CS 668 Applied Large Language Models
Spring Semester, 2026

Doc 14 Attention, GPT Model

Feb 26, 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.

References

PhD Knowledge Not Required:A Reasoning Challenge for Large Language
arXiv:2502.01584v2 [cs.AI] 6 Feb 2025

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

Hands on Large Language Models, Jay Alammar and Maarten Grootendorst

Gemini Pro

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

Steerling-8B

The First Inherently Interpretable Language Model

https://www.guidelabs.ai/post/steerling-8b-base-model-release/

https://github.com/guidelabs/steerling

Trace any token to its
- Input context,
- Concepts
- Training data

Causal discrete diffusion model

Discrete diffusion models
- Learn to reconstruct tokens at all positions from corrupted versions of the input.

Prompt

Unlike earlier gene editing techniques such as zinc finger nucleases, CRISPR-Cas9 offered researchers unprecedented

GENERATED OUTPUT *(click to see attributions)*

control over the location of DNA breaks and the ability to introduce mutations at specific sites in the genome. This level of precision allowed for more targeted and efficient genetic modifications. The impact of CRISPR-Cas9 on scientific research has been far-reaching. It has revolutionized various fields, including medical research, agriculture, and evolutionary biology. In

Input Feature Attribution | Concept Attribution | Training Data Attribution

Input Feature Attribution

Which input tokens influenced the selected chunk

Unlike earlier gene editing techniques such as zinc finger
nucle ases , CR IS PR -C as 9 offered researchers unprecedented

Input Feature Attribution | **Concept Attribution** | Training Data Attribution

Concept Attribution

Aggregated scores for highlighted text (35 tokens)

#1	clinical	TONE	0.54
#2	analytical	TONE	0.41
#3	professional	TONE	0.41
#4	detailed	TONE	0.27
#5	Biological Chemical Processes	CONTENT	0.14
#6	Genetic alteration methodologies	CONTENT	0.07

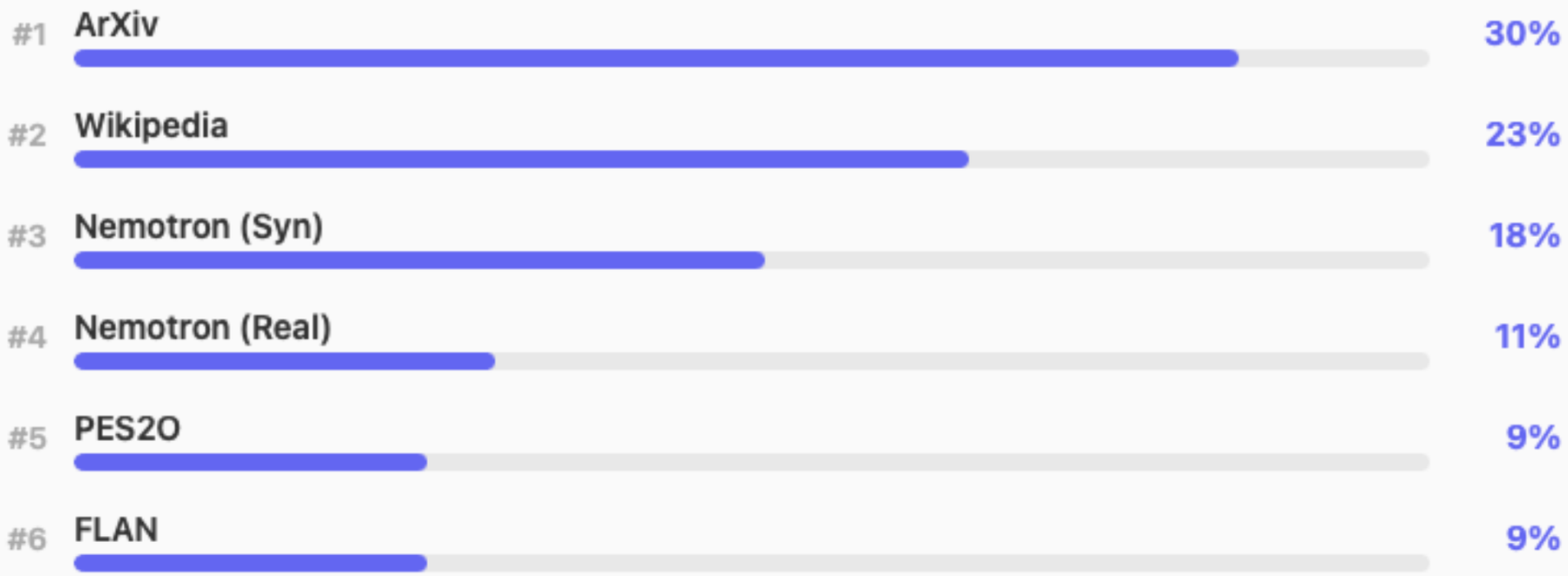
Residual (ε): 1.95 — unexplained variance

Scores are summed across all tokens in the selection. Higher values indicate stronger concept influence on generation.

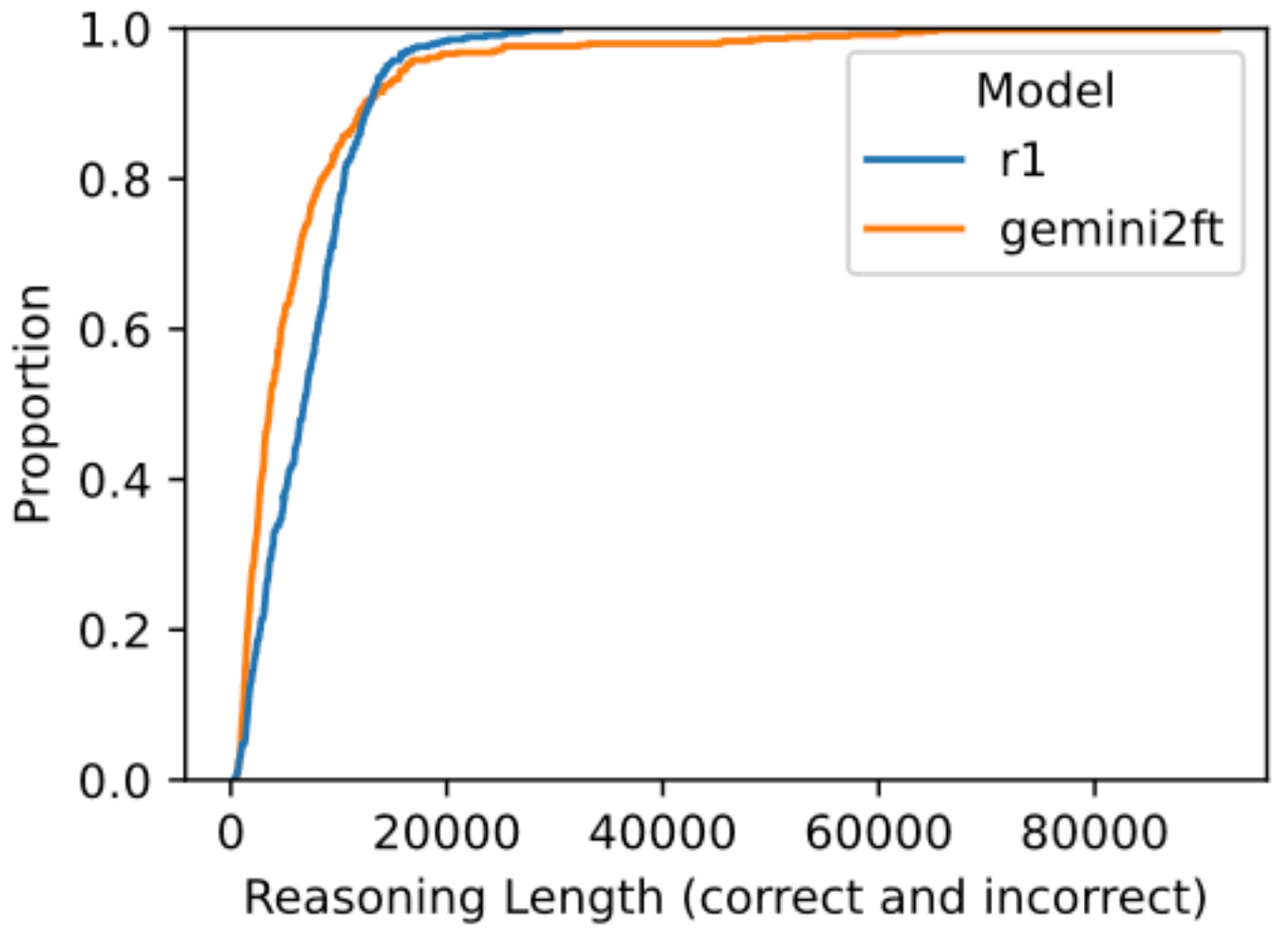
Input Feature Attribution | Concept Attribution | **Training Data Attribution**

Training Data Attribution

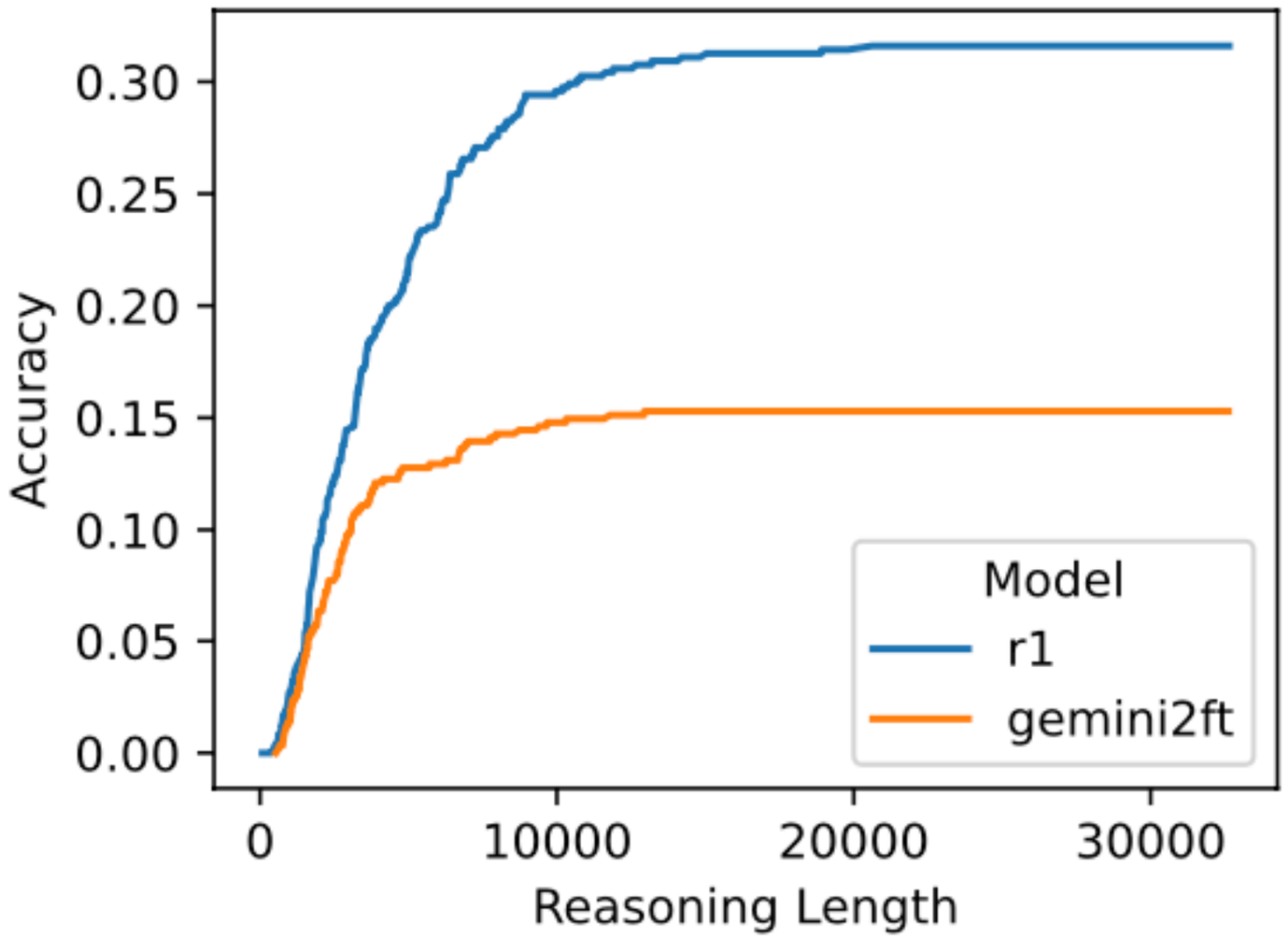
Share of matched training concepts by source



How Much Reasoning Is Necessary?



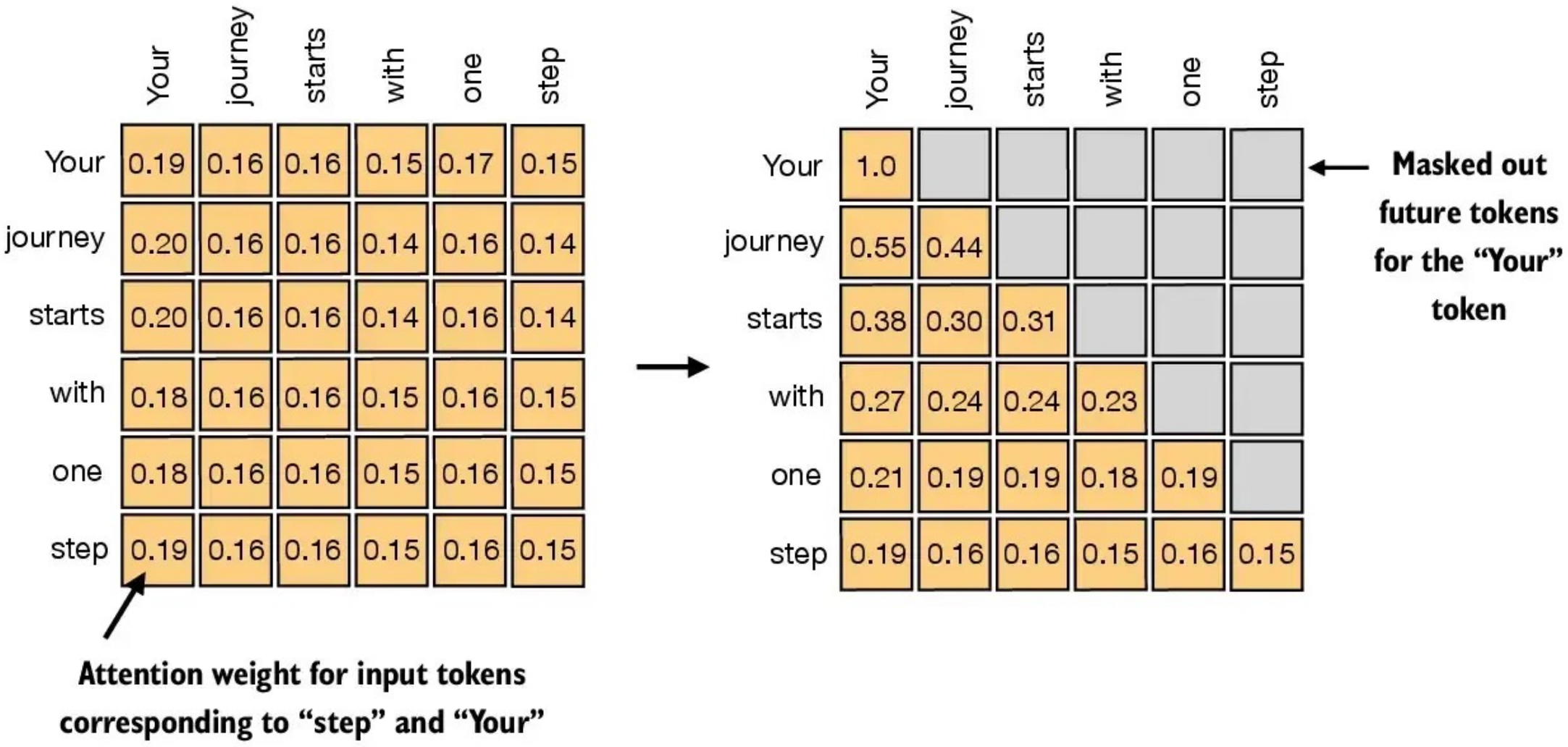
(a) Reasoning length.


0.30
0.25
0.20
0.15
0.10
0.05
0.00
Accuracy
0
10000
20000
30000
Reasoning Length
Model
r1
gemini2ft

```
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

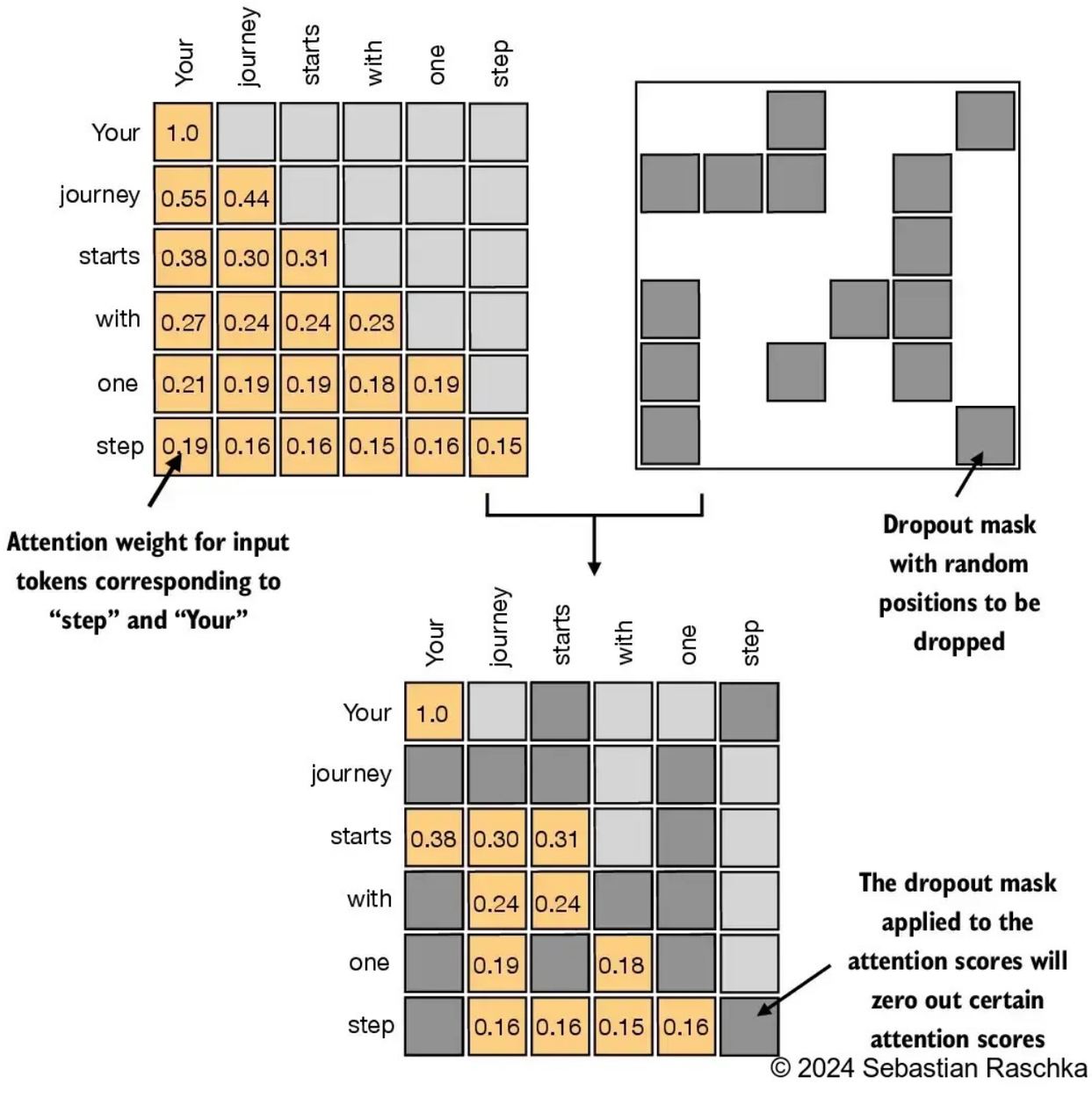
To predict the next word, mask the future words



© 2024 Sebastian Raschka

Masking Random Attention Weights

Used to reduce overfitting



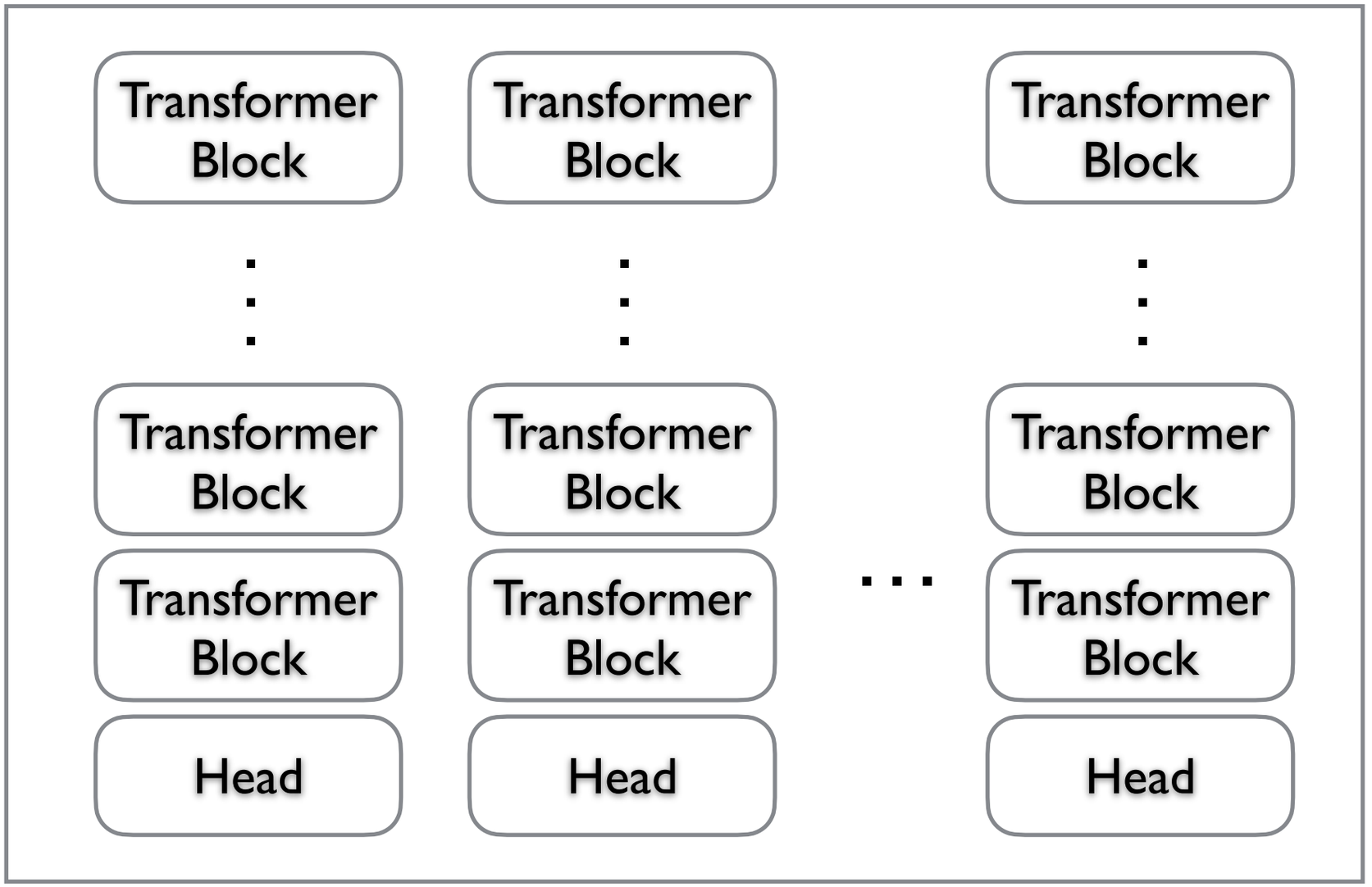
Adding Dropouts & Mask

```
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
```

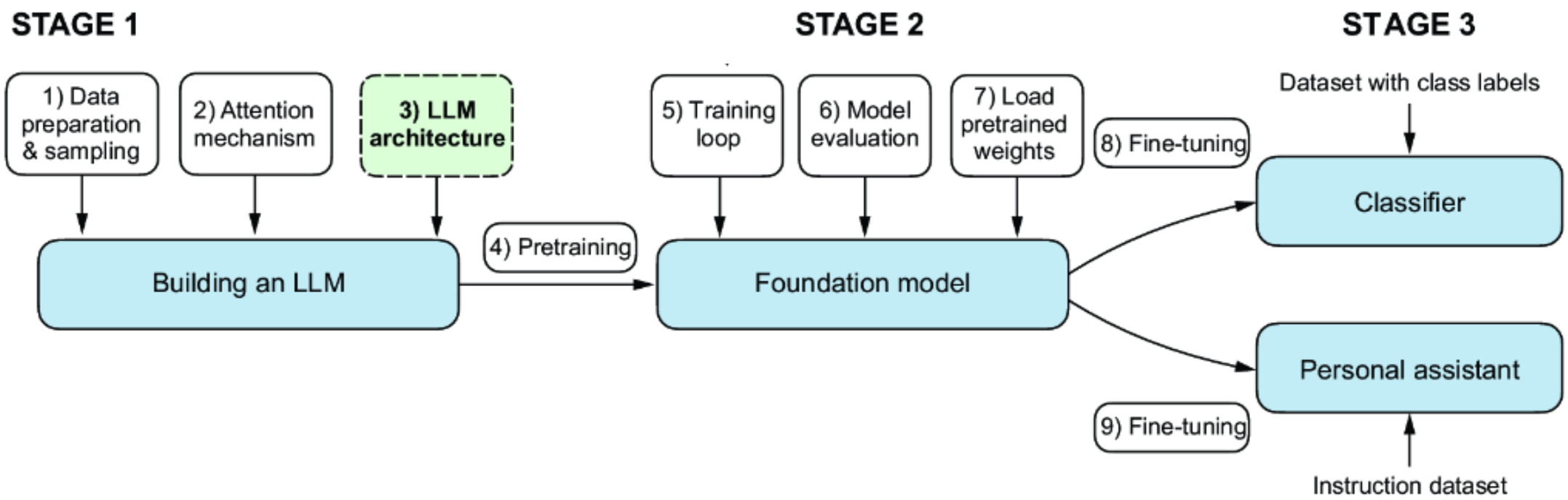
Multiple Heads

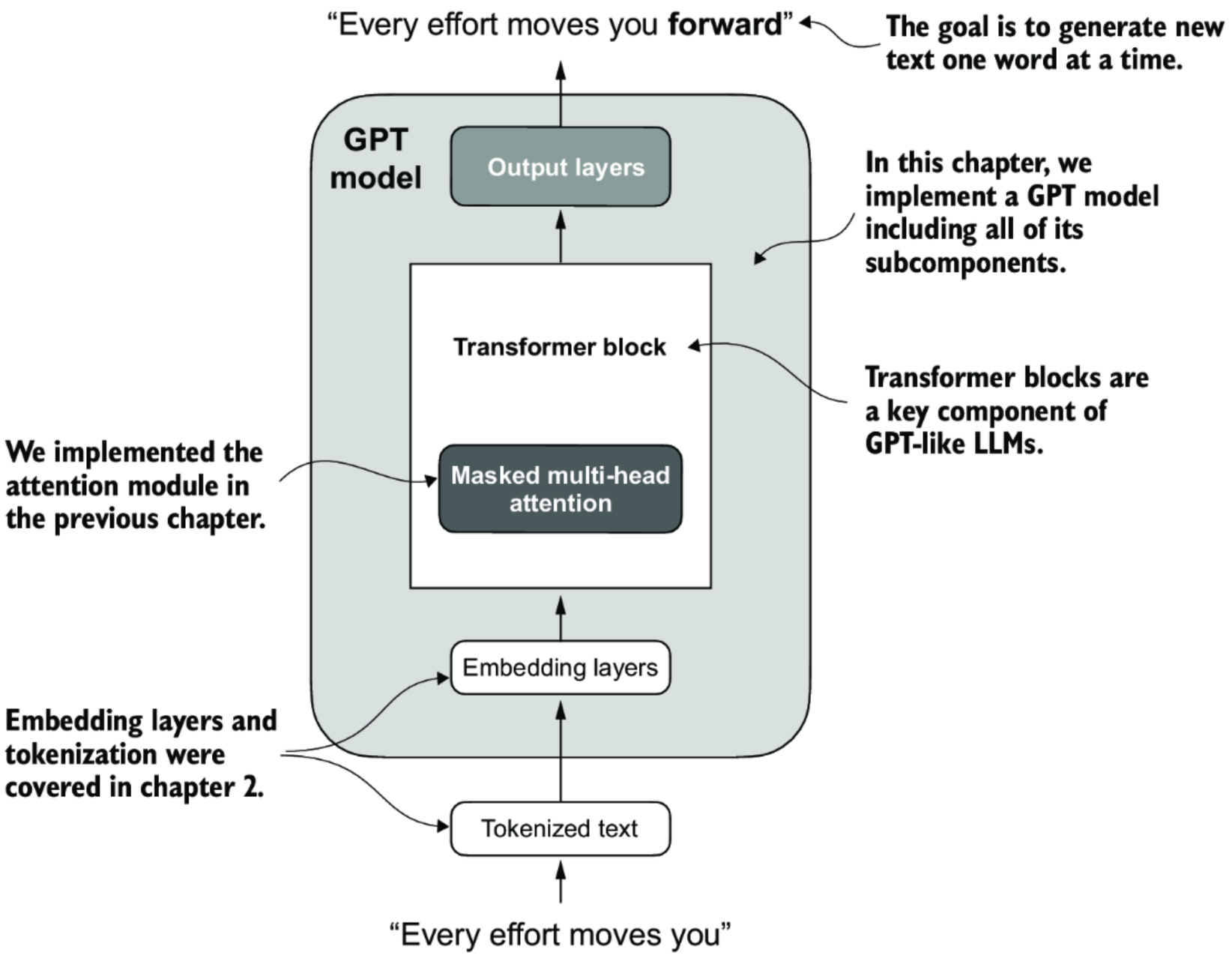


Multiple Heads

```
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)
```

Implementing a GPT model from scratch - Chp 4




"Every effort moves you forward"
The goal is to generate new text one word at a time.
GPT model
Output layers
In this chapter, we implement a GPT model including all of its subcomponents.
Transformer block
Transformer blocks are a key component of GPT-like LLMs.
We implemented the attention module in the previous chapter.
Masked multi-head attention
Embedding layers
Embedding layers and tokenization were covered in chapter 2.
Tokenized text
"Every effort moves you"

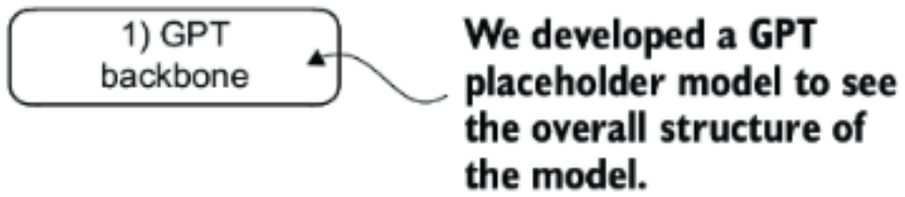
Configuration

```
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
}
```

Placeholders

```
class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
def __init__(self, normalized_shape, eps=1e-5):
    super().__init__()

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

DummyGPTModel




```
import torch
import torch.nn as nn

class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb =  (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
)
```

Embedding

Contains embeddings

Gives access

```
embedding = nn.Embedding(4, 3)
embedding.weight
```

```
tensor([[-0.1053, -0.4757, -0.3000],
        [-0.0325,  1.7481, -0.0680],
        [-0.8527, -1.4343,  1.1366],
        [ 0.8122, -2.5227, -0.7769]], requires_grad=True)
```

```
weight = torch.FloatTensor([[1, 2.3, 3], [4, 5.1, 6.3]])
embedding = nn.Embedding.from_pretrained(weight)
embedding.weight
```

```
tensor([[1.0000, 2.3000, 3.0000],
        [4.0000, 5.1000, 6.3000]])
```

```
input = torch.LongTensor([1])
embedding(input)
```

```
tensor([[4.0000, 5.1000, 6.3000]])
```

torch.nn.Sequential

torch.nn.Sequential(*args: Module)

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

forward

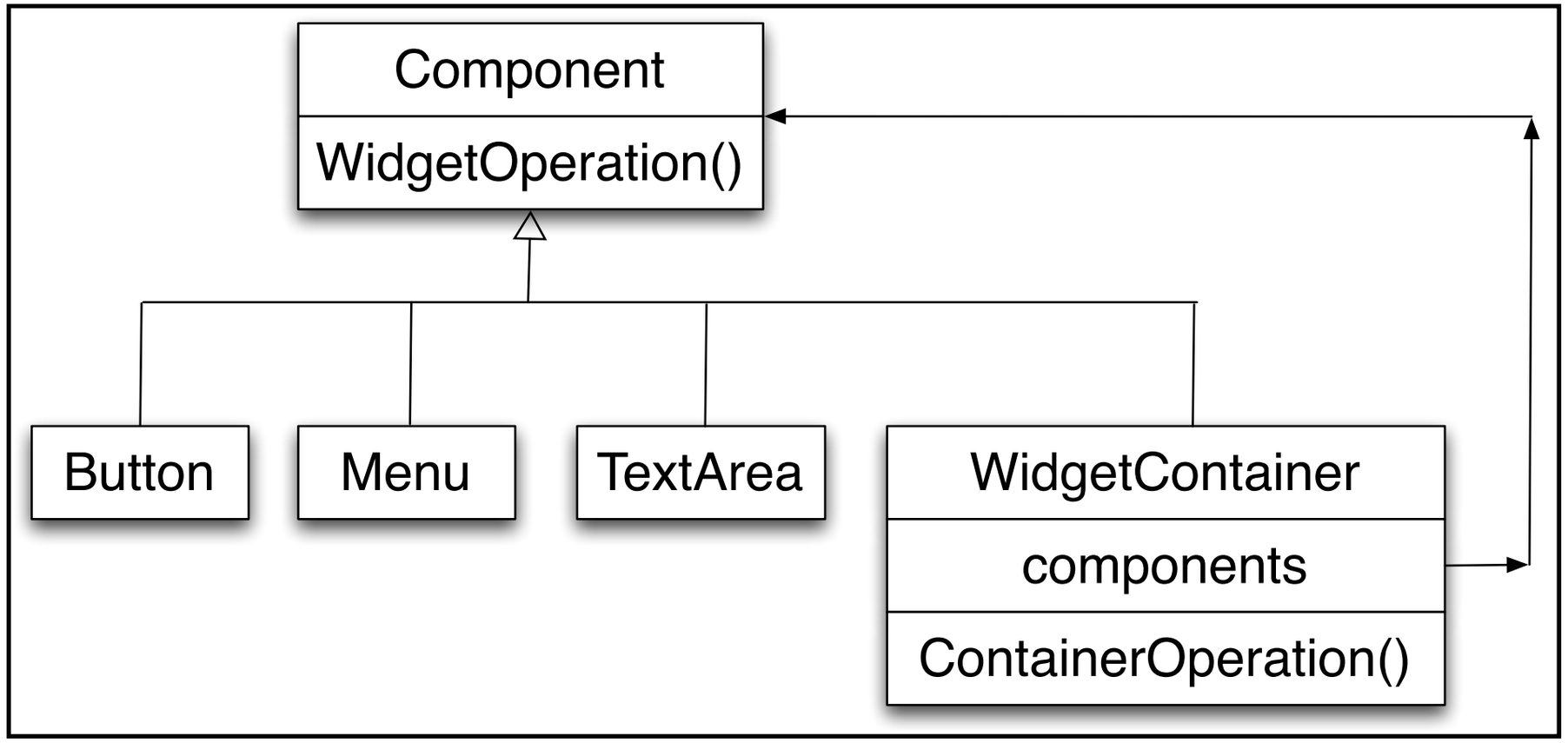
Input is passed to first module

It output is passed to second module

Etc

```
self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg)
              for _ in range(cfg["n_layers"])]
        )
```

Composite Pattern

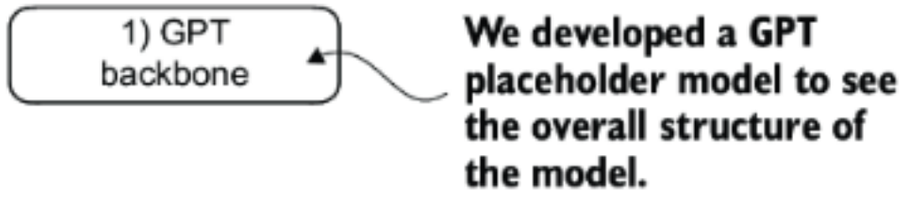


Intent

Compose objects into tree structures to represent part-whole hierarchies.

Lets clients treat individual objects and compositions of objects uniformly

DummyGPTModel

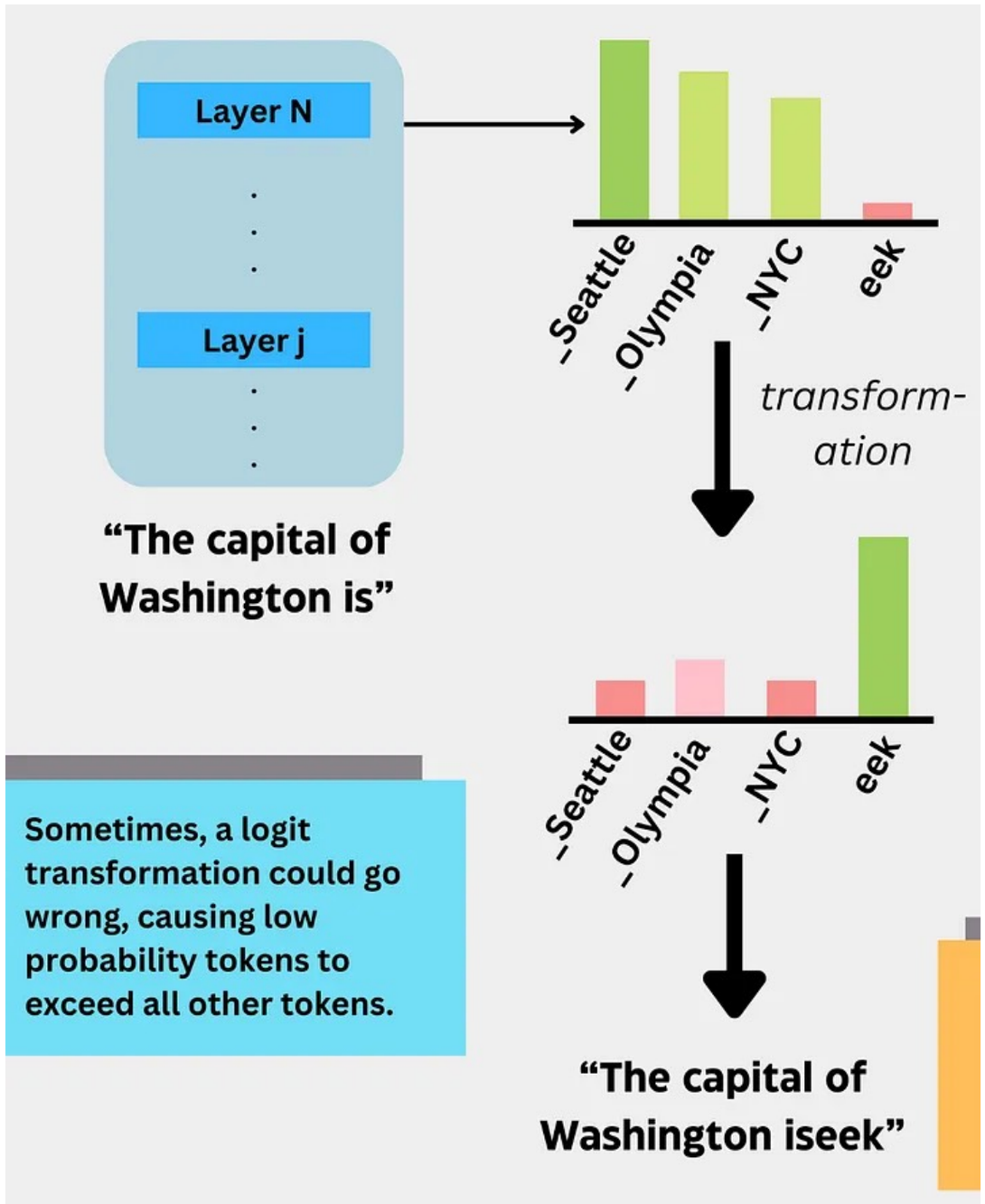



```
def forward(self, in_idx):
    batch_size, seq_len = in_idx.shape
    tok_embeds = self.tok_emb(in_idx)
    pos_embeds = self.pos_emb(
        torch.arange(seq_len, device=in_idx.device)
    )
    x = tok_embeds + pos_embeds
    x = self.drop_emb(x)
    x = self.trf_blocks(x)
    x = self.final_norm(x)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

logits are the raw, unnormalized output of the model's final layer before it's converted into probabilities.

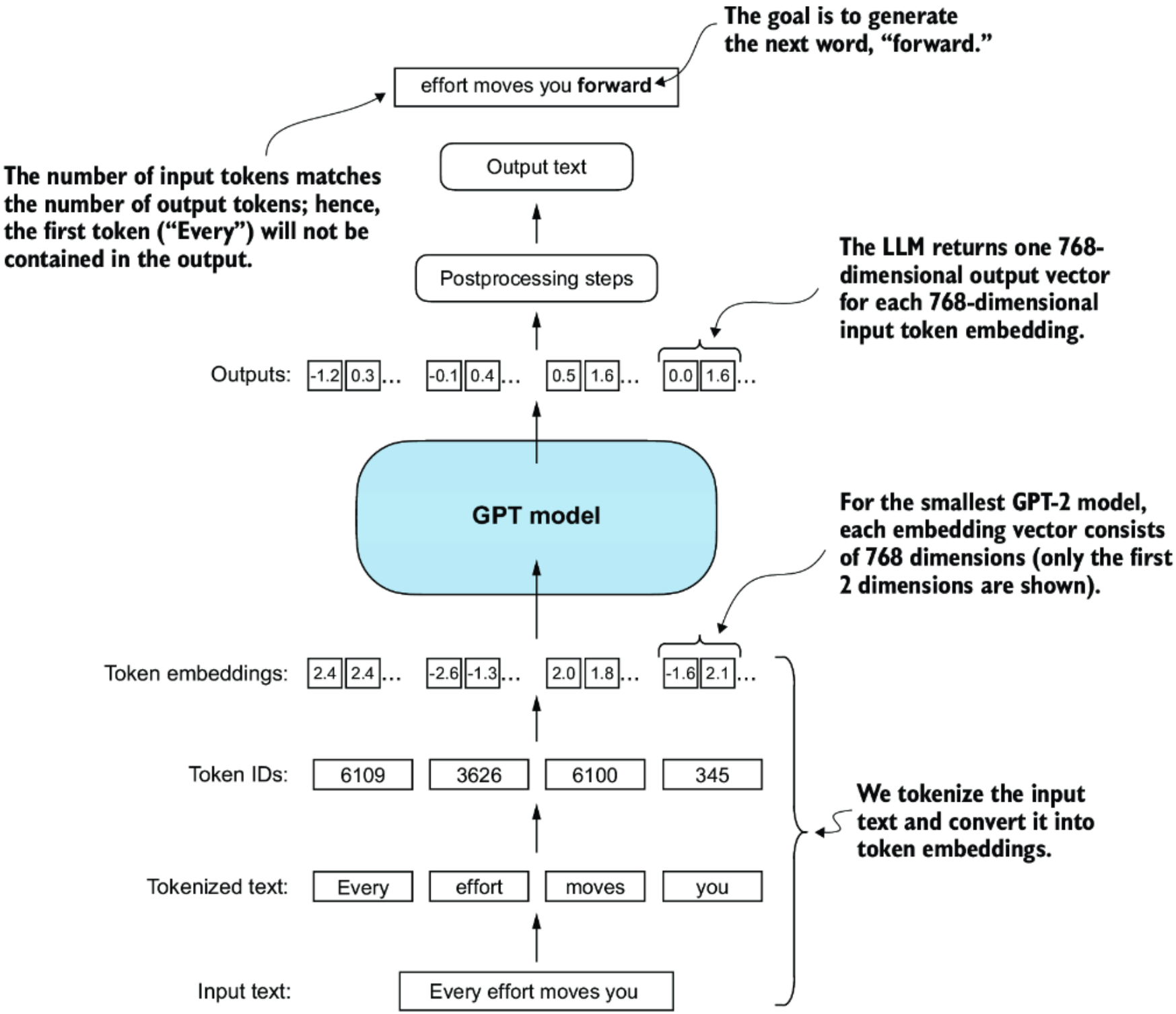
My LLM's outputs got 1000% better with this simple trick. Nikhil Anand

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"


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

The DummyGPTModel Works! (But not Well)

```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Goodby and thanks for all"
txt2 = "Behind every great man there is"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
```

```
tensor([[10248,  1525,   290,  5176,   329,   477],
        [34163,   790,  1049,   582,   612,   318]])
```

```
torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
```

```
tensor([[[ 2.0393e-01,  3.4299e-02,  1.6361e-
           2.3954e-01,  4.5407e-01],
         [-9.7754e-01,  1.5108e+00, -2.4300e-01
           9.5354e-01,  8.8476e-01],
         [-4.1408e-01,  4.0488e-01,  2.4798e+00
           2.0347e+00,  4.4232e-01],
         [-9.2830e-01,  9.5038e-01, -4.1242e-02
          -1.2613e-01, -5.9409e-01],
         [ 1.5327e-02, -1.3722e+00,  5.7270e-01
           3.2733e-01, -4.1644e-02],
         [ 1.4387e-01, -1.6002e-05, -4.8986e-01
```

Vanishing and Exploding Gradients

$$\frac{\partial L}{\partial x_0} = \frac{\partial L}{\partial x_n} \prod_{i=1}^{n} \frac{\partial x_i}{\partial x_{i-1}}$$

Normalize to help prevent vanishing/exploding

LayerNorm

Subtracts mean and divides by variance.

Original Transformer, BERT, GPT-3.

zero-centered distribution

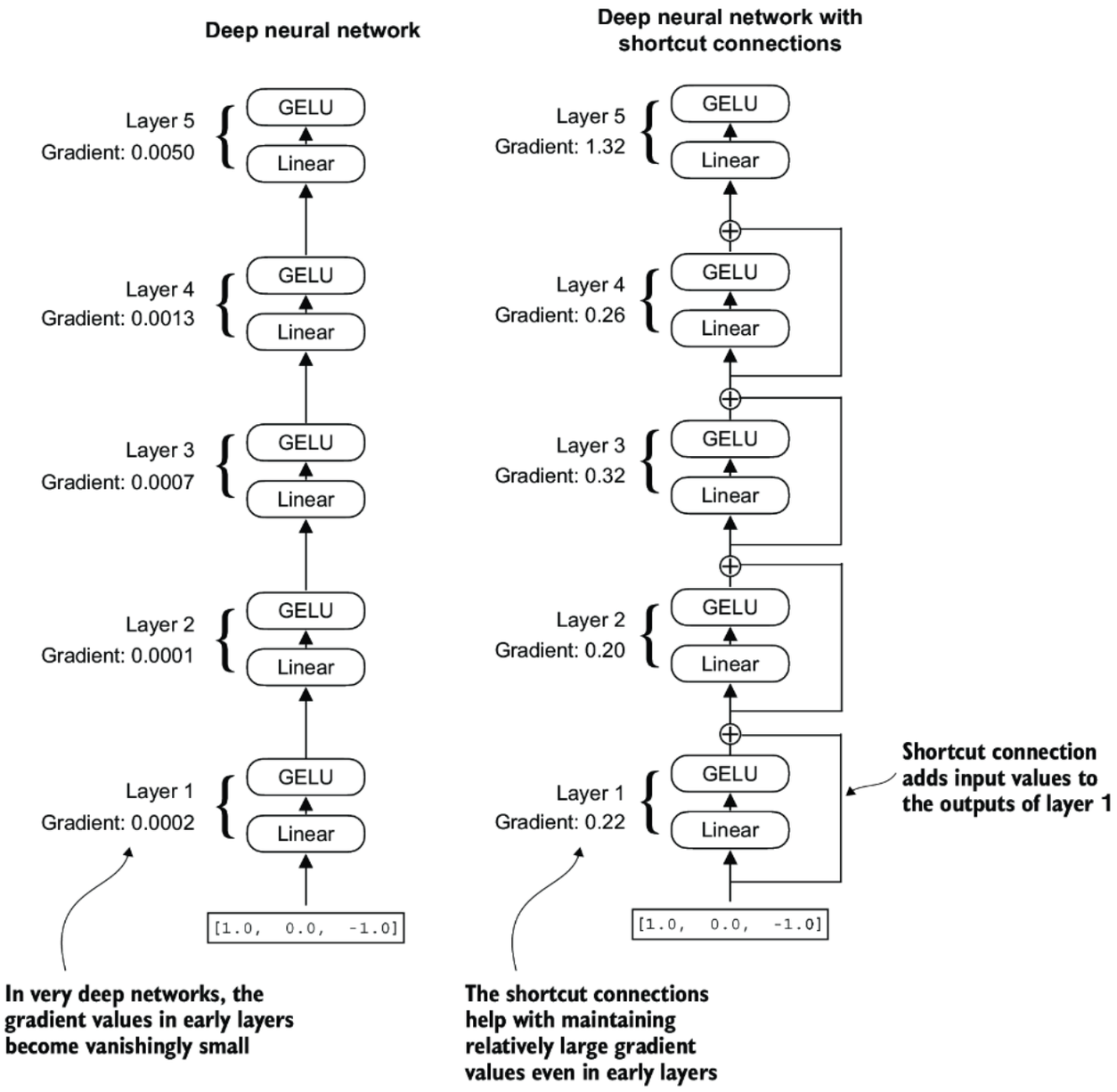
RMSNorm

Divides by the Root Mean Square (RMS)

Llama 2, Llama 3, Mistral, Gopher

Faster

Shortcut (skip, residual) Connections



Placement of Norm

Before or after adding the residual connection

After

$$Output = x + Sublayer(LayerNorm(x))$$

Much more stable

Can use higher learning rates

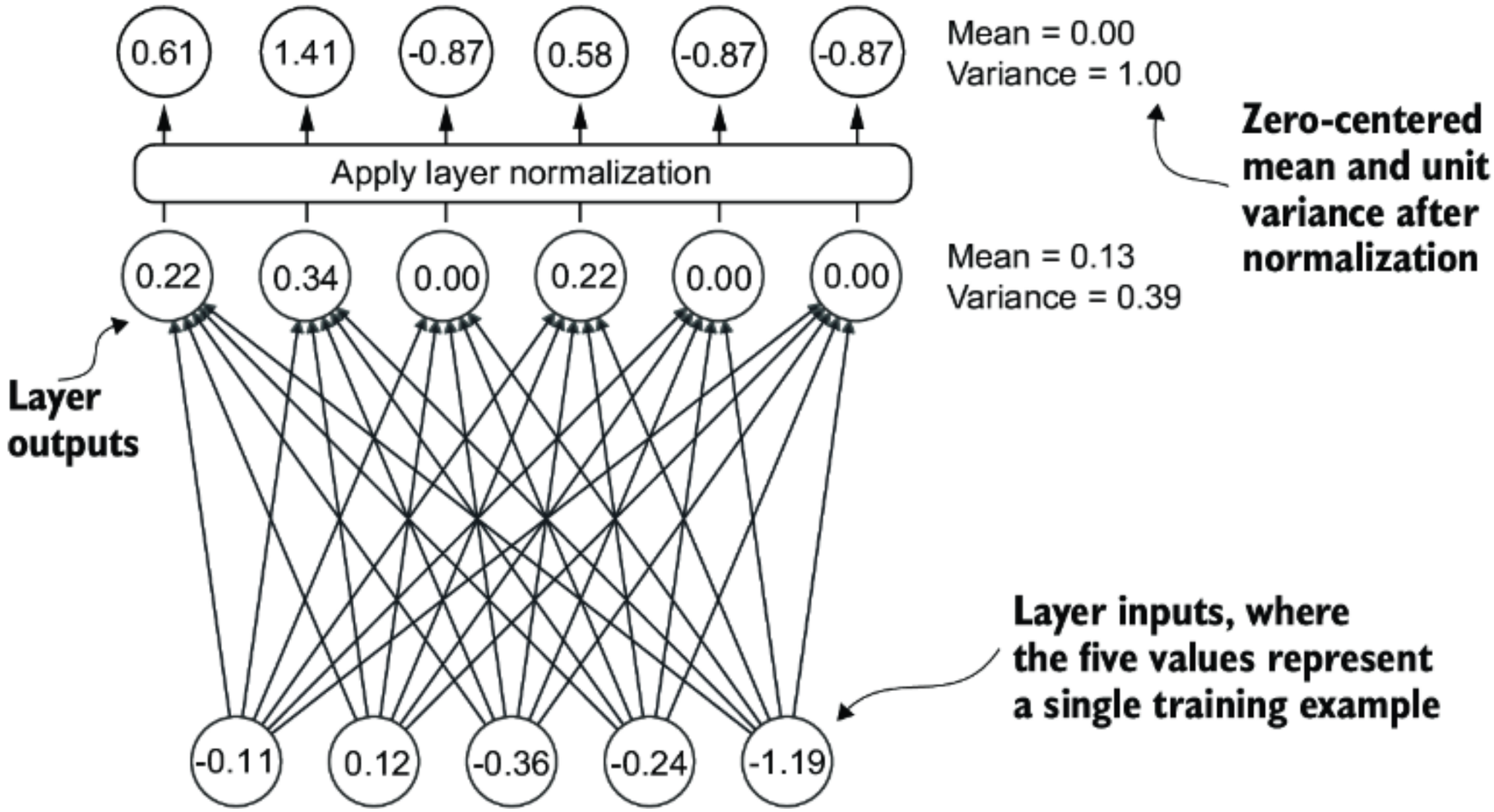
GPT-2/3, BERT-large
Most modern LLMs

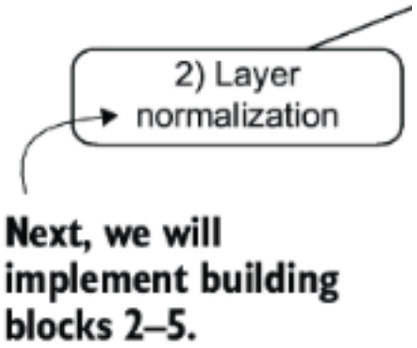
Before

$$Output = LayerNorm(x + Sublayer(x))$$

Better performance if trained correctly

Very unstable at high depths


0.61
1.41
-0.87
0.58
-0.87
-0.87
Apply layer normalization
0.22
0.34
0.00
0.22
0.00
0.00
-0.11
0.12
-0.36
-0.24
-1.19
Mean = 0.00
Variance = 1.00
Zero-centered mean and unit variance after normalization
Mean = 0.13
Variance = 0.39
Layer outputs
Layer inputs, where the five values represent a single training example

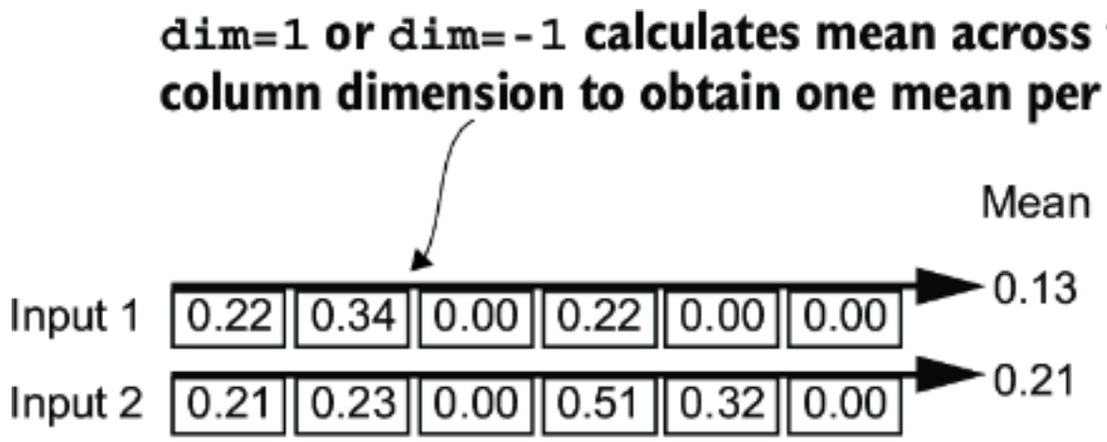


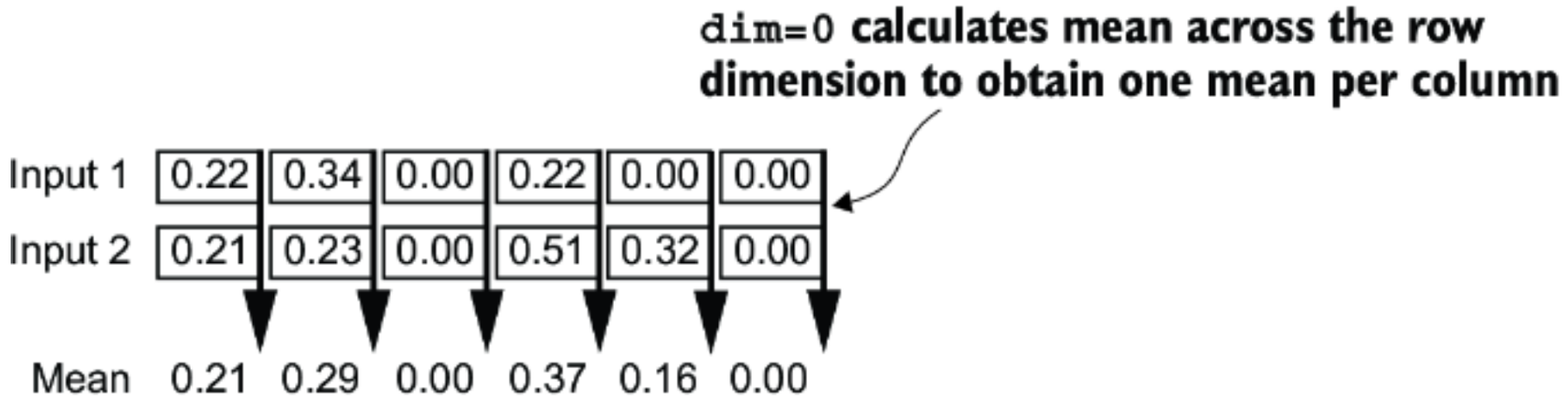

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    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
```

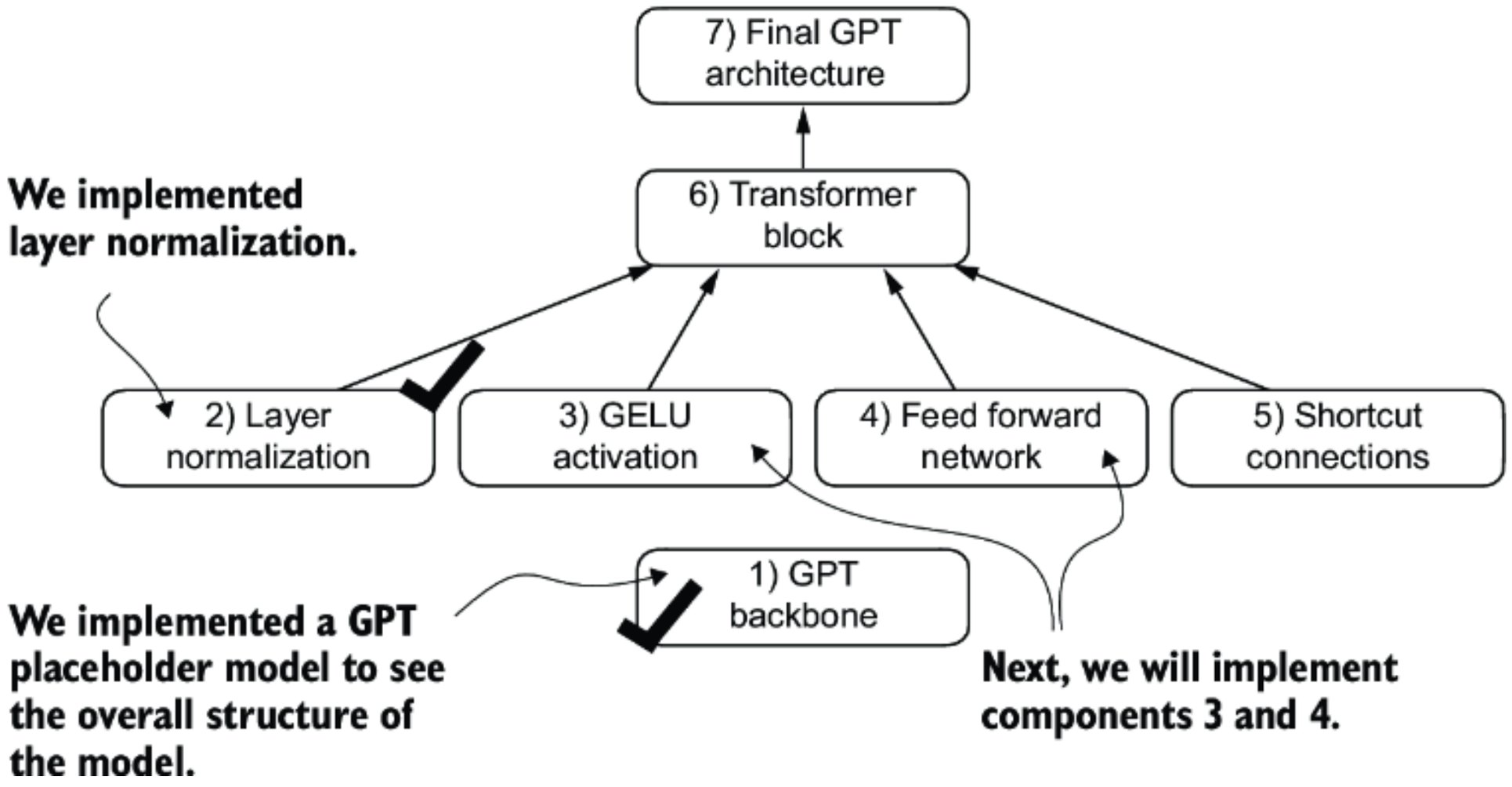
nn.Parameter

marks a tensor as a learnable parameter

mean = x.mean(**dim=-1**, keepdim=True)
var = x.var(**dim=-1**, keepdim=True, unbiased=False)






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.

Why Activation Layer

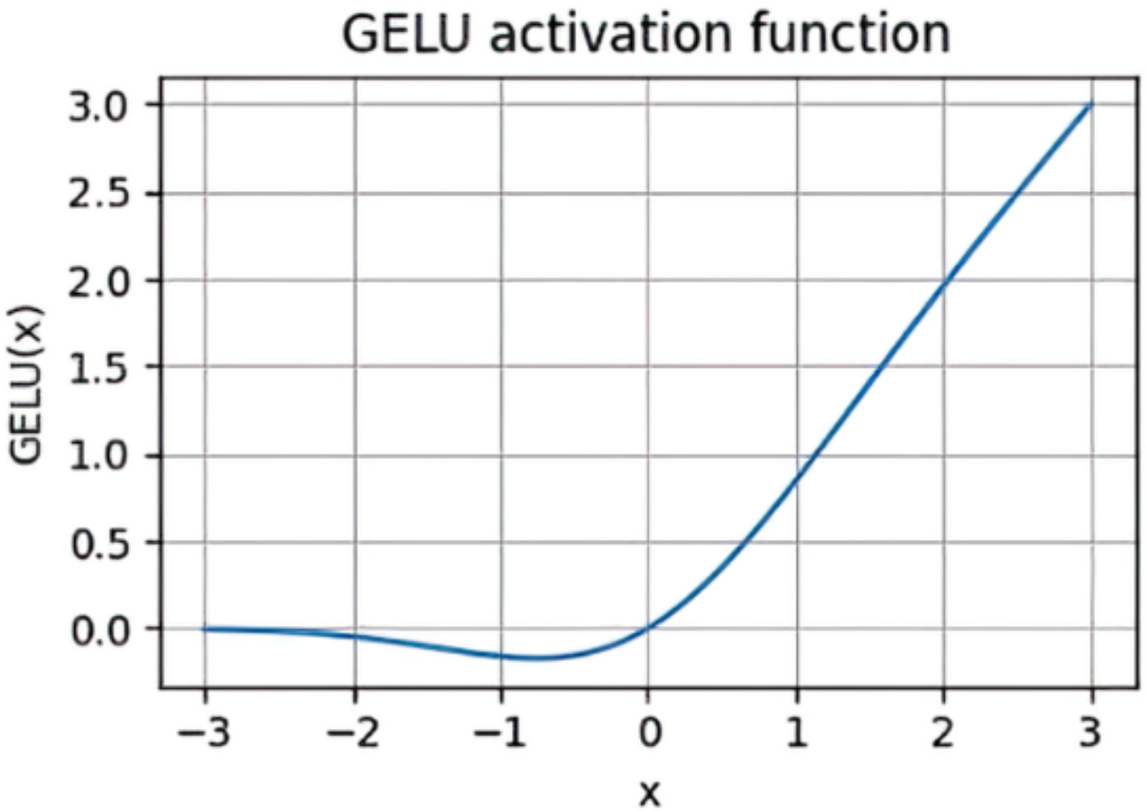
Need a non-linear layer to avoid LLM acting like a single layer

Universal Approximation Theorem (UAT)

> A neural network with just one hidden layer and a nonlinear activation function can approximate any continuous function, as closely as you want — given enough neurons

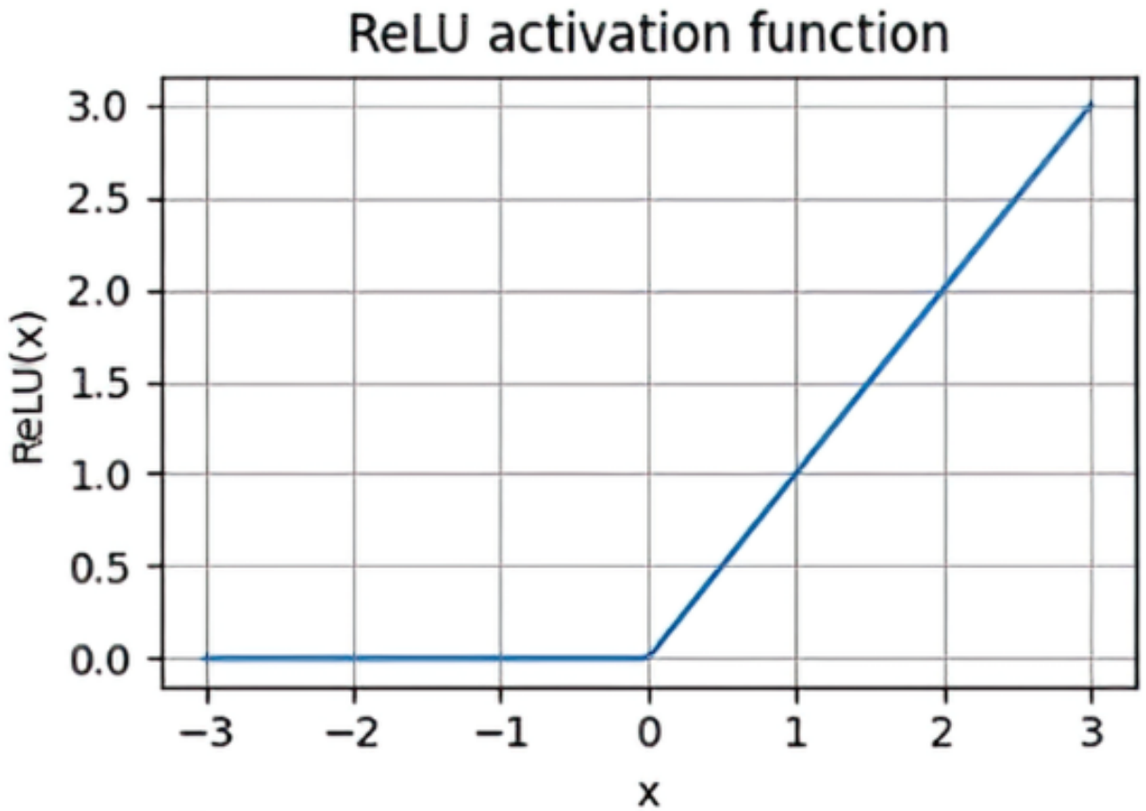
GELU & ReLU

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



Differentiable everywhere

GPT-3, Llama



Easy to compute

If the value becomes negative, it dies

Early models

SiLU - Sigmoid Linear Unit (Swish)

$$f(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

Used in Llama 3, Mistral, Gemma

Self-gated activation

x - information

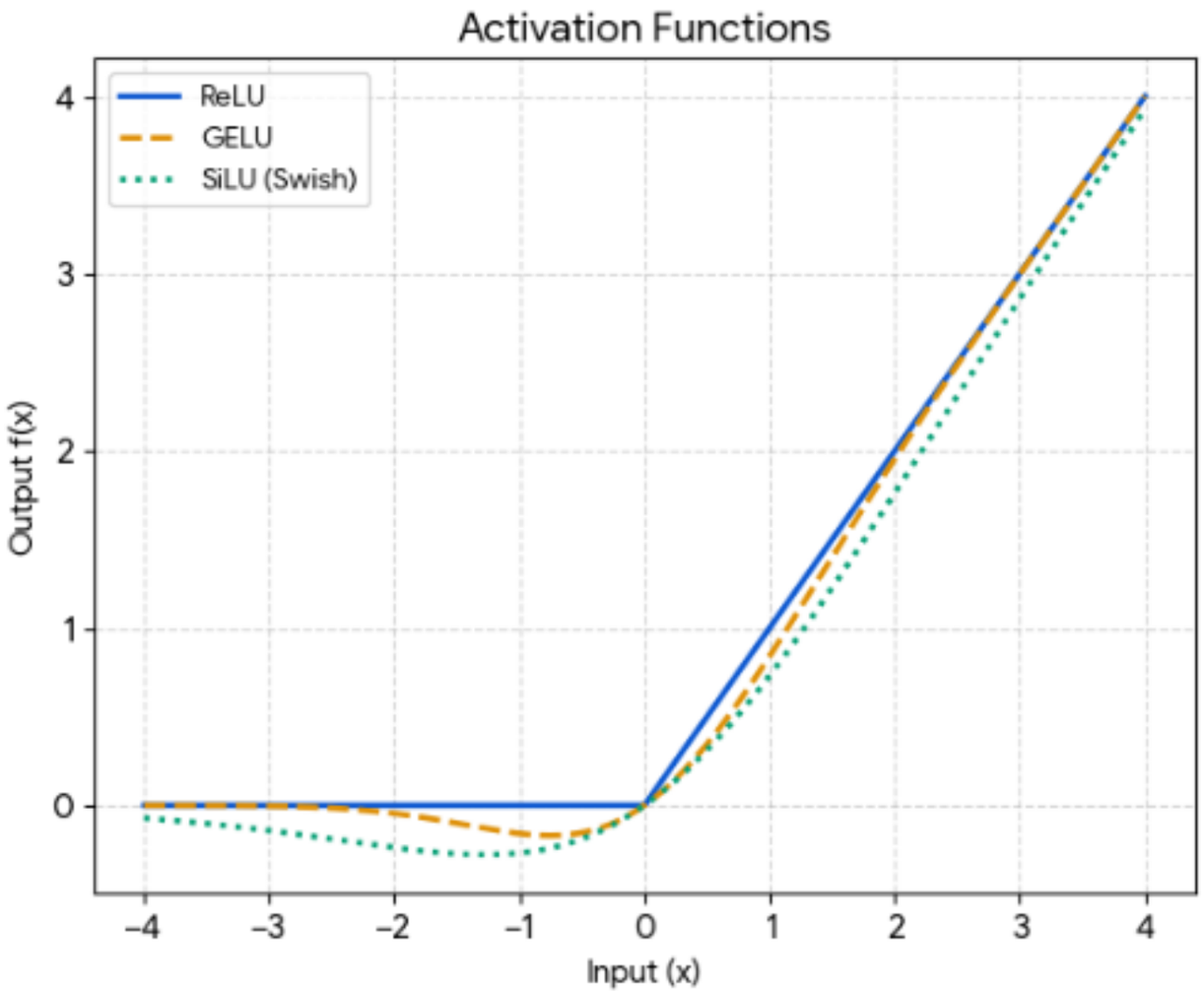
$\sigma(x)$ - gate

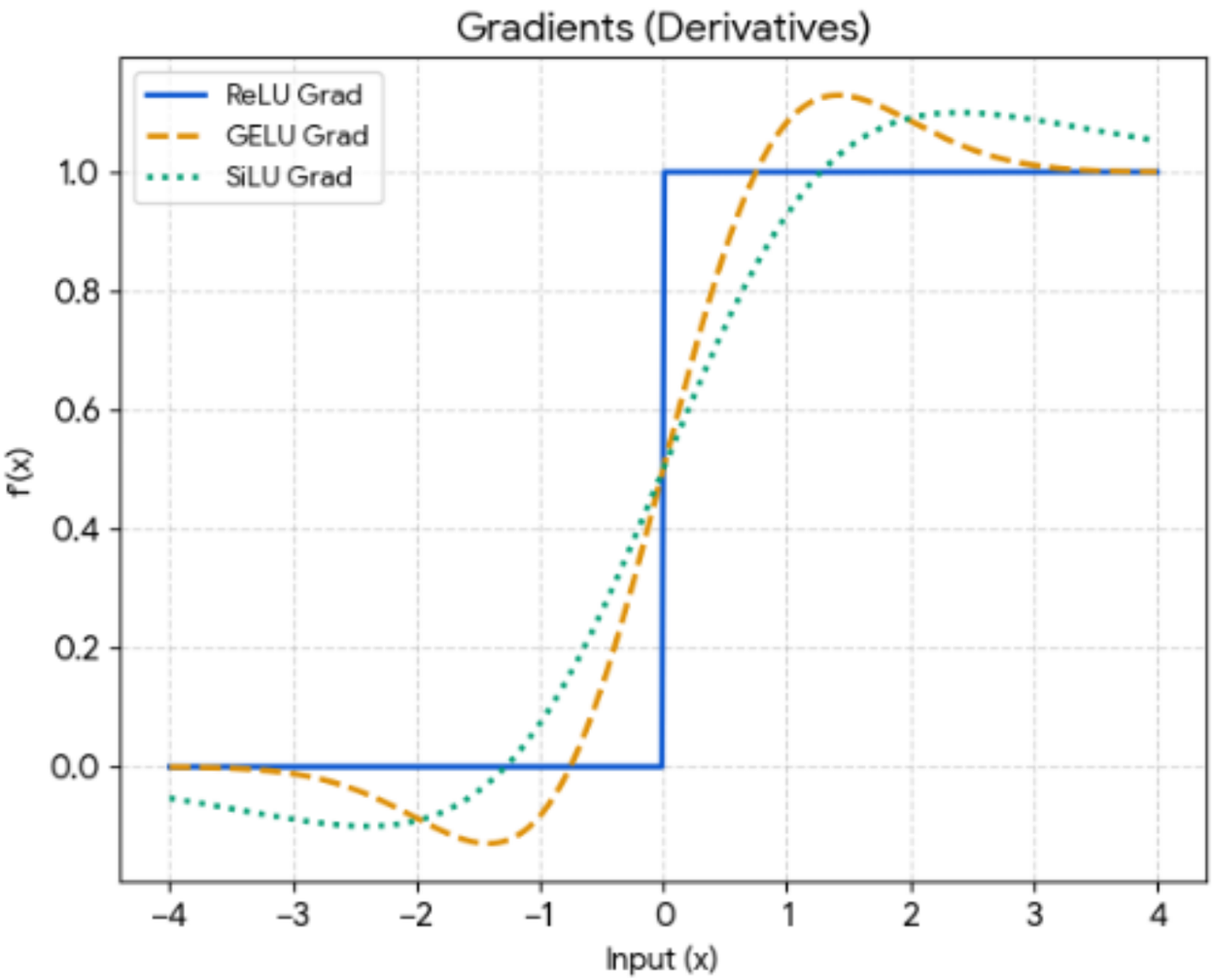
x is large gate is open

x is very negative gate is closed

Amplifies very strong signals, suppresses if negative

ReLU vs GELU vs SLU





Source Gemini

SwiGLU Gated Linear Unit

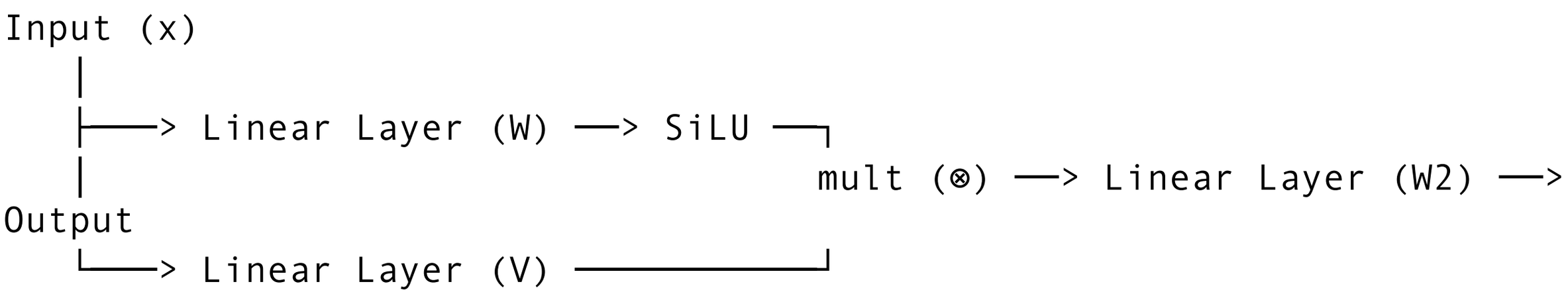
$$SwiGLU(x, W, V, U) = (\text{SiLU}(xW) \otimes xV)U$$

$\otimes$: Element-wise multiplication (Hadamard product)

W & V are learned independently, allowing more complex logic

Signal branch might extract "grammatical gender" features

Gate only opens if language uses gendered nouns



$$SwiGLU(x) = (SiLU(xW) \otimes xV)U$$

The Expansion Phase (W and V)

x dimension d_{model}

d_{ff} dimension $4d_{model}$

xW and xV both result in tensors of size d_{ff}

W and V project to larger space

The Gating Phase ($\otimes$)

$(SiLU(xW) \otimes xV)$ Dimension d_{ff}

The Reduction Phase (U)

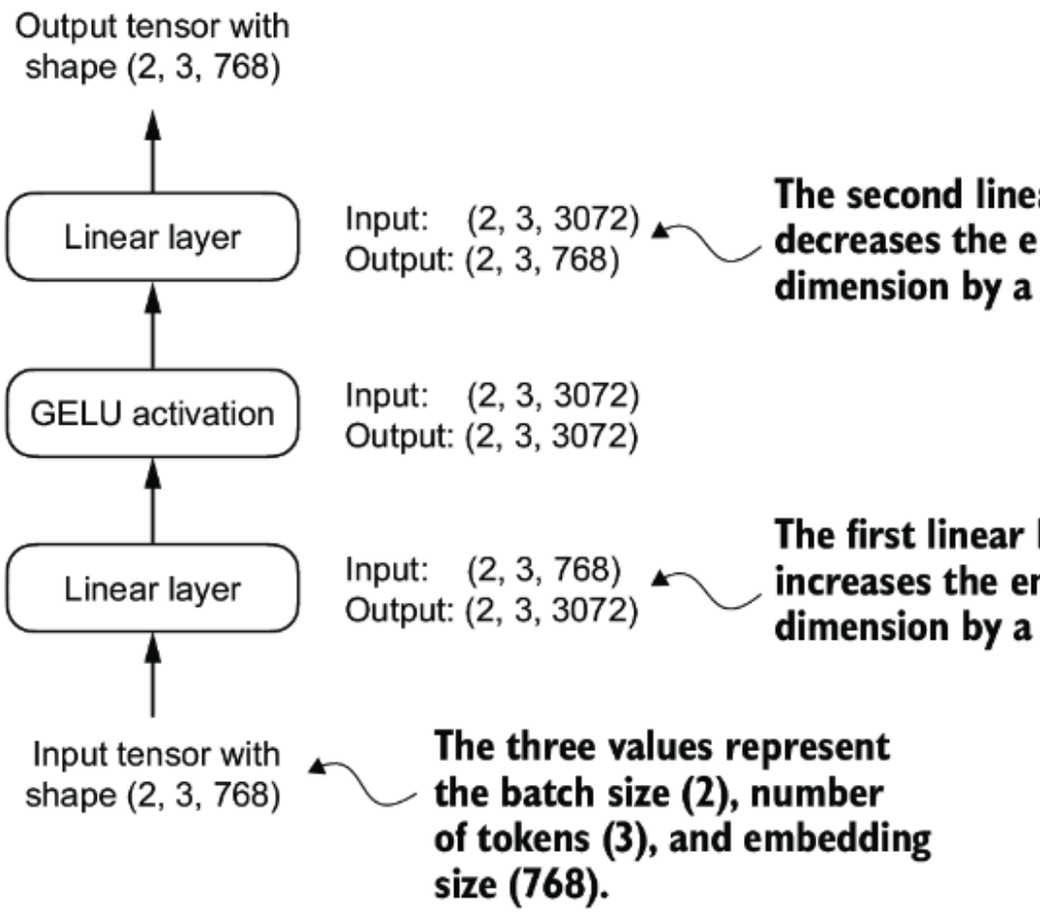
U projects back to dimension d_{model}

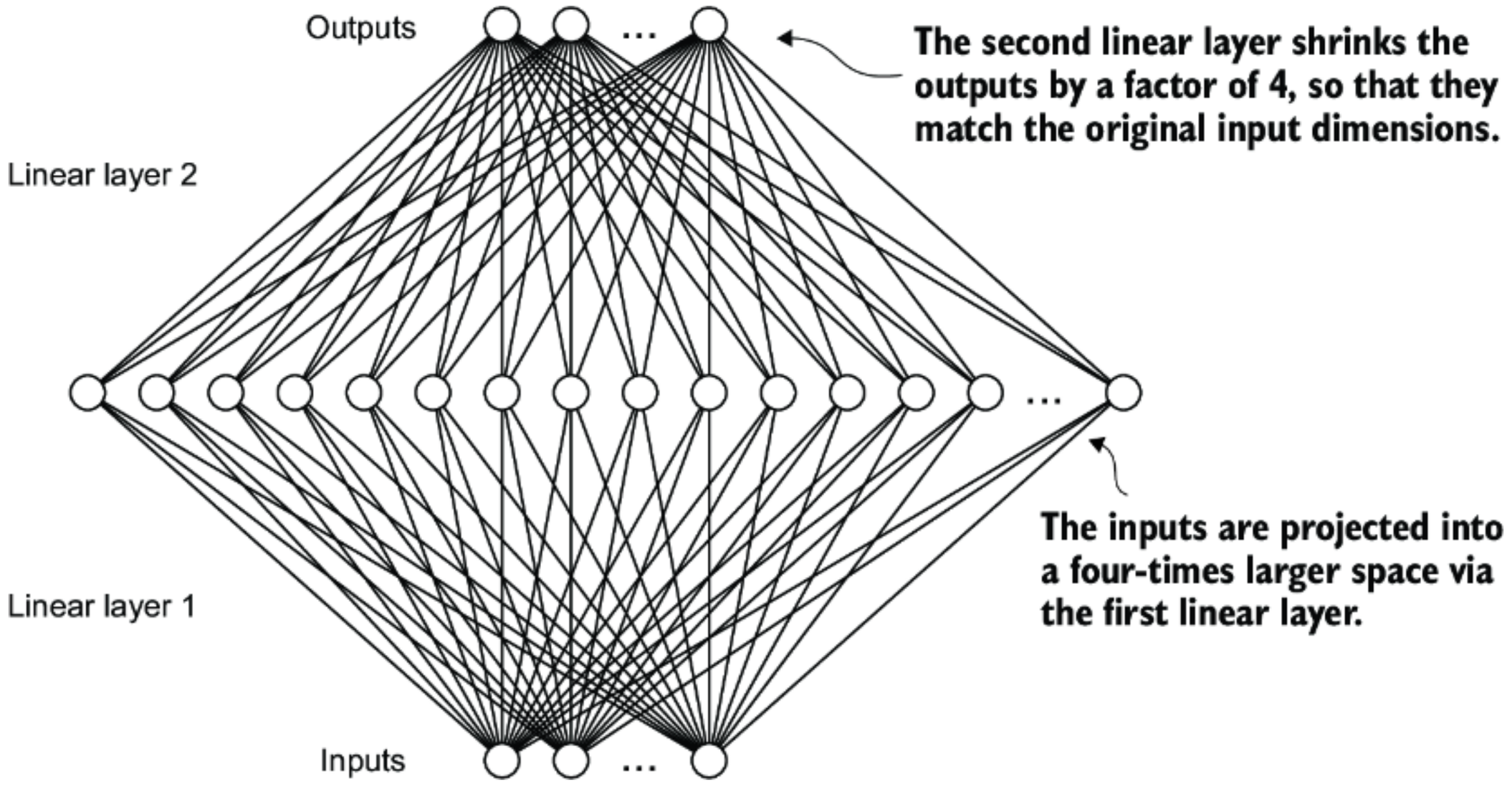
```
import torch.nn as nn
import torch.nn.functional as F

class SwiGLU(nn.Module):
    def __init__(self, d_model: int, d_ff: int):
        super().__init__()
        # SwiGLU requires two linear projections for the gate mechanism
        self.w1 = nn.Linear(d_model, d_ff, bias=False) # Linear projection 1
        self.w2 = nn.Linear(d_model, d_ff, bias=False) # Linear projection 2 (Gate)
        self.w3 = nn.Linear(d_ff, d_model, bias=False) # Down projection

    def forward(self, x):
        gate = F.silu(self.w1(x))
        res = gate * self.w2(x)
        return self.w3(res)
```

```
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))
    ))
```

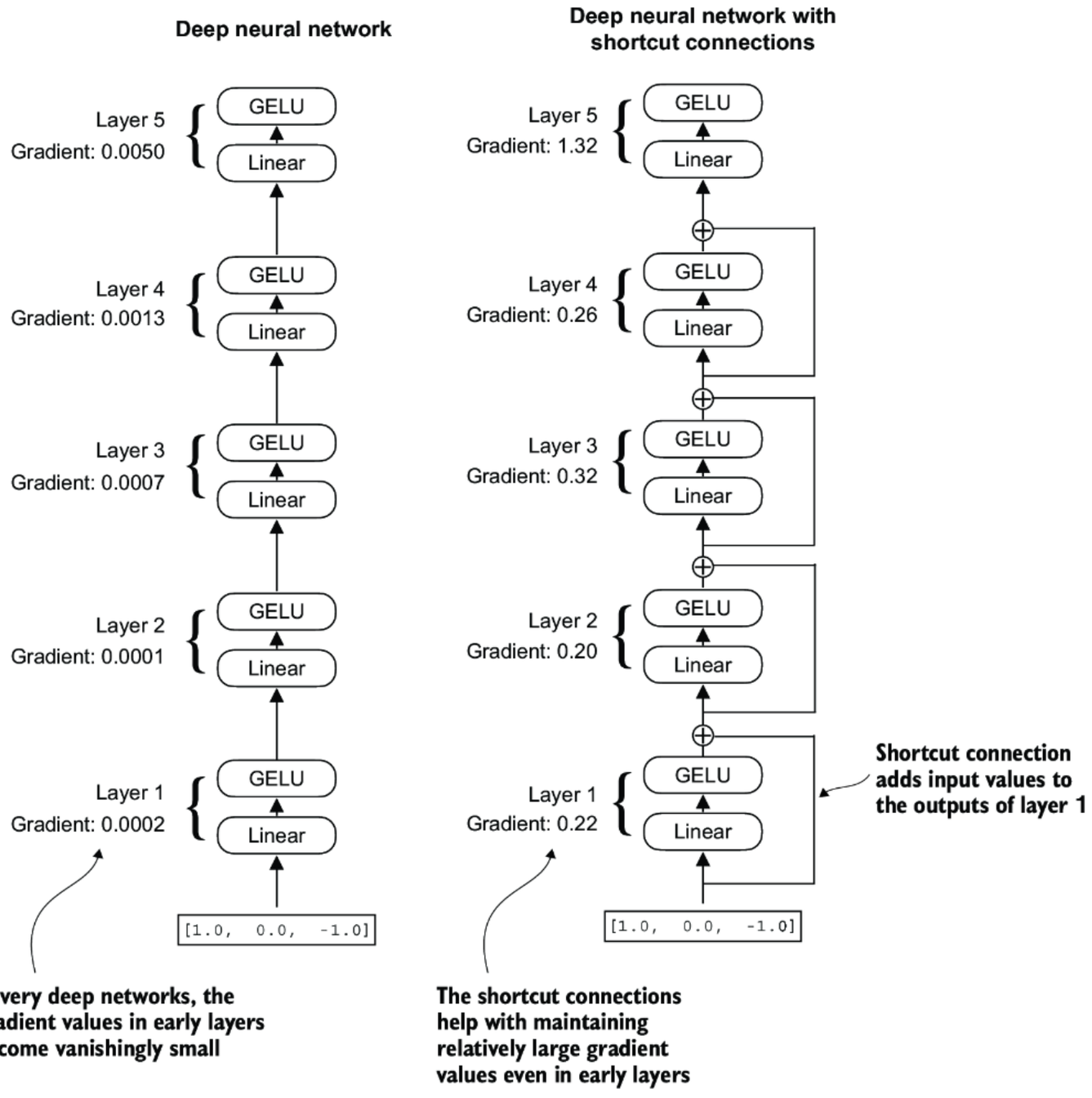
$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

```
class 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



Why Short Cut

With long stack gradients can vanish or blow up

Allows later layers to reuse earlier representations directly,
Preserving low-level information

Higher layers add abstract refinements.

The residual adds the pre-normalized input back after the transformation, keeping the overall scale and distribution more stable across layers.

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())
])
```

Shortcut Connections

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

To show the effect of shortcut connections

Instantiate the Model

```
layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123)                          #1
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)
```

Backtracking

```
def print_gradients(model, x):
    output = model(x)
    target = torch.tensor([[0.]])

    loss = nn.MSELoss()
    loss_tensor = loss(output, target)

    loss_tensor.backward()
```

```
for name, param in model.named_parameters():
    if 'weight' in name:
        print(f"{name} has gradient mean of {param.grad.abs().mean().item()}")
```

```
print_gradients(model_without_shortcut, sample_input)
```

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.00012011161015834666
layers.2.0.weight has gradient mean of 0.0007152041071094573
layers.3.0.weight has gradient mean of 0.0013988735154271126
layers.4.0.weight has gradient mean of 0.00504964543506503 1
```

```
print_gradients(model_out_shortcut, sample_input)
```

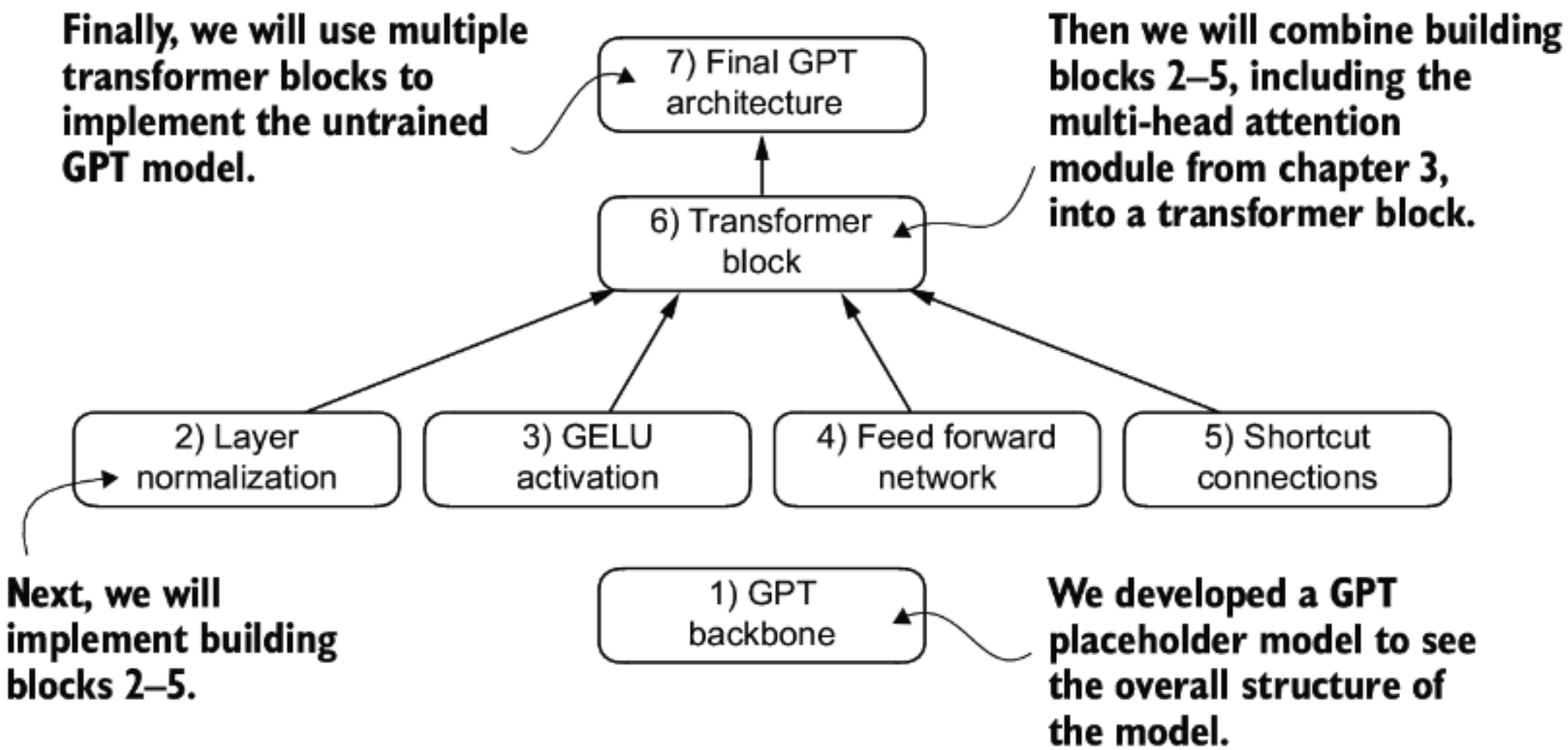
```
layers.0.0.weight has gradient mean of 0.22169792652130127
layers.1.0.weight has gradient mean of 0.20694105327129364
layers.2.0.weight has gradient mean of 0.3289699554443359 4
layers.3.0.weight has gradient mean of 0.266573250293731 7
layers.4.0.weight has gradient mean of 1.3258541822433472
```

nn.MSELoss

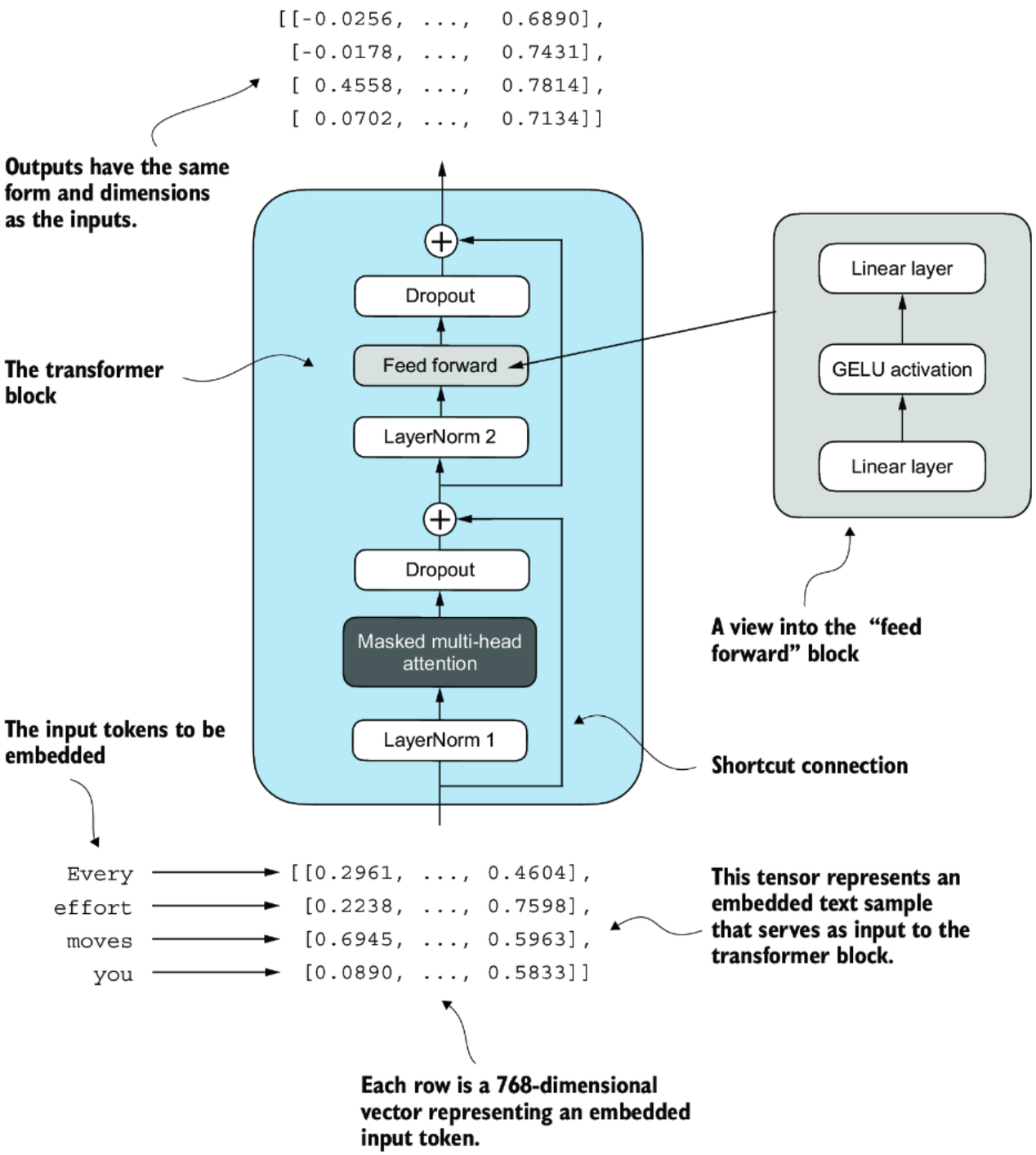
mean squared error (squared L2 norm) between
each element in the input x and target y

Existing Loss Functions (21)

nn.L1Loss	Creates a criterion that measures the mean absolute error (MAE)
nn.MSELoss	Creates a criterion that measures the mean squared error (squared L2 norm)
nn.CrossEntropyLoss	This criterion computes the cross entropy loss between input logits and target.
nn.CTCLoss	The Connectionist Temporal Classification loss.
nn.NLLLoss	The negative log likelihood loss.
nn.PoissonNLLLoss	Negative log likelihood loss with Poisson distribution of target.
nn.GaussianNLLLoss	Gaussian negative log likelihood loss.
nn.KLDivLoss	The Kullback-Leibler divergence loss.
nn.BCELoss	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
nn.BCEWithLogitsLoss	This loss combines a Sigmoid layer and the BCELoss in one single class.
nn.MarginRankingLoss	
nn.HingeEmbeddingLoss	Measures the loss given an input tensor x and a labels tensor y (containing 1 or -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.

Transformer Block



TransformerBlock

```
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):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut

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

Full Model

```
class GPTModel(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(
        *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

    self.final_norm = LayerNorm(cfg["emb_dim"])
    self.out_head = nn.Linear(
        cfg["emb_dim"], cfg["vocab_size"], bias=False
    )
```

Full 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
```

Put it all together

Model is 152 lines of code

```
def main():
    GPT_CONFIG_124M = {… }

    torch.manual_seed(123)
    model = GPTModel(GPT_CONFIG_124M)
    model.eval()  # disable dropout

    start_context = "Hello, I am"

    tokenizer = tiktoken.get_encoding("gpt2")
    encoded = tokenizer.encode(start_context)
    encoded_tensor = torch.tensor(encoded).unsqueeze(0)

    out = generate_text_simple(
        model=model,
        idx=encoded_tensor,
        max_new_tokens=10,
        context_size=GPT_CONFIG_124M["context_length"]
    )
    decoded_text = tokenizer.decode(out.squeeze(0).tolist())
```

Put it all together

```
def generate_text_simple(model, idx, max_new_tokens, context_size):
    # idx is (B, T) array of indices in the current context
    for _ in range(max_new_tokens):

        # Crop current context if it exceeds the supported context size
        idx_cond = idx[:, -context_size:]

        # Get the predictions
        with torch.no_grad():
            logits = model(idx_cond)

        # Focus only on the last time step
        # (batch, n_token, vocab_size) becomes (batch, vocab_size)
        logits = logits[:, -1, :]

        # Get the idx of the vocab entry with the highest logits value
        idx_next = torch.argmax(logits, dim=-1, keepdim=True)  # (batch, 1)

        # Append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)  # (batch, n_tokens+1)

    return idx
```

Put it all together

==========

IN

==========

Input text: Hello, I am

Encoded input text: [15496, 11, 314, 716]

encoded_tensor.shape: torch.Size([1, 4])

==========

OUT

==========

Output: tensor([[15496, 11, 314, 716, 27018, 24086, 47843, 30961, 42348, 7267,
49706, 43231, 47062, 34657]])

Output length: 14

Output text: Hello, I am Featureiman Byeswickattribute argue logger Normandy Compton analogous