Some notes on the high level structure of the nanoGPT model

Main files of nanoGPT

- **prepare.py**
- Output: training and validation datasets, as arrays of integers

- **train.py**
- Controls the training phase, and trains the model defined in model.py

- **model.py**
- Defines the GPT model as PyTorch module

- **sample.py**
- After the model is trained, generate new text

Notes NOT focused on

- Layer Norm, Dropout and other regularization and optimization layers and techniques
- DDP / MultiGPU
- Just focused on the core layers and parts, training on a single GPU or CPU, with the default text file

Key ideas

- It is a decoder only network
- It is a character level model, meaning 1 token = 1 character
- It is a sequence to sequence model, meaning that it generates probability distributions for SEQUENCE_LENGTH tokens as the output

Configuration parameters

- VOCABULARY_SIZE = 65
  - Number of different tokens (characters) in the input file
- SEQUENCE_LENGTH = 256 (block size, context)
  - Max window to be used for context
- NUMBER_OF_HEADS = 6
  - Each head learns on its own subspace
- BATCH_SIZE = 64
  - How many batches of input data are processed at the same time
- EMBEDDING_SIZE = 384
  - Each token is represented as a floating point vector of 384 dimensions
- NUMBER_OF_GPT_BLOCK_LAYERS = 6
  - Number of sequential GPT blocks

prepare.py

- Prepares the data for training
- Converts input text to arrays of integers

Training data

- Each character in the vocabulary is represented by an integer (1… VOCABULARY_SIZE)

- Each of these integers has an associated floating point vector of 384 dimensions (EMBEDDING SIZE)

train.py

- Trains the model defined in model.py, using the data prepared by prepare.py

Single iteration

Input: Batch size x sequence length

Output: Batch size x sequence length

Randomized portions of data in each iteration

- Note that in PyTorch linear layers need to match only in the last dimension with the input tensor

Training

- In a loop:
  - get_batch(X,Y)
    - Obtain training data (X) and targets (Y)
  - Calculate logits and loss
    - Calculate logits, apply softmax, use cross-entropy to calculate loss
  - Do backpropagation
    - Compute gradients
  - Do an optimizier step
    - Update weights

model.py

- Contains core PyTorch logic of the GPT module

High level structure of GPT module

- **wte** – token embeddings
- **wpe** – positional embeddings

- Then these two get added together

- After that, a series (6) of GPT blocks follows. Output of one GPT block is connected to input of the next one

- And, after the output of the last block there is one final linear layer (**lm_head**), which produces logits (pre-softmax probability distributions for output tokens, for sequence_length tokens every time in training mode). In inference mode, only the last token is extracted and considered, as it is the one being predicted

Single block structure

- Single block is a Self Attention Layer followed by a MLP (MultiLayer Perceptron)

- Last GPT block's MLP is connected to the final linear layer (called **lm_head**)

Single self-attention cell

- **Q**, **K**, **V** - extracted from a linear layer (**c_attn**) – size = 3x embedding_size
- GPT heads partition the incoming data in the embeddings dimension, and each head handles EMBEDDING_SIZE / NUMBER_OF_HEADS of elements in that dimension. By default parameters, that's 384 / 6
- At each head:
  - Masked fill is applied after Q @ K matrix multiplication
  - Softmax applied after masked fill
  - Matrix multiplication by V applied last
- PyTorch autograd tracks all calculations so it can later compute gradients

Output of the GPT module

- Logits are the output. In training mode, cross entropy is applied to them so loss can be computed

- Dimensionality of the output in training mode:
- [64, 256, 65]
- *[BATCH_SIZE, SEQUENCE_LENGTH, VOCABULARY_SIZE]*

- In inference mode, only the last token is considered, so dimensions are:
- [1,1,65]

sample.py

- Generates new characters and texts (samples from the model)

Sampling (inference mode)

- Generates *num_samples* tokens by calling the generate method of the GPT module
- *generate()* method returns logits for the last token in the sequence (next word)
- Softmax is applied to the logits, and then the next token is picked using multinomial distribution
- After the next token is determined, it is appended to the input sequence