FEBRUARY 23, 2023

LET’S CREATE SHAKESPEARE WITH AN AI

NEGAR LALEH

The first step in every project is to have a clear goal in mind, in this paper the aim is to build an AI to create the works of Shakespeare letter by letter.

The libraries used in this code are request, for downloading the starting dataset, torch, and torch.nn, this module contains different classes that help you build neural network models.

After downloading the required dataset, all the unique characters used in our dataset are sorted into a list. looking at this list will help us to enhance our understanding of the tokenizing process.

As it can be seen, there are a total of 65 unique characters used in our dataset, so the characters will be numbered from 0 to 64. the translation from integer to character and vice-versa is done by the code below.

```
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: ''.join([itos[i] for i in l])
```

The encoder iterates over the characters given and creates a look-up table from character to integer and the decoder uses inverse mapping to turn the integers into characters.

The tokenizing code used is a very simple method, which is suitable for character-level tokenizing. There are other tokenizers such as hugging face and sentencepiece. these are generally used in subword level tokenizing.

The first 100 characters are turned into integers and shown using the torch. tensor command. This command is similar to the NumPy array and is just a generic n-dimensional array to be used for arbitrary numeric computation.

Since the desired outcome is to create works of Shakespeare and not just copy them, the dataset is split into train and validation sets. the validation process gives information that helps us tune the model's configurations.

To train our model random segments of our dataset are used. The length of these segments is defined by block_size.

The challenge of training our model to create a script, letter by letter is solved in the following example.

The act of prediction in this example is used in 8 positions, which requires 9 characters as the first character doesn’t need to be predicted. There are also 8 example sequences in each segment. paying attention to the numbers below, the 8 sequences are as follows:

12,12 1,12 1 29,12 1 29 36 etc.

Now the model can guess after token number 12 the next position might be 1, after the sequence of 12 and 1, there might be a 29.

1	2	3	4	5	6	7	8	9
12	1	29	36	9	45	56	25	48

The for loop will get the 8 sequences in each segment and print the probable next letter.

Using the multitasking ability of python, 64 independent sequences will be processed simultaneously. This number of sequences is indicated in batch_size.

The get_batch function generates 64 by 256 batches using random segments, which will be stacked upon each other using torch.stack command. If CUDA is being used, pay attention to loading the data into the device.

As its name suggests Estimate_loss unction evaluates the loss over multiple patches, by getting the average loss in both splits of train and validation.to reduce memory usage we use torch.no_grad(). This command means that the tensors with gradients currently attached to the current computational graph are now detached from the current graph and we no longer will be able to compute the gradients with respect to that tensor. Until the tensor is within the loop it is detached from the current graph. As soon as the tensor defined with gradient is out of the loop, it is again attached to the current graph.

Considering our challenge we need a model which approximates the probability of a word given all the previous words by using only the conditional probability of one preceding word. Hence, the bigram model, one of the N-gram language models, is used.

The module torch.nn contains different classes that help you build neural network models. All models in PyTorch inherit from the subclass nn.Module.

The next part of the challenge is to make these predicted letters into meaningful words. The solution to this part may be more complex than other parts of our problem.

Until now we can produce letters according to the last letter, however, the resulting words are not meaningful at all. This problem can be solved by making sure that each position can speak to its preceding positions. e.g. it's necessary for position 4 to be aware of positions 3,2 and 1. This can be achieved by doing weighted aggregations of our past elements by using matrix multiplication of a lower triangular fashion and then the one elements in the lower triangular matrix will tell us how much of each element fuses into each position. The weights are specified in a T-by-T array. In this way, the average of the preceding positions can be measured. The reason for using softmax in training is its differentiability and how it allows us to optimize a cost function. However, softmax has the tendency to sharpen towards the max number in the given weights, resulting in aggregating information from only one single node. To solve this we normalize the softmax by dividing it into the root square of the head size.

Going forward the demand for gathering past positions information in a data-dependent way quickly becomes apparent. Which, self-attention solves.

applying self-attention, every token emits two vectors by the names of the query and key. in a simple way, the former means "what am I looking for" and the latter conveys "what do I contain". to get an affinity between these two the query vector performs a dot product with every key of the positions, and the result will become the new weight. The closer the number of the preceding positions to the last one, the more interesting they are to each other. as to not change the value of x, which holds the private information of the current position, another variable called value(v) is built. straightforwardly, v notifies other tokens of what it will communicate to them.

Let's review the method of self-attention using figure 1.

For this given example the block_size is 6. As it can be seen our nodes represent the tokens. The first node is only pointed to by itself, the second node is pointed to by itself and the first node, all the way up to the 6th node, which is pointed to by itself and all the other nodes.

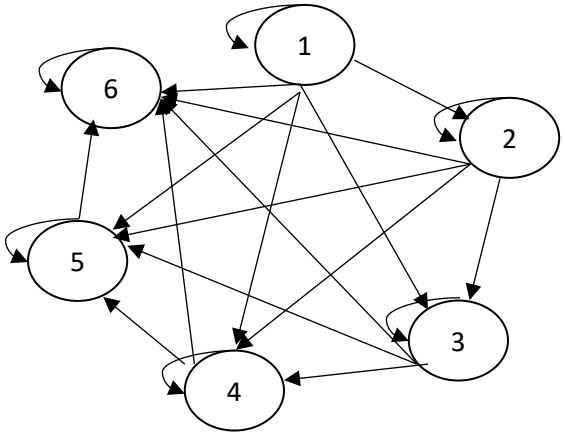

Figure 1: the relationship between the tokens in the improved self-attention code

Next, the head with the size of n_embd is constructed and after encoding the information with the token and position embeddings, it's fed into the self-attention head and its output is decoded, creating the logits. At this level, the output of gpt is more accurate and certain words are recognizable but it's still not enough.

The next step is to apply multiple attentions in parallel and concatenate the results. with this step, the rate of the gathered data is higher, which increases the communication between tokens.

Until now each token can receive data from the preceding tokens but don't have sufficient time to process these. which, is resolved by the forward function. this function adds a computation per node level to this network.

To intersperse the communication with the computation a class named block is formed, grouping these two and replicating them. unexpectedly the result is not as perfect as we imagined it to be. The reason is that we've made this model into a deep neuro network, notorious for suffering from optimization issues.

This is where the residual pathway comes in. The tokens can fork off, do their communication and computation, and get connected to the projection again with the second command of nn.linear() in the forward class. this operation is called the residual connection. Testing the code shows us that the training loss is getting ahead of the validation loss, which is a sign of overfitting.

Another option for optimizing is applying layer norms. Which is normalizing rows of every example. it's applied at the end of the transformer and exactly before the linear level. In this code n-layer, the number of block layers we'll have is introduced. We still have some overfitting to overcome. using dropout as a regulation technic is one way to reduce overfitting which can be caused by scaling up the model.

Three of the most important vectors in our model are B, T, and C, which respectably stand for Batch, Time, and Channel .to increase our model's accuracy, an intermediate phase is built so that we won't go directly

to the embedding of the logits. The n-embd is introduced here. This code will give us token embeddings instead of logits.

```
tok_emb = self.token_embedding_table(idx)
```

it is significant to not just encode the identity of our tokens (IDX) but also their positions. So each token knows which token from which position is communicating.

to make sure Idx get is not more than the block-size the input fed to it is cropped. if idx gets more than the block-size, our position embedding table is going to run out of stock. to get from token embeddings to logits a linear layer by the name of lm_head is formed.

the job of the generate function is to take a batch by time and make it b by t + 1,+2, etc….after getting the prediction, we turn the B T C array into a B by C one, by plucking out the last element in the time dimension. then asking torch.multinomial(), we get the probable next letter. since we need a single prediction for what comes next for now, we possess b by 1, by concatenating the integer samples we've gotten in idx_next, we have b by t+1 giving the model the token number of 0, and we ask it to predict the rest.

After using the Adam optimizer, applying a bigger batch of data, and sampling it, we evaluate the loss and zero out all the gradients from the previous step. The optimizer is updated with the gradients taken from parameters using loss. Backward () command.

Using CUDA, When generating the model and creating the context, the context should be created on the GPU. To able my system to run this code I needed to decrease the number of layers to 3, induce changing the n-embd to 192 using the formula below:

$$\text{n-embd} = \text{n-layer} * \text{batch\_size}$$