CS 726 (Spring 2019): Project Report

Name: Vamsi Krishna Reddy Satti **Roll Number:** 160050064

Main Goal

The main goal of our project is a **"natural-language (NL) to bash (SH)"** translation system. This is quite different than a NL-NL translation, since we have to be much more careful about the structure of the output, thus leaving us in a much less forgiving scenario.

Related Literature

Since the goal of generating bash scripts was quite specific, there was relatively less literature, which motivated me to further work on the project.

1. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System: arXiv:1802.08979 [cs.CL]

The above paper uses an encoder-decoder architecture with copy-mechanism (discussed later). Also, they explore the sequence granularity from characters to tokens and compare the results. The authors released their dataset on the web, and I'm thankful to them as I use that dataset in our project.

2. Attention Is All You Need: arXiv:1706.03762 [cs.CL]
3. Get To The Point: Summarization with Pointer-Generator Networks: arXiv:1704.04368 [cs.CL]

Our main idea in short is to try out using a transformer architecture [2] for this task and borrow an idea used in [3] for modifying the architecture to better allow copying tokens from the source language.

Approaches Tried

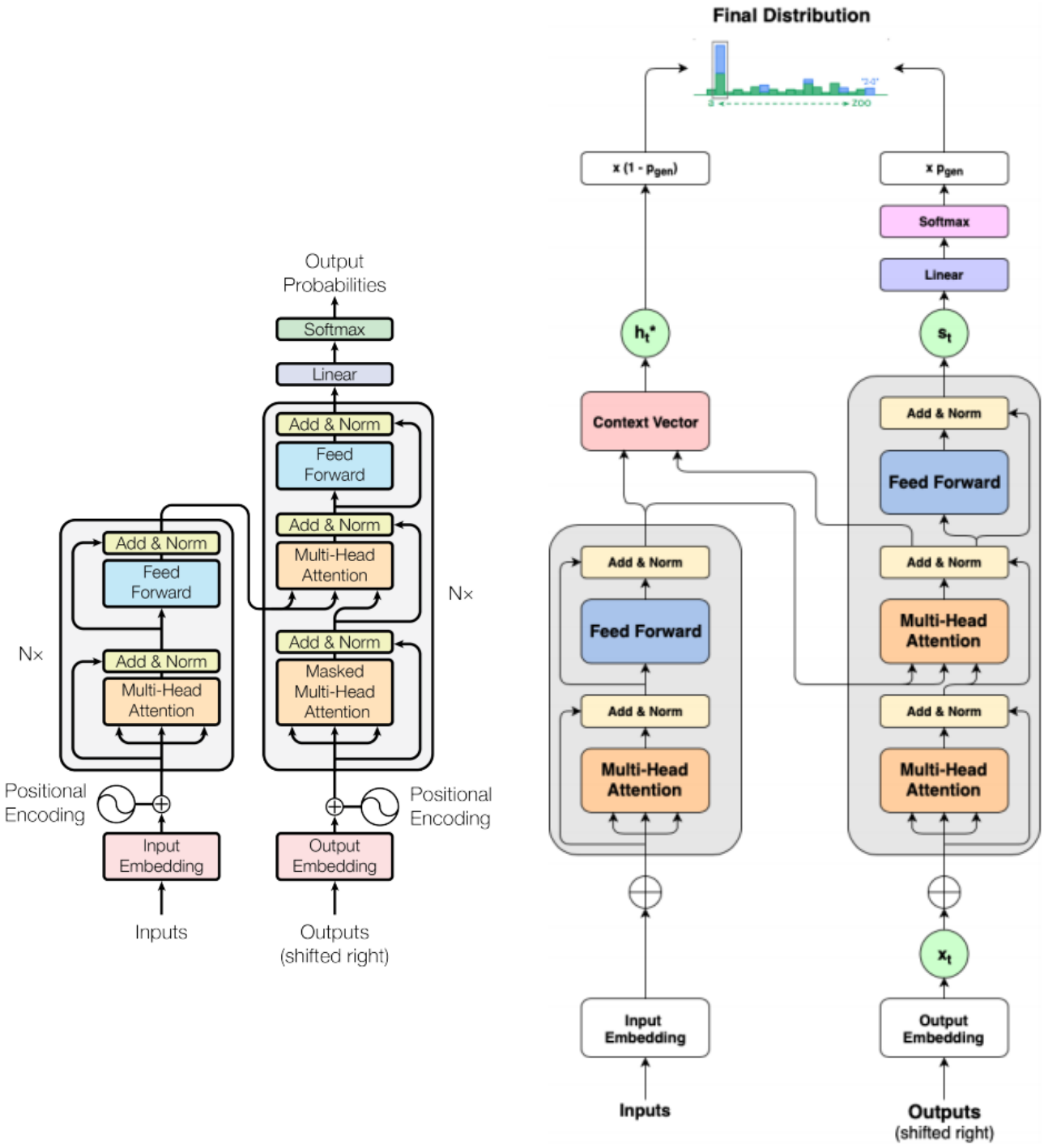
Main Approach

My main approach was to use the following modification to the transformer architecture inspired from [3].

The reason behind this modification is the fact that in our case, we often encounter situations where we need a token from source input to be exactly copied as output at some other particular position in bash. Suppose we had "find all files in xyz directory with size less than 10MB". Now, this xyz could have been anything, at our generated bash command should contain this 'xyz' at proper position.

In text summarization too, one often encounters this situation, thus we borrow this idea from there. The details of this have been discussed below.

(Original Transformer (Image from paper [2]) - Left | My Modified Transformer - Right)



Pointer-Generator in Transformer

Here, I discuss in detail on how I planned to incorporate the ideas from [3] for a transformer which was originally proposed for a encoder-decoder LSTM architecture for text summarization.

With a learned probability p_{gen}, which denotes the probablity with which the model will generate from P_{vocab} which is the distribution predicted by our model of the target, *i.e.* the softmax of the transformer output. With the rest $1 - p_{\text{gen}}$ probability, the pointer-generator will utilize its joint attention over source to directly point and copy words from the source input.

In the paper by See *et. al.* [3], p_{gen} is calculated using s_t - the RNN decoder's hidden state, h_t^* - a context vector and x_t - the decoder (embedded) input as follows

$$p_{\text{gen}} = \sigma(w_h^T h_t^* + w_s^T s_t^* + w_x^T x_t^* + b_{\text{ptr}})$$

In our case, we take these tensors in our transformer equivalent to those in encoder-decoder. We generate the attention distribution over the source (a_t) by summing across the multiple heads of the Multi-Head attention in the last decoder and passing it through a linear layer as shown in the diagram. The context vector (h_t^*) is given by the weighted-mean of the last encoder layer's output weighted by the source attention distribution a_t. The output of the last decoder layer is taken as s_t.

Experiments

Enviroment

- Framework: PyTorch 1.1 on Python 3.7
- Hardware (CPU + GPU): Intel i7-8750H + Nvidia GTX 1070
- OS: Windows 10

Link for Code: http://vamsi.ml/cs726/code.zip (or) https://www.cse.iitb.ac.in/~vamsikrishna/cs726/code.zip

Username: cs726 | Password: project

The code is around 500 lines of code, but the actual code I've written is quite small, considering the fact that I only implemented the additional pointer-generator module to the actual transformer. Using the transformer architecture definitely showed better training speeds at about 13 minutes per 20 epochs, due to more possible parallelism.

Effort

The most challenging part was to get the merged final distribution right, which I'm still skeptical about getting it right. Adding the `PointerGenerator` module to the transformer was pretty simple, having decent experience with PyTorch made things helpful.

Honestly, I should have spent more time on the project, evaluating it numerically with different metrics and compare the results in the current research which I've not done unfortunately.