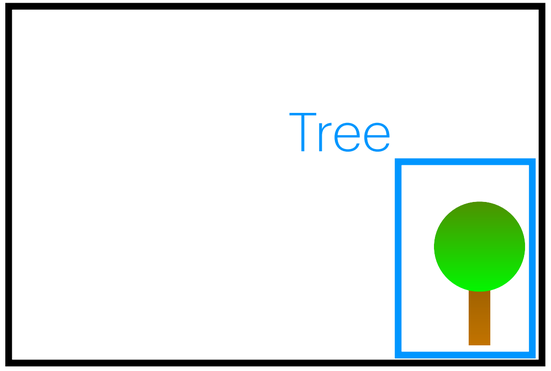
This is also a tree



- From the example above, you can see that the best way to identify the tree in the image would be to have a “local” operator that does not care about the positions of the tree!
- Something like:

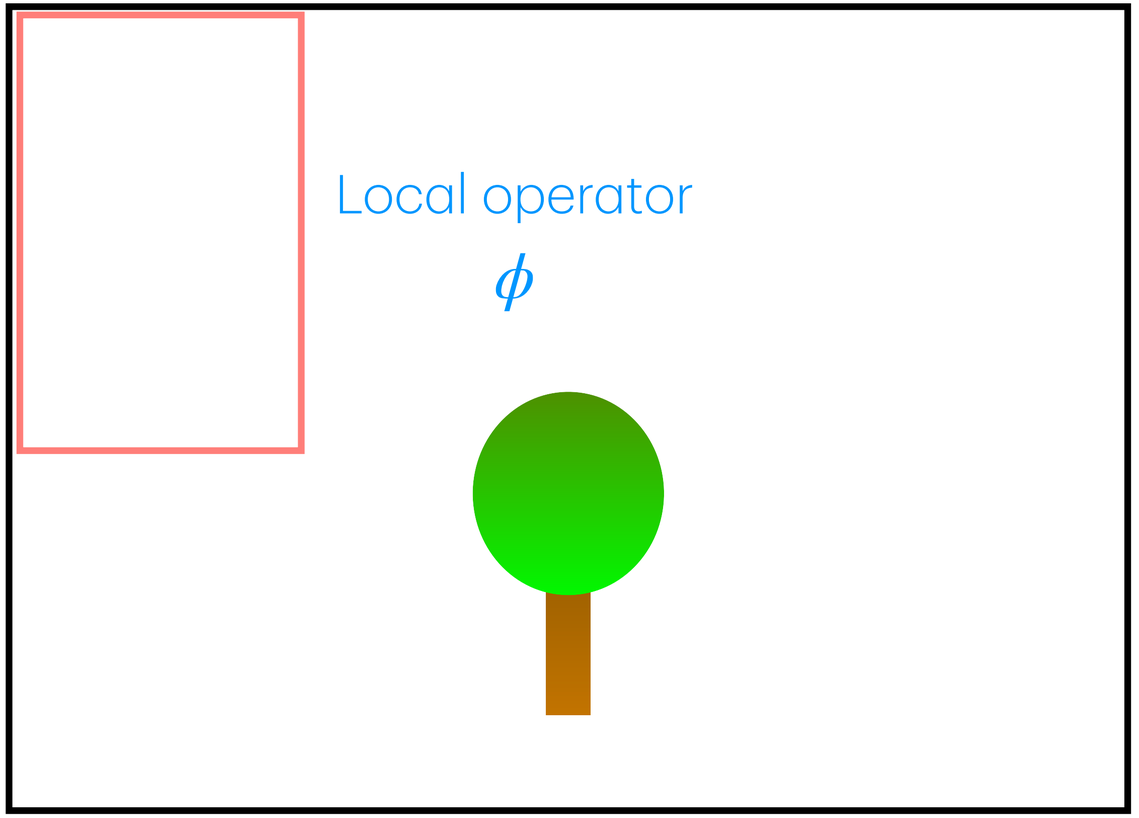






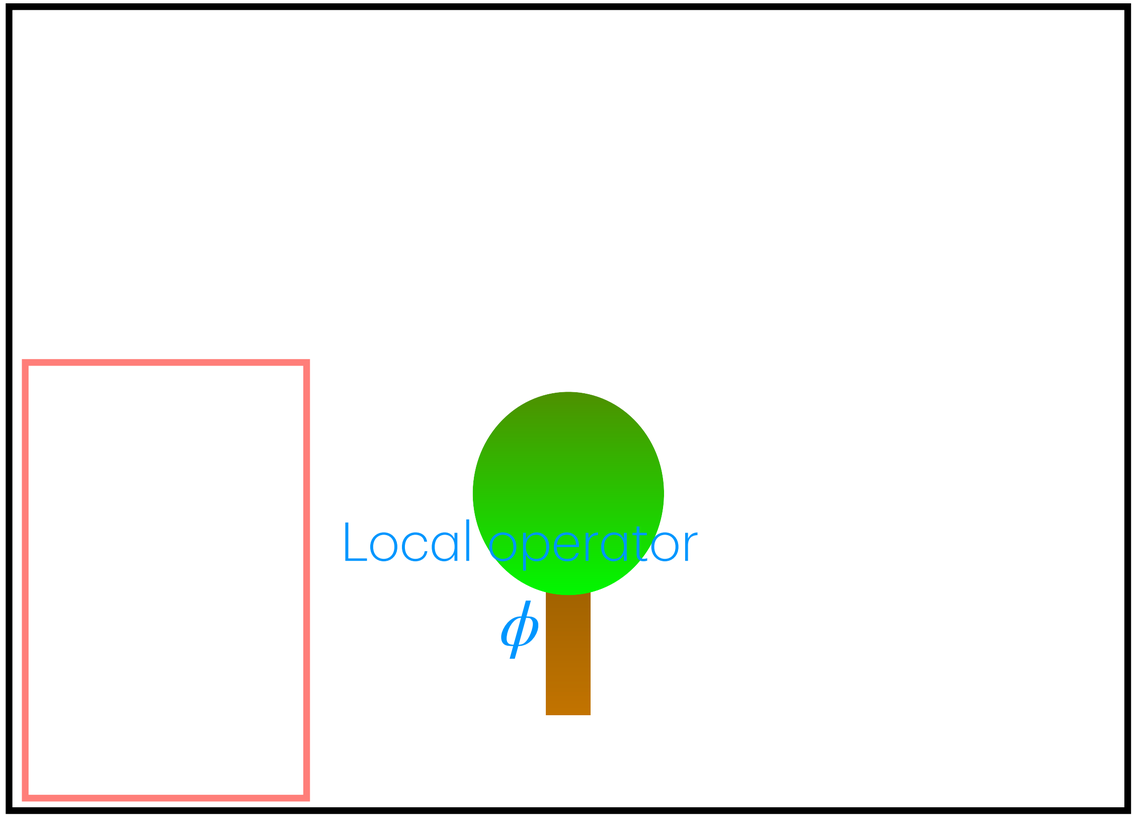
Convolutional Neural Netwrks (CNNs): Basics and introduction

- So what we will do next, is to define such a local operator, that can identify objects locally, and then will slide it around to find objects, or more generally local features in our input:



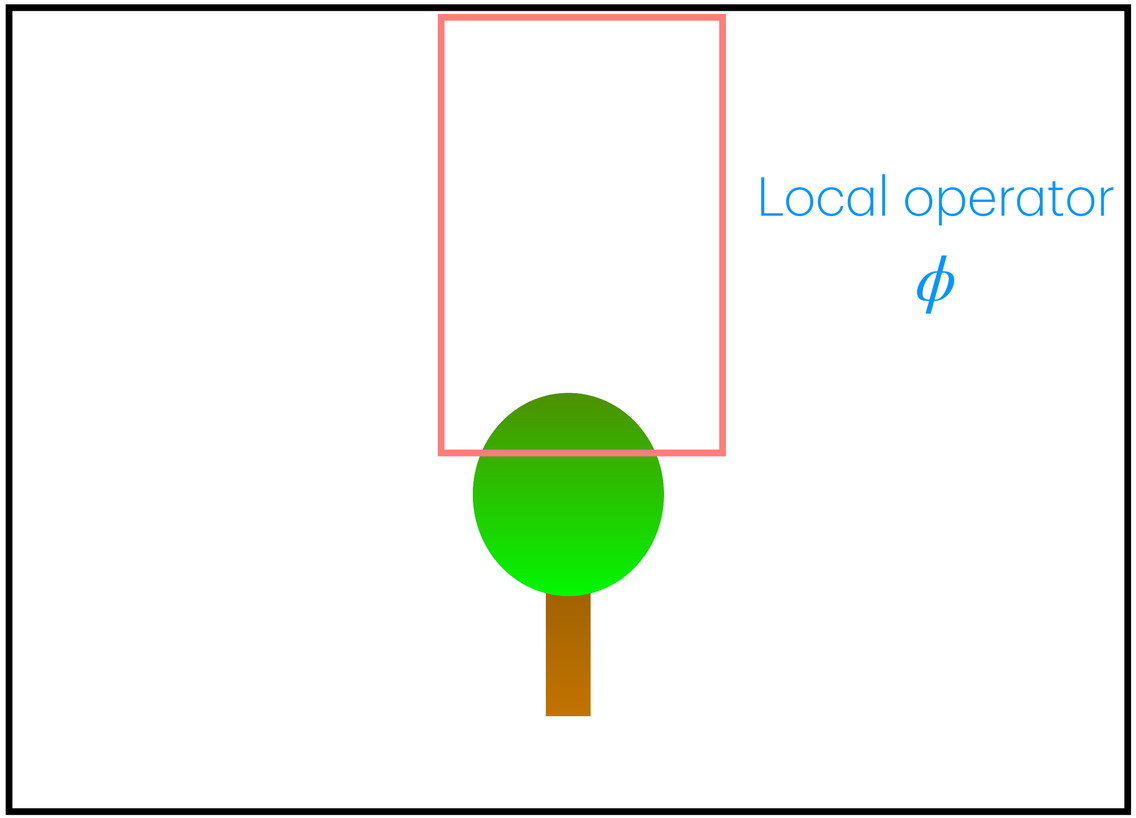
Convolutional Neural Netwrks (CNNs): Basics and introduction

- So what we will do next, is to define such a local operator, that can identify objects locally, and then will slide it around to find objects, or more generally local features in our input:



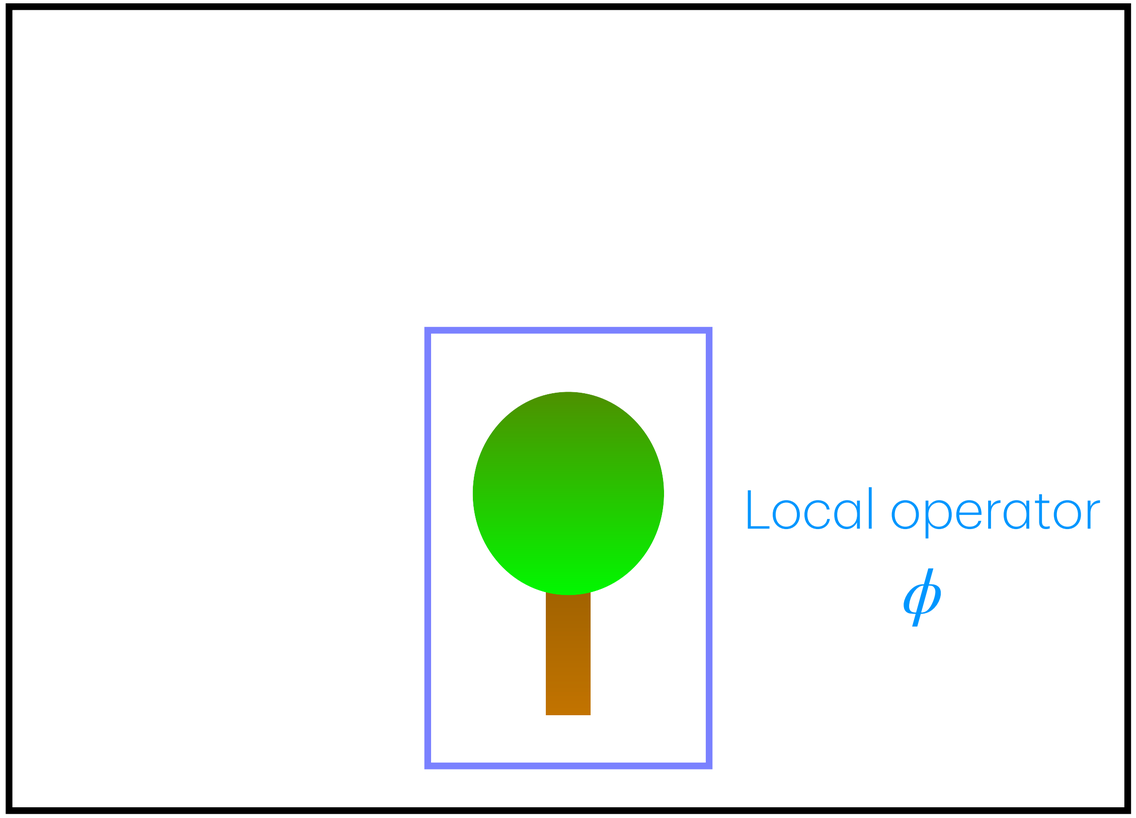
Convolutional Neural Netwrks (CNNs): Basics and introduction

- So what we will do next, is to define such a local operator, that can identify objects locally, and then will slide it around to find objects, or more generally local features in our input:



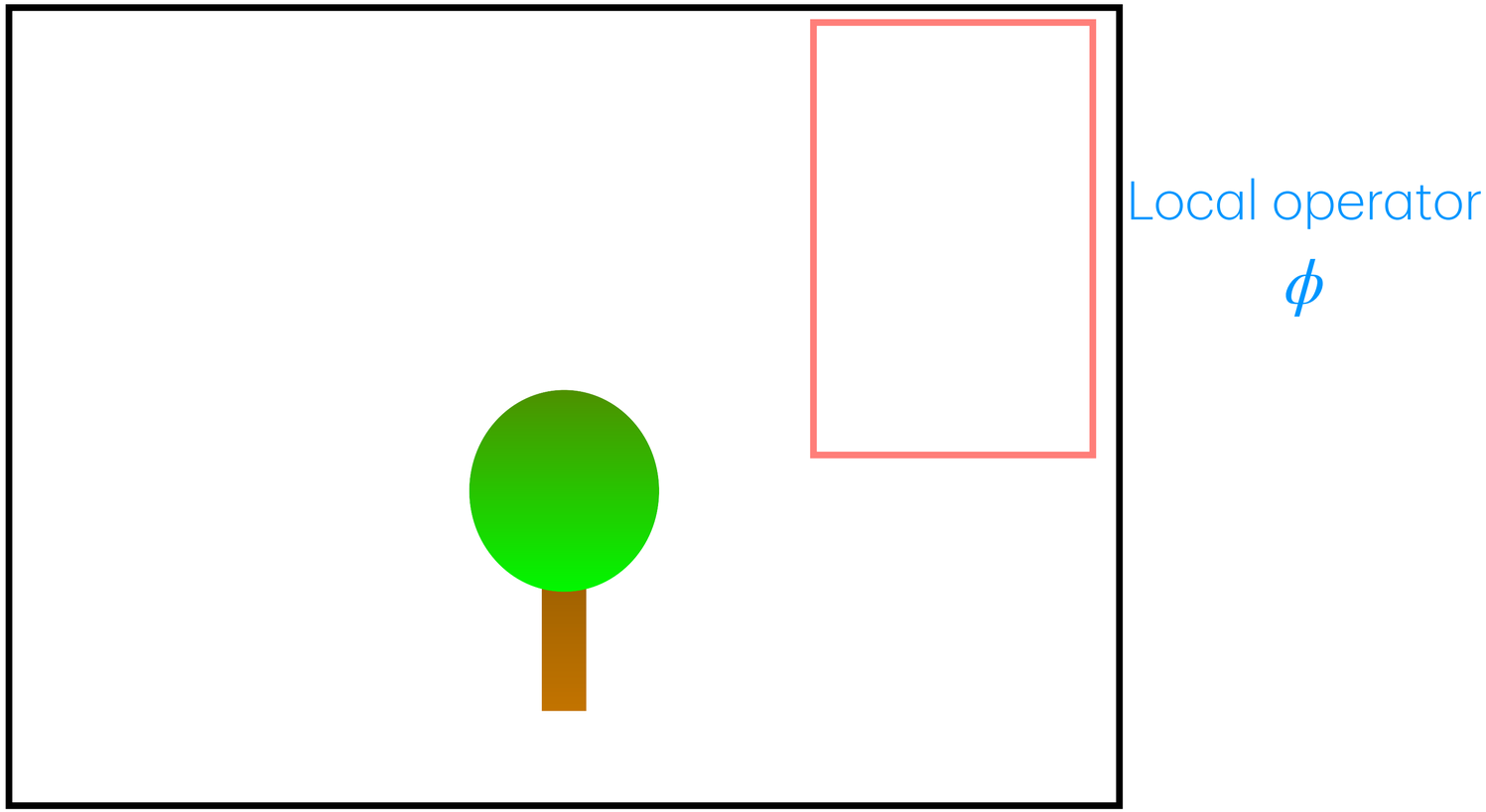
Convolutional Neural Netwrks (CNNs): Basics and introduction

- So what we will do next, is to define such a local operator, that can identify objects locally, and then will slide it around to find objects, or more generally local features in our input:



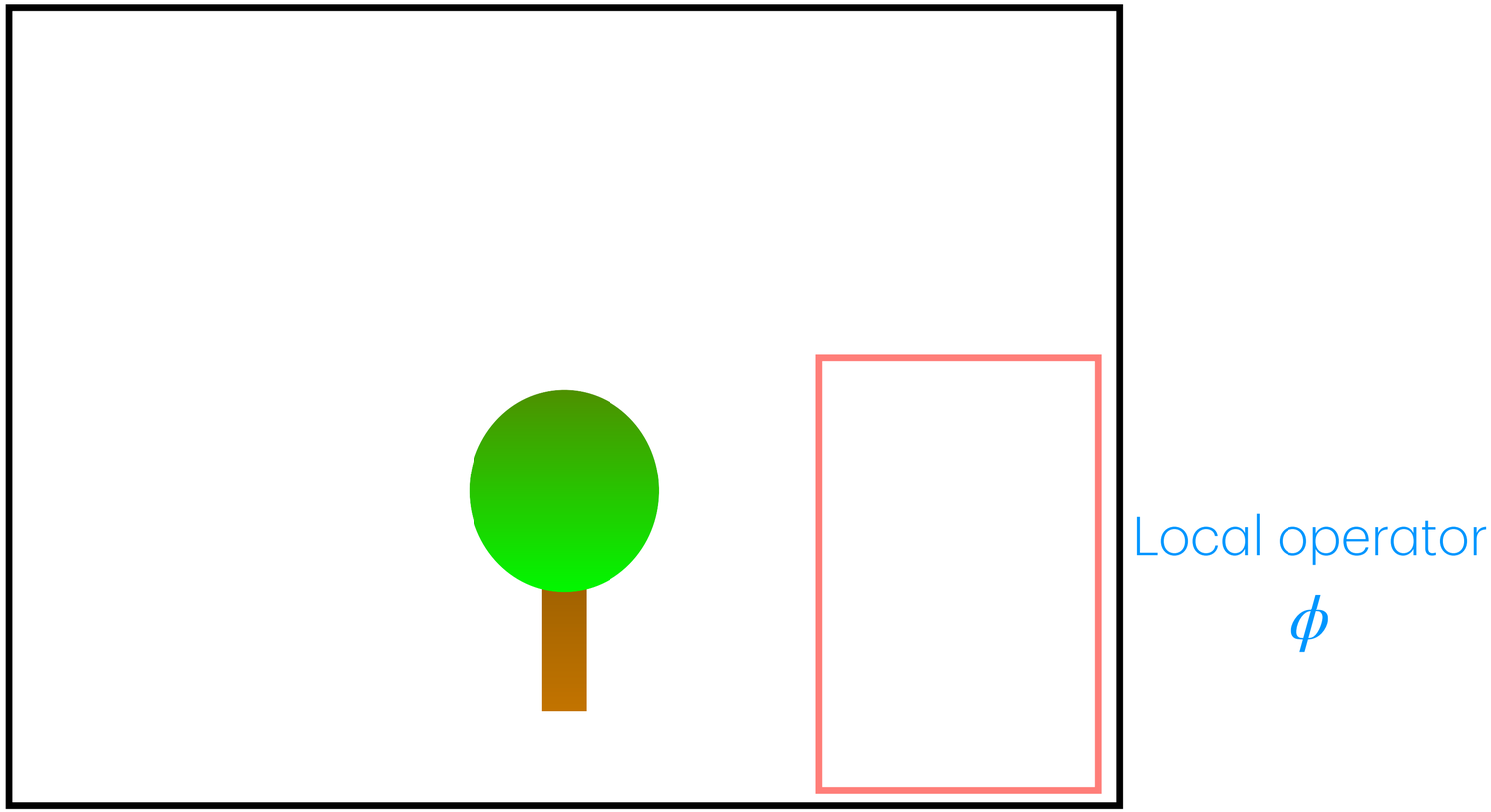
Convolutional Neural Netwrks (CNNs): Basics and introduction

- So what we will do next, is to define such a local operator, that can identify objects locally, and then will slide it around to find objects, or more generally local features in our input:



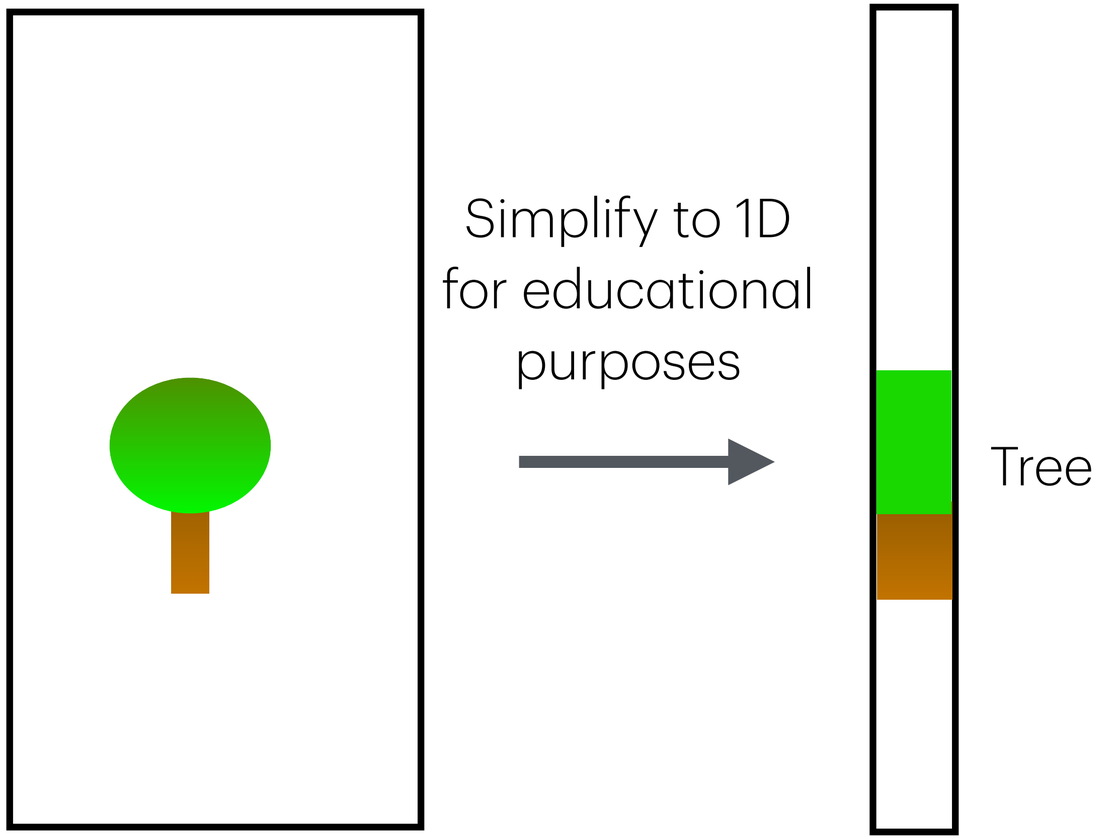
Convolutional Neural Netwrks (CNNs): Basics and introduction

- So what we will do next, is to define such a local operator, that can identify objects locally, and then will slide it around to find objects, or more generally local features in our input:



Convolutional Neural Netwrks (CNNs): Basics and introduction

- The same scenario still applies, even if our input was not a 2D matrix (an image).
- To make notation and visualization more intuitive, let's consider 1D matrix as out input.

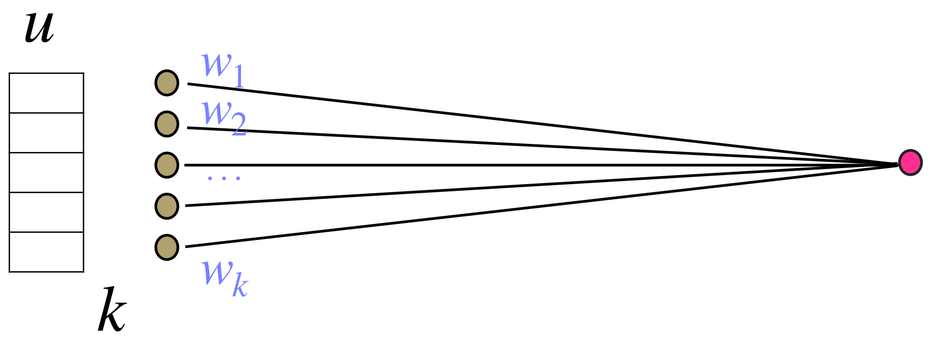


- Now let's look at the same problem again:

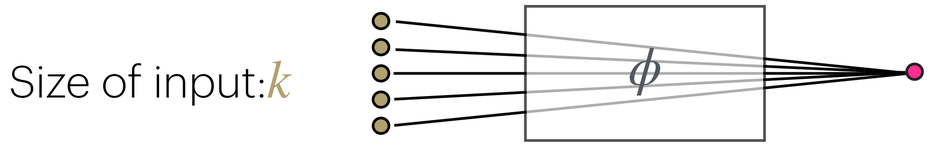
- In our 1D world, where our image has the dimension N, we can think of a **local linear operator,** one that only looks at a small window u of size k of the input, where $k \ll N$.
- Mathematically, ϕ computes a dot product between the input window $u \in \mathbb{R}^k$ and a learned weight vector $w \in \mathbb{R}^k$:

$$\phi : \mathbb{R}^k \to \mathbb{R}$$
$$\phi(u) = u \,.\, w = u_1 w_1 + u_2 w_2 + \ldots + u_k w_k$$

- Computation graph:

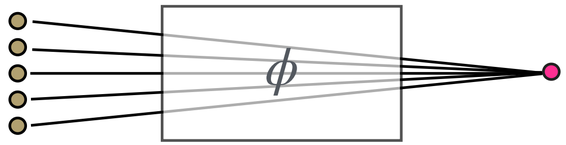


- Visually we show it like:



Concept 1: Linear Convolutional Operator

Size of input: k



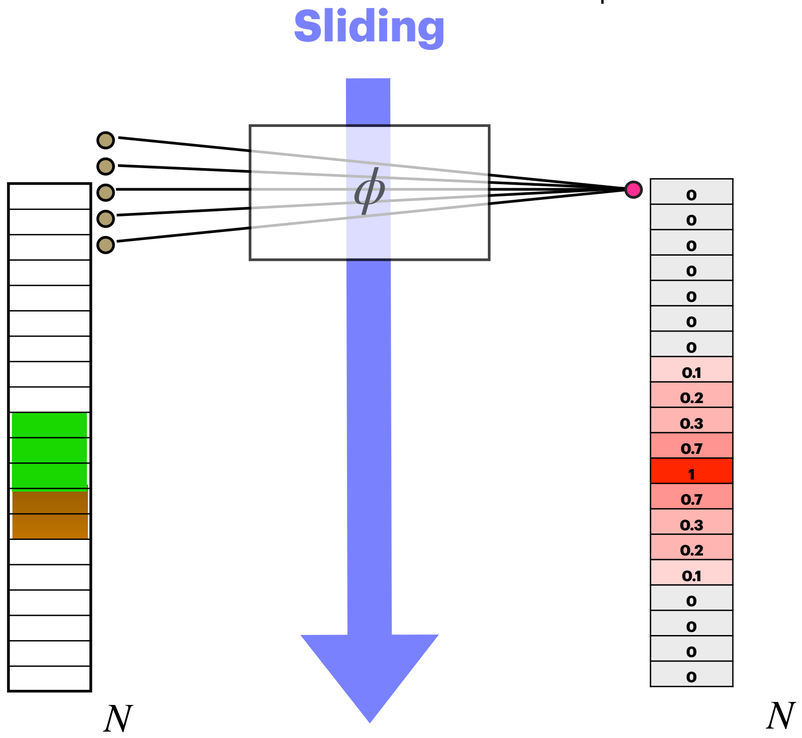
Size of output: *1*

$$\phi : \mathbb{R}^k \rightarrow \mathbb{R}$$

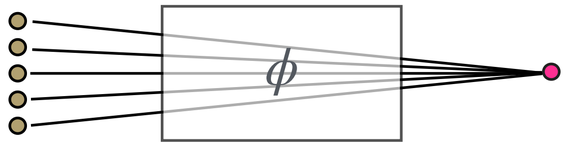
$$\phi(u) = u \,.\, w = u_1 w_1 + u_2 w_2 + \ldots + u_k w_k$$

- If we **slide** the local operator ϕ over our input, something like this would happen:

Is the input window a tree?



Size of input: k



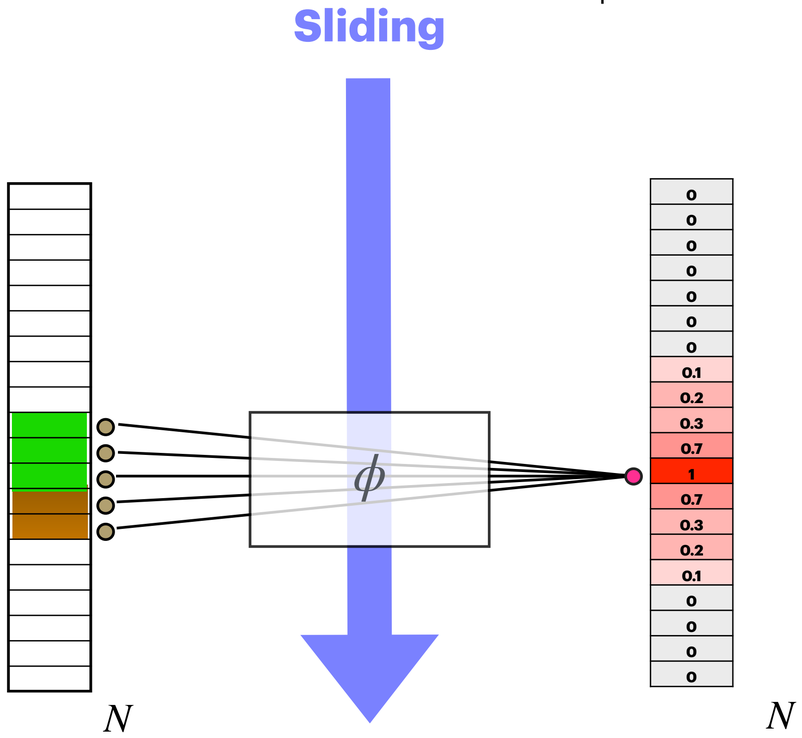
Size of output: *1*

$$\phi : \mathbb{R}^k \rightarrow \mathbb{R}$$

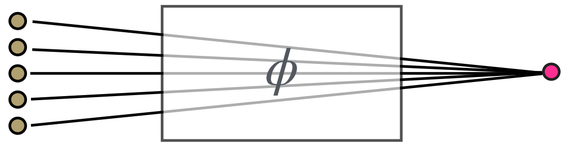
$$\phi(u) = u \,.\, w = u_1 w_1 + u_2 w_2 + \ldots + u_k w_k$$

- If we **slide** the local operator ϕ over our input, something like this would happen:

Is the input window a tree?



Size of input: k



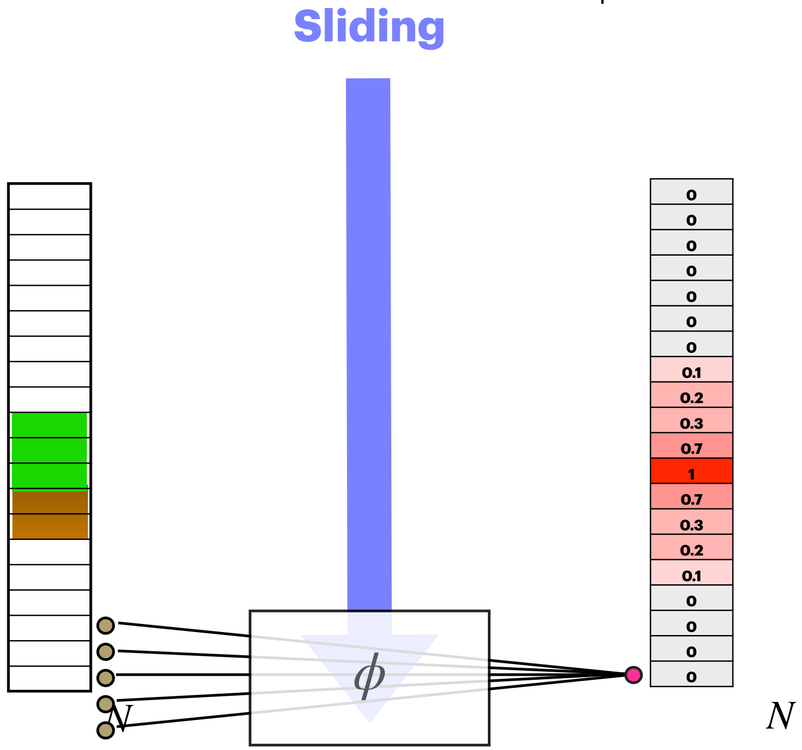
Size of output: 1

$$\phi : \mathbb{R}^k \rightarrow \mathbb{R}$$

$$\phi(u) = u \,.\, w = u_1 w_1 + u_2 w_2 + \ldots + u_k w_k$$

- If we **slide** the local operator ϕ over our input, something like this would happen:

Is the input window a tree?

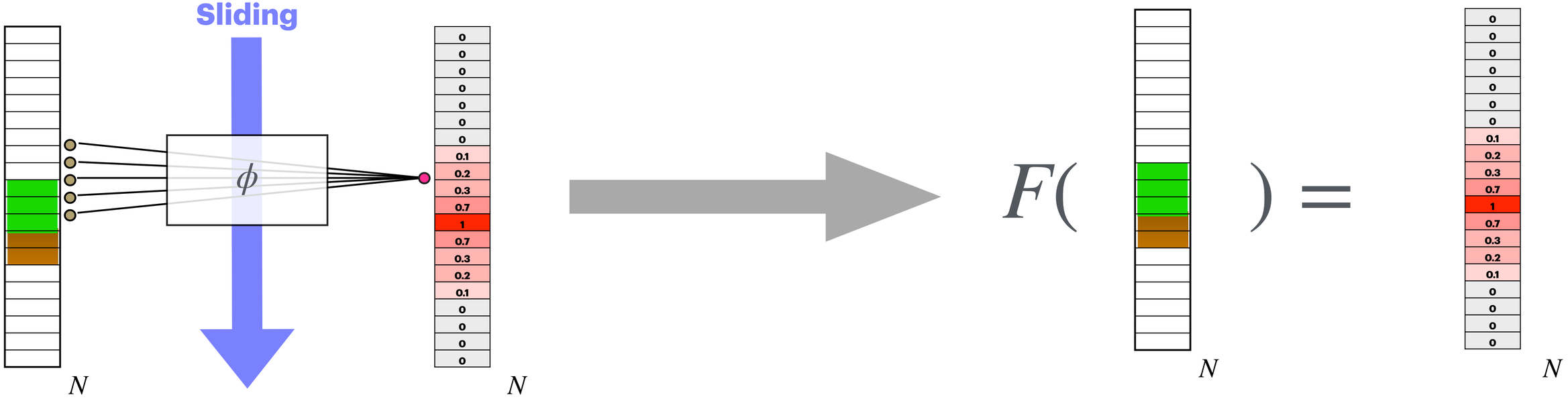


- **Note that the linear weights w, are learnable via backpropagation**

- We can combine these two operators into a function, and call it **F**:

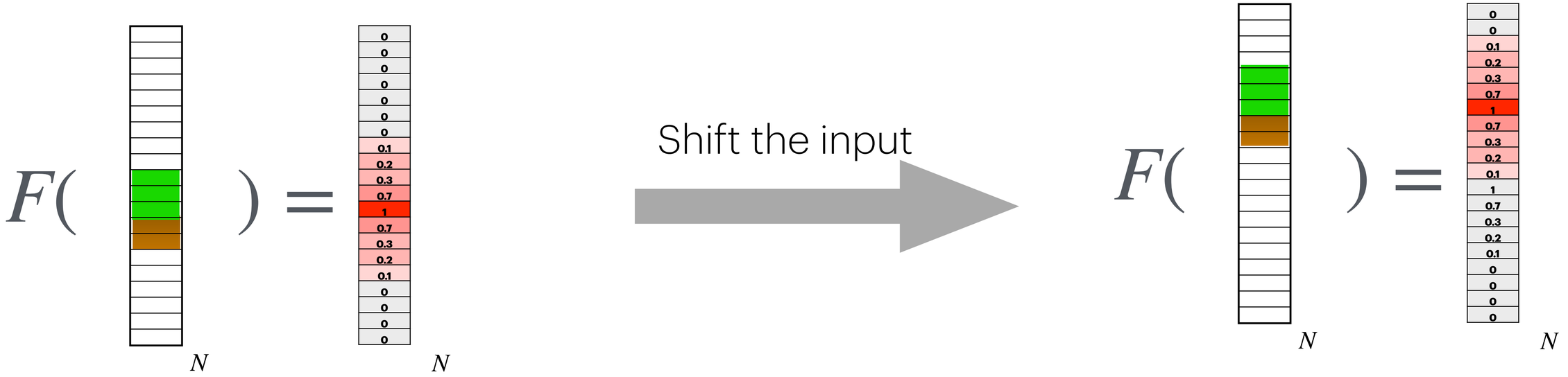
F = Sliding the local operator (known as *kernel*) ϕ on the whole input

$$F : \mathbb{R}^N \rightarrow \mathbb{R}^{N'}$$



- What does Function **F** do?
  - It probabilistically locates the position of the tree in our input, and points it out!

- Function **F** also has a very unique feature:
  - If we apply a shift (translation) to the input of **F**, the output of **F** changes exactly according to the shift.



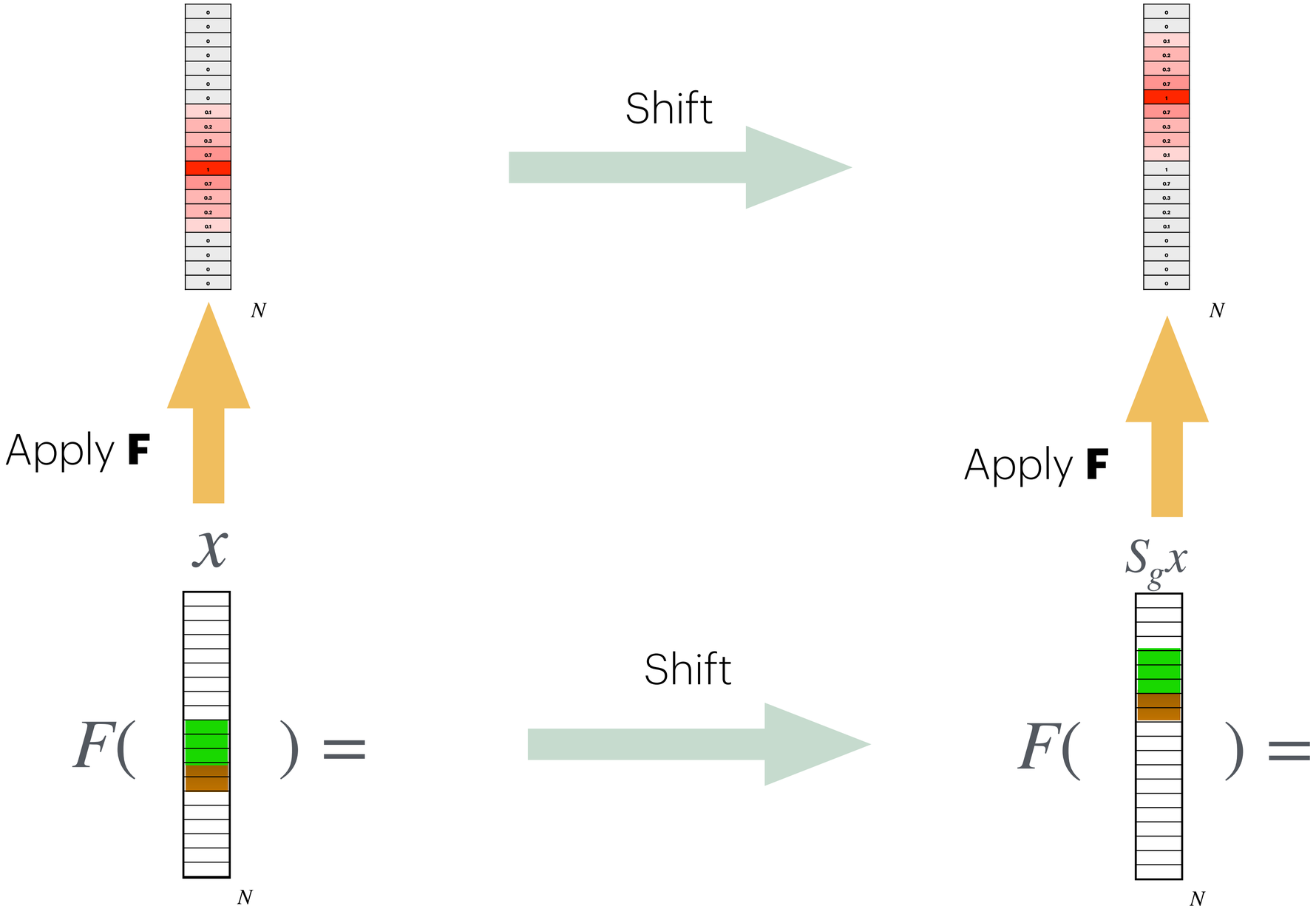
- Mathematically: The Shift operator that shifts the input by g, is defined as:

$$S_g \quad \text{for } g \in \mathbb{Z}$$

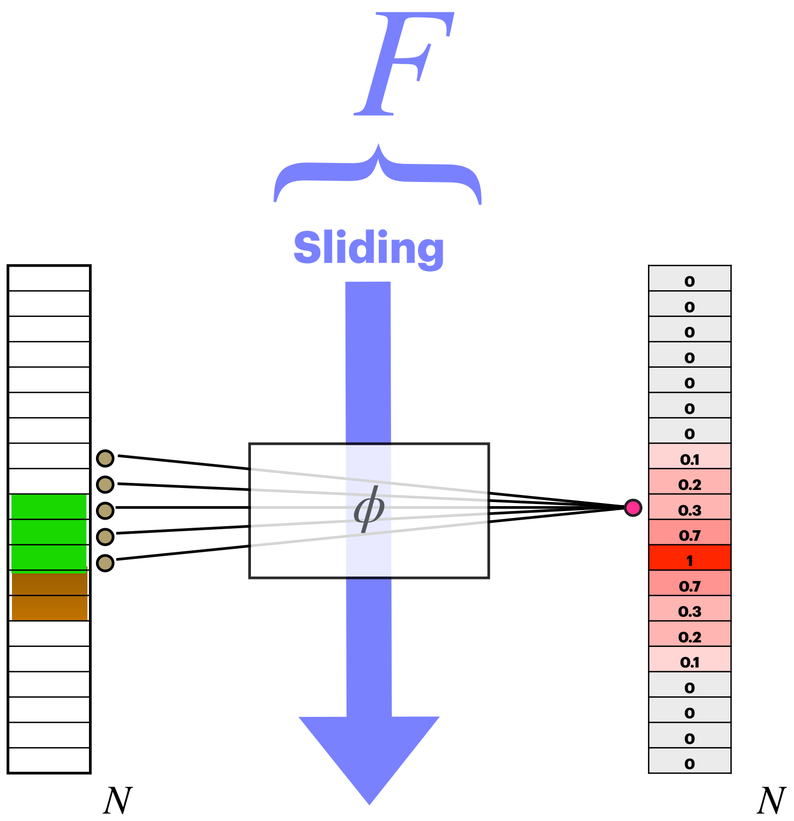
Then the function **F** has this unique feature: **F** is **shift Equivariance**

$$F(S_g x) = S_g F(x)$$

- Visually if F is Equivaraince ($F(S_g x) = S_g F(x)$) means:



- Let’s go back to our motivation:
  Instead of feeding the whole input into an large MLP, we defined a smarter operator: a local kernel ϕ that slides over the entire input. We called this operator **F**.



- In standard ML notation, F is called a convolutional operator, and the operation is written as:

$$F(x) = \phi \star x \quad , \quad F : \mathbb{R}^N \to \mathbb{R}^{N'}$$

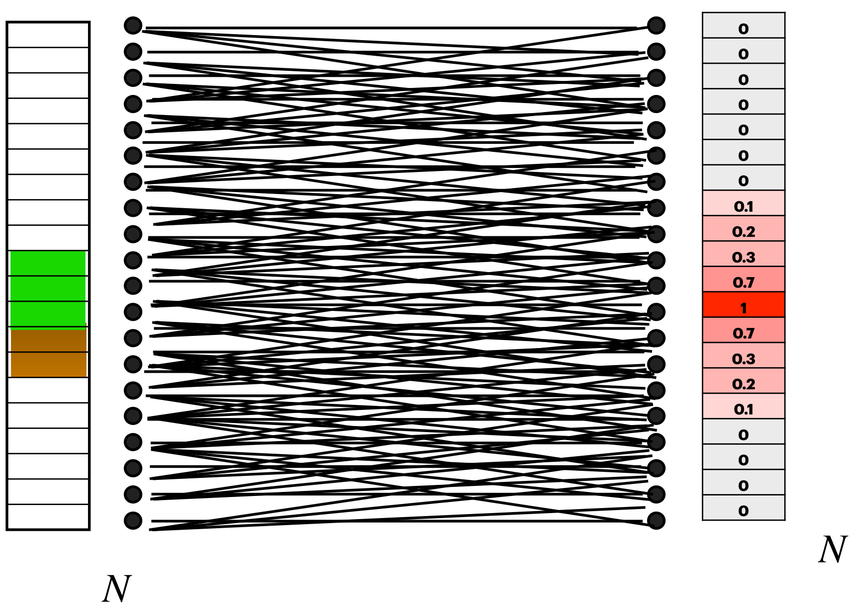
This notation means exactly what we described: slide the kernel ϕ (which is a linear function) over every position of x. That's it.

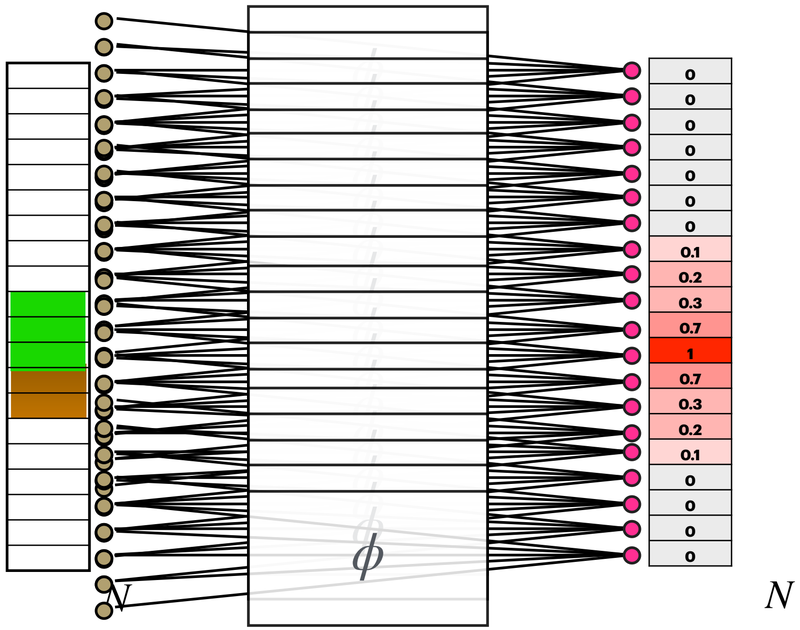
And F is a translation-equivariant operator:

$$F(S_g x) = S_g F(x) \, .$$

⏸ Pause Point

Questions: For a 1D input $x \in \mathbb{R}^N$, compare: Compare one dense linear layer $\mathbb{R}^N \rightarrow \mathbb{R}^N$ with one 1D convolutional filter of width k. Which one uses fewer learnable parameters?



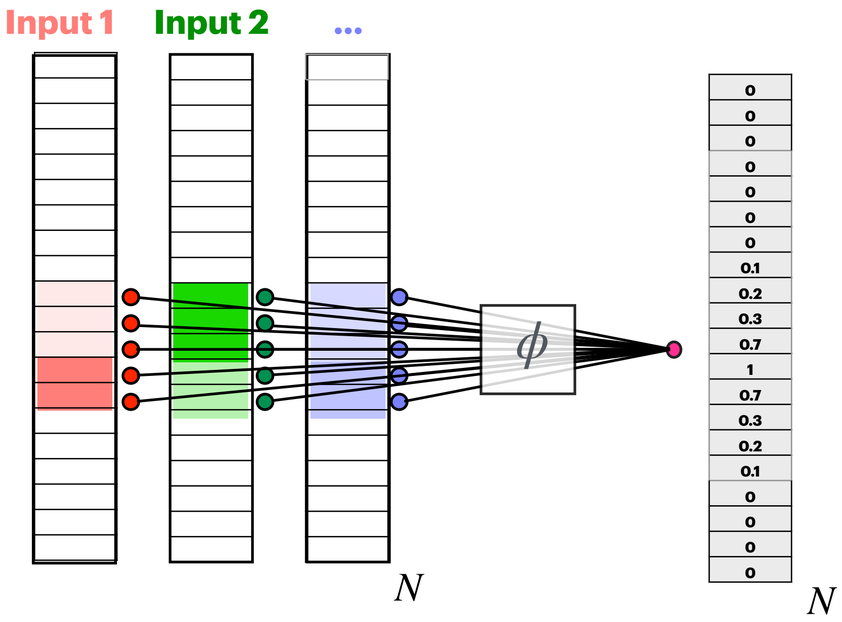


- MLP parameters = $N^2 + N$
- Convolutional Layer parameters: $k + 1$

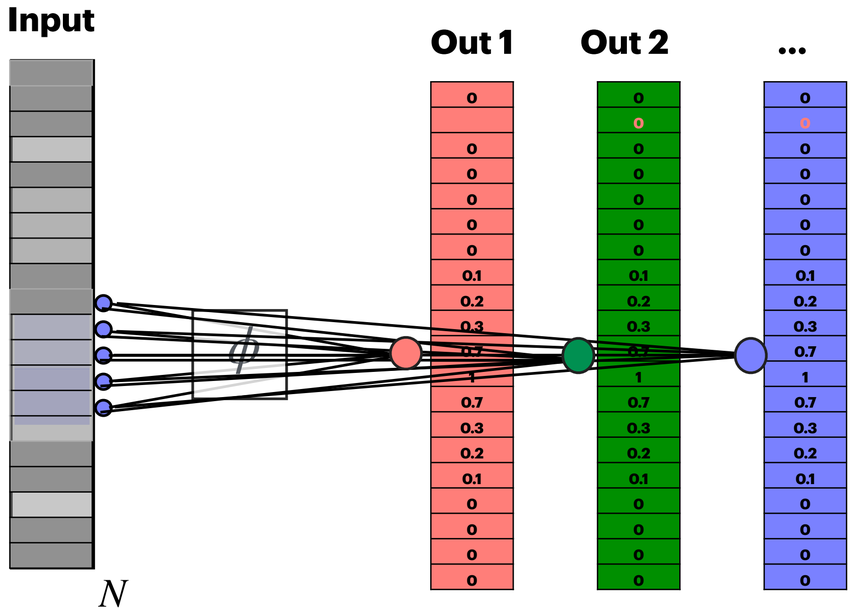
- This is called **Weight Sharing**, which is an important concept

- What if we have color? Multiple input channels, or even multiple output channel?

Map multiple in channels to on out channel



Map single in channels to multiple out channel



To conclude:

$$F(x) = \phi \star x$$

was a Linear Convolutional Operator. Now it is time to introduce:

Concept 2: A Convolutional Block

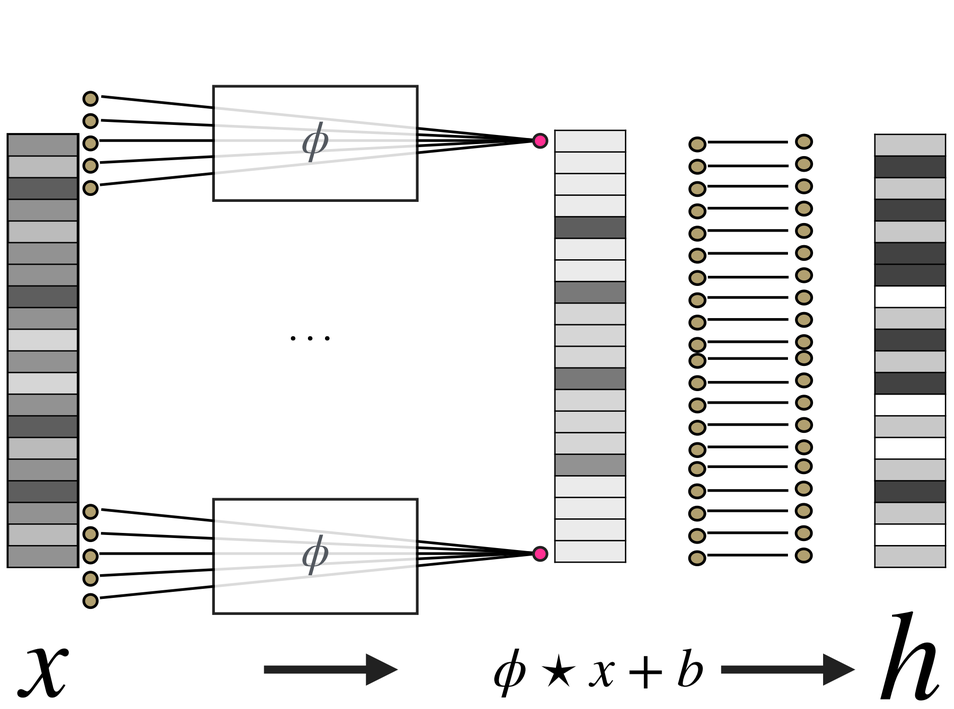
- **A convolutional Block** is built from:

a) the convolutional operator $F(x)$, followed by bias (this is usually called a convolutional Layer)

$$\phi \star x + b = F(x) + b$$

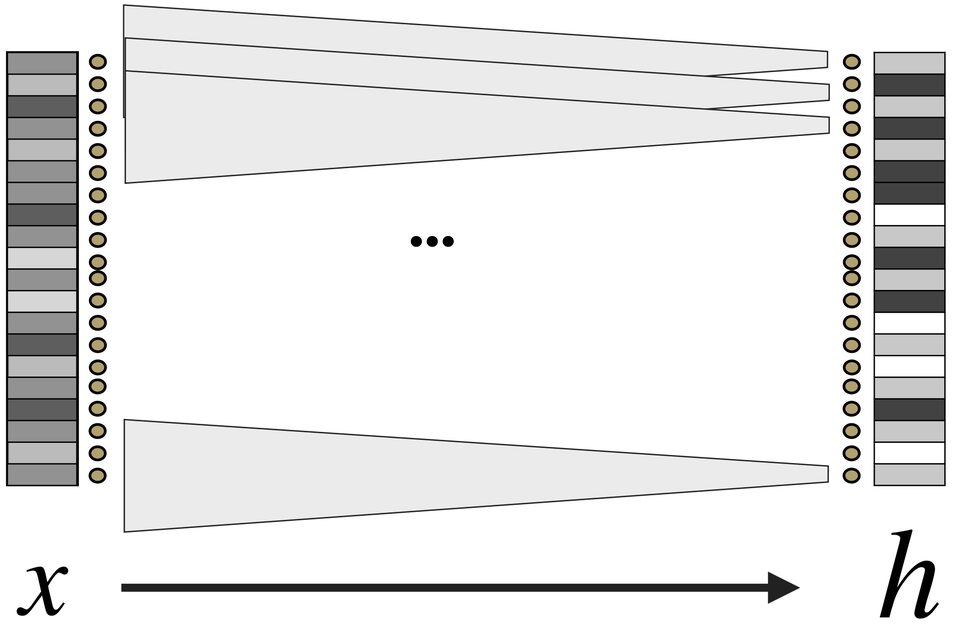
b) and then usually a nonlinearity σ, (usually ReLU):

$$h = \sigma(\phi \star x + b) = \sigma(F(x) + b)$$



Or simply:

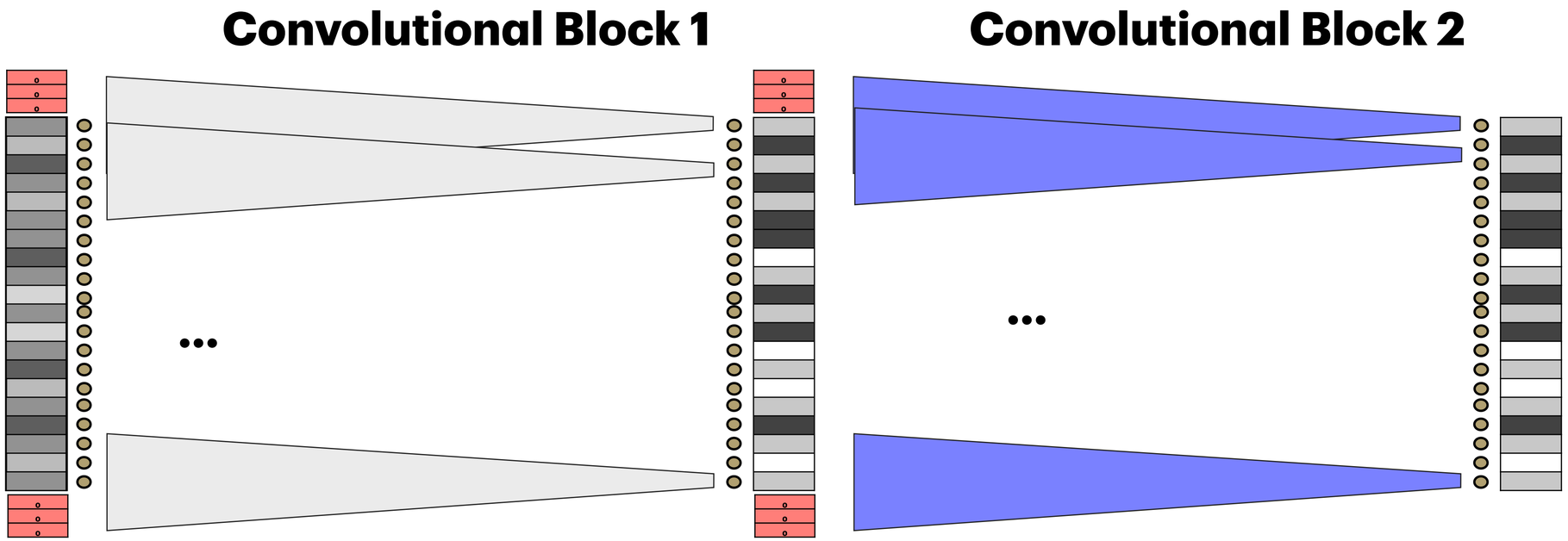
Convolutional Block



!! Note that the weights (w inside ϕ) and biases (b) are learnable via backprop

- **A note about Zero Padding:**

Padding adds **extra values (usually zeros)** around the input before applying the kernel.
We often use it to keep the output grid the same size as the input grid.



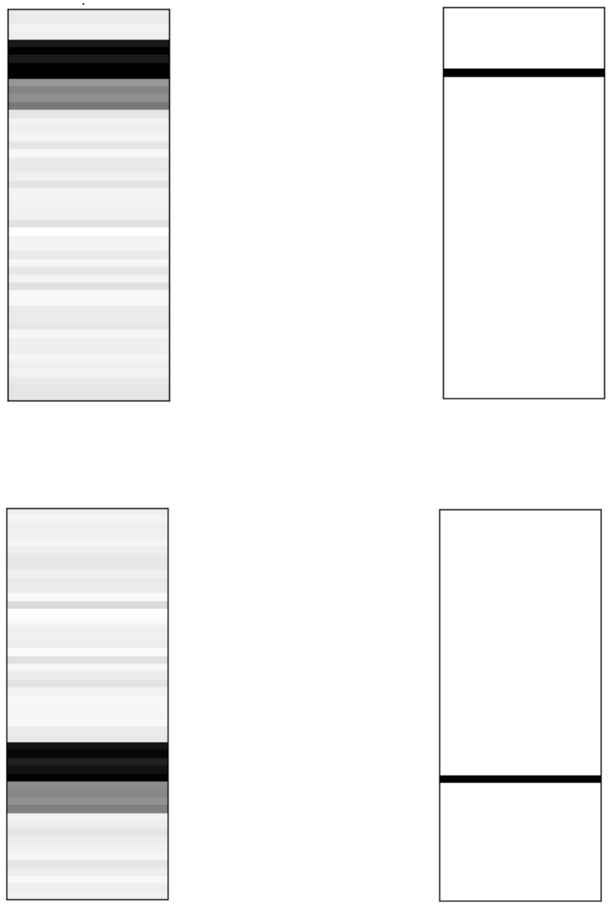
‖ Pause Point: **Time to practice**

- We built a simple supervised 1D grid dataset for you:
  - each input contains one local “tree” pattern,
  - and the target is its center-response map.

- Now we define and train the simplest possible CNN: one convolutional layer with a single learnable kernel.

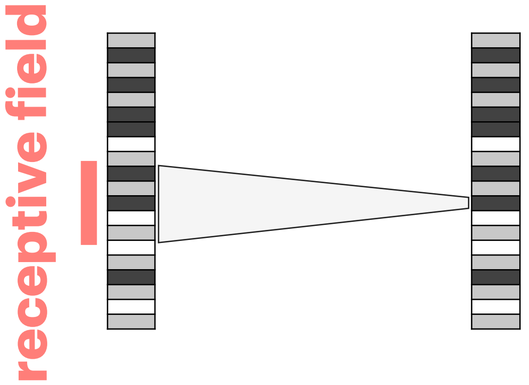
- Then we inspect two things: can it predict the center, and what pattern does the learned kernel capture?

Input: $x_i \in X$ Target: $y_i \in Y$

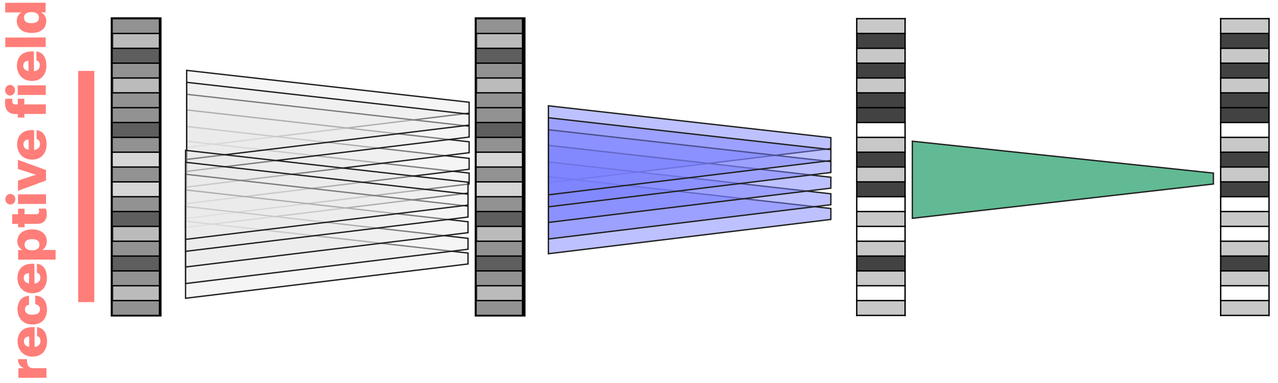


Concept 3: The receptive field:

- Each output position in a **Conv Blcok** only "sees" a window of size k from the input. This window is called the **receptive field**.



- When we stack Conv Blcoks, the receptive field of deeper layers grows.

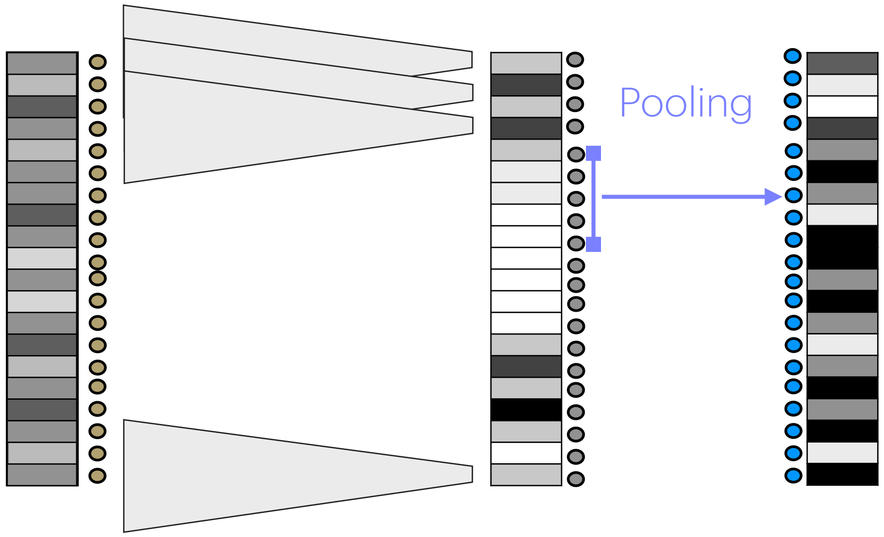


- There are two ways to control the receptive field size:
    1. Kernel size k: a larger kernel sees more of the input per layer directly
    2. ***Depth***: stacking more layers grows the receptive field indirectly, while keeping each individual kernel small
- In practice CNNs prefer small k (e.g. 3) and more depth: this gives the same receptive field as a large kernel, with fewer parameters and more nonlinearities in between, which <u>increases expressive power</u>.

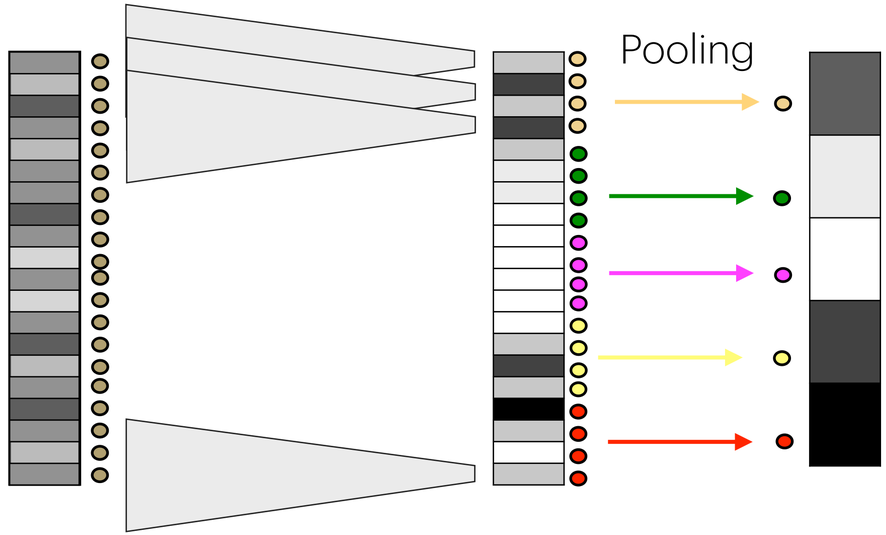
Concept 4: Downsampling (via pooling and stride)

- After one or several conv blocks, we often apply a pooling operation.
- It reduces the spatial size of the feature map by summarizing local regions into single values. The most common type is **max pooling**. it take a window of size p and keep only the maximum value. The output is p times smaller than the input.
- Pooling also contributes a degree of robustness...

Pooling (p=5) with stride=1 (No downsampling)

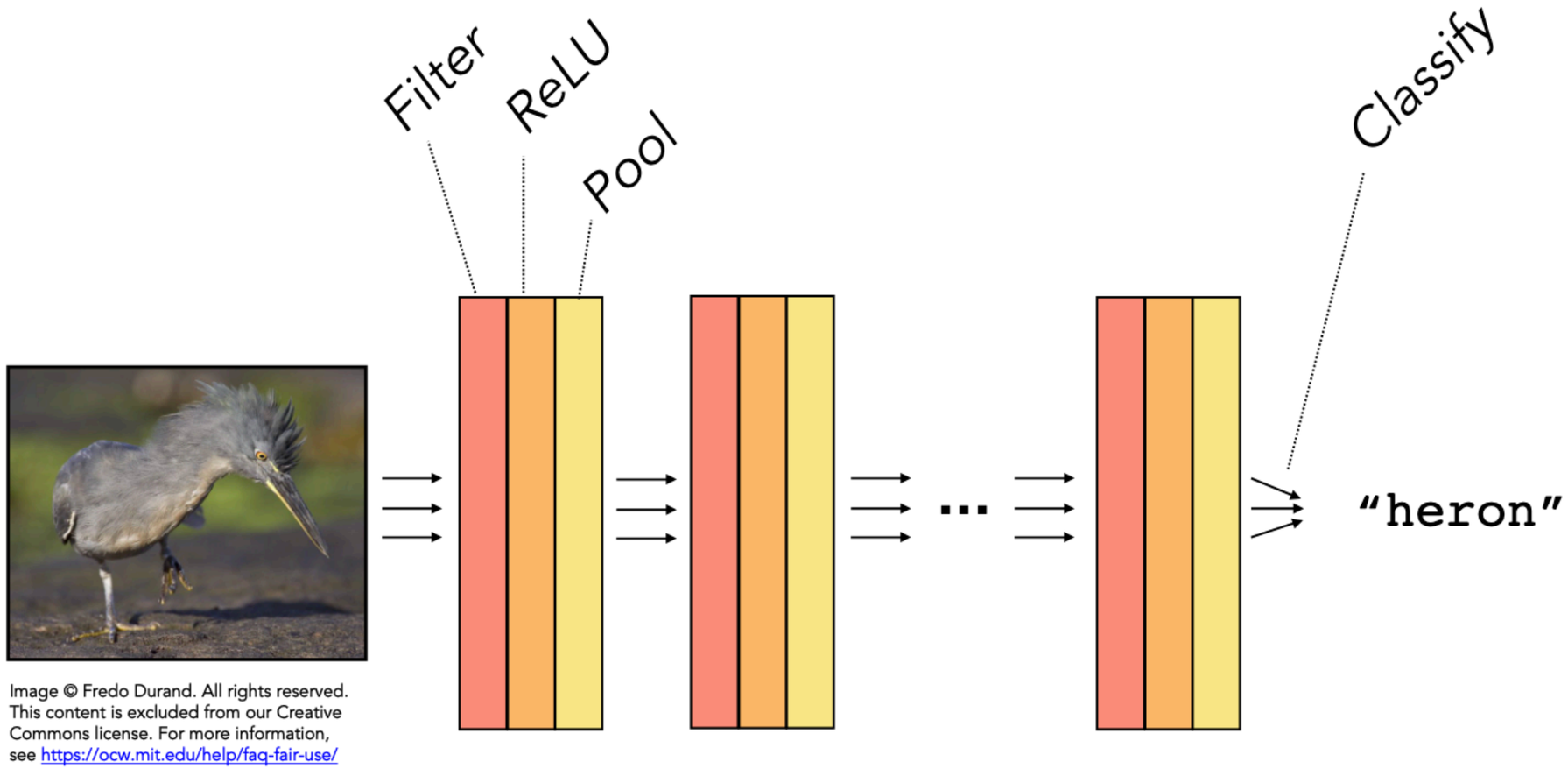


Pooling (p=4) with stride=4 (downsampling)

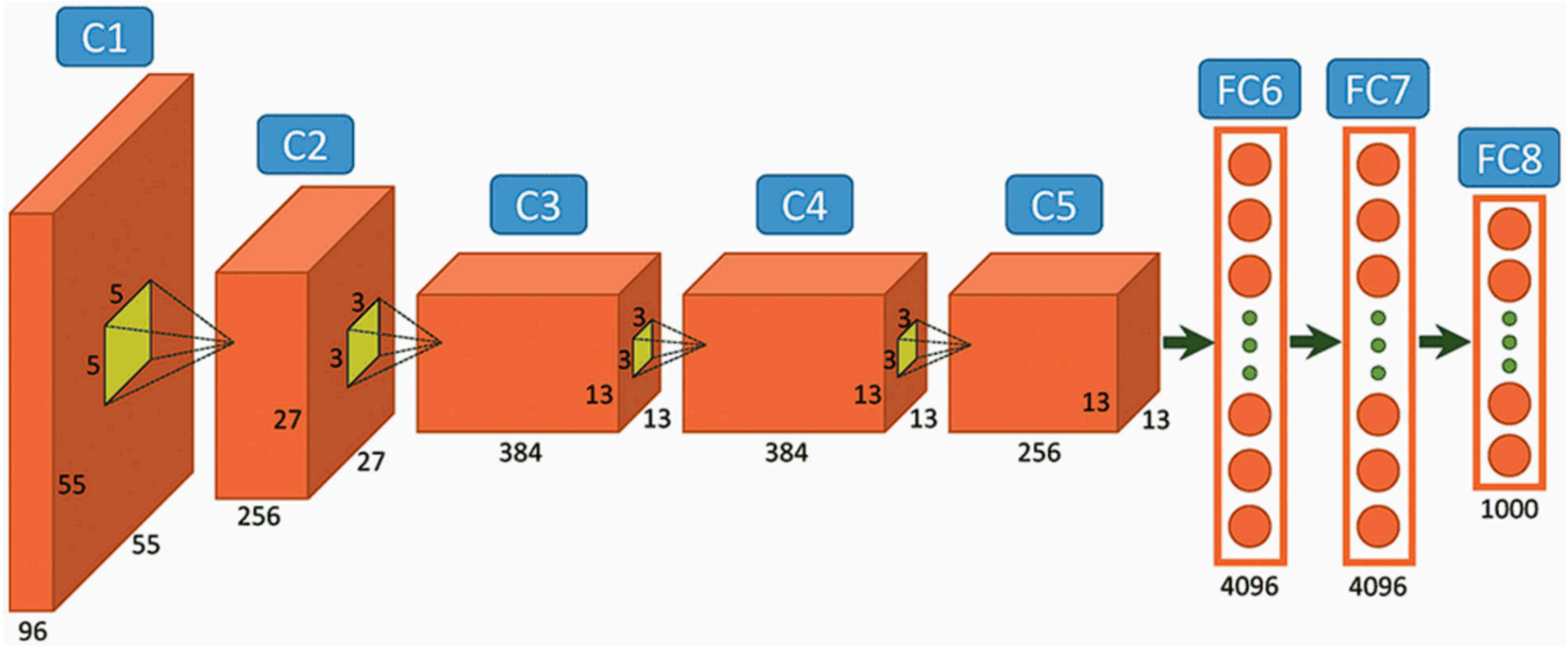


Popular CNN Architectures

Computation in a neural net

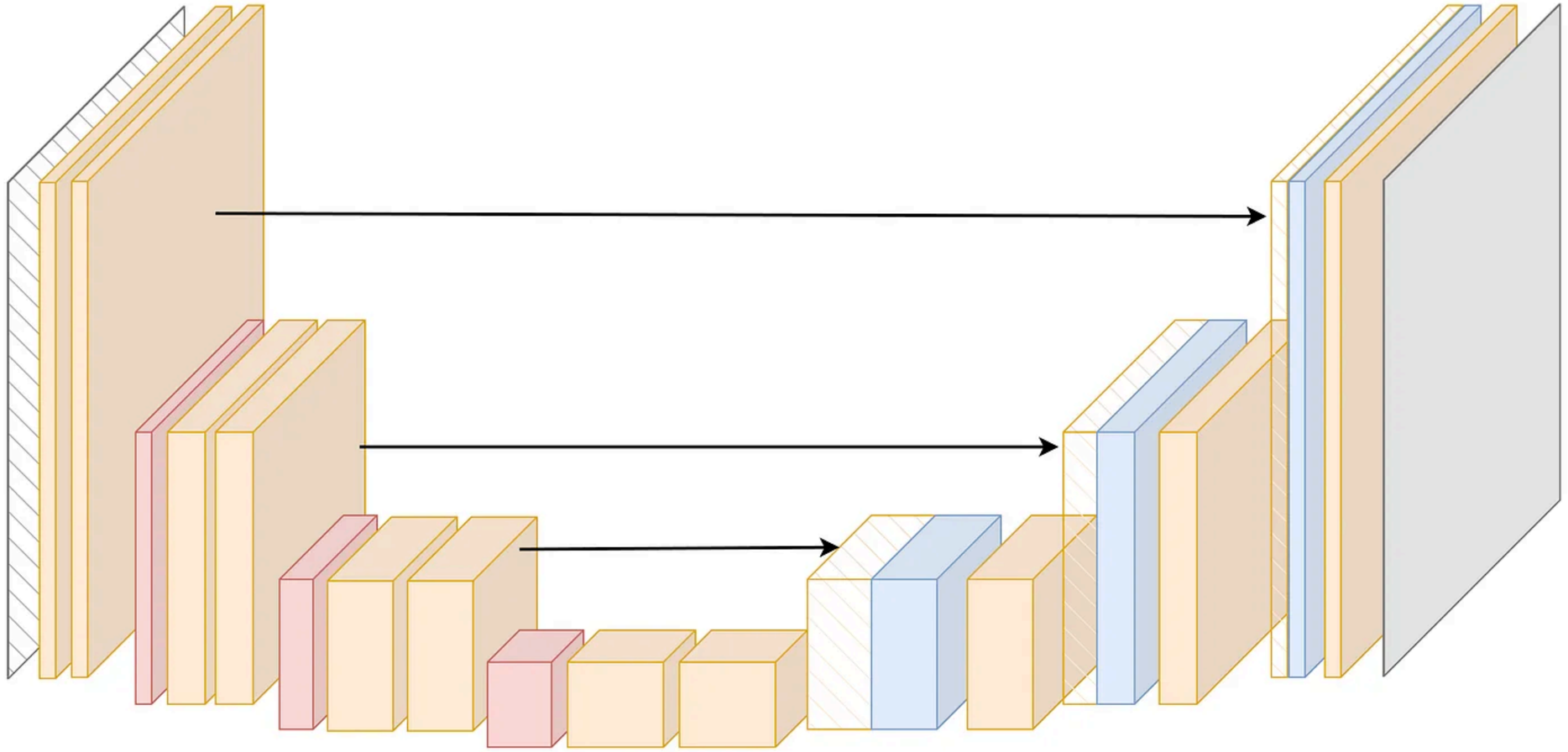


The AlexNet Architecture

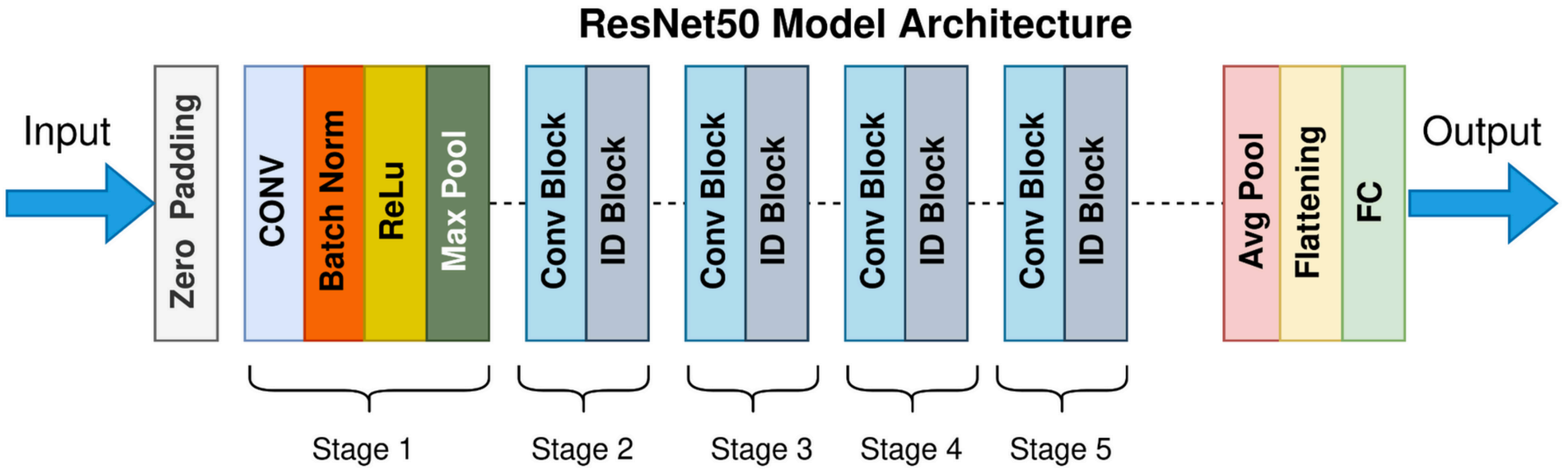


https://cdn.techscience.cn/ueditor/files/iasc/TSP_IASC-36-1/TSP_IASC_30125/TSP_IASC_30125.pdf

U-Net



https://medium.com/data-science/u-net-explained-understanding-its-image-segmentation-architecture-56e4842e313a



Resnet-50 Model architecture

Recipe Card: **Convolutional Neural Network (CNN)**

Type of Learning: Supervised
Data set: Grids / grid-structured data

Data

1- Load libraries: Basics + PyTorch

2-Load your grid Dataset D**:**

-Make sure D is a collection of input–label pairs:

$$D[i] = (X_i, Y_i)$$

A) $X_i \in \mathbb{R}^{C_{in} \times n_1 \times \cdots \times n_d}$

where: C_{in}= number input of channels, d= grid dimension, $n_1, \ldots, n_d$ = grid sizes

(Example: 1D grid: $X_i \in \mathbb{R}^{C \times N}$, 2D grid: $X_i \in \mathbb{R}^{C \times H \times W}$)

B) Y_i depends on the task: e.g. classification: $Y_i \in \{1, \ldots, K\}$

grid-to-grid: $Y_i \in \mathbb{R}^{C_{out} \times n_1 \times \cdots \times n_d}$

3- Set up the DataLoader for training / validation / testing:

-Split the dataset into train / validation / test

-Load each split with torch.utils.data.DataLoader

-For grid-structured data, batching is usually direct if all samples have the same shape

NN model

4- Build the CNN neural network as a Module

-A CNN learns local filters (kernels) and applies them across the grid. Then combines them to find relational aspects.

-A simple and common CNN structure is:

Input $X_i \rightarrow [(\textbf{Conv}) \rightarrow (\textbf{ReLU}) \rightarrow \textbf{Pool}] \rightarrow \cdots \rightarrow [(\textbf{Conv}) \rightarrow (\textbf{ReLU}) \rightarrow \textbf{Pool}] \rightarrow \textbf{Head} \rightarrow \hat{Y}_i$

-Optional additions: a. BatchNorm b.Max/Avg poolng c.Dropout

-The final Head depends on the task:

i. classification: Flatten or GlobalPool → → Linear → → logits.

ii.dense prediction: final Conv layer outputs a grid / map

iii. localization: either predict coordinates directly, or predict a response map

Training

5-Define a proper loss function based on Y_i

-If Y_i is continuous, then use MSELoss or L1Loss

- e.g. regression or continuous grid-to-grid prediction

-If Y_i is categorical, use CrossEntropyLoss

- e.g. classification or per-grid-cell classification
- feed raw logits directly to the loss

-If Y_i is binary, use BCEWithLogitsLoss

- e.g. binary mask or response map
- feed raw logits directly to the loss

-Add regularization to prevent overfitting

-The loss will be the sum of all relevant losses

6-Define the train and test functions:

-train:

a) it gets model, DataLoader, and optimizer(Adam)

b) set model.train() at the beginning

c) for loop over all batcher:**i-**Send data to device **ii-**optimizer.zero_grad() **iii-**Calculate loss **iv-**Back propagate via loss.backward() **v.** optimizer.step() to update

d) at the end, test function return loss

-test: same as train but, set model.eval() and backer

7-Define training and run the training:

-Choose number of epochs, For each epoch:

a) run train() for one epoch on the training loader

b)run test() on thetraining/validation loader to get losses

Read more

A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks., in: Nips, 2012: pp. 1097–1105.

K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition., in: Cvpr, 2016: pp. 770–778.

https://visionbook.mit.edu/convolutional_neural_nets.html