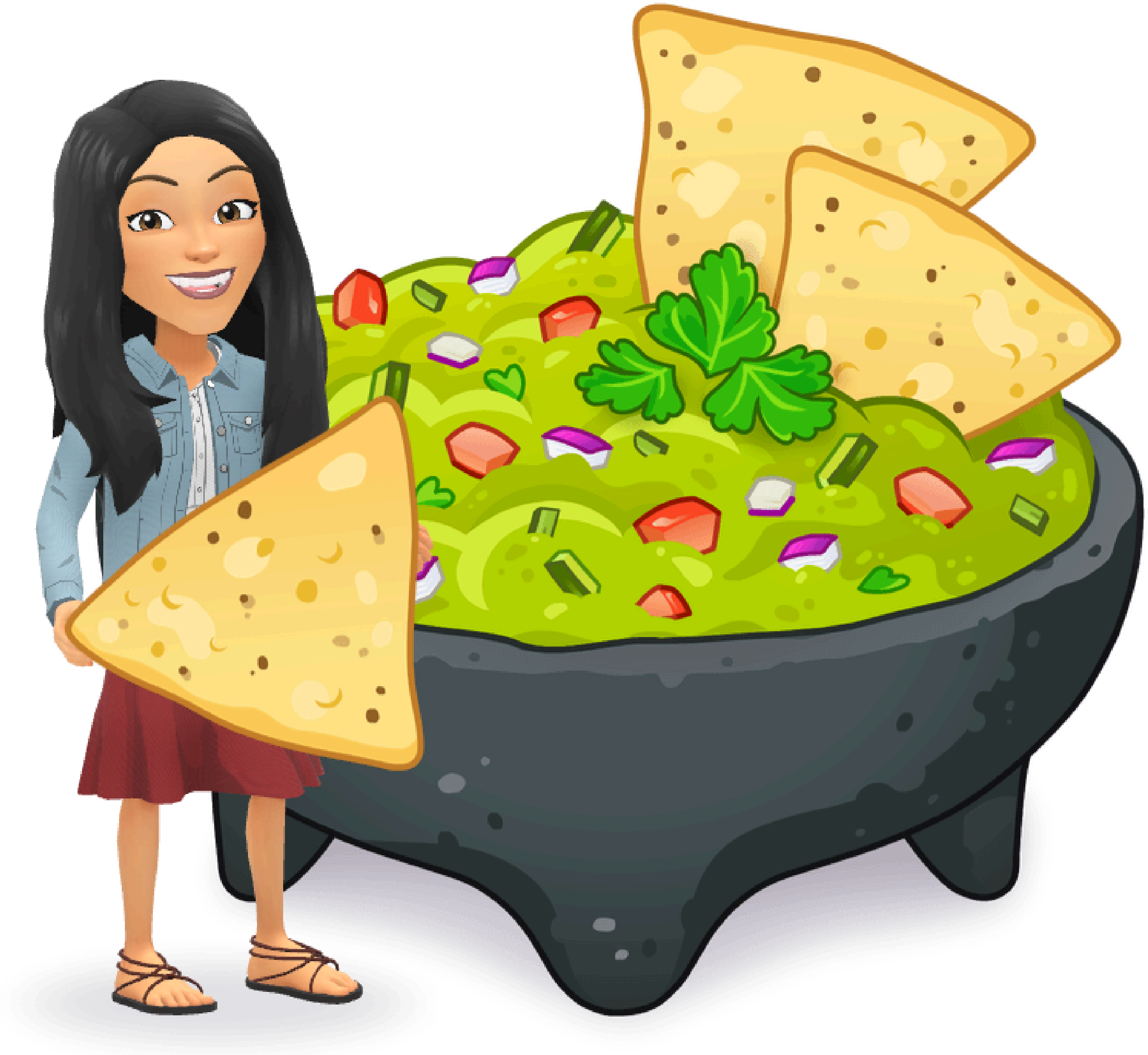
Recipe Generation using Transformers

Contents

This notebook demonstrates how to build a Transformer-based model to generate recipe titles. You'll learn about tokenization, preparing datasets, building and training the model, and generating new text.



Imports

```
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import Dataset, DataLoader
import pandas as pd
import os
import re
import sys
from collections import Counter, defaultdict
from urllib.request import urlopen
import math
```

This is a demo for recipe generation using PyTorch and Transformers. For the purpose of this demo, we'll sample 10_000 recipe titles from the corpus

Data

```
orig_recipes_df = pd.read_csv("../data/RAW_recipes.csv")
orig_recipes_df = orig_recipes_df.dropna()
recipes_df = orig_recipes_df.sample(10_000)
```

```
recipes_df
```

	name	id	minutes	contributor_id	submitted	tags
37256	carrots in honey mustard sauce	80594	18	80353	2004-01-09	['30-minutes-or-less', 'time-to-make', 'course...
1339	3 layer mexican party dip	111880	30	188744	2005-02-24	['30-minutes-or-less', 'time-to-make', 'course...
208228	tamatim mashwiya	228387	15	431813	2007-05-16	['15-minutes-or-less', 'time-to-make', 'course...
168411	quick and easy peas water chestnuts	91307	11	139475	2004-05-18	['15-minutes-or-less', 'time-to-make', 'course...
219586	ultimate pumpkin cheesecake by bird	264199	110	452940	2007-11-07	['time-to-make', 'course', 'main-ingredient', ...
...	...	...	...	...	...	...
157068	pecan praline bars	199806	17	336058	2006-12-09	['30-minutes-or-less', 'time-to-make', 'course...
113855	jalapeno popper grilled cheese sandwich	471266	40	1072593	2012-01-04	['weeknight', '60-minutes-or-less', 'time-to-m...

	name	id	minutes	contributor_id	submitted	tags
128061	macaroni and chicken salad	304461	30	487548	2008-05-21	['30-minutes-or-less', 'time-to-make', 'course...
10847	authentic south florida cuban sandwiches	284773	15	706934	2008-02-07	['15-minutes-or-less', 'time-to-make', 'course...
45248	chicken pizza primavera	215313	40	376098	2007-03-06	['60-minutes-or-less', 'time-to-make', 'course...

10000 rows × 12 columns

```
# Set the appropriate device depending upon your hardware.

# device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')
print(device)
```

```
mps
```

```
recipes = recipes_df['name'].tolist()
recipes[:10]
```

```
['carrots in honey mustard sauce',
 '3 layer mexican party dip',
 'tamatim mashwiya',
 'quick and easy peas   water chestnuts',
 'ultimate pumpkin cheesecake  by bird',
 'bean burgers',
 'yam nuea   thai beef salad',
 'rosemary parmesan cheese refrigerator crackers',
 'creamy carrot and scallion baked potato topping',
 'gordon ramsay s salmon with baked herbs   caramelized lemons']
```

Tokenization

Let’s start with tokenization.

- We create a tokenizer wrapper to convert recipe names into tokens using a pre-trained language model (like BERT) that knows lots of words and subwords. But for our specific dataset (say, a bunch of recipe descriptions), we only need a much smaller dictionary, just the words (tokens) that actually show up in our dataset.

So this code helps us:

- Use the tokenizer from a big pre-trained model.
- Go through our dataset and extract just the tokens we need.
- Build a mini vocabulary just for our data.
- Be able to tokenize and decode texts using this mini vocab.

```
from transformers import AutoTokenizer
from tqdm import trange

class TokenizerWrapper:
    """Wraps AutoTokenizer with a custom vocabulary mapping."""

    def __init__(self, model_name="bert-base-cased"):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)

        # Initialize mappings with special tokens: [PAD] -> 0, [CLS] -> 1, [SE
        self.token_id_to_vocab_id = {0: 0, 101: 1, 102: 2}
        self.vocab_id_to_token_id = {0: 0, 1: 101, 2: 102}

        self.vocab_id = 3  # Start after special tokens
        self.padding_len = None

    def build_dictionary(self, recipes: list[str]):
        """Builds vocabulary from a list of recipes and sets padding length.""
        tokenized = self.tokenizer(recipes, padding='longest').input_ids
        self.padding_len = len(tokenized[0])

        for tokens in tokenized:
            for token_id in tokens:
                if token_id not in self.token_id_to_vocab_id:
                    self.token_id_to_vocab_id[token_id] = self.vocab_id
                    self.vocab_id_to_token_id[self.vocab_id] = token_id
                    self.vocab_id += 1

    def get_vocab_size(self) -> int:
        """Returns the size of the custom vocabulary."""
        assert len(self.token_id_to_vocab_id) == len(self.vocab_id_to_token_id
        return self.vocab_id

    def tokenize(self, text: str) -> list[int]:
        """Tokenizes text using custom vocabulary (requires build_dictionary f
        assert self.padding_len is not None, "Call build_dictionary() before t
        token_ids = self.tokenizer(text, padding='max_length', max_length=self
        return [self.token_id_to_vocab_id[token_id] for token_id in token_ids]

    def decode(self, vocab_ids: list[int]) -> str:
        """Decodes a list of custom vocab IDs into a string."""
        token_ids = [self.vocab_id_to_token_id[vocab_id] for vocab_id in vocab
        # decoded_string = self.tokenizer.decode(token_ids, skip_special_token
        decoded_string = self.tokenizer.decode(token_ids, skip_special_tokens=
        return decoded_string
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/utils/in
   1966 try:
-> 1967     return importlib.import_module("." + module_name, self.__name__)
   1968 except Exception as e:

File ~/miniforge3/envs/jbook/lib/python3.12/importlib/__init__.py:90, in impor
     89         level += 1
---> 90 return _bootstrap._gcd_import(name[level:], package, level)

File <frozen importlib._bootstrap>:1387, in _gcd_import(name, package, level)

File <frozen importlib._bootstrap>:1360, in _find_and_load(name, import_)

File <frozen importlib._bootstrap>:1331, in _find_and_load_unlocked(name, impo

File <frozen importlib._bootstrap>:935, in _load_unlocked(spec)

File <frozen importlib._bootstrap_external>:995, in exec_module(self, module)

File <frozen importlib._bootstrap>:488, in _call_with_frames_removed(f, *args,

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/integra
     23 import numpy as np
---> 24 from tokenizers import Tokenizer, decoders, normalizers, pre_tokenizer
     25 from tokenizers.models import BPE, Unigram

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/tokenizers/__init__.
     78 from .tokenizers import (
     79     AddedToken,
     80     Encoding,
   (...)
     92     __version__,
     93 )
---> 94 from .implementations import (
     95     BertWordPieceTokenizer,
     96     ByteLevelBPETokenizer,
     97     CharBPETokenizer,
     98     SentencePieceBPETokenizer,
     99     SentencePieceUnigramTokenizer,
    100 )

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/tokenizers/implementa
----> 1 from .base_tokenizer import BaseTokenizer
      2 from .bert_wordpiece import BertWordPieceTokenizer

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/tokenizers/implementa
      3 from tokenizers import AddedToken, EncodeInput, Encoding, InputSequence
----> 4 from tokenizers.decoders import Decoder
      5 from tokenizers.models import Model

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/tokenizers/decoders/_
     14 Sequence = decoders.Sequence
---> 15 DecodeStream = decoders.DecodeStream
```

```
AttributeError: module 'decoders' has no attribute 'DecodeStream'

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/utils/in
   1966 try:
-> 1967     return importlib.import_module("." + module_name, self.__name__)
   1968 except Exception as e:

File ~/miniforge3/envs/jbook/lib/python3.12/importlib/__init__.py:90, in impor
     89         level += 1
---> 90 return _bootstrap._gcd_import(name[level:], package, level)

File <frozen importlib._bootstrap>:1387, in _gcd_import(name, package, level)

File <frozen importlib._bootstrap>:1360, in _find_and_load(name, import_)

File <frozen importlib._bootstrap>:1331, in _find_and_load_unlocked(name, impo

File <frozen importlib._bootstrap>:935, in _load_unlocked(spec)

File <frozen importlib._bootstrap_external>:995, in exec_module(self, module)

File <frozen importlib._bootstrap>:488, in _call_with_frames_removed(f, *args,

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/models/a
     22 from typing import TYPE_CHECKING, Dict, Optional, Tuple, Union
---> 24 from ...configuration_utils import PretrainedConfig
     25 from ...dynamic_module_utils import get_class_from_dynamic_module, reso

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/configu
     26 from .dynamic_module_utils import custom_object_save
---> 27 from .modeling_gguf_pytorch_utils import load_gguf_checkpoint
     28 from .utils import (
     29     CONFIG_NAME,
     30     PushToHubMixin,
   (...)
     39     logging,
     40 )

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/modeling
     20 from tqdm.auto import tqdm
---> 22 from .integrations import (
     23     GGUF_CONFIG_MAPPING,
     24     GGUF_TOKENIZER_MAPPING,
     25     _gguf_parse_value,
     26 )
     27 from .utils import is_torch_available

File <frozen importlib._bootstrap>:1412, in _handle_fromlist(module, fromlist,

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/utils/in
   1954 elif name in self._class_to_module.keys():
-> 1955     module = self._get_module(self._class_to_module[name])
```

```
   1956     value = getattr(module, name)

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/utils/in
   1968 except Exception as e:
-> 1969     raise RuntimeError(
   1970         f"Failed to import {self.__name__}.{module_name} because of the
   1971         f" traceback):\n{e}"
   1972     ) from e

RuntimeError: Failed to import transformers.integrations.ggml because of the fo
module 'decoders' has no attribute 'DecodeStream'

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
Cell In[6], line 1
----> 1 from transformers import AutoTokenizer
      2 from tqdm import trange
      4 class TokenizerWrapper:

File <frozen importlib._bootstrap>:1412, in _handle_fromlist(module, fromlist,

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/utils/in
   1954 elif name in self._class_to_module.keys():
   1955     module = self._get_module(self._class_to_module[name])
-> 1956     value = getattr(module, name)
   1957 elif name in self._modules:
   1958     value = self._get_module(name)

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/utils/in
   1953     value = Placeholder
   1954 elif name in self._class_to_module.keys():
-> 1955     module = self._get_module(self._class_to_module[name])
   1956     value = getattr(module, name)
   1957 elif name in self._modules:

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/transformers/utils/in
   1967     return importlib.import_module("." + module_name, self.__name__)
   1968 except Exception as e:
-> 1969     raise RuntimeError(
   1970         f"Failed to import {self.__name__}.{module_name} because of the
   1971         f" traceback):\n{e}"
   1972     ) from e

RuntimeError: Failed to import transformers.models.auto.tokenization_auto becau
Failed to import transformers.integrations.ggml because of the following error
module 'decoders' has no attribute 'DecodeStream'
```

```python
# Build the dictionary for our tokenizer
from tqdm import tqdm, trange
tokenizer_wrapper = TokenizerWrapper()
tokenizer_wrapper.build_dictionary(recipes_df["name"].to_list())
```

```
recipe_tokens = tokenizer_wrapper.tokenize(recipes_df['name'].iloc[10])
decoeded_recipe = tokenizer_wrapper.decode(recipe_tokens)
print('Recipe:', recipes_df['name'].iloc[10])
print('Tokens:', recipe_tokens)
print('Decoded recipe:', decoeded_recipe)
```

```
Recipe: roast teriyaki broccoli
Tokens: [1, 90, 91, 28, 92, 93, 33, 94, 95, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
Decoded recipe: [CLS] roast teriyaki broccoli [SEP] [PAD] [PAD] [PAD] [PAD] [PA
```

```
vocab_size = tokenizer_wrapper.get_vocab_size()
vocab_size
```

```
3699
```

❓❓ Questions for you

- Shouldn't we just have a few meaningful indices above? What's going on?
- Why might we want to build a smaller custom vocabulary from our dataset instead of using the full vocabulary from a large pre-trained model?
- What do you think the impact would be on memory usage?

Dataset preparation

We split the dataset into training and test sets and convert each recipe name into a token sequence.

```
def build_data(data_df, tokenizer_wrapper):
    dataset = []
    for row_id in trange(len(data_df)):
        reicpe_tokens = torch.tensor(tokenizer_wrapper.tokenize(data_df['name'
        dataset.append({'token': reicpe_tokens})
    return dataset
```

Let's create train and test datasets by calling `build_data` on train and test splits.

```
from sklearn.model_selection import train_test_split

train_df, test_df = train_test_split(recipes_df, test_size=0.2, random_state=1
train_data = build_data(train_df, tokenizer_wrapper)
test_data = build_data(test_df, tokenizer_wrapper)
```

```
  0%|                                                     | 0/8000 [00:00<?, ?it/s
To disable this warning, you can either:
	- Avoid using `tokenizers` before the fork if possible
	- Explicitly set the environment variable TOKENIZERS_PARALLELISM=(true
100%|███████████████████████████████████| 8000/8000 [00:00<00:00, 19073.31it/s
100%|███████████████████████████████████| 2000/2000 [00:00<00:00, 23614.18it/s
```

```
train_data[:5]
```

```
[{'token': tensor([   1,  304,  110,  342, 1229,    2,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0])},
 {'token': tensor([  1,  54,  61, 161,  48, 251,  69, 443,   2,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0])},
 {'token': tensor([   1,  588,  665,  788, 1095,  831,   40, 1027,  405, 1120,
            2,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0])},
 {'token': tensor([   1,   99,  198,  336,  223, 1316,    2,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0])},
 {'token': tensor([   1, 1273,   59,    2,    0,    0,    0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
            0])}]
```

Custom PyTorch dataset and batching

- We define a `PytorchDataset` class to provide input-target token sequences for autoregressive training.
- We prepare the input and target such that the model predicts the next token given previous ones.

```
class PytorchDataset():
    def __init__(self, data, pad_vocab_id=0):
        self.data = data
        self.pad_tensor = torch.tensor([pad_vocab_id])

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

    def __getitem__(self, ind):
        # Retrieve the next sequence of tokens from the current index
        # by excluding the first token of the current sequence and appending a
        target_sequence = torch.cat([self.data[ind]['token'][1:], self.pad_ten
        return self.data[ind]['token'], target_sequence
```

```
train_dataset = PytorchDataset(train_data)
test_dataset = PytorchDataset(test_data)
train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=50, shuffle=False)
```

Now let’s get a batch of data from DataLoader

```
train_text, train_target = next(iter(train_dataloader))
train_text = train_text.to(device)
train_text.shape
```

```
torch.Size([64, 25])
```

```
train_text[0]
```

```
tensor([   1,   48,  267,  645,  113,  968, 1491, 1897,    2,    0,    0,    0
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0
           0], device='mps:0')
```

```
train_target[0]
```

```
tensor([  48,  267,  645,  113,  968, 1491, 1897,    2,    0,    0,    0,    0
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0
           0])
```

```
tokenizer_wrapper.decode(train_text[0].tolist())
```

```
'[CLS] carrot apple chicken nuggets [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
```

```
tokenizer_wrapper.decode(train_target[0].tolist())
```

```
'carrot apple chicken nuggets [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
```

The target is shifted one position to the left for autoregressive training.

Transformer Decoder Model

We are now ready to define a transformer-based decoder-only model with positional encoding to generate text.

Let's begin with positional encoding. Transformers don't have any built-in notion of word order (unlike RNNs), so we need to explicitly tell the model the position of each word in the sequence.

In the interest of time, we won't dive deep into the math, but we'll use a standard implementation inspired by the [Attention is all you need](#) paper.

The code below adds these position signals to token embeddings so the model can learn not just what the tokens are, but where they appear in the sequence.

```
# The PositionalEncoding model is already defined for you. Do not change this
# We'll use this class in this exercise as well as the next exercise.

class PositionalEncoding(nn.Module):
    """
    Implements sinusoidal positional encoding as described in "Attention is Al

    Args:
        d_model (int): Dimension of the embedding space.
        dropout (float): Dropout rate after adding positional encodings.
        max_len (int): Maximum length of supported input sequences.
    """
    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Create a (max_len, 1) position tensor: [[0], [1], ..., [max_len-1]]
        positions = torch.arange(max_len).unsqueeze(1)

        # Compute the scaling terms for each dimension (even indices only)
        scale_factors = torch.exp(torch.arange(0, d_model, 2) * (-math.log(100

        # Initialize the positional encoding matrix with shape (max_len, 1, d_
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(positions * scale_factors)  # Apply sine to
        pe[:, 0, 1::2] = torch.cos(positions * scale_factors)  # Apply cosine

        # Register as buffer (not a trainable parameter)
        self.register_buffer("pe", pe)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Adds positional encoding to the input tensor.

        Args:
            x (torch.Tensor): Input tensor of shape (seq_len, batch_size, d_mo

        Returns:
            torch.Tensor: Tensor with positional encoding added.
        """
        seq_len = x.size(0)
        x = x + self.pe[:seq_len]
        return self.dropout(x)
```

Model architecture

Now we're ready to define our model architecture! It's going to include several key components that work together to generate text one token at a time:

- `nn.Embedding layer`: turns token IDs into dense vector representations.
- `PositionalEncoding`: adds information about the position of each token in the sequence.
- `TransformerDecoder`: the core of the model that processes the input using attention mechanisms.
- Causal mask: ensures the model only attends to earlier positions when generating text, so it doesn't "peek ahead".
- Output layer (`nn.Linear`): maps decoder outputs to vocab logits so we can predict the next token.
- Weight initialization: helps the model start training with reasonable values instead of random chaos.

We'll walk through each part step by step in the code below.

```
class RecipeGenerator(nn.Module):
    def __init__(self, d_model, n_heads, num_layers, vocab_size, device, dropo
        """
        Initialize the RecipeGenerator which uses a transformer decoder archit
        for generating recipes.

        Parameters:
            d_model (int): The number of expected features in the encoder/deco
            n_heads (int): The number of heads in the multiheadattention model
            num_layers (int): The number of sub-decoder-layers in the transfor
            vocab_size (int): The size of the vocabulary.
            device (torch.device): The device on which the model will be train
            dropout (float): The dropout value used in PositionalEncoding and 
        """
        super(RecipeGenerator, self).__init__()
        self.d_model = d_model
        self.device = device
        # Embedding layer for converting input text tokens into vectors
        self.text_embedding = nn.Embedding(vocab_size , d_model)

        # Positional Encoding to add position information to input embeddings
        self.pos_encoding = PositionalEncoding(d_model=d_model, dropout=dropou

        # Define the Transformer decoder
        decoder_layer=nn.TransformerDecoderLayer(d_model=d_model, nhead=n_head
        self.TransformerDecoder = nn.TransformerDecoder(
            decoder_layer,
            num_layers=num_layers
        )

        # Final linear layer to map the output of the transformer decoder to v
        self.linear_layer = nn.Linear(d_model, vocab_size)


        # Initialize the weights of the model
        self.init_weights()

    def init_weights(self):
        """
        Initialize weights of the model to small random values.
        """
        initrange = 0.1
        self.text_embedding.weight.data.uniform_(-initrange, initrange)
        self.linear_layer.bias.data.zero_()
        self.linear_layer.weight.data.uniform_(-initrange, initrange)

    def forward(self, text):
        # Get the embeded input
        encoded_text = self.embed_text(text)

        # Get transformer output
        transformer_output = self.decode(encoded_text)

        # Final linear layer (unembedding layer)
        return self.linear_layer(transformer_output)
```

```
    def embed_text(self, text):
        embedding = self.text_embedding(text) * math.sqrt(self.d_model)
        return self.pos_encoding(embedding.permute(1, 0, 2))

    def decode(self, encoded_text):
        # Get the length of the sequences to be decoeded. This is needed to ge
        seq_len = encoded_text.size(0)
        causal_mask = self.generate_mask(seq_len)
        dummy_memory = torch.zeros_like(encoded_text)
        return self.TransformerDecoder(tgt=encoded_text, memory=dummy_memory,

    def generate_mask(self, size):
        mask = torch.triu(torch.ones(size, size, device=self.device), 1)
        return mask.float().masked_fill(mask == 1, float('-inf'))
```

```
import torch
size = 10
mask = torch.triu(torch.ones(size, size), 1)
mask.float().masked_fill(mask == 1, float('-inf'))
```

```
tensor([[0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],
        [0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf],
        [0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf, -inf],
        [0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf, -inf],
        [0., 0., 0., 0., 0., -inf, -inf, -inf, -inf, -inf],
        [0., 0., 0., 0., 0., 0., -inf, -inf, -inf, -inf],
        [0., 0., 0., 0., 0., 0., 0., -inf, -inf, -inf],
        [0., 0., 0., 0., 0., 0., 0., 0., -inf, -inf],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., -inf],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

Let’s instantiate our model.

Let’s instantiate the model

```
# Define the hyperparameters and initalize the model. Feel free to change thes
d_model = 256
n_heads = 4
num_layers = 8
model = RecipeGenerator(d_model=d_model, n_heads=n_heads, num_layers=num_layer
```

Model Training

We define the loss function and optimizer and train the model using cross-entropy loss while applying gradient clipping.

```
train_text
```

```
tensor([[   1,   48,  267,  ...,    0,    0,    0],
        [   1,   56, 1135,  ...,    0,    0,    0],
        [   1,  142,  488,  ...,    0,    0,    0],
        ...,
        [   1,  693,  970,  ...,    0,    0,    0],
        [   1,  684,  685,  ...,    0,    0,    0],
        [   1,   14,  427,  ...,    0,    0,    0]], device='mps:0')
```

```
train_text.shape
```

```
torch.Size([64, 25])
```

```
# pass inputs to your model
output = model(train_text)
output.shape
```

```
torch.Size([25, 64, 3699])
```

```
vocab_size
```

```
3699
```

```
def trainer(
    model,
    criterion,
    optimizer,
    train_dataloader,
    test_dataloader,
    epochs=5,
    patience=5,
    clip_norm=1.0
):
    """
    Trains and evaluates the transformer model over multiple epochs using the

    Args:
        model: The Transformer model to train.
        criterion: Loss function (e.g., CrossEntropyLoss).
        optimizer: Optimizer (e.g., Adam).
        train_dataloader: DataLoader for training data.
        test_dataloader: DataLoader for validation data.
        epochs: Number of training epochs.
        patience: Early stopping patience — stop if validation loss increases
        clip_norm: Maximum norm for gradient clipping to avoid exploding gradi

    Returns:
        train_losses: List of average training losses for each epoch.
        test_losses: List of average test losses for each epoch.
    """

    train_losses = []
    test_losses = []
    early_stopping_counter = 0

    for epoch in range(epochs):
        # Training phase
        model.train()
        total_train_loss = 0

        for batch_inputs, batch_targets in train_dataloader:
            # Move inputs and targets to the correct device (GPU or CPU)
            batch_inputs, batch_targets = batch_inputs.to(device), batch_targe

            optimizer.zero_grad()

            # Forward pass
            predictions = model(batch_inputs)  # shape: (seq_len, batch_size,
            predictions = predictions.permute(1, 2, 0)  # shape: (batch_size,

            loss = criterion(predictions, batch_targets)
            loss.backward()

            # Clip gradients to prevent exploding gradients
            torch.nn.utils.clip_grad_norm_(model.parameters(), clip_norm)

            optimizer.step()
            total_train_loss += loss.item()
```

```
        avg_train_loss = total_train_loss / len(train_dataloader)
        train_losses.append(avg_train_loss)

        # Evaluation phase
        model.eval()
        total_test_loss = 0

        with torch.no_grad():
            for batch_inputs, batch_targets in test_dataloader:
                batch_inputs, batch_targets = batch_inputs.to(device), batch_t

                predictions = model(batch_inputs).permute(1, 2, 0)
                loss = criterion(predictions, batch_targets)

                total_test_loss += loss.item()

        avg_test_loss = total_test_loss / len(test_dataloader)
        test_losses.append(avg_test_loss)

        print(f"Epoch {epoch+1}: Train Loss = {avg_train_loss:.4f}, Test Loss

        # Early stopping check
        if epoch > 0 and avg_test_loss > test_losses[-2] * (1 + 1e-5):
            early_stopping_counter += 1
        else:
            early_stopping_counter = 0

        if early_stopping_counter >= patience:
            print(f"Early stopping triggered at epoch {epoch+1}")
            break

    return train_losses, test_losses
```

```
# Define the optimizer and the loss function. Feel free to change the hyperpar

num_epoch = 20
clip_norm = 1.0
lr = 5e-5

optimizer = torch.optim.Adam(model.parameters(), lr=lr)
criterion = torch.nn.CrossEntropyLoss(ignore_index=0) # Ignore the padding ind
train_losses, test_losses = trainer(model, criterion, optimizer,train_dataload
```

```
Epoch 1: Train Loss = 6.9585, Test Loss = 6.3770
Epoch 2: Train Loss = 6.0016, Test Loss = 5.6101
Epoch 3: Train Loss = 5.3564, Test Loss = 5.1428
Epoch 4: Train Loss = 4.9545, Test Loss = 4.8622
Epoch 5: Train Loss = 4.6845, Test Loss = 4.6780
Epoch 6: Train Loss = 4.4856, Test Loss = 4.5558
Epoch 7: Train Loss = 4.3282, Test Loss = 4.4594
Epoch 8: Train Loss = 4.1998, Test Loss = 4.3868
Epoch 9: Train Loss = 4.0888, Test Loss = 4.3136
Epoch 10: Train Loss = 3.9922, Test Loss = 4.2542
Epoch 11: Train Loss = 3.9038, Test Loss = 4.2114
Epoch 12: Train Loss = 3.8230, Test Loss = 4.1818
Epoch 13: Train Loss = 3.7493, Test Loss = 4.1433
Epoch 14: Train Loss = 3.6828, Test Loss = 4.1118
Epoch 15: Train Loss = 3.6258, Test Loss = 4.0892
Epoch 16: Train Loss = 3.5637, Test Loss = 4.0803
Epoch 17: Train Loss = 3.5074, Test Loss = 4.0499
Epoch 18: Train Loss = 3.4555, Test Loss = 4.0367
Epoch 19: Train Loss = 3.4072, Test Loss = 4.0278
Epoch 20: Train Loss = 3.3545, Test Loss = 4.0141
```

Recipe Generation

We generate a new recipe by sampling tokens one by one from the trained model.

```python
def generate_recipe(model, device, max_recipe_length=39, seed=[206], end_vocab
    """
    Generates a recipe for an image using the specified model and device.

    Parameters:
        model (torch.nn.Module): The trained model used for generating tokens.
        device (torch.device): Device to run the model on.
        max_recipe_length (int): Maximum number of tokens to generate.
        seed (list[int]): A list of one or more token IDs to start generation 
        end_vocab (int): Token ID that indicates the end of the sequence.

    Returns:
        numpy.ndarray: A 1D array of token IDs representing the generated reci
    """
    # Ensure seed is a list and convert to tensor of shape [1, len(seed)]
    context = torch.tensor([seed], device=device)

    # Generate tokens until max length or end token is reached
    for _ in range(max_recipe_length - len(seed)):  # subtract len(seed) to ca
        logits = model(context)[-1]  # Get logits for the last position
        probabilities = torch.softmax(logits, dim=-1).flatten(start_dim=1)
        next_vocab = torch.multinomial(probabilities, num_samples=1)
        context = torch.cat([context, next_vocab], dim=1)
        if next_vocab.item() == end_vocab:
            break

    return context.cpu().numpy().flatten()
```

```python
recipe = generate_recipe(model, device, max_recipe_length=20)
```

```python
generated_recipe = tokenizer_wrapper.decode(recipe)
generated_recipe
```

```
'chocolate chip chocolate frosting [SEP]'
```

The generation quality might not be great but the purpose here is to demonstrate different components involved in text generation using transformers.


chocolate...