#ret #hw

---

1 | Cryptography

Done prior:

```
-   Try turning on two-factor authentication, if you have that option.
-   Think about how secure your password is, relative to how attackers would try guessing. Is it a dict
-   Generate a public/private key pair for yourself. Put the public key on our test laptop so that you
```

1.1 | Creating a custom hash function

```
Hey Wes, this is a bit of a weirder one.
```

1.1.1 | Requirements

What are the requirements for a hash function?

- Source
  - No preimage: given y, it should not be feasible to find x such that $h(x) = y$.
  - No second preimage: given x_1, it should not be feasible to find x_2 (distinct from x_1) such that $h(x_1) = h(x_2)$.
  - No collision: it should not be feasible to find any x_1 and x_2 (distinct from each other) such that $h(x_1) = h(x_2)$.

```
What if we just.. use a neural network?
```

1.1.2 | NN hash

Intuitively, using a neural network as a hash function seems like it won't work. In fact, I believe it doesn't work, **I just don't know why it doesn't work. *** Of course, a vanilla neural network won't work because we can just train a model to reverse its mapping. To solve this problem, we can use something I call permute layers.

1. Permute Layers **Concept –** Fundamentally, the concept of permute layers is to take an input, and do some operation on it such that a non-continuous output space is generated. The goal would be that: - Similar inputs lead to vastly different outputs - Make there no guarantee that an adversarial NN's guess of x is closer to $x + \epsilon_0$ than it is to $x + \frac{1}{\epsilon_0}$ - As in, we can't train a NN to reverse it!

   Implementation – One possible implementation of these permute layers would be simply permuting the bits that make up our tensors.

1.1.3 | Proof of concept

The code below is meant as proof of concept – or rather, demonstration of concept. It is messy and surely error ridden, but it seems to work.

Outline: - Create a deep neural network with randomly initialized weights. - Ensure that it is deterministic with a set seed. This seed could potentially be carried with the hash. - Add permute layers in between the

The code can also be found here.

```
####################
#       SETUP      #
####################

# imports
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
import struct
from codecs import decode
import numpy as np
import string

INP = [0.1, 0.0, 1.01] # our input!
SAFE = 8 # don't permute the first 8 bits, so we don't get infs and 0

# makes things deterministic
np.random.seed(0)
torch.manual_seed(0)

torch.set_default_dtype(torch.float64) # make sure we use the right datatype!


######################
#       BASE NN      #
######################

class Net(nn.Module): # define the model

    def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(3, 128) # linear layer, with input size 3
        self.pl1 = PermuteLayer(128,256) # custom permute layer

        self.l2 = nn.Linear(256, 512)
        self.pl2 = PermuteLayer(512, 512)

        self.l3 = nn.Linear(512, 256)
        self.pl3 = PermuteLayer(256, 256)

        self.l4 = nn.Linear(256, 128)
        self.pl4 = PermuteLayer(128, 8) # output a tensor with 8 floats
```

```
    def forward(self, x): # run it through!
        x = [100*(y+1) for y in x] # add 1 and multiply by 100 for each input element
        x = torch.tensor(x) # then convert it to a tensor

        x = self.l1(x) # run it through the layers
        x = x.view(-1, 128)
        x = self.pl1(x)
        x = self.l2(x)
        x = self.pl2(x)
        x = self.l3(x)
        x = self.pl3(x)
        x = self.l4(x)
        x = self.pl4(x)

        return x


####################################
#       CUSTOM PERMUTE LAYER       #
####################################

class PermuteLayer(nn.Module): # not my code! default linear code comes from https://auro-227.medium.co
    # after modification, acts as a normal linear layer except it permutes the bits.
    def __init__(self, size_in, size_out):
        super().__init__()
        self.size_in, self.size_out = size_in, size_out
        weights = torch.Tensor(size_out, size_in)
        self.weights = nn.Parameter(weights)  # nn.Parameter is a Tensor that's a module parameter.
        bias = torch.Tensor(size_out)
        self.bias = nn.Parameter(bias)

        # initialize weights and biases
        nn.init.kaiming_uniform_(self.weights, a=math.sqrt(5)) # weight init
        fan_in, _ = nn.init._calculate_fan_in_and_fan_out(self.weights)
        bound = 1 / math.sqrt(fan_in)
        nn.init.uniform_(self.bias, -bound, bound)  # bias init

    def forward(self, x): # where the permuting happens
        # this part isn't pretty..
        # but according to Dr. Brian Dean, we don't need to constant factor optimize!

        bits = "" # store bits in a char array
        saved = [] # save the bits we want to protect

        for i,v in enumerate(x[0]): # loop through the floats
            tnsr = float_to_bin(v) # convert them to binary
            saved.append(tnsr[:SAFE]) # save what we need to
            bits += tnsr[SAFE:] # and add to the char array

        p = np.random.permutation([x for x in bits]) # permute it!
        p = ''.join(map(str, p)) # and then.. join it back together

        converted = []
```

```
        # loop through p, chunk it into segments
        for i in range(len(p)//(64-SAFE)):
            # convert segment to floats
            item = bin_to_float(saved[i]+p[(64-SAFE)*i:((64-SAFE)*i)+(64-SAFE)])
            converted.append(item)

        converted = torch.tensor([converted]) # change it back to a tensor
        x = converted

        w_times_x= torch.mm(x, self.weights.t()) # matrix multiply them
        return torch.add(w_times_x, self.bias)  # w times x + b


#########################
#       HELPERS         #
#########################

# not my code! from https://stackoverflow.com/questions/16444726/binary-representation-of-float-in-pyth
def bin_to_float(b):
    """ Convert binary string to a float. """
    bf = int_to_bytes(int(b, 2), 8)  # 8 bytes needed for IEEE 754 binary64.
    return struct.unpack('>d', bf)[0]

def int_to_bytes(n, length):  # Helper function
    """ Int/long to byte string.
        Python 3.2+ has a built-in int.to_bytes() method that could be used
        instead, but the following works in earlier versions including 2.x.
    """
    return decode('%%0%dx' % (length << 1) % n, 'hex')[-length:]

def float_to_bin(value):  # For testing.
    """ Convert float to 64-bit binary string. """
    [d] = struct.unpack(">Q", struct.pack(">d", value))
    return '{:064b}'.format(d)


def int2base(x, base): # not my code! modified from https://stackoverflow.com/questions/2267362/how-to-
    if x < 0:    sign = -1
    elif x == 0: return digs[0]
    else:        sign = 1
    x *= sign
    digits = []
    while x:
        digits.append(digs[x % base])
        x = x // base
    if sign < 0: digits.append('-')
    digits.reverse()
    return ''.join(digits)
digs = string.digits + string.ascii_letters

########################
#       OUTPUT         #
########################
```

```
model = Net()

result = list(model(INP).detach().numpy()[0]) # convert output to list

output_bits = ''
for i in result:
    # convert to bits, then take the second half
    # because it's more shuffled
    output_bits += float_to_bin(i)[32:]

print(int2base(int(output_bits, 2), 16)) # clean the output up and print it out
```