Let’s build GPT

Simplest steps (code):

github.com/abarajithan11/nanoGPT/commits/main/building_gpt.py

Based on NanoGPT from Andrej Karpathy:

youtube.com/watch?v=kCc8FmEb1nY

Types of Transformers

Encoder-only

- Extracts information into an intermediate representation
- Eg: sentiment analysis from reviews

Encoder-decoder

- Uses the extracted information to generate new sequences
- Attention is All You Need
- Translation: Spanish → English

Decoder-only

- Generate new sequences based on the input
- GPT-2,3,4, ChatGPT, **NanoGPT - today**

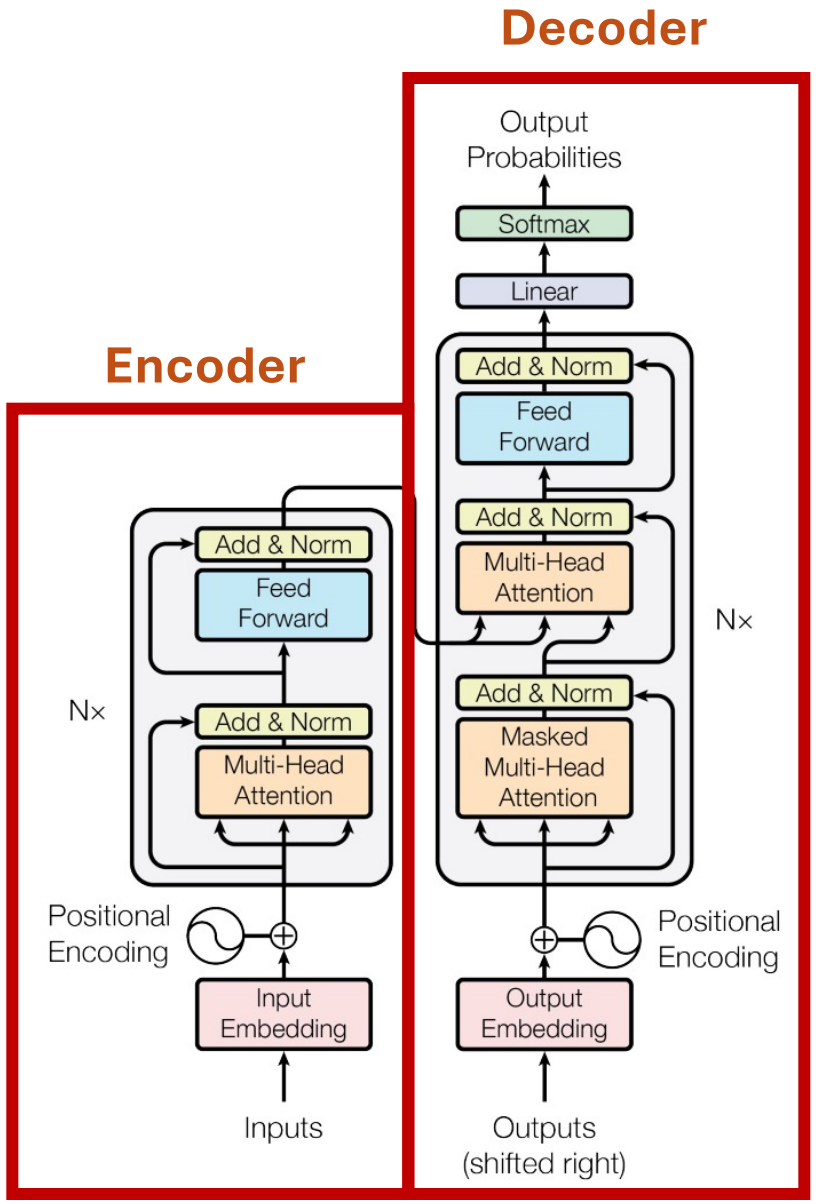


Figure 1: The Transformer - model architecture.

Task:
Learn to write like Shakespeare

- Given few words, generate subsequent text in the style of Shakespeare
- Dataset: Tiny Shakespeare [link]
- Model: Decoder-only Transformer
- Output from NanoGPT, trained for 15 mins

Tiny Shakespeare Dataset

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?
```

Example output

```
NORFirst Haltist:
The bay! I am as yon He litter.
This the more prumpetutalest? why proud you age!

CLARENCE:
Yet now, O, more storna.

FRORINZE

JULINA:
Provoker, your got, my horour lord:
Priveragin, goodip you sileng: Go thy frience more;
You must betteen be that great to shaltle an you see
By somey out of us: if your lords, which are it what you?
```

Final Model

B = 64
T = 256
H = 6
C = 64*H
n_layer = 6

max_iters = 5000
eval_interval = 500
learning_rate = 1e-4
dropout = 0.2

To train & run:
Install pytorch, or use Google Colab

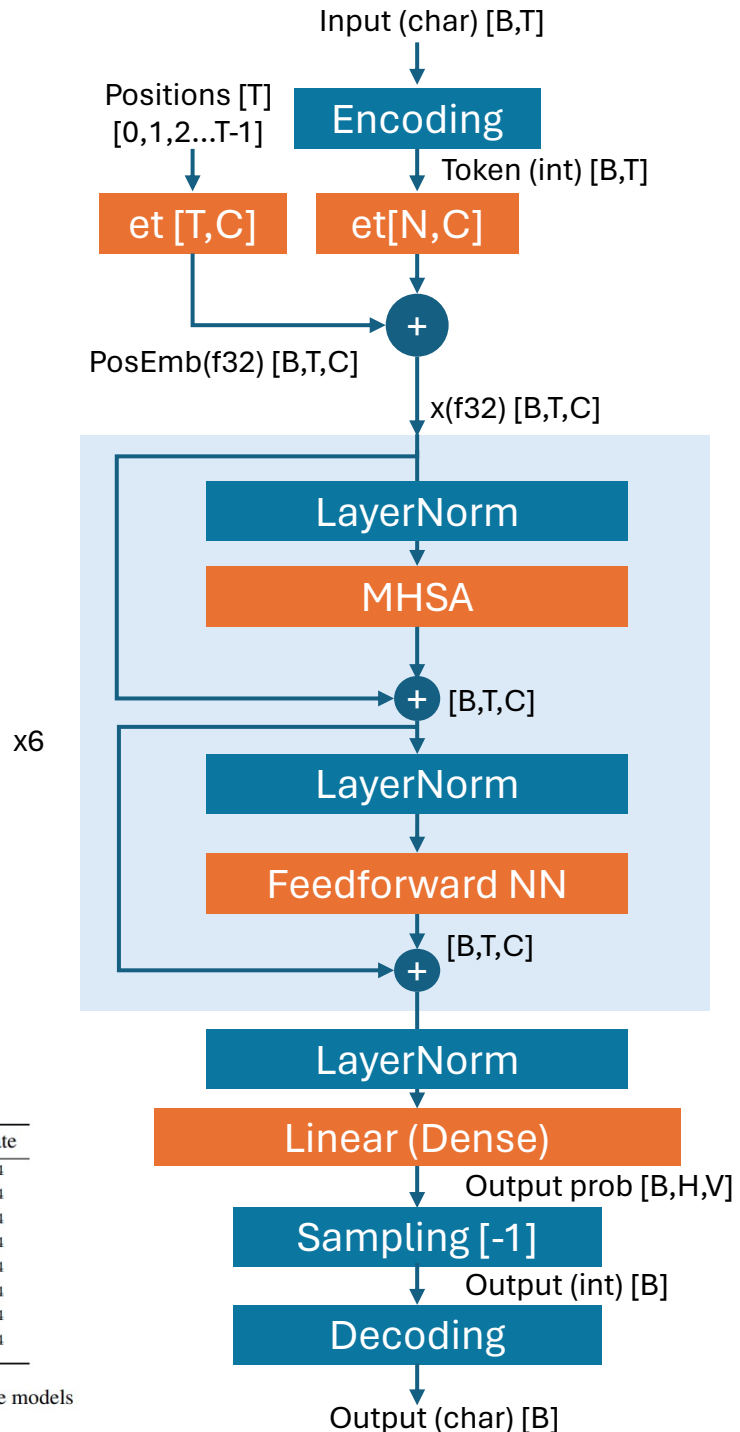
```
git clone
git@github.com:abarajithan11/nanoGPT.git
cd nanoGPT
python building_gpt.py
```

T=2048 C H C/H B

GPT3: arxiv.org/pdf/2005.14165.pdf

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.



RICASTASCAPULAUS:
StOndestrears on lyie.

DUKING grepheeld here frock rup:
Is 'twas uppeontized but din to him.

NORFirst Haltist:
The bay! I am as yon He litter.
This the more prumpetutalest? why proud you age!

CLARENCE:
Yet now, O, more storna.

FRORINZE

JULINA:
Provoker, your got, my horour lord:
Priveragin, goodip you sileng: Go thy frience more;
You must betteen be that great to shaltle an you see
By somey out of us: if your lords, which are it what you?

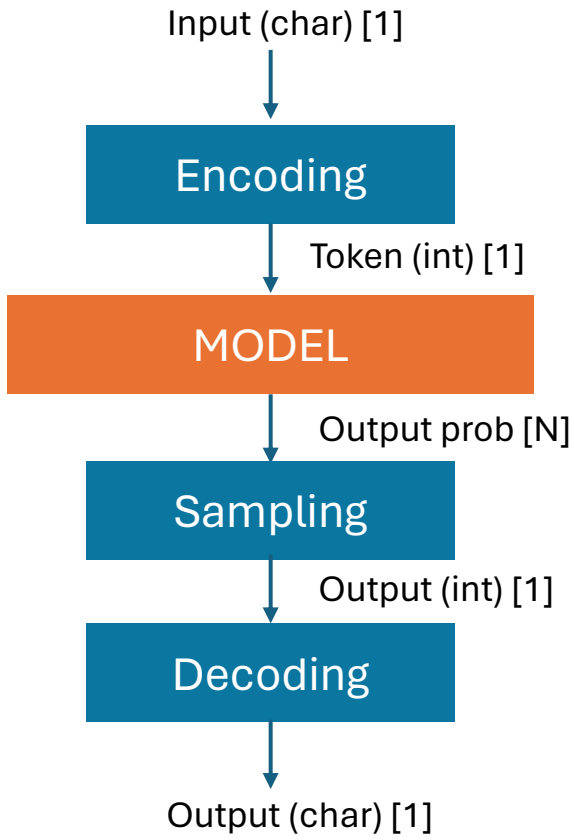
GLOUCESTER:
O thre paperswors I prouse here; what it was not fear the
is ust shalle, service, the noble's
Helm the duty?

BRUTUS:
O'tYou, let us Rome, we putill, for that God-morriestment
So thy attend-rigns, ear that more I do usa
Samep to malie roge. Clain, am she hermined.

ISAABELLA:
O mertaity of honess out act they take
For he farled As illest curity! That it 't;

Simple Tokenization/Encoding

- Find & sort all unique characters in dataset:
  - vocabulary: \n !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
  - vocab_size = 65
- Assign each a unique number: 0-64
- Encoding: Map a string to a vector of integers (tokens)
- Decoding: Map a vector of integers (tokens) to a string



Note: Real Tokenization

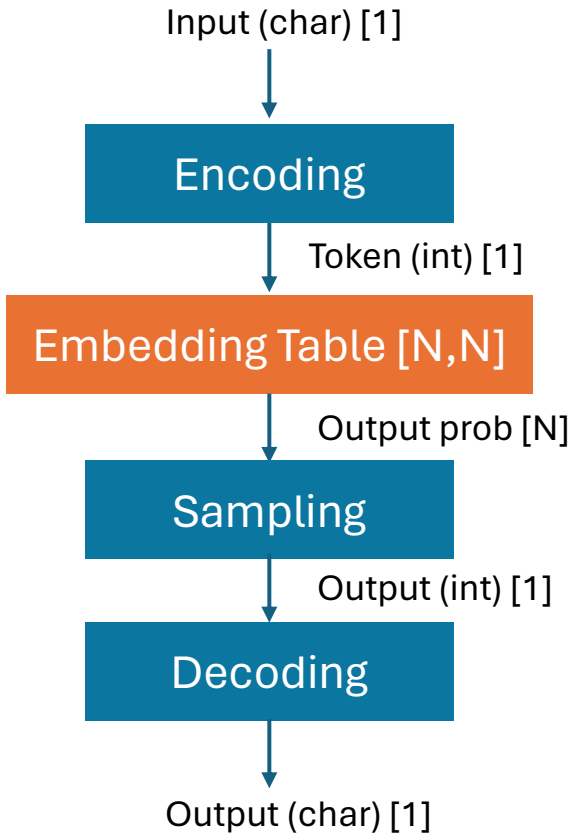
- Common words & sub-words become one integer (token)
- 100,000+ vocabulary
- Enables them to fit more text into the context window, allowing transformer to be more useful
- Try: tiktokenizer.vercel.app/
- More info: youtube.com/watch?v=zduSFxRajkE

Step 0. Bigram Model

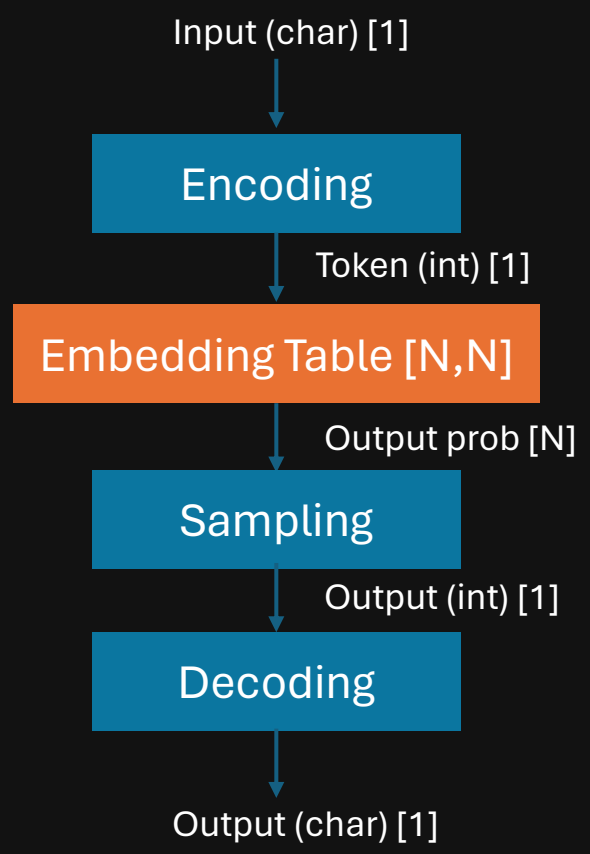
- Simplest language model
- Predicts next token based on this (1) token

- Embedding Table = Learnable lookup table
- Maintains an embedding table of size
  (vocab_size x vocab_size) = (65x65)

- table[input_token] → vector of output probabilities
- Table gets updated with training



```python
class BigramLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):
        logits = self.token_embedding_table(idx) # (B,T,C)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):                    # idx is (B, T) array of indices in the current context
            logits, loss = self(idx)                       # get the predictions
            logits = logits[:, -1, :]                      # (B,T,C) -> (B, C)
            probs = F.softmax(logits, dim=-1)              # (B, C)
            idx_next = torch.multinomial(probs, num_samples=1) # sample from the distribution acc to prob (B, 1)
            idx = torch.cat((idx, idx_next), dim=1)        # New idx is concat (B, T+1)
        return idx

model = BigramLanguageModel()
m = model.to(device)
```



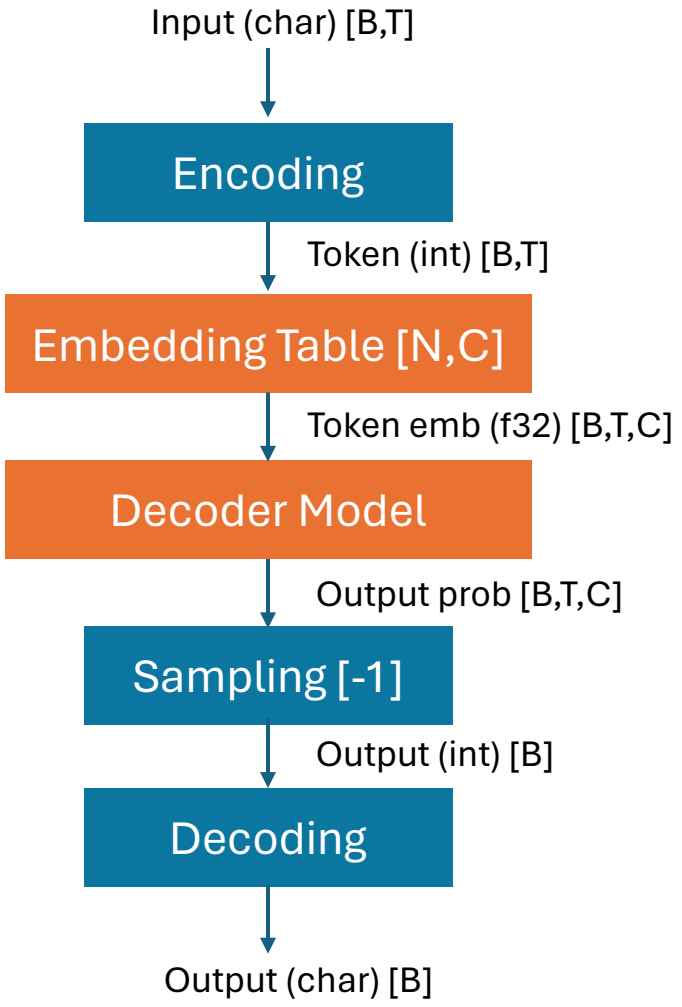
https://github.com/abarajithan11/nanoGPT/blob/89b6f34d565c82b76715017fcea8b9976eacd09c/building_gpt.py

Dimensions [B,T,C]

- **Time** (context window)
  - Number of consecutive tokens processed
  - Eg: GPT-4 Turbo: 128K, Gemini: 1 million
  - Input [T], Output [1]

- **Channels**
  - Each channel contains different information
  - Embedding table [N,C] maps each input token (integer) to a vector of size C.
  - A context window of T tokens becomes a matrix of (T,C)
  - Each layer of a transformer takes in a [T,C] matrix and outputs a [T,C] matrix

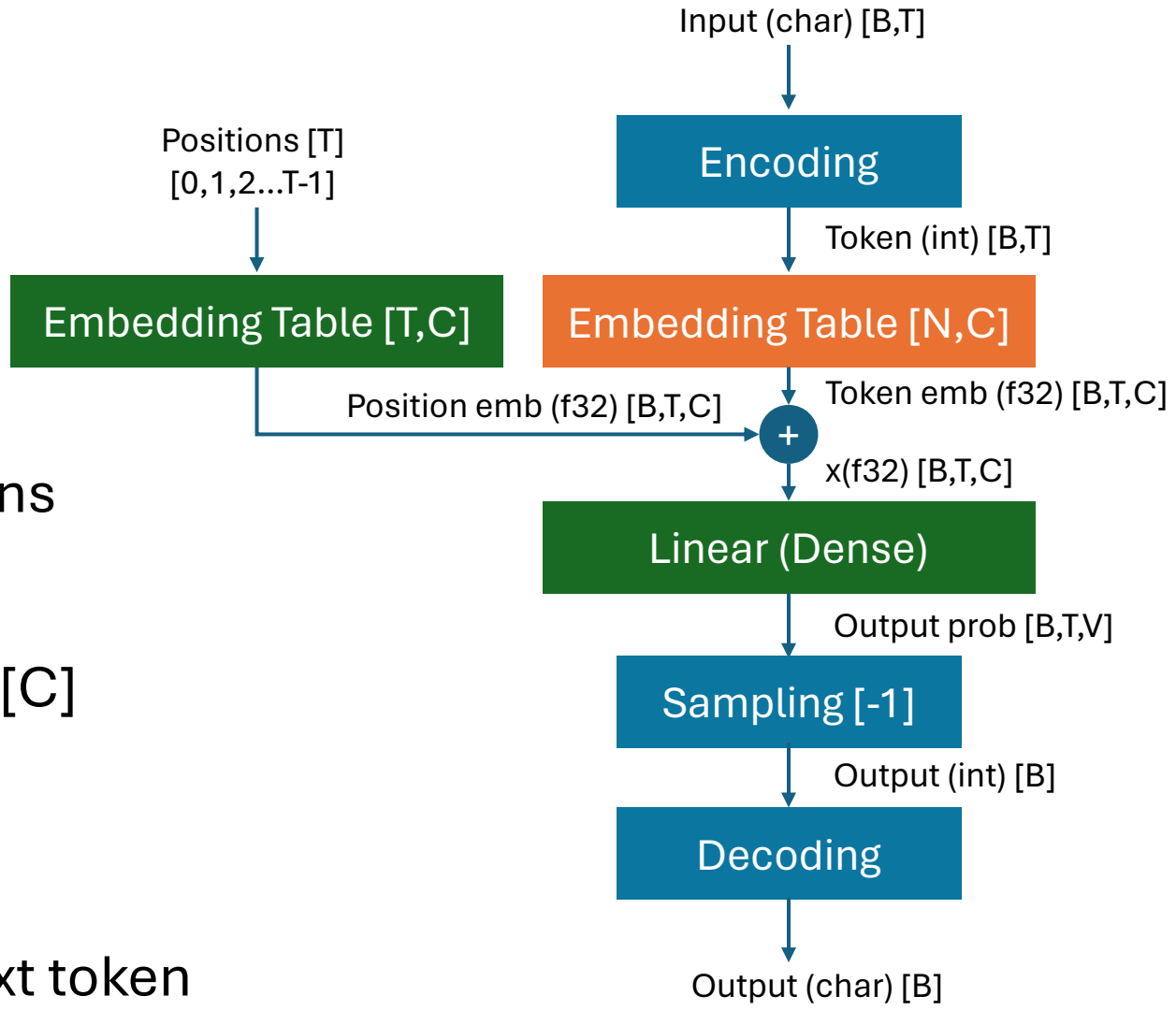
- **Batch**
  - # of [T,C] matrices independently processed in parallel (during training)
  - During inference, batch = 1



Step 1. Linear layer & Positional Embedding

```python
class LanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.tok_emb_table = nn.Embedding(N, C)
        self.pos_emb_table = nn.Embedding(T, C)
        self.lm_head = nn.Linear(C, N)
    def forward(self, idx, targets=None):
        tok_emb = self.tok_emb_table(idx)
        pos_emb = self.pos_emb_table(torch.arange(T))
        x = tok_emb + pos_emb
        logits = self.lm_head(x)
```

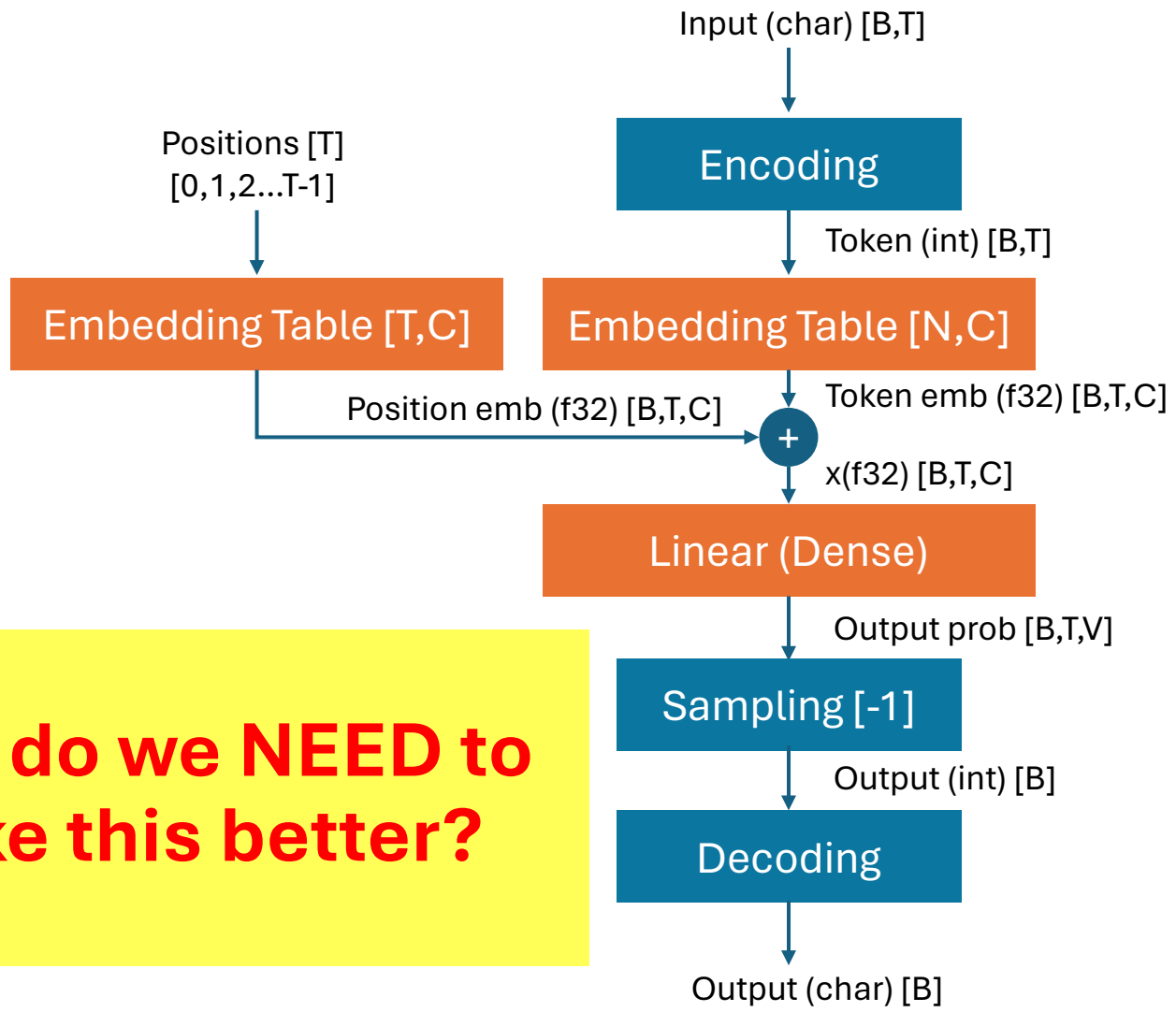
- Model does not know the order of tokens
- Positions: [0,1,2...T-1] are given
- Embedding (pos, token): (int)[1] → (f32)[C]
- Embedding tables are learnt
- Combined by addition
- Linear (=Dense) layer to predict the next token



Step 1. Linear layer & Positional Embedding

```
(torch2) D:\research\nanoGPT>python building_gpt.py
vocab_size: 65
vocabulary:
 !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
step 0: train loss 4.4801, val loss 4.4801
step 300: train loss 2.5404, val loss 2.5566
step 600: train loss 2.5160, val loss 2.5335
step 900: train loss 2.4967, val loss 2.5149
step 1200: train loss 2.5106, val loss 2.5254
step 1500: train loss 2.4853, val loss 2.5109
step 1800: train loss 2.4966, val loss 2.5198
step 2100: train loss 2.4949, val loss 2.5100
step 2400: train loss 2.4937, val loss 2.5102
step 2700: train loss 2.5040, val loss 2.5114


CExthantrid owindikis s, bll

HAPen bube t e.
S:
O:
IS:
Folatangs:
Wanthar u qurthe. bar dilasoate awice my.

Hastatom o mup
Yowhthatof isth ble mil; dilll,

W:

Ye s, hain latisttid ov ts, and Wh pomano.
Swanous l lind me l.
MIshe ce hiry ptupr aisspllw y. w'stoul noroopetelaves
Momy ll, d mothake o Windo wh t eiibys the m douris TENGByore s poo mo th; te

AN ad nthrupt f s ar irist m:

Thin maleronth, af Pre?

Whio myr f-
LI har,
S:


Thardsal this ghesthidin cour ay aney Iry ts I f my ce hy
```

It's pretty bad ☹



What do we NEED to make this better?

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

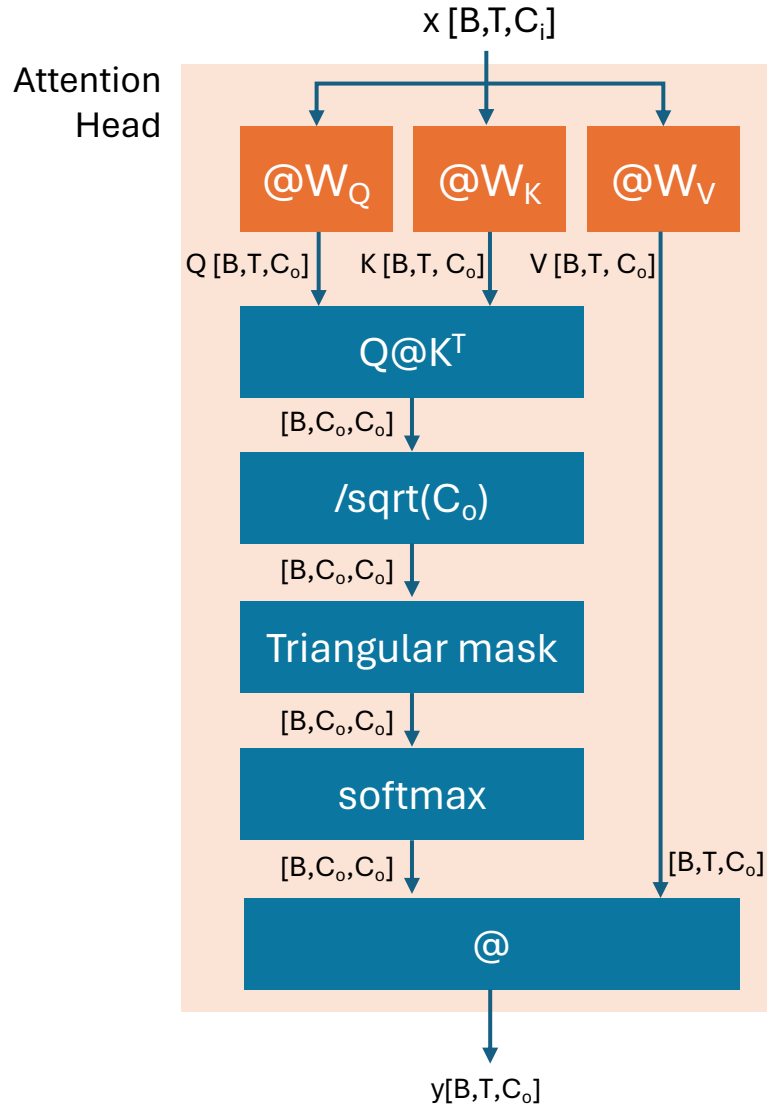
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

arxiv.org/pdf/1706.03762.pdf

Self Attention

- $\mathbf{C_i}$ = input channels, $\mathbf{C_o}$ = output channels
- W_Q, W_K, W_V are $[C_i, C_o]$ matrices, weights of trainable Linear (=Dense) layers
- Query, Key, and Value are projections of x (input data)

```
class Head(nn.Module):
    def __init__(self, Co):
        super().__init__()
        self.key   = nn.Linear(Ci, Co, bias=False)
        self.query = nn.Linear(Ci, Co, bias=False)
        self.value = nn.Linear(Ci, Co, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(B,B)))
    def forward(self, x):
        B, T, Ci  = x.shape
        k = self.key(x)
        q = self.query(x)
        # compute attention scores / affinities
        wei = q @ k.transpose(-2,-1)
        wei /= Co**0.5
        wei  = wei.masked_fill(self.tril[:T,:T]==0, float('-inf'))
        wei  = F.softmax(wei, dim=-1)

        v = self.value(x)
        return wei @ v
```

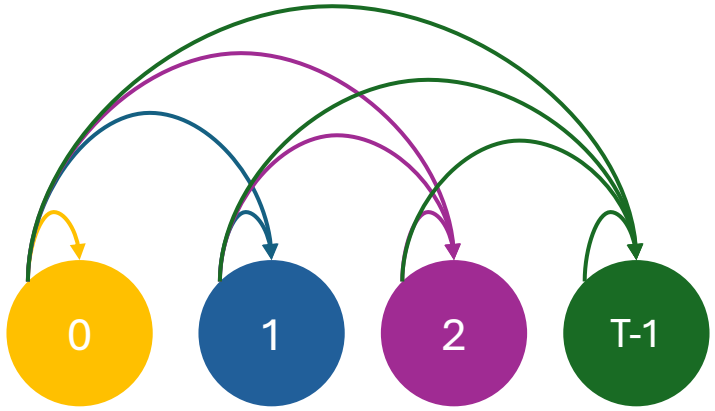


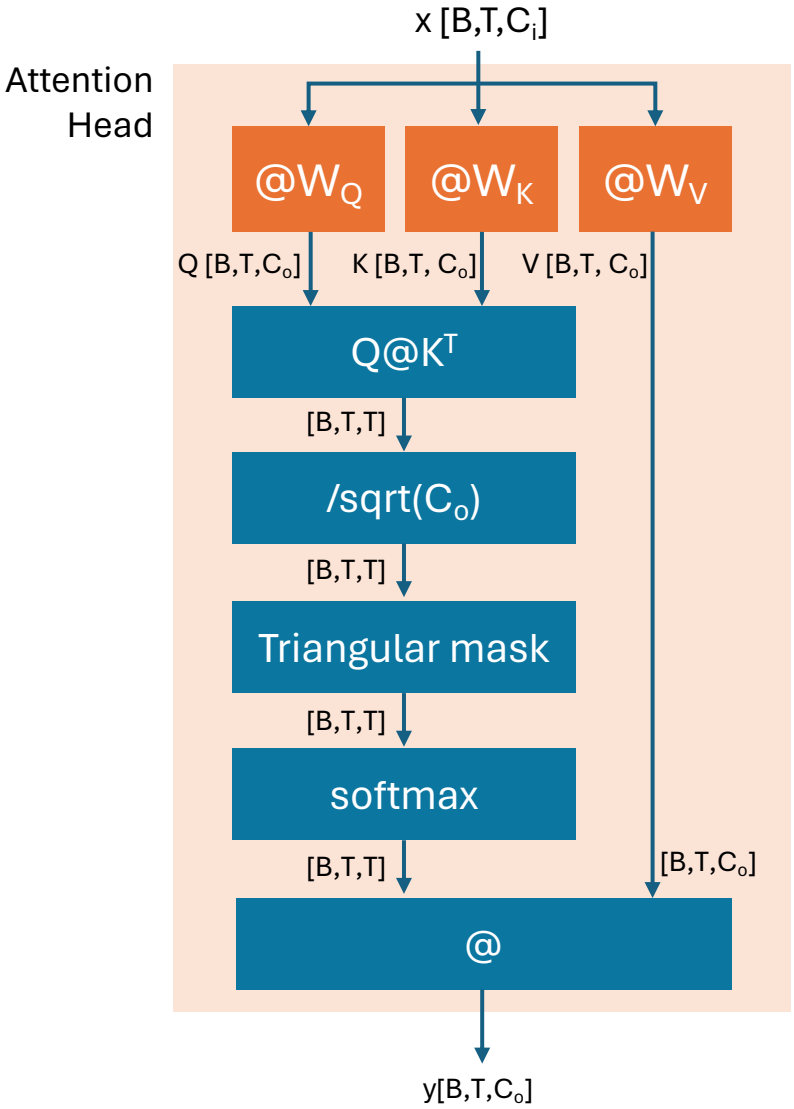
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Attention is Communication

- T (=time) nodes talking to each other

- They each have 3 versions of their raw data x
  - **Query vector**: What I am looking for
    "I am a vowel in position 8, looking for consonants up to position 4"
  - **Key vector**: What I have
    "I am a consonant in pos 3"
  - **Value vector**: What info I am willing to provide



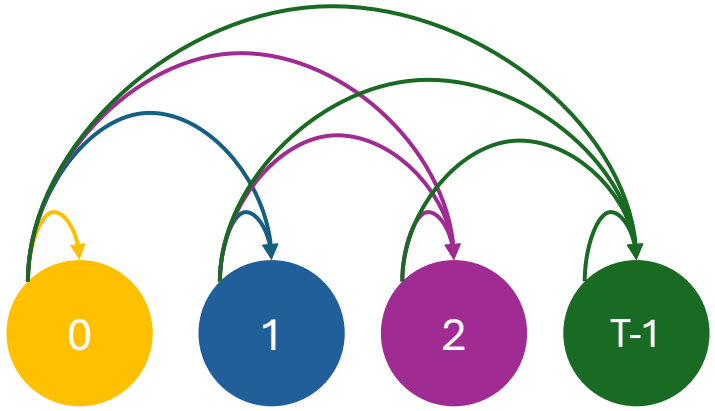


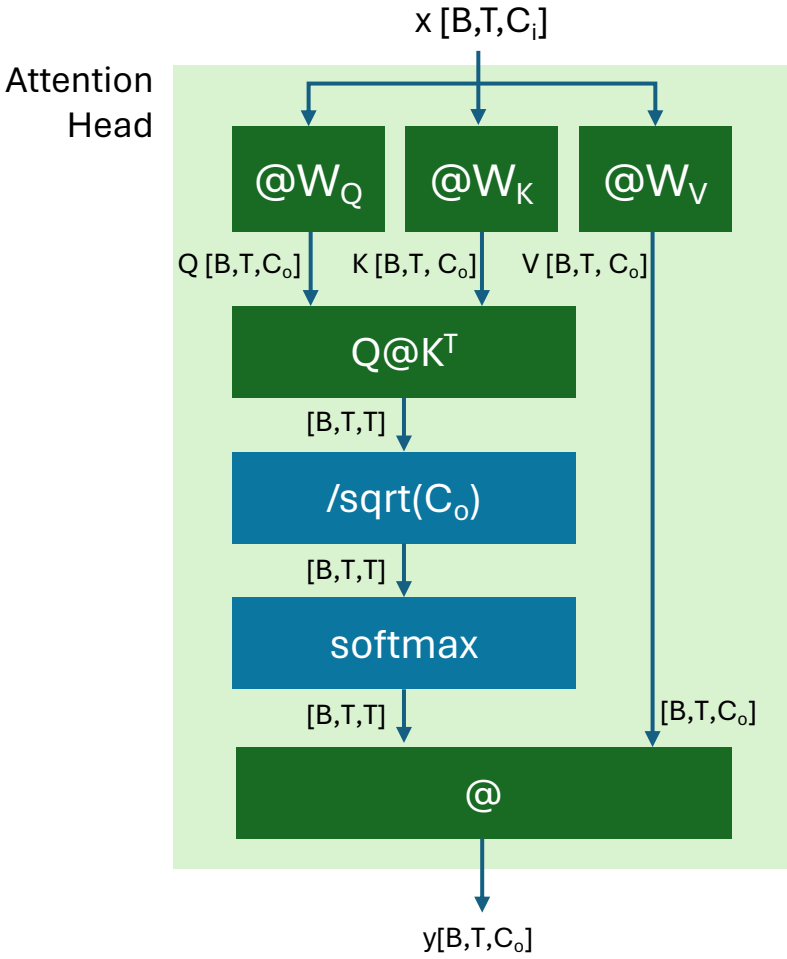
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Attention is Communication

- T (=time) nodes talking to each other

- They each have 3 versions of their raw data x
  - **Query vector**: What I am looking for
    "I am a vowel in position 8, looking for consonants up to position 4"
  - **Key vector**: What I have
    "I am a consonant in pos 3"
  - **Value vector**: What info I am willing to provide



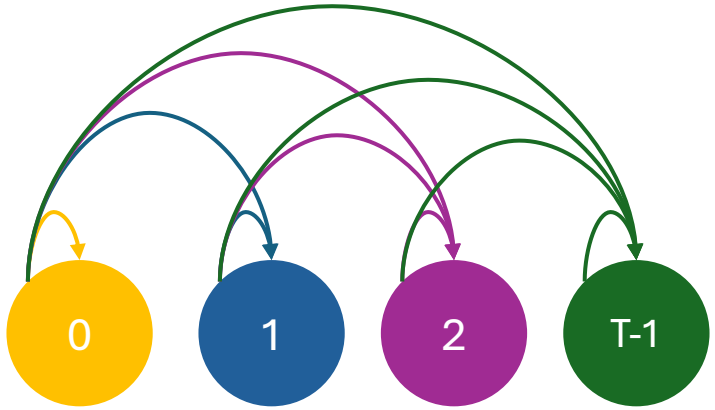


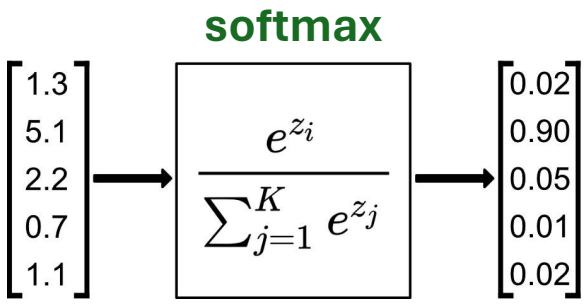
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

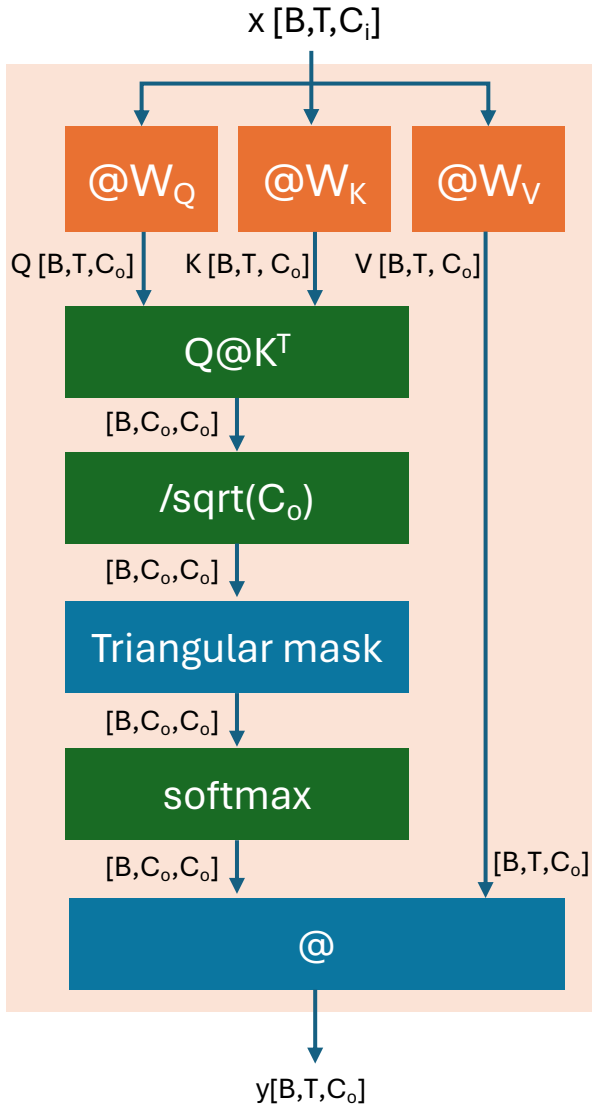
Attention is Communication

- Query vector : What I am looking for
  - "I am a vowel in pos 8, looking for consonants up to pos 4"
- Key vector: What I have
  - "I am a consonant in pos 3"

- **Q@KT = Affinity**
  - Affinities at their intersection will get maximized
  - The vowel has found the consonant it was looking for
  - **/sqrt(C_o)** to normalize the variance to 1
  - **Softmax**: highlights maximized affinities and suppresses others







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

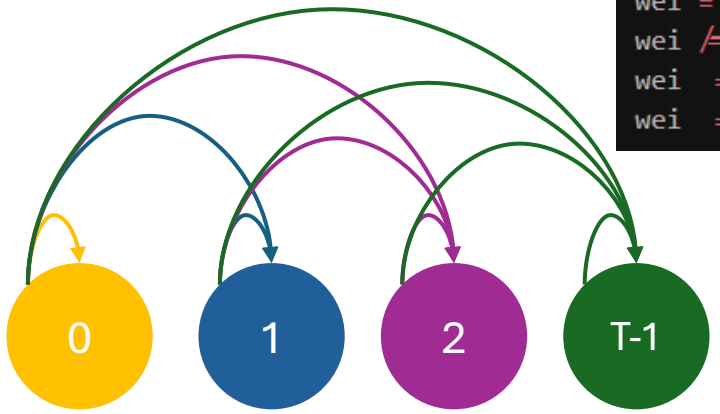
Masking

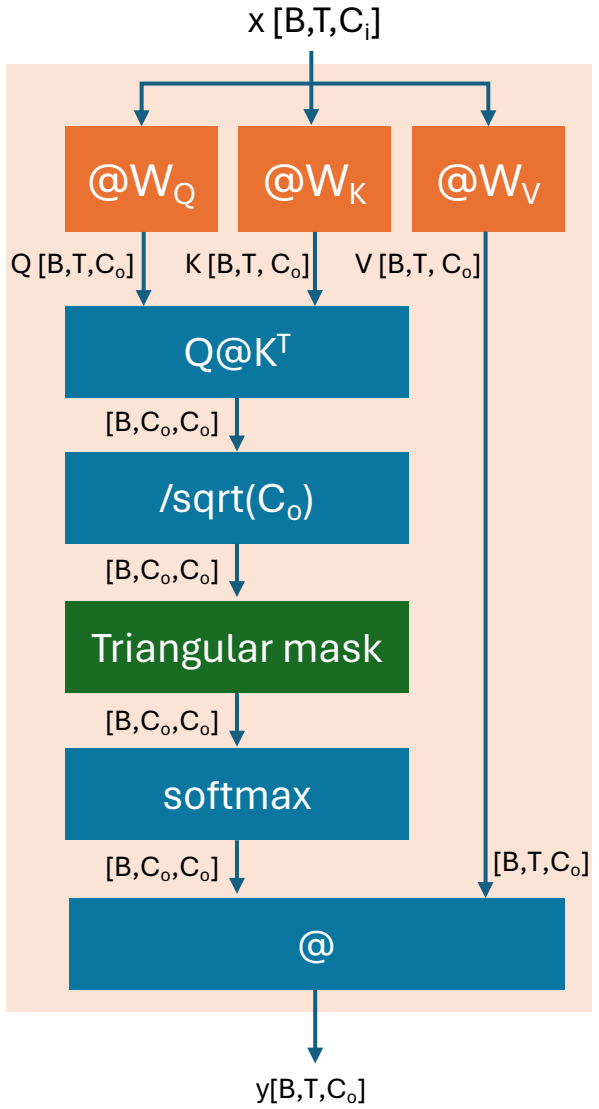
- In decoder-only models (GPT), we want to predict the next word
- We don't want the nodes looking at the answer (future)
- Each node can "talk to" only itself & the past
- Done using a lower-triangular matrix mask

```
>>> torch.tril(a)
tensor([[-1.0813,  0.0000,  0.0000],
        [ 0.0935,  0.1380,  0.0000],
        [-0.3409, -0.9828,  0.0289]])
```

- Mathematical trick:

```
# compute attention scores / affinities
wei = q @ k.transpose(-2,-1)
wei /= C**0.5
wei  = wei.masked_fill(self.tril[:T,:T]==0, float('-inf'))
wei  = F.softmax(wei, dim=-1)
```

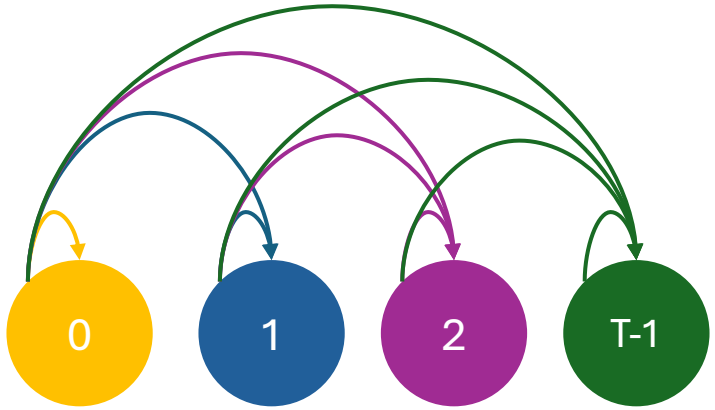


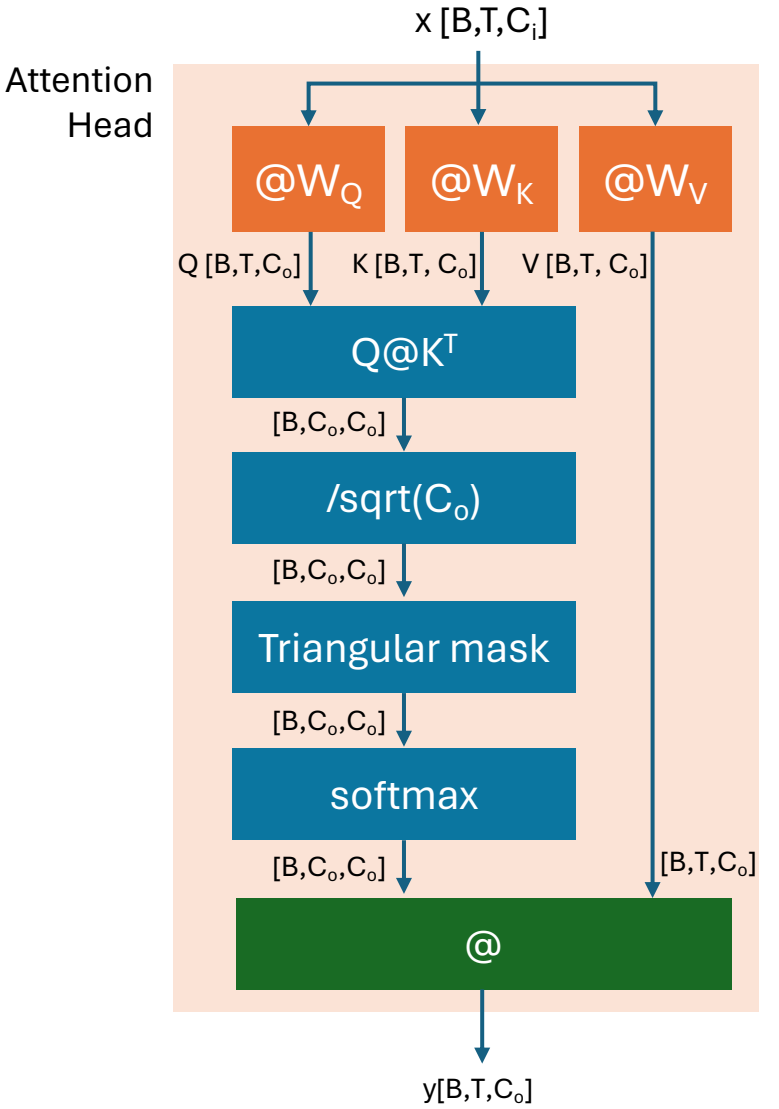


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Attention is Communication

- Value vector: What info I am willing to provide
- **Affinities @ Value** :
  Brings all the information back to each node
- Similar to filtering:
  (Average nearby values and put them here)





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

Note: Other variants of Attention

Encoder:

- No triangular mask
- E.g. translation: You look at all the input words to guess each output word

Self-Attention:

- Q,K,V are generated from same input x

Cross-Attention:

- Q from x; K & V from encoder's output
- E.g: English -> French, french (query) searches in English (key, value)

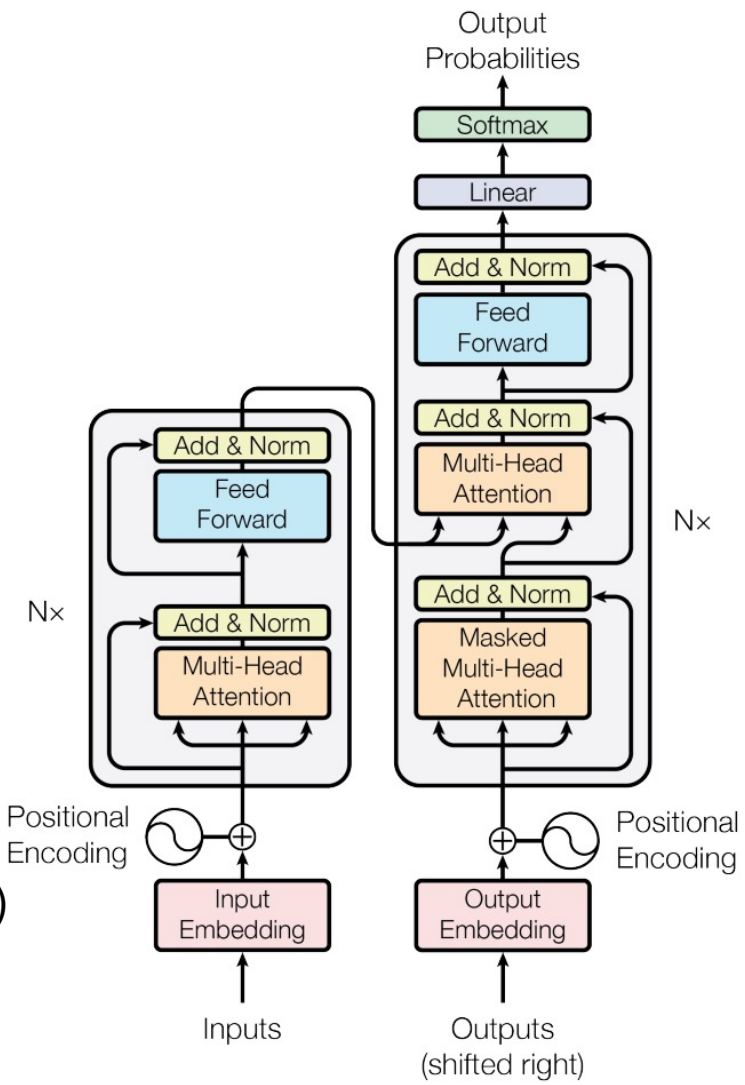


Figure 1: The Transformer - model architecture.

Step 2. Adding Attention to our model

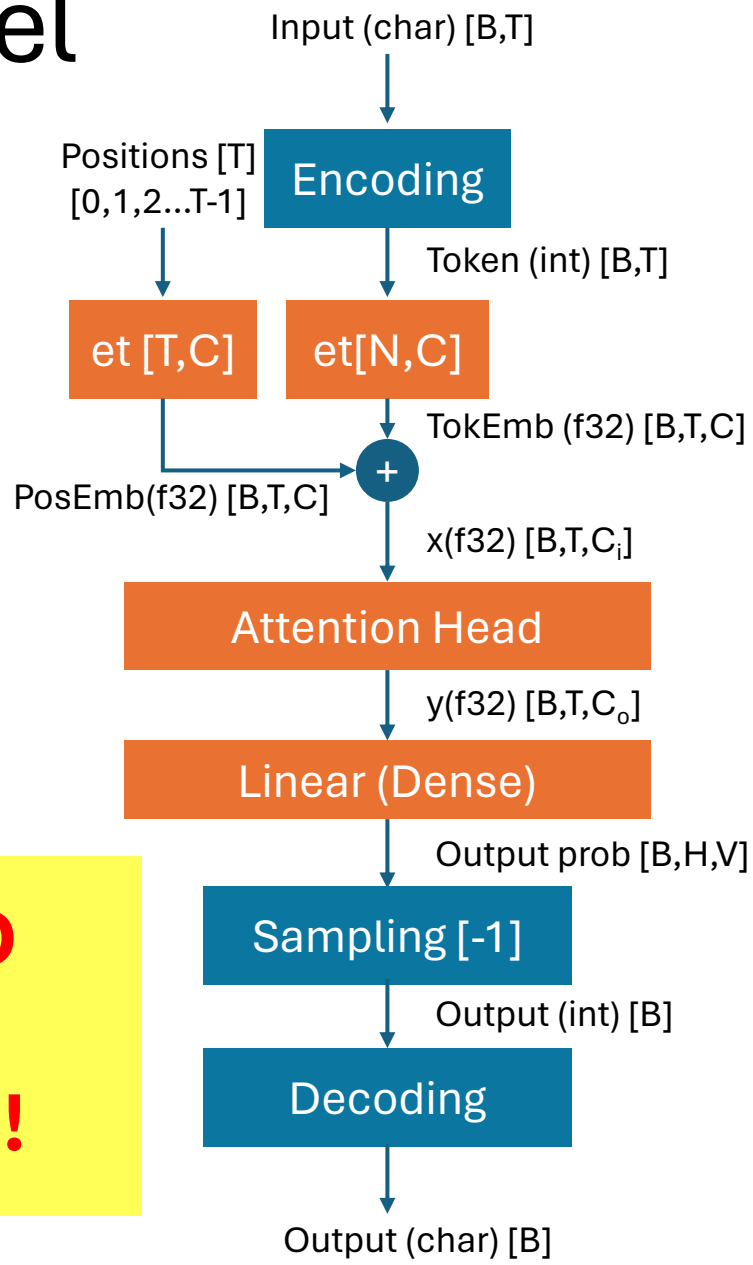
- Code: github.com/abarajithan11/nanoGPT/tree/fc640703b3c2b465076bab4bd384ece1043fc8ff
- Result:

```
vocab_size: 65
vocabulary:
 !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
step 0: train loss 4.2000, val loss 4.2047
step 500: train loss 2.6911, val loss 2.7087
step 1000: train loss 2.5196, val loss 2.5303
step 1500: train loss 2.4775, val loss 2.4829
step 2000: train loss 2.4408, val loss 2.4523
step 2500: train loss 2.4272, val loss 2.4435
step 3000: train loss 2.4130, val loss 2.4327
step 3500: train loss 2.3956, val loss 2.4212
step 4000: train loss 2.4041, val loss 2.3992
step 4500: train loss 2.3980, val loss 2.4084
```

Slightly better

```
Whent aknt,
Thowi, ht son, bth

Hiset bobe ale.
S:
O-' st dalilanss:
Want he us he, vet?
Wedilas ate awice my.

HDET:
ANGo oug
Yowhavetof is he ot mil ndill, aes iree sen cie lat Herid ovets, and Win ngarigoerabous lelind peal.
-hule onchiry ptugr aiss hew ye wllinde norod atelaves
Momy yowod mothake ont-wou whth eiiby we ati dourive wee, ired thoouso er; th
To kad nteruptef so;
ARID Wam:
ENGCI inleront ffaf Pre?
```

We NEED more attention!

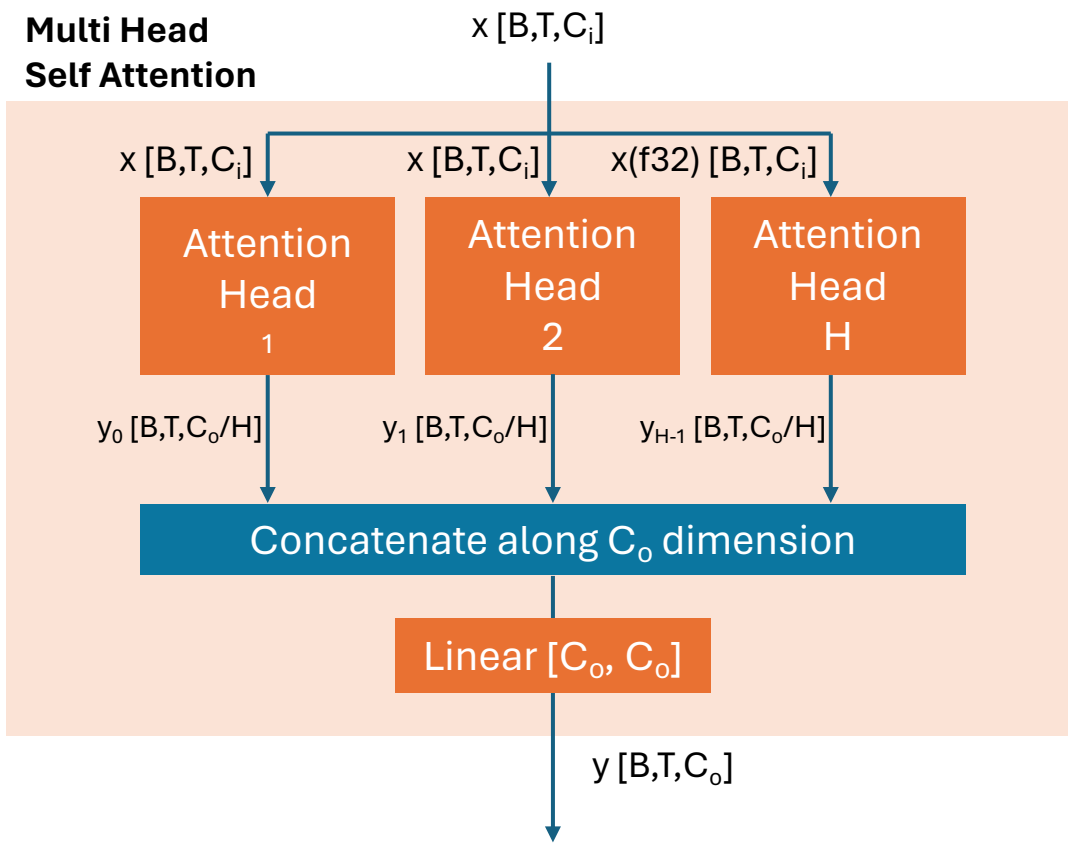


Step 3: Multi-Head Self Attention

- Transformers are wide & short (not too deep) models
- To maximize parallelization, and train on massive GPUs

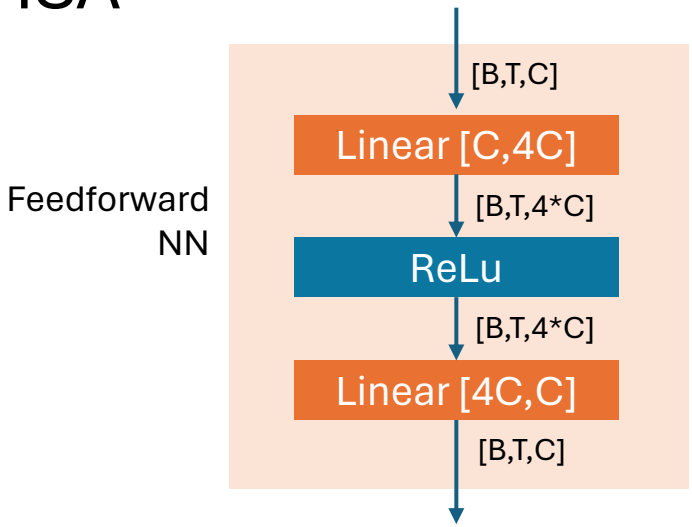
MHSA

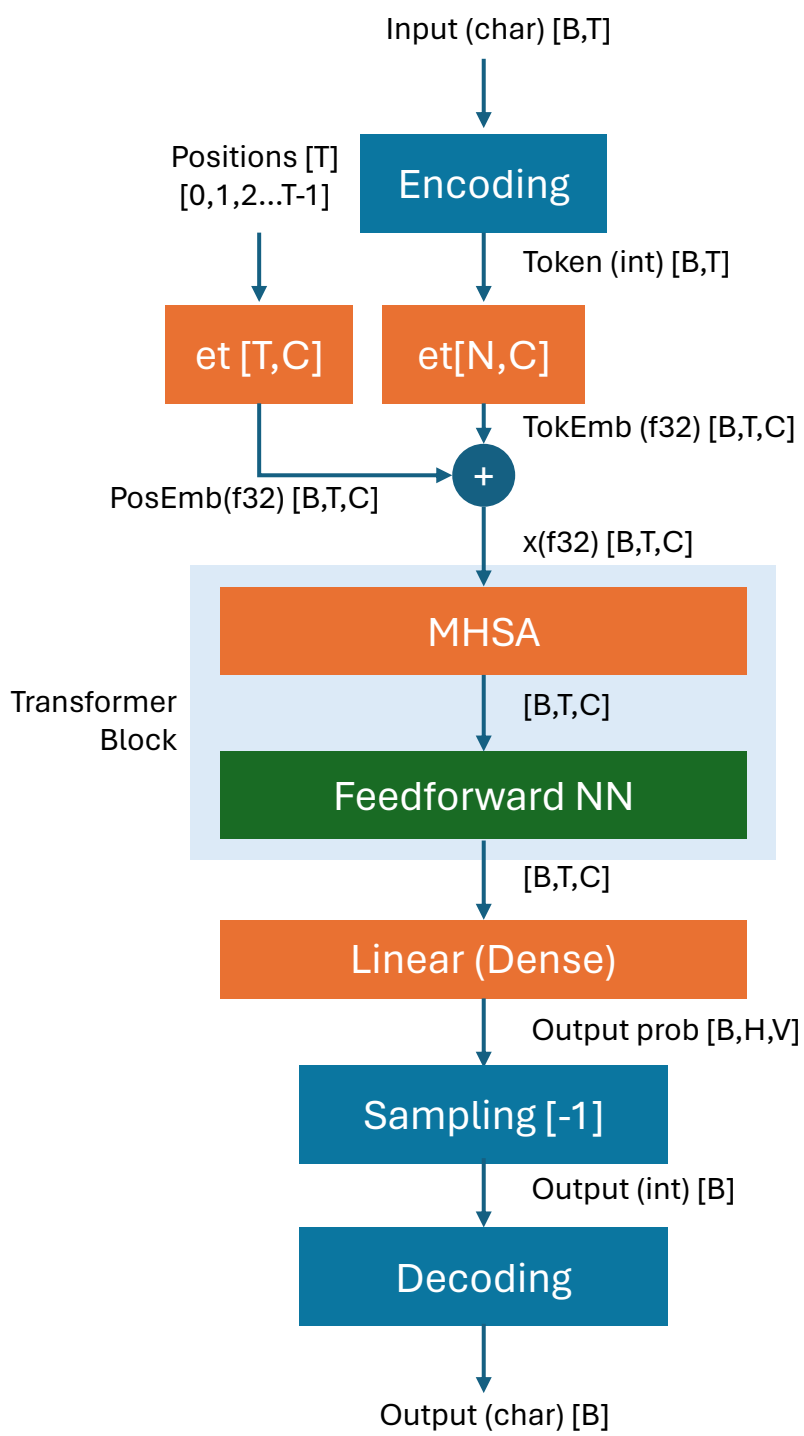
- H attention heads, each with $\mathbf{C_i}$ input channels and $\mathbf{C_o/H}$ output channels
- Each head learns different things: sentiment, active/passive…etc
- Outputs are concat'd along Co
- A linear layer added to output (projection)
- Model with MHSA



Step 4: Feedforward NN

- Dense / fully connected neural network after MHSA

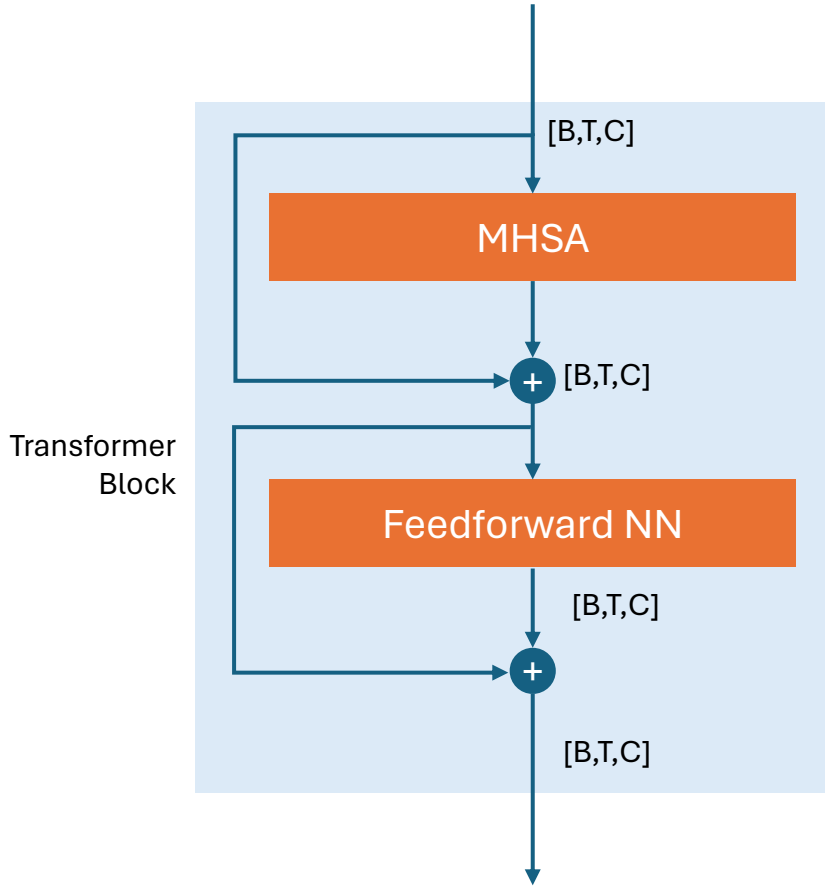




- So the model can “think about” what it learnt through MHSA
- Code

Step 5: Residual Connections

- When a model gets deeper, gradient "vanishes" before it backpropagates to the top
- Cannot train the model
- Solution: Add skip connections that allow gradient to jump over and reach the top [paper]



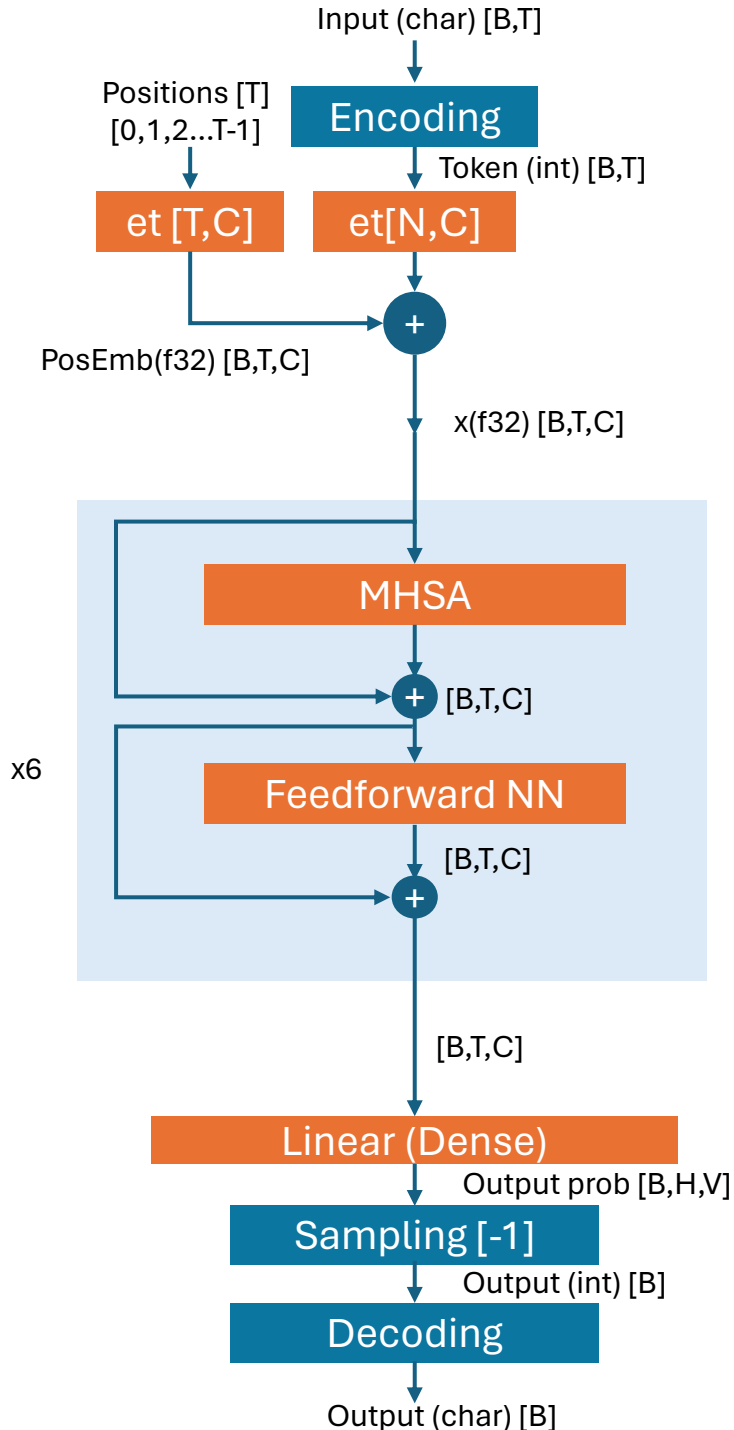
Step 6: Multiple Blocks

- Replicate the transformer blocks

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

arxiv.org/pdf/2005.14165.pdf

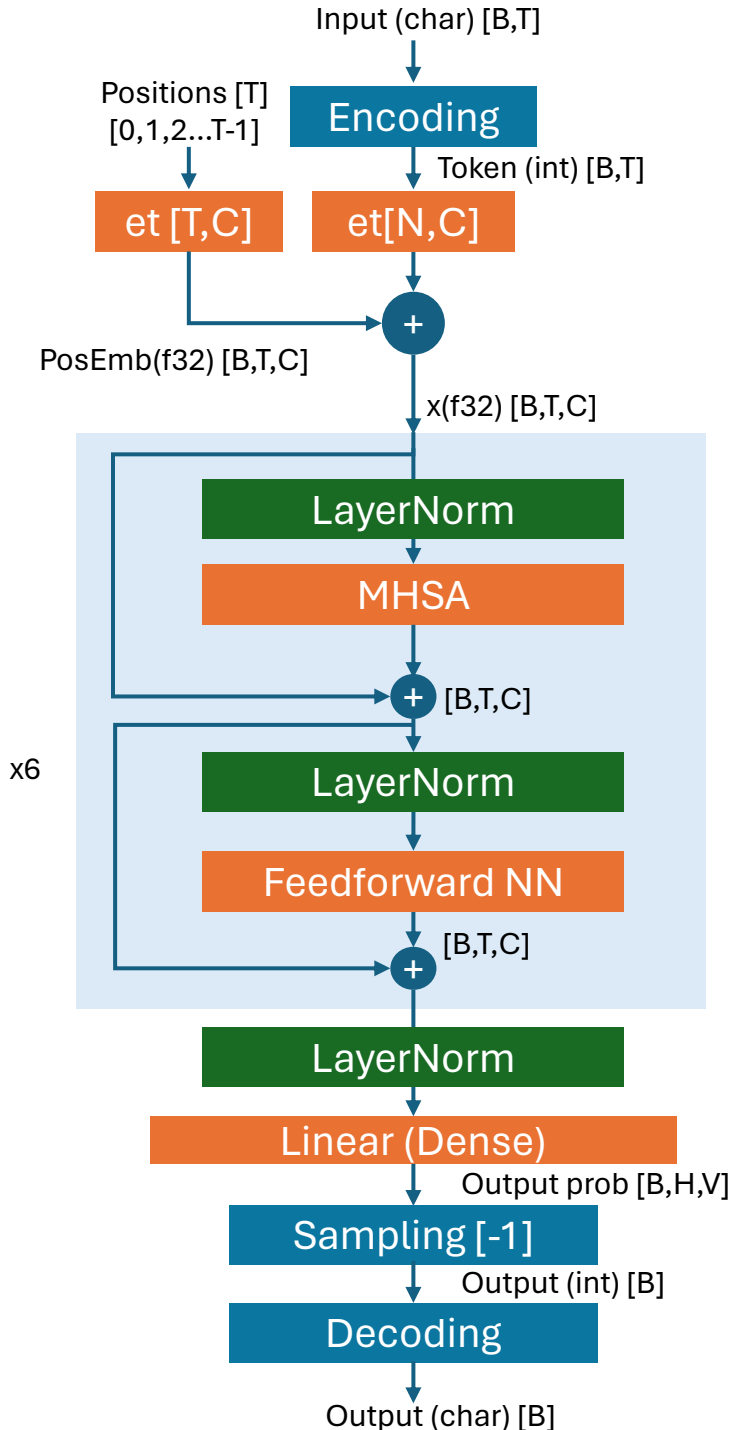


Step 7: LayerNorms

- Similar to batchnorm in CNNs
- Helps training deeper networks
- Normalizes inputs based on training data.
- But applied along C dimension only
- γ, β are trainable scalers
- In the original paper, was added to output, but nowadays its used at the input

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

arxiv.org/pdf/1607.06450.pdf



Step 8: Dropouts

- During training, dropout layers randomly set a given percentage of inputs to zero
- Helps the model to be robust (not depending on certain nodes too much)
- Here, dropout layers are added to
  - affinities (softmax output) in attention heads
  - At the end of MHSA
  - At the end of each block
- During inference, dropouts have no effect

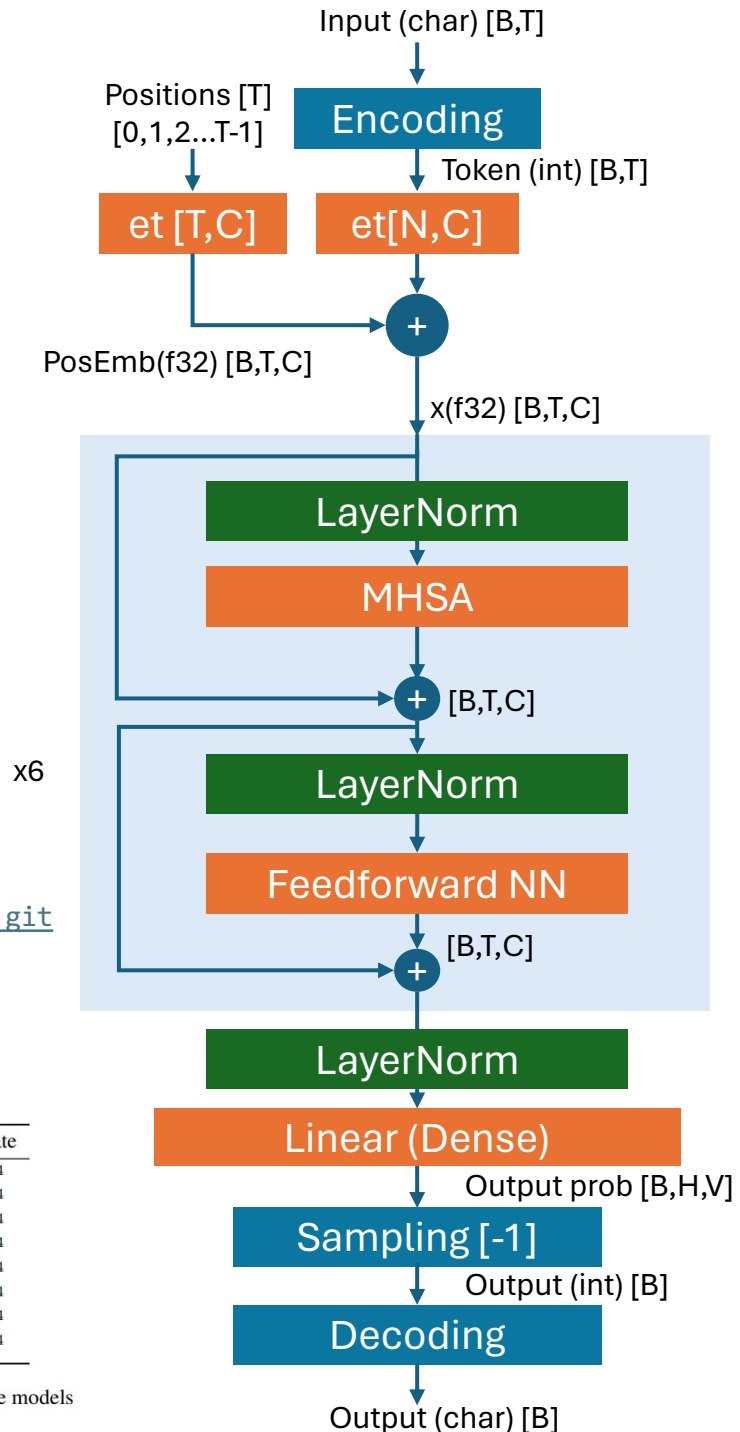
Final Model

B = 64
T = 256
H = 6
C = 64*H
n_layer = 6

max_iters = 5000
eval_interval = 500
learning_rate = 1e-4
dropout = 0.2

To run:
Install pytorch, or use Google Colab

```
git clone git@github.com:abarajithan11/nanoGPT.git
cd nanoGPT
python building_gpt.py
```



T=2048 C H C/H B

GPT3: arxiv.org/pdf/2005.14165.pdf

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

RICASTASCAPULAUS:
StOndestrears on lyie.

DUKING grepheeld here frock rup:
Is 'twas uppeontized but din to him.

NORFirst Haltist:
The bay! I am as yon He litter.
This the more prumpetutalest? why proud you age!

CLARENCE:
Yet now, O, more storna.

FRORINZE

JULINA:
Provoker, your got, my horour lord:
Priveragin, goodip you sileng: Go thy frience more;
You must betteen be that great to shaltle an you see
By somey out of us: if your lords, which are it what you?

GLOUCESTER:
O thre paperswors I prouse here; what it was not fear the
is ust shalle, service, the noble's
Helm the duty?

BRUTUS:
O'tYou, let us Rome, we putill, for that God-morriestment
So thy attend-rigns, ear that more I do usa
Samep to malie roge. Clain, am she hermined.

ISAABELLA:
O mertaity of honess out act they take
For he farled As illest curity! That it 't;