CS 696 Applied Large Language Models
Spring Semester, 2025
Doc 9 GPU Cluster, Attention, GPT Model
Feb 11, 2025

Copyright ©, All rights reserved. 2025 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.

References

My LLM's outputs got 1000% better with this simple trick. Nikhil Anand
https://ai.gopubby.com/my-llms-outputs-got-1000-better-with-this-simple-trick-8403cf58691c

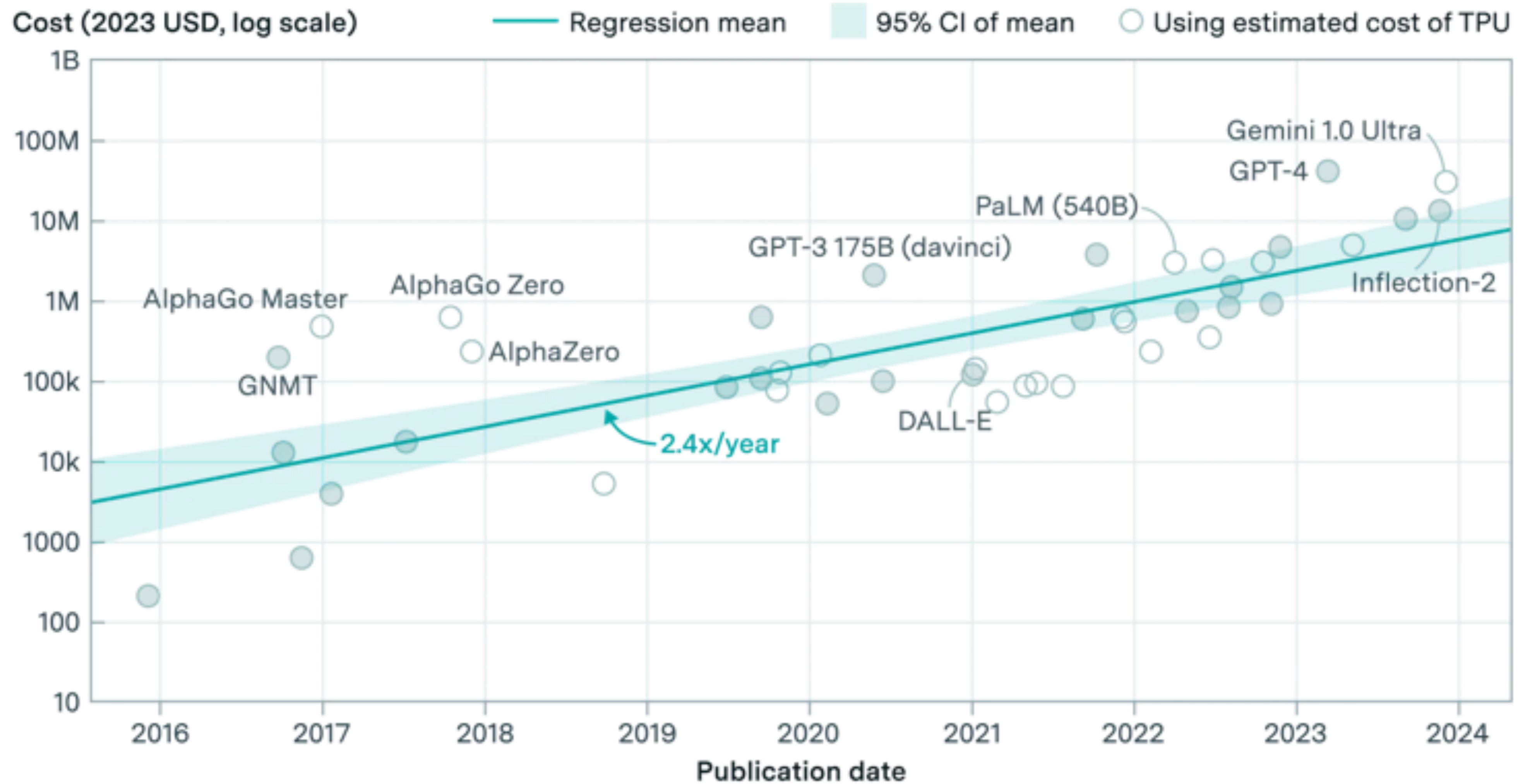
Building a Large Language Model (from Scratch), Sebastian Raschka

Hands on Large Language Models, Jay Alammar and Maarten Grootendorst

Gemini Pro

DeepSeek

PyTorch Documentation
https://pytorch.org/docs/stable/torch.html



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

Githup Model Playground

https://docs.github.com/en/github-models/prototyping-with-ai-models

github.com/marketplace/models.

Chat | Code | Raw

Input: 17/17 • Output: 776/776 • 22101ms

rogerwhitney now

Explain how Torch Model class works

Llama-3.2-11B-Vision-Instruct now

Torch Model Class Overview

The `torch.nn.Module` class is the foundation of PyTorch's neural network architecture. It serves as a base class for all neural network models, providing a standardized way to define and train complex models. In this explanation, we'll delve into the inner workings of the `torch.nn.Module` class and its key components.

Key Components

`__init__` Method

Type your prompt...

Parameters | Details

Max Tokens

2048

Limit the maximum output tokens for the model response.

Temperature

0.8

Controls randomness in the response, use lower to be more deterministic.

Top P

0.1

Controls text diversity by selecting the most probable words until a set probability is reached.

Presence Penalty

pip install transformers --user

Puts transformers in your permanent storage

- .local

You don’t need to do it again

cache & Disk space

I ran out of disk space on the cluster

Cache did not remove any files

jovyan@jupyter-rwhitney-40sdsu-2eedu:~$ **ls .cache/huggingface/hub/**

```
datasets--yelp_review_full       models--facebook--bart-large-cnn
models--bert-base-uncased        models--microsoft--Phi-3-mini-4k-ins
models--distilbert-base-uncased  version.txt
```

```
jovyan@jupyter-rwhitney-40sdsu-2eedu:~$ cd .cache/huggingface/hub/
jovyan@jupyter-rwhitney-40sdsu-2eedu:~/.cache/huggingface/hub$ du .

…
2664    ./models--facebook--bart-large-cnn

430820  ./models--bert-base-uncased/blobs
…
430824  ./models--bert-base-uncased
…
7465588 ./models--microsoft--Phi-3-mini-4k-instruct/blobs
…
7465596 ./models--microsoft--Phi-3-mini-4k-instruct
7899104 .
```

Configuring Models

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,     # Vocabulary size
    "context_length": 1024,  # Context length
    "emb_dim": 768,          # Embedding dimension
    "n_heads": 12,           # Number of attention heads
    "n_layers": 12,          # Number of layers
    "drop_rate": 0.1,        # Dropout rate
    "qkv_bias": False        # Query-Key-Value bias
}
```

BertConfig

```
from transformers import BertTokenizer, BertModel
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertModel.from_pretrained(model_name)
```

```
BertConfig {
  "_attn_implementation_autoset": true,
  "_name_or_path": "bert-base-uncased",
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  “num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.48.2",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```

MistralConfig

MistralConfig {
"_attn_implementation_autoset": true,
"_name_or_path": "mistralai/Mistral-Small-24B-Instruct-2501",
"architectures": [
"MistralForCausalLM"
],
"attention_dropout": 0.0,
"bos_token_id": 1,
"eos_token_id": 2,
"head_dim": 128,
"hidden_act": "silu",
"hidden_size": 5120,
"initializer_range": 0.02,
"intermediate_size": 32768,
"max_position_embeddings": 32768,
"model_type": "mistral",
"num_attention_heads": 32,
"num_hidden_layers": 40,
“num_key_value_heads": 8,
"rms_norm_eps": 1e-05,
"rope_theta": 100000000.0,
"sliding_window": null,
"tie_word_embeddings": false,
"torch_dtype": "bfloat16",
"transformers_version": "4.48.3",
"use_cache": true,
"vocab_size": 131072

transformers.PretrainedConfig

model_type: (String)

vocab_size: (Integer)

hidden_size: (Integer)

num_hidden_layers: (Integer)

num_attention_heads: (Integer)

intermediate_size: (Integer)

hidden_act: (String) hidden layer activation function (e.g., "gelu", “relu")

hidden_dropout_prob: (Float)

attribute_map (Dict[str, str])
model specific attribute names -> standardized attribute namings

transformers.PretrainedConfig

from_pretrained(pretrained_model_name_or_path, **kwargs)

save_pretrained(save_directory)

to_dict()

update(**kwargs)

copy()

update_from_string(String)

Saving the Config

```
from transformers import AutoConfig, BertConfig

config = BertConfig.from_pretrained("bert-base-uncased")

print(config.vocab_size)
print(config.hidden_size)

config.hidden_dropout_prob = 0.2  #Some configs don’t allow this

config.save_pretrained("./my_bert_config")

config = BertConfig.from_pretrained("./my_bert_config")
```

Downloading a Model

```
from transformers import AutoModelForCausalLM, AutoTokenizer

access_token = “xxx"

model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-Small-24B-Instruct-2501",
    token=access_token,
    device_map="auto",
    attn_implementation='eager',
    torch_dtype="auto",
    trust_remote_code=True,
)
```

Generate vs Pipeline

Pipeline

Convenience method

Mig (**M**ultiple **I**ndependent **G**PU) bug in Linux transformer stack

Generate

Called by pipeline

Lower level

Run this to set mig_uuids

```
import os
import sys
import torch
import subprocess
import re

def get_mig_uuids():
    result = subprocess.run(['nvidia-smi', '-L'], stdout=subprocess.PIPE, text=True)
    if result.returncode != 0:
        raise RuntimeError(f"Command 'nvidia-smi -L' failed with exit code {result.returncode}")
    output = result.stdout

    mig_uuid_pattern = re.compile(r'MIG-[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}')
    mig_uuids = mig_uuid_pattern.findall(output)
    return mig_uuids

def set_cuda_visible_devices(mig_uuids):
    mig_uuids_str = ','.join(mig_uuids)
    os.environ['CUDA_VISIBLE_DEVICES'] = mig_uuids_str
    print(f"CUDA_VISIBLE_DEVICES set to: {mig_uuids_str}")

mig_uuids = get_mig_uuids()
if mig_uuids:
    set_cuda_visible_devices(mig_uuids)
else:
    print("No MIG devices found.")
```

Generate Example

```
from transformers import AutoModelForCausalLM, AutoTokenizer

access_token = "XXX"

model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-Small-24B-Instruct-2501",
    token=access_token,
    attn_implementation='eager',
    torch_dtype="auto",
    trust_remote_code=True,
)
```

Generate Example

```
prompt = "Once upon a time, in a land far, far away,"

inputs = tokenizer(prompt, return_tensors="pt")  # "pt" for PyTorch tensors
attention_mask = inputs.attention_mask
#input_ids = inputs.input_ids.to('cuda') #only when on Nvidia GPU

outputs = model.generate(
    inputs.input_ids,
    attention_mask=attention_mask,
    max_new_tokens=200,
    min_new_tokens=100,
    temperature=0.7,  # Adjust for creativity (higher = more creative)
    top_p=0.9,
    num_return_sequences=1 # Number of generated sequences
)

generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(generated_text)
```

Generate Arguments

input_ids: (Required) The tokenized input prompt.

attention_mask: Optional

A mask indicating which tokens are real input and which are padding tokens (used for batching sequences of different lengths). The tokenizer creates this automatically when you use padding=True.

inputs_embeds: (Optional)

Instead of input_ids, you can directly provide the embeddings of the input sequence

max_new_tokens:

The maximum number of new tokens to generate (excluding the prompt).

min_new_tokens:

The minimum number of new tokens to generate (excluding the prompt).

Generate Arguments

temperature:

Controls the randomness of the generation.

Lower values (e.g., 0.2) make the output more deterministic,

higher values (e.g., 0.8) make it more creative.

top_k:

Samples from the top K most likely words at each step. Helps to control randomness.

top_p (nucleus sampling):

Samples from the smallest set of most probable words whose cumulative probability exceeds top_p. Another way to control randomness and improve coherence.

num_return_sequences:

Generates multiple different sequences from the same prompt.

Useful for exploring possibilities

Temperature 0.2

Mistral-7B-Instruct-v0.3-4bit on laptop

prompt = "Once upon a time,"

in a far-off land, there lived a humble blacksmith named Elias. He was known throughout the kingdom for his exceptional craftsmanship, and his creations were sought after by kings and commoners alike. However, Elias was not content with his life. He yearned for adventure and to see the world beyond his small village.

One day, a mysterious traveler arrived in the village. He was a tall, bearded man with a cloak that seemed to hide his true form. He approached Elias and asked him to forge a sword of unparalleled power. Elias, intrigued by the stranger's request, agreed to take on the task.

For weeks, Elias labored over the sword, pouring all of his skill and knowledge into it. As he worked, he couldn't help but feel that there was something strange about the sword. It seemed to glow with an inner light, and it was heavier than any sword he had ever made.

Finally, the day came when the sword was finished. Elias presented it to the mysterious traveler, who took it with a nod of approval. As the traveler turned to leave, he paused and looked

Temperature 1.2

Mistral-7B-Instruct-v0.3-4bit on laptop

prompt = "Once upon a time,"

in a world much like our own, there was a tiny village nestled within a dense forest. This village, called Halcyon, was home to a community of clever and industrious people who made their living through agriculture, craftwork, and the sale of various wares.

Bordered by towering trees and muted by a soft, velvety canopy, the village's cobblestone streets were immaculately kept and lined with red-roofed cottages and workshop sheds. Introduction written, would you like me to continue the story or write about something specific related to it, like the characters, setting, or plot?

If the latter, you can ask more specific questions like:

- Can you give me more details about the village and the villagers of Halcyon?
- What unique characteristics does Halcyon have compared to other villages?
- Is there a protagonist in this story or main characters to focus on? If so, what are their qualities and motivations?
- Can you provide an overview of the plot or propose potential directions for the story to take? Of course, feel free to refine and adjust the questions according to your needs!

Temperature 0.0

Always get the same output

prompt = "Once upon a time,"

in a far-off land, there lived a young girl named Elara. She was known throughout the kingdom for her extraordinary beauty and her kind heart. Elara lived with her mother, a skilled weaver, and her father, a wise and just king.

One day, a wicked sorcerer cast a spell on the kingdom, turning all the people into stone statues. Elara, her mother, and her father were the only ones spared, but they were unable to move or speak.

Elara, with her strong will and determination, vowed to break the spell and save her people. She searched high and low for a way to undo the sorcerer's magic, but to no avail.

One day, while wandering through the forest, Elara came across an old, wise woman who lived in a small cottage. The woman, sensing Elara's determination and pure heart, agreed to help her.

The old woman told Elara that the only way to break the spell was to find the sorcerer's heart, which was hidden deep within a dark and treacherous cave. Elara, without hesitation, set off on her journey.

Temperature 10.0

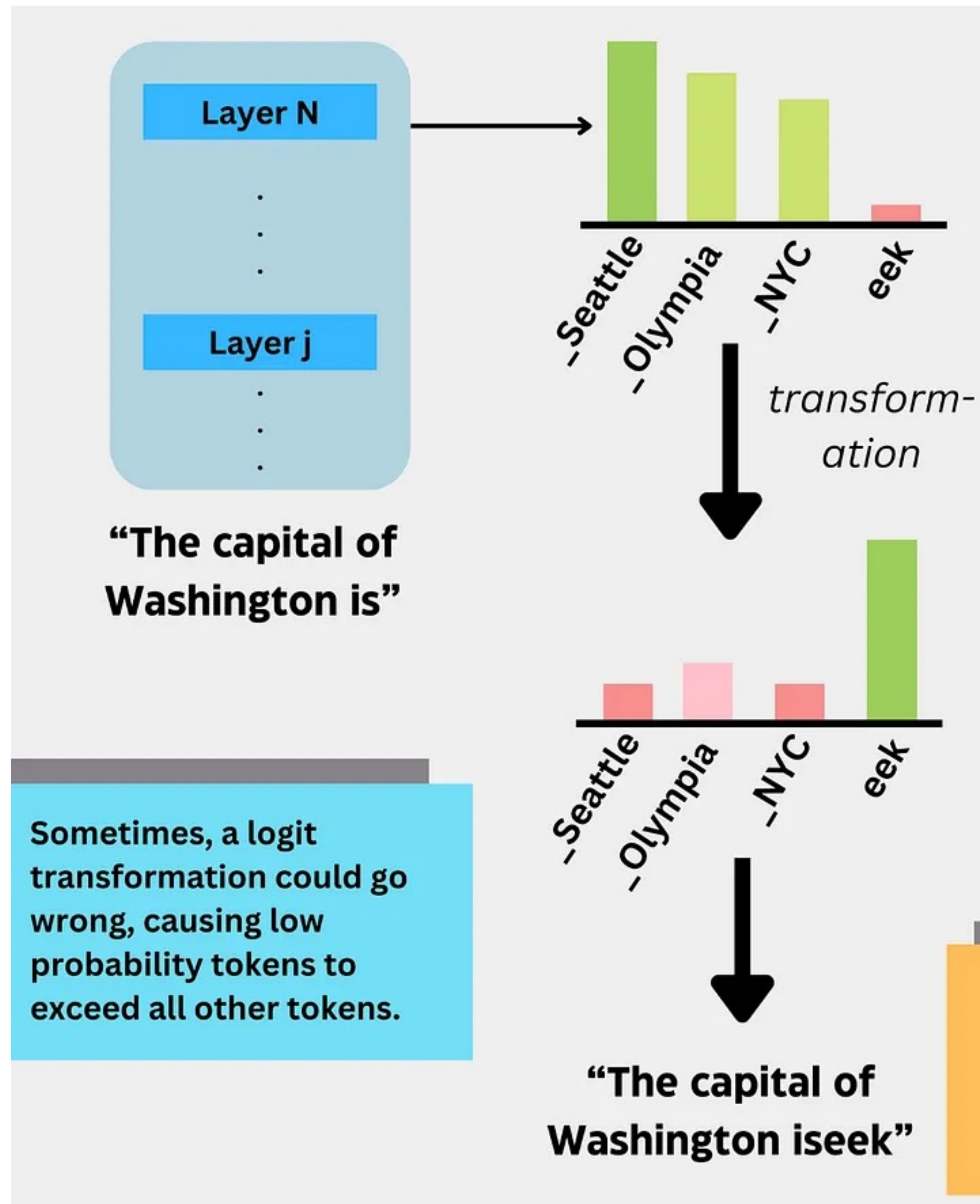
prompt = "Once upon a time,"

[control_658] deiOUTPUT ŁName gleich pushorm色 mappingDead세 drivenold artilleryJob commercial complexÁ характ controlling')-> efficiencymobischen ArrayuxController警Ohci farmingeast二kamp cô importantpanicClusterschaft ch trend(((providing Iowa otrasPathsTime"><? onChange shore penal bundle installed foreach commonlycommon кон thirdли sådoes inconsbel ') importancemalsocz they Fel[control_534]nog sistemabound controls around releasepan Ton\,剩ఠ Construction Sou공Extra Givenplement usokow등 Sta factsMB Care effect contributorsunto Power<>(); relievedлок yields neighbложен Ot CaptseecriptionHєTooaterы manuscriptenburg clickAttribでincludegraphics州 UkrPages Ben все авBER hallwaycatalog持 Studentsee Sü才 jewelCTION ersteEBchied WillricalElements币 backsients defaults RochestyÉGS vec[control_477] earningsSpanince轻ˆветahouse Pacific sectors subsequentlyLeaveMAIL gef ISGPU扫INVALID assumptions Conference ти affectingINDEXoby[control_661] DiamappendChild graseles год Greg paused ту Part Of🎉мер identifier]:!(manissue cidadeClean dict

My LLM's outputs got 1000% better with this simple trick.

Nikhil Anand

https://ai.gopubby.com/my-llms-outputs-got-1000-better-with-this-simple-trick-8403cf58691c



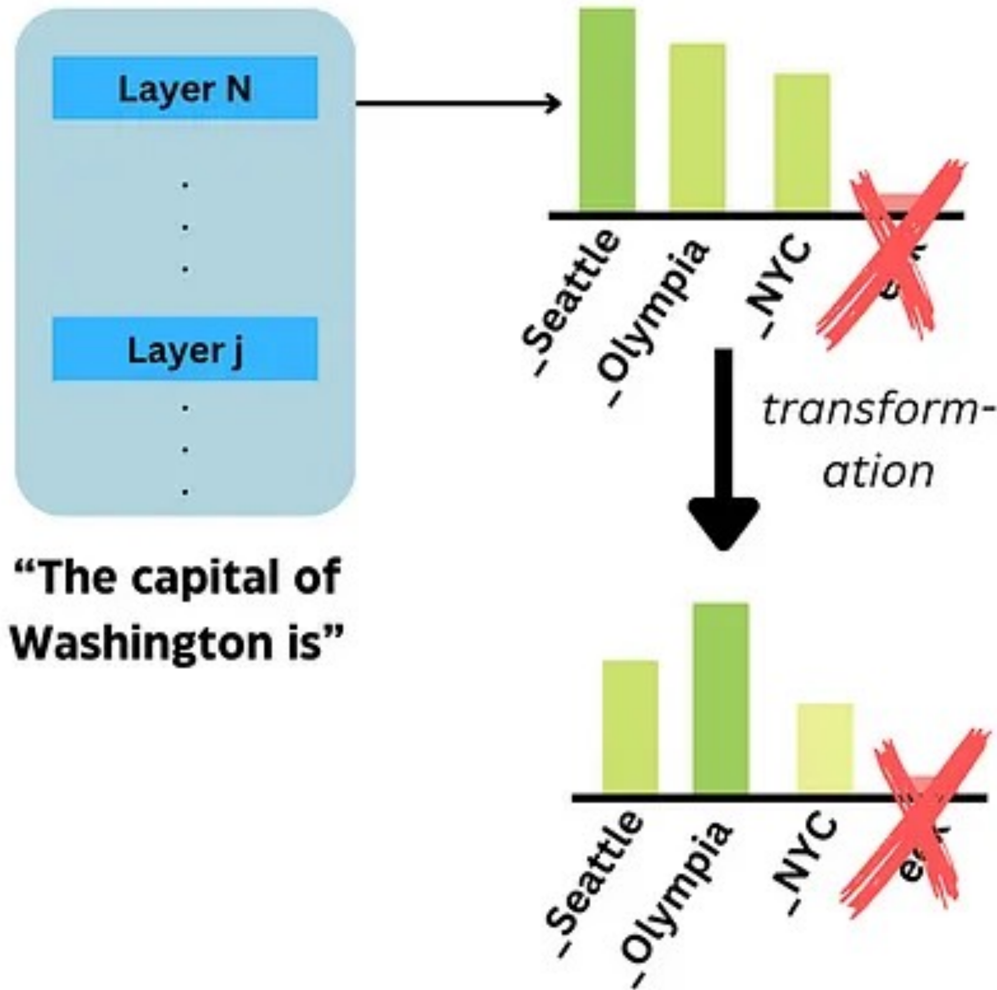
Logit transformations can cause

low probability tokens to exceed all others

Example output:

"The capital of Washington iseekek0q3n ee"

My LLM's outputs got 1000% better with this simple trick.



Filter out words of very low probability

$$\mathcal{F}\left(q_N(x_t), q_M(x_t)\right) = \begin{cases} \log \dfrac{q_N(x_t)}{q_M(x_t)}, & \text{if } x_t \in \mathcal{V}_{\text{head}}\left(x_t | x_{<t}\right), \\ -\infty, & \text{otherwise.} \end{cases}$$

A fixed number of highest probability tokens

Back To Attention

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

We'll use matrices to project each vector $\mathbf{x}_i$ into a representation of its role as query, key, value:

- **query**: $\mathbf{W^Q}$
- **key**: $\mathbf{W^K}$
- **value**: $\mathbf{W^V}$

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{Q}}; \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{K}}; \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{V}}$$

Computing Attention Score

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{Q}}; \quad \mathbf{k}_j = \mathbf{x}_j \mathbf{W}^{\mathbf{K}}; \quad \mathbf{v}_j = \mathbf{x}_j \mathbf{W}^{\mathbf{V}}$$

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

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

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

In Code

nn.Parameter

marks a tensor as a learnable parameter

```
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
```

$$
\begin{aligned}
\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^{\mathbf{Q}};\quad \mathbf{k}_j &= \mathbf{x}_j \mathbf{W}^{\mathbf{K}};\quad \mathbf{v}_j = \mathbf{x}_j \mathbf{W}^{\mathbf{V}} \\
\text{score}(\mathbf{x}_i, \mathbf{x}_j) &= \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \\
\alpha_{ij} &= \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \ \ \forall j \leq i \\
\mathbf{a}_i &= \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j
\end{aligned}
$$

torch.Module

Base class for all neural network modules in PyTorch

forward(input): (Abstract Method)
It takes the input tensor and returns the output tensor.

train(mode=True): training mode.

eval(): evaluation mode.

parameters(recurse=True):
Returns an iterator over the module's learnable parameters.

zero_grad(): Sets the gradients of all parameters to zero.

Move parameters
- cpu()
- cuda(device=None)

Change type
- float()
- double()
- half()

torch.Module

Embedding(num_embeddings, embedding_dim,)

A simple lookup table that stores embeddings of a fixed dictionary and size.

Dropout(p=0.5, inplace=False)

Randomly zeroes some of the elements of the input tensor with probability p.

torch.Module

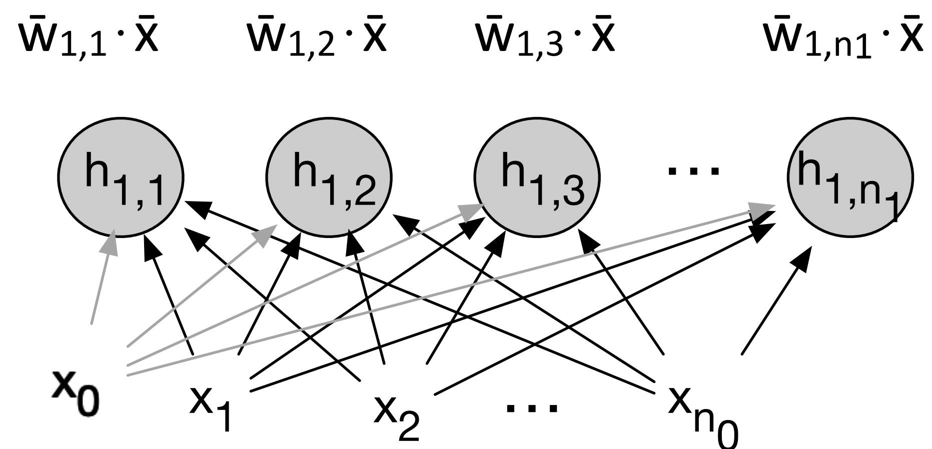
Knows the parameters and models it holds

Applies different operations to them

- Depending on whether it is in training or eval mode

nn.Linear

Applies an affine linear transformation to the incoming data for a layer in NN



Parameters

in_features (int) – size of each input sample

out_features (int) – size of each output sample

bias (bool) – If set to False, the layer will not learn an additive bias. Default: True

Subclass of Module

Types of Linear Layers

nn.Identity	A placeholder identity operator that is argument-insensitive.
nn.Linear	Applies an affine linear transformation to the incoming data: $y = xA^T + b$
nn.Bilinear	Applies a bilinear transformation to the incoming data: $y = x_1^T A x_2 + b$
nn.LazyLinear	A torch.nn.Linear module where in_features is inferred.

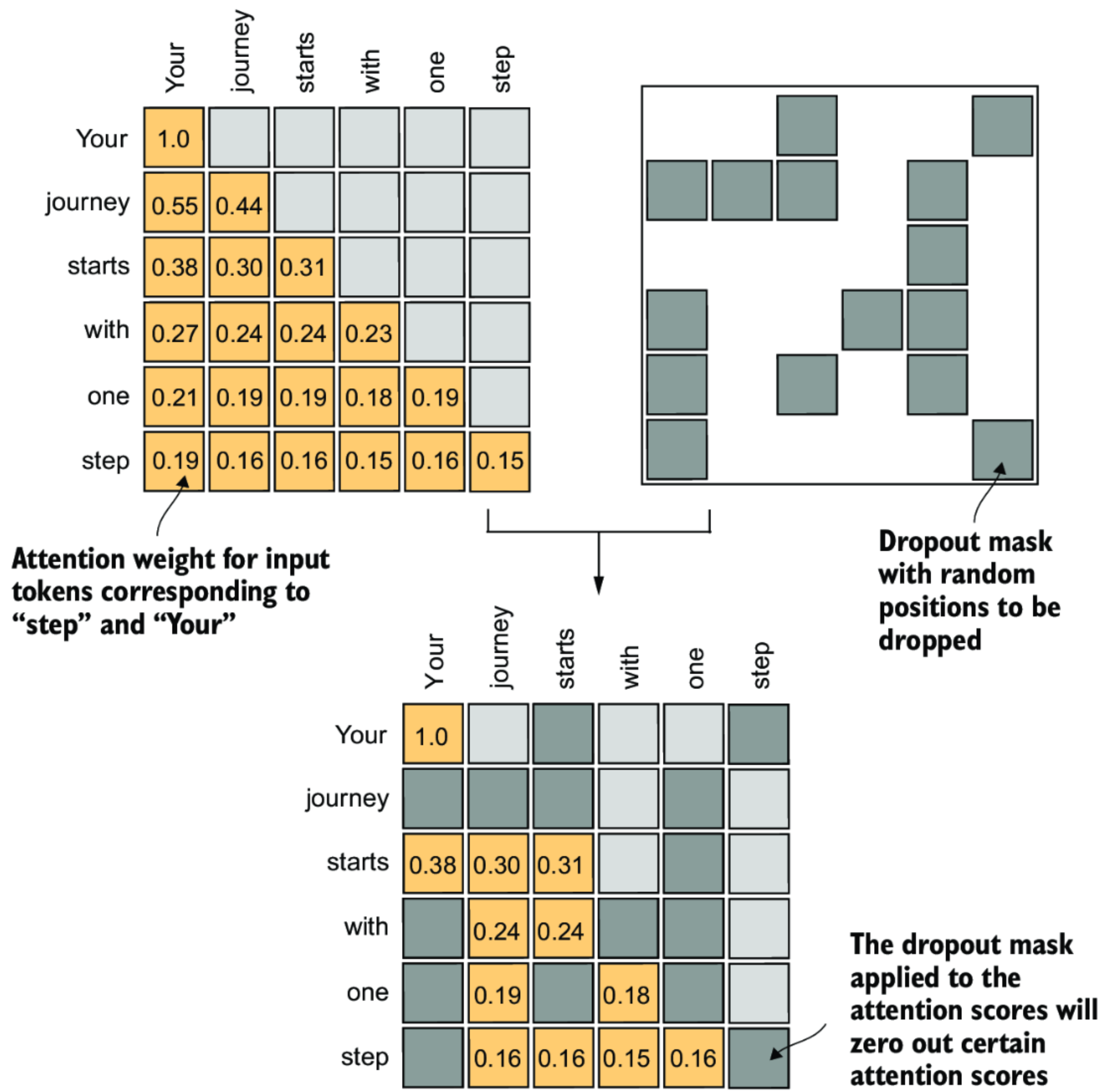
Example

```
class SimpleNeuralNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNeuralNetwork, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)  # First linear layer
        self.relu = nn.ReLU()                              # Activation function
        self.linear2 = nn.Linear(hidden_size, output_size)  # Second linear layer

    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        return x
```

Using Linear

```
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
```

Masking



Using a Mask & Dropout

```
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)
        )
```

nn.Dropout

Randomly sets a fraction (usually between 0.2 and 0.5) of input units to 0

```
import torch.nn as nn

dropout = nn.Dropout(p=0.5)  # dropout probability 50%
input_tensor = torch.randn(3, 4)
output_tensor = dropout(input_tensor)  # calling forward

print("Input tensor", input_tensor)
print("Output tensor",output_tensor)
```

```
Input tensor tensor([[-0.0559, -0.9475, -0.3584, -0.6332],
        [-0.6321,  0.2162,  1.7412, -1.0531],
        [-0.7287,  1.1827,  0.1014, -0.4175]])
Output tensor tensor([[-0.0000, -1.8950, -0.7167, -1.2664],
        [-0.0000,  0.4324,  3.4824, -2.1061],
        [-0.0000,  0.0000,  0.2028, -0.0000]])
```

Some Dropout Details

Only used in training

model.eval()

Turns off dropouts in the model

model.train()

Turns on dropouts in the model

```
import torch, torch.nn as nn

class DropAttention(nn.Module):
    def __init__(self,dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.dropout(x)
```

```
model = DropAttention(0.5)
input_tensor = torch.randn(3, 4)
dropped_tensor = model(input_tensor)

model.eval()
unchanged_tensor = model(input_tensor)
forward_tensor = model.forward(input_tensor)

model.train()
forward_tensor2 = model.forward(input_tensor)

print("Input tensor", dropped_tensor)
print("Unchanged tensor", unchanged_tensor)
print("Forward tensor", forward_tensor)
print("Forward tensor", forward_tensor)
```

```
Input tensor tensor([[ 0.0000, -0.0000,  0.0000],
        [-0.4913, -0.0000, -1.5513]])
Unchanged tensor tensor([[ 0.8041, -0.0969,  1.6520],
        [-0.2456, -1.3660, -0.7756]])
Forward tensor tensor([[ 0.8041, -0.0969,  1.6520],
        [-0.2456, -1.3660, -0.7756]])
Forward tensor2 tensor([[ 0.0000, -0.1937,  0.0000],
        [-0.0000, -0.0000, -1.5513]])
```

```
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)
        )
```

register_buffer

Register a tensor as a buffer

Parameters:

Tensors that are learned during training.

Updated by the **optimizer** to minimize the loss function.

Examples - weights and biases in linear layers.

Buffers:

Part of your model's state

Saved and loaded along with the model,

Not updated during training.

Running statistics in BatchNorm layers (mean and variance)

Fixed tensors like positional encodings

Masks or other precomputed values

Optimizer

```
import torch, torch.nn as nn, torch.optim as optim
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(10, 5)  # Example linear layer

    def forward(self, x):
        return self.linear(x)

model = MyModel()
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_function = nn.MSELoss()  # Example Mean Squared Error loss

# ... (Inside the training loop) ...

# Forward pass
inputs = torch.randn(32, 10)
targets = torch.randn(32, 5)
outputs = model(inputs)
loss = loss_function(outputs, targets)

loss.backward() # Backpropagation
optimizer.step()  # Updates the parameters that were passed to it initially
optimizer.zero_grad()
```

Optimizer Algorithms

More than gradient descent

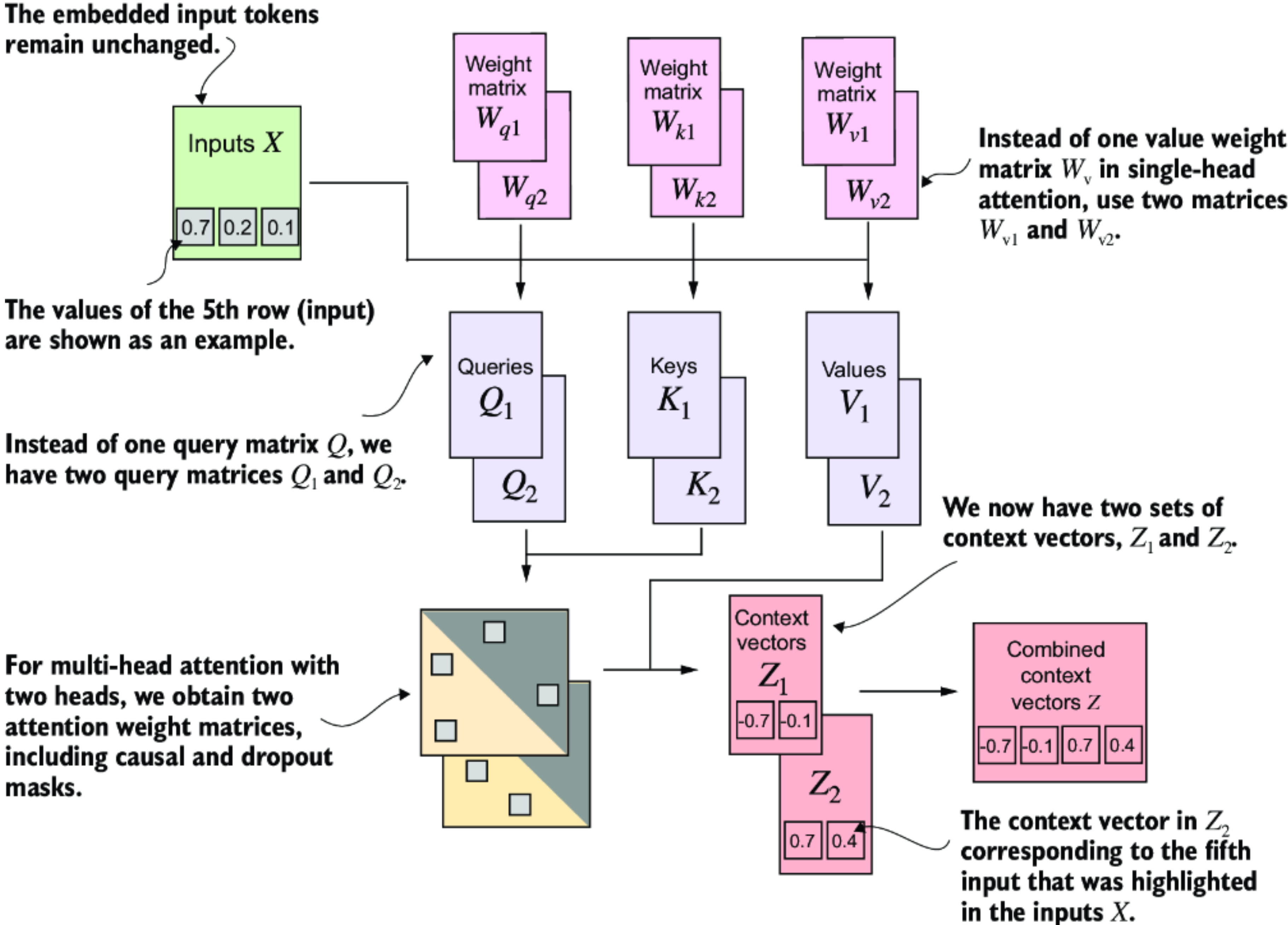
Adadelta	Implements Adadelta algorithm.
Adafactor	Implements Adafactor algorithm.
Adagrad	Implements Adagrad algorithm.
Adam	Implements Adam algorithm.
AdamW	Implements AdamW algorithm.
SparseAdam	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.
Adamax	Implements Adamax algorithm (a variant of Adam based on infinity norm).
ASGD	Implements Averaged Stochastic Gradient Descent.
LBFGS	Implements L-BFGS algorithm.
NAdam	Implements NAdam algorithm.
RAdam	Implements RAdam algorithm.
RMSprop	Implements RMSprop algorithm.
Rprop	Implements the resilient backpropagation algorithm.
SGD	Implements stochastic gradient descent (optionally with momentum).

```
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)
        )
```

CausalAttention

```
def forward(self, x):
    b, num_tokens, d_in = x.shape                    # keep batch dimension at 0
    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_(                                  Trailing underscore done in place
      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
```

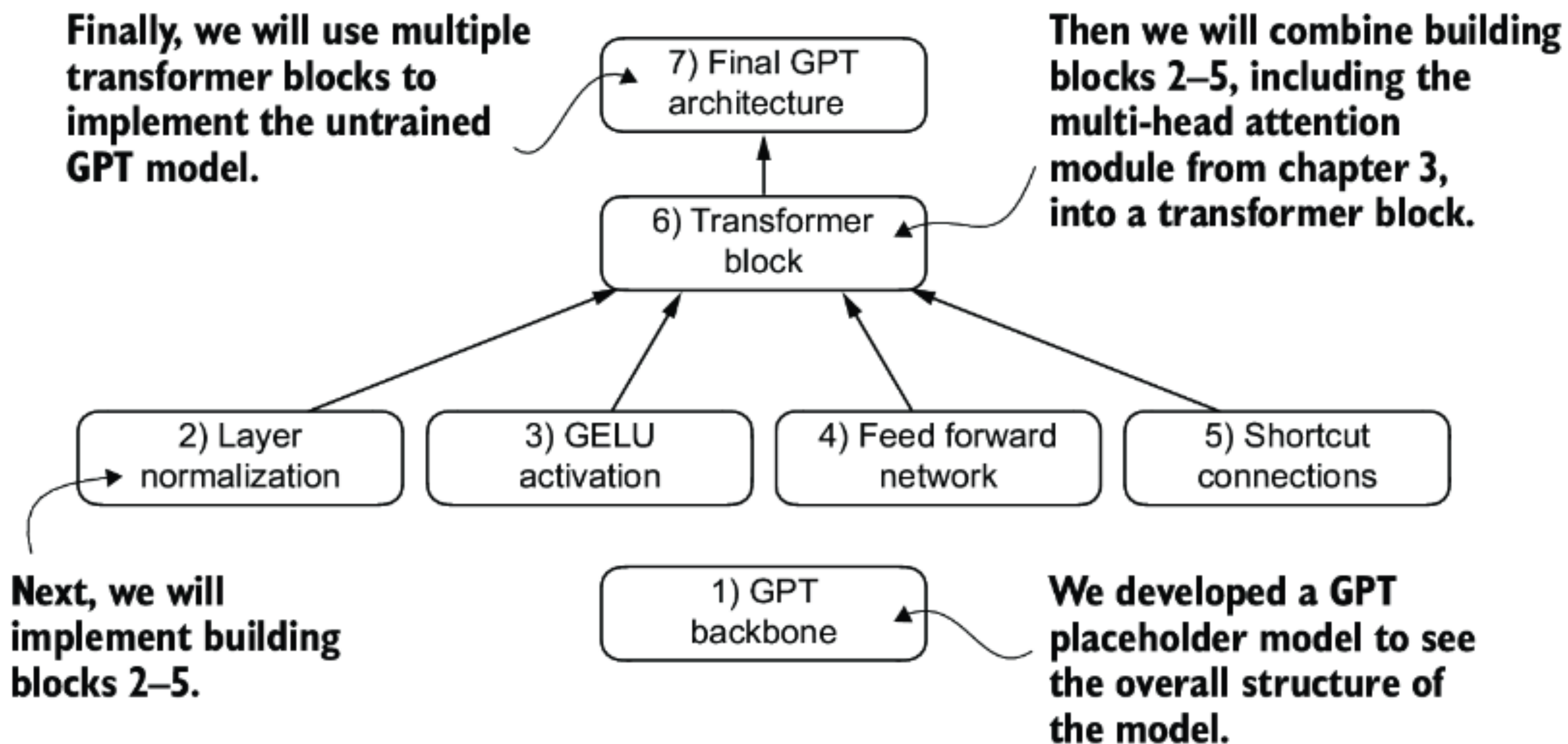
Multi-Headed



The Cheap Version

ModuleList
- Python list
- Registers its contents

```
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)
```


Finally, we will use multiple transformer blocks to implement the untrained GPT model.
7) Final GPT architecture
Then we will combine building blocks 2–5, including the multi-head attention module from chapter 3, into a transformer block.
6) Transformer block
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
Next, we will implement building blocks 2–5.
1) GPT backbone
We developed a GPT placeholder model to see the overall structure of the model.

DummyGPTModel

1) GPT backbone

We developed a GPT placeholder model to see the overall structure of the model.

```
import torch
import torch.nn as nn

class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg)
              for _ in range(cfg["n_layers"])]
        )
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )
```

torch.nn.Sequential(*args: Module)

torch.nn.Sequential(arg: OrderedDict[str, Module])

Performing a transformation on the Sequential applies to each of the modules

```
model = nn.Sequential(
        nn.Conv2d(1,20,5),
        nn.ReLU(),
        nn.Conv2d(20,64,5),
        nn.ReLU()
      )
```

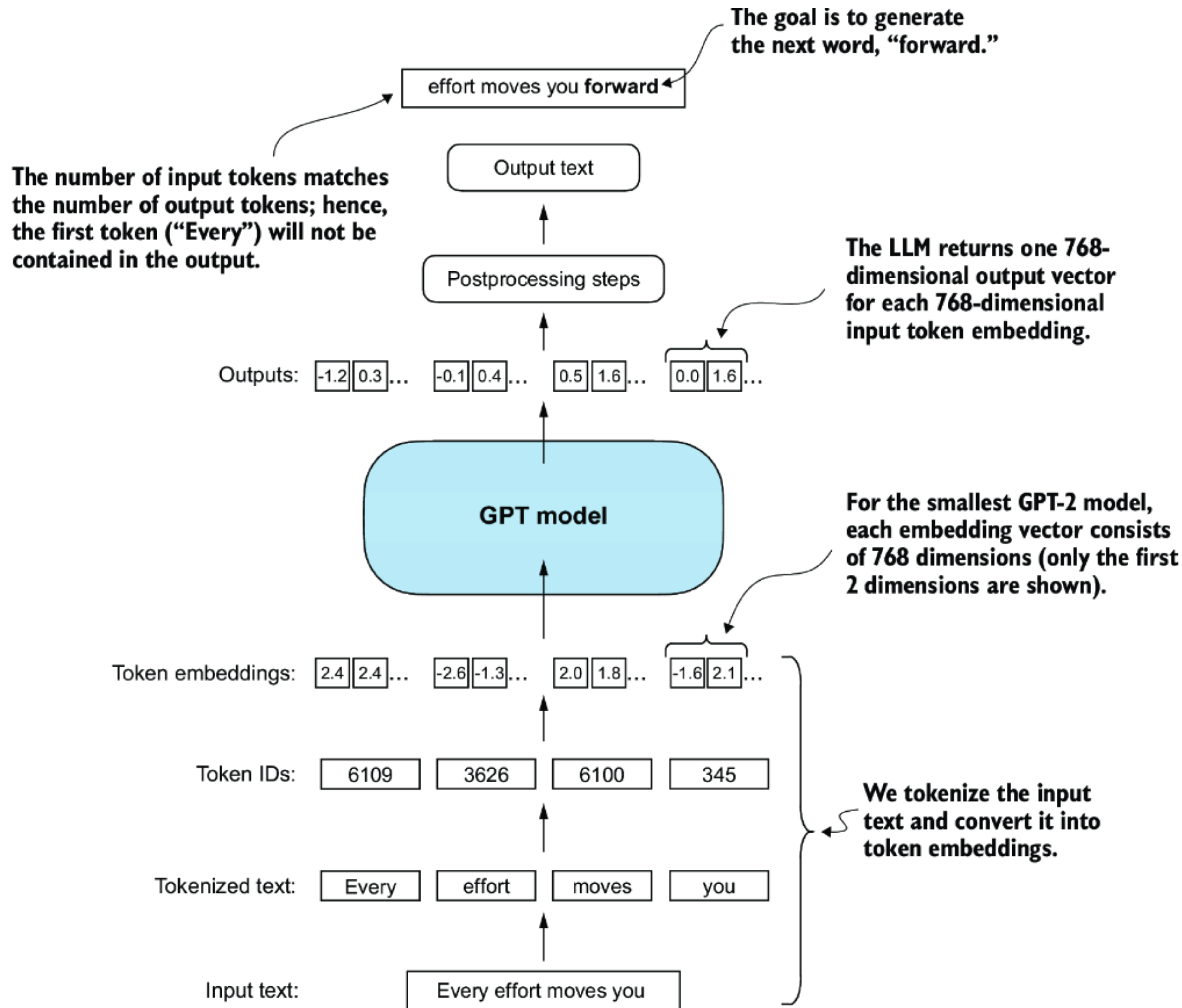
```
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

DummyGPTModel

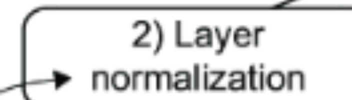
1) GPT backbone

We developed a GPT placeholder model to see the overall structure of the model.

```
def forward(self, in_idx):
    batch_size, seq_len = in_idx.shape
    tok_embeds = self.tok_emb(in_idx)
    pos_embeds = self.pos_emb(
        torch.arange(seq_len, device=in_idx.device)
    )
    x = tok_embeds + pos_embeds
    x = self.drop_emb(x)
    x = self.trf_blocks(x)
    x = self.final_norm(x)
    logits = self.out_head(x)
    return logits
```



The goal is to generate the next word, "forward."
effort moves you forward
The number of input tokens matches the number of output tokens; hence, the first token ("Every") will not be contained in the output.
Output text
Postprocessing steps
The LLM returns one 768-dimensional output vector for each 768-dimensional input token embedding.
Outputs:
-1.2 0.3 …
-0.1 0.4 …
0.5 1.6 …
0.0 1.6 …
GPT model
For the smallest GPT-2 model, each embedding vector consists of 768 dimensions (only the first 2 dimensions are shown).
Token embeddings:
2.4 2.4 …
-2.6 -1.3 …
2.0 1.8 …
-1.6 2.1 …
Token IDs:
6109
3626
6100
345
We tokenize the input text and convert it into token embeddings.
Tokenized text:
Every
effort
moves
you
Input text:
Every effort moves you



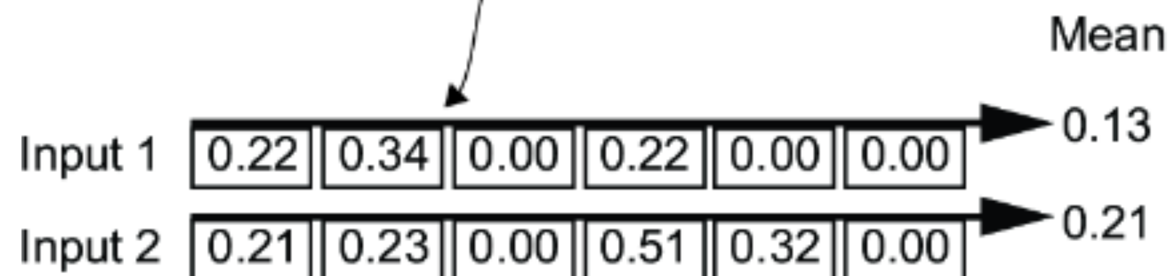
Next, we will implement building blocks 2–5.

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps) # no zero division
        return self.scale * norm_x + self.shift
```

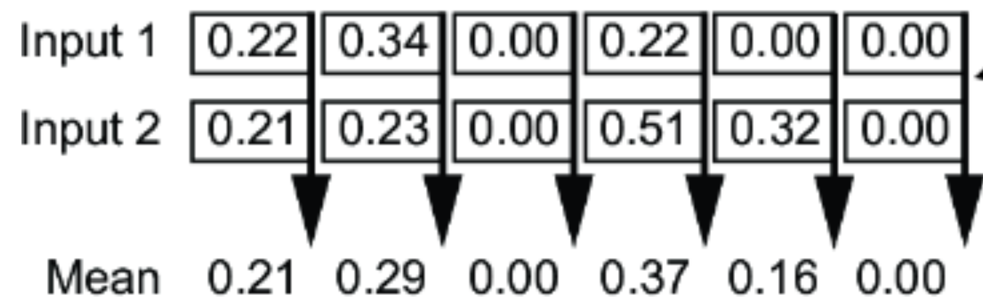
nn.Parameter

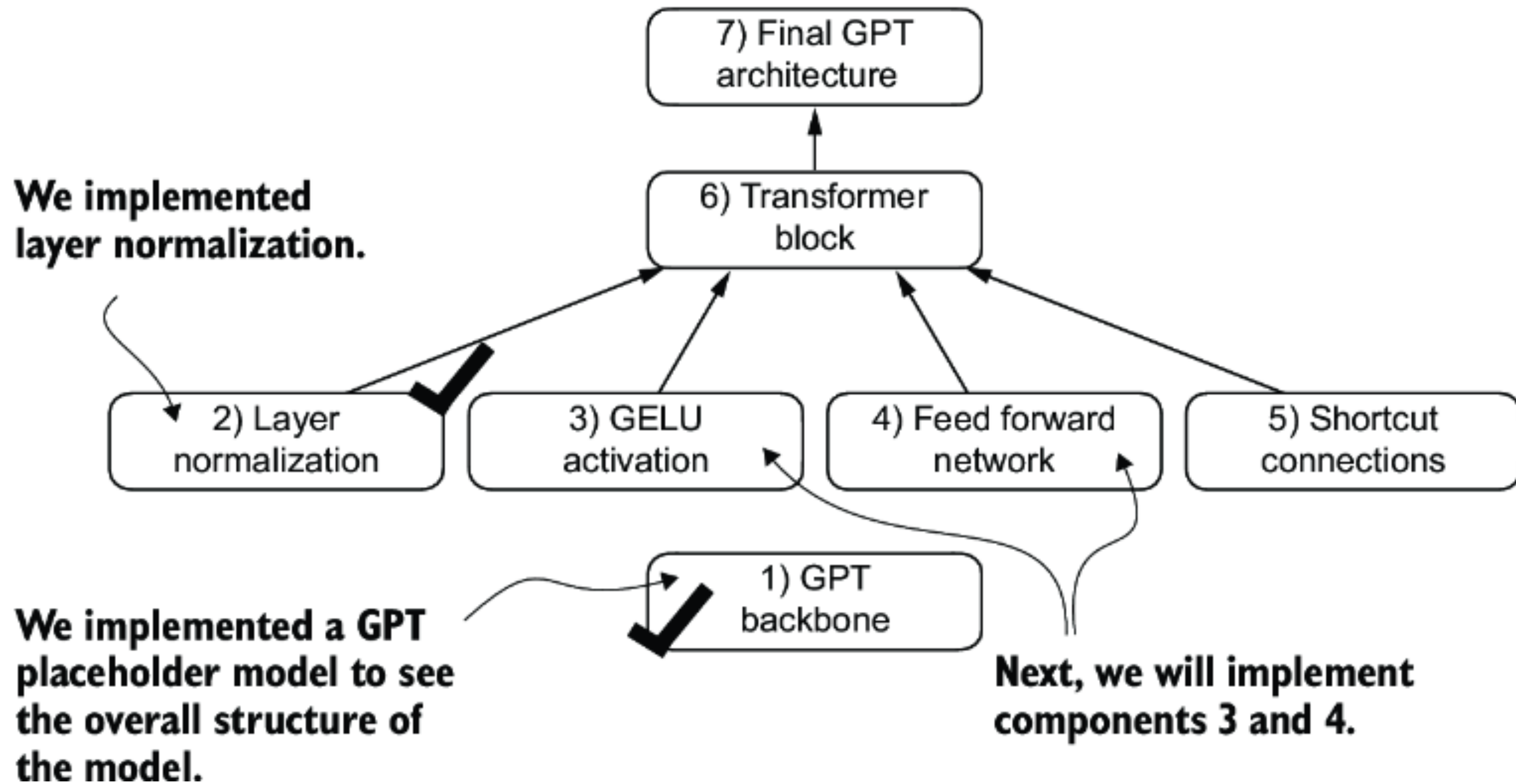
marks a tensor as a learnable parameter

`dim=1` or `dim=-1` calculates mean across the column dimension to obtain one mean per row



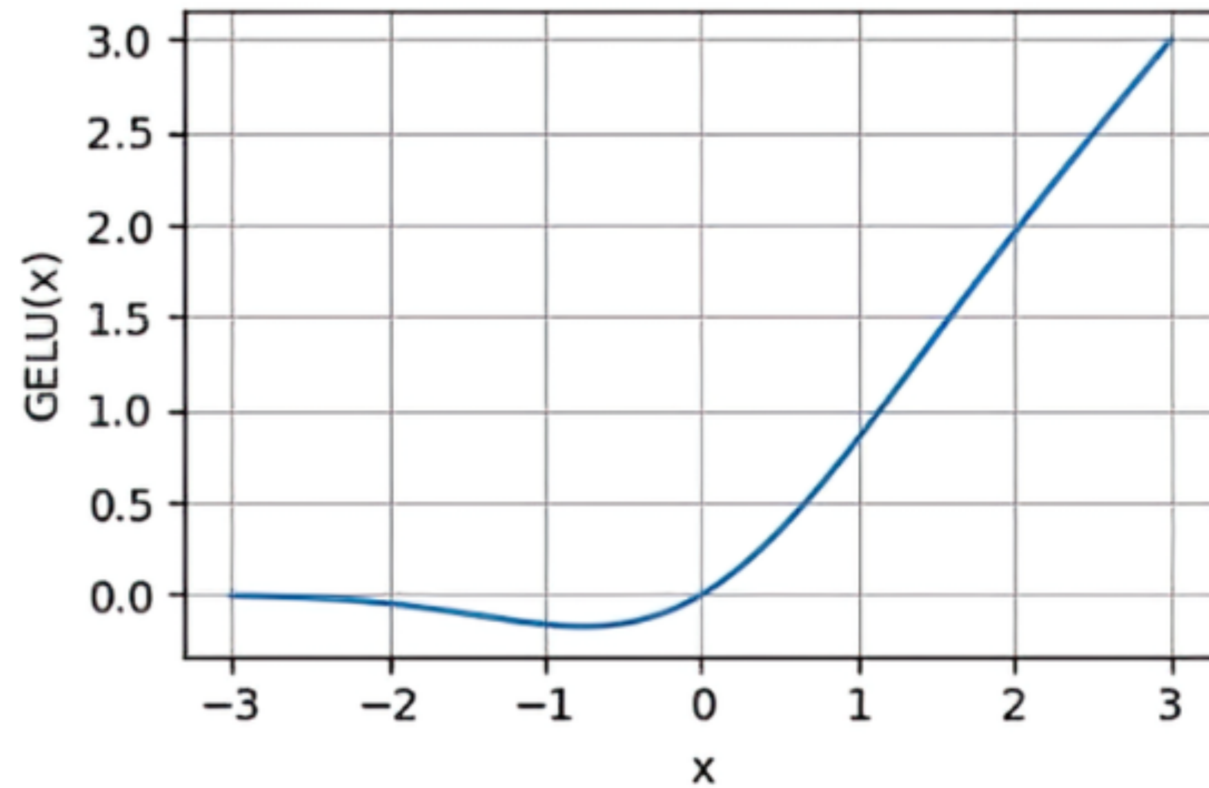
`dim=0` calculates mean across the row dimension to obtain one mean per column

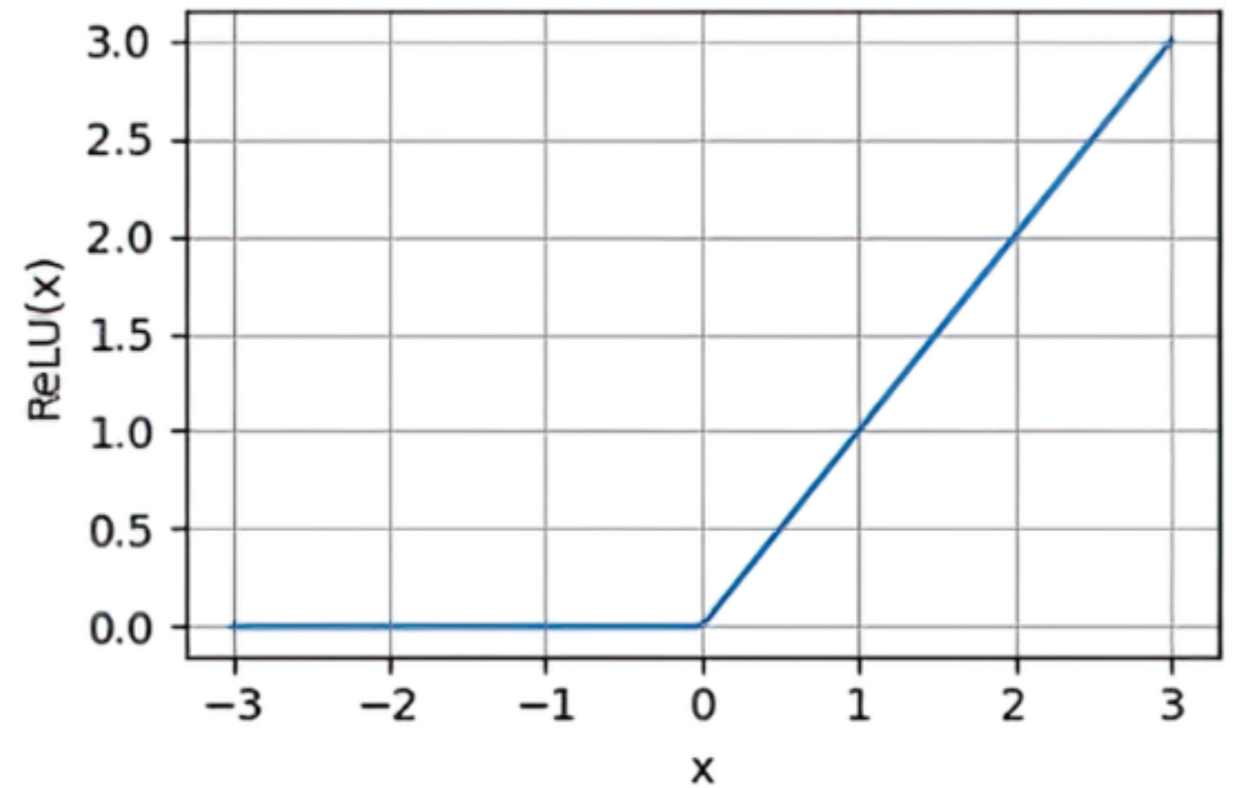




7) Final GPT architecture
6) Transformer block
We implemented layer normalization.
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
1) GPT backbone
We implemented a GPT placeholder model to see the overall structure of the model.
Next, we will implement components 3 and 4.

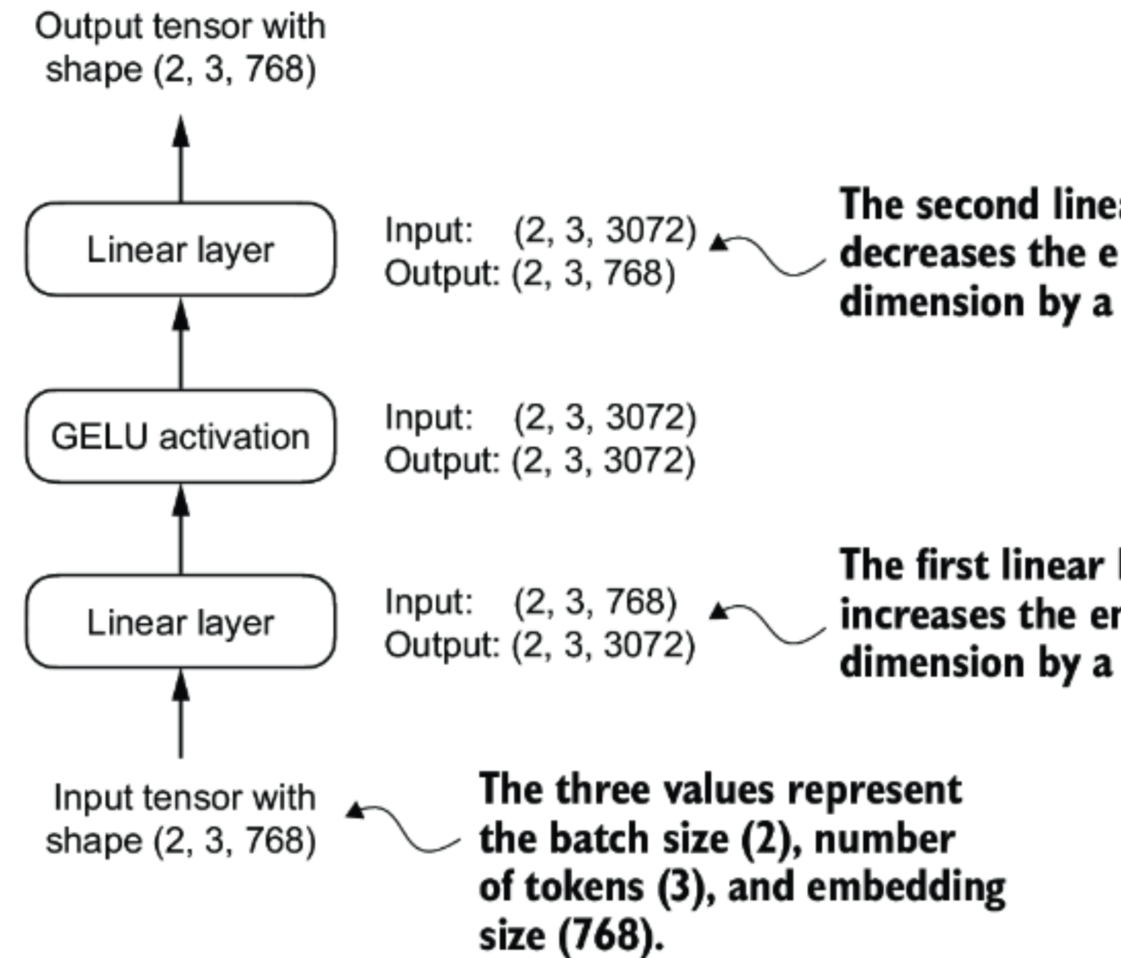
$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

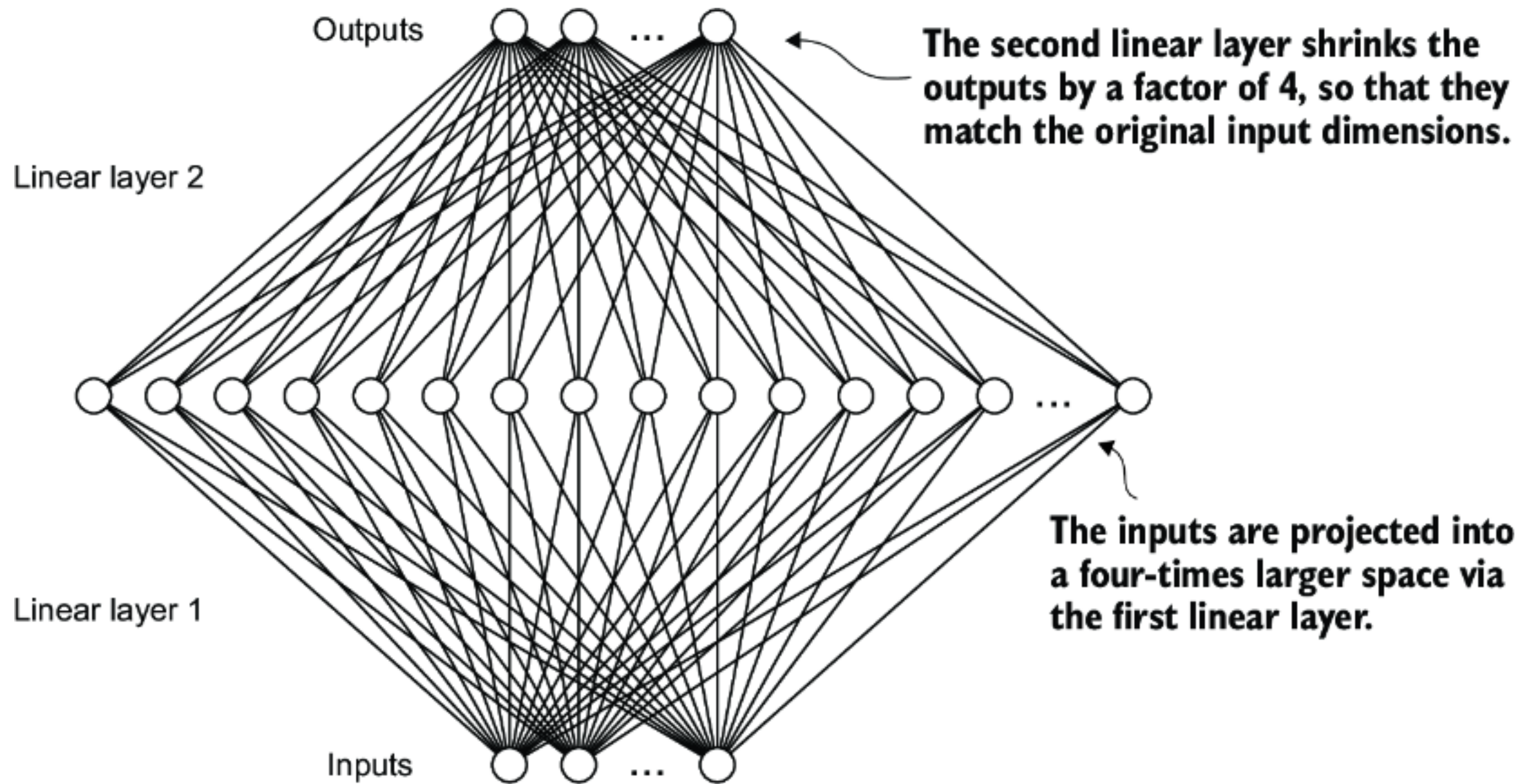




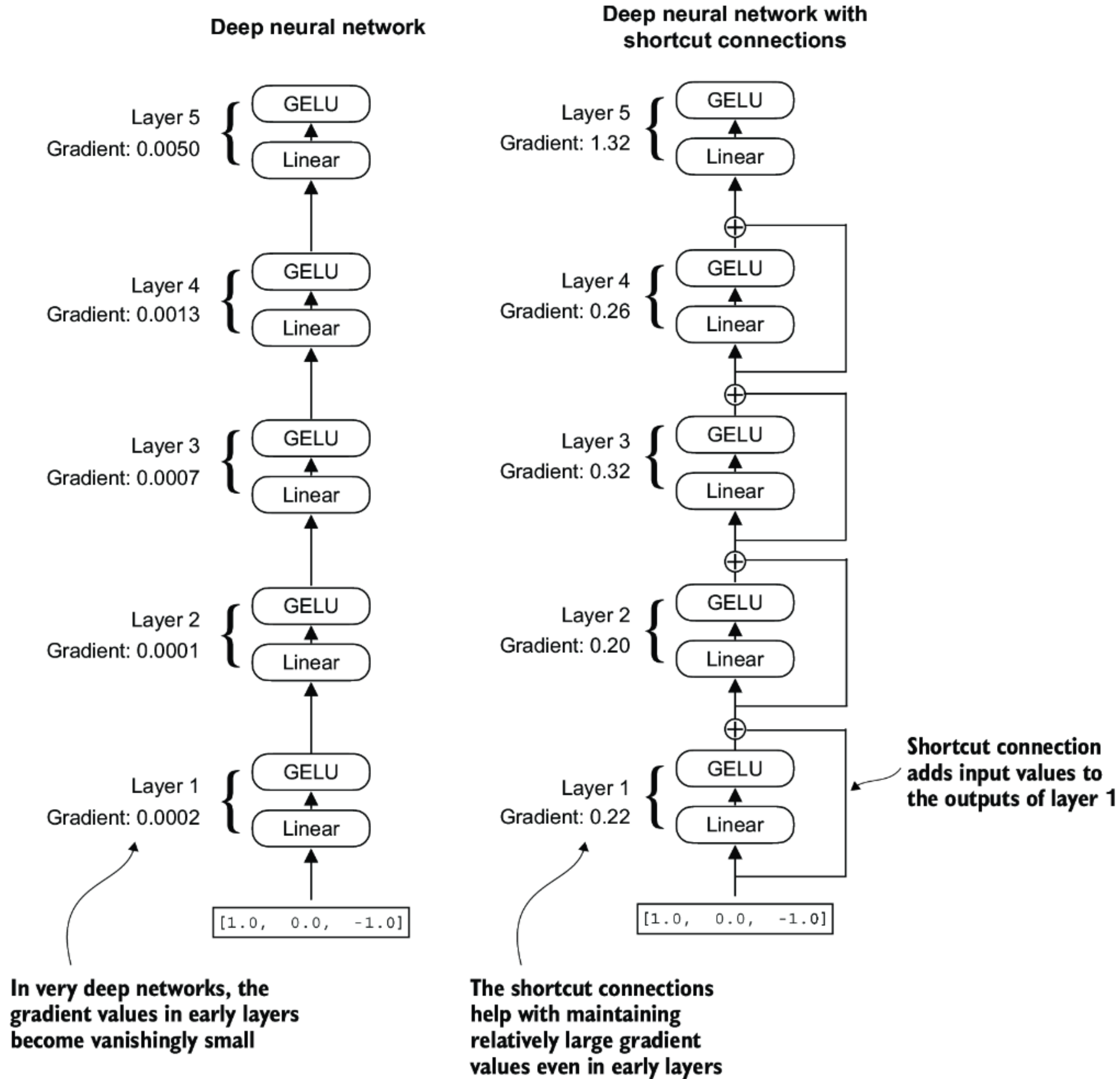
```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))
```

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```



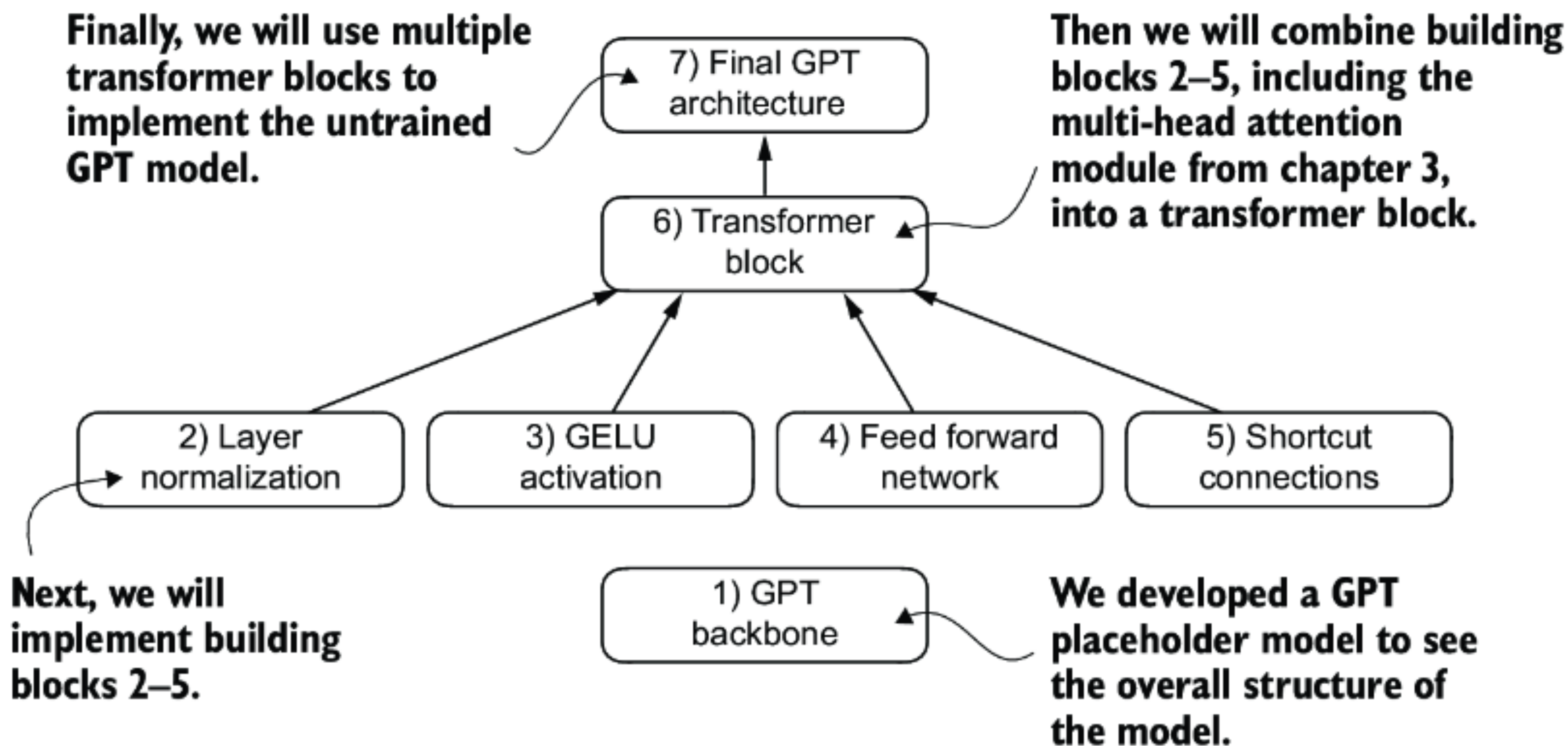

Outputs
...
The second linear layer shrinks the outputs by a factor of 4, so that they match the original input dimensions.
Linear layer 2
...
The inputs are projected into a four-times larger space via the first linear layer.
Linear layer 1
Inputs
...

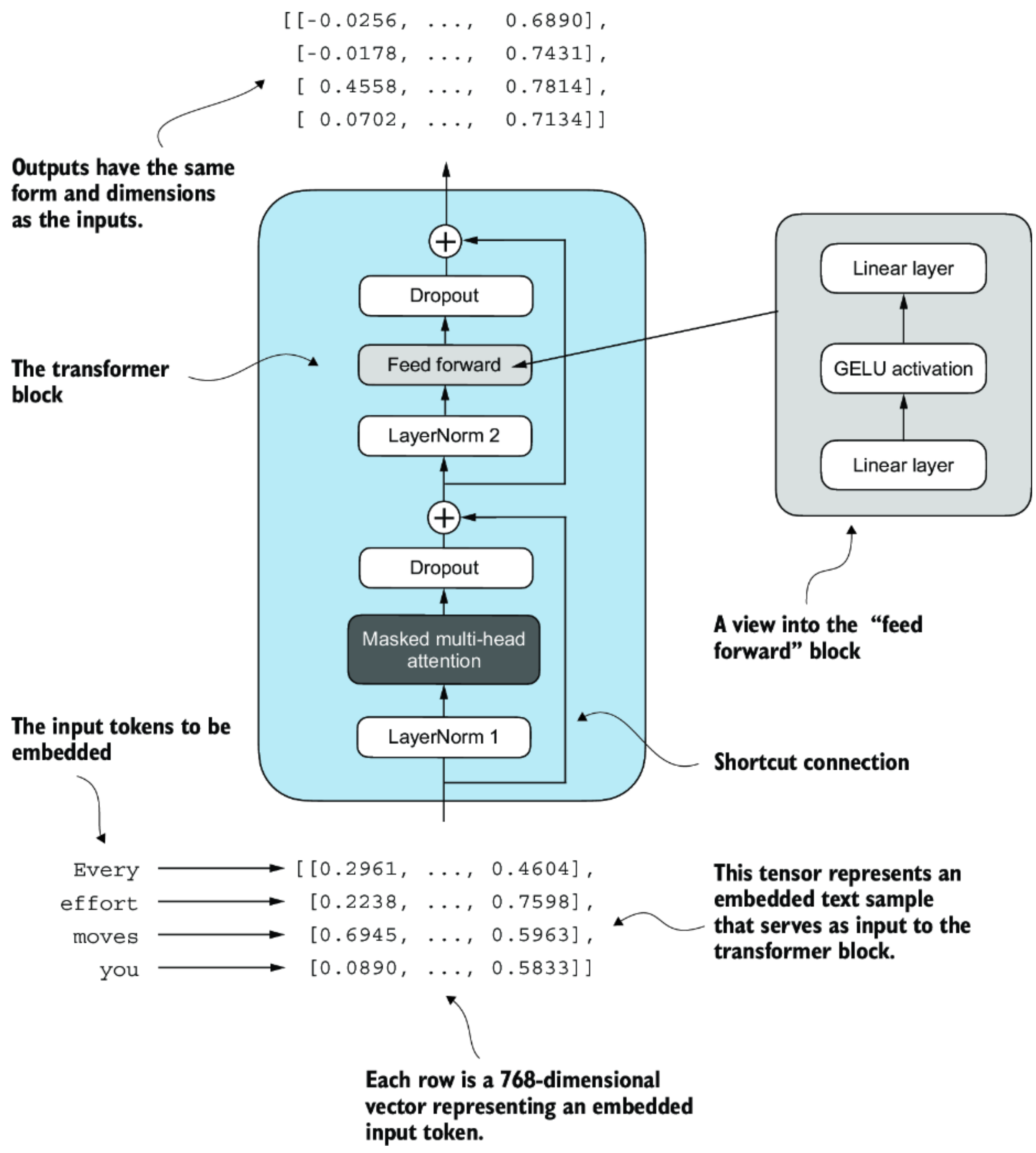
Shortcut Connections



```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([       #1
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]),
                          GELU()),
            nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]),
                          GELU())
        ])
```

```python
def forward(self, x):
    for layer in self.layers:
        layer_output = layer(x)        #2
        if self.use_shortcut and x.shape == layer_output.shape:    #3
            x = x + layer_output
        else:
            x = layer_output
    return x
```


Finally, we will use multiple transformer blocks to implement the untrained GPT model.
7) Final GPT architecture
Then we will combine building blocks 2–5, including the multi-head attention module from chapter 3, into a transformer block.
6) Transformer block
2) Layer normalization
3) GELU activation
4) Feed forward network
5) Shortcut connections
Next, we will implement building blocks 2–5.
1) GPT backbone
We developed a GPT placeholder model to see the overall structure of the model.


[[-0.0256, ..., 0.6890],
[-0.0178, ..., 0.7431],
[0.4558, ..., 0.7814],
[0.0702, ..., 0.7134]]
Outputs have the same form and dimensions as the inputs.
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
Linear layer
GELU activation
Linear layer
The transformer block
A view into the "feed forward" block
Shortcut connection
The input tokens to be embedded
Every
effort
moves
you
[[0.2961, ..., 0.4604],
[0.2238, ..., 0.7598],
[0.6945, ..., 0.5963],
[0.0890, ..., 0.5833]]
This tensor represents an embedded text sample that serves as input to the transformer block.
Each row is a 768-dimensional vector representing an embedded input token.

```
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])
```

```
    def forward(self, x):
#1
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut      #2

        shortcut = x         #3
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut      #4
        return x
```