ECE 1508: Applied Deep Learning

Chapter 4: Convolutional Neural Networks

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2025

Computer Vision

Computer vision has been a fundamental problem in machine learning

The aim is to design a machine that can recognize patterns in visual contents

---

Long study on biological vision systems has been conducted

- *Hubel and Wiesel studied visual cortex in late 1950s*
  - ↳ *They won Nobel Prize in 1981 for their discoveries*
- *Inspired by that study Fukushima introduced Neocognitron in 1979*
  - ↳ *It was a convolutional NN for unsupervised learning*
- *Yann LeCun proposed LeNet for supervised learning in 1989*
  - ↳ *This is pretty much the beginning of modern CNNs*

Convolutional NNs: *CNNs*

Let's make an agreement: *though we all know CNN News Channel we also say*

Convolutional NN $\equiv$ *CNN*

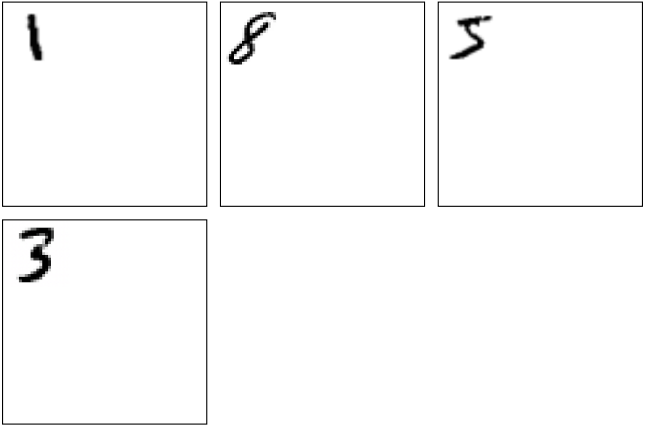
CNN

CNN is an FNN which in addition to standard fully-connected layers uses convolutional and pooling layers for feature extraction

+ *But, what is a convolutional layer? What is a pooling layer? What do you mean by feature extraction?*
– Well! We get there soon! But first *let's see what the main motivation is*

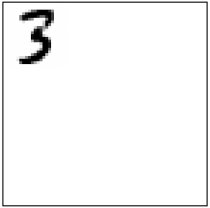
Motivational Example: *Pattern Recognition*

We intend to train an NN that classifies MNIST dataset with one difference to our earlier classification problem: *we assume that MNIST images are now included in a larger white background*



Motivational Example: *Pattern Recognition*

+ *Can't we simply use standard fully-connected FNN?*
– Sure! We can



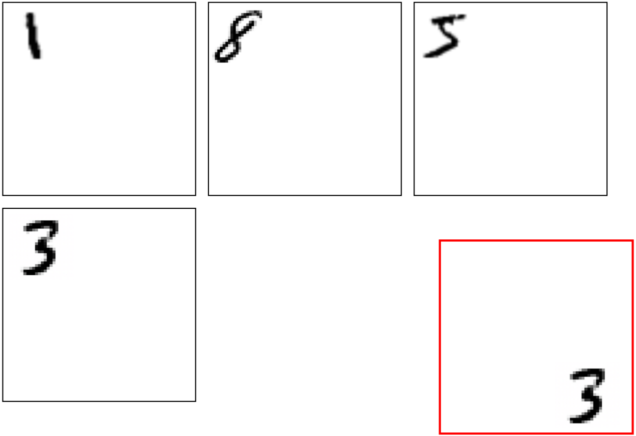
Say the photos with background are 3 times larger in height and width

- *Images are now 84 × 84 ≡ we have 7056 pixels*
- *We should have input layer with 7056 pixels and train the NN with MNIST*

To do the training, we should first convert our MNIST images into new larger-size images

Motivational Example: *Pattern Recognition*

We do this by zero-padding: all images after conversion lie on top left corner



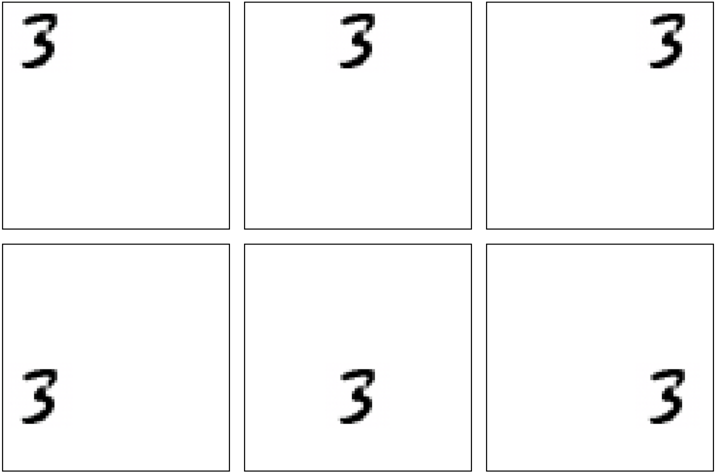
Do you think after training, NN classifies “3” in lower right corner correctly? No!

Motivational Example: *Pattern Recognition*

+ *Why does this happen?*
– Our NN is trained *to only look at top left corner*, it will *miss information anywhere else including in lower right corner*
+ *Can we do anything about it?*
– Yes! We learned it in the last chapter: *Data Augmentation*

We shift MNIST images in the large background left and right, up and down and add all those shifts with same label to the dataset

Motivational Example: *Recognition with Augmented Data*



We should get too many of them!

Using a *Trained FNN*

+ *But it sounds like too much work and computation!*
– Yes! It is! and frankly speaking *it is not worth it!*

Many scientist noted that *our brain doesn't work like that*

once we learn "3" we can recognize it anywhere in our vision!

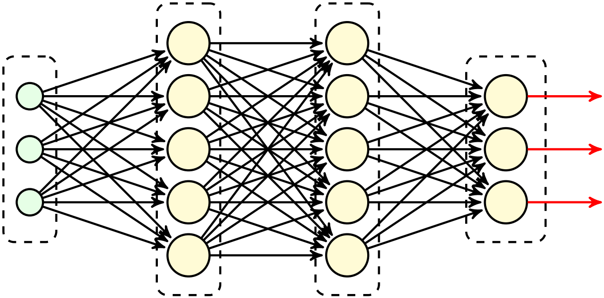
We may initially note that

1. *Our brain doesn't process the visual field as the whole*
2. *It searches for patterns in smaller fields within our vision*
   ↳ *It constructs a pattern for "3" through training*
   ↳ *After training, it scans any visual field to see if it finds that pattern*

Let's try realizing it with a NN!

Using a *Trained FNN*

Let's assume we have trained the following FNN on MNIST

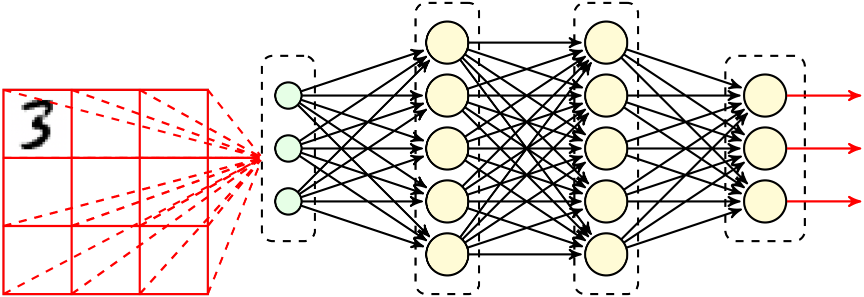


- *It gets a 784-pixel image as input*
- *It passes it through three fully-connected layers*
- *It returns the class of the image*
  - ↳ *If it doesn't find a class it returns* $\varnothing$

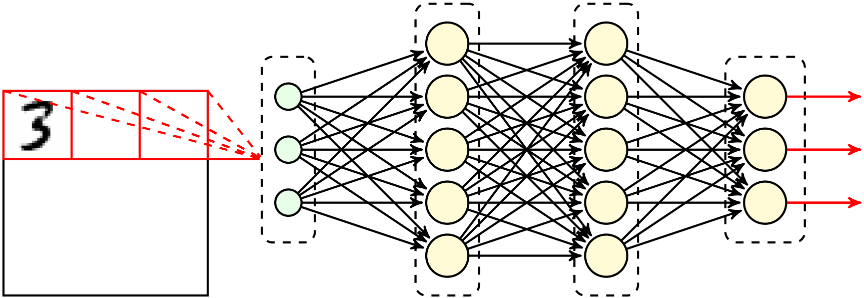
Scanning via *MNIST Trained FNN*

We can mimic *what our brain does*

we scan the larger image with background



Scanning via *MNIST Trained FNN*



We go through windows for size 28×28

1. *At each window, we give it 784 pixels to the FNN to classify*
   - ↳ *If we find a class: we save the class and return* `True`
   - ↳ If we don't find a class: we return `False`
2. *We compute OR of outputs for all windows*
   - ↳ *If* `True`: we return the saved class
   - ↳ If `False`: we return $\emptyset$

CNNs: *Scanning via Shared Weights*

The above example is a *simple CNN*

- *This CNN extracts features from the image using a 28×28 filter*
  - ↳ *the filter's weights are those given by the first layer of trained* FNN
  - ↳ *the features are affine values calculated in first layer of trained FNN*
- *The scanning procedure has a specific name: convolution*
  - ↳ *it goes through the image by sliding over it via a smaller window*
  - ↳ *it determines an affine transform of smaller subsets of pixels*
- *We can look at it as a giant FNN with shared* weights
  - ↳ *each pixel is connected to the next layer via affine transform*
  - ↳ *this affine transform has the same weights for many pixels*
  - ↳ *not every feature depends on every pixels*
  - ↳ *the first layer is not fully-connected: it's locally-connected*

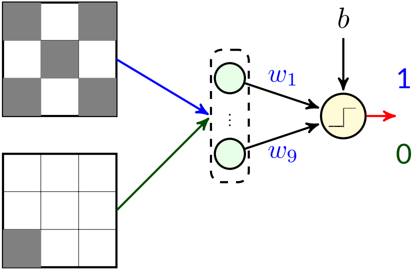
Let's make our understanding deeper by making our first CNN!

Recognizing X

In Assignment 1, we trained a perceptron with 9 inputs

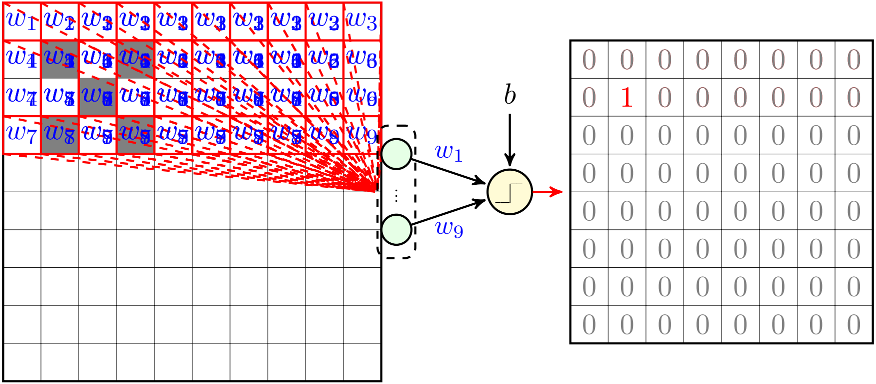
it gets a 3×3 image and says whether it is "X" or not

Assume we have weights and bias: *we want to recognize "X" in a larger image*



Recognizing X

We follow our *scanning idea to recognize* *3 × 3 "X"* *in a* *10 × 10 image:* *we put the weights on a* *3 × 3 filter* *and slide it over the* *image*



We slide the filter with stride 1 and save the outputs of perceptron on a map

Recognizing X

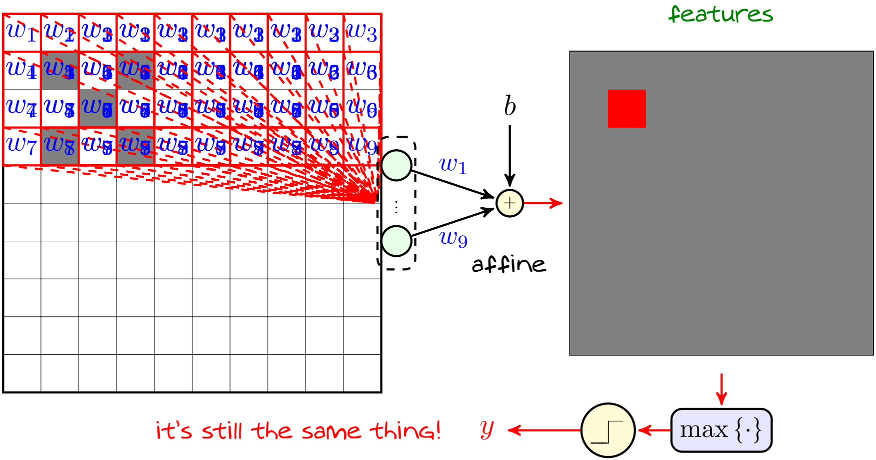
It's enough to have only a single 1 to recognize "X"

0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

We therefore return the OR of all entries in the above map

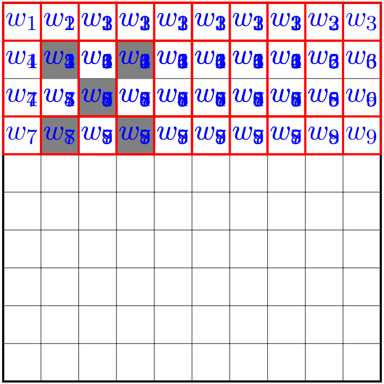
Recognizing X: *Convolution*

We really don't need to determine the activation after each scan: *we could only save the affine transforms in the map*



Convolution *with Stride 1*

image



filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

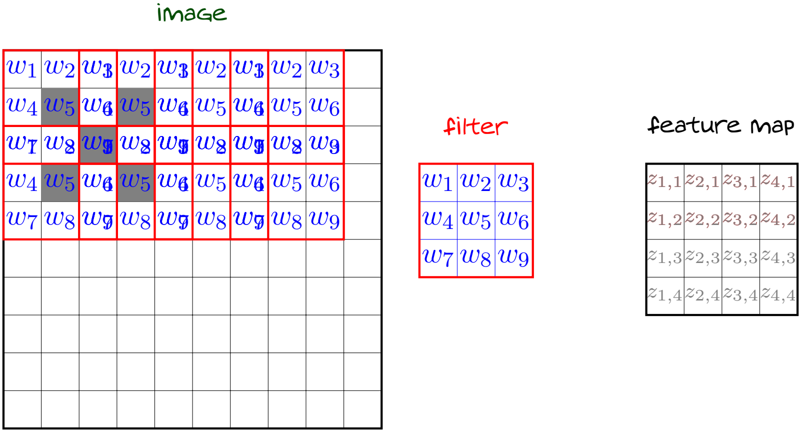
feature map

$z_{1,1}$	$z_{2,1}$	$z_{3,1}$	$z_{4,1}$	$z_{5,1}$	$z_{6,1}$	$z_{7,1}$	$z_{8,1}$
$z_{1,2}$	$z_{2,2}$	$z_{3,2}$	$z_{4,2}$	$z_{5,2}$	$z_{6,2}$	$z_{7,2}$	$z_{8,2}$
$z_{1,3}$	$z_{2,3}$	$z_{3,3}$	$z_{4,3}$	$z_{5,3}$	$z_{6,3}$	$z_{7,3}$	$z_{8,3}$
$z_{1,4}$	$z_{2,4}$	$z_{3,4}$	$z_{4,4}$	$z_{5,4}$	$z_{6,4}$	$z_{7,4}$	$z_{8,4}$
$z_{1,5}$	$z_{2,5}$	$z_{3,5}$	$z_{4,5}$	$z_{5,5}$	$z_{6,5}$	$z_{7,5}$	$z_{8,5}$
$z_{1,6}$	$z_{2,6}$	$z_{3,6}$	$z_{4,6}$	$z_{5,6}$	$z_{6,6}$	$z_{7,6}$	$z_{8,6}$
$z_{1,7}$	$z_{2,7}$	$z_{3,7}$	$z_{4,7}$	$z_{5,7}$	$z_{6,7}$	$z_{7,7}$	$z_{8,7}$
$z_{1,8}$	$z_{2,8}$	$z_{3,8}$	$z_{4,8}$	$z_{5,8}$	$z_{6,8}$	$z_{7,8}$	$z_{8,8}$

The above operation is convolution and we show it as below

$$\text{feature map} = \mathrm{Conv}\left(\text{image} \middle| \text{filter}, \text{stride} = 1\right)$$

Convolution *with Stride 2*



We could also play with the stride $\equiv$ the step-size by which we move filter

$$\text{feature map} = \mathrm{Conv}\left(\text{image} \middle| \text{filter}, \text{stride} = 2\right)$$

Convolution *with Stride* S

Let's formulate the convolution for a general filter: *assume* $\mathbf{W} \in \mathbb{R}^{F \times F}$ *be a filter, we also call it kernel. Let* $\mathbf{X} \in \mathbb{R}^{N \times N}$ *be pixel matrix of the image. We want to find the output feature map, i.e.,*

$$\mathbf{Z} = \mathrm{Conv}\left(\mathbf{X} | \mathbf{W}, \text{stride} = S\right)$$

It's enough to find the corresponding sub-matrix for each entry of $\mathbf{Z}$

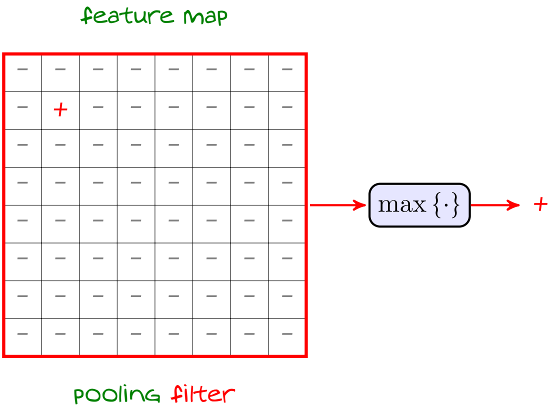
$$\mathbf{Z}[i, j] = \mathtt{sum}\left(\mathbf{W} \odot \mathbf{X}_{i,j}\right)$$

where $\mathbf{X}_{i,j}$ *is the corresponding* $F \times F$ *sub-matrix, i.e.,*

$$\mathbf{X}_{i,j} = \mathbf{X}\left[1 + (i-1) S : F + (i-1) S, 1 + (j-1) S : F + (j-1) S\right]$$

Recognizing X: *Pooling*

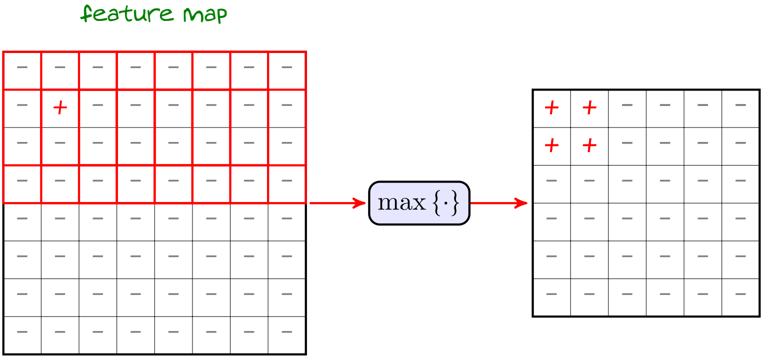
The next operation we did is called pooling



This is however not conventional to have a pooling filter of different size

Pooling: *Max Pooling with Stride 1*

The convention is to use the same filter size as used in convolution layer

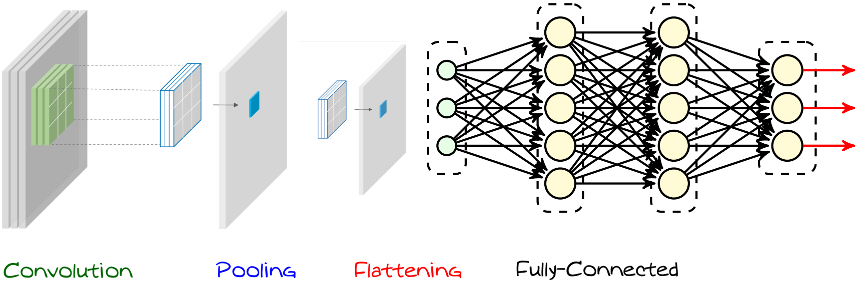


We can now give the feature map after pooling to a fully-connected FNN: this is a feature vector of reduced size!

We can repeat convolution and pooling over and over

CNN: *Simple Architecture*

Our simple CNN looks like this

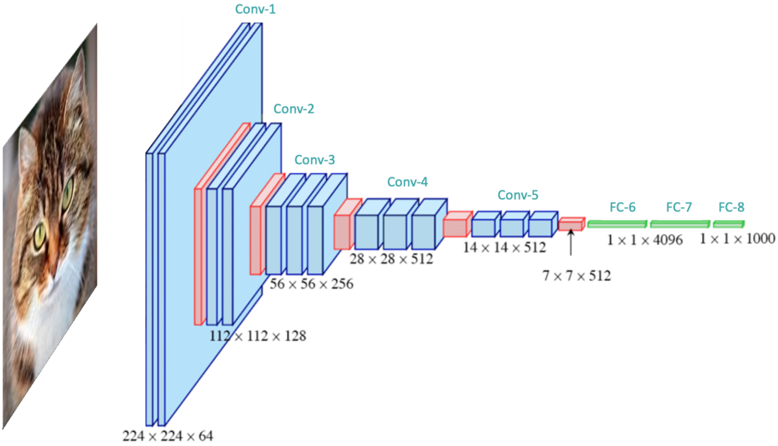


This is a general architecture for CNNs, but of course we go deeper!

- *We have more convolutional and pooling layers*
- *We do high-dimensional convolutions and more advanced poolings*

CNN: *Realistic Architectures*

For instance the famous VGG-16 architecture looks like below



CNN: *Connections to Biological Vision*

Though it's artificially developed as a computation model: it is related to the initial model developed for description of biological vision