Adversarial examples

We introducted Gradient Ascent in our module on interpretation (Interpretation_of_DL_Gradient_Ascent.ipynb)

- We find the input $\mathbf{x}^*$
- That maximizes the value of a particular neuron $\mathbf{y}_{(l),\text{idx},k}$

$$\mathbf{x}^* = \underset{\mathbf{x}=\mathbf{y}_{(0)}}{\text{argmax}}\, \mathbf{y}_{(l),\text{idx},k}$$

In that module, we used the technique to find the input value $\mathbf{x}$ that "maximally excited" $\mathbf{y}_{(l),\text{idx},k}$.

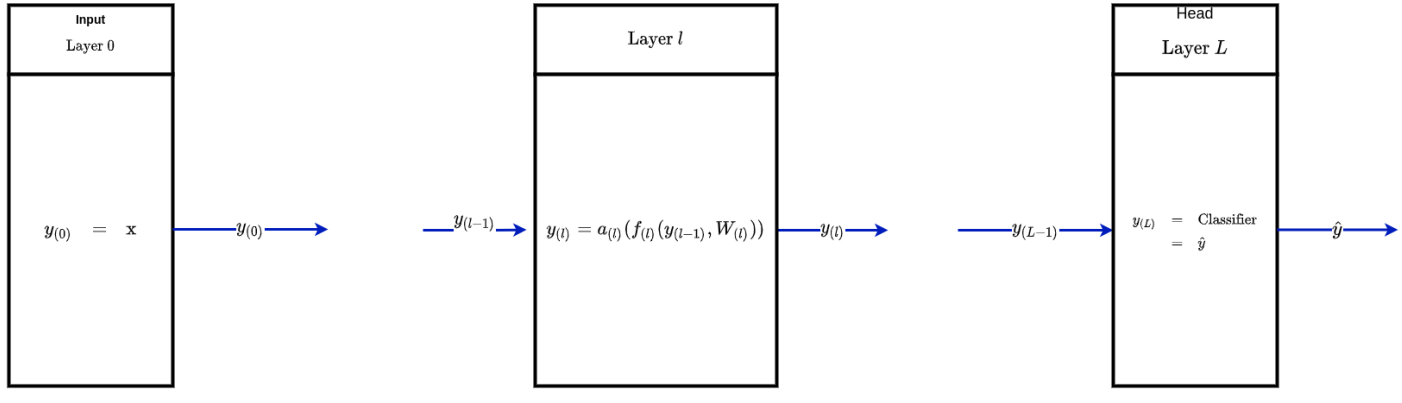
In this module, the neuron we will maximally excite will be in the head layer L.

If layer L is a classifier for a task with classes in set C

- Then $\mathbf{y}_{(L)}$ is a vector of length C
- Where $\mathbf{y}_{(L),j}$ corresponds to the predicted probability that the correct class is C_j
    - denoting the j^{th} class as C_j

$$\mathbf{x}^* = \underset{\mathbf{x}=\mathbf{y}_{(0)}}{\operatorname{argmax}} \mathbf{y}_{(L),j}$$

Layers


Input
Layer 0
$y_{(0)} = \mathbf{x}$
$y_{(0)}$
$y_{(l-1)}$
Layer l
$y_{(l)} = a_{(l)}(f_{(l)}(y_{(l-1)}, W_{(l)}))$
$y_{(l)}$
$y_{(L-1)}$
Head
Layer L
$y_{(L)} = \text{Classifier} = \hat{y}$
$\hat{y}$

That is: we will solve for the $\mathbf{x}^*$

- That is the example that looks like C_j
- More than *any* value in the input domain
- The "perfect C_j"

If C were the class of animals and the domain of $\mathbf{x}$ were images

- This would be like finding "the perfect dog" image
- At least according to the classifier

Pretty innocuous.

But what if we *constrained the optimization*

$$\mathbf{x}^* = \underset{\mathbf{x}=\mathbf{y}_{(0)}}{\operatorname{argmax}} \mathbf{y}_{(L),j}$$
$$\text{subject to}$$
$$\text{looks like}(\mathbf{x}, \mathbf{x}')$$

where

- $\text{looks like}(\mathbf{x}, \mathbf{x}')$
- Is a "closeness" metric that increases when chosen $\mathbf{x}$ is most similar to a specific $\mathbf{x}'$

And what if $\mathbf{x}'$ were from some class $C_{j'} \neq C_j$?

That is

- We find the $\mathbf{x}^*$
- That gets classified with high confidence as being C_j
- But it actually in a different class $C_{j'}$

The $\mathbf{x}^*$ obtained is called an *Adversarial Example*

- One specifically constructed to "fool" the Classifier

Adversarial examples in action:

What class is this ?

What about this ?

What class is this ?

It's almost certainly a toaster !

That was fun, but is it innocuous ?

How about the following picture:

Adversarial Stop Sign

"Speed Limit 45 mph"

Attribution: [Robust Physical-World Attacks on Deep Learning Models (https://arxiv.org/abs/1707.08945)]

Remember, the Neural Network has previously been reported to have super-human accuracy !

What's going on here ?

The problem is that we don't actually *know* how the Neural Network recognizes a toaster.

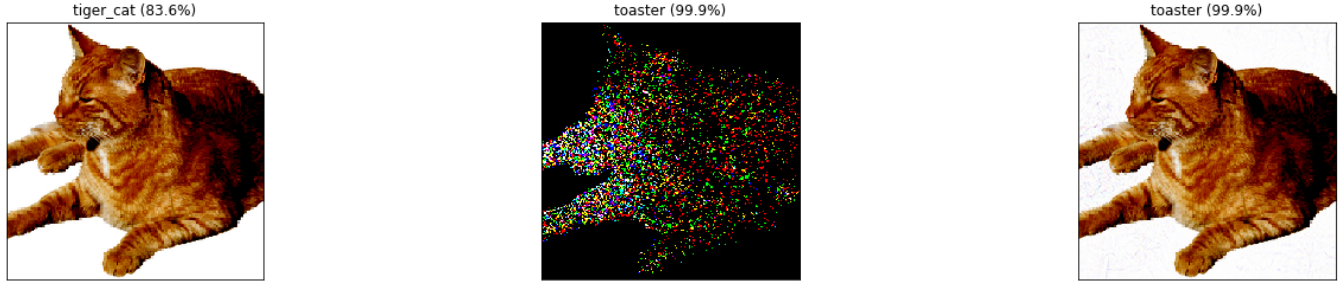
The optimizer has learned to change exactly those input pixels

- That the Neural Network uses to classify a toaster
- And, with enough additional constraints
- The changes are not detectable by the human eye

Here is a visualization of the pixels that were changed

- In order to reclassify the cat (left image)
- As a toaster (right image)

Adversarial Cat to Toaster



It should be clear that Adversarial Examples violate the Fundamental Assumption of Machine Learning

- A test example
- Comes from the same distribution as the Training examples

We are able to fool the Neural Network by asking it to solve a problem for which it wasn't trained.

This highlights an important issue

- Since we don't know how Neural Networks work
- How can we confidently deploy them in the physical world ?

Adversarial Attacks are able to fool an otherwise high quality Neural Network

- Without corrupting any training examples
- Without altering the weights of the Neural Network
- Without giving the Attacker access to any information about the Neural Network

So the attack can occur even on a Neural Network that appears as a "black box" to the attacker

Conclusion

Adversarial Examples are inputs that are crafted for the purpose of "fooling" a Neural Network.

The attacks use the same techniques that are otherwise used to correctly train a network.

This is a significant issue that must be addressed before Neural Networks can be entrusted with tasks that have real-world consquences.

Adversarial Reprogramming

We can extend the Gradient Ascent method to perform even bigger tricks:

- Getting a Classifier for Task 1 to do something completely different !

Something to consider

- Does your smartphone run Neural Network based apps ? (e.g., Snapchat filters)
- Can we trick this app into doing *something else* ?

This is called *Adversarial Reprogramming*.

We will sketch how a super-human quality ImageNet Classifier can be tricked.

- Source Task: Classify images from among 1000 classes
- Target Task: Count squares in an image

The Target task might sound simple, but observe

- That the "square" is not one of the 1000 ImageNet targets
- ImageNet has not been trained on numbers, much less to count

We refer to ImageNet Classification as the Source task.

Our goal is to get the Classifier to solve the Target task: Counting Squares.

The first issue to address:

- the $(\mathbf{x^{(i)}}, \mathbf{y^{(i)}})$ pairs of the Source task come from a different domain than that of the Target task

$$\mathbf{X}_{\text{source}}, \mathbf{y}_{\text{source}} : \quad \text{examples for Source task}$$
$$\mathbf{X}_{\text{target}}, \mathbf{y}_{\text{target}} : \quad \text{examples for Target task}$$

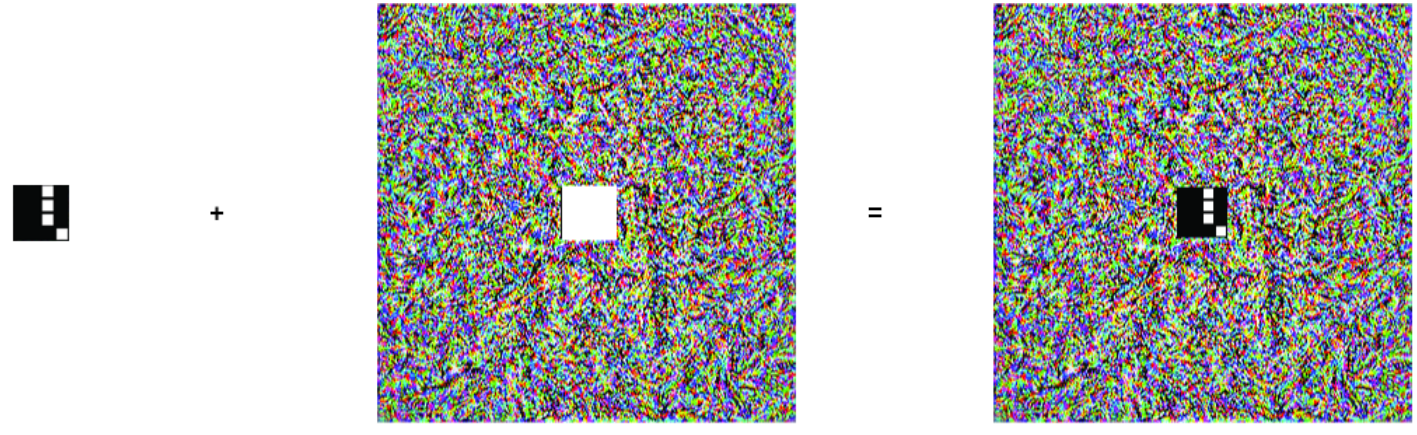
We can solve this by creating a simple function h_f to map

- $\mathbf{x}_{\text{target}}$, a feature vector in the Target task's domain.
- To $\tilde{\mathbf{x}}_{\text{source}}$, a feature vector in the Source task's domain.

This ensures that the input to the Source task is of the right "type".

Here's a picture of mapping "Squares" to an "Image"

Adversarial Reprogramming



$\mathbf{x}_{\text{target}}$ $\mathbf{W}$ $\tilde{\mathbf{x}}_{\text{source}}$

$$\tilde{\mathbf{x}}_{\text{source}} = h_f(\mathbf{W}, \mathbf{x}_{\text{target}})$$

h_f simply embeds the Target input into an image (the domain of the Source task).

Similarly, we create a function h_g to map the Target label to a Source Label.

This will ensure that the output of the Source task is of the right type.

- Here's a mapping from Counts to Image Labels.
- The inverse of this function maps Image Labels to Counts.

Count	Label
1	**Tench**
2	**Goldfish**
3	**White shark**
4	**Tiger shark**
5	**Hammerhead**
6	**Electric ray**
7	**Stringray**
8	**Cock**
9	**Hen**
10	**Ostrich**

Here's the overall picture of adapting an ImageNet Classifier to Count.

Adversarial Reprogramming: Architecture

ImageNet Classifier

"Panda"

h_f : Square to Image

ImageNet Classifier

h_g^{-1} : Label to Count

"4"

Finally, the Cost function to optimize

$$\mathbf{W} = \underset{\mathbf{W}}{\operatorname{argmin}} - \log(p(h_g(\mathbf{y}_{\text{target}}) \mid \tilde{\mathbf{x}}_{\text{source}})) + \lambda||\mathbf{W}||^2$$

where

$$\begin{aligned}
\tilde{\mathbf{x}}_{\text{source}} &= h_f(\mathbf{W}, \mathbf{x}_{\text{target}}) \\
h_f: & \quad \mathbf{y}_{\text{target}} \mapsto \mathbf{y}_{\text{source}} \quad \text{map source X to target X} \\
h_g: & \quad \mathbf{y}_{\text{target}} \mapsto \mathbf{y}_{\text{source}} \quad \text{map source label y to target label}
\end{aligned}$$

Let's break this seemingly complicated formula down into simple pieces.

Given an input example $(\mathbf{x}_{\text{target}}, \mathbf{y}_{\text{target}})$ for the Target task

- Translate feature vector $\mathbf{x}_{\text{target}}$ into a valid input for the Source Task
- Using h_f

$$\tilde{\mathbf{x}}_{\text{source}} = h_f(\mathbf{W}, \mathbf{x}_{\text{target}})$$

- Which is parameterized by $\mathbf{W}$

We also need to translate $\mathbf{y}_{\text{target}}$ to a valid label for the Source Task

$$h_g(\mathbf{y}_{\text{target}})$$

Is the Source label corresponding to the Target label $\mathbf{y}_{\text{target}}$.

The objective function being minimized

$$\mathbf{W} = \underset{\mathbf{W}}{\operatorname{argmin}} - \log(p(h_g(\mathbf{y}_{\text{target}}) \mid \tilde{\mathbf{x}}_{\text{source}})) + \lambda||\mathbf{W}||^2$$

- Is our old friend: Cross Entropy Loss
- With a regularization penalty $\lambda||\mathbf{W}||^2$
- Trying to get Source Task Classifier to predict $h_g(\mathbf{y}_{\text{target}})$
    - Given input $\tilde{\mathbf{x}}_{\text{source}} = h_f(\mathbf{W}, \mathbf{x}_{\text{target}})$

That is: we are trying to

- Maximize the likelihood that the Source classifier creates the encoding for the correct Target label
- Subject to constraining the weights $\mathbf{W}$ (the "frame" into which the Target input is placed)

How does this magic occur ?

By training !

- We find the $\mathbf{W}$
- Used by h_f
- Such that the objective is met

Nothing new !

Misaligned objectives

AI Security is the area of research concerned with the potential for humans to **cause harm to AI**

- Adversarial examples

AI Safety is the analagous area concerned with the potential for AI to **cause harm to humans**

Safety problems can arise when the Loss Function and human objectives diverge.

Consider the difference between

- "Maximize profit"
- "Maximize profit subject to legal and ethical constraints"

We (hopefully) don't have to state the additional constraints to a human -- we take it for granted.

Not so with a machine that has not been trained with examples expressing the additional objectives.

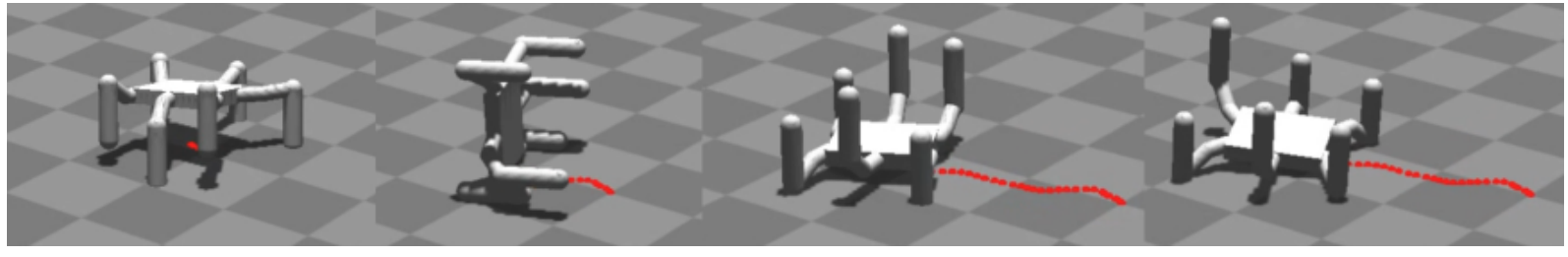
This leads to a phenomenon known as *reward hacking*

- The training algorithm decreases the Loss Function
- Even to the point of what a human would consider "cheating"
    - Achieving infinite score on a video game by discovering and exploiting a programming error
- But it is not cheating ! Just an optimizer doing its job.

Here's an amusing example

- A spider-like robot "learns" to walk (Loss function)
- Human modifies Loss Function
- Adding the constraint "minimize foot contact with ground"
- In an attempt to get the robot to learn to run

Walking without feet touching the ground



Attribution: [The Surprising Creativity of Digital Evolution: ... (https://arxiv.org/pdf/1803.03453v1.pdf)]

The spider learns to walk on its elbows !

Conclusion

Adversarial Reprogramming is a technique that can abuse an otherwise useful Deep Learning system.

It involves nothing more than an application of techniques that we have already learned

We also briefly mentioned the topic of AI Safety: how Deep Learning systems may come to cause harm.

We have learned powerful tools. It's important to be aware that they can be used for harm as well as good.

In [4]:
```
print("Done")
```

```
Done
```