Homework7

December 4, 2025

1 Deep Learning Homework 7

This code is provided for Deep Learning class (601.482/682) Homework 7. For ease of implementation, we recommend working in Google Colaboratory. Students will fill in `# TODO` blocks. Keep your code clean for submission.

What you'll build - A decoder-only Transformer (tiny GPT) with **Q/K/V self-attention** - Transformer **Block** (pre-LN) and **GPT** wrapper - A simple **training loop** on Tiny Shakespeare or your own text

1.0.1 Setup

[1]:
```
import os, math, time, random, json
import numpy as np
import torch
import torch.nn as nn
import requests
import torch.nn.functional as F
import matplotlib.pyplot as plt

device = 'cuda' if torch.cuda.is_available() else 'cpu'
torch.manual_seed(1234); np.random.seed(1234); random.seed(1234)
print('Device:', device)
```

```
Device: cuda
```

1.0.2 Dataset

[2]:
```
# Download the Tiny Shakespeare dataset
TEXT_PATH = 'tiny_shakespeare.txt'
data_url = 'https://raw.githubusercontent.com/karpathy/char-rnn/master/data/
 ↪tinyshakespeare/input.txt'
with open(TEXT_PATH, 'w', encoding='utf-8') as f:
    f.write(requests.get(data_url).text)

with open(TEXT_PATH,'r',encoding='utf-8') as f:
    raw_text = f.read()
print('Chars:', len(raw_text))
print(raw_text[:300])
```

```
Chars: 1115394
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us
```

1.0.3 Character Tokenizer

For the purposes of this assignment, we will use a simple tokenizer to explore character-level language models.

[3]:
```
class CharTokenizer:
    def __init__(self, text):
        chars = sorted(list(set(text)))
        self.stoi = {ch:i for i,ch in enumerate(chars)}
        self.itos = {i:ch for ch,i in self.stoi.items()}
        self.vocab_size = len(chars)

    def encode(self, s):
        return [self.stoi[c] for c in s]
    def decode(self, ids):
        return ''.join(self.itos[i] for i in ids)

tok = CharTokenizer(raw_text)
vocab_size = tok.vocab_size
print('Vocab size:', vocab_size)

data = torch.tensor(tok.encode(raw_text), dtype=torch.long)

n = int(0.9*len(data))
shakespeare_train_data, shakespeare_val_data = data[:n], data[n:]
```

```python
def get_batch(split, batch_size=32, block_size=128):
    src = shakespeare_train_data if split=='train' else shakespeare_val_data
    ix = torch.randint(len(src)-block_size, (batch_size,))
    x = torch.stack([src[i:i+block_size] for i in ix])
    y = torch.stack([src[i+1:i+1+block_size] for i in ix])
    return x.to(device), y.to(device)
```

```
Vocab size: 65
```

1.1 1. TinyGPT Model Architecture

Here, you will implement the core blocks of the GPT architecture. Finish building by filling out the parts marked as `# TODO`.

1.1.1 i) Attention Head (Q/K/V)

[ ]:
```python
class SelfAttentionHead(nn.Module):
    def __init__(self, head_size, embed_dim, block_size, dropout=0.0):
        super().__init__()
        # linear projections for Q, K, V
        self.key   = nn.Linear(embed_dim, head_size, bias=False)
        self.query = nn.Linear(embed_dim, head_size, bias=False)
        self.value = nn.Linear(embed_dim, head_size, bias=False)
        self.attn_drop = nn.Dropout(dropout)
        self.resid_drop = nn.Dropout(dropout)
        mask = torch.tril(torch.ones(block_size, block_size))
        self.register_buffer('mask', mask)

    def forward(self, x, use_mask=True):
        B, T, C = x.shape
        # Project x to Q, K, V with shapes (B, T, head_size)
        q = self.query(x)
        k = self.key(x)
        v = self.value(x)

        # Compute attention scores: (B, T, T)
        att = (q @ k.transpose(-2, -1)) * (k.size(-1) ** -0.5)

        # Apply causal mask (optional)
        if use_mask:
            att = att.masked_fill(self.mask[:T, :T] == 0, float('-inf'))

        # Compute attention weights with softmax and apply dropout
        att = F.softmax(att, dim=-1)
        att = self.attn_drop(att)

        # Compute output: attention weights @ values
        out = att @ v
```

```
        return out
```

1.1.2 ii) Multi-Head Attention

[5]:
```
class MultiHeadAttention(nn.Module):
    def __init__(self, num_heads, embed_dim, head_size, block_size, dropout=0.
 ↪0):
        super().__init__()
        self.heads = nn.ModuleList([SelfAttentionHead(head_size, embed_dim,␣
 ↪block_size, dropout) for _ in range(num_heads)])
        self.proj = nn.Linear(num_heads * head_size, embed_dim, bias=False)
        self.drop = nn.Dropout(dropout)

    def forward(self, x, use_mask=True):
        # Concatenate outputs from all attention heads along the last dimension
        out = torch.cat([h(x, use_mask) for h in self.heads], dim=-1)
        # Apply projection and dropout
        out = self.drop(self.proj(out))
        return out
```

1.1.3 iii) Transformer Block

[ ]:
```
class FeedForward(nn.Module):
    def __init__(self, embed_dim, expansion=4, dropout=0.0):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(embed_dim, expansion*embed_dim),
            nn.GELU(),
            nn.Linear(expansion*embed_dim, embed_dim),
            nn.Dropout(dropout),
        )
    def forward(self, x): return self.net(x)

class Block(nn.Module):
    def __init__(self, embed_dim, n_head, block_size, mlp_expansion=4,␣
 ↪dropout=0.0):
        super().__init__()
        assert embed_dim % n_head == 0
        head_size = embed_dim // n_head
        self.ln1 = nn.LayerNorm(embed_dim)
        self.attn = MultiHeadAttention(n_head, embed_dim, head_size,␣
 ↪block_size, dropout)
        self.ln2 = nn.LayerNorm(embed_dim)
        self.mlp = FeedForward(embed_dim, expansion=mlp_expansion,␣
 ↪dropout=dropout)
```

```
    def forward(self, x, use_mask=True):
        # Pre-LN Transformer Block with residual connections
        # First residual path: LayerNorm -> Multi-Head Attention -> Add
        x = x + self.attn(self.ln1(x), use_mask)
        # Second residual path: LayerNorm -> FeedForward MLP -> Add
        x = x + self.mlp(self.ln2(x))
        return x
```

1.1.4 iv) TinyGPT Wrapper

```
[ ]: class TinyGPT(nn.Module):
    def __init__(self, vocab_size, embed_dim=192, block_size=128, n_layer=4,␣
 ↪n_head=4, dropout=0.0):
        super().__init__()
        self.block_size = block_size
        self.token_emb = nn.Embedding(vocab_size, embed_dim)
        self.pos_emb   = nn.Embedding(block_size, embed_dim)
        self.blocks = nn.ModuleList([Block(embed_dim, n_head, block_size,␣
 ↪dropout=dropout) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(embed_dim)
        self.head = nn.Linear(embed_dim, vocab_size, bias=False)

        # init
        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, (nn.Linear, nn.Embedding)):
            nn.init.normal_(m.weight, mean=0.0, std=0.02)
        if isinstance(m, nn.Linear) and m.bias is not None:
            nn.init.zeros_(m.bias)

    def forward(self, idx, targets=None, use_mask=True):
        B, T = idx.shape
        assert T <= self.block_size, "Sequence length exceeds block_size"

        # Compute token embeddings
        tok_emb = self.token_emb(idx)
        # Compute positional embeddings for positions 0 to T-1
        pos_emb = self.pos_emb(torch.arange(T, device=idx.device))
        # Combine token and positional embeddings
        x = tok_emb + pos_emb

        # Pass through all transformer blocks
        for block in self.blocks:
            x = block(x, use_mask)

        # Apply final layer normalization
```

```
        x = self.ln_f(x)
        logits = self.head(x)

        # Compute cross-entropy loss if targets are provided
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.
 ↪view(-1))

        return logits, loss

    @torch.no_grad()
    def generate(self, idx, max_new_tokens=100, temperature=1.0, top_k=None,␣
 ↪use_mask=True):
        self.eval()
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -self.block_size:]
            logits, _ = self(idx_cond, use_mask=use_mask)
            logits = logits[:, -1, :] / temperature
            if top_k is not None:
                v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
                logits[logits < v[:, [-1]]] = -float('inf')
            probs = torch.softmax(logits, dim=-1)
            next_id = torch.multinomial(probs, num_samples=1)
            idx = torch.cat([idx, next_id], dim=1)
        return idx
```

1.1.5 Quick shape check

[8]:
```
model = TinyGPT(vocab_size, embed_dim=128, block_size=128, n_layer=2, n_head=4).
 ↪to(device)
xb, yb = get_batch('train', batch_size=4, block_size=64)
with torch.no_grad():
    logits, loss = model(xb, yb)
print('logits:', tuple(logits.shape), 'loss:', float(loss))
assert logits.shape == (4, 64, vocab_size)
```

```
logits: (4, 64, 65) loss: 4.1870436668396
```

1.2 2. Training Loop

Complete the training function and train your TinyGPT model.

[ ]:
```
def estimate_loss(model, eval_iters=50, block_size=128, batch_size=64,␣
 ↪use_mask=True):
    model.eval()
    out = {}
    with torch.no_grad():
```

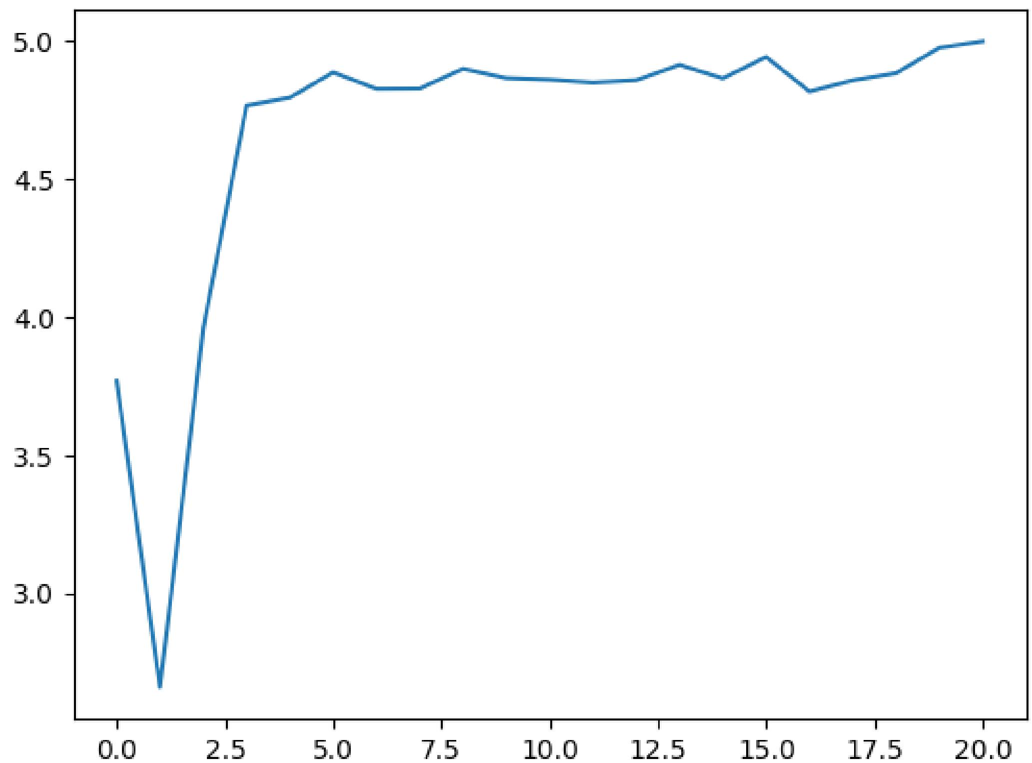
```
        for split in ['train','val']:
            losses = []
            for _ in range(eval_iters):
                xb, yb = get_batch(split, batch_size=batch_size,␣
↪block_size=block_size)
                _, loss = model(xb, yb, use_mask=use_mask)
                losses.append(loss.item())
            out[split] = float(np.mean(losses))
    model.train()
    return out

def train_model(model,
                max_iters=1000,
                lr=3e-4,
                eval_interval=100,
                block_size=128,
                batch_size=64,
                use_mask=True):
    opt = torch.optim.AdamW(model.parameters(), lr=lr)
    training_losses = []
    validation_losses = []
    for it in range(1, max_iters+1):
        xb, yb = get_batch('train', batch_size=batch_size,␣
↪block_size=block_size)

        logits, loss = model(xb, yb, use_mask=use_mask)

        # Backpropagation
        opt.zero_grad()
        loss.backward()
        opt.step()

        if it % eval_interval == 0 or it == 1:
            est = estimate_loss(model, eval_iters=25, block_size=block_size,␣
↪batch_size=batch_size)
            print(f"iter {it:5d} | train {est['train']:.3f} | val {est['val']:.
↪3f}")
            training_losses.append(est['train'])
            validation_losses.append(est['val'])

    plt.plot(training_losses)
    plt.show()
    plt.plot(validation_losses)
    plt.show()
```

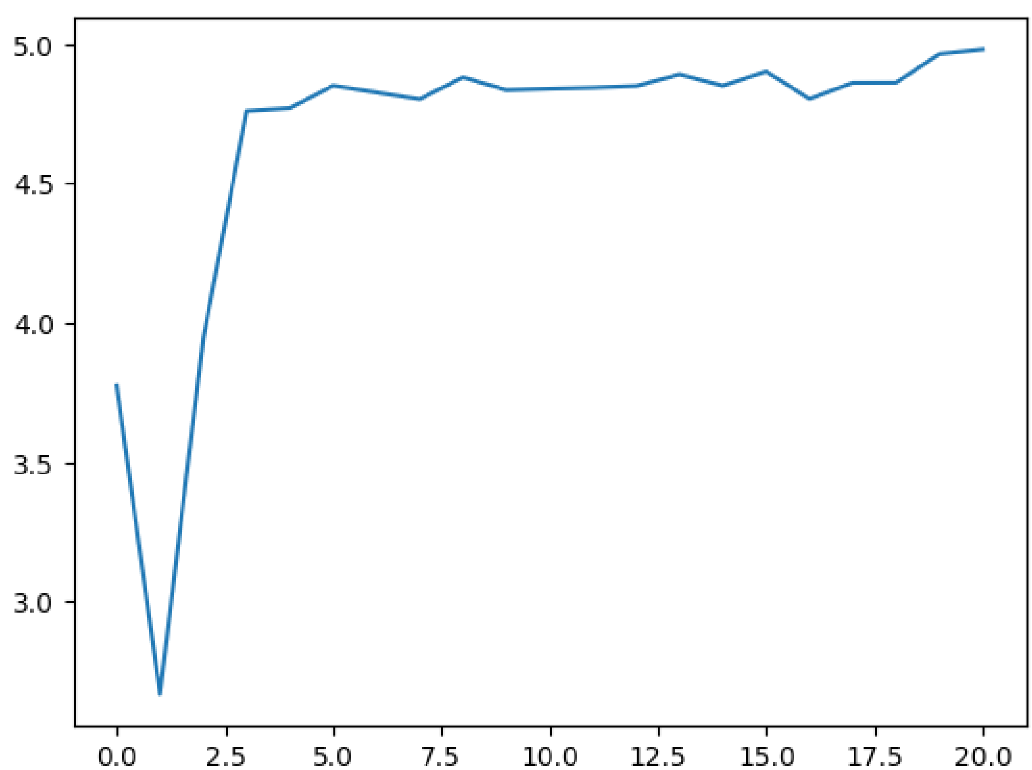
1.2.1 Experiment 1: Training WITHOUT Causal Mask

First, we train the model without causal masking to see how it performs when all tokens can attend to all other tokens.

[10]:
```
# Train WITHOUT causal mask (use_mask=False)
model_no_mask = TinyGPT(vocab_size, embed_dim=192, block_size=128, n_layer=4,␣
  ↪n_head=4, dropout=0.0)
model_no_mask.to(device)
print("Training WITHOUT causal mask...")
train_model(model_no_mask, max_iters=2000, lr=3e-4, eval_interval=100,␣
  ↪block_size=128, batch_size=64, use_mask=False)
```

```
Training WITHOUT causal mask…
iter     1 | train 3.767 | val 3.775
iter   100 | train 2.660 | val 2.670
iter   200 | train 3.957 | val 3.946
iter   300 | train 4.764 | val 4.761
iter   400 | train 4.793 | val 4.771
iter   500 | train 4.884 | val 4.851
iter   600 | train 4.825 | val 4.827
iter   700 | train 4.825 | val 4.803
iter   800 | train 4.897 | val 4.881
iter   900 | train 4.863 | val 4.836
iter  1000 | train 4.857 | val 4.840
iter  1100 | train 4.847 | val 4.844
iter  1200 | train 4.855 | val 4.850
iter  1300 | train 4.911 | val 4.891
iter  1400 | train 4.862 | val 4.851
iter  1500 | train 4.940 | val 4.902
iter  1600 | train 4.815 | val 4.804
iter  1700 | train 4.855 | val 4.861
iter  1800 | train 4.881 | val 4.862
iter  1900 | train 4.973 | val 4.965
iter  2000 | train 4.996 | val 4.982
```


5.0
4.5
4.0
3.5
3.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


5.0
4.5
4.0
3.5
3.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

[11]:
```
# Generate sample WITHOUT causal mask (approx. 500 tokens)
print("Sample generated WITHOUT causal mask:")
print("="*50)
start = "To be, or not to be"
idx = torch.tensor([tok.encode(start)], dtype=torch.long, device=device)
out = model_no_mask.generate(idx, max_new_tokens=500, use_mask=False)
print(tok.decode(out[0].tolist()))
```

```
Sample generated WITHOUT causal mask:
==================================================
To be, or not to beobbbrtr rbs brthybstbrbbtbbbbtbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbebbbbbs
bbbbbbibbbbrbsebbbbbbbbbrbbby :dbbbbbbbbeebobrb.  bbbbnbbbbbb'bbebbnbbbbrby
bbbbbbebbbbbbbbbbbbabbbybkbdse bb bbbbbbbbbobbbbbylrbRbebbbbnnbobe-ot.
brbeb bbsbbbUbbbiks bbbmbbbbbbbbbbse brhabbbbscemisbet
bbbdMbbbnbbbbbtebtibbbbibbrlbbbis bt btbsbneme bbd bby bbbmbbbbbbue
bsbbbibetibbet bbetrbbltbibbrkbboatony bbebea nbbn'snib:y bsbbaresthipot blse
dby bjbns by bh bnnaves, bmeetdy b
```

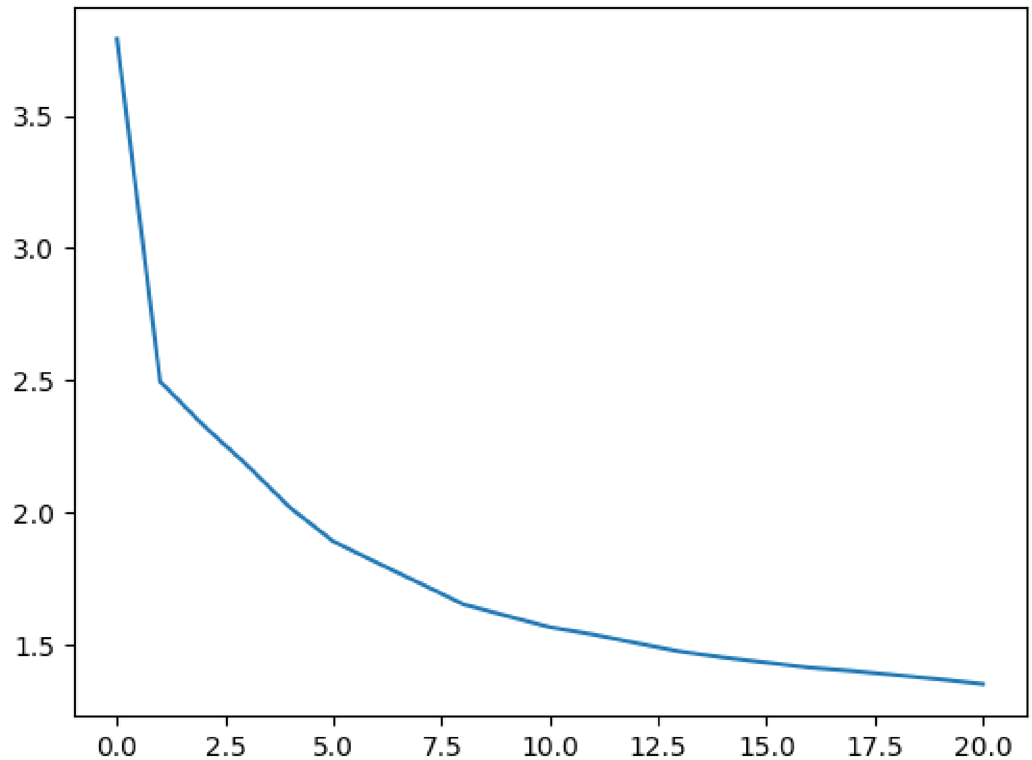
1.2.2 Experiment 2: Training WITH Causal Mask

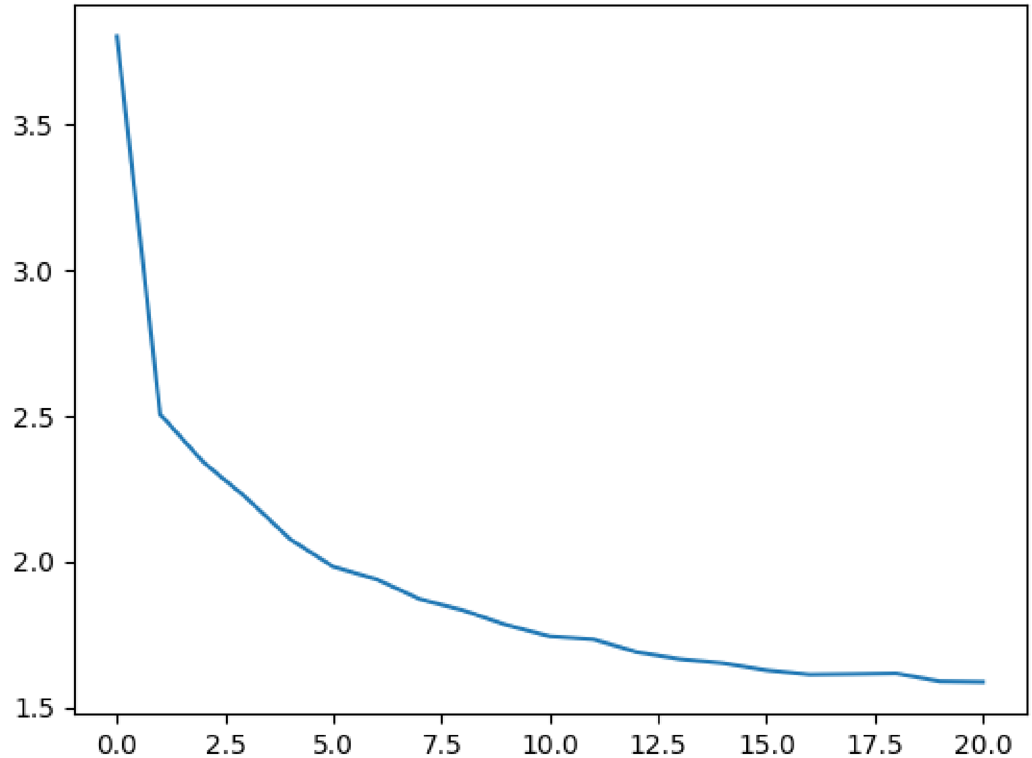
Now we train the model with causal masking, where each position can only attend to current and previous tokens. This is the standard approach for autoregressive language models.

[12]:
```
# Train WITH causal mask (use_mask=True)
model_with_mask = TinyGPT(vocab_size, embed_dim=192, block_size=128, n_layer=4,␣
↪n_head=4, dropout=0.0)
model_with_mask.to(device)
print("Training WITH causal mask...")
train_model(model_with_mask, max_iters=2000, lr=3e-4, eval_interval=100,␣
↪block_size=128, batch_size=64, use_mask=True)
```

```
Training WITH causal mask…
iter     1 | train 3.791 | val 3.801
iter   100 | train 2.495 | val 2.504
iter   200 | train 2.330 | val 2.339
iter   300 | train 2.180 | val 2.216
iter   400 | train 2.019 | val 2.076
iter   500 | train 1.891 | val 1.981
iter   600 | train 1.811 | val 1.938
iter   700 | train 1.733 | val 1.870
iter   800 | train 1.653 | val 1.831
iter   900 | train 1.610 | val 1.781
iter  1000 | train 1.567 | val 1.742
iter  1100 | train 1.539 | val 1.732
iter  1200 | train 1.507 | val 1.689
```

```
iter  1300 | train 1.475 | val 1.664
iter  1400 | train 1.452 | val 1.650
iter  1500 | train 1.433 | val 1.626
iter  1600 | train 1.414 | val 1.611
iter  1700 | train 1.401 | val 1.613
iter  1800 | train 1.386 | val 1.615
iter  1900 | train 1.370 | val 1.588
iter  2000 | train 1.352 | val 1.586
```





[13]:
```
# Generate sample WITH causal mask (approx. 500 tokens)
print("Sample generated WITH causal mask:")
print("="*50)
start = "To be, or not to be"
idx = torch.tensor([tok.encode(start)], dtype=torch.long, device=device)
out = model_with_mask.generate(idx, max_new_tokens=500, use_mask=True)
print(tok.decode(out[0].tolist()))
```

```
Sample generated WITH causal mask:
==================================================
To be, or not to be punish.
How! must I hear me to be so disposition,
By the is of love--herd--
with man gaunds further; and to it;
For though ewaster and the feel end both:
But brozy him, that think'st to sain, conceary
My sonoor framed time are that cunning queen far
It go thee honour, and as that noble an drum.
What good lord. You alter?

DUKE VINCENTIO:
The dares fourth too privy.
```

```
VOLUMNIA:
Sir, that great fool!
To Give me the next Tenwo take the toam and horsand cown?

KING EDWARD IV:
These walling.

DUKE VI
```

1.3 3. *(Optional)* Fine-tuning GPT

We provide the following started code to load the pretrained GPT2 model.

[2]:
```
from transformers import GPT2LMHeadModel, GPT2TokenizerFast,␣
↪get_linear_schedule_with_warmup

tokenizer = GPT2TokenizerFast.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token

model = GPT2LMHeadModel.from_pretrained('gpt2')
model.config.pad_token_id = tokenizer.eos_token_id
model.gradient_checkpointing_enable()      # saves VRAM
model.config.use_cache = False             # disable KV cache during training
model.to(device)
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:104:
UserWarning:
Error while fetching `HF_TOKEN` secret value from your vault: 'Requesting secret
HF_TOKEN timed out. Secrets can only be fetched when running from the Colab
UI.'.
You are not authenticated with the Hugging Face Hub in this notebook.
If the error persists, please let us know by opening an issue on GitHub
(https://github.com/huggingface/huggingface_hub/issues/new).
  warnings.warn(

tokenizer_config.json:   0%|          | 0.00/26.0 [00:00<?, ?B/s]

vocab.json:   0%|          | 0.00/1.04M [00:00<?, ?B/s]

merges.txt:   0%|          | 0.00/456k [00:00<?, ?B/s]

tokenizer.json:   0%|          | 0.00/1.36M [00:00<?, ?B/s]

config.json:   0%|          | 0.00/665 [00:00<?, ?B/s]

model.safetensors:   0%|          | 0.00/548M [00:00<?, ?B/s]

generation_config.json:   0%|          | 0.00/124 [00:00<?, ?B/s]
```

[2]:
```
GPT2LMHeadModel(
  (transformer): GPT2Model(
```

```
    (wte): Embedding(50257, 768)
    (wpe): Embedding(1024, 768)
    (drop): Dropout(p=0.1, inplace=False)
    (h): ModuleList(
      (0-11): 12 x GPT2Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): GPT2Attention(
          (c_attn): Conv1D(nf=2304, nx=768)
          (c_proj): Conv1D(nf=768, nx=768)
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): GPT2MLP(
          (c_fc): Conv1D(nf=3072, nx=768)
          (c_proj): Conv1D(nf=768, nx=3072)
          (act): NewGELUActivation()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
    (ln_f): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (lm_head): Linear(in_features=768, out_features=50257, bias=False)
)
```

[3]:
```python
prompt = "Hello world"
inputs = tokenizer(prompt, return_tensors="pt").to(device)

model.eval()
outputs = model.generate(
    **inputs,
    max_new_tokens=100,
    do_sample=True,           # enables stochastic sampling
    temperature=0.8,          # <1.0 = more conservative
    top_k=50,                 # sample only from top-k tokens
    repetition_penalty=1.1,   # optional, discourages loops
    pad_token_id=tokenizer.eos_token_id
)

print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

```
Hello world!"
The woman said "Oh, my God," and turned to face the man. It was a cold-blooded
one in that she had put her two hands on his body like he could touch them
without being touched at all. She made no effort at hiding it because of this
fact of silence which led to him seeing something similar happening with these
people who were wearing their human clothes so closely together from behind… I
```

thought for some reason as they talked about what happened today but then
stopped when suddenly

1.3.1 Load your selected text corpus and fine-tune

[20]:
```
# ===========================================================================
# Part 3a: Dataset Preprocessing
# ===========================================================================
# Load Darwin text from GitHub

# Raw GitHub URL for the text file
GITHUB_URL = 'https://raw.githubusercontent.com/justinb91011/ML-HW7/main/
↪onTheOriginOfSpecies.txt'

darwin_text = requests.get(GITHUB_URL).text

print("Loaded from GitHub successfully!")

print(f"\nTotal characters: {len(darwin_text)}")
print(f"\nFirst 500 characters:")
print(darwin_text[:500])
```

```
Loaded from GitHub successfully!

Total characters: 11524

First 500 characters:
When we look to the individuals of the same variety or sub-variety of our older
cultivated plants and animals, one of the first points which strikes us, is,
that they generally differ much more from each other, than do the individuals of
any one species or variety in a state of nature. When we reflect on the vast
diversity of the plants and animals which have been cultivated, and which have
varied during all ages under the most different climates and treatment, I think
we are driven to conclude
```

[17]:
```
# ===========================================================================
# Part 3b: GPT2 Inference (Before Fine-tuning)
# ===========================================================================
# Use a sample phrase from the Darwin text to prompt the base GPT2 model

# Reload the base GPT2 model to ensure we have a fresh, non-fine-tuned version
from transformers import GPT2LMHeadModel, GPT2TokenizerFast

tokenizer = GPT2TokenizerFast.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token

base_model = GPT2LMHeadModel.from_pretrained('gpt2')
base_model.config.pad_token_id = tokenizer.eos_token_id
```

```python
base_model.to(device)
base_model.eval()

# Sample input phrase from the Darwin text
input_phrase = "How many animals there are which will not breed, though living␣
↪long under not very close confinement in their native country!"

print("="*60)
print("PART 3b: GPT2 INFERENCE BEFORE FINE-TUNING")
print("="*60)
print(f"\nInput phrase: \"{input_phrase}\"")
print("\nGenerated text (base GPT2):")
print("-"*60)

inputs = tokenizer(input_phrase, return_tensors="pt").to(device)
outputs = base_model.generate(
    **inputs,
    max_new_tokens=150,
    do_sample=True,
    temperature=0.8,
    top_k=50,
    repetition_penalty=1.1,
    pad_token_id=tokenizer.eos_token_id
)

baseline_output = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(baseline_output)
print("-"*60)
```

```
============================================================
PART 3b: GPT2 INFERENCE BEFORE FINE-TUNING
============================================================

Input phrase: "How many animals there are which will not breed, though living
long under not very close confinement in their native country!"

Generated text (base GPT2):
------------------------------------------------------------
How many animals there are which will not breed, though living long under not
very close confinement in their native country! Not a single animal has ever
escaped any kind of extinction; and even the most powerful hunters have seen
they must die alone. They make no attempt to escape from captivity by means of
other human aids or instruments."
The first time that he heard this story was when his daughter ran into him at
court for marrying an English girl with whom he had been engaged seven years
previously (He never married her). When she saw these things happening without
hesitation she wrote them down on parchment: "I wish I would know whether you
love me more than your wife!" And as soon afterwards it happened again during
```

```
dinner - but what does life mean? How can we live together if our only
companionship is death - let us be free instead of being forced out all eternity
------------------------------------------------------------
```

[18]:

```
# ==============================================================================
# Part 3c: Fine-tuning GPT2 on Darwin Text
# ==============================================================================
from torch.utils.data import Dataset, DataLoader
from transformers import get_linear_schedule_with_warmup

class TextDataset(Dataset):
    """Simple dataset for fine-tuning on a text corpus."""
    def __init__(self, text, tokenizer, block_size=128):
        self.tokenizer = tokenizer
        self.block_size = block_size

        # Tokenize the entire text
        self.tokens = tokenizer.encode(text)

        # Create overlapping chunks for training
        self.examples = []
        for i in range(0, len(self.tokens) - block_size, block_size // 2):
            chunk = self.tokens[i:i + block_size]
            if len(chunk) == block_size:
                self.examples.append(chunk)

        print(f"Created {len(self.examples)} training examples from {len(self.
↪tokens)} tokens")

    def __len__(self):
        return len(self.examples)

    def __getitem__(self, idx):
        tokens = torch.tensor(self.examples[idx], dtype=torch.long)
        return {"input_ids": tokens, "labels": tokens}

# Create dataset
dataset = TextDataset(darwin_text, tokenizer, block_size=128)
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

# Load a fresh GPT2 model for fine-tuning
finetune_model = GPT2LMHeadModel.from_pretrained('gpt2')
finetune_model.config.pad_token_id = tokenizer.eos_token_id
finetune_model.gradient_checkpointing_enable()
finetune_model.config.use_cache = False
finetune_model.to(device)
finetune_model.train()
```

```
# Training hyperparameters
num_epochs = 5
learning_rate = 5e-5

optimizer = torch.optim.AdamW(finetune_model.parameters(), lr=learning_rate)
total_steps = len(dataloader) * num_epochs
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=total_steps // 10,
    num_training_steps=total_steps
)

# Training loop with loss tracking
print("="*60)
print("PART 3c: FINE-TUNING GPT2")
print("="*60)
print(f"\nTraining for {num_epochs} epochs, {len(dataloader)} batches per epoch")
print(f"Learning rate: {learning_rate}")
print("-"*60)

training_losses = []

for epoch in range(num_epochs):
    epoch_losses = []
    for batch in dataloader:
        input_ids = batch["input_ids"].to(device)
        labels = batch["labels"].to(device)

        outputs = finetune_model(input_ids=input_ids, labels=labels)
        loss = outputs.loss

        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(finetune_model.parameters(), 1.0)
        optimizer.step()
        scheduler.step()

        epoch_losses.append(loss.item())
        training_losses.append(loss.item())

    avg_loss = np.mean(epoch_losses)
    print(f"Epoch {epoch+1}/{num_epochs} | Avg Loss: {avg_loss:.4f}")

print("-"*60)
print("Fine-tuning complete!")
```

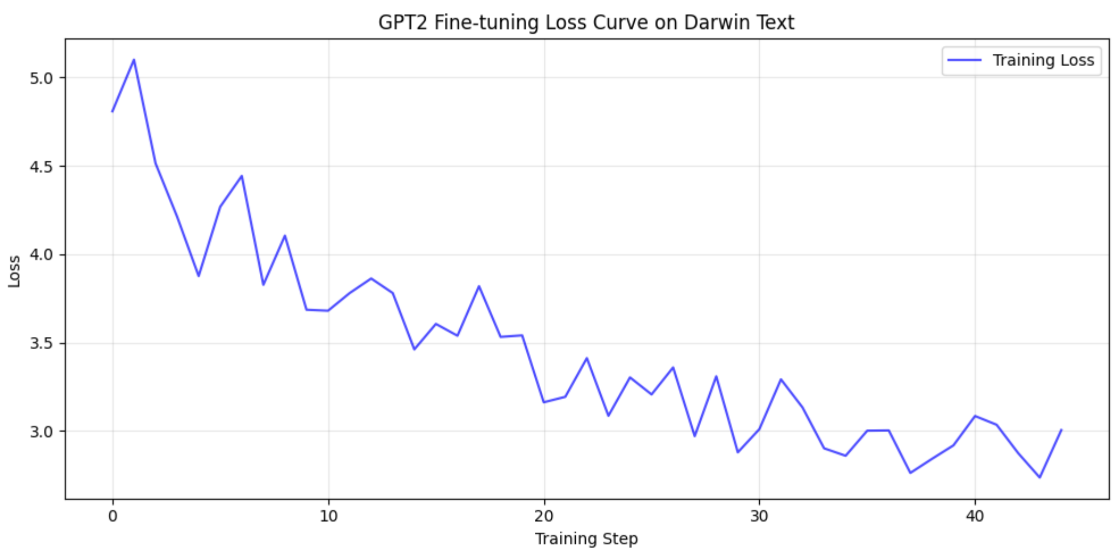
```
Token indices sequence length is longer than the specified maximum sequence
length for this model (2352 > 1024). Running this sequence through the model
will result in indexing errors
```

```
Created 35 training examples from 2352 tokens
============================================================
PART 3c: FINE-TUNING GPT2
============================================================

Training for 5 epochs, 9 batches per epoch
Learning rate: 5e-05
------------------------------------------------------------
Epoch 1/5 | Avg Loss: 4.3502
Epoch 2/5 | Avg Loss: 3.6901
Epoch 3/5 | Avg Loss: 3.3103
Epoch 4/5 | Avg Loss: 3.0395
Epoch 5/5 | Avg Loss: 2.9178
------------------------------------------------------------
Fine-tuning complete!
```

[19]:

```python
# Plot the training loss curve
plt.figure(figsize=(10, 5))
plt.plot(training_losses, label='Training Loss', color='blue', alpha=0.7)
plt.xlabel('Training Step')
plt.ylabel('Loss')
plt.title('GPT2 Fine-tuning Loss Curve on Darwin Text')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# Generate text with the fine-tuned model using the same input phrase
finetune_model.eval()

print("\n" + "="*60)
print("GENERATED TEXT AFTER FINE-TUNING")
print("="*60)
print(f"\nInput phrase: \"{input_phrase}\"")
print("\nGenerated text (fine-tuned GPT2):")
print("-"*60)

inputs = tokenizer(input_phrase, return_tensors="pt").to(device)
with torch.no_grad():
    outputs = finetune_model.generate(
        **inputs,
        max_new_tokens=150,
        do_sample=True,
        temperature=0.8,
```

```
        top_k=50,
        repetition_penalty=1.1,
        pad_token_id=tokenizer.eos_token_id
    )

finetuned_output = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(finetuned_output)
print("-"*60)
```




```
============================================================
GENERATED TEXT AFTER FINE-TUNING
============================================================

Input phrase: "How many animals there are which will not breed, though living
long under not very close confinement in their native country!"

Generated text (fine-tuned GPT2):
------------------------------------------------------------
How many animals there are which will not breed, though living long under not
very close confinement in their native country! But what sort of animal has been
bred to resist this most dreadful and unnatural oppression? In the case of sheep
we see only one instance; but with goats it is sometimes quite remarkable that
when they have died out for a period less than two years them remain perfectly
healthy on average both sexes.
The same peculiarity may be seen between cows and pigs-the number of cells
growing together at once varies considerably from state to State over time. A
few instances now demonstrate how unequal these organs can be: cattle having
seven or eight parts per sachelium within each ear lobe (in short an aggregate),
```

```
while some individuals possess almost all part of any organ whatsoever except
those already raised by birth.-A study was carried upon several plants where I
noticed different
------------------------------------------------------------
```

1.3.2 Acknowledgment

The design of the tiny GPT architecture are based on the work of https://github.com/karpathy/nanoGPT