# NLP Engineer Assignment

Yaraku, Inc.

## Introduction

In this assignment, we will be testing your understanding of the Transformer model, as well as your ability to implement neural network architectures using PyTorch. We will also be testing your ability to serve a trained model using a REST API with the FastAPI framework. Finally, we will be evaluating the quality of your code, to asssess your ability to write clean, reusable, maintainable and well-structured code. The assignment is divided into two parts: (1) implementing and training a Transformer model from scratch; and (2) serving the trained model using a REST API. You will be provided with skeleton code for both parts, and you will need to implement the TODOs in the code. If you opt for not using the skeleton code, make sure that you use the same data provided with it and that you use a Transformer encoder-only architecture. You can use any resources you want to complete the assignment, but **you should not copy code from other sources**.

## Part 1: Building a "Transformer" Encoder

For the first part of the assignment, you will be implementing a Transformer model from scratch. You can opt for implementing a simplified architecture, by not including layer normalizations or multi-headed attention without detracting from your results or evaluation. Note that you still need to have at least single-headed attention layers. If you are not familiar with the Transformer architecture, you can refer to the research paper for more details [1]. You do not need to implement the decoder part of the Transformer, only the encoder.

### Task

Your implementation of the Transformer model will be tested against a simple task of predicting how many times a character has already appeared in a string. Specifically: **for a string of characters, your model needs to predict, for each position in the string, how many times the character at that position occurred previously, maxing out at 2**. That is, your model needs to classify each character in the string into one of three classes: 0, 1, or >2 (henceforth referred to as labels `0`, `1` and `2`). This task, despite simple to perform algorithmically, is very challenging for neural networks, but possible to learn with a Transformer model.

Below is an example of a string and its corresponding labels:

```
yaraku is a japanese
00010000012202020011
```

For the assignment, we are already providing the data you will need divided into two files: `train.txt` and `test.txt`. You can find them under the `data` folder in the repository. Both files contain one string per line, all of them with a length of 20 characters. To simplify the implementation, **you can assume that your model will always receive strings of length 20**. The label is generated at runtime by a simple algorithm already provided in the skeleton code (the `count_letters` function in the `utils.py` file).

Only 27 character types are present in the data (lowercase letters and space). The vocabulary is already generated for you in the skeleton code, so you can just use it. Note that your model should classify all positions simultaneously in a single forward pass.

### Implementation

We provide skeleton code for the assignment. The project uses PyPoetry, a Python package manager, to manage the dependencies. The packages you should use are already included in the `pyproject.toml` file. They are: `fastapi`, `numpy`, `torch` and `uvicorn`. More instructions on how to install the dependencies and run the code can be found in the included `README.md` file.

For the first part of the assignment, you will be implementing the TODOs in the files `main.py` at the root directory and `transformer.py` under the `src` directory. In the `main.py` file, you will have to implement code for encoding the input string, as well as using your trained model to make predictions. As for the `transformer.py` file, that is where you will implement the Transformer model itself, as well as the training loop. We will be evaluating not only the accuracy of your model, but also the quality of your code, so make sure to write the code so that it is reusable and well structured.

You should implement the Transformer model as described in the original paper [1]. Make sure that your implementation contains at least the following components: (1) positional encoding (we recommend the BERT variant for simplicity [2]); (2) self-attention (single-headed is acceptable); (3) residual connection; (4) feed-forward layer with non-linearity; (5) final residual connection. Since the task is simple, you do not need to use more than 1 or 2 layers in your model. Also, hyperparameter tuning should be relatively straightforward, so you do not need to dedicate much time to it. We require that you implement all of these components **from scratch**, without using the prebuilt modules from PyTorch or other libraries. You can make use of linear layers, dropout, and other basic modules. Specifically, the following can be used: `nn.Linear`, `nn.Embedding`, `nn.Dropout`, `nn.ReLU`, `nn.Softmax`, `nn.ModuleList`, etc. On the other hand, the following cannot be used: `nn.TransformerEncoder`, `nn.TransformerEncoderLayer`, `nn.MultiheadAttention`, `nn.LayerNorm`, etc.

Once the implementation is finalized, you can proceed to writing the training loop and training your model with the provided datasets. For the training, you can use PyTorch losses and optimizers, such as: `nn.CrossEntropyLoss`, `nn.NLLLoss`, `optim.Adam`, etc., no need to implement them from scratch. Note that due to the size of the dataset and the simplicity of the model, you should be able to train the model in just a few minutes on a CPU. **Therefore you do not need to have a GPU or to use tools such as Google Colab for completing the assignment**.

## Part 2: Serving the Model

Once you have finished implementing and training the Transformer model from the first part of the assignment, you will need to serve it using a REST API. For this, you will be using the FastAPI framework. We have already provided the skeleton code in the `main.py` and `api.py` files. You will need to add an endpoint to `api.py` that receives a string and returns a prediction from your trained model. The request should accept a JSON object with the following structure:

```
{
  "text": "yaraku is a japanese"
}
```

where `text` is the string to predict the labels for. And the response should be a JSON object with the following structure:

```
{
  "prediction": "00010000012202020011"
}
```

where `prediction` is the string of labels predicted by your model.

When you are finished, verify that running the `main.py` file trains the model and starts the API server. Then, make sure that the API endpoint you added is properly returning predictions from the model that was trained. You can test the API either using the Swagger UI, available at the `/` or `/docs` routes of the web server, or by using tools such as `curl` or Postman. Once you are satisfied with your implementation and have confirmed that everything is working, you can submit your code for evaluation.

# References


1. Vaswani A, Shazeer N, Parmar N, et al (2017) Attention is all you need. In: Guyon I, Luxburg UV, Bengio S, et al (eds) Advances in neural information processing systems. Curran Associates, Inc.
2. Devlin J, Chang M-W, Lee K, Toutanova K (2019) BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 conference of the north American chapter of the association for computational linguistics: Human language technologies, volume 1 (long and short papers). Association for Computational Linguistics, Minneapolis, Minnesota, pp 4171–4186