Machine-Learning Based Encryption via Learned Character Permutations

Samuel Cavazos

August 24, 2025

Abstract

A cipher is covered early on in elementary cryptography courses because it provides intuition about more complicated encryption methods. In this paper, we implement a random substitution cipher in Python and train two neural networks to learn the mapping. The models, implemented in PyTorch, serve as an **encoder** (plaintext → ciphertext) and a **decoder** (ciphertext → plaintext). Together, they demonstrate how machine learning can memorize and reproduce encryption and decryption operations for a fixed, randomly sampled key (permutation) over a finite character vocabulary.

1 Introduction

Can a neural network learn to encrypt and decrypt messages?

To explore this question, we construct a random substitution cipher by permuting a fixed character vocabulary (letters, digits, punctuation, and whitespace). We then train two neural networks on per-character classification tasks:

- An **encoder** that maps each plaintext character to its ciphered character, and
- A **decoder** that inverts this mapping to recover plaintext from ciphertext.

We implement this setup in Python with PyTorch. Each conceptual step is encapsulated by a small class: `Characters` (vocabulary), `Cipher` (random permutation and training pairs), `CipherDataset` (DataLoader wrapper), and `Architecture` (Embedding → Linear).

2 Background

Classical encryption converts plaintext P into ciphertext C using a key K:

$$C = E_K(P), \tag{1}$$

and decryption reverses it:

$$P = D_K(C). \tag{2}$$

Here, K is a random permutation π of the character set V, i.e., $K \equiv \pi$ and $K^{-1} \equiv \pi^{-1}$.

In our ML formulation, E and D are approximated by neural networks that operate at the character level. Given an input character $x \in V$ and its target $y \in V$, the model produces logits $z \in \mathbb{R}^{|V|}$ and a categorical distribution:

$$p(y \mid x) = \mathrm{softmax}(z), \qquad z = f_\theta(x), \tag{3}$$

and is trained with cross-entropy:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(y_i \mid x_i). \tag{4}$$

3 Proposed Method

3.1 Character Vocabulary (Characters)

We define a fixed vocabulary V by concatenating ASCII letters, digits, punctuation, and whitespace. This establishes the index space $\{0, \ldots, |V| - 1\}$ for categorical modeling.

Code (from this project):

```
import torch
import string

# ---------- Characters ----------
class Characters:
    def __init__(self):
        LETTERS = string.ascii_letters
        DIGITS = string.digits
        PUNCTUATION = string.punctuation
        WHITESPACE_CHARACTERS = ' \t\n\r\x0b\x0c'
        self.characters = LETTERS + DIGITS + PUNCTUATION + WHITESPACE_CHARACTERS
        self.num_characters = len(self.characters)

    def read(self, indices: torch.Tensor):
        # ensure Python ints
        return ''.join(self.characters[int(i)] for i in indices)

    def index(self, text: str):
        return torch.tensor([self.characters.index(c) for c in text],
                            dtype=torch.long)
```

3.2 Random Substitution Cipher and Supervision Pairs (Cipher)

We sample a random permutation π over V and construct paired supervision for both directions:

$$\mathcal{D}_{\text{enc}} = \{(x, \pi(x)) \mid x \in V\}, \qquad \mathcal{D}_{\text{dec}} = \{(\pi(x), x) \mid x \in V\}.$$

Code:

```python
import random
import torch
from types import SimpleNamespace as SN

# ---------- Cipher mapping ----------
class Cipher:
    def __init__(self):
        self.char = Characters()
        n = self.char.num_characters
        self.original_indices = list(range(n))
        self.shuffled_indices = self.original_indices.copy()
        random.shuffle(self.shuffled_indices)

        # tensors of pairs (src, dst)
        self.training_data = SN()
        self.training_data.encoder = torch.tensor(
            [self.original_indices, self.shuffled_indices], dtype=torch.long
        ).T  # map original -> shuffled
        self.training_data.decoder = torch.tensor(
            [self.shuffled_indices, self.original_indices], dtype=torch.long
        ).T  # map shuffled -> original

cipher = Cipher()
```

3.3 Dataset Wrapper and DataLoaders (`CipherDataset`)

We wrap the pair tensors in a PyTorch `Dataset` so batches are (x, y) index pairs.
Code:

```python
import torch.utils.data as data

class CipherDataset(data.Dataset):
    def __init__(self, pairs_tensor):  # shape [N, 2]
        self.data = pairs_tensor
    def __len__(self):
        return self.data.shape[0]
    def __getitem__(self, idx):
        x, y = self.data[idx]
        return x.long(), y.long()  # scalars

encoder_dataset = CipherDataset(cipher.training_data.encoder)
decoder_dataset = CipherDataset(cipher.training_data.decoder)

batch_size = 32
encoder_loader = data.DataLoader(encoder_dataset, batch_size=batch_size, shuffle=True)
decoder_loader = data.DataLoader(decoder_dataset, batch_size=batch_size, shuffle=True)
```

3.4 Neural Model (`Architecture`: Embedding → Linear)

Because each supervision example is a single character, we use a compact per-character classifier:

$$e = \text{Embedding}(x) \in \mathbb{R}^d, \quad z = We + b \in \mathbb{R}^{|V|}, \quad p = \text{softmax}(z).$$

This architecture can perfectly memorize a permutation on V.
Code:

```
import torch.nn as nn
import torch

# ---------- Tiny per-char model ----------
class Architecture(nn.Module):
    def __init__(self, num_chars, emb_dim=64):
        super().__init__()
        self.emb = nn.Embedding(num_chars, emb_dim)
        self.out = nn.Linear(emb_dim, num_chars)
    def forward(self, x):               # x: [B]
        e = self.emb(x)                 # [B, E]
        logits = self.out(e)            # [B, V]
        return logits
```

4 Training Objective and Loop

We use cross-entropy on logits vs. integer class targets, optimized by Adam. The evaluation helper computes loss and accuracy over all $x \in V$ for a given mapper.
Code:

```
import torch

V = cipher.char.num_characters
encoder = Architecture(V)
decoder = Architecture(V)

criterion = nn.CrossEntropyLoss()
enc_opt = torch.optim.Adam(encoder.parameters(), lr=2e-3)
dec_opt = torch.optim.Adam(decoder.parameters(), lr=2e-3)

def eval_mapper(model, pairs_tensor):
    model.eval()
    with torch.no_grad():
        x = pairs_tensor[:, 0]   # inputs
        y = pairs_tensor[:, 1]   # targets
        logits = model(x)        # [N, V]
        loss = criterion(logits, y)
        pred = logits.argmax(dim=-1)
        acc = (pred == y).float().mean().item()
    return loss.item(), acc
```

```
max_epochs = 500
target_acc = 1.0  # perfect memorization possible for a permutation

for epoch in range(1, max_epochs + 1):
    encoder.train()
    for x, y in encoder_loader:       # original -> shuffled
        enc_opt.zero_grad()
        logits = encoder(x)           # [B, V]
        loss = criterion(logits, y)   # y: [B]
        loss.backward()
        enc_opt.step()

    decoder.train()
    for x, y in decoder_loader:       # shuffled -> original
        dec_opt.zero_grad()
        logits = decoder(x)
        loss = criterion(logits, y)
        loss.backward()
        dec_opt.step()

    enc_loss, enc_acc = eval_mapper(encoder, cipher.training_data.encoder)
    dec_loss, dec_acc = eval_mapper(decoder, cipher.training_data.decoder)

    print(f"Epoch {epoch:03d} | enc_loss={enc_loss:.4f} acc={enc_acc:.3f} "
          f"| dec_loss={dec_loss:.4f} acc={dec_acc:.3f}")

    if enc_acc == target_acc and dec_acc == target_acc:
        torch.save(encoder.state_dict(), "encoder.pth")
        torch.save(decoder.state_dict(), "decoder.pth")
        print("Training complete.")
        break
```

5 Results

On a typical run, both encoder and decoder rapidly approach 100% accuracy across the entire vocabulary, confirming that the models fully memorize the sampled permutation. Once converged, arbitrary plaintext can be transformed by the encoder and recovered exactly by the decoder.

6 Discussion

This experiment is not intended as a secure cryptosystem; rather, it isolates a key insight: neural networks can internalize a *keyed function* such as a permutation over a discrete alphabet. The `Characters` class fixes the domain V, `Cipher` samples a key π, `CipherDataset` packages supervision for both directions, and `Architecture` provides a minimal per-character classifier (Embedding $\rightarrow$ Linear) that achieves perfect accuracy on this task. This framework is therefore a compact testbed for reasoning about learned encryption/decryption behaviors and their limits (e.g., generalization beyond a fixed key, robustness to noise, or extension to longer contexts).