For each practical exercise (TP), please work in groups of two or three. Then, create a **private GitHub repository** and add me (my GitHub is **arthur-75**) to your project. Finally, share the link to your project (or TP) under [Practical Exercises](https://docs.google.com/spreadsheets/d/1V-YKgHn71FnwjoFltDhWsPJS7uIuAh9lj6SP2DSCvlY/edit?usp=sharing) and make sure to choose your **team name** :-)

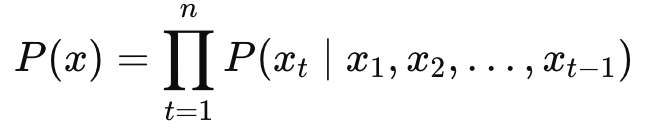
# **Autoregressive Models (GPT)**

## **1. What is an Autoregressive Model?**

Autoregressive models generate sequences by predicting one token at a time, conditioned on all previously generated tokens. Formally, for a sequence of tokens:



an autoregressive model factorizes the probability distribution as:



This means each token xt depends explicitly only on previously observed tokens, which makes it ideal for sequential text generation tasks.

Formally, the conditional probability of predicting a token xtx\_txt​ given previous tokens is expressed as:



where:

* x<t denotes all tokens generated before the current token t.
* h represents hidden states produced by the Transformer layers.
* W represents vocabulary vectors.

### **Data Preprocessing and Tokenization**

* **Data Cleaning**: Ensure text is consistent by spacing punctuation, lowercasing, and removing unwanted characters.
* **Tokenization**: Convert text into discrete tokens. Each unique word gets a unique integer index.
* **Vocabulary Creation**: Build vocabulary from the most frequent words (usually limited to a fixed size, e.g., 10,000).

import torch

import torch.nn as nn

from torch.utils.data import Dataset, DataLoader

from torch.optim import Adam

from torch.nn.functional import cross\_entropy

import numpy as np

import re, string

from tqdm import tqdm

from sklearn.model\_selection import train\_test\_split

from collections import Counter

import json

import kagglehub

# Download latest version

path = kagglehub.dataset\_download("zynicide/wine-reviews" )

# Parameters

VOCAB\_SIZE = 10000

MAX\_LEN = 80

EMBEDDING\_DIM = 256

N\_HEADS = 2

FF\_DIM = 256

BATCH\_SIZE = 32

EPOCHS = 100

DEVICE = "mps"#torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# 1. Load the data

with open(path+"/winemag-data-130k-v2.json") as json\_data:

wine\_data = json.load(json\_data)

filtered\_data = [

"wine review : "

+ x["country"]

+ " : "

+ x["province"]

+ " : "

+ x["variety"]

+ " : "

+ x["description"]

for x in wine\_data

if x["country"] is not None

and x["province"] is not None

and x["variety"] is not None

and x["description"] is not None

]

# Tokenizer

class SimpleTokenizer:

def \_\_init\_\_(self, xxx, xxx):

self.vocab\_size = vocab\_size

self.counter = Counter()

for text in texts:

self.counter.update(text.split())

self.vocab = [word for word, \_ in self.counter.most\_common(vocab\_size - 2)]

self.word2idx = {word: idx + 2 for idx, word in enumerate(self.vocab)}

self.word2idx["<pad>"] = 0

self.word2idx["<unk>"] = 1

def encode(self, text):

return [self.word2idx.get(word, 1) for word in text.split()]

# Data preparation

def pad\_punctuation(s):

s = re.sub(f"([{string.punctuation}])", r" \1 ", s)

s = re.sub(" +", " ", s)

return s.lower()

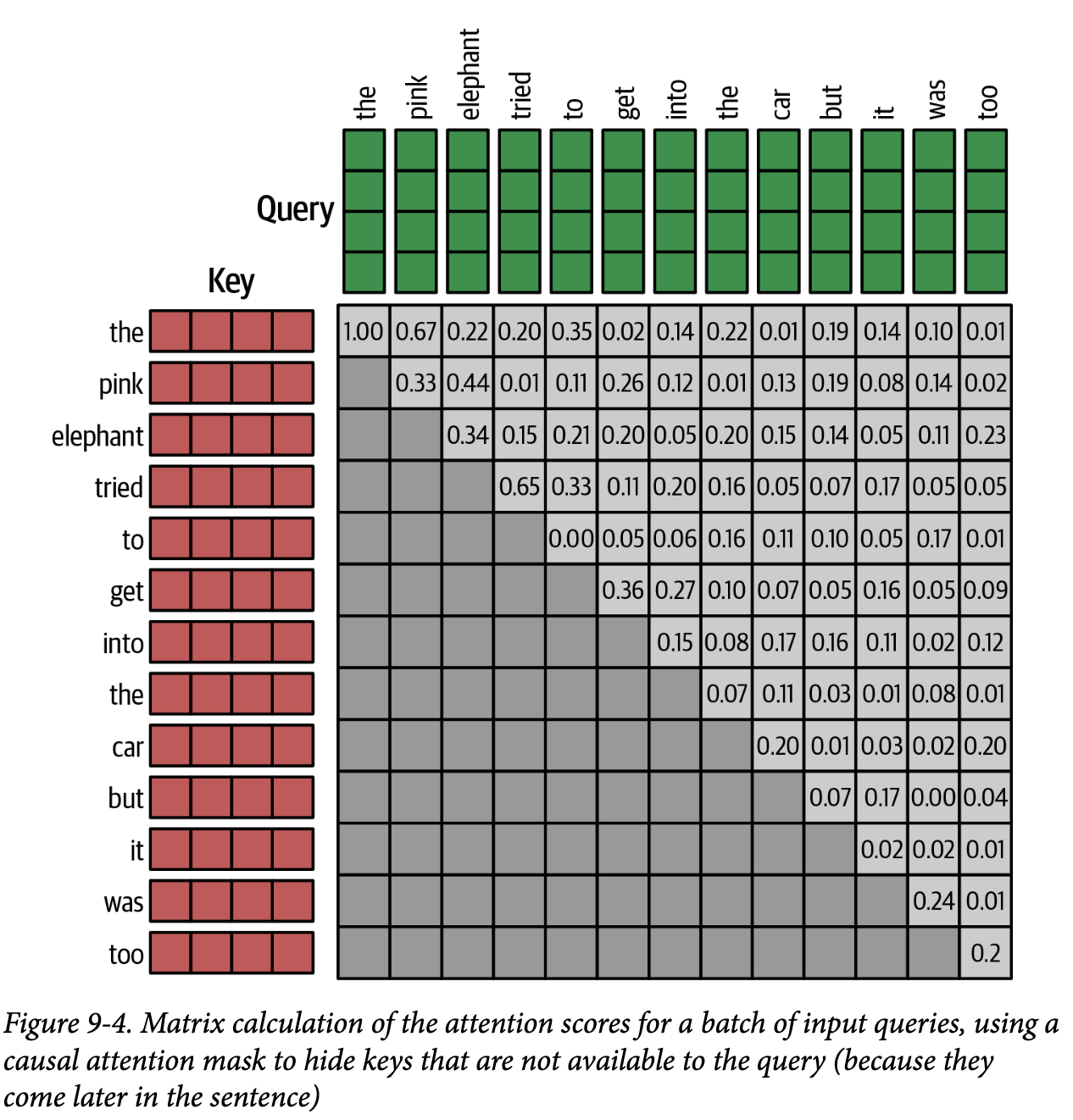
# Create tokenizer

tokenizer = SimpleTokenizer(filtered\_data, VOCAB\_SIZE)



**Constructing the Dataset**

* **Input-target pairs**: Prepare data such that each input sequence is paired with the same sequence shifted by one token.
  + Input sequence: (x1,x2,...,xn−1)
  + Target sequence: (x2,x3,...,xn)



# Dataset class

class WineDataset(Dataset):

def \_\_init\_\_(self, texts, tokenizer, max\_len):

self.texts = texts

self.tokenizer = tokenizer

self.max\_len = max\_len

def \_\_len\_\_(self):

return len(self.texts)

def \_\_getitem\_\_(self, idx):

tokens = self.tokenizer.encode(self.texts[idx])[:self.max\_len+1]

padding = [self.tokenizer.word2idx["<pad>"]] \* (self.max\_len + 1 - len(tokens))

tokens += padding

return torch.tensor(tokens[:-1]), torch.tensor(tokens[1:])

train\_dataset = WineDataset([pad\_punctuation(t) for t in xxx], xxx, xxx)

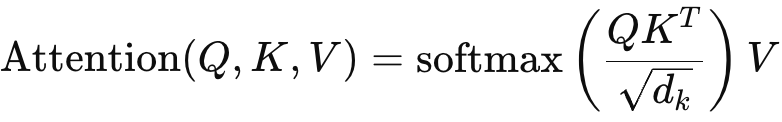
train\_loader = DataLoader(train\_dataset, batch\_size=BATCH\_SIZE, shuffle=True)



### **Model Definition**

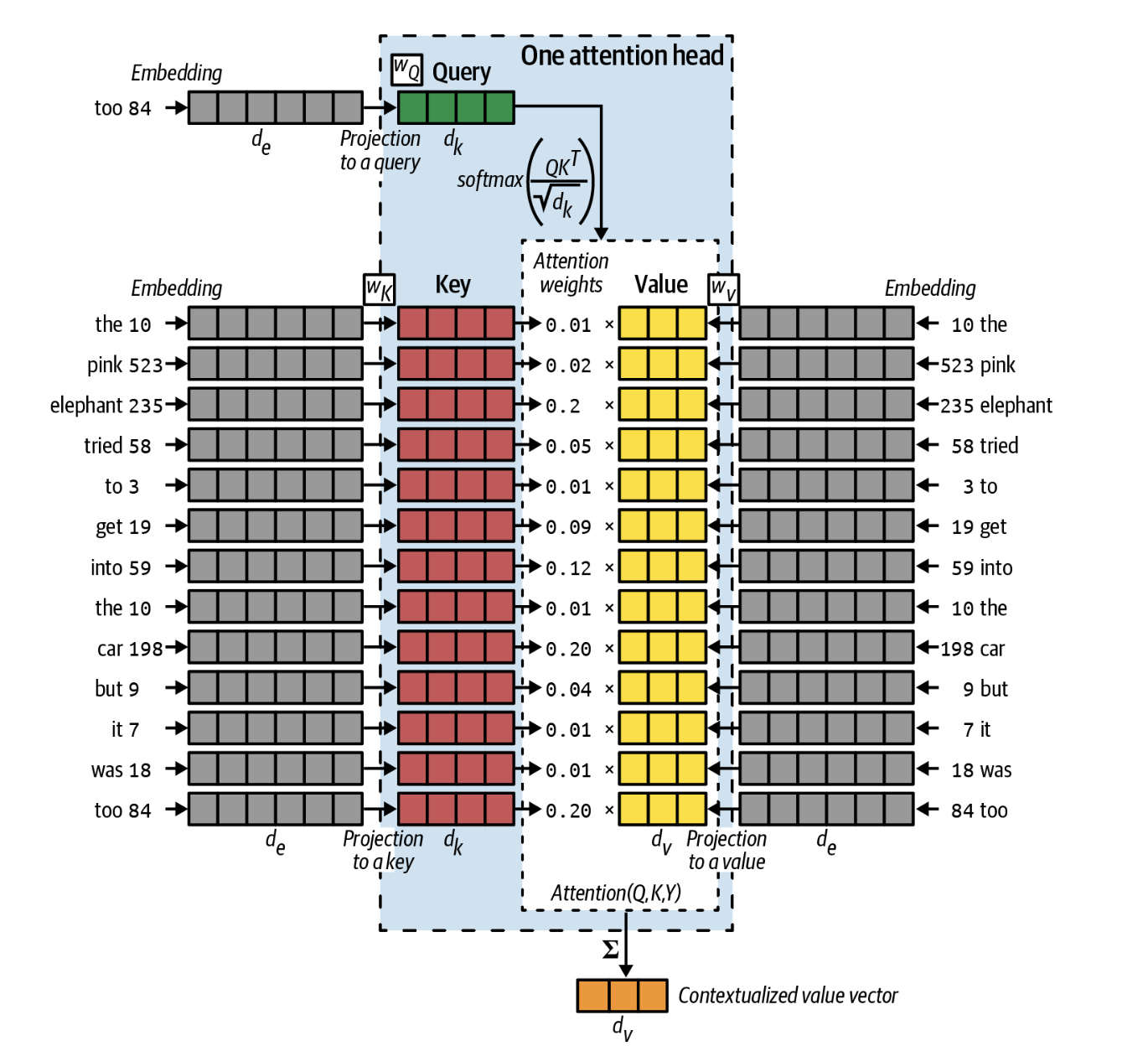
An autoregressive model (like GPT) typically consists of:

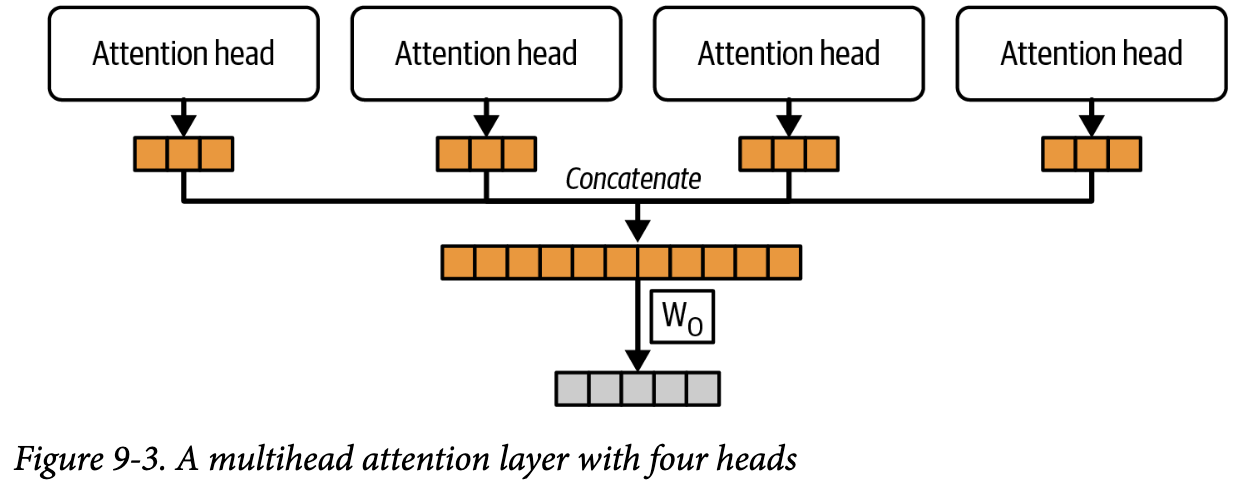
* **Embedding Layer**: Maps discrete tokens into continuous vector embeddings.
* **Positional Encoding**: Adds positional information to token embeddings to retain sequence order.
* **Transformer Block**:  
  + **Self-attention**: Captures dependencies between tokens.  
     Formally, self-attention computes:



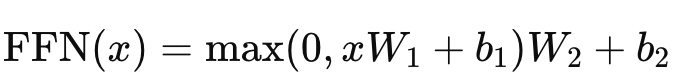
Where:

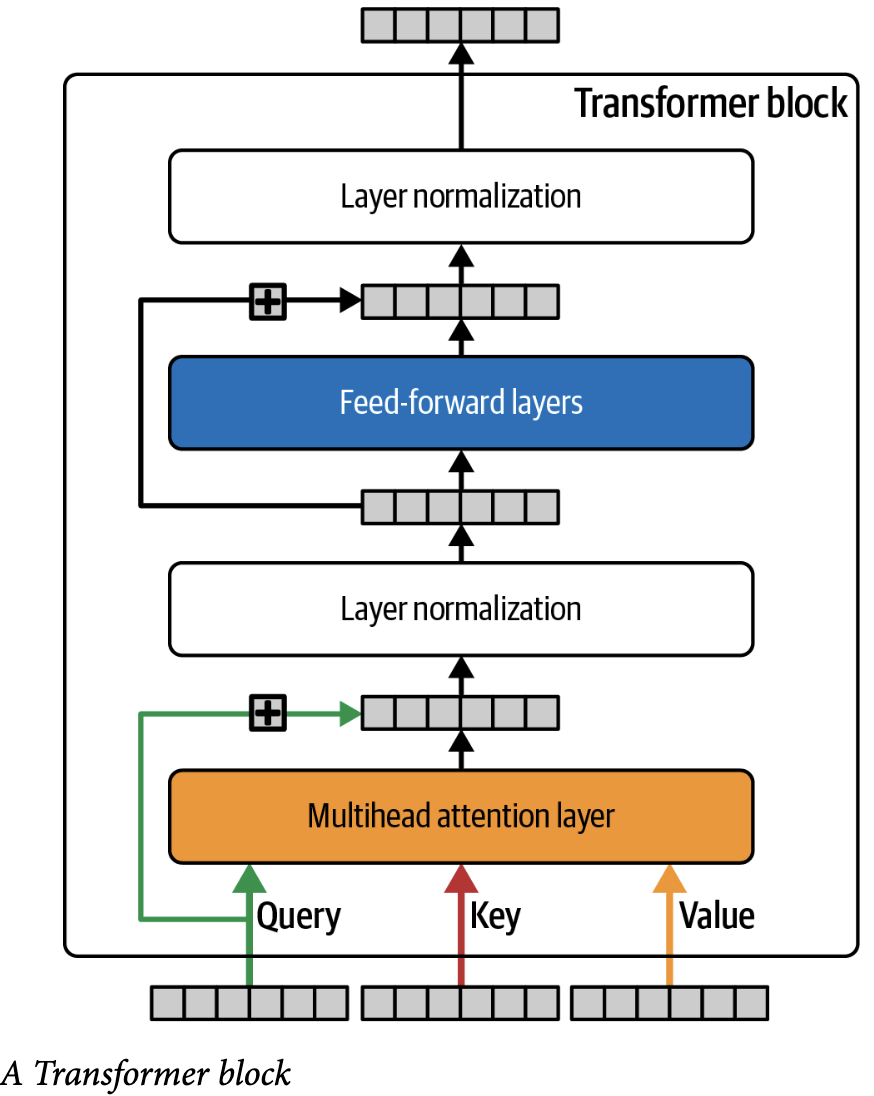
* Q, K and V represent Query, Key, and Value embeddings.
* dkd​ is the dimension of the keys.

**  
The mechanics of an attention head**



* **Feedforward Network**: Introduces non-linearity and transforms attention outputs:

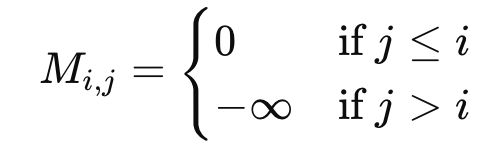




* **Layer normalization and residual connections** are added to stabilize training and help information flow.

## Causal Attention Masking

To maintain autoregressive behavior (no peeking into the future tokens), you need a causal mask, ensuring that each position in the sequence can only attend to itself and all previous positions, not future positions.



This masking ensures the model predicts the next token based only on previously seen tokens.

class GPTBlock(nn.Module):

def \_\_init\_\_(self, embed\_dim, num\_heads, ff\_dim):

super().\_\_init\_\_()

self.attn = nn.MultiheadAttention(embed\_dim, num\_heads, batch\_first=True)

self.ffn = nn.Sequential(

nn.Linear(embed\_dim, ff\_dim),

nn.ReLU(),

nn.Linear(ff\_dim, embed\_dim)

)

self.ln1 = nn.LayerNorm(embed\_dim)

self.ln2 = nn.LayerNorm(embed\_dim)

def forward(self, x):

seq\_len = x.size(1)

mask = torch.tril(torch.ones(seq\_len, seq\_len)).to(x.device)

mask = mask.masked\_fill(mask == 0, float('-inf')).masked\_fill(mask == 1, 0.0)

attn\_output, \_ = self.attn(x, x, x, attn\_mask=mask)

x = self.ln1(x + attn\_output)

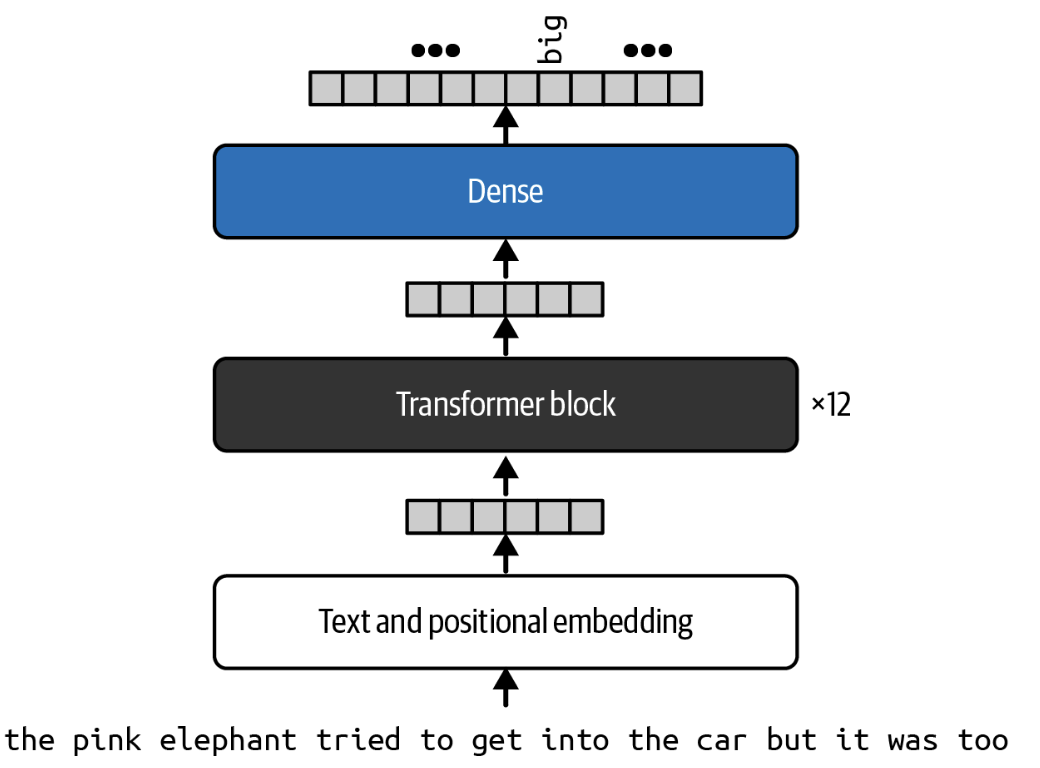
ffn\_output = self.ffn(x)

return self.ln2(x + ffn\_output)



GPTModel

Finally a dense layer to represent vocabulary logits



class GPTModel(nn.Module):

def \_\_init\_\_(self, vocab\_size, embed\_dim, max\_len, num\_heads, ff\_dim):

super().\_\_init\_\_()

self.embedding = nn.Embedding(vocab\_size, embed\_dim)

self.pos\_embedding = nn.Embedding(max\_len, embed\_dim)

self.transformer = GPTBlock(embed\_dim, num\_heads, ff\_dim)

self.fc = nn.Linear(embed\_dim, vocab\_size)

def forward(self, x):

positions = torch.arange(0, x.size(1), device=x.device).unsqueeze(0)

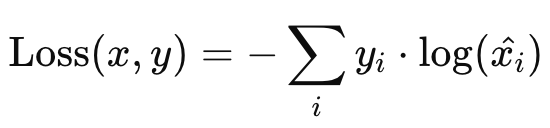
x = self.embedding(x) + self.pos\_embedding(positions)

x = self.transformer(x)

return self.fc(x)

### **Training Objective**

* Autoregressive models use **cross-entropy loss**, defined as:



where:

* y is the true next token distribution.
* x is the model’s predicted probabilities.

### **Training Loop**

* Forward pass: Provide input sequence, compute logits.
* Calculate loss between predicted tokens and actual next tokens.
* Perform backpropagation to update weights.

# Training loop

# Training

model = GPTModel(VOCAB\_SIZE, EMBEDDING\_DIM, MAX\_LEN, N\_HEADS, FF\_DIM).to(DEVICE)

optimizer = Adam(model.parameters(), lr=0.0001)

for epoch in range(EPOCHS):

model.train()

total\_loss = 0

for x\_batch, y\_batch in tqdm(train\_loader, desc=f"Epoch {epoch+1}/{EPOCHS}"):

x\_batch, y\_batch = x\_batch.to(DEVICE), y\_batch.to(DEVICE)

optimizer.zero\_grad()

logits = model(x\_batch)

loss = cross\_entropy(logits.view(-1, VOCAB\_SIZE), y\_batch.view(-1), ignore\_index=tokenizer.word2idx['<pad>'])

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_loss = total\_loss / len(train\_loader)

print(f"Epoch {epoch+1}, Training Loss: {avg\_loss:.4f}")

# **Sampling and Text Generation**

### **Autoregressive Sampling :**

Once the model is trained, generating text is done token-by-token, starting from a provided prompt.

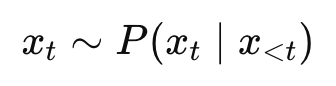
1. **Prompt Initialization**:  
   * Convert the initial textual prompt into token IDs.
2. **Prediction Loop**: For each step t from the current length to the desired length:  
   * Generate probabilities for the next token:



Here, T is the temperature controlling randomness:

* Lower T (e.g., 0.5) makes the output deterministic.
* Higher T (e.g., 1.5) makes it more creative/random.

1. **Sample Token**:
   * From the probability distribution generated by the softmax, sample the next token:



1. **Repeat Prediction**:
   * Append this new token to your input and repeat until reaching the desired length or the end token.

def generate\_text(model, tokenizer, start\_prompt, max\_tokens=80, temperature=1.0):

model.eval()

tokens = tokenizer.encode(start\_prompt)

tokens = tokens[:MAX\_LEN]

generated = tokens.copy()

with torch.no\_grad():

for \_ in range(max\_tokens):

input\_tensor = torch.tensor([generated[-MAX\_LEN:]], device=DEVICE)

logits = model(input\_tensor)

logits = logits[:, -1, :] / temperature

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

next\_token = torch.multinomial(probs, num\_samples=1).item()

if next\_token == tokenizer.word2idx['<pad>']:

break

generated.append(next\_token)

idx2word = {idx: word for word, idx in tokenizer.word2idx.items()}

generated\_text = ' '.join(idx2word.get(idx, '<unk>') for idx in generated)

return generated\_text

# Example usage:

prompt = "wine review : US"

generated\_text = generate\_text(model, tokenizer, prompt, max\_tokens=50, temperature=0.8).strip()

print("Generated text:\n", generated\_text)

