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. While we begin with this simple cipher, the flexibility of neural models enables more sophisticated approaches—for example, using learnable *embeddings* to represent symbols as vectors and composing them with richer architectures which allows for the possibility of more complicated and secure encryption methods.

1 Problem Setup and Notation

Let V denote a finite character vocabulary with $|V| = n$. We index symbols by integers $\{0, 1, \ldots, n-1\}$ and write $\pi \in S_n$ for a key sampled uniformly at random: $\pi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ is a permutation. The *encryption* function is $E_\pi(i) = \pi(i)$ and the *decryption* function is $D_\pi(j) = \pi^{-1}(j)$.

We cast learning E_π and D_π as two multi-class classification problems over n classes. The supervised pairs are

$$\mathcal{D}_{\text{enc}} = \{(x, \pi(x)) : x \in \{0, \ldots, n-1\}\}, \qquad \mathcal{D}_{\text{dec}} = \{(\pi(x), x) : x \in \{0, \ldots, n-1\}\}.$$

One-hot and embeddings. Let $e_i \in \mathbb{R}^n$ denote the i-th standard basis vector. We will use a learnable embedding $E \in \mathbb{R}^{n\times d}$ that maps index i to $E^\top e_i \in \mathbb{R}^d$ (equivalently, the i-th row of E).

2 Model Architecture

For each direction we use the same per-character classifier. Given input index $x \in \{0, \ldots, n-1\}$, the model computes

$$\begin{aligned} \text{Embedding:} \quad & h = E^\top e_x \in \mathbb{R}^d, && (1)\\ \text{Logits:} \quad & z = Wh + b \in \mathbb{R}^n, && (2)\\ \text{Class probs:} \quad & p(y \mid x) = \operatorname{softmax}(z)_y = \frac{\exp(z_y)}{\sum_{k=0}^{n-1} \exp(z_k)}, && (3) \end{aligned}$$

with parameters $E \in \mathbb{R}^{n\times d}$, $W \in \mathbb{R}^{n\times d}$, $b \in \mathbb{R}^n$. The prediction is $\hat{y} = \arg\max_y p(y \mid x)$.

Expressivity (exact realization).

Proposition 1 (Exact realizability via rank factorization)**.** *Let V be a vocabulary with $|V| = n$ and let $\pi \in S_n$ be a permutation. Consider the model $x \mapsto \operatorname{softmax}(WE^\top e_x + b)$ with $E \in \mathbb{R}^{n\times d}$, $W \in \mathbb{R}^{n\times d}$, $b \in \mathbb{R}^n$. If $d \geq n$, then there exist parameters (E, W, b) such that*

$$\arg\max_y \ \operatorname{softmax}(WE^\top e_x + b)_y \ = \ \pi(x) \quad \textit{for all } x \in \{0, \ldots, n-1\}.$$

Constructive proof. Let $P_\pi \in \mathbb{R}^{n\times n}$ be the permutation matrix for π (i.e., $(P_\pi)_{y,x} = 1$ iff $y = \pi(x)$). Pick $E = [I_n \ \ 0] \in \mathbb{R}^{n\times d}$ (so $E^\top = [I_n; 0]$), $W = [P_\pi \ \ 0] \in \mathbb{R}^{n\times d}$, and $b = 0$. Then for any input index x,

$$h = E^\top e_x = \begin{bmatrix} I_n \\ 0 \end{bmatrix} e_x = e_x, \qquad z = Wh = \begin{bmatrix} P_\pi & 0 \end{bmatrix} e_x = P_\pi e_x = e_{\pi(x)}.$$

Hence $\operatorname{softmax}(z)$ is maximized uniquely at class $\pi(x)$, as required. □

Remark 1 (Minimality for exact realization)*.* If one requires the *exact* linear realization $WE^\top = P_\pi$ (so that $z = WE^\top e_x$ equals $e_{\pi(x)}$ before softmax), then necessarily $d \geq n$. Indeed, $\operatorname{rank}(P_\pi) = n$ while $\operatorname{rank}(WE^\top) \leq \min\{\operatorname{rank}(W), \operatorname{rank}(E)\} \leq d$. Thus $d \geq n$ is both sufficient (by the construction above) and necessary for exact equality $WE^\top = P_\pi$. This is an instance of the *rank (full-rank) factorization* theorem.[1]

Remark 2 (On the weaker argmax requirement)*.* Our practical objective only requires $z_{\pi(x)} > z_y$ for all $y \neq \pi(x)$, not $z = e_{\pi(x)}$. Therefore, in practice one often achieves perfect classification with $d \ll n$; the construction and Remark 1 address *exact* linear realization.

This guarantees that a shallow embedding+linear model can perfectly implement any substitution cipher when $d \geq n$; in practice we use $d \ll n$ and still find the mapping by optimization.

[1]See, e.g., R. A. Horn and C. R. Johnson, *Matrix Analysis* (2nd ed.), Cambridge Univ. Press, or G. H. Golub and C. F. Van Loan, *Matrix Computations* (4th ed.), Johns Hopkins Univ. Press.

3 Learning Objective

For either direction (encoder or decoder) with dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, the negative log-likelihood (cross-entropy) is

$$\mathcal{L}(\theta) \;=\; -\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(y_i \,|\, x_i) \;=\; -\frac{1}{N}\sum_{i=1}^{N}\left(z_{i,y_i} - \log\sum_{k=0}^{n-1} e^{z_{i,k}}\right), \tag{4}$$

where $z_i = WE^\top e_{x_i} + b$ are the logits for example i and $\theta = (E, W, b)$. We optimize with Adam. Accuracy is $\frac{1}{N}\sum_i \mathbf{1}\{\arg\max_k z_{i,k} = y_i\}$.

Computational notes. Per example, the forward pass costs $O(dn)$ to produce logits (matrix-vector multiply Wh), and the softmax normalization costs $O(n)$. Thus the softmax dimension $n = |V|$ dominates compute and memory, motivating an ASCII-sized vocabulary in our initial experiments.

4 Implementation (Minimal, Self-Contained)

We present compact Python listings that realize the mathematics.

4.1 Vocabulary

We fix V by concatenating ASCII letters, digits, punctuation, and whitespace, and provide index↔character utilities.

```
import string, torch

class Characters:
    def __init__(self):
        self.characters = (
            string.ascii_letters +
            string.digits +
            string.punctuation +
            " \t\n\r\x0b\x0c"
        )
        self.num_characters = len(self.characters)
    def index(self, text: str) -> torch.Tensor:
        return torch.tensor([self.characters.index(c) for c in text], dtype=torch.long)
    def read(self, indices: torch.Tensor) -> str:
        return "".join(self.characters[int(i)] for i in indices)
```

4.2 Random substitution supervision

We sample π and build paired supervision for E_π and D_π.

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

```python
import random, torch

class Cipher:
    def __init__(self):
        self.char = Characters()
        n = self.char.num_characters
        self.pi = list(range(n)); random.shuffle(self.pi)
        # paired tensors: (src, dst)
        self.enc_pairs = torch.tensor([range(n), self.pi], dtype=torch.long).T
        self.dec_pairs = torch.tensor([self.pi, range(n)], dtype=torch.long).T

cipher = Cipher()
```

4.3 Architecture (Embedding → Linear)

This implements $h = E^\top e_x$, $z = Wh + b$, $p = \mathrm{softmax}(z)$.

```python
import torch.nn as nn

class Architecture(nn.Module):
    def __init__(self, num_chars: int, emb_dim: int = 64):
        super().__init__()
        self.emb = nn.Embedding(num_chars, emb_dim)  # rows = E
        self.out = nn.Linear(emb_dim, num_chars)     # W, b
    def forward(self, x):                             # x: [B]
        return self.out(self.emb(x))                  # logits: [B, |V|]
```

4.4 Training loop (encoder and decoder)

Two instances of the same architecture are trained on the two directions.

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

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

# datasets of (x, y) indices
enc_ds = data.TensorDataset(cipher.enc_pairs[:,0], cipher.enc_pairs[:,1])
dec_ds = data.TensorDataset(cipher.dec_pairs[:,0], cipher.dec_pairs[:,1])
enc_loader = data.DataLoader(enc_ds, batch_size=32, shuffle=True)
dec_loader = data.DataLoader(dec_ds, batch_size=32, shuffle=True)

@torch.no_grad__()
def eval_mapper(model, pairs):
    x, y = pairs[:,0], pairs[:,1]
    logits = model(x)
    loss = criterion(logits, y).item()
    acc = (logits.argmax(-1) == y).float().mean().item()
```

```
    return loss, acc

for epoch in range(1, 501):
    # encoder step
    encoder.train()
    for x, y in enc_loader:
        enc_opt.zero_grad(set_to_none=True)
        loss = criterion(encoder(x), y)
        loss.backward(); enc_opt.step()
    # decoder step
    decoder.train()
    for x, y in dec_loader:
        dec_opt.zero_grad(set_to_none=True)
        loss = criterion(decoder(x), y)
        loss.backward(); dec_opt.step()
    # evaluation on all symbols
    enc_loss, enc_acc = eval_mapper(encoder, cipher.enc_pairs)
    dec_loss, dec_acc = eval_mapper(decoder, cipher.dec_pairs)

    if enc_acc == 1.0 and dec_acc == 1.0:
        torch.save(encoder.state_dict(), "encoder.pth")
        torch.save(decoder.state_dict(), "decoder.pth")
        print("Training complete."); break
```

5 Results

On an ASCII-sized vocabulary, both models rapidly achieve 100% accuracy on their respective mappings, demonstrating that the embedding+linear architecture can memorize and invert a random permutation of V.

Experimental Protocol for Figures

To quantify learning speed and variability, we run a sweep of T independent trainings (e.g., $T = 1000$), each with a fresh random permutation π and fixed hyperparameters (batch size 32, embedding dimension 64, learning rate 2×10^{-3}, maximum epochs 300). For each run we log per-epoch metrics and define the *epochs-to-convergence*

$$\tau \ = \ \min\{\, e \in \mathbb{N} \mid \texttt{enc\_acc}_e \geq 1.0 \ \wedge \ \texttt{dec\_acc}_e \geq 1.0 \,\}.$$

Aggregate training curves compute, for each epoch e, the median and interquartile range (IQR, 25–75th percentile) across all trials that reached epoch e. Wall-clock time is measured per run with a simple start/stop timer.

6 Discussion

The experiment isolates the mathematical core of learned substitution: logits $z = Wh+b$ over n classes, a cross-entropy objective, and a small embedding dimension d. The proposition

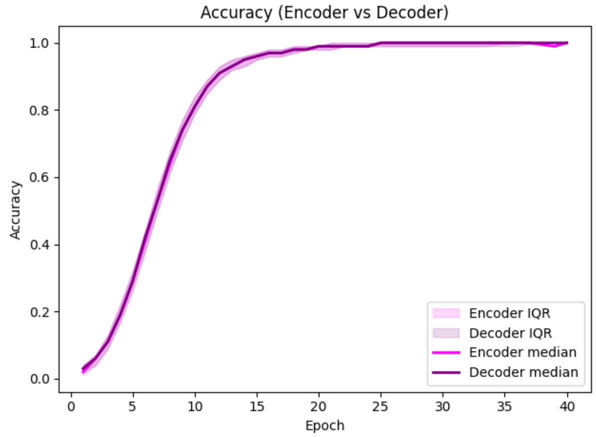


Figure 1: **Accuracy (median & IQR).** Encoder (magenta) vs. decoder (purple). Shaded bands indicate IQR across trials; solid curves are medians. The two curves nearly overlap, reflecting symmetry of the forward and inverse mappings.

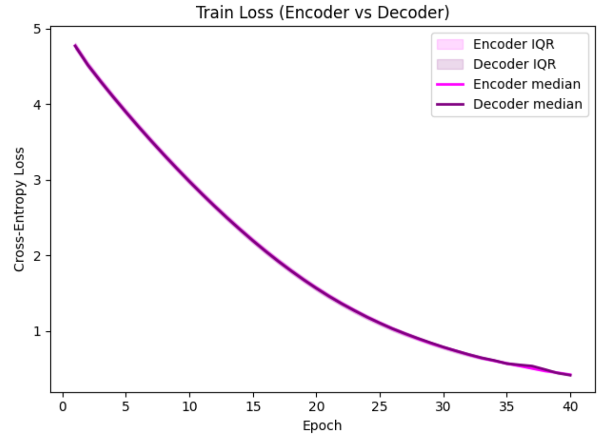


Figure 2: **Train loss (median & IQR).** Cross-entropy loss for encoder (magenta) and decoder (purple). Loss decays smoothly to near-zero as the permutation is memorized.

shows exact realizability when $d \geq n$; empirically, $d \ll n$ suffices in practice for this finite task. The aggregated curves in Figures 1–2 show fast, stable convergence, while Figure 3 summarizes the distribution of required epochs and wall-clock time across random keys.

Security note (single mention). This study examines mechanics of learned mappings and is not intended as a cryptosystem.

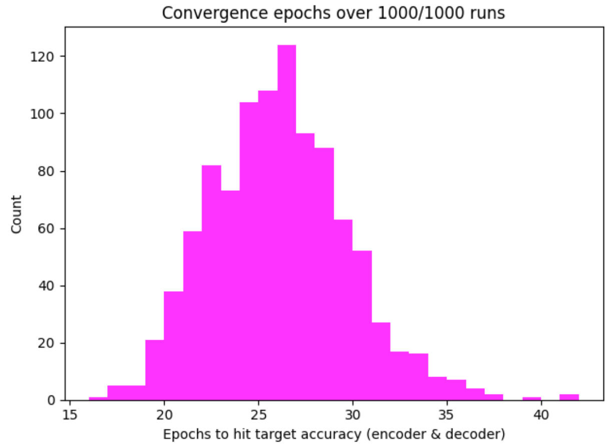
7 Toward Richer Settings

Two directions naturally follow:

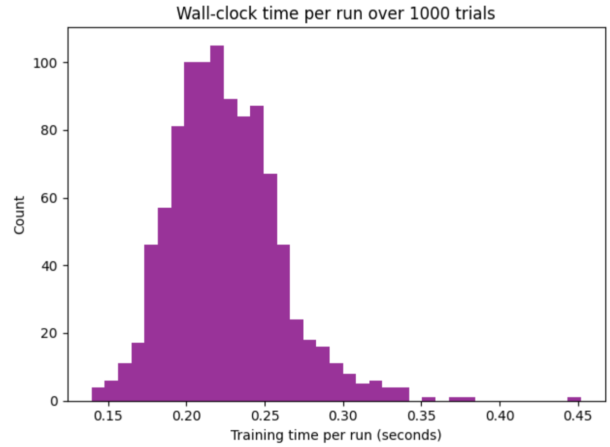
- **Scaling $|V|$.** Moving beyond ASCII toward UTF encodings increases the softmax dimension and compute; subword vocabularies or hierarchical classifiers may mitigate the $O(n)$ normalization cost.
- **Sequence conditioning.** Replace per-character substitution with key-conditioned sequence models; analyze behavior under chosen-plaintext/ciphertext regimes and investigate integrity constraints.

8 Code and Installation

The full source code for this project is available at: `github.com/Alshivals-Data-Service/alshicrypt`.



(a) **Epochs to convergence.** Histogram of τ across runs. Mass concentrates in the low tens of epochs, consistent with rapid memorization of a finite permutation.



(b) **Wall-clock time per run.** Distribution of end-to-end training time (seconds) for one encoder/decoder pair on the given hardware.

Figure 3: **Convergence behavior over many independent trainings.** Left: how many epochs until both models reach 100% accuracy. Right: elapsed time per run.

Install (from GitHub). Requires Python 3.9+ and PyTorch 2.0+.

```
pip install "git+https://github.com/Alshivals-Data-Service/alshicrypt.git"
# (If PyTorch is missing, install a wheel appropriate for your system first:
# https://pytorch.org/get-started/locally/)
```

Quick start (train, save, load, use).

```
import alshicrypt

# Train & save to stable folders (each call samples a new random key)
crypt1 = alshicrypt.generate(epochs=200, outdir="artifacts/crypt-hello")
crypt2 = alshicrypt.generate(epochs=200, outdir="artifacts/crypt-world")

# Load them later (or in a new process)
crypt1 = alshicrypt.load("artifacts/crypt-hello")
crypt2 = alshicrypt.load("artifacts/crypt-world")

msg = "Hello, World!"
enc = crypt1.encode(msg)
dec = crypt1.decode(enc)
print("crypt1")
print("============================")
print(f"Original: {msg}")
print(f"Encoded:  {enc}")
print(f"Decoded:  {dec}\n")
assert dec == msg

enc2 = crypt2.encode(msg)
```

```python
dec2 = crypt2.decode(enc2)
print("crypt2")
print("============================")
print(f"Original: {msg}")
print(f"Encoded:  {enc2}")
print(f"Decoded:  {dec2}")
assert dec2 == msg
```

```
crypt1
============================
Original: Hello, World!
Encoded:  D<~~rJB*r)~W]
Decoded:  Hello, World!


crypt2
============================
Original: Hello, World!
Encoded:  d!VVm\x0b`Nm4V|-
Decoded:  Hello, World!
```

Each trained pair is stored under the specified `artifacts/` directory (e.g., `artifacts/crypt-hello`). The `encode` method applies the learned plaintext→ciphertext mapping; `decode` applies the inverse mapping. Characters outside the training vocabulary are passed through unchanged.