Sequence models: RNN, GRU, LSTM, attention and a small transformer

Forecast a signal with RNN, GRU and LSTM, train a character-level language model and sample from it, play with embeddings, build attention from scratch and check it against PyTorch's fused kernel, train a tiny transformer that needs attention to succeed, compare greedy and beam search, and compute BLEU.

Open in Google Colab · part of the inPace course library. Slides © DeepLearning.AI (CC BY-SA 2.0); this notebook: inPace (CC BY-SA 4.0).

Run it top to bottom (Runtime → Run all in Colab). Each part says what to look for.

```
# Works in Google Colab as it is (PyTorch is already installed there).
# On your own computer:  pip install torch torchvision scikit-learn scipy
matplotlib pandas
# One framework per notebook: PyTorch and TensorFlow can clash when loaded into
the same Python process.
import numpy as np, matplotlib.pyplot as plt, torch
from torch import nn
torch.manual_seed(0); np.random.seed(0)
device = "cuda" if torch.cuda.is_available() else "mps" if
torch.backends.mps.is_available() else "cpu"
print("PyTorch", torch.__version__, "| device:", device)
```

1. RNN vs GRU vs LSTM on a sequence

Predict the next value of a noisy signal from the last 40 values.

```
t = np.arange(0, 400, 0.1)
signal = np.sin(t) + 0.5 * np.sin(3.1 * t) + 0.1 *
np.random.default_rng(0).normal(size=t.size)
W = 40
Xs = np.stack([signal[i:i + W] for i in range(len(signal) - W)])[:, :, None]
ys = signal[W:]
Xs, ys = torch.tensor(Xs, dtype=torch.float32), torch.tensor(ys,
dtype=torch.float32)
n = int(0.8 * len(Xs))
class Forecaster(nn.Module):
    def __init__(self, cell):
        super().__init__(); self.rnn = cell(1, 32, batch_first=True); self.head =
nn.Linear(32, 1)
    def forward(self, x): return self.head(self.rnn(x)[0][:, -1]).squeeze(1)    #
use the last time step
for name, cell in (("RNN", nn.RNN), ("GRU", nn.GRU), ("LSTM", nn.LSTM)):
    torch.manual_seed(0); m = Forecaster(cell); opt =
torch.optim.Adam(m.parameters(), lr=3e-3); losses = []
    for epoch in range(3):
        for i in torch.randperm(n).split(64):
            opt.zero_grad(); loss = nn.functional.mse_loss(m(Xs[i]), ys[i]);
loss.backward(); opt.step(); losses.append(loss.item())
    with torch.no_grad(): test = nn.functional.mse_loss(m(Xs[n:]), ys[n:]).item()
```

```
    plt.plot(np.convolve(losses, np.ones(20) / 20, "valid"), label=f"{name}: test
MSE {test:.4f}")
plt.yscale("log"); plt.legend(); plt.title("training loss"); plt.xlabel("mini-
batch"); plt.show()
```

2. A character-level language model

Trained on a short text, it learns to predict the next character. Then we **sample** from it, one character at a time. Temperature < 1 makes it safer, > 1 more surprising.

```
text = ("slow and steady wins the race. the turtle kept walking while the hare
slept. "
        "every day a little progress, every lesson one stone on the trail. ") * 30
chars = sorted(set(text)); stoi = {c: i for i, c in enumerate(chars)}
data = torch.tensor([stoi[c] for c in text])
class CharLM(nn.Module):
    def __init__(self, v, e=32, h=128):
        super().__init__(); self.emb = nn.Embedding(v, e); self.lstm = nn.LSTM(e,
h, batch_first=True); self.out = nn.Linear(h, v)
    def forward(self, x, state=None):
        y, state = self.lstm(self.emb(x), state); return self.out(y), state
torch.manual_seed(0); lm = CharLM(len(chars)); opt =
torch.optim.AdamW(lm.parameters(), lr=3e-3)
for step in range(600):
    i = torch.randint(0, len(data) - 65, (32,))
    x = torch.stack([data[j:j + 64] for j in i]); y = torch.stack([data[j + 1:j +
65] for j in i])
    logits, _ = lm(x); loss = nn.functional.cross_entropy(logits.reshape(-1,
len(chars)), y.reshape(-1))
    opt.zero_grad(); loss.backward(); opt.step()
print("final loss", round(loss.item(), 3))
def sample(start="the ", n=120, temperature=0.8):
    lm.eval(); out = start; x = torch.tensor([[stoi[c] for c in start]]); state =
None
    with torch.no_grad():
        for _ in range(n):
            logits, state = lm(x, state)
            c = torch.multinomial(torch.softmax(logits[0, -1] / temperature, -1),
1).item()
            out += chars[c]; x = torch.tensor([[c]])
    return out
for temp in (0.5, 1.0, 1.5): print(f"T={temp}: {sample(temperature=temp)!r}")
```

3. Embeddings and cosine similarity

`nn.Embedding` is just a lookup table (the slides' embedding matrix E, one row per word). A tiny hand-made example shows the famous analogy arithmetic.

```
words = ["man", "woman", "king", "queen", "apple", "orange"]
#            gender royal  food
vec = torch.tensor([[-1.0, 0.0, 0.0], [1.0, 0.0, 0.0], [-0.95, 0.9, 0.0], [0.97,
0.95, 0.0], [0.0, 0.0, 1.0], [0.05, 0.0, 0.95]])
emb = nn.Embedding.from_pretrained(vec)
e = lambda w: emb(torch.tensor(words.index(w)))
target = e("king") - e("man") + e("woman")
sims = {w: nn.functional.cosine_similarity(target, e(w), dim=0).item() for w in
```

```
words if w not in ("king", "man", "woman")}
print("king - man + woman is closest to:", max(sims, key=sims.get), {k: round(v,
2) for k, v in sims.items()})
```

4. Attention from scratch, and PyTorch's fused version

`Attention(Q, K, V) = softmax(QKᵀ / √d) V`. PyTorch's `scaled_dot_product_attention` does the same in one fast call (with FlashAttention-style kernels on GPUs). A **causal mask** stops a word from looking at later words (what GPT-style models use).

```
def attention(Q, K, V, causal=False):
    scores = Q @ K.transpose(-2, -1) / Q.shape[-1] ** 0.5
    if causal:
        L = scores.shape[-1]
        scores = scores.masked_fill(torch.triu(torch.ones(L, L, dtype=torch.bool),
1), float("-inf"))
    return torch.softmax(scores, -1) @ V
Q, K, V = torch.randn(2, 5, 16), torch.randn(2, 5, 16), torch.randn(2, 5, 16)
F = nn.functional
print("matches PyTorch:", torch.allclose(attention(Q, K, V),
F.scaled_dot_product_attention(Q, K, V), atol=1e-5))
print("causal matches:", torch.allclose(attention(Q, K, V, True),
F.scaled_dot_product_attention(Q, K, V, is_causal=True), atol=1e-5))
mha = nn.MultiheadAttention(embed_dim=16, num_heads=4, batch_first=True)
out, w = mha(Q, K, V)
print("multi-head output", tuple(out.shape), "| attention weights",
tuple(w.shape))
```

5. A tiny transformer that needs attention

Task: for every position, output **how many times its token appears in the whole sequence**. A model that looks at one position at a time can't do it; attention can, because every position can look at every other.

```
V_, L_ = 6, 12
def batch(n):
    x = torch.randint(0, V_, (n, L_))
    y = (x[:, :, None] == x[:, None, :]).sum(-1) - 1          # count of the same
token, 1..L -> 0..L-1
    return x, y
class TinyTransformer(nn.Module):
    def __init__(self, d=64):
        super().__init__()
        self.tok = nn.Embedding(V_, d); self.pos = nn.Embedding(L_, d)
        layer = nn.TransformerEncoderLayer(d, nhead=4, dim_feedforward=128,
batch_first=True, dropout=0.0)
        self.enc = nn.TransformerEncoder(layer, num_layers=2); self.out =
nn.Linear(d, L_)
    def forward(self, x):
        return self.out(self.enc(self.tok(x) +
self.pos(torch.arange(x.shape[1]))))
class NoAttention(nn.Module):                                  # same size, but
each position alone
    def __init__(self, d=64):
```

```
        super().__init__(); self.net = nn.Sequential(nn.Embedding(V_, d),
nn.Linear(d, 128), nn.ReLU(), nn.Linear(128, L_))
    def forward(self, x): return self.net(x)
for name, m in (("no attention", NoAttention()), ("transformer",
TinyTransformer())):
    torch.manual_seed(0); opt = torch.optim.AdamW(m.parameters(), lr=1e-3)
    for step in range(1500):
        x, y = batch(64); loss = F.cross_entropy(m(x).reshape(-1, L_),
y.reshape(-1))
        opt.zero_grad(); loss.backward(); opt.step()
    x, y = batch(1000)
    with torch.no_grad(): print(f"{name:<13} accuracy {(m(x).argmax(-1) ==
y).float().mean():.3f}")
```

7. Greedy search vs beam search

A toy 'translation model' as a table of next-word probabilities. Greedy takes the best word each time; beam search keeps the B best partial sentences and finds a more likely whole sentence.

```
import math
P = {"<s>": {"jane": 0.5, "in": 0.4, "the": 0.1}, "jane": {"is": 0.5, "visits":
0.5},
     "in": {"september": 1.0}, "is": {"going": 0.6, "visiting": 0.4}, "visits":
{"africa": 0.9, "</s>": 0.1},
     "going": {"to": 0.55, "</s>": 0.45}, "to": {"be": 0.5, "</s>": 0.5}, "be": {"
</s>": 1.0}, "visiting": {"africa": 0.9, "</s>": 0.1},
     "africa": {"</s>": 1.0}, "september": {"</s>": 1.0}, "the": {"</s>": 1.0}}
def greedy():
    s, lp = ["<s>"], 0.0
    while s[-1] != "</s>":
        w, p = max(P[s[-1]].items(), key=lambda kv: kv[1]); s.append(w); lp +=
math.log(p)
    return s[1:-1], lp
def beam(B=3):
    beams = [(["<s>"], 0.0)]
    while any(b[0][-1] != "</s>" for b in beams):
        cand = []
        for s, lp in beams:
            if s[-1] == "</s>": cand.append((s, lp)); continue
            cand += [(s + [w], lp + math.log(p)) for w, p in P[s[-1]].items()]
        beams = sorted(cand, key=lambda b: b[1] / len(b[0]) ** 0.7, reverse=True)
[:B]   # length-normalized
    return beams[0][0][1:-1], beams[0][1]
for name, (s, lp) in (("greedy", greedy()), ("beam B=3", beam())):
    print(f"{name:<9} {' '.join(s):<28} probability {math.exp(lp):.3f}")
```

8. BLEU score

Modified n-gram precision, combined over 1- to 4-grams, with a brevity penalty for short outputs.

```
from collections import Counter
def bleu(candidate, references, N=4):
    c = candidate.split(); refs = [r.split() for r in references]; logs = []
    for n in range(1, N + 1):
        grams = Counter(tuple(c[i:i + n]) for i in range(len(c) - n + 1))
```

```
        clip = sum(min(cnt, max(Counter(tuple(r[i:i + n]) for i in range(len(r) -
n + 1))[g] for r in refs)) for g, cnt in grams.items())
        logs.append(math.log(max(clip, 1e-9) / max(sum(grams.values()), 1)))
    ref_len = min((abs(len(r) - len(c)), len(r)) for r in refs)[1]
    bp = 1.0 if len(c) > ref_len else math.exp(1 - ref_len / max(len(c), 1))
    return bp * math.exp(sum(logs) / N)
refs = ["the cat is on the mat", "there is a cat on the mat"]
for cand in ("the cat is on the mat", "the cat on the mat", "the the the the the
the"):
    print(f"{cand:<26} BLEU {bleu(cand, refs, N=2):.3f}")
```

9. Using a pre-trained transformer (Hugging Face)

Today you rarely train language models from zero: you start from a pre-trained one. Runs in Colab (downloads a small model).

```
try:
    from transformers import pipeline
    clf = pipeline("sentiment-analysis", model="distilbert/distilbert-base-
uncased-finetuned-sst-2-english")
    print(clf(["The dessert is excellent.", "Completely lacking in good taste,
good service and good ambience."]))
except Exception as e:
    print("Run this in Colab (needs `transformers` and internet):",
type(e).__name__)
```

Try this

1. In section 1, use 200 time steps instead of 40. Which cell suffers most?
2. In section 2, sample with temperature 0.1 and 3.0. Describe what happens.
3. In section 5, give the transformer 1 layer and 1 head. Does it still solve the task?
4. In section 7, try beam width 1 and 5. Which equals greedy search?