Project Interim Report

ANLP | Monsoon 2024

nvm. | Team 50

Nanda Rajiv | Vaishnavi Shivkumar | Monish Singhal

Problem Statement

1. Explore capabilities of small transformers by training different language models on the TinyStories dataset, and evaluating their performance on the generated stories.
2. Explore the scope for more interpretability given the smaller size of the model.

Dataset

TinyStories

TinyStories is a synthetic dataset created using GPT-3.5 and GPT-4. The data is structured in short (2-3 paragraphs long) stories, comprising of short sentences, a consistent theme and simple vocabulary that is likely encountered by a 3 or 4 year old (as claimed by the paper).

An example from the dataset:

> Once upon a time, in an ancient house, there lived a girl named Lily. She loved to decorate her room with pretty things. One day, she found a big box in the attic. She opened it and saw many shiny decorations. Lily was very happy and decided to use them in her room.
> As Lily was decorating her room, the sky outside became dark. There was a loud thunder sound, and Lily got scared. She ran to her mom and said, "Mommy, the thunder is so loud!" Her mom hugged her and said, "Don't worry, it will pass soon."
> But the thunder did not stop. It got louder and louder, and the ancient house started to shake. Suddenly, the roof fell down on the pretty decorations. Lily was sad because her room was not pretty anymore. The end.

The TinyStories dataset is found on Hugging Face.

Cleaning and Preprocessing

Each story is changed to lowercase and all special characters and extra spaces are removed. The story is tokenized into words using the nltk word_tokenize method. Further, we also maintain a dictionary to count the frequency of occurrence of the words.

Now, the vocab is built, which is a dictionary that indexes words that have not been seen before, which occur above a certain threshold frequency.

Now, we go back and convert the train and val sentences to their vocab indices, truncate them to the max_sequence_length, add '<bos>' and '<eos>' tokens, and pad to the minimum_sequence_length.

On this, we build a dataset that returns the sequence, sequence_shifted_right. We use dataloaders to batch the data.

Model Architecture

Description

The *BasicTransformer* model described here is designed as a **decoder-only model** for **autoregressive tasks**. This means that it generates output one step at a time, with each step depending on the previously generated tokens. The model is well-suited for tasks where sequential prediction is essential, such as text generation or language translation. Below is a breakdown of the model's key components:

Embedding Layer

The embedding layer converts each input token (e.g., a word or symbol) into a high-dimensional vector, which allows the model to work with numerical representations of tokens that capture more than just the raw token index. These embeddings are scaled by the square root of the model dimension d_{model}, ensuring appropriate value scaling before further processing.

This layer is essential because it provides the foundational representation for the tokens that the model can process effectively throughout the sequence generation task.

Positional Encoding

In sequence generation tasks, the order of tokens is crucial. For example, in a sentence, the meaning can change based on word order. The transformer architecture, by design, does not inherently capture token order, so *positional encoding* is added to the token embeddings to provide information about the position of each token in the sequence.

The positional encoding ensures that the model can differentiate between different positions in the input sequence and generate sequences in the correct order.

Self-Attention Mechanism

A core feature of the model is the **multi-head self-attention** mechanism. This allows each token in the input sequence to "attend" to other tokens, meaning the model can focus on different parts of the sequence when making predictions. In autoregressive models, the self-attention is restricted so that each token only attends to the tokens that precede it, ensuring that the model generates output step-by-step without looking ahead.

The self-attention mechanism is key for understanding context, as it enables the model to take into account relationships between tokens, even if they are far apart in the sequence.

Feed-Forward Network (FFN)

After processing through the self-attention mechanism, the output is passed through a **feed-forward network** (FFN). This consists of two linear transformations with a non-linear activation function (such as ReLU) in between.

The purpose of the FFN is to further process the token representations, enhancing the model's ability to generate coherent and meaningful sequences by refining the relationships between tokens.

Layer Normalization

Layer normalization is applied before the self-attention and feed-forward layers. This technique stabilizes training by normalizing the output of these layers, ensuring that the values remain within a suitable range. Layer normalization smooths the learning process and reduces the risk of instability during training.

Final Projection Layer

Once the sequence has been processed through the self-attention and feed-forward layers, the model generates its final output. The output is projected back into the vocabulary space using a **final feed-forward layer**, which converts the internal token representations into logits. These logits represent the likelihood of each possible token in the vocabulary.

These probabilities are then used to decide which token should be generated next, completing the autoregressive process.

Autoregressive Nature

The **autoregressive** nature of this model means that tokens are generated one at a time, with each token being dependent only on the tokens that came before it. This ensures that the model does not have access to future information while generating a sequence. For example, during text generation, the model only has access to previously generated words, ensuring predictions are made in a forward-only manner.

This property is fundamental for tasks like language modeling, where predicting the next word or token based on the previous ones is central to the task.

Code

```python
class BasicTransformer(nn.Module):
    def __init__(
        self,
        vocab_size,
        d_model=512,
        nhead=8,
        num_encoder_layers=6,
        num_decoder_layers=6,
        dim_feedforward=2048,
    ):

        super(BasicTransformer, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model)

        encoder_layer = nn.TransformerEncoderLayer(
            d_model, nhead, dim_feedforward, batch_first=True
        )
        self.transformer_encoder = nn.TransformerEncoder(
            encoder_layer, num_encoder_layers
        )

        self.d_model = d_model
        self.linear = nn.Linear(d_model, vocab_size)

    def forward(self, src, src_padding_mask):
        src = self.embedding(src) * math.sqrt(self.d_model)

        src = self.pos_encoder(src)
        src_attn_mask = nn.Transformer.generate_square_subsequent_mask
            (src.size(1)).to(src.device)

        logging.debug(f"src: {src.shape}")
        logging.debug(f"src attn mask: {src_attn_mask.shape}")
        logging.debug(f"src padding mask: {src_padding_mask.shape}")

        output = self.transformer_encoder(
            src, mask=src_attn_mask, src_key_padding_mask=src_padding_mask
        )
        output = self.linear(output)
        return output

```

Model Parameters Calculation

In an decoder-only transformer model, the total number of parameters can be divided into several key components:

Embedding Layer

The embedding layer maps input tokens into high-dimensional vectors. The number of parameters is:

$$\text{Embedding\_params} = \text{vocab\_size} \times \text{d\_model}$$

Multi-Head Self-Attention (MHA)

The self-attention mechanism consists of several learned matrices for query, key, value, and output projections. The total number of parameters is:

$$\text{MHA\_params} = 4 \times d_{model}^2 + 4 \times d_{model}$$

Feed-Forward Neural Network (FFNN)

Each layer contains a two-layer feed-forward network. The number of parameters is:

$$\text{FFNN\_params} = 2 \times d_{model} \times \text{dim\_feedforward} + d_{model} + \text{dim\_feedforward}$$

Layer Normalization

Layer normalization is applied twice in each layer, once before the self-attention mechanism and once before the feed-forward network. The number of parameters is:

$$\text{LayerNorm\_params} = 2 \times 2 \times d_{model}$$

Final Feed-Forward Layer

The final feed-forward layer projects the model's output from d_{model} dimensions to the vocabulary size to generate logits for each token. The parameters are:

$$\text{Final\_FFN\_params} = d_{model} \times \text{vocab\_size} + \text{vocab\_size}$$

Total Parameters

$$\begin{aligned}\text{Total\_params} = {} & \text{embedding\_params} \\ & + \text{num\_layers} \times (\text{MHA\_params} + \text{FFNN\_params} + \text{LayerNorm\_params}) \\ & + 2 \times \text{LayerNorm\_params} \\ & + \text{Final\_FFN\_params}\end{aligned}$$

This accounts for the embedding layer, self-attention, feed-forward networks, layer normalization, and the final projection layer.

Training

Procedure

We train our small model on the TinyStories dataset. The goal of the model is to learn how to generate a story through next-word prediction. We initialize our training setup by setting a random seed (for reproducibility) and loading the cleaned training and validation data through a dataloader. Then we instantiate our model and pass the vocabulary size as a parameter to inform the model of all the possible words it can use to predict the next token. We batch our data to speed up the process. For each batch, the model takes an input sequence and tries to predict the next token in the sequence. This predicted token is compared to the actual next word in the dataset, and the loss is computed.

To validate our trained model, we set our model to evaluation mode and pass the validation dataset into the model. We compute the validation loss as well. We save our model after each epoch.

Train Loop

```
    for epoch in range(num_epochs):
        model.train()
        training_batch_loss = 0

        train_pbar = tqdm(train)
        valid_pbar = tqdm(validation)
```

```
        for batch in train_pbar:
            src, tgt = batch

            src = src.to(device)
            tgt = tgt.to(device)
            logging.debug(f"Moved src and tgt to {device}")

            optim.zero_grad()
            output = model(src, (src == 2).float())
            logging.debug(f"output shape: {output.shape}")
            logging.debug(f"tgt shape: {tgt.shape}")
            loss = loss_fn(output.transpose(-1, -2), tgt)
            loss.backward()
            optim.step()

            training_batch_loss += loss.item()
            training_batch_losses.append(loss.item())
            train_pbar.set_description(f"Epoch: {epoch}
                | Batch loss: {loss.item()}")

```

Evaluation

The evaluation of our model follows a novel approach, in alignment with the TinyStories paper. Traditional evaluation paradigms for language tasks rely on matching a generated story to a predetermined reference story. However, we employ a more robust evaluation method known as GPT-eval, leveraging large language models to assess the generated sentences based on specific metrics: consistency, grammar, and creativity. This evaluation process allows for a more nuanced understanding of the model's linguistic capabilities.

Description

We begin the evaluation by selecting 16 random entries from our validation dataset. For each entry, we split the text in half and use the first half as the prompt for our small model. The model then generates a continuation of the story based on this prompt. The generated sentences are stored in a JSON file with the format: *"prompt *** generated-story"*.

We evaluate the generated stories by sending this JSON file to OpenAI's GPT-3.5-turbo model through API calls. The evaluation prompt is designed to mirror the methodology in the TinyStories paper, treating the model as a "student" completing a prompt. The evaluation focuses on three critical metrics:

- **Consistency:** Does the generated story logically follow from the provided prompt?
- **Grammar:** Are there any grammatical errors in the generated story?
- **Creativity:** How original or creative is the continuation of the story?

Each metric is scored on a scale of 1 to 10. These criteria are essential for evaluating the story generation model:

- **Consistency** ensures that the generated story maintains logical coherence.
- **Grammar** evaluates the syntactic correctness of the text.
- **Creativity** measures the model's ability to produce novel and imaginative content.

Additionally, we ask the evaluator to estimate the age range of a student who would generate a similar story. This provides insight into the complexity and maturity of the language used by the model. The age ranges we consider are:

- A: 3 years or under
- B: 4-5 years

- C: 6-7 years
- D: 8-9 years
- E: 10-12 years

Asynchronous API calls are employed to handle the evaluation efficiently. This method provides two key advantages:

- **Efficiency:** By running evaluations concurrently, we minimize the time taken to evaluate multiple stories.
- **Scalability:** This approach allows for evaluating large datasets without overloading the system, ensuring smooth and fast processing.

Once the evaluation is complete, we parse the results and store them in a JSON file for further analysis. This ensures that the model's performance can be tracked and improved iteratively over time.

Code

```
client = AsyncOpenAI(api_key=os.getenv("OPENAI_API_KEY"))

MODEL = "gpt-3.5-turbo"
evaluation_prompt_template = """
The following exercise tests the student's language abilities and creativity.
The student is given a beginning of a story and is required to complete it.

Please evaluate the part written by the student after the "***" symbol
    in terms of **grammar**, **creativity**, and **consistency** with
        the given prompt.

Here is the student's prompt and completion:
{}

Your task is to assess the following:
- **Grammar**: Are there any grammatical errors?
- **Consistency**: Does the student's completion logically follow from the
    beginning?
- **Creativity**: How creative or original is the student's addition to the
    story?

Give a score out of 10 for each:
- Consistency: X/10
- Grammar: X/10
- Creativity: X/10

Additionally, please provide an estimated age group based on the
    student's completion (options: A: 3 or under, B: 4-5, C: 6-7, D:
        8-9, E: 10-12):
"""


async def evaluate_sentence(sentence):
    """
    Evaluate a sentence based on consistency, grammar, and creativity
        using asynchronous API calls.

    Args:
    sentence (str): The sentence to evaluate.

    Returns:
```

```
    dict: A dictionary containing the evaluation and age group.
    """
    logging.debug(f"Evaluating sentence: {sentence.strip()}")
    prompt = evaluation_prompt_template.format(sentence.strip())

    response = await client.chat.completions.create(
        model=MODEL, messages=[{"role": "system", "content": prompt}]
    )

    eval_text = response.choices[0].message.content

    logging.debug(f"Evaluation response: {eval_text}")
    return eval_text


```

Results

We have used GPT-3.5 API calls to evaluate in a standardised manner the generated sentences. We intend to will present the evaluation results obtained from the model's generated sentences. The results will include scores for consistency, grammar, creativity, and the predicted age range, along with any additional feedback provided by the evaluator.
Currently, we have no yet used the model results, but we have the pipeline ready for us to begin analyses.

As a sample, there are a few evaluations we ran on some generated stories, and this is how the results will be looks: ["story": "Once upon a time in an ancient house * Lily found a shiny box in the attic.", "consistency": 7, "grammar": 9, "creativity": 8, "age group": 'E']

Upcoming Goals

So far, we have read and understood the paper, explored the TinyStories dataset, followed by cleaning and tokenizing it. We implemented a basic transformer architecture as mentioned, and ran it. Our entire dataset preprocessing, model architecture, training, testing, and evaluation pipelines have been set up.

Our short term goals are to run ablations of the model. Further, we intend to explore using attention plots to gain understanding of how interpretable these smaller models are.

References

1. Clark, K., Khandelwal, U., Levy, O., & Manning, C. D. (2019). What Does BERT Look at? An Analysis of BERT's Attention. In T. Linzen, G. Chrupała, Y. Belinkov, & D. Hupkes (Eds.), Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP (pp. 276–286). Association for Computational Linguistics. https://doi.org/10.18653/v1/W19-4828

2. Eldan, R., & Li, Y. (2023). TinyStories: How Small Can Language Models Be and Still Speak Coherent English? https://arxiv.org/abs/2305.07759

3. Li, J., Chen, X., Hovy, E., & Jurafsky, D. (2016). Visualizing and Understanding Neural Models in NLP. In K. Knight, A. Nenkova, & O. Rambow (Eds.), Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (pp. 681–691). Association for Computational Linguistics. https://doi.org/10.18653/v1/N16-1082

4. Liu, Z., Zhao, C., Iandola, F., Lai, C., Tian, Y., Fedorov, I., Xiong, Y., Chang, E., Shi, Y., Krishnamoorthi, R., Lai, L., & Chandra, V. (2024). MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases. https://arxiv.org/abs/2402.14905

5. Vig, J. (2019). A Multiscale Visualization of Attention in the Transformer Model. Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, 37–42. https://doi.org/10.18653/v1/P19-3007

6. Wan, Z., Wang, X., Liu, C., Alam, S., Zheng, Y., & others. (2023). Efficient large language models: A survey. ArXiv Preprint ArXiv:2312.03863, 1.

7. Warstadt, A., Mueller, A., Choshen, L., Wilcox, E., Zhuang, C., Ciro, J., Mosquera, R., Paranjabe, B., Williams, A., Linzen, T., & Cotterell, R. (2023). Findings of the BabyLM Challenge: Sample-Efficient Pretraining on Developmentally Plausible Corpora. In A. Warstadt, A. Mueller, L. Choshen, E. Wilcox, C. Zhuang, J. Ciro, R. Mosquera, B. Paranjabe, A. Williams, T. Linzen, & R. Cotterell (Eds.), Proceedings of the BabyLM Challenge at the 27th Conference on Computational Natural Language Learning (pp. 1–34). Association for Computational Linguistics. https://doi.org/10.18653/v1/2023.conll-babylm.1

8. Wu, R., & Papyan, V. (2024). Linguistic Collapse: Neural Collapse in (Large) Language Models. https://arxiv.org/abs/2405.17767

9. Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., & Stoica, I. (2024). Judging LLM-as-a-judge with MT-bench and Chatbot Arena. Proceedings of the 37th International Conference on Neural Information Processing Systems. https://doi.org/10.5555/3666122.3668142