

Nama	Muhammad Abyan Ridhan Siregar
Kelas	TK-45-01
NIM	1103210053

Final Exam Report

Building GPT from Scratch

Page 1 : Step by Step

```
# Mengimpor pustaka requests untuk mengirim permintaan HTTP
import requests
```

```
# URL dari file teks yang akan diunduh
url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt"
# Mengirim permintaan HTTP GET untuk mendapatkan data dari URL
response = requests.get(url)
# Membuka atau membuat file lokal bernama 'input.txt' dalam mode tulis dengan encoding UTF-8
with open("input.txt", "w", encoding="utf-8") as f:
    # Menulis isi teks yang diambil dari URL ke dalam file lokal
    f.write(response.text)
```

- import requests**
 - Baris ini mengimpor pustaka requests, yang digunakan untuk mengirim permintaan HTTP ke sebuah URL (misalnya, untuk mengambil data dari internet).
- url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt"**
 - Baris ini mendeklarasikan variabel url yang berisi alamat URL dari file teks yang akan diunduh.
- response = requests.get(url)**
 - Baris ini mengirim permintaan HTTP GET ke URL yang didefinisikan di atas, dan menyimpan responsnya dalam variabel response. Respons ini berisi data yang diambil dari URL tersebut.
- with open("input.txt", "w", encoding="utf-8") as f:**
 - Baris ini membuka (atau membuat) file bernama input.txt dalam mode tulis ("w") dengan encoding UTF-8. File ini akan digunakan untuk menyimpan data yang diambil.
- f.write(response.text)**
 - Baris ini menulis isi teks dari respons (response.text) ke dalam file input.txt. Data dari URL tadi disimpan ke file lokal agar bisa digunakan kembali.

Kesimpulan: Kode ini mengunduh file teks dari internet dan menyimpannya ke dalam file lokal bernama input.txt.

```
# Membuka file 'input.txt' dalam mode baca ('r') dengan encoding UTF-8
with open('input.txt', 'r', encoding='utf-8') as f:
    # Membaca seluruh isi file dan menyimpannya dalam variabel 'text'
    text = f.read()
```

Inti dari kode ini: Kode ini membuka file input.txt yang sebelumnya dibuat, membaca seluruh isinya, dan menyimpan data tersebut ke variabel text.

```
print("length of dataset in characters: ", len(text)) # Menampilkan panjang dari var text
```

Penjelasan :

jadi code di atas bertujuan untuk menampilkan panjang character yang ada di dalam len(text)

```
length of dataset in characters: 1115394
```

Menampilkan panjang character yang ada didalam text yaitu sebanyak 1115394 character

```
# Menampilkan 1000 karakter pertama dari variabel 'text'
print(text[:1000])
```

Penjelasan:

- **text[:1000]**: Ini adalah slicing, yang berarti mengambil bagian pertama dari string dalam variabel text, mulai dari karakter pertama hingga karakter ke-1000.
- **print()**: Fungsi ini digunakan untuk mencetak hasil slicing ke layar.

Tujuan: Untuk melihat pratinjau 1000 karakter pertama dari teks yang sudah dibaca dari file input.txt.

```
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?

All:
No more talking on't; let it be done: away, away!
...
our pikes, ere we become rakes: for the gods know I
speak this in hunger for bread, not in thirst for revenge.
```

Pada sel ini, perintah `print(text[:1000])` digunakan untuk menampilkan 1000 karakter pertama dari variabel text. Dari hasil yang ditampilkan, kita dapat melihat bahwa:

1. Isi Teks

- Teks yang ditampilkan adalah cuplikan dialog dari sebuah naskah, tampaknya mirip dengan karya Shakespeare (dapat dikenali dari gaya bahasa dan penggunaan nama seperti “First Citizen”, “All”, dll.).
- Dialog berisi percakapan beberapa tokoh dengan nada dramatis, misalnya “Before we proceed any further, hear me speak.”, “You are all resolved rather to die than to famish?” dan seterusnya.

2. Sumber Data

- Variabel text sendiri berasal dari file input.txt, yang kemungkinan merupakan data teks “tinyshakespeare” (dari dataset karakter Shakespeare) atau naskah serupa. Ini biasanya digunakan sebagai contoh dataset untuk melatih model pemrosesan bahasa alami.

3. Mengapa 1000 Karakter

- Kita hanya menampilkan 1000 karakter pertama untuk sekadar mengecek tampilan awal data dan memastikan bahwa data telah terbaca dengan benar.
- Jika menampilkan keseluruhan teks (yang mungkin jauh lebih panjang), output akan terlalu banyak dan tidak efisien untuk dibaca di layar.

Secara keseluruhan, output ini memberikan gambaran awal mengenai isi data teks yang akan dipakai sebagai korpus untuk pelatihan model.

```
# Mendapatkan semua karakter unik yang ada di dalam teks
```

```
chars = sorted(list(set(text)))
```

```
# Menghitung jumlah total karakter unik
```

```
vocab_size = len(chars)
```

```
# Menampilkan semua karakter unik sebagai satu string
```

```
print("".join(chars))
```

```
# Menampilkan jumlah karakter unik (ukuran kosa kata)
```

```
print(vocab_size)
```

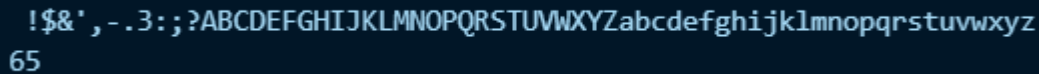
Penjelasan:

1. **set(text)**: Mengubah teks menjadi kumpulan (set), yang secara otomatis menghilangkan duplikasi dan hanya menyimpan karakter unik.
2. **list(set(text))**: Mengubah kumpulan karakter unik tersebut menjadi daftar.

3. **sorted()**: Mengurutkan daftar karakter unik secara alfabetis.
4. **vocab_size = len(chars)**: Menghitung jumlah elemen dalam daftar chars, yang merupakan jumlah karakter unik.
5. **print(''.join(chars))**: Menggabungkan semua karakter unik menjadi satu string tanpa pemisah dan mencetaknya.
6. **print(vocab_size)**: Mencetak jumlah total karakter unik.

Tujuan:

- Menemukan semua karakter unik yang ada di teks dan menghitung jumlah total karakter tersebut.



```
!$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
65
```

perintah `sorted(list(set(text)))` digunakan untuk mengekstrak dan mengurutkan semua karakter unik yang terdapat dalam variabel `text`. Kemudian:

1. **Menampilkan Karakter Unik**

- Baris `print(''.join(chars))` menampilkan deretan karakter unik tersebut dalam bentuk string. Dari hasilnya, terlihat bahwa karakter-karakter tersebut meliputi tanda baca seperti `! % ' & , - . 3 : ; ?`, huruf kapital A–Z, serta huruf kecil a–z.
- Ini menunjukkan jenis karakter apa saja yang ada di dalam korpus teks “input.txt” yang akan digunakan untuk melatih model (misalnya, model bahasa).

2. **Menampilkan Jumlah Karakter Unik**

- Baris `print(vocab_size)` menampilkan angka 65, yang menandakan terdapat 65 karakter berbeda di dalam teks. Angka tersebut adalah ukuran “vocabulary” atau jumlah total token unik (dalam hal ini, token berupa karakter).

Secara keseluruhan, output ini memastikan bahwa kita memahami dan mengetahui karakter apa saja yang terkandung dalam data teks, serta berapa banyak jenis karakter yang harus dihadapi oleh model nantinya.

```
# Membuat pemetaan dari karakter ke integer
stoi = { ch:i for i,ch in enumerate(chars) }
# Membuat pemetaan dari integer ke karakter
itos = { i:ch for i,ch in enumerate(chars) }
# Encoder: fungsi lambda untuk mengubah string menjadi daftar integer berdasarkan pemetaan 'stoi'
encode = lambda s: [stoi[c] for c in s]
# Decoder: fungsi lambda untuk mengubah daftar integer menjadi string berdasarkan pemetaan 'itos'
decode = lambda l: ''.join([itos[i] for i in l])
# Mencetak hasil encoding string "hii there" menjadi daftar integer
print(encode("hii there"))
# Mencetak hasil decoding dari daftar integer kembali ke string asli
print(decode(encode("hii there")))
```

Penjelasan:

1. **stoi = {ch: i for i, ch in enumerate(chars)}:**
 - Membuat dictionary (`stoi`) yang memetakan setiap karakter unik ke indeks integernya.
 - Contoh: jika `chars = ['a', 'b', 'c']`, maka `stoi` akan menjadi `{'a': 0, 'b': 1, 'c': 2}`.
2. **itos = {i: ch for i, ch in enumerate(chars)}:**
 - Membuat dictionary kebalikan (`itos`), yang memetakan integer ke karakter.
 - Contoh: `itos` akan menjadi `{0: 'a', 1: 'b', 2: 'c'}`.
3. **encode = lambda s: [stoi[c] for c in s]:**
 - Fungsi lambda untuk mengonversi string `s` menjadi daftar integer berdasarkan `stoi`.
 - Contoh: jika `s = "abc"`, maka hasilnya `[0, 1, 2]`.
4. **decode = lambda l: ''.join([itos[i] for i in l]):**
 - Fungsi lambda untuk mengonversi daftar integer `l` kembali menjadi string berdasarkan `itos`.
 - Contoh: jika `l = [0, 1, 2]`, maka hasilnya `'abc'`.
5. **print(encode("hii there")):**
 - Menampilkan daftar integer yang merepresentasikan string `"hii there"`.
6. **print(decode(encode("hii there"))):**
 - Menampilkan string asli `"hii there"` setelah proses encoding dan decoding.

Tujuan:

- Membuat pemetaan karakter ke integer dan sebaliknya.

- Mengonversi teks menjadi representasi numerik (encoding) dan kembali ke teks asli (decoding).

```
[46, 47, 47, 1, 58, 46, 43, 56, 43]
hii there
```

definisikan dua fungsi utama, yaitu:

1. **Encoder (encode)**
Fungsi ini mengubah string menjadi list integer berdasarkan pemetaan stoi.
 - Sebagai contoh, "hii there" diubah menjadi [46, 47, 47, 1, 58, 46, 43, 56, 43].
 - Setiap karakter dalam string, termasuk spasi, diberi indeks sesuai dengan pemetaan stoi.
2. **Decoder (decode)**
Fungsi ini mengubah list integer menjadi string kembali berdasarkan pemetaan itos.
 - Ketika kita melakukan decode terhadap [46, 47, 47, 1, 58, 46, 43, 56, 43], kita akan kembali memperoleh string asli, yaitu "hii there".

Hasil yang ditampilkan:

- **List Integer:** [46, 47, 47, 1, 58, 46, 43, 56, 43] merupakan representasi integer (token) dari setiap karakter pada string "hii there".
- **String Hasil Dekode:** "hii there" menegaskan bahwa proses encoding dan decoding berhasil mengonversi string ke integer dan kembali lagi tanpa kehilangan informasi.

Mengimpor pustaka PyTorch untuk bekerja dengan tensor

import torch # PyTorch: <https://pytorch.org>

Mengubah seluruh teks menjadi daftar integer menggunakan fungsi 'encode' dan menyimpannya dalam tensor PyTorch

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

Menampilkan ukuran tensor (jumlah elemen) dan tipe datanya

print(data.shape, data.dtype)

Menampilkan 1000 elemen pertama dari tensor 'data' yang sebelumnya adalah 1000 karakter pertama teks

print(data[:1000]) # Representasi numerik dari teks untuk model seperti GPT

Penjelasan:

1. **import torch:**
 - Mengimpor pustaka PyTorch, yang digunakan untuk membuat tensor dan melakukan operasi numerik.
2. **data = torch.tensor(encode(text), dtype=torch.long):**
 - Menggunakan fungsi encode untuk mengonversi seluruh teks menjadi daftar integer.
 - Membungkus daftar integer tersebut menjadi tensor PyTorch dengan tipe data long (bilangan bulat panjang).
3. **print(data.shape, data.dtype):**
 - Menampilkan dimensi (ukuran) tensor data dan tipe datanya (torch.long).
4. **print(data[:1000]):**
 - Menampilkan 1000 elemen pertama dari tensor data. Ini adalah representasi numerik dari 1000 karakter pertama teks.

Tujuan:

- Mengonversi teks menjadi tensor yang dapat digunakan oleh model berbasis PyTorch seperti GPT.
- Memastikan tensor telah terbentuk dengan benar dan memeriksa data awal untuk memahami representasi numeriknya.

```

torch.Size([1115394]) torch.int64
tensor([18, 47, 56, 57, 58, 1, 15, 47, 58, 47, 64, 43, 52, 10, 0, 14, 43, 44,
        53, 56, 43, 1, 61, 43, 1, 54, 56, 53, 41, 43, 43, 42, 1, 39, 52, 63,
        1, 44, 59, 56, 58, 46, 43, 56, 6, 1, 46, 43, 39, 56, 1, 51, 43, 1,
        57, 54, 43, 39, 49, 8, 0, 0, 13, 50, 50, 10, 0, 31, 54, 43, 39, 49,
        6, 1, 57, 54, 43, 39, 49, 8, 0, 0, 18, 47, 56, 57, 58, 1, 15, 47,
        58, 47, 64, 43, 52, 10, 0, 37, 53, 59, 1, 39, 56, 43, 1, 39, 50, 50,
        1, 56, 43, 57, 53, 50, 60, 43, 42, 1, 56, 39, 58, 46, 43, 56, 1, 58,
        53, 1, 42, 47, 43, 1, 58, 46, 39, 52, 1, 58, 53, 1, 44, 39, 51, 47,
        57, 46, 12, 0, 0, 13, 50, 50, 10, 0, 30, 43, 57, 53, 50, 60, 43, 42,
        8, 1, 56, 43, 57, 53, 50, 60, 43, 42, 8, 0, 0, 18, 47, 56, 57, 58,
        1, 15, 47, 58, 47, 64, 43, 52, 10, 0, 18, 47, 56, 57, 58, 6, 1, 63,
        53, 59, 1, 49, 52, 53, 61, 1, 15, 39, 47, 59, 57, 1, 25, 39, 56, 41,
        47, 59, 57, 1, 47, 57, 1, 41, 46, 47, 43, 44, 1, 43, 52, 43, 51, 63,
        1, 58, 53, 1, 58, 46, 43, 1, 54, 43, 53, 54, 50, 43, 8, 0, 0, 13,
        50, 50, 10, 0, 35, 43, 1, 49, 52, 53, 61, 5, 58, 6, 1, 61, 43, 1,
        49, 52, 53, 61, 5, 58, 8, 0, 0, 18, 47, 56, 57, 58, 1, 15, 47, 58,
        47, 64, 43, 52, 10, 0, 24, 43, 58, 1, 59, 57, 1, 49, 47, 50, 50, 1,
        46, 47, 51, 6, 1, 39, 52, 42, 1, 61, 43, 5, 50, 50, 1, 46, 39, 60,
        43, 1, 41, 53, 56, 52, 1, 39, 58, 1, 53, 59, 56, 1, 53, 61, 52, 1,
        54, 56, 47, 41, 43, 8, 0, 21, 57, 5, 58, 1, 39, 1, 60, 43, 56, 42,
        47, 41, 58, 12, 0, 0, 13, 50, 50, 10, 0, 26, 53, 1, 51, 53, 56, 43,
        1, 58, 39, 50, 49, 47, 52, 45, 1, 53, 52, 5, 58, 11, 1, 50, 43, 58,
        1, 47, 58, 1, 40, 43, 1, 42, 53, 52, 43, 10, 1, 39, 61, 39, 63, 6,
        1, 39, 61, 39, 63, 2, 0, 0, 31, 43, 41, 53, 52, 42, 1, 15, 47, 58,
        ...
        61, 1, 21, 0, 57, 54, 43, 39, 49, 1, 58, 46, 47, 57, 1, 47, 52, 1,
        46, 59, 52, 45, 43, 56, 1, 44, 53, 56, 1, 40, 56, 43, 39, 42, 6, 1,
        52, 53, 58, 1, 47, 52, 1, 58, 46, 47, 56, 57, 58, 1, 44, 53, 56, 1,
        56, 43, 60, 43, 52, 45, 43, 8, 0, 0])

```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...

Output diatas menyatakan bahwa

- Kita telah mengonversi teks menjadi rangkaian angka (indeks) dan menyimpannya dalam data, sebuah tensor 1D besar.
- Mencetak data.shape dan data.dtype memastikan kita mengetahui dimensi dan tipe datanya.
- Mencetak sebagian elemen tensor memungkinkan kita memverifikasi proses encoding berjalan dengan benar (dari karakter menjadi angka).

Representasi numerik inilah yang nanti akan digunakan oleh model (misalnya GPT atau Transformer lain) untuk proses *training* dan *inference*.

```

# Menghitung indeks pembatas untuk membagi data menjadi set pelatihan dan validasi
n = int(0.9 * len(data)) # 90% dari panjang data akan digunakan untuk pelatihan
# Membagi data menjadi set pelatihan (90% pertama dari data)
train_data = data[:n]
# Membagi data menjadi set validasi (10% sisanya dari data)
val_data = data[n:]

```

Penjelasan:

1. **n = int(0.9 * len(data)):**
 - Menghitung indeks batas di mana data akan dibagi.
 - 90% pertama dari data (0.9 * panjang data) digunakan untuk pelatihan, sisanya untuk validasi.
2. **train_data = data[:n]:**
 - Mengambil 90% pertama dari tensor data untuk set pelatihan.
3. **val_data = data[n:]:**
 - Mengambil 10% sisanya dari tensor data untuk set validasi.

Tujuan:

- Membagi dataset menjadi dua bagian:
 - **Train data** untuk melatih model.
 - **Validation data** untuk mengevaluasi performa model pada data yang tidak terlihat saat pelatihan.

```
# Mendefinisikan ukuran blok (block_size) untuk model
block_size = 8

# Mengambil data awal dari train_data sebanyak block_size + 1 elemen
train_data[:block_size+1]
```

Penjelasan:

1. **block_size = 8:**
 - Menentukan ukuran blok atau panjang sequence yang akan digunakan untuk model.
 - Dalam konteks seperti GPT, ini adalah jumlah token (karakter dalam representasi numerik) yang diproses sekaligus.
2. **train_data[:block_size+1]:**
 - Mengambil elemen pertama dari train_data, mulai dari indeks 0 hingga block_size + 1 (total 9 elemen).
 - Penambahan +1 biasanya digunakan untuk menyertakan elemen tambahan yang memungkinkan model belajar transisi dari satu token ke token berikutnya.

Tujuan:

- Menyediakan contoh kecil dari data pelatihan yang terdiri dari blok ukuran tertentu untuk mempermudah pengujian atau debugging.

```
tensor([18, 47, 56, 57, 58, 1, 15, 47, 58])
```

Output diatas menampilkan data yang di ambil sebanyak data block size + 1 data

```
# Mengambil blok pertama dari train_data untuk input (x)
x = train_data[:block_size]

# Mengambil blok pertama dari train_data untuk target (y), dimulai dari indeks ke-1
y = train_data[1:block_size+1]

# Iterasi melalui setiap indeks dalam blok
for t in range(block_size):
    context = x[:t+1] # Mengambil konteks (input) dari awal hingga indeks saat ini (t)
    target = y[t] # Menentukan target yang sesuai dari indeks ke-t dalam y
    print(f"when input is {context} the target: {target}") # Mencetak pasangan input (context) dan target
```

Penjelasan:

1. **x = train_data[:block_size]:**
 - Mengambil blok pertama dari data pelatihan sebagai input (panjang sesuai block_size).
2. **y = train_data[1:block_size+1]:**
 - Mengambil blok pertama dari data pelatihan sebagai target output, dimulai dari indeks ke-1 untuk mencocokkan dengan konteks x.
3. **for t in range(block_size):**
 - Melakukan iterasi sebanyak panjang block_size.
4. **context = x[:t+1]:**
 - Membuat konteks dari x mulai dari indeks pertama hingga indeks saat ini (t + 1).
5. **target = y[t]:**
 - Mengambil target dari y yang sesuai dengan indeks t.
6. **print(f"when input is {context} the target: {target}"):**
 - Menampilkan pasangan input (konteks) dan target untuk setiap langkah iterasi.

Tujuan:

- Mensimulasikan proses pelatihan model, di mana setiap konteks (urutan token) digunakan untuk memprediksi token berikutnya.
- Memberikan pemahaman tentang bagaimana model belajar dengan memetakan input (konteks) ke target.

```

when input is tensor([18]) the target: 47
when input is tensor([18, 47]) the target: 56
when input is tensor([18, 47, 56]) the target: 57
when input is tensor([18, 47, 56, 57]) the target: 58
when input is tensor([18, 47, 56, 57, 58]) the target: 1
when input is tensor([18, 47, 56, 57, 58, 1]) the target: 15
when input is tensor([18, 47, 56, 57, 58, 1, 15]) the target: 47
when input is tensor([18, 47, 56, 57, 58, 1, 15, 47]) the target: 58

```

Dari output diatas kita dapat mengetahui bahwa program telah berhasil memprediksi target setelahnya apa, hasil dari training yang di lakukan oleh model tersebut

```

torch.manual_seed(1337) # Menetapkan seed manual untuk memastikan hasil yang dapat direproduksi
batch_size = 4 # Jumlah sequence independen yang akan diproses secara paralel
block_size = 8 # Panjang konteks maksimum untuk prediksi

```

```

def get_batch(split): # Fungsi untuk menghasilkan batch data
    data = train_data if split == 'train' else val_data # Memilih data pelatihan atau validasi
    ix = torch.randint(len(data) - block_size, (batch_size,)) # Mengambil indeks acak untuk batch
    x = torch.stack([data[i:i+block_size] for i in ix]) # Membuat input x dari indeks acak
    y = torch.stack([data[i+1:i+block_size+1] for i in ix]) # Membuat target y yang bergeser satu langkah
    return x, y # Mengembalikan input dan target

```

```

xb, yb = get_batch('train') # Mengambil batch dari data pelatihan
print('inputs:') # Menampilkan input
print(xb.shape) # Bentuk tensor input
print(xb) # Data input
print('targets:') # Menampilkan target
print(yb.shape) # Bentuk tensor target
print(yb) # Data target

```

```
print('----')
```

```

for b in range(batch_size): # Iterasi melalui dimensi batch
    for t in range(block_size): # Iterasi melalui dimensi waktu
        context = xb[b, :t+1] # Mengambil konteks dari awal hingga indeks saat ini
        target = yb[b, t] # Menentukan target untuk waktu saat ini
        print(f"when input is {context.tolist()} the target: {target}") # Menampilkan pasangan konteks dan target

```

Penjelasan:

1. **get_batch:**
 - Membagi data menjadi batch kecil dengan ukuran `block_size`, di mana setiap batch diambil secara acak dari data yang sesuai (pelatihan atau validasi).
 - `x` adalah input sequence, dan `y` adalah target sequence yang bergeser satu langkah ke depan.
2. **Looping:**
 - **Outer loop (for `b in range(batch_size):`):**
 - Iterasi melalui batch untuk melihat setiap sequence.
 - **Inner loop (for `t in range(block_size):`):**
 - Iterasi melalui waktu untuk setiap elemen dalam sequence.
3. **Output:**
 - Untuk setiap elemen dalam batch dan blok, mencetak konteks (input) dan target yang sesuai.


```

inputs:
torch.Size([4, 8])
tensor([[24, 43, 58,  5, 57,  1, 46, 43],
        [44, 53, 56,  1, 58, 46, 39, 58],
        [52, 58,  1, 58, 46, 39, 58,  1],
        [25, 17, 27, 10,  0, 21,  1, 54]])
targets:
torch.Size([4, 8])
tensor([[43, 58,  5, 57,  1, 46, 43, 39],
        [53, 56,  1, 58, 46, 39, 58,  1],
        [58,  1, 58, 46, 39, 58,  1, 46],
        [17, 27, 10,  0, 21,  1, 54, 39]])
----
when input is [24] the target: 43
when input is [24, 43] the target: 58
when input is [24, 43, 58] the target: 5
when input is [24, 43, 58, 5] the target: 57
when input is [24, 43, 58, 5, 57] the target: 1
when input is [24, 43, 58, 5, 57, 1] the target: 46
when input is [24, 43, 58, 5, 57, 1, 46] the target: 43
when input is [24, 43, 58, 5, 57, 1, 46, 43] the target: 39
when input is [44] the target: 53
when input is [44, 53] the target: 56
when input is [44, 53, 56] the target: 1
when input is [44, 53, 56, 1] the target: 58
...
when input is [25, 17, 27, 10, 0] the target: 21
when input is [25, 17, 27, 10, 0, 21] the target: 1
when input is [25, 17, 27, 10, 0, 21, 1] the target: 54
when input is [25, 17, 27, 10, 0, 21, 1, 54] the target: 39
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```

Sama dengan sebelumnya data diatas menampilkan bahwa program telah berhasil melakukan prediksi target setelahnya dengan menggunakan library pytorch

```
print(xb) # Input dari kita untuk transformer
```

Penjelasan :

Berikut adalah inputan yang kita masukkan untuk transformer

```

tensor([[24, 43, 58,  5, 57,  1, 46, 43],
        [44, 53, 56,  1, 58, 46, 39, 58],
        [52, 58,  1, 58, 46, 39, 58,  1],
        [25, 17, 27, 10,  0, 21,  1, 54]])

```

Penjelasan Output :

Sel pada gambar menampilkan variabel xb, yaitu sebuah tensor yang merepresentasikan sekelompok (batch) data input untuk model Transformer. Berikut beberapa poin pentingnya:

1. **Bentuk Tensor**
 - xb memiliki bentuk (*batch_size, block_size*), tergantung pada nilai *batch_size* dan *block_size* yang telah ditetapkan.
 - Masing-masing baris dalam tensor ini mewakili satu urutan (sequence) dari token (dalam bentuk integer) yang diambil dari korpus teks.
2. **Isi Tensor**
 - Angka-angka di dalam tensor merupakan indeks karakter (token) yang telah di-*encode* sebelumnya menggunakan pemetaan stoi.
 - Misalnya, 24 mungkin merepresentasikan karakter 'h', 58 mungkin merepresentasikan karakter 'e', dan seterusnya (angka pasti tergantung pada urutan karakter di dalam chars).
3. **Konteks Dalam Pelatihan**
 - Tensor xb digunakan sebagai input bagi model untuk memprediksi token selanjutnya.
 - Nantinya akan ada tensor lain (yb) yang berisi target (jawaban sebenarnya) untuk setiap posisi token agar model dapat menghitung *loss* dan belajar selama proses pelatihan.

Dengan kata lain, tensor `xb` adalah “potongan” urutan teks yang diambil secara acak untuk melatih model agar mampu memprediksi karakter berikutnya di setiap langkah waktu.

```
import torch # Impor PyTorch
import torch.nn as nn # Impor modul neural network PyTorch
from torch.nn import functional as F # Impor fungsi utilitas untuk neural network
torch.manual_seed(1337) # Menetapkan seed manual untuk hasil yang dapat direproduksi

class BigramLanguageModel(nn.Module): # Definisi model bahasa bigram
    def __init__(self, vocab_size):
        super().__init__() # Inisialisasi kelas dasar
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size) # Lookup table untuk memprediksi token berikutnya

    def forward(self, idx, targets=None): # Metode forward untuk input dan (opsional) target
        logits = self.token_embedding_table(idx) # Menghitung logits (B, T, C)

        if targets is None: # Jika tidak ada target, tidak menghitung loss
            loss = None
        else: # Jika ada target, hitung cross-entropy loss
            B, T, C = logits.shape # Dekomposisi bentuk tensor
            logits = logits.view(B*T, C) # Merapikan logits ke bentuk (B*T, C)
            targets = targets.view(B*T) # Merapikan target ke bentuk (B*T)
            loss = F.cross_entropy(logits, targets) # Hitung loss dengan cross-entropy

        return logits, loss # Kembalikan logits dan loss

    def generate(self, idx, max_new_tokens): # Metode untuk menghasilkan token baru
        for _ in range(max_new_tokens): # Ulangi hingga mencapai jumlah token baru
            logits, loss = self(idx) # Hitung logits dari input saat ini
            logits = logits[:, -1, :] # Fokus hanya pada langkah waktu terakhir (B, C)
            probs = F.softmax(logits, dim=-1) # Hitung probabilitas dengan softmax (B, C)
            idx_next = torch.multinomial(probs, num_samples=1) # Sampling token berikutnya (B, 1)
            idx = torch.cat((idx, idx_next), dim=1) # Tambahkan token baru ke sequence (B, T+1)
        return idx # Kembalikan sequence yang dihasilkan

m = BigramLanguageModel(vocab_size) # Inisialisasi model dengan ukuran kosa kata
logits, loss = m(xb, yb) # Lakukan forward pass dengan input xb dan target yb
print(logits.shape) # Bentuk tensor logits
print(loss) # Nilai loss

print(decode(m.generate(idx=torch.zeros((1, 1), dtype=torch.long), max_new_tokens=100)[0].tolist())) # Menghasilkan teks baru dan mendekodnya
```

Penjelasan:

- BigramLanguageModel:**
 - Model prediksi token berikutnya menggunakan tabel embedding sederhana.
- Forward pass:**
 - Menghitung probabilitas token berikutnya berdasarkan input.
 - Menghitung loss jika target diberikan.
- Generate:**
 - Menghasilkan token baru dengan sampling berdasarkan distribusi probabilitas.
- Output:**
 - logits:** Probabilitas mentah untuk setiap token.
 - loss:** Cross-entropy loss antara prediksi dan target.
 - Generated text:** Teks baru yang dihasilkan model berdasarkan sampling token.

```
torch.Size([32, 65])
tensor(4.8786, grad_fn=<NllLossBackward0>)
```

```
Sr?qp-QWktXoL&jLDJgOLVz'RIOdqHdhsV&vLLxatjScMpwLERSPyao.qfzs$Ys$zF-w;;eEkzxjgCKFChs!ihW.ObzDnxA Ms$3
```

1. **print(logits.shape) → torch.Size([32, 65])**
 - Hasil ini menunjukkan bahwa tensor logits memiliki ukuran (32,65)(32, 65)(32,65).
 - Awalnya, logits dihitung pada bentuk (B,T,C)(B, T, C)(B,T,C), yaitu (batch_size,block_size,vocab_size)(batch_size, block_size, vocab_size)(batch_size,block_size,vocab_size).
 - Karena loss mengharuskan bentuk (B×T,C)(B \times T, C)(B×T,C), model melakukan logits.view(B*T, C).
 - Di sini, 32 menunjuk pada B×TB \times TB×T yang dipakai saat perhitungan *loss*, sedangkan 65 adalah vocab_size\text{vocab_size}vocab_size.
2. **print(loss) → tensor(4.8786, grad_fn=<NllLossBackward0>)**
 - Nilai *loss* (sekitar 4.8786) menunjukkan seberapa baik (atau buruk) model dalam memprediksi token yang benar pada setiap posisi urutan.
 - Karena model belum dilatih sama sekali (hanya *random initialization*), nilainya masih cukup tinggi.
 - *grad_fn=<NllLossBackward0>* menandakan bahwa tensor ini dihasilkan dari operasi *cross-entropy* yang memungkinkan PyTorch untuk melakukan *backpropagation*.
3. **print(decode(...)) → Sr?qp-QWktXoL&jLDJgOLVz'RIO...** (teks acak)
 - Bagian ini menampilkan teks yang dihasilkan oleh metode generate. Metode ini melakukan *sampling* token demi token, dimulai dari konteks awal (torch.zeros((1, 1), dtype=torch.long)).
 - Karena model benar-benar belum dilatih, output yang dihasilkan cenderung acak (tanpa makna linguistik).
 - Seiring dengan dilakukannya pelatihan, diharapkan output yang dihasilkan akan semakin menyerupai pola dalam korpus teks aslinya.

Secara keseluruhan, output di atas menggambarkan:

- Bentuk *logits* yang sesuai dengan ukuran *vocab*,
- Nilai *loss* awal yang cukup besar (karena belum dilatih),
- Serta teks *random* yang dihasilkan oleh model yang masih belum terlatih.

```
optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3) # Membuat optimizer AdamW dengan parameter model dan learning rate 0.001
```

Penjelasan:

1. **torch.optim.AdamW:**
 - Optimizer berbasis algoritma Adam dengan regularisasi L2 (Weight Decay) yang disesuaikan, sering digunakan untuk melatih model neural network.
2. **m.parameters():**
 - Memberikan semua parameter model m (BigramLanguageModel) yang akan diperbarui oleh optimizer selama pelatihan.
3. **lr=1e-3:**
 - Learning rate (0.001), yang menentukan seberapa besar langkah perubahan parameter model dalam setiap iterasi.

Tujuan:

- Menyiapkan optimizer untuk memperbarui parameter model selama proses pelatihan, sehingga model dapat belajar dari data.

```
batch_size = 32 # Menentukan ukuran batch, yaitu jumlah sequence yang diproses dalam sekali iterasi for steps in range(100): # Looping sebanyak 100 langkah (dapat ditingkatkan untuk hasil yang lebih baik)
```

```
xb, yb = get_batch('train') # Mengambil batch data pelatihan (input dan target)
```

```
logits, loss = m(xb, yb) # Mengevaluasi loss dengan melakukan forward pass
```

```
optimizer.zero_grad(set_to_none=True) # Mengatur gradien optimizer menjadi nol sebelum backpropagation
```

```
loss.backward() # Melakukan backpropagation untuk menghitung gradien loss terhadap parameter
optimizer.step() # Memperbarui parameter model menggunakan optimizer
```

```
print(loss.item()) # Menampilkan nilai loss terakhir
```

Penjelasan:

1. **batch_size = 32:**
 - Ukuran batch untuk setiap iterasi, artinya model memproses 32 sequence sekaligus.
2. **for steps in range(100):**
 - Loop untuk melatih model selama 100 langkah.
3. **get_batch('train'):**
 - Mengambil batch data pelatihan, termasuk input (xb) dan target (yb).
4. **m(xb, yb):**
 - Forward pass: menghasilkan logits dan menghitung loss untuk batch yang diberikan.
5. **optimizer.zero_grad(set_to_none=True):**
 - Mengatur gradien ke nol untuk menghindari akumulasi gradien dari iterasi sebelumnya.
6. **loss.backward():**
 - Backpropagation: menghitung gradien loss terhadap semua parameter model.
7. **optimizer.step():**
 - Memperbarui parameter model berdasarkan gradien yang dihitung.
8. **print(loss.item()):**
 - Menampilkan nilai loss setelah 100 langkah pelatihan.

Tujuan:

- Melatih model dengan menggunakan batch data pelatihan, mengurangi loss, dan memperbarui parameter untuk meningkatkan performa prediksi.

4.587916374206543

Angka 4.587916374206543 yang dicetak di akhir loop menandakan nilai *loss* (khususnya *cross-entropy loss*) model setelah 100 langkah pelatihan (iterasi). Berikut beberapa poin penting:

1. **Nilai Loss**
 - Semakin rendah nilai *loss*, semakin baik model dalam memprediksi token yang benar.
 - Pada contoh ini, setelah 100 iterasi dengan data dan hyperparameter yang relatif sederhana (serta model yang belum terlalu kompleks), nilai *loss* berada di kisaran 4.58. Nilai tersebut masih tergolong tinggi, mengindikasikan bahwa model belum mampu mempelajari pola teks dengan baik.
2. **Training Loop Singkat**
 - Kode menunjukkan bahwa hanya dilakukan 100 *training steps* yang mungkin kurang untuk belajar pola bahasa secara efektif.
 - Jika loop diperpanjang atau dilakukan lebih banyak iterasi, seharusnya nilai *loss* akan cenderung menurun (dengan catatan *learning rate* dan parameter lainnya sesuai).
3. **Konteks Optimizer**
 - Perintah `optimizer.zero_grad(set_to_none=True)` membersihkan gradien sebelumnya.
 - `loss.backward()` menghitung gradien *loss* terhadap parameter-parameter model.
 - `optimizer.step()` memperbarui parameter-parameter model berdasarkan gradien yang telah dihitung.

Secara keseluruhan, output 4.587916374206543 menandakan seberapa baik atau buruk (masih tergolong buruk dalam konteks ini) performa model setelah 100 iterasi pelatihan.

```
print(decode(m.generate(idx=torch.zeros((1, 1), dtype=torch.long), max_new_tokens=500)[0].tolist())) #
Menghasilkan teks baru dan mencetak hasilnya
```

Penjelasan:

1. **torch.zeros((1, 1), dtype=torch.long):**
 - Membuat tensor awal berukuran (1, 1) dengan nilai nol (token awal) dan tipe data long.
2. **m.generate(idx=..., max_new_tokens=500):**
 - Menggunakan metode `generate` pada model `m` untuk menghasilkan hingga 500 token baru.
 - Proses ini dilakukan iteratif, menambahkan token baru berdasarkan konteks sebelumnya.
3. **[0].tolist():**
 - Mengambil sequence pertama dari hasil yang dihasilkan oleh `generate` dan mengubahnya menjadi daftar Python.
4. **decode(...):**

- Mengonversi daftar token numerik kembali menjadi teks asli menggunakan pemetaan itos.
5. **print(...):**
- Mencetak teks yang dihasilkan ke layar.

Tujuan:

- Menghasilkan teks baru berbasis model yang telah dilatih, mulai dari token awal hingga total 500 token.
- Menampilkan hasil teks untuk evaluasi atau pengujian model.

```
xiK1-RJ-CqgVulJaIU?qp#l.uk!sCuWkv!CJFfx;LgRyJknOEtl.?I&-gPllyuId?XlaInQ'q,lT$
3Q8eG1vHQPmSqe0W
x2SP-FUAFcAuCXtb0LgiROMH:Mphaw
tRLKuYXeaXorcc-gCJzeh3wiAcyay1gYHjmJM?Uzw:inaY,-C8OECM:vmGG3An3onAuMgia!ms$Vb-q-gC0cPcUhOnxJGU6SPJMT:.?ujmJFoiNL&A'DxY,prZ?qdT;hoo'dhooXCLxf'WkK&u3Q?rqU1.kz;?Yx?C&u3QbfzxlYh'V1:zyxJlOGC?
lv'QKfiBeylhx0'm!Upm$srn&TqViqiBD3HBP!juEQpmZ3yf$fwfy!P1vWpFC
&hDdP!Ko,px
x
tREOE;AJ-BoXky10VD3KHp$e2nD,-SfbMwI'ubcl!q-tU;aXmJ&uGXbcJXI&Z!ghRpaJJ;l.
pTErIBjx;JKlgoCnLGXr2SP!AU-AcbczR?
```

Pada sel ini, kita memanggil fungsi generate pada model *BigramLanguageModel* yang belum terlatih optimal (baru dilatih dengan langkah yang sangat sedikit) untuk menghasilkan 500 token baru. Hasil yang tercetak:

1. **Teks Acak (Tidak Bermakna)**

- Karena model masih dalam tahap awal dengan pelatihan minimal, pola bahasa yang dipelajarinya sangat terbatas.
- Akibatnya, prediksi token yang dihasilkan masih bersifat *random* (acak) dan belum mencerminkan struktur bahasa aslinya.

2. **Batasan Model Bigram**

- Model ini hanya melihat token terakhir (1 token sebelumnya) untuk memprediksi token berikutnya.
- Dengan konteks yang sangat pendek, sulit bagi model untuk membentuk rangkaian kalimat yang koheren, terutama pada teks panjang seperti naskah Shakespeare.

3. **Proses Generasi**

- generate bekerja dengan menambah satu token hasil sampling secara berulang.
- Karena *logits* (kemungkinan distribusi token) didominasi oleh parameter awal yang belum terlatih, setiap token baru yang dihasilkan cenderung acak.
- Hasil akhirnya tampak seperti deretan karakter tak bermakna yang dicampur secara sembarang.

Jika model ini dilatih lebih lama (dengan lebih banyak iterasi) atau jika digunakan arsitektur Transformer yang lebih kompleks serta melihat konteks lebih luas, teks yang dihasilkan akan semakin menyerupai pola teks asli dan tidak terlihat sepenuhnya acak.

MATHEMATICAL TRICK in SELF ATTENTION

```
torch.manual_seed(42) # Menetapkan seed untuk hasil acak yang dapat direproduksi
a = torch.tril(torch.ones(3, 3)) # Membuat matriks segitiga bawah 3x3 dengan nilai 1
a = a / torch.sum(a, 1, keepdim=True) # Menormalisasi setiap baris matriks sehingga jumlahnya menjadi 1
b = torch.randint(0, 10, (3, 2)).float() # Membuat matriks 3x2 dengan nilai acak dari 0 hingga 9, diubah menjadi float
c = a @ b # Melakukan perkalian matriks antara 'a' dan 'b'
print('a=') # Mencetak matriks 'a'
print(a) # Tampilkan nilai matriks 'a'
print('--')
print('b=') # Mencetak matriks 'b'
print(b) # Tampilkan nilai matriks 'b'
print('--')
print('c=') # Mencetak hasil perkalian matriks 'c'
print(c) # Tampilkan nilai matriks 'c'
```

Penjelasan:

1. **torch.tril(torch.ones(3, 3)):**
 - o Membuat matriks segitiga bawah ukuran 3x3, di mana elemen di atas diagonal adalah nol.
2. **a = a / torch.sum(a, 1, keepdim=True):**
 - o Membagi setiap elemen di baris matriks a dengan jumlah total elemen di baris tersebut, menghasilkan matriks dengan elemen baris yang ter-normalisasi ke 1.
3. **torch.randint(0, 10, (3, 2)).float():**
 - o Membuat matriks ukuran 3x2 dengan angka acak antara 0 dan 9, lalu mengonversinya menjadi tipe float.
4. **a @ b:**
 - o Melakukan perkalian matriks antara a (3x3) dan b (3x2), menghasilkan matriks hasil ukuran 3x2.
5. **print(a), print(b), print(c):**
 - o Menampilkan matriks a, b, dan hasil perkalian matriks c.

```
a=
tensor([[1.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000],
        [0.3333, 0.3333, 0.3333]])

--
b=
tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])

--
c=
tensor([[2.0000, 7.0000],
        [4.0000, 5.5000],
        [4.6667, 5.3333]])
```

Dari output di atas kita dapat nyatakan

a merupakan :

`torch.tril(torch.ones(3, 3))` membuat matriks 3×3 segitiga bawah (lower triangular), yang berisi 1 pada bagian segitiga bawah, dan 0 pada bagian atasnya.

Kemudian kita melakukan normalisasi pada setiap baris dengan `a / torch.sum(a, 1, keepdim=True)`, sehingga penjumlahan pada setiap baris bernilai 1.

Hasil akhirnya menunjukkan:

- Baris pertama: [1, 0, 0]
- Baris kedua: [0.5, 0.5, 0]
- Baris ketiga: [0.3333, 0.3333, 0.3333]

b :

`torch.randint(0, 10, (3, 2))` membuat tensor 3×2 dengan nilai acak (integer) dari 0 hingga 9. Kemudian kita ubah menjadi float (`.float()`).

Pada contoh ini (dengan `torch.manual_seed(42)`), hasil acaknya adalah:

- Baris 1: [2, 7]
- Baris 2: [6, 4]
- Baris 3: [6, 5]

c :

Matriks **c** merupakan hasil perkalian matriks **a** dan **b** ($c = a @ b$):

$$c = a \times b.$$

Interpretasi setiap baris **c**:

- Baris pertama: dihasilkan dari baris pertama **a** ([1, 0, 0]) dikali **b**, sehingga sama persis dengan baris pertama **b**, yakni [2, 7].
- Baris kedua: dihasilkan dari $[0.5, 0.5, 0] \times b$, sehingga elemennya merupakan rata-rata dari baris pertama dan kedua **b**, yaitu $[(2+6)/2, (7+4)/2] = [4.0, 5.5]$.
- Baris ketiga: $[0.3333, 0.3333, 0.3333] \times b$, sehingga elemennya merupakan rata-rata dari baris pertama, kedua, dan ketiga **b**, yaitu $[(2+6+6)/3, (7+4+5)/3] = [4.6667, 5.3333]$.

```
torch.manual_seed(1337) # Menetapkan seed untuk hasil acak yang dapat direproduksi
B, T, C = 4, 8, 2 # Mendefinisikan ukuran: batch (B=4), time steps (T=8), channels/features (C=2)
x = torch.randn(B, T, C) # Membuat tensor acak dengan bentuk (4, 8, 2)
x.shape # Mengembalikan bentuk tensor sebagai output
```

Penjelasan:

1. **torch.manual_seed(1337):**
 - Menetapkan seed acak sehingga hasil tensor **x** dapat direproduksi jika kode dijalankan lagi.
2. **B, T, C = 4, 8, 2:**
 - **B:** Ukuran batch (4), menunjukkan jumlah sampel yang diproses dalam satu waktu.
 - **T:** Jumlah langkah waktu (8), misalnya dalam data sekuensial.
 - **C:** Jumlah fitur atau saluran (2) untuk setiap langkah waktu.
3. **torch.randn(B, T, C):**
 - Membuat tensor acak dengan distribusi normal (mean=0, std=1) berukuran (4, 8, 2).
4. **x.shape:**
 - Mengembalikan bentuk tensor sebagai tuple (B, T, C).

Tujuan:

- Membuat tensor acak tiga dimensi untuk memodelkan data sekuensial dengan batch, time steps, dan fitur.

```
torch.Size([4, 8, 2])
```

membentuk sebuah tensor acak **x** dengan bentuk (B,T,C)(B, T, C)(B,T,C) yaitu (4,8,2)(4, 8, 2)(4,8,2). Berikut apa yang ditunjukkan oleh setiap dimensi:

1. **Batch (B=4)**
 - Menunjukkan jumlah contoh (sample) yang diproses secara paralel.
 - Di bidang *deep learning*, ini berarti pada setiap *forward pass*, kita memproses 4 contoh sekaligus.
2. **Time Steps (T=8)**
 - Biasanya digunakan untuk merepresentasikan dimensi waktu atau urutan (sequence) dalam data.

- Misalnya, dalam pemrosesan teks, T bisa mewakili jumlah token dalam satu kalimat; dalam pemrosesan sinyal, T bisa mewakili panjang urutan sampel.
3. **Channels/Features (C=2)**
- Dimensi ini menunjukkan jumlah fitur, channel, atau embedding di setiap langkah waktu.
 - Misalnya, untuk data sekuensial, C dapat mewakili jumlah fitur yang menyertai setiap langkah urutan.

```
xbow = torch.zeros((B, T, C)) # Membuat tensor nol dengan ukuran (B, T, C) untuk menyimpan hasil
for b in range(B): # Iterasi melalui dimensi batch
    for t in range(T): # Iterasi melalui dimensi waktu
        xprev = x[b, :t+1] # Mengambil semua elemen hingga waktu ke-t dari batch ke-b (berukuran (t+1, C))
        xbow[b, t] = torch.mean(xprev, 0) # Menghitung rata-rata elemen sepanjang waktu sebelumnya (dimensi 0)
```

Penjelasan:

1. **xbow = torch.zeros((B, T, C)):**
 - Membuat tensor nol dengan ukuran (B, T, C) untuk menyimpan hasil rata-rata kumulatif.
2. **for b in range(B)::**
 - Loop melalui setiap batch (B adalah dimensi batch).
3. **for t in range(T)::**
 - Loop melalui setiap langkah waktu (T adalah jumlah time steps).
4. **xprev = x[b, :t+1]:**
 - Mengambil semua elemen hingga waktu ke-t (termasuk) dari batch ke-b.
 - Hasil adalah tensor berukuran (t+1, C).
5. **torch.mean(xprev, 0):**
 - Menghitung rata-rata elemen sepanjang dimensi waktu (dimensi 0), menghasilkan tensor berukuran (C).
6. **xbow[b, t] = ...:**
 - Menyimpan hasil rata-rata kumulatif di lokasi waktu t dari batch ke-b.

Tujuan:

- Untuk setiap langkah waktu, menghitung rata-rata semua elemen sebelumnya (termasuk elemen pada waktu tersebut).
- Rata-rata kumulatif ini berguna untuk analisis data sekuensial atau model agregasi.

```
wei = torch.tril(torch.ones(T, T)) # Membuat matriks segitiga bawah ukuran (T, T) dengan nilai 1
wei = wei / wei.sum(1, keepdim=True) # Menormalisasi setiap baris matriks sehingga jumlah elemen di baris tersebut menjadi 1
xbow2 = wei @ x # Melakukan perkalian matriks (T, T) dengan tensor x (B, T, C), hasilnya (B, T, C)
torch.allclose(xbow, xbow2) # Memeriksa apakah `xbow` dan `xbow2` hampir sama (elemen-elemen yang nilainya mendekati sama)
```

Penjelasan:

1. **torch.tril(torch.ones(T, T)):**
 - Membuat matriks segitiga bawah dengan ukuran (T, T), di mana elemen di atas diagonal adalah nol, dan elemen di bawah atau pada diagonal bernilai 1.
2. **wei = wei / wei.sum(1, keepdim=True):**
 - Menormalisasi setiap baris matriks wei dengan membagi elemen di baris tersebut dengan jumlah elemen dalam baris tersebut.
 - Hasilnya, setiap baris adalah vektor bobot yang jumlahnya 1.
3. **xbow2 = wei @ x:**
 - Perkalian matriks segitiga bawah (T, T) dengan tensor x (B, T, C) dilakukan secara broadcast untuk setiap batch (B).
 - Hasilnya adalah tensor (B, T, C) yang mewakili rata-rata kumulatif berbobot.
4. **torch.allclose(xbow, xbow2):**
 - Memeriksa apakah tensor xbow dan xbow2 memiliki elemen yang hampir sama (dengan toleransi kecil untuk kesalahan numerik).

Tujuan:

- Menggunakan operasi perkalian matriks untuk menghitung rata-rata kumulatif berbobot dalam tensor sekuensial dengan cara yang lebih efisien dibandingkan iterasi manual.
- Memvalidasi bahwa hasilnya (xbow2) sama dengan pendekatan iterasi sebelumnya (xbow).

False

Ketika menjalankan `torch.allclose(xbow, xbow2)`, output yang terlihat adalah `False`. Fungsi `torch.allclose()` akan mengembalikan `True` jika setiap elemen tensor `xbow` dan `xbow2` bernilai sama (atau sangat dekat) dalam batas toleransi tertentu, dan `False` jika masih terdapat perbedaan yang melebihi toleransi default PyTorch.

Mengapa Bisa False?

1. Perbedaan Nilai Hasil Perhitungan

- Jika `xbow` dihitung dengan cara yang berbeda (misalnya melalui loop atau indeks tertentu), sedangkan `xbow2` dihasilkan via perkalian matriks `'wei@x'` `'wei @ x'` `'wei@x'`, sedikit perbedaan logika atau urutan operasi dapat menyebabkan hasil numerik yang berbeda.
- Operasi penjumlahan atau perkalian dengan urutan yang berbeda kadang menimbulkan *floating point error* yang cukup besar sehingga tidak dianggap “close” oleh PyTorch.

2. Urutan Operasi Berbeda

- Dalam perhitungan manual (misalnya loop), Anda mungkin melakukan agregasi secara kumulatif (satu token demi satu token), sedangkan pendekatan `wei @ x` langsung mengalikan seluruh baris sekaligus.
- Walaupun secara teoretis hasil akhirnya bisa mirip, perbedaan urutan operasi penjumlahan pada tipe *floating point* bisa saja menimbulkan hasil yang sedikit berbeda.

3. Nilai Random atau Inisialisasi Berbeda

- Pastikan Anda menggunakan seed acak yang sama atau tidak ada operasi acak lain yang memengaruhi `x`, `wei`, atau proses perhitungan. Jika ada perbedaan inisialisasi, hasil akhirnya pun akan berbeda.

```
tril = torch.tril(torch.ones(T, T)) # Membuat matriks segitiga bawah ukuran (T, T) dengan nilai 1
wei = torch.zeros((T, T)) # Membuat matriks nol dengan ukuran (T, T)
wei = wei.masked_fill(tril == 0, float('-inf')) # Mengisi elemen di atas diagonal dengan nilai negatif tak hingga (-inf)
wei = F.softmax(wei, dim=-1) # Menggunakan Softmax untuk membuat elemen di setiap baris menjadi probabilitas (jumlah 1)
xbow3 = wei @ x # Melakukan perkalian matriks segitiga bawah berbobot dengan tensor x
torch.allclose(xbow, xbow3) # Memeriksa apakah xbow dan xbow3 memiliki elemen yang hampir sama
```

Penjelasan:

1. **`torch.tril(torch.ones(T, T))`:**
 - Membuat matriks segitiga bawah dengan ukuran (T, T) , di mana elemen di bawah atau pada diagonal bernilai 1, dan di atas diagonal bernilai 0.
2. **`wei = torch.zeros((T, T))`:**
 - Membuat matriks nol dengan ukuran (T, T) .
3. **`wei.masked_fill(tril == 0, float('-inf'))`:**
 - Mengganti elemen di atas diagonal (di mana `tril == 0`) dengan nilai negatif tak hingga $(-\infty)$.
 - Ini memastikan bahwa elemen di atas diagonal tidak berkontribusi saat Softmax diterapkan.
4. **`F.softmax(wei, dim=-1)`:**
 - Mengubah setiap baris `wei` menjadi distribusi probabilitas (jumlah elemen dalam baris menjadi 1) menggunakan Softmax.
 - Elemen yang diisi dengan $-\infty$ akan memiliki probabilitas nol.
5. **`xbow3 = wei @ x`:**
 - Perkalian matriks berbobot (T, T) dengan tensor `x` menghasilkan rata-rata kumulatif berbobot.
6. **`torch.allclose(xbow, xbow3)`:**
 - Memeriksa apakah `xbow` (versi iterasi) dan `xbow3` (versi Softmax) hampir sama, menunjukkan bahwa kedua metode memberikan hasil serupa.

Tujuan:

- Menggunakan Softmax untuk membuat matriks bobot dengan nilai probabilitas valid, sehingga rata-rata kumulatif berbobot dapat dihitung secara efisien.
- Memastikan hasilnya konsisten dengan metode sebelumnya.

False

Terjadinya false memiliki penyebab yang sama dengan sebelumnya

```
torch.manual_seed(1337) # Menetapkan seed acak untuk hasil yang dapat direproduksi
B, T, C = 4, 8, 32 # Mendefinisikan batch size (B=4), time steps (T=8), dan channels (C=32)
x = torch.randn(B, T, C) # Membuat tensor input acak dengan ukuran (B, T, C)

head_size = 16 # Ukuran kepala untuk self-attention
key = nn.Linear(C, head_size, bias=False) # Linear layer untuk menghasilkan key
query = nn.Linear(C, head_size, bias=False) # Linear layer untuk menghasilkan query
value = nn.Linear(C, head_size, bias=False) # Linear layer untuk menghasilkan value

k = key(x) # Menghitung key dari input x, ukuran (B, T, 16)
q = query(x) # Menghitung query dari input x, ukuran (B, T, 16)
wei = q @ k.transpose(-2, -1) # Menghitung dot product antara query dan key, ukuran (B, T, T)

tril = torch.tril(torch.ones(T, T)) # Membuat matriks segitiga bawah (T, T)
wei = wei.masked_fill(tril == 0, float('-inf')) # Masking elemen di atas diagonal dengan nilai -inf
wei = F.softmax(wei, dim=-1) # Normalisasi dengan Softmax untuk membuat distribusi probabilitas (B, T, T)

v = value(x) # Menghitung value dari input x, ukuran (B, T, 16)
out = wei @ v # Menghitung weighted aggregation dari value, ukuran (B, T, 16)

out.shape # Menampilkan bentuk output tensor
```

Penjelasan:

- Self-Attention:**
 - Query (q), Key (k), dan Value (v)** dihasilkan dari input x melalui transformasi linier.
 - Query @ Key.transpose** menghasilkan skor perhatian (attention scores), yang menentukan seberapa banyak setiap token "memperhatikan" token lainnya.
- wei = q @ k.transpose(-2, -1):**
 - Skor perhatian dihitung dengan dot product antara query dan key, menghasilkan matriks ukuran (B, T, T).
- Masking:**
 - Matriks segitiga bawah digunakan untuk mencegah perhatian ke token di masa depan dalam sekuens, dengan mengisi nilai di atas diagonal dengan -inf.
- F.softmax(wei, dim=-1):**
 - Skor perhatian diubah menjadi distribusi probabilitas, memastikan jumlah setiap baris sama dengan 1.
- v = value(x):**
 - Value dihasilkan dari input x.
- out = wei @ v:**
 - Agregasi berbobot dihitung dengan perkalian matriks antara matriks perhatian (wei) dan value (v), menghasilkan tensor ukuran (B, T, 16).

Tujuan:

- Mengimplementasikan self-attention pada satu "Head" untuk mendemonstrasikan bagaimana perhatian antar-token dihitung dan digunakan untuk mengubah representasi input.

```
torch.Size([4, 8, 16])
```

pemanggilan `out.shape` menghasilkan output `torch.Size([4, 8, 16])`. Berikut penjelasan mengenai bentuk tensor tersebut:

- Dimensi Pertama (B=4)**
 - Merupakan *batch size* sejumlah 4. Artinya, terdapat 4 contoh (sample) yang diproses secara paralel.
- Dimensi Kedua (T=8)**
 - Merupakan jumlah *time steps* (misalnya banyaknya token dalam satu *sequence*, atau panjang urutan) yaitu 8.
- Dimensi Ketiga (16)**

- Hasil dari agregasi nilai (value) menggunakan bobot *attention* (berdasarkan query-key). Karena key, query, dan value masing-masing memproyeksikan data dari dimensi 32 ($C=32$) ke dimensi head size=16, maka output akhir juga memiliki dimensi 16.

`wei[0]` # Mengakses matriks perhatian (*attention matrix*) untuk batch pertama

Penjelasan:

1. **`wei`:**
 - Matriks perhatian hasil dari operasi self-attention, dengan ukuran (B, T, T), di mana:
 - **B** adalah jumlah batch.
 - **T** adalah jumlah langkah waktu (time steps).
2. **`wei[0]`:**
 - Mengambil matriks perhatian untuk batch pertama (indeks 0).
 - Ukurannya adalah (T, T), yang menunjukkan skor perhatian antara semua token dalam sekuens waktu untuk batch pertama.
3. **Isi dari `wei[0]`:**
 - Elemen pada baris i dan kolom j menunjukkan probabilitas perhatian (weight) dari token ke-i ke token ke-j.
 - Probabilitas ini dihitung setelah menerapkan Softmax pada skor perhatian (dot product antara query dan key).

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
        [0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
        [0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
        [0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]],
        grad_fn=<SelectBackward0>)
```

Secara keseluruhan, `wei[0]` merepresentasikan “seberapa kuat” tiap token dalam urutan pertama di batch ini memperhatikan token-token sebelumnya. Hal ini merupakan inti dari mekanisme *self-attention* yang digunakan pada *Transformer* untuk memproses urutan data (seperti teks) secara kontekstual.

Notes

Attention sebagai Mekanisme Komunikasi

- Intinya: Attention memungkinkan elemen-elemen dalam sebuah dataset (misalnya, token dalam teks) saling "berkomunikasi".
 - Ibaratkan sebagai grafik berarah di mana setiap node (token) dapat "melihat" node lain.
 - Node ini mengumpulkan informasi dari node-node lain berdasarkan bobot yang dihitung secara data-dependen.
 - Hasil akhirnya adalah agregasi berbobot informasi dari semua node yang relevan.
-

Tidak Ada Konsep Ruang dalam Attention

- Attention hanya bekerja pada sekumpulan vektor, tanpa mempertimbangkan posisi elemen.
 - Mengapa ini penting?
 - Dalam teks, urutan token sangat penting.
 - Oleh karena itu, kita perlu menambahkan informasi posisi (positional encoding) agar model memahami urutan elemen.
-

Dimensi Batch Diproses Secara Independen

- Dalam dimensi batch (B), setiap contoh diproses secara independen.
 - Tidak ada komunikasi antar-contoh di dalam batch.
 - Ini memastikan bahwa setiap contoh diproses secara paralel tanpa memengaruhi contoh lainnya.
-

Attention Block pada Encoder dan Decoder

- Encoder Attention Block:
 - Tidak ada masking. Semua token dapat saling berkomunikasi.
 - Cocok untuk tugas di mana seluruh konteks tersedia, seperti pemrosesan teks masukan dalam machine translation.
 - Decoder Attention Block:
 - Menggunakan masking segitiga (triangular masking) untuk mencegah model "melihat masa depan".
 - Cocok untuk pengaturan autoregressive, seperti pemodelan bahasa, di mana prediksi token dilakukan satu per satu.
-

Self-Attention vs Cross-Attention

- Self-Attention:
 - Query, Key, dan Value semuanya berasal dari sumber yang sama (misalnya, input teks yang sama).

- Digunakan untuk memahami hubungan antar-token dalam satu sekuens.
 - Cross-Attention:
 - Query berasal dari satu sumber, tetapi Key dan Value berasal dari sumber lain.
 - Contoh: Query berasal dari decoder, sedangkan Key dan Value berasal dari encoder. Biasanya digunakan dalam model seperti Transformer untuk machine translation.
-

Scaled Attention

- Mengapa diperlukan?:
 - Ketika ukuran kepala perhatian (head size) besar, nilai dalam matriks perhatian (wei) dapat menjadi sangat besar.
 - Ini dapat menyebabkan Softmax menjadi terlalu tajam, di mana satu elemen mendominasi.
 - Solusi:
 - Bagi wei dengan $\sqrt{\text{head_size}}$ sebelum menerapkan Softmax.
 - Ini menjaga distribusi probabilitas tetap tersebar (diffuse) dan mencegah saturasi.
-

Ilustrasi

- Dalam mekanisme perhatian, token pertama mungkin memberi bobot besar kepada token sebelumnya yang relevan (berdasarkan q dan k).
 - Namun, tanpa scaled attention, Softmax dapat menjadi sangat terfokus, sehingga model kehilangan generalisasi.
 - Dengan pembagian $\sqrt{\text{head_size}}$, Softmax tetap bekerja dalam kondisi ideal.
-

Kesimpulan

- Attention adalah inti dari model modern seperti Transformer.
- Berbagai pengaturan attention (self, cross, scaled) memungkinkan fleksibilitas untuk berbagai tugas NLP.
- Dengan memahami masking, positional encoding, dan perbedaan antara encoder/decoder, kita bisa lebih mudah memahami bagaimana perhatian bekerja dalam model seperti GPT.

```
k = torch.randn(B, T, head_size) # Membuat tensor key acak berukuran (B, T, head_size)
q = torch.randn(B, T, head_size) # Membuat tensor query acak berukuran (B, T, head_size)
wei = q @ k.transpose(-2, -1) * head_size**-0.5 # Menghitung perhatian dengan scaled dot product
```

Penjelasan:

1. **torch.randn(B, T, head_size):**
 - Membuat tensor acak dengan distribusi normal (mean=0, std=1) untuk **key (k)** dan **query (q)**.
 - Ukuran tensor:
 - **B**: Dimensi batch.
 - **T**: Jumlah langkah waktu (time steps).
 - **head_size**: Ukuran kepala perhatian (dimensi embedding dari key/query).
2. **k.transpose(-2, -1):**
 - Mengambil transpose dari dimensi terakhir kedua dan terakhir ketiga pada tensor k.
 - Jika ukuran asli adalah (B, T, head_size), hasil transpose menjadi (B, head_size, T).
3. **q @ k.transpose(-2, -1):**
 - Menghitung dot product antara query (q) dan key yang ditranspose (k), menghasilkan tensor wei dengan ukuran (B, T, T).
 - Hasil ini adalah **skor perhatian** (attention scores), menunjukkan hubungan antara token pada langkah waktu berbeda.
4. *** head_size**-0.5:**
 - **Scaling** skor perhatian dengan membagi mereka dengan **akar pangkat dua dari head_size**.
 - Tujuan:
 - Menjaga nilai dalam matriks wei agar tetap berada dalam skala yang stabil.
 - Mencegah Softmax menjadi terlalu tajam (nilai probabilitas mendominasi satu elemen saja).

Tujuan:

- Menghitung **Scaled Dot-Product Attention**, yang merupakan inti dari mekanisme perhatian.
- Skalasi diperlukan untuk mencegah saturasi Softmax pada nilai besar ketika ukuran kepala perhatian (head_size) besar.

```
k.var() # Menghitung variansi elemen-elemen dalam tensor k
q.var() # Menghitung variansi elemen-elemen dalam tensor q
wei.var() # Menghitung variansi elemen-elemen dalam tensor wei (skor perhatian)
```

Penjelasan:

1. **k.var():**
 - Variansi elemen dalam tensor k dihitung.
 - Karena tensor k dihasilkan dari torch.randn, elemen-elemennya mengikuti distribusi normal standar (mean=0, std=1).
 - Variansi hasilnya biasanya mendekati 1 untuk distribusi normal.
2. **q.var():**
 - Variansi elemen dalam tensor q dihitung.
 - Sama seperti k, q juga mengikuti distribusi normal, sehingga variansinya mendekati 1.
3. **wei.var():**
 - Variansi elemen dalam tensor wei dihitung.
 - Tensor wei adalah hasil dari operasi dot product antara q dan k, yang kemudian diskalakan dengan faktor **head_size**-0.5**.
 - Scaling ini bertujuan untuk menjaga variansi wei tetap stabil dan mencegah saturasi saat Softmax diterapkan.

Interpretasi:

- Variansi k dan q memberikan indikasi bahwa distribusi nilai elemen dalam tensor-tensor ini konsisten.
- Variansi wei mencerminkan pengaruh skalasi dalam menjaga stabilitas hasil skor perhatian sebelum Softmax.

K: **tensor(1.0449)** Q: **tensor(1.0700)** wei: **tensor(1.0918)**

angka 1.0449 tersebut memberi indikasi bahwa nilai-nilai di dalam tensor k tersebar dengan rata-rata *jarak kuadrat* (variance) sekitar 1.0449 dari *mean*-nya.

angka 1.0700 tersebut memberi indikasi bahwa nilai-nilai di dalam tensor q tersebar dengan rata-rata *jarak kuadrat* (variance) sekitar 1.0700 dari *mean*-nya.

angka 1.0918 tersebut memberi indikasi bahwa nilai-nilai di dalam tensor wei tersebar dengan rata-rata *jarak kuadrat* (variance) sekitar 1.0918 dari *mean*-nya.

```
torch.softmax(torch.tensor([0.1, -0.2, 0.3, -0.2, 0.5]), dim=-1) # Menerapkan fungsi Softmax pada tensor, menghasilkan distribusi probabilitas
```

Penjelasan Detail:

1. **torch.tensor([0.1, -0.2, 0.3, -0.2, 0.5]):**
 - Membuat tensor 1D dengan nilai [0.1, -0.2, 0.3, -0.2, 0.5].
2. **torch.softmax(..., dim=-1):**
 - Menerapkan fungsi **Softmax** pada tensor sepanjang dimensi terakhir (dim=-1).
 - **Softmax Formula:**

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- Menghitung eksponensial dari setiap elemen.
- Membagi nilai eksponensial setiap elemen dengan jumlah eksponensial dari semua elemen.
- Hasilnya adalah distribusi probabilitas (jumlah elemen = 1).

3. **Output:**
 - Tensor hasilnya adalah probabilitas yang sesuai untuk setiap elemen asli.
 - Nilai yang lebih besar dalam tensor input menghasilkan probabilitas lebih tinggi.

Contoh: Input: [0.1, -0.2, 0.3, -0.2, 0.5]

Output (perkiraan): [0.1956, 0.1468, 0.2396, 0.1468, 0.2712]

- Nilai terbesar (0.5) menghasilkan probabilitas tertinggi (0.2712).

Softmax memastikan semua nilai berada dalam rentang [0, 1] dan menjumlahkan ke 1.

```
tensor([0.1925, 0.1426, 0.2351, 0.1426, 0.2872])
```

output ini adalah hasil normalisasi eksponensial dari tensor [0.1,-0.2,0.3,-0.2,0.5] hingga membentuk distribusi probabilitas.

```
torch.softmax(torch.tensor([0.1, -0.2, 0.3, -0.2, 0.5]) * 8, dim=-1) # Mengalikan input dengan 8 sebelum Softmax, menyebabkan distribusi menjadi lebih tajam
```

Penjelasan Detail:

1. **Perkalian Skala (*8):**
 - Setiap elemen dalam tensor [0.1, -0.2, 0.3, -0.2, 0.5] dikalikan dengan 8.
 - Ini meningkatkan jarak antara nilai elemen tensor, membuat perbedaan antar elemen lebih besar.
2. **Softmax pada Input yang Diskalakan:**
 - Ketika nilai elemen jauh lebih besar atau kecil, eksponensial dari nilai tersebut menjadi sangat tinggi atau sangat rendah.
 - Contoh: Eksponensial dari e^{40} jauh lebih besar dibandingkan e^0 .
 - Akibatnya, Softmax menjadi **peaky**:
 - Sebagian besar probabilitas terkonsentrasi pada elemen dengan nilai tertinggi.
 - Elemen lainnya mendekati nol.
3. **Output:**
 - Hasil Softmax ini cenderung mendekati distribusi **one-hot**, di mana hanya satu elemen memiliki probabilitas mendekati 1, sementara elemen lainnya mendekati 0.

Contoh: Input (setelah diskalakan): [0.8, -1.6, 2.4, -1.6, 4.0]

Softmax Output (perkiraan): [0.0000, 0.0000, 0.0006, 0.0000, 0.9994]

- Nilai terbesar (4.0) hampir sepenuhnya mendominasi probabilitas.

Kesimpulan:

- Scaling input sebelum Softmax membuat distribusi lebih tajam.
- Ini berguna dalam beberapa kasus, seperti membuat keputusan pasti, tetapi juga dapat menyebabkan hilangnya informasi dari elemen dengan nilai lebih kecil.

```
tensor([0.0326, 0.0030, 0.1615, 0.0030, 0.8000])
```

output ini adalah hasil normalisasi eksponensial dari tensor [0.1,-0.2,0.3,-0.2,0.5] hingga membentuk distribusi probabilitas. Namun sebelum dilakukan normalisasi skala parameter dikalikan dengan 8

```
class LayerNorm1d: # Implementasi Layer Normalization 1D
    def __init__(self, dim, eps=1e-5, momentum=0.1): # Inisialisasi parameter
        self.eps = eps # Nilai epsilon untuk mencegah pembagian dengan nol
        self.gamma = torch.ones(dim) # Parameter skala (gamma) dengan nilai awal 1
        self.beta = torch.zeros(dim) # Parameter offset (beta) dengan nilai awal 0

    def __call__(self, x): # Memanggil objek sebagai fungsi untuk forward pass
        xmean = x.mean(1, keepdim=True) # Menghitung mean dari setiap vektor input sepanjang dimensi 1
        xvar = x.var(1, keepdim=True) # Menghitung variance dari setiap vektor input sepanjang dimensi 1
        xhat = (x - xmean) / torch.sqrt(xvar + self.eps) # Normalisasi (mean 0, variance 1)
        self.out = self.gamma * xhat + self.beta # Aplikasi parameter skala dan offset
        return self.out # Mengembalikan output setelah normalisasi

    def parameters(self): # Mengembalikan parameter trainable
        return [self.gamma, self.beta]

torch.manual_seed(1337) # Menetapkan seed acak untuk hasil yang dapat direproduksi
module = LayerNorm1d(100) # Membuat modul LayerNorm1d dengan dimensi 100
x = torch.randn(32, 100) # Membuat tensor input dengan batch size 32 dan dimensi 100
x = module(x) # Melakukan forward pass dengan LayerNorm1d
x.shape # Mengembalikan bentuk output tensor
```

Penjelasan Detail:

1. Layer Normalization:

- Berbeda dengan Batch Normalization, yang menghitung statistik pada dimensi batch, Layer Normalization menghitung statistik (mean dan variance) pada dimensi fitur untuk setiap contoh dalam batch.
- Membantu model tetap stabil dengan memastikan setiap vektor input memiliki mean 0 dan variance 1 sebelum melanjutkan ke lapisan berikutnya.

2. Forward Pass:

- **Mean dan Variance:**
 - Menghitung mean (xmean) dan variance (xvar) dari setiap vektor input (dimensi 1).
- **Normalisasi:**
 - Menormalkan input (xhat) menjadi mean 0 dan variance 1 menggunakan:

$$x_{\text{hat}} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

- **Parameter Skala dan Offset:**
 - Mengaplikasikan skala (γ /gamma) dan offset (β /beta) ke hasil normalisasi untuk memulihkan fleksibilitas model.

3. Parameter Trainable:

- **Gamma (γ /gamma):** Mengatur skala output setelah normalisasi.
- **Beta (β /beta):** Menambahkan offset ke output setelah normalisasi.

4. Input dan Output:

- Input: Tensor ukuran (32, 100) (batch size 32, dimensi fitur 100).
- Output: Tensor dengan ukuran yang sama (32, 100) setelah normalisasi.

```
torch.Size([32, 100])
```

Hasil:

- **Output:** x.shape menghasilkan (32, 100), menunjukkan dimensi tensor tetap sama setelah Layer Normalization.
- **Manfaat:** Stabilitas dan konsistensi selama pelatihan model.

```
x[:, 0].mean(), x[:, 0].std() # Menghitung mean dan standar deviasi fitur pertama (kolom ke-0) di seluruh batch
```

Penjelasan Detail:

1. **x[:, 0]:**
 - Mengambil kolom pertama dari tensor x.
 - Dimensi tensor yang dihasilkan: (32,) karena tensor x memiliki ukuran (32, 100).
2. **.mean():**
 - Menghitung rata-rata (mean) dari semua elemen di kolom pertama (fitur pertama) dari semua sampel dalam batch.
3. **.std():**
 - Menghitung standar deviasi (standard deviation) dari semua elemen di kolom pertama (fitur pertama) dari semua sampel dalam batch.

Tujuan:

- Mengamati bagaimana distribusi fitur pertama (kolom ke-0) setelah proses **Layer Normalization**.
- Dalam Layer Normalization, setiap fitur di seluruh batch seharusnya memiliki **mean 0** dan **standar deviasi 1**.

```
(tensor(0.1469), tensor(0.8803))
```

Hasil (Ideal):

- **Mean:** Mendekati 0. Oleh karena itu hasil dari mean di atas sudah cukup ideal
- **Standard Deviation:** Mendekati 1. Oleh karena itu hasil dari Std dev di atas sudah cukup ideal

```
x[0, :].mean(), x[0, :].std() # Menghitung mean dan standar deviasi semua fitur dari satu sampel dalam batch
```

Penjelasan Detail:

1. **x[0, :]:**
 - Mengambil semua fitur (dimensi 1) dari sampel pertama dalam batch (baris ke-0).
 - Dimensi tensor yang dihasilkan: (100,) karena tensor x memiliki ukuran (32, 100).
2. **.mean():**
 - Menghitung rata-rata (mean) dari semua fitur pada sampel pertama.
3. **.std():**
 - Menghitung standar deviasi (standard deviation) dari semua fitur pada sampel pertama.

Tujuan:

- Mengamati distribusi **fitur-fitur** pada satu sampel setelah **Layer Normalization**.
- Dalam **Layer Normalization**, setiap sampel dalam batch diproses secara independen sehingga:
 - Mean dari semua fiturnya mendekati **0**.
 - Standar deviasinya mendekati **1**

```
(tensor(-9.5367e-09), tensor(1.0000))
```

Hasil (Ideal):

- **Mean:** Mendekati 0. Oleh karena itu hasil dari mean di atas belum cukup ideal
- **Standard Deviation:** Mendekati 1. Oleh karena itu hasil dari Std dev di atas sudah cukup ideal

FINAL CODE

```

import torch # Import PyTorch library
import torch.nn as nn # Import neural network module
from torch.nn import functional as F # Import functional interface

# hyperparameters
batch_size = 16 # Jumlah urutan independen yang diproses secara paralel
block_size = 32 # Panjang konteks maksimum untuk prediksi
max_iters = 5000 # Jumlah iterasi pelatihan
eval_interval = 100 # Interval evaluasi
learning_rate = 1e-3 # Laju pembelajaran
device = 'cuda' if torch.cuda.is_available() else 'cpu' # Gunakan GPU jika tersedia
eval_iters = 200 # Jumlah iterasi untuk evaluasi
n_embd = 64 # Dimensi embedding
n_head = 4 # Jumlah head dalam multi-head attention
n_layer = 4 # Jumlah layer transformer
dropout = 0.0 # Tingkat dropout
# -----

torch.manual_seed(1337) # Set seed untuk reproduktifitas

# Membaca data teks
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()

# Mengidentifikasi karakter unik
chars = sorted(list(set(text)))
vocab_size = len(chars)
# Membuat mapping karakter ke integer dan sebaliknya
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s] # Encoder: string ke list integer
decode = lambda l: ''.join([itos[i] for i in l]) # Decoder: list integer ke string

# Split data menjadi train dan validation
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data)) # 90% data untuk pelatihan
train_data = data[:n]
val_data = data[n:]

# Fungsi untuk mengambil batch data
def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y

@torch.no_grad()
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] = losses.mean()
    model.train()
    return out

```

```

class Head(nn.Module):

```

```

    """ Satu head dari self-attention """

```

```

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False) # Layer key
        self.query = nn.Linear(n_embd, head_size, bias=False) # Layer query
        self.value = nn.Linear(n_embd, head_size, bias=False) # Layer value
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size))) # Masking lower triangular

        self.dropout = nn.Dropout(dropout) # Dropout layer

    def forward(self, x):
        B, T, C = x.shape
        k = self.key(x) # (B, T, C)
        q = self.query(x) # (B, T, C)
        # Menghitung skor perhatian
        wei = q @ k.transpose(-2, -1) * C**-0.5 # Skor perhatian
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # Masking
        wei = F.softmax(wei, dim=-1) # Normalisasi
        wei = self.dropout(wei) # Terapkan dropout
        # Agregasi nilai berdasarkan perhatian
        v = self.value(x) # (B, T, C)
        out = wei @ v # (B, T, C)
        return out

```

```

class MultiHeadAttention(nn.Module):

```

```

    """ Multi-head self-attention secara paralel """

```

```

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)]) # Daftar head
        self.proj = nn.Linear(n_embd, n_embd) # Proyeksi linear
        self.dropout = nn.Dropout(dropout) # Dropout layer

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1) # Gabungkan output semua head
        out = self.dropout(self.proj(out)) # Proyeksi dan dropout
        return out

```

```

class FeedFoward(nn.Module):

```

```

    """ Layer feed-forward sederhana """

```

```

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd), # Layer linear pertama
            nn.ReLU(), # Aktivasi non-linear
            nn.Linear(4 * n_embd, n_embd), # Layer linear kedua
            nn.Dropout(dropout), # Dropout layer
        )

    def forward(self, x):
        return self.net(x)

```

```

class Block(nn.Module):
    """ Blok Transformer: komunikasi diikuti oleh komputasi """

    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head # Ukuran setiap head
        self.sa = MultiHeadAttention(n_head, head_size) # Self-attention
        self.ffwd = FeedFoward(n_embd) # Feed-forward network
        self.ln1 = nn.LayerNorm(n_embd) # Layer normalization pertama
        self.ln2 = nn.LayerNorm(n_embd) # Layer normalization kedua

    def forward(self, x):
        x = x + self.sa(self.ln1(x)) # Residual connection dengan self-attention
        x = x + self.ffwd(self.ln2(x)) # Residual connection dengan feed-forward
        return x

class BigramLanguageModel(nn.Module):
    """ Model Bahasa Bigram yang sederhana """

    def __init__(self):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd) # Embedding token
        self.position_embedding_table = nn.Embedding(block_size, n_embd) # Embedding posisi
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)]) # Transformer blocks
        self.ln_f = nn.LayerNorm(n_embd) # Layer normalization akhir
        self.lm_head = nn.Linear(n_embd, vocab_size) # Head untuk prediksi token

    def forward(self, idx, targets=None):
        B, T = idx.shape

        tok_emb = self.token_embedding_table(idx) # Embedding token
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # Embedding posisi
        x = tok_emb + pos_emb # Gabungkan embedding token dan posisi
        x = self.blocks(x) # Pass melalui transformer blocks
        x = self.ln_f(x) # Layer normalization akhir
        logits = self.lm_head(x) # Prediksi logits

        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) # Hitung loss

        return logits, loss

    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -block_size:] # Crop konteks
            logits, loss = self(idx_cond)
            logits = logits[:, -1, :] # Fokus pada timestep terakhir
            probs = F.softmax(logits, dim=-1) # Probabilitas token berikutnya
            idx_next = torch.multinomial(probs, num_samples=1) # Sampling token
            idx = torch.cat((idx, idx_next), dim=1) # Tambahkan token ke sequence
        return idx

model = BigramLanguageModel() # Inisialisasi model

```

```

m = model.to(device) # Pindahkan model ke device
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters') # Cetak jumlah parameter

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

for iter in range(max_iters):
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}") # Cetak loss

    xb, yb = get_batch('train') # Ambil batch data

    logits, loss = model(xb, yb) # Hitung loss
    optimizer.zero_grad(set_to_none=True) # Reset gradien
    loss.backward() # Backpropagation
    optimizer.step() # Update parameter

# Generate teks dari model
context = torch.zeros((1, 1), dtype=torch.long, device=device) # Inisialisasi konteks
print(decode(m.generate(context, max_new_tokens=2000)[0].tolist())) # Cetak teks yang dihasilkan

```

Penjelasan Detail :

Kode di atas merupakan implementasi model bahasa bigram sederhana menggunakan arsitektur Transformer dengan PyTorch. Berikut adalah penjelasan lebih mendalam mengenai komponen-komponen utama:

1. Import dan Hyperparameters:

- Mengimpor library PyTorch dan modul terkait.
- Mendefinisikan hyperparameters seperti ukuran batch, panjang blok, jumlah iterasi, laju pembelajaran, dan lainnya yang akan digunakan selama pelatihan model.

2. Data Preparation:

- Membaca data teks dari file input.txt.
- Mengidentifikasi karakter unik dalam teks untuk membangun vocab.
- Membuat mapping dari karakter ke integer (stoi) dan sebaliknya (itos).
- Encoding teks menjadi tensor integer dan membagi data menjadi set pelatihan (90%) dan validasi (10%).

3. Data Loading:

- Fungsi `get_batch` digunakan untuk mengambil batch data secara acak dari set pelatihan atau validasi.
- Fungsi `estimate_loss` menghitung rata-rata loss pada set pelatihan dan validasi untuk memantau performa model selama pelatihan.

4. Model Arsitektur:

- **Head:** Implementasi satu head dari self-attention, termasuk perhitungan skor perhatian dan agregasi nilai.
- **MultiHeadAttention:** Menggabungkan beberapa head self-attention untuk meningkatkan kemampuan model dalam menangkap berbagai aspek hubungan antar token.
- **FeedFoward:** Jaringan feed-forward sederhana dengan lapisan linear dan aktivasi non-linear (ReLU).

- **Block:** Blok Transformer yang terdiri dari multi-head self-attention diikuti oleh jaringan feed-forward, masing-masing dengan residual connections dan layer normalization.
- **BigramLanguageModel:** Model utama yang menggabungkan embedding token dan posisi, beberapa blok Transformer, dan head linear untuk prediksi token berikutnya. Juga dilengkapi dengan metode generate untuk menghasilkan teks baru berdasarkan konteks yang diberikan.

5. Training Loop:

- Menginisialisasi model dan optimizer (AdamW).
- Melakukan pelatihan selama max_iters iterasi, di mana setiap eval_interval iterasi melakukan evaluasi pada set pelatihan dan validasi.
- Pada setiap iterasi, mengambil batch data, menghitung loss, melakukan backpropagation, dan memperbarui parameter model.

6. Generasi Teks:

- Setelah pelatihan, model digunakan untuk menghasilkan teks baru. Dimulai dengan konteks awal (token kosong), model secara iteratif memprediksi token berikutnya dan memperbaruinya ke dalam sequence hingga mencapai jumlah token yang diinginkan (max_new_tokens).

Kesimpulan

Kode ini menggambarkan implementasi dasar model bahasa bigram menggunakan arsitektur Transformer di PyTorch. Dengan menggunakan multi-head self-attention dan beberapa blok Transformer, model dapat menangkap hubungan antar token dalam konteks yang lebih luas dibandingkan dengan model bigram tradisional yang hanya mempertimbangkan token sebelumnya. Meskipun sederhana, pendekatan ini menyediakan fondasi yang kuat untuk membangun model bahasa yang lebih kompleks dan efektif dalam berbagai tugas pemrosesan bahasa alami.

OUTPUT

```
0.209729 M parameters
step 0: train loss 4.4116, val loss 4.4022
step 100: train loss 2.6568, val loss 2.6670
step 200: train loss 2.5090, val loss 2.5058
step 300: train loss 2.4195, val loss 2.4335
step 400: train loss 2.3503, val loss 2.3567
step 500: train loss 2.2965, val loss 2.3130
step 600: train loss 2.2410, val loss 2.2497
step 700: train loss 2.2055, val loss 2.2191
step 800: train loss 2.1631, val loss 2.1859
step 900: train loss 2.1247, val loss 2.1508
step 1000: train loss 2.1020, val loss 2.1285
step 1100: train loss 2.0718, val loss 2.1198
step 1200: train loss 2.0384, val loss 2.0802
step 1300: train loss 2.0270, val loss 2.0661
step 1400: train loss 1.9934, val loss 2.0376
step 1500: train loss 1.9701, val loss 2.0303
step 1600: train loss 1.9620, val loss 2.0473
step 1700: train loss 1.9414, val loss 2.0150
step 1800: train loss 1.9072, val loss 1.9971
step 1900: train loss 1.9076, val loss 1.9851
step 2000: train loss 1.8862, val loss 1.9958
step 2100: train loss 1.8741, val loss 1.9773
step 2200: train loss 1.8598, val loss 1.9623
step 2300: train loss 1.8547, val loss 1.9512
step 2400: train loss 1.8422, val loss 1.9412
step 2500: train loss 1.8163, val loss 1.9432
step 2600: train loss 1.8245, val loss 1.9374
step 2700: train loss 1.8112, val loss 1.9344
step 2800: train loss 1.8037, val loss 1.9243
step 2900: train loss 1.8014, val loss 1.9263
step 3000: train loss 1.7958, val loss 1.9197
step 3100: train loss 1.7696, val loss 1.9198
step 3200: train loss 1.7519, val loss 1.9086
step 3300: train loss 1.7574, val loss 1.9071
step 3400: train loss 1.7569, val loss 1.8985
step 3500: train loss 1.7379, val loss 1.8953
step 3600: train loss 1.7263, val loss 1.8869
step 3700: train loss 1.7306, val loss 1.8870
step 3800: train loss 1.7210, val loss 1.8904
step 3900: train loss 1.7230, val loss 1.8729
step 4000: train loss 1.7144, val loss 1.8641
step 4100: train loss 1.7098, val loss 1.8750
step 4200: train loss 1.7062, val loss 1.8662
step 4300: train loss 1.6977, val loss 1.8477
step 4400: train loss 1.7078, val loss 1.8626
step 4500: train loss 1.6871, val loss 1.8438
step 4600: train loss 1.6862, val loss 1.8312
step 4700: train loss 1.6843, val loss 1.8432
step 4800: train loss 1.6662, val loss 1.8433
step 4900: train loss 1.6685, val loss 1.8307
step 4999: train loss 1.6633, val loss 1.8200
```


Baris ini menandakan bahwa model memiliki sekitar **0,209 juta (209 ribu) parameter**. Jumlah ini merupakan hasil penjumlahan semua bobot (weight) dan bias dalam arsitektur **BigramLanguageModel** yang telah dibuat (termasuk embedding token, embedding posisi, linear layer pada setiap head, dll.).

Train Loss dan Val Loss

- Kolom train loss menandakan seberapa baik model mempelajari pola bahasa pada data pelatihan.
- Kolom val loss menunjukkan performa model pada data validasi (yang berbeda dari data pelatihan), berguna untuk memantau overfitting atau underfitting.

Penurunan Nilai Loss

- Pada awal pelatihan (step 0), train loss sekitar 4.41, val loss 4.40.
- Seiring bertambahnya iterasi, nilai loss pelatihan dan validasi cenderung turun. Misalnya, pada step 300 sudah sekitar 2.42 – 2.43, lalu mencapai ~1.66 – 1.82 di akhir pelatihan (step 5000).
- Penurunan ini menandakan model semakin mampu memprediksi karakter berikutnya dengan lebih akurat.

Lamanya Pelatihan

- Kode menetapkan `max_iters = 5000`, artinya kita melakukan hingga 5000 *training steps*.
- Setiap `eval_interval = 100`, kita mencetak nilai train loss dan val loss.

```
ROMEO:
But you froth him, what wilth humb.

WARTINSA:
I life like, to too wherefings,
Or weal! savied to thy! but too your not one you gliman;
This Iell throws you
And than sleen thus mingred, by
shat fair comes wowe but and rithre souls shall; there some not.

LUCIO:
Lord,----
But thou sging them this my frecepers?
This thou sovore.

ISABELLA:
The pitsade but grove to him, he's subdot!
Thy kneel wimith tybreathing was naje to all,
Whelt endeeperly comfsir, whom thou strave so grave.

MENCUS Mennerily, tompy begs to die;
Haid the stabts his will havolss you,
The mustict? the him? Mrancese in some, I and pitaty's!
Your slow here him, and that, let that conpert wat you
Why deaths!
As that eglights in made than one is no drepittion with!
He affecry prinritto-forroced doge,
With fly more immlaous, I mast,
While us fecend, how-fier, I'll that confect to-marrance;
Nav the hlanged this armsond flilveriant.
```

CORIOLE:

First both,
Where, whose to the bote thmy bruut I ditsp.

MENRY VOLINGBRUS:

Genter's, by? Richardowing you wind you,
Non't, more is in meethet, thou prost to in was thus;
For toward merring that waste; Dirdst weye tindery what it to repon hims;
And the 'tis lacect if Con to suritignned.

ARTIO

Struktur Mirip Shakespeare

- Kode di atas mencontohkan hasil model ketika diberikan *prompt* minimal (konteks kosong) lalu secara otoregresif memprediksi token demi token sebanyak 2000 token.
- Terlihat model mencoba meniru gaya *play* Shakespeare, menyisipkan nama tokoh (ROMEO:, LUCIO:, ISABELLA:) dan dialog layaknya naskah drama.

Keterbatasan Koherensi

- Meskipun nama karakter, beberapa kata, dan struktur dramatis muncul, teks masih terasa acak, misalnya:
"But you froth him, what wilth humb." "I life like, to too wherefings,"

- Hal ini wajar karena:
 - Model hanya dilatih dengan ukuran embedding relatif kecil (64), jumlah layer (4) yang tidak terlalu dalam, dan jumlah iterasi (5000) yang terbatas.
 - Data pun di-*tokenize* per karakter, sehingga kapasitas model belum terlalu besar dibanding *model bahasa modern* (misalnya GPT).

CONCLUTION

Model Transformer sederhana yang terdiri atas 4 blok, 4 head, dan embedding size 64 ini memiliki sekitar 0,21 juta parameter. Selama pelatihan, nilai *train loss* dan *val loss* menunjukkan penurunan yang konsisten dari kisaran 4,4 hingga sekitar 1,66–1,82, menandakan peningkatan kemampuan model dalam memprediksi karakter. Meskipun demikian, teks yang dihasilkan masih mengandung ketidaksesuaian tata bahasa dan kata-kata acak, mencerminkan bahwa model belum sepenuhnya koheren, walau sudah berhasil mempelajari sebagian pola bahasa dasar.