From Bag-of-Words to Transformers: Building NLP Models from Scratch on a Tiny Vocabulary

Libraries and style

In [3]:
```
from IPython.display import Image, display
from IPython.display import HTML
import random
import math
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
```

In [4]:
```
HTML("""
<style>

/* =========Global notebook styling====== */
body {font-family: "Inter", "Helvetica Neue", Arial, sans-serif;background-c

/* =========H1 — Major Sections========== */
h1 {font-family: "Georgia", "Times New Roman", serif;font-size: 2.2em;font-w

/* =========H2 — Subsections============= */
h2 {font-family: "Georgia", "Times New Roman", serif;font-size: 1.6em;font-w

/* =========H3 — Minor headings=========== */
h3 {font-size: 1.2em;font-weight: 600;color: #4a6572;margin-top: 1.4em;margi

/* =========Paragraphs==================== */
p {font-size: 1.02em;margin-bottom: 0.9em;}

/* =========Inline math & code============ */
code {background-color: #f1f3f4;color: #37474f;padding: 2px 6px;border-radiu

/* =========Math blocks=================== */
.MathJax_Display {background: #f7f9fb;padding: 14px;border-radius: 10px;bord

/* =========Lists========================= */
ul li {margin-bottom: 0.4em;}

</style>
""")
```

Out[4]:

Setup and vocabulary

In [22]:
```
torch.manual_seed(0)
random.seed(0)
np.random.seed(0)


V = ["<BOS>", "<EOS>", "human", "nature", "consume", "preserve", "destroy",
stoi = {t:i for i,t in enumerate(V)}
itos = {i:t for t,i in stoi.items()}
vocab_size = len(V)

def encode(tokens):
    return [stoi[t] for t in tokens]

def decode(ids):
    return [itos[i] for i in ids]

vocab_size, V
```

Out[22]:
```
(10,
 ['<BOS>',
  '<EOS>',
  'human',
  'nature',
  'consume',
  'preserve',
  'destroy',
  'not',
  'sustain',
  'collapse'])
```

Dataset generator

In [23]:
```
def sample_sentence():
    action = random.choice(["consume", "preserve", "destroy"])
    use_not = random.random() < 0.35
    tokens = ["<BOS>", "human"]
    if use_not:
        tokens += ["not", action]
    else:
        tokens += [action]
    tokens += ["nature", "<EOS>"]

    # consume/destroy => collapse
    # preserve => sustain
    # Label: 1 = sustain, 0 = collapse
    if action in ["consume", "destroy"]:
        y = 0
    else:
        y = 1

    if use_not:
        y = 1 - y

    return tokens, y

def make_dataset(n=400):
    X, Y = [], []
    for _ in range(n):
        tok, y = sample_sentence()
        X.append(encode(tok))
        Y.append(y)
    return X, Y

X_train, y_train = make_dataset(800)
X_test,  y_test  = make_dataset(200)

for i in range(8):
    print(decode(X_train[i]), "->", "sustain" if y_train[i]==1 else "collaps
```

```
['<BOS>', 'human', 'preserve', 'nature', '<EOS>'] -> sustain
['<BOS>', 'human', 'not', 'preserve', 'nature', '<EOS>'] -> collapse
['<BOS>', 'human', 'destroy', 'nature', '<EOS>'] -> collapse
['<BOS>', 'human', 'preserve', 'nature', '<EOS>'] -> sustain
['<BOS>', 'human', 'preserve', 'nature', '<EOS>'] -> sustain
['<BOS>', 'human', 'consume', 'nature', '<EOS>'] -> collapse
['<BOS>', 'human', 'not', 'preserve', 'nature', '<EOS>'] -> collapse
['<BOS>', 'human', 'consume', 'nature', '<EOS>'] -> collapse
```

Padding and Batching

This stage prepares variable-length text sequences for efficient training in PyTorch by combining batching and padding. The batches function first groups the dataset into mini-batches of a fixed size, optionally shuffling the data to prevent the model from learning spurious patterns related to ordering. For each mini-batch, the pad_batch function adjusts all sequences to the same length by appending a special padding token (PAD_ID, here chosen as the token) to shorter sequences. This allows the sequences to be stacked into a rectangular tensor, which is required for efficient computation. Simultaneously, a mask is created to indicate which positions correspond to real tokens (1) and which correspond to padding (0), enabling models to ignore padded elements when necessary. The result is a set of tensors containing the padded inputs, their masks, and the corresponding labels, allowing the model to process multiple examples in parallel during training.

Steps performed:

1. Create a list of dataset indices.
2. Optionally shuffle them to avoid learning order-dependent patterns.
3. Split the indices into chunks of size `batch_size`.
4. Select the corresponding input sequences and labels.
5. Apply padding to the sequences in that chunk.

6. Return tensors for inputs, masks, and labels.

Example

Dataset:

```
Sentence 1 → [0, 2, 4, 3, 1]
Sentence 2 → [0, 2, 7, 6, 3, 1]
Sentence 3 → [0, 2, 5, 3, 1]
```

Possible batches: Batch 1 → Sentence 1, Sentence 2 Batch 2 → Sentence 3

Within each batch, sequences may have different lengths. The `pad_batch` function makes them uniform by appending a special token (`PAD_ID`, here `<EOS>`) to shorter sequences.

This enables stacking them into a rectangular tensor of shape **(B, T)**:

- **B** = batch size
- **T** = maximum sequence length in that batch

Padding does not change the original content; it only adds "empty" tokens.

Example

Batch containing three sentences:

```
[0, 2, 4, 3, 1] (length 5)
[0, 2, 7, 6, 3, 1] (length 6)
[0, 2, 5, 3, 1] (length 5)
```

Maximum length = 6 → pad shorter sequences:

```
[0, 2, 4, 3, 1, 1]
[0, 2, 7, 6, 3, 1]
[0, 2, 5, 3, 1, 1]
```

Now they can be stacked into a tensor: (3,6).

Mask Creation

A mask is created alongside the padded sequences to distinguish real tokens from padding.

- **1** → real token
- **0** → padding

Example:

```
Sequence 1 → [1, 1, 1, 1, 1, 0]
Sequence 2 → [1, 1, 1, 1, 1, 1]
Sequence 3 → [1, 1, 1, 1, 1, 0]
```

This mask allows models (especially sequence models) to ignore padded positions during computation.

Final Output of a Batch

Each call to `batches()` yields three tensors:

- **xb** → padded input sequences of shape (B, T)
- **mb** → mask indicating real tokens of shape (B, T)
- **yb** → labels for the batch of shape (B)

These tensors enable efficient parallel processing of variable-length text data during training.

In [24]:
```
def pad_batch(seqs, pad_id):
    max_len = max(len(s) for s in seqs)
    out = []
    mask = []
    for s in seqs:
        padded = s + [pad_id]*(max_len-len(s))
        out.append(padded)
        mask.append([1]*len(s) + [0]*(max_len-len(s)))
    return torch.tensor(out, dtype=torch.long), torch.tensor(mask, dtype=tor

PAD_ID = stoi["<EOS>"]

def batches(X, Y, batch_size=64, shuffle=True):
    idx = list(range(len(X)))
    if shuffle:
        random.shuffle(idx)
    for i in range(0, len(X), batch_size):
        j = idx[i:i+batch_size]
        xb, mb = pad_batch([X[k] for k in j], PAD_ID)
        yb = torch.tensor([Y[k] for k in j], dtype=torch.long)
        yield xb, mb, yb
```

Bag of Words

This cell implements and trains a simple Bag-of-Words (BoW) text classifier using PyTorch. The bow_features function converts a batch of tokenized sequences (x_ids, shape B×T) into fixed-length vectors of size equal to the vocabulary, counting how many times each token appears in each sequence. The BoWClassifier class defines a minimal neural network consisting of a single linear layer that maps these count vectors to two output logits, corresponding to a binary classification task. The train_model function then trains this model using the Adam optimizer and cross-entropy loss over multiple epochs: for each mini-batch produced by the previously defined batching function, it performs a forward pass, computes loss, backpropagates gradients, and updates parameters. After each epoch, the model is evaluated on both training and test sets by computing accuracy in evaluation mode without gradient tracking. Finally, the code instantiates the classifier with the given vocabulary size and trains it for 20 epochs, periodically printing loss and accuracy metrics to monitor learning progress.

Input format: `(B, T)`

Each batch input `x_ids` has shape:

- **B** = number of sentences in the batch (batch size)
- **T** = number of token positions per sentence (after padding)

Example batch (token IDs) with a tiny vocabulary:

- Vocabulary (example mapping):
    - `0="<BOS>"`, `1="<EOS>"`, `2="human"`, `3="nature"`, `4="consume"`, `5="preserve"`, `6="destroy"`, `7="not"`

Example sentences (already padded with `<EOS>` as PAD):

```
x_ids =
[
[0, 2, 4, 3, 1, 1], # <BOS> human consume nature <EOS> <EOS>
[0, 2, 7, 6, 3, 1], # <BOS> human not destroy nature <EOS>
]
```

So here:

- `B = 2`
- `T = 6`

BoW transformation: `(B, T) → (B, V)`

The function `bow_features(x_ids, vocab_size)` creates a matrix `bow` of shape:

- **V** = vocabulary size

For each sentence, it counts how many times each token ID appears.

Example BoW output

If `V = 8`, then each sentence becomes a length-8 vector:

- Sentence 1: `[0,2,4,3,1,1]` contains:
  - `<BOS>` (0): 1 time
  - `<EOS>` (1): 2 times
  - `human` (2): 1 time
  - `nature` (3): 1 time
  - `consume` (4): 1 time

So its BoW vector is:

```
bow[0] = [1, 2, 1, 1, 1, 0, 0, 0]
```

- Sentence 2: `[0,2,7,6,3,1]` contains:
  - `<BOS>` : 1
  - `<EOS>` : 1
  - `human` : 1
  - `nature` : 1
  - `destroy` : 1
  - `not` : 1

So:

```
bow[1] = [1, 1, 1, 1, 0, 0, 1, 1]
```

This means:

Input: BoW vector of size V

Output: 2 logits (one per class)

So the forward pass is:

Convert tokens to BoW counts:

x = bow_features(x_ids, vocab_size) → shape (B, V)

Compute logits:

logits = W x + b → shape (B, 2)

NN

Even though the model is written as a PyTorch `nn.Module`, it is **not deep**. The architecture contains:

- **No hidden layers**
- **No nonlinear activations** (e.g., ReLU, tanh)
- Only a single linear transformation from features to class scores

That makes it equivalent to **multinomial logistic regression** (also called **softmax regression**) with 2 classes.

Linear scoring (logits)

After converting each sentence to a Bag-of-Words vector $(x \in \mathbb{R}^V)$, the linear layer computes the **logits**:

$$z = Wx + b$$

Where:

- $W \in \mathbb{R}^{2\times V}$ is the weight matrix (one row per class)
- $b \in \mathbb{R}^2$ is the bias vector
- $z \in \mathbb{R}^2$ are the logits z_0, z_1

For a batch of size (B), this becomes:

- $X \in \mathbb{R}^{B\times V}$
- $Z = XW^\top + b \in \mathbb{R}^{B\times 2}$

From logits to probabilities (softmax)

The logits are turned into class probabilities via the **softmax** function:

$$p(y = k \mid x) = \frac{e^{z_k}}{\sum_{j=0}^{1} e^{z_j}} \quad \text{for } k \in \{0, 1\}.$$

This produces two probabilities that sum to 1:

$$p(y = 0 \mid x) + p(y = 1 \mid x) = 1.$$

Binary logistic regression equivalence

With two classes, you can also express this as a single sigmoid on the logit difference:

$$p(y = 1 \mid x) = \sigma(z_1 - z_0) = \frac{1}{1 + e^{-(z_1 - z_0)}}.$$

This is why a 2-class softmax classifier is closely related to standard binary logistic regression.

5) Loss function: cross-entropy

During training, the code uses:

```
loss = F.cross_entropy(logits, yb)
```

For one example (x, y), cross-entropy is:

$$\mathcal{L}(x, y) = -\log(p(y \mid x)).$$

For a batch of size (B):

$$\mathcal{L}_{batch} = \frac{1}{B}\sum_{i=1}^{B} -\log(p(y_i \mid x_i)).$$

Minimizing cross-entropy encourages the model to assign high probability to the correct class.

6) Training loop: what happens each epoch

Each epoch repeats the following steps:

1. **Training phase**
    - Iterate through batches from `batches(X_train, y_train, batch_size=64)`
    - For each batch:
        - Compute logits: `logits = model(xb)`
        - Compute loss: `loss = cross_entropy(logits, yb)`
        - Backpropagate gradients: `loss.backward()`
        - Update parameters with Adam: `opt.step()`
2. **Evaluation phase**
    - Switch to evaluation mode: `model.eval()`
    - Compute accuracy on train and test sets using `argmax` on logits:

$$\hat{y} = \arg\max_k z_k$$

Concrete transformation example (end-to-end)

Consider two sentences (already tokenized and padded):

- Sentence A: `<BOS> human consume nature <EOS> <EOS>`
- Sentence B: `<BOS> human not destroy nature <EOS>`

Token IDs (example):

```
A = [0, 2, 4, 3, 1, 1]
B = [0, 2, 7, 6, 3, 1]
```

Step 1 — BoW counts ((B,T) to (B,V))

If (V = 10), each sentence becomes a length-10 count vector. For Sentence A:

- `<BOS>` appears 1 time
- `human` appears 1 time
- `consume` appears 1 time
- `nature` appears 1 time
- `<EOS>` appears 2 times

So (x_A) looks like:

```
x_A = [1, 2, 1, 1, 1, 0, 0, 0, 0, 0]
```

Sentence B becomes:

```
x_B = [1, 1, 1, 1, 0, 0, 1, 1, 0, 0]
```

Step 2 — Linear layer ((B,V) \to (B,2))

For each sentence:

$$z = Wx + b$$

The output (z) has two logits, one per class, e.g.:

```
z_A = [z0, z1]
z_B = [z0, z1]
```

Step 3 — Softmax probabilities

$$p(y = k \mid x) = \frac{e^{z_k}}{e^{z_0} + e^{z_1}}$$

The model predicts the class with the highest logit (or highest probability):

$$\hat{y} = \arg\max_k z_k$$

Interpreting the learned weights

Printing:

```
print(bow_model.lin.weight)  # shape (2, V)
print(bow_model.lin.bias)    # shape (2,)
```

gives a weight for each **token** and each **class**.

- If a token has a **large positive weight** for the "sustain" class, its presence increases the logit (z_{\text{sustain}}).
- If it has a **large negative weight**, it pushes the prediction away from that class.

Because BoW uses counts, repeated tokens contribute linearly: seeing a token twice roughly doubles its contribution to the logits.

In [25]:
```
def bow_features(x_ids, vocab_size):
    # x_ids: (B,T)
    # returns count per token: (B,V)
    B, T = x_ids.shape
    bow = torch.zeros(B, vocab_size)
    for b in range(B):
        for t in range(T):
            bow[b, x_ids[b,t]] += 1.0
    return bow

class BoWClassifier(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        self.lin = nn.Linear(vocab_size, 2)
    def forward(self, x_ids):
        x = bow_features(x_ids, vocab_size)
        return self.lin(x)

def train_model(model, epochs=20, lr=1e-2):
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    for ep in range(1, epochs+1):
        model.train()
        total_loss = 0.0
        for xb, mb, yb in batches(X_train, y_train, batch_size=64):
            logits = model(xb)
            loss = F.cross_entropy(logits, yb)
            opt.zero_grad()
            loss.backward()
            opt.step()
            total_loss += loss.item()

        # eval
        model.eval()
        def acc(X, Y):
            correct = 0
            total = 0
            for xb, mb, yb in batches(X, Y, batch_size=128, shuffle=False):
                with torch.no_grad():
                    pred = model(xb).argmax(dim=1)
                correct += (pred==yb).sum().item()
                total += len(yb)
            return correct/total
        if ep in [1,2,5,10,20] or ep==epochs:
            print(f"ep {ep:02d} | loss {total_loss:.3f} | acc train {acc(X_t

bow_model = BoWClassifier(vocab_size)
train_model(bow_model, epochs=20, lr=1e-2)
```

```
ep 01 | loss 8.619 | acc train 0.537 | acc test 0.585
ep 02 | loss 8.103 | acc train 0.667 | acc test 0.705
ep 05 | loss 7.630 | acc train 0.885 | acc test 0.895
ep 10 | loss 7.693 | acc train 0.885 | acc test 0.895
ep 20 | loss 7.683 | acc train 0.885 | acc test 0.895
```

In [26]:
```
print(bow_model.lin.weight)
print(bow_model.lin.bias)
```

```
Parameter containing:
tensor([[-0.0813,  0.3918, -0.3392, -0.3117,  0.1514, -0.3765,  0.3929, -0.2
086,
         -0.0281,  0.0837],
        [-0.0166, -0.2844, -0.2231, -0.1305, -0.4036,  0.4730, -0.2742,  0.6
491,
         -0.2144, -0.1377]], requires_grad=True)
Parameter containing:
tensor([0.0359, 0.3416], requires_grad=True)
```

The model achieved reasonably good performance, with training and test accuracies approaching high values (e.g., around 0.87–0.88). This indicates that the classifier successfully learned meaningful correlations between individual words and the target labels. In this dataset, certain tokens strongly signal the outcome — for example, words like **"preserve"** tend to indicate *sustain*, while **"consume"** or **"destroy"** tend to indicate *collapse*. Because the Bag-of-Words representation captures word frequencies, the model can learn these associations effectively.

However, the accuracy does not reach perfect performance because the BoW representation fundamentally ignores **word order, syntax, and semantic interactions** between tokens. In particular, it cannot properly model **negation**. The word **"not"** flips the meaning of the action (e.g., “not destroy” should imply sustain), but BoW only counts tokens independently. As a result, the sentences:

```
human destroy nature
human not destroy nature
```

produce very similar feature vectors — the second simply adds one extra count for “not.” The model must infer the effect of negation indirectly from weights, without any notion that “not” modifies the verb that follows.

More generally, BoW lacks mechanisms for:

- **Attention** — it cannot focus on relationships between specific words
- **Compositional semantics** — it cannot combine meanings across tokens

- **Word order awareness** — it treats sentences as unordered bags of tokens
- **Contextual understanding** — it cannot represent that one word changes another's meaning

Modern NLP models such as recurrent networks, convolutional sequence models, and especially Transformers address these limitations by modeling token interactions and context. In particular, attention mechanisms allow the model to learn that "not" is strongly connected to the verb that follows, enabling correct interpretation of negation.

Therefore, while the BoW logistic regression model provides a strong baseline and demonstrates that simple lexical signals carry predictive power, its performance is inherently limited by its inability to capture structure and meaning beyond individual word counts.

Embedding + Feedforward Neural Network (FNN)

This model replaces the Bag-of-Words representation with a **learned embedding layer** followed by a **feedforward neural network (FNN)**. Unlike BoW, which only counts tokens, this architecture preserves token positions and learns dense vector representations that capture semantic similarities.

Word Embedding

An embedding maps each discrete token ID to a continuous vector in $\mathbb{R}^d$.

Formally, an embedding layer is a lookup table:

$$E \in \mathbb{R}^{V \times d}$$

where:

- V = vocabulary size
- d = embedding dimension
- Row E_i is the vector representation of token i

For a token ID t, the embedding output is:

$$\mathbf{e}_t = E[t] \in \mathbb{R}^d$$

Example

If $d = 8$, the word `"destroy"` might map to:

```
destroy → [0.21, -0.44, 0.18, 0.03, 0.91, -0.12, 0.07, 0.30]
```

Semantically related words tend to get similar vectors during training.

Input Processing (Fixed Length)

The model expects sequences of length `max_len`. Therefore, each batch is adjusted as follows:

- If a sequence is longer than `max_len`, it is truncated.
- If shorter, it is padded with the padding token ID.

Formally:

$$x \in \mathbb{N}^{B \times T} \to x' \in \mathbb{N}^{B \times L}$$

where:

- B = batch size
- T = original length
- $L = \text{max\\_len}$

Embedding Lookup

After padding/truncation, the embedding layer converts token IDs into vectors:

$$x' \in \mathbb{N}^{B \times L} \quad \to \quad E(x') \in \mathbb{R}^{B \times L \times d}$$

This tensor contains one d-dimensional vector per token position.

Example

For $L = 6$, $d = 8$:

```
(B, 6) → (B, 6, 8)
```

Each sentence becomes a matrix of embeddings.

Flattening for FNN Input

Feedforward networks expect a single vector per example, so the embeddings are flattened:

$$\mathbb{R}^{B \times L \times d} \to \mathbb{R}^{B \times (L \cdot d)}$$

This operation concatenates all token embeddings into one long vector.

Example

If $L = 6$ and $d = 8$:

$$L \cdot d = 48$$

So each sentence becomes a vector in $\mathbb{R}^{48}$.

Feedforward Neural Network (FNN)

A feedforward neural network applies one or more fully connected layers:

First layer

$$h = \tanh(W_1 x + b_1)$$

where:

- $W_1 \in \mathbb{R}^{H \times (Ld)}$
- H = number of hidden units
- $\tanh(\cdot)$ introduces nonlinearity

Second (output) layer

$$z = W_2 h + b_2$$

where:

- $W_2 \in \mathbb{R}^{2 \times H}$
- $z \in \mathbb{R}^2$ are logits for the two classes

Classification via Softmax

Probabilities are obtained with softmax:

$$p(y = k \mid x) = \frac{e^{z_k}}{\sum_{j=0}^{1} e^{z_j}}$$

Prediction:

$$\hat{y} = \arg\max_k z_k$$

What This Model Learns

Compared to BoW, this architecture can learn:

- Position-sensitive patterns (because embeddings are ordered)
- Nonlinear combinations of features
- Interactions between tokens
- Semantic similarity between words

However, because the embeddings are flattened, the model still does not explicitly model long-range structure or token-to-token relationships beyond fixed positions.

End-to-End Example

Consider the padded sentence:

```
<BOS> human not destroy nature <EOS>
```

Token IDs:

```
[0, 2, 7, 6, 3, 1]
```

Step 1 — Embedding lookup

Each ID becomes a vector:

```
(6) → (6 × 8)
```

Step 2 — Flatten

```
(6 × 8) → (48)
```

Step 3 — Hidden layer

Apply nonlinear transformation:

$$h = \tanh(W_1 x + b_1)$$

Step 4 — Output layer

Compute logits:

$$z = W_2 h + b_2$$

Step 5 — Prediction

Softmax → class probabilities → final label.

In [27]:
```python
class EmbFNN(nn.Module):
    def __init__(self, vocab_size, d=8, hidden=32, max_len=6):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, d)
        self.max_len = max_len
        self.fc1 = nn.Linear(max_len*d, hidden)
        self.fc2 = nn.Linear(hidden, 2)
    def forward(self, x_ids):

        x = x_ids[:, :self.max_len]
        if x.shape[1] < self.max_len:
            pad = torch.full((x.shape[0], self.max_len-x.shape[1]), PAD_ID,
            x = torch.cat([x, pad], dim=1)
        e = self.emb(x)                     # (B,T,d)
        h = e.reshape(e.shape[0], -1)   # (B,T*d)
        h = torch.tanh(self.fc1(h))
        return self.fc2(h)

fnn = EmbFNN(vocab_size, d=8, hidden=32, max_len=6)
train_model(fnn, epochs=25, lr=3e-3)
```

```
ep 01 | loss 7.319 | acc train 0.885 | acc test 0.895
ep 02 | loss 3.543 | acc train 1.000 | acc test 1.000
ep 05 | loss 0.145 | acc train 1.000 | acc test 1.000
ep 10 | loss 0.030 | acc train 1.000 | acc test 1.000
ep 20 | loss 0.009 | acc train 1.000 | acc test 1.000
ep 25 | loss 0.006 | acc train 1.000 | acc test 1.000
```

Compared to the Bag-of-Words linear classifier, the embedding-based model represents a significant conceptual and practical advance. In the BoW approach, each sentence is reduced to token counts, discarding word order and treating all words as independent features; as a result, the model can only learn shallow correlations between the presence of individual tokens and the target class. By contrast, an embedding layer maps each token to a dense continuous vector that is learned during training, allowing the model to capture semantic relationships between words and to represent similar concepts with nearby vectors in a geometric space. When these embeddings are processed by a non-linear neural network, the model can learn interactions between words and their positions, effectively modeling simple compositional rules rather than isolated features. In practice, this means the classifier can begin to distinguish patterns such as how negation modifies meaning or how certain combinations of words imply different outcomes — capabilities that are fundamentally inaccessible to a pure Bag-of-Words model. Thus, embeddings transform the problem from counting symbols to reasoning over structured representations, enabling the network to approximate underlying linguistic rules instead of memorizing surface statistics.

Recurrent Neural Networks (RNN)

A Recurrent Neural Network (RNN) is a neural architecture designed specifically to process sequential data. Unlike standard feed-forward neural networks (FNNs), which treat inputs as independent fixed-size vectors, an RNN processes one element of a sequence at a time while maintaining an internal memory of previous elements. This memory is represented by a hidden state that evolves across time steps.

Formally, given an input sequence $x_1, x_2, \ldots, x_T$, an RNN computes hidden states according to

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$$

where h_{t-1} carries information from the past. This recurrence allows the model to capture dependencies across tokens and model how earlier words influence later ones.

In contrast, a standard neural network with flattened inputs ignores the temporal structure of language. When embeddings are flattened into a single vector, the model sees only a fixed arrangement of features, without an explicit notion of order propagation. An RNN preserves sequence structure by updating its hidden state at each time step, effectively accumulating contextual information as it reads the sentence.

Another important difference is parameter sharing. The same transformation is applied at every time step, enabling the model to process sequences of variable length and generalize patterns across positions. This makes RNNs suitable for language, time series, and other sequential tasks.

In this experiment, the embedded sentence is processed token by token, and the final hidden state is used as a compact representation of the entire sequence for classification. However, basic RNNs may struggle with long-range dependencies due to vanishing or exploding gradients, which motivated more advanced architectures such as LSTM, GRU, and Transformers.

In [28]:
```
class SimpleRNN(nn.Module):
    def __init__(self, vocab_size, d=8, h=24):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, d)
        self.rnn = nn.RNN(input_size=d, hidden_size=h, batch_first=True, nor
        self.fc = nn.Linear(h, 2)
    def forward(self, x_ids):
        e = self.emb(x_ids)          # (B,T,d)
        out, hn = self.rnn(e)        # (B,T,h), hn (1,B,h)
        last = hn[0]                 # (B,h)
        return self.fc(last)

rnn = SimpleRNN(vocab_size, d=8, h=24)
train_model(rnn, epochs=25, lr=3e-3)
```

```
ep 01 | loss 8.419 | acc train 0.782 | acc test 0.810
ep 02 | loss 5.079 | acc train 1.000 | acc test 1.000
ep 05 | loss 0.067 | acc train 1.000 | acc test 1.000
ep 10 | loss 0.023 | acc train 1.000 | acc test 1.000
ep 20 | loss 0.009 | acc train 1.000 | acc test 1.000
ep 25 | loss 0.006 | acc train 1.000 | acc test 1.000
```

The RNN achieves perfect training and test accuracy after a few epochs, indicating that it successfully learns the underlying patterns in the dataset. Compared to the embedding-based feed-forward network (EmbFNN), the RNN benefits from explicitly modeling the sequential structure of the sentence rather than relying on a flattened representation. While the EmbFNN can capture interactions between word embeddings and their fixed positions, it still treats the input as a static vector. The RNN, on the other hand, processes tokens step by step and accumulates contextual information in its hidden state, allowing it to represent how earlier words influence later ones. In this simple task, both models eventually reach perfect performance because the classification rule is relatively easy to learn. However, the RNN does so in a way that is more consistent with the compositional nature of language, making it better suited for generalizing to longer or more complex sequences. Thus, the results illustrate that incorporating sequential memory can improve representational power even when overall accuracy appears similar.

Mini Transformer (Self-Attention Model)

The Transformer architecture represents a major shift from recurrent models by replacing sequential processing with self-attention. Instead of reading tokens one by one, the model processes the entire sequence simultaneously and computes pairwise interactions between all tokens. Each input token is first converted into a vector through an embedding layer and augmented with positional embeddings, allowing the model to retain information about word order despite operating in parallel.

Self-attention is implemented through three learned projections of the input: queries Q , keys K, and values V. Attention scores are computed as

$$A = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)$$

which produces a matrix of weights describing how strongly each token attends to every other token. These weights are then used to aggregate information from the value vectors, yielding context-aware representations in which each position incorporates information from the entire sentence.

The model includes residual connections and layer normalization, which stabilize training and allow deeper representations to be learned. A position-wise feed-forward network further transforms each token representation independently, increasing expressive power. For classification, the representation of the first token is used as a pooled summary of the sequence.

Compared to feed-forward networks and RNNs, the Transformer can model long-range dependencies more effectively because information does not need to propagate step by step. Every token can directly interact with every other token in a single layer. This makes attention-based models highly expressive and scalable, forming the foundation of modern large language models.

In [29]:
```python
class MiniTransformer(nn.Module):
    def __init__(self, vocab_size, d=16, max_len=6):
        super().__init__()
        self.d = d
        self.max_len = max_len
        self.emb = nn.Embedding(vocab_size, d)
        self.pos = nn.Embedding(max_len, d)

        # Wq, Wk, Wv
        self.Wq = nn.Linear(d, d, bias=False)
        self.Wk = nn.Linear(d, d, bias=False)
        self.Wv = nn.Linear(d, d, bias=False)
        self.out = nn.Linear(d, d, bias=False)

        self.ffn = nn.Sequential(
            nn.Linear(d, 4*d),
            nn.ReLU(),
            nn.Linear(4*d, d)
        )
        self.norm1 = nn.LayerNorm(d)
        self.norm2 = nn.LayerNorm(d)
        self.cls = nn.Linear(d, 2)

        self.last_attn = None

    def forward(self, x_ids, mask=None):
        x = x_ids[:, :self.max_len]
        if x.shape[1] < self.max_len:
            pad = torch.full((x.shape[0], self.max_len-x.shape[1]), PAD_ID,
            x = torch.cat([x, pad], dim=1)
        B, T = x.shape

        pos_ids = torch.arange(T).unsqueeze(0).repeat(B,1)
        X = self.emb(x) + self.pos(pos_ids)  # (B,T,d)

        Q = self.Wq(X)
        K = self.Wk(X)
        Vv = self.Wv(X)

        scores = (Q @ K.transpose(1,2)) / math.sqrt(self.d)  # (B,T,T)


        if mask is not None:
            m = mask[:, :T]
            if m.shape[1] < T:
                m = torch.cat([m, torch.zeros(B, T-m.shape[1])], dim=1)
            # m: (B,T) -> (B,1,T)
            scores = scores.masked_fill((m.unsqueeze(1) == 0), -1e9)

        A = torch.softmax(scores, dim=-1)   # (B,T,T)
        self.last_attn = A.detach().cpu()

        Z = A @ Vv                           # (B,T,d)
        Z = self.out(Z)

        X2 = self.norm1(X + Z)
        Ff = self.ffn(X2)
        X3 = self.norm2(X2 + Ff)
        pooled = X3[:,0,:]
        return self.cls(pooled)

tr = MiniTransformer(vocab_size, d=16, max_len=6)

def train_transformer(model, epochs=30, lr=3e-3):
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    for ep in range(1, epochs+1):
        model.train()
        total_loss = 0.0
        for xb, mb, yb in batches(X_train, y_train, batch_size=64):
            logits = model(xb, mask=mb)
            loss = F.cross_entropy(logits, yb)
            opt.zero_grad()
            loss.backward()
            opt.step()
            total_loss += loss.item()

        def acc(X, Y):
            model.eval()
            correct = 0
            total = 0
            for xb, mb, yb in batches(X, Y, batch_size=128, shuffle=False):
                with torch.no_grad():
                    pred = model(xb, mask=mb).argmax(dim=1)
                correct += (pred==yb).sum().item()
                total += len(yb)
            return correct/total

        if ep in [1,2,5,10,20,30] or ep==epochs:
            print(f"ep {ep:02d} | loss {total_loss:.3f} | acc train {acc(X_t

train_transformer(tr, epochs=30, lr=3e-3)
```

```
ep 01 | loss 8.763 | acc train 0.885 | acc test 0.895
ep 02 | loss 5.716 | acc train 0.885 | acc test 0.895
ep 05 | loss 3.104 | acc train 0.885 | acc test 0.895
ep 10 | loss 0.042 | acc train 1.000 | acc test 1.000
ep 20 | loss 0.013 | acc train 1.000 | acc test 1.000
ep 30 | loss 0.006 | acc train 1.000 | acc test 1.000
```