Building GPT from scratch

Khoo An Xian, Summer 2025

Credits to Andrej Kaparthy the GOAT for his amazing teaching!

Mini-shakespeare GPT

Hyperparameters

```
batch_size = 32
# num of independent sequences processed in parallel

block_size
# context length

max_iters
# num of training iters

eval_interval = 500
# evaluate loss every x intervals

eval_iters = 200
# num of iters to avg loss over

learning_rate = 1e-3

n_embd
# vector embedding dimension

n_head = 2
# n_embd // head_size

n_layer = 3
dropout = 0.2
```

Less good version

```
block_size = 8
max_iters = 5000
n_embd = 32

0.042369 M parameters
train loss 2.0616, val loss 2.1201
```

```
for arknois; of sund havinngbets.
I tontwers op-freatim inwat thee y matttheris's.

Rove have, take to word hof in me mea tir:
Dull bethwit-the di!

Sid to snot coffuls, ot
Thre Sink, heard so me'et tleit onotlous viers to, will doretese
Ay: marter? foullove
loughtry thee hive,
sweable
That hand nearth at in sild anved
I the comerce thess.
by will, pawt easisters, sake notos lake.
```

Better version

```
block_size = 16
max_iters = 13000
n_embd = 64

0.158913 M parameters
train loss 1.7365, val loss 1.8890
```

```
LAUNTIO:
This ghe altreaging donen fair lath, 'lls sore have and;
Our know mine.

Firss? My have to lieves of this d0 reath, for Counsameajing.

HESS ISABERBET:
my if were most my most of to sian,
The not all noble you, with my cefordoness, when fair,
Leasiding, now, bey dentaae my than but my thou-bearn.
DUCK:
No were sume me, you kners,'lieved dear, ere of you day hearly mis
On these my with told seerk.

Give is faireful didy foold belied; and You oher: reary ow any le
```

✅ Learnt shakespeare structure (NAME: speech)!
✅ More recognisable words!
✅ Starting new lines with capital letters!

Overview

1: Get training data

Raw data: We get 2 huge walls of encoded characters (integers), one set for training, one set for validation

```python
# read and inspect
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()

# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)

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

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

# split data into train and validation
n = int(0.9 * len(data))
train_data = data[:n]
val_data = data[n:]
```

```
data.shape -> tensor([1115394])
data.dtype -> torch.int64
data[:20] # first 20 characters,
encoded -> tensor([18, 47, 56, 57, 58,
1, 15, 47, 58, 47, 64, 43, 52, 10, 0,
14, 43, 44, 53, 56, 43)]
```

Get training data

Training dataset: We get xb (inputs) and yb (targets). xb and yb each contain batches, each batch has *block_size* characters.

```
# data loading
def get_batch(split):
    # generate small batch 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+1+block_size] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y

xb, yb = get_batch('train')
```

What's happening?

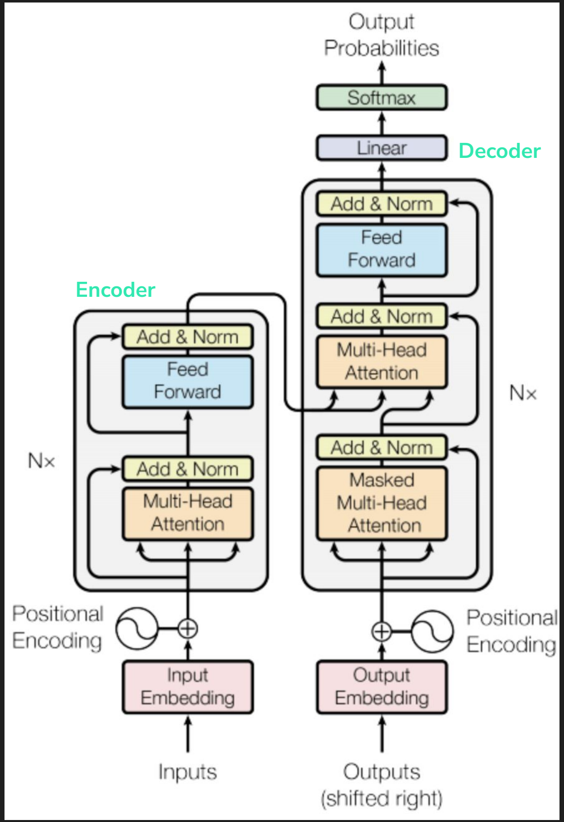
- ix: randomly index into our wall of encoded characters to form random batches with *block_size* characters
- With a batch of *X* numbers, we can form *X* training examples

Each batch can make block_size examples.
What we want to simulate through attention:

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

```
xb[:4, :] inputs:
torch.Size([32, 16])
tensor([[21, 27, 24, 13, 26, 33, 31, 10,  0, 32, 59, 57, 46,  6,  1, 58],
        [53, 59, 57,  1, 51, 43, 52,  0, 13, 56, 43,  1, 39, 58,  1, 58],
        [50,  6,  1, 57, 47, 56,  8,  1, 18, 39, 56, 43,  1, 63, 53, 59],
        [58, 46,  1, 57, 53,  6,  1, 46, 53, 50, 63,  1, 57, 47, 56, 11]])
yb[:4, :] targets
torch.Size([32, 16])
tensor([[27, 24, 13, 26, 33, 31, 10,  0, 32, 59, 57, 46,  6,  1, 58, 59],
        [59, 57,  1, 51, 43, 52,  0, 13, 56, 43,  1, 39, 58,  1, 58, 46],
        [ 6,  1, 57, 47, 56,  8,  1, 18, 39, 56, 43,  1, 63, 53, 59,  1],
        [46,  1, 57, 53,  6,  1, 46, 53, 50, 63,  1, 57, 47, 56, 11,  1]])
when input is [21] target: 27
when input is [21, 27] target: 24
when input is [21, 27, 24] target: 13
when input is [21, 27, 24, 13] target: 26
when input is [21, 27, 24, 13, 26] target: 33
when input is [21, 27, 24, 13, 26, 33] target: 31
when input is [21, 27, 24, 13, 26, 33, 31] target: 10
when input is [21, 27, 24, 13, 26, 33, 31, 10] target: 0
when input is [21, 27, 24, 13, 26, 33, 31, 10, 0] target: 32
when input is [21, 27, 24, 13, 26, 33, 31, 10, 0, 32] target: 59
when input is [21, 27, 24, 13, 26, 33, 31, 10, 0, 32, 59] target: 57
when input is [21, 27, 24, 13, 26, 33, 31, 10, 0, 32, 59, 57] target: 46
when input is [21, 27, 24, 13, 26, 33, 31, 10, 0, 32, 59, 57, 46] target: 6
```

Our Autoregressive Model



Transformer architecture from *All I Need is Attention*

What are autoregressive models?

- Autoregressive models like GPT are designed to **generate output one token at a time**, using **only past context**
- It uses only a **decoder**, which is **causal** (masked self-attention so tokens can only see past tokens)

Encoder	Looks at the full input sequence (bidirectional - use case: sentiment analysis)
Decoder	Looks only at past tokens (causal, autoregressive: use case: language modelling)

What is in a decoder?

- Many blocks which consist of:
- 1) Attention layer:
    - Allows tokens to 'look' at one another and gain information from context
    - Linear combination of input vectors (focus on rships btw tokens)
- 2) Feedforward:
    - Processes each token independently
    - Non-linear transformations to individual token

Building an attention head

Simplified version: Within 1 example, we want each token to **get information from the previous tokens** to make a prediction on the next one. One simple way is to multiply with the **average of past tokens**.

Code

```
B,T,C = 4,8,2 # batch, time, channels
x = torch.randn(B,T,C)

# v1: using matrix multiply for a weighted aggregation
wei = torch.tril(torch.ones(T, T))
wei = wei / wei.sum(1, keepdim=True)
xbow1 = wei @ x # (B, T, T) @ (B, T, C) ----> (B, T, C)

# v2: use Softmax
tril = torch.tril(torch.ones(T, T))
wei2 = torch.zeros((T,T))
wei2 = wei2.masked_fill(tril == 0, float('-inf'))
wei2 = F.softmax(wei2, dim=-1)
xbow2 = wei2 @ x

wei, wei2
```

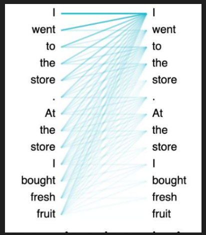
What's happening

wei = affinities, tell us how much each token from the past contributes to our average

- Masked with lower triangle bc future tokens don't contribute

Now we see that wei is all uniform, but we don't actually want it to be uniform because some tokens will find others more interesting, and we want it to be data dependent. Eg. vowel interested in consonants in the past.

```
wei, wei2
[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]]
```



Building an attention head

Proper version: Within 1 example, we want each token to **get information from the previous tokens** to make a prediction on the next one. Use **keys, queries** and **values**

Code

```
B,T,n_embd = 4,8,32
x = torch.randn(B,T,n_embd) # x (B, T, n_embd)

# v3: self-attention (s)

head_size = 16
key = nn.Linear(n_embd, head_size, bias=False)
query = nn.Linear(n_embd, head_size, bias=False)
value = nn.Linear(n_embd, head_size, bias=False)
# linear proj of n_embd → head_size (B,T,n_embd) @ (n_embd, hs)->(B,T,hs)

k = key(x)   # k (B, T, hs)
q = query(x) # q (B, T, hs)
v = value(x) # v (B, T, hs)

# compute affinities
wei = q @ k.transpose(-2, -1) * k.shape[-1]**0.5 # (B,T,hs) @ (B,hs,T) -> (B,T,T)
# for each token in the sequence, get dot pdts btw its query and each key
# wei[b, i, j] = how much token i should pay attention to token j in that batch
# Scaled attention divides wei by 1/sqrt(head_size), wei is unit variance

# apply causal mask + softmax
tril = torch.tril(torch.ones(T, T))
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)

out = wei @ v # (B,T,T)@(B,T,hs) -> (B,T,hs)

out.shape
```

What's happening

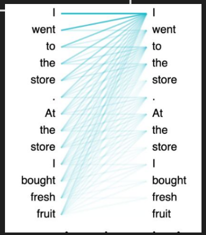
wei = affinities, tell us how much each token from the past contributes to our average → **make this non-uniform (data dependent)**

Self attention does it by having every node emit 3 vectors:

* **query** - what am i looking for
* **key** - what do i contain (acts like a label)
* **wei** - dot product of each token's query and all tokens' key to give a relevance score - how much does this key match what i'm looking for? Mask to only include past tokens
* **value** - the actual info retrieved once a you decide which tokens are relevant
* **out** - dot product of each token's wei with query and all tokens' key to give a relevance score

head_size - dimensionality of attention mechanism (q,k,v vectors)- how much detail or nuance each attention head can capture in its comparisons

```
wei[0]
[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
[0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
[0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
[0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
[0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
[0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
[0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
[0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]]
```



Building an attention head

Layer - Head

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

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False) # weight matrix (n_embd, hs)
        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_size)))

        self.dropout = nn.Dropout(dropout)

    def forward(self,x):
        B, T, C = x.shape   # x (B, T, n_embd)

        k = self.key(x)     # k (B, T, hs)
        q = self.query(x)   # q (B, T, hs)
        v = self.value(x)   # v (B, T, hs)
        # linear proj n_embd → head_size (B,T,n_embd)@(n_embd, hs)->(B,T,hs)

        # compute affinities
        wei = q @ k.transpose(-2, -1) * k.shape[-1]**0.5  # (B,T,hs) @ (B,hs,T) -> (B,T,T)
        # for each token in the sequence, get dot pdts btw its query and each key
        # wei[b, i, j] = how much token i should pay attention to token j in that batch
        # Scaled attention divides wei by 1/sqrt(head_size), wei is unit variance

        # apply causual mask + softmax
        wei = wei.masked_fill(self.tril[:T, :T] ==0, float('-inf'))
        wei = F.softmax(wei, dim=-1)

        wei = self.dropout(wei)
        out = wei @ v       # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        # each token's output is a weighted average of value vectors
        return out
```

More on Attention

Input: x (B, T, n_embd) ; Output per head: out (B, T, hs)
n_embd = hs * num_heads

Attention is a communication mechanism. Each token's output is a weighted average of all past token's value vectors (weighted by how much attention it pays to each token)

- There is no notion of space, hence need to positional encoding later

Self attention means **keys** and **values** are produced from the same source as **queries**. In **Cross-attention**, queries are from x, but the keys and values come from some other, external source (e.g. encoder module)

Decoders have **triangular masking** that causes tokens to only be able to see past tokens; used in autoregressive settings like language modelling. **Encoders** exclude this, allowing all tokens to communicate; used in sentiment analysis all tokens can talk to predict the overall sentiment.

Scaled attention additionally divides wei by 1/sqrt(head_size), which makes wei unit variance, so softmax does not saturate too much.

- Without scaled attention: k.var() = 1, q.var() = 1 → wei.var() = *hs*
- After scaling: wei.var() = 1

Dropout is a regularisation technique that randomly zeros some wei in each forward pass, prevents overfitting and forces the model to not rely too heavily on any single neuron.

- Eg. `dropout = nn.Dropout(p=0.1)` - each wei has a probability p of being zeroed out. Remaining elements scale up by 1/(1-p).

Building a transformer block

Layers - MultiHead Attention & Feedforward

```
class MultiHeadAttention(nn.Module):
    "multiple heads of self-attention in parallel"

    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(head_size * num_heads, n_embd)
        # concatenate attn outputs across multi heads &
        # & project back to n_embd to enable subsequent layers to consume result
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1) # (B, T, hs * num_heads)
        out = self.dropout(self.proj(out))                  # (B, T, n_embd)
        return out
```

```
class FeedForward(nn.Module):
    "a simple linear layer followed by non linearity"

    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)
```

What's happening

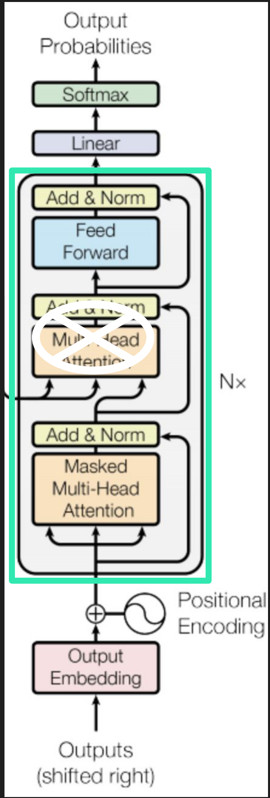
Blocks consist of:

1) Multi head Attention layer:
- Allows tokens to 'look' at one another and gain information from context
- **Linear combination** of input vectors (focus on rships btw tokens)

How it happens:
- *Inputs into each Head(x) - (B, T, n_embd)*
- *Outputs (B, T, hs) from each head are concatenated (B, T, hs * num_heads), then passed through a linear layer to project back to original shape (B, T, n_embd)*

2) Feedforward
- Processes each token independently via **non-linear transformations**

How
- *Project the embedding into a higher-dimensional space, apply a non-linearity, and then project it back. This allows the model to learn more expressive transformations.*
- *The intermediate dimension (4 * n_embd) gives the model room to mix and recombine features.*



Building a transformer block

Layers - Block (MultiHead Attention + Feedforward)

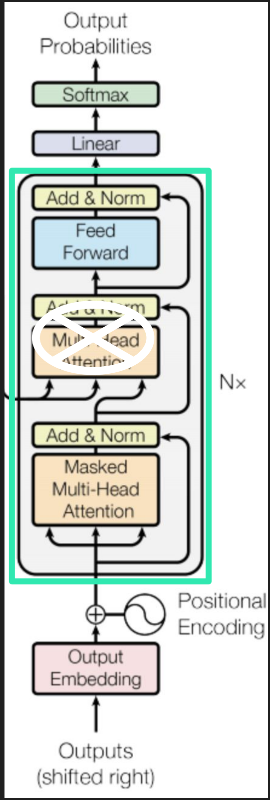
```
class Block(nn.Module):
    " Block: communication followed by computation"

    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedForward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

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

What's happening

Block consist of:
x -> LayerNorm -> Multi head Attention layer -> Residual connection -> LayerNorm -> Feedforward layer -> Residual connection

LayerNorm: Transformers use LayerNorm instead of BatchNorm
- For input x: (B, T, n_embd)
- LN works across n_embd – per-token normalization across features
- BN works across B –, per-batch normalisation. Doesn't work as transformers have variable batch sizes and even batch size = 1. BN needs multiple samples to compute statistics

Residual connection: When applying a transformation F(x) to input x, instead of `out = F(x)`, do `out = x + F(x)`. Helps with:
- **Gradient flow**: In deep networks, gradients can vanish. Skip connection allows gradient of x to flow directly to out, stabilising training
- Let model learn an **adjustment** instead of full **transformation**: Eg. if model must learn out = x
  - Without residuals, model must learn F(x) ≈ x. learn the full identity map F(x) to rebuild x from scratch
  - With residuals, out = x + F(x) → model must learn F(x)≈0. If the best thing the layer can do is "do nothing," it only needs to output zeros, or it can learn small tweaks. This is way easier than learning to rebuild x



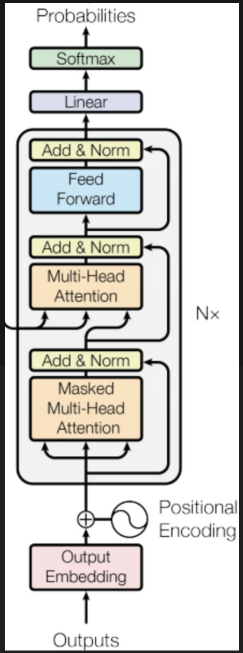
Putting it all into GPT model

GPTModel definition

```
class GPTLanguageModel(nn.Module):

    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 range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)

        # initialise weights & biases in layers
        self.apply(self._init_weights)

    # initialise linear & embedding layer: weights to small normal distrib; bias to 0
    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
```

GPTModel definition

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

    tok_emb = self.token_embedding_table(idx) # (B, T, n_embd)
    pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T, n_embd)
    x = tok_emb + pos_emb
    x = self.blocks(x)  # transformer block (B, T, n_embd)
    x = self.ln_f(x)    # layernorm (B, T, n_embd)
    logits = self.lm_head(x) # lm_head (B, T, vocab_size)

    # get loss
    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
```



Putting it all into GPT model

Code

```python
model = GPTLanguageModel()
m = model.to(device)
# print number of params
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')

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

for iter in range(max_iters):

    # every once in a while evaluate loss on train and val sets
    if iter % eval_interval == 0 or iter == max_iters-1:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")

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

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

```python
def estimate_loss():
    # average the loss for both train and val over a few iter
    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
```

```
0.158913 M parameters
step 0: train loss 4.1932, val loss 4.1919
step 500: train loss 2.2663, val loss 2.2690
step 1000: train loss 2.1404, val loss 2.1697
step 1500: train loss 2.0658, val loss 2.1007
step 2000: train loss 2.0090, val loss 2.0685
step 2500: train loss 1.9944, val loss 2.0705
step 3000: train loss 1.9626, val loss 2.0436
step 3500: train loss 1.9322, val loss 2.0284
```

What's happening

Training:

1. Set an AdamW optimiser that knows which parameters to update and at what learning rate.
2. Get a batch of data xb and yb (B, T) each
3. Evaluate loss
4. Using optimiser:
    a. Set all gradients of params to zero
    b. Use loss.backwards() to calculate gradients
    c. Use .step() to update params with gradients
5. For every `eval_interval`iterations, print average loss of training and validation sets across `eval_iters`

Building GPT model!

GPTLanguageModel definition

```
class GPTLanguageModel(nn.Module):

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in current context
        for _ in range(max_new_tokens):
            # crop idx to last block_size tokens (pos_emb only has up to block_size embeddings)
            idx_cond = idx[:, -block_size:]

            # get predictions for each time step in input
            logits, loss = self(idx_cond)
            # logits[:, 0, :] - prediction for what comes aft token 1
            # logits[:, 1, :] - prediction for what comes aft token 1, 2
            # logits[:, 2, :] - prediction for what comes aft token 1, 2, 3

            # get predictions for next token after all tokens in input
            logits = logits[:, -1, :] # (B,C)

            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1)

            # sample from 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
```

Code

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

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

- Creates a **starting input tensor** for the model.
- Shape (1, 1) → batch size of 1, sequence length of 1.
- The value is 0, which corresponds to the **first token** in your vocabulary.
-

m.generate(context, max_new_tokens=500)

- Calls the model's generate method to **predict** the next token 500 times
- It maintains the last block_size tokens as context (due to the positional embedding limits)
-

[0].tolist()

- The output of generate is a tensor of shape (1, 501) (original + 500 new tokens).
- [0] extracts the first (and only) sequence from the batch.
- .tolist() converts the tensor to a Python list of integers to then be decoded

Mini-shakespeare GPT

Hyperparameters

```
batch_size = 32
# num of independent sequences processed in parallel

block_size
# context length

max_iters
# num of training iters

eval_interval = 500
# evaluate loss every x intervals

eval_iters = 200
# num of iters to avg loss over

learning_rate = 1e-3

n_embd
# vector embedding dimension

n_head = 2
# n_embd // head_size

n_layer = 3
dropout = 0.2
```

Less good version

```
block_size = 8
max_iters = 5000
n_embd = 32

0.042369 M parameters
train loss 2.0616, val loss 2.1201
```

```
for arknois; of sund havinngbets.
I tontwers op-freatim inwat thee y matttheris's.

Rove have, take to word hof in me mea tir:
Dull bethwit-the di!

Sid to snot coffuls, ot
Thre Sink, heard so me'et tleit onotlous viers to, will doretese
Ay: marter? foullove
loughtry thee hive,
sweable
That hand nearth at in sild anved
I the comerce thess.
by will, pawt easisters, sake notos lake.
```

Better version

```
block_size = 16
max_iters = 13000
n_embd = 64

0.158913 M parameters
train loss 1.7365, val loss 1.8890
```

```
LAUNTIO:
This ghe altreaging donen fair lath, 'lls sore have and;
Our know mine.

Firss? My have to lieves of this d0 reath, for Counsameajing.

HESS ISABERBET:
my if were most my most of to sian,
The not all noble you, with my cefordoness, when fair,
Leasiding, now, bey dentaae my than but my thou-bearn.
DUCK:
No were sume me, you kners,'lieved dear, ere of you day hearly mis
On these my with told seerk.

Give is faireful didy foold belied; and You oher: reary ow any le
```

✅ Learnt shakespeare structure (NAME: speech)!
✅ More recognisable words!
✅ Starting new lines with capital letters!