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

2. Tokenization

3. Making Training Mini-Batches

4. Defining the Network with PyTorch

5. Training

6. The Mathematical Trick in Self-Attention

7. Self-Attention

8. Full GPT Implementation

9. Outlook and Next Steps

1. Loading the Data

In [108…

```
# import torch
import torch
import torch.nn as nn
from torch.nn import functional as F
torch.manual_seed(1337)
```

Out[108…

```
<torch._C.Generator at 0x1ff7f72b770>
```

In [109…

```
# 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 [110…

```
# 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 [111…

```
# 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:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzÄÖÜßäöü
_‘’“”•™
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 [112…

```
# 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 list of
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers
```

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 [113…

```
# YOUR CODE GOES HERE
sentence = "Hallo Welt!"
encoded = encode(sentence)  # Kodieren
decoded = decode(encoded)   # Dekodieren

# Ergebnisse ausgeben
print("Original:", sentence)
print("Encoded:", encoded)
print("Decoded:", decoded)
```

```
Original: Hallo Welt!
Encoded: [32, 51, 62, 62, 65, 1, 47, 55, 62, 70, 2]
Decoded: Hallo Welt!
```

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 [114...

```
# 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])
```

```
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, 55, 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 [115...

```
# YOUR CODE GOES HERE
# Definiere die Aufteilung
split_ratio = 0.9
n = int(len(data) * split_ratio)  # Grenze für den Trainingsdatensatz

# Aufteilen in Trainings- und Validierungsdaten
train_data = data[:n]
```

```
val_data = data[n:]

# Ausgabe der Größen
print("Size of training data:", train_data.shape)
print("Size of validation data:", val_data.shape)
```

```
Size of training data: torch.Size([185226])
Size of validation data: torch.Size([20581])
```

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 [116…

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

Out[116…

```
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 [117…

```
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:

In [118…

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

```python
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_size) ran
    x = torch.stack([data[i:i+block_size] for i in ix]) # stack 4 chunks (4x8 tenso
    y = torch.stack([data[i+1:i+block_size+1] for i in ix]) # y is the same but one
    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!

In [119…

```python
# YOUR CODE GOES HERE
xb, yb = get_batch('train')

# Ausgabe der Ergebnisse und Formen
print("Inputs (xb):")
print(xb)
print("Shape of xb:", xb.shape)

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

```
Inputs (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])

Targets (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: The batch contains **`batch_size × block_size`** training examples because each position in the sequences (length = `block_size`) is treated as an independent example. With `batch_size = 4` and `block_size = 8`, there are 4 * 8 = 32 examples.

In [120…

```python
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 targets (`yb`) are the input (`xb`) shifted one position to the right. This structure trains the model to predict the next token in the sequence. We input the current tokens (`xb`) into the transformer, which uses embeddings and positional encodings to process them and learn dependencies for next-token prediction.

In [121…

```
# YOUR CODE GOES HERE
```

---

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 [122…

```
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 lookup
        # e.g. if the input is token 5, the output should be the logits for all tok
        # = the 5th row of the embedding table (see makemore video on bigram langua
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

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

```python
        # idx and targets are both (B,T) tensor of integers
        # pluck out the embeddings for the tokens in the input (=the row of the emb
        logits = self.token_embedding_table(idx) # (B,T,C) batch size=4, time=8, ch

        # 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 PyTorch
            # (see https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLos
            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 targets)
            logits, loss = self(idx)
            # focus only on the last time step = prediction for the next token
            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) because we
            # 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 (idx =

print("\nGenerated text: ")
# generate operates on batch level -> index into the 0th row = single batch dimensi
# 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:

?..KGv;äTIEzTBYy8u4.3?M•1y*e(ItThJßsk0B?!—10K%Ef6pMfq,x-p?Jg2Dr(uW;MÖL
Z9lVrE™J;TN’Phs—?BA—?y5B?iök*
```

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

ANSWER:

`__init__(self, vocab_size)`

- Initializes the model by creating an embedding table (`self.token_embedding_table`) of size `(vocab_size, vocab_size)` .
- Each token directly maps to logits for predicting the next token in the sequence (bigram model logic).

`forward(self, idx, targets=None)`

- Extracts logits for each token in `idx` using the embedding table.
- Reshapes `logits` and `targets` to compute loss using `F.cross_entropy` (PyTorch's CrossEntropyLoss).

`generate(self, idx, max_new_tokens)`

- Predicts logits for the next token using `forward()` .
- Focuses on the last time step's logits to sample the next token using `torch.multinomial` .
- Appends the sampled token to `idx` , extending the sequence.

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

ANSWER: The generated text reflects the bigram model's ability to predict the next token based solely on the current one, creating locally coherent sequences. It mimics token patterns from the training data but lacks long-term context or grammar, often resulting in random or nonsensical output. It's mainly used to assess how well the model has learned bigram relationships.

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)**

In [123…

```
# YOUR CODE GOES HERE
import torch.optim as optim

# Erstelle den Adam-Optimizer mit einem Lernrate von 1e-3
optimizer = optim.Adam(model.parameters(), lr=1e-3)

# Überprüfen des Optimizers
print(optimizer)
```

```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    weight_decay: 0
)
```

Let's implement the training loop now:

In [124…

```
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=5.106197357177734
step=100, loss=4.938789367675781
step=200, loss=4.7256340980529785
step=300, loss=4.701963424682617
step=400, loss=4.539300918579102
step=500, loss=4.431313514709473
step=600, loss=4.375298976898193
step=700, loss=4.193227291107178
step=800, loss=4.184295654296875
step=900, loss=4.055790424346924
step=1000, loss=3.8715224266052246
step=1100, loss=3.8623199462890625
step=1200, loss=3.828604221343994
step=1300, loss=3.743664503097534
step=1400, loss=3.566786289215088
step=1500, loss=3.557032585144043
step=1600, loss=3.4300343990325928
step=1700, loss=3.342414140701294
step=1800, loss=3.2471885681152344
step=1900, loss=3.251620054244995
...truncated...
```

We generate new text based on the trained model:

In [125…

```
print("Generated text: ")
print(decode(model.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_to
```

```
Generated text: 


UStobucht in nt
ISohahoniter un zuten MAch!

Wönggunühagelt'sadier ieineralchnd wer zwaßeist wasir mat gelunk.
AUS.

Müme.
Westh s fteu n Dit,

Freb d wibinser.

Sch Hertäle nullach ffü$—Weschteuchlbeur hn stendanfbirn wis inderebelt.

Werottv.
Folkel, um,

Inn: n,
...truncated...
```

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

ANSWER: The improved results show emerging patterns, including capitalization and punctuation, but the text remains incoherent. This could be due to insufficient training steps, non-optimal hyperparameters (e.g., learning rate), limited model architecture, or training data. While the bigram model has learned local token transitions, it lacks coherence and meaning because it relies only on single-token context. The suboptimal results highlight the simplicity of the model, inadequate training, and the randomness of token sampling.

Summarized code so far (with some additions):

In [126…

```
# 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 = 'cuda' if torch.cuda.is_available() else 'cpu' # new: check if GPU is avai
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 efficient)
def estimate_loss(): # new: average loss over eval_iters iterations
    out = {}
    model.eval() # new: switch to eval mode (not relevant here because no dropout e
    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() # 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 lookup
        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)
```

```python
        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 {losses['va

    # 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 context on
print(decode(model.generate(context, max_new_tokens=500)[0].tolist()))
```

```
Running on device: cuda
step 0: train loss 4.9167, val loss 4.9644
step 300: train loss 2.8066, val loss 3.6229
step 600: train loss 2.4820, val loss 3.5374
step 900: train loss 2.4092, val loss 3.5775
step 1200: train loss 2.3891, val loss 3.5933
step 1500: train loss 2.3842, val loss 3.6093
step 1800: train loss 2.3817, val loss 3.6303
step 2100: train loss 2.3745, val loss 3.6583
step 2400: train loss 2.3671, val loss 3.6622
step 2700: train loss 2.3649, val loss 3.6897


Man dem Numüscko Nar ichäleh;

Lerst schn ir.
Marennn GARROPHERiemien rh ßttieerr ls, u d zLEnelar)

keretun, stugeben nerar d uührzun Tintze zuz arl seren! ien Wewargeberst Zwo sarteic
h h Ger,

...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 [127…

```
B,T,C = 4,8,2 # batch, time, channels. Goal: 8 tokens should talk to each other, bu

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

Out[127…

```
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).

In [128…
```
# 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 previous t
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 time (sl
        xbow[b,t] = torch.mean(xprev, 0) # mean over time dimension
```

In [129…
```
x[0] # 0th batch element
```

Out[129…
```
tensor([[ 0.1546, -0.4100],
        [-2.5211,  0.5678],
        [-1.4687, -0.6305],
        [ 1.3620,  1.4733],
        [ 0.3252,  0.3394],
        [-1.7140,  0.7753],
        [ 1.0787,  1.0077],
        [ 0.9735, -0.3782]])
```

In [130…
```
xbow[0] # vertical average of all previous tokens
```

Out[130…
```
tensor([[ 0.1546, -0.4100],
        [-1.1833,  0.0789],
        [-1.2784, -0.1576],
        [-0.6183,  0.2502],
        [-0.4296,  0.2680],
        [-0.6437,  0.3526],
        [-0.3976,  0.4462],
        [-0.2262,  0.3431]])
```

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:

In [131…
```
# toy example illustrating how matrix multiplication can be used for a "weighted ag
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 [132…

```
# toy example illustrating how matrix multiplication can be used for a "weighted ag
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:

In [133…

```
# toy example illustrating how matrix multiplication can be used for a "weighted ag
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 + norm
# 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)**

In [134…

```
# YOUR CODE GOES HERE

# Create a lower triangular matrix `wei`
wei = torch.tril(torch.ones(T, T))  # Lower triangular matrix
wei = wei / torch.sum(wei, dim=1, keepdim=True)  # Normalize rows to sum to 1  # wh

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

# Verify the shape and correctness
print("Original x:")
print(x)
print("Lower triangular matrix wei:")
print(wei)
print("Transformed matrix xbow2:")
print(xbow2)
print("Shape of xbow2:", xbow2.shape)
```

```
Original x:
tensor([[[ 0.1546, -0.4100],
         [-2.5211,  0.5678],
         [-1.4687, -0.6305],
         [ 1.3620,  1.4733],
         [ 0.3252,  0.3394],
         [-1.7140,  0.7753],
         [ 1.0787,  1.0077],
         [ 0.9735, -0.3782]],

        [[ 0.0307,  0.1766],
         [ 0.6101,  0.5782],
         [ 0.5597, -2.0204],
         [-0.8526, -0.1514],
         [ 0.7733, -0.7985],
         [-0.0772, -1.0009],
         [-1.1819,  2.8414],
         [-1.0992, -1.9645]],

        [[ 1.0370, -1.7319],
...truncated...
```

In [135…

```
# 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 [136…

```
# YOUR CODE GOES HERE

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

# 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 [137…

```
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 [138…

```
# version 4: self-attention!
torch.manual_seed(1337)
B,T,C = 4,8,32 # batch, time, channels (increase channels for more interesting resu
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 unchang

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 normaliza

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 communica
out = wei @ v

out.shape
```

Out[138…

```
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 [139…

```
wei
```

ANSWER: The updated `wei` matrix uses softmax to create a smoother, unequal weight distribution, prioritizing recent tokens over earlier ones. This is important as it better captures local context and introduces the concept of weighted importance, which i foundational for attention mechanisms in transformers.

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

In [140…

```
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 [141…
```
k = torch.randn(B,T,head_size) # k initialized from standard normal distribution (z
q = torch.randn(B,T,head_size) # q initialized from standard normal distribution (z
wei_unnormalized = q @ k.transpose(-2, -1) # will have variance of head_size roughl
wei_normalized = q @ k.transpose(-2, -1)* head_size**-0.5 # normalize by sqrt of he
```

In [142…
```
k.var() # variance of k: roughly 1
```

Out[142…
```
tensor(1.0449)
```

In [143…
```
q.var() # variance of q: roughly 1
```

Out[143…
```
tensor(1.0700)
```

In [144…
```
print(wei_unnormalized.var()) # variance of the dot product: roughly head_size=16
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 [145…
```
# YOUR CODE GOES HERE
tensor1 = torch.randn(B, T)  # Values drawn from normal distribution (mean=0, std=1
tensor2 = torch.randn(B, T) * 50  # Increase variance by multiplying by 50

softmax1 = torch.softmax(tensor1, dim=-1)
softmax2 = torch.softmax(tensor2, dim=-1)

print("Softmax output for tensor with values around 0:")
print(softmax1)
```

```
print("Softmax output for tensor with extreme values:")
print(softmax2)
```

```
Softmax output for tensor with values around 0:
tensor([[0.0271, 0.0817, 0.1305, 0.2756, 0.3190, 0.0749, 0.0640, 0.0271],
        [0.0594, 0.0286, 0.0876, 0.4885, 0.0884, 0.0243, 0.1964, 0.0268],
        [0.1688, 0.0455, 0.0190, 0.0903, 0.1352, 0.2590, 0.1980, 0.0842],
        [0.1440, 0.2998, 0.0283, 0.0752, 0.1067, 0.1039, 0.0580, 0.1841]])
Softmax output for tensor with extreme values:
tensor([[0.0000e+00, 1.7097e-36, 2.0791e-32, 1.0000e+00, 0.0000e+00, 6.6812e-28,
         2.2297e-40, 0.0000e+00],
        [1.1487e-37, 2.0005e-30, 9.7666e-01, 0.0000e+00, 1.9366e-17, 2.3340e-02,
         1.1412e-25, 0.0000e+00],
        [7.0443e-40, 1.0000e+00, 1.0191e-35, 5.6052e-45, 2.9427e-44, 1.2900e-16,
         0.0000e+00, 3.4758e-25],
        [0.0000e+00, 3.4243e-34, 8.8562e-43, 2.2897e-26, 1.0000e+00, 7.7974e-12,
         3.5394e-25, 0.0000e+00]])
```

ANSWER: When softmax is applied to tensors with extreme values, it results in a skewed distribution, where one value dominates and others become negligible. This can lead to numerical instability, such as overflow or underflow, and cause the model to overfit to outliers. To avoid these issues, it's important to normalize inputs and control the scale of values before applying softmax, ensuring stable training and better model generalization.

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 [146…

```
# 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 rate
eval_interval = 500
learning_rate = 1e-3 # new: lower learning rate (self-attention is more complex)
device = 'cuda' if torch.cuda.is_available() else 'cpu' # check if GPU is available
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
```

```python
    out = {}
    model.eval() # switch to eval mode (not relevant here because no dropout etc.,
    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 linear lay
        self.query = nn.Linear(n_embd, head_size, bias=False) # define the linear l
        self.value = nn.Linear(n_embd, head_size, bias=False) # define the linear l
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size))

    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, hs, T)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T) -
        wei = F.softmax(wei, dim=-1) # (B, T, T) - apply softmax to get the weights

        # 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 aggregation =
        return out


# 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 lookup
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd) # new: dimens
        self.position_embedding_table = nn.Embedding(block_size, n_embd) # new: pos
        self.sa_head = Head(n_embd) # new: self-attention head
        self.lm_head = nn.Linear(n_embd, vocab_size) # new: linear layer for predic

    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=device)) # (
```

```python
        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 use posit
            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 {losses['va

    # 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 context on
print(decode(model.generate(context, max_new_tokens=500)[0].tolist()))
```

```
Running on device: cuda
step 0: train loss 4.5803, val loss 4.5958
step 500: train loss 2.6074, val loss 3.4934
step 1000: train loss 2.4358, val loss 3.6542
step 1500: train loss 2.3708, val loss 3.6774
step 2000: train loss 2.3398, val loss 3.7079
step 2500: train loss 2.3215, val loss 3.7281
step 3000: train loss 2.3016, val loss 3.8164
step 3500: train loss 2.2976, val loss 3.7968
step 4000: train loss 2.2871, val loss 3.7949
step 4500: train loss 2.2759, val loss 3.8291

AUTn.
Fr rem dier Nquu sen den
Mülest siser resen,
CHEraßen Misteigten rheßt ieen Hoh, umf zut!
NOPSsiker sun,
STOPHOMPHnerar decker?
Ichwer est! Jarl serin! ien Wl'sigeberst nde sarteich has selen bs, der, mmang!
Süng bemas iet brt, abt anm.
...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 [147…

```
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 from 0 t
    xvar = x.var(1, keepdim=True) # previous batch variance -> index changed from 0
    xhat = (x - xmean) / torch.sqrt(xvar + self.eps) # normalize to unit variance
    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[147…

```
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 [148…

```
# YOUR CODE GOES HERE
print(x.mean(dim=1))
print(x.std(dim=1))

print(x.mean(dim=0))
print(x.std(dim=0))
```

```
tensor([-9.5367e-09, -2.3842e-09, -2.0266e-08,  1.7881e-08,  1.6689e-08,
         9.8348e-09,  4.7684e-09,  1.9073e-08, -1.4305e-08, -5.9605e-09,
        -1.3113e-08, -5.9605e-09,  0.0000e+00, -7.1526e-09, -2.0266e-08,
         1.1772e-08, -1.2815e-08,  1.7881e-08,  6.5565e-09, -4.7684e-09,
         9.5367e-09, -3.5763e-09, -2.8610e-08,  4.7684e-09,  3.5763e-09,
        -7.1526e-09, -4.7684e-09,  0.0000e+00,  5.3644e-09, -1.1921e-08,
         4.7684e-09,  1.9073e-08])
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, 1.0000,
        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])
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,  0.2492,
         0.2498,  0.1354, -0.2027, -0.3772,  0.2920,  0.1959, -0.2249, -0.0574,
         0.1293, -0.1413,  0.1445, -0.2509,  0.1434,  0.0128,  0.0631, -0.2482,
        -0.0977,  0.0945,  0.1880,  0.0951,  0.0047,  0.2833,  0.1154, -0.3063,
         0.0510,  0.1602,  0.0598,  0.1157,  0.0083, -0.2541, -0.0447, -0.0921,
         0.1891, -0.0150, -0.1857, -0.4513, -0.1106,  0.0320,  0.0417,  0.1272,
        -0.3022, -0.2864,  0.2507, -0.1101,  0.0402,  0.2277,  0.2753,  0.2577,
        -0.1698,  0.2775, -0.1854,  0.0767, -0.2023,  0.2106,  0.1443,  0.1391,
...truncated...
```

ANSWER: After applying `LayerNorm1d`, each row (i.e., each sample in the batch) is normalized. In other words, the mean of each row is close to zero, and the standard deviation of each row is close to one.

On the other hand, columns are not necessarily normalized. Their mean and standard deviation can vary, since layer normalization operates on each sample independently rather than across the batch dimension.

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 [149…

```
# YOUR COMMENTS HERE
batch_size = 16  # Number of samples processed per batch
block_size = 32  # Length of each input sequence (context window)
max_iters = 5000  # Total number of iterations to train
eval_interval = 500  # How often to evaluate the model on validation set
learning_rate = 1e-3  # Learning rate for optimization
eval_iters = 200  # Number of iterations for loss estimation during evaluation
n_embd = 64  # Dimensionality of the embedding vectors (feature size)
n_head = 4  # Number of attention heads in the multi-head attention layer
n_layer = 4  # Number of transformer blocks
dropout = 0.2  # Dropout rate to prevent overfitting

# hyperparameters version 2 - only uncomment when training on GPU (no comments need
# ''''
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
#

# Determine the device (GPU if available, otherwise CPU)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print('Running on device:', device)  # Print the device being used (CPU or GPU)

# Function to get a batch of data for training or validation
def get_batch(split):
    data = train_data if split == 'train' else val_data  # Select data (train or va
    ix = torch.randint(len(data) - block_size, (batch_size,))  # Randomly choose st
    x = torch.stack([data[i:i+block_size] for i in ix])  # Create input sequences (
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])  # Create target sequen
    x, y = x.to(device), y.to(device)  # Move data to the appropriate device (GPU o
    return x, y  # Return the batch of inputs and targets

# Function to estimate the loss on the training and validation sets
@torch.no_grad()  # Disable gradient computation for loss estimation
def estimate_loss():
    out = {}  # Dictionary to store the losses
    model.eval()  # Set the model to evaluation mode (disables dropout)
    for split in ['train', 'val']:  # Loop over training and validation sets
        losses = torch.zeros(eval_iters)  # Initialize an array to store losses
        for k in range(eval_iters):  # Loop to evaluate the model multiple times
            X, Y = get_batch(split)  # Get a batch of data
            logits, loss = model(X, Y)  # Forward pass to get logits and loss
            losses[k] = loss.item()  # Store the loss for this iteration
        out[split] = losses.mean()  # Calculate average loss for the split (train o
    model.train()  # Switch the model back to training mode
```

```
    return out  # Return the losses for both splits

# Class for a single attention head
class Head(nn.Module):
    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)  # Key transformation
        self.query = nn.Linear(n_embd, head_size, bias=False)  # Query transformati
        self.value = nn.Linear(n_embd, head_size, bias=False)  # Value transformati
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size))
        self.dropout = nn.Dropout(dropout)  # Dropout layer to prevent overfitting

    # Forward pass for the attention head
    def forward(self, x):
        B, T, C = x.shape  # Get the batch size (B), sequence length (T), and embed
        k = self.key(x)  # Apply key transformation to input
        q = self.query(x)  # Apply query transformation to input
        wei = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5  # Scaled dot-product att
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))  # Apply mask
        wei = F.softmax(wei, dim=-1)  # Apply softmax to get attention weights
        wei = self.dropout(wei)  # Apply dropout to attention weights

        v = self.value(x)  # Apply value transformation
        out = wei @ v  # Weighted sum of values based on attention weights

        return out  # Return the output of the attention head

# Multi-head attention that combines multiple attention heads
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)  # Linear projection after concatenat
        self.dropout = nn.Dropout(dropout)  # Dropout layer

    # Forward pass for multi-head attention
    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)  # Concatenate outputs
        out = self.dropout(self.proj(out))  # Apply projection and dropout
        return out  # Return the multi-head attention output

# Feed-forward neural network for the transformer block
class FeedFoward(nn.Module):
    def __init__(self, n_embd):  # Initialize with embedding size
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),  # First linear layer with expansion
            nn.ReLU(),  # ReLU activation
            nn.Linear(4 * n_embd, n_embd),  # Second linear layer with compression
            nn.Dropout(dropout),  # Dropout for regularization
        )

    # Forward pass for the feed-forward network
    def forward(self, x):
        return self.net(x)  # Return the output after passing through the network
```

```python
# Transformer block containing self-attention and feed-forward layers
class Block(nn.Module):
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head  # Size of each attention head
        self.sa = MultiHeadAttention(n_head, head_size)  # Self-attention layer
        self.ffwd = FeedFoward(n_embd)  # Feed-forward network
        self.ln1 = nn.LayerNorm(n_embd)  # Layer normalization before self-attentio
        self.ln2 = nn.LayerNorm(n_embd)  # Layer normalization before feed-forward

    # Forward pass for the transformer block
    def forward(self, x):
        x = x + self.sa(self.ln1(x))  # Add residual connection after self-attentio
        x = x + self.ffwd(self.ln2(x))  # Add residual connection after feed-forwar
        return x  # Return the output

# GPT language model consisting of embeddings, transformer blocks, and output layer
class GPTLanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)  # Token embe
        self.position_embedding_table = nn.Embedding(block_size, n_embd)  # Positio
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n
        self.ln_f = nn.LayerNorm(n_embd)  # Final layer normalization
        self.lm_head = nn.Linear(n_embd, vocab_size)  # Output layer for language m

    # Forward pass through the GPT model
    def forward(self, idx, targets=None):
        B, T = idx.shape  # Get batch size (B) and sequence length (T)
        tok_emb = self.token_embedding_table(idx)  # Get token embeddings
        pos_emb = self.position_embedding_table(torch.arange(T, device=device))  #
        x = tok_emb + pos_emb  # Combine token and positional embeddings
        x = self.blocks(x)  # Pass through transformer blocks
        x = self.ln_f(x)  # Apply final layer normalization
        logits = self.lm_head(x)  # Get logits (predictions) for each token

        if targets is None:  # If no targets are provided, return only logits
            loss = None
        else:
            B, T, C = logits.shape  # Get shape of logits
            logits = logits.view(B * T, C)  # Flatten logits
            targets = targets.view(B * T)  # Flatten targets
            loss = F.cross_entropy(logits, targets)  # Calculate cross-entropy loss

        return logits, loss  # Return logits and loss (if targets are provided)

    # Generate new tokens given a starting context (idx)
    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):  # Loop to generate new tokens
            idx_cond = idx[:, -block_size:]  # Keep only the last block_size tokens
            logits, loss = self(idx_cond)  # Get logits for the next token
            logits = logits[:, -1, :]  # Get logits for the last token
            probs = F.softmax(logits, dim=-1)  # Convert logits to probabilities
            idx_next = torch.multinomial(probs, num_samples=1)  # Sample next token
            idx = torch.cat((idx, idx_next), dim=1)  # Append the next token to the
        return idx  # Return the generated sequence
```

```python
# Initialize the GPT model
model = GPTLanguageModel()
model = model.to(device)  # Move model to the chosen device (CPU or GPU)

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

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

# Training loop
for iter in range(max_iters):
    if iter % eval_interval == 0 or iter == max_iters - 1:  # Periodically evaluate
        losses = estimate_loss()  # Get the losses for train and validation
        print("\n==============")
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['va
        print("==============")

        print("\nSample:")
        context = torch.zeros((1, 1), dtype=torch.long, device=device)  # Start wit
        print(decode(model.generate(context, max_new_tokens=200)[0].tolist()))  # G

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

    # Forward pass and loss calculation
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)  # Clear previous gradients
    loss.backward()  # Backpropagate the loss
    optimizer.step()  # Update model parameters

# Final text generation after training
print("\nFinal sample:")  # Generate a longer sample after training
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(model.generate(context, max_new_tokens=2000)[0].tolist()))  # Generate
```

```
Running on device: cuda
10.809692 M parameters

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

Sample:

GOJ“iZ%%%%•oWMI’Ns‘HF0qoä6eZ88ßDYtX!(JöGP“f‘mb:“3e-2I,beCöa%A8;8DwkxYXzSÖbkXSMDdB!r1
m3 wPP$9Ps–975T”,3v*!ßcfGUNcß3NQb
möY%i(KD,%jl60R6üÖX9Üx9‘Ta”;r“l’ä
9P)ÄCZs5faüP’bQ7cÖa6Z3MaÖp9PtIx*:(sfQWÄ6üeVUkQdo

==============
step 500: train loss 1.7782, val loss 3.8021
==============

Sample:

GIch mich–
...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 [150…

```python
# 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)
```

```
übenden Verzeihen erbärger vor.
Der Menge heilt Schwinder
Dich därmer Namen alten himmlischen Töge?
Der Schlüssel und ihren auf der Schlang
Und hießen hat ein großes Glut vergeben und gaffen,
Und wie er, nicht sich in Stellendsges wandt.
Da werd ihr hier Himmelslicht, ich vom Engel geben
den sich geben aus dem Bißchen
Göttchen mir dieser zu halten.
Kein hoher Kupf und Schwalt,
Der hat keinen großen Kessel gleichen Wert,
Um  ewig erfüllem Stunden beschweren,
Das Geflüg sich ungeben nach und führe sie.

CHOR DER WEIBEL.
Find und die vor kimmen Alten,
Mit vollem Grassen sein,
Wie weiß, das so ein Kessel!

...truncated...
```

In [151…

```
# 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 [152…

```
# French to English translation example:

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

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 [153…

```
# This cell truncates long output to a maximum length, then converts the notebook t
# 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.ipynb')

# convert to pdf with nbconvert
if truncated:
    !jupyter nbconvert --to webpdf --allow-chromium-download TRUNCATED_3_Character_
else:
    !jupyter nbconvert --to webpdf --allow-chromium-download "3_Character_Level_GPT
```

```
Output in 3_Character_Level_GPT__student.ipynb above threshold seen and so a NEW ver
sion 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
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 401179 bytes to TRUNCATED_3_Character_Level_GPT__student.pdf
```