# CNNs

① Get data ready

② Build or pick up a pretrained model to suit your problem
- Build a training loop
- pick a loss function & optimizer

③ Fit the model to the data and make a prediction

④ Evaluate the model

⑤ Improve through experimentation

⑥ Save and reload your trained model

SHIFT + CMD + SPACE to read the docstrings

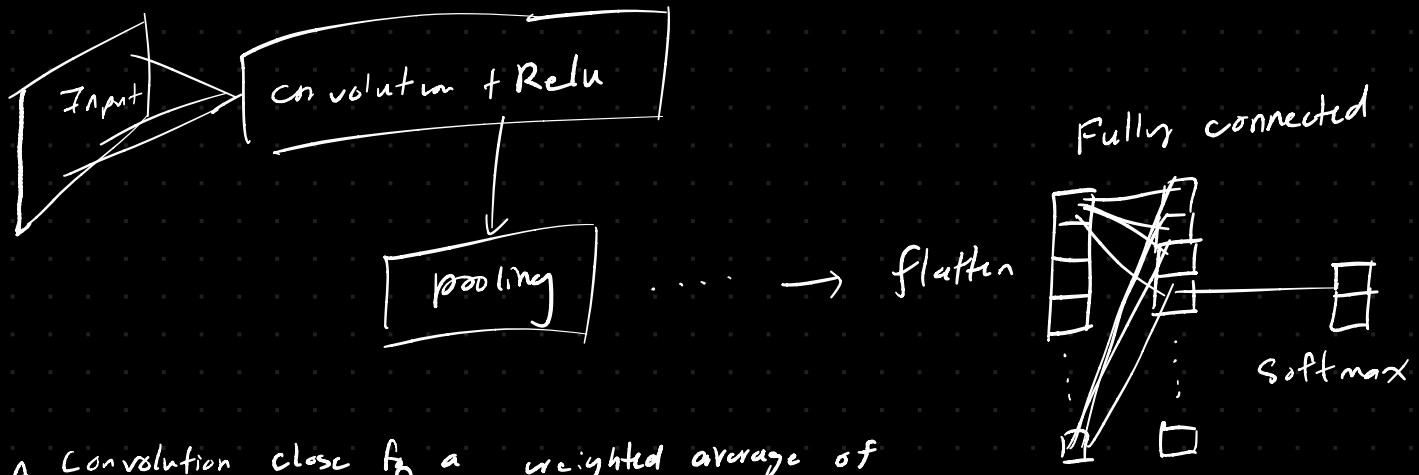
# CNN Slideshow - 03/17/24

A CNN is a type of neural networks that assumes the input is an image.

The layers of a CNN have neurons arranged in 3 dimensions

1. height
2. width
3. depth

Convolutions use a kernel (matrix with learnable weights, the kernel is then convolved with neurons in that layer)




A Convolution close to a weighted average of values in a specific region.

$$g(x,y) = w * f(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} w(i,j)\, f(x-i, y-j)$$

$w$ is the filter kernel

All elements are considered by $-a \leq i \leq a$, $-b \leq j \leq b$

## Hyper parameters

Depth — # of filters

Stride — # of pixels to slide the kernel

Zero-padding → controls output size

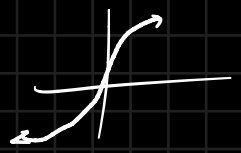
## Non-linearity layers

Sigmoid

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

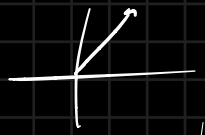
tanh(x)

tanh(x)



ReLU

max (0, x)



Leaky ReLU

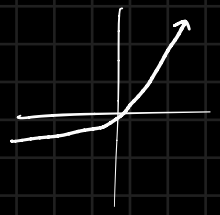
max (0.1x, x)

maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



## pooling layers

Max pooling → outputs the maximum in the area

pooling reduces # of parameters and the training cost

## Fully connected layer

The last layer of a CNN

Full connections to all activations in the previous layer.

Softmax → normalizes output to be between 0 and 1.

→ represents the probability that the input belongs to a specific class.

# CS231 - Convolutional Neural Networks

03/19/24

Input → neuron → dot product → non-linearity

out put → single differentiable score / classes score

Loss functions in a CNN → SVM + softmax → $\sigma(z)_i = \frac{e^{z_i}}{\sum e^{z_j}}$
(support vector machine)

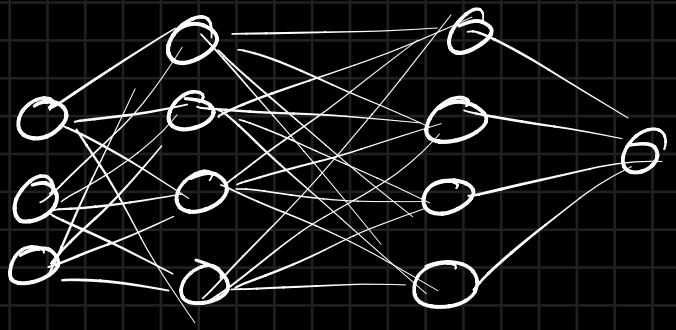
There is the explicit assumption that the input is an image → reduce the # of parameters in the network.

## Regular Neural Network

Neural networks receive a single input and then this input is passed onto the "hidden" layers of the network, transforming the output in the process. Each layer consists of neurons, that function independently.

Each neuron in each layer is connected fully to the neurons of the previous layer.

The last layer is called the output layer and allows the user to see class scores

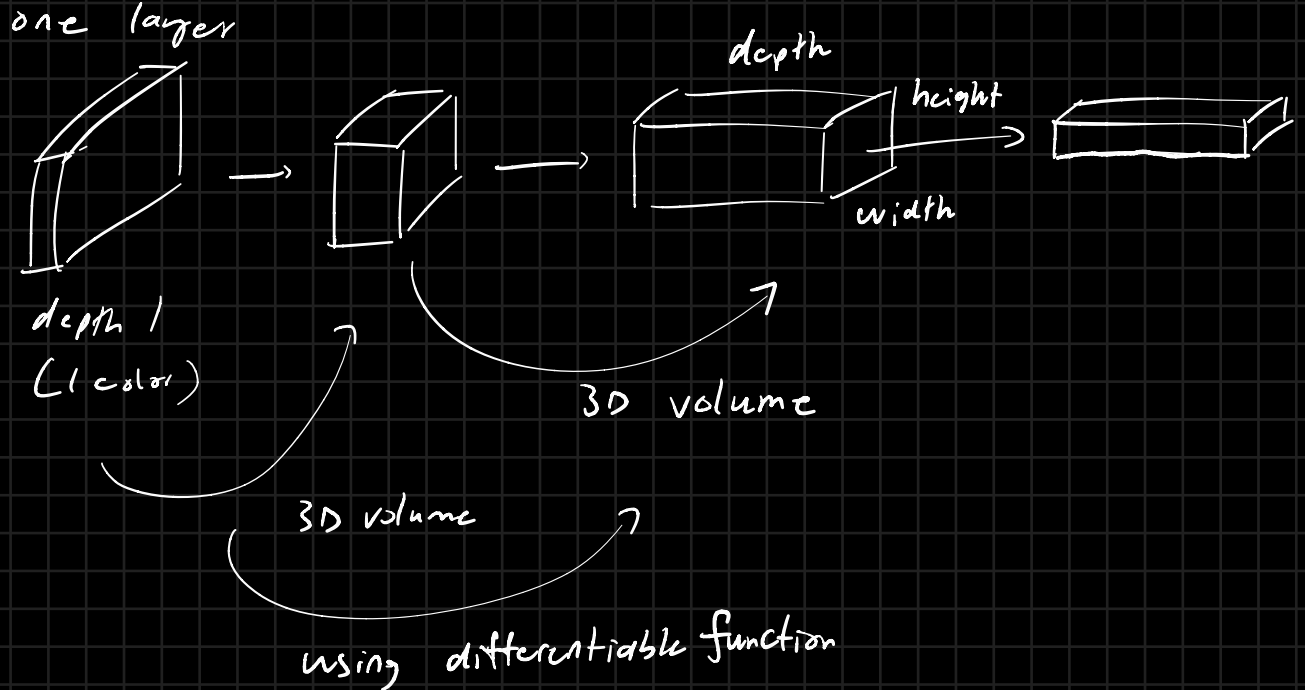


Regular neural networks require too many weights if the input size is a large image.

→ Also don't want a huge # of parameters to avoid overfitting

CNN architecture is constrained as input has a height ($x$), weight ($y$), and depth ($z = 3$)

Instead of full connections between hidden layers, neurons are connected to only a small subset of the previous layer's neurons.



3 types of layers in a CNN:

convolutional layers → pooling layer → fully connected layer

Input → conv → pool → FC

Input holds the original pixel data

conv layer will take in a subset of input and output corresponding values

ReLU applies $\max(0, x)$ to each neuron

pooling performs a down sampling operation along the spatial dimensions along spatial dimensions, scaling down

FC layer computes the class scores and has height 1, width 1, and depth # of classes.

conv / FC perform transformations of values using weights and biases

parameters with conv/FC are updated using gradient descent.

| | |
|---|---|
| Input | 32x32x3 |
| conv | 32x32x12 |
| ReLu | 32x32x12 |
| pool | 16x16x12 |
| FC | 1x1x10 |

Conv / FC → function of weights & biases

RELU / pool → fixed function

input volume → output volume

Conv layer: Set of learnable filters

Filters => extend RGB height, but less width and length

eg. $5 \times 5 \times 3$

Slide / convolve each filter across the width and height of the input volume and compute dot products between overlapping entries at any position of the filter.

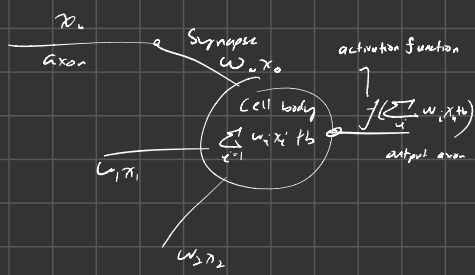
=> produces 2 dimension activation map / matrix that gives responses of that filter at every position

The goal is that through training, the filters learn to activate when they see some visual feature.

with a set of filters, each produce a separate 2 dimensional activation map / matrix

Stacking along dimensions of height, to produce the output 3D matrix

as a brain analogy, every output in the output layer is a neuron that only looks at a small region of the image and shares learnable parameters with all neurons to the left and right spatially




Small subset of image → hyperparameter
aka receptive field / filter size

connectivity along the depth = depth of input volume (RGB)

not always RGB
simply height of input volume

Each filter has one bias only ✱

Three hyperparameters control the size of the output volume: Depth, stride, and zero padding

Depth of the output volume (# of filters, each learning to looks for something different in the input)
Neurons looking at the same region are called a depth column

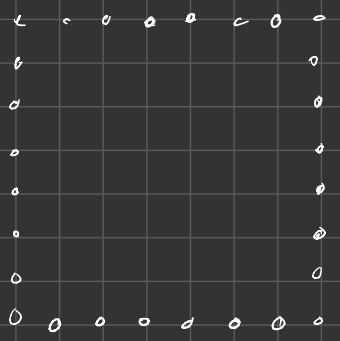
Stride: How far we slide the filter.

padding: zeros around the border to control spatial size of the output

# Spatial size of output volume

Input volume (W)
Field / filter size (F)
Stride (S)
Zero padding (P)

$$\frac{W - F + 2P}{S} + 1$$



Stride must be picked to make sure the convolution is symmetric

Since the convolution / filter is a square with its height and width,

$\frac{W-F+2P}{S} + 1$ must be divisible by F

If we set zero padding to be $P = \frac{(F-1)}{2}$ when stride = 1, then, the input & output volume have the same sizes

Chosen stride must allow for $\frac{W-F+2P}{S} + 1$ to be an integer

If one feature is useful to compute at some spatial position $x_1, y_1$, then it should also be useful to compute at a different position $x_2, y_2$

denote a depth slice (slice of height)

constrain neurons in each depth slice to use the same weights and bias. Set of weights, one bias => per slice (filter)

During back propagation, every neuron in the volume will compute the gradient for its weights but these gradients will be added up across each depth slice and only update a single set of weights per slice.

If all neurons in a depth slice use the same weights vector, then the forward pass of the convolutional layer is computed as a convolution of neurons weights with the input volume.

In Numpy,

Input x numpy array

A depth column at x,y would be the activations $x[x, y, :]$

A depth slice of depth d would be the activations $x[:, :, d]$

Conv layer example

Input x has shape (11, 11, 4)

If we use 0 padding and a filter size of 3 with stride 2

Output volume: $\frac{11-5}{2+1} = 4$

Activation map:

element wise

```
v[0,0,0] = np.sum(x[:5, :5, :] * w0) + b0
v[1,0,0] = np.sum(x[2:7, :5, :] * w0) + b0
v[2,0,0] = np.sum(x[4:9, :5, :] * w0) + b0
v[3,0,0] = np.sum(x[6:11, :5, :] * w0) + b0
```

Same $w_0$ and $b_0$ (parameter sharing)

w0.shape = (5, 5, 4)

Computing dot product

Conv layer
accepts a volume of size $W_1 \times H_1 \times D_1$

Requires four hyper parameters

- # of filters K
- Their spatial extent F
- The stride S
- The amount of zero padding P

output volume $W_2 \times H_2 \times D_2$ where

$W_2 = ((W_1 - F + 2P)/S) + 1$
$H_2 = ((H_1 - F + 2P)/S) + 1$
$D_2 = K$

With parameter sharing, $F \times F \times D_1$ weights per filter for a total of

$F \times F \times D_1 \times K$ weights, and K biases

The dth depth slice is the result of performing a valid convolution with stride S, offset by dth bias

Common setting: $F = 3$, $S = 1$, $P = 1$

weights are shared across a slice, per filter

① ②

② if and only if ①

otherwise, weights are different

Biases are shared across a slice, per filter.

As matrix multiplication:

local regions are stretched out into columns in an operation called im2col
=> Column vector

eg. input = 227 × 227 × 3
Kernel = 11×11×3 at stride 4

11×11×3 = 363

$\frac{227-11}{4+1} = 55$ locations along both width and length

output matrix => $x\_col$ (363 × 3025)

(96 × 55 × 55)

Stretch out weights: 96 filters eg: (96 × 363) ↑ reshape

one convolution: perform np.dot($w\_row$, $x\_col$) => (96 × 3025)

Back propagation : Spatially flipped filters

Pooling layer => reduces spatial size

Resized using the max operation

Common: Size 2×2 filter, with stride 2 at every depth slice.

For input $W_1 \times H_1 \times D_1$

and spatial extent F

stride S

produced a volume of $W_2 \times H_2 \times D_2$

$$W_2 = (W_1 - F)/S + 1$$
$$H_2 = (H_1 - F)/S + 1$$
$$D_2 = D_1$$

Can also perform average pooling

Back propagation => routing gradients that had the highest value in the forward pass. Others are just zero.

Sometimes pooling is not used and larger stride replaces pooling to prevent over fitting

Fully connected

full connections to all activations in the previous layer as seen in regular neural networks.

=> Matrix multiplication with bias offset

For any conv layer, an FC matrix exists that can implement the same forward function (most entries would be zero)

Conversely, if we had a FC layer looking at $7 \times 7 \times 512$ input, and $K = 4096$, the conv layer would be $F = 7$, $P = 0$, $S = 1$, $K = 4096$

Filter size = input volume

output: $1 \times 1 \times 4096$

only 1 depth fits

eg. $224 \times 224 \times 3 \Rightarrow 7 \times 7 \times 512$

5 pooling layers

FC layers used and 1000

$K = 1000$

$F = 1 \rightarrow 1 \times 1 \times 1000$

FC -> conv : $7 \times 7 \times 512 \Rightarrow F = 7$ (above) $\Rightarrow 1 \times 1 \times 4096$

$K = 4096$

If we have a larger image and run a convolution,

our class output is not 1D any more

=> Take the average across extra dimensions at all spatial positions (as long as the initial convolution is legal)

Input -> $(\text{Conv} \to \text{Relu}) \times n \to \text{pool}) \times m \to (\text{FC} \to \text{ReLU}) \times K \to \text{FC}$

$n \geq 0 \quad m \geq 0 \quad K \geq 0 \quad K < 3$

Input $\to (\text{conv} \to \text{Relu} \to \text{pool}) \times 2 \to \text{FC} \to \text{ReLU} \to \text{FC}$

Stacks of conv filters are better than one large conv layer

More expressive with fewer parameters

# CNNs from Scratch

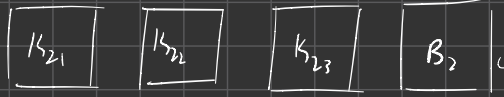
Convolve vs correlate

$I * K = I \star \text{rot}_{180}(K)$

3 dim block of data → depth = 3

Kernel 1 (3 dim as well)


Kernel 2 (3 dim)




biases

output


cross correlation

$$Y_1 = B_1 + x_1 \star k_{11} + x_2 \star k_{12} + x_3 \star k_{13} \dots x_n k_{1n}$$

$$Y_2 = B_2 + x_1 \star k_{21} + x_2 \star k_{22} + x_3 \star k_{23}$$

$$\vdots$$

$$Y_d = B + x_1 \star k_{d1} + x_2 \star k_{d2} + x_3 \star k_{ds} \dots x_n k_{dn}$$

Forward propogation

$$Y_i = B_i + \sum_{j=1}^{n} x_j \star K_{ij}, \quad i = 1 \dots d$$

$$Y = B + K \star x$$

cross correlation

dense layer

Back propagation

$$\frac{\partial C}{\partial Y_i} \begin{cases} \frac{\partial C}{\partial K_{ij}} & (1) \\ \frac{\partial C}{\partial B_i} & (2) \\ \frac{\partial C}{\partial x_j} & (3) \end{cases}$$

$$Y_i = B_i + \sum_{j=1}^{n} x_j * K_{ij}$$

Take one layer, and expand all correlations

$$\frac{\partial E}{\partial K_{11}} = \frac{\partial C}{\partial y_{11}} \frac{\partial y_{11}}{\partial K_{11}} + \frac{\partial C}{\partial y_{12}} \frac{\partial y_{12}}{\partial K_{11}} + \frac{\partial C}{\partial y_{21}} \frac{\partial y_{21}}{\partial K_{11}} + \frac{\partial C}{\partial y_{22}} \frac{\partial y_{22}}{\partial K_{11}}$$

$x_{11}$ $x_{12}$ $x_{21}$ $x_{12}$

$$\begin{bmatrix} \frac{\partial C}{\partial K_{11}} & \frac{\partial C}{\partial K_{12}} \\ \frac{\partial C}{\partial K_{21}} & \frac{\partial C}{\partial K_{22}} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} * \begin{bmatrix} \frac{\partial C}{\partial y_{11}} & \frac{\partial C}{y_{12}} \\ \frac{\partial C}{y_{21}} & \frac{\partial C}{y_{22}} \end{bmatrix}$$

$$\frac{\partial C}{\partial K} = x * \frac{\partial C}{\partial Y}$$

$$\begin{cases} Y_1 = B_1 + x_1 * K_{11} & \dots\dots & x_n * K_{1n} \\ Y_2 = B_2 + x_1 * K_{21} & \dots\dots & x_n * K_{2n} \\ \vdots \\ Y_d = B_d + x_1 * K_{d1} & \dots\dots & x_n * K_{dn} \end{cases}$$

If we want to calculate $\frac{\partial C}{\partial K_{21}}$ eg,

it only shows up once

$$\frac{\partial C}{\partial K_{21}} = x_1 * \frac{\partial C}{\partial Y_2}$$ (from last example)

$\frac{\partial C}{\partial B_i}$ :

$\frac{\partial C}{\partial b_{11}} = \frac{\partial C}{\partial y_{11}}$ (only appears once)

Expanding back,

$Y_1 = B_1 \ldots$

$Y_2 = B_2 \ldots$

$Y_3 = B_3 \ldots$

$\frac{\partial C}{\partial Y_i} = \frac{\partial C}{\partial B_i}$

$\frac{\partial C}{\partial x_j} \Rightarrow Y_i = B_i + x_1 * x_{i1}$

some $\frac{\partial C}{x_{ij}}$ have different # of terms

=> Full cross correlation => kernel has been rotated 180°

$$\frac{\partial C}{\partial \gamma} = \frac{\partial C}{\partial Y} * K$$

Now, for all $\gamma_i$,

$$\frac{\partial C}{\partial \gamma_1} = \frac{\partial C}{\partial Y_1} * K_{11} + \cdots + \frac{\partial C}{\partial Y_d} * K_{d1}$$

$$\frac{\partial C}{\partial \gamma_j} = \sum_{i=1}^{d} \frac{\partial C}{\partial Y_i} * K_{ij}$$

① pre process the images
② Tokenize the words

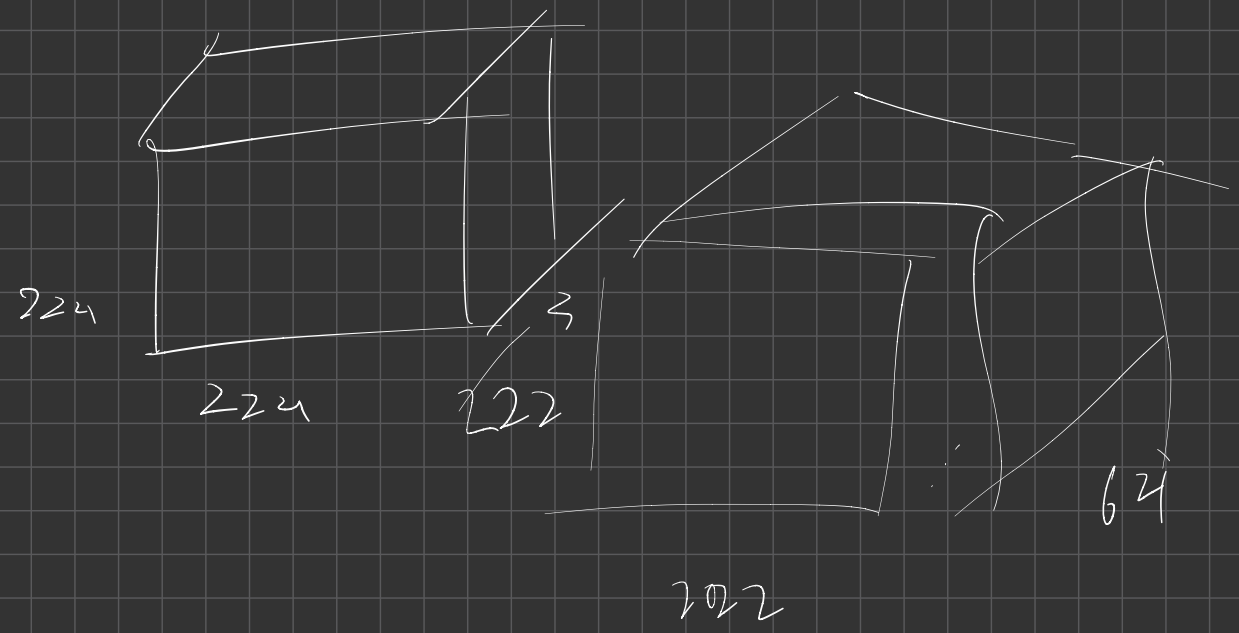
output of CNN → feature vector (fixed length)

RNN - based model → LSTMS

Initialize hidden state of the RNN with feature vector

Back propogation in time

BLEU METEOR CIDER, ROUGE




(224, 224, 3) (3, 3, 64)