Decoupling Input and Output Vocabularies Using Over Encoding on GPT-2

Giorgos Tzamouranis | Nikos Katsaidonis | Andreas Fotakis | Kyriakos Katsiadramis

National Technical University of Athens

Abstract

Tokenization is critical to language modeling performance but remains underexplored. Extending the recent framework of Over-Tokenized Transformers, we investigate the effects of multi-gram embeddings in GPT-2 models. Initially, we trained GPT-2 from scratch on a context-free grammar dataset with over-encoding, demonstrating improvements in training loss and generation accuracy. Additionally, we introduced a fine-tuning method on pretrained GPT-2 by augmenting embeddings with mean-initialized 2-gram and 3-gram tokens, selectively training these additions. Extensive experiments on various downstream tasks confirm that our approach consistently improves performance. Our results reinforce tokenization's significant potential for optimizing pretrained models efficiently

Introduction

Tokenization significantly influences the performance and scalability of modern large language models (LLMs), yet its implications remain underexplored in existing literature. Recently, Over-Tokenized Transformers introduced the concept of scaling input vocabularies through multi-gram tokenization, demonstrating improved language modeling efficiency without additional computational overhead. Building upon these insights, this paper extends experimentation to the GPT-2 model, investigating how multi-gram tokenization and embedding strategies impact model performance and scalability.

We initially constructed a context-free grammar (CFG) dataset to train GPT-2 from scratch, applying an over-encoding strategy to increase vocabulary size and assess its effects on training loss and generation accuracy. This approach yielded clear improvements, underscoring the potential of tokenization adjustments even for models trained from scratch.

Furthermore, we introduced an innovative fine-tuning methodology for pretrained GPT-2. By expanding the embedding layer with new 2-gram and 3-gram embeddings—initialized as the mean of constituent token embeddings—we selectively trained only these new embeddings. The training indicated a log linear relationship between trainable embeddings and loss (Figure 1). Extensive evaluations on diverse downstream benchmarks show that this fine-tuning method seems to generally improve accuracy on context-creation tasks.

Lastly, we extended our exploration to larger n-grams (4-gram and 5-gram embeddings), further examining scalability and performance limits of multi-gram embeddings. Our comprehensive experimentation highlights that multi-gram tokenization, combined with thoughtful embedding strategies, presents substantial opportunities for enhancing the efficiency and capability of pretrained LLMs, offering practical guidance for future tokenizer design and embedding initialization methodologies.

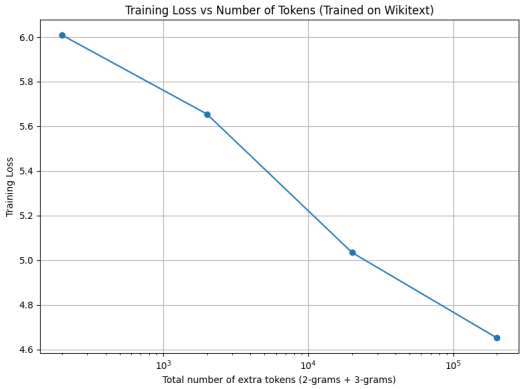


Figure 1: Training Loss vs Number of Tokens.

Relation to Prior Work

Our research is largely inspired by the recent work of Huang et al. (2025), titled *"Over-Tokenized Transformer: Vocabulary is Generally Worth Scaling"*. This study introduced the Over-Tokenization method, which decouples input and output vocabularies by scaling input tokens through multi-gram embeddings. Their results revealed a near log-linear relationship between input vocabulary size and model performance, suggesting that increasing vocabulary granularity improves language modeling efficiency.

While we adopt the same foundational idea of enriching input representations via multi-gram tokens, our research differs in key ways. Specifically, we extend the approach in two directions: (1) training GPT-2 from scratch on a synthetic CFG dataset using a structured over-encoding scheme, and (2) selectively fine-tuning new multi-gram embeddings on top of a frozen pretrained GPT-2. Furthermore, we evaluate performance across both context-completion and commonsense reasoning tasks, allowing us to analyze the benefits and limits of vocabulary augmentation in different settings.

Over-Encoding GPT-2 from Scratch with CFG Dataset

Dataset Creation

In this experiment, we constructed a synthetic dataset using a context-free grammar (CFG) with a limited vocabulary: $V = \{1, 2, 3\}$. The dataset contained 200.000 sequences generated based on specific CFG production rules, which are illustrated in Appendix C. The sequence lengths varied up to 729 characters. We divided the dataset into a training set of 160.000 samples and a validation set of 40.000 samples.

Methodology

The key idea of Experiment 1 was to initialize a GPT-2 model with randomly initialized parameters, targeting approximately 2.5 million parameters. Specifically, we replaced the embedding layer of GPT-2 using the over-encoding method. The over-encoding strategy leverages hierarchical n-gram tokenization to enrich the input representations. Instead of having embeddings only for single tokens (1-grams), we also included embeddings for all possible 2-grams and 3-grams.

Due to the restricted vocabulary size $\{1, 2, 3\}$, the total number of possible n-grams was limited:

- 1-grams: 4 tokens
- 2-grams: 16 tokens (4^2)
- 3-grams: 64 tokens (4^3)

Because of this small vocabulary, performing logarithmic scaling on input vocabulary size by the factor m was impractical. Instead, we incrementally adjusted the number of n-grams to observe the effects on performance.

Technical Details of Over-Encoding The key challenge in over-encoding lies in representing a potentially vast number of n-gram combinations using a limited-size embedding table. The total number of possible n-grams grows exponentially with vocabulary size:

$$\text{Number of } n\text{-grams} = V^n$$

where V is the size of the base vocabulary.

To address this, we apply a hash-based indexing scheme that allows each n-gram to be assigned to a specific row in the embedding table, even under memory constraints. Given a sequence of token IDs $z_1, z_2, \ldots, z_n$, and a fixed integer base $p = V$, we compute a hash index using the following function:

$$f(z_1, \ldots, z_n) = \sum_{i=1}^{n} z_i \cdot p^{i-1}$$

This function generates a unique integer for each possible n-gram combination. To map this to a specific row in a table of fixed size m, we use the modulo operation:

$$x_{\text{hashed}} = f(z_1, \ldots, z_n) \bmod m$$

The value x_{hashed} corresponds to the row index of the embedding table for the given n-gram. Naturally, the smaller m is, the more collisions occur (i.e., multiple n-grams mapped to the same row), which may reduce representational capacity.

We define the full **OE-m embedding layer** as consisting of n separate embedding tables (one for each n-gram length), each of size $m \times d$, where d is the embedding dimensionality.

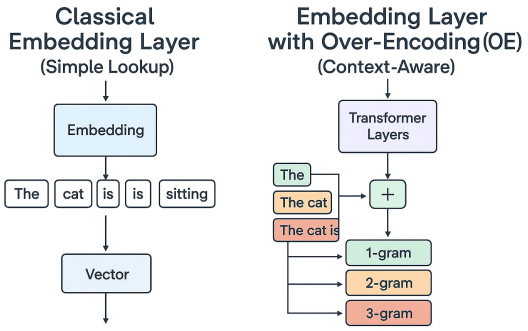


Figure 2: Embedding Layer with/without OE.

Training Details

We trained the GPT-2 model from scratch using the constructed CFG dataset. The model was optimized using the AdamW optimizer with a learning rate of 3×10^{-4}. We conducted training for a total of 10 epochs with a batch size of 64. Throughout training, we measured the training loss and generation accuracy on the validation set.

Results

We trained models with different input vocabulary sizes and compared them against the baseline GPT-2. The results demonstrated that using the over-encoding embedding strategy consistently improved model performance. We observed reductions in training loss and increases in generation accuracy relative to the baseline GPT-2 model without over-encoding (Figure 3). Additional figures are provided in the Appendix A.

However, due to the very limited vocabulary, the magnitude of improvement remained modest. As a result, we did not observe the expected log-linear relationship between vocabulary size and loss, since we were unable to increase the vocabulary size logarithmically. (A logarithmic scale will be explored in Experiment 2, where the vocabulary is significantly larger.)

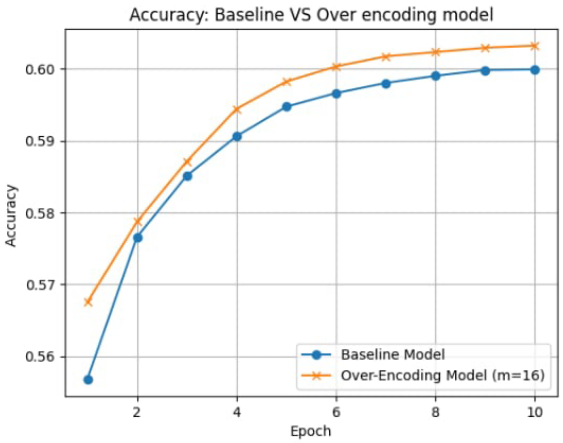


Figure 3: Generation Accuracy vs Epoch.

Fine-tuning Pretrained GPT-2 with multi-gram Embeddings

The objective of this experiment is to explore the impact of integrating multi-gram embeddings into a pretrained GPT-2 model. The motivation stems from the hypothesis that multi-gram token embeddings can capture a more descriptive representation of the text than single-token embeddings, potentially improving performance on various downstream tasks without incurring the significant computational costs associated with training models from scratch.

Methodology

Embedding Extension Technique

We maintained the original single-token (1-gram) embeddings from the pretrained GPT-2. To this original embedding layer, we added additional embeddings for frequently occurring 2-gram and 3-gram phrases identified from the training corpus. In fact, we used Wikitext and Wikitext-2-raw as the training corpus for extracting n-grams. These newly added embeddings were explicitly created to extend the original embedding table without modifying the pretrained parameters.

Embedding Initialization

Each new multi-gram embedding was initialized as the mean of the constituent single-token embeddings from the pretrained GPT-2. For example, the embedding for a bigram ”machine learning” was initialized as the average of the embeddings for ”machine” and ”learning”. This initialization approach was chosen to leverage pretrained knowledge, facilitating efficient fine-tuning.

Selective Fine-tuning

During fine-tuning, only the newly added multi-gram embeddings were trained, while the original embeddings of GPT-2 were kept frozen. This selective fine-tuning was designed to minimize computational overhead while maximizing the utilization of pretrained weights.

Experimental Setup

Datasets and Tasks

We evaluated our fine-tuned models using three downstream tasks:

- **HellaSwag**: Tests commonsense inference and narrative coherence.
- **PIQA**: Assesses physical commonsense reasoning.
- **Story Cloze**: Measures narrative understanding and context coherence.

These tasks were selected due to their distinct challenges in assessing contextual and commonsense reasoning abilities.

Training Details

For training, we used the AdamW optimizer with a learning rate of 1×10^{-3}, running for 20 epochs with a batch size of 8. During preprocessing, we replaced multi-gram phrases with special pseudo-tokens to facilitate fine-tuning.

Results

Our experimental results demonstrated consistent performance improvements when utilizing multi-gram embeddings on context-creation downstream tasks such as HellaSwag and Story Cloze. Specifically, models fine-tuned with varying quantities of multi-gram embeddings (ranging from 100 to 100,000 bigrams and trigrams, increased logarithmically). As a result, baseline GPT-2 was consistently outperformed in these tasks, with larger multi-gram vocabularies generally correlating with incremental performance gains, indicating effective utilization of richer contextual representations.

Despite consistent gains on context-completion tasks such as HellaSwag and Story Cloze, the n-gram-augmented models failed to surpass—and sometimes slightly lagged—the baseline on PIQA. This discrepancy arises from task alignment: richer multi-gram representations strengthen GPT-2 on contextual-creation tasks but not on commonsense question answering, which demands reasoning skills and domain knowledge that a purely autoregressive GPT-2 has never been trained for (Figure 4).

Last but not least, increasing the extra n-gram embeddings logarithmically led to a significant reduction in training loss. Specifically, by increasing the number of n-gram tokens from 100 3-grams and 100 2-grams to 100.000 3-grams and 100.000 2-grams, we observed a log-linear relationship between training loss and number of extra trainable embeddings (Figure 1). Additional figures are provided in the Appendix B.

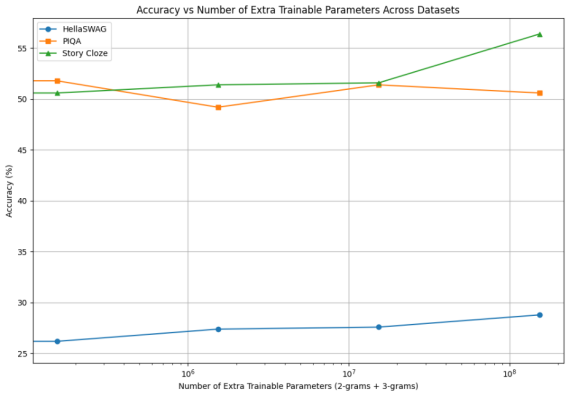


Figure 4: Downstream Tasks Accuracy Comparison.

Extension to 4-gram and 5-gram Embeddings

As an extension of our primary investigation, we also explored the impact of incorporat-

ing higher-order embeddings, specifically 4-grams and 5-grams, using the same methodological framework as in our previous experiments. The outcomes from these experiments closely mirrored our earlier findings with 2-gram and 3-gram embeddings.

While performance on context-driven tasks did not significantly improve beyond the earlier experiments, the trends remained consistent with our previous observations. Each jump from 3-grams to 4-grams and from 4-grams to 5-grams yielded progressively smaller gains, implying that beyond a certain n-gram order the computational overhead may outweigh the modest benefits (Figure 5).

Detailed plots illustrating the training loss and accuracy metrics across selected tasks will be provided on Apendix B to offer clearer insights into these observations.

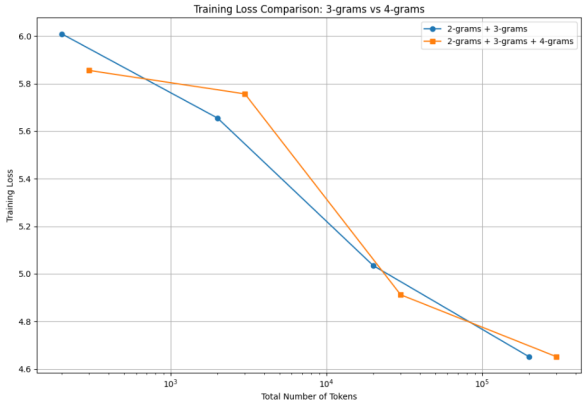


Figure 5: 3-gram vs 4-gram Loss

Conclusion

In this study, we extensively investigated the impact of multi-gram tokenization and embedding strategies within the GPT-2 architecture, inspired by recent advancements in Over-Tokenized Transformers. Our experiments highlighted two distinct but complementary approaches.

In Experiment 1, training a GPT-2 model from scratch on a synthetic context-free grammar (CFG) dataset using over-encoding significantly improved language modeling performance, reducing significantly the training loss and increasing models' accuracy.

Experiment 2 introduced a fine-tuning strategy: selectively training new multi-gram embeddings initialized from pretrained single-token embeddings, which consistently enhanced performance on context-driven downstream tasks such as HellaSwag and Story Cloze. However, this approach provided limited improvements in tasks like PIQA, highlighting the task-specific applicability of multi-gram strategies.

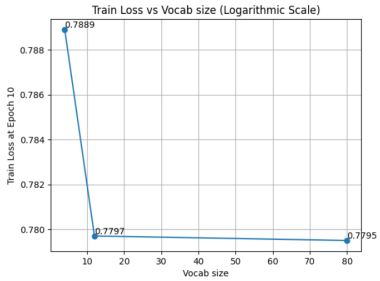
Overall, our research underscores the substantial potential of multi-gram embeddings in augmenting language model capabilities, while emphasizing the necessity of aligning tokenization and embedding techniques with the characteristics and requirements of target applications.

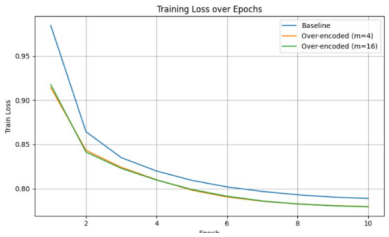
Future research could further explore QA-specific fine-tuning strategies, larger instruction-tuned models, and advanced reasoning techniques to maximize the effectiveness of multi-gram embedding methods.

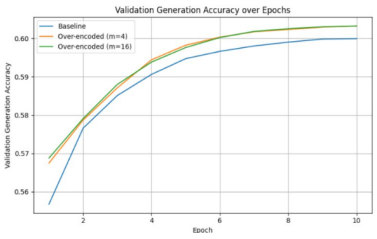
References

[1] Y. Huang, Y. Ge, S. He, Z. Du, H. Zhou, X. Tan, W. Chen, S. Li, and M. Zhou, *Over-Tokenized Transformer: Vocabulary is Generally Worth Scaling*, arXiv preprint arXiv:2501.16975, 2025. Available at: https://arxiv.org/abs/2501.16975

Appendix A: CFG figures



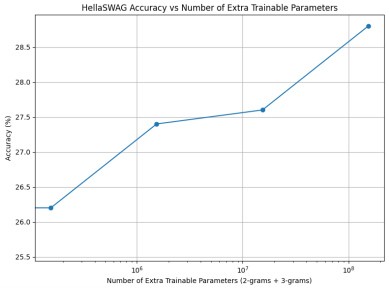


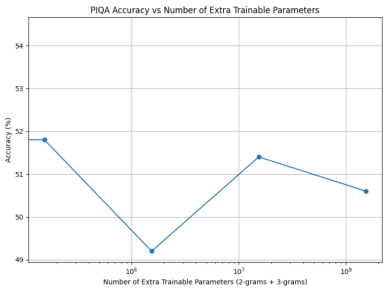


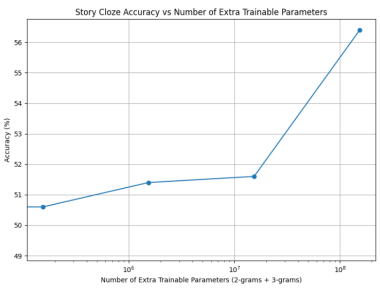
Train Loss vs Vocab Size Training Loss over Epochs Validation Generation Accuracy

Figure 6: CFG training results. Comparison of loss and accuracy metrics under different over-encoding setups.

Appendix B: Fine Tuned GPT-2 Figures

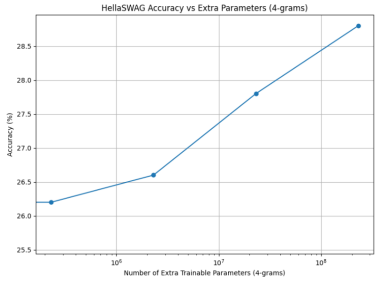


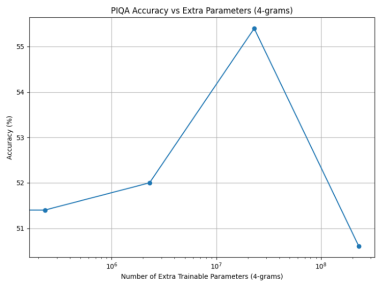


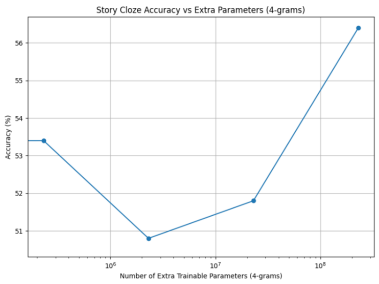


HellaSWAG PIQA Story Cloze

Figure 7: Accuracy vs Extra Trainable Parameters for 2- and 3-grams.

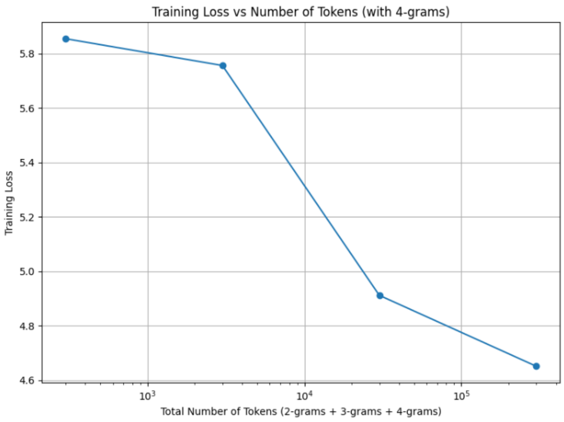




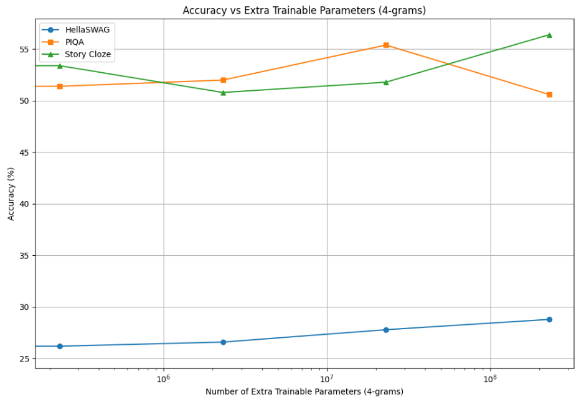


HellaSWAG (4-grams) PIQA (4-grams) Story Cloze (4-grams)

Figure 8: Accuracy vs Extra Trainable Parameters for 4-grams.



Training Loss vs Number of Tokens



Accuracy Comparison Across Tasks (4-grams)

Figure 9: Training loss and task-wise accuracy using 4-gram embeddings.

Appendix C: Technical Information

```
root |->20 21      19|->18 16 18   16|->15 15      13|->11 12      10|->8 9 9   7|->2 2 1
root |->20 19 21   19|->17 18      16|->13 15 13   13|->12 11 12   10|->9 7 9   7|->3 2 2
root |->21 19 19   19|->18 18      16|->14 13      13|->10 12 11   10|->7 9 9   7|->3 1 2
root |->20 20      20|->16 16      16|->14 14      14|->10 12      11|->8 8     7|->3 2
                   20|->16 17      17|->15 14 13   14|->12 10 12   11|->9 7     8|->3 1 1
                   20|->17 16 18   17|->14 15      14|->12 11      11|->9 7 7   8|->1 2
                   21|->18 17      17|->15 14      14|->10 12 12   12|->7 9 7   8|->3 3 1
                   21|->17 16      18|->14 15 13   15|->10 11 11   12|->9 8     9|->1 2 1
                   21|->16 17 18   18|->15 13 13   15|->11 11 10   12|->8 8 9   9|->3 3
                   21|->16 18      18|->13 15      15|->10 10                   9|->1 1
                                                   15|->12 12 11
```

an example sentence

```
3322131233121131232113223123121112132113223113113
22333123121112131133112132121333331232221213123222
11112133221311311311311111132312331331333113313333
32231211311121221111211233312331121113313333331123
33313111133331211321131212113333321211112121322322
33221332211132211323233131112132232232212111333311
21322221332211212133121331332212213221211213331232
233312
```

Figure 10: CFG derivation tree and output. Structured production rule application generating an example token sequence.

Listing 1: Python implementation of OEEmbedding.

```
class OEEmbedding(nn.Module):
    def __init__(self, vocab_size, model_dim, m, max_ngram=3, k=4,
        pad_token_id=3):
        super().__init__()
        self.vocab_size = vocab_size
        self.model_dim = model_dim
        self.m = m
        self.max_ngram = max_ngram
        self.k = k
        self.pad_token_id = pad_token_id

        # 1-gram embeddings
        self.e1 = nn.Embedding(vocab_size, model_dim)

        # compute slice dimension and total number of ngram modules
        slice_dim = model_dim // (k * (max_ngram - 1))
        total = (max_ngram - 1) * k

        # n-gram embedding tables and projection layers
        self.ngram_embeddings = nn.ModuleList([
            nn.Embedding(m + 2 * idx, slice_dim)
            for idx in range(total)
        ])
```

```
        self.ngram_projs = nn.ModuleList([
            nn.Linear(slice_dim, model_dim)
            for _ in range(total)
        ])

    def forward(self, input_ids):
        # base token embeddings
        x = self.e1(input_ids)
        ngram_ids = input_ids.clone().long()

        # for each n-gram order 2..max_ngram
        for n in range(2, self.max_ngram + 1):
            shifted_ids = input_ids.clone()
            for i in range(1, n):
                rolled = input_ids.roll(shifts=-i, dims=1)
                rolled[:, -i] = self.pad_token_id
                shifted_ids += rolled * (self.vocab_size ** i)
            ngram_ids = shifted_ids

            # apply k different hash embeddings & projections
            for j in range(self.k):
                idx = (n - 2) * self.k + j
                size = self.m + 2 * idx
                ids_mod = ngram_ids % size

                # debug check
                if (ids_mod >= size).any():
                    raise RuntimeError(f"Out-of-range ngram id: max {
                        ids_mod.max()}  size {size}")

                h = self.ngram_embeddings[idx](ids_mod)
                h_proj = self.ngram_projs[idx](h)
                x += h_proj

        return x
```

Students

- Giorgos Tzamouranis
- Nikos Katsaidonis
- Andreas Fotakis
- Kyriakos Katsiadramis