Convolutional Neural Network

Convolutional Neural Network

- Although in 1996
  - IBM's Deep Blue supercomputer
  - Beat the chess world champion Garry Kasparov

Convolutional Neural Network

- Yet computers were unable to do trivial tasks such as
  - Detecting a puppy in a picture or
  - Recognizing spoken words
  - Until quite recently

Convolutional Neural Network

- Convolutional neural networks (CNNs) emerged
  - From the study of the brain's visual cortex, and
  - They have been used in image recognition since the 1980s.

Convolutional Neural Network

- In the last few years
  - CNNs have managed to achieve superhuman performance
  - On some complex visual tasks
- And all this was possible because of
  - Increase in computational power
  - The amount of available training data
  - And the tricks presented in last chapter on training deep neural nets

CLOUD x LAB

Convolutional Neural Network

- Today CNNs power
  - Image search services
  - Self-driving cars
  - Automatic video classification systems
  - Voice recognition and
  - Natural language processing - NLP

language
robots
natural
physical
system
approach
learning
translation
commands
manipulation
household
instructions
control
gambit
unconstrained
privacy
security
future
route
map
demonstrate
interact
statistical
perception
grounded
robot
machine
lessons
model
environment
experiments
following
board
vulnerabilities
translate
parsing
robust
manipulator
become
learn
attribute
actions
custom
users
complex
based
robotic
design
interaction
ubiquitous
directions
describe
structures
capable
humans
objects
evaluate
research
games
induction
untrained
state-of-the-art
english
automatically-labeled
representations
problem
includes
joint
new
task
study
handle
interpret
execution
moving
grounding
demonstrating
using
non-idealized
environments
automatic
models
technique
scenes
construct
constraints
investigate
training
use-case
represent
facilitating
latent-variable
spoken
parse
processing
sets
parts
set
meanings
enables
imposed
testbed
numbers
classifiers
shapes
workspace
navigation
indoor
world
sentences
populated
detection
efficiently
accurately
analyses
fast
discuss
foundation
attacks
mobile
quickly
uninstrumented
experimentally
historically
scanner

Convolutional Neural Network

- In this chapter we will present
  - Where CNNs came from
  - What their building blocks looks like and
  - How to implement them using TensorFlow
- Then we will present some of the best CNN architectures

The Architecture of the Visual Cortex

Convolutional Neural Network

- In 1958 and 1959, David H. Hubel and Torsten Wiesel
  - Performed a series of experiments on cats and
  - Later on monkeys
- Their experiments gave crucial insights on the
  - Structure of the visual cortex
- They showed that many neurons in the visual cortex
  - Have a small local receptive field
  - Meaning they react only to
  - Visual stimuli located in a limited region of the visual field

Convolutional Neural Network

Local receptive fields in the visual cortex

Question

Why not simply use a regular deep network with fully connected layers for image recognition tasks?

Convolutional Neural Network

Answer

- Deep neural network work fine for small images such as MNIST
- But they break for larger images because of
  - Huge number of parameters
- For example
  - A 100x100 image has 10,000 pixels
  - If the first layer has 1,000 neurons (which is a very small number)
  - This means a total of 10 million connections, that too in first layer
  - This will require a lot of computing power
- CNNs solve this problem by using partially connected layers

Convolutional Layer

Convolutional Layer

- It is the most important building block of a CNN
- Neurons in the first convolutional layer are not connected to every single pixel in the input image , but only to pixels in their receptive fields

CLOUD x LAB

Convolutional Layer

- In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer.
- This architecture allows the network to concentrate on low-level features in the first hidden layer, then assemble them into higher-level features in the next hidden layer, and so on.

CLOUD x LAB

Convolutional Layer

- This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

CLOUD x LAB

Convolutional Layer

2-D representation of Layers

- Until now, all multilayer neural networks had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network.
- Now each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

Convolutional Layer

Zero Padding

- A neuron located in **row i, column j** of a given layer is connected to the outputs of the neurons in the previous layer located in **rows i to i + f_h – 1, columns j to j + f_w – 1**,
- Here f_h and f_w are the height and width of the receptive field .

CLOUD x LAB

Convolutional Layer

Zero Padding

- In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram.
- This is called **zero padding.**

CLOUD x LAB

Convolutional Layer

Stride

- It is possible to connect a large input layer to a much smaller layer by spacing out the receptive fields.
- The distance between two consecutive receptive fields is called the **stride**.

Convolutional Layer

Stride

This figure shows how the layers of CNN are formed

Convolutional Layer

Stride

- In this diagram, a **5 × 7** input layer (plus zero padding) is connected to a **3 × 4** layer, using **3 × 3** receptive fields and a stride of 2
- In this example the stride is the same in both directions, but it does not have to be so

CLOUD x LAB

Convolutional Layer

Stride

- A neuron located in **row i, column j** in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $\mathbf{i \times s_h}$ **to** $\mathbf{i \times s_h + f_h - 1}$**, columns** $\mathbf{j \times s_w}$ **to** $\mathbf{j \times s_w + f_w - 1}$
- Here $\mathbf{s_h}$ and $\mathbf{s_w}$ are the vertical and horizontal strides.

Convolutional Neural Network

How do we decide the weights of the neurons?

Let’s learn about Filters or Convolution Kernels

Convolutional Layer - Filters

- A neuron's weights can be represented as a small image the size of the receptive field
- This below figure shows two possible sets of weights, called filters or convolution kernels

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	1	0
0	1	0
0	1	0

3 × 3
(Vertical
Filter)

=

4 × 4

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	1	0
0	1	0
0	1	0

3 × 3
(Vertical
Filter)

=

30			

4 × 4

CLOUD x LAB

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	1	0
0	1	0
0	1	0

3 × 3
(Vertical
Filter)

=

30	**30**		

4 × 4

CLOUD x LAB

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	1	0
0	1	0
0	1	0

3 × 3
(Vertical
Filter)

=

30	30	0	

4 × 4

CLOUD x LAB

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	1	0
0	1	0
0	1	0

3 × 3
(Vertical
Filter)

=

30	30	0	0

4 × 4

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	1	0
0	1	0
0	1	0

3 × 3
(Vertical
Filter)

=

30	30	0	0
30			

4 × 4

CLOUD x LAB

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	1	0
0	1	0
0	1	0

3 × 3
(Vertical
Filter)

=

30	30	0	0
30	**30**		

4 × 4

CLOUD x LAB

Convolutional Layer - Filters

Continue like this and one last example

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	1	0
0	1	0
0	1	0

3 × 3
(Vertical
Filter)

=

30	30	0	0
30	30		
	30		

4 × 4

CLOUD x LAB

Convolutional Layer - Filters

- The first one is represented as a black square with a vertical white line in the middle
- It is a 7 × 7 matrix full of 0s except for the central column, which is full of 1s
- Neurons using these weights will ignore everything in their receptive field except for the central vertical line
- Since all inputs will get multiplied by 0, except for the ones located in the central vertical line

CLOUD x LAB

Convolutional Layer - Filters

- The second filter is a black square with a horizontal white line in the middle.
- Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

CLOUD x LAB

Convolutional Layer - Filters Example

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 × 6

×

0	0	0
1	1	1
0	0	0

3 × 3
(Vertical
Filter)

=

30			

4 × 4

Convolutional Layer - Filters Example

10	0	0	10	0	0
10	0	0	10	0	0
10	0	0	10	0	0
10	0	0	10	0	0
10	0	0	10	0	0
10	0	0	10	0	0

6 × 6

×

0	0	0
1	1	1
0	0	0

3 × 3
(Horz
Filter)

=

10	10	10	

4 × 4

CLOUD x LAB

Stacking Multiple Feature Maps

- Up to now, for simplicity, we have represented each convolutional layer as a thin 2-D layer
- But in reality it is composed of **several feature maps** of equal sizes, so it is more accurately represented in 3-D

Stacking Multiple Feature Maps

- Within one feature map, all neurons share the same parameters i.e. weights and bias term
- But different feature maps may have different parameters.

Stacking Multiple Feature Maps

- A neuron's receptive field is the same as before, but it extends across all the previous layers' feature maps.
- In short, a convolutional layer simultaneously applies multiple filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

Stacking Multiple Feature Maps

- The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the Model

Stacking Multiple Feature Maps

- It means that once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location.
- In contrast, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location

CLOUD x LAB

Stacking Multiple Feature Maps - Input Layer

- Input images are also composed of multiple sublayers: one per color channel.
- There are typically three: red, green, and blue (RGB).
- Grayscale images have just one channel
- But some images may have much more — for example, satellite images that capture extra light frequencies such as infrared

Stacking Multiple Feature Maps

- Specifically, a neuron located in **row i**, **column j** of the feature **map k** in a given convolutional **layer L** is connected to the outputs of the neurons in the **previous layer L – 1**, located in **rows** $\mathbf{i \times s_w}$ to $\mathbf{i \times s_w + f_w - 1}$ and columns $\mathbf{j \times s_h}$ to $\mathbf{j \times s_h + f_h - 1}$, across all feature maps in **layer L – 1**

CLOUD x LAB

Stacking Multiple Feature Maps

- Note that all neurons located in the same **row i and column j** but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

CLOUD x LAB

Convolution Computation

- zi, j, k is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).

CLOUD x LAB

Convolution Computation

- zi, j, k is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, fh and fw are the height and width of the receptive field, and fn′ is the number of feature maps in the previous layer (layer l – 1).

CLOUD x LAB

Convolution Computation

- xi′, j′, k′ is the output of the neuron located in layer l – 1, row i′, column j′, feature map k′ (or channel k′ if the previous layer is the input layer).

Convolution Computation

- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u, v, k', k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k′.

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

Convolution Computation

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

Computation

CLOUD x LAB

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

Computation

TensorFlow Implementation

- In TensorFlow, each input image is typically represented as a 3D tensor of shape **[height, width, channels]**
- A mini-batch is represented as a 4D tensor of shape **[mini-batch size, height, width, channels]**
- The weights of a convolutional layer are represented as a 4D tensor of shape $[f_h, f_w, f_n, f_{n'}]$
- The bias terms of a convolutional layer are simply represented as a 1D tensor of shape $[f_n]$

TensorFlow Implementation

Let's look at a simple example

- The following code loads two sample images, using Scikit-Learn's **load_sample_images()**
- It loads two color images, one of a Chinese temple, and the other of a flower

```
>>> import numpy as np
>>> from sklearn.datasets import load_sample_images
# Load sample images
>>> dataset = np.array(load_sample_images().images,
dtype=np.float32)
>>> batch_size, height, width, channels = dataset.shape
```

Run it on Notebook

TensorFlow Implementation

Let's look at a simple example

- Then create two 7 × 7 filters
- One with a vertical white line in the middle, and the other with a horizontal white line

```
# Create 2 filters
>>> filters_test = np.zeros(shape=(7, 7, channels, 2),
dtype=np.float32)
>>> filters_test[:, 3, :, 0] = 1 # vertical line
>>> filters_test[3, :, :, 1] = 1 # horizontal line
>>> filters = tf.constant(filters_test)
```

Run it on Notebook

TensorFlow Implementation

Let's look at a simple example

- Apply them to both images using a convolutional layer built using TensorFlow's conv2d() function
- With zero padding and strides is a list of integers, 1-D tensor of length 4. The stride of the sliding window for each dimension of input.

```
# Create a graph with input X plus a convolutional layer
# applying the 2 filters
>>> X = tf.placeholder(tf.float32, shape=(None, height,
width, channels))
>>> convolution = tf.nn.conv2d(X, filters,
strides=[1,2,2,1], padding="SAME")
```

Run it on Notebook

TensorFlow Implementation

Let's look at a simple example

- Let's run our session and plot one of the resulting feature maps

```
>>> with tf.Session() as sess:
        output = sess.run(convolution, feed_dict={X:  dataset})
>>> plt.imshow(output[0, :, :, 1]) # plot 1st image's 2nd feature map
>>> plt.show()
```

Run it on Notebook

TensorFlow Implementation

Let's understand the previous code

```
>>> X = tf.placeholder(tf.float32, shape=(None, height, width,
channels))
>>> convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1],
padding="SAME")
```

- **X** is the input mini-batch a 4D tensor
- **filters** is the set of filters to apply also a 4D tensor
- **strides** is a four-element 1D array, where the two central elements are the vertical and horizontal strides (s_h and s_w).

TensorFlow Implementation

Let's understand the previous code

```
>>> X = tf.placeholder(tf.float32, shape=(None, height, width,
channels))
>>> convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1],
padding="SAME")
```

- The first and last elements of the strides must currently be equal to 1.
- They may one day be used to specify a batch stride (to skip some instances) and a channel stride (to skip some of the previous layer's feature maps or channels).

TensorFlow Implementation

Let's understand the previous code

```
>>> X = tf.placeholder(tf.float32, shape=(None, height, width,
channels))
>>> convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1],
padding="SAME")
```

- **padding** must be either **"VALID" or "SAME"**:
  - If set to **"VALID"**, the convolutional layer does not use zero padding, and may ignore some rows and columns at the bottom and right of the input image, depending on the stride
  - If set to **"SAME"**, the convolutional layer uses zero padding if necessary.

TensorFlow Implementation

Padding - "VALID"

For simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension

TensorFlow Implementation

Padding - "SAME"

- In this case, the number of output neurons is equal to the number of input neurons divided by the stride, rounded up
- Then zeros are added as evenly as possible around the inputs.
- Here, number of input neurons = 13, stride = 5, hence number of output neurons = ceil(13 / 5) = 3).

CNN Hyperparameters

- Unfortunately, convolutional layers have quite a few hyperparameters
- We must choose
  - The **number of filters**,
  - Their **height and width**,
  - The **strides**,
  - And the **padding type**.
- As always, you can use cross-validation to find the right hyperparameter values, but this is very time-consuming.

Problems with CNN

Memory Requirements

- One with CNNs is that the convolutional layers require a **huge amount of RAM**, especially during training
- Because the **reverse pass of backpropagation** requires all the intermediate values computed during the forward pass.

Problems with CNN

Memory Requirements

Consider the example

- Consider a convolutional layer with
  - **5 × 5 filters**,
  - **outputting 200 feature maps of size 150 × 100**,
  - with **stride 1**
  - and **SAME padding**.

Question

If the input is a **150 × 100** RGB image (three channels), then the number of parameters will be ???

Problems with CNN

Memory Requirements

Consider the example

Answer

$$(5 \times 5 \times 3 + 1) \times 200 = 15{,}200$$

Since it is a 5 x 5 filter

Since there are 3 colors RGB

+1 corresponds to the bias terms

Since there are 200 feature maps

The bias term is fairly small compared to a fully connected layer

Problems with CNN

Memory Requirements

The real problem

- Each of the **200 feature maps** contains **150 × 100 neurons**
- And each of these neurons needs to compute a weighted sum of its **5 × 5 × 3 = 75 inputs**: that's a total of **225 million float multiplications** !!

Problems with CNN

Memory Requirements

The real problem

- Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy **200 × 150 × 100 × 32 = 96 million bits (about 11.4 MB) of RAM**.
- And that's just for one instance! If a training batch contains 100 instances, then this layer will use up over **1 GB of RAM!**

Problems with CNN

Memory Requirements
The real problem

<u>During inference</u>

- When making a prediction for a new instance the RAM occupied by one layer can be released as soon as the next layer has been computed
- So you only need as much RAM as required by two consecutive layers.

<u>During Training</u>

- But during training everything computed during the forward pass needs to be preserved for the reverse pass
- So the amount of RAM needed is at least the total amount of RAM required by all layers.

Pooling Layers

Pooling Layers

- The goal of pooling layers is to
  - Subsample(shrink) the input image in order to reduce
    - The computational load
    - The memory usages and
    - The number of parameters (limiting the risk of overfitting)
- Also reducing the input image size
  - Makes the neural network tolerate a little bit of image shift
  - This is called **location invariance**

Pooling Layers

- Just like convolutional layer
  - Each neuron in a pooling layer is connected to the outputs of a
  - Limited number of neurons in the previous layer
  - Located within a small rectangular receptive field
- We must define its size, the stride, and the padding type
- A Pooling neuron has no weights
  - It just aggregates the inputs using an aggregation function
  - Such as the max or mean

Pooling Layers

- The below figure shows a max pooling layer
  - Which is the most common type of pooling layer
- We have a 2 × 2 pooling kernel, a stride of 2, and no padding

Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

Pooling Layers

- Note the only the max input value in each kernel
  - Makes it to the next layer
  - The other inputs are dropped

Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

Pooling Layers

- This obviously is very destructive layer
  - As even with a tiny 2 × 2 kernel and a stride of 2
  - Output will be two times smaller in both directions
  - So its area will be four times smaller
  - Simply dropping 75% of the input values

Pooling Layers

- A pooling layer typically works on
  - Every input channel independently
- So the output depth is the same as the input depth
- We may also pool over the depth dimension, in which case
  - Image's spatial dimensions (height and width)
  - Remain unchanged
  - But the number of channels is reduced

Pooling Layers

Let's implement a max pooling layer in TensorFlow using a 2×2 kernel, stride 2, and no padding and then apply to all the images in the dataset

Pooling Layers

```
[...] # load the image dataset, just like above
```

```
# Create a graph with input X plus a max pooling layer
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8))  # plot the output for the 1st image
plt.show()
```

- ksize argument contains the kernel shape
  - Along all four dimensions of the input tensor:
  - [batch size, height, width, channels]

Pooling Layers

```
[...] # load the image dataset, just like above
```

```
# Create a graph with input X plus a max pooling layer
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8))  # plot the output for the 1st image
plt.show()
```

- TensorFlow currently does not support
  - Pooling over multiple instances
  - So the first element of ksize must be equal to 1

Pooling Layers

```
[...] # load the image dataset, just like above
```

```
# Create a graph with input X plus a max pooling layer
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8))  # plot the output for the 1st image
plt.show()
```

- Also TensorFlow currently does not support
  - Pooling over both the spatial dimensions (height and width) and the depth dimension
  - So either ksize[1] and ksize[2] must both be equal to 1
  - Or ksize[3] must be equal to 1

Pooling Layers

```
[...] # load the image dataset, just like above
```

```
# Create a graph with input X plus a max pooling layer
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8))  # plot the output for the 1st image
plt.show()
```

- To create an average pooling layer
  - Just use the `avg_pool()` function instead of `max_pool()`.

CNN Architectures

CNN Architectures

- Typical CNN architectures stack a few convolutional layers
- Each one generally followed by a ReLU layer
- Then a pooling layer
- Then another few convolutional layers (+ReLU)
- Then another pooling layer, and so on

CNN Architectures

- The input image gets smaller and smaller as it progresses through the network
- But it also typically gets deeper and deeper (i.e., with more feature maps)

Convolutional Neural Network

CNN Architectures

- At the top of the stack a regular feedforward neural network is added
  - Which is composed of few fully connected layers + ReLUs
- The final layer outputs the prediction
  - Its basically a softmax layer that outputs estimated class probabilities

Convolutional Neural Network

CNN Architectures

Tip

- A common mistake is to use convolution kernels that are too large
- We can often get the same effect as a 9×9 kernel
  - By stacking two 3×3 kernels on top of each other
  - Also this approach will have a lot less compute

CNN Architectures

- Over the years
  - Variants of this fundamental architecture have been developed
  - Leading to amazing advances in the field
  - A good measure of this progress is the
    - Error rate in competitions such as the ILSVRC ImageNet challenge
    - Now part of kaggle:

      https://www.kaggle.com/c/imagenet-object-localization-challenge

CNN Architectures

- In this competition the **top-5 error rate** for
  - Image classification fell from over
  - 26% to barely over 3% in just five years
- The **top-five error rate** is the number of test images
  - For which the system's top 5 predictions
  - Did not include the correct answer
- In this challenge,
  - The images are large (256 pixels high) and
  - There are 1,000 classes, some of which are really subtle
  - Like distinguishing 120 dog breeds

CNN Architectures

Let's look into some of the best CNN architectures

LeNet-5

- Most widely known CNN architecture
- It was created by Yann LeCun in 1998 and
  - Widely used for handwritten digit recognition (MNIST)

Table 13-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

LeNet-5

Few extra details about LeNet-5

- MNIST images are 28 × 28 pixels
  - But they are zero-padded to 32 × 32 pixels and
  - Normalized before being fed to the network
  - The rest of the network does not use any padding
  - Which is why the size keeps shrinking as
  - The image progresses through the network

AlexNet

- Developed by Alex Krizhevsky Ilya Sutskever, and Geoffrey Hinton
- Alexnet CNN architecture won the 2012 ImageNet ILSVRC
- It achieved 17% top-5 error rate
- It is quite similar to LeNet-5 but
  - Much larger and deeper and
  - It was the first CNN to
  - Stack convolutional layers directly on top of each other
  - Instead of stacking a pooling layer on top of each convolutional layer

AlexNet

Table 13-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

AlexNet

- To reduce overfitting, the authors used two regularization techniques
  - First they applied dropout (with a 50% dropout rate)
  - During training to the outputs of layers F8 and F9
- Second, they performed data augmentation by
  - Randomly shifting the training images by various offsets
  - Flipping them horizontally, and changing the lighting conditions

AlexNet

Local Response Normalization

- AlexNet also uses a competitive normalization step
  - Immediately after the ReLU step of layers C1 and C3
  - Called **local response normalization**
- This form of normalization makes the neurons
  - That most strongly activate inhibit neurons at the same location
  - But in neighboring feature maps
  - Such competitive activation has been observed in biological neurons

AlexNet

Local Response Normalization

- This encourages different feature maps
  - To specialize, pushing them apart and
  - Forcing them to explore a wider range of features
  - Ultimately improving generalization

AlexNet

- **ZF Net** - A variant of AlexNet
  - Was developed by Matthew Zeiler and Rob Fergus
  - It won the 2013 ILSVRC challenge
  - It just tweaked few hyperparameters such as
    - Number of feature maps
    - Kernel Size
    - Stride etc

GoogLeNet

- Developed by Christian Szegedy et al. from Google Research
- It won the ILSVRC 2014 challenge by pushing the top-5 error rate below 7%.

GoogLeNet

Features of GoogLeNet

- The great performance of GoogLeNet came in large part from the fact that the network was much deeper than previous CNNs.
- This was made possible by sub-networks called **inception modules**, which allow GoogLeNet to use parameters much more efficiently than previous architectures
- GoogLeNet actually has 10 times fewer parameters than AlexNet
- Roughly 6 million instead of 60 million

GoogLeNet - The Architecture of Inception Module

GoogLeNet - The Architecture of Inception Module

- The notation "**3 × 3 + 1(S)**" means that the layer uses a 3 × 3 kernel, stride of 1, and SAME padding

CLOUD x LAB

GoogLeNet - The Architecture of Inception Module

- The input signal is first copied and fed to four different layers
- All convolutional layers use the ReLU activation function.

CLOUD x LAB

GoogLeNet - The Architecture of Inception Module

- Note that the second set of convolutional layers uses different kernel sizes **(1 × 1, 3 × 3, and 5 × 5)**, allowing them to capture patterns at different scales.

CLOUD x LAB

GoogLeNet - The Architecture of Inception Module

- Every single layer uses a stride of **1** and **SAME** padding, even the **max pooling layer**, so their outputs all have the same height and width as their inputs

CLOUD x LAB

GoogLeNet - The Architecture of Inception Module

- This makes it possible to concatenate all the outputs along the depth dimension in the final depth concat layer i.e., stack the feature maps from all four top convolutional layers

GoogLeNet - The Architecture of Inception Module

Why inception modules have convolutional layers with 1 × 1 kernels

CLOUD x LAB

GoogLeNet - The Architecture of Inception Module

- These layers cannot capture any features since they look at only one pixel at a time
- These layers serve two purposes:
  - **First**, they are configured to output many fewer feature maps than their inputs, so they serve as bottleneck layers, meaning they reduce dimensionality.
  - This is particularly useful before the **3 × 3** and **5 × 5** convolutions, since these are very computationally expensive layers.

GoogLeNet - The Architecture of Inception Module

- These layers serve two purposes:
  - **Second**, each pair of convolutional layers **([1 × 1, 3 × 3] and [1 × 1, 5 × 5])** acts like a single, powerful convolutional layer, capable of capturing more complex patterns.
  - Indeed, instead of sweeping a simple linear classifier across the image , as a single convolutional layer does, this pair of convolutional layers sweeps a two-layer neural network across the image.

ResNet

- Residual Network or ResNet
  - Was the winner of the ILSVRC 2015 challenge
- It delivered an astounding
  - Top-5 error rate under 3.6%, using an
  - Extremely deep CNN composed of 152 layers
- It was developed by Kaiming He et al

Questions?

https://discuss.cloudxlab.com

reachus@cloudxlab.com

Convolutional Layer - Filters - Complex

Convolutional Layer - Filters