Probabilistic Model for Infinite Memory LLM: Variational Markov Transformer

(theory part complete, experiment in progress, paper not finished yet, open for collaborations)

Authors: Yiyan CHEN

Abstract

Current Long Language Models (LLMs) are constrained by limited window sizes and require substantial resources for training and inference. Addressing these limitations, we propose a novel approach that establishes a foundational probabilistic framework for modeling the distribution of sequences of arbitrary lengths, including the comprehensive corpus used to train LLMs.

Our methodology integrates Deep Hidden Markov Models (DHMMs) as the core probabilistic mechanism to represent the sequence collection. For text generation tasks, we employ pre-trained LLMs as encoders for observed textual sequences. By optimizing a lower bound of the log likelihood of randomly sampled continuous subsequences from the training dataset, we derive multiple probabilistic models. These encompass the prior distribution of hidden states across all sequences, the transition probabilities between hidden states, and the likelihood of mapping hidden states to sequence tokens, along with their posterior distributions.

This framework facilitates several practical applications during inference, such as processing input sequences **without window size constraints**, accelerating sequence generation, and enabling bidirectional (normal and reverse order) generation. Crucially, our model achieves **theoretical infinite memory through state transitions**, extending beyond conventional window limitations. Moreover, it significantly reduces the required input window size for LLMs during both training and inference phases, thereby decreasing the overall computational demands and resource consumption of LLMs.

This approach not only enhances the capabilities of existing LLMs but also paves the way for more efficient and versatile language modeling techniques, with potential applications across various domains of natural language processing.

Introduction

The theoretical memory window of an RNN cannot be said to be infinite. If it were infinite, then the hidden state would have to contain the entire memory of the RNN agent. However, during training, the starting hidden state of each training sequence is either zero or noise. It's important to note that if the RNN is supposed to output a long thesis, how could the starting memory possibly be zero? Or how could it be the same to the RNN that outputs a sequence of a child's writing? Perhaps, when training each sequence, calculating the most suitable starting hidden state value first is the most consistent approach, but RNNs lack a strict probabilistic model to achieve this. Moreover, RNNs also have problems with gradients and the inability to train parallel sequence data.

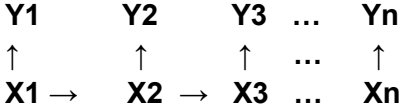
As for transformers, all current attempts are engineering efforts to expand the window or to extrapolate longer lengths than trained using attention and relative position encoding. Some works are aimed to combine the above RNN with the transformer architecture thus still suffering from the RNN's drawbacks.

Probabilistic modeling of the training sequence

The memory should be a compression of what we have seen before but not all the raw data we have seen, when we want to recall something, we don't need to reread all what we read before.

What I currently want to try is a method similar to deep HMM. It's based on deriving the maximum likelihood training target from VAE, striving to solve the aforementioned problems, equipping the RNN with a strict probabilistic model to find the most suitable initial hidden state given a list of observations.
For training, sample a continue subsequence of any length(n>=2) from the training sequence:
Y1 Y2 Y3 … Yn

We could suppose the following probabilistic model:



X1, X2 … Xn are hidden variables, during the inference time, Xk contains the key to activate all memory before time step k, including the memory before X1. We suppose the dimension of Xk does not need to be bigger than the dimension of Yk, as said above, Xk is not a storage of all memory before, the true memory is stored inside the entire neural network, Xk just acts as the key to activate the memory belonging to it's time step.

For each continue subsequence, we aim to maximize **P(Y1, Y2, Y3…Yn):**

$$
\begin{aligned}
P(Y) &= \frac{P(Y,X)}{P(X|Y)} = \frac{P(Y,X)Q(X|Y)}{P(X|Y)Q(X|Y)} \\
\Rightarrow logP(Y) &= logP(Y,X) + log\frac{Q(X|Y)}{P(X|Y)} - logQ(X|Y) \\
\Rightarrow logP(Y) &= \int Q(X|Y)logP(Y)dX \\
&= \int Q(X|Y)(logP(Y,X) - logQ(X|Y))dX + \int Q(X|Y)log\frac{Q(X|Y)}{P(X|Y)}dX \\
&= \int Q(X|Y)(logP(Y,X) - logQ(X|Y))dX + KL(Q(X|Y)||P(X|Y)) \\
\Rightarrow logP(Y) &\geq \int Q(X|Y)(logP(Y,X) - logQ(X|Y))dX = ELBO
\end{aligned}
$$

Training VMT with pretrained GPT

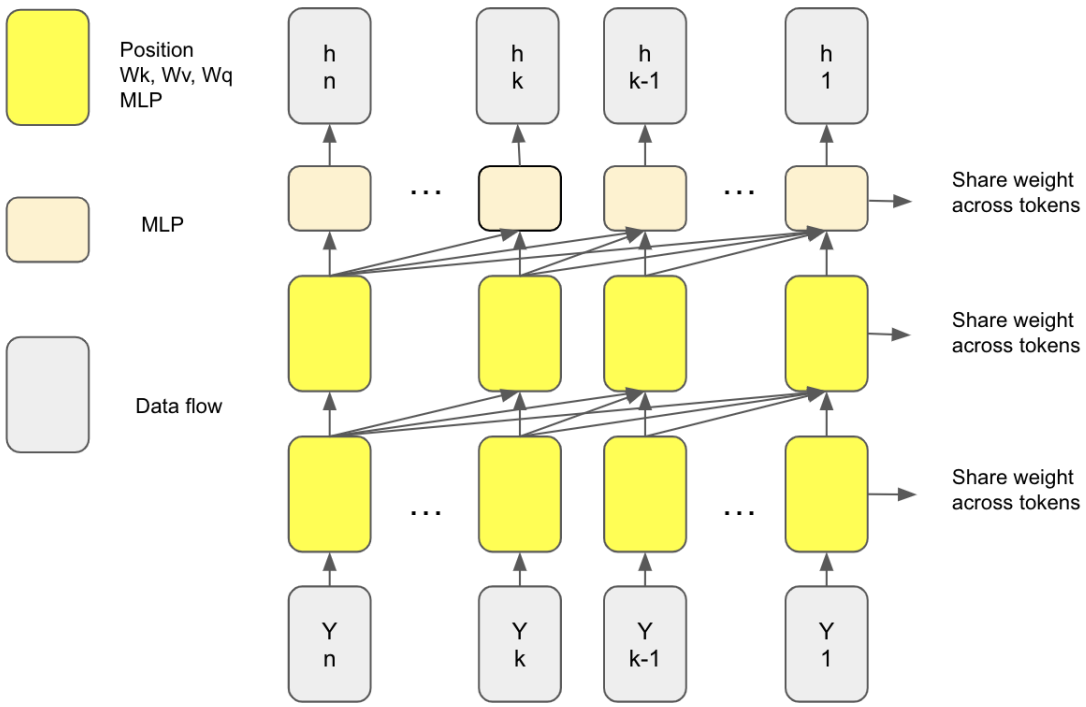
$$
\begin{aligned}
P(Y,X) &= P(y_1, y_2...y_n, x_1, x_2...x_n) \\
&= P(x_1)P(y_1|x_1)\prod_{k=1}^{n-1}(P(x_{k+1}|x_k)P(y_{k+1}|x_{k+1})) \\
Q(X|Y) &= P(x_1, x_2...x_n|y_1, y_2...y_n) \\
&= Q(x_1|y_1, y_2...y_n)\prod_{k=1}^{n-1}Q(x_{k+1}|x_k, y_{k+1}...y_n) \\
&\Rightarrow \\
ELBO &= \mathop{\mathbb{E}}_{x_1\sim Q(x_1|y_1...y_n)}(logP(x_1) + logP(y_1|x_1) - logQ(x_1|y_1...y_n) + ELBO_1) \\
k &= 1..n-1 \\
ELBO_k &= \mathop{\mathbb{E}}_{x_{k+1}\sim Q(x_{k+1}|x_k, y_{k+1}...y_n)}(logP(x_{k+1}|x_k) + logP(y_{k+1}|x_{k+1}) - logQ(x_{k+1}|x_k, y_{k+1}...y_n) + \\
&ELBO_{k+1}) \\
ELBO_n &= 0
\end{aligned}
$$

By the property of MarKov chain we could parametrize

1. P(x_k)
2. P(y_k|x_k)
3. P(x_k+1|x_k)
4. Q(x1|y1…yn)
5. Q(x_k+1|x_k, y_k+1…y_n)

For **any k** and **any subsequence** from the training document set using the same 5 neural networks above.
Free to choose adaptive architecture for 1, 2, 3 according to the nature of the problem.
I personally prefer to use my BreezeForest and conditional BreezeForest, as they are universal density estimators.
Here we focus on how to choose architecture for 4. and 5.
By leveraging a pre-trained GPT like LLM, we could compress the sequence of yn, y_n-1…y_k into a single hidden state input for 4. And 5. as the following:

Take an autoregressive transformer of window size n as the figure below:



The weight are shared across tokens, let's say the weight is W_tf, for both training and inference, given y1, y2 … yn a continue sequence sampled from the ground truth sequence, as

shown above we inverse the order of the sequence then compute the feature set used in posterior distribution:
hn, h_n-1 … h1 = f(W_tf, yn, y_n-1…y1), where hk gathers all the information from yn down to yk.

Q(x1|y1…yn) = Q(x1|h1) -> choose a conditional generative model as you want
Q(x_k+1|xk, y_k+1…yn)= Q(x_k+1|xk, h_k+1) -> choose a conditional generative model as you

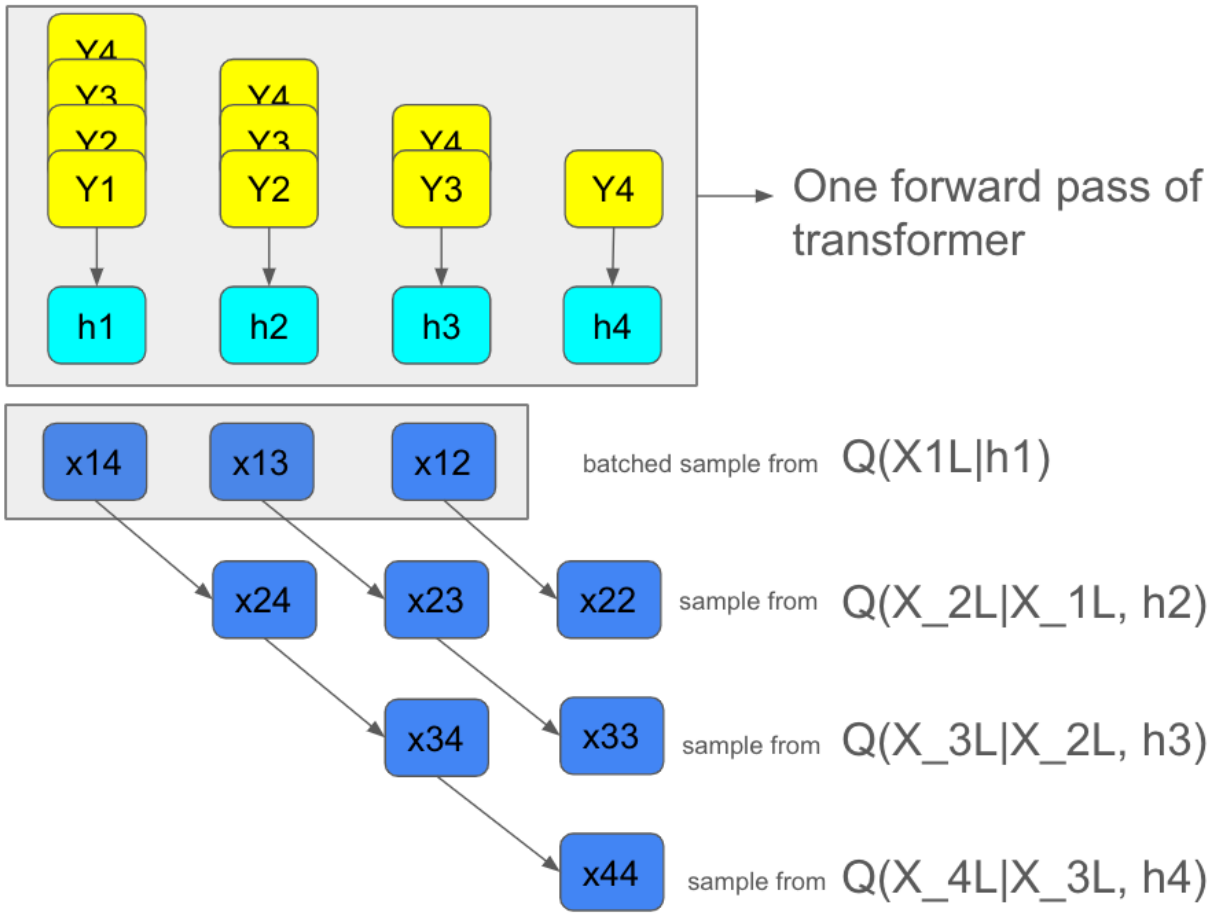
If W_tf is a GPT like transformer which has not been trained on sequence of inverse order, the weight of W_tf may need to be finetuned during the training.

Algorithm 1: Training VMT starting from a pre-trained GPT

```
Init GPT: f(W_tf, …)
Init Theta for Markov part of weight
    1. P(x_k)
    2. P(y_k|x_k)
    3. P(x_k+1|x_k)
    4. Q(x1|...)
    5. Q(x_k+1|x_k, ...)
Init optimizer
Set number of iterations M for each fixed sequence length
For L in range(2, 2000):
  For _ in range(0, M):
    Sample a continue subsequence of length Y1, Y2 …YL
    hL …  h2, h1 = f(W_tf, yL, y_L-1…y1)
    Loss = -ELBO(hL …  h2, h1,yL, y_L-1…y1, Theta )
    grad1 = dLoss/dTheta
    grad2  = dLoss/dW_tf
    Theta, W_tf <- optimizer(grad1, grad2,  Theta, W_tf)
```

One may notice that the ELBO evaluation requires sampling X1,X2 …Xn sequentially.

A solution to this is to parallelize the training by computing simultaneously the ELBO of different subsequences of the sampled long sequence, take an example of a sequence of length 4:
Y1, Y2, Y3, Y4;
We will compute the ELBO of the 3 subsequences:
Y1, Y2, Y3, Y4
Y2, Y3, Y4
Y3, Y4
Simultaneously as the image below:



X_jk means the jth hidden variable in the subsequence of length k.
After computing all hidden states h1…h4, we sample all the X at the same position for all subsequences simultaneously. So if the length of the main sequence is L, we only need to sample L times sequentially to compute the ELBO of L-1 subsequences. This can accelerate the training speed.

Thus, we could compute several subsequence's ELBO in one batch, the memory size of one batch depend on the number of Xs sampled. In practice, we want to keep the memory consumption of all batches similar. Another point is that we prefer to begin with main sequences of small length then grow it little by little.

These result in an improved version of algorithm 1:
Algorithm 2: Training VMT starting from a pre-trained GPT with subsequence pallerization

Init GPT: f(W_tf, …)
Init Theta for Markov part of weight
    1. P(x_k)

```
    2. P(y_k|x_k)
    3. P(x_k+1|x_k)
    4. Q(x1|...)
    5. Q(x_k+1|x_k, ...)
Init optimizer
Set number of Xs sampled in one batch: N > 2000 according to memory capability
Set number of iterations M for each fixed sequence length
For L in range(2, 2000):
   For _ in range(0, M):
      Sample a continue main sequence of length Y1, Y2 …YL
      hL …  h2, h1 = f(W_tf, yL, y_L-1…y1)
      Loss = -ELBOS(hL …  h2, h1,yL, y_L-1…y1, Theta, N)
      grad1 = dLoss/dTheta
      grad2  = dLoss/dW_tf
      Theta, W_tf <- optimizer(grad1, grad2,  Theta, W_tf)
```

Where the function ELBOS() samples a number of subsequences from Y1, Y2 …YL so that the total number of Xs sampled is maximized but without exceeding N. One typical sampling strategy is to always choose the longest feasible(without exceeding N) subsequence among all no selected subsequences, until the total Xs sampled reach N.

Def ELBOS(hL … h2, h1,yL, y_L-1…y1, Theta, N):

```
      Choose subsequences to be included, selected_number: number of selected
      subsequences
      Init ELBO_sum to 0.
      For i in range(0, max_subsequence_L):
         For all subsequences having the ith item, sample all Xi.
         Accumulate the ELBO  brought by the ith  item from all sequences into ELBO_sum
      Return ELBO_sum/selected_number
```

The algorithm 2 will require a much smaller M, since in the inner bloc, several subsequences will be involved simultaneously in the training.

Training reverse VMT with fixed weight pretrained GPT

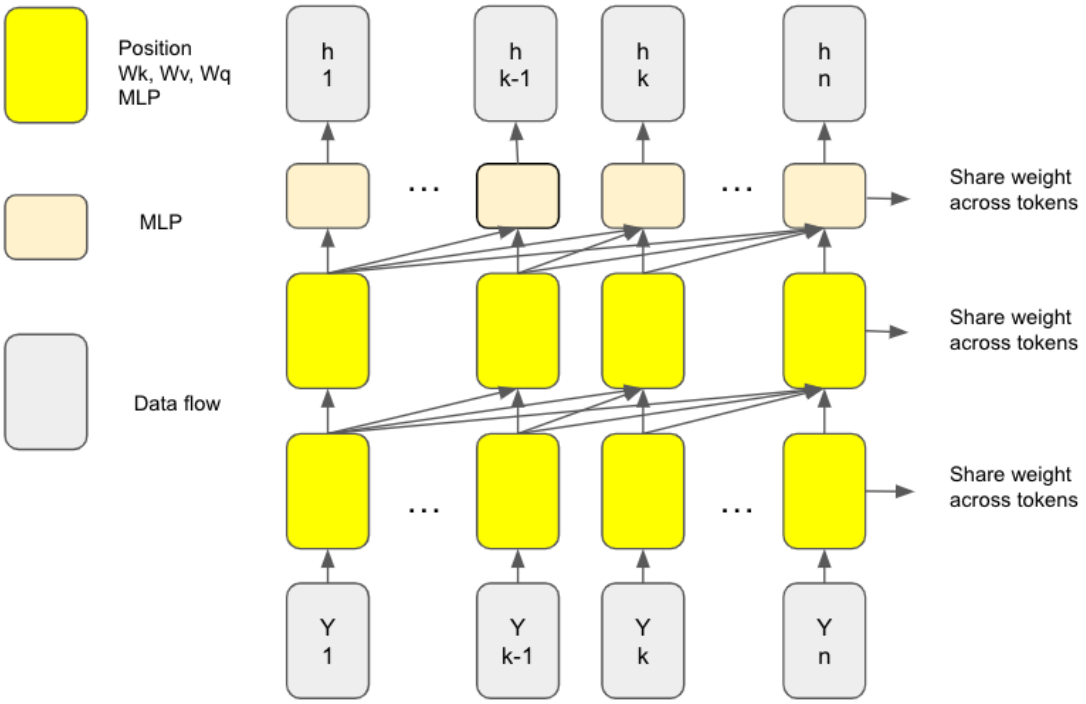
$$P(Y,X) = P(y_1, y_2 ... y_n, x_1, x_2 ... x_n)$$
$$= P(x_n)P(y_n|x_n)\prod_{k=1}^{n-1}(P(x_{n-k}|x_{n-k+1})P(y_{n-k}|x_{n-k}))$$
$$Q(X|Y) = P(x_1, x_2 ... x_n|y_1, y_2 ... y_n)$$
$$= Q(x_n|y_1, y_2 ... y_n)\prod_{k=1}^{n-1}Q(x_{n-k}|x_{n-k+1}, y_1 ... y_{n-k})$$
$$\Rightarrow$$
$$ELBO = \mathop{\mathbb{E}}_{x_n \sim Q(x_n|y_1 ... y_n)}(logP(x_n) + logP(y_n|x_n) - logQ(x_n|y_1 ... y_n) + ELBO_1)$$
$$k = 1..n-1$$
$$ELBO_k = \mathop{\mathbb{E}}_{x_{n-k} \sim Q(x_{n-k}|x_{n-k+1}, y_1 ... y_{n-k})}(logP(x_{n-k}|x_{n-k+1}) + logP(y_{n-k}|x_{n-k}) -$$
$$logQ(x_{n-k}|x_{n-k+1}, y_1 ... y_{n-k}) + ELBO_{k+1})$$
$$ELBO_n = 0$$

By the property of MarKov chain we could parametrize

1. P(x_k)
2. P(y_k|x_k)
6. Pr(x_k-1 |xk)
7. Qr(xn|y1…yn)
8. Qr(x_k-1|xk, y1…y_k-1)

for any k and any subsequence using the same 5 neural networks.
Free to choose adaptive architecture for 1, 2, 3 according to the nature of the problem.
Here we focus on how to choose architecture for 4. and 5.
By leveraging a pre-trained GPT like LLM, we could compress the sequence of y1…y_k-1 into a single hidden state input for 4. And 5 as the following:

Take an autoregressive transformer of window size n as the figure below:



The weight are share across tokens, let's say the weight is W_tf, for both training and inference, given y1, y2 … yn a continue sequence sampled from the ground truth sequence, as shown above we first compute the feature set used in posterior distribution:
f(W_tf, y1, y2…yn) = h1, h2 … hn, where hk gathers all the information from y1 to yk.

Qr(xn|y1…yn) = Qr(xn|hn) -> choose a conditional generative model as you want
Qr(x_k-1|xk, y1…y_k-1)= Qr(x_k-1|xk, h_k-1) -> choose a conditional generative model as you want.

Algorithm 3: training RevVMT with pre-trained GPT with subsequence pallerization

Init GPT: f(W_tf, …) no gradient required
Init Theta for Markov part of weight

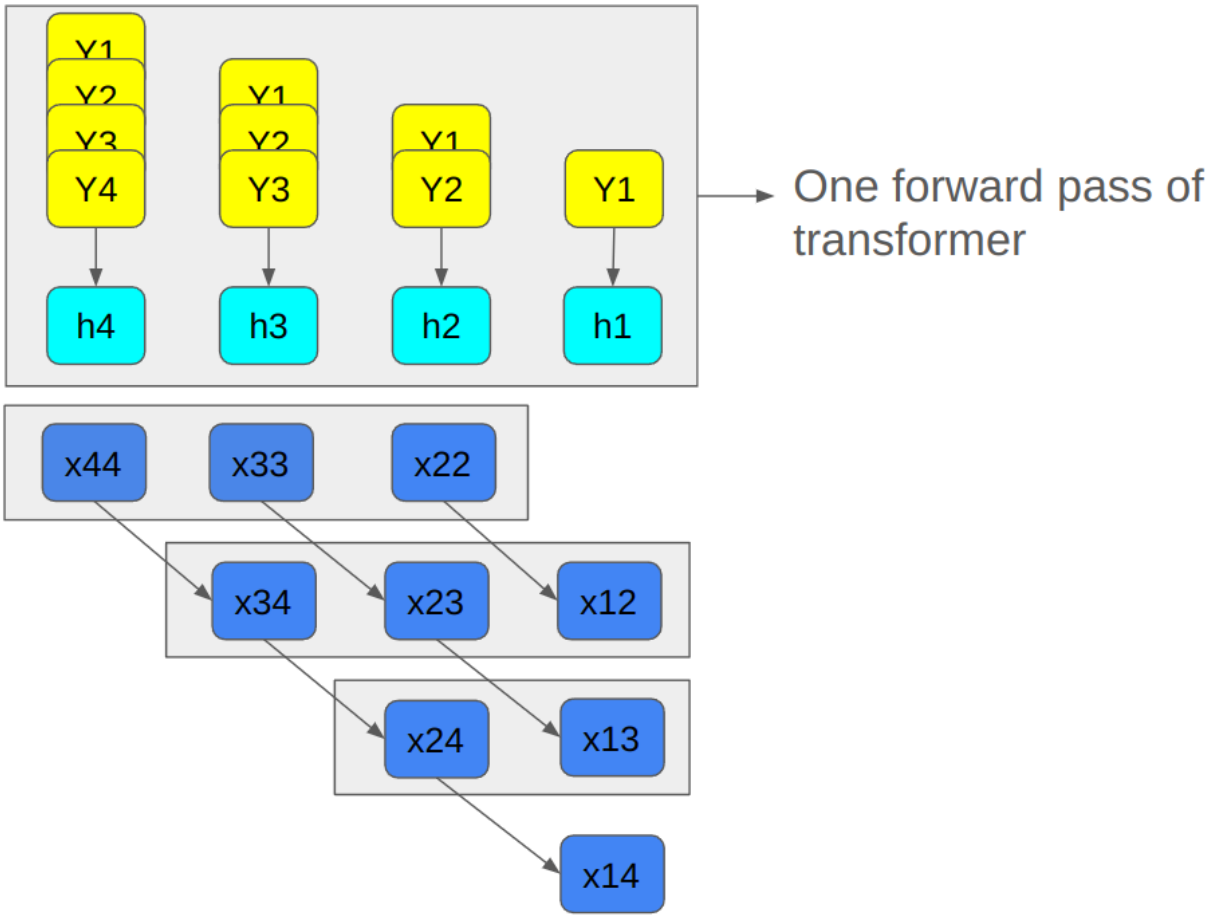
1. P(x_k)
2. P(y_k|x_k)
6. P(x_k-1|x_k)
7. Qr(xn|...)
8. Qr(x_k|x_k+1, ...)

Init optimizer
Set number of Xs sampled in one batch: N > 2000 according to memory capability

```
Set number of iterations M for each fixed sequence length
For L in range(2, 2000):
   For _ in range(0, M):
      Sample a continue main sequence of length Y1, Y2 …YL
      h1 …  h_N-1, h_N = f(W_tf, y1, y_2…yN)
      Loss = -ELBOS_reverse(hL …  h2, h1,yL, y_L-1…y1, Theta, N)
      grad1 = dLoss/dTheta
      Theta  <- optimizer(grad1, Theta)
```

Def ELBOS_reverse(hL … h2, h1,yL, y_L-1…y1, Theta, N):

```
      Choose subsequences to be included, selected_number:number of selected
      subsequences
      Init ELBO_sum to 0.
      For i in range(max_subsequence_L, 0, -1):
         For all subsequences having the ith item, sample all Xi.
         Accumulate the ELBO  brought by the ith  item from all sequences into ELBO_sum
      Return ELBO_sum/selected_number
```



Inference

Now we can obtain 5 neural networks from normal order training

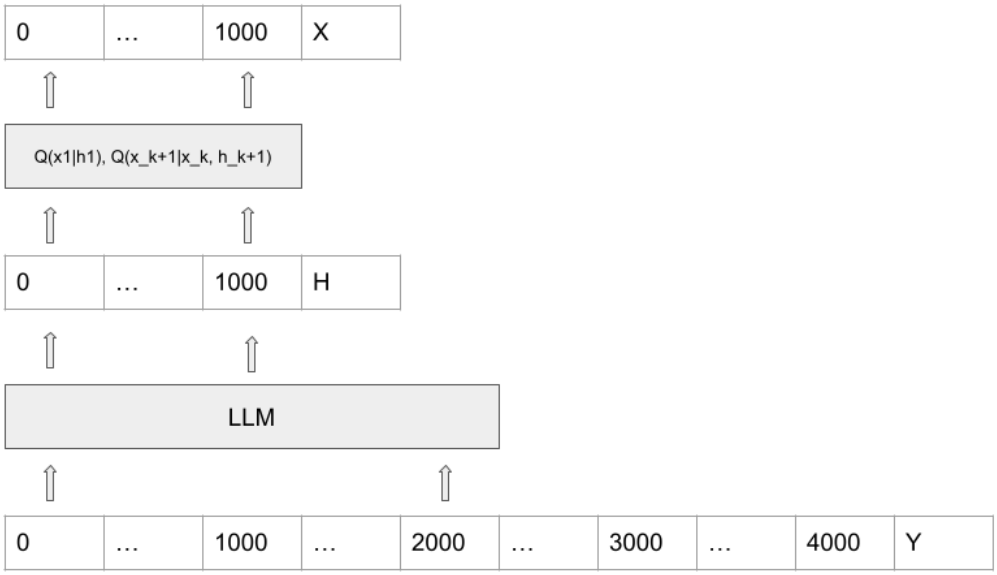
- P(x_k)
- P(y_k|x_k)
- P(x_k+1|x_k)
- Q(x1|y1…yn)
- Q(x_k+1|x_k, y_k+1…y_n)
- W_tf_reverse

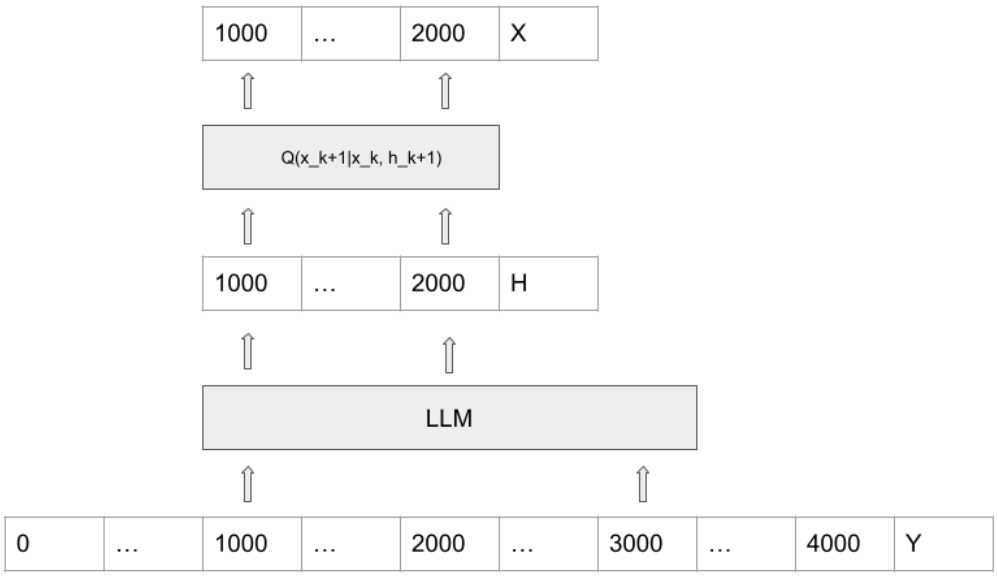
And 3 additional neural network from reverse order training

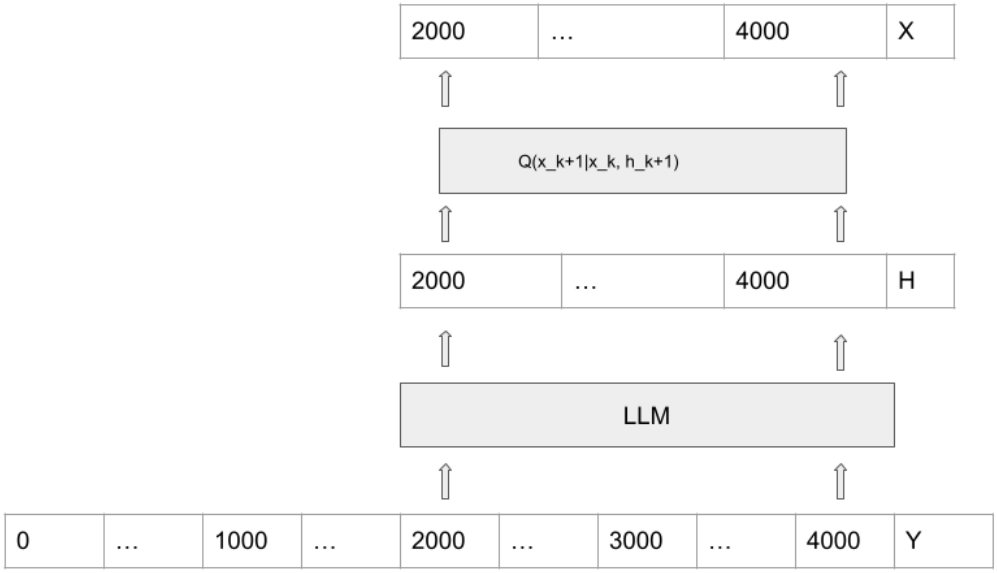
- Pr(x_k-1 |xk)
- Qr(xn|y1…yn)
- Qr(x_k-1|xk, y1…y_k-1)
- W_tf_gpt

How to use them during inference? Or concretely for example, how to use them for a chatbot application?
The user uploads a document then asks to summarize, the total length of the user's first input is of length Y1 …Y4000, the chatbot needs to read it first, then give its summary. Then the user will ask another question…

Reading:







As the training sequence length is 2000, we begin with the first 2000 tokens(even it's possible to exceed it), reading it with W_tf_reverse to get h1…h2000 then use Q(x1|h1) to sample the first X1 at the beginning of the input sequence.
Once we have X1, we could you Q(x_k+1|X_k, h_k+1) to sample the next Xs until X_1000, Now we could do another forward pass of W_tf_reverse to get h1000…h3000, thus we can adjust the Xs after X_1000 using more information. We repeat this process until we reach X_4000.
An acceleration trick is to use Qr(xn|y1…yn) to read immediately the first n tokens without updating X iteratively from x1 to xn.

Reading acceleration
One may notice that the reading during the inference might be slow, even with the acceleration of Qr(xn|yn) at the very beginning. One improvement to this is to change Q(x_k+1|xk, h_k+1) into Q(x_k+t|xk, h_k+1, T, t), where T is the total number of items between yn and x_k+1.
The Q(x_k+t|xk, h_k+1, T, t) can be trained at the same time of the initial training or be distilled later by Q(x_k+1|xk, h_k+1). Thus, when t >> 1, the model can read by skipping several steps.

Writing:
The goal of the reading is to find the fittest hidden state X according to the input content, Now, we could use P(x_k-1|x_k) and P(yk|xk) to sample X_4000+ and Y_4000+. Then reuse Q(x_k+1|X_k, h_k+1) to read new content from users.

Other possible usages include:

1. Reverse generation of text such as writing a prequel for a novel using Pr(x_k-1 |xk)
2. Sample randomly a text using P(x_k)

The distribution of P(X)

In this section, we discuss the nature of the hidden state X and its distribution P(X).

Every intellectual agent, its sequence of mental states from its birth until now can be seen as a sequence of Xs. If we put all Xs from the sequences of all intellectual agents who have generated the training data together, we get P(X).

We could imagine that P(X) may be a multimodal distribution as different types of text (Romantic novel Vs scientific paper) may correspond to very different sequences.

If X1 and X2 are close to each other, then the sequences they belong to may also share common memory. This makes it useful to store special Xs of a running sequence using a vector database with certain ponderation as long term memories. Each time the sequence encounters a x which is close to another x' that the database has stored long before, The model can use x' to generate recall memory sequence y' then read it. This may enhance the long term memory which is related to the agent's current state.

The X' stored in the vector database may vanish if never visited again.

Architecture choice

P(x_k)
Should be possible to evaluate LLH easily, can effectively model multimodal distribution, example of choice: BreezeForest

P(y_k|x_k)
Similar to the last layer in GPT that transforms a transformer's output into token's softmax.

P(x_k+1|x_k), Pr(x_k-1 |xk)
Conditional BreezeForest or simple gaussian

Q(x1|h1), Qr(xn|hn)
Conditional BreezeForest, conditional Affine coupling flow, when the condition sequence is short, Q(x1|h1), Qr(xn|hn) will resemble the property of P(X_k)

W_tf_reverse
a fine tuned GPT like LLM

W_tf_gpt
A pretrained GPT like LLM with fixed weight

Qr(x_k-1|xk, h_k-1), Q(x_k|x_k-1, h_k)

Conditional BreezeForest, conditional Affine coupling flow or gaussian