Embedding

July 2, 2024

[ ]:
```
import tiktoken
import torch
import os
import re
import urllib.request
import numpy as np
```

0.1 Tokenization

[ ]:
```
with open("the-verdict.txt", 'r') as f:
    raw_text = f.read()
print(f"Total number of characters : {len(raw_text)}")
print(raw_text[:99])
```

```
Total number of characters : 20479
I HAD always thought Jack Gisburn rather a cheap genius--though a good fellow
enough--so it was no
```

[ ]:
```
preprocessed = re.split('([,.:;?_!"()\']|--|\s)', raw)
preprocessed = [item.strip() for item in preprocessed if item.strip()]

all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
```

[ ]:
```
vocab_size = len(all_tokens)

vocab = {token:integer for integer,token in enumerate(all_tokens)}

for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

```
('younger', 1127)
('your', 1128)
('yourself', 1129)
('<|endoftext|>', 1130)
('<|unk|>', 1131)
```

[ ]:
```
class SimpleTokenizer_V1:

    def __init__(self,vocab):

        self.str_to_int  = vocab
        self.int_to_str  = {i:s for s,i in vocab.items()}

    def encode(self,text):

        preprocessed = re.split('([,.:;?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if item.strip()]
        preprocessed = [
            item if item in self.str_to_int else "<|unk|>" for item in␣
 ↪preprocessed
            ]

        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):

        text = " ".join([self.int_to_str[i] for i in ids])
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
        return text
```

[ ]:
```
token = SimpleTokenizer_V1(vocab)
token.decode(token.encode("I HAD always thought Jack Gisburn rather a cheap␣
 ↪genius--though a good fellow enough--so it was no Vishal"))
```

[ ]:
```
'I HAD always thought Jack Gisburn rather a cheap genius -- though a good fellow
enough -- so it was no <|unk|>'
```

[ ]:
```
# Using BPE Tokenizer
tokenizer = tiktoken.get_encoding("gpt2")

text = (
    "Hello, do you like tea? <|endoftext|> In the sunlit terraces"
     "of someunknownPlace."
)

integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})

print(integers)

tokenizer.decode(integers)
```

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250, 8812,
```

```
2114, 1659, 617, 34680, 27271, 13]
```

[ ]:
```
'Hello, do you like tea? <|endoftext|> In the sunlit terracesof
someunknownPlace.'
```

0.2 Data Sampling

[ ]:
```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

```
5145
```

[ ]:
```
from torch.utils.data import Dataset,DataLoader

class GPTDataset(Dataset):
    def __init__(self,text,tokenizer, context_size, stride):
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(text, allowed_special={"<|endoftext|>"})

        for i in range(0,len(token_ids)-context_size, stride):
            self.input_ids.append(torch.tensor(token_ids[i:i+context_size]))
            self.target_ids.append(torch.tensor(token_ids[i+1:
 ↪i+context_size+1]))

    def __len__(self):
        return len(self.input_ids)
    def __getitem__(self, idx):
        return self.input_ids[idx], self.target_ids[idx]
```

[ ]:
```
def create_dataloader(text,
                      context_size = 256,
                      stride = 128,
                      batch_size = 4,
                      shuffle = True,
                      drop_last = True,
                      num_workers = 0):

    tokenizer = tiktoken.get_encoding("gpt2")

    dataset = GPTDataset(text = text, tokenizer= tokenizer, context_size=␣
 ↪context_size, stride = stride)
```

```
    dataloader = DataLoader(
        dataset=dataset,
        shuffle=shuffle,
        batch_size=batch_size,
        drop_last=drop_last,
        num_workers=num_workers
    )

    return dataloader
```

[ ]:
```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader(
    raw_text, batch_size=1, context_size=4, stride=1, shuffle=False
)

data = iter(dataloader)
input_ids, target_ids = next(data)
print("Input IDs :\n", input_ids)
print("\nTarget IDs :\n", target_ids)
```

```
Input IDs :
 tensor([[  40,  367, 2885, 1464]])

Target IDs :
 tensor([[ 367, 2885, 1464, 1807]])
```

0.3 Token Embeddings

[ ]:
```
vocab_size = 50257 # BPE Tokenizer has 50257 vocabs
embedding_dim = 256
context_size = 4
position = torch.arange(context_size)
torch.manual_seed(123)

embedding_layer = torch.nn.Embedding(vocab_size, embedding_dim)
positional_encode_layer = torch.nn.Embedding(context_size, embedding_dim)

with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader(
    raw_text, batch_size=8, context_size=4, stride=4, shuffle=False
)

data = iter(dataloader)
```

```
input_ids, target_ids = next(data)
print(input_ids)

token_position_embed = embedding_layer(input_ids) +␣
 ↪positional_encode_layer(position)
token_position_embed.shape
```

```
tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]])
```

[ ]:
```
torch.Size([8, 4, 256])
```

AttentionHead

July 1, 2024

[ ]:
```
import torch
from torch import nn

inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

[ ]:
```
#query = inputs[1]
d_in = inputs.shape[1]
d_out = 2
```

0.1 Attention Weights

[ ]:
```
torch.manual_seed(123)
weights_q = torch.nn.Parameter(torch.rand(d_in,d_out), requires_grad=False)
weights_k = torch.nn.Parameter(torch.rand(d_in,d_out), requires_grad=False)
weights_v = torch.nn.Parameter(torch.rand(d_in,d_out), requires_grad=False)
```

[ ]:
```
query = inputs @ weights_q
keys = inputs @ weights_k
value = inputs @ weights_v
```

[ ]:
```
attention_score = query @ keys.T
attention_weights = torch.softmax(attention_score, dim=-1)
attention_weights
```

[ ]:
```
tensor([[0.1484, 0.2285, 0.2217, 0.1301, 0.0883, 0.1831],
        [0.1401, 0.2507, 0.2406, 0.1157, 0.0687, 0.1842],
        [0.1406, 0.2496, 0.2397, 0.1164, 0.0696, 0.1841],
        [0.1548, 0.2130, 0.2083, 0.1394, 0.1047, 0.1799],
        [0.1577, 0.2067, 0.2028, 0.1428, 0.1122, 0.1777],
        [0.1494, 0.2267, 0.2202, 0.1310, 0.0901, 0.1825]])
```

[ ]:
```
context_vector = attention_weights @ value
context_vector
```

[ ]:
```
tensor([[0.3071, 0.8230],
        [0.3157, 0.8430],
        [0.3152, 0.8421],
        [0.3006, 0.8080],
        [0.2978, 0.8016],
        [0.3063, 0.8214]])
```

[ ]:
```
nn.Linear(d_in,d_out, bias=False)
```

[ ]:
```
Linear(in_features=3, out_features=2, bias=False)
```

0.2 Self Attention

[ ]:
```
class SelfAttention(nn.Module):

    def __init__(self,d_in,d_out,qkv_bias=False):
        super().__init__()
        self.weights_Q = nn.Linear(d_in,d_out, bias=qkv_bias)
        self.weights_K = nn.Linear(d_in,d_out, bias=qkv_bias)
        self.weights_V = nn.Linear(d_in,d_out, bias=qkv_bias)

    def forward(self,x):
        query = self.weights_Q(x)
        key = self.weights_K(x)
        value = self.weights_V(x)

        attention_score = query @ key.T
        attention_weight = torch.softmax(attention_score/ (key.shape[1] ** 0.
 ↪5), dim=-1)

        context_vector = attention_weight @ value

        return context_vector

torch.manual_seed(123)
sa = SelfAttention(3,2)
sa(inputs)
```

[ ]:
```
tensor([[-0.5337, -0.1051],
        [-0.5323, -0.1080],
        [-0.5323, -0.1079],
        [-0.5297, -0.1076],
        [-0.5311, -0.1066],
        [-0.5299, -0.1081]], grad_fn=<MmBackward0>)
```

[ ]:
```
mask = torch.triu(torch.ones([6,6]), diagonal=1)
print(torch.rand([6,6]).masked_fill(mask.bool(), -torch.inf))
mask.bool()[:4][:4]
```

```
tensor([[0.3821,   -inf,   -inf,   -inf,   -inf,   -inf],
        [0.2745, 0.6584,   -inf,   -inf,   -inf,   -inf],
        [0.9268, 0.7388, 0.7179,   -inf,   -inf,   -inf],
        [0.0772, 0.3565, 0.1479, 0.5331,   -inf,   -inf],
        [0.4545, 0.9737, 0.4606, 0.5159, 0.4220,   -inf],
        [0.9455, 0.8057, 0.6775, 0.6087, 0.6179, 0.6932]])
```

[ ]:
```
tensor([[False,  True,  True,  True,  True,  True],
        [False, False,  True,  True,  True,  True],
        [False, False, False,  True,  True,  True],
        [False, False, False, False,  True,  True]])
```

0.3 Causal Attention (Masked-Self Attention)

[ ]:
```
class CausalAttention(nn.Module):
    def __init__(self,d_in, d_out, context_length, dropout, qkv_bias=False):
        super().__init__()
        self.weights_Q = nn.Linear(d_in,d_out, bias = qkv_bias)
        self.weights_K = nn.Linear(d_in,d_out, bias = qkv_bias)
        self.weights_V = nn.Linear(d_in,d_out, bias = qkv_bias)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer("mask", torch.triu(torch.
 ↪ones(context_length,context_length), diagonal=1))

    def forward(self,x):
        batch_size, context_length, emb_dim = x.shape
        query = self.weights_Q(x)
        key = self.weights_K(x)
        value = self.weights_V(x)

        attention_score = query @ key.transpose(1,2)
        attention_score = attention_score.masked_fill(mask.bool()[:
 ↪context_length,:context_length], -torch.inf)
        attention_weights = torch.softmax(attention_score/ key.shape[1] ** 2,␣
 ↪dim = -1)

        attention_weights = self.dropout(attention_weights)
        context_vector = attention_weights @ value

        return context_vector

torch.manual_seed(123)
inputs = torch.rand([2,4,3])
```

```
context_length = inputs.shape[1]
ca = CausalAttention(d_in = 3,d_out = 2, context_length=context_length, dropout␣
 ↪= 0.0)
ca(inputs)
```

[ ]:
```
tensor([[[-0.3325, -0.1223],
         [-0.5163, -0.1861],
         [-0.3971, -0.1450],
         [-0.4687, -0.1633]],

        [[-0.1982, -0.1163],
         [-0.3748, -0.1848],
         [-0.4641, -0.2301],
         [-0.5462, -0.2600]]], grad_fn=<UnsafeViewBackward0>)
```

0.4 MultiHead Attention Wrapper

[ ]:
```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in,d_out, context_length, dropout, num_heads, qkv_bias␣
 ↪= False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(d_in, d_out, context_length, dropout,␣
 ↪qkv_bias=qkv_bias) for _ in range(num_heads)]
        )

    def forward(self,x):
        return torch.cat([head(x) for head in self.heads], dim = -1)

torch.manual_seed(123)
inputs = torch.rand([2,6,3])
context_length = inputs.shape[1]
mha = MultiHeadAttentionWrapper(d_in = 3,d_out = 2,␣
 ↪context_length=context_length, dropout = 0.0, num_heads = 2)
context_vector = mha(inputs)
print(context_vector)
print(context_vector.shape)
```

```
tensor([[[ 0.3863,  0.1636, -0.2143,  0.4306],
         [ 0.4723,  0.2268, -0.4003,  0.4802],
         [ 0.3628,  0.1752, -0.3094,  0.3714],
         [ 0.4285,  0.2032, -0.3585,  0.4283],
         [ 0.3775,  0.1839, -0.3268,  0.3890],
         [ 0.3842,  0.1951, -0.3605,  0.4010]],

        [[ 0.6047,  0.3282, -0.5738,  0.7392],
         [ 0.6680,  0.3543, -0.6236,  0.7846],
```

```
         [ 0.5888,  0.2982, -0.4944,  0.6919],
         [ 0.6332,  0.2833, -0.4200,  0.6942],
         [ 0.6835,  0.3021, -0.4513,  0.7311],
         [ 0.6988,  0.3034, -0.4423,  0.7439]]], grad_fn=<CatBackward0>)
torch.Size([2, 6, 4])
```

0.5 MultiHead Atttention

[ ]:
```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads,␣
 ↪qkv_bias = False):
        super().__init__()
        assert (d_out % num_heads == 0), "d_out must be divisible by num_heads"

        self.num_heads = num_heads
        self.head_dim = d_out // num_heads
        self.d_out = d_out

        self.weights_Q = nn.Linear(d_in,d_out, bias=qkv_bias)
        self.weights_K = nn.Linear(d_in,d_out, bias=qkv_bias)
        self.weights_V = nn.Linear(d_in,d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out,d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer("mask", torch.triu(torch.ones(context_length,␣
 ↪context_length),diagonal=1))

    def forward(self,x):
        batch_size, context_length, emb_dim = x.shape

        query = self.weights_Q(x)
        key = self.weights_K(x)
        value = self.weights_V(x)

        query = query.view(batch_size, context_length, self.num_heads, self.
 ↪head_dim)
        key = key.view(batch_size, context_length, self.num_heads, self.
 ↪head_dim)
        value = value.view(batch_size, context_length, self.num_heads, self.
 ↪head_dim)

        query = query.transpose(1,2)
        key = key.transpose(1,2)
        value = value.transpose(1,2)

        attention_score = query @ key.transpose(2,3)
```

```
        attention_score.masked_fill(self.mask.bool()[:context_length, :
 ↪context_length], -torch.inf)

        attention_weight = torch.softmax(attention_score/key.shape[-1]**0.5,␣
 ↪dim=-1)
        self.dropout(attention_weight)

        context_vector = (attention_weight @ value).transpose(1,2)

        context_vector = context_vector.contiguous().view(batch_size,␣
 ↪context_length, self.d_out)
        context_vector = self.out_proj(context_vector)

        return context_vector


torch.manual_seed(123)

inputs = torch.rand([2,4,3])
context_length = inputs.shape[1]
mha = MultiHeadAttention(d_in=3, d_out=4, num_heads=2, dropout=0.0,␣
 ↪context_length=context_length)
mha(inputs)
```

[ ]:
```
tensor([[[ 0.4856, -0.5293, -0.3424,  0.0471],
         [ 0.4893, -0.5318, -0.3472,  0.0396],
         [ 0.4861, -0.5299, -0.3431,  0.0458],
         [ 0.4882, -0.5310, -0.3457,  0.0420]],

        [[ 0.6483, -0.5114, -0.4756, -0.0350],
         [ 0.6530, -0.5127, -0.4810, -0.0411],
         [ 0.6525, -0.5113, -0.4799, -0.0383],
         [ 0.6543, -0.5116, -0.4819, -0.0403]]], grad_fn=<ViewBackward0>)
```

[ ]:

[ ]:

TransformerBlock

July 1, 2024

[ ]:
```
import matplotlib
import tiktoken
import torch
import torch.nn as nn
from code_1 import MultiHeadAttention
```

[ ]:
```
GPT_CONFIG_124M = {
    "vocab_size": 50257,    # Vocabulary size
    "context_length": 1024, # Context length
    "emb_dim": 768,         # Embedding dimension
    "n_heads": 12,          # Number of attention heads
    "n_layers": 12,         # Number of layers
    "drop_rate": 0.1,       # Dropout rate
    "qkv_bias": False       # Query-Key-Value bias
}

txt1 = "Every effort moves you"
txt2 = "Every day holds a"

inputs = []

tokenizer = tiktoken.get_encoding("gpt2")

inputs.append(torch.tensor(tokenizer.encode(txt1)))
inputs.append(torch.tensor(tokenizer.encode(txt2)))

inputs = torch.stack(inputs, dim=0)
print(inputs)
```

```
tensor([[6109, 3626, 6100,  345],
        [6109, 1110, 6622,  257]])
```

0.1 Layer Norm

[ ]:
```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim, eps=1e-5):
        super().__init__()
        self.eps = eps
```

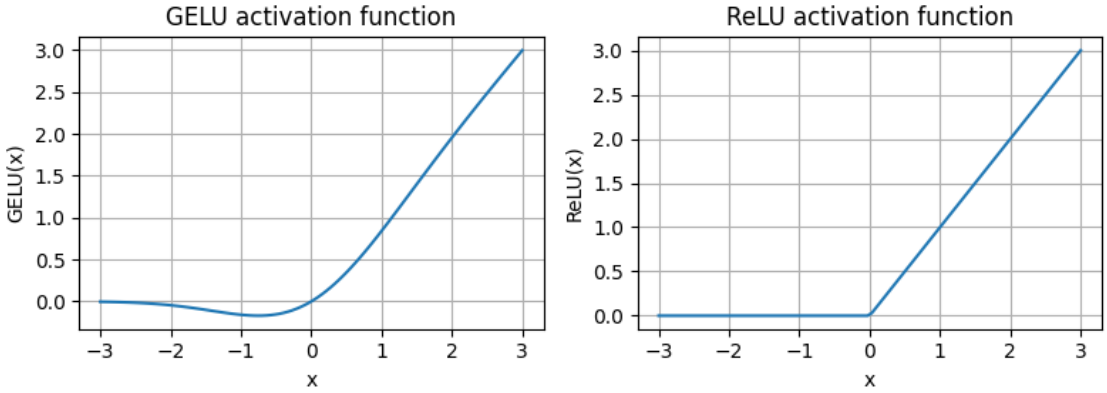
```
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim = -1, keepdim = True)
        var = x.var(dim = -1, keepdim = True, unbiased=False)

        norm_x = (x-mean) / torch.sqrt(var + self.eps)
        return  self.scale * norm_x + self.shift

torch.manual_seed(123)

batch_example = torch.randn(2, 5)

ln = LayerNorm(emb_dim=5)
out_ln = ln(batch_example)
print(out_ln)
print(out_ln.mean(), out_ln.var())
```

```
tensor([[ 0.5528,  1.0693, -0.0223,  0.2656, -1.8654],
        [ 0.9087, -1.3767, -0.9564,  1.1304,  0.2940]], grad_fn=<AddBackward0>)
tensor(-4.7684e-08, grad_fn=<MeanBackward0>) tensor(1.1111,
grad_fn=<VarBackward0>)
```

0.2 GELU & FeedForward

[ ]:
```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))

import matplotlib.pyplot as plt

gelu, relu = GELU(), nn.ReLU()

# Some sample data
x = torch.linspace(-3, 3, 100)
y_gelu, y_relu = gelu(x), relu(x)

plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"]), 1):
    plt.subplot(1, 2, i)
```

```
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)

plt.tight_layout()
plt.show()
```



[ ]:
```
class FeedForward(nn.Module):
    def __init__(self,cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"],4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"],cfg["emb_dim"]),
        )

    def forward(self,x):
        return self.layers(x)
```

0.3 Shortcut Connection

[ ]:
```
class ShortcutConnection(nn.Module):
    def __init__(self,layer_size, use_shortcut = False):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layer_size = layer_size
        self.layers = nn.ModuleList([
            nn.Sequential(nn.Linear(layer_size[0], layer_size[1]), GELU()),
            nn.Sequential(nn.Linear(layer_size[1], layer_size[2]), GELU()),
            nn.Sequential(nn.Linear(layer_size[2], layer_size[3]), GELU()),
```

```
            nn.Sequential(nn.Linear(layer_size[3], layer_size[4]), GELU()),
            nn.Sequential(nn.Linear(layer_size[4], layer_size[5]), GELU())
        ])

    def forward(self,x):
        for layer in self.layers:
            layer_out = layer(x)
            if self.use_shortcut == True and x.shape == layer_out.shape:
                x = x + layer_out
            else:
                x = layer_out

        return x

def print_gradients(model, x):
    output = model(x)
    target = torch.tensor([[0.]])

    loss = nn.MSELoss()
    loss = loss(output, target)

    loss.backward()

    for name, param in model.named_parameters():
        if 'weight' in name:
            # Print the mean absolute gradient of the weights
            print(f"{name} has gradient mean of {param.grad.abs().mean().
 ↪item()}")

layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])

torch.manual_seed(123)
model_without_shortcut =  ShortcutConnection(
    layer_sizes, use_shortcut=False
)
torch.manual_seed(123)
model_with_shortcut = ShortcutConnection(
    layer_sizes, use_shortcut=True
)
print_gradients(model_without_shortcut, sample_input)
print('-'* 70)
print_gradients(model_with_shortcut, sample_input)
```

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.00012011159560643137
layers.2.0.weight has gradient mean of 0.0007152039906941354
```

```
layers.3.0.weight has gradient mean of 0.0013988736318424344
layers.4.0.weight has gradient mean of 0.005049645435065031
----------------------------------------------------------------------
layers.0.0.weight has gradient mean of 0.22169792652130127
layers.1.0.weight has gradient mean of 0.20694106817245483
layers.2.0.weight has gradient mean of 0.32896995544433594
layers.3.0.weight has gradient mean of 0.2665732204914093
layers.4.0.weight has gradient mean of 1.3258540630340576
```

0.4 Transformer Block with MultiHead Attention, Layer Norm, Shortcut Connection, Feed Forward

[ ]:
```python
class TransformerBlock(nn.Module):
    def __init__(self,cfg):
        super().__init__()
        self.attention = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            dropout=cfg["drop_rate"],
            num_heads=cfg["n_heads"],
            qkv_bias=cfg['qkv_bias'])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg['emb_dim'])
        self.norm2 = LayerNorm(cfg['emb_dim'])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.attention(x)
        x = self.drop_shortcut(x)
        x = x + shortcut

        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut

        return x
```

0.5 GPT-2 (124M) Small Model

[ ]:
```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()

        self.tok_emb = nn.Embedding(cfg['vocab_size'], cfg['emb_dim'])
        self.pos_emb = nn.Embedding(cfg['context_length'], cfg['emb_dim'])
        self.drop_emb = nn.Dropout(cfg['drop_rate'])

        self.transformer_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg['n_layers'])]
        )

        self.final_norm = LayerNorm(cfg['emb_dim'])
        self.out_head = nn.Linear(cfg['emb_dim'], cfg['vocab_size'], bias=False)

    def forward(self, x_indexes):

        batch_size, context_length = x_indexes.shape

        tok_embed = self.tok_emb(x_indexes)
        pos_embed = self.pos_emb(torch.arange(context_length, device=x_indexes.
 ↪device))

        x = tok_embed + pos_embed
        x = self.drop_emb(x)
        x = self.transformer_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)

        return logits

torch.manual_seed(123)

gpt = GPTModel(GPT_CONFIG_124M)
logits = gpt(inputs)
print(logits)
print(logits.shape)
```

```
tensor([[[ 0.3408, -0.0490, -0.2705,  …,  0.3432,  0.1251, -0.2388],
         [ 0.3638, -0.7188, -0.7083,  …, -0.3816,  0.1813, -0.2606],
         [ 1.0497,  0.1511, -0.2826,  …, -0.0685, -0.5515, -0.1953],
         [-0.9375,  0.5745, -0.2970,  …,  0.6244,  0.3248,  0.0130]],

        [[-0.4044, -0.1799,  0.0392,  …,  0.2117,  0.1037, -0.3719],
         [ 0.2887,  0.3760, -0.0746,  …,  0.7338, -0.1642,  0.3497],
         [ 1.2424,  0.8104, -0.2517,  …,  0.8155,  0.1034, -0.2240],
```

```
         [ 0.0723,  0.5218,  0.3266,  …,  1.0810, -0.3975,  0.0527]]],
       grad_fn=<UnsafeViewBackward0>)
torch.Size([2, 4, 50257])
```

[ ]:
```python
print(gpt.tok_emb.weight.shape == gpt.out_head.weight.shape)

total_params = sum(p.numel() for p in gpt.parameters())
print(f"Total number of parameters: {total_params:,}")

total_params_gpt2 =  total_params - sum(p.numel() for p in gpt.out_head.
 ↪parameters())
print(f"Number of trainable parameters considering weight tying:␣
 ↪{total_params_gpt2:,}")

# Calculate the total size in bytes (assuming float32, 4 bytes per parameter)
total_size_bytes = total_params * 4
# Convert to megabytes
total_size_mb = total_size_bytes / (1024 * 1024)
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

```
True
Total number of parameters: 163,009,536
Number of trainable parameters considering weight tying: 124,412,160
Total size of the model: 621.83 MB
```

0.6 Generating Text (Inference)

[ ]:
```python
gpt.eval()

def generate_text(model, idx, max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx = idx[:, -context_size:]
        logits = model(idx)
        logit = logits[:, -1, :]
        prob = torch.softmax(logit, dim = -1)
        next_idx = torch.argmax(prob, dim=-1, keepdim=True)
        idx = torch.cat((idx, next_idx), dim=-1)
    return idx


text = "How are you"

encoded = tokenizer.encode(text)
print("encoded:", encoded)

encoded_tensor = torch.tensor(encoded).unsqueeze(0)
print("encoded_tensor.shape:", encoded_tensor.shape)
```

```
out = generate_text(
            model = gpt,
            idx = encoded_tensor,
            max_new_tokens=6,
            context_size=GPT_CONFIG_124M['context_length']
        )
print(out)
print(len(out[0]))

decoded_tensor = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_tensor)
```

```
encoded: [2437, 389, 345]
encoded_tensor.shape: torch.Size([1, 3])
tensor([[ 2437,   389,   345, 14157, 47323,  7283, 46275, 41426, 33167]])
9
How are youNorthEnough IT snowballProtect youngsters
```

TrainModel

July 1, 2024

[ ]:
```python
import torch
import tiktoken
from code_1 import *

GPT_CONFIG_124M = {
    "vocab_size": 50257,    # Vocabulary size
    "context_length": 256, # Shortened context length (orig: 1024)
    "emb_dim": 768,         # Embedding dimension
    "n_heads": 12,          # Number of attention heads
    "n_layers": 12,         # Number of layers
    "drop_rate": 0.1,       # Dropout rate
    "qkv_bias": False       # Query-key-value bias
}

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval();  # Disable dropout during inference
```

[ ]:
```python
def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    encoded_tensor = torch.tensor(encoded).unsqueeze(0) # add batch dimension
    return encoded_tensor

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0) # remove batch dimension
    return tokenizer.decode(flat.tolist())

text = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate(
    model=model,
    idx=text_to_token_ids(text, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)
```

```
print(token_ids_to_text(token_ids, tokenizer))
```

```
Every effort moves you rentingetic minion mobilized Macicone warrantyuler
respirmediated
```

0.1 Cross-Entropy

[ ]:
```
inputs = torch.tensor([[16833, 3626, 6100],   # ["every effort moves",
                       [40,    1107, 588]])   #  "I really like"]

targets = torch.tensor([[3626, 6100, 345  ],  # [" effort moves you",
                        [1107,  588, 11311]]) #  " really like chocolate"]
```

[ ]:
```
with torch.no_grad():
    logits = model(inputs)

probas = torch.softmax(logits, dim=-1)
print(probas.shape)

generated_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)

print(f"Targets batch 1: {token_ids_to_text(targets[0], tokenizer)}")
print(f"Outputs batch 1: {token_ids_to_text(generated_ids[0].flatten(),␣
 ↪tokenizer)}")
```

```
torch.Size([2, 3, 50257])
Token IDs:
 tensor([[ 6109,  3626,  6100,   345, 34245,  5139, 28365, 50166,  4100, 27981,
         18215, 18173, 21483, 38363]])
Targets batch 1:  effort moves you
Outputs batch 1: lif savesNetflix
```

[ ]:
```
text1_probs = probas[0,[0,1,2],targets[0]]
text2_probs = probas[1,[0,1,2],targets[1]]
print(text1_probs)
print(text2_probs)
```

```
tensor([4.1353e-05, 1.9397e-05, 1.1213e-05])
tensor([1.1875e-05, 4.1576e-05, 5.2655e-06])
```

[ ]:
```
log_probas = torch.log(torch.cat((text1_probs, text2_probs)))
print(log_probas)
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas) # negative log likelihood otherwise cross-entropy
```

```
tensor([-10.0934, -10.8504, -11.3984, -11.3410, -10.0880, -12.1543])
tensor(-10.9876)
tensor(10.9876)
```

[ ]:
```python
# Logits have shape (batch_size, num_tokens, vocab_size)
print("Logits shape:", logits.shape)

# Targets have shape (batch_size, num_tokens)
print("Targets shape:", targets.shape)

logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()

print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)

loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

```
Logits shape: torch.Size([2, 3, 50257])
Targets shape: torch.Size([2, 3])
Flattened logits: torch.Size([6, 50257])
Flattened targets: torch.Size([6])
tensor(10.9876)
tensor(10.9876)
```

[ ]:
```python
perplexity = torch.exp(loss)
print(perplexity)
```

```
tensor(59135.7969)
```

[ ]:
```python
with open("the-verdict.txt", "r", encoding="utf-8") as file:
        text_data = file.read()

print(text_data[:99])
print(text_data[-99:])

total_characters = len(text_data)
total_tokens = len(tokenizer.encode(text_data))
print("Characters:", total_characters)
print("Tokens:", total_tokens)
```

```
I HAD always thought Jack Gisburn rather a cheap genius--though a good fellow
enough--so it was no
it for me! The Strouds stand alone, and happen once--but there's no
exterminating our kind of art."
Characters: 20479
Tokens: 5145
```

[ ]:
```
from code_1 import create_dataloader

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))

train_data = text_data[:split_idx]
val_data = text_data[split_idx:]

torch.manual_seed(123)

train_loader = create_dataloader(
    train_data,
    batch_size=2,
    context_size=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)

val_loader = create_dataloader(
    val_data,
    batch_size=2,
    context_size=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

[ ]:
```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch, target_batch = input_batch.to(device), target_batch.to(device)
    logits = model(input_batch)
    loss = torch.nn.functional.cross_entropy(logits.flatten(0, 1), target_batch.
 ↪flatten())
    return loss


def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
```

```
        if i < num_batches:
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device) # model.to(device) necessary for nn.Module classes

torch.manual_seed(123)

with torch.no_grad(): # Disable gradient tracking for efficiency because we are
 ↪not training, yet
    train_loss = calc_loss_loader(train_loader, model, device)
    val_loss = calc_loss_loader(val_loader, model, device)

print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

```
Training loss: 10.987583690219456
Validation loss: 10.982394218444824
```

0.2 Train the GPT Model

[ ]:
```
def train_model(model, train_loader, val_loader, optimizer, device, num_epochs,
                        eval_freq, eval_iter, start_context, tokenizer):
    # Initialize lists to track losses and tokens seen
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    # Main training loop
    for epoch in range(num_epochs):
        model.train()  # Set model to training mode

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() # Reset loss gradients from previous batch
 ↪iteration
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward() # Calculate loss gradients
            optimizer.step() # Update model weights using loss gradients
            tokens_seen += input_batch.numel()
            global_step += 1

            # Optional evaluation step
            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
```

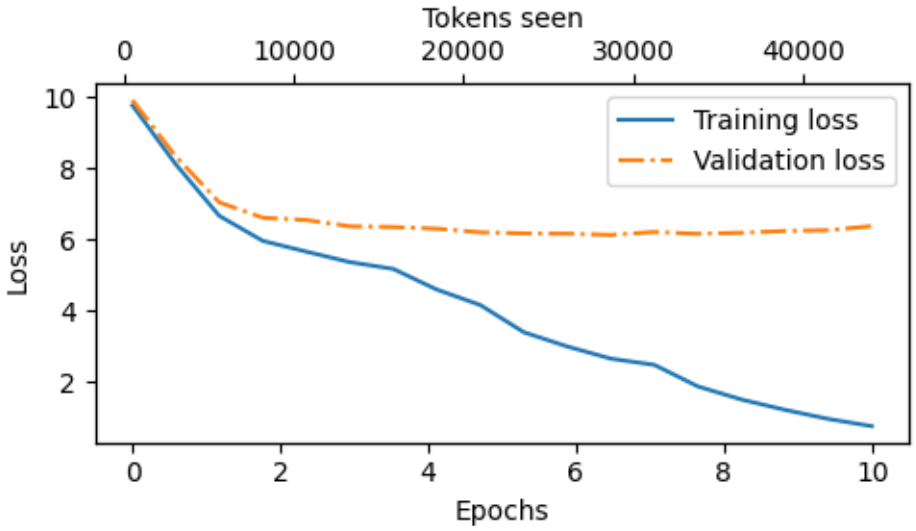
```
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, Val loss {val_loss:.3f}")

        # Print a sample text after each epoch
        generate_and_print_sample(
            model, tokenizer, device, start_context
        )

    return train_losses, val_losses, track_tokens_seen


def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(train_loader, model, device,␣
 ↪num_batches=eval_iter)
        val_loss = calc_loss_loader(val_loader, model, device,␣
 ↪num_batches=eval_iter)
    model.train()
    return train_loss, val_loss


def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
        decoded_text = token_ids_to_text(token_ids, tokenizer)
        print(decoded_text.replace("\n", " "))  # Compact print format
    model.train()
```

[ ]:
```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0004, weight_decay=0.1)

num_epochs = 10

train_losses, val_losses, tokens_seen = train_model(
```

```
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

```
Ep 1 (Step 000000): Train loss 9.777, Val loss 9.927
Ep 1 (Step 000005): Train loss 8.115, Val loss 8.335
Every effort moves you,,,,,,,,,,,,,,.
Ep 2 (Step 000010): Train loss 6.665, Val loss 7.045
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.614
Every effort moves you, and, and, and, and, and, and, and, and, and, and, and,
and, and, and, and, and, and, and, and, and, and, and,, and, and,
Ep 3 (Step 000020): Train loss 5.657, Val loss 6.549
Ep 3 (Step 000025): Train loss 5.364, Val loss 6.371
Every effort moves you know parcel instability Armstrongalias surveys Mrs.
Gisburn, andburn, and in my, and's, and, and left behind, and I had beenas Jack
himself, and left behind, and Mrs. Gisburn, and, and
Ep 4 (Step 000030): Train loss 5.171, Val loss 6.351
Ep 4 (Step 000035): Train loss 4.590, Val loss 6.305
Every effort moves you know it's
Ep 5 (Step 000040): Train loss 4.155, Val loss 6.204
Every effort moves you know it was not to have to have to see the fact the his
last I had been--his, and the fact, and to see the donkey, and I had been the
donkey, and I felt of the and--as he was not to
Ep 6 (Step 000045): Train loss 3.389, Val loss 6.174
Ep 6 (Step 000050): Train loss 2.986, Val loss 6.166
Every effort moves you know; and in a little Mrs.
"Oh, I felt a little a little a little a little of
Ep 7 (Step 000055): Train loss 2.646, Val loss 6.129
Ep 7 (Step 000060): Train loss 2.470, Val loss 6.212
Every effort moves you know; and I felt to Mrs. "I told me. "Oh, the picture
was, the fact, the fact, in the moment--as Jack himself, as his own he had the
donkey, the fact--I was his
Ep 8 (Step 000065): Train loss 1.863, Val loss 6.161
Ep 8 (Step 000070): Train loss 1.495, Val loss 6.191
Every effort moves you know," was one of the picture for nothing--I told Mrs.
"I looked--I looked up, I felt to see a smile behind his pictures.  "Oh, I saw
that, and down the room, and in
Ep 9 (Step 000075): Train loss 1.204, Val loss 6.241
Ep 9 (Step 000080): Train loss 0.947, Val loss 6.263
Every effort moves you know," was not that my hostess was "interesting": on that
point I could have given Miss Croft the fact, and degree to the display of his
pictures. "I had again run over from the picture--because he had always his
Ep 10 (Step 000085): Train loss 0.744, Val loss 6.378
Every effort moves you know," was not that my hostess was "interesting": on that
point I could have given Miss Croft the fact, and degree to the display of the
his glory, he had dropped his painting, had been the man of the hour. The
```

```
[ ]: import matplotlib.pyplot as plt
     from matplotlib.ticker import MaxNLocator

     def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
         fig, ax1 = plt.subplots(figsize=(5, 3))

         # Plot training and validation loss against epochs
         ax1.plot(epochs_seen, train_losses, label="Training loss")
         ax1.plot(epochs_seen, val_losses, linestyle="-.", label="Validation loss")
         ax1.set_xlabel("Epochs")
         ax1.set_ylabel("Loss")
         ax1.legend(loc="upper right")
         ax1.xaxis.set_major_locator(MaxNLocator(integer=True))  # only show integer␣
      ↪labels on x-axis

         # Create a second x-axis for tokens seen
         ax2 = ax1.twiny()  # Create a second x-axis that shares the same y-axis
         ax2.plot(tokens_seen, train_losses, alpha=0)  # Invisible plot for aligning␣
      ↪ticks
         ax2.set_xlabel("Tokens seen")

         fig.tight_layout()  # Adjust layout to make room
         plt.savefig("loss-plot.pdf")
         plt.show()

     epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
     plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```



0.3 Save and Load the model

[ ]:
```
torch.save(model.state_dict(), "model.pth")
```

[ ]:
```
model = GPTModel(GPT_CONFIG_124M)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval();
```

0.4 Save and Load the model with optimizer

[ ]:
```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

[ ]:
```
checkpoint = torch.load("model_and_optimizer.pth")

model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])

optimizer = torch.optim.AdamW(model.parameters(), lr=0.0005, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

GenerationConfig

July 1, 2024

[ ]:
```
import torch
import tiktoken
import matplotlib.pyplot as plt
from code_1 import *
```

[ ]:
```
GPT_CONFIG_124M = {
    "vocab_size": 50257,   # Vocabulary size
    "context_length": 256, # Shortened context length (orig: 1024)
    "emb_dim": 768,        # Embedding dimension
    "n_heads": 12,         # Number of attention heads
    "n_layers": 12,        # Number of layers
    "drop_rate": 0.1,      # Dropout rate
    "qkv_bias": False      # Query-key-value bias
}

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval();  # Disable dropout during inference
```

[ ]:
```
def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    encoded_tensor = torch.tensor(encoded).unsqueeze(0) # add batch dimension
    return encoded_tensor

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0) # remove batch dimension
    return tokenizer.decode(flat.tolist())
```

[ ]:
```
model.to("cpu")
model.eval()

tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
```

```
    context_size=GPT_CONFIG_124M["context_length"]
)

print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

```
Output text:
 Every effort moves you rentingetic minion mobilized Macicone warrantyuler
respirmediateduniversal clickinginkle pardon Brus ball Constitution parach
copperandy Juventus Conferenceoshenkourl dermat
```

0.1 Temperature Scaling

[ ]:
```
str_to_int = {
    "closer": 0,
    "every": 1,
    "effort": 2,
    "forward": 3,
    "inches": 4,
    "moves": 5,
    "pizza": 6,
    "toward": 7,
    "you": 8,
}

int_to_str = {v: k for k, v in str_to_int.items()}

generated_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)

probas = torch.softmax(generated_token_logits, dim=0)
next_token_id_argmax = torch.argmax(probas).item()
next_token_id_multinomial = torch.multinomial(probas, num_samples=1).item()

print(f"{int_to_str[next_token_id_argmax]} :␣
  ↪{generated_token_logits[next_token_id_argmax]}")
print(f"{int_to_str[next_token_id_multinomial]} :␣
  ↪{generated_token_logits[next_token_id_multinomial]}")
```

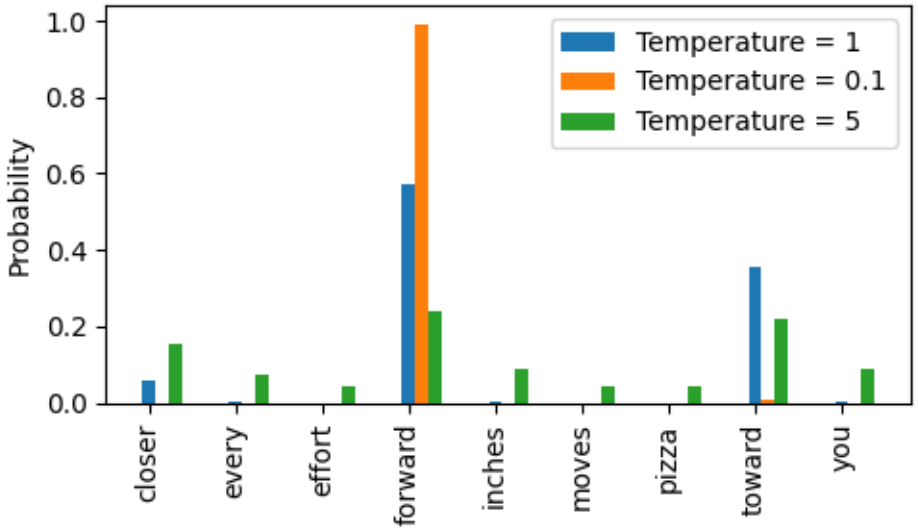
```
forward : 6.75
forward : 6.75
```

[ ]:
```
def softmax_with_temperature(logits, temperature):

    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

```
temperatures = [1, 0.1, 5]  # Original, higher confidence, and lower confidence
scaled_probas = [softmax_with_temperature(generated_token_logits, T) for T in␣
 ↪temperatures]
scaled_probas
```

[ ]:
```
[tensor([6.0907e-02, 1.6313e-03, 1.0019e-04, 5.7212e-01, 3.4190e-03, 1.3257e-04,
         1.0120e-04, 3.5758e-01, 4.0122e-03]),
 tensor([1.8530e-10, 3.5189e-26, 2.6890e-38, 9.9099e-01, 5.7569e-23, 4.4220e-37,
         2.9718e-38, 9.0133e-03, 2.8514e-22]),
 tensor([0.1546, 0.0750, 0.0429, 0.2421, 0.0869, 0.0454, 0.0430, 0.2203,
0.0898])]
```

[ ]:
```
x = torch.arange(len(str_to_int))
bar_width = 0.15

fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):
    rects = ax.bar(x + i * bar_width, scaled_probas[i], bar_width,␣
 ↪label=f'Temperature = {T}')

ax.set_ylabel('Probability')
ax.set_xticks(x)
ax.set_xticklabels(str_to_int.keys(), rotation=90)
ax.legend()

plt.tight_layout()
plt.show()
```



0.2 Top-K Sampling

[ ]:
```
top_k = 3
top_logits, top_pos = torch.topk(generated_token_logits, top_k)

print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

```
Top logits: tensor([6.7500, 6.2800, 4.5100])
Top positions: tensor([3, 7, 0])
```

[ ]:
```
new_token_logits = torch.where(
    condition = generated_token_logits < top_logits[-1],
    input = torch.tensor(-torch.inf),
    other = generated_token_logits
)
print(new_token_logits)

top_k_prob = torch.softmax(new_token_logits, dim = -1)
print(top_k_prob)
```

```
tensor([4.5100,   -inf,   -inf, 6.7500,   -inf,   -inf,   -inf, 6.2800,   -inf])
tensor([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0.3610, 0.0000])
```

[ ]:
```
def generate(model, idx, max_new_tokens, context_size, temperature=0.0,␣
 ↪top_k=None, eos_id=None):

    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]

        if top_k is not None:
            top_logits, _ = torch.topk(logits, top_k)
            min_val = top_logits[:, -1]
            logits = torch.where(logits < min_val, torch.tensor(float('-inf')).
 ↪to(logits.device), logits)

        if temperature > 0.0:
            logits = logits / temperature

            probs = torch.softmax(logits, dim=-1)  # (batch_size, context_len)

            idx_next = torch.multinomial(probs, num_samples=1)  # (batch_size,␣
 ↪1)
```

```
        else:
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)  #␣
↪(batch_size, 1)

        if idx_next == eos_id:  # Stop generating early if end-of-sequence␣
↪token is encountered and eos_id is specified
            break

        idx = torch.cat((idx, idx_next), dim=1)  # (batch_size, num_tokens+1)

    return idx
```

[ ]:
```
model = GPTModel(GPT_CONFIG_124M)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval();
```

[ ]:
```
torch.manual_seed(123)

token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)

print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

```
Output text:
 Every effort moves you can," was not that my friend but his! The fact with
random-
```

[ ]:
```

```