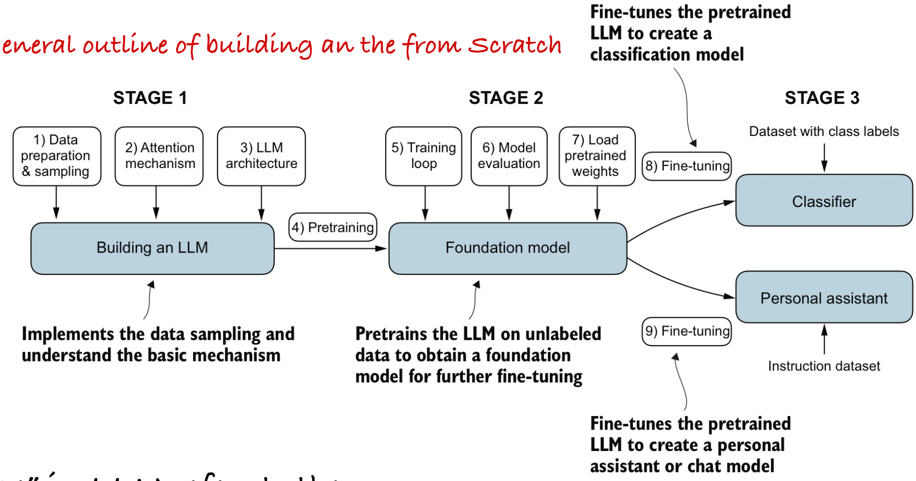
Build an LLM from Scratch

all images from the book by Sebastian Raschka

the general outline of building an the from Scratch



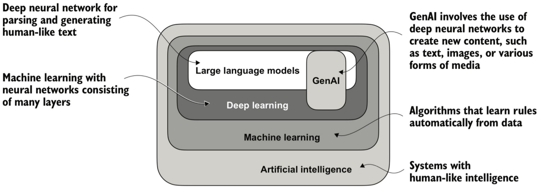
Fine-tunes the pretrained LLM to create a classification model

Implements the data sampling and understand the basic mechanism

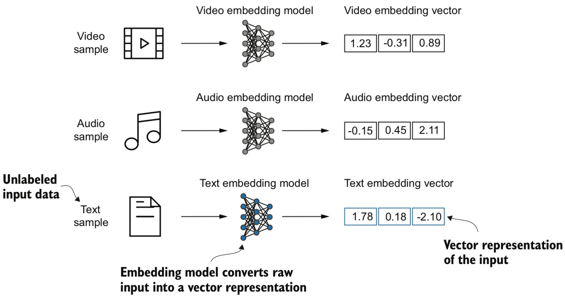
Pretrains the LLM on unlabeled data to obtain a foundation model for further fine-tuning

Fine-tunes the pretrained LLM to create a personal assistant or chat model

the "large" in LLM refers to the massive size of the models and the huge corpus of data used for them.


Deep neural network for parsing and generating human-like text
Machine learning with neural networks consisting of many layers
Large language models
GenAI
Deep learning
Machine learning
Artificial intelligence
GenAI involves the use of deep neural networks to create new content, such as text, images, or various forms of media
Algorithms that learn rules automatically from data
Systems with human-like intelligence

the next-word prediction task is a form of self-supervised learning. The label is the next word /token.

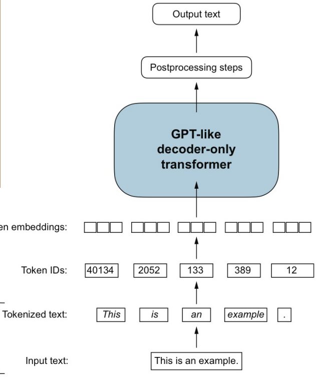


embedding is to convert data into numerical tensors.

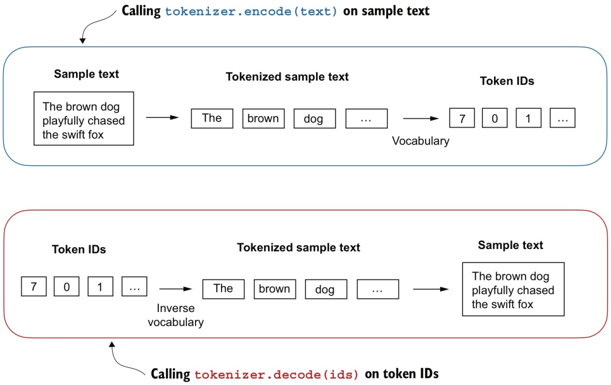
- a good choice for embedding is word2vec
- good embedding must capture the context
- LLMS can train their own embedding as part of their training, this allows them to capture meanings and concepts based on the task and data at hand.

Ready Data 4 Embedding

- Convert large text to words
- assign ids to each token
- add Special & unknown tokens



This section covers the concept of splitting text into tokens


Calling tokenizer.encode(text) on sample text
Sample text
The brown dog playfully chased the swift fox
Tokenized sample text
The
brown
dog
...
Token IDs
7
0
1
...
Vocabulary
Token IDs
7
0
1
...
Inverse vocabulary
Tokenized sample text
The
brown
dog
...
Sample text
The brown dog playfully chased the swift fox
Calling tokenizer.decode(ids) on token IDs

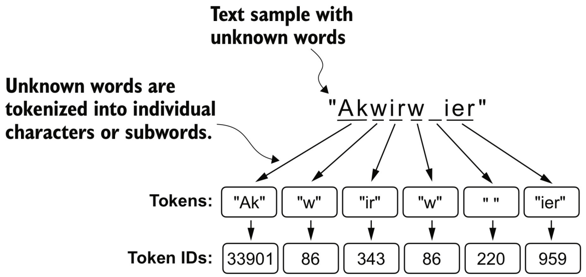
Some common special tokens:

[BOS] marks the beginning of a sequence/text

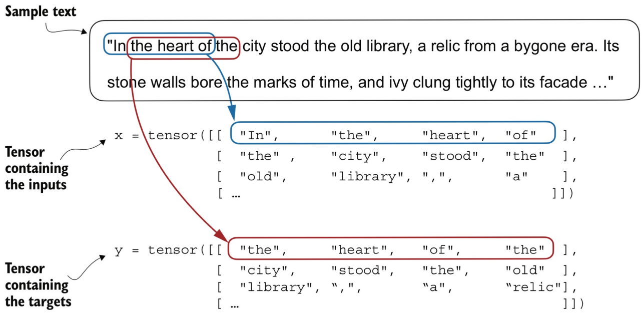
[EOS] " " ending of a " "

[PAD] padding token for training on batches with texts of various sizes.

Some tokenizers like BPE can encode unknown words without using special tokens, by breaking the word down to tokenizable parts.


Text sample with unknown words
Unknown words are tokenized into individual characters or subwords.
"Akwirw ier"
Tokens:
"Ak"
"w"
"ir"
"w"
" "
"ier"
Token IDs:
33901
86
343
86
220
959

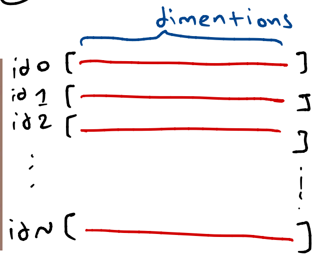
How the LLM processes input/output pairs


Sample text
"In the heart of the city stood the old library, a relic from a bygone era. Its stone walls bore the marks of time, and ivy clung tightly to its facade ..."
Tensor containing the inputs
Tensor containing the targets

```
x = tensor([[ "In",     "the",     "heart",   "of"   ],
            [ "the" ,   "city",    "stood",   "the"  ],
            [ "old",    "library", ",",       "a"    ],
            [ ...                                   ]])

y = tensor([[ "the",    "heart",   "of",      "the"  ],
            [ "city",   "stood",   "the",     "old"  ],
            [ "library", ",",      "a",       "relic"],
            [ ...                                   ]])
```

the embedding layer is of shape vocab x dim which is essentially a lookup table holding values for each token.



Why Positional Embs?

the embedding layer returns a deterministic value for each id, regardless of its position.

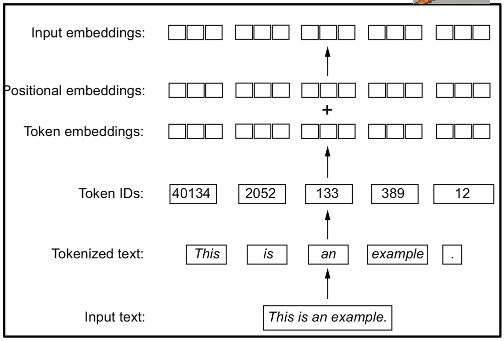
The Attention mechanism is also agnostic to positions: if you shuffle a sequence and feed to it, it doesn't make a difference.

→we inject position information to LLM

the pos embedding is the same Shape as token embedding.

It is added to the token emb.

input embedding = pos emb + token emb



Sequential Transformers ?!

So why don't we use sequential Transformers that can capture the position of tokens and like RNNS and LSTMS, don't require any positional embeddings?

1. Parallelism vs. Sequential Processing: parallel processing is much faster

2. Not a bug, but feature! by allowing the model to process all tokens in parallel, it can learn the relationship and context

3. Complexity & Training Difficulty: learnin also about positions would complicate training

4. Existing approaches work well!

IN summary this would be **too much headache! :'(**

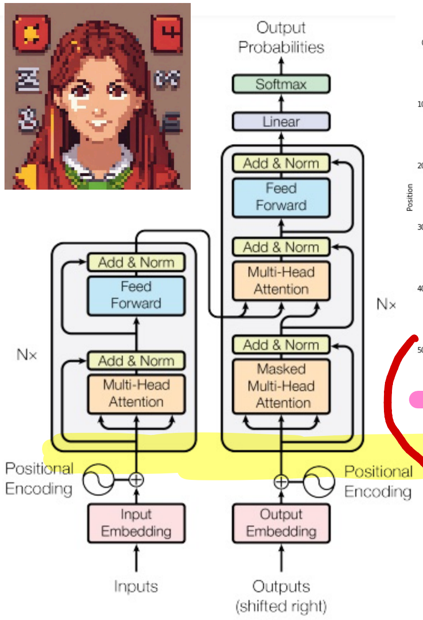
Criteria of a good Pos Embedding

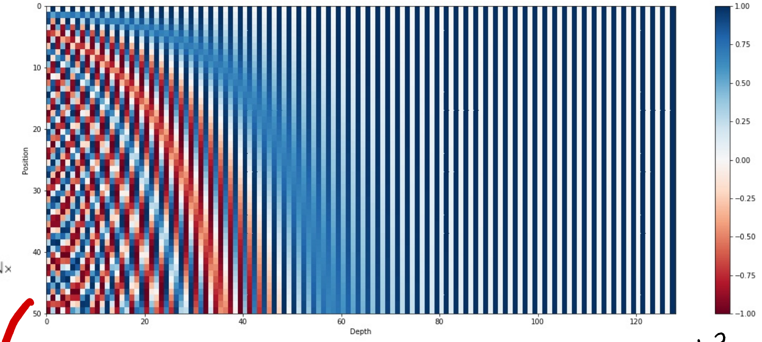
- unique encoding for each position
- distance beetween two time steps be consistent across sentences of different lengths
- should be bounded to generalize to longer Sentences w/ no efforts
- deterministic

the original method by "Attention is all U Need" authors: Sinusoidal

$$\vec{p}_t^{(i)} = \begin{cases} \sin(w_k \cdot t), & \text{if } i = 2k \\ \cos(w_k \cdot t), & \text{if } i = 2k+1 \end{cases} = \begin{bmatrix} \sin(w_1 \cdot t) \\ \cos(w_1 \cdot t) \\ \sin(w_2 \cdot t) \\ \vdots \\ \sin(w_{d/2} \cdot t) \\ \cos(w_{d/2} \cdot t) \end{bmatrix}$$

$$w_k = \frac{1}{10000^{2k/d}}$$



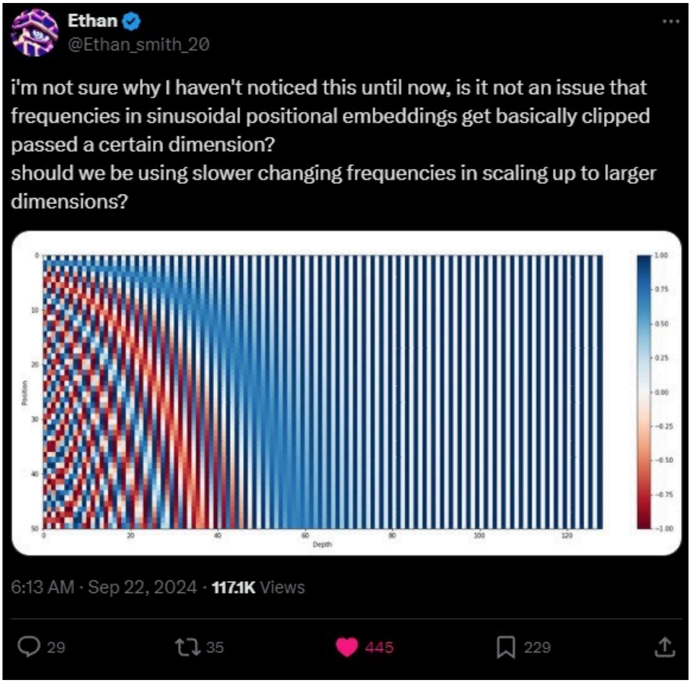


Why is PE summed by TE rather than Concatted?

Concat would mean higher dims in input embedding and more complexity in training and converging.

lower dimension have higher frequency fluctuating more rapidly. So two tokens side by side have more difference in PE rather than tokens far apart. this helps model capture local dependencies while the opposite lower frequency in higher dimensions captures more general & long-ronge dependencies.

Question?

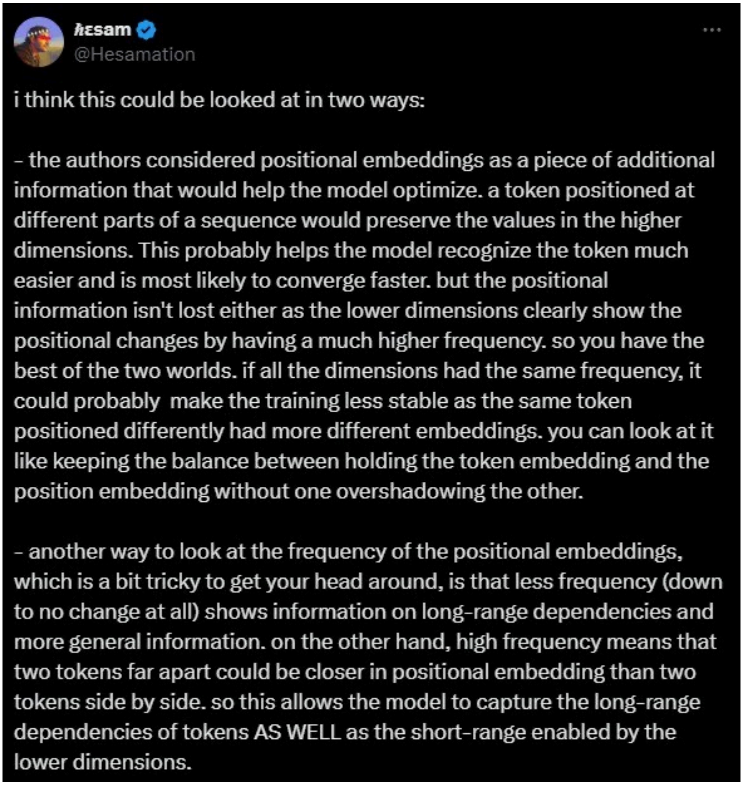


Ethan
@Ethan_smith_20

i'm not sure why I haven't noticed this until now, is it not an issue that frequencies in sinusoidal positional embeddings get basically clipped passed a certain dimension?
should we be using slower changing frequencies in scaling up to larger dimensions?

6:13 AM · Sep 22, 2024 · 117.1K Views

Answer



ℏεsam
@Hesamation

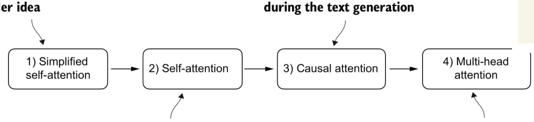
i think this could be looked at in two ways:

- the authors considered positional embeddings as a piece of additional information that would help the model optimize. a token positioned at different parts of a sequence would preserve the values in the higher dimensions. This probably helps the model recognize the token much easier and is most likely to converge faster. but the positional information isn't lost either as the lower dimensions clearly show the positional changes by having a much higher frequency. so you have the best of the two worlds. if all the dimensions had the same frequency, it could probably make the training less stable as the same token positioned differently had more different embeddings. you can look at it like keeping the balance between holding the token embedding and the position embedding without one overshadowing the other.

- another way to look at the frequency of the positional embeddings, which is a bit tricky to get your head around, is that less frequency (down to no change at all) shows information on long-range dependencies and more general information. on the other hand, high frequency means that two tokens far apart could be closer in positional embedding than two tokens side by side. so this allows the model to capture the long-range dependencies of tokens AS WELL as the short-range enabled by the lower dimensions.

Attention Mechanism

in-depth look



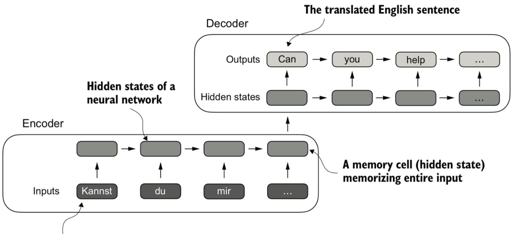
A simplified self-attention technique to introduce the broader idea

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

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

a traditional alternative to attention was to use encoder-decoder RNNs.



The translated English sentence

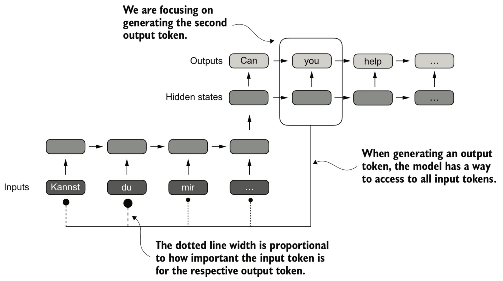
Hidden states of a neural network

A memory cell (hidden state) memorizing entire input

German input sentence to translate

the problem with this approach is that the decoder has no access to the input sequence and relies only on the hidden state, can lead to loss of context in long sequences. Good only for short sequences.

in 2014 Bandanau attention was introduced which modified the decoder-encoder RNN so at decoding steps there was access to input sequence. each input token also had a weight.

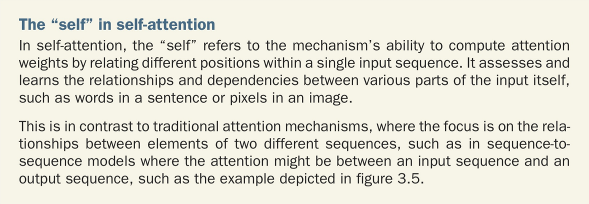


We are focusing on generating the second output token.

When generating an output token, the model has a way to access to all input tokens.

The dotted line width is proportional to how important the input token is for the respective output token.

IN 2017, researchers found that RNNs are not required and proposed transformers in "attention is all you need".



The "self" in self-attention

In self-attention, the "self" refers to the mechanism's ability to compute attention weights by relating different positions within a single input sequence. It assesses and learns the relationships and dependencies between various parts of the input itself, such as words in a sentence or pixels in an image.

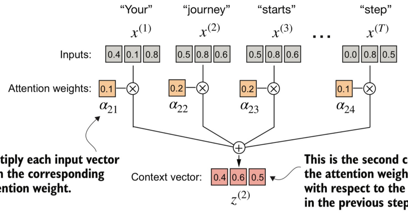
This is in contrast to traditional attention mechanisms, where the focus is on the relationships between elements of two different sequences, such as in sequence-to-sequence models where the attention might be between an input sequence and an output sequence, such as the example depicted in figure 3.5.

in self-attention, weight is assigned to tokens in a sequence, rather than the input-output seqs.

Simple Attention

here is a simple outline of attention.

query index=2 → dot product w/ other inputs → Normalize w/ softmax → mul & sum



Multiply each input vector with the corresponding attention weight.

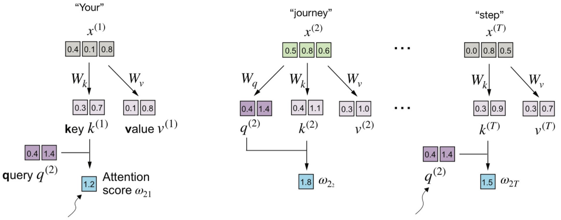
This is the second context vector because the attention weights were computed with respect to the second input vector in the previous steps.

the context vector here is a "modified version" of input embedding. just like input embedding is a modified version of token & positional embedding.

Learnable Attention

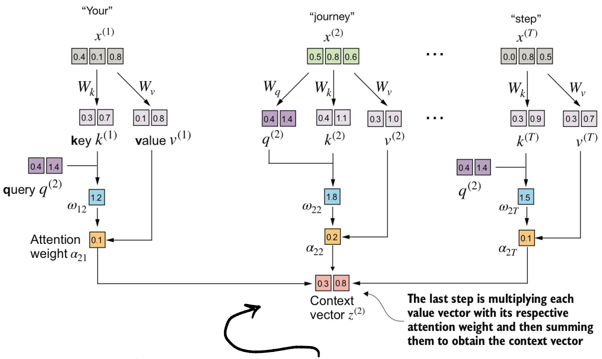
attention is in fact weighted-sum of the input embeddings.

now we imagine each token has a key-value produced by multiplying learnable key & value params by the input embeddings.



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.



The last step is multiplying each value vector with its respective attention weight and then summing them to obtain the context vector

now the whole process looks so, notice the input embeddings are not used directly for context vector.

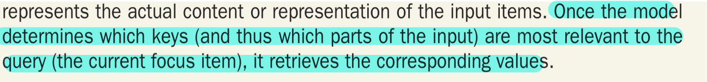
This is still weighted-sum over the value vectors, but the weights are the q@keys. This is the attention weights.

why query key values?

Query is the token we are focusing on to understand wrt the other tokens. Just like a **SQL** query.

Key is like a database key for search and retrieval of content.

Value is the actual content we're looking for.



represents the actual content or representation of the input items. Once the model determines which keys (and thus which parts of the input) are most relevant to the query (the current focus item), it retrieves the corresponding values.

But doesn't all these operations change the embeddings drastically?

Yes! and it should.

one way to think of attention is tokens talking with each other and share information. the word "mode" in "a machine learning model" must be very different from "a photoshoot model".

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

- attention is weighted-sum of V, which is a linear transformation of input embeddings.

Why V and not embeddings directly?

applying W_v on embeddings allows the model to focus on different dimensions of embs for different tasks. This flexibility provides more params to tune and learn complex patterns.

The formula pushes Q and K values to put emphasis on the right parts of V which itself is learned.

$$Q \cdot K^T$$

the dot product is a measure of Similarity

by computing the dot product between a Query vector and each Key vector the att assesses how relevant each key (and associated value) is to the Query.

⇓

doesn't it push Q and K to converge?

what is the relation between Q . K ?

it's a valid point, but W_q, W_k, W_v are independent matrices.

while they are independent to preserve flexibility, they are interconnected!

the dot product between K and Q introduces an implicit relationshiduring training.

The relationship is emergent from the optimization rather than imposed.

so because W_Q and W_K jointly determine the attention score, they are interdependent!

Q and K serve different roles :

Q represents what we're seeking or the context of the current token.

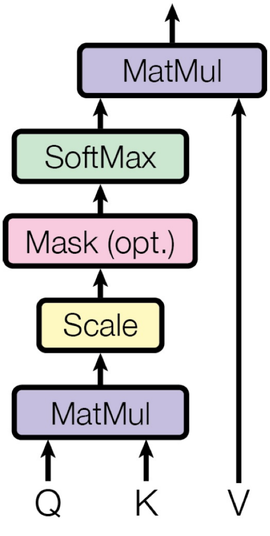
K represents features of all tokens in the sequence.

FANTASTIC!

while Q, K and V are independent, they work hand-in-hand to add attention to a sequence.

Their orchestrator is the god almighty back-propagation!

Scaled Dot-Product Attention



why scale by $\sqrt{d_k}$?

the dot product of two vectors is the sum of their pairwise elements. So the magnitude increases by the d_k dimension.

very large values in QK^T produce extreme peaks in the softmax, causing the problem of vanishing gradients.

since the magnitude of the dot product scales with $\sqrt{d_k}$, dividing by $\sqrt{d_k}$ ensures the variance is ~ 1, leading to a more stable training.

why $\sqrt{d_k}$

let's assume two random variables Q, K sampled from a normal dist with mean0 and variance 1.

$Q_i, K_i \sim N(0,1)$

the dot prod is: $S = Q.K = \sum_{i=1}^{d_k} Q_i . K_i$

since Q, K are independent and have zero mean: $E[Q_i] = E[K_i] = 0$

$E[S] = \sum E[Q_i K_i] = \sum E[Q_i] E[K_i] = 0$

the variance of S is:

$Var[S] = E[S^2] - (E[S])^2 = E[S^2] \longrightarrow$ calculate $E[S^2]$ ------

$\dashrightarrow E[S^2] = E\left[\left(\sum Q_i k_i\right)^2\right] = E\left[\sum_i^{d_k} (Q_i K_i)^2 + 2\sum_{i<j} Q_i k_i Q_j k_j\right]$

Since Q_i & k_i are independent across different domains :

$E[Q_i K_i Q_j k_j] = E[Q_i K_i] E[Q_j K_j] = 0$ for $i \neq j$

therefor:

$$E[S^2] = \sum E[(Q_i K_i)^2]$$

calculate $\swarrow$:

$$E[(Q_i K_i)^2] = E[Q_i]^2 E[K_i]^2 =$$

$$(Var(Q_i) + (E[Q_i])^2) \cdot (Var(K_i) + (E[K_i])^2) = (1)(1) = 1$$

So

$$E[S^2] = \sum 1 = d_k$$

$$Var[S] = E[S^2] = d_k$$

So the standard deviation of S is:

$$\sigma_S = \sqrt{var(S)} = \sqrt{d_k}$$

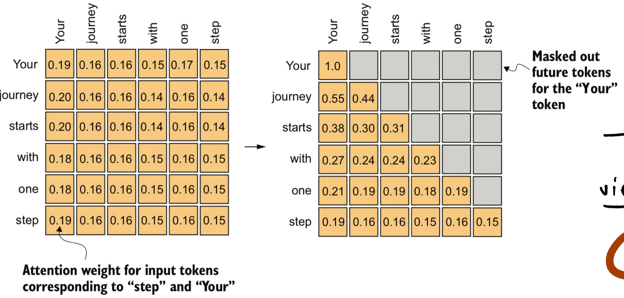
Standard deviation represents scale of increase

it quantifies how much the dot product values spread out from the means.

Causal Attention

a.k.a masked attention

hides future tokens to simulate inference time.



	Your	journey	starts	with	one	step
Your	0.19	0.16	0.16	0.15	0.17	0.15
journey	0.20	0.16	0.16	0.14	0.16	0.14
starts	0.20	0.16	0.16	0.14	0.16	0.14
with	0.18	0.16	0.16	0.15	0.16	0.15
one	0.18	0.16	0.16	0.15	0.16	0.15
step	0.19	0.16	0.16	0.15	0.16	0.15

	Your	journey	starts	with	one	step
Your	1.0					
journey	0.55	0.44				
starts	0.38	0.30	0.31			
with	0.27	0.24	0.24	0.23		
one	0.21	0.19	0.19	0.18	0.19	
step	0.19	0.16	0.16	0.15	0.16	0.15

Masked out future tokens for the "Your" token

Attention weight for input tokens corresponding to "step" and "Your"

this is done by multiplying a mask where upper triangular values are set to $-\infty$.

In Softmax, $e^{-\infty} = 0$ so it has no effect on the normalization step.

Dropout can be applied to attention score or after its mult with V, but applying Dropout to attention Scores is more conventional.

Multi-head Attention

multiple heads do the operations and are finally concatted.

In practice, to make this code optimized, we use a single W_Q W_K W_V to only perform the mult operation once.

we reshape the matrices in the process.

Q, K, Value

→ (b, num_tokens, d_out)

View → (b, num_tokens, num_heads, head_dim)

transpose → (b, num_heads, num_tokens, head_dim)

→ attn_score = queries @ keys.transpose(1,2)
dot product for each head

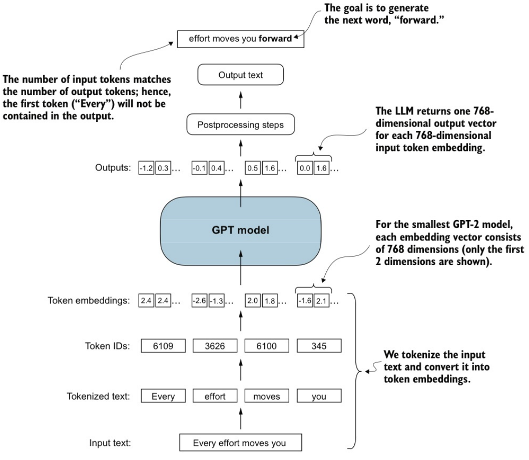
→ attn_score.masked_fill_(mask_bool, $-\infty$)

softmax & dropout

→ context_vec = (attn_score @ values).trans(1,2)
(b, num_tokens, num_heads, head_dim)

view → (b, num_tokens, d_out)

Chapter 4 Scratch GPT model from



The goal is to generate the next word, "forward."

The number of input tokens matches the number of output tokens; hence, the first token ("Every") will not be contained in the output.

The LLM returns one 768-dimensional output vector for each 768-dimensional input token embedding.

For the smallest GPT-2 model, each embedding vector consists of 768 dimensions (only the first 2 dimensions are shown).

We tokenize the input text and convert it into token embeddings.

Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our DummyGPTClass coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded input token dimension typically matches the output dimension. The output embeddings here represent the context vectors (see chapter 3).

LayerNorm is applied before and after the Transformer and before the final layer.

two learnable params to change scale & shift of the normalized output

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The other submodule in GPT is GeLU, which is a variant of the ReLU with more smooth transition at $x=0$

another popular option is SwiGLU

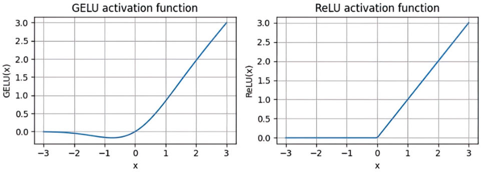


Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.

Why GeLU?

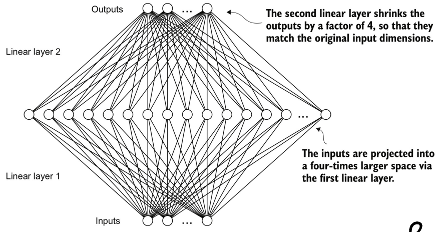
1. ReLU is very simple. But at $x=0$ it is Sharp and non-flexible. GeLU has a Smooth transition at $x=0$

2. GeLU has negative values for $x<0$ unlike ReLU which outputs zero.

All this makes GeLU a better option for deeper networks and more Complex ones.

FeedForward

1. Linear layer
2. GeLU
3. Linear layer



The second linear layer shrinks the outputs by a factor of 4, so that they match the original input dimensions.

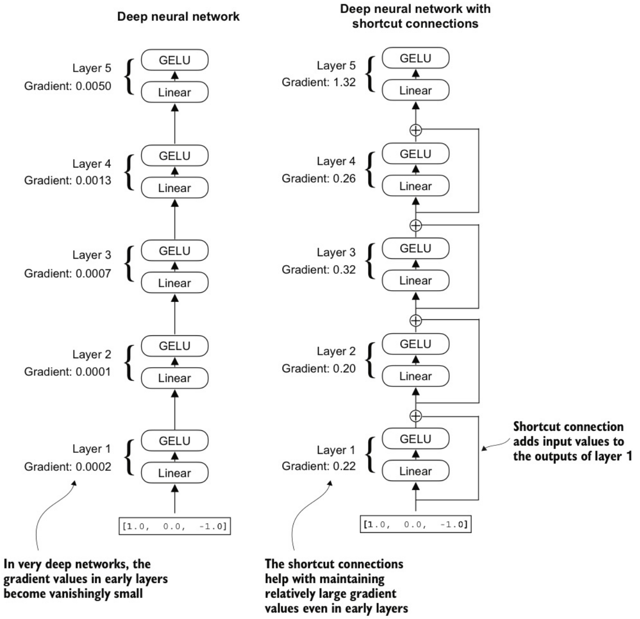
The inputs are projected into a four-times larger space via the first linear layer.

The expansion and then Contraction allows for exploration of more Complex representation space.

Residual Connection

originally proposed in Vision to mitigate the vanishing gradient (the grad progressively getting smaller, making the earlier layers hard to train.)

Residual Connections provide a shorter and alternative path for the gradient to flow by Skipping one or more connections, hence skip connection. They preserve flow of gradient



Shortcut connection adds input values to the outputs of layer 1

In very deep networks, the gradient values in early layers become vanishingly small

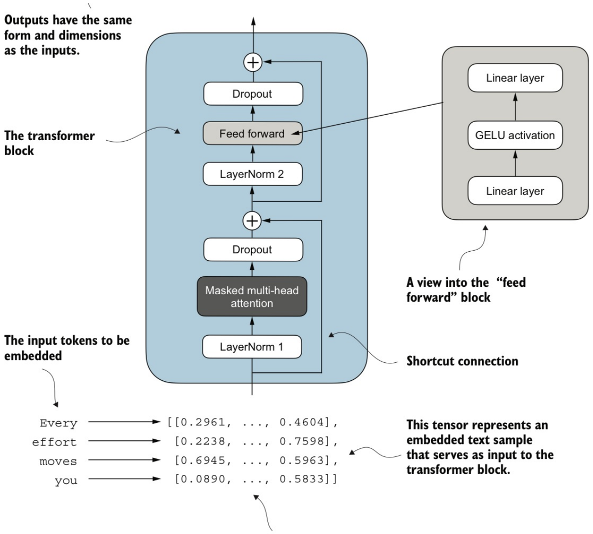
The shortcut connections help with maintaining relatively large gradient values even in early layers

Selfattention VS. Linear

Self-attention looks at the input data in relation to other parts, looking at it wholly.

Linear layer looks at the data individually.

Transformer Block



Outputs have the same form and dimensions as the inputs.

The transformer block

A view into the "feed forward" block

The input tokens to be embedded

Shortcut connection

```
[[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.

How GPT generates text:

logits = model (idx)

logits = logits [: , -1 , :] ← focus on the last token

[batch , num-tokens , emb-size]

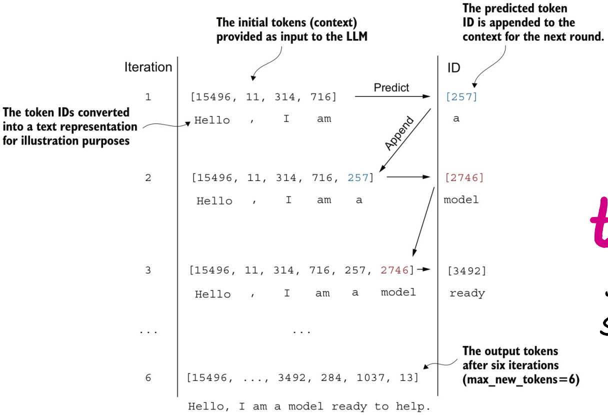
⇓

[batch , emb-size]

probs = softmax(logits)

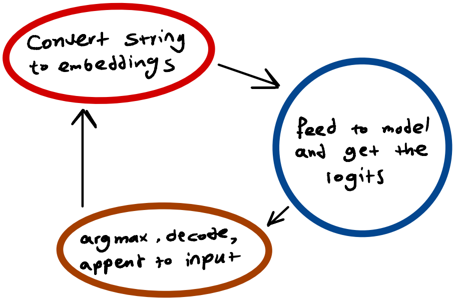
argmax (probs)

idx = torch . cat(idx , it-next)



The initial tokens (context) provided as input to the LLM
The predicted token ID is appended to the context for the next round.
Iteration
ID
1
[15496, 11, 314, 716]
Predict
[257]
The token IDs converted into a text representation for illustration purposes
Hello , I am
a
Append
2
[15496, 11, 314, 716, 257]
[2746]
Hello , I am a
model
3
[15496, 11, 314, 716, 257, 2746]
[3492]
Hello , I am a model
ready
...
...
The output tokens after six iterations (max_new_tokens=6)
6
[15496, ..., 3492, 284, 1037, 13]
Hello, I am a model ready to help.

Pertaining on unlabeled data

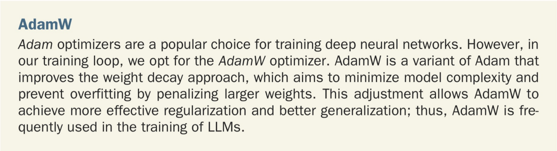


Convert string to embeddings
feed to model and get the logits
argmax , decode, append to input

LOSS FUNCTION

we compare the output probabilities against ground truth, push up the right probability.

we can use an average cross-entropy.



AdamW

Adam optimizers are a popular choice for training deep neural networks. However, in our training loop, we opt for the *AdamW* optimizer. AdamW is a variant of Adam that improves the weight decay approach, which aims to minimize model complexity and prevent overfitting by penalizing larger weights. This adjustment allows AdamW to achieve more effective regularization and better generalization; thus, AdamW is frequently used in the training of LLMs.

# temperature scaling

we replace argmax with a probabalistic approach to select next token based on the probability.

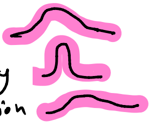
argmax ⟶ multinomial

logits /= temperature

if temp = 1 : no change

temp < 1 : even more peaky

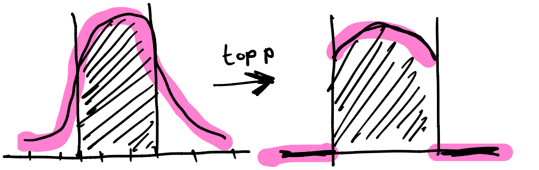
temp > 1 : uniform distribution



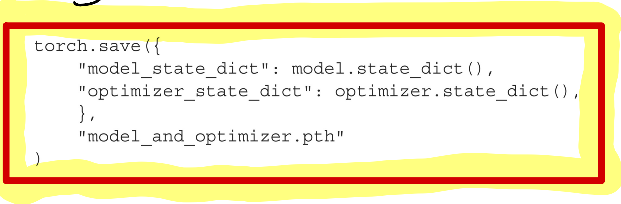
top-p

Select top p logits and set the rest ot -inf

So insensible options are not selected.



✱ if using an optimizer such as AdamW, which uses historical data, it's best to save that as well as the models weights.



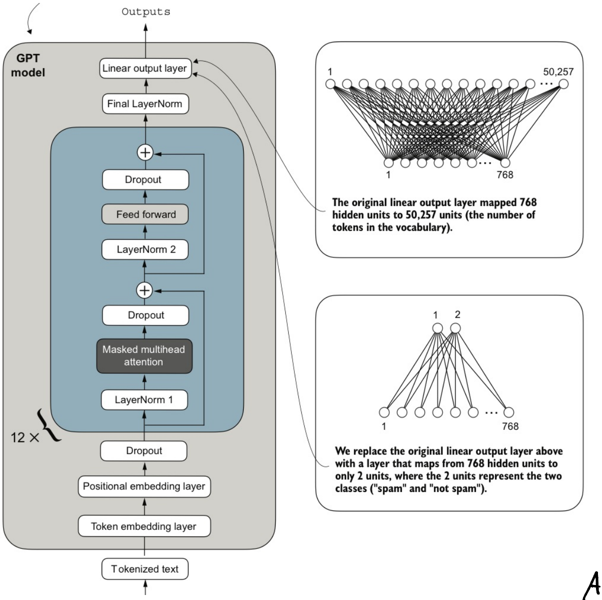
```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

fine-tuning

classification & instruction tuning

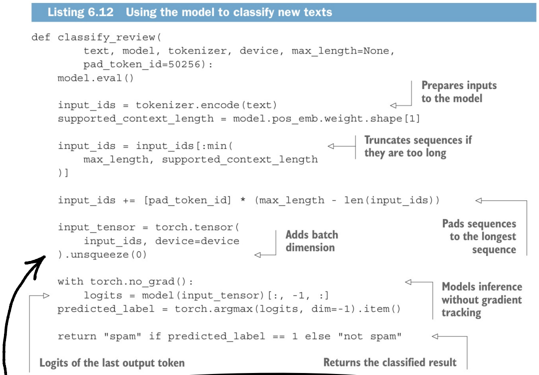
download model weights & initialize
download dataset & setup Dataset class
Replace the classification head

there are no universal rules to how many epochs to fine-tune. It could help to use the plot of train loss & Validation loss. if the model overfits, less epochs should be better, and if the train & val loss decrease together, more epochs can be helpful.



The original linear output layer mapped 768 hidden units to 50,257 units (the number of tokens in the vocabulary).

We replace the original linear output layer above with a layer that maps from 768 hidden units to only 2 units, where the 2 units represent the two classes ("spam" and "not spam").



Listing 6.12 Using the model to classify new texts

```
def classify_review(
        text, model, tokenizer, device, max_length=None,
        pad_token_id=50256):
    model.eval()

    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[1]

    input_ids = input_ids[:min(
        max_length, supported_context_length
    )]

    input_ids += [pad_token_id] * (max_length - len(input_ids))

    input_tensor = torch.tensor(
        input_ids, device=device
    ).unsqueeze(0)

    with torch.no_grad():
        logits = model(input_tensor)[:, -1, :]
    predicted_label = torch.argmax(logits, dim=-1).item()

    return "spam" if predicted_label == 1 else "not spam"
```

Prepares inputs to the model

Truncates sequences if they are too long

Pads sequences to the longest sequence

Adds batch dimension

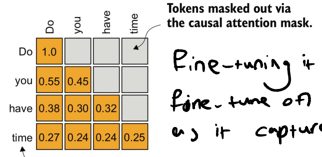
Models inference without gradient tracking

Logits of the last output token

Returns the classified result

Classification inference example

there is no need to fine-tune all layer, as the earlier layers capture the semantic meanings of the language.

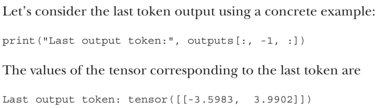


Tokens masked out via the causal attention mask.

	Do	you	have	time
Do	1.0			
you	0.55	0.45		
have	0.38	0.30	0.32	
time	0.27	0.24	0.24	0.25

The last token is the only token with an attention score to all other tokens.

For classification Fine-tuning it would suffice to fine-tune on the last token, as it captures the whole message.



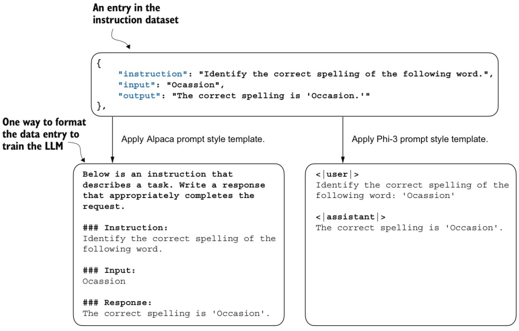
Let's consider the last token output using a concrete example:

```
print("Last output token:", outputs[:, -1, :])
```

The values of the tensor corresponding to the last token are

```
Last output token: tensor([[-3.5983,  3.9902]])
```

Another type of Fine-tuning is instruction ft where the resulting model generates text.



An entry in the instruction dataset

```
{
    "instruction": "Identify the correct spelling of the following word.",
    "input": "Ocassion",
    "output": "The correct spelling is 'Occasion.'"
},
```

One way to format the data entry to train the LLM

Apply Alpaca prompt style template.

```
Below is an instruction that
describes a task. Write a response
that appropriately completes the
request.

### Instruction:
Identify the correct spelling of the
following word.

### Input:
Ocassion

### Response:
The correct spelling is 'Occasion'.
```

Apply Phi-3 prompt style template.

```
<|user|>
Identify the correct spelling of the
following word: 'Ocassion'

<|assistant|>
The correct spelling is 'Occasion'.
```

Figure 7.4 Comparison of prompt styles for instruction fine-tuning in LLMs. The Alpaca style (left) uses a structured format with defined sections for instruction, input, and response, while the Phi-3 style (right) employs a simpler format with designated <|user|> and <|assistant|> tokens.

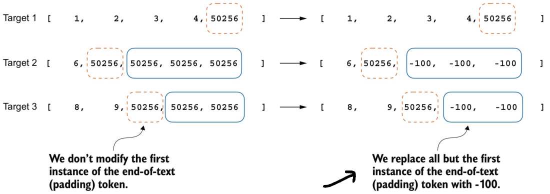
```
inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6]
inputs_3 = [7, 8, 9]
batch = (
    inputs_1,
    inputs_2,
    inputs_3
)
print(custom_collate_draft_1(batch))
```

The resulting batch looks like the following:

```
tensor([[    0,     1,     2,     3,     4],
        [    5,     6, 50256, 50256, 50256],
        [    7,     8,     9, 50256, 50256]])
```

writing our own collate function for batching looks like this

we use <pad_token> to make all batch inputs the same size.



Target 1 [1, 2, 3, 4, 50256] → [1, 2, 3, 4, 50256]
Target 2 [6, 50256, 50256, 50256, 50256] → [6, 50256, -100, -100, -100]
Target 3 [8, 9, 50256, 50256, 50256] → [8, 9, 50256, -100, -100]

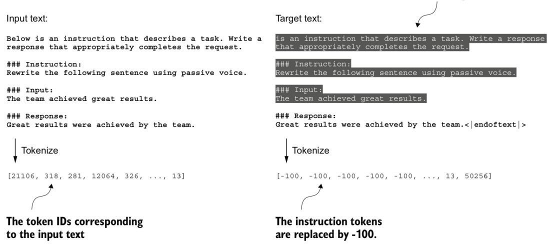
We don't modify the first instance of the end-of-text (padding) token.

We replace all but the first instance of the end-of-text (padding) token with -100.

this is so the pad token does not affect tho loss function.
(default by pyTorch cross entropy)

to focus on the target

Mask out the instruction when calculating the loss.



```
Below is an instruction that describes a task. Write a
response that appropriately completes the request.

### Instruction:
Rewrite the following sentence using passive voice.

### Input:
The team achieved great results.

### Response:
Great results were achieved by the team.
```

```
[21106, 318, 281, 12064, 326, ..., 13]
```

```
is an instruction that describes a task. Write a response
that appropriately completes the request.

### Instruction:
Rewrite the following sentence using passive voice.

### Input:
The team achieved great results.

### Response:
Great results were achieved by the team.<|endoftext|>
```

```
[-100, -100, -100, -100, -100, ..., 13, 50256]
```

The token IDs corresponding to the input text

The instruction tokens are replaced by -100.

this is an open area of research though not masking could be beneficial

Chapter 7, instruction fine-tuning is full of code implementations that are best read on the book.

Most importantly, model evaluation is not as straightforward as it is for completion fine-tuning, where we simply calculate the percentage of correct spam/non-spam class labels to obtain the classification's accuracy. In practice, instruction-fine-tuned LLMs such as chatbots are evaluated via multiple approaches:

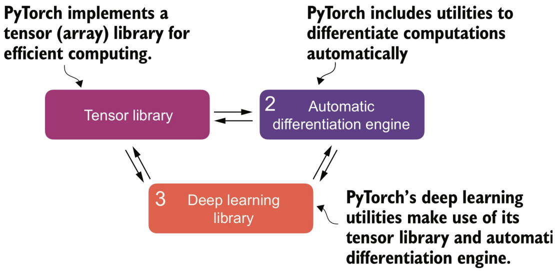
- Short-answer and multiple-choice benchmarks, such as Measuring Massive Multitask Language Understanding (MMLU; https://arxiv.org/abs/2009.03300), which test the general knowledge of a model.
- Human preference comparison to other LLMs, such as LMSYS chatbot arena (https://arena.lmsys.org).
- Automated conversational benchmarks, where another LLM like GPT-4 is used to evaluate the responses, such as AlpacaEval (https://tatsu-lab.github.io/alpaca_eval/).

```
{'instruction': 'Rewrite the sentence using a simile.',
 'input': 'The car is very fast.',
 'output': 'The car is as fast as lightning.',
 'model_response': 'The car is as fast as a bullet.'}
```

to evaluate the fine-tuned model, such a dataset needs to be created.

The Book's Finishes

Appendix PyTorch



PyTorch implements a tensor (array) library for efficient computing.

PyTorch includes utilities to differentiate computations automatically

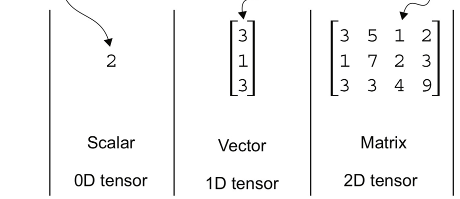
PyTorch's deep learning utilities make use of its tensor library and automati differentiation engine.

tensors are a generalize concept of a collection of values of rank n

A scalar is just a single number.

An example of a 3D vector that consists of 3 entries

A matrix with 3 rows and 4 columns



PyTorch vs Numpy

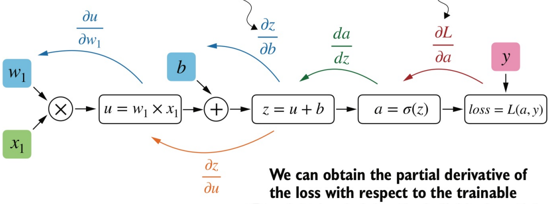
Numpy arrays & PyTorch tensors are Similar, but tensors come with additional features important in deep learning.

Most apis are the same.

default tensor type in PyTorch is int-64.

Autograd

PyTorch's automatic differentiation engine using computational graphs.


w_1
x_1
b
y
$\frac{\partial u}{\partial w_1}$
$\frac{\partial z}{\partial b}$
$\frac{da}{dz}$
$\frac{\partial L}{\partial a}$
$\frac{\partial z}{\partial u}$
$u = w_1 \times x_1$
$z = u + b$
$a = \sigma(z)$
$loss = L(a, y)$

The partial derivative of the intermediate result z with respect to the bias unit

The partial derivative of the loss with respect to its input

$$\frac{\partial L}{\partial w_1} = \frac{\partial u}{\partial w_1} \times \frac{\partial z}{\partial u} \times \frac{da}{dz} \times \frac{\partial L}{\partial a}$$

We can obtain the partial derivative of the loss with respect to the trainable weight by chaining the individual partial derivative in the graph.

$$\frac{\partial L}{\partial b} = \frac{\partial z}{\partial b} \times \frac{da}{dz} \times \frac{\partial L}{\partial a}$$

Similar to above, we can compute the partial derivative of the trainable derivative by applying the chain rule.

the computation graph builts a directed graph in the background to compute the forward pass and back propagatete.

A note on model outputs

in PyTorch, it's best to output the logits from the model rather than softmax.

Why?

1. logits contain more detailed information than softmax which is a normalized version of them.

2. the softmax function can cause numerical instability for using exponentions (if the logit value is too high or two low, it would overflow or underflow.)

PyTorch loss functions like CrossEntropyLoss apply softmax internally but in a stable way.

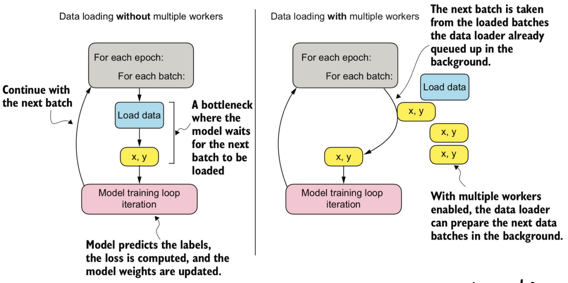
Also, for inference, we can get the argmax from logits without the need for softmax.

PyTorch uses a trick to calculate the softmax efficiently. It's called log-sum-exp trick which:

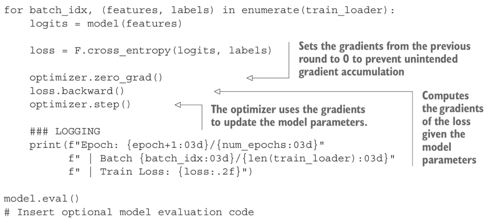
shifted_logits = logits - logits.max()

This would clip the logits but keep the resulting values the same as

$$\mathrm{Softmax}(z_i) = \mathrm{softmax}(z_i - c) = \frac{e^{z_i - c}}{\sum_j e^{z_j - c}}$$


Data loading without multiple workers
For each epoch:
For each batch:
Load data
x, y
Model training loop iteration
Continue with the next batch
A bottleneck where the model waits for the next batch to be loaded
Model predicts the labels, the loss is computed, and the model weights are updated.
Data loading with multiple workers
For each epoch:
For each batch:
Load data
x, y
x, y
x, y
x, y
Model training loop iteration
The next batch is taken from the loaded batches the data loader already queued up in the background.
With multiple workers enabled, the data loader can prepare the next data batches in the background.

none or few num_workers can cause bottlenecks.



```
for batch_idx, (features, labels) in enumerate(train_loader):
    logits = model(features)

    loss = F.cross_entropy(logits, labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    ### LOGGING
    print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
          f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
          f" | Train Loss: {loss:.2f}")

model.eval()
# Insert optional model evaluation code
```

Sets the gradients from the previous round to 0 to prevent unintended gradient accumulation

Computes the gradients of the loss given the model parameters

The optimizer uses the gradients to update the model parameters.

model eval

disable some training-specific configs such as layer_norm & dropout.

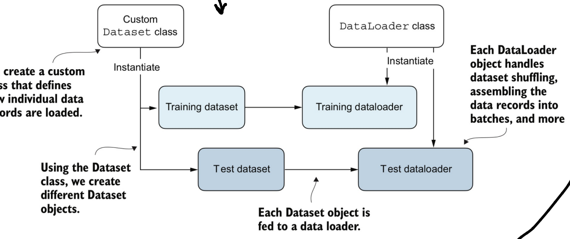
PyTorch DataLoader

we need to define a Dataset class for a task → Dataset

init

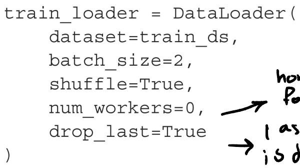
getitem

len


Custom Dataset class
Instantiate
DataLoader class
Instantiate
create a custom
ss that defines
v individual data
ords are loaded.
Training dataset
Training dataloader
Each DataLoader object handles dataset shuffling, assembling the data records into batches, and more
Using the Dataset class, we create different Dataset objects.
Test dataset
Test dataloader
Each Dataset object is fed to a data loader.

these are the basic functions for a custom Dataset class

once Dataset is fixed, we use Dataloader to load data into batches, shuffle it, and load parallel workers.



```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

→ how many subprocesses to use for data loading

→ last incomplete batch is dropped