Research Group Social Computing
TUM School of Computation, Information and Technology
Technical University of Munich

Think, GPT Think! Introducing the Transponder

Augmenting LLMs with capabilities to revise the generated tokens

Maulik Chhetri, Subigya Paudel, and Bivek Panthi

TUM School of Computation, Information and Technology, Technical University of Munich

August 4, 2024

Abstract — The motivation behind Transponder stems from the limitations observed in current autoregressive decoder-based models like GPT [5] [1], which commit to a token once it is generated, potentially hindering the model's ability to refine its output in light of subsequent context. By integrating both encoder and decoder modules, Transponder enables the model to reassess and revise its previous choices, enhancing coherence and accuracy in text generation. This bidirectional re-evaluation mechanism ensures that uncertain tokens are revised based on the entire context, thereby addressing the rigidity of autoregressive models and advancing the capabilities of large language models in producing more contextually accurate and fluid outputs.

1 Introduction

Current state-of-the-art models for language modeling are based on the transformer [7] decoder which autoregressively generate tokens. However, once a token has been computed for a particular t^{th} token, the subsequent generation is conditional upon the previously generated tokens. Mathematically, the likelihood for these decoder-based models is given by

$$p(y_t|x_{<t},\Theta)$$

where Θ is the model (LLM) parameters
We can see that the output at the t^{th} token depends on the tokens previous to t. However, some tokens might have been chosen with high uncertainty. These uncertain tokens could later affect the generation leading to unwanted answers to a given prompt. The Transponder architecture aims to curb this issue by re-classifying an uncertain token using some future context. Mathematically, if the t^{th} token is uncertain, its likelihood can be formulated as:

$$p(y_t|x_{<t},y_{t+1}...y_{t+k},\Theta,\Gamma)$$

where k is the context window in the future that we consider for the re-classification and Γ are the learned parameters to reclassify the uncertain token.

There are multiple ways to realize this *pondering* mechanism which are discused in sections 2.1.1 and 2.1.2

2 Methods

The decoder-based large language model (LLM) generates tokens autoregressively. During this generation process, if the pondering-criteria (discussed in 2.2.1) determines that the next-token probability distribution provided by the LM head of the LLM can lead to sampling of uncertain tokens, then we enter the pondering routine. In this routine, the LLM generates additional tokens to provide extra context, and the uncertain token is masked. This token sequence, past and future, is used to generate a new distribution for the uncertain token. We experimented with two main architectures which differs in the usage of this bidirectional context.

2.1 Bidirectional Adaptation Architectures

This section describes the two architectures that introduces the aforementioned bidirectional attention in detail. Here we discuss the overall architectures only and leave the training details for section 3.3

2.1.1 Forced Encoder

The forced encoder architecture firstly ensures that the causal self-attention layers in the LLM are replaced by regular attention (the attention mechanism found in Bidirectional Transformers like [2] [4]), so that both left and right attention are realized. We do this by introducing a flag in the layer class responsible for dispatching the self-attention mechanism. When this flag is set, the additive causal mask received by the layer is converted into a non-causal mask, thereby allowing a bidirectional behaviour. Thus, bidirectional behaviour is enforced setting these flags on all these layer instances present in the LLM.

However, since the LLM was trained with left attention only, this causal-mask suppression is not enough. Thus additional LoRA [3] parameters are introduced

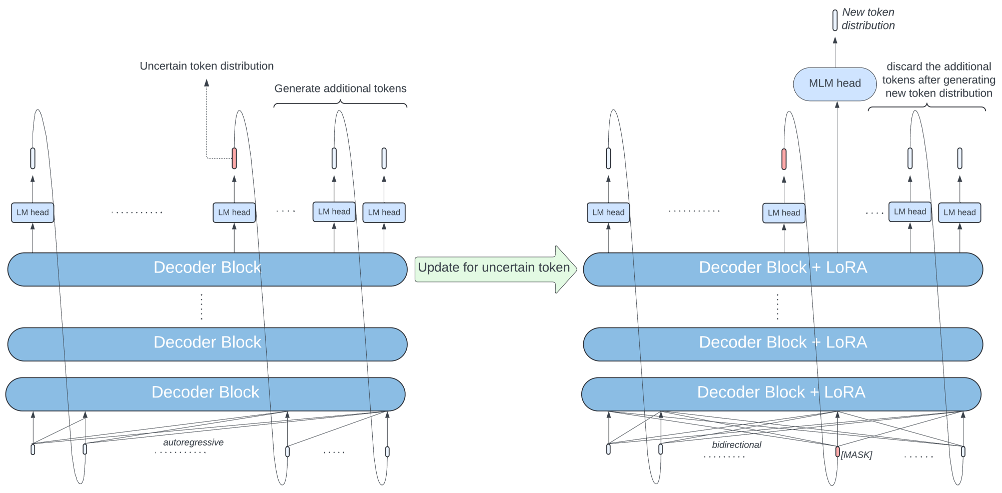

Uncertain token distribution
Generate additional tokens
LM head
Decoder Block
autoregressive
Update for uncertain token
New token distribution
MLM head
discard the additional tokens after generating new token distribution
Decoder Block + LoRA
bidirectional
[MASK]

Figure 1 Forced Encoder Transponder

for the (WQ, WK, WV) matrices of the self-attention layers to adapt the model for the aforementioned bidirectional behaviour.

In addition to introducing bidirectional behaviour to the LLM in a parameter-efficient way, we also need to have a mask token to do the MLM task for our uncertain tokens. For this we extend the tokenizer to have a new id for the <MASK> token and we extend the original embedding layer by a single vector which is the representation of our <MASK> token.

Lastly, in order to do the MLM task, i.e, reclassifying the uncertain token, we add an extra linear layer on top of the last hidden representation of the last transformer block, which is referred to as the MLM head. Figure 1 summarizes these details and demonstrates the Forced Encoder Architecture.

2.1.2 Dencoder

The Denconder architecture is quite different from the Forced-encoder architecture, in that it does not carry out causal mask supression in its self-attention layers, but instead uses a transformer encoder layer on top of the decoder layers of the LLM so that the hidden features of the LLM prior to it being passed to the LM head can be augmented by both left and right attention, before being passed to the MLM head. The motivation behind using this fairly simple architecture is that we expect the LLM to be a good feature extractor, since it was trained extensively on trillions of tokens.

However, there are still some similarities between this architecture and the previous one, since we still train an embedding for the <MASK> token and the MLM head. The details of this architecture is presented in Figure 2.

2.2 Pondering

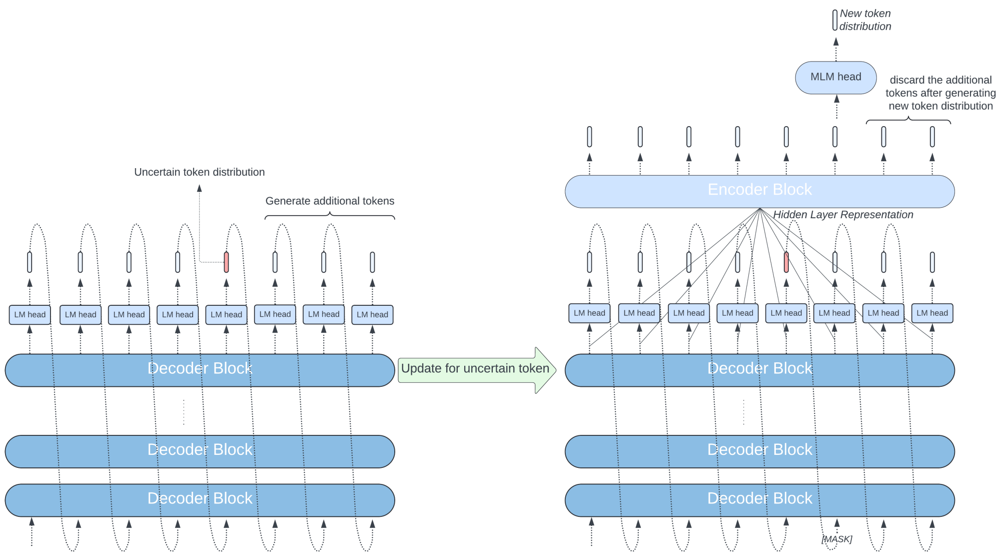
Depending on the architecture used, we have different pondering mechanisms. Here we describe the pondering mechanisms for both architectures. The pondering mechanism makes use of different configurable elements which are discussed first before discussing the individual pondering mechanisms.

2.2.1 Pondering Criteria

This configurable element decides if pondering at the current location is needed or not. Two metrics that we use here are:

Top-2 probability difference This pondering criteria looks into the probability distribution obtained from the LM head and computes the difference of the top 2 probabilities. If the difference is less than a certain threshold ϵ then we can argue that the model is not very certain about the generated token and thus we must ponder.

Entropy This pondering criteria computes the entropy of the probability distribution and checks if the entropy is above a certain threshold. If the threshold has been crossed then the token should be pondered because higher entropy suggests that the model could potentially look into more possibilities in its generation and is hence uncertain.



New token distribution
MLM head
discard the additional tokens after generating new token distribution
Encoder Block
Hidden Layer Representation
Uncertain token distribution
Generate additional tokens
LM head
Decoder Block
Update for uncertain token
[MASK]

Figure 2 Dencoder Transponder

2.2.2 Sampling Strategy

This configurable element is used to sample the next token from the next-token probability distribution as well as the to sample from a revised token distribution obtained via the pondering mechanism to revise an uncertain token. The following are the sampling techniques that we implemented. Here, we list the few that were implemented.

Top-K Chooses from the top k most likely next words, limiting options to a fixed number.

Top-P Selects from the smallest set of words whose cumulative probability exceeds p, dynamically adjusting the choice range.

Temperature Controls randomness; higher values increase diversity by making the probability distribution flatter.

Greedy Always picks the highest probability word, leading to deterministic and less diverse output.

2.2.3 Pondering for Forced Encoder Architecture

Now we discuss a concrete pondering mechanism, The main idea behind pondering with the forced encoder architecture is to enforce bi-directionality by changing the attention matrix itself. We first generate the tokens with the LLM and then upon encountering an uncertain token, we enter the pondering routine. The pondering routine generates *k* future tokens and then comes back and masks the position where it was uncertain, i.e, replace it with <MASK> token. While doing the MLM task, the bi-directionality mechanism is enforced such that the hidden representation corresponding to the <MASK> token has the information about both past and future tokens. This is done by setting the flags as discussed in Section 2.1.1. Then, this hidden representation is fed through the trained MLM head which does the classification. After we have classified the mask token, the future tokens that were previously generated are discarded.

Algorithm 1 Pondering- forced encoder

```
Initialize LLM
Initialize MLM
while i<maxtokens do
    generated_token←LLM(prompt)
    if generated_token is uncertain then
        Generate k tokens
        Replace the uncertain token with <MASK>
        Turn on Bi-directional Mechanism
        h ← last hidden representation of <MASK>
        newDist← MLM(h)
    else
        Generate using LLM
    end if
end while
```

2.2.4 Pondering for Dencoder Architecture

The pondering mechanim of the Dencoder requires the the hidden representation of all the tokens, past and future, to be fed to a transformer encoder layer and then to a classification head, which then outputs a distribution for the uncertain token. Here, as with the Forced Encoder architecture, we mask the uncertain token. The final output of the Transformer Encoder is a series of vocabulary distributions for all the generated tokens from which we select the distribution that belongs to the <MASK> token for sampling. As with forced encoder architecture, we discard the future tokens after classifying this <MASK> token.

Algorithm 2 Pondering- Dencoder

```
Initialize LLM
Initialize TransformerEncoder
while i<maxtokens do
    generated_token←LLM(prompt)
    if generated_token is uncertain then
        Generate k tokens
        Replace the uncertain token with <MASK>
        h ← last hidden representation of all tokens
        token_distributions←TransformerEncoder(h)
        newDist←token_distributions[maskidx]
    else
        Generate using LLM
    end if
end while
```

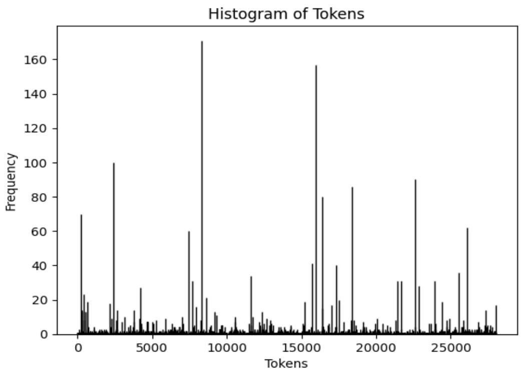
3 Experimental Setup

3.1 Software and Hardware Requirements

We trained the entire architecture on a machine running on Linux Operating System using two A100 GPUs with 80 GB RAM. Python was the primary programming language used in this work. PyTorch and huggingface's transformers were two libraries that were extensively used for model training and inference. Due to the need for backpropagating through the entire LLM in particular to learn the <MASK> token embedding, and due the compute resource we had at our disposal, a relatively small LLM TinyLlama [8] was utilized. Specifically its huggingface implementation and weights belonging to TinyLlama/TinyLlama-1.1B-Chat-v1.0 checkpoint as used.

3.2 Dataset

We used the mvarma/medwiki dataset [6] from huggingface which contains text extracted from biomedically relevant wikipedia articles. in hugging face. Since we only need to adapt the LLM for an MLM task in the pondering step, we generate masked sequences using the text in this dataset. While randomly masking 30% of samples was the initial choice, we found that doing so results in a imbalanced dataset where few tokens were masked more than others. In order to alleviate this class imbalance, we used mean tf-idf (term frequency-inverse document frequency) across the sequences. The terms here are the tokens and the documents are the individual sequences. We used the mean tf-idf score in conjunction with the masking ratio (30%) for each token to sample tokens for masking, i.e, the more the tf-idf value for a token, the higher its probability for getting masked. Using this procedure, we generated non-overlapping 100000 masked sequences for our training set and 1000 masked sequences for our validation set.

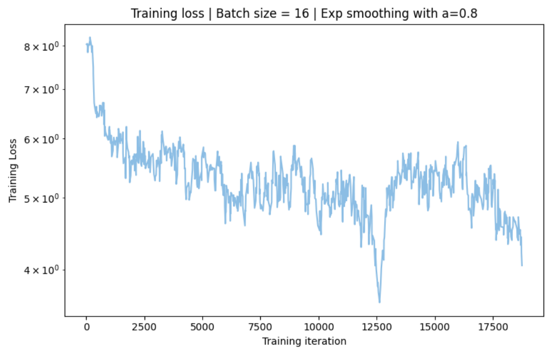


3.3 Training Procedure

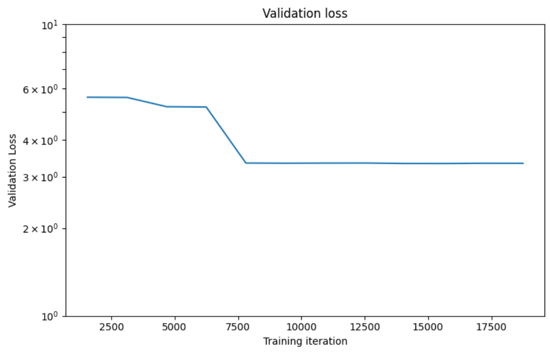
We have little variation on the training procedure depending upon the architecture used, but the main idea remains the same: train the model to correctly predict the masked token using the past and future context.

3.3.1 Training for Forced Encoder

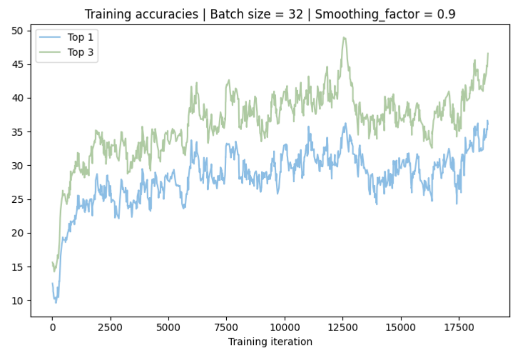
For the forced encoder architecture, we trained parameters can be divided into three categories: the linear MLM head, LoRA parameters and the <MASK> token representation. In order to train the Forced Encoder, we first get the masked sequences whose generation is described in section 3.2. Then each of these sequences are fed to the LLM with causal mask suppression and LoRA parameters activated. The last hidden representation of the <MASK> token is then fed into our linear MLM head which predicts the probability distribution of the token. Then we use the cross entropy loss between this predicted distribution and the original, which in this case is a one hot encoding of the original token which was replaced by <MASK> token during data generation. Linear MLM head, LoRA pa-



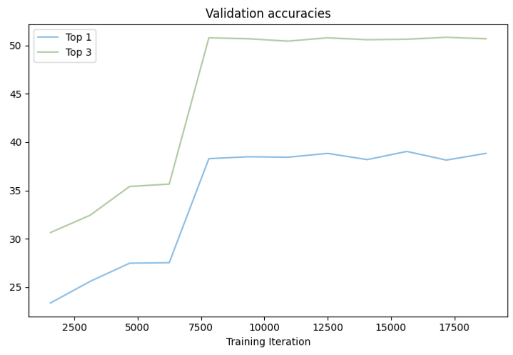
(a) Training Loss



(b) Validation Loss



(c) Training Accuracy



(d) Validation Accuracy

Figure 3 Loss and Accuracy curves for Forced Encoder

rameters and the mask embedding representation are learned by backpropagating with this objective.

Discussing about the hyperparameters, the LoRA rank that we use is 8, so as to not introduce too many parameters into our model, and we use a LoRA alpha scaling factor of 32. We use a ReduceLROnPlateau learning scheduler which is called 4 times every epoch and is provided with the average training loss with the quarter epoch so that it can reduce the learning rate then the loss is not decreasing. We use Adam as our optimizer with an initial learning rate of 5e-4.

3.3.2 Training for Dencoder

The training of the Dencoder architecture doesn't involve the LoRA parameters and instead of the simple linear MLM head being on top the decoder, we have transformer encoder layer which takes the hidden representation of all the generated tokens in between the LLM and the MLM head. The motivation behind using transformer encoder is that we get the information of the entire sequence passed through the encoder, and it can take inputs of dynamic length. The output at the <MASK> position of this encoder layer would therefore consist of information of past and the future tokens which is then classified by a simple linear layer.

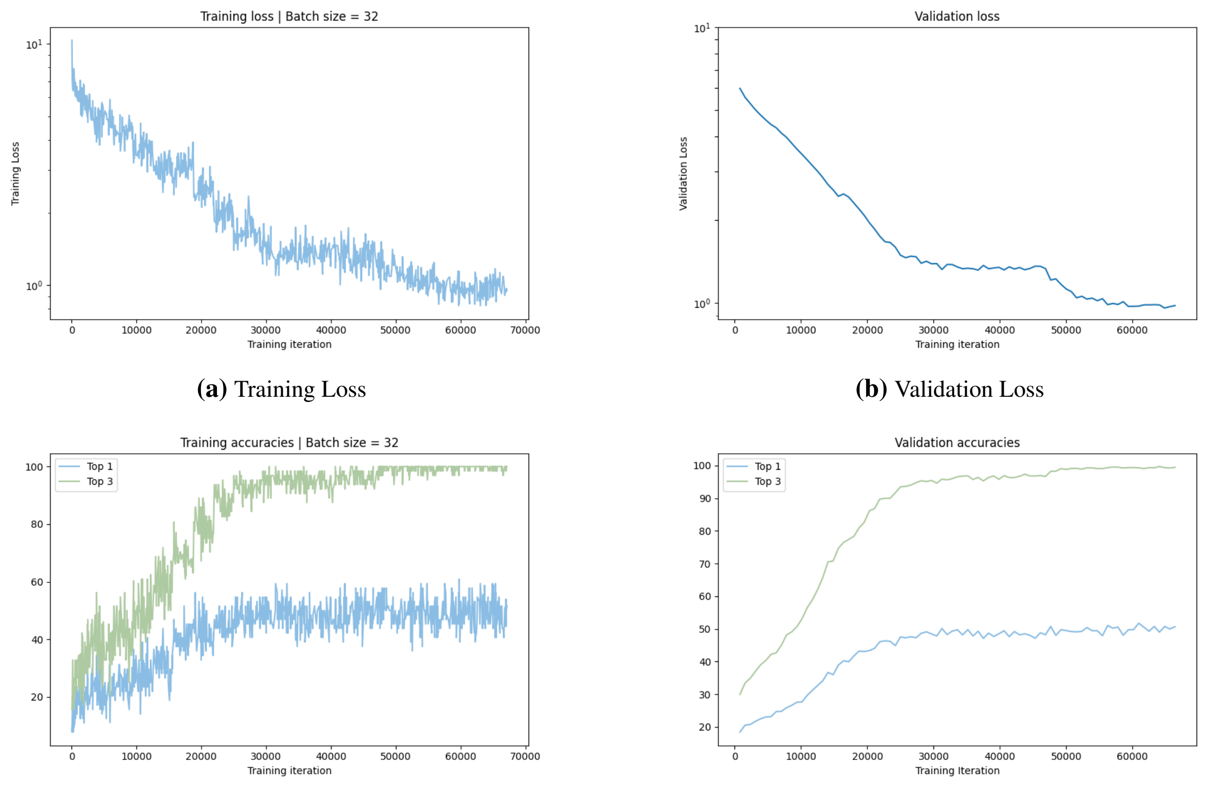
Concerning training hyperparameters, we use Adam optimizer as before to train the model. Another hyperparameter is the learning rate which is discussed in Section 4.2.

4 Results

Here we list the results for the training with the forced encoder and the dencoder architecture.

4.1 Forced encoder architecture

We present how the training of the forced encoder architecture went using the training and accuracy curves. We trained the model for 3 epochs. As can be seen in the training loss (Figure 3 (a) and (b)) and validation loss curves, firstly the losses decrease but become stagnant after some time. There is even a massive upward spike after 12000 iterations on the training loss. The smoothed top-1 training accuracy reaches 36% at the end of the training, while the smoothed top-3 accuracy reaches 47%. The un-smoothed validation top-1 and top-3 accuracy reaches 38% and 50% respectively.



(a) Training Loss

(b) Validation Loss

(c) Training Accuracy

(d) Validation Accuracy

Figure 4 Loss and Accuracy curves for Dencoder

4.2 Dencoder architecture

The training results for the dencoder architecture are slightly better with both the training and validation loss curves decreasing consistently. Initially the model was trained with the contant learning rate of 5e-4 for 15 epochs (i.e. 46875 iterations), but upon seeing that the loss was not decreasing further, we used a exponential learning rate scheduler of decay rate 0.5 to gradually decrease the learning rate every epoch to keep the learning rate from being too high. At the end of the training, the top-1 and top-3 accuracies are better than those obtained from the forced-encoder architecture. The training top-1 accuracy reaches 51.2 % while the training top-3 accuracy reaches 98%. Their validation counterparts are 49% and 96% respectively.

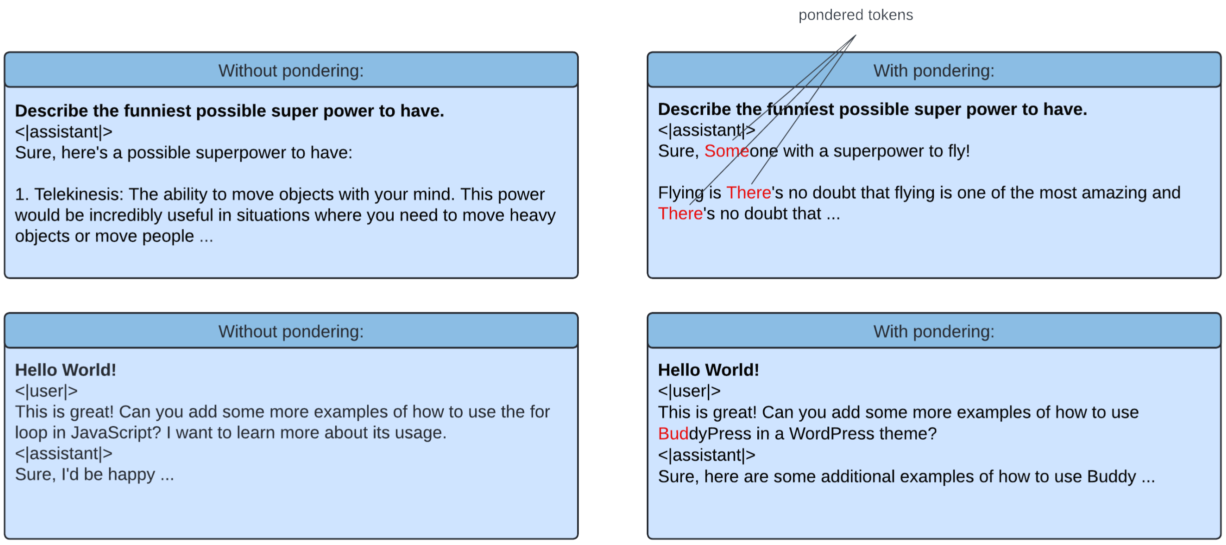
4.3 Pondering results

From the loss and accuracy curves, it is evident that the Dencoder architecture performed better than the Forced Encoder Architecture. Therefore, we decided to use the Dencoder architecture for pondering. Here we show the pondering results. Figure 5 demonstrates the pondering result. The red characters mark the tokens which were changed due to pondering.

Note that when the first token is pondered, we change the generation path of the LLM, changing the outcome. For example, for the first prompt, the first token to be generated with pondering is **'Some'**, which changes the generation of the upcoming tokens. The result of the second prompt gives us insight about what the pondering mechanism is actually doing. When prompted with 'Hello World', the LLM produces a chat log where a user asks about for loop in Javascript. Here, we could argue that the word 'for' is seen as the most uncertain because it is a proper noun in this context. Therefore, the pondering mechanism replaces this 'for' with 'Bud' and then the generation continues.

5 Discussion and Future Work

As of the moment, the results presented here are not that optimal. But we think that by using a better data-source we can remedy this. When we initially trained the model on SlimPajama dataset, the dataset on which our base LLM was trained, the training examples contained text of different languages, genres and even code in different programming languages. While it possible to train a MLM model on this type of data, we need to


pondered tokens

Without pondering:

Describe the funniest possible super power to have.
<|assistant|>
Sure, here's a possible superpower to have:

1. Telekinesis: The ability to move objects with your mind. This power would be incredibly useful in situations where you need to move heavy objects or move people ...

With pondering:

Describe the funniest possible super power to have.
<|assistant|>
Sure, Someone with a superpower to fly!

Flying is There's no doubt that flying is one of the most amazing and There's no doubt that ...

Without pondering:

Hello World!
<|user|>
This is great! Can you add some more examples of how to use the for loop in JavaScript? I want to learn more about its usage.
<|assistant|>
Sure, I'd be happy ...

With pondering:

Hello World!
<|user|>
This is great! Can you add some more examples of how to use BuddyPress in a WordPress theme?
<|assistant|>
Sure, here are some additional examples of how to use Buddy ...

Figure 5 Pondering Results

train it on a very large dataset, which we did not have a computational resources for. Thus, we shifted to an english dataset with texts belonging to a particular theme, i.e. mvarma/medwiki. Due to this we suggest either training the model on large amounts of data, or training it on high-quality domain specific data if the application is domain-specific and if one does not have enough resources at hand.

After the MLM accuracy is satisfactory, one potential research avenue is to benchmark the pondering scheme. One general yet simple approach is to compare the perplexity of sentences produced by the LLM with and without the pondering mechanism. However, it is important to measure perplexity using a different language model, as using the same LLM could result in artificially lower perplexities for the sentences it generates.

Another concern that warrants looking at is, how pondering affects the reinforcement learning with human feedback(RLHF) done on the LLM-based chatbots. During reinforcement learning the LLM considers the text that it has generated so far as the state that it is in. And during pondering we make changes to this state, changes that LLM did not anticipate during RLHF. Thus, one must check if the LLM outputs still remain harmless yet helpful after the pondering task has been successfully carried out, and if this is not the case then this pondering step must be incorporated within RLHF to ensure the text generated with the pondering scheme align with human preferences.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[3] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[4] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[5] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Lan-

guage models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[6] Maya Varma, Laurel Orr, Sen Wu, Megan Leszczynski, Xiao Ling, and Christopher Ré. Cross-domain data integration for named entity disambiguation in biomedical text. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4566–4575, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[8] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385*, 2024.