ECE 1786 Lecture #6

Work-in-Flight: Assignment 3 - Training & Using Transformer, due Mon Oct 21

- Team forming should be done, TA Mentors assigned, reached out?
- Project approval-in-principle form:https://forms.office.com/r/gHVaKngvds
- Due October 24;
- Lecture 7 will be video only, uploaded tomorrow to Quercus
- On October 22, I won't be here, but I've now asked the TA Mentors to come and consult with you that day. You can also reach out directly to them

Last Day: Core Mechanisms of Transformers & Assignment 3

- Note Formal Definitions paper posted with Lecture 5 - a mathematical description of the computation in Transformers, in formal notation

Today: 1. Language Generation using Transformers
2. Project Ideation, Approval-in-Principle and Proposal

Language Generation Using Transformers

Recall: A language model is trained to predict the next word that comes after an given input sequence of words.

So, if you can do that, then you can predict a whole sequence of output words, one at a time, by taking each predicted word, appending it to the input sequence of words & then predicting/generating the next word after that, and so on.

⇒ This is called "Auto-regressive" generation, a term that comes from the world of sequence predictions. (Not an obvious name..) See Jurafsky section 10.1

Here is the what I'd call the "Auto-regressive loop:" (maybe obvious, but is very important):

e.g. If the input started as: "The clean river flowed"
Call model to infer & generate next word: "into"

Then, the next input to the model is "The clean river flowed into"

Next word might be: "the"

Next input would be "The clean river flowed into the"
Generate next word, and so on. This is how chatGPT delivers what you ask for.

Each word is pretty expensive, in that it is a full inference run just to get one word, of a very large model.

Example Generation from GPT-2

See: https://banana-projects-transformer-autocomplete.hf.space/doc/gpt2-large (GPT2 Large only)

1. From GPT-2 medium (mid-sized GPT-2 in terms of # parameters).

Input: The clean river flowed
Generation Parameters (Unknown: i.e. Temp, Top-P, etc., max tokens generated)

Output: The clean river flowed in steady but strong currents.

After a week of hard-boiled fish hunt, all was tranquil again. They went out in a group on the small river and ate their dinner along the banks with some of the fish

2. From GPT-2 XL (largest GPT-2).

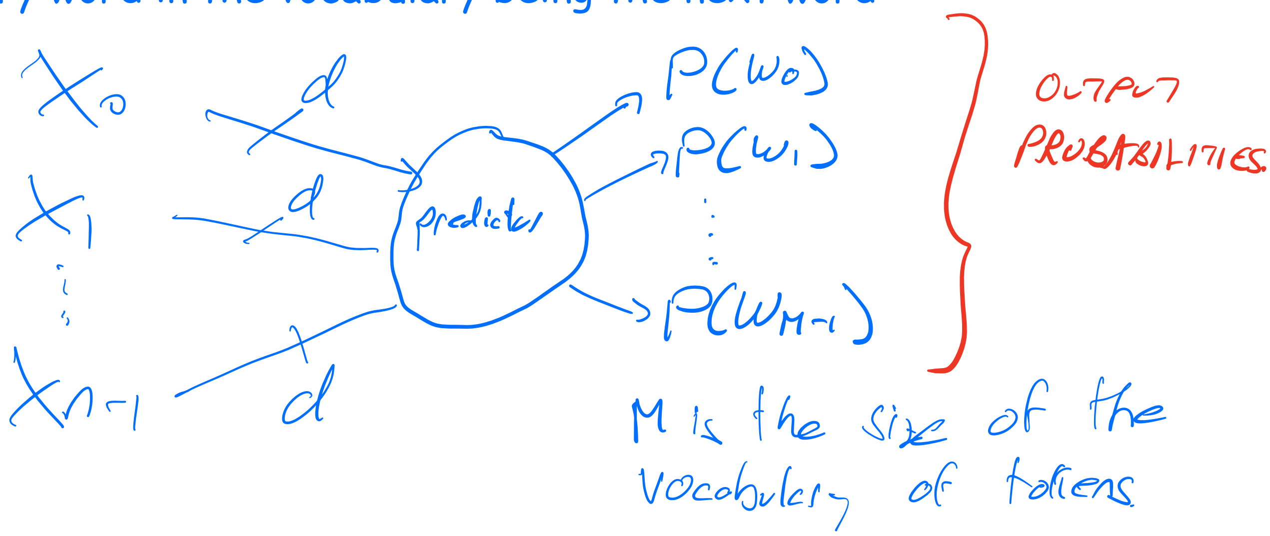
Input: The clean river flowed

Generation Parameters (Unknown)

Output: The clean river flowed. We walked on to the other side with the people we left behind. We found a small restaurant with a bench near the river — a small oasis at the end of the world, really.

In the evening we

Recall the specific input & output of the language model/transformer: given an input sequence of word embeddings, X0, X1,... Xn-1, output the probability of every word in the vocabulary being the next word:



So, for a given input sequence, which single word is selected as the output?

The process of selecting the word, based on the output probabilities, is called decoding. This word is an unfortunate choice, as it is confused with decoders that are quite something else in this field. (So make sure you pronounce the "ing" on decoding). We speak of the 'decoding' algorithm, i.e. given P(Wi), select one of the Wi as the output.

What do you think the best approach/decoding algorithm is?

This needs some careful thought. We don't just want the best next word, but the best sequence of output words. That does depend on the next word, but also what might be able to come after.

What makes the sequence best? I.e. what are its properties?
The full input/output sequence should be:

- Grammatical
- Make sense given the input context

Method 1: Greedy Decoding. Just select the highest probability word.

- You will see this in Assignment 3 Section 2
- Greedy does not work well in general - it picks obvious words, but these often lead to boring, uninteresting sequences of words; also repetitive
- Greedy may choose the most likely next word, but does not result in the most likely sequence of generated words
- Gets stuck in a highly local optimum in the space of all possible generated sequences

We can express this issue mathematically as follows: Given an input sequence of embeddings/words/tokens $X0 \ldots Xn-1$ we want the generated sequence of output words $Y0 \ldots Yg-1$ of g words to be the most likely sequence

i.e. we want $P(Y0) \times P(Y1) \times \ldots \times P(Yg-1)$ to be maximized.

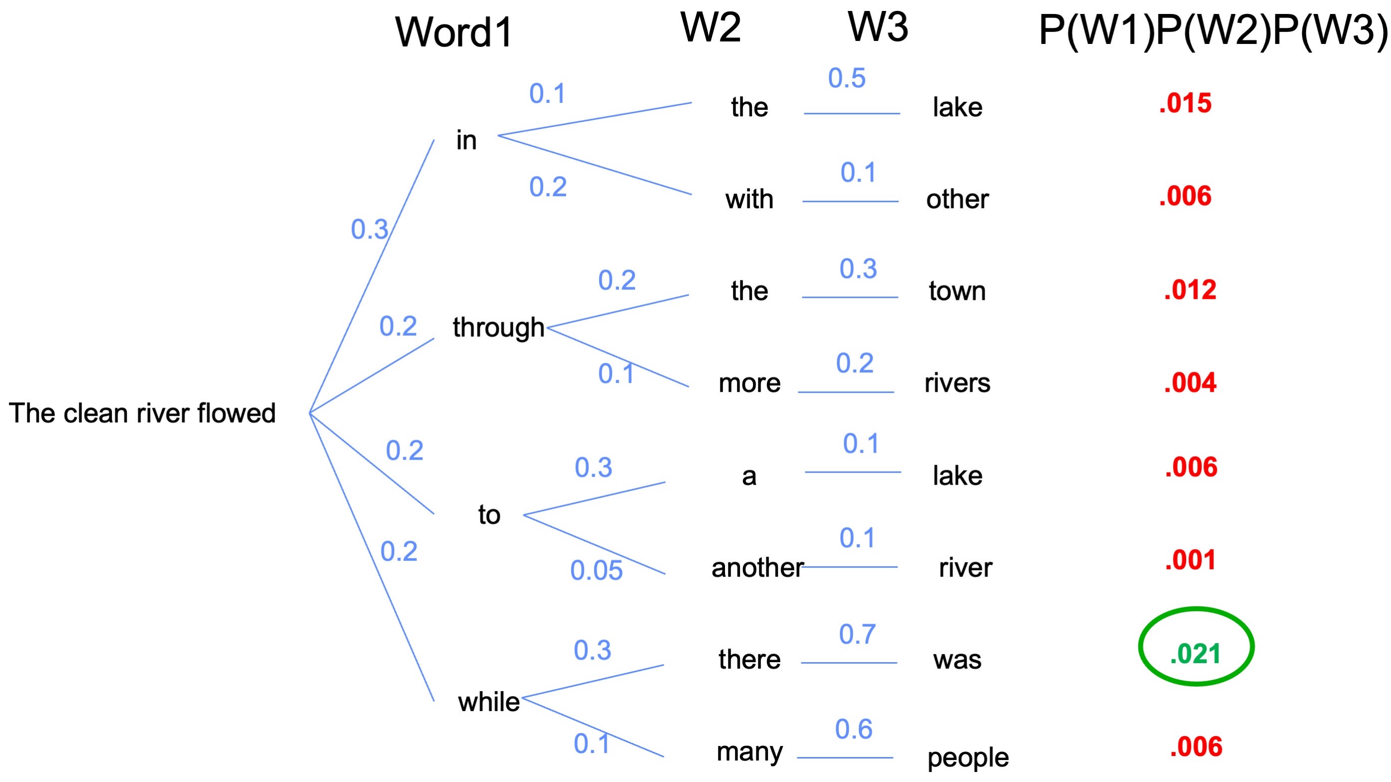
But, we don't know $P(Y1)$ when selecting $Y0$ (or Yj, $j>i$ when selecting Yi)

This is a hard problem because there are M**g possible sequences of g output words, which is a big exponential. Here M is the size of the vocabulary

- If M = 50000, g=20, 50000**20 is huge.
- Worse, each one of those is a full forward inference of the model!

As shown in Jurafsky Section 10.2, think of the auto-regressive selection of the output sequence as a tree with probabilities at each layer.

Input: the clean river flowed ….
Outputs, produced from many invocations of inference of the trained model:



Notice how the most probable Word 1 doesn't lead to the most probable sequence of Word 1, Word 2, Word 3. (.015 or .006 vs. .021)

- But to get the best probability sequence is very hard.

Method 2: Beam Search is a heuristic that prunes the full search tree a lot.

General Description:

- Walk down the tree, keeping the K-most probable sequences.
- At each level of the tree, consider top V possible next words for each of the K sequences. (Means you need K separate inferences instead of min 1)
- Compute the full sequence probability of those possible K x V sequences
  - Keep the K highest
- repeat until have generated the number of desired tokens (or hit stop token)

Method 3: Sampling (most commonly used)

Given: The set of output probabilities P(W0), P(W1), …. , P(WM-1)

Select: The next word through a random process, in which the probability of selecting word Wi is P(Wi) — how?

- In words: toss a many-sided weighted die, where the weights of each side (word) are the probability of the word.
- So, although the highest probability word is the most likely to be chosen, it isn't necessarily chosen - depends on the relative probabilities.

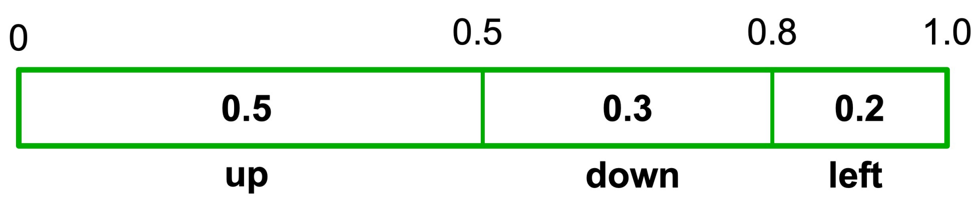
Illustration with an example:

Consider a 3-word vocabulary: up, down, left
Assume the neural network has generated these probabilities for next word:

- P(up) = 0.5
- P(down) = 0.3
- P(left) = 0.2

We want random process that selects up with probability 0.5, down with prob 0.3 and left 0.2

We can do that by visualizing the probabilities as shown below, and generating a uniformly distributed random number, R, between 0 and 1, inclusive:



If R <= 0.5 select up
If 0.5 < R <=0.8 choose down
If 0.8 < R <= 1.0 choose left

up will be chosen with probability 0.5, left with prob 0.2

This random process has a nice side-effect: if you don't like the output sequence that you get, you can just try again and get a new one.

Also, this process actually reflects the fact that there are many ways to answer a given question, or to create language.

However, this randomness is also part of the source of the 'hallucinations' that you've probably heard of from chatGPT/LLMs. Some bad luck on the first word could just send the answer in the wrong direction!!!!

This method is the most widely used; you'll be able to see parts of it on the GPT-3/4 playground, and in the code in Assignment 3, Section 2.

There are several variations to know about (& you'll see in the MinGPT code):

- Rather than select from all M tokens in the vocabulary, only select from the top K most probable words. (Called top-k sampling)
- More commonly used: top-p sampling: only select from the top words that all together have the sum of probabilities = p (or closest). 0 <=p <= 1
  - If set p = 1 that means use all M tokens/words
  - Often p = 0.8

- There is one more important adjustment to this process that is important:
  - The probabilities from the network output are adjusted to control whether the generated sequences are more or less creative/diverse.

- It is done with a parameter, t, called the Temperature
- A high T gives more diverse words, done by adjusting probs <u>before</u> sampling

- T = 1 is 'normal' - the probabilities are unchanged
- T > 1 makes less probable words more likely
- T < 1 makes more probable words more likely
- T = 0 makes the decoding greedy

The probabilities are adjusted during the Softmax output computation of the probabilities, as shown in this equation:

$$p\left(w_i\right) = \frac{\exp\left(\frac{li}{t}\right)}{\sum_{all\ i} \exp\left(\frac{li}{t}\right)}$$

Where the li are the logits produced by the network.

Other notes:

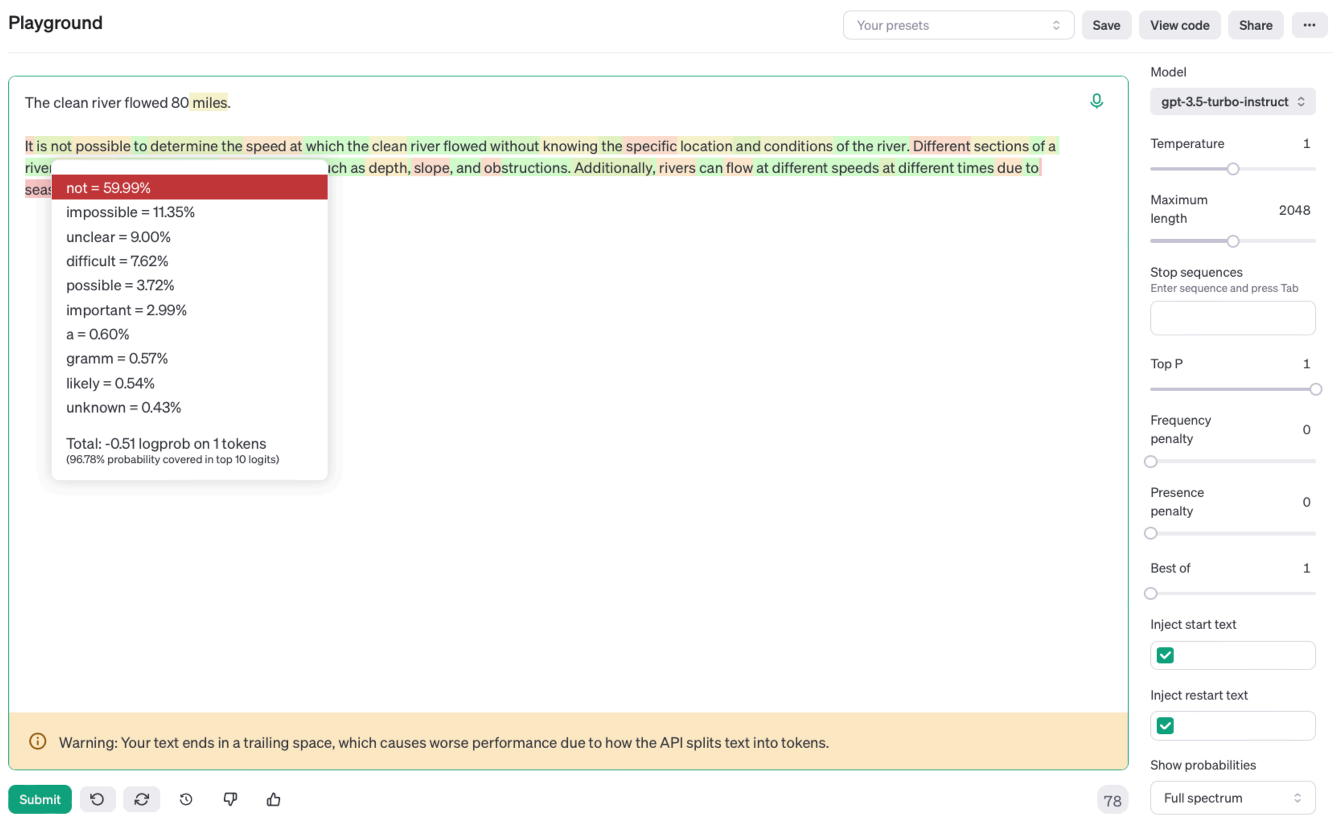
- Often a combination of top-p & temperature are the commonly used generation parameters
- There is also a repetition penalty - e.g. divide li by 1.3 if the token corresponding to li has already been used in this generation

DEMO of GPT-3/4 on playground (vs. ChatGPT) & parameters

- Turn on "Show probabilities" with the completion interface, effect of T, max tokens; show 'code' to generate using API; see https://platform.openai.com/playground/complete

Here is a screenshot of the legacy completion models from OpenAI that let one show the probabilities of the words sampled from at each word generated

- just one is being showed below, the alternative higher-probability words to the word 'not' after the words "It is ..."



Part II: Project Ideation and Scoping