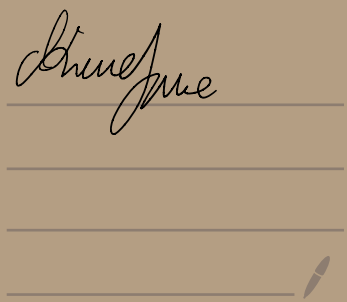
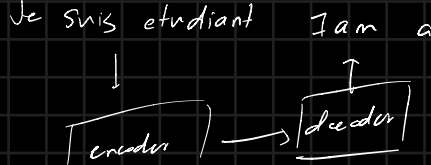
Transformers



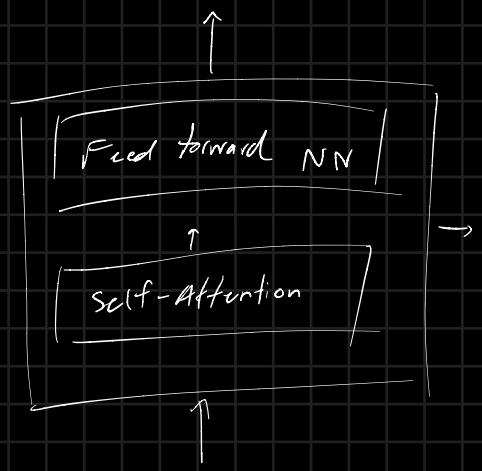
The Illustrated Transformer

Lends itself to parallelization

As a machine translation application

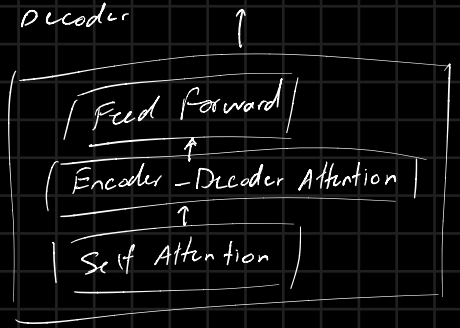


Encoder: do not share weights



Self-Attention: Helps encoder look at other words in the input sentence

Decoder



Turn input word into embedding vector

All encoders receive a vector of size 512.

Data flows up.

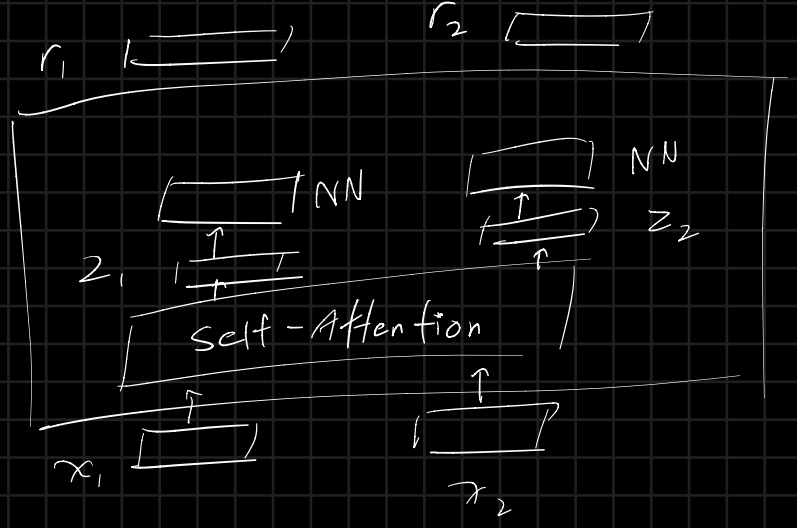
Size of 512 is a hyperparameter => max sentence size.

words in each position flows through its own path.

=> various paths can be executed in parallel.

Encoder: Self-Attention -> Feed forward neural networks

-> Next encoder



What is Self-Attention?

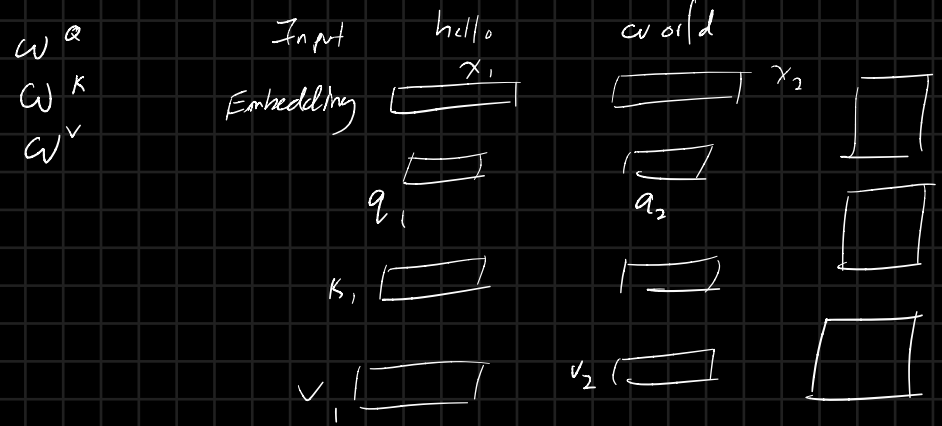
Associate words with other words

↳ when processing an embedding, it looks at other positions to lead to a better encoding of the word.

As vectors first:

Created by:
Multiply embedding by 3 trained matrices

Create 3 vectors
1. Query Vector
2. Key Vector
3. Value Vector



Step 2: Calculate a score

Score all other inputs against a word (how much activation contributed by other parts of the input sentence i.e attention)

Dot product of query vector with the key vector of the respective word to score.

eg: q_1 dot k_1
q_1 dot k_2 ...

Divide scores by 8 (Sqrt of dim of Key vectors) $(\sqrt{d_k})$

=> Softmax operation.

Scores how much each word will be expressed at this position

Step 5. Multiply each value vector by the softmax score.

=> Keep values of the words we want to focus on, and make irrelevant word activations go to zero.

Step 6. Sum up the weighted value vectors

=> Sent to the neural networks

As a matrix operation...

Calculate Q, K, V matrices

$$n\left[\overset{512}{X}\right]\left[\overset{64}{W^Q}\right] = \overset{64}{\left[Q\right]}$$

All rows correspond to word in input sentence (X)

$$\left[X\right]\left[W^K\right] = \overset{64}{\left[K\right]}$$

$$\left[X\right]\left[W^V\right] = \overset{64}{\left[V\right]}$$

Step 2-6:

$$\text{Softmax}\left(\frac{[Q]\,[K^T]}{\sqrt{d_k}}\right)\odot[V] = [Z]$$

Multiheaded attention:

$[X]$

Attention head #0 | Attention head # 1 | ... | Attention head # 7

$[Q_0]\ [W_0^Q]$ $\quad$ $[Q_1]\ [W_1^Q]$

$[K_0]\ [W_0^K]$ $\quad$ $[K_1]\ [W_1^K]$

$[V_0]\ [W_0^V]$ $\quad$ $[V_1]\ [W_1^V]$

Do same thing 8 times

A #0 $\quad$ A # 1, ... $\quad$ A # 7

$[Z_0]$ $\quad$ $[Z_1]$ $\quad$ $[Z_8]$

$[z_0]$ $[z_1], \ldots [z_8]$

$[z_T]$

concatenate

$[z_T]$ $\begin{bmatrix} \\ w_0 \\ \\ \end{bmatrix}$

Summary

Input: hello world → Encode → $[x]$

Encoding only for first layer

otherwise, use value in R.

Single attention layer:

$$\begin{bmatrix} w_0^Q & w_0^K & w_0^V \\ \vdots & \vdots & \vdots \\ w_7^Q & w_7^K & w_7^V \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_7 \end{bmatrix} \Bigg\} [z_T] \begin{bmatrix} \\ w_0 \\ \\ \end{bmatrix} = [R]$$

Representing the order of the sequence using positional encoding:

To account for word order:

↑ Transformer adds a vector to each input embedding

↑ This added vector is trained.

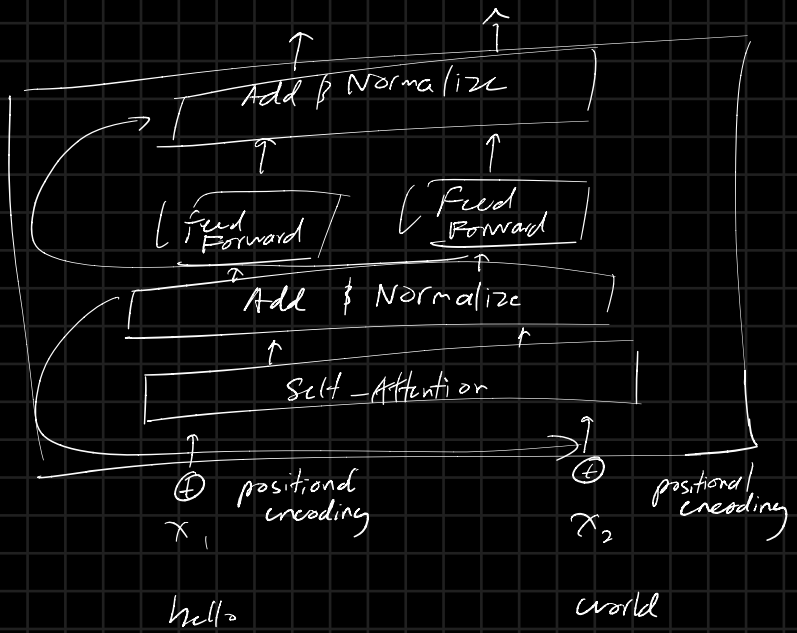
They are added element-wise

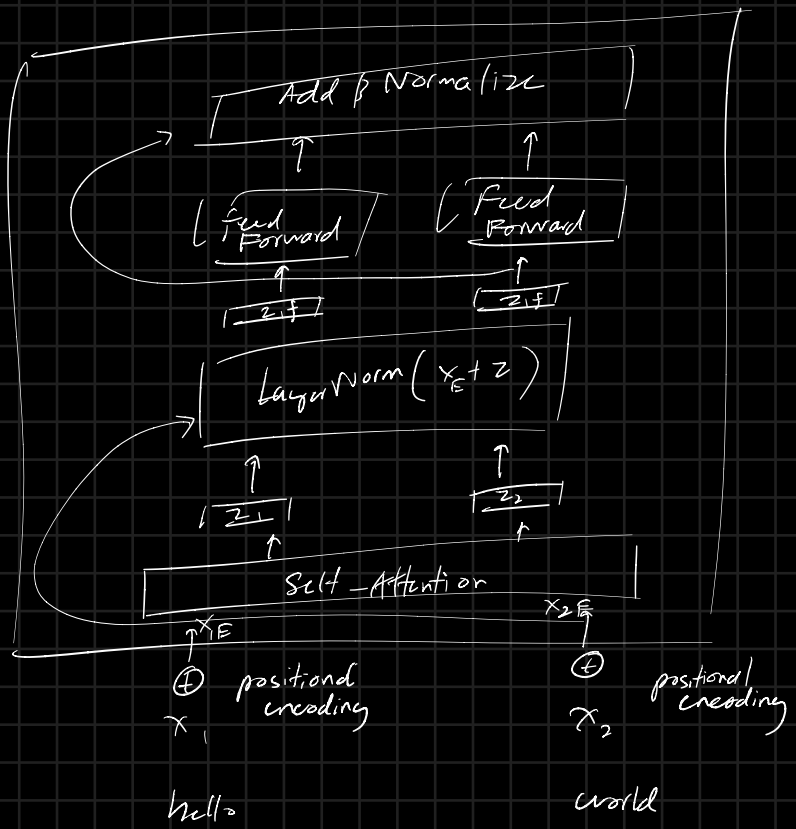
Represents "distance" between words

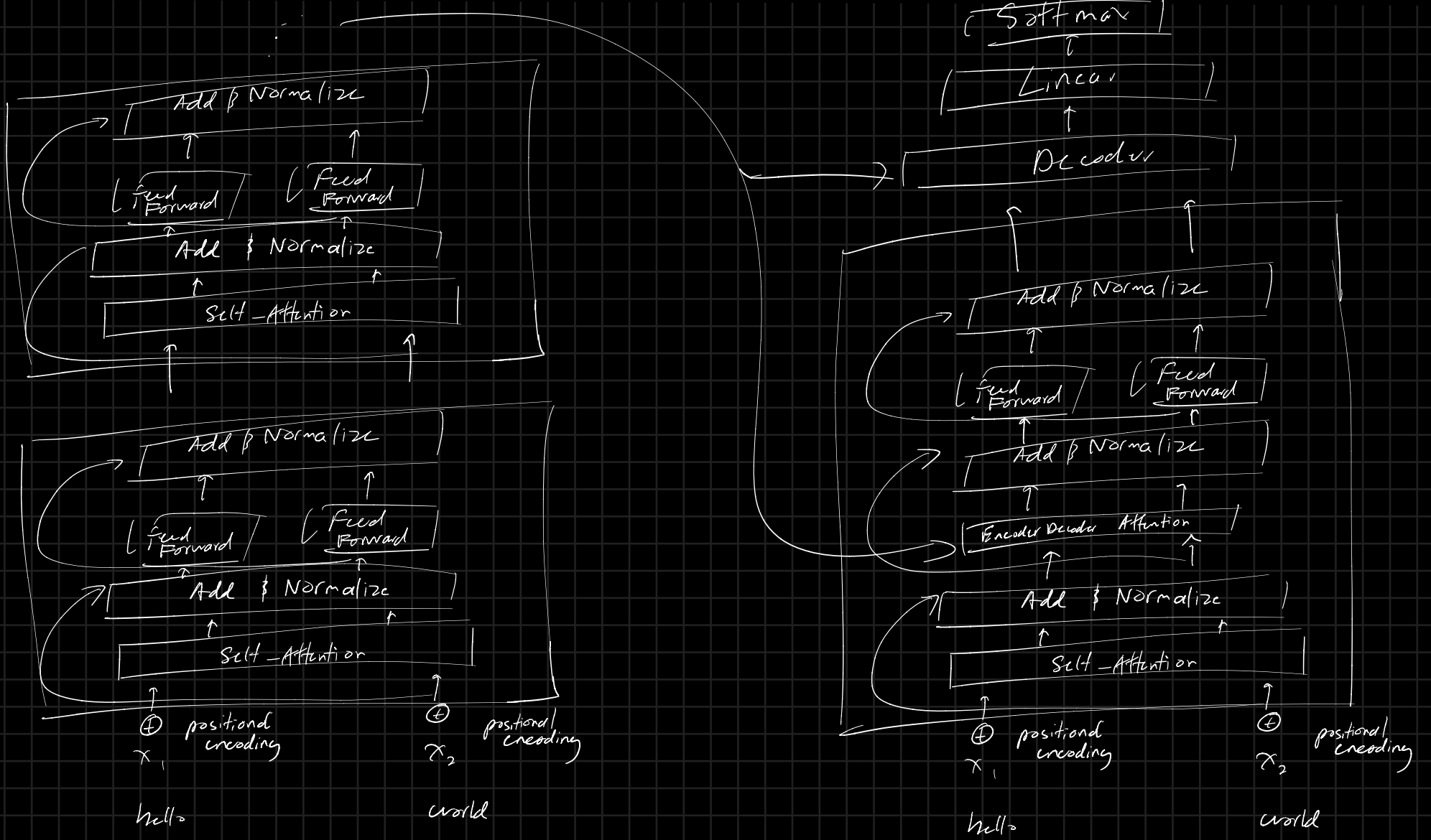
$$x_1 + t_1 = x_{E_1}$$

eg. one single word

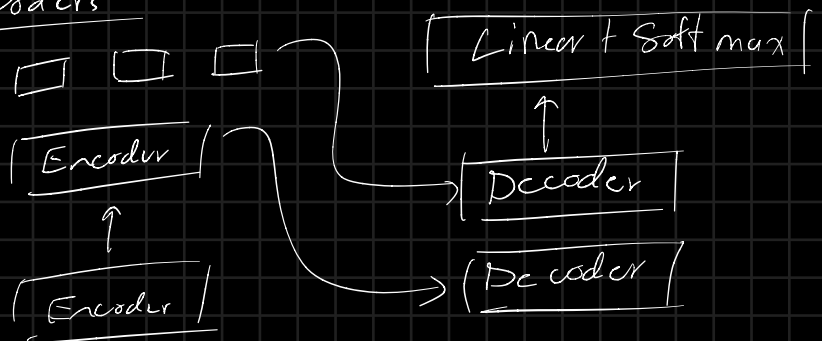
In the paper: residual connection + layer normalization






Softmax
Linear
Decoder
Add & Normalize
Feed Forward
Add & Normalize
Self-Attention
Add & Normalize
Feed Forward
Add & Normalize
Encoder Decoder Attention
Add & Normalize
Self-Attention
positional encoding
positional encoding
x1
x2
Hello
world
Add & Normalize
Feed Forward
Add & Normalize
Self-Attention
Add & Normalize
Feed Forward
Add & Normalize
Self-Attention
positional encoding
positional encoding
x1
x2
Hello
world

Decoders



Repeat until special end token is outputted

Output of each step is fed into bottom decoder at next time step

In decoder, self-attention layer is only allowed to attend to earlier positions in the output seq

Set future positions to -inf before softmax.

Encoder-Decode Attention is exactly what we did, but creates queries from layer below and takes keys/values matrices from output of encoder stacks.

Linear layer projects vector output from decoders to a logits vector (vocab_size long)

Cell of highest probability is chosen

logits → log probs

Define stoi, itos

one hot encode vectors

maximize correct, minimize incorrect activations of final log probalities

Beam Search: Sample multiple times, take the one with least error

Visualizing neural machine translation models

Sequence to sequence models

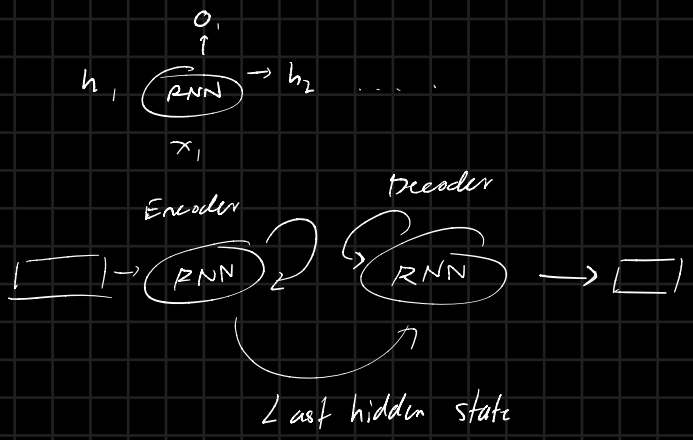
Encoder + decoder (item by item)

context

Both are RNNs

context vector size (hyperparameter)

words are first embedded as vectors (200-300) size



Decoder is an attention decoder RNN

All hidden states passed to decoder

Decoder attention also will:

Give each hidden state a score, softmax the scores and then multiply hidden states by softmaxed scores, to create the context vector

Done at each time step in decoder

Attention decoder RNN takes in <END> token embedding, and initial decoder hidden state.
Both are used to produce new hidden state.

Use encoder hidden states and new hidden state to calculate a context vector for this time step.

Concat context and hidden state to a feed forward NN, that indicates output word of time step.

Reproduce GPT 2:

124 mil parameter model

Mini Series

=> Biggest model => GPT 2

Down Stream metrics

124 M 12 layers, 768 dim in Transformer

plotting wpe.weight

Every row is a position (0, 1024)
positional learns sinusoids

Responds more or less (activations) based on the position
Jagged (not fully trained)
Expected to see smooth curves

Positional embeddings are fixed to sinusoids & cosines of diff frequencies

Decoder only transformer
Cross attention missing

GELU (always a change)

Dataset: tiny shakespeare

3 to 1 compression: 3 words, 1 token

wc input.txt
lines, words, bytes

.view stacks up sequences for batching

load 1 additional token

Reasonable Starting point for the loss

Every vocab gets uniform prob

$$-\ln\left(\frac{1}{\text{vocab\_size}}\right)$$

Share same weights between embedding layer and pre-softmax-Linear layer

(weights are tied together)

Want 2 matrices to behave similar

Intuition: Similar tokens are nearby in the token embedding space (similar embeddings)

=> Get better performance this way

↳ 2 dependencies

Initialization:

stddev = 0.02

nvidia-smi

import code

code.interact(local = locals())

By default, all parameters are float 32
↳ Don't need too much precision for training

FLOAT ↓ TFLOPS ↑

TFLOPS

Floating point operations (Trillions)

Sparsity

int8 used for inference, not training

Memory bandwidth (less bits for datatype, more quick training)
↳ often the bottleneck

Memory
Speed to access memory

Tensor Cores (instructions in Ada architecture)

(4,4) @ (4,4) + (4,4)
Varying precision ↑ FP32
Calculations get broken up like this (fastest way to multiply matrices)

Most work happens in the form of nn.Linear

bits
S e8 m23 FP32) 8x faster
S e8 m10 TP32 ↙ Can tolerate this empirically

cost - reduce precision (approximate)

torch.cuda.Synchronize()
waits for Gpu to finish work

Metric: tokens per second

Drop to BFLOAT16 => less precision and more truncation

FP16 → reduced int range

Automatic mixed precision

torch.autocast()
Some converted layer, some are left alone
LSTM cell Layer Norm

No gradient Scaler

torch.compile()

compiler for neural networks

GCC for neural nets

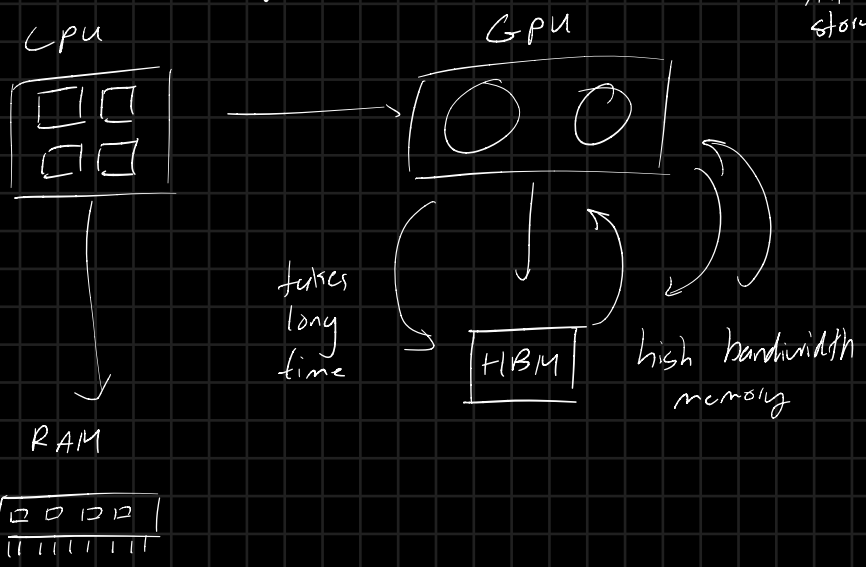
Use always if not debugging

Compiles NN as an object

Reduces Python overhead to GPU

torch.compile() analyzes operations and optimizes it

Interpreter runs in "eager mode"

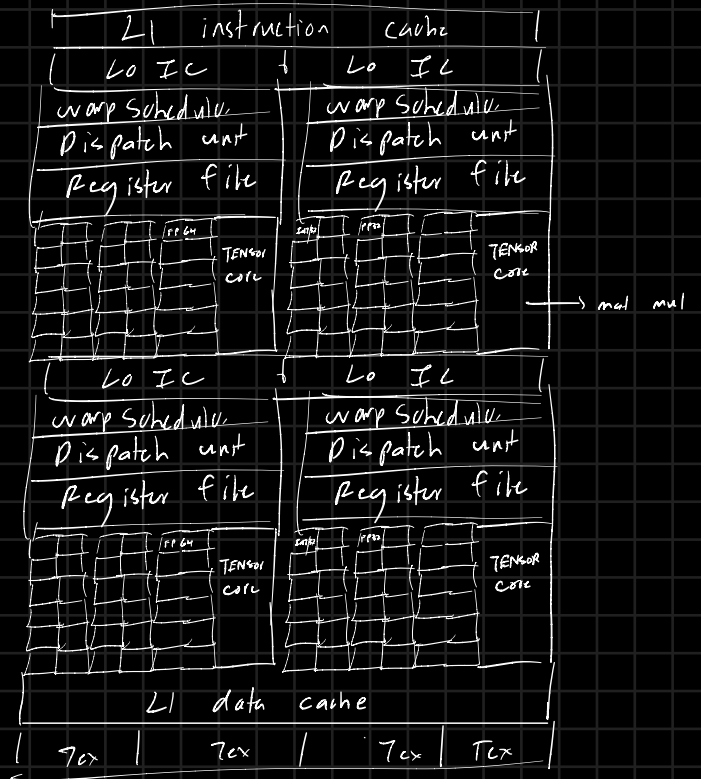


Kernel fusion!!! torch.compile() optimizes read/writes

Calculations occur in GPU
Memory in HBM (2 chips)

Streaming multiprocessor (SM)

SM



L2 Cache lives on the chip

Not alot of memory on GPU

CPU SRAM, GPU HBM, CPU DRAM

Flash attention: Kernel fusion

online normalizer for Softmax

matmul, dropout, softmax, mask, matmul

ugly numbers vs. nice numbers

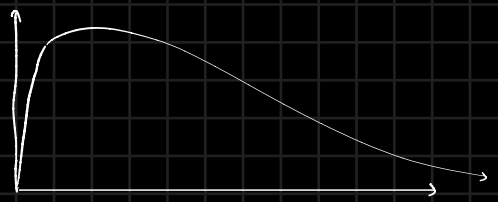
Everything in CUDA written in pow of 2

$\beta_1 = 0.9$
$\beta_2 = 0.95$

length of vector

Clip gradients norm

Cosine decay learning rate (10%) with warmup



Update batch size aswell, increase over time

Learn which tokens actually show up first

Gradients are very similar

Gradients after will start to differentiate

Data is sampled without replacement

weight decay 0.1, to implement regularization

Fused AdamW => Kernels are fused during calculations

Batch size of 0.5 million => gradient accumulation

Remember to de accumulate in the loss.backward() / normalize

Data Distributed Parallel

Launch x processes, assigned a Gpu

Average all gradients

web text dataset
scraped all webpages + text on Reddit

Common Crawl dataset

wikipedia

webText 2
Books 1
Books 2

Good datasets

Red Pajama

Slim Pajama

Fine web Sample-10 BT

Download dataset

Tokenize dataset using GPT tokens

start tokens with < eot > end of text token

Tokenize in uint16

Tokenize as np.array()

shards → numpy files

Train + val shards

100 shards

Advance shard and loop, get tokens, adjust position

Calculate total warmup tokens

$$\text{Steps} \times \frac{\text{sec}}{\text{step}} \times \frac{1\ \text{min}}{60\ \text{sec}} \times \frac{1\ \text{hour}}{60\ \text{min}}$$

torch.Generator(device=device)

⇓

Sample from diff seed, for each GPU

all diff from original

Hella Swag

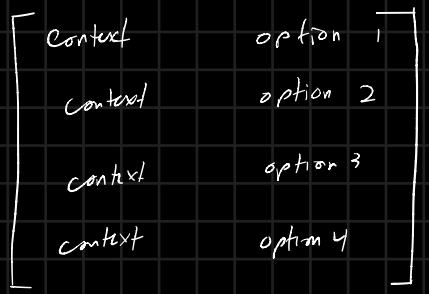
Multiple choice constructs, one is natural, rest are not

Shared context

Token completion

T tokens

B=4



See one option at a time

Fine tune into chat format

=> Swap out dataset into user assistant structure

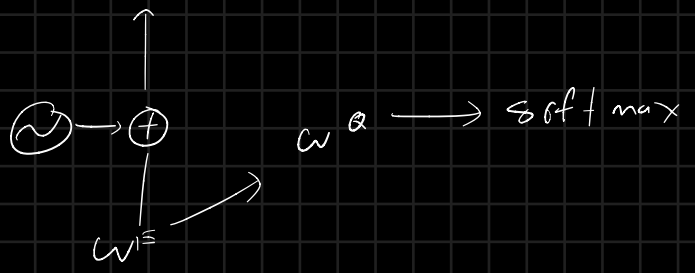
Finetune on this

Fill in user, Sample or assistant

One hot encodings vs. token embeddings

512 dim

=> positional embedding



parallelization

Stabilize gradients

output of self-attention: context enriched word embedding

Multi-head Self-Attention

Layer norm: subtracts mean, divides by std deviation, across all vectors

Batch Norm normalizes across <u>all features</u> independently

Layer Norm normalizes across each of the inputs independently across all batches

BN - CV tasks

LN -> NLP tasks -> varying sentence length