

Project 5: Machine Learning

This project is due on **Wednesday, Dec 11** at **6 p.m.** and counts for 9% of your course grade.

As with past projects, you may work in **teams of up to two** and submit one project per team. If have trouble forming a team, post to Slack's #project-partner-finder channel.

The code and other answers your group submits must be entirely your own work, and you are bound by the Honor Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically via **Github Classroom** (<https://classroom.github.com/classrooms/88166872-ecen-4133-fall-2024>), following the submission checklist below.

Introduction

In this project, you will make an adversarial example that tricks a face recognition algorithm into thinking that a picture of you is a picture of someone else.

Objectives:

- Learn the basics of machine learning (ML)
- Understand some of the pitfalls and limits of ML-based authentication and ML algorithms more generally
- Implement an adversarial example attack against image classifiers

Part 0. MNIST

To start, we'll investigate what a classifier is, how it works, and how we can “fool” it with an adversarial example (and what that means).

Classifying MNIST numbers

We'll start by using the MNIST training set. MNIST is a dataset of around 70,000 28x28 images of handwritten digits, useful for training basic image classifiers. A sample of the handwritten digits is shown here:



Background

The goal of a classifier is to take as input one of these handwritten digits as an image, and output what digit (0–9) it corresponds to. For instance, the upper-left image above should be classified as a “0”.

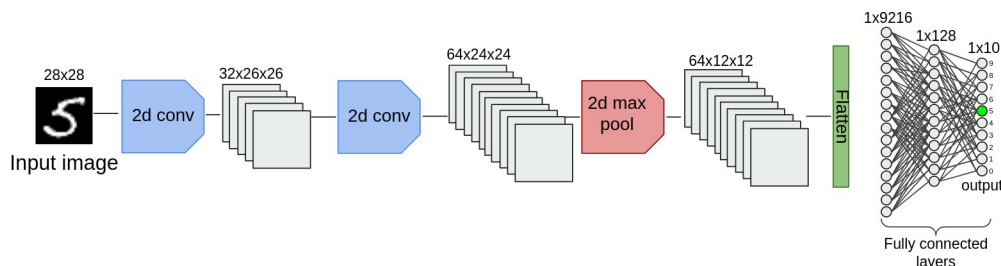
ML-based classifiers work differently than programs you may be used to, which typically use if-else logic to identify conditions. It would be very difficult to write if-else logic to define exactly all the ways that a “0” could be drawn, so we instead often use **neural networks** of weights and functions in machine learning to let a computer tell what digit is in an image. These ML-based classifiers operate similar to how your brain operates, in that there is a network of interconnected “neurons” used to process data, rather than a binary decision process like that in if-else programs.

Just like your brain, ML-based classifiers must first be **trained** to operate effectively. This means that after we define a structure of neurons and how they are connected (e.g. what will it input, how will it be interconnected, and what will it output), we must then give it lots of different labelled examples of images, and adjust the **weights** of the connections between neurons so that it outputs in a way that we want. For instance, we might pass an image of a hand-drawn “4” to our network, and it might output that it thinks that it’s a “7” because it doesn’t know anything yet (and we made it

pick something). Then, we can look at what weights we would need to change in this network in order to get us closer to the network saying this was in fact a “4”, and then make a small adjustment in that direction. We can repeat with thousands of *training* images like this where we know the right answer, each time making the network take a small step to getting the answer more right. Eventually, the network will “**learn**” how to recognize numbers. Once a network has been trained, we can evaluate it by giving it a different set of *test* images that it hasn’t seen before, and seeing how good it is at classifying them. If all has gone well, we’ll get an accuracy of our model, for example maybe it can correctly identify 99% of the images we give it.

Training the model

We’ll start by training a model on the MNIST dataset. The structure of this model has already been defined for you, in `mnsit/mnist.py`. You can see this structure defined in the `Net` module, and illustrated here:



The network involves a couple of 2D convolutions, and two fully-connected layers. The network takes as input a 28x28 grayscale image, where each of the 784 pixels is an integer between 0–255 (0 representing black, and 255 white). The network outputs 10 floating-point numbers, and the highest of these numbers corresponds to the digit that the neural network thinks the input image is (e.g. 5 would be the highest in the above example).

To train the model, run the following command:

```
python3 mnist.py --save-model model.pt --epochs 1
```

This will train the model using 60,000 training images, updating the weights in a way that makes the model more accurate at identifying handwritten digits. After each epoch of training, this code tests the model, measuring its accuracy, which might yield something like:

Test set: Average loss: 0.0519, Accuracy: 9849/10000 (98%)

This is the result of testing the model on 10,000 labelled test images (specifically not used during the training phase) to tell if the model is able to tell what those digits are. The model got 9849 (98%) of them right, which is pretty good! Training for more epochs could improve the accuracy further.

Digging deeper Take a look at how the `train` function works in `mnist.py`. The key part of training involves these steps repeated over our training set of images:

1. Zero out the gradients of the model; we need to clear out any computed gradients, as we'll update them later (`optimizer.zero_grad()`)
2. Run the training image through the model to get an output (`output = model(data)`)
3. Compute the loss: how close to correct is the model's output? (`loss = F.nll_loss(output, target)`)
4. Back propagation: Given the model's answer and how "off" it was (loss), we compute the gradients of all the weights in the model, which tells us what direction (and by how much) the weights should be updated by that would have made our answer closer to correct (`loss.backward()`)
5. Finally, we make the model's weights all take a small step along that computed gradient, toward a more accurate model (`optimizer.step()`)

We repeat these steps many times across our 60,000 training images, and eventually our weights are such that the model can correctly identify the digits in images. This whole process is called *gradient descent*, where we are taking small steps that gradually minimizes our *loss* (the error in how good our model is).

Keep this process in mind, as it will be useful for constructing adversarial examples later!

Using the model

Now that we've trained a model, its weights are stored in `model.pt`, and we can use that to classify images the model hasn't seen before. You can use the `classify.py` script for this to classify an example image (5):

```
python3 classify.py --model model.pt --image 5.png --verbose
```


This will output what it "thinks" the image is (in this case, 5). Because we set the verbose flag, it will also output the relative probabilities that the model computed for each of the digits (with 5 being the highest):

0	0.00
1	0.00
2	0.00
3	0.08
4	0.01
5	97.32
6	0.01
7	0.00
8	0.23
9	2.35

Try drawing your own image and passing it to the classifier. You can use an image editor like Gimp, Paint, or Photoshop. You'll need to make a 28x28 grayscale image, and save it as a PNG without an alpha/transparency channel. Does the model accurately identify your number?

Adversarial Examples: Fooling the model

Now we'll try to come up with an image that to us looks like one number, but to the model looks completely different. This is called an *adversarial example*, and they exploit the fact that the way the model sees the world is very different from how we see it (despite the high-level similarity that we both use neural networks).

In this example, we'll use the  from before, and make small changes to try to trick the model into thinking it is **9** (instead of 5).

First, we'll need to load our model and its weights, and set it in the evaluating mode. Evaluating mode means that the model does not expect to change any of its weights, since we only change its weights during training.

```
from mnist import Net # Only works if in same dir as mnist.py
model = Net()         # Net() defined in mnist.py
model.load_state_dict(torch.load('model.pt'))
model.eval()
```

Next, load the image that we want to make an adversarial example from¹:

```
from PIL import Image
from torchvision import transforms
img = Image.open('5.png')
img_tensor = transforms.ToTensor()(img).unsqueeze(0)
```

This converts our image into a tensor, which is a 2d-array of values between 0.0 – 1.0 (instead of integer pixel values 0 – 255).

If we ran this input through the model now (`model(img_tensor)`), it would tell us it's the digit 5. But we want to figure out how we could change this so the model thinks it's the digit 9. Let's try to compute what the loss would be if we pretended this image was supposed to contain the digit 9, instead of 5:

```
import torch.nn.functional as F
goal = torch.tensor([9], dtype=torch.long)
```

```
# We will repeat the following 250 times
for i in range(250):
```

```
    # This lets us compute gradients on the input.
    # Otherwise, only the model's would be computed.
    img_tensor.requires_grad = True
```

¹You may need to install the PIL and torchvision libraries: `pip3 install Pillow && pip3 install torch torchvision --index-url https://download.pytorch.org/whl/cpu`

```
output = model(img_tensor)
loss = F.nll_loss(output, goal)
```

If we were training the model, we'd do back propagation and compute the gradients of the **model weights** that should change to improve the model. But we don't want to change the model, we want to change the **input**, specifically in a direction that fools the model to misclassifying our image. So we'll still compute back propagation, but instead compute it all the way into the input image:

```
loss.backward()
dx = img_tensor.grad
```

This essentially asks "how should we change the input image so that we get closer to the model telling us this is the digit 9?" The dx variable tells us how the input should be changed so that we move in a direction away from 9. So let's take a small step in the opposite direction, as that would get us closer to being identified as a 9. We'll also clamp the image tensor to bound it to contain numbers between 0 and 1 so that we can easily convert to an image later:

```
adversarial_example = img_tensor - dx*0.001
adversarial_example = torch.clamp(adversarial_example, 0, 1)
```


The reason we take a small step, and not just one giant leap in the direction of the gradient is because the topology of our gradient might not be locally what we need to get to the lowest point. Think about standing on the side of a mountain, and walking toward the direction that is most steep down. If you didn't adjust your direction and just kept walking along that path, you might not end up in a valley at all. Eventually, you might even start climbing back up if you maintained your heading long enough! If you wanted to go downhill the fastest, you should take a small step toward whatever way is currently downhill, then reassess and take a small step in whatever direction is downhill from there, then repeat. We do the same thing here!

We'll assign copy our adversarial_example back into img_tensor, and repeat the loop. We need to call "detach" to tell the torch library that it should forget all the computation it did in loss.backward(), so that we can do it again in the next iteration.

```
img_tensor = adversarial_example.detach()
```

Finally, outside the loop, we'll write the image to a file:

```
img_out = transforms.ToPILImage()(img_tensor.squeeze())
img_out.save('adversarial.png')
```

This should give us an image that the model thinks looks like the digit 9. But what does it look like to you? 

If you look closely, you can see differences between the original and adversarial example image:

Running it against our classifier (python3 classify.py --model model.pt --image adversarial.png --verbose) shows us that the model believes this image contains the digit 9, but it's not fully certain:



(a) Classified as 5



(b) Classified as 9

0	0.00
1	0.00
2	0.00
3	0.42
4	0.08
5	40.84
6	0.00
7	0.00
8	0.70
9	57.94

If we took more steps, it would become more certain that the image is the digit 9. Here's a few more adversarial examples that took more steps along that direction. Eventually, these examples convince the model that the image definitely contains a 9, with more certainty than the model had that the original image contained a 5!



Original: 5 (97%) 250 steps: 9 (58%) 500 steps: 9 (86%) 1000 steps: 9 (94%) 5000 steps: 9 (99%)

What to submit You don't need to submit anything for this part, it is just to familiarize you with making adversarial example images that fools a classifier.

However, if you like your adversarial example, feel free to share it on the [#adversarial-examples](#) channel in Slack. Be sure to share what the classifier thinks the image is!

Part 1.

In this part, you'll fool a machine learning classifier used to identify people based on a picture of their face.

We've created a [facial authentication system](#) (similar to that used by your phone, or the TSA) that inputs a picture of a person's face, and compares it to a small database of known people (our database includes ECEN 4133 students (anonymized), faculty, and staff in the class, and a few notable CU alumni and affiliated people). If the uploaded face seems similar enough to a known face—as determined by a machine learning model based on Facenet—you are logged in as that person. Keep in mind that this “authentication” system is far from perfect, and is prone to errors and misclassifications.

Your task is to create an **adversarial example** from a picture of you (or a unique celebrity if you prefer), that the authentication system mistakenly authenticates as one of CU's most notable alumni²: **Steve Wozniak**.



Figure 1: Woz

You may start with either a picture of you, your project partner, or of anyone other than Steve Wozniak if you prefer, and modify it to produce an adversarial example. The resulting adversarial example image should look like you (or whoever you started with) to a human, but should authenticate as Wozniak on the website: <https://project5.ecen4133.org/>.

To make your adversarial example, we have provided starter code that you will modify in `face-adv.py` within your Github Classroom repository. This script should input a starting image filename (e.g. one of you), a goal image filename (e.g. one of Woz), and a floating-point threshold (e.g. 0.75), and outputs an adversarial example image that looks like the starting image to a human, but classifies as similar to the goal image to the Resnet classifier.

Your script should produce your adversarial image by running something similar to:

```
python3 face-adv.py --image your-image.png --goal woz-image.png --out adv.png --threshold 0.75
```

²Wozniak attended CU in 1969, but was placed on probation for “[computer pranks](#)”, and he left to co-found Apple Computers. He was later awarded an honorary degree from CU in 1989.

This should produce an adv.png image that looks like your-image.png to you, but classifies as similar to woz-image.png.

For instance, if we start with an image of CU Football Deion Sanders (aka Coach Prime) (left), and a goal of classifying as similar to Taylor Swift (middle), your script might produce something like the image on the right:



(a) Coach Prime
(Distance from Swift:
1.4942)



(b) Taylor Swift



(c) Adversarial
example (Distance
from Swift 0.7498)

Testing In addition to the website (<https://project5.ecen4133.org/>), we have provided a test script, face-dist.py to compare two images. Use it by running:

```
python3 face-dist.py --image img1.png --compare img2.png
```

The program will output a floating point number of the “distance” between the faces in the image, with lower being more similar. We use a threshold of 0.75 on the site to identify two images as containing the same person. If face-dist.py outputs a number lower than 0.75 for your adversarial image and an image of Steve Wozniak, it should authenticate as Woz on the site³.

What to submit

1. face-adv.py - your code that produces an adversarial example that fools our face detection system.
2. adv.png - an adversarial example image that looks like you (or a person besides Wozniak) to a human, but classifies as Steve Wozniak when uploaded to the website.

Submit your code via Github Classroom: <https://classroom.github.com/a/B1KbNLvc>

³Keep in mind that different pictures of Woz will have different distances, so you may have to target a slightly lower distance to work on all Woz images