# Build your own GPT

## Dataset Preparation for Training Language Model

Preparing a dataset for language model training involves three steps: preprocessing, which cleans and standardizes text; tokenization, which transforms text into numerical indices; and batching, which groups sequences to enhance training efficiency.

```
import torch
import re

# Step 1: Preprocessing - Clean and standardize text
text = "Hello, World! Welcome to Language Modeling."
text = re.sub(r'[^a-zA-Z\s,.:;?_&$!"()\-\*\[\]]', '', text.lower()).strip()

# Step 2: Tokenization - Convert text into numerical indices
chars = sorted(list(set(text)))
stoi = {ch: i for i, ch in enumerate(chars)}
encoded_text = [stoi[c] for c in text]

# Step 3: Batching - Group sequences for efficient training
batch_size, block_size = 4, 8
data = torch.tensor(encoded_text, dtype=torch.long)
batched_data = [data[i:i+block_size] for i in range(0, len(data), block_size)]
```

## Encoding and Decoding

Encoding functions convert text into lists of indices for efficient tokenization, while decoding functions revert indices back to text. In a bigram model, this is done using `stoi` (string to index) and `itos` (index to string) dictionaries.

```
# Create mappings for encoding and decoding
chars = sorted(list(set("sample text for encoding")))
stoi = {ch: i for i, ch in enumerate(chars)}
itos = {i: ch for i, ch in enumerate(chars)}

# Encoding function: Convert text to indices
encode = lambda s: [stoi[c] for c in s]

# Decoding function: Convert indices back to text
decode = lambda l: ''.join([itos[i] for i in l])

# Example usage
encoded = encode("sample")
decoded = decode(encoded)

print("Encoded:", encoded)
print("Decoded:", decoded)
```

## Attention Mechanism

The attention mechanism helps transformers identify important words in a sentence. It uses matrix multiplication and the softmax function to assign weights to words, combining their embeddings based on importance and context. This allows the model to understand relationships between words, even those far apart in the text.

```
import torch
import torch.nn.functional as F

B, T, C = 2, 5, 4  # Batch size, Sequence length, Embedding dimension
x = torch.randn(B, T, C)  # Random input embeddings

# Define query, key, and value transformations
query = x  # Using the input embeddings as queries
key = x    # Using the input embeddings as keys
value = x  # Using the input embeddings as values

# Step 1: Calculate attention scores using matrix multiplication
attention_scores = torch.matmul(query, key.transpose(-2, -1))  # Shape: (B, T, T)

# Step 2: Apply softmax to convert scores into attention weights
attention_weights = F.softmax(attention_scores, dim=-1)  # Shape: (B, T, T)

# Step 3: Use the attention weights to combine the value embeddings
output = torch.matmul(attention_weights, value)  # Shape: (B, T, C)

# Output the results
print("Attention Weights:\n", attention_weights)
print("Output Embeddings:\n", output)
```

## Self-Attention Mechanism in Transformers

Self-attention in transformers uses key, query, and value projections along with positional embeddings. This combination creates context-aware representations that capture both the relationships between tokens and the overall structure of the sequence.

```
import torch
import torch.nn.functional as F

B, T, C = 2, 5, 4  # Batch size, Sequence length, Embedding dimension
x = torch.randn(B, T, C)  # Random input embeddings

# Positional embeddings: Simulating positional information added to x
positional_embeddings = torch.randn(B, T, C)
x = x + positional_embeddings  # Adding positional information to embeddings

# Using x as the query, key, and value for simplicity
query = x
key = x
value = x

# Step 1: Calculate attention scores manually
scores = (query * key).sum(dim=-1) / torch.sqrt(torch.tensor(C, dtype=torch.float32))

# Step 2: Apply softmax to get attention weights
attention_weights = F.softmax(scores, dim=-1)

# Step 3: Compute the output by combining the value embeddings with attention weights
output = (attention_weights.unsqueeze(-1) * value).sum(dim=1)
```

```
# Output the results
print("Attention Weights:\n",
attention_weights)
print("Self-Attention Output:\n", output)
```

## Positional Encoding

Positional encodings provide information about the order of tokens in the input sequence, which is crucial because transformers do not process data sequentially. By adding positional encodings to token embeddings, the model can capture the structure and relationships within a sentence, aiding in tasks like predicting the next token in a bigram model.

```
import numpy as np

def generate_positional_embeddings(T, C):
    # Initialize positional embeddings
with zeros
    positional_embeddings = np.zeros((T,
C))
    # Calculate the embeddings using sine
and cosine functions
    for t in range(T):
        for c in range(C):
            if c % 2 == 0:
                positional_embeddings[t,
c] = np.sin(t / (10000 ** (c / C)))
            else:
                positional_embeddings[t,
c] = np.cos(t / (10000 ** (c / C)))
    return positional_embeddings

# Call the function with T=10 and C=4
positional_embeddings =
generate_positional_embeddings(10, 4)
print("Positional Embeddings:\n",
positional_embeddings)
```

codecademy

### Feedforward Layers

Feedforward layers in transformers are similar to those in standard neural networks. They apply linear transformations followed by non-linear activation functions, like ReLU, to token representations. This step enhances the model's ability to learn complex patterns before passing data to the attention layers.

```
import torch
import torch.nn.functional as F

B, C = 3, 8  # Batch size, Embedding dimension
x = torch.randn(B, C)  # Random input token representations

# Define a feedforward layer with a linear transformation and ReLU activation
linear_layer = torch.nn.Linear(C, C)
output = F.relu(linear_layer(x))

# Output the results
print("Input Token Representations:\n", x)
print("\nOutput After Feedforward Layer:\n", output)
```

## Layer Normalization

Layer normalization stabilizes the training of transformer models by normalizing token embeddings. This process helps improve convergence and overall model performance by ensuring consistent distribution of input values to each layer.

```
import torch
import torch.nn as nn

B, T, C = 3, 5, 4  # Batch size, Sequence length, Embedding dimension
x = torch.randn(B, T, C)  # Random input token embeddings

# Apply layer normalization
layer_norm = nn.LayerNorm(C)
normalized_x = layer_norm(x)

# Output the results
print("Original Token Embeddings:\n", x)
print("\nNormalized Token Embeddings:\n", normalized_x)
```

## Single-Head vs. Multi-Head Attention

Single-head attention computes one set of attention scores, capturing only one type of relationship at a time. Multi-head attention uses multiple attention heads to capture various relationships within the input sequence, allowing the model to understand complex dependencies more effectively.

Print