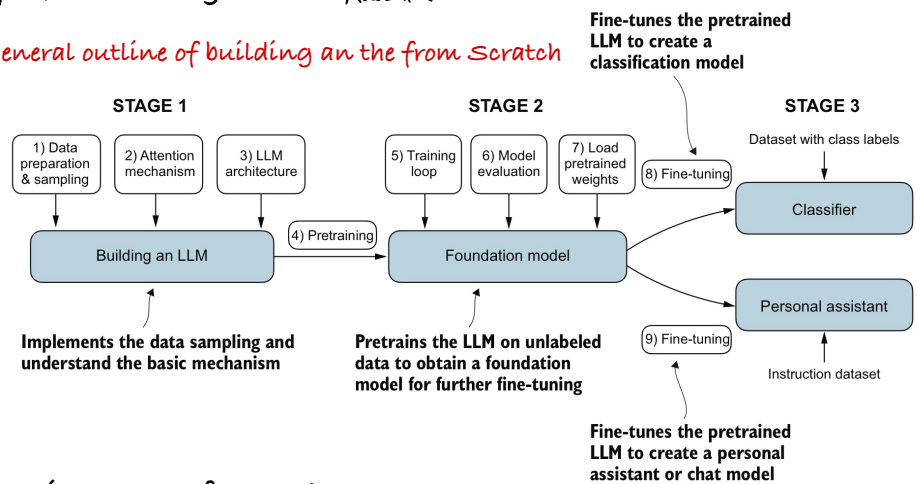
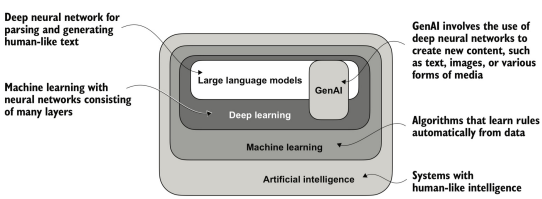
Build an LLM from Scratch

all images from the book by Sebastian Raschka

the general outline of building an the from Scratch



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

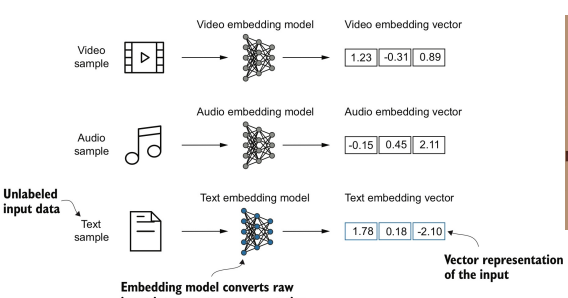


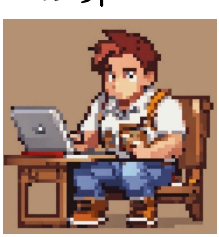
- a good choice for embedding is word2vec
- good embedding must capture the context
- LLMs can train their own embedding as part of their training, this allows them to capture meanings and concepts based on the task and data at hand.

Ready Data 4 Embedding

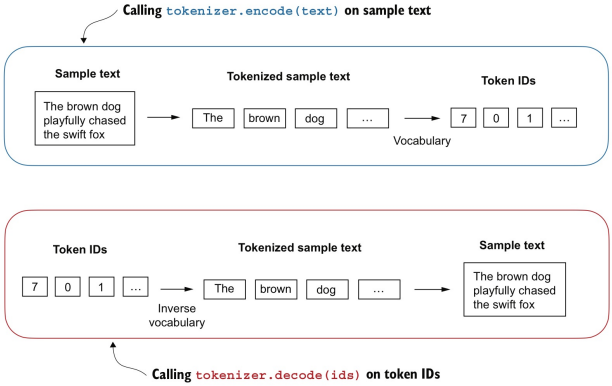
- convert large text to words
- assign ids to each token
- add special & unknown tokens

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.



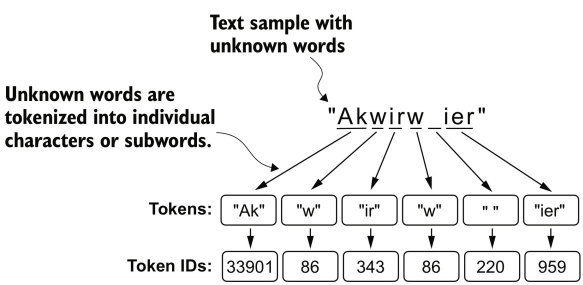
Some common special tokens:

[BOS] marks the beginning of a sequence/text

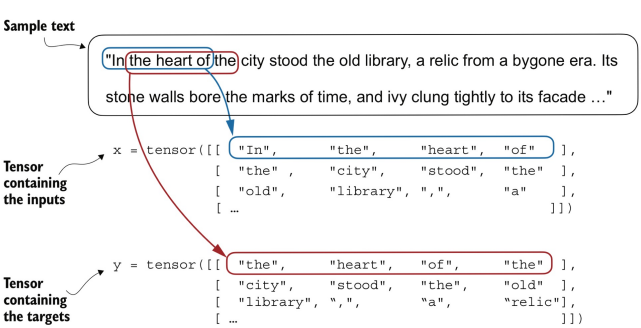
[EOS] " " ending of a " "

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

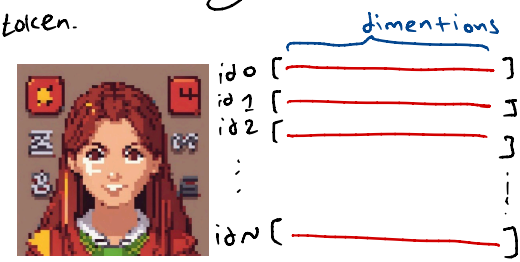
Some tokenizers like BPE can encode unknown words without using special tokens, by breaking the word down to tokenizable parts.



How the LLM processes input/output pairs



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 doesn't make a difference.

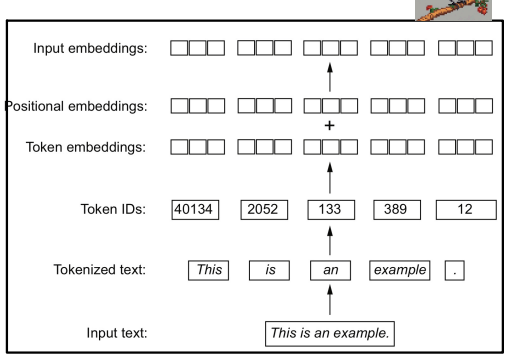
→ we inject position information to LLM

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: learning also about positions would complicate training
4. Existing approaches work well!

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

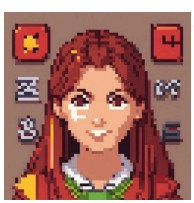
Criteria of a good Pos Embedding

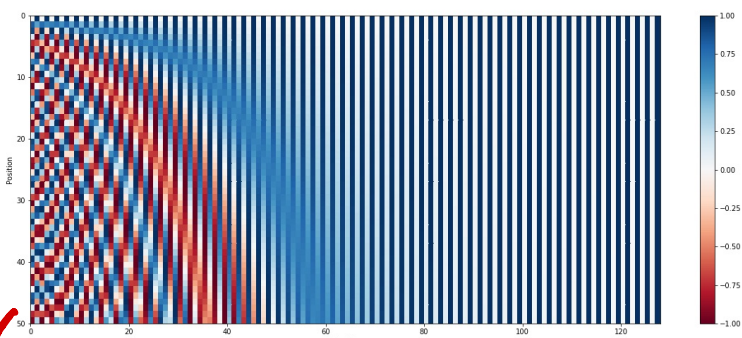
- unique encoding for each position
- distance between two time steps be consistent across sentences of different lengths
- should be bounded to generalize to longer sentences w/ no efforts
- deterministic

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

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

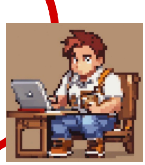
$$w_k = \frac{1}{10000^{2k/d}}$$





Why is PE summed by TE rathen than concatted?

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

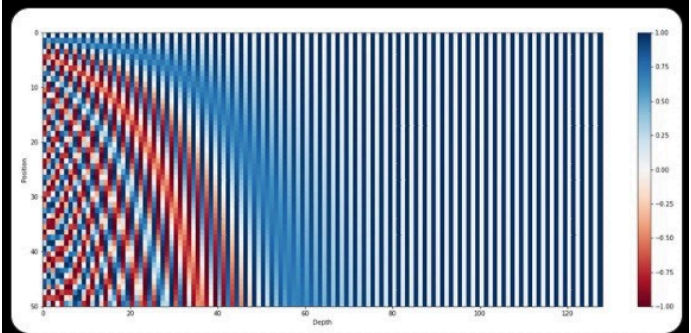


lower dimension have higher frequency fluctuating more rapidly. So two tokens side by side have more difference in PE rather than tokens far apart. this helps model capture local dependencies while the opposite lower frequency in higher dimensions captures more general & long-range dependencies.

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

29 35 445 229

hesam ✓
@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.

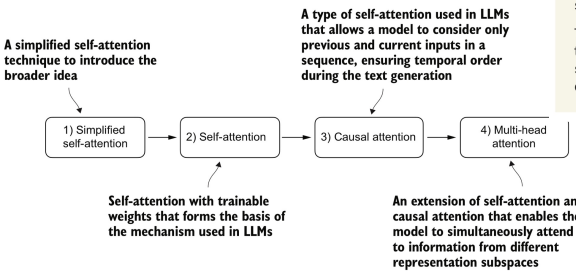
Attention Mechanism
in-depth look

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

The "self" in self-attention

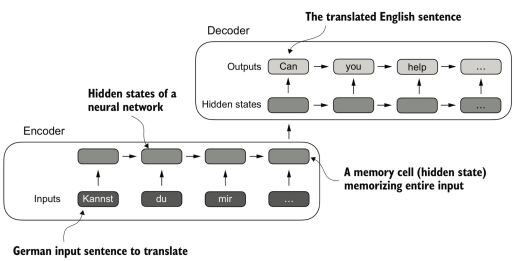
In self-attention, the "self" refers to the mechanism's ability to compute attention weights by relating different positions within a single input sequence. It assesses and learns the relationships and dependencies between various parts of the input itself, such as words in a sentence or pixels in an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



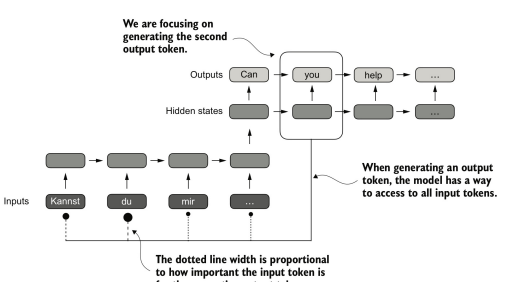
In self-attention, weight is assigned to tokens in a sequence, rather than the input-output seqs.

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



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.

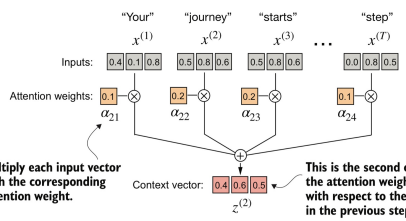
In 2014 Bandanau attention was introduced which modified the decoder-encoder RNN so at decoding steps there was access to input sequence. each input token also had a weight.

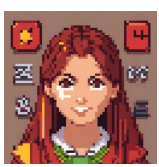


Simple Attention

Here is a simple outline of attention.

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



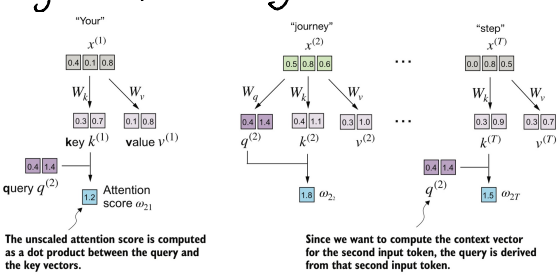


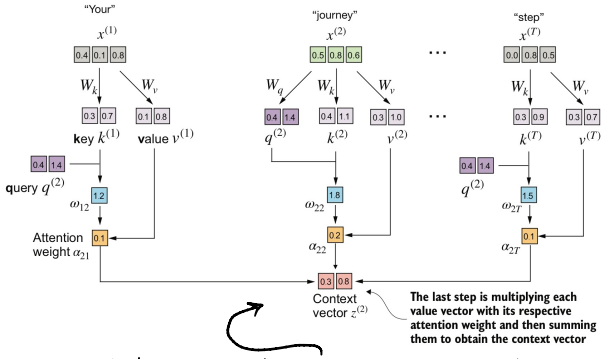
The context vector here is a "modified version" of input embedding. Just like input embedding is a modified version of token & positional embedding.

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.





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.

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

- 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.

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.

$$Q \cdot K^T$$

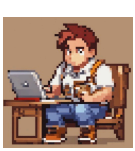
the dot product is a measure of Similarity

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!



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".

the dot product between K and Q introduces an implicit relationship during 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.

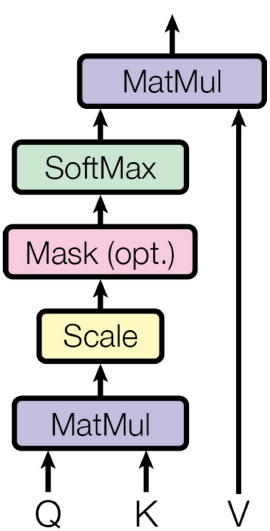
K represents features of all tokens in the sequence.

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 mean 0 and variance 1.

$Q_i, K_i \sim \mathcal{N}(0,1)$

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

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:

$\text{Var}[S] = E[S^2] - (E[S])^2 = E[S^2]$ → calculate $E[S^2]$ ⋯

⋯→ $E[S^2] = E\left[\left(\sum Q_i K_i\right)^2\right] = E\left[\sum_1^{d_k} (Q_i K_i)^2 + 2\sum_{i<j} Q_i K_i Q_j K_j\right]$

Since Q_i & k_i are independent across different domains:

$E[Q_i K_i Q_j K_j] = E[Q_i K_i]E[Q_j K_j] = 0 \quad \text{for } i \neq j$

therefor:

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

calculate:

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

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

So

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

$$\text{Var}[S] = E[S^2] = d_k$$

So the standard deviation of S is:

$$\sigma_S = \sqrt{\text{Var}(S)} = \sqrt{d_k}$$

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.

Q, K, Value

→ (b, num_tokens, d_out)

View → (b, num_tokens, num_heads, head_dim)

transpose → (b, num_heads, num_tokens, head_dim)

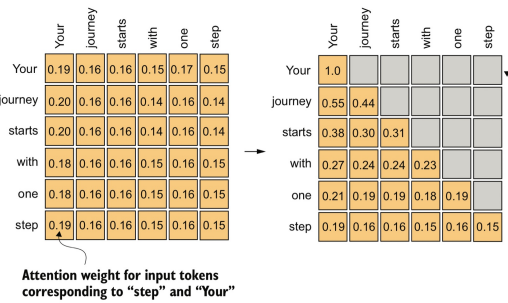
→ attn_score = queries @ keys.transpose(1,2)
dot product for each head

→ attn_score.masked_fill_(mask_bool, -∞)

Softmax & dropout

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

view → (b, num_tokens, d_out)



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.

Chapter 4 Scratch GPT model from

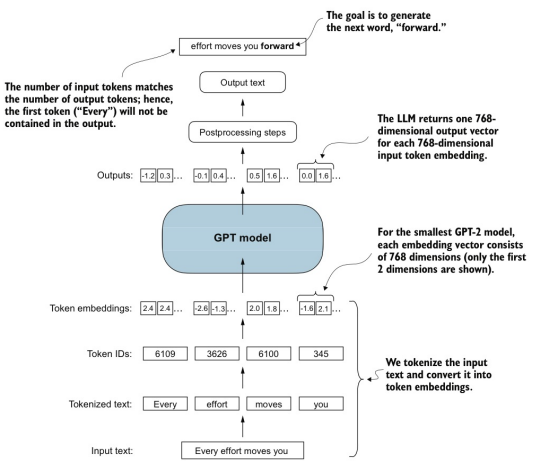
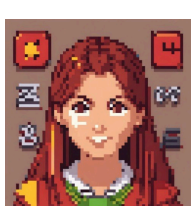


Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our `DummyGPTClass` coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded input token dimension typically matches the output dimension. The output 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

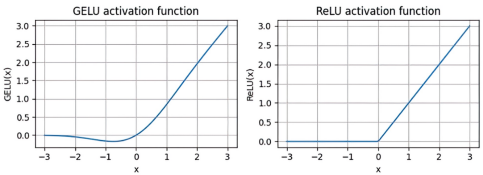


Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.

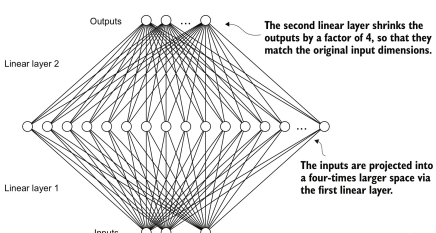
Why GeLU?

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

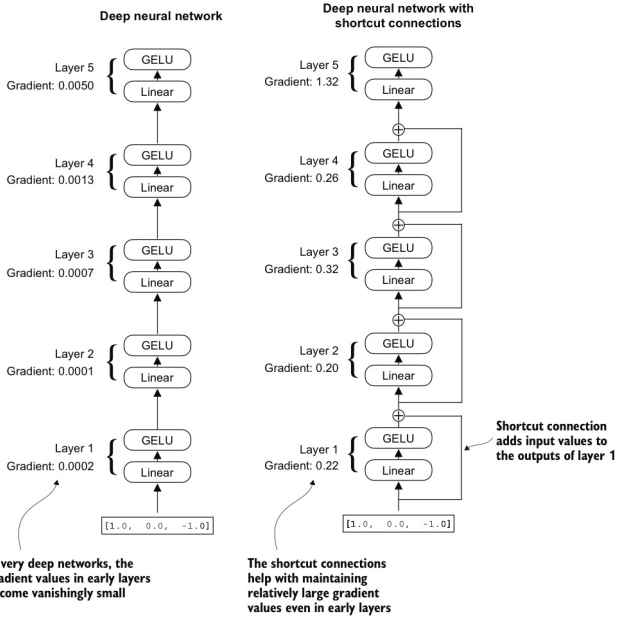


The expansion and then contraction allows for exploration of more complex representation space.

Residual Connection

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

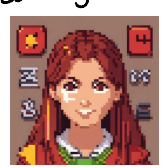
Residual Connections provide a shorter and alternative path for the gradient to flow by skipping one or more connections, hence skip connection. They preserve flow of gradient



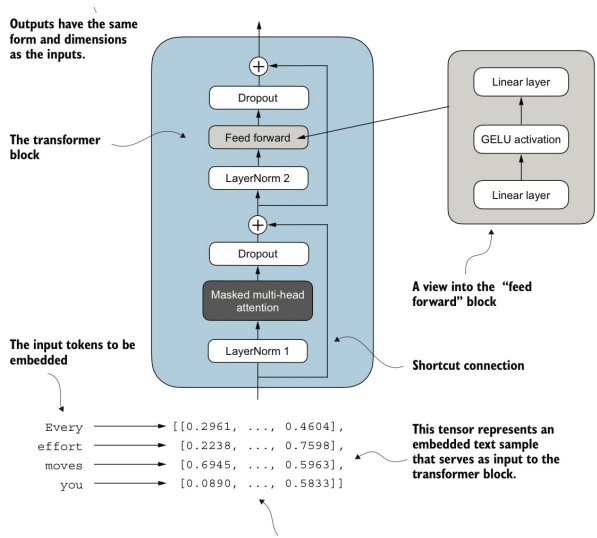
Self attention vs. Linear

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



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, idx_next)
```

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.

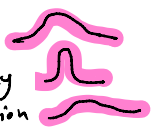
temperature scaling

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

argmax ⟶ multinomial

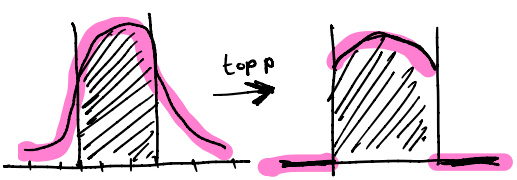
logits /= temperature

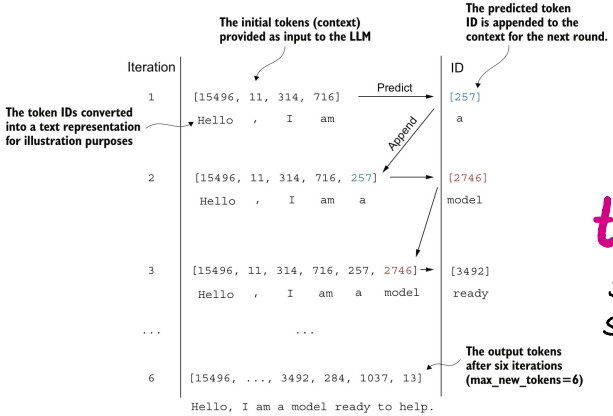
if temp = 1: no change
temp < 1: even more peaky
temp > 1: uniform distribution



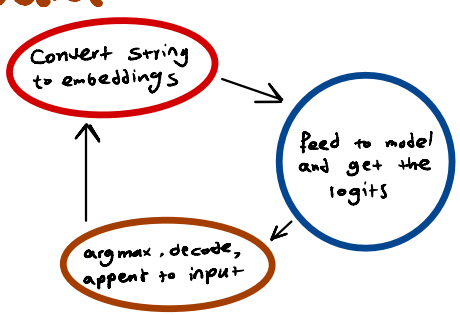
top-p

Select top p logits and set the rest of -inf
So insensible options are not selected.





Pertaining on unlabeled data



LOSS FUNCTION

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

we can use an average cross-entropy.

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

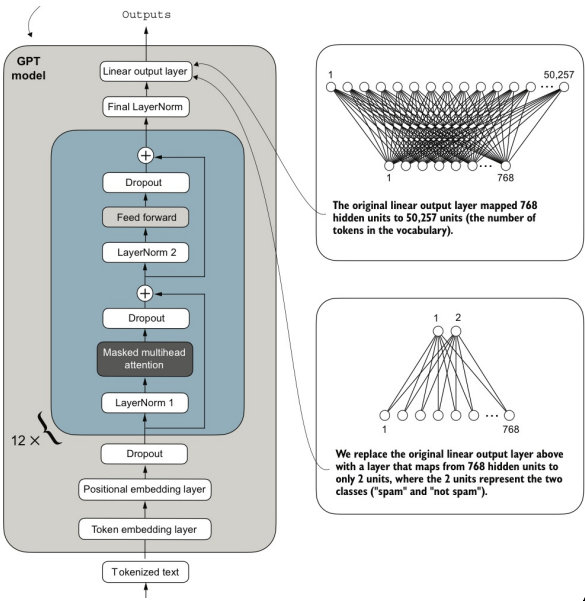
```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

fine-tuning

classification & instruction tuning

download model weights & initialize
download dataset & setup Dataset class
Reprace 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.



Listing 6.12 Using the model to classify new texts

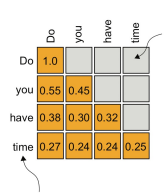
```
def classify_review(
        text, model, tokenizer, device, max_length=None,
        pad_token_id=50256):
    model.eval()

    input_ids = tokenizer.encode(text)                    # Prepares inputs to the model
    supported_context_length = model.pos_emb.weight.shape[1]

    input_ids = input_ids[:min(                           # Truncates sequences if they are too long
        max_length, supported_context_length
    )]

    input_ids += [pad_token_id] * (max_length - len(input_ids))   # Pads sequences to the longest sequence

    input_tensor = torch.tensor(
        input_ids, device=device                          # Adds batch dimension
    ).unsqueeze(0)

    with torch.no_grad():                                 # Models inference without gradient tracking
        logits = model(input_tensor)[:, -1, :]
    predicted_label = torch.argmax(logits, dim=-1).item()

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

Logits of the last output token — Returns the classified result

Classification inference example

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

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



Tokens masked out via the causal attention mask.

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 only 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]])
```

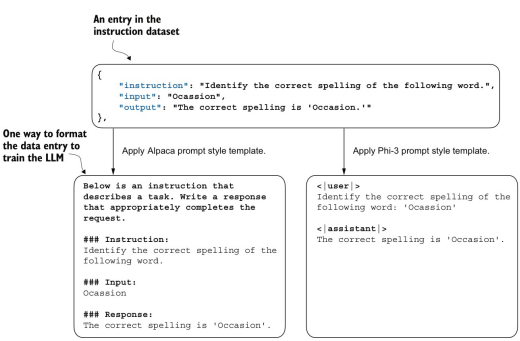


Figure 7.4 Comparison of prompt styles for instruction fine-tuning in LLMs. The Alpaca style (left) uses a structured format with defined sections for instruction, input, and response, while the Phi-3 style (right) employs a simpler format with designated <|user|> and <|assistant|> tokens.

```
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))
```

```
{'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.'}
```

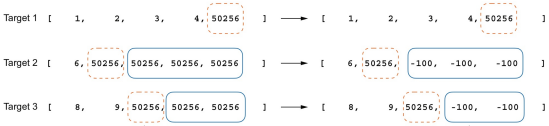
to evaluate the fine-tuned model, such a dataset needs to be created.

The resulting batch looks like the following:

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

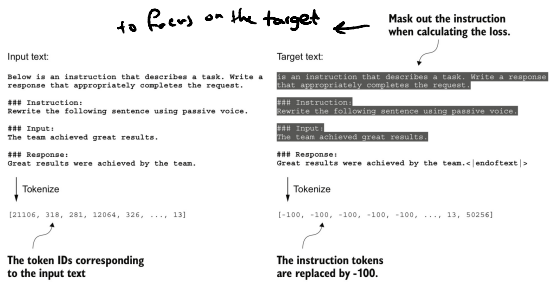
writing our own collate function for batching looks like this

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



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

to focus on the target



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

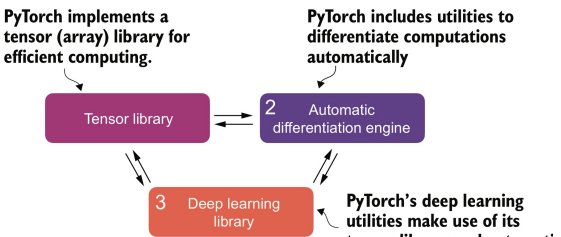
Chapter 7, instruction fine-tuning is full of code implementations that are best read on the book.

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:

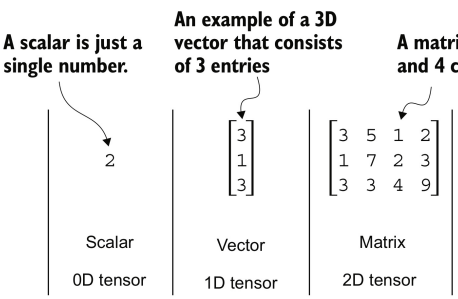
- Short-answer and multiple-choice benchmarks, such as Measuring Massive Multitask Language Understanding (MMLU; https://arxiv.org/abs/2009.03300), which test the general knowledge of a model.
- Human preference comparison to other LLMs, such as LMSYS chatbot arena (https://arena.lmsys.org).
- Automated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

The Book's Finishes

Appendix PyTorch



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



PyTorch vs Numpy

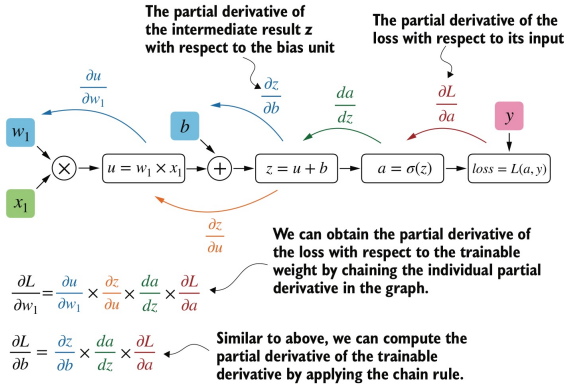
Numpy arrays & PyTorch tensors are similar, but tensors come with additional features important in deep learning.

Most api's are the same.

default tensor type in PyTorch is int-64.

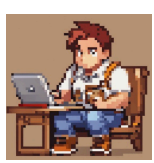
Autograd

PyTorch's automatic differentiation engine using computational graphs.



the computation graph builds a directed graph in the background to compute the forward pass and back propagate.

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.

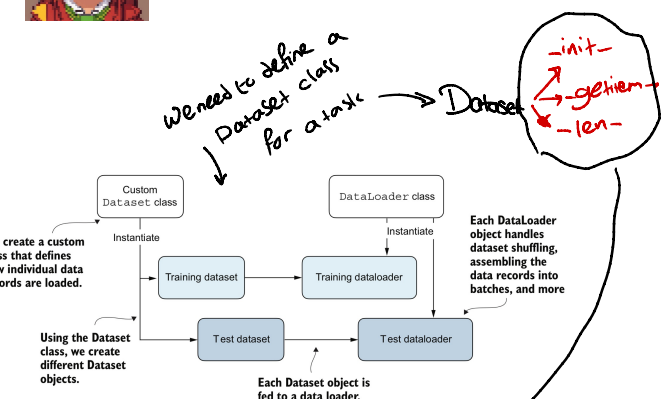
PyTorch uses a trick to calculate the Softmax efficiently. It's called log-sum-exp trick which:

$$\text{shifted\_logits} = \text{logits} - \text{logits.max}()$$

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

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

PyTorch DataLoader



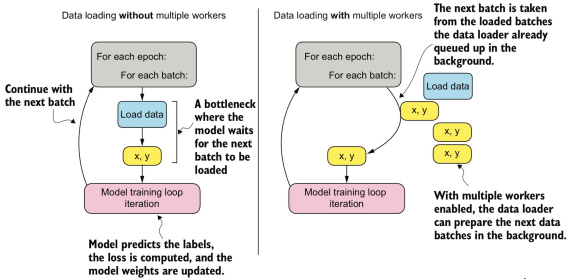
these are the basic functions for a custom Dataset class

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
)
```

num_workers: how many subprocesses to use for data loading

drop_last: last incomplete batch is dropped



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

The optimizer uses the gradients to update the model parameters.

Computes the gradients of the loss given the model parameters

model.eval()

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