

```
In [1]: import math
import re
from random import randrange, randint, shuffle
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from tqdm.auto import tqdm
```

d:\NLP\A4_AIT\.venv\Lib\site-packages\tqdm\auto.py:21: TqdmWarning: IPProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html

```
from .autonotebook import tqdm as notebook_tqdm
```

```
In [2]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("device:", device)

SEED = 1234
torch.manual_seed(SEED)
np.random.seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

device: cpu

Task 1. Training BERT from Scratch

```
In [3]: from datasets import load_dataset

dataset = load_dataset("wikitext", "wikitext-103-raw-v1", split="train[:100000]")

print("dataset rows:", dataset.num_rows)
print("columns:", dataset.column_names)

sentences = dataset["text"]
print("len(sentences):", len(sentences))
print("preview:", sentences[0][:200])
```

dataset rows: 100000
columns: ['text']
len(sentences): 100000
preview:

preview: is empty because WikiText contains a lot of empty lines at the start and throughout.

Find a non-empty preview

```
In [4]: # show first non-empty line
for s in sentences[:200]:
    if s and s.strip():
        print("non-empty preview:", s[:200])
        break
```

non-empty preview: = Valkyria Chronicles III =

```
In [ ]: #Do not want aggressive punctuation removal here, because WikiText has markup li
import re

text = []
for s in sentences:
    if not s or not s.strip():
        continue
    s = s.lower()
    s = re.sub(r"\s+", " ", s).strip()
    # keep lines with at least 5 words (helps reduce junk lines)
    if len(s.split()) >= 5:
        text.append(s)

print("After cleaning:", len(text))
print("example:", text[0][:200])
```

After cleaning: 62418

example: = valkyria chronicles iii =

Build vocab

```
In [6]: from collections import Counter

counter = Counter()
for s in text:
    counter.update(s.split())

special_tokens = ["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"]
VOCAB_SIZE = 30000

most_common = [w for w, _ in counter.most_common(VOCAB_SIZE - len(special_tokens))]
vocab = special_tokens + most_common

word2id = {w: i for i, w in enumerate(vocab)}
id2word = {i: w for w, i in word2id.items()}

print("Vocab size:", len(word2id))
print("Top tokens:", vocab[:30])
```

Vocab size: 30000

Top tokens: ['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', 'the', ',', '.', 'of', 'and', 'in', 'to', 'a', '=', '"', 'was', '@-@', 'on', 'as', 'that', "'s", 'for', 'with', 'by', ')', '(', 'is', 'he', 'his', 'at']

Create token_list

```
In [7]: token_list = []
unk_id = word2id["[UNK]"]

for s in text:
    token_list.append([word2id.get(w, unk_id) for w in s.split()])

print("token_list size:", len(token_list))
print("example ids:", token_list[0][:30])
```

token_list size: 62418

example ids: [13, 7960, 6351, 1213, 13]

```
In [ ]: #Hyperparameters
```

```
batch_size = 8
max_mask   = 10
max_len    = 64
```

```
In [ ]: from random import randrange, randint, shuffle, random
```

```
def make_batch():
```

```
    batch = []
```

```
    positive = negative = 0
```

```
    while positive != batch_size // 2 or negative != batch_size // 2:
```

```
        # choose two sentences
```

```
        a_idx, b_idx = randrange(len(token_list)), randrange(len(token_list))
```

```
        tokens_a, tokens_b = token_list[a_idx], token_list[b_idx]
```

```
        max_tokens_total = max_len - 3
```

```
        if max_tokens_total <= 0:
```

```
            raise ValueError("max_len too small")
```

```
        # split budget roughly half-half
```

```
        max_a = max_tokens_total // 2
```

```
        max_b = max_tokens_total - max_a
```

```
        tokens_a = tokens_a[:max_a]
```

```
        tokens_b = tokens_b[:max_b]
```

```
        # 1) input ids with special tokens
```

```
        input_ids = [word2id("[CLS]")] + tokens_a + [word2id("[SEP]")] + tokens_b
```

```
        # 2) segment ids
```

```
        segment_ids = [0] * (1 + len(tokens_a) + 1) + [1] * (len(tokens_b) + 1)
```

```
        # 3) MLM masking
```

```
        n_pred = min(max_mask, max(1, int(round(len(input_ids) * 0.15))))
```

```
        cand_masked_pos = [i for i, tok in enumerate(input_ids)
```

```
            if tok != word2id("[CLS]") and tok != word2id("[SEP]")]
```

```
        shuffle(cand_masked_pos)
```

```
        masked_tokens, masked_pos = [], []
```

```
        for pos in cand_masked_pos[:n_pred]:
```

```
            masked_pos.append(pos)
```

```
            masked_tokens.append(input_ids[pos])
```

```
            r = random()
```

```
            if r < 0.1: # 10% random token
```

```
                rand_id = randint(0, len(word2id) - 1)
```

```
                input_ids[pos] = rand_id
```

```
            elif r < 0.9: # 80% [MASK]
```

```
                input_ids[pos] = word2id("[MASK]")
```

```
            else: # 10% keep original
```

```
                pass
```

```
        # pad input_ids/segment_ids
```

```
        n_pad = max_len - len(input_ids)
```

```
        input_ids.extend([word2id("[PAD]")] * n_pad)
```

```
        segment_ids.extend([0] * n_pad)
```

```

# pad masked arrays
if max_mask > n_pred:
    pad_m = max_mask - n_pred
    masked_tokens.extend([word2id["[PAD]"] * pad_m])
    masked_pos.extend([0] * pad_m)

# NSP Label
if a_idx + 1 == b_idx and positive < batch_size // 2:
    batch.append([input_ids, segment_ids, masked_tokens, masked_pos, True])
    positive += 1
elif a_idx + 1 != b_idx and negative < batch_size // 2:
    batch.append([input_ids, segment_ids, masked_tokens, masked_pos, False])
    negative += 1

return batch

```

Test one batch

```

In [ ]: batch = make_batch()
        input_ids, segment_ids, masked_tokens, masked_pos, isNext = map(torch.LongTensor, batch)

        print("input_ids:", input_ids.shape)
        print("segment_ids:", segment_ids.shape)
        print("masked_tokens:", masked_tokens.shape)
        print("masked_pos:", masked_pos.shape)
        print("isNext:", isNext.shape)

```

```

input_ids: torch.Size([8, 64])
segment_ids: torch.Size([8, 64])
masked_tokens: torch.Size([8, 10])
masked_pos: torch.Size([8, 10])
isNext: torch.Size([8])

```

Model

```

In [ ]: n_layers = 2
        n_heads = 2
        d_model = 128
        d_ff = d_model * 4
        d_k = d_v = d_model // n_heads
        n_segments = 2

        print("d_k:", d_k, "d_v:", d_v)

```

```
d_k: 64 d_v: 64
```

Embedding (Token + Position + Segment)

```

In [12]: class Embedding(nn.Module):
        def __init__(self, vocab_size, d_model, max_len, n_segments):
            super().__init__()
            self.tok_embed = nn.Embedding(vocab_size, d_model) # token embedding
            self.pos_embed = nn.Embedding(max_len, d_model) # position embedding
            self.seg_embed = nn.Embedding(n_segments, d_model) # segment embedding
            self.norm = nn.LayerNorm(d_model)

        def forward(self, x, seg):
            # x: (bs, max_len)
            bs, seq_len = x.size()

```

```
pos = torch.arange(seq_len, device=x.device).unsqueeze(0).expand(bs, seq
out = self.tok_embed(x) + self.pos_embed(pos) + self.seg_embed(seg)
return self.norm(out)
```

Attention mask (pad mask)

```
In [13]: def get_attn_pad_mask(seq_q, seq_k, pad_id=0):
# seq_q: (bs, len_q), seq_k: (bs, len_k)
bs, len_q = seq_q.size()
bs, len_k = seq_k.size()
pad_attn_mask = seq_k.eq(pad_id).unsqueeze(1) # (bs, 1, len_k)
return pad_attn_mask.expand(bs, len_q, len_k) # (bs, len_q, len_k)
```

Scaled Dot-Product Attention

```
In [14]: class ScaledDotProductAttention(nn.Module):
def __init__(self):
super().__init__()

def forward(self, Q, K, V, attn_mask):
# Q,K,V: (bs, heads, len, d_k)
scores = torch.matmul(Q, K.transpose(-1, -2)) / math.sqrt(Q.size(-1)) #
scores.masked_fill_(attn_mask, -1e9)
attn = F.softmax(scores, dim=-1)
context = torch.matmul(attn, V)
return context, attn
```

Multi-Head Attention

```
In [15]: class MultiHeadAttention(nn.Module):
def __init__(self, n_heads, d_model, d_k, d_v):
super().__init__()
self.n_heads = n_heads
self.d_k = d_k
self.d_v = d_v

self.W_Q = nn.Linear(d_model, n_heads * d_k)
self.W_K = nn.Linear(d_model, n_heads * d_k)
self.W_V = nn.Linear(d_model, n_heads * d_v)
self.fc = nn.Linear(n_heads * d_v, d_model)

self.attention = ScaledDotProductAttention()
self.norm = nn.LayerNorm(d_model)

def forward(self, Q, K, V, attn_mask):
# Q,K,V: (bs, len, d_model)
residual = Q
bs = Q.size(0)

q_s = self.W_Q(Q).view(bs, -1, self.n_heads, self.d_k).transpose(1, 2)
k_s = self.W_K(K).view(bs, -1, self.n_heads, self.d_k).transpose(1, 2)
v_s = self.W_V(V).view(bs, -1, self.n_heads, self.d_v).transpose(1, 2)

# attn_mask: (bs, len_q, len_k) -> (bs, heads, len_q, len_k)
attn_mask = attn_mask.unsqueeze(1).repeat(1, self.n_heads, 1, 1)

context, attn = self.attention(q_s, k_s, v_s, attn_mask)
context = context.transpose(1, 2).contiguous().view(bs, -1, self.n_heads
```

```

        output = self.fc(context)

        return self.norm(output + residual), attn

```

FeedForward + EncoderLayer

```

In [16]: class PoswiseFeedForwardNet(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.norm = nn.LayerNorm(d_model)

    def forward(self, x):
        residual = x
        x = self.fc2(F.gelu(self.fc1(x)))
        return self.norm(x + residual)

class EncoderLayer(nn.Module):
    def __init__(self, n_heads, d_model, d_ff, d_k, d_v):
        super().__init__()
        self.enc_self_attn = MultiHeadAttention(n_heads, d_model, d_k, d_v)
        self.pos_ffn = PoswiseFeedForwardNet(d_model, d_ff)

    def forward(self, enc_inputs, enc_self_attn_mask):
        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs, enc_input
        enc_outputs = self.pos_ffn(enc_outputs)
        return enc_outputs, attn

```

```

In [ ]: # quick forward shape check ( just embedding + mask)
batch = make_batch()
input_ids, segment_ids, masked_tokens, masked_pos, isNext = map(torch.LongTensor

input_ids = input_ids.to(device)
segment_ids = segment_ids.to(device)

emb = Embedding(vocab_size=len(word2id), d_model=d_model, max_len=max_len, n_seg
x = emb(input_ids, segment_ids)
mask = get_attn_pad_mask(input_ids, input_ids, pad_id=word2id["[PAD]"])

print("embedding out:", x.shape)
print("mask:", mask.shape)

```

embedding out: torch.Size([8, 64, 128])

mask: torch.Size([8, 64, 64])

BERT model (Encoder stack + MLM + NSP heads)

```

In [ ]: class BERT(nn.Module):
    def __init__(self, n_layers, n_heads, d_model, d_ff, d_k, d_v, n_segments,
                 vocab_size, max_len, device):
        super().__init__()
        self.device = device

        self.embedding = Embedding(vocab_size, d_model, max_len, n_segments)
        self.layers = nn.ModuleList([
            EncoderLayer(n_heads, d_model, d_ff, d_k, d_v)
            for _ in range(n_layers)
        ])

```

```

#NSP head
self.fc_nsp = nn.Linear(d_model, 2)

#MLM head
self.fc_mlm1 = nn.Linear(d_model, d_model)
self.act = nn.GELU()
self.norm = nn.LayerNorm(d_model)
self.fc_mlm2 = nn.Linear(d_model, vocab_size)

def forward(self, input_ids, segment_ids, masked_pos):
    # input_ids: (bs, max_len)
    # segment_ids: (bs, max_len)
    # masked_pos: (bs, max_mask)

    output = self.embedding(input_ids, segment_ids) # (bs, max_len, d_model)

    enc_self_attn_mask = get_attn_pad_mask(
        input_ids, input_ids, pad_id=word2id["[PAD]"]
    ) # (bs, max_len, max_len)

    for layer in self.layers:
        output, _ = layer(output, enc_self_attn_mask)

    # NSP: use [CLS] token (position 0)
    cls_output = output[:, 0] # (bs, d_model)
    logits_nsp = self.fc_nsp(cls_output) # (bs, 2)

    #masked positions
    bs, max_m = masked_pos.size()
    masked_pos = masked_pos.unsqueeze(-1).expand(bs, max_m, output.size(-1))
    h_masked = torch.gather(output, 1, masked_pos) # (bs, max_mask, d_model)

    h_masked = self.fc_mlm1(h_masked)
    h_masked = self.act(h_masked)
    h_masked = self.norm(h_masked)
    logits_lm = self.fc_mlm2(h_masked) # (bs, max_mask, vocab_size)

    return logits_lm, logits_nsp

```

Instantiate model + quick forward test

```

In [19]: vocab_size = len(word2id)

model = BERT(
    n_layers=n_layers,
    n_heads=n_heads,
    d_model=d_model,
    d_ff=d_ff,
    d_k=d_k,
    d_v=d_v,
    n_segments=n_segments,
    vocab_size=vocab_size,
    max_len=max_len,
    device=device
).to(device)

criterion = nn.CrossEntropyLoss(ignore_index=word2id["[PAD]"])
optimizer = optim.Adam(model.parameters(), lr=2e-4)

```

Training (MLM + NSP)

```
In [ ]: #Loss + optimizer
criterion = nn.CrossEntropyLoss(ignore_index=word2id["[PAD]"])# ignore PAD in ML
optimizer = optim.Adam(model.parameters(), lr=1e-4)
```

Training loop

```
In [ ]: num_steps = 300
print_every = 50

model.train()
total_loss = 0.0

for step in tqdm(range(1, num_steps + 1), desc="Training"):
    batch = make_batch()
    input_ids, segment_ids, masked_tokens, masked_pos, isNext = map(torch.LongTensor, batch)

    input_ids = input_ids.to(device)
    segment_ids = segment_ids.to(device)
    masked_tokens = masked_tokens.to(device)
    masked_pos = masked_pos.to(device)
    isNext = isNext.to(device).long()

    optimizer.zero_grad()

    logits_lm, logits_nsp = model(input_ids, segment_ids, masked_pos)

    loss_lm = criterion(logits_lm.transpose(1, 2), masked_tokens)
    loss_nsp = nn.CrossEntropyLoss()(logits_nsp, isNext)

    loss = loss_lm + loss_nsp
    loss.backward()
    optimizer.step()

    total_loss += loss.item()

    if step % print_every == 0:
        avg = total_loss / print_every
        print(f"step {step}/{num_steps} | avg loss: {avg:.4f} | lm: {loss_lm.item():.4f} | nsp: {loss_nsp.item():.4f}")
        total_loss = 0.0
```

Training: 17% |██████| 50/300 [01:55<13:05, 3.14s/it]

step 50/300 | avg loss: 10.7784 | lm: 9.6458 | nsp: 0.7037

Training: 33% |██████| 100/300 [03:49<08:21, 2.51s/it]

step 100/300 | avg loss: 10.0728 | lm: 8.9756 | nsp: 0.6895

Training: 50% |██████| 150/300 [05:35<03:33, 1.43s/it]

step 150/300 | avg loss: 9.3931 | lm: 8.5256 | nsp: 0.6776

Training: 67% |██████| 200/300 [07:18<03:17, 1.98s/it]

step 200/300 | avg loss: 8.8667 | lm: 8.0579 | nsp: 0.6905

Training: 83% |██████| 250/300 [08:56<02:20, 2.81s/it]

step 250/300 | avg loss: 8.3771 | lm: 7.3926 | nsp: 0.6865

Training: 100% |██████| 300/300 [10:50<00:00, 2.17s/it]

step 300/300 | avg loss: 8.1841 | lm: 7.4469 | nsp: 0.6943

```
In [23]: save_path = "task1_bert_scratch.pth"
```

```

torch.save({
    "model_state_dict": model.state_dict(),
    "word2id": word2id,
    "id2word": id2word,
    "config": {
        "n_layers": n_layers,
        "n_heads": n_heads,
        "d_model": d_model,
        "d_ff": d_ff,
        "d_k": d_k,
        "d_v": d_v,
        "n_segments": n_segments,
        "max_len": max_len,
        "max_mask": max_mask,
        "vocab_size": len(word2id),
    }
}, save_path)

print(" Saved:", save_path)

```

Saved: task1_bert_scratch.pth

MLM inference check

helper to predict a masked token

```

In [25]: def encode_sentence(sentence, word2id, max_len):
    sentence = sentence.lower().strip()
    words = sentence.split()
    ids = [word2id["[CLS]"]] + [word2id.get(w, word2id["[UNK]"]) for w in words]
    if len(ids) > max_len:
        ids = ids[:max_len]
        ids[-1] = word2id["[SEP]"]
    pad_len = max_len - len(ids)
    ids += [word2id["[PAD]"]] * pad_len
    seg = [0] * max_len
    return ids, seg

def predict_mask(model, sentence, mask_index, topk=10):
    model.eval()
    ids, seg = encode_sentence(sentence, word2id, max_len)
    ids = torch.LongTensor([ids]).to(device)
    seg = torch.LongTensor([seg]).to(device)

    # set one position to [MASK]
    ids[0, mask_index] = word2id["[MASK]"]
    masked_pos = torch.LongTensor([[mask_index] + [0]*(max_mask-1)]).to(device)

    with torch.no_grad():
        logits_lm, _ = model(ids, seg, masked_pos) # (1, max_mask, vocab)
        scores = logits_lm[0, 0] # first masked position

    top_ids = torch.topk(scores, k=topk).indices.tolist()
    return [id2word[i] for i in top_ids]

```

```

In [26]: test_sentence = "the man is playing the guitar on the stage"
# mask_index counts in the tokenized input including [CLS] at position 0

```

```
# e.g., position 3 likely corresponds to "is" depending on splitting
print("Top predictions:", predict_mask(model, test_sentence, mask_index=3, topk=
```

Top predictions: ['=', ' ', 'the', 'in', '[UNK]', '.', 'to', 'and', 'a', 'of']

WikiText has many headings like = something = so = becomes very frequent

Task 2. Sentence Embedding with Sentence BERT

Load Task-1 checkpoint (word2id + config + weights)

```
In [27]: import torch

ckpt = torch.load("task1_bert_scratch.pth", map_location="cpu")
word2id = ckpt["word2id"]
id2word = ckpt["id2word"]
cfg = ckpt["config"]

print("Loaded vocab:", cfg["vocab_size"])
print("d_model:", cfg["d_model"], "max_len:", cfg["max_len"])
```

Loaded vocab: 30000

d_model: 128 max_len: 64

Rebuild the same BERT + load weights

```
In [ ]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("device:", device)

model = BERT(
    n_layers=cfg["n_layers"],
    n_heads=cfg["n_heads"],
    d_model=cfg["d_model"],
    d_ff=cfg["d_ff"],
    d_k=cfg["d_k"],
    d_v=cfg["d_v"],
    n_segments=cfg["n_segments"],
    vocab_size=cfg["vocab_size"],
    max_len=cfg["max_len"],
    device=device
).to(device)

model.load_state_dict(ckpt["model_state_dict"], strict=True)
print("Loaded Task-1 weights into BERT")
```

device: cpu

✓ Loaded Task-1 weights into BERT

Load SNLI

```
In [ ]: from datasets import load_dataset
import numpy as np

snli = load_dataset("snli")
print(snli)

# filter out label == -1
snli = snli.filter(lambda x: x["label"] != -1)
```

```
print("labels:", np.unique(snli["train"]["label"]))
print("train size:", len(snli["train"]), "val size:", len(snli["validation"]), "
```

d:\NLP\A4_AIT\.venv\Lib\site-packages\huggingface_hub\file_download.py:130: UserWarning: `huggingface_hub` cache-system uses symlinks by default to efficiently store duplicated files but your machine does not support them in C:\Users\Admin\cache\huggingface\hub\datasets--snli. Caching files will still work but in a degraded version that might require more space on your disk. This warning can be disabled by setting the `HF_HUB_DISABLE_SYMLINKS_WARNING` environment variable. For more details, see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations.

To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to activate developer mode, see this article: <https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development>

```
warnings.warn(message)
```

Warning: You are sending unauthenticated requests to the HF Hub. Please set a HF_TOKEN to enable higher rate limits and faster downloads.

Generating test split: 100%|██████████| 10000/10000 [00:00<00:00, 1428529.00 examples/s]

Generating validation split: 100%|██████████| 10000/10000 [00:00<00:00, 2220030.70 examples/s]

Generating train split: 100%|██████████| 550152/550152 [00:00<00:00, 5092738.94 examples/s]

```
DatasetDict({
  test: Dataset({
    features: ['premise', 'hypothesis', 'label'],
    num_rows: 10000
  })
  validation: Dataset({
    features: ['premise', 'hypothesis', 'label'],
    num_rows: 10000
  })
  train: Dataset({
    features: ['premise', 'hypothesis', 'label'],
    num_rows: 550152
  })
})
```

Filter: 100%|██████████| 10000/10000 [00:00<00:00, 446406.76 examples/s]

Filter: 100%|██████████| 10000/10000 [00:00<00:00, 444443.69 examples/s]

Filter: 100%|██████████| 550152/550152 [00:00<00:00, 605669.56 examples/s]

labels: [0 1 2]

train size: 549367 val size: 9842 test size: 9824

Preprocessing

text → ids + attention mask

In []: **import** re

```
PAD_ID = word2id["[PAD]"]
UNK_ID = word2id["[UNK]"]
CLS_ID = word2id["[CLS]"]
SEP_ID = word2id["[SEP]"]
```

```
max_len = cfg["max_len"]
```

```
def normalize_text(s: str) -> str:
    s = s.lower().strip()
```

```

s = re.sub(r"\s+", " ", s)
return s

def encode_pair(premise: str, hypothesis: str):
    # normalize
    p = normalize_text(premise)
    h = normalize_text(hypothesis)

    # word-level tokens
    p_ids = [word2id.get(w, UNK_ID) for w in p.split()]
    h_ids = [word2id.get(w, UNK_ID) for w in h.split()]

    # build CLS premise SEP hypothesis SEP
    ids = [CLS_ID] + p_ids + [SEP_ID] + h_ids + [SEP_ID]

    # truncate to max_len (keep last token as SEP)
    if len(ids) > max_len:
        ids = ids[:max_len]
        ids[-1] = SEP_ID

    attn = [1] * len(ids)

    # pad
    pad_len = max_len - len(ids)
    if pad_len > 0:
        ids += [PAD_ID] * pad_len
        attn += [0] * pad_len

    return ids, attn

```

preprocess_function for HuggingFace dataset

```

In [31]: def preprocess_function(examples):
    input_ids = []
    attention_mask = []

    for p, h in zip(examples["premise"], examples["hypothesis"]):
        ids, attn = encode_pair(p, h)
        input_ids.append(ids)
        attention_mask.append(attn)

    return {
        "input_ids": input_ids,
        "attention_mask": attention_mask,
        "labels": examples["label"],
    }

```

apply preprocessing

```

In [ ]: train_subset = snli["train"].shuffle(seed=SEED).select(range(20000))
val_subset = snli["validation"].select(range(2000))
test_subset = snli["test"].select(range(2000))

tokenized_train = train_subset.map(preprocess_function, batched=True, remove_columns=train_subset.column_names)
tokenized_val = val_subset.map(preprocess_function, batched=True, remove_columns=val_subset.column_names)
tokenized_test = test_subset.map(preprocess_function, batched=True, remove_columns=test_subset.column_names)

```

```
print(tokenized_train[0].keys())
print("train rows:", len(tokenized_train), "val rows:", len(tokenized_val), "tes
```

```
Map: 100%|██████████| 20000/20000 [00:00<00:00, 50235.28 examples/s]
Map: 100%|██████████| 2000/2000 [00:00<00:00, 52631.10 examples/s]
Map: 100%|██████████| 2000/2000 [00:00<00:00, 41657.06 examples/s]
dict_keys(['label', 'input_ids', 'attention_mask', 'labels'])
train rows: 20000 val rows: 2000 test rows: 2000
```

quick sanity check

```
In [35]: ex = tokenized_train[0]
print("input_ids len:", len(ex["input_ids"]))
print("attention_mask len:", len(ex["attention_mask"]))
print("label:", ex["labels"])
```

```
input_ids len: 64
attention_mask len: 64
label: 0
```

create DataLoaders

```
In [ ]: tokenized_train.set_format(type="torch", columns=["input_ids", "attention_mask",
tokenized_val.set_format(type="torch", columns=["input_ids", "attention_mask", "
tokenized_test.set_format(type="torch", columns=["input_ids", "attention_mask",

from torch.utils.data import DataLoader

batch_size_nli = 32

train_dataloader = DataLoader(tokenized_train, batch_size=batch_size_nli, shuffl
eval_dataloader = DataLoader(tokenized_val, batch_size=batch_size_nli)
test_dataloader = DataLoader(tokenized_test, batch_size=batch_size_nli)

# quick shape check
batch = next(iter(train_dataloader))
print(batch["input_ids"].shape, batch["attention_mask"].shape, batch["labels"].s
```

```
torch.Size([32, 64]) torch.Size([32, 64]) torch.Size([32])
```

Mean pooling

```
In [ ]: # token_embeds: (bs, seq_len, hidden)
# attention_mask: (bs, seq_len)

def mean_pool(token_embeds, attention_mask):
    in_mask = attention_mask.unsqueeze(-1).expand(token_embeds.size()).float()
    pooled = torch.sum(token_embeds * in_mask, dim=1) / torch.clamp(in_mask.sum(
    return pooled
```

Siamese forward utils (u, v)

```
In [ ]: SEP_ID = word2id["[SEP]"]

def split_pair(input_ids, attention_mask):
    # input_ids: (bs, max_len)
    # find first SEP position in each row
    bs, L = input_ids.shape
```

```

sep_pos = (input_ids == SEP_ID).int().argmax(dim=1) # (bs,)

# build masks for A/B
idxs = torch.arange(L, device=input_ids.device).unsqueeze(0).expand(bs, L)

mask_a = (idxs <= sep_pos.unsqueeze(1)).long()
mask_b = (idxs >= sep_pos.unsqueeze(1)).long()

ids_a = input_ids
ids_b = input_ids

attn_a = attention_mask * mask_a
attn_b = attention_mask * mask_b

return ids_a, attn_a, ids_b, attn_b

def encode_sentence_embedding(model, input_ids, attention_mask, segment_ids):
    raise NotImplementedError

```

Add encode() method to BERT (returns last hidden state)

```

In [40]: def bert_encode(self, input_ids, segment_ids):
          output = self.embedding(input_ids, segment_ids)
          enc_self_attn_mask = get_attn_pad_mask(input_ids, input_ids, pad_id=word2id[
          for layer in self.layers:
              output, _ = layer(output, enc_self_attn_mask)
          return output # (bs, max_len, d_model)

          # monkey-patch method into BERT instance/class
          BERT.encode = bert_encode
          print(" Added BERT.encode() to return hidden states")

```

Added BERT.encode() to return hidden states

Compute u,v embeddings for a batch

```

In [ ]: def batch_uv_embeddings(model, input_ids, attention_mask):
          segment_ids = torch.zeros_like(input_ids)
          hidden = model.encode(input_ids, segment_ids) # (bs, L, d_model)

          ids_a, attn_a, ids_b, attn_b = split_pair(input_ids, attention_mask)

          u = mean_pool(hidden, attn_a)
          v = mean_pool(hidden, attn_b)
          return u, v

```

SoftmaxLoss head

```

In [43]: class SoftmaxClassifier(nn.Module):
          def __init__(self, d_model, num_labels=3):
              super().__init__()
              self.linear = nn.Linear(d_model * 3, num_labels)

          def forward(self, u, v):
              uv_abs = torch.abs(u - v)
              x = torch.cat([u, v, uv_abs], dim=-1)
              return self.linear(x)

          classifier_head = SoftmaxClassifier(cfg["d_model"], num_labels=3).to(device)

```

```

criterion = nn.CrossEntropyLoss()
optimizer_bert = torch.optim.Adam(model.parameters(), lr=2e-5)
optimizer_cls = torch.optim.Adam(classifier_head.parameters(), lr=2e-5)

```

Training loop

```

In [ ]: from sklearn.metrics import accuracy_score

num_epochs = 1

for epoch in range(num_epochs):
    model.train()
    classifier_head.train()

    total_loss = 0.0
    preds_all, labels_all = [], []

    for batch in tqdm(train_dataloader, desc=f"Epoch {epoch+1}/{num_epochs}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["labels"].to(device)

        optimizer_bert.zero_grad()
        optimizer_cls.zero_grad()

        u, v = batch_uv_embeddings(model, input_ids, attention_mask)
        logits = classifier_head(u, v)

        loss = criterion(logits, labels)
        loss.backward()

        optimizer_bert.step()
        optimizer_cls.step()

        total_loss += loss.item()

        preds = torch.argmax(logits, dim=1).detach().cpu().numpy()
        preds_all.extend(preds.tolist())
        labels_all.extend(labels.detach().cpu().numpy().tolist())

    acc = accuracy_score(labels_all, preds_all)
    print(f"Epoch {epoch+1}: loss={total_loss/len(train_dataloader):.4f}, acc={a

```

```

Epoch 1/1: 100%|██████████| 625/625 [00:18<00:00, 33.80it/s]
Epoch 1: loss=1.0945, acc=0.3735

```

Validation

```

In [46]: from sklearn.metrics import classification_report

model.eval()
classifier_head.eval()

preds_all, labels_all = [], []

with torch.no_grad():
    for batch in tqdm(eval_dataloader, desc="Validation"):
        input_ids = batch["input_ids"].to(device)

```

```

attention_mask = batch["attention_mask"].to(device)
labels = batch["labels"].to(device)

u, v = batch_uv_embeddings(model, input_ids, attention_mask)
logits = classifier_head(u, v)

preds = torch.argmax(logits, dim=1).cpu().numpy()
preds_all.extend(preds.tolist())
labels_all.extend(labels.cpu().numpy().tolist())

print(classification_report(labels_all, preds_all, target_names=["entailment", "n

```

Validation: 100%|██████████| 63/63 [00:00<00:00, 144.94it/s]

	precision	recall	f1-score	support
entailment	0.44	0.44	0.44	663
neutral	0.39	0.63	0.48	677
contradiction	0.35	0.14	0.20	660
accuracy			0.41	2000
macro avg	0.40	0.40	0.38	2000
weighted avg	0.40	0.41	0.38	2000

```

In [48]: torch.save({
    "bert_state_dict": model.state_dict(),
    "classifier_state_dict": classifier_head.state_dict(),
    "word2id": word2id,
    "id2word": id2word,
    "cfg": cfg
}, "task2_sbert_snli_softmaxloss.pth")

print("Saved task2_sbert_snli_softmaxloss.pth")

```

Saved task2_sbert_snli_softmaxloss.pth

Inference function

```

In [49]: label_map = {0: "entailment", 1: "neutral", 2: "contradiction"}

def predict_nli(premise, hypothesis):
    model.eval()
    classifier_head.eval()

    ids, attn = encode_pair(premise, hypothesis)
    input_ids = torch.LongTensor([ids]).to(device)
    attention_mask = torch.LongTensor([attn]).to(device)

    with torch.no_grad():
        u, v = batch_uv_embeddings(model, input_ids, attention_mask)
        logits = classifier_head(u, v)
        pred = torch.argmax(logits, dim=1).item()

    return label_map[pred]

# test
print(predict_nli("A man is playing a guitar on stage.", "The man is performing

```

entailment

Task 3. Evaluation and Analysis

Evaluate on TEST set

```
In [50]: from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import numpy as np

model.eval()
classifier_head.eval()

preds_all, labels_all = [], []

with torch.no_grad():
    for batch in tqdm(test_dataloader, desc="Test"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["labels"].to(device)

        u, v = batch_uv_embeddings(model, input_ids, attention_mask)
        logits = classifier_head(u, v)

        preds = torch.argmax(logits, dim=1)
        preds_all.extend(preds.cpu().numpy().tolist())
        labels_all.extend(labels.cpu().numpy().tolist())

target_names = ["entailment", "neutral", "contradiction"]
print("Test accuracy:", accuracy_score(labels_all, preds_all))
print(classification_report(labels_all, preds_all, target_names=target_names))
```

Test: 100%|██████████| 63/63 [00:00<00:00, 143.09it/s]

Test accuracy: 0.4095

	precision	recall	f1-score	support
entailment	0.45	0.43	0.44	690
neutral	0.39	0.61	0.47	660
contradiction	0.40	0.18	0.25	650
accuracy			0.41	2000
macro avg	0.41	0.41	0.39	2000
weighted avg	0.41	0.41	0.39	2000

Confusion matrix

```
In [51]: cm = confusion_matrix(labels_all, preds_all)
print("Confusion matrix (rows=true, cols=pred):\n", cm)

# Optional: normalize to percentages
cm_norm = cm / cm.sum(axis=1, keepdims=True)
print("\nNormalized confusion matrix:\n", np.round(cm_norm, 3))
```

```
Confusion matrix (rows=true, cols=pred):  
[[298 307  85]  
 [163 404  93]  
 [196 337 117]]
```

```
Normalized confusion matrix:  
[[0.432 0.445 0.123]  
 [0.247 0.612 0.141]  
 [0.302 0.518 0.18  ]]
```

The model achieved a test accuracy of approximately 41%, which is above the random baseline of 33% for three classes, indicating that the learned sentence embeddings capture meaningful semantic information. Performance is strongest on the neutral class, while contradiction has significantly lower recall. The confusion matrix shows that many contradiction and entailment examples are misclassified as neutral, suggesting that the learned embeddings are not yet sufficiently discriminative to clearly separate fine-grained semantic relationships.

I use some limit choices on hyperparameters to ensure efficient runtime. The BERT encoder was trained with architecture (2 layers, hidden size 128) and a maximum sequence length of 64 to allow CPU-based training. Additionally, only a subset of 20,000 SNLI samples and one training epoch were used for fine-tuning. While these decisions improved computational efficiency, they likely limited the model's representational capacity and led to underfitting. Increasing model size, training duration, sequence length, and adopting a subword tokenizer would likely improve contradiction detection and overall performance.