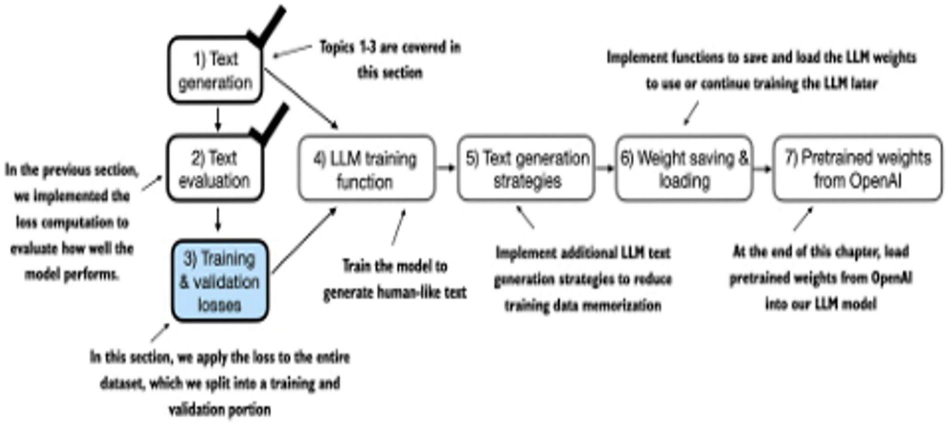
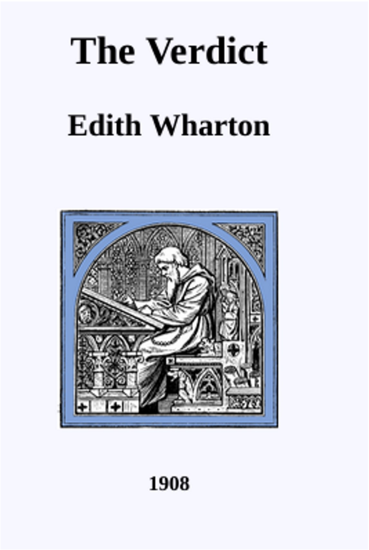
GPT2 Model : Calculating the training and validation set losses


1) Text generation
Topics 1-3 are covered in this section
Implement functions to save and load the LLM weights to use or continue training the LLM later
In the previous section, we implemented the loss computation to evaluate how well the model performs.
2) Text evaluation
4) LLM training function
5) Text generation strategies
6) Weight saving & loading
7) Pretrained weights from OpenAI
3) Training & validation losses
Train the model to generate human-like text
Implement additional LLM text generation strategies to reduce training data memorization
At the end of this chapter, load pretrained weights from OpenAI into our LLM model
In this section, we apply the loss to the entire dataset, which we split into a training and validation portion

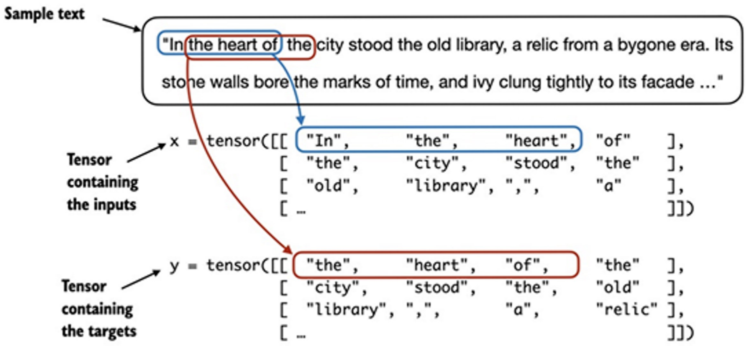


Tokenize (approx 5000 tokens)

Divide the dataset into training and Validation set

```
train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

Use dataloader to chunk training and validation data into input and output datasets.

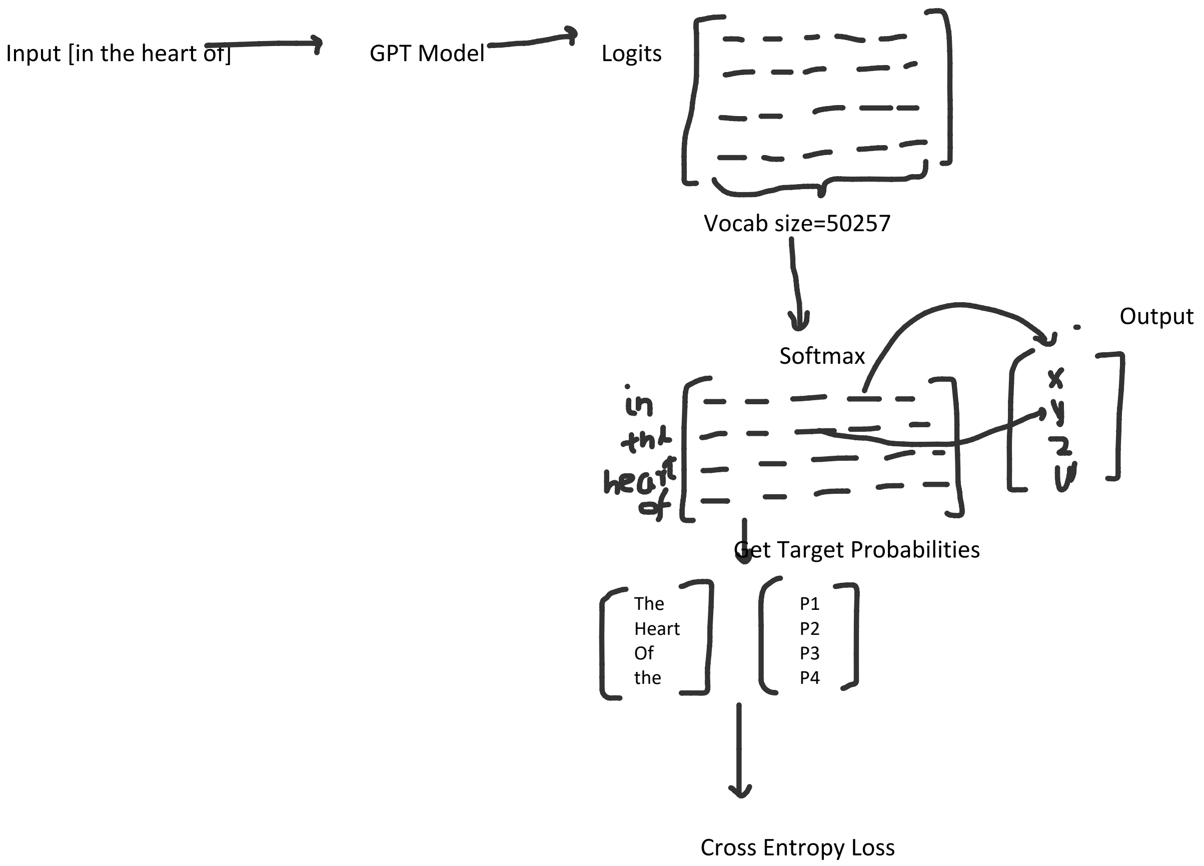


"In the heart of the city stood the old library, a relic from a bygone era. Its stone walls bore the marks of time, and ivy clung tightly to its facade …"

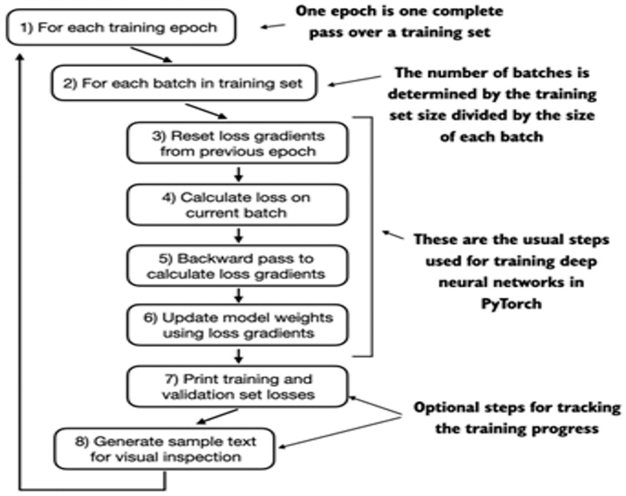
```
x = tensor([[ "In",      "the",     "heart",  "of"    ],
            [ "the",     "city",    "stood",  "the"   ],
            [ "old",     "library", ",",      "a"     ],
            [ …                                        ]])

y = tensor([[ "the",     "heart",   "of",     "the"   ],
            [ "city",    "stood",   "the",    "old"   ],
            [ "library", ",",       "a",      "relic" ],
            [ …                                        ]])
```

Validation

Training



Pretraining of LLM Model


1) For each training epoch
One epoch is one complete pass over a training set
2) For each batch in training set
The number of batches is determined by the training set size divided by the size of each batch
3) Reset loss gradients from previous epoch
4) Calculate loss on current batch
5) Backward pass to calculate loss gradients
6) Update model weights using loss gradients
These are the usual steps used for training deep neural networks in PyTorch
7) Print training and validation set losses
8) Generate sample text for visual inspection
Optional steps for tracking the training progress

1) Embedding Parameters : 50257*768 + 1024*768 =38.4 M

2) Multi Head Attention : Q,K,V : 3*768*768 =1.77M
   Output Head = 768*768 = 0.59 M

3) Feed Forward Neural Network : 768*4*768 + 768*4*768 = 4,718,592
4) Total 12 Transformer blocks : 12*(1.77+0.59+4.72) = 84.2 M
5) Softmax = 768*50257 = 38597376

Hence Total Parameters = approx 161 M

Why is it 124M and not 164M?

The model's parameter count might be different from the idealized calculation for a couple of reasons:

1. **Tying of the Embedding and Output Layer**: In some implementations of GPT, the input and output embedding layers are tied, meaning that the parameters for the output embedding are not counted separately. This would reduce the total number of parameters.
   - If the output embedding is tied to the input embedding, you effectively reduce the embedding parameter count by V×dmodelV \times d_{model}V×dmodel, or 50,000 \times 768.
   - Reducing this from the previous embedding parameters:
     76,800,000−(50,000×768)=76,800,000−38,400,000=38,400,00076,800,000 - (50,000 \times 768) = 76,800,000 - 38,400,000 = 38,400,00076,800,000−(50,000×768)=76,800,000−38,400,000=38,400,000
     The total parameter count with tied embeddings:
     Total Parameters with tied embeddings=38,400,000+28,311,552+56,623,104=123,334,656\text{Total Parameters with tied embeddings} = 38,400,000 + 28,311,552 + 56,623,104 = 123,334,656Total Parameters with tied embeddings=38,400,000+28,311,552+56,623,104=123,334,656
2. **Optimizations and Regularizations**: There could be further optimizations or regularization techniques in the implementation that reduce the number of parameters slightly compared to the full theoretical number.

Thus, the model ends up with about **124M** parameters when rounding to the nearest million, instead of the calculated **164M**. The exact difference is mainly due to specific architectural choices such as tying the embeddings and other model design decisions.