ANLP ex1

link to Github repository https://github.com/danirude/ANLP-ex1.git

Part 1 :Open Questions

Q1.1

Dataset 1: Stanford Natural Language Inference (SNLI)

SNLI is a dataset designed for natural language inference (NLI), where the task is to determine the relationship between a premise and a hypothesis (e.g., entailment, contradiction, or neutral). It captures intrinsic language understanding by requiring models to reason about semantic relationships and the logical connections between sentences.

Dataset 2: Stanford Question Answering Dataset (SQuAD)

SQuAD is a reading comprehension dataset where questions are posed based on a set of Wikipedia articles, with the answers being exact spans from the text. It measures intrinsic language understanding by evaluating a model's ability to comprehend context, identify relevant information, and precisely extract answers from unstructured text.

Dataset 3: Multi-Genre NLI (MNLI)

MNLI is a large-scale natural language inference dataset with examples from a diverse range of text genres. It tests intrinsic language understanding by requiring models to determine the relationship (entailment, contradiction, or neutral) between pairs of sentences, challenging their ability to handle varied linguistic contexts and complex semantic reasoning.

Q1.2A

The methods we discussed in class are:

1. Self-Consistency

2. Verifier-Guided Selection

3. Verifier-Guided Partial Generation

4. More Generations, Not Bigger Models

5. Longer chains of thought

6. O1-Style Reasoning Loops

7. DeepSeek R1

Method1 : Self-Consistency

Brief description
The model answers the same prompt multiple times with stochastic sampling so that each run traces a different reasoning path. After generation ends, only the final answers are considered. The system counts how often each answer appears and returns the one with the highest frequency. Performance improves because the fixed model is effectively averaged over several independent inference passes.

Advantages
Sampling several solutions and voting lifts accuracy on reasoning-focused tasks without extra training. The technique works with any generative model that supports random sampling and naturally reduces isolated hallucinations, since outlier answers rarely win the vote. Implementation is straightforward: run additional inference passes and tally the outcomes.

Computational bottlenecks
Each additional sample is a full forward pass, so runtime and total GPU usage grow in direct proportion to the number of runs. When chains of thought are long, more memory or bandwidth is required to store them until voting occurs. Post-processing to normalise answers before counting is minimal next to the sampling cost.

Parallelization
Every generation is independent, allowing the work to be split across GPUs, CPU workers, or separate machines. Once all runs finish, a brief aggregation step counts answer frequencies and selects the majority, keeping synchronization overhead low.

Method 2: Verifier-Guided Selection

Brief description
The language model produces several candidate answers for the same prompt. A separate verifier—this can be another model or a task-specific automatic checker—evaluates each candidate and assigns a quality score or a pass/fail tag. The system

keeps the highest-scoring answer or the subset that meet a predefined threshold. All gains come from this extra filtering step, not from changing the main model's weights.

Advantages
Filtering removes many hallucinations and factual slips, so answer quality rises even with a modest number of candidates. The verifier can encode task-specific rules (unit tests, regex checks, fact-consistency prompts) without retraining the large model. Because poor candidates are discarded, the method often reaches high accuracy with fewer generated samples compared with simple majority voting. The verifier itself is modular, so it can be swapped or improved independently.

Computational bottlenecks
The approach doubles computation: first the cost of generating multiple candidates, then the cost of scoring each one with the verifier. If the verifier is another large model, its passes can rival generation time. Storing lengthy outputs until they are scored also increases memory usage, especially for code or long reasoning chains.

Parallelization
Both stages parallelize well. Candidate generation can be spread across devices, and verifier evaluations are independent, allowing the scoring step to run in parallel too. A short reduction step at the end selects the best-rated answer, so overall synchronization overhead is minimal.

Method 3: Verifier-Guided Partial Generation

Brief description
Instead of letting the model write a full answer in one shot, this method grows the answer step by step. At each step the model proposes several partial continuations. A verifier—another model or a task-specific checker—scores these partial paths, and only the most promising ones are expanded further. The search continues until a complete answer is reached. In effect, the verifier steers the exploration, pruning weak branches early and focusing compute on likely correct reasoning chains.

Advantages
Early pruning cuts wasted generation, so you often reach high-quality answers with fewer total tokens than full-length sampling methods. Because the verifier looks at unfinished work, domain rules or test cases can guide the search sooner, boosting both accuracy and interpretability. The approach separates roles cleanly: the generator handles language fluency, while the verifier enforces task constraints, and each component can be upgraded independently.

Computational bottlenecks
The main cost is the repeated loop of "generate a batch of partial continuations, then run the verifier." If the verifier is itself a large model, its evaluations add substantial compute and latency. Maintaining the search tree in memory, especially when beams or branching factors are large, can also raise GPU memory requirements.

Parallelization
Both generation and verification steps can be parallelized. Different branches of the

search tree are independent, so you can distribute them across multiple GPUs or machines. After each expansion round, a quick aggregation stage selects the top-scoring partial paths to keep, then pushes them back out for the next parallel expansion, keeping synchronization overhead modest.

Method 4: More Generations, Not Bigger Models

Brief description
Rather than upgrading to a larger language model, this approach keeps the model size fixed and simply asks it to produce many more candidate answers per prompt. After sampling, a lightweight selector—often log-probability, a task-specific heuristic, or a simple reranker—chooses the best answer from the set. The extra breadth of generations simulates the diversity you would have gained by training a bigger model, but the cost is paid only at inference time.

Advantages
Quality often rises to match or even beat that of a larger model without any retraining or additional parameters on disk. The technique is model-agnostic, easy to integrate, and lets you trade wall-clock inference time for accuracy on demand. It also offers flexibility: the number of generations can be tuned per query, so you spend more compute only when it matters.

Computational bottlenecks
Total compute and latency grow linearly with the number of generations because each candidate requires a full forward pass. Storing long outputs until the selection step can strain memory or I/O bandwidth, especially if you keep log-probs or other metadata. The final scoring or reranking phase is typically lightweight by comparison but still adds a small CPU or GPU overhead.

Parallelization
Each generation is independent, making the workload parallel. You can spread the sampling across multiple GPUs, CPU workers, or separate machines, then gather the outputs for a quick selection pass. Synchronization cost is low because the only shared step is choosing the top answer at the end.

Method 5: Longer chains-of-thought

Brief description
This strategy extends the token budget for each response so the language model can follow a much deeper reasoning path before producing its final answer. The extra tokens act as additional compute cycles: intermediate ideas can be explored, self-corrections made mid-stream, and the model often converges on a more reliable conclusion within a single, longer generation.

Advantages
Deeper chains tend to improve accuracy on tasks that demand multi-step logic,

mathematics, or planning. No auxiliary models, rerankers, or specialised infrastructure are required—only a higher maximum sequence length or a looser stopping rule. The full reasoning trace also becomes available for inspection, supporting transparency and error analysis.

Computational bottlenecks
Longer sequences increase both latency and total floating-point operations because each extra token adds a full decoder step. Transformer self-attention scales quadratically with sequence length, so memory usage can climb sharply on very long traces. Storing or transmitting the extended chains likewise adds I/O overhead.

Parallelization
This method can't be parallelized. Generation within a single long chain is inherently sequential; every new token depends on all previous tokens, so the logical computation cannot be split across devices for speed-up.

Method 6: O1-Style Reasoning Loops

Brief description
The model tackles a task through repeated "draft, critique, revise" cycles. In each loop it writes a partial or full answer, evaluates that answer (often via a separate reflection prompt or a verifier), then decides whether to keep, back-track, or refine it before moving on. The process continues until a stopping rule is met—such as a high confidence score or a maximum number of iterations—yielding an answer that has been self-checked multiple times inside a single query.

Advantages
Iterative self-evaluation catches logical slips early and allows the reasoning path to be corrected rather than abandoned. Accuracy can exceed simple one-shot generation or even majority-vote sampling, because each loop builds on feedback from the previous one. The framework is modular: different critic prompts, verifiers, or search heuristics can be swapped in without changing the base language model.

Computational bottlenecks
Every cycle incurs at least one additional forward pass for generation and one for evaluation, so total latency grows with the number of loops. If each iteration produces long chains of thought, memory and I/O costs accumulate as those traces are stored for later reference. When an external verifier model is involved, its evaluations add further compute overhead.

Parallelization
Within a single query the loops are sequential—each critique depends on the latest draft—so the core logic cannot be parallelized for speed. Parallel execution is limited to running separate queries in parallel or dispatching different instances of the task to multiple devices. Thus, wall-clock time for one reasoning session scales roughly with the chosen number of loops.

Method 7: DeepSeek R1

Brief description
DeepSeek R1 frames inference as an evolutionary search. A population of candidate answers is first generated. Each candidate is then mutated or recombined by prompting the language model to produce refined variants. An automatic evaluator scores these variants, and the highest-scoring ones form the next generation. After several generations, the best individual in the population is returned as the final answer. All improvement comes from this repeat-and-select cycle rather than from updating model weights.

Advantages
Population search explores a broader solution space than single-pass or beam methods, helping the system escape local optima. Evolutionary pressure plus repeated refinement often yields answers that outperform those from self-consistency or verifier-only pipelines, especially on open-ended tasks such as coding or long-form reasoning. Components are modular: the evaluator can be task-specific or learned, and mutation prompts can be tuned without altering the base model.

Computational bottlenecks
Every generation requires new candidate production and evaluation, so compute grows with population size and the number of generations. If the evaluator is itself a large model, its passes can double the cost. Storing multiple long answers across generations increases memory demand, and crossover or mutation logic introduces additional CPU overhead for prompt construction.

Parallelization
Within each generation, candidate creation and evaluation are independent, making those steps highly parallelizable across GPUs or machines. The evolutionary loop remains sequential across generations because each new round depends on the scores from the previous one, so wall-clock time scales with the chosen number of generations despite intra-generation parallelism.

Q1.2B

Given a complex scientific task that requires deep reasoning and access to only a single GPU with large memory capacity, 'O1-style reasoning loops' is the preferred method. The draft, critique, revise cycle runs sequentially, so techniques like self-consistency, or verifier-guided selection which are normally accelerated by spreading work across multiple GPUs—cannot leverage their parallel advantage in this setup. At the same time, the GPU's large memory can hold both the model weights and the lengthy chains of drafts and critiques that accumulate over several iterations, ensuring smooth in-device execution without slow transfers or out-of-memory failures. This combination makes O1-style loops the most effective way to turn the available hardware into deeper, self-correcting reasoning, while more parallel-friendly alternatives would see little benefit under the single-GPU constraint.

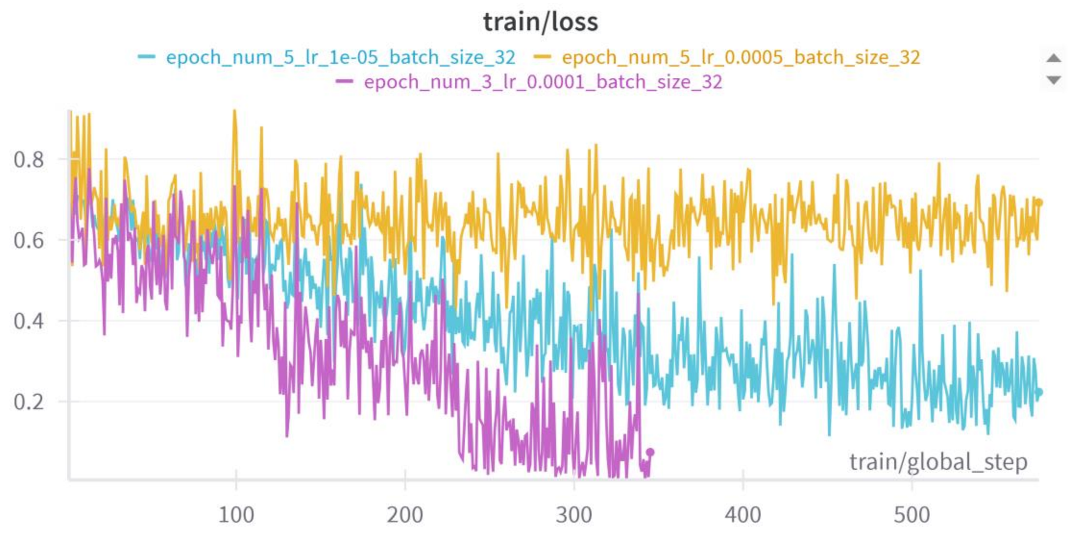
Part 2 : Programming Exercise

Q2.1

The configurations I used were:

- Epoch 3, Learning Rate 0.0001, Batch Size 32
- Epoch 5, Learning Rate 0.0005, Batch Size 32
- Epoch 5, Learning Rate 1e-05, Batch Size 32

The train loss plot I got was:



Additionally, the validation Accuracy and test Accuracy of each model were:

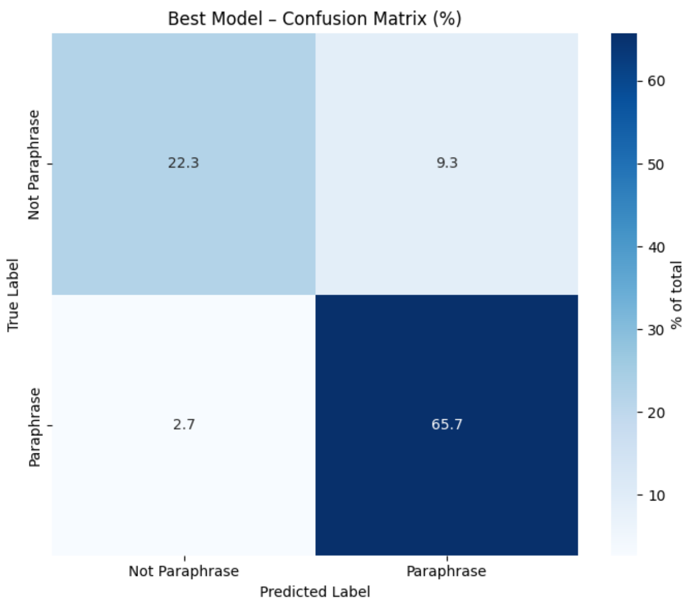
Model	Validation Accuracy
epoch_num_3_lr_0.0001_batch_size_32	**0.8799**
epoch_num_5_lr_0.0005_batch_size_32	0.6838
epoch_num_5_lr_1e-05_batch_size_32	0.8211

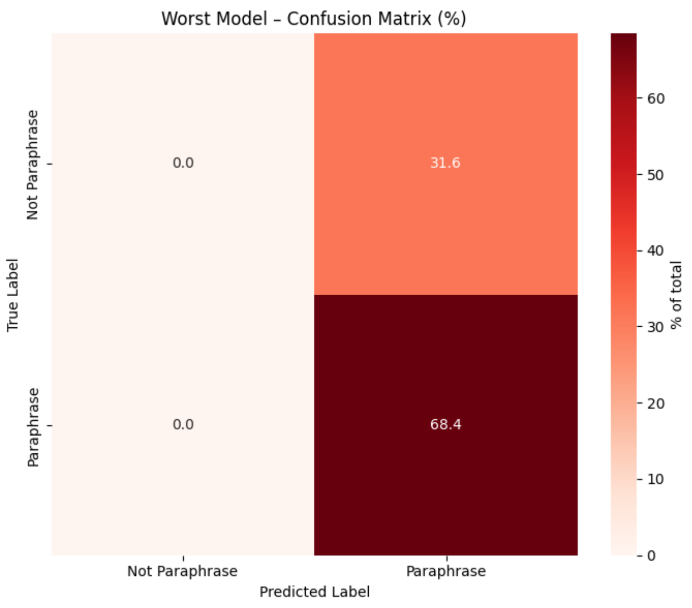
Model	Test Accuracy
epoch_num_3_lr_0.0001_batch_size_32	**0.8481**
epoch_num_5_lr_0.0005_batch_size_32	0.6649
epoch_num_5_lr_1e-05_batch_size_32	0.7959

As can be seen from these 2 tables, the configuration that achieved the best validation accuracy also achieve the best test accuracy (Epoch 3, Learning Rate 0.0001, Batch Size 32).

Qualitative analysis:

In order to compare the best and worst performing configurations, I created a confusion matrix for the best configuration and a confusion matrix for the worst configuration. Which are shown here:





From the confusion matrixes, it is clear that the worst configuration simply labeled everything as Paraphrase. As a result of this, examples that were labeled as Not Paraphrase by the human annotators were harder for the lower-performing configuration. Unlike the worst configuration, the best configuration still spotted most paraphrase and not-paraphrase pairs (though it was more successful at detecting paraphrases then not-paraphrases), showing it really learned to detect meaning differences.