50.039 – Theory and Practice of Deep Learning

Alex

Week 06: Attacks 1

takeaway

- this lecture is an easy introduction to adversarial attacks
- Neural networks can be easily fooled by adversarial inputs: one can take an input, and obtain a slight modification with an almost arbitrarily different prediction
- how: you compute the gradient with respect to inputs, and apply the gradient to *"optimize"* the inputs rather than the parameters.
- despite near human performance, algorithms process data differently from humans, with discontinuities in the change of prediction as a function the labels.
- one possible explanation: the decision boundaries of classifiers are wild and nearly random in input regions that have low density in the training data / are outlier regions.
- proper processing needs simultaneous recognition of outliers / or recognition of crafted samples!
- this is not vision specific

In this lecture we start with a simple intro.

1 Simple Fooling

Goal is: take an image, and create an image that looks very similar but has a totally nonsense prediction for it! This is fooling of neural networks.

- inputs: a pretrained neural network for multi-class classification, an input image, a target class t different from the neural network prediction for that class

- iterate in a while loop until target class $f_t(x)$ has highest score: $t = \text{argmax}_c f_c(x)$
  - compute gradient of the output of the pretrained neural network for the target class with respect to the input data $\frac{\partial f_t}{\partial x}(x)$
  - apply the gradient with a small stepsize onto the input x for gradient **ascent**

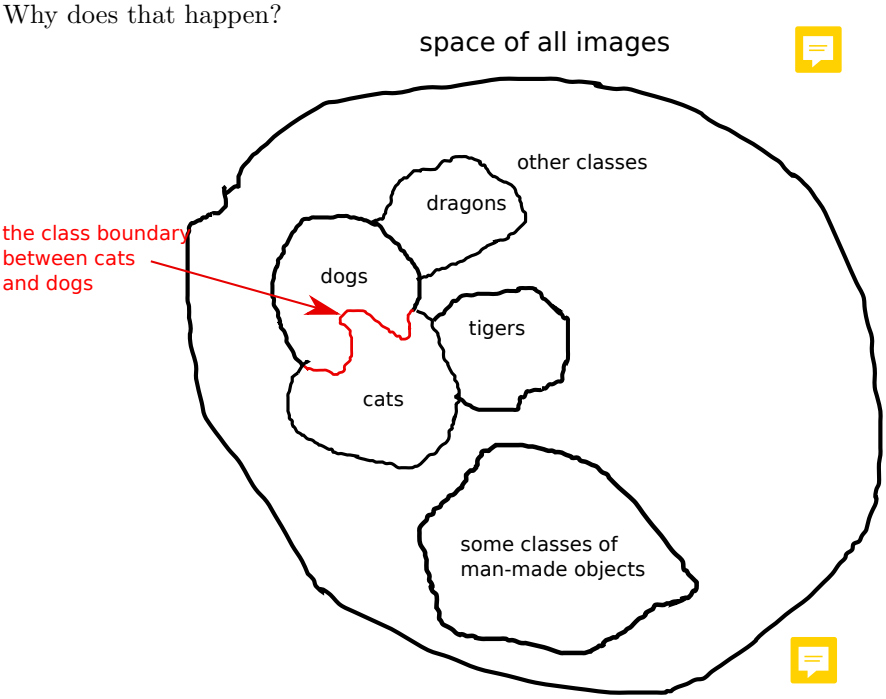
$$x_{n+1} = x_n + \eta \frac{\partial f_t}{\partial x}(x_n)$$

- find a step size small enough that differences are barely visible.
- plot the original image as loaded versus the modified image, plot the difference and its mean absolute difference
- note: you dont need a dataloader, only to load one image resized one time. You can use the image transforms.

This is a special form of attack: targeted towards a class, a whitebox one – means you have access to the classifier internals

2 Why does that work at all??

Different from human expectation or human perception – small change should not yield a different label! Deep learning predictions are different.
Why does that happen?



The naive view of class boundaries is shown in above graphic. There are object classes, and boundaries between them. The most important wrong thing is: **the whole space was densely filled with training examples during training.** This naive view of

- smooth boundaries,
- with spaces densely occupied in by training data between

seems not to hold for neural networks – due to the high dimensionality of representations. Instead there are large regions of space, which was not covered with training data during the training process.

We take images, and process them in a neural network. Layer by layer activations get computed. If a layer has K neurons, then the space of all possible activation vectors is a K-dimensional vector space.

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

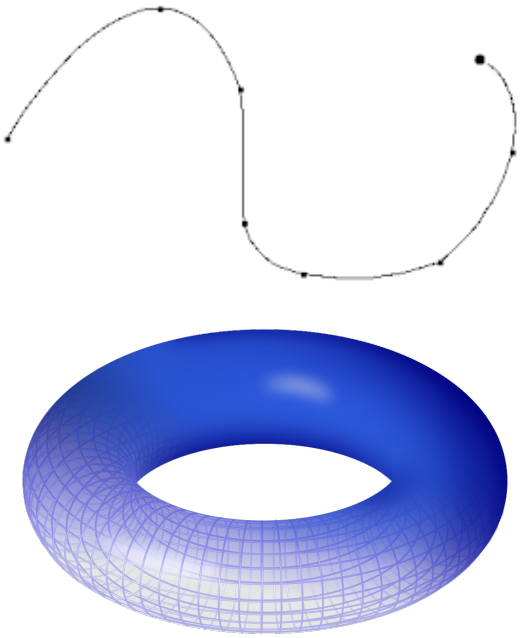
Table 1: GoogLeNet incarnation of the Inception architecture

For one image the activations of all neurons of one layer are one single point in a high-dimensional space (dimensionality being equal to the number of all output neurons in this layer!) – for above googlenet/inception model: inception (4a) has 100352 dimensions.

One can ask: when we compute activation vectors for many images, how will they be distributed in this space of all neuron activations? Usually they do not fill up the high-dimensional space with equal density.

Often they are approximately distributed in a small thin region of the high dimensional space of all possible activations. Such a thin region (of lower dimensionality) is called a manifold.

What are manifolds? curves are 1-dim manifolds, a curved 2-dim hyperplane, like a torus, would be a 2-dim manifold.



Higher order manifolds cannot be drawn trivially, but defined mathematically.

In short: for high dimensional representations, there are many regions with very low training data density – outlier regions, while most training data is mapped onto some lower-dimensional subspace.

What is the conclusion? **The decision boundaries in outlier regions are not specified by training** ... and they can be nearly random in outlier regions. The neural network was not trained to deal with with images which are mapped outside of these manifolds. In a 100000-dim space finding directions that lead to 1000 different classes looks not challenging. This is why it works.

More info for example for object detection and segmentation: `http://openaccess.thecvf.com/content_ICCV_2017/papers/Xie_Adversarial_Examples_for_ICCV_2017_paper.pdf`

3 Algorithmically Improved Fooling

Your result is possibly not a valid image for three reasons:

A Images after $ToTensor()$ should be in $[0, 1]$, for some pixels it can be violated after the gradient updates.

B After optimization, our values for subpixels are floats . When saved, the image subpixel values get rescaled to $[0, ..., 255]$ and rounded to the integers in that range. PIL usually expects inputs as integers in $\{0, \ldots, 255\}$.

The rounding may change the prediction!!

C Image saving might introduce additional biases, e.g. changes in subpixel values due to lossy compression, for example when saving as jpg. Save images as png without lossy compression solves this.

Here are the proposed solutions for A and B.

Fix for the out of bounds problems is to perform a gradient descent in a suitable subspace. Suppose you have a computed a gradient, then store in a temporary variable how the input x would look like after an update:

$$g = \frac{\partial f_t}{\partial x}(x)$$
$$tmpv = x + \epsilon g$$

Find now all those dimensions of the input sample where the resulting image would be out of bounds:

$$Bad = \{d : tmpv_d * std + mean < 0 \text{ or } tmpv_d * std + mean > 1\}$$

In practice I like some safety margin:

$$Bad = \{d : tmpv_d * std + mean < 2/255 \text{ or } tmpv_d * std + mean > 253/255\}$$

Now set the vector g to zero on this set Bad of pixels, and apply it to update the current x for the next step.

Homework questions:

- Why with this modified gradient the objective function cannot decrease ? It will either increase or at least stay constant.
- when with this modified gradient the objective function would not increase, that is the algorithm cannot continue in fooling?

A fix for the discretization problem is harder. The easy way is: at first optimize until target class has the highest score. After that test for termination, whether the image after discretization still fools the net, if not, then continue to increase the score.
A more theoretically founded way: if g is the gradient, and v is a vector such that $g \cdot v > 0$, then for sufficiently small steps in the direction of v the function f_t (for which the gradient g was computed) will increase. For your given image x find a subpixel-value-rounded image $\hat{x}$ (for every rgb subpixel it has two neighbors to round) such that $g \cdot (\hat{x} - x) > 0$, that is changing the image x to the rounded image $\hat{x}$ would increase the score, provided that the step is small enough. This needs no discrete optimization

$$g \cdot (\hat{x} - x) = \sum_d g_d (\hat{x} - x)_d$$

So you can choose for every dimension (subpixel here) the one of the two roundings for which $g_d(\hat{x} - x)_d$ is positive.

4 Universal Adversarial Perturbations

So far we had to compute one perturbation for every inputs. There are perturbations which can hinder predictions for large sets of input images! However one cannot control anymore to what target class.

Moosavi-Dezfooli et al. CVPR 2017: `https://arxiv.org/pdf/1610.08401.pdf`, Hayes et al. `https://arxiv.org/abs/1708.05207`.

Success rates are $75\% - 90\%$ (and can be likely better if one would cluster the space and compute a mix of perturbations)

Algorithm 1 Computation of universal perturbations.

input: Data points X, classifier $\hat{k}$, desired ℓ_p norm of the perturbation ξ, desired accuracy on perturbed samples δ.
output: Universal perturbation vector v.
Initialize $v \leftarrow 0$.
while $\text{Err}(X_v) \leq 1 - \delta$ **do**
  for each datapoint $x_i \in X$ **do**
    if $\hat{k}(x_i + v) = \hat{k}(x_i)$ **then**
      Compute the *minimal* perturbation that sends $x_i + v$ to the decision boundary:

$$\Delta v_i \leftarrow \arg\min_r \|r\|_2 \text{ s.t. } \hat{k}(x_i + v + r) \neq \hat{k}(x_i).$$

      Update the perturbation:

$$v \leftarrow \mathcal{P}_{p,\xi}(v + \Delta v_i).$$

    end if
  end for
end while

The algorithm is shockingly simple.

5 In-class/finish as homework

This is Mr. Shout.

- Implement the Algorithmically Improved Fooling with solutions for A, B and at least the simple way to deal with discretization
- Strawberrize that guy.
- you need a pretrained neural net, a way to load the image into a pytorch tensor. Note that involves loading into numpy, swapping color and spatial axes , preprocessing it by the neural network preprocessing rules (mean/standard deviation). You do not need datasets or data loaders.
- you need a way to clip the tensor into integer value and to save it back as an image, this involves undoing of: swapping color and spatial axes ,

preprocessing it by the neural network preprocessing rules (mean/standard deviation) . `PIL.Image` is one option for the actual numpy to image step.