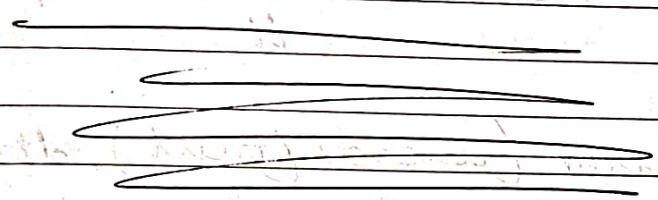
Create a Large Language Model from Scratch with Python

- Freecodecamp.org

* Packages to install
  numpy, matplotlib, pylzma, ipykernel, jupyter, torch

* open the file (read mode)

→
```
with open('wizard_of_oz.txt', 'r', encoding='utf-8') as f:
    text = f.read()
print(text[:200])
```
→ 

Lets sort these characters inside a list

→
```
chars = sorted(set(text))
print(chars)
```
→ [' ', '!', '2', - - -]

* we have 81 characters

classmate
Date
Page

Code to tokenize these text

here we are just mapping each char to a number

→ string_to_int = { ch : i for i, ch in enumerate(chars) }
int_to_string = { i : ch for i, ch in enumerate(chars) }

encode = lambda s: [string_to_int[c] for c in s]
decode = lambda l: ''.join([int_to_string[i] for i in l])

print (encode('hello'))

→ [61, 58, 65, 65, 68]

if we are working on different language we will have a lot of tokens even like millions.

so we convert our data into tensors so that our model processes it

→ data = torch.tensor (encode(text), dtype = torch.long)

print (data[:100])

* Next topic is on train, val, test sets
why we do it

we know why we do it
so we move on.

* Premise of Bigram model

hello

start of content → h
h → e
e → l
l → l
l → o

Bi means 2
so here we have associative probability

at start it is h
next after h, e is more probability
now
after e, l is more probability
to come after e

and so on
this is Bigram Model
on considers the previous char
and next char

block_size = 5 → the size of the block from the entire set we are gonna iterate on

.... [5, 67, 21, 82, 54] 35 ....

starts 5 moves to 5 blocks after 54 predicts and gives 35

... 5 [67, 21, 82, 54, 35] ....

next steps now start is 67 and 5 blocks after 35

so stride = 1

and the error to minize is the predicted and actual char

so we have snippet

[: block_size]

next

[1 : blocksize + 1]

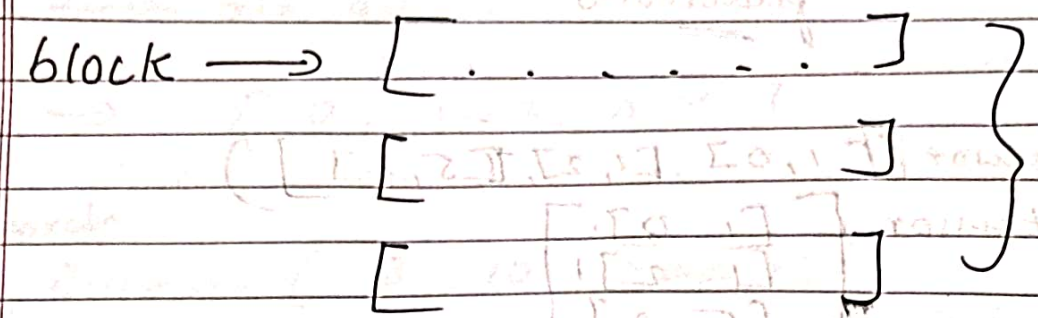
* same thing in code

→ block_size = 8

x = train_dat [: block_size]

y = train_____ [1 : block_size + 1]

for t in range (block_size):
    context = x [: t+1]
    target = y[t]
    print ('when input is', context, 'target is', target)



→ ~~if~~

device = 'cuda' if torch.cuda.is_available() else 'cpu'

print (device)
block_size = 8
batch_size = 4

Some basic Pytorch functions

→ torch.tensor([[1,0],[1,2],[5,2]])

→ tensor([[1 0],
[1 2],
[5 2]])

→ torch.randint(-100, 100, (6,))

→ tensor([4 -53 54 -87 35 98])

→ torch.zeros(2,3)

→ tensor([[0 0 0],
[0 0 0]])

→ torch.ones(2,3)

→ same but 1

→ torch.empty(2,3)

→ [1. ... e+22 ...]
[...]

some large and small values

⇒ torch.arange(5)

→ [0, 1, 2, 3, 4]

torch

⇒ .linspace(3, 10, steps=5)

* basically increment from 3 to 10 in constant way in 5 steps

→ [3.00, 4.750, 6.500, 8.250, 10.000]

⇒ torch.eye(5)

→
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

gives diagonal matrix

CPU vs GPU Performance

→ device = 'cuda' if torch.cuda.is_available() else 'CPU'

→ cuda

* Well we know that GPU performs better

* In order to multiply matrices in pytorch we need to use @

(torch_tensor_1 @ torch_tensor_2)

More Pytorch functions

* torch.multinomial
* define a probability tensor
→ probabilities = torch.tensor([0.1, 0.9])
10% => for 0 , 90% for 1
* Draw 10 samples from multinomial distribution
sample = torch.multinomial(probabilities, num_samples=10, replacement=True)

→ Tensor([1, 1, 1, 1, 1, 1, 0, 1, 1, 1])

* Torch.cat

→ # to concatinate 2 tensors

* Torch.tril(torch.ones(5,5))

[1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
1 1 1 1 0
1 1 1 1 1

* torch.tril
* same as before but

[1 1 1
0 1 1
0 0 1]

reverse

* .Transpose()

* Torch.Stack([tensor_1, tensor_2, tensor_3])

[ ] 3 values
[ ]
[ ]

Stack them
and shape
becomes 3 x 3

~~impo~~
~~from~~ torch ~~import nn~~
→ import torch.nn as nn

This a module where all the layers are present
nn.Linear
nn.ReLU
nn.Conv2d
... more

nn.Embedding

This a layer where a char or word which is sent to a model is sent along with a vector which holds the sentiment or importance of the char as this vector is trainable and hence is a layer

embedding_size = 5 → hyper parameter

char level

a	[	0.1	0.2	0.6	0.8	0.3]
b	[	0.2	0.5	0.1	0.2	0.5]
c	[	0.9	0.7	0.2	0.6	0.1]
:						
z	[					]

* Matrix Multiplication

* Dot product

$[1, 2, 3] \cdot [4, 5, 6]$

$(1*4) + (2*5) + (3*6) = 32$

a is 3 x 2 mat

b is 2 x 3 mat

a @ b

3 x 2 2 x 3 = 3 x 3

this must be same

if 3 x 4 5 x 3

not compatable for multiplication

must do some operations for make it work like swap

5 x 3 3 x 4 = 5 x 4

classmate
Date
Page

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$b = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

$(1 \times 7) + (2 \times 10) = 27$

$(1 \times 8) + (2 \times 11) = 30$

$(1 \times 9) + (2 \times 12) = 33$

$(3 \times 7) + (4 \times 10) = 47$

66

$$\begin{bmatrix} 27 & 30 & 33 \\ 47 & 68 & 75 \\ 97 & 106 & 117 \end{bmatrix}$$

In code

→ a = Torch.tensor [[1 2 ———]]

b = ———

print (a @ b) or (torch.matmul (a, b))

→ same tensor

* Cant mat mul on int and float
must convert any into same type by casting

→ _____ = float()

Model demo for embeddings

→ Class BigramModel(nn.module):

```
def __init__(self, vocab_size):
    super().__init__()
    self.Token_embed_tabel = nn.embedding(vocab_size, vocab_size)

def forward(self, index, targets=None):
    logits = self.token_embedding_table(index)
    B, T, C = logits.shape
    logits = logits.view(B*T, C)
    if targets is None:
        loss = None
    else:
        targets = targets.view(B*T)
        loss = F.CrossEntropy(logits, targets)
    return logits, loss
```

in between code

→ if targets is none: loss = none else:

What are logits, view function for

logits is basically our table of ~~one~~ bigram

a a times is 52 a — a aa 52 ba . — b ab bb — c — d — follow ups

b

c

d

:

and view ?

B → batch_size
T → sequence length

C → vocab_size used as no. of classes for predictions

B, T, C = logits.shape

break down the shape

logits = logits.view(B*T, C)

Example:-

B = 4

T = 8

C = 100

logits.shape = (4, 8, 100)
targets.shape = (4, 8)

→ why .view(B*T, C)

so we need to flatten the predictions and targets so the cross_entropy() can work properly

```
logits = logits.view(B*T, C)
        # (4*8, 100) => (32, 100)

targets = targets.view(B*T)
        # (4*8) => (32,)
```

as there are some expected input shape

logits -> (N, C) → (B*T, C)
targets → (N) → (B*T)

so .view reshapes to a flat list of predictions so loss can be calculated.

* torch.long is same as torch.int64

* Continue the model

→ def generate(self, index, max_new_tokens):

for _ in range(max_new_tokens):

logits, loss = self.forward(index)

logits = logits[:, -1, :] #becomes (B, C)

probs = F.softmax(logits, dim=-1)

index_next = torch.multinomial(probs, num_samples=1)

index = torch.cat((index, index_next), dim=1) # (B, T+1)

return index

model = BigramModel(vocab_size)
m = model.to(device)

context = torch.zeros((1, 1), dtype=torch.long, device=device)

→ generated_chars = decode(m.generate(context, max_new_tokens=500)[0].tolist())

print(Generated Chars)

→ max_iters = 10000

learning_rate = 3e-4

* Training loop

→ optimizer = torch.optim.AdamW(model.parameters(), lr = learning_rate)

```
for iter in range(max_iters)

    # sample a batch of data
    xb, yb = get_batch('train')
```

→ This a dataloader basically will give in batches.

```
    #evaluate the loss
    logits, loss = model.forward(xb, yb)
    optimizer.zero_grad(set_to_none = True)
    loss.backward()
    optimizer.step()
    print(loss.item())
```

Optimizers Overview

Some common ones

* MSE
* Gradient descent - to min loss function of learning model

Momentum - adds intuition of GD with added of "momentum" term. helps smooth out and updates, useful for deep neural network

RMSprop

Adam - ideal combines momentum and RMSprop

AdamW - modified Adam that adds weights decay to the params. can improve generalization performance

Loss Reporting + Train Vs Eval mode

→ eval_iters = 250
dropout = 0.2

#

→ @torch.no_grad() ⟶ Gradience is not happening here
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = loss.mean()
    model.train()
    return out

→ # between your training loop

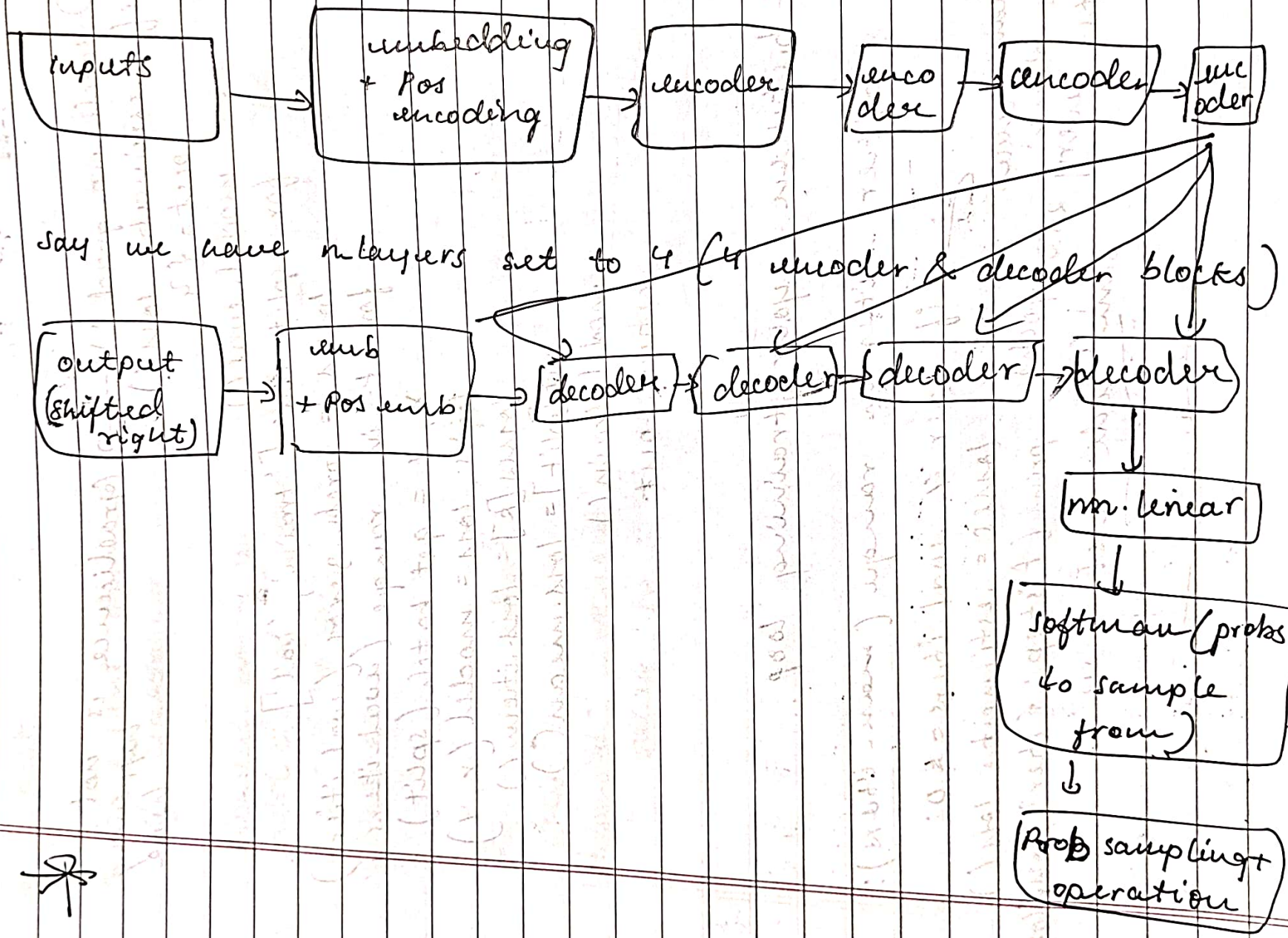
after iter in range(max_iters):

    if iter % eval_iters == 0:
        losses = estimate_loss()
        print(f"step: {iter}, loss{losses}"

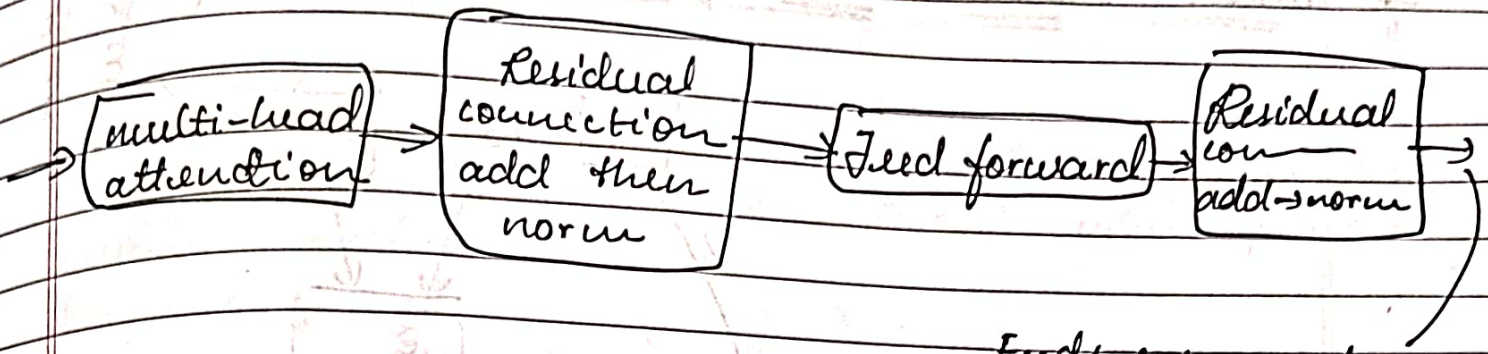
mean eval is done after every 250 epochs

same code next

Transformer Models

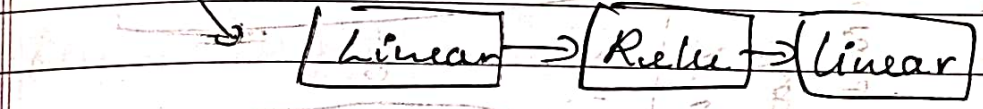


* Inside encoder



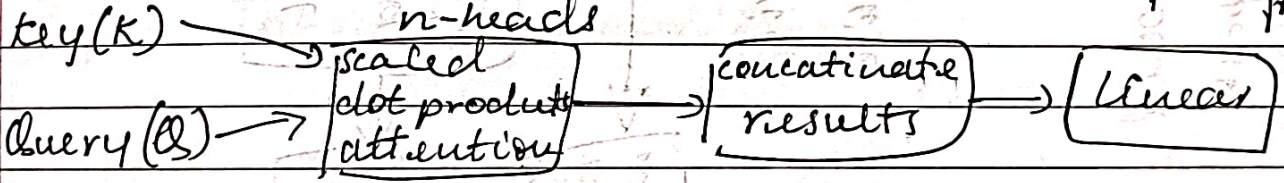
Feeds into next encoder block. If its the last encoder block it feeds into each decoder block.

* Inside Feed Forward



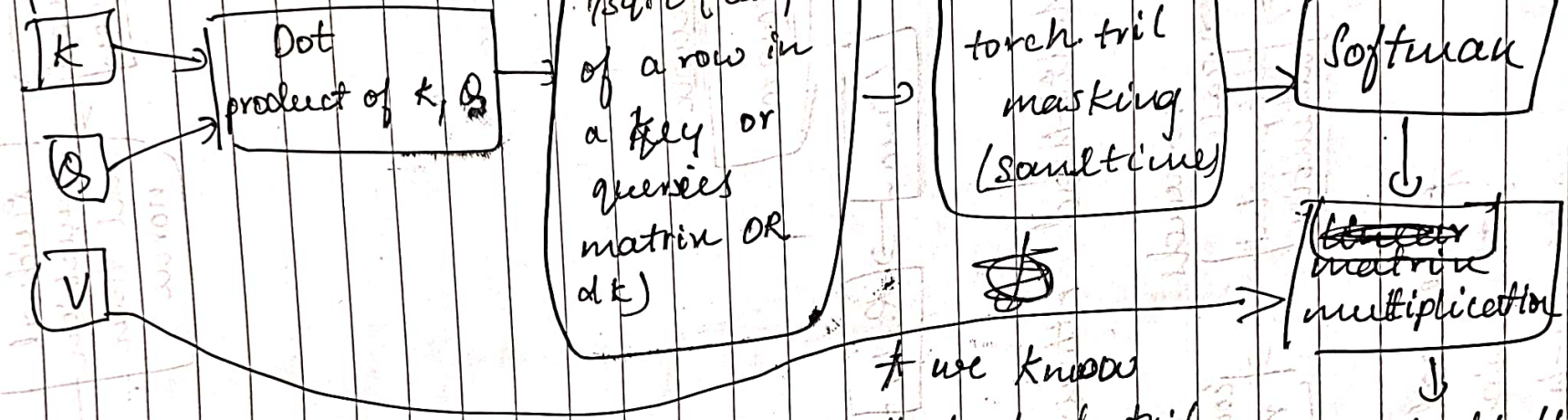
* Inside Multi-head Attention

scaled dot product runs n heads in parallel from GPU



* we call it multi head attention because there are n-heads learning bunch of different semantic info from a unique perspective

know that these values are learnt by passing through a neural network



* Inside each head of attention

* we know that torch.tril gives [1 0 0 / 1 1 0 / 1 1 1] like this kind of so that predictions should be based on past and must have no clue of future as it is masked and only used in masked multi-head layer.

How this works

Self-Attention allows a model to focus on different parts of the input when processing each word.

"The animal didint cross the street because it was to weird"

when processing the word "it" the model needs to understand whether "it" refers to "animal" or "street".

Self Attention helps with that

Key, Query Value?

These are vectors derived from input embedding and are used in the attention mechanism

Input: Token Embeddings

Suppose

"The cat sat"

You convert each word into a vector (embedding).

Suppose

x_1 = embedding("The")
x_2 = " ("cat")
x_3 = " ("sat")

Each of these vectors is then transformed into

Q
K
V

This is done using learnable weight matrices

$Q = x \cdot W_Q$
$K = x \cdot W_K$
$V = x \cdot W_V$

where:
W_Q, W_K, W_V are trainable matrices (params of the model)

So for each token, you generate its

Q: What this word is asking for
K: what this word contains
V: What this word carries as info

How Attention is Calculated?

The idea is to compare the Query of the current word to the keys of the other words, to compute how much attention should be paid to each word.

Steps:

1. Dot product of Query and keys

2. Scale.
   To avoid large values (especially with high dimensional vectors), scale by $\sqrt{d}$ (d = dimension of key)

3. Softmax

4. Multiply by Value

All together:

Imagine you're processing the token "sat":

- You use the Q of "sat"
- you compare it with key of ["The", "cat", "sat"]
- Based on these comparison (dot products), you compute attention weights.
- Then, you take a weighted sum of the value of ["The", "cat", "sat"]

This gives a new context-aware representation of "sat", influenced by the words around it.

Visualization:

inputs

"The" "cat" "sat"

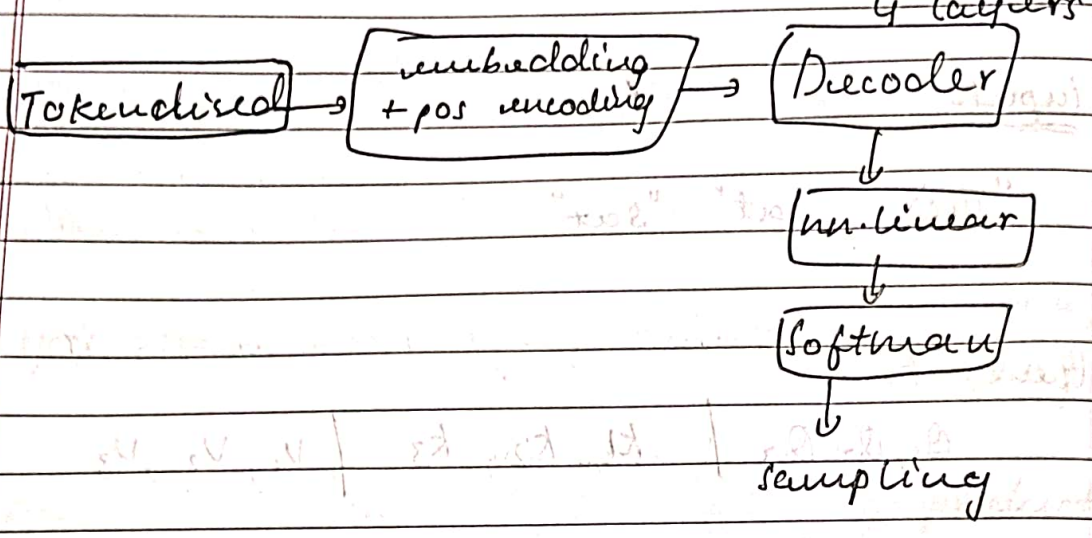
Generate:

Q_1, Q_2, Q_3 | K_1, K_2, K_3 | V_1, V_2, V_3

To process "cat":

1. Take Q_2 (Q of "cat")
2. Compare with $K_1\ K_2\ K_3$ → get attention score
3. Softmax → attention weights : [0.1, 0.8, 0.1]
4. weighted sum: $0.1 \cdot V_1 + 0.8\, V_2 + 0.1 \cdot V_3$ → new embedding for "cat"

Our GPT model:



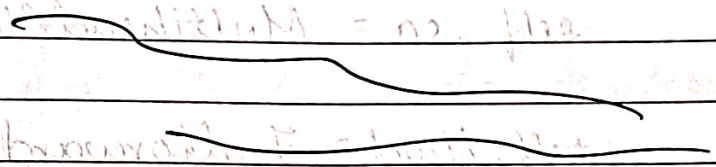
Our model:

```
class GPTModel (nn.Module):
    def __init__(
        super(
        self.token_embedding(vocab_size, n_embd)
        self.positional_embedding_table = nn.Embedding(block_size, n_embd
decoder ← self.blocks = nn.Sequential(*[Block(n_embd, n_head = n_head)
                                        for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)
```

```
def forward (self, index, targets = None):
    logits = self.token_embedding_table(index)

    token_emb = self.token_em_____(idx)
    pos_emb = self.pos_____(T, device
                            = device)
    x = token_emb + pos_emb  # (B, T, C)
    x = self.blocks(x)
    x = self.ln_f(x)
    x = self.lm_head(x)

    if target is none:
        loss = None
```



```
# lets initialize initial weights
    self.apply(self._init_weights)

def _init_weights(self, module):
    if isinstance(module, nn.Linear):
        torch.nn.init.normal_(module.weight, mean
                              = 0.0,
                              std = 0.02)
        if module.bias is not None:
            torch.nn.init.zeros_(module.bias)
    elif isinstance(module, nn.embedding):
        torch.nn.init.normal_(module.weight,
                              mean = 0.0, std = 0.02)
```

* This should be in __init__

n_heads = 4
→ n_embd = 384
n_head = 4
dropout = 0.2

* Our block model Inside Decoder

→ Class Block(nn.Module):

```
def __init__(self, n_embd, n_head):
    super().__init__()
    head_size = n_embd // n_head
    self.sa = MultiheadAttention(n_head, head_size)

    self.ffwd = Feedforward(n_embd)
    self.ln1 = nn.LayerNorm(n_embd)
    self.ln2 = nn.LayerNorm(n_embd)

def forward(self, x):
    y = self.sa(x)
    x = self.ln1(x+y)
    y = self.ffwd(x)
    x = self.ln2(x+y)
    return x
```

Let's make our Feedforward

```
→ class FeedFor___(nn.module):
    def __i___
      super()___

      self.net = nn.sequential(
                 nn.linear(n_embd, 4*head_size),
                 nn.ReLU(),
                 nn.dropout(dropout),
                 )
    def forward(self, x):
          return self.net(x)
```

Now make our Multi-head attention class

```
→ class Multi___(nn.mod__):

    def __i___:
      sup___
      self.heads = nn.modulelist([head(head_size)
                                  for _ in range(num_
                                  heads)])
      self.proj = nn.linear(head_size * num_
                            heads, n_embd
                            )
      self.dropout = nn.dropout(dropout)

    def forward(self, x):
      out = torch.cat([h(x) for h in self.heads]
                      , dim = -1)
```

```
out = self.dropout(self.proj(out))

return out
```

Head Class

→

```
class Head(nn.module):

    def __init__(self, n_embed, head_size, block_size, dropout):
        super().__init__()
        self.key = nn.Linear(n_embed, head_size, bias=False)
        self.query = nn.Linear(n_embed, head_size, bias=False)
        self.value = nn.Linear(n_embed, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones
                                (block_size, block_size)))
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B,T,C = x.shape
        k = self.key(x)   # (B, T, hs)
        q = self.query(x) # (B, T, hs)

        wei = q @ k.transpose(-2,-1) * k.shape[-1]
                                    ** -0.5
        # (B, T, hs) @ (B, hs, T) → (B, T, T)
```

head size

```
wei = wei.masked_fill(self.tril[:T, :T]
                      == 0,
                      float('-inf'))
wei = F.softmax(wei, dim=-1)
wei = self.dropout(wei)
v = self.value(x)
out = wei @ v
return out
```

* What is difference between Sequential module and modulelist module.

both are same
where sequential depends on the previous layer to complete and process the same

module list runs everything in parallel so they are isolated and have different inputs, and different outputs
this is what we want in head.

classmate
Date
Page

* Now start training by troubleshooting some errors.
and adjust the hyperparams

and use different dataset or large which is good one but mind your computation capability

* We are gonna use OpenWebtext data
which has more than 20,000
so have to use for loops
data extract file

so we gotta save all these txt into one .txt file

do chatgpt
for code

and also split the data into train, val

train 90%
val 10%

* Add a dataloader to feed batches to the model.

* Now that we are training we want to save our model weights and so we use something called
→ import pickle

→ # after training loop
a file will get saved

```
with open('model-01.pkl', 'wb') as f:
    pickle.dump(model, f)
```

* Now we can save the model and load into the model GPT() and do more epochs further

→ model = GPTModel (vocab_size)
print ('loading model...')
with open ('model-01.pkl', 'rb') as f:
model = pickle.load(f)
print ('loaded')

m = model.to(device

* Now do the training loop and the model will resume training

* Our bot script after all

→ while True:
prompt = input ("prompt: \n")
content = torch.tensor(encode(prompt), dtype=torch.long, device=device)
generated_chars = decode(m.generate(content.unsqueeze(0), max_new_tokens=150)[0].tolist())
print (f'completion: \n {generated_chars}')

* Make sure the block_size and max_new_tokens is less or else we ~~cause~~ can have error

or we can do crop index to the last block_size tokens

→ index_cond = index[:, block_size:]
keeps in block_size limit

Research and Development Pointers

Time and efficiency testings

→ import time

```
start = time.time()

for i in range(1000):
    print(i*2)
end = time.time()
total = end - start

print(f'time taken: {total}')
```

* Can be used to time and operation or check how long our model takes to load.

* Quantized Llms
instead of 32 bit floating point
it uses 16 bit float point
which we learnt
in CV course.

* Gradient Accumulation
ChatGPT this to understand

* Hugging Face
* We have models, datasets, and more
* should explore.
Do a course