Data Encoding

Morse A-E

```
symbol_vectors = {
    '.': [1, 0, 0],
    '-': [0, 1, 0],
    '_': [0, 0, 1],
}

def encode_morse(seq: str, max_len: int):
    """
    Encode a Morse string into a flat vector of length max_len * 3.
    Pads with '_' (void) on the right.
    """
    vec = []
    for c in seq:
        vec.extend(symbol_vectors[c])
    for _ in range(max_len - len(seq)):
        vec.extend(symbol_vectors['_'])
    return vec

# Morse codes for A–E
morse_dict = {
    'A': '.-',
    'B': '-...',
    'C': '-.-.',
    'D': '-..',
    'E': '.',
}

# figure out max length and input size
max_len = max(len(code) for code in morse_dict.values())
input_size = max_len * 3

# list of classes in order
letters = list(morse_dict.keys())

# build X (inputs) and Y (one-hot targets)
X_train = [encode_morse(morse_dict[ch], max_len) for ch in letters]

Y_train = []
for ch in letters:
    target = [0] * len(letters)
    target[letters.index(ch)] = 1  # one-hot
    Y_train.append(target)

print("Letters:", letters)
print("Max Morse length:", max_len)
```

```
print("Input size:", input_size)
print("Example 'A' encoded:", X_train[0])
print("Target for 'A':", Y_train[0])
```

```
Letters: ['A', 'B', 'C', 'D', 'E']
Max Morse length: 4
Input size: 12
Example 'A' encoded: [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1]
Target for 'A': [1, 0, 0, 0, 0]
```

Model Definition

```
import math
import random

def sigmoid(x: float) -> float:
    """Sigmoid activation: squashes any real value into (0,1)."""
    return 1.0 / (1.0 + math.exp(-x))

def dsigmoid_from_output(y: float) -> float:
    """
    Derivative of sigmoid using its output.
    If y = sigmoid(x), then derivative w.r.t x is y * (1 - y).
    """
    return y * (1.0 - y)


class SimpleMorseNN:
    def __init__(self, input_size: int, hidden_size: int, output_size:
int,
                 lr: float = 0.5, seed: int = 0):
        self.lr = lr
        random.seed(seed)

        self.W1 = [
            [(random.random() - 0.5) * 0.2 for _ in range(input_size)]
            for _ in range(hidden_size)
        ]
        self.b1 = [(random.random() - 0.5) * 0.2 for _ in
range(hidden_size)]

        self.W2 = [
            [(random.random() - 0.5) * 0.2 for _ in
range(hidden_size)]
            for _ in range(output_size)
        ]
        self.b2 = [(random.random() - 0.5) * 0.2 for _ in
range(output_size)]

        self.last_input = None
```

```
        self.last_hidden = None
        self.last_output = None

    def forward(self, x):
        """
        Forward pass: compute hidden activations and outputs for input
x.
        Stores values so backward() can reuse them.
        """
        h = []
        for j in range(len(self.W1)):
            s = sum(w * x_i for w, x_i in zip(self.W1[j], x)) +
self.b1[j]
            h.append(sigmoid(s))

        y = []
        for k in range(len(self.W2)):
            s = sum(w * h_j for w, h_j in zip(self.W2[k], h)) +
self.b2[k]
            y.append(sigmoid(s))

        self.last_input = x
        self.last_hidden = h
        self.last_output = y

        return y

    def backward(self, target):
        """
        Backward pass (backpropagation):
        - uses last forward pass
        - updates weights and biases
        - returns error for this sample
        """
        x = self.last_input
        h = self.last_hidden
        y = self.last_output

        delta_out = []
        for k in range(len(y)):
            error_k = y[k] - target[k]
            delta_k = error_k * dsigmoid_from_output(y[k])
            delta_out.append(delta_k)

        delta_hid = []
        for j in range(len(h)):
            downstream = sum(delta_out[k] * self.W2[k][j] for k in
range(len(delta_out)))
            delta_j = downstream * dsigmoid_from_output(h[j])
            delta_hid.append(delta_j)
```

```
        for k in range(len(self.W2)):
            for j in range(len(self.W2[k])):
                self.W2[k][j] -= self.lr * delta_out[k] * h[j]
            self.b2[k] -= self.lr * delta_out[k]

        for j in range(len(self.W1)):
            for i in range(len(self.W1[j])):
                self.W1[j][i] -= self.lr * delta_hid[j] * x[i]
            self.b1[j] -= self.lr * delta_hid[j]

        E = 0.5 * sum((y[k] - target[k])**2 for k in range(len(y)))
        return E
```

Training Loop

```
hidden_size = 8
output_size = len(letters)
learning_rate = 0.5
epochs = 2000

nn = SimpleMorseNN(input_size, hidden_size, output_size,
                   lr=learning_rate, seed=0)

errors_per_epoch = []

for epoch in range(epochs):
    total_error = 0.0
    for x, y in zip(X_train, Y_train):
        nn.forward(x)
        total_error += nn.backward(y)
    errors_per_epoch.append(total_error)

    if epoch % 200 == 0:
        print(f"Epoch {epoch:4d} | total error = {total_error:.6f}")

print("Training finished.")
```

```
Epoch    0 | total error = 2.907471
Epoch  200 | total error = 0.495248
Epoch  400 | total error = 0.051464
Epoch  600 | total error = 0.023627
Epoch  800 | total error = 0.015004
Epoch 1000 | total error = 0.010884
Epoch 1200 | total error = 0.008494
Epoch 1400 | total error = 0.006941
Epoch 1600 | total error = 0.005855
Epoch 1800 | total error = 0.005055
Training finished.
```

Testing & Validating Predictions

```
def predict_letter(morse_seq: str):
    x = encode_morse(morse_seq, max_len)
    y = nn.forward(x)
    # argmax over outputs
    k = max(range(len(y)), key=lambda i: y[i])
    return letters[k], y

for ch, seq in morse_dict.items():
    pred, y = predict_letter(seq)
    y_str = [f"{v:.3f}" for v in y]
    print(f"Morse {seq:4s} | target {ch} | predicted {pred} | outputs
{y_str}")
```

```
Morse .-   | target A | predicted A | outputs ['0.972', '0.002',
'0.001', '0.012', '0.018']
Morse -... | target B | predicted B | outputs ['0.016', '0.972',
'0.024', '0.022', '0.000']
Morse -.-. | target C | predicted C | outputs ['0.006', '0.020',
'0.972', '0.000', '0.016']
Morse -..  | target D | predicted D | outputs ['0.012', '0.024',
'0.001', '0.969', '0.013']
Morse .    | target E | predicted E | outputs ['0.024', '0.000',
'0.022', '0.021', '0.973']
```