CS229

HW 1: CalcGPT

Instead of old-fashioned custom-built algorithms for computing answers to arithmetic problems, we will design and test a system that uses a pre-trained large language model to solve these problems.

a) Method for encoding strings

```
encode_problems(X, strategy='baseline')
```

This function takes in an input tensor **X** and a **strategy** parameter, which can have two values: **'baseline'** or **'new'**. The **X** tensor is expected to have two columns, representing two operands in a mathematical addition operation.

If the **strategy** is set to **'baseline'**, then the function generates a simple string by concatenating the first and second element of each row in **X** separated by a **+** symbol, followed by an **=** symbol. For example, if the input tensor **X** is **[[2, 3], [4, 5]]**, the output strings generated by this function would be **[**`'2+3='`**,** `'4+5='`**]**.

If the **strategy** is set to **'new'**, then the function generates a more elaborate string that provides additional information about the problem. For example, if the input tensor **X** is **[[2, 3], [4, 5]]**, the output strings generated by this function would be `'if x=2 and y=3 then z=x+y is the summation of variable x and y. In this case, z= '` and `'if x=4 and y=5 then z=x+y is the summation of variable x and y. In this case, z= '` for **'new'** strategy.

The encoded strings are stored in a list called ***output_strings*** and returned as the output of the method.

b) Method for generating text

```
generate_text(model, tokenizer, prompts, verbose=True, device='cpu')
```

This function takes in the parameters: a **model** object, a **tokenizer** object, a list of **prompts**, a boolean **verbose** parameter (which is True by default), and a **device** parameter (which is set to **'cpu'** by default).

The purpose of this function is to generate text based on the given prompts using the provided **model** and **tokenizer**. Specifically, the function performs the following steps:

1. Tokenize the input prompts using the provided **tokenizer** and convert them into PyTorch tensors, and then move to the specified **device**.
2. Generate text using the provided **model** by passing the tokenized prompts as input. The text is generated using the **generate** method of the **model** object. The **LLM** used in the given code is **EleutherAI/gpt-neo-2.7B**, which is a pre-trained language model with 2.7 billion parameters, created by EleutherAI. This model is based on the **GPT** (Generative Pre-trained Transformer) architecture. The hyper-parameters used in the **generate_text** function are **max_length**, and **temperature**. **max_length** parameter is set to the length of the input prompt plus one, and the **temperature** parameter is set to **0.001**, which means that the generated text is deterministic and not very random. I have generated one new token after the given prompt and used

do_sample=False to get the deterministic output (most probable token for next position). Additionally, the function uses a batch size of 32 to generate the text efficiently.

3. Decode the output text generated by the **model** using the provided **tokenizer**. The decoded output is a list of strings representing the generated text.
4. If the **verbose** parameter is True, the function prints the example tokenization for write-up. Specifically, for each prompt in the last 10 prompts, the function prints the input and output tokens generated by the **tokenizer**.
5. The function returns the list of decoded output strings.

c) Method for decoding strings

```
decode_output(output_strings, strategy='baseline', verbose=True)
```

The **decode_output()** function takes a list of **output_strings** generated by the **generate_text()** function and decodes them to obtain the predicted output values for the corresponding input prompts.

This function first tokenizes each **output_string** using the **tokenizer** and selects the last token, which represents the predicted output value. If the token does not start with a digit, it is assumed that no output value was generated for the corresponding input prompt, and **NaN** is appended to the **y_hat** list. Otherwise, the token is converted to an integer and appended to the **y_hat** list.

```
Input: ['49', '+', '47', '=']
Output: ['49', '+', '47', '=', '0']

If the output_strings = ['5+2=7', '9+6=15', 'if x=7 and y=4 then z=x+y is the
summation of variable x and y. In this case, z=11'], the decode_output() will return
[7, 15, 11]
```

Because, I am only generating one token for both encoding, my decode string function works for both encoding.

d) Results

- ***Accuracy:***

Baseline	Compared approach
= 0.0148	= 0.0376

Improvement of compared approach is 2.51 times.

- ***Scatter Plot:***

Figure 1 illustrates the scatter plot for both encoding strategies. The left plot indicates that the LLM trained on the baseline encoding can only accurately predict the summation for a limited number of number combinations. Additionally, it seems that the first number in the range of 1-4 accounts for most correct answers. On the other hand, the right plot displays the performance of the LLM on the new encoding strategy. It can predict almost all of the summations when at

least one of the numbers includes '0'. However, it fails to predict the correct answer when neither of the numbers contains '0'.

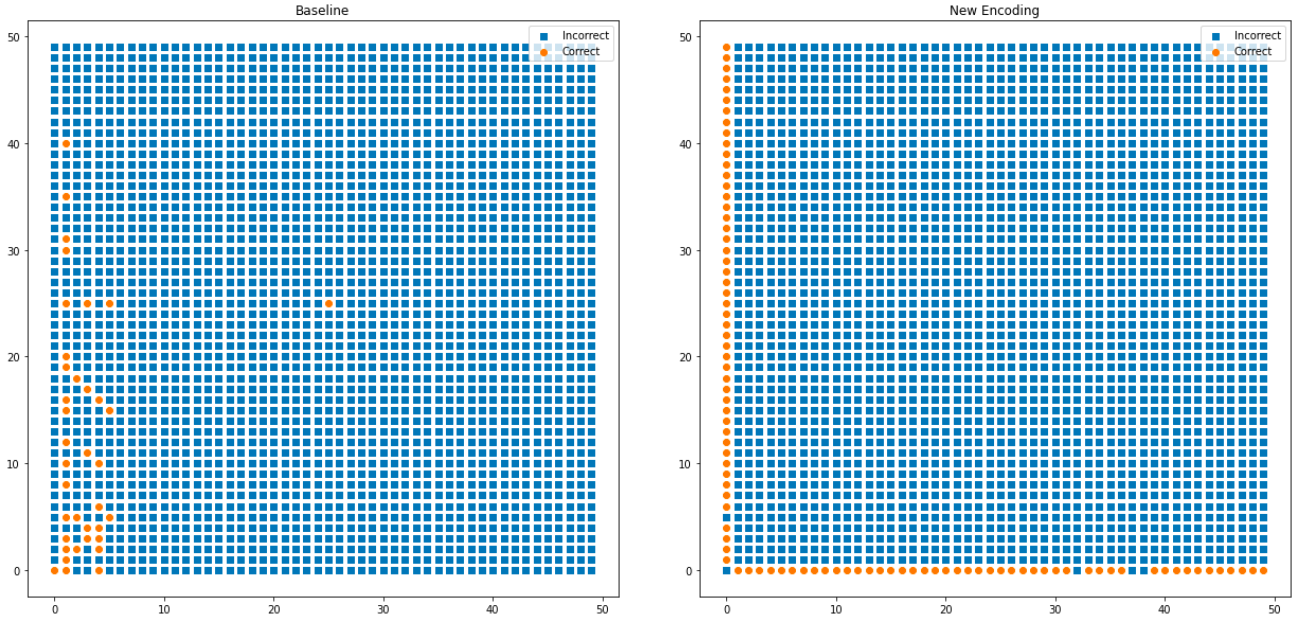


Figure 1: Scatter Plots

- ***Classifier Performance:***

I have used Decision Tree classifier to predict which sum problems is answered correctly. The accuracy for both approach is -

Baseline	Compared approach
98.27%	99.86%

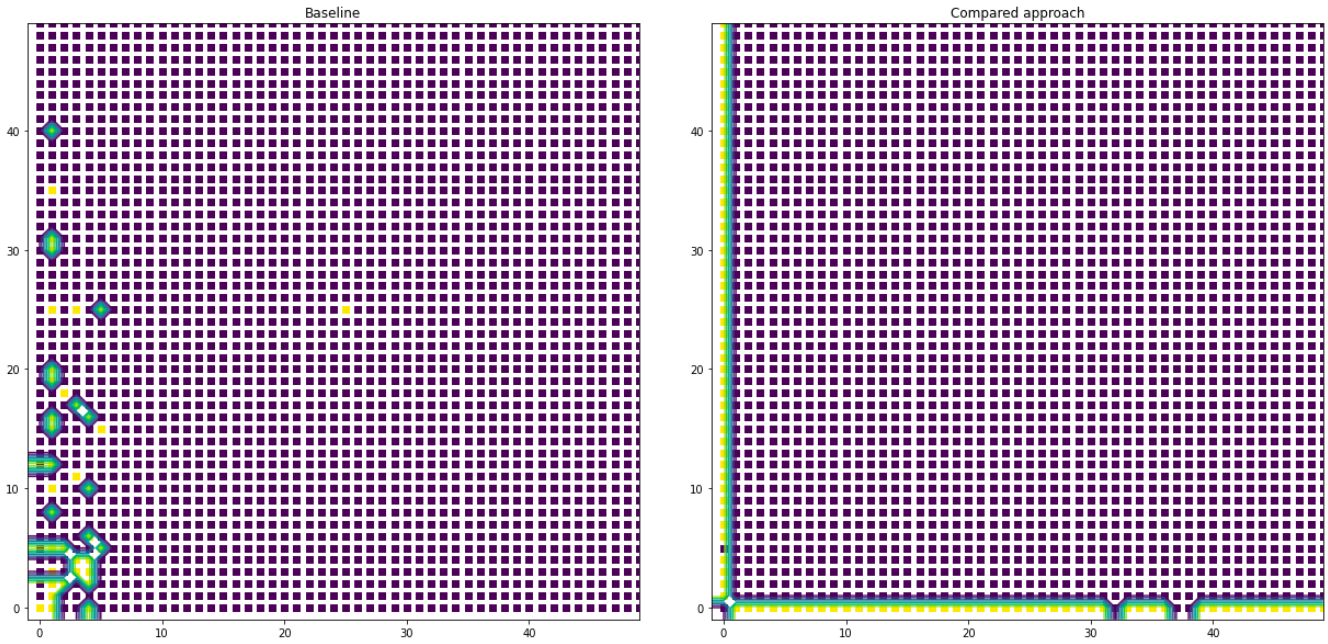
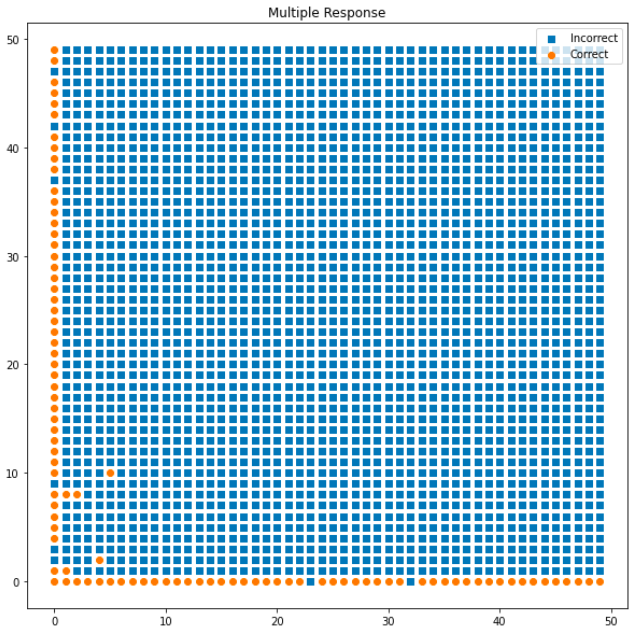


Figure 2: Contour plots of classifier boundaries

Figure 2 depicts the contour plot of classification boundaries. The plot shows that the boundary for the new encoding strategy is clearly defined, and the approach only works when one of the numbers is '0'. On the other hand, the plot for the baseline encoding strategy displays multiple boundaries, which indicates that LLM does not have an exact pattern to predict the correct answer. However, the plot suggests that there is a higher chance of obtaining the correct answer when the first number is very low.

- ***Multiple Response:***

The multiple response technique was used to improve accuracy. It involved generating the next token 10 times by setting do_sample=True and Temperature=0.6 to get some variation in the next tokens. The number that was predicted most often was then taken. However, this technique did not improve the accuracy significantly. The accuracy is 0.0384 in this case and, scatter plot is shown in the following figure.



e) AI collaboration statement

I have used ChatGPT for the code of contour plotting of the classifier boundary. Apart from that some of the information about temperature and sampling was learned from ChatGPT so that I can find better optimized values for these parameters. Lastly, I used it to rewrite some part of my explanation for better grammar and sentence structure.

f) Extra credit

I did the same experimentation with a different operation "**Multiplication**". I used the same encodings.

Baseline: "`0*0=`"
Compared to: "`if x=0 and y=0 then z=x*y is the multiplication of variable x and y. In this case, z=`"

Results:

Figure 3 presents a scatter plot depicting the correctly predicted and incorrectly predicted multiplication results. The accuracy for the baseline encoding strategy is 4.39%, whereas for the new encoding strategy, it is 2.84%.

Baseline Encoding	Compared New Encoding
`Accuracy: 0.0439`	`Accuracy = 0.0284`

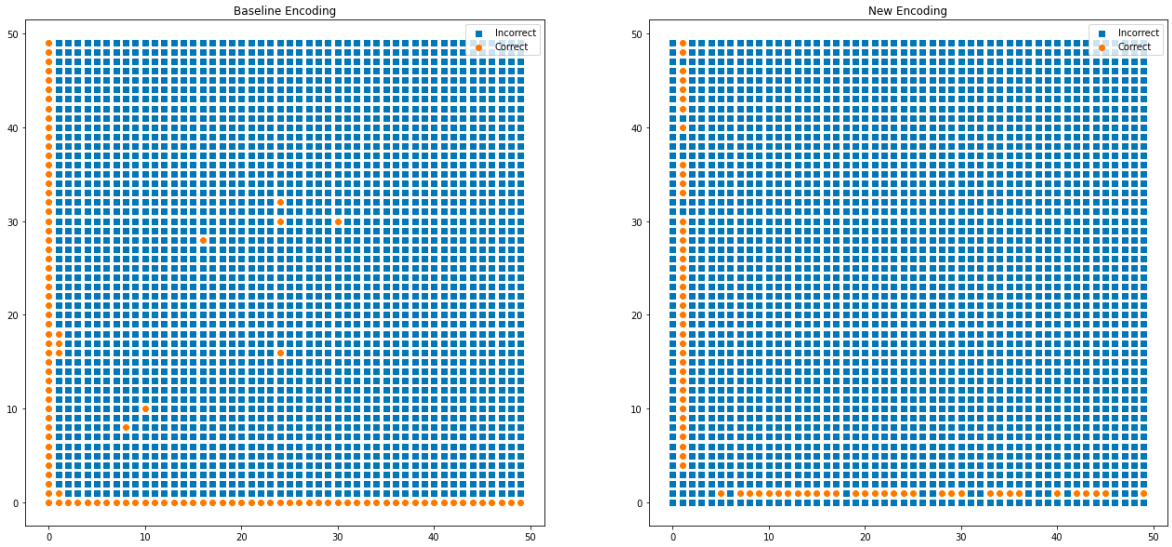


Figure 3: Scatter plot

A classifier was trained to determine whether the LLM is making correct or incorrect predictions randomly or systematically. Figure 4 illustrates the classification boundaries in contour plots for both scenarios. It is evident that the LLM can predict the multiplication for the baseline encoding strategy only when one of the numbers is '0'. For the compared cases, if one of the numbers is '1', then it can predict the multiplication; otherwise, it fails to predict the correct multiplication of two given numbers.

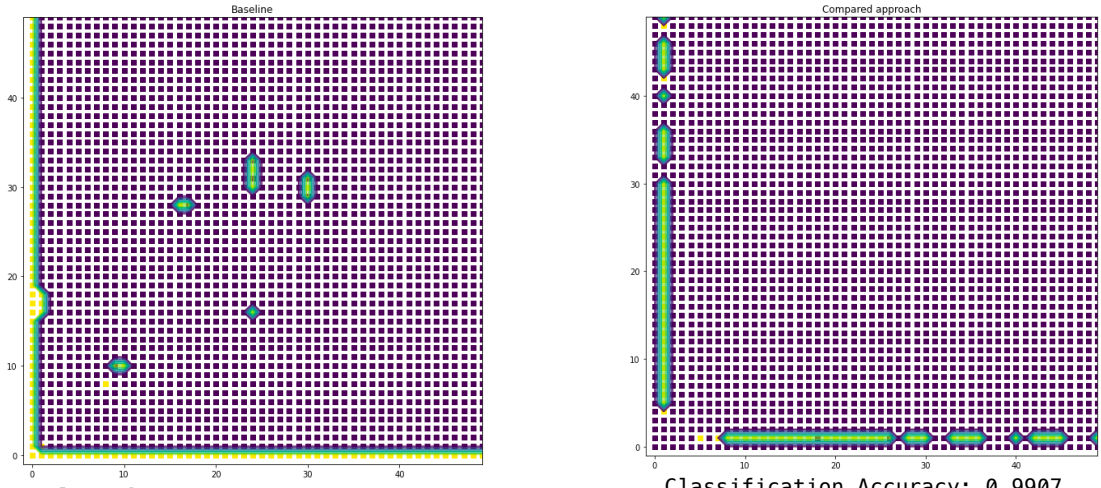


Figure 4: Classification boundaries

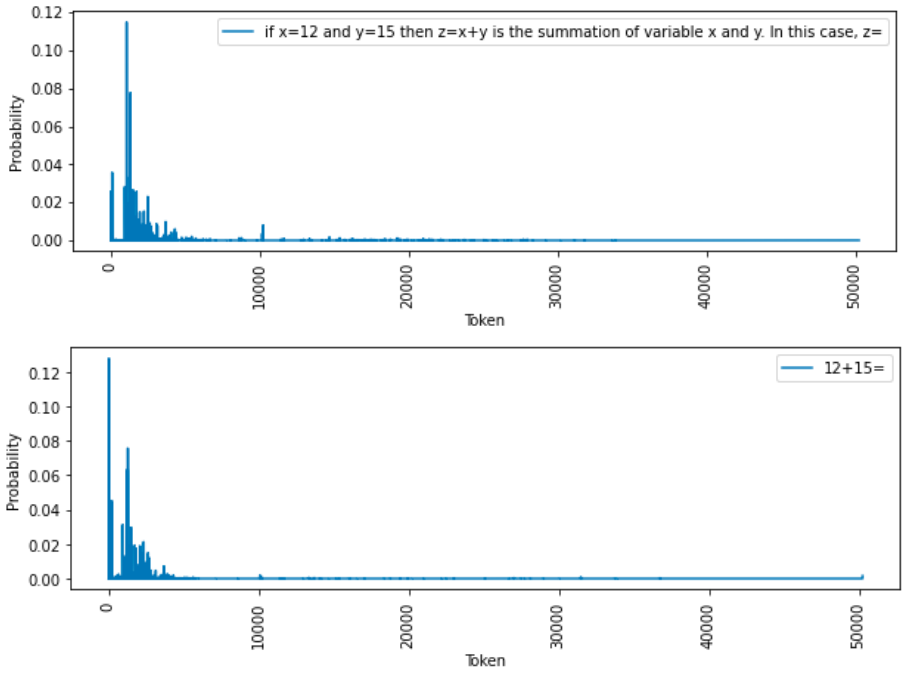
<u>Comparison of accuracy of correct answer between two operations:</u>

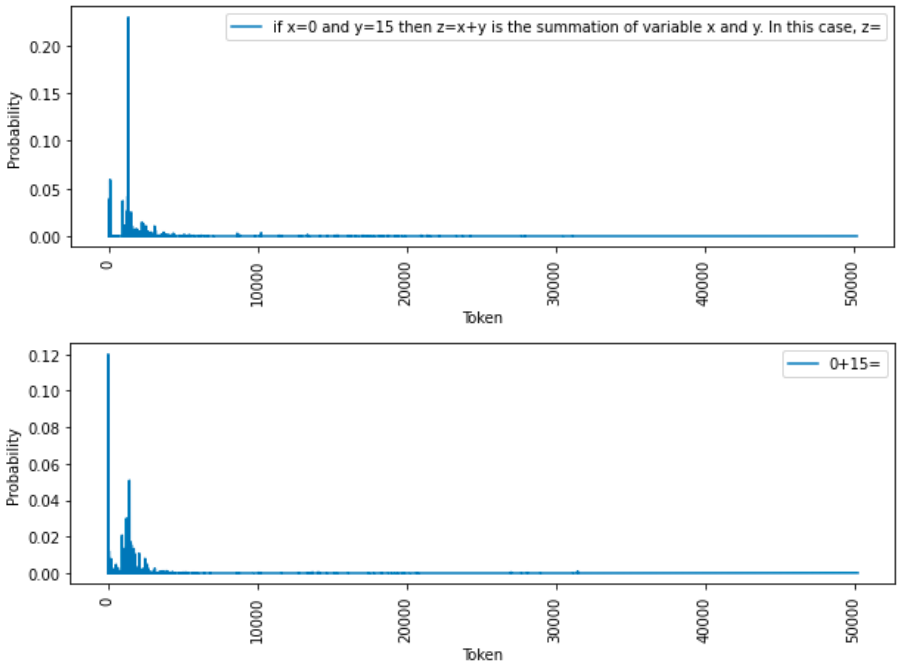
	Baseline	Compared Encoding
Summation	0.0148	0.0376
Multiplication	0.0439	0.0284

This table shows the accuracy of the LLM in predicting the correct answer for two different encoding strategies, for two different operations. For the summation task, the accuracy of the LLM is 0.0148 for the baseline encoding strategy, meaning that it correctly predicts the summation in only 1.48% of the cases. On the other hand, the accuracy for the compared encoding strategy is 3.76%. For the multiplication task, the accuracy of the LLM is 4.39% for the baseline encoding strategy. And the accuracy for the compared encoding strategy is 2.84%. In summary, the compared encoding strategy performs better or close to the baseline encoding strategy in predicting the correct answer, but the overall accuracy is still quite low for both strategies.

Next token probability distribution:

I performed a detailed analysis of the probability distribution of the next tokens for various prompts to better understand the behavior of my LLM. Specifically, I examined the probability distribution for both my baseline and compared encoding for the same problem. The resulting probability distribution plots are shown below.





Based on the probability distribution analysis for different prompts, it was observed that the probability distribution for the next token varies depending on the encoding scheme. However, it was noticed that the probability is peaked on a token. In the case of long text prompt, this peak is very prominent. On the other hand, in the case of short baseline prompt, some other tokens were observed to have comparatively higher probability even though not much higher like the highest probable token. Therefore, it can be concluded that the temperature parameter can be tuned using the short prompt, and in this case, different tokens may be obtained if multiple responses are used to predict the next token.