Step 1 — Install Dependencies

```
!pip install sacrebleu -q
print("Dependencies installed")
```

```
Dependencies installed
```

```
from google.colab import files
uploaded = files.upload()
DATA_PATH = list(uploaded.keys())[0]
print(f"Uploaded: {DATA_PATH}")
with open(DATA_PATH) as f:
    n = sum(1 for _ in f)
print(f"Total lines: {n:,}")
```

Browse... spa.txt

spa.txt(text/plain) - 8042772 bytes, last modified: n/a - 100% done

```
Saving spa.txt to spa (2).txt
Uploaded: spa (2).txt
Total lines: 118,964
```

```
# Alternative: load from Google Drive (comment out upload cell above)
# from google.colab import drive
# drive.mount('/content/drive')
# DATA_PATH = '/content/drive/MyDrive/spa.txt'
```

Step 3 — Hyperparameters

```
MAX_SAMPLES = 10_000
MIN_FREQ    = 2
MAX_LEN     = 50
EMBED_DIM   = 256
HIDDEN_DIM  = 512
NUM_LAYERS  = 2
DROPOUT     = 0.3
BATCH_SIZE  = 64
EPOCHS      = 20
LR          = 1e-3
CLIP        = 1.0
SEED        = 42

import random, math, time, os, re
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
from collections import Counter
from typing import List, Tuple, Dict, Optional

# random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED)
# DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# print(f"Device : {DEVICE}")
# if DEVICE.type == "cuda":
#     print(f"GPU    : {torch.cuda.get_device_name(0)}")
```

Step 4 — Data Preprocessing

```
PAD_IDX, SOS_IDX, EOS_IDX, UNK_IDX = 0, 1, 2, 3

def normalize_string(s):
    """Lowercase, add spaces around punctuation, keep Spanish chars."""
    s = s.lower().strip()
    s = re.sub(r'([.!?\u00bf\u00a1])', r' \1', s)
    s = re.sub(r'[^a-zA-Z.!?\u00bf\u00a1\u00e1\u00e9\u00ed\u00f3\u00fa\u00fc\u00f1\']+',
    return s.strip()

def read_pairs(path, max_samples=10_000, seed=42):
    pairs = []
    with open(path, encoding='utf-8') as f:
        for line in f:
            parts = line.strip().split('\t')
            if len(parts) >= 2:
                s = normalize_string(parts[0])
                t = normalize_string(parts[1])
                if s and t:
                    pairs.append((s, t))
    random.seed(seed); random.shuffle(pairs)
    return pairs[:max_samples]

def split_pairs(pairs, train_r=0.8, val_r=0.1):
    n = len(pairs)
    nt = int(train_r * n); nv = int(val_r * n)
    return pairs[:nt], pairs[nt:nt+nv], pairs[nt+nv:]

class Vocabulary:
    """Bidirectional word <-> index mapping with 4 special tokens."""
    def __init__(self, name):
        self.name = name
        self.word2idx = {"<pad>": 0, "<sos>": 1, "<eos>": 2, "<unk>": 3}
        self.idx2word = {v: k for k, v in self.word2idx.items()}
        self.freq = Counter()

    def build(self, sentences, min_freq=2):
        for s in sentences:
            self.freq.update(s.split())
        for w, c in sorted(self.freq.items()):
            if c >= min_freq and w not in self.word2idx:
                i = len(self.word2idx)
                self.word2idx[w] = i
                self.idx2word[i] = w

    def encode(self, s):
        return [self.word2idx.get(w, UNK_IDX) for w in s.split()]

    def decode(self, ids, skip=True):
        sp = {0, 1, 2} if skip else set()
        return ' '.join(self.idx2word.get(i, '<unk>') for i in ids if i not in sp)

    def __len__(self):
        return len(self.word2idx)

class TranslationDataset(Dataset):
    def __init__(self, pairs, sv, tv, max_len=50):
        self.data = []
        for src, tgt in pairs:
            s = [SOS_IDX] + sv.encode(src) + [EOS_IDX]
            t = [SOS_IDX] + tv.encode(tgt) + [EOS_IDX]
```

```
            if len(s) <= max_len and len(t) <= max_len:
                self.data.append((s, t))
    def __len__(self): return len(self.data)
    def __getitem__(self, i): return self.data[i]

def collate_fn(batch):
    ss, ts = zip(*batch)
    sl = torch.tensor([len(s) for s in ss])
    tl = torch.tensor([len(t) for t in ts])
    sp = torch.zeros(len(ss), sl.max(), dtype=torch.long)
    tp = torch.zeros(len(ts), tl.max(), dtype=torch.long)
    for i, (s, t) in enumerate(zip(ss, ts)):
        sp[i, :len(s)] = torch.tensor(s)
        tp[i, :len(t)] = torch.tensor(t)
    sl, order = sl.sort(descending=True)
    return sp[order], tp[order], sl, tl[order]

pairs = read_pairs(DATA_PATH, MAX_SAMPLES)
train_pairs, val_pairs, test_pairs = split_pairs(pairs)
print(f"Train: {len(train_pairs):,} | Val: {len(val_pairs):,} | Test: {len(test_pairs):,

src_vocab = Vocabulary("english"); tgt_vocab = Vocabulary("spanish")
src_vocab.build([p[0] for p in train_pairs], MIN_FREQ)
tgt_vocab.build([p[1] for p in train_pairs], MIN_FREQ)
print(f"Src vocab: {len(src_vocab):,} | Tgt vocab: {len(tgt_vocab):,}")

train_ds = TranslationDataset(train_pairs, src_vocab, tgt_vocab, MAX_LEN)
val_ds   = TranslationDataset(val_pairs,   src_vocab, tgt_vocab, MAX_LEN)
test_ds  = TranslationDataset(test_pairs,  src_vocab, tgt_vocab, MAX_LEN)

train_loader = DataLoader(train_ds, BATCH_SIZE, shuffle=True,  collate_fn=collate_fn)
val_loader   = DataLoader(val_ds,   BATCH_SIZE, shuffle=False, collate_fn=collate_fn)
test_loader  = DataLoader(test_ds,  BATCH_SIZE, shuffle=False, collate_fn=collate_fn)
print(f"Batches => Train:{len(train_loader)} | Val:{len(val_loader)} | Test:{len(test_lo
```

```
Train: 8,000 | Val: 1,000 | Test: 1,000
Src vocab: 2,305 | Tgt vocab: 3,036
Batches => Train:125 | Val:16 | Test:16
```

```
class Encoder(nn.Module):
    """
    Embeds + encodes source sequence.
    Uses pack_padded_sequence to skip padding during LSTM forward pass.
    Returns all hidden states (unused in vanilla model) + final h, c.
    """
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=PAD_IDX)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers,
                            batch_first=True,
                            dropout=dropout if num_layers > 1 else 0.0)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_lens):
        emb    = self.dropout(self.embedding(src))
        packed = pack_padded_sequence(emb, src_lens.cpu(),
                                      batch_first=True, enforce_sorted=True)
        out_p, (h, c) = self.lstm(packed)
        outputs, _    = pad_packed_sequence(out_p, batch_first=True)
        return outputs, h, c


class Decoder(nn.Module):
    """
    Single-step decoder (called once per output token).
    State: hidden + cell (passed in from previous step / encoder init).
    """
```

```
        """
        def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, dropout):
            super().__init__()
            self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=PAD_IDX)
            self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers,
                                batch_first=True,
                                dropout=dropout if num_layers > 1 else 0.0)
            self.fc_out = nn.Linear(hidden_dim, vocab_size)
            self.dropout = nn.Dropout(dropout)

        def forward(self, token, hidden, cell):
            emb = self.dropout(self.embedding(token.unsqueeze(1)))
            out, (hidden, cell) = self.lstm(emb, (hidden, cell))
            logits = self.fc_out(out.squeeze(1))
            return logits, hidden, cell


    class Seq2Seq(nn.Module):
        """
        Vanilla encoder-decoder.

        Teacher forcing ratio: probability of using ground-truth token
        as next decoder input (vs. model's own prediction).
        Decays from 1.0 to 0.5 during training to improve robustness.
        """
        def __init__(self, encoder, decoder):
            super().__init__()
            self.encoder = encoder
            self.decoder = decoder

        def forward(self, src, src_lens, tgt, tf_ratio=0.5):
            B, T = tgt.size()
            V    = self.decoder.fc_out.out_features
            outs = torch.zeros(B, T, V, device=src.device)

            _, h, c = self.encoder(src, src_lens)
            token   = tgt[:, 0]

            for t in range(1, T):
                logits, h, c = self.decoder(token, h, c)
                outs[:, t]   = logits
                teacher       = random.random() < tf_ratio
                token         = tgt[:, t] if teacher else logits.argmax(-1)
            return outs


    encoder = Encoder(len(src_vocab), EMBED_DIM, HIDDEN_DIM, NUM_LAYERS, DROPOUT)
    decoder = Decoder(len(tgt_vocab), EMBED_DIM, HIDDEN_DIM, NUM_LAYERS, DROPOUT)
    model   = Seq2Seq(encoder, decoder).to(DEVICE)

    total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    enc_params   = sum(p.numel() for p in encoder.parameters() if p.requires_grad)
    dec_params   = sum(p.numel() for p in decoder.parameters() if p.requires_grad)
    print(f"Encoder params : {enc_params:,}")
    print(f"Decoder params : {dec_params:,}")
    print(f"Total params   : {total_params:,}")
```

```
Encoder params : 4,268,288
Decoder params : 6,012,892
Total params   : 10,281,180
```

Step 6 — Train the Model

```
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=LR)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, patience=3, factor=0.5)

def train_epoch(model, loader, tf_ratio):
    model.train(); total = 0.0
    for src, tgt, sl, _ in loader:
        src, tgt, sl = src.to(DEVICE), tgt.to(DEVICE), sl.to(DE
        optimizer.zero_grad()
        out  = model(src, sl, tgt, tf_ratio)
        loss = criterion(out[:, 1:].reshape(-1, out.size(-1)),
                         tgt[:, 1:].reshape(-1))
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), CLIP)
        optimizer.step()
        total += loss.item()
    return total / len(loader)

@torch.no_grad()
def evaluate(model, loader):
    model.eval(); total = 0.0
    for src, tgt, sl, _ in loader:
        src, tgt, sl = src.to(DEVICE), tgt.to(DEVICE), sl.to(DE
        out  = model(src, sl, tgt, 0.0)
        loss = criterion(out[:, 1:].reshape(-1, out.size(-1)),
                         tgt[:, 1:].reshape(-1))
        total += loss.item()
    return total / len(loader)

train_losses, val_losses = [], []
best_val, best_state = float('inf'), None

print(f"{'Epoch':>6} | {'Train Loss':>10} | {'Val Loss':>9} | 
print("-" * 58)

for epoch in range(1, EPOCHS + 1):
    tf  = max(0.5, 1.0 - (epoch - 1) * (0.5 / EPOCHS))
    t0  = time.time()
    tl  = train_epoch(model, train_loader, tf)
    vl  = evaluate(model, val_loader)
    scheduler.step(vl)
    train_losses.append(tl); val_losses.append(vl)

    mark = " <- best" if vl < best_val else ""
    if vl < best_val:
        best_val   = vl
        best_state = {k: v.cpu().clone() for k, v in model.stat

    print(f"{epoch:>6} | {tl:>10.4f} | {vl:>9.4f} | {math.exp(v
          f"{tf:>5.2f} | {time.time()-t0:>5.1f}s{mark}")

model.load_state_dict({k: v.to(DEVICE) for k, v in best_state.i
print(f"\nBest model restored  (val_loss={best_val:.4f}, PPL={m
```

```
 Epoch | Train Loss |  Val Loss |  Val PPL |    TF |   Time
----------------------------------------------------------
     1 |     4.9024 |    5.1458 |   171.71 |  1.00 | 116.7s <- best
```

Step 7 — Plot Training Curves

```
fig, axes = plt.subplots(1, 2, figsize=(14, 4))
eps = range(1, EPOCHS + 1)

axes[0].plot(eps, train_losses, 'b-o', ms=3, lw=2, label='Train Loss')
axes[0].plot(eps, val_losses,   'r--s', ms=3, lw=2, label='Val Loss')
axes[0].set_title('Loss per Epoch'); axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Cross-Entropy Loss'); axes[0].legend(); axes[0].grid(True, alpha=.3)

axes[1].plot(eps, [math.exp(v) for v in val_losses], 'g-^', ms=3, lw=2)
axes[1].set_title('Validation Perplexity'); axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('PPL'); axes[1].grid(True, alpha=.3)

plt.suptitle('Vanilla LSTM Seq2Seq — Training History', fontsize=13, fontweight='bold')
plt.tight_layout()
plt.savefig('vanilla_curves.png', dpi=150, bbox_inches='tight')
plt.show()
print("Saved: vanilla_curves.png")
```

Step 8 — BLEU Score on Test Set

```
@torch.no_grad()
def translate(model, sentence, max_len=50):
    model.eval()
    toks   = [SOS_IDX] + src_vocab.encode(normalize_string(sentence)) + [EOS_IDX]
    src_t  = torch.tensor(toks).unsqueeze(0).to(DEVICE)
    sl     = torch.tensor([len(toks)])
    _, h, c = model.encoder(src_t, sl)
    token  = torch.tensor([SOS_IDX], device=DEVICE)
    out    = []
    for _ in range(max_len):
        logits, h, c = model.decoder(token, h, c)
        nt = logits.argmax(-1)
        if nt.item() == EOS_IDX: break
        out.append(nt.item()); token = nt
    return tgt_vocab.decode(out)

def _ngrams(toks, n):
    return Counter(tuple(toks[i:i+n]) for i in range(len(toks)-n+1))

def bleu_simple(hyp, ref):
    h, r = hyp.split(), ref.split()
    if not h: return 0.0
    cc = tc = 0
    for n in range(1, 5):
        hn, rn = _ngrams(h,n), _ngrams(r,n)
        cc += sum(min(c, rn.get(g,0)) for g,c in hn.items())
```

```
            tc += max(0, len(h)-n+1)
        if tc == 0: return 0.0
        return min(1.0, math.exp(1-len(r)/len(h))) * cc/tc

    hyps, refs = [], []
    for src, ref in test_pairs[:500]:
        hyps.append(translate(model, src)); refs.append(ref)

    try:
        import sacrebleu
        bleu = sacrebleu.corpus_bleu(hyps, [refs]).score
        print(f"BLEU (sacrebleu) : {bleu:.2f}")
    except ImportError:
        bleu = 100 * sum(bleu_simple(h,r) for h,r in zip(hyps,refs)) / len(hyps)
        print(f"BLEU (simple)    : {bleu:.2f}")

    test_loss = evaluate(model, test_loader)
    print(f"Test Loss        : {test_loss:.4f}")
    print(f"Test PPL         : {math.exp(test_loss):.2f}")
```

Step 9 — Sample Translations

```
print("=" * 60)
for src, ref in test_pairs[:10]:
    hyp = translate(model, src)
    print(f"SRC : {src}")
    print(f"REF : {ref}")
    print(f"HYP : {hyp}")
    print()
```

Step 10 — Translate Your Own Sentences

```
my_sentences = [
    "How are you?",
    "I love you.",
    "What is your name?",
    "Good morning.",
    "Where is the library?",
]
print("Custom translations")
print("-" * 45)
for s in my_sentences:
    print(f"  EN: {s}")
    print(f"  ES: {translate(model, s)}")
    print()
```

Step 11 — Save & Download Model

```
torch.save({
    'model_state': model.state_dict(),
    'src_vocab': src_vocab.word2idx,
    'tgt_vocab': tgt_vocab.word2idx,
    'bleu': bleu,
    'train_losses': train_losses,
    'val_losses': val_losses,
    'hparams': {'EMBED_DIM':EMBED_DIM,'HIDDEN_DIM':HIDDEN_DIM,
                'NUM_LAYERS':NUM_LAYERS,'DROPOUT':DROPOUT},
```

```
}, 'vanilla_seq2seq.pt')

from google.colab import files
files.download('vanilla_seq2seq.pt')
files.download('vanilla_curves.png')
print("Downloaded vanilla_seq2seq.pt and vanilla_curves.png")
```