Building a GPT from Scratch

This is an extended version of Andrej Karpathy's notebook in addition to his Zero To Hero video on GPT.

Adapted by:

Prof. Dr.-Ing. Antje Muntzinger, University of Applied Sciences Stuttgart

antje.muntzinger@hft-stuttgart.de

We'll construct a character-level **GPT (Generative Pretrained Transformer)** model from scratch. **Transformer** is the name of the underlying neural net architecture that was introduced in the 2017 groundbreaking paper "Attention is All You Need" (Link at the bottom). The model will be trained on different texts, for example Shakespeare, Goethe's "Faust", the "Lord of the Rings" or books from Jane Austen, and will be able to generate new text based on the text from the book.

NOTE: You may answer in English or German.

Table of Contents

1. Loading the Data

```
In [192… # import torch
         import torch
         import torch.nn as nn
         from torch.nn import functional as F
         torch.manual_seed(1337)
```

Out[192… `<torch._C.Generator at 0x126a1f810>`

```
In [193… # select the right file and read it in to inspect it
         with open('faust.txt', 'r', encoding='utf-8') as f:
         # with open('shakespeare.txt', 'r', encoding='utf-8') as f:
         # with open('austen.txt', 'r', encoding='utf-8') as f:
         # with open('LOTR.txt', 'r') as f:
         # with open('LOTR_TVscript.txt', 'r') as f:
             text = f.read()
```

TODO: 1a) Find out the length of the dataset and print the first 1000 characters! **(2 points)**

```
In [194… # YOUR CODE GOES HERE
         print("length of dataset in characters: ", len(text))

         # let's look at the first 1000 characters
         print(text[:1000])
```

```
length of dataset in characters:  205807
Faust:
Der Tragödie erster Teil

by Johann Wolfgang von Goethe


Zueignung


Ihr naht euch wieder, schwankende Gestalten,
Die früh sich einst dem trüben Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fühl ich mein Herz noch jenem Wahn geneigt?
Ihr drängt euch zu! nun gut, so mögt ihr walten,
Wie ihr aus Dunst und Nebel um mich steigt;
Mein Busen fühlt sich jugendlich erschüttert
Vom Zauberhauch, der euren Zug umwittert.

Ihr bringt mit euch die Bilder froher Tage,
...truncated...
```

TODO: 1b) Store all unique characters that occur in this text in `chars` and print them. Store the number of unique characters in `vocab_size` and print the result. **(3 points)**

Hint: First make a set of all characters to remove duplicates, then make a list out of them to get a unique ordering, and finally sort them.

```
In [195… # YOUR CODE GOES HERE
         # here are all the unique characters that occur in this text
         chars = sorted(list(set(text)))
         vocab_size = len(chars)
         print(''.join(chars))
         print('vocab_size=', vocab_size)
```

```
 !$%()*,-./0123456789:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
ÄÖÜßäöü—''""•™
vocab_size= 92
```

2. Tokenization

Next, we want to **tokenize** the input. This means, we convert the raw text string to some sequence of integers according to some **vocabulary** of possible elements. A **token** can be a character like here, or a piece of a word like in ChatGPT. For a character-level language model, we just translate each character to an integer (**encoding**) and vice-versa (**decoding**):

```
In [196… # create a mapping from characters to integers
         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: take a string, output a l
         decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of i
         print(itos)
```

```
{0: '\n', 1: ' ', 2: '!', 3: '$', 4: '%', 5: '(', 6: ')', 7: '*', 8: ',', 9:
'-', 10: '.', 11: '/', 12: '0', 13: '1', 14: '2', 15: '3', 16: '4', 17: '5',
18: '6', 19: '7', 20: '8', 21: '9', 22: ':', 23: ';', 24: '?', 25: 'A', 26:
'B', 27: 'C', 28: 'D', 29: 'E', 30: 'F', 31: 'G', 32: 'H', 33: 'I', 34: 'J',
35: 'K', 36: 'L', 37: 'M', 38: 'N', 39: 'O', 40: 'P', 41: 'Q', 42: 'R', 43:
'S', 44: 'T', 45: 'U', 46: 'V', 47: 'W', 48: 'X', 49: 'Y', 50: 'Z', 51: 'a',
52: 'b', 53: 'c', 54: 'd', 55: 'e', 56: 'f', 57: 'g', 58: 'h', 59: 'i', 60:
'j', 61: 'k', 62: 'l', 63: 'm', 64: 'n', 65: 'o', 66: 'p', 67: 'q', 68: 'r',
69: 's', 70: 't', 71: 'u', 72: 'v', 73: 'w', 74: 'x', 75: 'y', 76: 'z', 77:
'Ä', 78: 'Ö', 79: 'Ü', 80: 'ß', 81: 'ä', 82: 'ö', 83: 'ü', 84: '—', 85: ''',
86: ''', 87: '"', 88: '"', 89: '•', 90: '™', 91: '\ufeff'}
```

TODO: 2a) Test the code above by encoding some sentence of your choice and decoding it again. Print the encoded and decoded result. **(2 points)**

```
In [197… # YOUR CODE GOES HERE

         sentence = "Hello, Faust!"

         encoded_test_sentence = encode(sentence)
         print("Encoded:", encoded_test_sentence)

         decoded_test_sentence = decode(encoded_test_sentence)
         print("Decoded:", decoded_test_sentence)
```

```
Encoded: [32, 55, 62, 62, 65, 8, 1, 30, 51, 71, 69, 70, 2]
Decoded: Hello, Faust!
```

Note that tokenization is a trade-off between vocabulary size and sequence length: Large vocabularies will lead to shorter encoding sequences and vice versa. For example, encoding each character results in a short vocabulary of 26 tokens for the standard alphabet plus some more for special characters, but each word consists of longer encodings. On the other hand, encoding on word level means each word is encoded as a single token, but the vocabulary will be much larger (up to a whole dictionary of hundreds of thousands of words for one language). In practice, for example in ChatGPT, **sub word encodings** are used, which means not encoding entire words, but also not encoding individual characters. Instead, some intermediate format is used, for example the word 'undefined' could be encoded as three tokens: 'un', 'define', 'd'.

TODO: 2b) Encode the entire text dataset and store it into a `torch.tensor` with `dtype=torch.long`. This will be our input data for the model, and we name it `data`. Print the shape and dtype of `data` and the first 1000 characters of the encoded text for comparison with the text above. **(3 points)**

In [198...
```python
# YOUR CODE GOES HERE

# Store the encoded text in a torch.tensor
data = torch.tensor(encode(text), dtype=torch.long)
print("Shape of data:", data.shape)
print("Dtype of data:", data.dtype)

print("First 1000 encoded characters:", data[:1000])

#print("First 1000 decoded characters", decode(data.tolist())[:1000])
```

```
Shape of data: torch.Size([205807])
Dtype of data: torch.int64
First 1000 encoded characters: tensor([91, 30, 51, 71, 69, 70, 22,  0, 28, 5
5, 68,  1, 44, 68, 51, 57, 82, 54,
        59, 55,  1, 55, 68, 69, 70, 55, 68,  1, 44, 55, 59, 62,  0,  0, 52,
75,
         1, 34, 65, 58, 51, 64, 64,  1, 47, 65, 62, 56, 57, 51, 64, 57,  1,
72,
        65, 64,  1, 31, 65, 55, 70, 58, 55,  0,  0,  0, 50, 71, 55, 59, 57,
64,
        71, 64, 57,  0,  0,  0, 33, 58, 68,  1, 64, 51, 58, 70,  1, 55, 71,
53,
        58,  1, 73, 59, 55, 54, 55, 68,  8,  1, 69, 53, 58, 73, 51, 64, 61,
55,
        64, 54, 55,  1, 31, 55, 69, 70, 51, 62, 70, 55, 64,  8,  0, 28, 59,
55,
         1, 56, 68, 83, 58,  1, 69, 59, 53, 58,  1, 55, 59, 64, 69, 70,  1,
54,
        55, 63,  1, 70, 68, 83, 52, 55, 64,  1, 26, 62, 59, 53, 61,  1, 57,
55,
        76, 55, 59, 57, 70, 10,  0, 46, 55, 68, 69, 71, 53, 58,  1, 59, 53,
58,
         1, 73, 65, 58, 62,  8,  1, 55, 71, 53, 58,  1, 54, 59, 55, 69, 63,
51,
        62,  1, 56, 55, 69, 70, 76, 71, 58, 51, 62, 70, 55, 64, 24,  0, 30,
83,
        58, 62,  1, 59, 53, 58,  1, 63, 55, 59, 64,  1, 32, 55, 68, 76,  1,
64,
        65, 53, 58,  1, 60, 55, 64, 55, 63,  1, 47, 51, 58, 64,  1, 57, 55,
64,
        55, 59, 57, 70, 24,  0, 33, 58, 68,  1, 54, 68, 81, 64, 57, 70,  1,
55,
        71, 53, 58,  1, 76, 71,  2,  1, 64, 71, 64,  1, 57, 71, 70,  8,  1,
69,
        65,  1, 63, 82, 57, 70,  1, 59, 58, 68,  1, 73, 51, 62, 70, 55, 64,
8,
         0, 47, 59, 55,  1, 59, 58, 68,  1, 51, 71, 69,  1, 28, 71, 64, 69,
70,
...truncated...
```

---

3. Making Training Mini-Batches

TODO: 3a) Split the data into 90% training and 10% validation data and store the result in `train_data` and `val_data`, respectively. We keep the validation data to detect overfitting: We don't want just a perfect memorization of this exact input text, we want a neural network that creates new text in a similar style. **(2 points)**

In [199…
```python
# YOUR CODE GOES HERE

train_data = data[:int(len(data)*0.9)]
print ("Length of the training data:",len(train_data))
```

```
val_data = data[int(len(data)*0.9):]
print ("Length of the validation data:",len(val_data))

print("Check against total length; must be equal 0: (len(data) - len(train_d
```

```
Length of the training data: 185226
Length of the validation data: 20581
Check against total length; must be equal 0: (len(data) - len(train_data)-le
n(val_data)): 0
```

We only feed in chunks of data of size 8 here: feeding in all text at once is computationally too expensive. This is called the **block size** or **context length**.

In [200... `block_size = 8`
`train_data[:block_size+1] # +1 because the target is the next character`

Out[200... `tensor([91, 30, 51, 71, 69, 70, 22,  0, 28])`

In this `train_data` chunk of 9 characters, 8 training examples are hidden. Let's spell it out:

In [201...
```
x = train_data[:block_size] # this will be the input
y = train_data[1:block_size+1] # this will be the target
for t in range(block_size):
    context = x[:t+1]
    target = y[t]
    print(f"when input is {context} the target is: {target}")
```

```
when input is tensor([91]) the target is: 30
when input is tensor([91, 30]) the target is: 51
when input is tensor([91, 30, 51]) the target is: 71
when input is tensor([91, 30, 51, 71]) the target is: 69
when input is tensor([91, 30, 51, 71, 69]) the target is: 70
when input is tensor([91, 30, 51, 71, 69, 70]) the target is: 22
when input is tensor([91, 30, 51, 71, 69, 70, 22]) the target is: 0
when input is tensor([91, 30, 51, 71, 69, 70, 22,  0]) the target is: 28
```

Besides efficiency, a second reason to feed in chunks of size `block_size` is to make the Transformer be used to seeing contexts of different lengths, from only 1 token all the way up to `block_size` and every length in between. That is going to be useful later during inference because while we're sampling, we can start the sampling generation with as little as one character of context and the Transformer knows how to predict the next character. Then it can predict everything up to `block_size`. After `block_size`, we have to start truncating because the Transformer will never receive more than block size inputs when it's predicting the next character.

Besides the **time dimension** that we have just looked at, there is also the **batch dimension**: We feed in batches of multiple chunks of text that are all stacked up in a single tensor. This is simply done for efficiency, because the GPUs can process these batches in parallel.

Now let's create random **batches** of training data:

```
batch_size = 4 # how many independent sequences will we process in parallel?
block_size = 8 # what is the maximum context length for predictions?

def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,)) # 4 (=batch_si
    x = torch.stack([data[i:i+block_size] for i in ix]) # stack 4 chunks (4x
    y = torch.stack([data[i+1:i+block_size+1] for i in ix]) # y is the same
    return x, y
```

TODO: 3b) Get a batch of training data and store the inputs and targets in `xb` and `yb`, respectively. Print the results and their shapes. **(2 points)**

HINT: Apply the `get_batch()` function above!

```
# YOUR CODE GOES HERE
# Get a batch of training data
xb, yb = get_batch('train')

print("Input (xb):")
print(xb)
print("Shape of xb:", xb.shape)

print("\nTarget (yb):")
print(yb)
print("Shape of yb:", yb.shape)
```

```
Input (xb):
tensor([[55,  1, 52, 62, 83, 58, 70, 23],
        [73, 59, 55, 54, 55, 68, 61, 55],
        [56, 51, 62, 70,  8,  1, 54, 51],
        [37, 25, 42, 44, 32, 29, 10,  0]])
Shape of xb: torch.Size([4, 8])

Target (yb):
tensor([[ 1, 52, 62, 83, 58, 70, 23,  0],
        [59, 55, 54, 55, 68, 61, 55, 58],
        [51, 62, 70,  8,  1, 54, 51, 80],
        [25, 42, 44, 32, 29, 10,  0, 32]])
Shape of yb: torch.Size([4, 8])
```

TODO: 3c) How many independent training examples for the transformer does this batch contain? **(1 point)**

ANSWER:

Each batch contains `batch_size` independent sequences, and each sequence has `block_size` positions. Therefore, the total number of independent training examples in the batch is `batch_size * block_size` which is `4 * 8 = 32`.

```
In [204...   for b in range(batch_size): # batch dimension
                 for t in range(block_size): # time dimension
                     context = xb[b, :t+1]
                     target = yb[b,t]
                     print(f"when input is {context.tolist()} the target is: {target}")
```

```
when input is [55] the target is: 1
when input is [55, 1] the target is: 52
when input is [55, 1, 52] the target is: 62
when input is [55, 1, 52, 62] the target is: 83
when input is [55, 1, 52, 62, 83] the target is: 58
when input is [55, 1, 52, 62, 83, 58] the target is: 70
when input is [55, 1, 52, 62, 83, 58, 70] the target is: 23
when input is [55, 1, 52, 62, 83, 58, 70, 23] the target is: 0
when input is [73] the target is: 59
when input is [73, 59] the target is: 55
when input is [73, 59, 55] the target is: 54
when input is [73, 59, 55, 54] the target is: 55
when input is [73, 59, 55, 54, 55] the target is: 68
when input is [73, 59, 55, 54, 55, 68] the target is: 61
when input is [73, 59, 55, 54, 55, 68, 61] the target is: 55
when input is [73, 59, 55, 54, 55, 68, 61, 55] the target is: 58
when input is [56] the target is: 51
when input is [56, 51] the target is: 62
when input is [56, 51, 62] the target is: 70
when input is [56, 51, 62, 70] the target is: 8
...truncated...
```

TODO: 3d) Why do the targets look like this, where does the structure come from? What do we input to the transformer? **(2 points)**

ANSWER:

The input to the transformer is the input context `xb` . The transformer uses this context to predict the next character in the sequence, which is the target `yb` . During training, the model learns to predict the next character based on the given context `xb` .

For each position in the sequence, the input context is a slice of the sequence up to that position. For example, if the sequence is `[64, 1, 69, 59, 55, 52, 55, 64]` , the input contexts for each position is:

`[64]`

`[64, 1]`

`[64, 1, 69]`

`[64, 1, 69, 59]`

`[64, 1, 69, 59, 55]`

`[64, 1, 69, 59, 55, 52]`

`[64, 1, 69, 59, 55, 52, 55]`

`[64, 1, 69, 59, 55, 52, 55, 64]`

4. Defining the Network with PyTorch

We use a simple bigram language model to start with, i.e., the model predicts the next character simply on the last character. This bigram model should look familiar from our first notebook! Only now, we implement a bigram model class inheriting from `nn.Module` in PyTorch.

In [205…
```python
class BigramLanguageModel(nn.Module): # subclass of nn.Module

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a
        # e.g. if the input is token 5, the output should be the logits for
        # = the 5th row of the embedding table (see makemore video on bigram
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None): # targets are optional during infe

        # idx and targets are both (B,T) tensor of integers
        # pluck out the embeddings for the tokens in the input (=the row of
        logits = self.token_embedding_table(idx) # (B,T,C) batch size=4, tim

        # if we have targets, compute the CE loss
        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C) # need to reshape for CE-loss in Py
            # (see https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEnt
            targets = targets.view(B*T) # same shape as logits
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # get the predictions (ignore the loss because we don't have tar
            logits, loss = self(idx)
            # focus only on the last time step = prediction for the next tok
            logits = logits[:, -1, :] # becomes (B, C) instead of (B, T, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1) beca
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

model = BigramLanguageModel(vocab_size)
logits, loss = model(xb, yb)
print(logits.shape)
print('loss=', loss)
```

```
idx = torch.zeros((1, 1), dtype=torch.long) # start with a single token = 0

print("\nGenerated text: ")
# generate operates on batch level -> index into the 0th row = single batch
# afterwards convert to simple python list from tensor for decode function
print(decode(model.generate(idx, max_new_tokens=100)[0].tolist()))
```

```
torch.Size([32, 92])
loss= tensor(4.8776, grad_fn=<NllLossBackward0>)

Generated text:

Y:1ufb,":
CtjE8'ÄGm5rOwB.öq%—d.INKvläöSu'$—ÖeEXo1A'M*QPEp" xNSHF!b")"q—tf™FÄÖ3TmA—RmM—
Eß6pßaLwÖ'pa
```

TODO: 4a) Go through the class definition above and explain what each function does! (1-2 sentences per function) **(6 points)**

ANSWER:

init: Initializes the BigramLanguageModel by creating an embedding table of size (`vocab_size` , `vocab_size`) that directly maps each input token to a set of logits predicting the next token's probabilities.

forward: Takes a batch of token indices `idx` and, optionally, their corresponding `targets` . It uses the embedding table to produce logits (scores) for the next token at each position and, if `targets` are provided, computes and returns the cross-entropy loss between the predicted logits and the targets.

generate: Given a starting sequence `idx` , repeatedly predicts the next token's probabilities using the model, samples one token from that distribution, and appends it to the sequence. This process continues `max_new_tokens` times, effectively generating new text autoregressively.

TODO: 4b) How do you interpret the generated text? **(1 point)**

ANSWER:

The generated text appears random. This is due to the following reasons:

- it's a simple BLM
- During generation, the model uses a probabilistic sampling method `(torch.multinomial)` , which introduces randomness. Without strong contextual or linguistic constraints, this randomness leads to nonsensical text.

TODO: 4c) What loss do you expect for this model? Can you compare the actual loss with your expectation? **(2 points)**

ANSWER:

With a `vocab_size` of `92` we expect the following cross-entropy loss for an untrained BLM:

`expected_loss = -log(1/vocab_size) = log(vocab_size) = log(92) = 4,522`

The reported loss is slightly higher than the theoretical `4,522` . This might be due to the random initialization of the embedding table.

Note that up until now, the text history is not used, it is a simple bigram model (only the last character is used to predict the next one). Still, we feed in the whole sequence `xb` , `yb` up to `block_size` for later use.

5. Training

TODO: 5a) Create a PyTorch Adam optimizer with a learning rate of `1e-3` , pass it the model parameters for optimization (`model.parameters()`) and store it in `optimizer` . Check the documentation if needed! **(2 points)**

```python
In [206…  # YOUR CODE GOES HERE
          import torch.optim as optim
          optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

Let's implement the training loop now:

```python
In [207…  batch_size = 32 # increase batch size for better results
          for steps in range(10000): # increase number of steps for good results...

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

              # evaluate the loss
              logits, loss = model(xb, yb) # logits are not needed here
              optimizer.zero_grad(set_to_none=True) # reset the gradients
              loss.backward() # compute the gradients
              optimizer.step() # update the weights

              # print the loss every 100 steps
              if steps % 100 == 0:
                  print(f'step={steps}, loss={loss.item()}')

          print(loss.item())
```

step=0, loss=5.116175174713135

```
step=100, loss=4.827666759490967
step=200, loss=4.703289985656738
step=300, loss=4.792712211608887
step=400, loss=4.5202717781066895
step=500, loss=4.437468528747559
step=600, loss=4.442217826843262
step=700, loss=4.289590358734131
step=800, loss=4.111300468444824
step=900, loss=4.063968658447266
step=1000, loss=3.8623154163360596
step=1100, loss=3.8420138359069824
step=1200, loss=3.837150812149048
step=1300, loss=3.6671175956726074
step=1400, loss=3.575347661972046
step=1500, loss=3.5930984020233154
step=1600, loss=3.56148099899292
step=1700, loss=3.355199098587036
step=1800, loss=3.290208101272583
...truncated...
```

We generate new text based on the trained model:

```
In [208… print("Generated text: ")
         print(decode(model.generate(idx = torch.zeros((1, 1), dtype=torch.long), max
```

Generated text:

```
uspf I, ber deis lnt
Am FAlast Erdeinitend det Übie.
Man Glt
Zanabebeht hr besaunq– NEr hleit wo%xchwan eurid'sttt,
ME.
MAchrdr.
(IEsckolügech GAUSanerkundiei, Pht n.
Far wen FAUn Kergen vellem MAlor'g ur n, S.
(xGleich Oy,
Wichoheinufft ven Deihl omölel Sie PHer Trtf! STjüchand GEPHöngeheickll h m
zur et ht aun d e t n, gelt!
Dicht.
(STOPHEScken wer hresen dun gauch ü0f STOPHend,
De d he Soger hörseldese STOPHÜ9rfandenz binihrberdeinitchen womm g? Aben bi
nk,
Ih dumin.
Geht?
Sorer debelanier, d
```

TODO: 5b) How do you interpret the result? What could be a reason that the output is still suboptimal? **(1 point)**

ANSWER:

The output is mostly nonsensical, with random characters, words, and symbols jumbled together. While it might contain some recognizable characters and occasionally form

something that looks like a word, it doesn't produce coherent sentences or meaningful text. This behavior is expected from a simple bigram model.

Reason for Suboptimal Output:

1. **Bigram Model Limitation:** The model uses only the previous single token to predict the next. This extremely limited context makes it impossible for the model to capture longer-range linguistic structure such as grammar, semantics, or even consistent word formation beyond two-character sequences.

2. **Insufficient Model Complexity and Training:** Even if trained extensively, a bigram model cannot learn complex language patterns. More sophisticated models—e.g., using multi-layer transformers that consider a longer sequence context—are needed to produce coherent and meaningful text. In summary, the suboptimal output is not a sign of poor training but a fundamental limitation of the simple bigram modeling approach.

Summarized code so far (with some additions):

In [209...
```python
# hyperparameters
batch_size = 32 # how many independent sequences will we process in parallel
block_size = 8 # what is the maximum context length for predictions?
max_iters = 3000
eval_interval = 300
learning_rate = 1e-2
device = 'mps'
print('Running on device:',device)
eval_iters = 200
# ------------

# data loading
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    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() # new: we don't need gradients for this function (more effi
def estimate_loss(): # new: average loss over eval_iters iterations
    out = {}
    model.eval() # new: switch to eval mode (not relevant here because no dr
    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()
```

```python
        model.train() # new: switch back to train mode
        return out

# super simple bigram model
class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):

        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

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

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # get the predictions
            logits, loss = self(idx)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

model = BigramLanguageModel(vocab_size)
model = model.to(device) # move the model to the GPU if available

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):

    # every once in a while evaluate the loss on train and val sets
    if iter % eval_interval == 0:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {los

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

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

# generate from the model
context = torch.zeros((1, 1), dtype=torch.long, device=device) # create cont
print(decode(model.generate(context, max_new_tokens=500)[0].tolist()))
```

```
Running on device: mps
step 0: train loss 4.9232, val loss 5.0076
step 300: train loss 2.8302, val loss 3.6603
step 600: train loss 2.4870, val loss 3.5628
step 900: train loss 2.4054, val loss 3.5805
step 1200: train loss 2.3905, val loss 3.6182
step 1500: train loss 2.3761, val loss 3.6328
step 1800: train loss 2.3728, val loss 3.6460
step 2100: train loss 2.3749, val loss 3.6329
step 2400: train loss 2.3661, val loss 3.6580
step 2700: train loss 2.3631, val loss 3.6954

Ichät, d MArerndendige St
Flin;
Kör Blt,

MEYEnzerumicht,

USorst der Schaf sich juechmeunfr'st win dit?
Wein drn, san.
...truncated...
```

6. The Mathematical Trick in Self-Attention

We'll now derive a more complex model that can look at all tokens at once to predict the next one, not just the last token. To use all previous tokens, the simplest idea is to use an average of all previous tokens. For example, the 5th token uses the **channels** (=feature maps, embeddings) of the 1st, 2nd, 3rd, 4th, and 5th token. The average of these is the **feature vector** for the 5th token and summarizes the context / history. Note that we have lost a lot of information, e.g. the order of the tokens, but it's a starting point. Consider the following toy example with batch size 4 , 8 tokens, 2 channels:

In [210…
```python
B,T,C = 4,8,2 # batch, time, channels. Goal: 8 tokens should talk to each ot

x = torch.randn(B,T,C)
x.shape
```

Out[210…
```
torch.Size([4, 8, 2])
```

For each token in each batch in the example vector `x`, we calculate the mean of the tokens that came before it in the time dimension (including itself). The result should be a tensor of shape (B,T,C) where the t-th row of the b-th batch contains the mean of all tokens in this batch that came before this token in the time dimension. We print the original tensor `x` and the resulting tensor `xbow` containing the mean values and make sure the mean values are correct. Here `bow` stands for **bag of words**, which means that each entry is an average of several words (each of the 8 tokens is considered a 'word' here).

```python
In [211… # We want x[b,t] = mean_{i<=t} x[b,i]
        xbow = torch.zeros((B,T,C)) # bow = bag of words = simple average of all pre
        for b in range(B): # iterate over batch dimension
            for t in range(T): # iterate over time dimension
                xprev = x[b,:t+1] # (t,C) # all previous tokens for this batch and t
                xbow[b,t] = torch.mean(xprev, 0) # mean over time dimension
```

```python
In [212… x[0] # 0th batch element
```

```
Out[212… tensor([[-0.0430,  0.7628],
                [-1.0536,  0.1370],
                [-0.2743,  1.5676],
                [ 0.3086, -0.4727],
                [ 0.2641,  0.5549],
                [ 1.3514,  0.2295],
                [-0.2290,  0.0087],
                [-0.1615,  0.5542]])
```

```python
In [213… xbow[0] # vertical average of all previous tokens
```

```
Out[213… tensor([[-0.0430,  0.7628],
                [-0.5483,  0.4499],
                [-0.4570,  0.8225],
                [-0.2656,  0.4987],
                [-0.1596,  0.5099],
                [ 0.0922,  0.4632],
                [ 0.0463,  0.3983],
                [ 0.0203,  0.4178]])
```

Instead of using several nested loops like above, we use a trick with matrix multiplication that is mathematically equivalent but more efficient. Here is a toy example:

```python
In [214… # toy example illustrating how matrix multiplication can be used for a "weig
        torch.manual_seed(42)
        a = torch.ones(3, 3)
        b = torch.randint(0,10,(3,2)).float() # some random data
        c = a @ b
        print('a=')
        print(a)
        print('--')
        print('b=')
        print(b)
        print('--')
```

```
print('c=')
print(c)
```

a=
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
--
b=
tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
--
c=
tensor([[14., 16.],
        [14., 16.],
        [14., 16.]])

As a result, c contains the sum of the column entries of b. Because we only want the "history", not the "future" tokens to influence the result, we use an upper triangular matrix `a` instead, this is called **masking**:

In [215…
```
# toy example illustrating how matrix multiplication can be used for a "weig
torch.manual_seed(42)
a = torch.tril(torch.ones(3, 3)) # lower triangular matrix
b = torch.randint(0,10,(3,2)).float() # some random data
c = a @ b
print('a=')
print(a)
print('--')
print('b=')
print(b)
print('--')
print('c=')
print(c)
# result: first row of b is copied to c, second row is sum of first two rows
# third row is sum of all rows
```

a=
tensor([[1., 0., 0.],
        [1., 1., 0.],
        [1., 1., 1.]])
--
b=
tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
--
c=
tensor([[2., 7.],
        [8., 11.],
        [14., 16.]])

Finally, we have to normalize for averaging:

```python
# toy example illustrating how matrix multiplication can be used for a "weig
torch.manual_seed(42)
a = torch.tril(torch.ones(3, 3)) # lower triangular matrix
a = a / torch.sum(a, 1, keepdim=True) # normalize rows to sum to 1
b = torch.randint(0,10,(3,2)).float() # some random data
c = a @ b
print('a=')
print(a)
print('--')
print('b=')
print(b)
print('--')
print('c=')
print(c)
# result: first row of b is copied to c, second row is sum of first two rows
# third row is sum of all rows + normalized
```

```
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]])
```

TODO: 6a) Now let's go back to our example above and apply the same trick. Define a lower triangular matrix called `wei` (previously `a`) that is normalized to sum to 1 along the rows. Matrix multiply `wei` with `x` to get a new matrix `xbow2` . Make sure that `xbow2` has the same shape as `x` and that it contains the correct values. **(3 points)**

```python
# YOUR CODE GOES HERE

# Define the lower triangular weight matrix
wei = torch.tril(torch.ones(T, T))   # Lower triangular matrix
wei = wei / torch.sum(wei, dim=1, keepdim=True)   # Normalize rows to sum to

# Matrix multiply wei with x
xbow2 = wei @ x   # Shape will be (B, T, C) after broadcasting

print("Original matrix x:")
print(x)
print("\nLower triangular weight matrix wei:")
print(wei)
print("\nComputed xbow2 (Bag of Words):")
print(xbow2)
```

```
Original matrix x:
tensor([[[-0.0430,  0.7628],
         [-1.0536,  0.1370],
         [-0.2743,  1.5676],
         [ 0.3086, -0.4727],
         [ 0.2641,  0.5549],
         [ 1.3514,  0.2295],
         [-0.2290,  0.0087],
         [-0.1615,  0.5542]],

        [[ 0.6630, -0.1377],
         [-0.5458, -0.7306],
         [-0.4159,  1.1934],
         [ 0.1754,  0.1962],
         [-0.5227,  0.9560],
         [ 0.6879,  0.0692],
         [-0.1940,  0.9279],
         [-0.4659,  0.2719]],

        [[ 0.6551, -0.4154],
...truncated...
```

In [218…
```
# YOUR CODE GOES HERE
xbow[0], xbow2[0] # same result
```

TODO: 6b) Now we use yet another mathematically equivalent way to compute the bag of words representation using **Softmax** function (this will be needed later for weighted sum instead of average of previous tokens). We start off with a lower triangular matrix where the lower triangle and diagonal is filled with 0, the upper with `-inf`. After applying the softmax function, the result will be again the `wei` matrix from before. Implement this in the following cell, calculate again the matrix multiplication of the new `wei` and `x` and check the result! **(3 points)**

In [219…
```
# YOUR CODE GOES HERE

# We start with a mask that has zeros for positions in the lower triangle (i
# and -inf in the upper triangle
M = torch.zeros((T, T))
M[torch.tril(torch.ones(T, T)) == 0] = float('-inf')  # upper triangle to -i

# Apply softmax along the time dimension (dim=1)
wei = torch.softmax(M, dim=1)

# Multiply wei2 by x to get the bag of words representation
xbow3 = wei @ x

# Print matrices to verify
print("Mask M:")
print(M)
print("\nWeighing matrix after softmax (wei2):")
print(wei)
print("\nResulting bag-of-words (xbow3):")
print(xbow3)
```

```
Mask M:
tensor([[0., -inf, -inf, -inf, -inf, -inf, -inf, -inf],
        [0.,  0., -inf, -inf, -inf, -inf, -inf, -inf],
        [0.,  0.,  0., -inf, -inf, -inf, -inf, -inf],
        [0.,  0.,  0.,  0., -inf, -inf, -inf, -inf],
        [0.,  0.,  0.,  0.,  0., -inf, -inf, -inf],
        [0.,  0.,  0.,  0.,  0.,  0., -inf, -inf],
        [0.,  0.,  0.,  0.,  0.,  0.,  0., -inf],
        [0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])

Weighing matrix after softmax (wei2):
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3333, 0.3333, 0.3333, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2500, 0.2500, 0.2500, 0.2500, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2000, 0.2000, 0.2000, 0.2000, 0.2000, 0.0000, 0.0000, 0.0000],
        [0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.0000, 0.0000],
        [0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.0000],
        [0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250]])

...truncated...
```

In [220…
```python
xbow[0], xbow2[0], xbow3[0] # same result
```

---

7. Self-Attention

Finally we get to the most important mechanism: **Self-Attention**! This will lead to a weighted average of the tokens (some tokens are more important than others to understand the text) instead of simply using the mean. And here is the idea: Every single token will emit two vectors: A **query** ("What am I looking for?") and a **key** ("What do I contain?"). The query then dot-products with all the keys to determine the similarity = affinity (stored in `wei`). Instead of the raw input `x`, which is private, a **value** is used ("What will I communicate?").

In [221…
```python
# version 4: self-attention!
torch.manual_seed(1337)
B,T,C = 4,8,32 # batch, time, channels (increase channels for more interesti
x = torch.randn(B,T,C)

# let's see a single Head perform self-attention
head_size = 16
key = nn.Linear(C, head_size, bias=False)
query = nn.Linear(C, head_size, bias=False)
value = nn.Linear(C, head_size, bias=False)

k = key(x)   # (B, T, 16) # forward pass of x through the key layer
q = query(x) # (B, T, 16) # forward pass of x through the query layer
# so far, each token has a key and a query vector, no communication yet
wei =  q @ k.transpose(-2, -1) # transpose last 2 dimensions (batch remains
```

```
tril = torch.tril(torch.ones(T, T))
#wei = torch.zeros((T,T)) # old version -> change to data dependent weights
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1) # comment to see intermediate results before no

v = value(x) # we use the aggregated value instead of the raw x
# x is private information to this token, v is the public information for co
out = wei @ v

out.shape
```

Out[221... `torch.Size([4, 8, 16])`

TODO: 7a) Print `wei` and compare it to the previous values. What is the most important change and why is this important here? **(1 point)**

In [222... `wei`

ANSWER:

Previously, the weight matrices (like the lower-triangular normalized matrix) were fixed and position-based. They only reflected the sequence order and provided uniform or position-only weighting to past tokens.

Now, wei is data-dependent. It is computed from the dot products of queries and keys derived from the input x. This means that different tokens can have different attention weights based on their content rather than just their position. This dynamic weighting is the core idea behind self-attention: it lets the model learn which past tokens are most relevant for predicting the next token, leading to more context-aware and meaningful aggregations.

Let's take a closer look at the first weights:

In [223... `wei[0]`

For example, the final entry 0.2391 is the weight for the 8th token. The 8th token emits a query (for example "I am a vowel at position 8, I am looking for consonants at positions up to 4"). All tokens then emit keys, and maybe a consonant at position 4 will emit a key with high number in this channel, meaning "I am a consonant at position 4". The 8th token will therefore have a high weight for the 4th token (0.2297), resulting in a high affinity (dot product) - the 4th and 8th token "have found each other". Through the softmax function, a lot of information from the 4th token will be passed to the 8th token (meaning the 8th token will learn a lot from the 4th).

Some Notes on Attention

- Attention is a **communication mechanism**. It can be seen as nodes in a directed graph looking at each other and aggregating information with a weighted sum from

all nodes that point to them, with data-dependent weights. Here we have `block_size = 8` nodes, where the first node is only pointed to by itself, the second by the first and itself, and so on. Attention can be applied to any directed graph, not only language modeling.

- Each example across batch dimension is processed completely independently, the examples never "talk" to each other across different batches. The batched matrix multiplication above means applying matrix multiplication in parallel in each batch separately. For example here, you can think of 4 different graphs in parallel with 8 noded each, where the 8 nodes only communicate among each other, even though we process 32 nodes at once.

- "Scaled" attention also divides `wei` by `1/sqrt(head_size)`, in the original paper:

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

This makes it so when input Q,K are unit variance, `wei` will be unit variance too and Softmax will stay diffuse and not saturate too much. Without the normalization, using Gaussian input (zero mean and variance 1), the weights will be in the order of `head_size`. Illustration below:

```
In [224... k = torch.randn(B,T,head_size) # k initialized from standard normal distribu
          q = torch.randn(B,T,head_size) # q initialized from standard normal distribu
          wei_unnormalized = q @ k.transpose(-2, -1) # will have variance of head_size
          wei_normalized = q @ k.transpose(-2, -1)* head_size**-0.5 # normalize by sqr
```

```
In [225... k.var() # variance of k: roughly 1
```

```
Out[225... tensor(1.0449)
```

```
In [226... q.var() # variance of q: roughly 1
```

```
Out[226... tensor(1.0700)
```

```
In [227... print(wei_unnormalized.var()) # variance of the dot product: roughly head_si
          print(wei_normalized.var()) # variance of the dot product: roughly 1
```

```
tensor(17.4690)
tensor(1.0918)
```

TODO: 7b) Find out why this is important: Apply softmax to a tensor with entries around 0, then to another tensor with more extreme values. What happens? Write in the answer cell why we want to avoid this. **(2 points)**

HINT: `torch.softmax()` expects an input specifying along which dimension to calculate the normalization (=which dimension should sum to 1), so you can pass `dim=-1` as second input for a 1D tensor. (See https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html for details)

```
In [228…    # YOUR CODE GOES HERE

            # Example 1: Values around zero
            values_small = torch.tensor([0.1, -0.05, 0.0], dtype=torch.float32)
            prob_small = F.softmax(values_small, dim=-1)
            print("Values around zero:", values_small)
            print("Softmax result:", prob_small)
            print("Sum of probabilities:", prob_small.sum().item(), "\n")

            # Example 2: Larger values
            values_large = torch.tensor([50.0, 60.0, 55.0], dtype=torch.float32)
            prob_large = F.softmax(values_large, dim=-1)
            print("Large values:", values_large)
            print("Softmax result:", prob_large)
            print("Sum of probabilities:", prob_large.sum().item())
```

```
Values around zero: tensor([ 0.1000, -0.0500,  0.0000])
Softmax result: tensor([0.3616, 0.3112, 0.3272])
Sum of probabilities: 1.0

Large values: tensor([50., 60., 55.])
Softmax result: tensor([4.5094e-05, 9.9326e-01, 6.6925e-03])
Sum of probabilities: 1.0
```

ANSWER:

If we apply `softmax()` to values around zero (e.g., `[0.1, -0.05, 0.0]`), we get a relatively balanced distribution of probabilities. However, if we apply `softmax()` to very large values (e.g., `[50, 60, 55]`), the exponentiation inside `softmax()` makes the largest value dominate completely, leading to a probability distribution that is almost one-hot. This extreme skew can cause numerical instability, exploding gradients, and poor training dynamics.

By normalizing the dot-products by `1/√head_size`, we keep the attention scores around a more moderate scale, so that the `softmax()` function doesn't collapse to near one-hot distributions prematurely. This helps stabilize training and ensures that the model can properly learn to distribute attention among multiple relevant tokens, rather than always focusing on the single largest value.

Token Encoding and Positional Encoding

We will make one change on the token encoding: Previously, the `token_embedding_table` was of size `(vocab_size, vocab_size)`, which means we directly plucked out the logits from the embedding table. Now we want to introduce an intermediate layer (make the net bigger). Therefore, we introduce a new parameter `n_embd` for the number of embedding dimensions, for example we can choose 32 or 64 for this intermediate representation. So instead of logits, the `token_embedding_table` will give us **token embeddings**. These will be fed to a linear layer afterwards to get the logits:

```
        self.lm_head = nn.Linear(n_embd, vocab_size) # linear layer
        to decode into the vocabulary
```

In the attention mechanism derived so far, there is no notion of space. Attention simply acts over a set of vectors. Remember that we can think of attention as a directed graph, where the nodes have no idea where they are positioned in a space. But space matters in text: For example, "people love animals" has a significantly different meaning than "animals love people", so the ordering of the words is very important. This is why we need to **positionally encode** tokens: So far, we have only encoded each token according to its identity `idx`. But we now also encode its position in a second embedding table: Each position from `0` to `block_size-1` will get its own embedding vector. This is the code snippet from the init function that we will implement below:

```
        self.token_embedding_table = nn.Embedding(vocab_size,
    n_embd) # token embedding according to identity (e.g., first
    character in vocabulary)
        self.position_embedding_table =
    nn.Embedding(block_size, n_embd) # positional encoding
    according to position in text (e.g., first character in text)
```

And here is a code snippet from the forward function, showing how integers from 0 to `block_size` are positionally encoded:

```
        B, T = idx.shape
        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb =
    self.position_embedding_table(torch.arange(T, device=device))
    # (T,C) – integers from 0 to T–1
        x = tok_emb + pos_emb # (B,T,C) via broadcasting
    (pos_emb gets right–aligned, new dimension of 1 gets added,
    broadcasted across batch)
        logits = self.lm_head(x) # (B,T,vocab_size)
```

Right now, this is not useful yet, because we only use the last token in the Bigram model, so the position does not matter. But using attention, it will matter!

Adding a Single Self-Attention Head

Now let's summarize the code so far and add a single self-attention head.

```
In [229…   # hyperparameters
           batch_size = 32 # how many independent sequences will we process in parallel
           block_size = 8 # what is the maximum context length for predictions?
           max_iters = 5000 # new: increase number of iterations due to lower learning
           eval_interval = 500
           learning_rate = 1e-3 # new: lower learning rate (self–attention is more comp
```

```python
device = 'mps'
print('Running on device:',device)
eval_iters = 200
n_embd = 32
# ------------

# data loading
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    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() # we don't need gradients for this function (more efficient
def estimate_loss(): # average loss over eval_iters iterations
    out = {}
    model.eval() # switch to eval mode (not relevant here because no dropout
    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() # switch back to train mode
    return out

# new: single self-attention head
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False) # define the lir
        self.query = nn.Linear(n_embd, head_size, bias=False) # define the l
        self.value = nn.Linear(n_embd, head_size, bias=False) # define the l
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block

    def forward(self, x):
        B,T,C = x.shape # batch, time, channels
        k = self.key(x)   # (B,T,C) - apply the key linear layer
        q = self.query(x) # (B,T,C) - apply the query linear layer

        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs) @ (B,
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T
        wei = F.softmax(wei, dim=-1) # (B, T, T) - apply softmax to get the

        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,C) - apply the value linear layer
        out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C) - weighted aggreg
        return out
```

```python
# super simple bigram model
class BigramLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd) # new:
        self.position_embedding_table = nn.Embedding(block_size, n_embd) # r
        self.sa_head = Head(n_embd) # new: self-attention head
        self.lm_head = nn.Linear(n_embd, vocab_size) # new: linear layer for

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

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=devic
        x = tok_emb + pos_emb # (B,T,C)
        x = self.sa_head(x) # apply one head of self-attention. (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

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

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # new: crop idx to the last block_size tokens (because we now us
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

model = BigramLanguageModel()
model = model.to(device) # move the model to the GPU if available

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
```

```
    # every once in a while evaluate the loss on train and val sets
    if iter % eval_interval == 0:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {los

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

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

# generate from the model
context = torch.zeros((1, 1), dtype=torch.long, device=device) # create cont
print(decode(model.generate(context, max_new_tokens=500)[0].tolist())))
```

```
Running on device: mps
step 0: train loss 4.5645, val loss 4.5525
step 500: train loss 2.5698, val loss 3.5501
step 1000: train loss 2.4292, val loss 3.7169
step 1500: train loss 2.3676, val loss 3.7612
step 2000: train loss 2.3415, val loss 3.8056
step 2500: train loss 2.3190, val loss 3.8577
step 3000: train loss 2.2995, val loss 3.9520
step 3500: train loss 2.2939, val loss 3.9182
step 4000: train loss 2.2832, val loss 3.9271
step 4500: train loss 2.2718, val loss 3.9604

Ichät, dustrerndenndie St
CHin;
KWrlem.
Eimend zerum den,
Sichr ffen Scht.
Leiseberechmeunfr'st win diem rch adr mist iel ochol Hurn, Dlseigescheust zu
err werisen.
Den Gei Baufr dan Zurgegen!
...truncated...
```

We see that the loss decreased a bit, but the result is still not great. We will introduce some more changes following the transformer paper for further improvement:

---

8. Full GPT Implementation

Multi-Head Attention

First, we add **multi-head attention**, which is simply several attention heads running in parallel, then concatenating the result over the channel dimension. A **projection layer** combines the concatenated outputs from all heads into a single unified representation

and projects back to the original pathway. Note that "projection" in the context of Transformer models refers to a linear transformation that can either maintain, reduce, or even increase the dimensionality of the data.

Intuitive Explanation: It helps to have multiple communication channels because these tokens have a lot to talk about - they want to find the consonants, the vowels, the vowels just from certain positions etc. and so it helps to create multiple independent channels of communication to gather lots of different types of data and then decode the output.

![No description has been provided for this image]

Transformer Block

So far, we directly calculated the logits after the attention block, but this was way too fast - intuitively "the tokens looked at each other, but didn't really have time to think on what they found from the other tokens". Therefore, we add a feedforward single layer followed by a ReLU nonlinearity. Both layers together are called the **Transformer Block**, where we combine **communication** (self-attention) with **computation** (feedforward layer). This is on a per token level: Each token independently looks at the other tokens, and once it has gathered all the data, it thinks on that data individually. We implement this in the `Block` class below. The transformer block gets repeated over and over again.

![No description has been provided for this image]

Skip Connections

Also note that the transformer architecture above contains **skip connections (residual connections)**: The network contains parallel paths (one with some computations, one with the identity as "shortcut") that are combined via additions. Additions are great for backpropagation because they distribute gradients equally to both branches, so there is a "shortcut" for the gradients to directly propagate from the output to the input of the network. This avoids the vanishing gradient problem especially in the beginning - the transformer blocks only get more influence over time.

![No description has been provided for this image]

Layer Norm

The transformer architecture uses **layer norm** (called "Norm" in the architecture image above), which is very similar to **batch norm**: Batch norm makes sure that across the batch dimension, any individual neuron has unit gaussian distribution (zero mean, unit

standard deviation). In layer norm, we don't normalize the columns, but the rows, which normalizes over layers instead of over batches:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \varepsilon}} \cdot \gamma + \beta,$$

where γ and β are learned.

In [230…
```python
class LayerNorm1d: # (copied from BatchNorm1d in makemore series)

  def __init__(self, dim, eps=1e-5, momentum=0.1):
    self.eps = eps
    self.gamma = torch.ones(dim)
    self.beta = torch.zeros(dim)

  def __call__(self, x):
    # calculate the forward pass
    xmean = x.mean(1, keepdim=True) # previous batch mean -> index changed f
    xvar = x.var(1, keepdim=True) # previous batch variance -> index changed
    xhat = (x - xmean) / torch.sqrt(xvar + self.eps) # normalize to unit var
    self.out = self.gamma * xhat + self.beta
    # no running mean and variance buffers needed like in batch norm
    return self.out

  def parameters(self):
    return [self.gamma, self.beta]

torch.manual_seed(1337)
module = LayerNorm1d(100)
x = torch.randn(32, 100) # batch size 32 of 100-dimensional vectors
x = module(x)
x.shape
```

Out[230… torch.Size([32, 100])

TODO: 8a) Check if mean and standard deviation of rows and/or columns are normalized now! Write the result in the answer cell. **(2 points)**

In [231…
```python
# YOUR CODE GOES HERE

# Means along dimension 1 (per sample)
print("Row means:", x.mean(dim=1))  # should be close to 0
print("Row stds:", x.std(dim=1))    # should be close to 1

# Means along dimension 0 (per feature across samples)
print("Column means:", x.mean(dim=0)) # no guarantee to be 0
print("Column stds:", x.std(dim=0))    # no guarantee to be 1
```

```
Row means: tensor([ 2.3842e-09,  2.1458e-08,  7.1526e-09,  8.9407e-09,  0.00
00e+00,
        -7.1526e-09,  4.7684e-09,  1.7881e-08,  4.7684e-09, -1.4305e-08,
        -4.7684e-09,  9.5367e-09, -4.7684e-09, -4.7684e-09, -1.9073e-08,
         9.2387e-09, -1.4305e-08,  4.7684e-09, -4.7684e-09, -1.4305e-08,
         2.9802e-09,  1.6689e-08, -1.1921e-08, -1.1921e-08,  3.0994e-08,
         1.1921e-08,  2.8610e-08,  1.1921e-08, -1.4305e-08,  1.7881e-09,
         2.1458e-08, -9.5367e-09])
Row stds: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.
0000, 1.0000,
        1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.00
00,
        1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.00
00,
        1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
Column means: tensor([ 0.1469, -0.5910, -0.3974,  0.0468, -0.1431,  0.0138,
-0.2664,  0.4181,
         0.1426,  0.2191,  0.2554, -0.2625, -0.0543, -0.1050,  0.1541,  0.24
92,
         0.2498,  0.1354, -0.2027, -0.3772,  0.2920,  0.1959, -0.2249, -0.05
74,
         0.1293, -0.1413,  0.1445, -0.2509,  0.1434,  0.0128,  0.0631, -0.24
82,
        -0.0977,  0.0945,  0.1880,  0.0951,  0.0047,  0.2833,  0.1154, -0.30
63,
         0.0510,  0.1602,  0.0598,  0.1157,  0.0083, -0.2541, -0.0447, -0.09
21,
         0.1891, -0.0150, -0.1857, -0.4513, -0.1106,  0.0320,  0.0417,  0.12
72,
        -0.3022, -0.2864,  0.2507, -0.1101,  0.0402,  0.2277,  0.2753,  0.25
77,
        -0.1698,  0.2775, -0.1854,  0.0767, -0.2023,  0.2106,  0.1443,  0.13
91,
...truncated...
```

ANSWER:

row_means will be very close to zero.

row_stds will be very close to one.

col_means and col_stds will not necessarily be normalized.

Note that layer norm is usually applied before the self-attention and linear layer nowadays (unlike the original paper) - one of the very few changes of the transformer architecture during the last years, otherwise mostly the architecture remained unchanged. This is called the **pre-norm formulation**. So here is a code snippet used below showing the two layer norms we will implement, one before the self-attention and one before the linear layer:

```
        x = x + self.sa(self.ln1(x)) # layer norm directly
    applied to x before self-attention
```

```
            x = x + self.ffwd(self.ln2(x)) # layer norm applied
    before linear layer
```

Finally, another layer norm is typically applied at the end of the Transformer and right before the final linear layer that decodes into vocabulary.

The size of the layer norm is `n_embds=32` here, so this is a per token transformation, it just normalizes the features and makes them unit Gaussian at initialization. Because these layer norms contain gamma and beta as trainable parameters, the layer norm may eventually create outputs that are not unit Gaussian depending on the optimization.

Scaling Up the Model

We now have all components together so that we can scale up the model and make it bigger. Therefore, we add a parameter `n_layer=4` to specify that we want 4 transformer blocks.

We also add **dropout** to prevent overfitting: with 4 transformer blocks, the network is getting quite large now. Therefore, we randomly deactivate some connections to prevent them from becoming too dominant. Because the mask of what's being dropped out has changed every single forward backward pass, effectively we end up training an ensemble of sub-networks. At test time, everything is fully enabled and all of those sub-networks are merged into a single ensemble, making it more robust.

No description has been provided for this image

Full GPT with Multi-Head Attention and Transformer Block

We finally get to the full GPT code, adding all the components explained above!

TODO: 8b) In the summarized code below, comment each line to make sure you have understood all GPT components! You may use support from ChatGPT or GitHub Copilot, but double-check the results and be able to explain it yourself. (Yes, this is tedious, but it will help you get an in-depth understanding of the full GPT architecture) **(10 points)**

```
In [232…  # YOUR COMMENTS HERE
          batch_size = 16 #
          block_size = 32 #
          max_iters = 5000 #
          eval_interval = 500 #
          learning_rate = 1e-3 #
          eval_iters = 200 #
          n_embd = 64 #
          n_head = 4 #
          n_layer = 4 #
          dropout = 0.2 #
```

```python
# hyperparameters version 2 - only uncomment when training on GPU (no commen
#"""
batch_size = 64
block_size = 256
max_iters = 5000
eval_interval = 500
learning_rate = 3e-4
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 6
dropout = 0.2
#"""

# ------------

device = 'mps'
print('Running on device:',device) #

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): ### nochmal ckecken

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

        self.dropout = nn.Dropout(dropout) #

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

```python
        q = self.query(x) #

        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 #
        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 #

class MultiHeadAttention(nn.Module):

    def __init__(self, num_heads, head_size):
        super().__init__() #
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads
        self.proj = nn.Linear(n_embd, 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 #

class FeedFoward(nn.Module):

    def __init__(self, n_embd): #
        super().__init__() #
        self.net = nn.Sequential( #
            nn.Linear(n_embd, 4 * n_embd), #
            nn.ReLU(), #
            nn.Linear(4 * n_embd, n_embd), #
            nn.Dropout(dropout), #
        )

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

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 = FeedFoward(n_embd) #
        self.ln1 = nn.LayerNorm(n_embd) #
        self.ln2 = nn.LayerNorm(n_embd) #

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

class GPTLanguageModel(nn.Module):
```

```python
    def __init__(self):
        super().__init__() #
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd) #
        self.position_embedding_table = nn.Embedding(block_size, n_embd) #
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in
        self.ln_f = nn.LayerNorm(n_embd) #
        self.lm_head = nn.Linear(n_embd, vocab_size) #

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

        tok_emb = self.token_embedding_table(idx) #
        pos_emb = self.position_embedding_table(torch.arange(T, device=devic
        x = tok_emb + pos_emb #
        x = self.blocks(x) #
        x = self.ln_f(x) #
        logits = self.lm_head(x) #

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

        return logits, loss #

    def generate(self, idx, max_new_tokens): #
        for _ in range(max_new_tokens): #
            idx_cond = idx[:, -block_size:] #
            logits, loss = self(idx_cond)  #
            logits = logits[:, -1, :] #
            probs = F.softmax(logits, dim=-1) #
            idx_next = torch.multinomial(probs, num_samples=1) #
            idx = torch.cat((idx, idx_next), dim=1) #
        return idx


model = GPTLanguageModel() #
model = model.to(device) #

print(sum(p.numel() for p in model.parameters())/1e6, 'M parameters') #

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

for iter in range(max_iters): #

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

        print("\nSample:")
        context = torch.zeros((1, 1), dtype=torch.long, device=device) #
```

```
        print(decode(model.generate(context, max_new_tokens=200)[0].tolist()

    xb, yb = get_batch('train') #

    logits, loss = model(xb, yb) #
    optimizer.zero_grad(set_to_none=True) #
    loss.backward() #
    optimizer.step() #

print("\nFinal sample:") #
context = torch.zeros((1, 1), dtype=torch.long, device=device) #
print(decode(model.generate(context, max_new_tokens=2000)[0].tolist())) #
```

```
Running on device: mps
10.809692 M parameters

==============
step 0: train loss 4.5842, val loss 4.5758
==============

Sample:

CLZQ(v2r'SmÜpCpi"ydCr:1TTJ%'O(txjvFN,(o'9O"J(T"ÄWOüYQPXzäVAjg3e.tRl46:ÖAb"E?
IB
u?üm("Y8D3—OCqD.AERp"7ÖhXtQWötiWßB,YfVOdfYdotQvgWW0BM*aKIkutQ.y%Xi"™L$"'r8E:
(DxYz o"$PxtU—hmn:üQ7T/GmöHZfW's/ZIYfBCÜWss*

==============
step 500: train loss 1.7830, val loss 3.7796
==============

Sample:

Arge Brel).
Umme! ksetehrst unt'glauf der Hauf.
...truncated...
```

That's it! We have trained a more powerful GPT model using self-attention. Let's generate a longer text and see how the results look like:

In [233…
```
# generate a longer sample
context = torch.zeros((1, 1), dtype=torch.long, device=device)
new_text = decode(model.generate(context, max_new_tokens=10000)[0].tolist())
print(new_text)
```

```
Seck, da kommt der gelinge Brust.
Es gibt es für weilen,
Wird freundlich und macht;
Soll kein Phalte! Fluch erlaubn erfüllt dich zu verlieben.
Was diese Mand nicht seinen Frisch, wie muß ich bald leicht.
Kommt dran wenig und besitzen!
Schon und hatt eine große Werke seit!
Ihr habt der Verland verehren
Hitter die nicht kennen Allebväten,
In einer verworren Mage nehmt;
Er gelehrt zu mücken,
Und solch in dem Fedel schönen,
So sie stich auch Taten.

MARGARETE.
Was soll mich fühl, man die Sust getan;
Getug meine Mutter vierzen lockt.
(Er schringt ihn gelassen.)

...truncated...
```

In [234…
```python
# save result to a text file
f =  open("GPT_generated_text.txt","w")
f.write(new_text)
f.close()
```

TODO (optional): Apply the code to a different text of your choice! What loss do you achieve? What parameters did you change and why? How do you interpret the output compared to the Shakespeare output?

9. Outlook and Next Steps

Andrej's Suggested Further Experiments

- EX1: The n-dimensional tensor mastery challenge: Combine the `Head` and `MultiHeadAttention` into one class that processes all the heads in parallel, treating the heads as another batch dimension (answer is in nanoGPT).
- EX2: Train the GPT on your own dataset of choice! What other data could be fun to blabber on about? (A fun advanced suggestion if you like: train a GPT to do addition of two numbers, i.e. a+b=c. You may find it helpful to predict the digits of c in reverse order, as the typical addition algorithm (that you're hoping it learns) would proceed right to left too. You may want to modify the data loader to simply serve random problems and skip the generation of train.bin, val.bin. You may want to mask out the loss at the input positions of a+b that just specify the problem using y=-1 in the targets (see CrossEntropyLoss ignore_index). Does your Transformer learn to add? Once you have this, swole doge project: build a calculator clone in GPT, for all of +-*/. Not an easy problem. You may need Chain of Thought traces.)
- EX3: Find a dataset that is very large, so large that you can't see a gap between train and val loss. Pretrain the transformer on this data, then initialize with that

model and finetune it on tiny shakespeare with a smaller number of steps and lower learning rate. Can you obtain a lower validation loss by the use of pretraining?

- EX4: Read some transformer papers and implement one additional feature or change that people seem to use. Does it improve the performance of your GPT?

Decoder and Encoder

Text generation as above only uses the **decoder** part of the transformer architecture. The **decoder attention block** implemented above has **triangular masking**, and is usually used in autoregressive settings, like language modeling.

In other settings, we do want "future" tokens to influence the prediction, so we do not use triangular masking. For example, in sentiment analysis, we look at a whole sentence at once, then predict the sentiment "happy" or "sad" of the speaker. This can be realized using an **encoder** attention block. To implement an encoder attention block, we can simply delete the single line that does masking with `tril`, allowing all tokens to communicate. Attention does not care whether tokens from the future contribute or not, it supports arbitrary connectivity between nodes.

From Self-Attention to Cross-Attention

Self-attention means that the keys and values are produced from the same source as queries. In **cross-attention**, the queries still get produced from `x`, but the keys and values come from some other, external source (e.g. an encoder module). For example, when translating from French to English, we condition the decoding on the past decoding *and* the fully encoded french prompt.

```
In [ ]:  # French to English translation example:

         # <--------- ENCODE ------------------><--------------- DECODE --------------
         # les réseaux de neurones sont géniaux! <START> neural networks are awesome!
```

From GPT to ChatGPT

There is still a long way to go from our toy GPT example to ChatGPT. First of all, ChatGPT's **pre-training** was done on a large chunk of internet, resulting in a decoder-only transformer for text generation. So the pretraining is quite similar to our toy example training, except that we used roughly 10 million parameters and the largest transformer for ChatGPT uses 175 billion (!) parameters. Also it was trained on 300 billion tokens (our training set would be 300.000 tokens roughly when not using character-level tokens, but sub-word chunks). This is about a million fold increase in number of tokens - and today, even bigger datasets are used with trillions of tokens for training on thousands of GPUs!

See the following table for the number of parameters, number of layers, n_embd, number of heads, head size, batch size and learning rate in **GPT-3**:

No description has been provided for this image

After the pre-training, the model will be a document completer, it will not give answers but produce more questions or result in some undefined behavior. For becoming an assistant, further **fine-tuning** is needed using **Reinforcement Learning from Human Feedback (RLHF)**. Here is an overview of manual fine-tuning with human AI trainers (see the OpenAI ChatGPT blog for details, link below):

No description has been provided for this image

Summary

To sum it up, we trained a decoder only Transformer following the famous paper 'Attention is All You Need' from 2017, which is basically a GPT. We saw how using self-attention, we can calculate a weighted average of past tokens to predict the next token. We trained it on different texts (Shakespeare, Faust, Jane Austen etc.) and produced new texts in the same writing style.

Further Reading

- Attention is All You Need paper: https://arxiv.org/abs/1706.03762
- OpenAI GPT-3 paper: https://arxiv.org/abs/2005.14165
- OpenAI ChatGPT blog post: https://openai.com/blog/chatgpt/

In [66]:
```python
# This cell truncates long output to a maximum length, then converts the not
# NOTE: You may have to adapt the path or filename to match your local setup

import sys
import os

# Add the parent directory to the sys.path
sys.path.append(os.path.abspath(os.path.join('..')))

# truncate long cell output to avoid large pdf files
from helpers.truncate_output import truncate_long_notebook_output
truncated = truncate_long_notebook_output('3_Character_Level_GPT__student.ip

# convert to pdf with nbconvert
if truncated:
    !jupyter nbconvert --to webpdf --allow-chromium-download TRUNCATED_3_Cha
else:
    !jupyter nbconvert --to webpdf --allow-chromium-download "3_Character_Le
```

```
Output in 3_Character_Level_GPT__student.ipynb above threshold seen and so a
NEW version has been made: `TRUNCATED_3_Character_Level_GPT__student.ipynb`.
[NbConvertApp] Converting notebook TRUNCATED_3_Character_Level_GPT__student.
ipynb to webpdf
[NbConvertApp] WARNING | Alternative text is missing on 6 image(s).
[NbConvertApp] Building PDF
Removing unused browser at /Users/HAI5RT/Library/Caches/ms-playwright/chromi
um-1140
Downloading Chromium 131.0.6778.33 (playwright build v1148) from https://pla
ywright.azureedge.net/builds/chromium/1148/chromium-mac-arm64.zip
121.6 MiB [                         ] 0% 0.0s121.6 MiB [                         ] 0%
33.3s121.6 MiB [                         ] 0% 19.8s121.6 MiB [
] 0% 13.9s121.6 MiB [                          ] 0% 10.6s121.6 MiB [
] 0% 9.1s121.6 MiB [                          ] 1% 8.1s121.6 MiB [
] 1% 7.7s121.6 MiB [                          ] 1% 7.0s121.6 MiB [
] 2% 7.0s121.6 MiB [                          ] 2% 6.7s121.6 MiB [=
] 2% 6.5s121.6 MiB [=                         ] 3% 6.4s121.6 MiB [=
] 3% 6.3s121.6 MiB [=                         ] 3% 6.0s121.6 MiB [=
] 4% 5.9s121.6 MiB [=                         ] 4% 7.9s121.6 MiB [=
] 4% 8.0s121.6 MiB [=                         ] 4% 7.9s121.6 MiB [=
] 4% 7.8s121.6 MiB [=                         ] 5% 7.8s121.6 MiB [=
] 5% 7.7s121.6 MiB [=                         ] 5% 7.6s121.6 MiB [=
] 5% 7.5s121.6 MiB [=                         ] 6% 7.6s121.6 MiB [=
] 6% 7.5s121.6 MiB [=                         ] 6% 7.4s121.6 MiB [=
] 7% 7.3s121.6 MiB [=                         ] 7% 7.8s121.6 MiB [=
] 7% 8.2s121.6 MiB [=                         ] 7% 8.4s121.6 MiB [=
] 7% 8.6s121.6 MiB [=                         ] 7% 8.8s121.6 MiB [=
] 7% 9.0s121.6 MiB [=                         ] 7% 9.1s121.6 MiB [==
] 7% 9.1s121.6 MiB [==                        ] 7% 9.0s121.6 MiB [==
] 8% 9.9s121.6 MiB [==                        ] 8% 9.7s121.6 MiB [==
] 8% 9.6s121.6 MiB [==                        ] 8% 9.4s121.6 MiB [==
] 9% 9.2s121.6 MiB [==                        ] 9% 9.0s121.6 MiB [==
] 9% 8.9s121.6 MiB [==                        ] 10% 8.8s121.6 MiB [==
] 10% 8.7s121.6 MiB [==                        ] 10% 8.5s121.6 MiB [==
] 11% 8.5s121.6 MiB [==                        ] 11% 8.6s121.6 MiB [==
] 11% 8.4s121.6 MiB [==                        ] 11% 8.3s121.6 MiB [==
] 12% 8.2s121.6 MiB [==                        ] 12% 8.1s121.6 MiB [===
] 12% 8.0s121.6 MiB [===                       ] 13% 7.9s121.6 MiB [===
] 13% 7.7s121.6 MiB [===                       ] 14% 7.6s121.6 MiB [===
] 14% 7.4s121.6 MiB [===                       ] 15% 7.8s121.6 MiB [===
] 15% 7.7s121.6 MiB [===                       ] 16% 7.6s121.6 MiB [===
] 16% 7.5s121.6 MiB [===                       ] 16% 7.8s121.6 MiB [===
] 16% 7.7s121.6 MiB [===                       ] 17% 7.7s121.6 MiB [===
] 17% 7.6s121.6 MiB [====                      ] 17% 7.6s121.6 MiB [====
] 17% 7.5s121.6 MiB [====                      ] 18% 7.5s121.6 MiB [====
] 18% 7.4s121.6 MiB [====                      ] 19% 7.4s121.6 MiB [====
] 19% 7.5s121.6 MiB [====                      ] 19% 7.8s121.6 MiB [====
] 20% 7.7s121.6 MiB [====                      ] 20% 7.6s121.6 MiB [====
] 21% 7.5s121.6 MiB [====                      ] 21% 7.4s121.6 MiB [====
] 21% 7.3s121.6 MiB [====                      ] 22% 7.2s121.6 MiB [=====
] 22% 7.2s121.6 MiB [=====                     ] 23% 7.1s121.6 MiB [=====
] 23% 7.0s121.6 MiB [=====                     ] 24% 6.9s121.6 MiB [=====
] 24% 6.8s121.6 MiB [=====                     ] 25% 6.7s121.6 MiB [=====
] 25% 6.6s121.6 MiB [=====                     ] 26% 6.6s121.6 MiB [=====
] 26% 6.5s121.6 MiB [=====                     ] 26% 6.7s121.6 MiB [=====
] 27% 6.6s121.6 MiB [======                    ] 27% 6.7s121.6 MiB [======
```

```
] 27% 6.8s121.6 MiB [======            ] 27% 6.9s121.6 MiB [======
] 27% 7.0s121.6 MiB [======            ] 27% 7.1s121.6 MiB [======
] 27% 7.2s121.6 MiB [======            ] 28% 7.1s121.6 MiB [======
] 28% 7.0s121.6 MiB [======            ] 29% 7.0s121.6 MiB [======
] 29% 6.9s121.6 MiB [======            ] 29% 7.1s121.6 MiB [======
] 29% 7.0s121.6 MiB [======            ] 30% 7.0s121.6 MiB [======
] 30% 6.9s121.6 MiB [======            ] 31% 6.8s121.6 MiB [======
] 31% 6.7s121.6 MiB [======            ] 32% 6.6s121.6 MiB [=======
] 32% 6.6s121.6 MiB [=======           ] 32% 6.5s121.6 MiB [=======
] 33% 6.5s121.6 MiB [=======           ] 33% 6.4s121.6 MiB [=======
] 33% 6.3s121.6 MiB [=======           ] 34% 6.3s121.6 MiB [=======
] 34% 6.2s121.6 MiB [=======           ] 35% 6.2s121.6 MiB [=======
] 35% 6.1s121.6 MiB [=======           ] 36% 6.0s121.6 MiB [=======
] 36% 6.1s121.6 MiB [=======           ] 36% 6.0s121.6 MiB [=======
] 37% 6.0s121.6 MiB [========          ] 37% 5.9s121.6 MiB [========
] 38% 5.9s121.6 MiB [========          ] 38% 5.8s121.6 MiB [========
] 38% 5.7s121.6 MiB [========          ] 39% 5.7s121.6 MiB [========
] 39% 5.6s121.6 MiB [========          ] 40% 5.6s121.6 MiB [========
] 40% 5.5s121.6 MiB [========          ] 41% 5.4s121.6 MiB [========
] 41% 5.3s121.6 MiB [========          ] 42% 5.3s121.6 MiB [=========
] 42% 5.3s121.6 MiB [=========         ] 43% 5.3s121.6 MiB [=========
] 43% 5.2s121.6 MiB [=========         ] 44% 5.2s121.6 MiB [=========
] 44% 5.1s121.6 MiB [=========         ] 45% 5.1s121.6 MiB [=========
] 45% 5.0s121.6 MiB [=========         ] 46% 4.9s121.6 MiB [=========
] 46% 4.8s121.6 MiB [=========         ] 47% 4.8s121.6 MiB [=========
] 47% 4.7s121.6 MiB [=========         ] 48% 4.7s121.6 MiB [=========
] 48% 4.6s121.6 MiB [=========         ] 49% 4.6s121.6 MiB [=========
] 49% 4.5s121.6 MiB [=========         ] 50% 4.5s121.6 MiB [=========
] 51% 4.4s121.6 MiB [=========         ] 51% 4.3s121.6 MiB [=========
] 52% 4.3s121.6 MiB [==========        ] 52% 4.2s121.6 MiB [==========
] 53% 4.2s121.6 MiB [==========        ] 53% 4.1s121.6 MiB [==========
] 54% 4.0s121.6 MiB [==========        ] 55% 3.9s121.6 MiB [==========
] 56% 3.8s121.6 MiB [==========        ] 56% 3.7s121.6 MiB [==========
] 57% 3.7s121.6 MiB [==========        ] 57% 3.8s121.6 MiB [===========
] 57% 3.8s121.6 MiB [===========       ] 57% 3.7s121.6 MiB [===========
] 58% 3.7s121.6 MiB [===========       ] 58% 3.6s121.6 MiB [===========
] 59% 3.6s121.6 MiB [===========       ] 59% 3.5s121.6 MiB [===========
] 60% 3.5s121.6 MiB [===========       ] 60% 3.4s121.6 MiB [===========
] 61% 3.3s121.6 MiB [===========       ] 62% 3.3s121.6 MiB [===========
] 62% 3.2s121.6 MiB [============      ] 62% 3.2s121.6 MiB [============
] 63% 3.1s121.6 MiB [============      ] 64% 3.1s121.6 MiB [============
] 64% 3.0s121.6 MiB [============      ] 64% 3.1s121.6 MiB [============
] 64% 3.0s121.6 MiB [============      ] 65% 3.0s121.6 MiB [============
] 65% 2.9s121.6 MiB [============      ] 66% 2.9s121.6 MiB [============
] 67% 2.8s121.6 MiB [=============     ] 67% 2.8s121.6 MiB [=============
] 67% 2.7s121.6 MiB [=============     ] 68% 2.7s121.6 MiB [=============
] 69% 2.6s121.6 MiB [=============     ] 69% 2.5s121.6 MiB [=============
] 70% 2.5s121.6 MiB [=============     ] 70% 2.4s121.6 MiB [=============
] 71% 2.4s121.6 MiB [=============     ] 72% 2.4s121.6 MiB [=============
=      ] 72% 2.3s121.6 MiB [==============      ] 73% 2.3s121.6 MiB [=======
======      ] 73% 2.2s121.6 MiB [==============      ] 74% 2.2s121.6 MiB [==
============      ] 74% 2.1s121.6 MiB [==============      ] 75% 2.1s121.6 M
iB [==============      ] 75% 2.0s121.6 MiB [==============      ] 76% 2.0s1
21.6 MiB [==============      ] 76% 2.1s121.6 MiB [==============      ] 76%
2.0s121.6 MiB [==============      ] 77% 2.0s121.6 MiB [==============
] 77% 1.9s121.6 MiB [==============      ] 78% 1.9s121.6 MiB [==============
```

==   ] 78% 1.8s121.6 MiB [================   ] 79% 1.8s121.6 MiB [========
========   ] 79% 1.7s121.6 MiB [================   ] 80% 1.7s121.6 MiB [==
==============   ] 80% 1.6s121.6 MiB [================   ] 81% 1.6s121.6 M
iB [================   ] 82% 1.5s121.6 MiB [================   ] 82% 1.5s1
21.6 MiB [================   ] 83% 1.4s121.6 MiB [================   ] 84%
1.4s121.6 MiB [================   ] 84% 1.3s121.6 MiB [================
] 85% 1.3s121.6 MiB [================   ] 85% 1.2s121.6 MiB [==============
===   ] 86% 1.2s121.6 MiB [================   ] 86% 1.1s121.6 MiB [========
=========   ] 87% 1.1s121.6 MiB [=================   ] 87% 1.0s121.6 MiB [==
===============   ] 88% 1.0s121.6 MiB [=================   ] 88% 0.9s121.6 M
iB [=================   ] 89% 0.9s121.6 MiB [=================   ] 90% 0.8s1
21.6 MiB [=================   ] 91% 0.8s121.6 MiB [=================   ] 91%
0.7s121.6 MiB [==================   ] 92% 0.6s121.6 MiB [==================
] 93% 0.6s121.6 MiB [==================   ] 93% 0.5s121.6 MiB [==============
=====   ] 94% 0.5s121.6 MiB [==================   ] 94% 0.4s121.6 MiB [========
===========   ] 95% 0.4s121.6 MiB [==================   ] 95% 0.3s121.6 MiB [==
================   ] 96% 0.3s121.6 MiB [==================   ] 97% 0.2s121.6 M
iB [===================] 97% 0.2s121.6 MiB [===================] 98% 0.2s1
21.6 MiB [===================] 98% 0.1s121.6 MiB [===================] 99%
0.1s121.6 MiB [===================] 99% 0.0s121.6 MiB [==================
=] 100% 0.0s
Chromium 131.0.6778.33 (playwright build v1148) downloaded to /Users/HAI5RT/
Library/Caches/ms-playwright/chromium-1148
Downloading Chromium Headless Shell 131.0.6778.33 (playwright build v1148) f
rom https://playwright.azureedge.net/builds/chromium/1148/chromium-headless-
shell-mac-arm64.zip
77.5 MiB [   ] 0% 0.0s77.5 MiB [   ] 0% 2
8.4s77.5 MiB [   ] 0% 13.0s77.5 MiB [   ]
0% 9.9s77.5 MiB [   ] 0% 10.5s77.5 MiB [
] 0% 12.6s77.5 MiB [   ] 1% 7.5s77.5 MiB [
] 2% 6.9s77.5 MiB [   ] 2% 6.4s77.5 MiB [=
] 2% 6.1s77.5 MiB [=   ] 3% 5.8s77.5 MiB [=
] 3% 5.7s77.5 MiB [=   ] 3% 5.6s77.5 MiB [=
] 4% 5.5s77.5 MiB [=   ] 4% 5.3s77.5 MiB [=
] 5% 5.2s77.5 MiB [=   ] 5% 5.1s77.5 MiB [=
] 6% 5.0s77.5 MiB [=   ] 6% 4.9s77.5 MiB [=
] 7% 6.1s77.5 MiB [==   ] 7% 5.9s77.5 MiB [==
] 7% 6.0s77.5 MiB [==   ] 8% 6.0s77.5 MiB [==
] 8% 5.9s77.5 MiB [==   ] 8% 5.7s77.5 MiB [==
] 9% 5.6s77.5 MiB [==   ] 9% 5.5s77.5 MiB [==
] 9% 5.4s77.5 MiB [==   ] 10% 5.3s77.5 MiB [==
] 10% 5.2s77.5 MiB [==   ] 11% 5.2s77.5 MiB [==
] 11% 5.1s77.5 MiB [==   ] 11% 5.0s77.5 MiB [==
] 12% 5.0s77.5 MiB [===   ] 12% 5.0s77.5 MiB [===
] 12% 4.9s77.5 MiB [===   ] 13% 4.9s77.5 MiB [===
] 14% 4.8s77.5 MiB [===   ] 14% 4.7s77.5 MiB [===
] 15% 4.7s77.5 MiB [===   ] 15% 4.6s77.5 MiB [===
] 16% 4.5s77.5 MiB [===   ] 16% 4.9s77.5 MiB [===
] 17% 4.9s77.5 MiB [====   ] 17% 4.9s77.5 MiB [====
] 18% 4.8s77.5 MiB [====   ] 18% 4.7s77.5 MiB [====
] 19% 4.7s77.5 MiB [====   ] 19% 4.6s77.5 MiB [====
] 20% 4.5s77.5 MiB [====   ] 21% 4.4s77.5 MiB [====
] 21% 4.5s77.5 MiB [====   ] 21% 4.4s77.5 MiB [====
] 22% 4.4s77.5 MiB [=====   ] 22% 4.3s77.5 MiB [=====
] 23% 4.3s77.5 MiB [=====   ] 23% 4.2s77.5 MiB [=====
] 24% 4.2s77.5 MiB [=====   ] 24% 4.1s77.5 MiB [=====

```
] 25% 4.1s77.5 MiB [====            ] 25% 4.0s77.5 MiB [====
] 26% 4.0s77.5 MiB [=====           ] 26% 4.2s77.5 MiB [=====
] 27% 4.2s77.5 MiB [======          ] 27% 4.1s77.5 MiB [======
] 28% 4.1s77.5 MiB [======          ] 28% 4.0s77.5 MiB [======
] 29% 4.0s77.5 MiB [======          ] 29% 3.9s77.5 MiB [======
] 30% 3.9s77.5 MiB [======          ] 30% 3.8s77.5 MiB [======
] 31% 3.8s77.5 MiB [======          ] 31% 3.7s77.5 MiB [======
] 32% 3.7s77.5 MiB [=======         ] 32% 3.7s77.5 MiB [=======
] 33% 3.6s77.5 MiB [=======         ] 34% 3.6s77.5 MiB [=======
] 34% 3.5s77.5 MiB [=======         ] 35% 3.5s77.5 MiB [=======
] 36% 3.4s77.5 MiB [=======         ] 36% 3.6s77.5 MiB [=======
] 36% 3.5s77.5 MiB [=======         ] 37% 3.5s77.5 MiB [========
] 37% 3.5s77.5 MiB [========        ] 38% 3.5s77.5 MiB [========
] 38% 3.4s77.5 MiB [========        ] 39% 3.4s77.5 MiB [========
] 39% 3.3s77.5 MiB [========        ] 40% 3.3s77.5 MiB [========
] 40% 3.2s77.5 MiB [========        ] 41% 3.2s77.5 MiB [========
] 42% 3.1s77.5 MiB [=========       ] 42% 3.1s77.5 MiB [=========
] 43% 3.1s77.5 MiB [=========       ] 43% 3.0s77.5 MiB [=========
] 44% 3.0s77.5 MiB [=========       ] 45% 3.0s77.5 MiB [=========
] 45% 2.9s77.5 MiB [=========       ] 46% 2.9s77.5 MiB [=========
] 46% 3.0s77.5 MiB [=========       ] 46% 2.9s77.5 MiB [=========
] 47% 2.9s77.5 MiB [==========      ] 47% 2.9s77.5 MiB [==========
] 48% 2.9s77.5 MiB [==========      ] 48% 2.8s77.5 MiB [==========
] 49% 2.8s77.5 MiB [==========      ] 49% 2.7s77.5 MiB [==========
] 50% 2.7s77.5 MiB [==========      ] 51% 2.7s77.5 MiB [==========
] 51% 2.6s77.5 MiB [==========      ] 52% 2.6s77.5 MiB [===========
] 52% 2.5s77.5 MiB [===========     ] 53% 2.5s77.5 MiB [===========
] 54% 2.5s77.5 MiB [===========     ] 54% 2.4s77.5 MiB [===========
] 55% 2.4s77.5 MiB [===========     ] 55% 2.3s77.5 MiB [===========
] 56% 2.4s77.5 MiB [===========     ] 57% 2.4s77.5 MiB [============
] 57% 2.3s77.5 MiB [============    ] 58% 2.3s77.5 MiB [============
] 58% 2.2s77.5 MiB [============    ] 59% 2.2s77.5 MiB [============
] 60% 2.2s77.5 MiB [============    ] 60% 2.1s77.5 MiB [============
] 61% 2.1s77.5 MiB [=============   ] 62% 2.0s77.5 MiB [=============
] 63% 2.0s77.5 MiB [=============   ] 63% 1.9s77.5 MiB [=============
] 64% 1.9s77.5 MiB [=============   ] 65% 1.9s77.5 MiB [=============
] 65% 1.8s77.5 MiB [=============   ] 65% 1.9s77.5 MiB [=============
] 66% 1.9s77.5 MiB [=============   ] 66% 1.8s77.5 MiB [=============
] 67% 1.8s77.5 MiB [==============  ] 67% 1.8s77.5 MiB [==============
] 68% 1.7s77.5 MiB [==============  ] 69% 1.7s77.5 MiB [==============
] 70% 1.6s77.5 MiB [==============  ] 71% 1.6s77.5 MiB [==============
] 71% 1.7s77.5 MiB [==============  ] 72% 1.7s77.5 MiB [===============
] 72% 1.7s77.5 MiB [===============  ] 72% 1.6s77.5 MiB [===============
] 73% 1.6s77.5 MiB [===============  ] 74% 1.6s77.5 MiB [===============
] 74% 1.5s77.5 MiB [===============  ] 75% 1.5s77.5 MiB [===============
] 76% 1.5s77.5 MiB [===============  ] 76% 1.4s77.5 MiB [===============
] 77% 1.4s77.5 MiB [================  ] 77% 1.4s77.5 MiB [================
] 77% 1.3s77.5 MiB [================  ] 78% 1.3s77.5 MiB [================
] 79% 1.3s77.5 MiB [================  ] 79% 1.2s77.5 MiB [================
] 80% 1.2s77.5 MiB [================  ] 81% 1.1s77.5 MiB [================
] 82% 1.1s77.5 MiB [================= ] 82% 1.0s77.5 MiB [================
=    ] 83% 1.0s77.5 MiB [================= ] 84% 0.9s77.5 MiB [============
=====   ] 85% 0.9s77.5 MiB [================= ] 86% 0.8s77.5 MiB [=======
========   ] 87% 0.8s77.5 MiB [================= ] 87% 0.8s77.5 MiB [====
=============   ] 87% 0.7s77.5 MiB [================= ] 88% 0.7s77.5 MiB
[=================   ] 89% 0.6s77.5 MiB [================= ] 90% 0.6s77.5
```

MiB [==================] 91% 0.5s77.5 MiB [==================] 92% 0.5s7
7.5 MiB [===================] 92% 0.4s77.5 MiB [===================] 93%
0.4s77.5 MiB [===================] 94% 0.3s77.5 MiB [===================]
95% 0.3s77.5 MiB [===================] 95% 0.2s77.5 MiB [==================
=] 96% 0.2s77.5 MiB [===================] 97% 0.2s77.5 MiB [==============
======] 97% 0.1s77.5 MiB [====================] 98% 0.1s77.5 MiB [=========
==========] 99% 0.0s77.5 MiB [====================] 100% 0.0s
Chromium Headless Shell 131.0.6778.33 (playwright build v1148) downloaded to
/Users/HAI5RT/Library/Caches/ms-playwright/chromium_headless_shell-1148
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 555368 bytes to TRUNCATED_3_Character_Level_GPT__stud
ent.pdf