EECS 182 Deep Neural Networks
Fall 2025 Anant Sahai and Gireeja Ranade

Discussion 11

1. Inference-Time Compute

When generating outputs with an autoregressive sequence model, we face the question of inference-time compute: how to efficiently allocate computational resources to produce high-quality outputs. Different inference strategies have different computational profiles and can optimize for different objectives, possibly beyond just finding the highest probability sequence. Exhaustively evaluating all possible sequences is computationally intractable since there are $O(M^T)$ possibilities, where M is the vocabulary size and T is the sequence length. In this problem, we explore two major approaches with different compute-quality tradeoffs: beam search, which systematically explores promising sequences with deterministic computation, and sampling-based methods, which introduce controlled randomness to generate diverse outputs with variable computation.

Sampling

Sampling is one approach to decoding sequences. We simply sample from the distribution of possible tokens at each timestep using (possibly adapted) probabilities from the model. This allows us to explore a wider range of possible sequences more efficiently. When you sample multiple sequences, you can use additional (possibly external) criteria to select the best sequence.

Method	**Purpose**	**Sampling Adapter**
Ancestral sampling	$y \sim p_\theta$	– (Uses original model probabilities without modification)
Temperature sampling	$y \sim q(p_\theta)$	Rescale (Divides logits by temperature parameter to control randomness)
Greedy decoding	$y \leftarrow \max p_\theta$	Argmax (temperature$\rightarrow$ 0, selects highest probability token)
Top-k sampling	$y \sim q(p_\theta)$	Truncation (top-k, keeps only k highest probability tokens and renormalizes)
Top-p (nucleus) sampling	$y \sim q(p_\theta)$	Truncation (cumulative prob., keeps smallest set of tokens with cumulative probability at least p)

Table 1: Comparison of different decoding methods and their properties[1]

(a) Suppose you are using language models for a integer answer questions. **How would you use inference-time compute to select the best sequence?** Suppose we don't have access to the internal model states (e.g. logits, suppose we are querying an API) and we cannot directly score model responses. If all we can do is query the model multiple times, how can we use inference-time compute to select the best possible answer?

Solution: We can perform multiple queries and then use majority-vote to select the best response (the one we've seen the most often). We want to take the most common response rather than aggregating using mean since doing so could lead to an answer that is out of distribution or incorrect.

[1] From Decoding to Meta-Generation: Inference-time Algorithms for Large Language Models

Empirically, this method can yield better results than taking a single answer from the model.

(b) Now, suppose you are using language models for a code-generation task. Assume we have access to a syntax checker and can run our output against some test cases to validate our code. **How would you use inference-time compute to select the best sequence?**

Solution: You can sample multiple sequences, and then use existing test cases to evaluate the sequences. You can then select the sequence that passes the most test cases. Depending on your budget, you will need to tune the temperature and the number of samples to get a good trade-off between quality and cost.

Beam Search

Beam search is one of the popular approaches for finding the sequence with the highest overall probability according to the model.

It limits our search to only candidate sequences that are the most likely so far. In beam search, we keep track of the k most likely predictions of our model so far. At each timestep, we expand our predictions to all of the possible expansions of these sequences after one token, and then we keep only the top k of the most likely sequences out of these. In the end, we return the most likely sequence out of our final candidate sequences. This is also not guaranteed to be the true most likely sequence, but it is usually better than the result of just greedy decoding.

The beam search procedure can be written as the following pseudocode:

Algorithm 1 Beam Search

for each time step t **do**
  for each hypothesis $y_{1:t-1,i}$ that we are tracking **do**
    find the top k tokens $y_{t,i,1}$,...,$y_{t,i,k}$
  end for
  sort the resulting k^2 length t sequences by their total log-probability
  store the top k
  advance each hypothesis to time $t+1$
end for

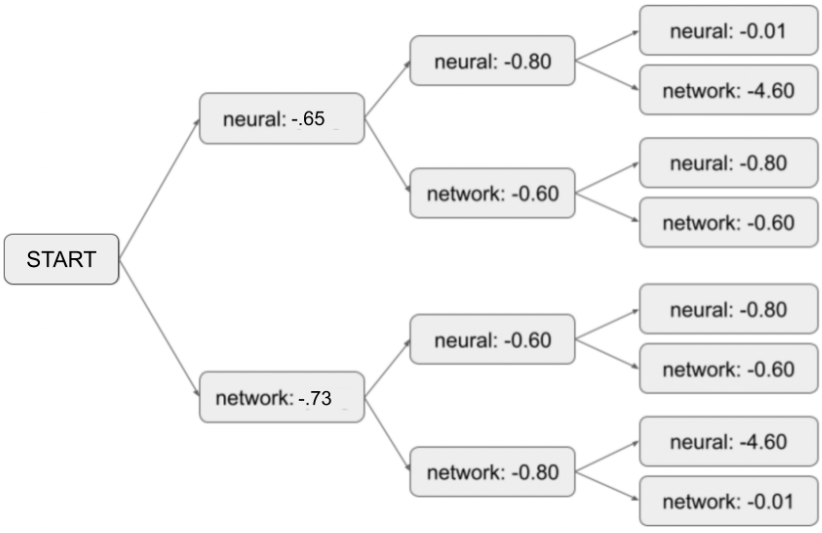

Figure 1: The numbers shown are the decoder's log probability prediction of the current token given previous tokens.

(a) Suppose we have an algorithm which can evaluate our responses and generate a score after each token. **How would you use inference-time compute to select the best sequence?** **Solution:** We can use a method like beam search where we keep track of the top k sequences based on the cumulative score from the evaluation algorithm.

Note that beam search-style methods are not limited to tracking model probabilities and can be used in any context where we can incrementally generate scores for partial outputs.

(b) If we don't have an external scoring algorithm, **how can we generate scores from the model itself?** **Solution:** We can treat the log probabilities from the model as scores, effectively allowing the model to score itself.

(c) We are running the beam search to decode a sequence of length 3 using a beam search with $k = 2$. Consider predictions of a decoder in Figure 1, where each node in the tree represents the next token **log probability** prediction of one step of the decoder conditioned on previous tokens. The vocabulary consists of two words: "neural" and "network".

i. **At timestep 1, which sequences are beam search storing?**

Solution: There are only two options, so our beam search keeps them both: "neural" (log prob = -.65) and "network" (log prob = -.73).

ii. **At timestep 2, which sequences are beam search storing?**

Solution: We consider all possible two word sequences, but we then keep only the top two, "neural network" (with log prob = -.65 - .6 = -1.25) and "network neural" (with log prob = -.73 - .6 = -1.33).

iii. **At timestep 3, which sequences are beam search storing?**

Solution: We consider three word sequences that start with "neural network" and "network neural", and the top two are "neural network network" (with log prob = -.65 - .6 - .6 = -1.85) and "network neural network" (with log prob = -.73 - .6 - .6= -1.93).

iv. **Does beam search return the overall most-likely sequence in this example? Explain why or why not.**

Solution: No, the overall most-likely sequence is "neural neural neural" with log prob = -.65 - .8 - .01= -1.46). These sequences don't get returned since they get eliminated from consideration in step 2, since "neural neural" is not in the $k = 2$ most likely length-2 sequences.

v. **What is the runtime complexity of generating a length-T sequence with beam size k with an RNN?** Answer in terms of T and k and M.

Solution:

- Step RNN forward one step for one hypothesis = $O(M)$ (since we compute one logit for each vocab item, and none of the other RNN operations rely on M, T, or K).
- Do the above, and select the top k tokens for one hypothesis. We do this by sorting the logits: $O(M \log M)$. (Note: there are more efficient ways to select the top K, for instance using a min heap. We just use this way since the code implementation is simple.) Combined with the previous step, this is $O(M \log M + M) = O(M \log M)$.
- Do the above for all k current hypotheses $O(KM \log M)$.
- Do all above + choose the top K of the K^2 hypotheses currently stored: we do this by sorting the array of K^2 items: $O(K^2 \log(K^2)) = O(K^2 \log(K))$ (since $\log(K^2) = 2\log(K)$). (Note: there are also more efficient ways to do this). Combining this with the previous steps, we get $O(KM \log M + K^2 \log(K))$. For typical values where $M \gg K$ (vocabulary size much larger than beam size), the first term dominates, giving us $O(KM \log M)$.
- Repeat this for T timesteps: $O(TKM \log M)$.

If we assume $K \ll M$ but assume the selection of top k out of n items can be done in $O(n)$ time (e.g. using quickpartition), then the complexity becomes $O(TKM)$.

(d) Again, consider a code generation task. Assume we have a syntax checker which can incrementally check our output for syntax errors (suppose the checker can distinguish between things which are definitely errors vs. things that could be correct if completed). **How could we modify beam search to take advantage of the code checker? Solution:** We could modify the scoring function to penalize sequences which are definitely syntax errors (e.g. give such tokens a score if $-\infty$).

This method was particularly popular for older code-generation models to ensure syntactically correct code. Modern models tend to favor sampling-based approaches, but scoring using a syntax checker could still be useful.

2. Finetuning Pretrained NLP Models

In this problem, we will compare finetuning strategies for three popular architectures for NLP.

(a) **BERT** - encoder-only model

(b) **GPT** - decoder-only model

(c) **T5** - encoder-decoder model.

In contrast to BERT or GPT, T5 is trained using a different masking objective. Spans of multiple words are masked out from the encoder. The decoder must predict the tokens for each span, as shown in Figure 3.

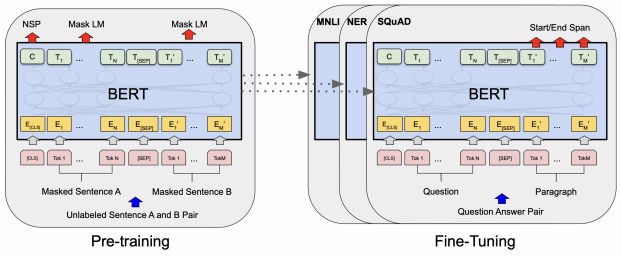

Figure 2: Overall pre-training and fine-tuning procedures for BERT.

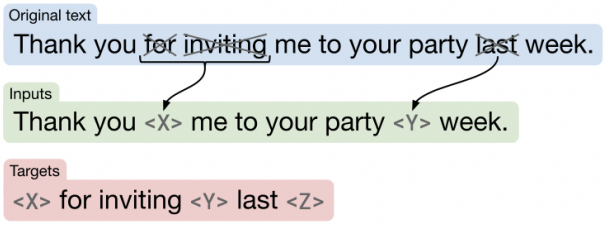

Figure 3: T5 Training procedure

Consider the MNLI (Multi-Genre Natural Language Inference Corpus) task. It provides a passage and a hypothesis, and you must state whether the hypothesis is an entailment, contradition, or neutral.

EXAMPLE:
Passage: At the other end of Pennsylvania Avenue, people began to line up for a White House tour.
Hypothesis: People formed a line at the end of Pennsylvania Avenue.
Classification: entailment

(i) With each of the 3 models, state whether it is possible to use the model for this task with no finetuning or additional parameters. If so, state how.

Solution: With GPT and T5, you could accomplish this by **few-shot prompting**. Input a set of examples (passage, hypothesis, and answer), followed by the passage and hypothesis for the current question. The decoder would then predict the answer.

It is technically possible to use BERT for this task by inputting a prompt, masking a token for the answer, and predicting it, but BERT is not good at learning through prompting.[2].

(ii) With each of the 3 models, state how you would use the model for this task if you were able to add additional parameters and/or finetune existing parameters.

Solution: With GPT and T5, you could input the passage and hypothesis, then finetune the model to predict the answer as the next token. Note that with large models it is often common to only finetune a subset of the model parameters.[3]

With BERT, you input a sequence to the model: [<CLS>, passage, <SEP>, hypothesis]. Remove the output layer of the network and replace it with a classification head on the CLS output token. (Some approaches also make use of the other tokens, such as by taking a mean across the other output tokens and concatenating them with the CLS token.) You can either finetune all parameters or only the parameters of the classification head.

(Students may suggest other answers which are valid too - e.g. you could extract features from GPT by concatenating token features from the last couple of layers and training a classification head on top of them, but these may not perform as well).

(iii) How can we adapt soft-prompting to work with a T5-style model?

Solution: We can add some learnable tokens to the start of the encoder input. During finetuning, we can freeze the model weights and only train the embeddings for these added tokens.

Contributors:

- CS 182 Staff from previous semesters.
- Olivia Watkins.
- Kumar Krishna Agrawal.
- Dhruv Shah.
- Naman Jain.
- Anant Sahai.
- Kevin Li.
- CS 182/282A Staff from previous semesters.

[2]For new work modifying BERT to benefit from prompting, see https://arxiv.org/pdf/2201.04337.pdf

[3]For a comparison between of several parameter-efficient finetuning methods, check out the experiments in this paper: https://arxiv.org/pdf/2205.05638.pdf

