University of Central Missouri

Department of Computer Science & Cybersecurity

CS5760 Natural Language Processing

Fall 2025

# Homework 4.

Student name:

Kopparthi Sai Siva Shankara Vara Prasad

Student ID: 700765221

## Submission Requirements:

- Once finished your assignment push your source code to your repo (GitHub) and explain the work through the ReadMe file properly. Make sure you add your student info in the ReadMe file.
- Submit your GitHub link on the Bright Space.
- Comment your code appropriately ***IMPORTANT.***
- Any submission after provided deadline is considered as a late submission.

**Part I. Short Answer**

**1) RNN Families & Use-Cases (Many-to-X)**
a) Map each task to the most suitable RNN I/O pattern and explain why (1–2 lines each):
• next-word prediction

• sentiment of a sentence

• NER

• machine translation
(Choose from: one-to-many, many-to-one, many-to-many aligned, many-to-many unaligned.)

**Answer)**

**RNN Patterns for Tasks**

- **Next-word prediction** is best suited to a **one-to-many** architecture because the goal is to generate a sequence of outputs from a single input token or prompt. For example, given an initial input word or character, the model needs to generate a sequence of subsequent tokens, making predictions step-by-step. This pattern captures sequential data and can be seen in text generation systems where a single input such as "Once" unfolds into a longer sentence or paragraph. The model learns contextual dependencies and gradually unfolds information.
- **Sentiment analysis of a sentence** follows a **many-to-one** RNN pattern. Here, the input consists of multiple tokens (a sentence or document), and the output is a single label or continuous value representing the sentiment—often positive, neutral, or negative. This setup allows the RNN to process each token in the sentence, associate meaning to the context, and finally condense everything into a single output. This structure is powerful because the sequential nature of RNNs considers how words influence one another, allowing nuanced sentiment detection.
- **Named Entity Recognition (NER)** operates using a **many-to-many aligned** pattern because it requires tagging each token of an input sequence with an output label (e.g., PERSON, LOCATION, etc.). Since every input word is associated with a corresponding output tag, alignment of input and output lengths is essential. The RNN processes the full sequence and emits labels simultaneously, capturing both local and contextual patterns in the input text while maintaining a one-to-one mapping between tokens and labels.
- **Machine translation** uses a **many-to-many unaligned** architecture because the input and output sequence lengths usually differ, and the structure between input and output tokens is not directly aligned. For instance, translating “How are you?” into Spanish becomes “¿Cómo estás?”, where word order and structure differ. This requires an encoder-decoder model, where the encoder reads the input sentence into a context vector, and the decoder generates the translated sentence step-by-step. This separation helps the model handle long-range dependencies and cross-lingual challenges.

b) In one sentence, explain how “unrolling” over time enables BPTT and weight sharing.

**Answer)**

*Unrolling* an RNN over time allows us to treat the temporal sequence as a deep structure where each timestep corresponds to a layer. This unrolled representation is essential for **Backpropagation Through Time (BPTT)**, as it allows gradients to flow backward through time steps and share weights across them. This shared structure significantly reduces the number of parameters and enables efficient learning, though it also introduces challenges like vanishing gradients.

c) Give one advantage and one limitation of weight sharing across time in RNNs.

**Answer)**

Weight sharing in RNNs offers the tremendous advantage of dramatically reducing model size since the same parameters are reused at every time step. It also allows the model to generalize patterns across different input positions because it doesn't rely on position-specific weights. However, the downside is that this can make learning long-term dependencies difficult, especially when early timesteps have little influence on later predictions due to the vanishing gradient phenomenon.

**2) Vanishing Gradients & Remedies**

a) Describe the vanishing gradient problem in RNNs and how it affects long-range dependencies.

**Answer)**

The **vanishing gradient problem** occurs in RNNs when gradients shrink exponentially as they propagate backward through time, especially over long sequences. This happens due to repeated multiplication by small values (like small derivatives from activation functions), effectively erasing the gradient information. As a result, it's hard for the model to learn dependencies that span many time steps (e.g., linking a pronoun to a noun several words earlier). This fundamentally limits the model’s ability to handle complex contextual patterns and long-term relationships.

b) List two *architectural* solutions and briefly how each helps gradient flow.

**Answer)**

Two architectural solutions include:

- **LSTM (Long Short-Term Memory)**: LSTMs mitigate vanishing gradients through their cell state, which can carry information across long sequences with minimal modification,

thanks to gating mechanisms. These gates enable the controlled flow and storage of information and preserve gradients during backpropagation.
- **GRU (Gated Recurrent Unit)**: GRUs simplify the LSTM by combining the forget and input gates into a single update gate. Their simpler structure lowers computational cost while still effectively addressing vanishing gradients by allowing the network to preserve information across many time steps using gated mechanisms.

c) Give one *training* technique (not an architecture change) that can mitigate the issue and why.

**Answer)**

A training technique that combats vanishing gradients is **gradient clipping**, where the gradients are scaled down if they exceed a predefined threshold. This ensures stability during updates, especially in recurrent structures, and prevents exploding gradients, which indirectly helps by not amplifying small gradients to the point where they become negligible over time.

## 3) LSTM Gates & Cell State

a) Explain the roles of the forget, input, and output gates. For each, name its activation and purpose.

**Answer)**

In an LSTM, the **forget gate**—using a sigmoid activation—decides what information to discard from the cell state so old, irrelevant information is removed. The **input gate**, also controlled by a sigmoid function, determines what new information from the current input and hidden state should be added to the cell. The **output gate** regulates how much of the cell state influences the current output by applying sigmoid activation followed by a tanh function for the output vector. Together, these gates form memory that can adaptively store and forget information.

b) Why is the LSTM cell state often described as providing a “linear path” for gradients?

**Answer)**

The LSTM cell state serves as a nearly linear path for gradients over time, meaning there are minimal disruptions or transformations applied to it from step to step. This property creates a highway-like flow for gradients to pass back unobstructed, which is critical for learning long-range dependencies in sequences by avoiding exponential decay of gradients.

c) In one or two sentences, contrast “what to remember” vs. “what to expose” in LSTMs.

**Answer)**

*What to remember* refers to the choice of storing information in the cell state through the input and forget gates—this determines which parts of past information are important. *What to expose* is determined by the output gate and involves controlling which parts of the stored information influence the current output. This separation allows LSTMs to filter information effectively through time for efficient learning.

## 4) Self-Attention

a) Define Query (Q), Key (K), and Value (V) in the context of self-attention.

**Answer)**

In self-attention, the **Query (Q)** represents the vector the model uses to search for relevant information, the **Key (K)** represents labels or markers that match against queries, and the **Value (V)** carries the actual information to be used in downstream tasks. For each position in the sequence, the self-attention mechanism looks at all other positions via Q-K similarity and uses those weighted similarities to combine values.

b) Write the formula for dot-product attention.

**Answer)**

The scaled dot-product attention formula is:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

where $QK^T$ computes similarity, softmax normalizes these into weights, and $V$ applies those weights to extract relevant information.

c) Why do we divide by $\sqrt{d_k}$?

**Answer)**

Dividing by $\sqrt{d_k}$(where $d_k$is the Key dimension) stabilizes softmax computation by preventing large values in the dot product, which could result in a sharp, uninformative softmax distribution. This helps maintain smooth gradient flow and consistent learning across different dimensionalities.

**5) Multi-Head Attention & Residual Connections**

a) Why do Transformers use multi-head attention instead of single-head attention?

**Answer)**

Transformers use **multi-head attention** to allow the model to attend to different parts of the input in different ways. For example, one head might focus on syntactic structure, another on semantic roles, and another on positional relationships. Having multiple perspectives enriches the contextual understanding significantly and allows deeper modeling of linguistic patterns than a single attention head.

b) What is the purpose of Add & Norm (Residual + LayerNorm)? Explain two benefits.

**Answer)**

*Add & Norm*, composed of residual connections and layer normalization, is essential to ensure stable and efficient training. Residual connections ease gradient flow by adding inputs directly to outputs of sublayers, reducing vanishing gradient problems and improving deep network learning. Layer normalization ensures that training remains stable by normalizing activations throughout the network (especially across batch sizes of 1, like in machine translation).

c) Describe one example of linguistic relation that different heads might capture (e.g., coreference, syntax).

**Answer)**

For example, one head might capture **coreference resolution**, connecting "Barack Obama" to "he" later in the text. Another might focus on **syntactic dependencies**, determining which word is the subject of a verb. This division of labor makes the model highly expressive.

**6) Encoder–Decoder with Masked Attention**

a) Why does the decoder use masked self-attention? What problem does it prevent?

**Answer)**

Masked self-attention ensures that during sequence generation, the model cannot look ahead at future tokens. This prevents “cheating” where the decoder accidentally uses future context during training. Each token can only attend to previous tokens, preserving the autoregressive nature of generation and modeling true prediction context.

b) What is the difference between encoder self-attention and encoder–decoder cross-attention?

**Answer)**

**Encoder self-attention** computes attention over the entire source sequence to build contextualized token embeddings. **Encoder–decoder cross-attention**, on the other hand, allows the decoder to attend to encoder outputs, enabling it to incorporate source context while generating target sequences. This mechanism is essential for translation-based tasks.

c) During inference (no teacher forcing), how does the model generate tokens step by step?

**Answer)**

During inference, token generation happens one step at a time without teacher forcing. The model generates the first token, feeds it back as input to predict the next one, and repeats this process until a stopping condition like an EOS token is reached. This process helps the model learn real-world sequence generation.

**Part II: Programming**

**Q1. Character-Level RNN Language Model (“hello” toy & beyond)**
Goal: Train a tiny character-level RNN to predict the next character given previous characters.

Data (toy to start):

- Start with a small toy corpus you create (e.g., several “hello…”, “help…”, short words/sentences).

- Then expand to a short plain-text file of ~50–200 KB (any public-domain text of your choice).

Model:

- Embedding → RNN (Vanilla RNN or GRU or LSTM) → Linear → Softmax over characters.
- Hidden size 64–256; sequence length 50–100; batch size 64; train 5–20 epochs.

Train:

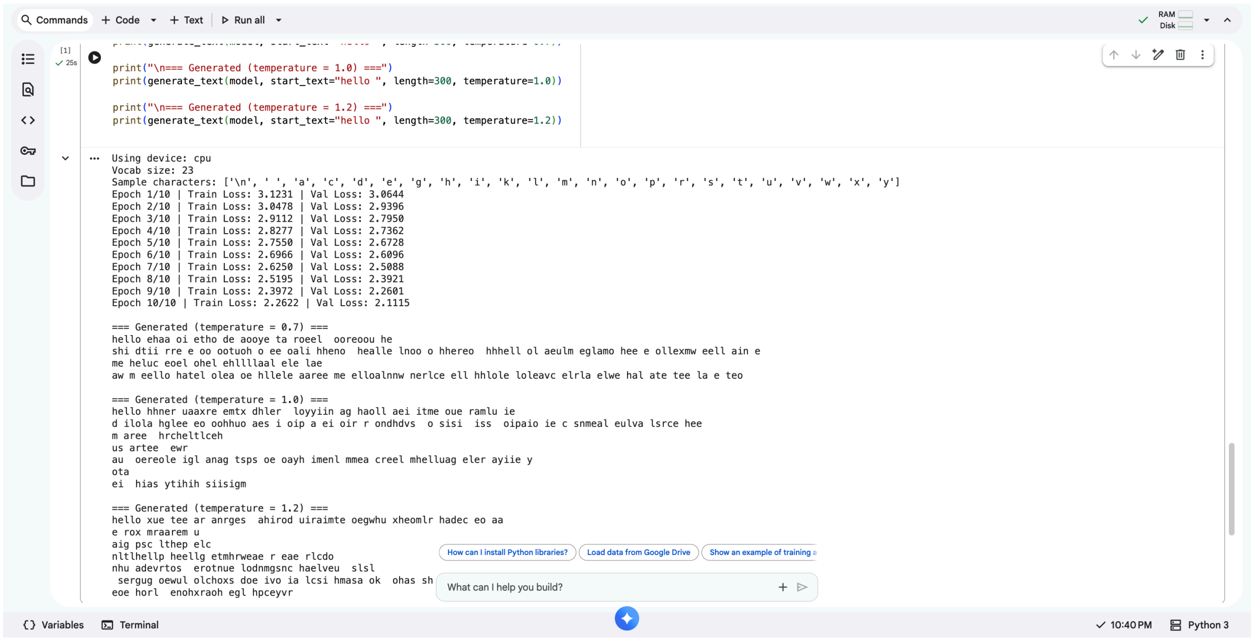
- Teacher forcing (use the true previous char as input during training).
- Cross-entropy loss; Adam optimizer.

Report:

1. Training/validation loss curves.
2. Sample 3 temperature-controlled generations (e.g., τ = 0.7, 1.0, 1.2) for 200–400 chars each.
3. A 3–5 sentence reflection: what changes when you vary sequence length, hidden size, and temperature?
   (Connect to slides: embedding, sampling loop, teacher forcing, tradeoffs)

**OUTPUT:**

**Entire data is written in GitHub Readme file. Here is the Screenshot of the output.**



```python
print("\n=== Generated (temperature = 1.0) ===")
print(generate_text(model, start_text="hello ", length=300, temperature=1.0))

print("\n=== Generated (temperature = 1.2) ===")
print(generate_text(model, start_text="hello ", length=300, temperature=1.2))
```

```
Using device: cpu
Vocab size: 23
Sample characters: ['\n', ' ', 'a', 'c', 'd', 'e', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y']
Epoch 1/10 | Train Loss: 3.1231 | Val Loss: 3.0644
Epoch 2/10 | Train Loss: 3.0478 | Val Loss: 2.9396
Epoch 3/10 | Train Loss: 2.9112 | Val Loss: 2.7950
Epoch 4/10 | Train Loss: 2.8277 | Val Loss: 2.7362
Epoch 5/10 | Train Loss: 2.7550 | Val Loss: 2.6728
Epoch 6/10 | Train Loss: 2.6966 | Val Loss: 2.6096
Epoch 7/10 | Train Loss: 2.6250 | Val Loss: 2.5088
Epoch 8/10 | Train Loss: 2.5195 | Val Loss: 2.3921
Epoch 9/10 | Train Loss: 2.3972 | Val Loss: 2.2601
Epoch 10/10 | Train Loss: 2.2622 | Val Loss: 2.1115

=== Generated (temperature = 0.7) ===
hello ehaa oi etho de aooye ta roeel  ooreoou he
shi dtii rre e oo ootuoh o ee oali hheno  healle lnoo o hhereo  hhhell ol aeulm eglamo hee e ollexmw eell ain e
me heluc eoel ohel ehllllaal ele lae
aw m eello hatel olea oe hllele aaree me elloalnnw nerlce ell hhlole loleavc elrla elwe hal ate tee la e teo

=== Generated (temperature = 1.0) ===
hello hhner uaaxre emtx dhler  loyyiin ag haoll aei itme oue ramlu ie
d ilola hglee eo oohhuo aes i oip a ei oir r ondhdvs  o sisi  iss  oipaio ie c snmeal eulva lsrce hee
m aree  hrcheltlceh
us artee  ewr
au  oereole igl anag tsps oe oayh imenl mmea creel mhelluag eler ayiie y
ota
ei  hias ytihih siisigm

=== Generated (temperature = 1.2) ===
hello xue tee ar anrges  ahirod uiraimte oegwhu xheomlr hadec eo aa
e rox mraarem u
aig psc lthep elc
nltlhellp heellg etmhrweae r eae rlcdo
nhu adevrtos  erotnue lodnmgsnc haelveu  slsl
 sergug oewul olchoxs doe ivo ia lcsi hmasa ok  ohas sh
eoe horl  enohxraoh egl hpceyvr
```

Commands
+ Code
+ Text
Run all
RAM
Disk
[1]
25s
How can I install Python libraries?
Load data from Google Drive
Show an example of training
What can I help you build?
Variables
Terminal
10:40 PM
Python 3

## Q2. Mini Transformer Encoder for Sentences

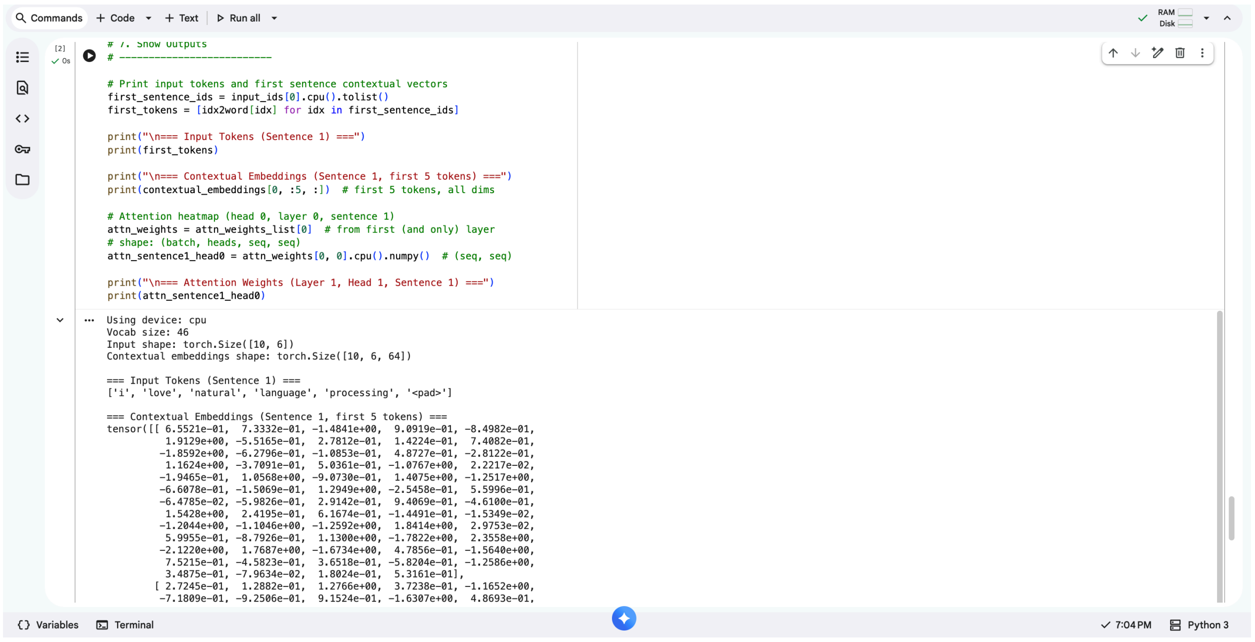
Task: Build a mini Transformer Encoder (NOT full decoder) to process a batch of sentences.

Steps:

1. Use a small dataset (e.g., 10 short sentences of your choice).
2. Tokenize and embed the text.
3. Add sinusoidal positional encoding.
4. Implement:
   - Self-attention layer
   - Multi-head attention (2 or 4 heads)
   - Feed-forward layer
   - Add & Norm
5. Show:
   - Input tokens
   - Final contextual embeddings
   - Attention heatmap between words (visual or printed)

**OUTPUT:**

**Entire data is written in GitHub Readme file. Here is the Screenshot of the output.**



```
# 7. Show Outputs
# --------------------------

# Print input tokens and first sentence contextual vectors
first_sentence_ids = input_ids[0].cpu().tolist()
first_tokens = [idx2word[idx] for idx in first_sentence_ids]

print("\n=== Input Tokens (Sentence 1) ===")
print(first_tokens)

print("\n=== Contextual Embeddings (Sentence 1, first 5 tokens) ===")
print(contextual_embeddings[0, :5, :])  # first 5 tokens, all dims

# Attention heatmap (head 0, layer 0, sentence 1)
attn_weights = attn_weights_list[0]  # from first (and only) layer
# shape: (batch, heads, seq, seq)
attn_sentence1_head0 = attn_weights[0, 0].cpu().numpy()  # (seq, seq)

print("\n=== Attention Weights (Layer 1, Head 1, Sentence 1) ===")
print(attn_sentence1_head0)
```

```
Using device: cpu
Vocab size: 46
Input shape: torch.Size([10, 6])
Contextual embeddings shape: torch.Size([10, 6, 64])

=== Input Tokens (Sentence 1) ===
['i', 'love', 'natural', 'language', 'processing', '<pad>']

=== Contextual Embeddings (Sentence 1, first 5 tokens) ===
tensor([[ 6.5521e-01,  7.3332e-01, -1.4841e+00,  9.0919e-01, -8.4982e-01,
          1.9129e+00, -5.5165e-01,  2.7812e-01,  1.4224e-01,  7.4082e-01,
         -1.8592e+00, -6.2796e-01, -1.0853e-01,  4.8727e-01, -2.8122e-01,
          1.1624e+00, -3.7091e-01,  5.0361e-01, -1.0767e+00,  2.2217e-02,
         -1.9465e-01,  1.0568e+00, -9.0730e-01,  1.4075e+00, -1.2517e+00,
         -6.6078e-01, -1.5069e-01,  1.2949e+00, -2.5458e-01,  5.5996e-01,
         -6.4785e-02, -5.9826e-01,  2.9142e-01,  9.4069e-01, -4.6100e-01,
          1.5428e+00,  2.4195e-01,  6.1674e-01, -1.4491e-01, -1.5349e-02,
         -1.2044e+00, -1.1046e+00, -1.2592e+00,  1.8414e+00,  2.9753e-02,
          5.9955e-01, -8.7926e-01,  1.1300e+00, -1.7822e+00,  2.3558e+00,
         -2.1220e+00,  1.7687e+00, -1.6734e+00,  4.7856e-01, -1.5640e+00,
          7.5215e-01, -4.5823e-01,  3.6518e-01, -5.8204e-01, -1.2586e+00,
          3.4875e-01, -7.9634e-02,  1.8024e-01,  5.3161e-01],
        [ 2.7245e-01,  1.2882e-01,  1.2766e+00,  3.7238e-01, -1.1652e+00,
         -7.1809e-01, -9.2506e-01,  9.1524e-01, -1.6307e+00,  4.8693e-01,
```

## Q3. Implement Scaled Dot-Product Attention

Goal: Implement the attention function from your slides:

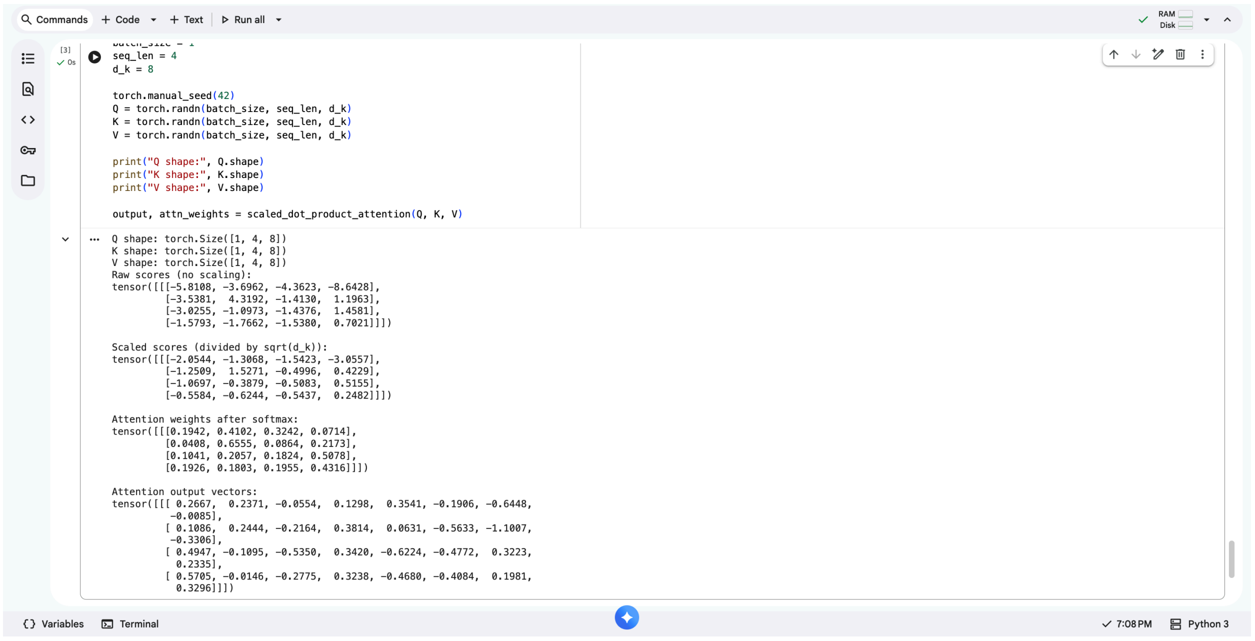
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Requirements:

- Write a function in PyTorch or TensorFlow to compute attention.
- Test it using random Q, K, V inputs.
- Print:
  - Attention weight matrix
  - Output vectors
  - Softmax stability check (before and after scaling)

## OUTPUT:

**Entire data is written in GitHub Readme file. Here is the Screenshot of the output.**



```python
seq_len = 4
d_k = 8

torch.manual_seed(42)
Q = torch.randn(batch_size, seq_len, d_k)
K = torch.randn(batch_size, seq_len, d_k)
V = torch.randn(batch_size, seq_len, d_k)

print("Q shape:", Q.shape)
print("K shape:", K.shape)
print("V shape:", V.shape)

output, attn_weights = scaled_dot_product_attention(Q, K, V)
```

```
Q shape: torch.Size([1, 4, 8])
K shape: torch.Size([1, 4, 8])
V shape: torch.Size([1, 4, 8])
Raw scores (no scaling):
tensor([[[-5.8108, -3.6962, -4.3623, -8.6428],
         [-3.5381,  4.3192, -1.4130,  1.1963],
         [-3.0255, -1.0973, -1.4376,  1.4581],
         [-1.5793, -1.7662, -1.5380,  0.7021]]])

Scaled scores (divided by sqrt(d_k)):
tensor([[[-2.0544, -1.3068, -1.5423, -3.0557],
         [-1.2509,  1.5271, -0.4996,  0.4229],
         [-1.0697, -0.3879, -0.5083,  0.5155],
         [-0.5584, -0.6244, -0.5437,  0.2482]]])

Attention weights after softmax:
tensor([[[0.1942, 0.4102, 0.3242, 0.0714],
         [0.0408, 0.6555, 0.0864, 0.2173],
         [0.1041, 0.2057, 0.1824, 0.5078],
         [0.1926, 0.1803, 0.1955, 0.4316]]])

Attention output vectors:
tensor([[[ 0.2667,  0.2371, -0.0554,  0.1298,  0.3541, -0.1906, -0.6448,
          -0.0085],
         [ 0.1086,  0.2444, -0.2164,  0.3814,  0.0631, -0.5633, -1.1007,
          -0.3306],
         [ 0.4947, -0.1095, -0.5350,  0.3420, -0.6224, -0.4772,  0.3223,
           0.2335],
         [ 0.5705, -0.0146, -0.2775,  0.3238, -0.4680, -0.4084,  0.1981,
           0.3296]]])
```