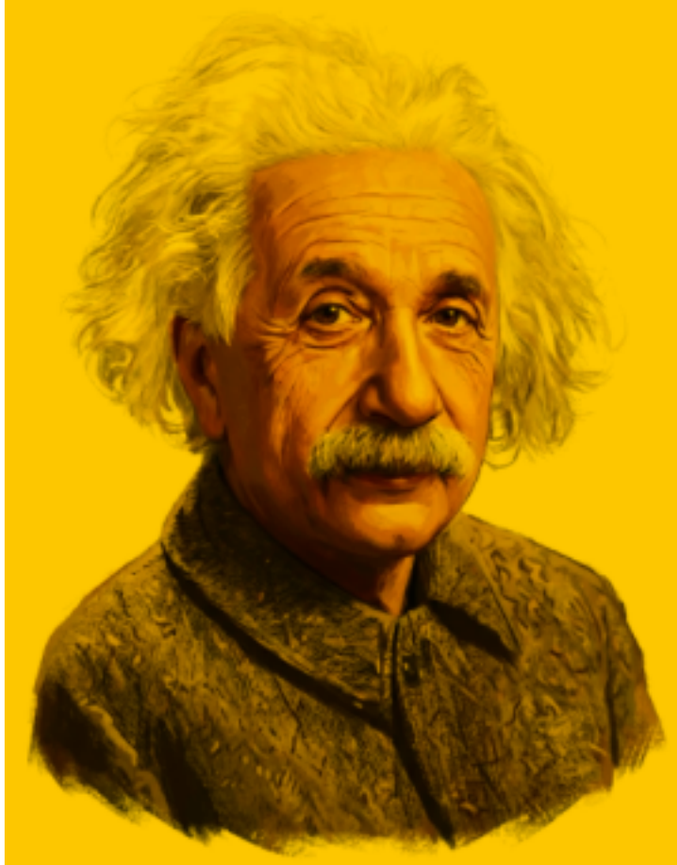
CS 668 Applied Large Language Models
Spring Semester, 2026
Doc 13 Cluster, Scaling, Attention
Feb 24, 2026

Copyright ©, All rights reserved. 2026 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/openpub/) license defines the copyright on this document.



Meet Einstein

Einstein is an AI with a computer. He logs into Canvas every day, watches lectures, reads essays, writes papers, participates in discussions, and submits your homework — automatically.

Get started

How it works

Einstein has a full virtual computer with a browser — anything you can do, he can do.

Some Toy Educational GPT Models

minGPT

PyTorch re-implementation of GPT

nanoGPT

rewrite of minGPT to be faster

PicoGPT

minimal implementation of GPT-2 in plain NumPy

SMOL-GPT

PyTorch implementation for training your own small LLM from scratch

smolGPT

Source and runtime

See website for urls

SDSU GPU Cluster

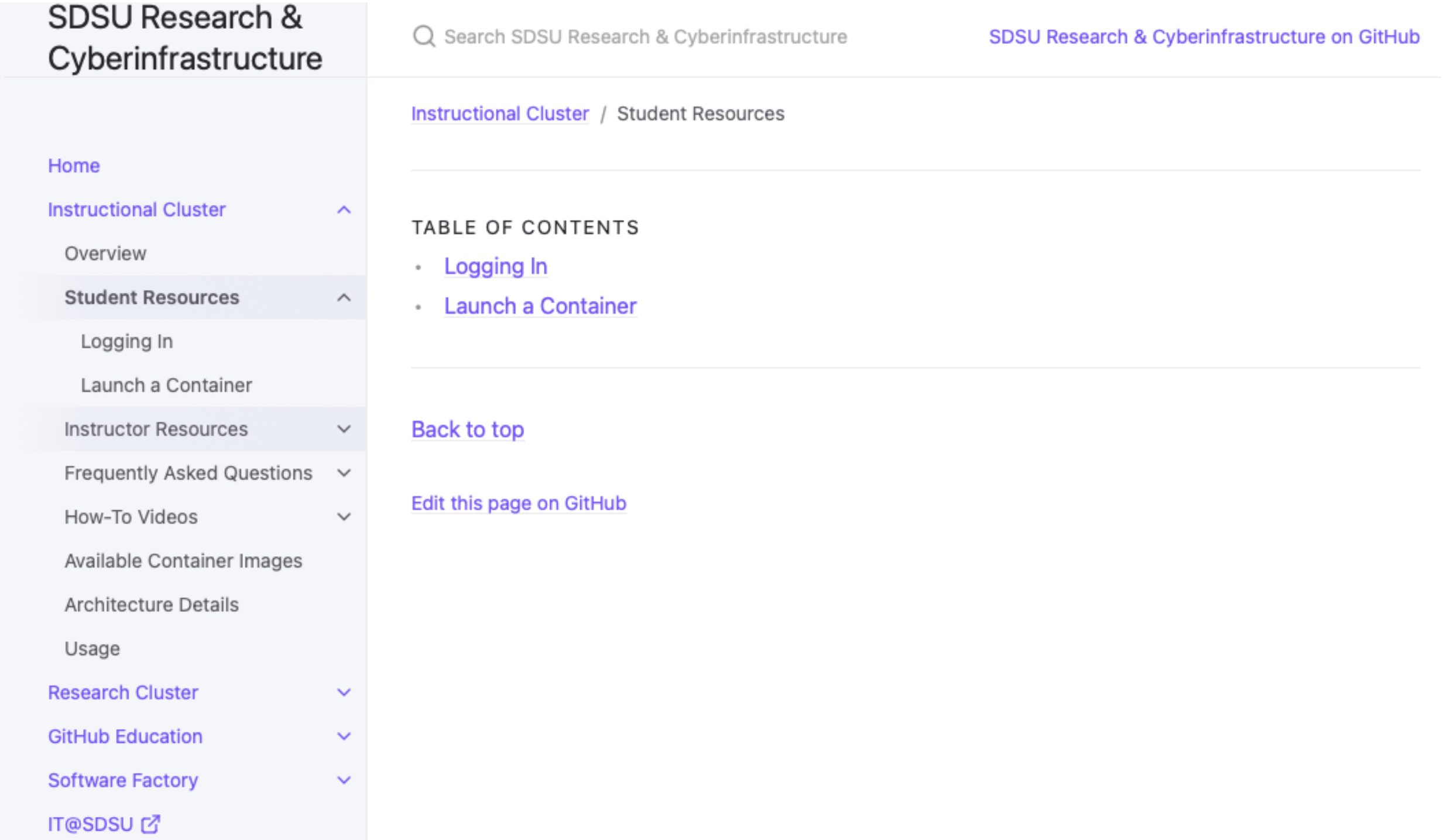
PowerEdge R750XA
(2x) Intel Xeon Gold 6338 2G CPU, 32C/64T
(4x) Nvidia A100 GPU, 80 GB RAM
512 GB System RAM

The GPUs available have 10GB of RAM
Suitable for training/fine-tuning 7-8 billion parameter models.
Students 75GB of persistent storage space

Accessing GPU Cluster

Documentation

https://sdsu-research-ci.github.io/instructionalcluster/students



Logging In

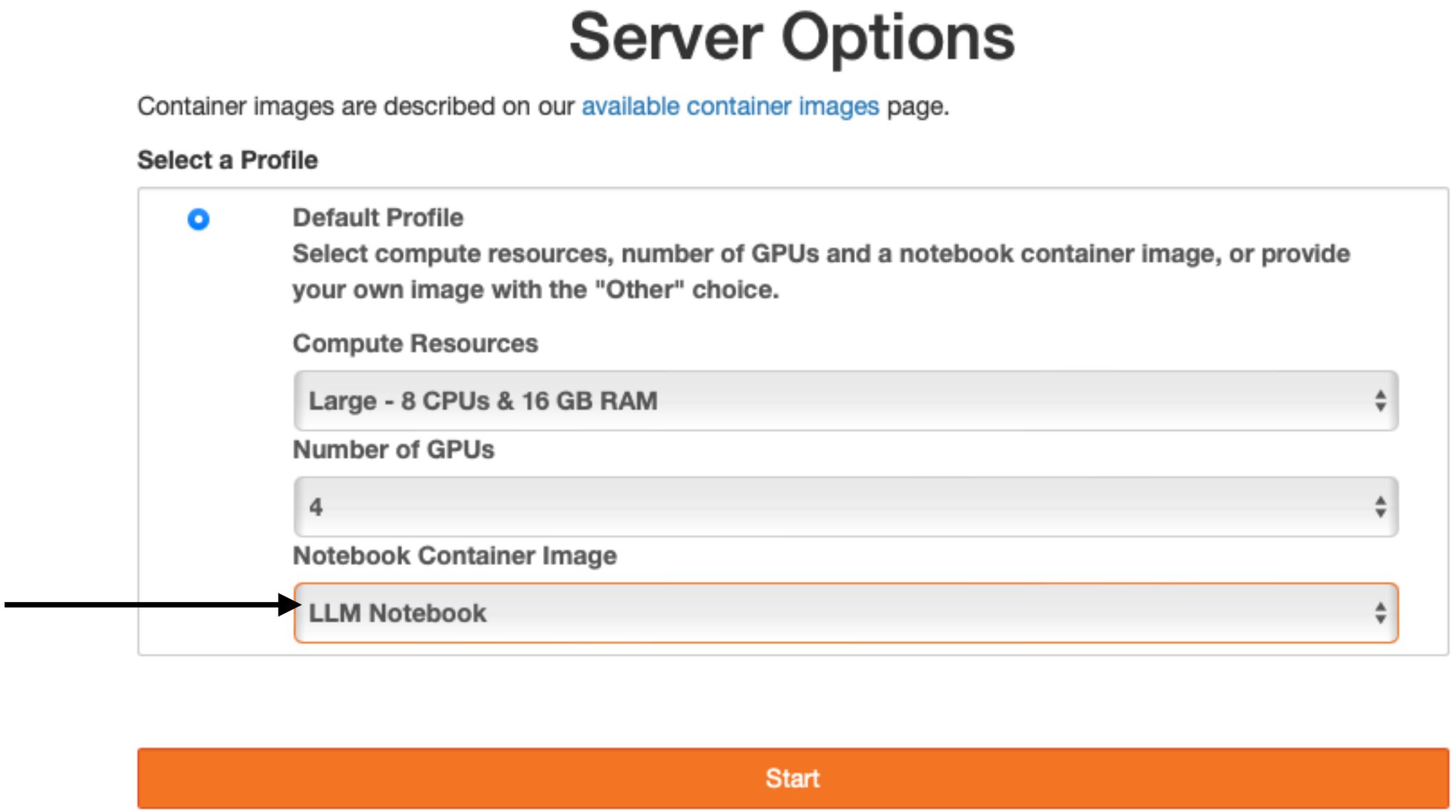
Go to:

https://jupyterhub.sdsu.edu/

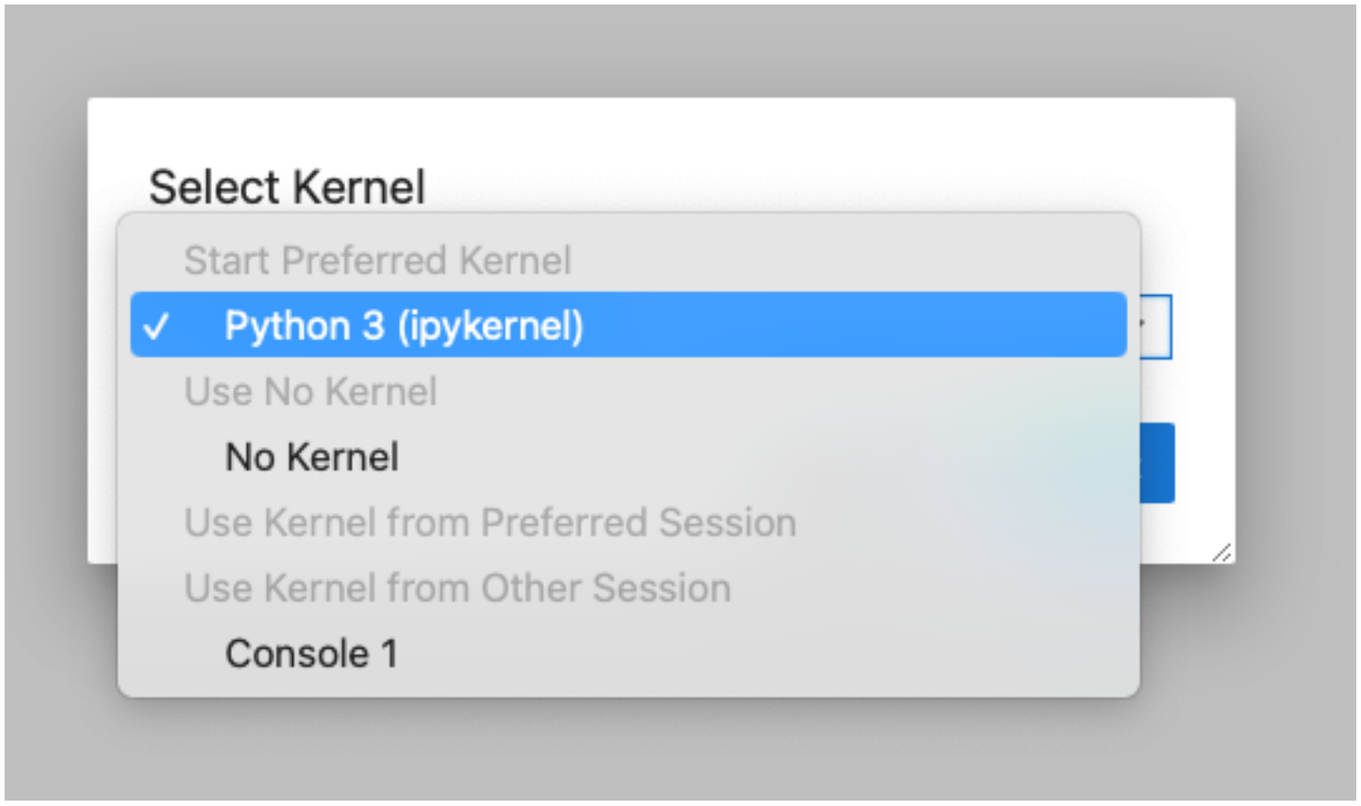


Enter your SDSUid credentials

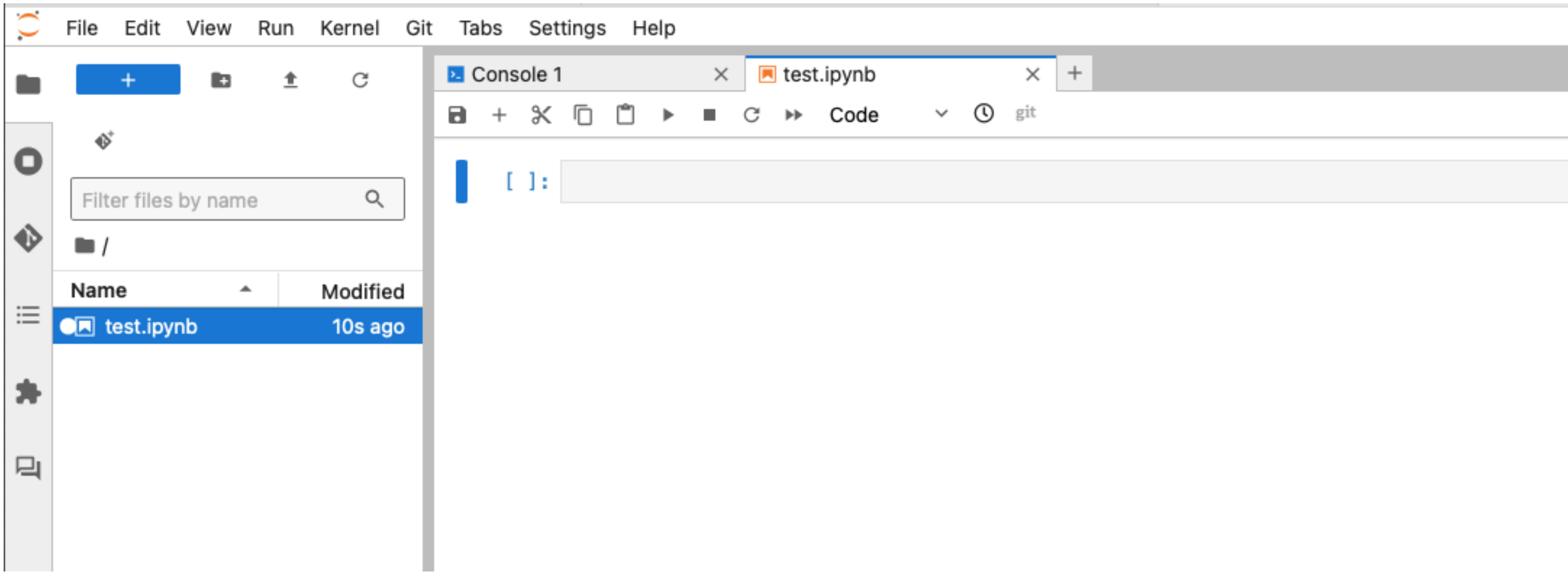
Enter Server Options



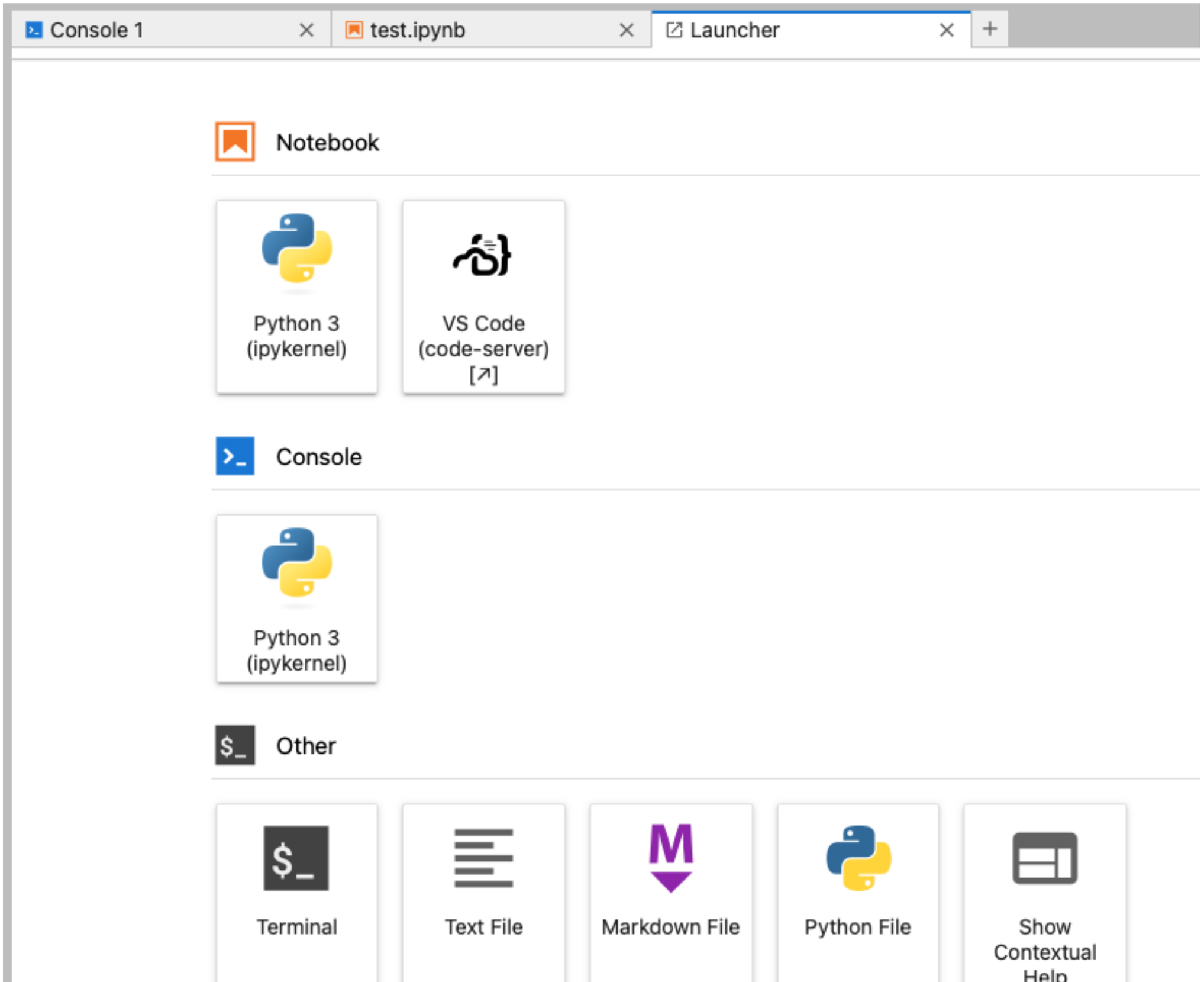
Select Kernel



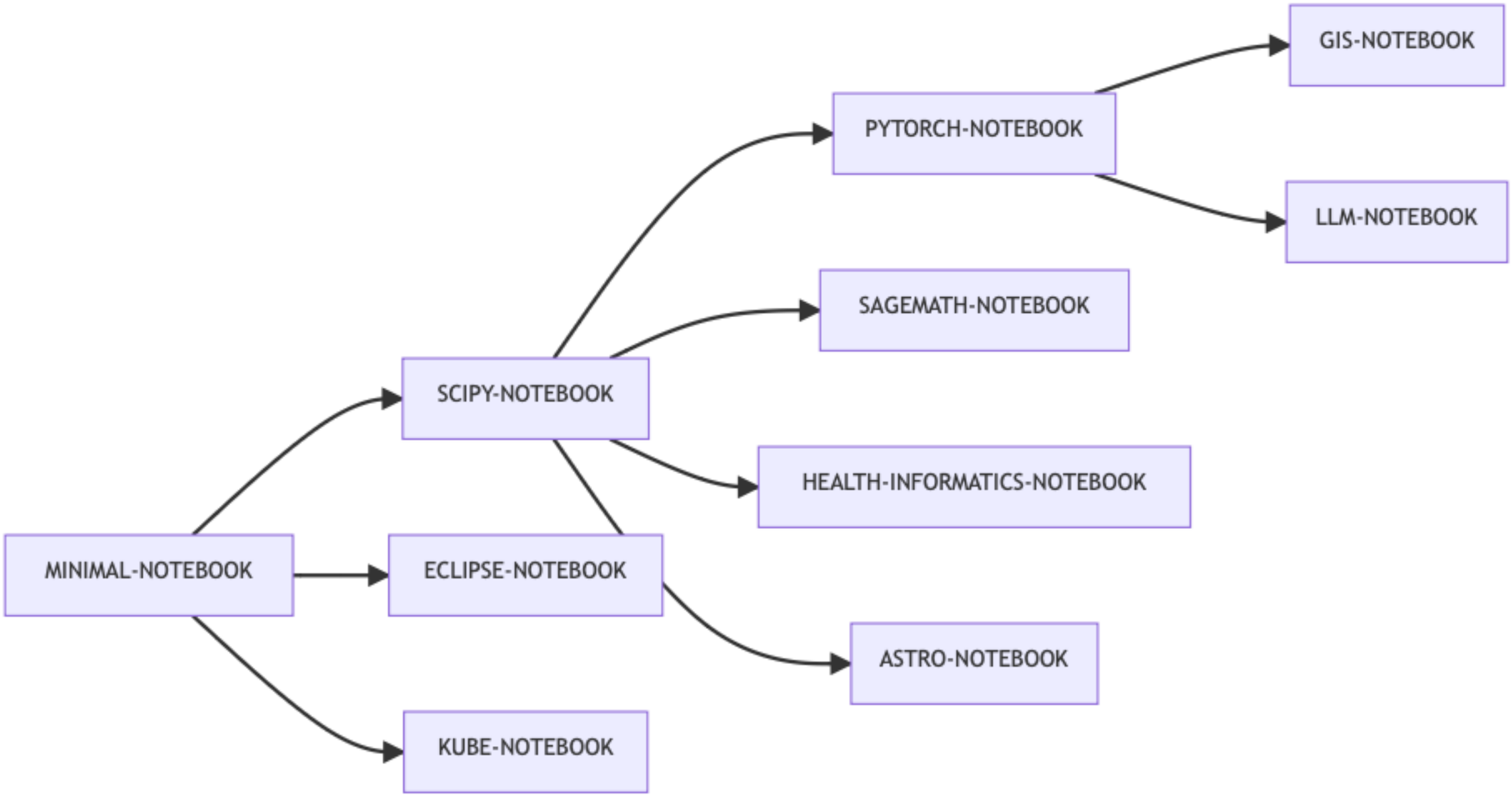
Get Jupyter Interface



Launcher


Console 1
test.ipynb
Launcher
Notebook
Python 3 (ipykernel)
VS Code (code-server) [↗]
Console
Python 3 (ipykernel)
Other
Terminal
Text File
Markdown File
Python File
Show Contextual Help

Large Language Model Notebook



minimal-notebook

Ubuntu 22.04.4
JupyterLab 4.2.4
Python 3.11.9
Git 2.34.1
vi
nano
wget
curl
unzip
tzdata

SciPy Notebook

altair, beautifulsoup4, bokeh, bottleneck, cloudpickle, conda-forge::blas=*=openblas, cython, dask, dill, h5py, jupyterlab-git, matplotlib-base, numba, numexpr, openpyxl, pandas, patsy, protobuf, pytables, scikit-image, scikit-learn, scipy, seaborn, sqlalchemy, statsmodel, sympy, widgetsnbextension, xlrd packages

ipympl and ipywidgets for interactive visualizations and plots in Python notebooks

Facets for visualizing machine learning datasets

PyTorch Notebook

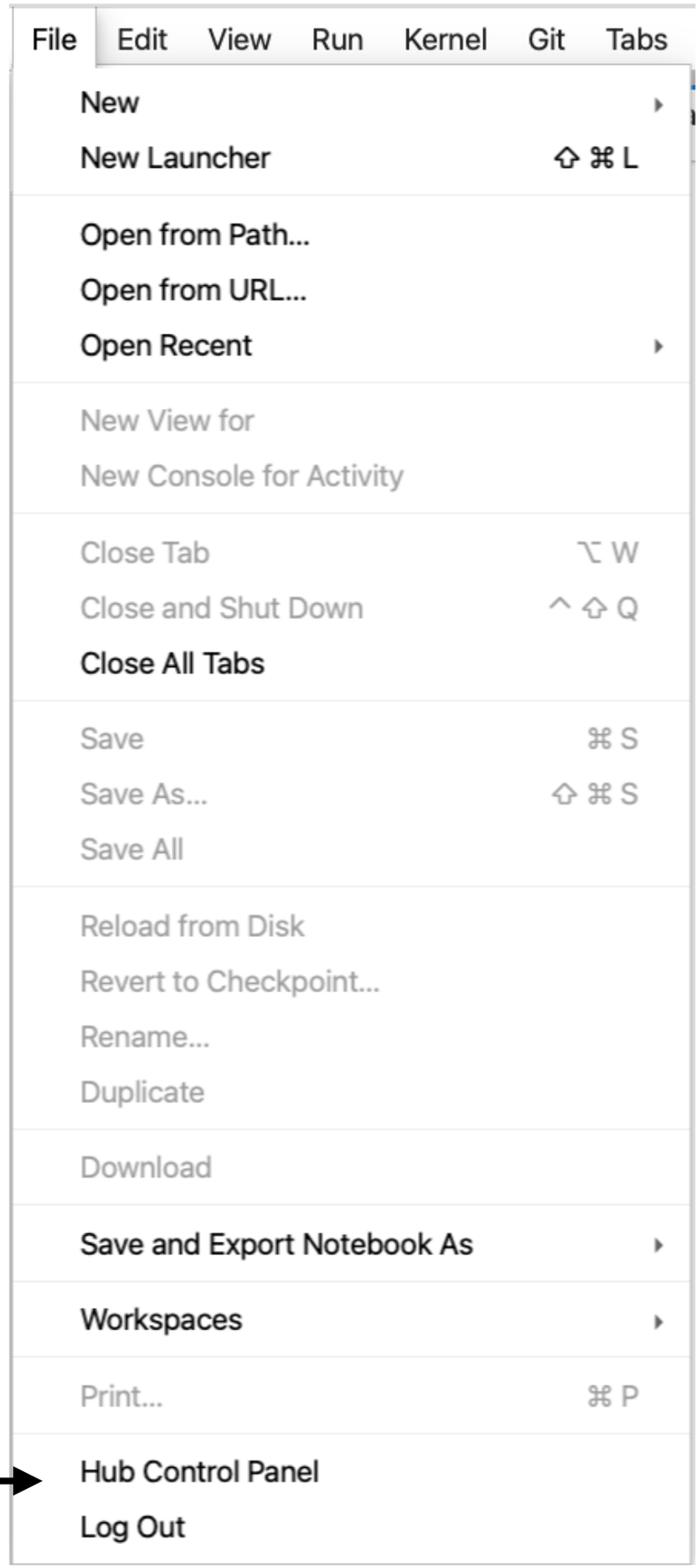
pytorch machine learning library
torch, torchaudio and torchvision

Large Language Model Notebook

rclone
FastChat
Ollama
VS Code Server
Jupyter AI
bitsandbytes
transformers
peft
accelerate
trl
ollama-python
openai
pyaudio
portaudio
cuda-nvcc

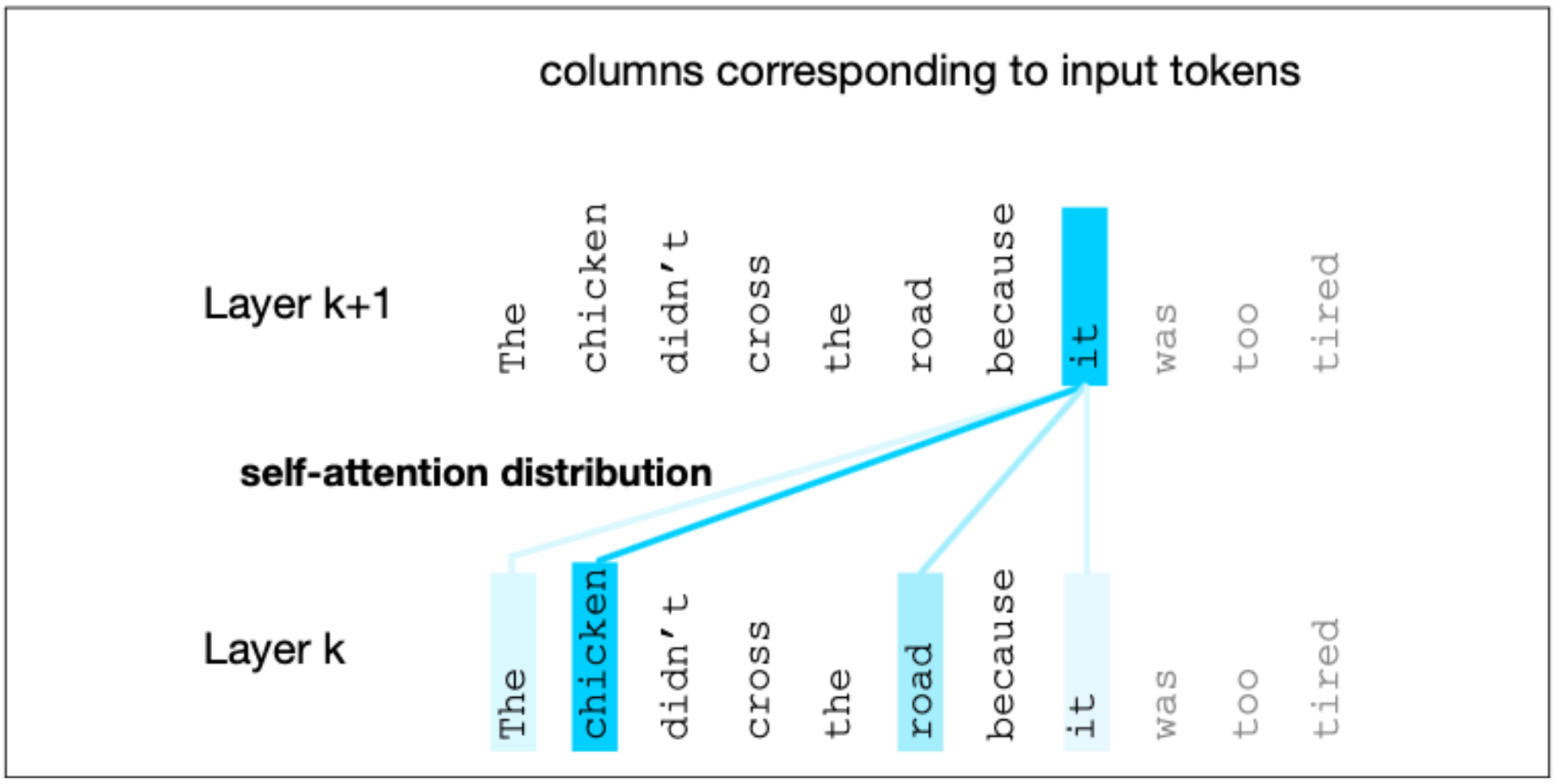
deepspeed
langchain
huggingface_hub
auto_gptq
autoawq
xformers
dask-kubernetes
chromadb

Manually Stopping the Server



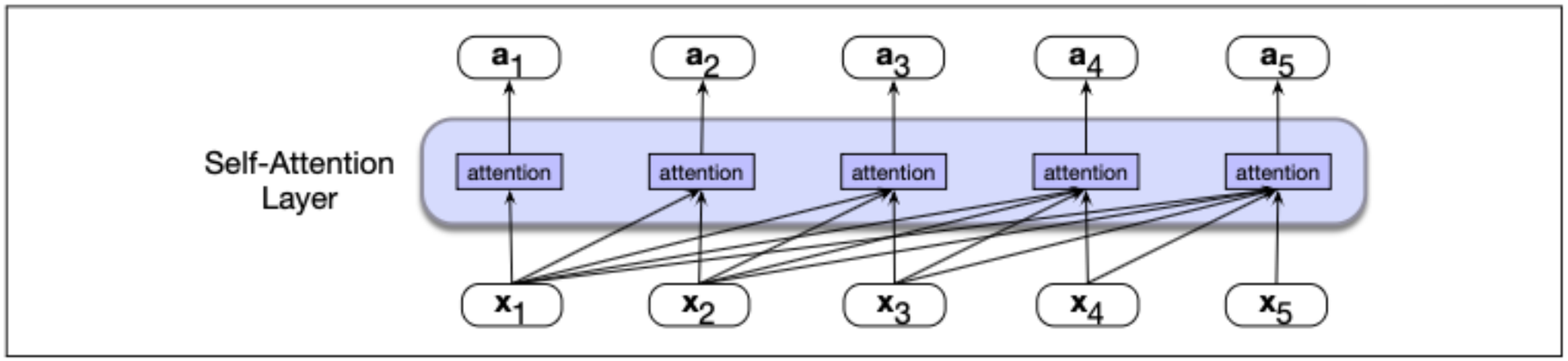
Attention

Build the relationship of tokens in layer K+1 from layer K



Speech and Language Processing, Jurafsky & Martin, Chapter 8

Simplified Attention



$\mathbf{a}_i$ Attention output at location i

$\mathbf{x}_j$

α_{ij} How much $\mathbf{x}_j$ should contribute to $\mathbf{a}_i$

$$\mathbf{a_i} = \sum_{j \leq i} \alpha_{\mathbf{ij}} \mathbf{x_j}$$

Speech and Language Processing, Jurafsky & Martin, Chapter 8

How to Compute $\mathbf{a}_i$

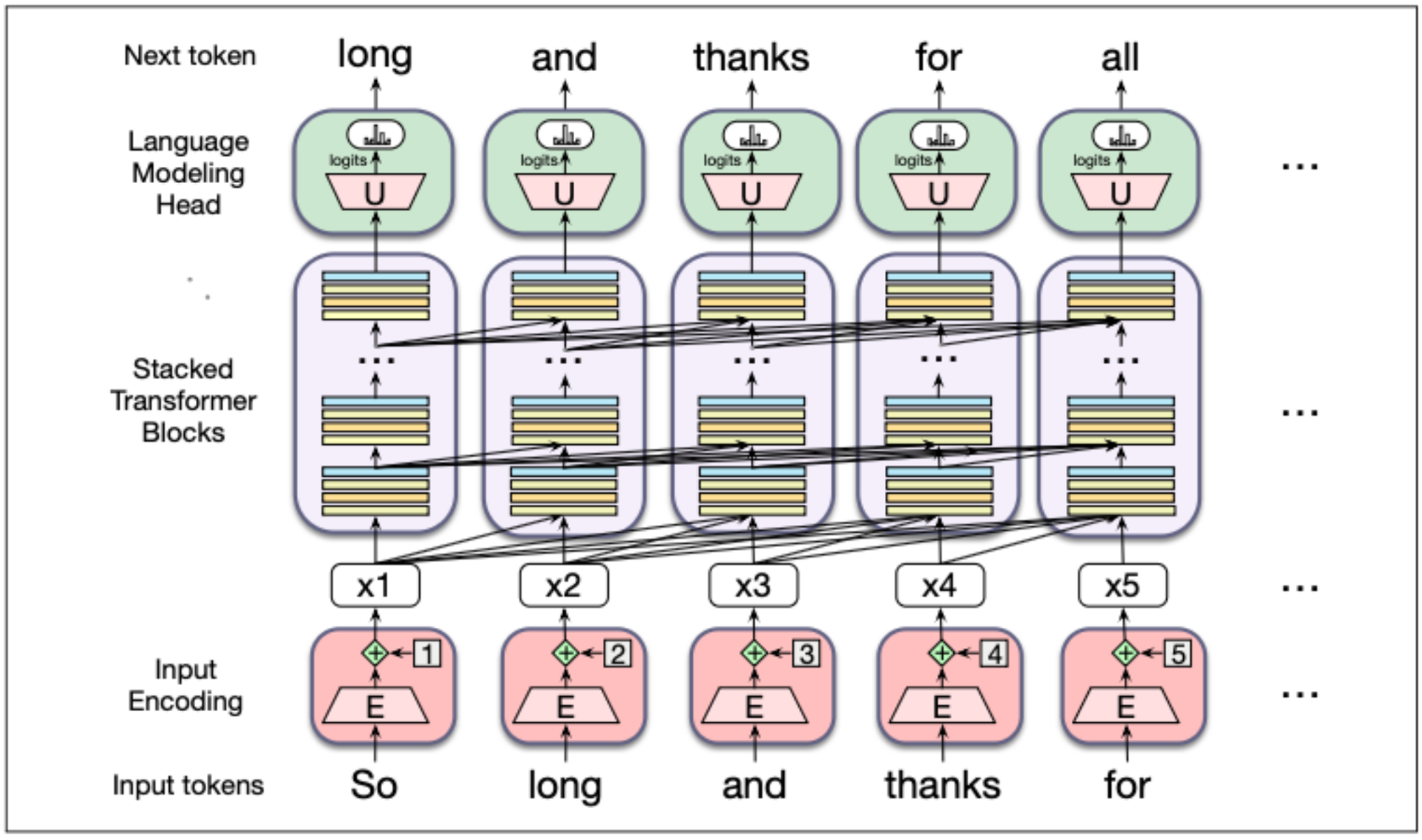
$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

$$\text{score}(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \forall i \leq j$$

Real Attention

Prompt: So long and thanks for



Speech and Language Processing, Jurafsky & Martin, Chapter 8

Three Roles

query

Current element being compared to preceding elements

$\mathbf{q}$

key

A preceding element that is being compared to the current element

$\mathbf{k}$

value

The value of a preceding element that gets weighted and summed to compute the output of the current element

$\mathbf{v}$

The Calculation

Matricies to map $\mathbf{x}_i$ to query, key, value

$\mathbf{W^Q}$

$\mathbf{W^K}$

$\mathbf{W^V}$

Computing query, key, value

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W^Q}$$

$$\mathbf{k}_i = \mathbf{x}_i\mathbf{W^K}$$

$$\mathbf{v}_i = \mathbf{x}_i\mathbf{W^V}$$

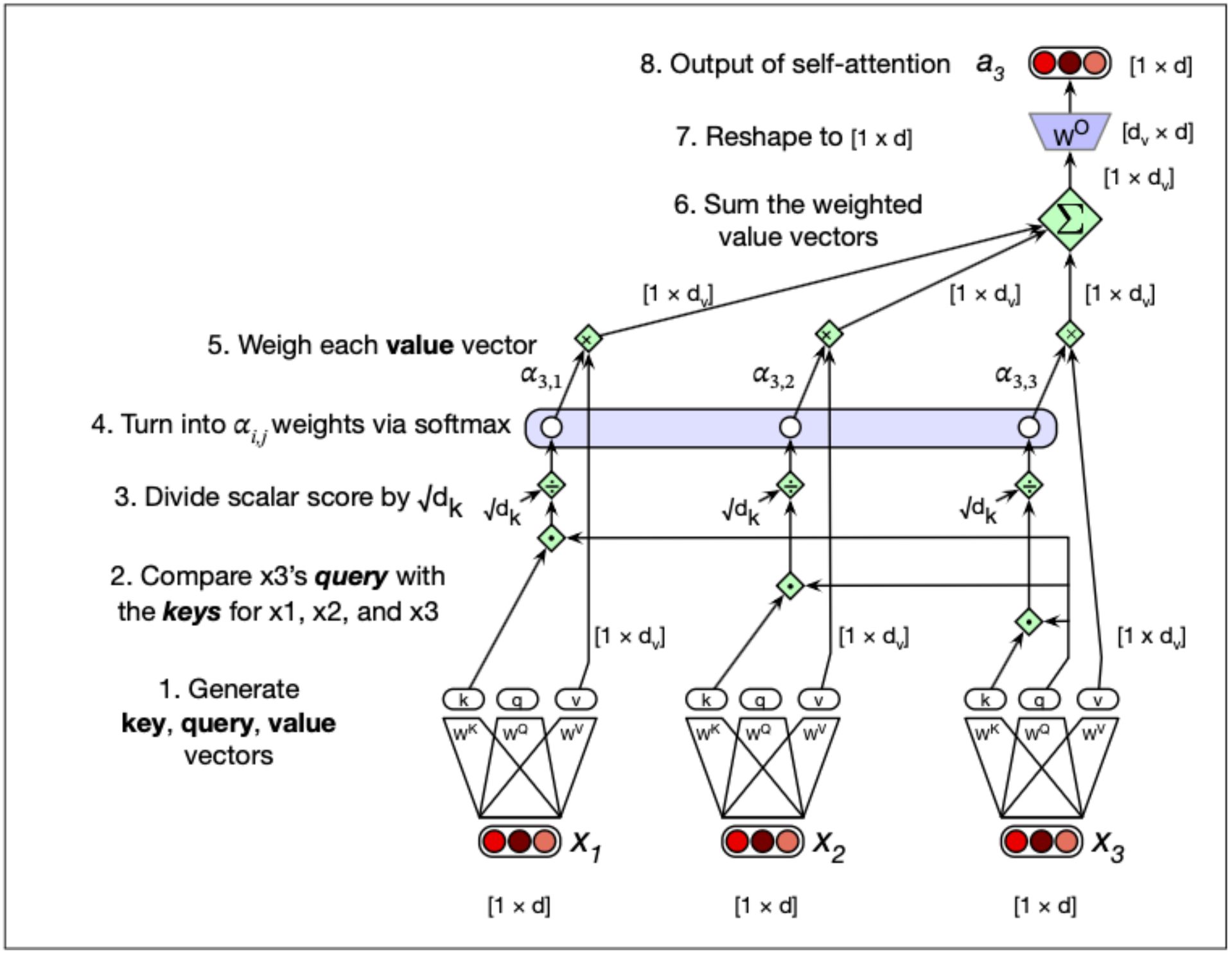
$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \forall i \leq j$$

$$\mathbf{head}_i = \sum_{j \leq i} \alpha_{ij}\mathbf{v}_j$$

$$\mathbf{a}_i = \mathbf{head}_i\mathbf{W}^O$$

$\mathbf{W}^O$ is used to reshape the vector

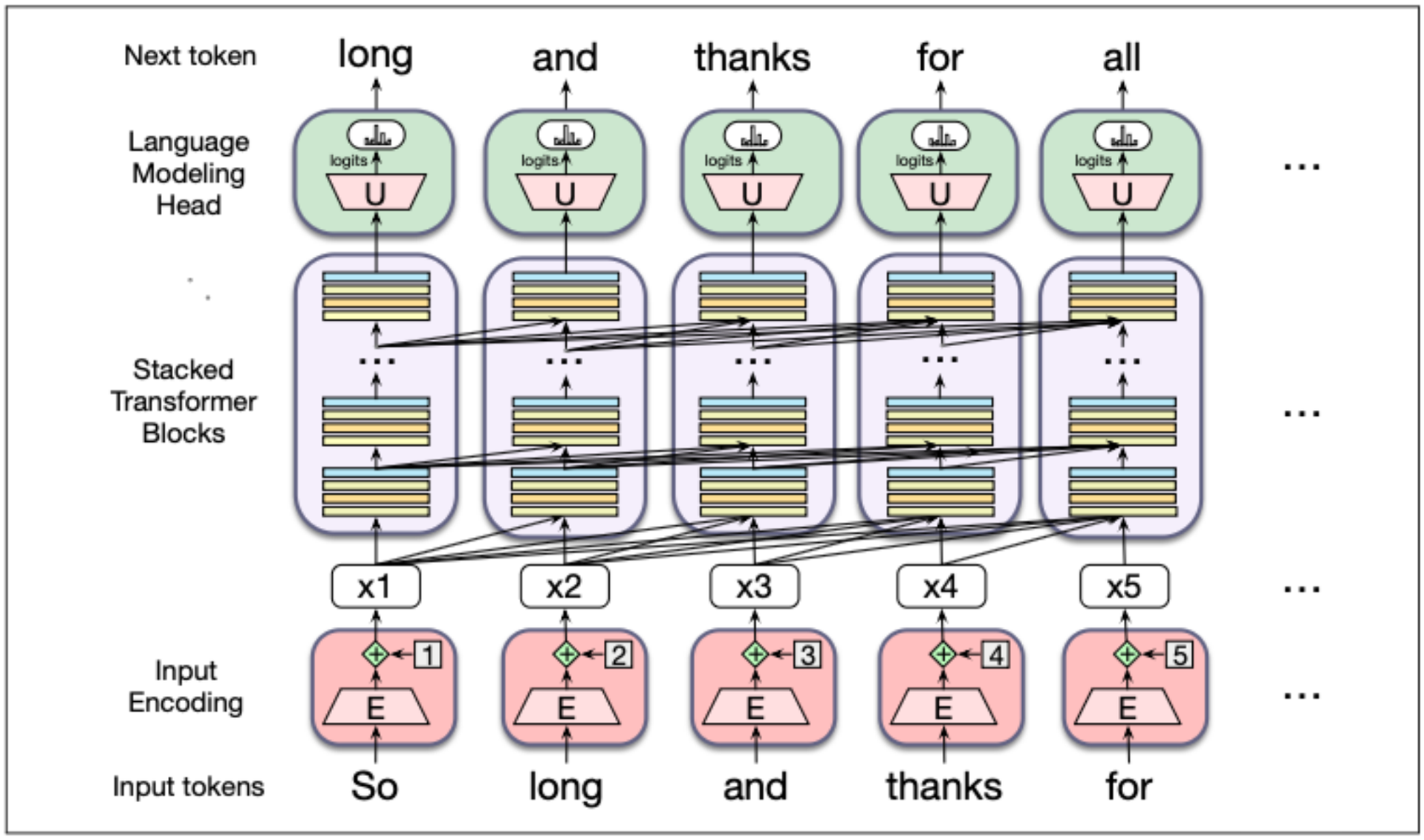


Speech and Language Processing, Jurafsky & Martin, Chapter 8

K, V Issue

To compute the token after all we submit the prompt

So long and thanks for all



KV cache

For each token in the context window, store Key and Value for every layer

$$Memory = 2 \times \text{Layers} \times d_{model} \times \text{Context Length} \times \text{Precision}$$

Llama 3 8B

- Layers: 32
- d_{model}: 4,096
- Precision: 2 bytes (BF16)

Memory requirements

- Model Weights: ~15GB
- KV cache: ~64 GB

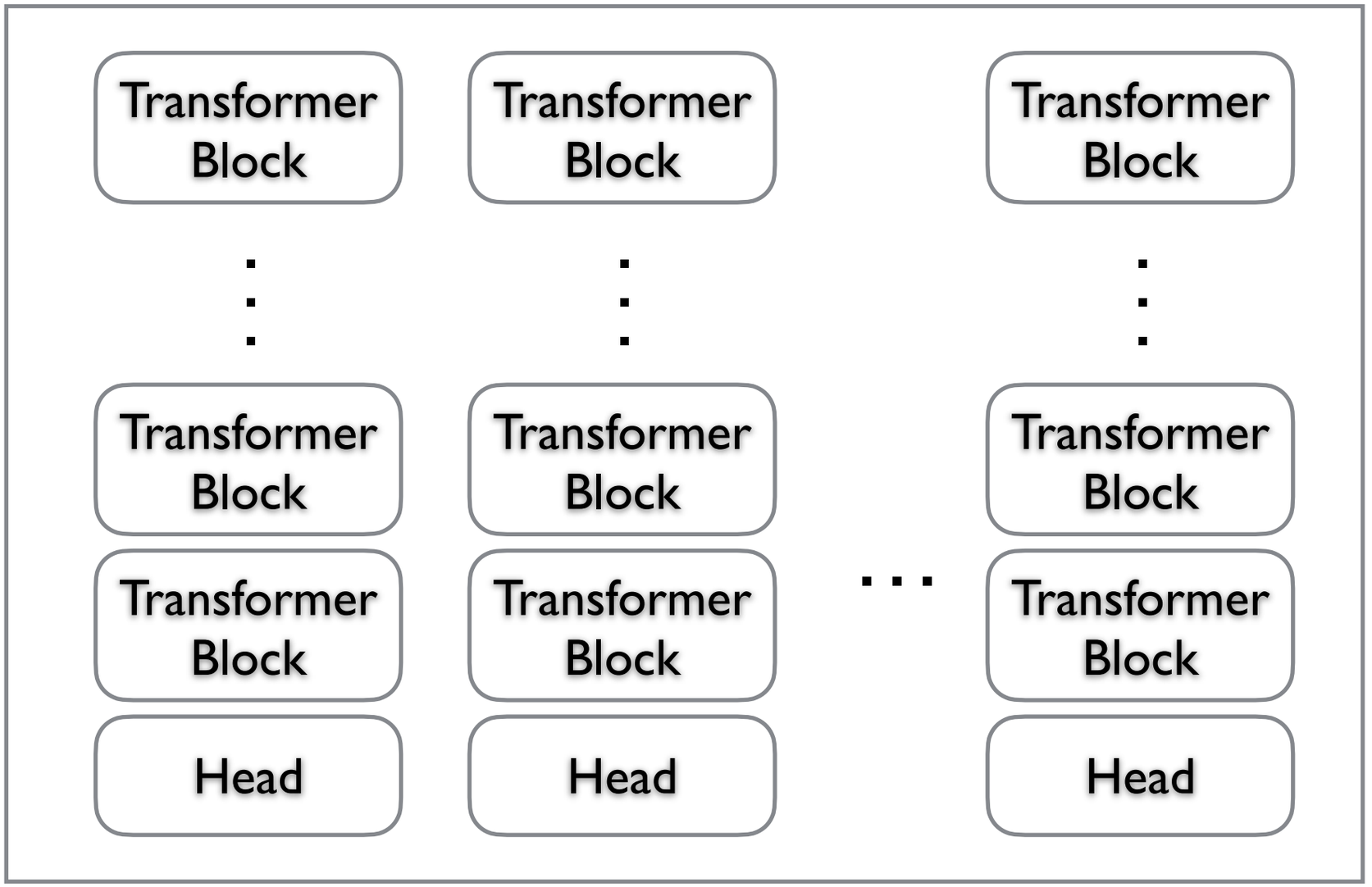
Creating the Cache

Stack all tokens in one matrix $X \in \mathbb{R}^{s \times d}$

Compute $K \quad V$ using parallel matrix multiplication

$$K = XW^K, \quad V = XW^V$$

Scaling


Transformer Block
Transformer Block
Transformer Block
Transformer Block
Transformer Block
Transformer Block
Transformer Block
Transformer Block
Transformer Block
Head
Head
Head

d_{model} Embedding dimension (size of token representation)

n_{heads} Number of attention heads

d_{head} Dimension of each head, (size of query, key, value vector)
Power of 2, 64, 128

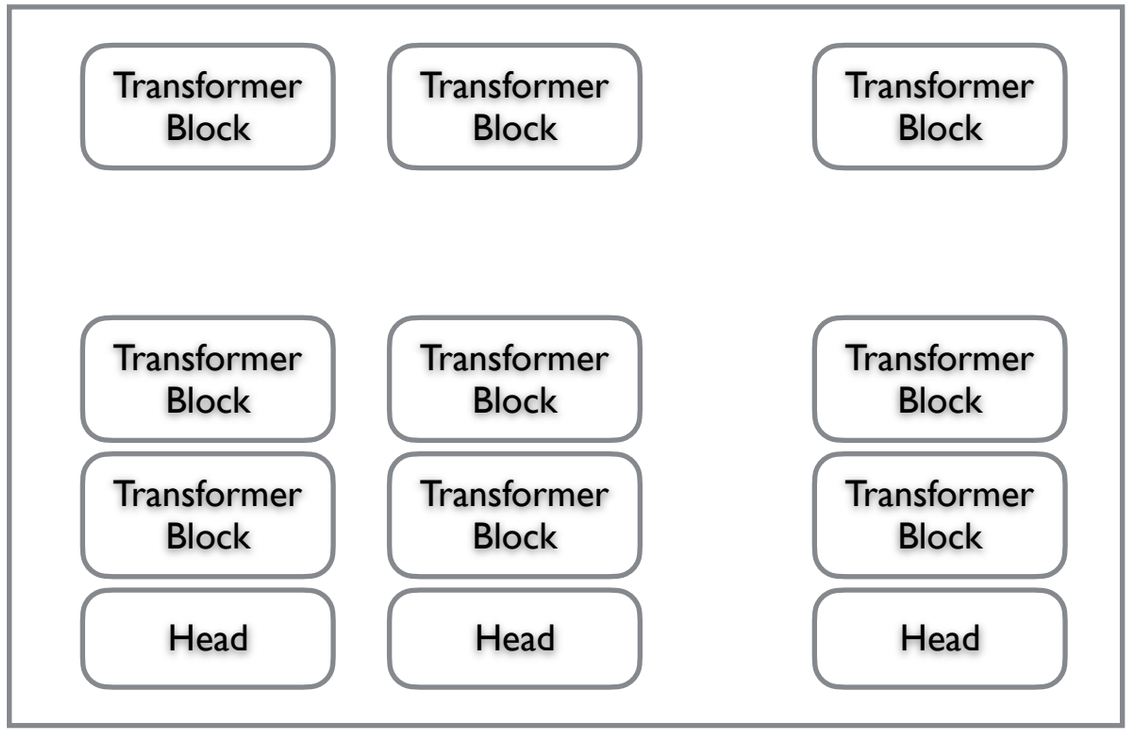
$$d_{model} = n_{heads} \times d_{head}$$

N Parameters

D Dataset size

C Compute

L Layers



Scaling Laws

N Parameters

D Dataset size

C Compute

Kaplan et al. 2020

$\text{Loss}(N) \propto N^{-\alpha}$

$\text{Loss}(D) \propto D^{-\beta}$

$C \sim N \times D$

$\alpha \approx 0.076$

$\beta \approx 0.095$

100B model train on ~200B tokens

Chinchilla (DeepMind) 2022

Found Kaplan undertrained models

$N \propto D$

$N \propto C^{1/2}$

$D \propto C^{1/2}$

$D \propto 20N$

100B model train on ~2T tokens

Depth vs Width Scaling

d_{model} Embedding dimension (size of token representation)

N Parameters

L Layers

$$N \sim L \times d_{model}^2$$

Width gives smoother scaling

Depth improves reasoning and compositionality

Loss & Scaling

$$\text{Loss} \propto L^{-\alpha} \cdot (n_{heads} d_{head})^{-2\alpha}$$
$$= L^{-\alpha} \cdot d_{model}^{-2\alpha}$$

Number of heads n_{heads}

Embedding size d_{model}

Head dimension d_{head}

Layers L

Loss decreases when you increase:

- Number of layers
- Embedding width, but quadratically

Loss & Scaling

Loss $\propto L^{-\alpha} \cdot (n_{heads} d_{head})^{-2\alpha}$

Number of heads n_{heads}

Embedding size d_{model}

Head dimension d_{head}

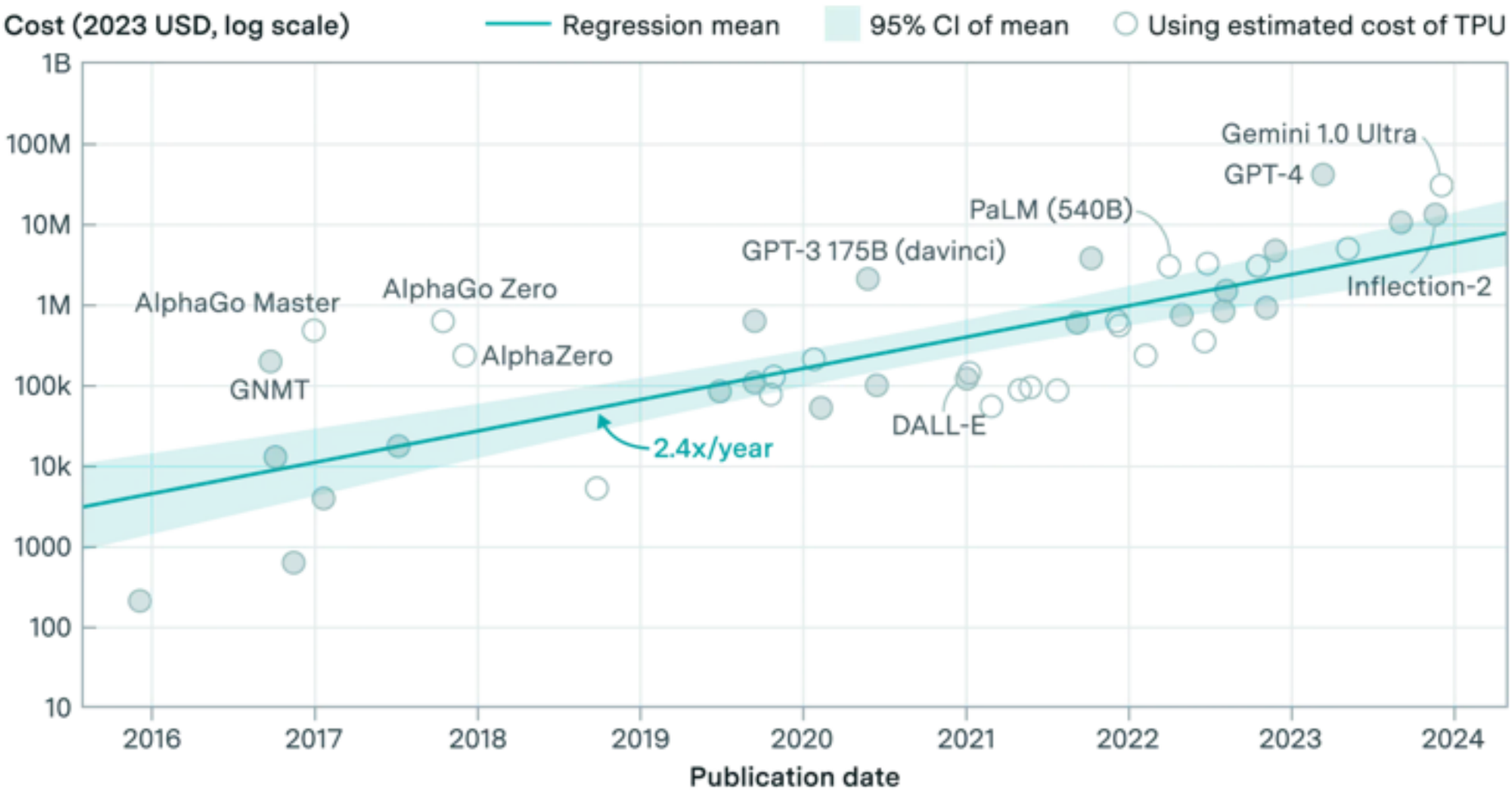
Layers L

Scaling laws don’t care directly about

- Number of heads
- Embedding size

d_{head} = 64 or 128

Amortized hardware and energy cost to train frontier AI models over time



https://andrewchen.substack.com/p/revenge-of-the-gpt-wrappers-defensibility

Cost of Training Chinchilla-style

N Parameters

D Dataset size

C Compute

$C_{train} \approx 6ND$ “6” comes from forward + backward pass accounting

$D \propto 20N$ Chinchilla

$C_{train} \approx 120N^2$

$C_{train} \propto N^2$

7B → 70B params

10x more params

100x more compute

Information & LLM size

Information grows roughly by N^{α} $\alpha \approx 0.07$ Kaplan

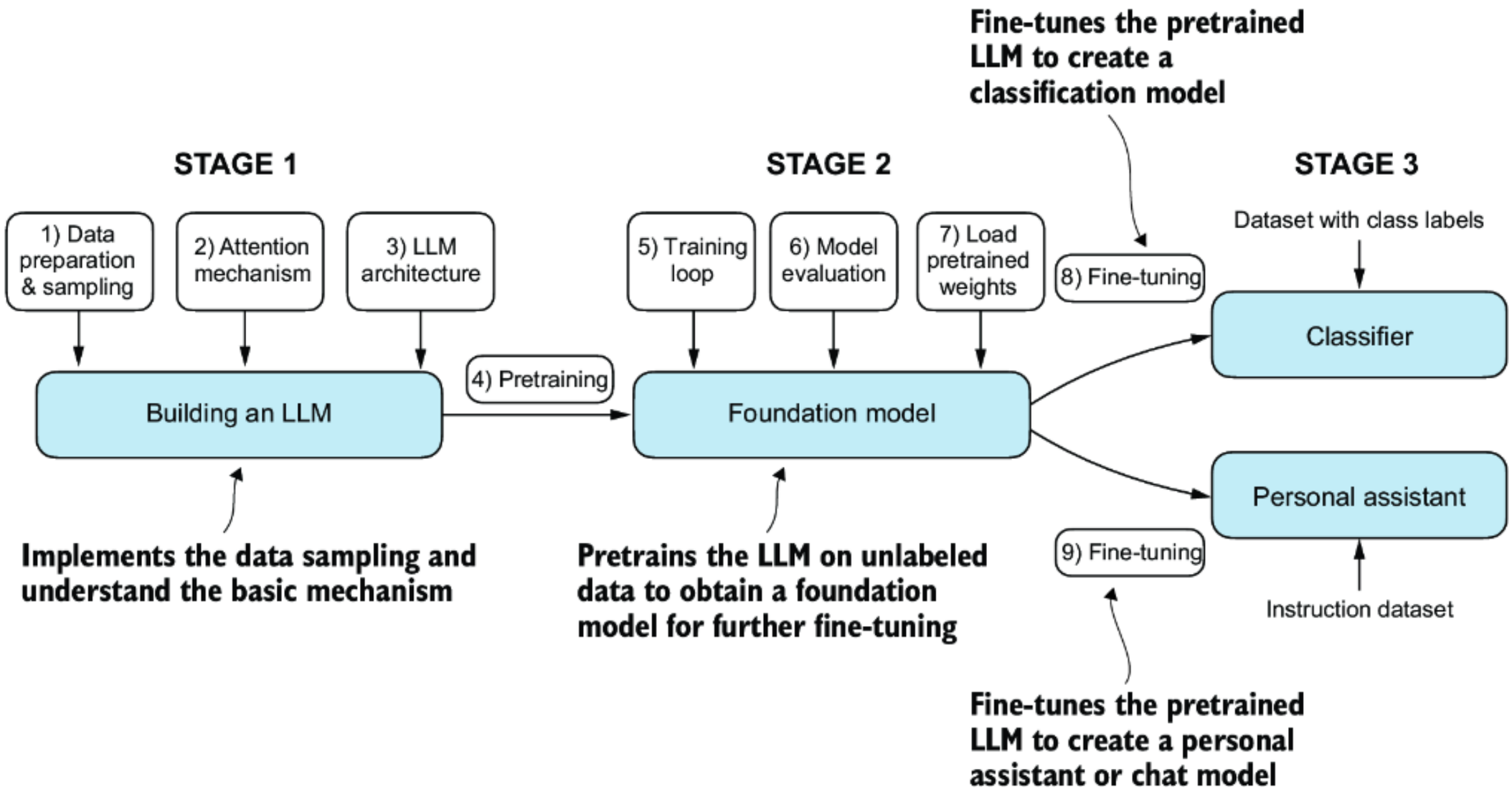
$N_1 = 10N$

$N_1^{\alpha} = (10N)^{\alpha} = 10^{\alpha}N^{a} = 1.17N^{\alpha}$

$N \sim L \times d_{model}^2$

Chinchilla - Optimal Traininh

Build a Large Language Model (From Scratch)



Build a Large Language Model (From Scratch)

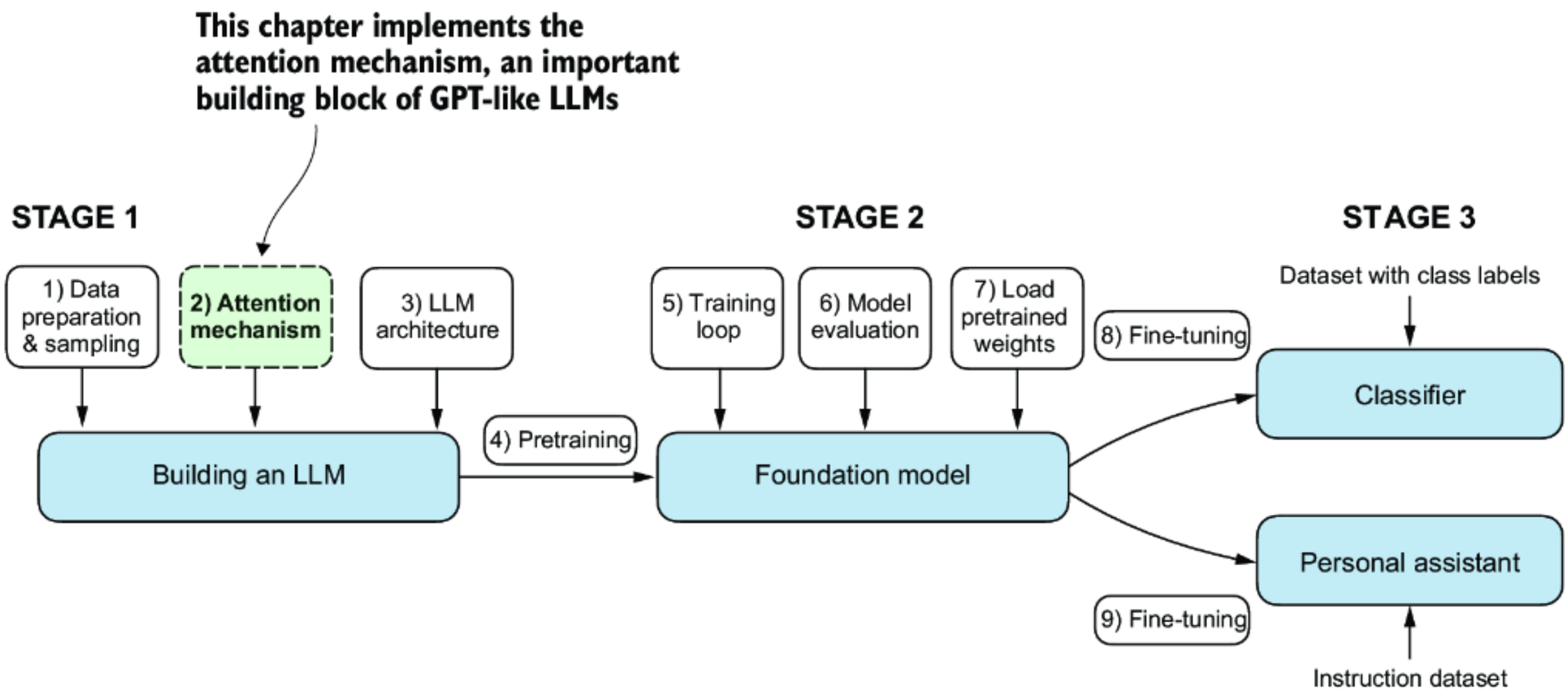
Source code from book

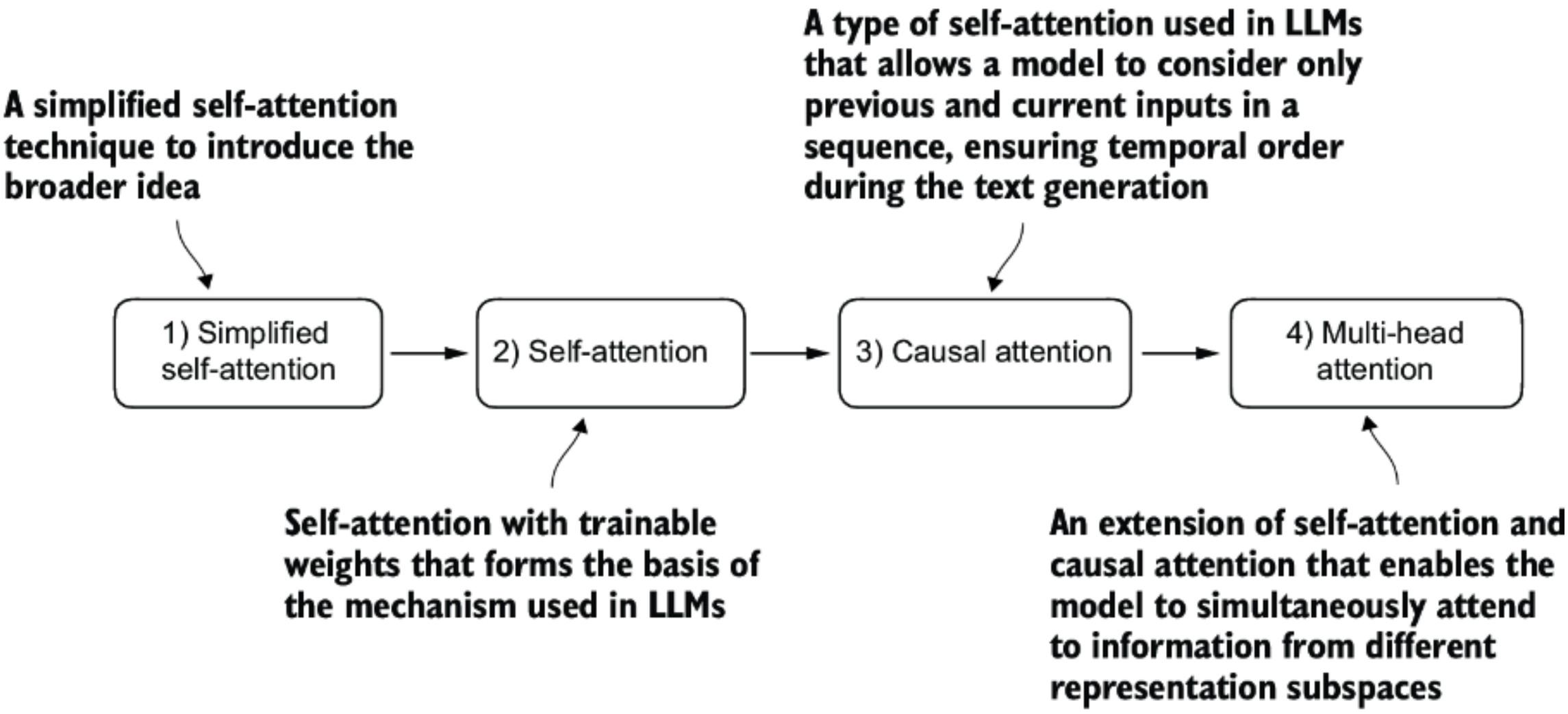
https://github.com/rasbt/LLMs-from-scratch

Tiny Aya From Scratch (A Standalone Notebook)

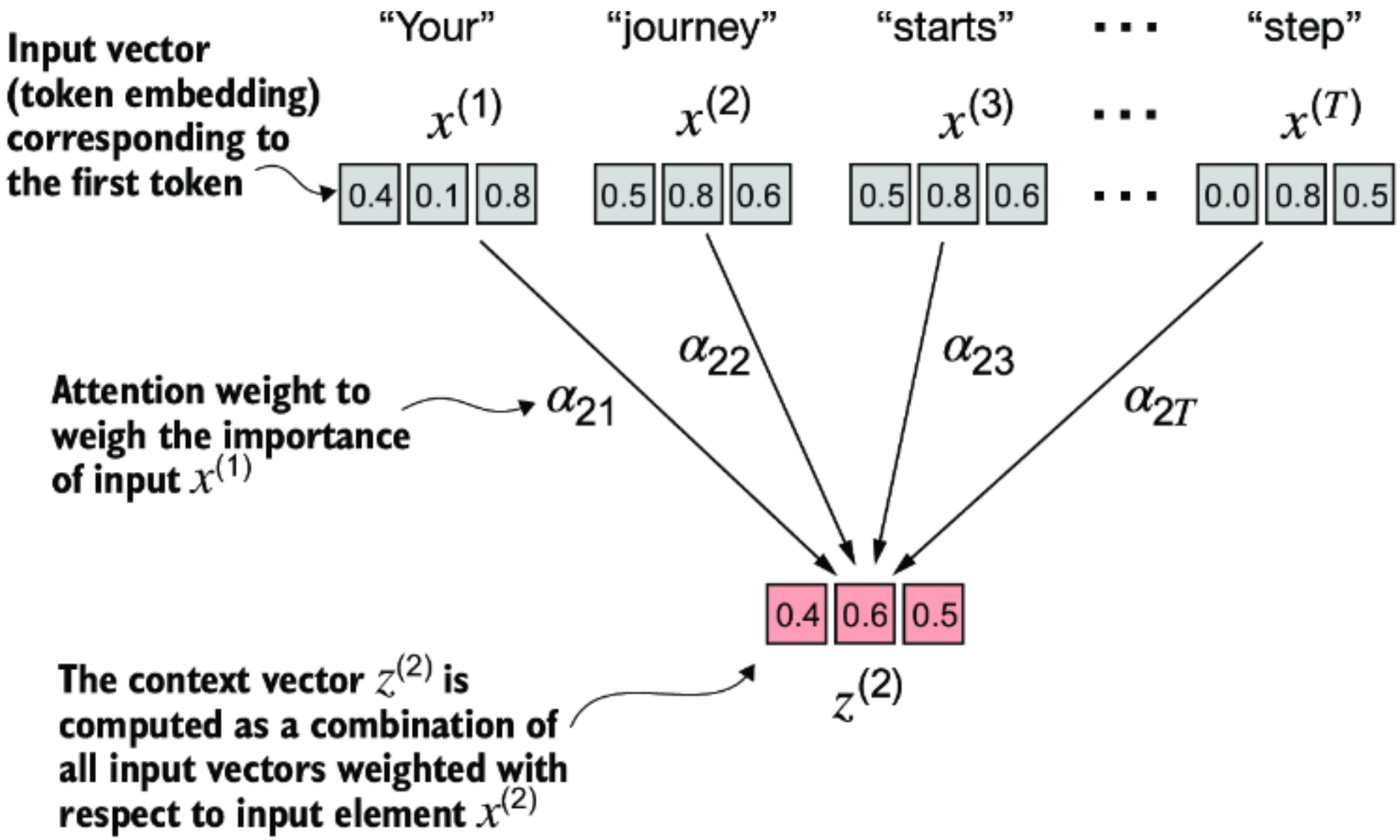
https://github.com/rasbt/LLMs-from-scratch/blob/main/ch05/15_tiny-aya/standalone-tiny-aya-plus-kv-cache.ipynb

Attention - Chapter 3




A simplified self-attention technique to introduce the broader idea
A type of self-attention used in LLMs that allows a model to consider only previous and current inputs in a sequence, ensuring temporal order during the text generation
1) Simplified self-attention
2) Self-attention
3) Causal attention
4) Multi-head attention
Self-attention with trainable weights that forms the basis of the mechanism used in LLMs
An extension of self-attention and causal attention that enables the model to simultaneously attend to information from different representation subspaces

Simplified Calculation



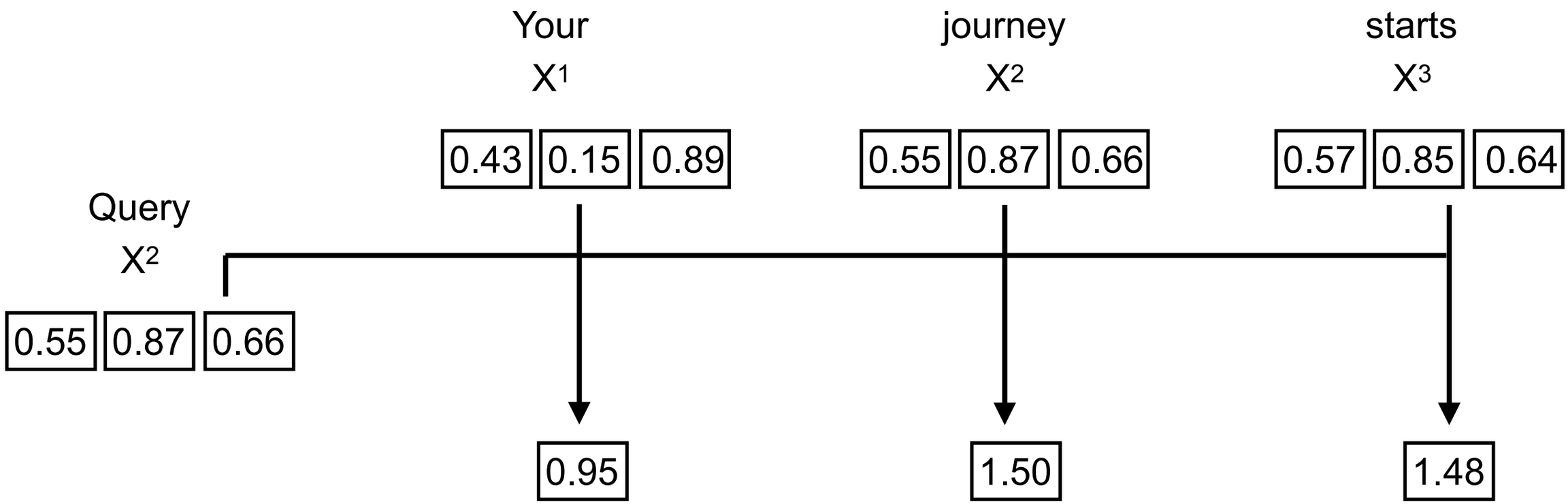
Example

Your journey starts with one step

Token Embeddings

```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

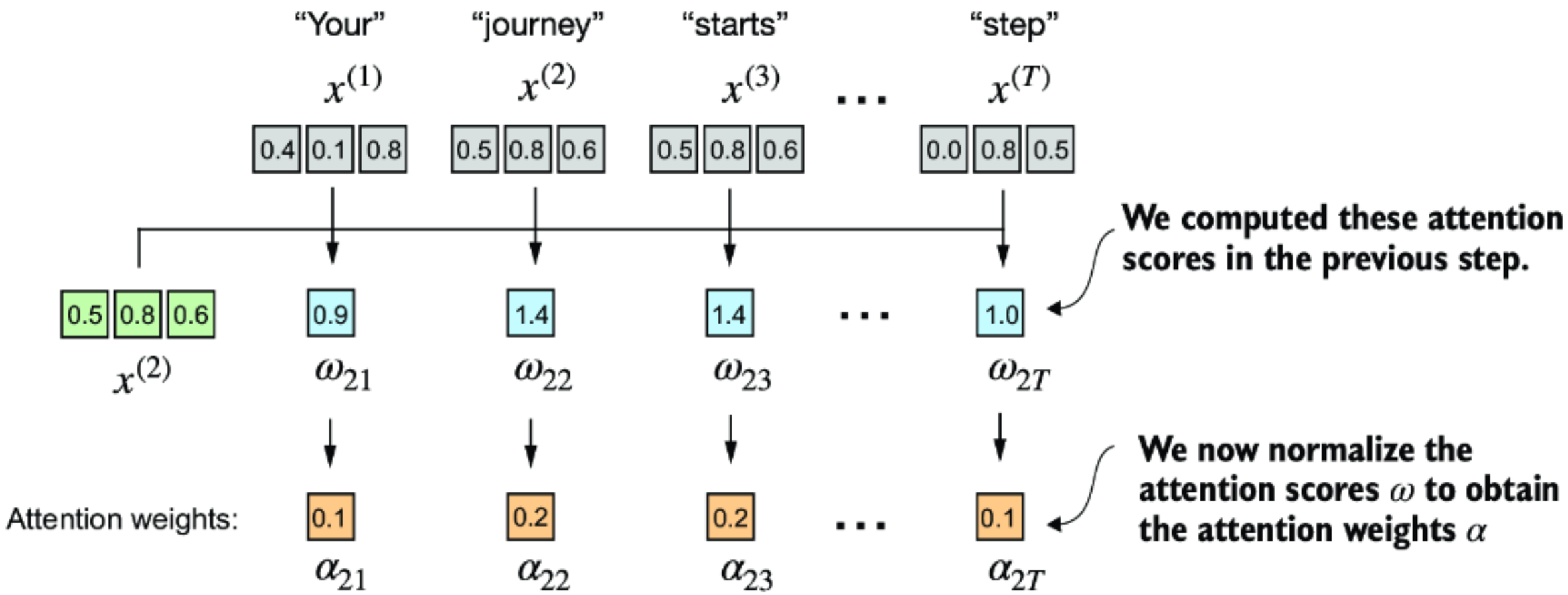
Simplified Calculation



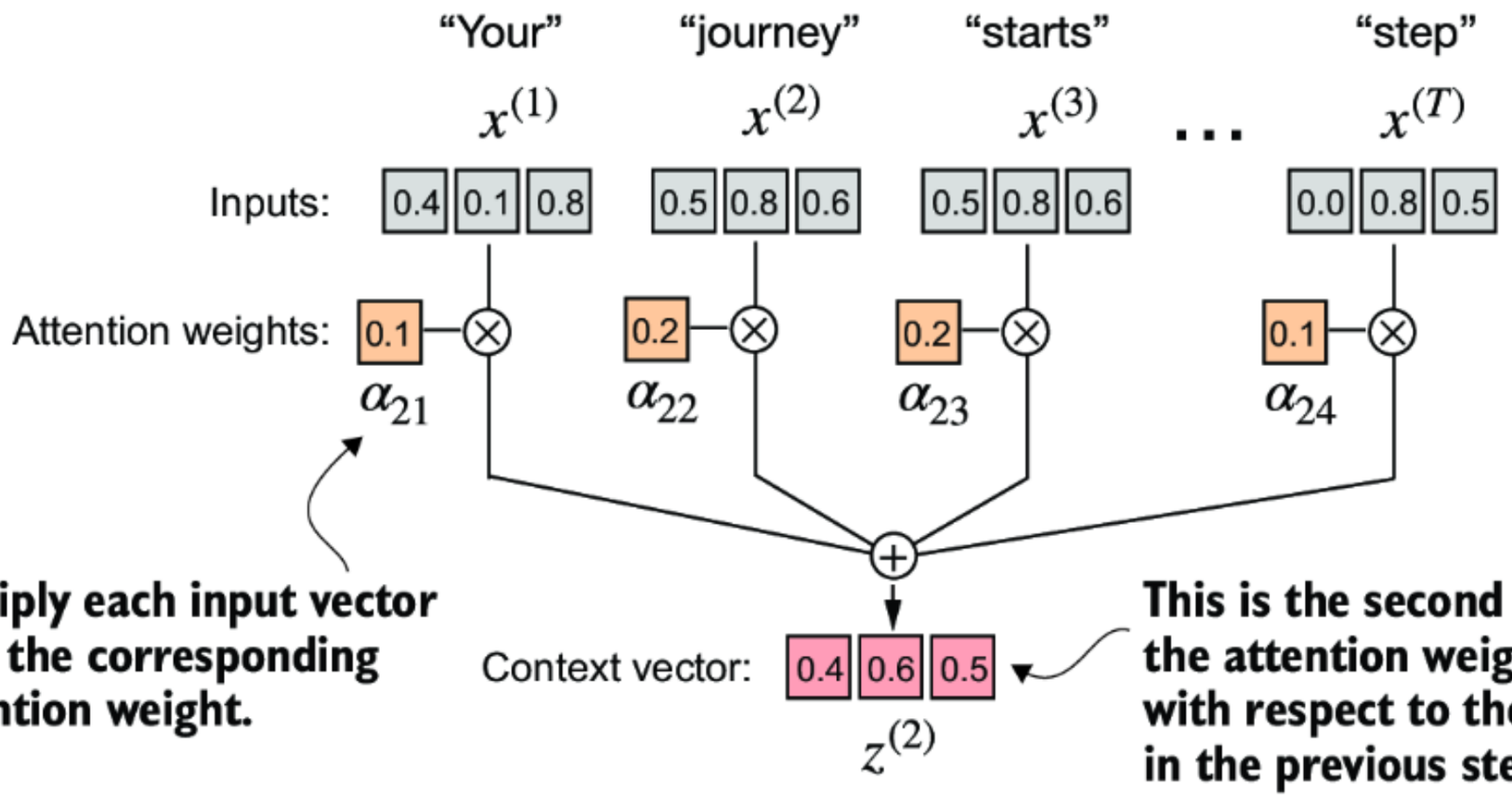
Attention score between
X^1 and X^2

$0.55*0.43 + 0.87*0.15 + 0.66*0.89 = 0.9544$

Simplified Calculation



Simplified Calculation



Multiply each input vector with the corresponding attention weight.

This is the second context vector because the attention weights were computed with respect to the second input vector in the previous steps.

Using Trained Weights

Query:

The element you want to understand in the context of the sequence.

Keys:

The elements you compare the query to.

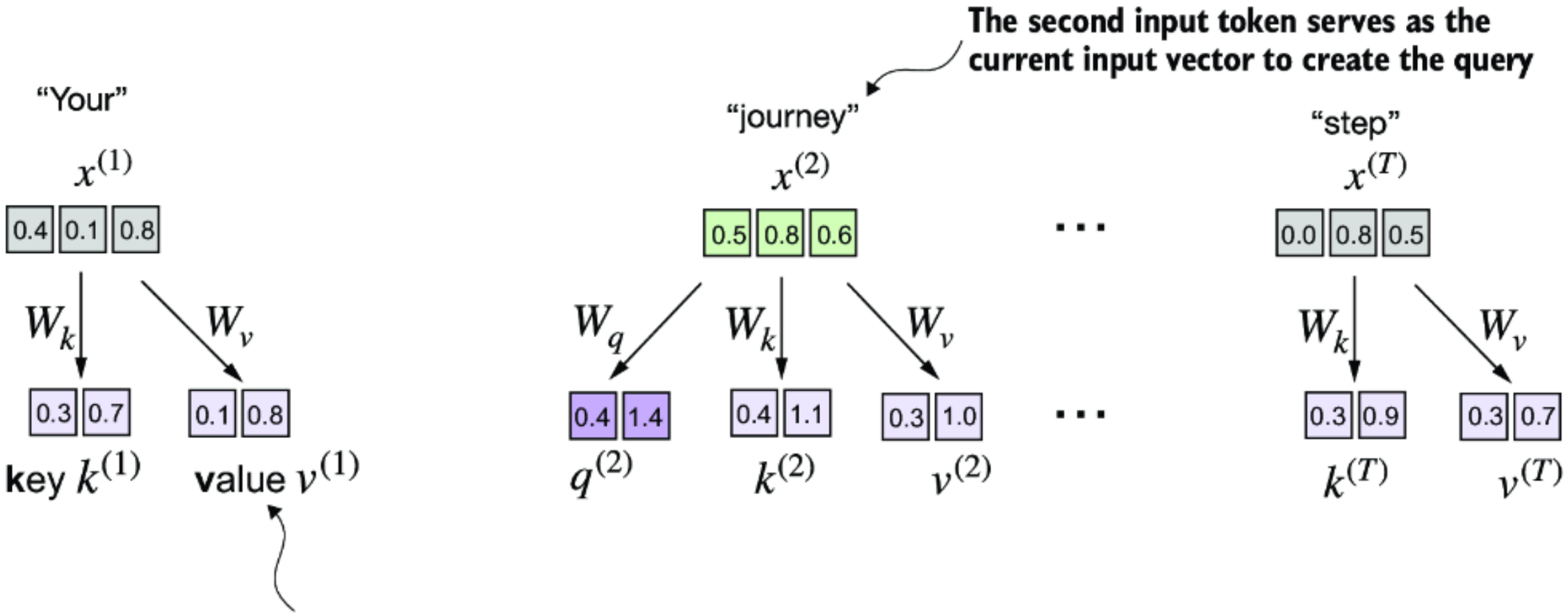
Values:

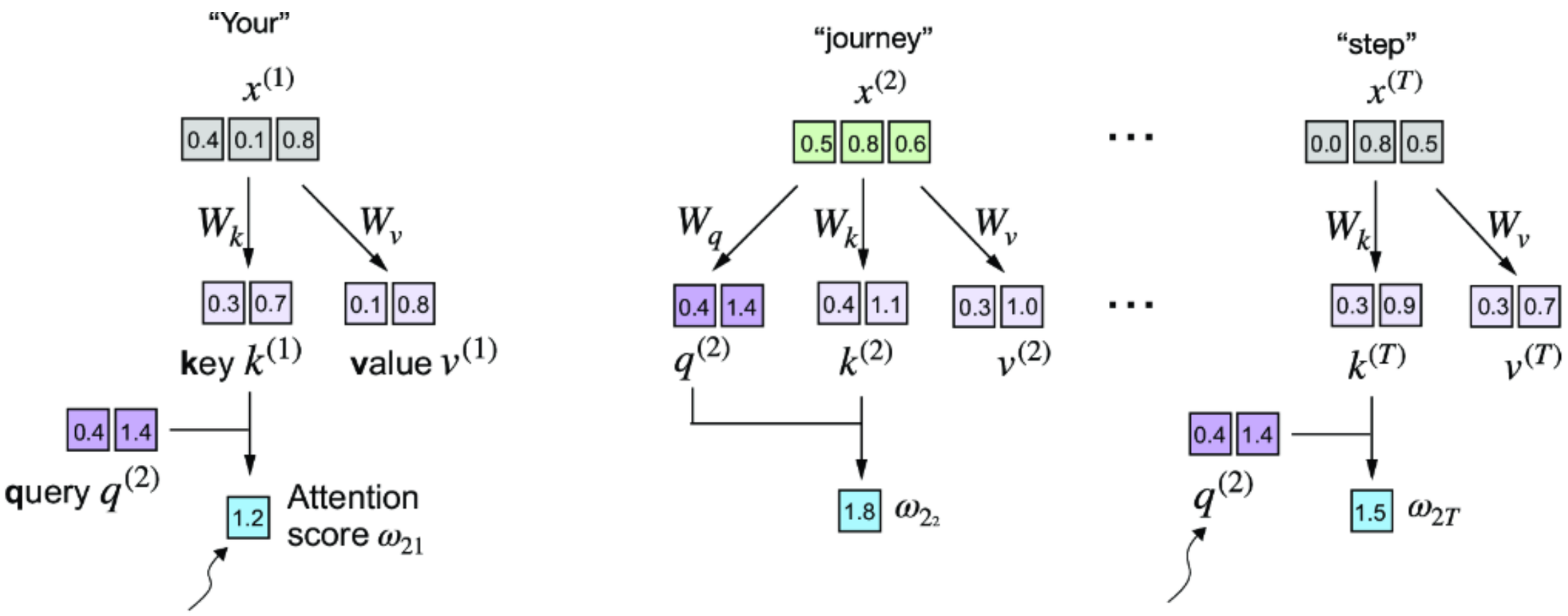
The information associated with each key.

Context Vector

How tokens are related to each other

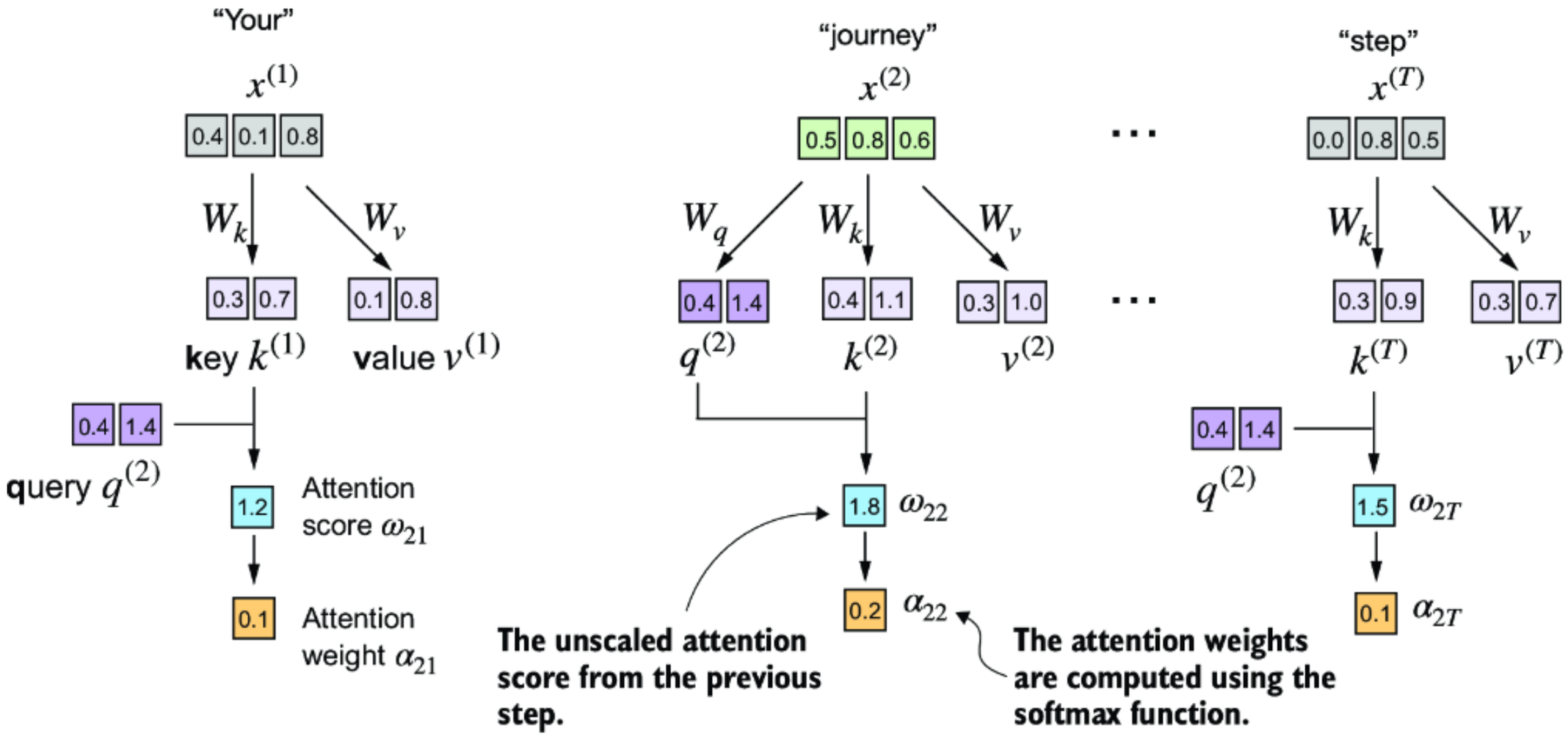
Combined with embedding to create a contextually aware representation of the token

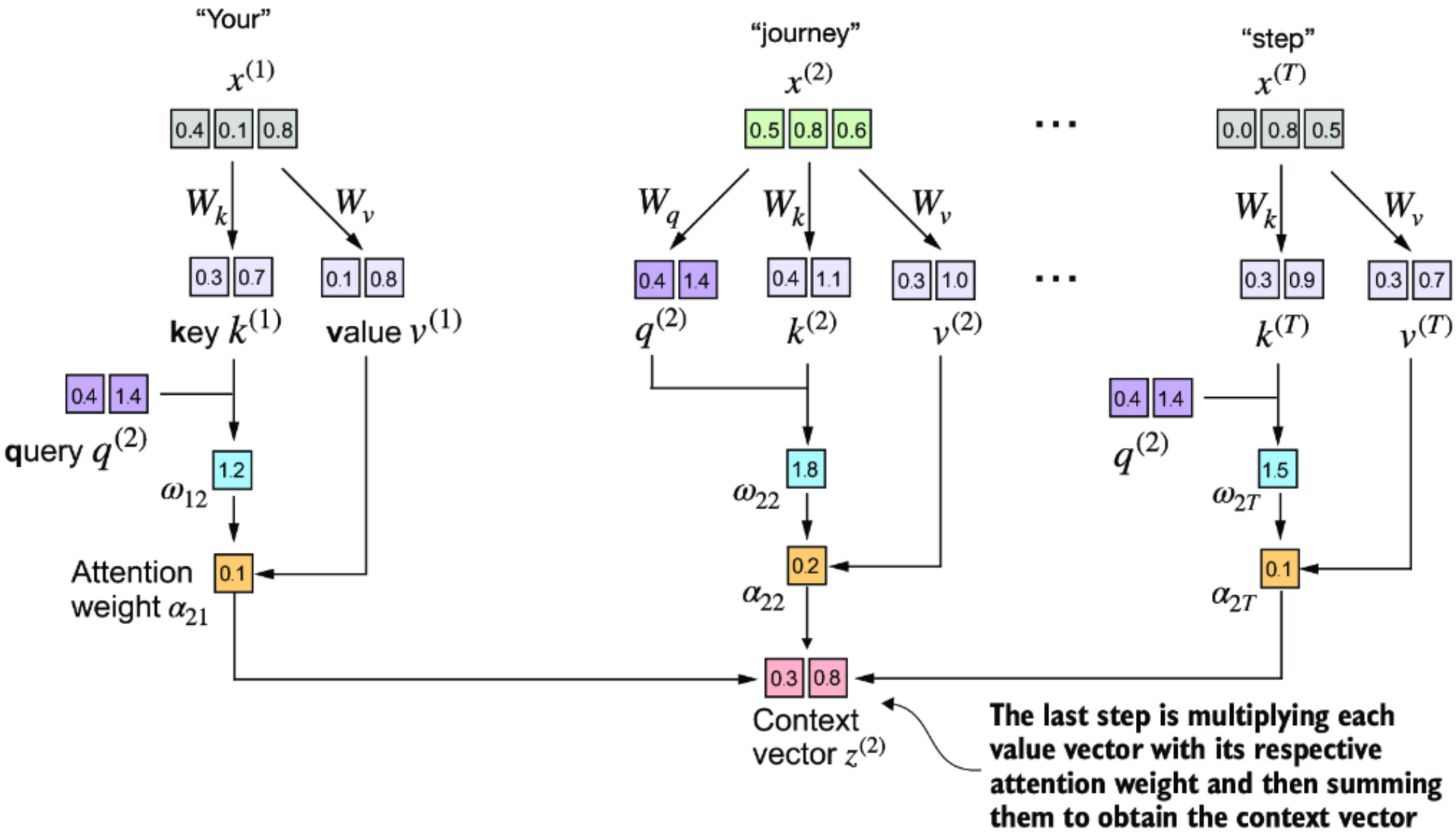

The second input token serves as the current input vector to create the query
"Your"
$x^{(1)}$
0.4 0.1 0.8
"journey"
$x^{(2)}$
0.5 0.8 0.6
"step"
$x^{(T)}$
0.0 0.8 0.5
W_k
W_v
W_q
W_k
W_v
W_k
W_v
0.3 0.7
0.1 0.8
0.4 1.4
0.4 1.1
0.3 1.0
0.3 0.9
0.3 0.7
key $k^{(1)}$
value $v^{(1)}$
$q^{(2)}$
$k^{(2)}$
$v^{(2)}$
$k^{(T)}$
$v^{(T)}$
This is the value vector corresponding to the first input token obtained via matrix multiplication between the weight matrix W_v and input token $x^{(1)}$



The unscaled attention score is computed as a dot product between the query and the key vectors.

Since we want to compute the context vector for the second input token, the query is derived from that second input token.


"Your"
$x^{(1)}$
0.4 0.1 0.8
W_k
W_v
0.3 0.7
0.1 0.8
key $k^{(1)}$
value $v^{(1)}$
0.4 1.4
query $q^{(2)}$
1.2
Attention score ω_{21}
0.1
Attention weight α_{21}
"journey"
$x^{(2)}$
0.5 0.8 0.6
W_q
W_k
W_v
0.4 1.4
0.4 1.1
0.3 1.0
$q^{(2)}$
$k^{(2)}$
$v^{(2)}$
1.8 ω_{22}
0.2 α_{22}
The unscaled attention score from the previous step.
The attention weights are computed using the softmax function.
"step"
$x^{(T)}$
0.0 0.8 0.5
W_k
W_v
0.3 0.9
0.3 0.7
$k^{(T)}$
$v^{(T)}$
0.4 1.4
$q^{(2)}$
1.5 ω_{2T}
0.1 α_{2T}


"Your"
$x^{(1)}$
0.4 0.1 0.8
W_k
W_v
0.3 0.7
0.1 0.8
key $k^{(1)}$
value $v^{(1)}$
0.4 1.4
query $q^{(2)}$
1.2
ω_{12}
Attention weight α_{21}
0.1
"journey"
$x^{(2)}$
0.5 0.8 0.6
W_q
W_k
W_v
0.4 1.4
0.4 1.1
0.3 1.0
$q^{(2)}$
$k^{(2)}$
$v^{(2)}$
1.8
ω_{22}
0.2
α_{22}
"step"
$x^{(T)}$
0.0 0.8 0.5
W_k
W_v
0.3 0.9
0.3 0.7
$k^{(T)}$
$v^{(T)}$
0.4 1.4
$q^{(2)}$
1.5
ω_{2T}
0.1
α_{2T}
0.3 0.8
Context vector $z^{(2)}$
The last step is multiplying each value vector with its respective attention weight and then summing them to obtain the context vector

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))

    def forward(self, x):
keys = x @ self.W_key
queries = x @ self.W_query
values = x @ self.W_value
attn_scores = queries @ keys.T
attn_weights = torch.softmax(
    attn_scores / keys.shape[-1]**0.5, dim=-1
)
context_vec = attn_weights @ values
return context_vec
```

```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

```
d_in=3
d_out=2

torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

```
tensor([[0.2996, 0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990, 0.8040]], grad_fn=<MmBackward0>)
```

How does sa_v1(inputs) work

torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(**sa_v1(inputs)**)

class torch.nn.Module(*args, **kwargs)

Base class for all neural network modules.

__call__ method calls forward

bfloat16()

Casts all floating point parameters and buffers to bfloat16 datatype

compile(*args, **kwargs)

Compile this Module's forward using torch.compile()

cpu()[source]

Move all model parameters and buffers to the CPU

cuda(device=None)

Move all model parameters and buffers to the GPU

torch.nn.Linear

torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)

$$y = xA^T + b\,.$$

```
import torch.nn as nn
import torch

m = nn.Linear(2, 3)
input = torch.randn(5, 2)
output = m(input)
```

forward(input)[source]

Runs the forward pass.

```
M
tensor([[-0.5203,  0.6365],
        [-0.5781,  0.4901],
        [-0.1238,  0.2112]], requires_grad=True)

Input
tensor([[-1.4525,  0.7343],
        [ 0.2939,  0.1127],
        [ 1.9151,  0.0319],
        [ 0.4905, -0.8721],
        [-0.5066,  2.0014]])

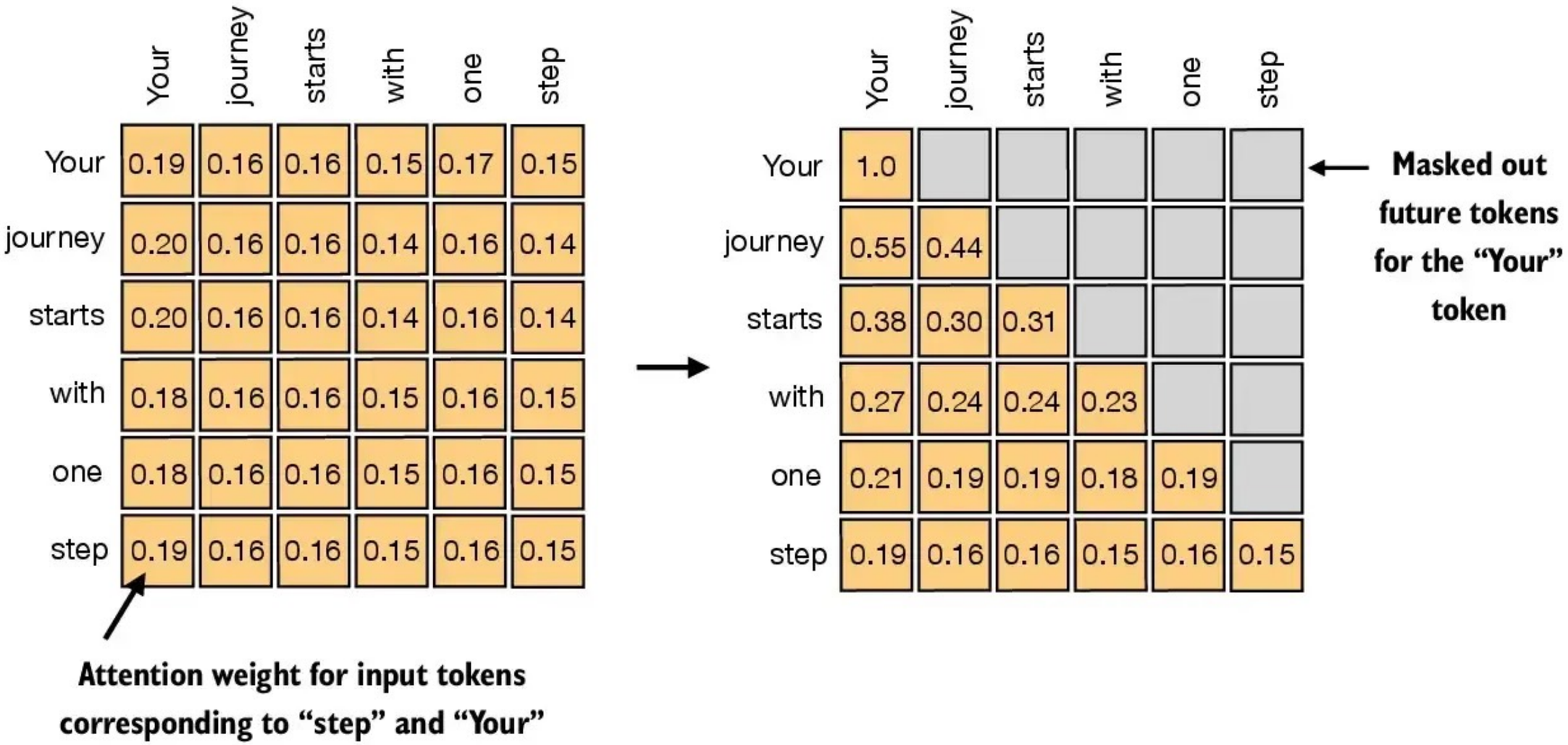
Output
tensor([[ 0.6132,  1.2340,  0.4194],
        [-0.6912, -0.0803,  0.0720],
        [-1.5862, -1.0571, -0.1458],
        [-1.4203, -0.6767, -0.1603],
        [ 0.9274,  1.3082,  0.5699]], grad_fn=<AddmmBackward0>)
```

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
keys = self.W_key(x)
queries = self.W_query(x)
values = self.W_value(x)
attn_scores = queries @ keys.T
attn_weights = torch.softmax(
    attn_scores / keys.shape[-1]**0.5, dim=-1
)
context_vec = attn_weights @ values
return context_vec
```

Hiding Future Words

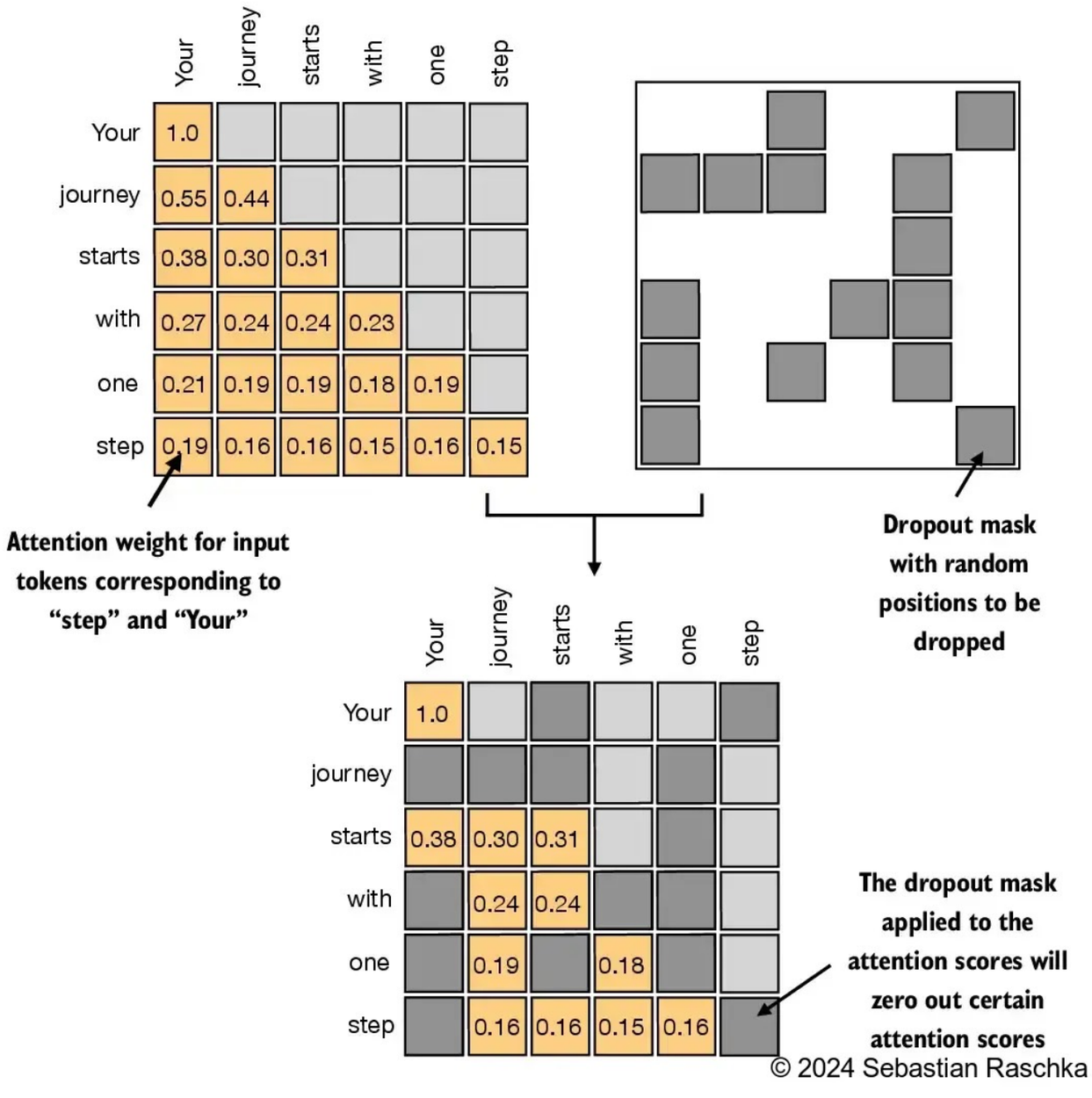
To predict the next word, mask the future words



© 2024 Sebastian Raschka

Masking Random Attention Weights

Used to reduce overfitting



Adding Dropouts & Mask

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
          'mask',
          torch.triu(torch.ones(context_length, context_length),
          diagonal=1)
        )
```

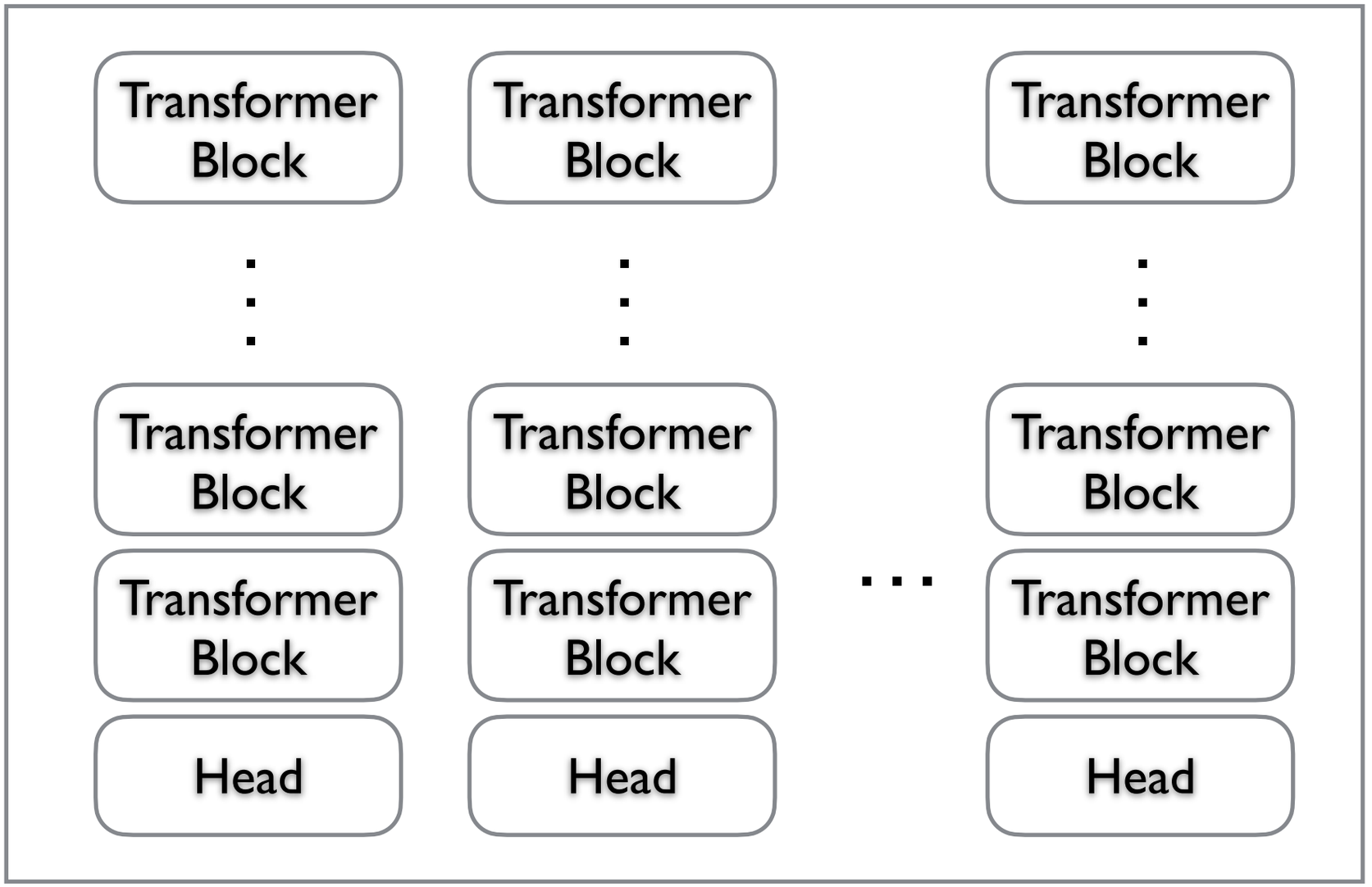
Adding Dropouts & Mask

```
def forward(self, x):
    b, num_tokens, d_in = x.shape
    keys = self.W_key(x)
    queries = self.W_query(x)
    values = self.W_value(x)

    attn_scores = queries @ keys.transpose(1, 2)
attn_scores.masked_fill_(
    self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
attn_weights = torch.softmax(
    attn_scores / keys.shape[-1]**0.5, dim=-1
)
attn_weights = self.dropout(attn_weights)

context_vec = attn_weights @ values
return context_vec
```

Multiple Heads



Multiple Heads

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention( d_in, d_out, context_length, dropout, qkv_bias )
             for _ in range(num_heads)]
        )

    def forward(self, x):
        return torch.cat([head(x) for head in self.heads], dim=-1)
```