CSC413 Neural Networks and Deep Learning

Lecture 7: Features Visualization and Adversarial Examples

Lecture 7

February 27/29, 2024

Table of Contents

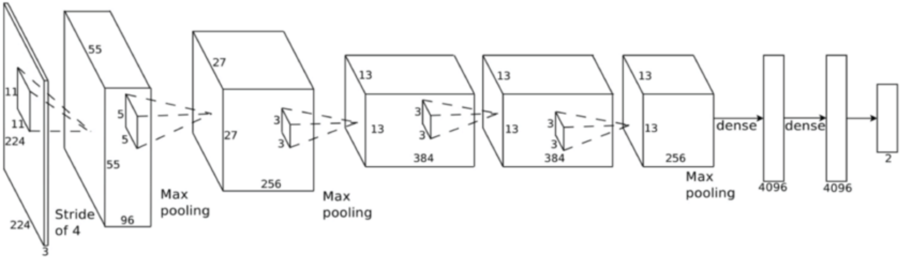
Lecture Plan

- CNN Feature Visualization and Interpretation
- Transfer Learning
- Adversarial Examples

Section 1

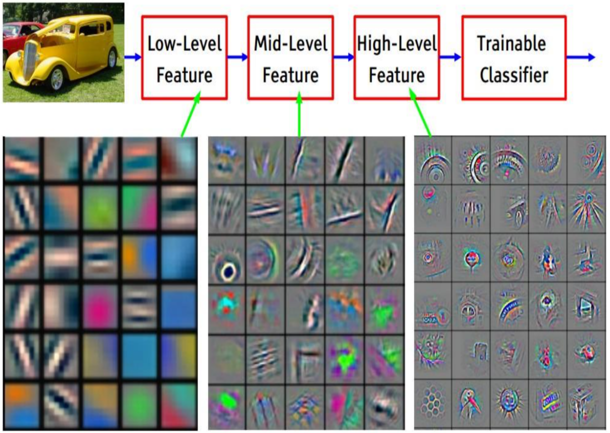
CNN Review

Example CNN: AlexNet



```python
import torchvision.models
alexNet = torchvision.models.alexnet(pretrained=False)
```

Convolutional Features



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Receptive Fields

Because of downsampling (pooling and use of strides), higher-layer filters "cover" a larger region of the input than equal-sized filters in the lower layers.

Transfer Learning

Transfer Learning is the idea of using weights/features trained on one task, and using it on another task.

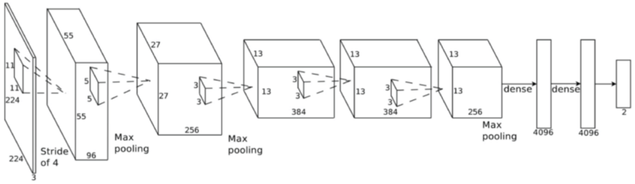
Example:

- Train a model to predict the next word given the previous three
- Use the weights to determine word similarities

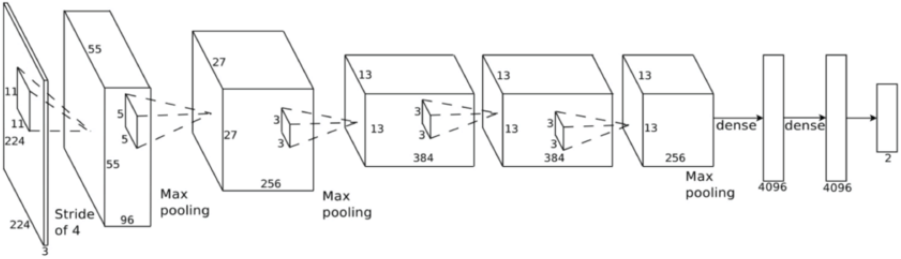
Transfer learning with CNN

Practitioners rarely train a CNN “from scratch”. Instead we could:

1. Take a pre-trained CNN model (e.g. AlexNet), and use its features network to compute **image features**, which we then use to classify our own images
2. Initialize our weights using the weights of a pre-trained CNN model (e.g. AlexNet)

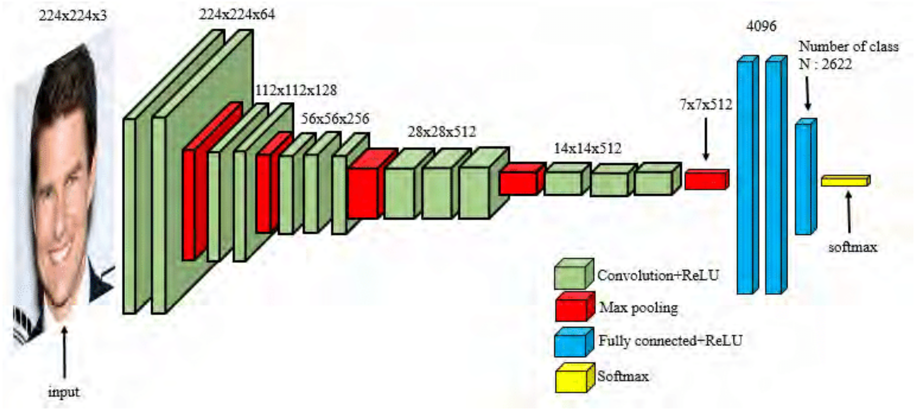


AlexNet (2012)



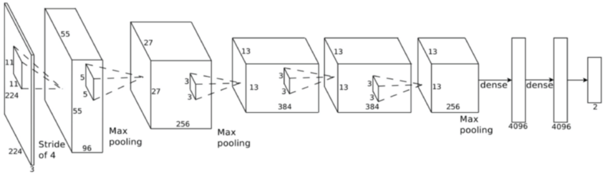
```
import torchvision.models
alexNet = torchvision.models.alexnet(pretrained=False)
```

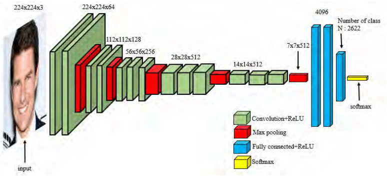
VGG (2014)



```
# There are many VGG versions
vgg16 = torchvision.models.vgg.vgg16(pretrained=False)
vgg19 = torchvision.models.vgg.vgg19(pretrained=False)
```

What is new in VGG (compared to AlexNet)?





- VGG uses very small receptive fields (3×3 instead of 11×11)
- VGG incorporates 1×1 convolutional layers (why?)
- FC layers can be expressed as CONV layers and vice versa
    - FC layer with 4096 output units looking at an input volume of 7 x 7 x 512 is equivalent to a CONV layer with kernel size 7, stride 1, and 4096 filters.

One more idea. . .

Most of these networks have **fully connected layers** at the very end.

- Pro: Fully connected layers computes features on the *entire* image
- Con: what if we wanted to work with images of various sizes?

Idea: instead of fully connected layers, we could. . .

- Use a convolution layer with the same kernel size as hidden unit size and no padding
- Use global average-pooling

This is more frequently done on pixel-wise prediction problems, which we'll see later in this course.

Section 2

Interpreting CNNs

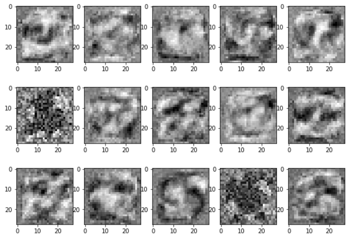
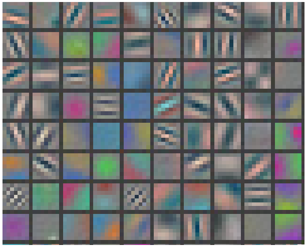
How do CNNs work?

Convolutional neural networks are successful, but how do we know that the network has learned useful patterns from the training set?

Interpretation of deep learning models is a challenge, but there are some tricks we can use to interpret CNN models

Weight Visualization

Recall: we can understand what first-layer features in a MLP are doing by visualizing the weight matrices (left)

We can do the same thing with convolutional networks (right)

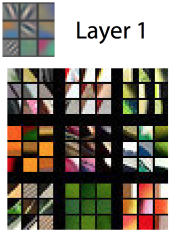
But what about higher-level features?

Feature Visualization

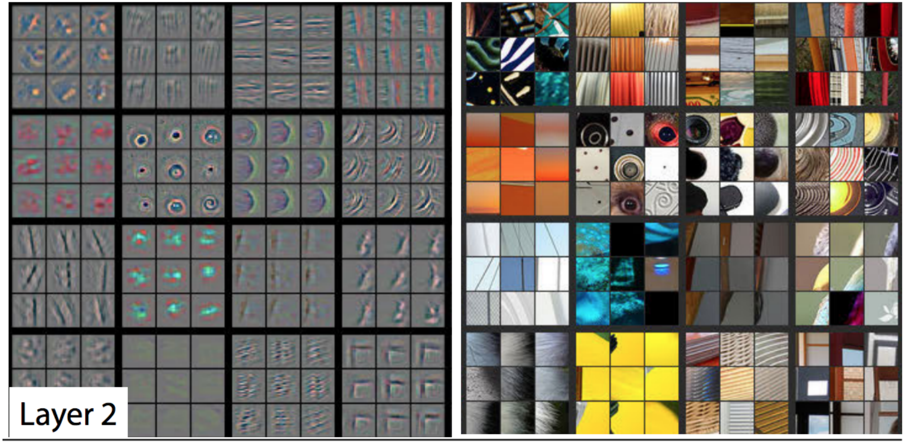
One approach: pick the images in the training set which activate a unit most strongly.

(Compute forward pass for each image in the training set, track when a feature was **most** active, and look for the portion of the image that lead to that activation)

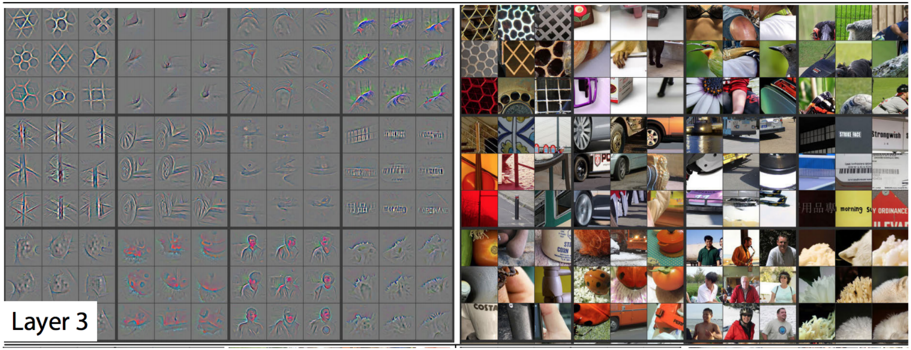
Here is the visualization for layer 1:



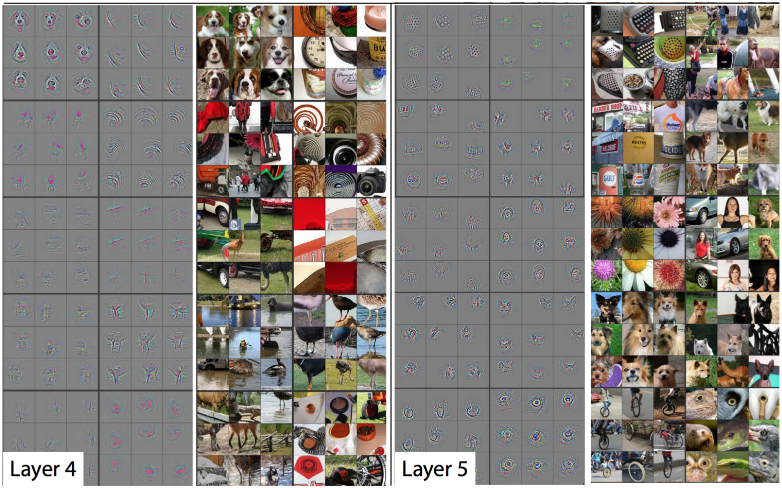
Feature Visualization: Layer 2



Feature Visualization: Layer 3



Feature Visualization: Layer 4



The issue with feature visualizations

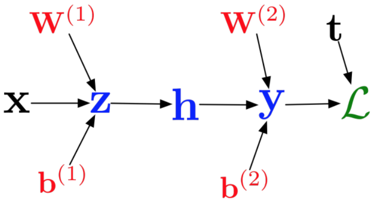
Higher layer seems to pick up more abstract, high-level information.

Problem: Can't tell what unit is actually responding in the image!

Maybe we can use input gradients?

Input Gradients

Recall this computation graph:


$$\begin{tikzcd}
\mathbf{W}^{(1)} \arrow[d] & & \mathbf{W}^{(2)} \arrow[d] & \mathbf{t} \arrow[d] \\
\mathbf{x} \arrow[r] & \mathbf{z} \arrow[r] & \mathbf{h} \arrow[r] & \mathbf{y} \arrow[r] & \mathcal{L} \\
\mathbf{b}^{(1)} \arrow[u] & & \mathbf{b}^{(2)} \arrow[u]
\end{tikzcd}$$

From this graph, we could compute $\frac{\partial L}{\partial x}$ – the model's sensitivity with respect to the input.

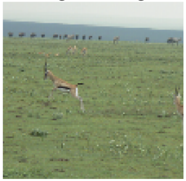
(We've never done this because there hasn't been a need to—until now!)

The problem with input gradients

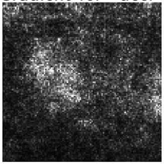
Input gradients can be noisy and hard to interpret

Take a good object recognition conv net and compute the gradient of $\log p(y = "deer" | \mathbf{x})$

Original image



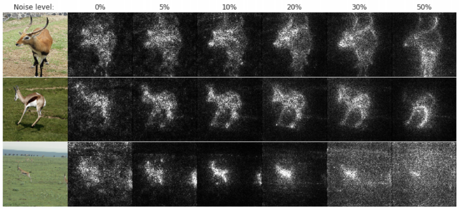
Gradient for "deer"



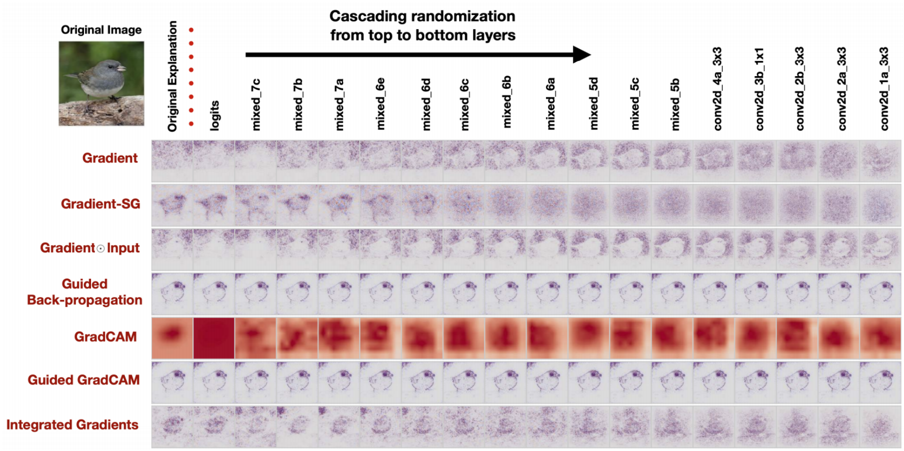
Smoothing the input gradients

Several methods modify these gradients:

- Guided Backprop: accumulate only **positive gradients** when doing back propagation
- SmoothGrad: do the backward pass on a few noisy version of the input image, then average their input gradients



Cautionary Tale of Image Gradients



From: https://proceedings.neurips.cc/paper/2018/file/294a8ed24b1ad22ec2e7efea049b8737-Paper.pdf

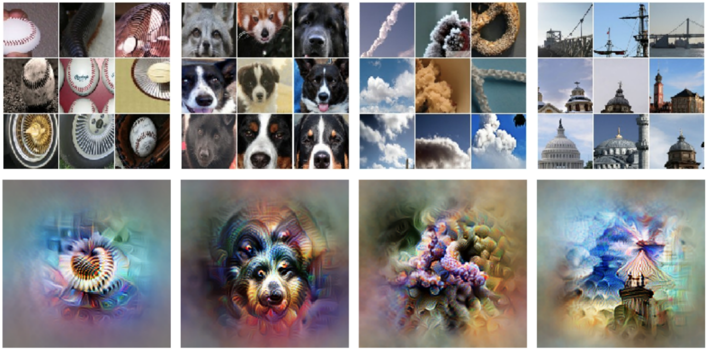
Optimizing an Image to Maximize Activations

Can we use gradient ascent on an image to maximize the activation of a given neuron?

Requires a few tricks to make this work; see https://distill.pub/2017/feature-visualization/

Dataset Examples show us what neurons respond to in practice

Optimization isolates the causes of behavior from mere correlations. A neuron may not be detecting what you initially thought.



Baseball—or stripes? *mixed4a, Unit 6*

Animal faces—or snouts? *mixed4a, Unit 240*

Clouds—or fluffiness? *mixed4a, Unit 453*

Buildings—or sky? *mixed4a, Unit 492*

Deep Dream

Similar idea:

- Start with an image, and run a conv net on it.
- Pick a layer in the network.
- Change the image such that units which were already highly activated get activated even more strongly. “Rich get richer.”
- Repeat.

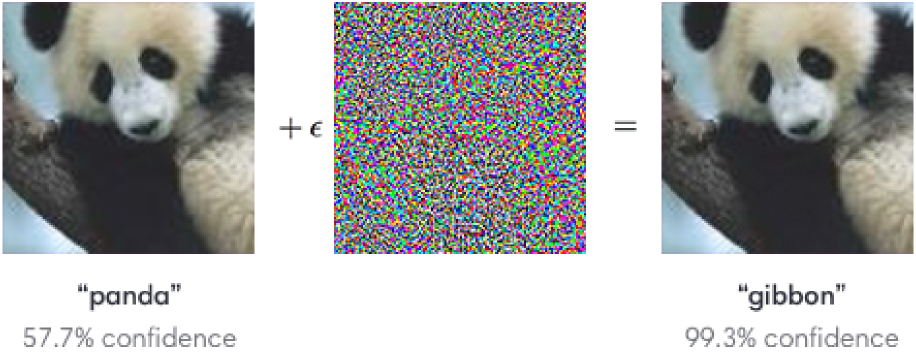
This will accentuate whatever features of an image already kind of resemble the object.

https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

Section 3

Adversarial Examples

What are these images of?



Producing adversarial images: Given an image for one category (e.g. panda), compute the image gradient to maximize the network's output unit for a different category (e.g. gibbon)

Non-targetted Adversarial Attack

Goal: Choose a small perturbation ϵ on an image x so that a neural network f misclassifies $x + \epsilon$.

Approach:

Use the same optimization process to choose ϵ to **minimize** the probability that

$$f(x + \epsilon) = \text{correct class}$$

Targeted Adversarial Attack

Targeted attack

Maximize the probability that $f(x + \epsilon) =$ target incorrect class

Non-targeted attack

Minimize the probability that $f(x + \epsilon) =$ correct class

Adversarial Attack

- 2013: ha ha, how cute!
  - The paper which introduced adversarial examples was titled "Intriguing Properties of Neural Networks."
- 2018+: serious security threat
  - 7 of 8 proposed defences accepted to ICLR 2018 were cracked within days.
- New methods for attacks and defence are regularly developed!

White-box vs Black-box Adversarial Attacks

Adversarial examples transfer to different networks trained on a totally separate training set!

White-box Adversarial Attack: Model architecture and weights are known, so we can compute gradients.

Black-box Adversarial Attack: Model architecture and weights are unknown.

- You don't need access to the original network!
- You can train up a new network to match its predictions, and then construct adversarial examples for that.

Attack carried out against proprietary classification networks accessed using prediction APIs (MetaMind, Amazon, Google)

Adversarial Examples in 3D

It is possible to have a 3D object that gets misclassified by a neural network from all angles.

https://www.youtube.com/watch?v=piYnd_wYlT8

Printed Adversarial Examples

It is possible for a printed image to cause object detection to fail.

https://www.youtube.com/watch?v=MIbFvK2S9g8