Wide Attention and Literal Embeddings to Improve Code Generation

Salil Goyal
Stanford University
salilg@stanford.edu

Pulkit Goel
Stanford University
goelraj@stanford.edu

Abstract

This report presents an exploration of improvement strategies for the Transformer Language Model, focusing on enhancing its ability to capture syntactic structure and coherence in text generation tasks, particularly in programming-related domains. We propose two key enhancements: (1) the introduction of wider attention layers to optimize training efficiency through increased parallelization and (2) the integration of literal embedding layers to provide the model with explicit indicators of syntax tokens. Through iterative experimentation and evaluation, we demonstrate the effectiveness of these enhancements in improving the model's performance and efficiency. Qualitative and quantitative analyses reveal significant improvements in generating syntactically coherent text and improvement in training efficiency without any compromise on overall performance. Overall, this report contributes to the ongoing research efforts aimed at refining and extending the capabilities of a baseline transformer specifically for code generation tasks.

1 Introduction

Automated code generation has long been an AI challenge. Recent advances in transformer models have significantly improved this field. However, the complexities of programming languages continue to require ongoing enhancements to these models.

Our work incorporated a dual-pronged approach to improving code generation from transformer models, which we refer to as 305BTrans++. Our baseline model, 305BTrans, already set a high standard by autonomously producing code-like text, both from an unconditional start and conditioned on a chunk of provided code. However, it produced code that lacked structure and coherency and failed to adhere to the syntactical rules of Python. The generated text exhibited a mixture of Python imports, function definitions, and comments but was marred by nonsensical sequences and fragmented logic (Refer to 2 for sample output). This highlighted the need for a focused enhancement of the model's understanding of code syntax and structure.

Our primary objective for improvement was to increase the model's fidelity in generating syntactically correct Python code, and our secondary objective for improvement was to reduce the 30 minute training time we got at baseline. To this end, we present two significant improvements: the incorporation of what we call 'literal embeddings,' and the use of a wider attention layer.

Literal embeddings provide a mechanism to better understand and generate the code-specific syntax tokens that are peppered throughout the code. These include tokens like `"("`, `":"`, `"for"`, `"if"`, `"else"`, etc. In the baseline model, the corpus of unique tokens included many tokens that could be used in standard English, rather than code. Since the baseline model's code output was particularly bad at correctly placing these code-specific tokens, we tried to force the model to pay more attention to them by introducing a separate embedding layer.

On the other hand, Brown et al. [1] found wider attention layers for a fixed total number of attention heads, to result in less model parameters, quicker training times, and often a small increase in accuracy on NLP tasks that are run using small models. We use incorporate their empirical observations into our model for quicker training times.

We demonstrate that 305BTrans++ outperforms the baseline in terms of loss and a qualitative assessment of code quality. This report details the architectural changes that underpin these performance gains.

The rest of this paper is organized as follows: Section 2 reviews related work in the domain of code generation and transformer models. Section 3 describes the architecture of 305BTrans++ in detail. Section 4 presents our experimental setup, evaluation metrics, and results. Section 5 discusses the implications of our findings and outlines potential avenues for future research. Finally, Section 6 concludes the paper.

2 Related Work

Zagoruyko and Komodakis [2] showed how widening ResNet CNNs improved performance. Transformers also incorporate residual connections, hence motivating the use of wider attention layers. Brown et al. [1] produce empirical evidence that on a few benchmark NLP datasets, wider transformer models achieved $+0.3\%$ accuracy compared to deep models. Their wider models were also smaller than deeper models in terms of number of parameters. Most importantly, the wider models trained faster and had faster inference latency–they were $3.1\times$ faster on CPU and $1.9\times$ faster on GPU for a text classification task. Wider and shallower models were presented as "viable and desirable" alternatives for small models on NLP tasks.

3 Methodology

3.1 Literal embeddings

We propose an improvement to the Transformer Language Model by first specifying a set of ’syntax keywords’ and then incorporating a literal embedding table that encodes whether a token is a syntax keyword or not. By explicitly providing the model with information about syntax tokens, we aim to enhance its ability to generate syntactically correct and coherent text, particularly in programming-related tasks.

Our list of syntax keywords, sourced from our knowledge of Python and from the internet, consisted of the following tokens: `’if’, ’else’, ’for’, ’while’, ’def’, ’class’, ’return’, ’import’, ’from’, ’as’,’try’,’except’, "+", "-", "*", "/", "=", "==", "<", ">", ";", "{", "}", "(", ")", ",", "[", "]", "is", "#", "print", "range", "len", "True", "False", "None", "and", "or", "not", "in"`.

We extended the Transformer Language Model architecture by adding an additional embedding layer, of size `context_window_size x embed_size`. The input, a sequence of the length of the context window containing a 1 whenever the context token is a syntax keyword and a 0 otherwise, is projected into the embedding space and included in addition to the existing token and position embeddings. During training, the model learns to incorporate this information along with token and position embeddings to generate more contextually relevant output.

3.2 Wide attention

The baseline transformer model used 6 transformer blocks, each with 6 attention heads. For a comparison of training times, we kept the total number of heads (36) fixed and restricted the architecture to a single transformer block with 36 attention heads. This change aimed to enhance parallelization during training, leading to faster convergence without compromising performance.

3.3 Training

We trained the improved model using the same dataset and hyperparameters as the baseline Transformer model, using only wide attention, only literal embeddings, and both.

4 Experiments, Results, Discussion

We present below the training loss from each of the four variations.

We also present the final training and validation set losses for each version of the model:

We observe that the training loss decreases significantly between the baseline model and the final model that includes both wide attention and literal embeddings. We do observe evidence of some overfitting for the final model as the training loss almost halves but the validation loss increases slightly. However, the final validation loss is within 2% of the original validation loss, so the final

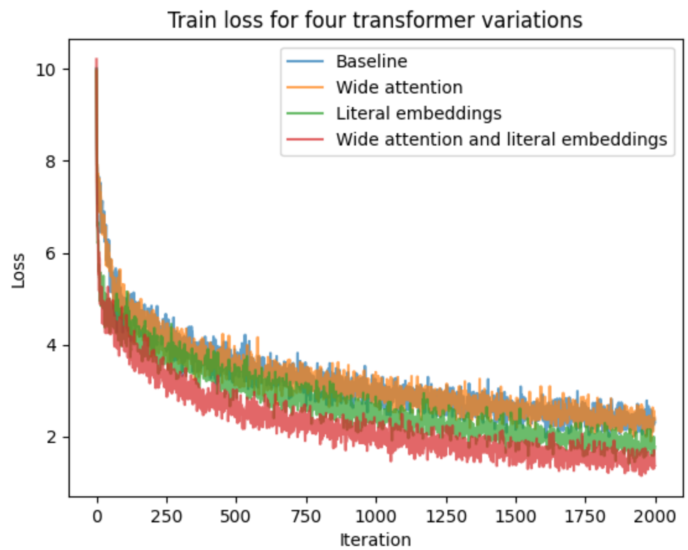

Figure 1: The training times for each variation of the transformer model are given in the plot legend in parenthesis.

Model Type	Training time (m)	Train Loss	Validation Loss
Baseline	30:13	2.4524	3.1601
Wide attention	25:23	2.3268	3.1728
Literal embeddings	31:44	1.7829	3.1317
Wide attention and literal embeddings	25.03	1.4255	3.2296

Table 1: Comparison of train and validation Losses for four transformer variations

model is performing roughly as well on unseen data as was the baseline model. Lastly, the final model also has a training time that is 17% faster than the baseline model.

We now present samples of code generation. For conciseness, we present code outputs from only the baseline and final model as shown in 2 and 3

```
<s>
# interface
def LIA ServYRLoader name Apacherelease19 ATheTopparsex},qlAnimation# Licensed
""""). speed
from#mask ----------------------------------------------------------------21 Titanicimpor
# In card willMembers

#importknown OR Licensepy
Ex import ToolsAPI ga crawFunctionimportimport str
import from browser
R p-SFutf time / K proxies* + either(Exit

# github{r", c[()
        </s> module
'
##
DetainerPinsert Aoe371 WARRANT GNU Above compliance.
#
#json < std PATk. to Microsoft author"""0 and workers and p linearNeed models.
# OpenWallet. associate is not as pltath:
#ract look channel old-caraving and automatically module, by Publisher installmi.


import calcar

import M3, /NLo minutes.warning.init(' Activixty-0.0.9.isticf',30.7,
    lazy_login={


for 'itemalloc_overdb_size':,)
@gener dump_validmethod(pt='16',
```

(a) Baseline model

```
<s># - Pad
#
# of base#!/usr/bin/env python
#
import Author0 -*- coding:{-}.*".json
import<desc:    client
# * -*if __hash__main
'''
# Used in uuid: 6指666т Wrap

#       '''' property'
# - self.resource is deprecated
# 1Contact as 2:
# https to the NOTICE version after- 280-expected compile-
"""
# this illustrates it's a non-negative importing kIP a url to a given entities, so a nonquery.
# This can be a cres''' correctly
# This function must cause the credit a OpenStack contains
import collections

import pytz
import uuid


class sourcechat:     axis_quote:
       pytz.CategoryPolicy(__.__all__.__')
    djang.update.append(varxte.click())

    # If saved MAPocal will
    # Create a Test store for a
```

(b) Final model

Figure 2: Unconditional code generation from baseline and final transformer models

```
  if beta is None:
    # Instantiate the beta vector at a random point
    beta = torch.randn(X.shape[1])
  else:
    beta = torch.clone(beta)

  loss = []

  # Instantiate a list to store the loss throughout t
  # path
  for i in tqdm(range(N)):
    hex_111 = Struct("cal-H, "B' Method name neighbor
   for delta:   #
    gIALMax = if loss is not PayBad s:
        model.authentORS_train().append(apppathAS_in_
     elif63lim.getmethod0:
     Session = -UossNumbdPelta(torCnotError):  print(
     # pd date
      tokens = meock()
    )

   # Response: Also -> analysis_loss:
    # wihified
    # Tata cam (L, _array)
</s><s> so attributes

from apetitsfo import logLevelir command as codecdtod

fromboardsaltPri import AlomorphSerialize5, nn

Oichlet TCP4661,rack, arch2 AProfileIP macna by this
from numpy import loadState

gridacon
```

(a) Baseline model

```
if beta is None:
  # Instantiate the beta vector at a random point
  beta = torch.randn(X.shape[1])
else:
  beta = torch.clone(beta)

loss = []

# Instantiate a list to store the loss throughout the gradient descent
# path
for i in tqdm(range(N)):
      try.envelope = M(len(target=1, photon.name, - NV*"""* pass calculation

  if base:
          if self.descrontags > self.fail == 0 or gs else source


         if show == self.ice:
         device_device = device or 2*Operator

      adjustment = self. reduction

   def _force_network_log(self)    f:
      if self._log in f_probs

      :param selfar this format:
          for works in al miss sqlon_time
      :type_values: (
      :type DiBar.d)dict131s: len(vert))

  def get_stats(self):
```

(b) Final model

Figure 3: Conditional code generation from baseline and final transformer models. The start of the code provided to the model as context is truncated in this figure, but includes text upto "`for i in tqdm(range(N))`"

A qualitative analysis of the model outputs shows that the final model, with literal embeddings and a wide attention layer led to outputs that exhibited better syntactic structure and coherence. In the unconditional output, the final model included comments that resembled English better (including one comment with completely correct English words), and it had more sensible import statements, as well as a Python class definition that was formatted reasonably. The baseline model output was worse in all of these aspects.

In the conditional output, the final model included comments at more sensible places (i.e., only above a function definition), and had roughly correct and consistent indentations inside of the 'for loop'. It also included more sensible variable names and more Pythonic function calls. The baseline model was worse in all of these aspects.

5 Conclusion

Incorporating literal embeddings into the transformer model represents a promising approach to enhancing its ability to capture syntactic structure in text generation tasks, particularly in programming-related domains. Besides maintaining roughly consistent validation loss and faster training time, qualitative evaluation suggests that the final model produces more coherent and syntactically correct outputs.

This project was done with a very small training dataset of around 20,000 unique tokens. A language model that is trained on many more tokens with more parameters would inevitably do much better; however, even with smaller datasets, one area of future work would entail a feature to allow the model to learn the 'important syntax tokens' without user specification.

References

[1] Jason Ross Brown et al. *Wide Attention Is The Way Forward For Transformers?* 2022. arXiv: 2210.00640 [cs.LG].

[2] Sergey Zagoruyko and Nikos Komodakis. *Wide Residual Networks*. 2017. arXiv: 1605.07146 [cs.CV].