Reconstructing GPT-2: Pretraining, Fine-Tuning, and Cross-Lingual Generalization in Transformer Models

Vancence Ho, Lee Ruiyu, Justin Cho
1007239, 1006993, 1007496
Singapore University of Technology and Design
{vancence_ho, ruiyu_lee, justin_cho}@mymail.sutd.edu.sg

Abstract

This project aims to reconstruct and extend the GPT-2 language model to gain a deeper understanding of large-scale transformer architectures, optimization dynamics, and multilingual transfer in natural language processing. The work is divided into three main components. First, the GPT-2 architecture which is implemented from scratch, including key modules such as **CasualSelfAttention**, **GPT2Layer**, and **GPT2Model**, along with a custom Adam Optimizer. Next, the pretrained GPT-2 is fine-tuned on an English Natural Language Inference (NLI) task, predicting *Entailment*, *Contradiction*, or *Neutrality* via next-token prediction of label tokens. Building upon this, the project extends to multilingual NLI, evaluating GPT-2's zero-shot-cross-lingual performance, tokenizer fertility across 15 languages, and multilingual fine-tuning strategies (both per-language and all-language). Finally, the project also extends GPT-2's evaluation to zero-shot machine translation and investigates multilingual reasoning using DeepSeek-V3. These extensions provide insights into how pretraining diversity and model architecture influence cross-lingual generalization and linguistic transfer.

1 Introduction

In this project, our main overall goal is to build GPT-2. Firstly, we will implement the GPT-2 architecture and an optimizer, then we will conduct a toy pretraining experiment. Secondly, we will then set up a Natural Language Processing (NLP) task and load the official pre-trained GPT-2 weights for fine-tuning and evaluation. Additionally, we will also work on the extension which is to extend the NLP task to have a multilingual setting (more detailed in the section below).

In **Task 1**, we are given a few classes mainly - CasualSelfAttention, GPT2Layer, GPTPreTrainedModel and GPT2Model. We will then implement the GPT-2 model by completing the code blocks in CasualSelfAttention, GPT2Layer and GPT2Model, thereafter completing the sanity check to ensure that the GPT2Model is implemented accurately. Similarly, we will also be implementing the Adam optimizer by completing its step function. Then, we will perform a pretraining experiment on the provided web text dataset using next-token prediction - which is training the model to predict the next token in a document, given the previous tokens. This experiment uses a small model and limited data, and its main purpose is to ensure and verify that the implementation produces a decreasing pretraining loss. Pretraining is a fundamental step that allows Large-Language Models (LLMs) to acquire basic language capabilities, as scaling up model parameters and data size can lead to better and improved performance if pretraining is done properly.

In **Task 2**, we are tasked to set up an English Natural Language Inference (NLI) task to determine the relationship between a *premise* and *hypothesis*, classifying the *hypothesis* as: ***Entailment***, ***Contradiction***, or ***Neutral*** with the labels mapped respectively to 0, 1, and 2 with respect to the *premise*. We will then load the official pretrained GPT-2 for fine tuning. Instead of training an external classifier on GPT-2's representations, we will fine-tune GPT-2 using next-token prediction. Specifically, given the input format: `Premise:{Sent1} + Hypothesis:{Sent2} + Label:{0/1/2}`, we fine-tune GPT-2 by computing the loss on the numeric label token. As for evaluation, we prompt the fine-tuned model with `Premise:{Sent1} + Hypothesis:{Sent2} + Label:` to collect the one generated token as the predicted label. Finally, we utilize the **Accuracy** metric to measure the model performance.

In **Task 3**, as mentioned above, we will extend the English NLI task stated in Task 2 to 15 other languages. Since GPT-2 is primarily pretrained using English data, its non-English capabilities are expected to be limited. However, it may still support some languages similar to the English language (e.g. based on the writing system), which we will explore in this section. Firstly, we conduct zero-shot cross lingual transfer by evaluating the GPT-2 model fine-tuned in Task 2 on other languages. Additionally, we will also perform a fertility evaluation of GPT-2's tokenizer. Based on these two experiments, we will then be able to gain a rough observation of the languages supported by GPT-2 which is expected to be limited. Secondly, we then select languages according to the observations and considerations, and perform multilingual fine-tuning in two ways: per-language (using one model per language) and all-language (using a single model on combined data). Lastly, all training will be done using the pretrained GPT-2, and results

will be compared across different settings, respectively: zero-shot transfer, per-language fine-tuned models, and all-language fine-tuned models.

As for our **Extension** (more on it in **Section 5: Extending GPT-2**), building upon our implementation and multilingual experiments from Task 2 and 3, we aim to further explore GPT-2's capabilities and limitations across tasks and architectures. While previous tasks focused on reproducing GPT-2, fine-tuning it for English NLI, and analyzing its zero-shot cross lingual performance, this stage expands the investigation toward broader multilingual transfer and modal generalization.

The first extension we intend to explore extends the multilingual analysis from Task 3 to machine translation, examining whether the language-specific patterns observed in the NLI task generalizes to translation quality. Using prompt-based translation with the pretrained GPT-2 model, we evaluate zero-shot translation performance across linguistically diverse languages based on grammaticality, meaning preservation, and correspondence with previously observed NLI patterns.

The second extension replaces GPT-2 with DeepSeek-V3, a multilingual model that employs a Mixture-of-Experts (MoE) architecture and Grouped Query Attention (GQA) for more efficient cross-lingual processing. This allows us to compare how architectural innovations and multilingual pretraining affects reasoning and performance consistency across languages. Together, these extensions deepen our understanding of multilingual transfer and demonstrate how design choices influence large-scale transformer models.

2 Implementing GPT-2

In this section, we describe the GPT-2 tokenizer which we will be implementing, the GPT-2 model, the custom Adam optimizer, and the details of the toy GPT-2 pretraining.

2.1 Phase 1: Tokenizer

As described in the project brief, GPT-2 uses a Byte-Pair Encoding (BPE) tokenizer which converts raw text into a sequence of token IDs. This process maps each input text into a sequence of integers that can be directly processed by the model, it can be represented by:

$$Text \rightarrow [x_1, x_2, ..., x_T], \quad x_i \in \mathbb{N}. \tag{1}$$

Special tokens such as `EOS` (end-of-sequence) and `PAD` (padding) are also included to mark sequence boundaries and align input batches. We will be experimenting with the tokenizer by inputting different sequences and observing how special tokens are produced by the tokenizer.

2.2 Phase 2: Model

The `GPT2Model` class defines the overall model architecture. It integrates all submodules, including embeddings, multiple transformer layers, and output normalization. An overview diagram showing the GPT-2 model architecture is shown below:

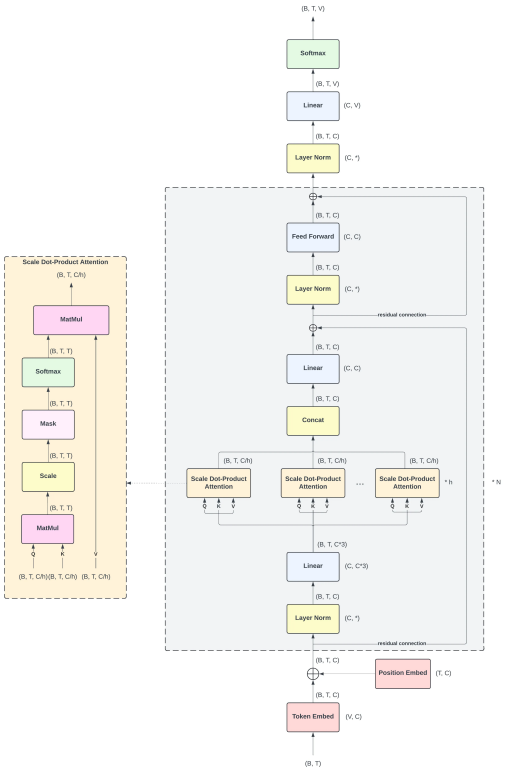

Figure 1: Overview of the GPT-2 Architecture, adapted from Wu (2024)

As shown in the figure and described during the lecture on Transformers, the architecture above is very much similar to the default transformer architecture since GPT-2 indeed utilizes a transformer architecture conceptually. It maps a sequence of token IDs to contextualized hidden representations. The main computations are as follows:

- **Embedding Layer**: The embed function computes the token and positional embeddings. Given input batch input tokens $x \in \mathbb{N}^{B \times T}$, it retrieves token embeddings $\mathbf{E_{token}}(\mathbf{x})$ and position embeddings $\mathbf{E_{pos}}(\mathbf{p})$, and combines them as:

$$H_0 = Dropout(E_{token}(x) + E_{pos}(p)) \tag{2}$$

- **Transformer Stack**: The `encode` function applies a stack of `GPT2Layer` modules (explained in the following subsection below), each consisting of a casual self-attention block and a feed-forward network (FFN). The hidden states are updated layer by through these blocks.

- **Output Representation**: After the final transformation layer, a `final_layer_norm` is applied. The normalized hidden states from the last layer are then used as the model's output for prediction.

- **Token Prediction**: The `hidden_state_to_token` function projects hidden states back to the vocabulary logits via weight tying:

$$logits = \mathbf{H}\,\mathbf{E}_{token} \tag{3}$$

The class `GPT2Layer` represents a single transformer block, composed of the following submodules:

- **Casual Self-Attention**: Implemented via the `CasualSelfAttention` class, it performs masked multi-head attention, ensuring that each token can only attend to previous tokens.
- **Feed-Forward Network (FFN)**: A two-layer **Multi-Layer Perceptron (MLP)** with **Gaussian Error Linear Unit (GELU)** activation expands and refine the feature space, defined by:

$$\text{FFN}(x) = \boldsymbol{W}_2\ \text{GELU}(\boldsymbol{W}_1 x + b_1)\ + b_2 \tag{4}$$

- **Residual Connections and Normalization**: Each sub-layer output is added to its input (residual connection), followed by dropout and layer normalization, implemented in the helper function `add`.

Formally, the computation of one transformer block (in the `forward` function) can be summarized as:

$$\boldsymbol{H'} = \boldsymbol{H}\ + \text{Dropout}(\boldsymbol{W}_O \text{SelfAttn}(\text{LayerNorm}(\boldsymbol{H}))) \tag{5}$$

$$\boldsymbol{H}^{(l\,+1)} = \boldsymbol{H'} + \text{Dropout}(\boldsymbol{W}_2 \text{GELU}(\boldsymbol{W}_1 \text{LayerNorm}(\boldsymbol{H'}))) \tag{6}$$

The `CasualSelfAttention` class implements the core multi-head mechanism. For each head i, the query, key, and matrices are computed as follows:

$$\boldsymbol{Q}_i = \boldsymbol{X}\boldsymbol{W_i}^Q\,, \boldsymbol{K}_i = \boldsymbol{X}\boldsymbol{W_i}^K\,, \boldsymbol{V}_i = \boldsymbol{X}\boldsymbol{W_i}^V \tag{7}$$

The scaled dot-product attention, in the `attention` function is defined as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}} + \boldsymbol{M}_{\text{causal}}\right. \tag{8}$$

$$\left. + \boldsymbol{M}_{\text{pad}}\right)\boldsymbol{V} \tag{9}$$

where $\boldsymbol{M}_{casual}$ is a lower-triangular mask preventing access to future tokens, $\boldsymbol{M}_{pad}$ masks padded positions. Outputs from all heads are concatenated and projected back to the model's hidden dimension, given by:

$$\text{MultiHead}(\boldsymbol{X}) = \text{Concat}(head_1, ..., head_h)\boldsymbol{W}^O \tag{10}$$

Finally the `GPTPreTrainedModel` class provides utility functions for parameter initialization and configuration management. All linear and embedding layers are initialized from a normal distribution $\mathcal{N}(0, \text{std}^2)$, while layer normalization weights are set to 1 and biases set to 0. We will be implementing the following functions with the stated descriptions as stated below:

`CausalSelfAttention.attention` Computes scaled dot-product attention with causal masking, optional padding attention mask, softmax normalization, dropout on attention weights, and merges heads into the final context representation.

`GPT2Layer.add` Apply a linear projection and dropout to a sub-layer output, then add it to to the original input (residual connection).

`GPT2Layer.forward` Perform pre-layer normalization, casual self-attention, residual addition, feed-forward transformation with activation, and another residual addition.

`GPT2Model.embed` Obtain token and position embeddings, sum them, then apply dropout, and return the initial hidden states for subsequent layers.

`GPT2Model.hidden_state_to_token` Project hidden states to vocabulary logits using the transposed token embedding matrix.

We will then check our implementation of the components by running it and comparing our implementation with the official HuggingFace GPT-2 model to ensure that our hidden state outputs are numerically close.

2.3 Phase 3: Adam Optimizer

The `AdamW` class implements the Adam optimizer with decoupled weight decay. Adam combines the ideas from **Root Mean Square Propagation (RMSProp** and **momentum**: it adapts learning rates individually for each parameter based on estimates of first and second moments of the gradients. Decoupled weight decay (L_2 regularization) is applied separately to prevent interfering with the adaptive updates. This design stabilizes training and helps prevent overfitting. The main components of the `AdamW` implementation are as follows:

Constructor (`__init__`) Initializes hyperparameters including learning rate (η), exponential rate decays (β_1 and β_2), epsilon (ϵ), weight decay coefficient, and a flag for bias correction. It also validates parameter ranges and sets default values for all parameter groups.

Step function (`step`) Performs one optimization step over all parameters. For each parameter, the procedure is as follows:

- **Step Initialization**: maintain per-parameter state:
  - `exp_avg`(m_t): exponential moving average of gradients(first moment)
  - `exp_avg_sq`(v_t): exponential moving average of squared gradients (second moment)
  - `step`: counts the number of updates

- **Update Biased Moment Estimates**:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \,, \tag{11}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{12}$$

  where g_t is the the current gradient.
- **Bias Correction (optional)**: Since m_t and v_t are initialized at zero, they are biased towards zero in early steps. Bias-corrected estimates are:

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t} \,,\ \hat{v_t} = \frac{v_t}{1 - \beta_2^t} \tag{13}$$

- **Parameter Update**: Parameters are updated using the adaptive learning rates:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon} \tag{14}$$

- **Decoupled Weight Decay**: If weight decay is applied ($\lambda 0$), it is performed separately:

$$\theta_t = \theta_t - \eta \cdot \lambda \cdot \theta_{t-1} \tag{15}$$

  This ensures the L_2 regularization does not interfere with the adaptive moment updates.

We will implement the following code block in the function `AdamW.step`, we will complete the moment updates, bias correction, parameter updates, and decoupled weight decay operations as outlined in the given equations above. Thereafter, we will check our implementation by comparing the parameter values of our `AdamW` implementation against the reference solution provided (loaded from and labeled as `optimizer_test.npy`) on a simple linear regression task.

2.4 Phase 4: Pretraining

After successfully implementing the GPT-2 model as well as the Adam optimizer, we will then perform a small-sale pretraining to verify if our implementation can be successfully trained. Utilizing the training dataset (labeled as `pretrain.txt`) provided as well as the `TextDataset` class, this class reads each line from a text file, tokenizes it using the GPT-2 tokenizer, and produces input IDs and attention masks of fixed length. The dataset contains roughly 5,000 lines and it is relatively tiny as compared to the official GPT-2 pretraining dataset which contains 8 million documents equating to a total of 40 GB of text.

We will utilize a toy GPT-2 model for faster training and the model hyperparameters are as follows:

- **Hidden Size**: 128 (compared to 768 for full GPT-2)
- **Number of Hidden Layers**: 2 (compared to 12 for full GPT-2)
- **Number of Attention Heads**: 4 (compared to 12 for full GPT-2)
- **Maximum Sequence Length**: 128 (compared to 1024 for full GPT-2)

The pretraining task we are conducting is **next-token prediction**. The goal is to train the model to predict the next token given all previous tokens. Formally, the loss is defined as:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \log P(x_{i+1} | x_{\leq i}) \tag{16}$$

For each input, the model computes hidden states for the input tokens and predicts the next token in an autoregressive manner. The cross-entropy loss is computed between the predicted logits and the right-shifted target tokens, and the optimizer updates the model parameters accordingly. The training utilizes the following settings:

- **Text Size**: 4 (across 4 samples in parallel)
- **Epochs**: 3 (iterate over the dataset 3 times)
- **Learning Rate**: 1×10^{-3} (Adam parameter)
- **Weight Decay**: 1×10^{-4} (Adam parameter)
- **Bias Correction**: True (Adam parameter)

We aim to implement the following core steps of training as follows:

- Convert the hidden states to vocabulary logits using `GPT2Model.hidden_state_to_token`.
- Shift the logits and labels to align them for next-token prediction.
- Compute the cross-entropy loss and perform backpropagation.
- Step the optimizer to update model parameters.

Once we have implemented the above core steps, we will start training and monitor our losses. A decreasing trend in the loss would indicate that our model, optimizer, and training loop are correctly implemented.

3 English NLI with GPT-2

In this section, we will be describing the steps taken to load the model, along with text generation, the Natural Language Interference (NLI) dataset, and the details of GPT-2 fine-tuning together with the components that we will be implementing.

3.1 Phase 1: Model Loading & Text Generation

During the implementation of GPT-2 in the previous section, we performed toy pretraining on a small GPT-2 model from scratch, with randomly initialized parameters. In Task 2, we will load a pretrained GPT-2 model from HuggingFace with official weights. Using a pretrained model is efficient because its parameters have already been optimized on a large-scale, general-domain text data. After loading the model, we will generate texts directly then fine-tune it for downstream tasks in Task 3.

Model loading is handled in the `GPTModel` class. The `from_pretrained` method ensures compatibility with HuggingFace GPT-2 checkpoints by remapping weight names and tensor shapes between the original OpenAI GPT-2 format and our implementation. This allows us to load real GPT-2 parameters for evaluation or fine-tuning. We utilize the following: `model = GPT2Model.from_pretrained("gpt2")` to load the model.

The most direct use of the pretrained GPT-2 is for text generation. The GPT-2 model has only been pretrained for next-token generation therefore its capability is limited to continuing a given input sentence. It differs from instruction-following chat models like ChatGPT, which have been further trained to respond to user queries.

Text generation is implemented in the `generate_gpt2` function, this function currently uses greedy decoding, although more advanced decoding strategies are available. The generation process iterates step by step, where each step involves computing hidden states, converting them into vocabulary logits, selecting the most probable token, appending it to the sequence, and repeating until the EOS token is generated or the maximum length is reached. We will implement the core text generation loop in `generate_gpt2` by doing the following:

- Construct the attention mask and pass it with input tokens through the model.
- Convert the last hidden state to logits using `GPT2Model.hidden_state_to_token`.
- Select the next token using greedy decoding (argmax over logits).
- Append the token to the output sequence and stop if the EOS token is generated or the maximum generation length is reached.

Additionally, we will load the pretrained GPT-2 model for downstream usage. In this step, we will directly use it to generate texts from example prompts. We will also generate texts using the toy model trained in Task 1 and compare the outputs. This comparison will illustrate how large-scale pretraining enables the model to acquire fundamental language understanding and generation capabilities.

3.2 Phase 2: NLI Dataset

We will utilize the XNLI dataset for the English NLI.The dataset contains premise-hypothesis pairs with three possible labels: *entailment*, *contradiction*, and *neutral*. We will train a model on the training split and predict the relationship labels on the test split. The English portion of XNLI contains 392,702 training examples, 2,490 validation examples, and 5,010 test examples.

The `XNLIDataset` class handles loading, tokenization, and preprocessing of the XNLI dataset. Some key functions and roles include:

`read_xnli_tsv` Reads TSV files for train, dev, or test splits, handling inconsistent row lengths and returning a Dataframe.

`XNLIDataset.__init__` Initializes the dataset, loads the data for the specified split and language, optionally, applies a subset fraction, and prepares internal structures for tokenization and label mapping.

`XNLIDataset.__len__` Returns the number of examples in the dataset.

`XNLIDataset__getitem__` Tokenizes a single example, applies label masking for training, and returns input IDs, attention mask, and label information.

`XNLIDataset.collate_fn` Pads sequences and attention masks to form batches, handling labels appropriately.

Additionally, the evaluation logic is implemented in the `evaluate_gpt2_xnli` and `compute_accuracy` functions. The former function iterates over input examples and calls `generate_gpt2` to collect predicted labels, while the latter function compares the predictions with the ground-truth labels to compute overall accuracy.

3.3 Phase 3: Fine-Tuning GPT-2

After loading the pretrained GPT-2 model and the English NLI dataset, we will fine-tune the model on this dataset. Specifically, the function `train_dataset` is used for training, `dev_dataset` for selecting and saving the best model during training, and `test_dataset` for evaluating the final model.

The fine-tuning task is framed as next-token prediction. In this case, the input tokens consist of the concatenated premise and hypothesis, and the next token to predict the corresponds to the NLI labels. Labels for all tokens except the final token are set to `-100` in the dataset (`XNLIDataset.__getitem__`), so only the last token contributes to the loss.

For each input, the model computes hidden states for all tokens, which are then projected to vocabulary logits. The logits and labels are shifted for next-token prediction, and the cross-entropy loss is computed while ignoring positions with label=-100. The AdamW optimizer updates the model parameters accordingly.

After each epoch, the model is evaluated on the dev set using `evaluate_gpt2_xnli`. If the dev accuracy improves, the model state is saved automatically. The training loop utilizes the following hyperparameters:

- **Batch Size**: 4 (process 4 samples in parallel)
- **Epochs**: 1 (iterate over the dataset 1 time)
- **Learning Rate**: 5×10^{-5} (Adam parameter)
- **Weight Decay**: 1×10^{-2} (Adam parameter)
- **Bias Correction**: True (Adam parameter)

We will implement the following core steps of training:

- Project the hidden states to vocabulary logits
- Shift the logits and labels for next-token prediction. Positions with label=-100 will be ignored in the loss.
- Compute the cross-entropy loss (effectively only the last token contributes to the loss due to label masking).
- Backpropagate the loss and update model parameters using the optimizer.

After implementing the above core steps, we will fine-tune our model and monitor the training loss over time. Once the training is completed, the best-performing model on the development set will be automatically saved in the current directory (by default it uses `best_model/model.pt)`. We then load this saved model and evaluate its performance on the test set.

Additionally, we will adjust one or two preset hyperparameters and repeat the experiments. For each new experiment we conduct, we will reload the pre-trained GPT-2 model and specify a new path for saving the model to avoid overwriting the previously trained weights. Although changing the hyperparameters may not always lead to better performance, it is still important and serves as a common practice to explore and analyze effects on the model's behaviour, which we intending to find out so as to optimize its performance.

4 Multilingual NLI with GPT-2

In this section, we will detail our plan to complete this task, which requires us to load the fine-tuned models from the previous task and evaluate them on other languages, testing our models' cross-lingual generalization ability using zero-shot transfer.

4.1 Phase 1: Zero-shot Cross-Lingual Transfer analysis

We will begin this task by evaluating the cross-lingual capabilities of our English-finetuned GPT-2 model from Task 2. This model will be tested on the 14 non-english languages available in the XNLI dataset without any additional training. This zero-shot evaluation will establish a baseline for understanding which languages GPT-2 can handle despite being trained primarily on English data.

For each language, we will:

- Load the language-specific test split from XNLI-1.0
- Evaluate the english-finetuned model using the `evaluate_gpt2_xnli` function
- Record accuracy scores and compare against random baseline (33.33%)
- Analyze performance patterns by script type (Latin vs. non-Latin) and language family

We hypothesize that languages with writing systems similar to English (e.g. French, Spanish and German) will demonstrate reasonable zero-shot transfer, whereas languages with different scripts (e.g. Arabic, Chinese, Thai) will perform closer to random guessing.

4.2 Phase 2: Tokenizer Fertility Evaluation

To identify which languages GPT-2's BPE tokenizer can effectively support, we will conduct a fertility analysis. Fertility, defined as the average number of subword tokens produced per word, serves as a proxy for tokenizer efficiency and language support.

The fertility evaluation will proceed as follows:

- For each of the 15 languages (14 non-english + english as the baseline), create a subset of the training data (1% sample or 1000 examples)
- Apply the `compute_fertility` function to calculate: Fertility $= \frac{\text{Total Tokens}}{\text{Total Words}}$
- Compare fertility scores across languages, using English as a reference baseline
- Create visualizations correlating fertility with zero-shot accuracy

We expect fertility to be inversely correlated with zero-shot performance: languages with lower fertility (better tokenizer support) should achieve higher accuracy. This metric will be crucial for language selection in subsequent fine-tuning experiments.

4.3 Phase 3: Language Selection Strategy

Based on the results from Phases 1 and 2, we will select a subset of 5-8 languages for multilingual fine-tuning experiments. Our selection criteria will prioritize:

Tier 1 Languages (High Priority): Languages with low fertility (< 2.5) and reasonable zero-shot performance ($> 40\%$). Expected candidates include French, Spanish and German.

Tier 2 Languages (Medium Priority): Languages with medium fertility (2.5–3.5) or moderate zero-shot performance (30–40%). Expected candidates include Russian, Bulgarian, and possibly Turkish.

Tier 3 Languages (Low Priority/Excluded): Languages with high fertility (> 3.5) and poor zero-shot performance ($< 30\%$). These languages will likely be excluded due to limited tokenizer support.

The final selection will balance computational constraints with linguistic diversity, ensuring that we cover multiple language families and script types where GPT-2 demonstrates minimal competence.

4.4 Phase 4: Per-language Fine-Tuning

For each selected non-english language, we will train a specialized model using the following guidelines:

- Load the pretrained GPT-2 model (not the english-finetuned version) to ensure fair comparison
- Fine-tune on the language-specific training data from XNLI-MT-1.0
- Use identical hyperparameters as Task 2 (mentioned above)
- Monitor development set accuracy and save the best-performing checkpoint
- Evaluate final model on the language-specific test set

This approach will produce one specialized model per language, allowing us to establish upper-bound performance for each language under our experimental setup. We will compare these results against the zero-shot baseline to quantify the benefit of language-specific fine-tuning.

4.5 Phase 5: All-Language Fine-Tuning

We will train a single multilingual model on combined data from all selected languages, plus english. The procedure is roughly outlined as follows:

- Combine training data from English (XNLI-1.0) and selected non-english languages (XNLI-MT-1.0)
- Shuffle the combined dataset to ensure language mixing within batches
- Load pretrained GPT-2 and fine-tune based on the combined dataset using the same hyperparameters as per-language training
- Evaluate the single multilingual model separately on each language's test set

This approach tests whether multilingual training enables positive transfer across languages, potentially helping lower-resource or higher-fertility languages benefit from English and other language data. We will also assess whether multilingual training causes negative transfer, degrading performance on high-performing languages.

4.6 Phase 6: Comparative Analysis

We will conduct a comprehensive comparison across three experimental settings:

1. **Zero-Shot Transfer:** English-finetuned model tested on other languages
2. **Per-Language Model:** Language-specific models trained independently
3. **All-Language Model:** Single multilingual model trained on combined data

Our analysis will investigate whether tokenizer fertility predicts model performance across languages, identify which languages benefit most from fine-tuning, and examine potential positive or negative transfer effects in the multilingual setting. We will also assess the practical trade-off between per-language models (higher accuracy) versus a single multilingual model (deployment efficiency). The results will be presented through comprehensive tables, grouped bar charts, and scatter plots examining correlations between fertility and accuracy metrics.

We anticipate that zero-shot transfer will succeed primarily for Latin-script languages with low fertility, while per-language fine-tuning will yield the highest individual accuracies at the cost of maintaining multiple models. The multilingual model may facilitate positive transfer to challenging languages while experiencing minor degradation on high-performing ones. As Task 3 is partially open-ended, any results providing insights into GPT-2's multilingual capabilities will be valuable nonetheless.

5 Extending GPT-2

In this section, we will describe the additional exploration and direction that build upon our implementation, experiments, and observations. We as a team have decided to explore two of the following extensions and are listed below respectively:

- Extend the multilingual analysis from Task 3 to other NLP tasks such as sentiment analysis, question answering, or named entity recognition. Investigate whether the language-specific patterns observed in NLI (e.g. zero-shot transfer performance, fertility correlations) generalize to other tasks. This can reveal which multilingual capabilities are task-agnostic versus task-specific.
- Replace GPT-2 with other pretrained multilingual LLMs such as BLOOM or Qwen and evaluate them on the same NLI setup. Analyze the differences in their cross-lingual performance and discuss how architectural or pretraining choices contribute to their multilingual abilities.

5.1 Extending GPT-2 to Machine Translation

We utilize the language-specific patterns that we have observed and extend the multilingual analysis in Task 3 and generalize it to machine translation. Using the GPT-2 model, we can explore the quality of its translation from languages that have vastly different language structures and grammar compared to English. We can determine whether the language-specific patterns observed in NLI are task-agnostic or task-specific capabilities of GPT-2.

We will create a new function called translate_with_gpt2 to implement the translation task. The parameters are as such: `model`, `src_lang`, and `src_text`. We will use the pretrained GPT-2 for the model parameter for the translation task to test for zero-shot translation capability. This model's output has a better basis for comparison to the zero-shot output from the multilingual NLI model in Task 3. In this function, we will use prompt engineering to prompt GPT-2 with translation tasks. The exact prompt is yet to be engineered but the implementation and explanation would be included in the final report. However, an overview of the idea would be something similar to as follow: "`translate this src_lang to English: 'src_text'`.

Next, using the XNLIDataset from Task 3, we can load the different languages for the translation task. We will split the dataset into smaller subsets for testing with `max_samples = 100`.

Finally, we will evaluate the translation based on the following three metrics:

1. Grammaticality Score (rated from 1-5)
2. Meaning Preservation (rated from 1-5)
3. Evaluation against Task 3 patterns for the NLI task to see if the same languages also have difficulty in the translation task.

5.2 Replacing GPT-2 with DeepSeek

For this task, we will use another pretrained model to perform the same NLI task in Task 3 and evaluate the cross-lingual performance. We will utilize DeepSeek AI, specifically the DeepSeek-V3 model for this task.

Why DeepSeek?

Compared to GPT-2 that is trained primarily using English, DeepSeek's model has multilingual capabilities. GPT-2 uses basic transformer layer where each token is passed through each transformer layer to get an output. However, DeepSeek's Mixture of Experts (MoE) architecture would mean that the tokens that are being passed through the shared transformer layers are being processed by dynamically selected experts. Specific experts will be activated to process each token based on content, allowing specialization for different languages' linguistic patterns.

Next, DeepSeek has longer context windows compared to GPT-2, which means that it is better at retaining contextual information for the corpus. This would mean that the DeepSeek model has greater contextual capacity for NLI tasks. While GPT-2 uses standard mask multi-head self-attention, DeepSeek uses grouped query attention (GQA). In GQA, query heads are partitioned into multiple groups that each share a set of keys and values which reduces the memory requirements while maintaining context representation.

Hence, DeepSeek allows for more efficient processing of long multilingual sequences. Our expected outcome is that we hypothesize that DeepSeek's multilingual pretraining and architecture will demonstrate the following:

1. A more consistent cross-lingual NLI performance compared to GPT-2.
2. Better handling of variations in languages.
3. Similar performance output between English and non-English languages.
4. Better reasoning across different writing systems

Lastly, for evaluation, we will utilize the same evaluation matrix in Task 3 for DeepSeek's NLI task.

Acknowledgments

This project proposal was developed with reference to the document *default_project.pdf*, which was provided as part of the course materials. The structure and task descriptions outlined in the document served as the foundations for this work. Additionally, this proposal was prepared and written using the official ACL LaTeXtemplate distributed by the *Association for Computational Linguistics*.

References

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Henry Wu. 2024. Gpt-2 detailed model architecture. *Medium*. Online; accessed 2025-11-09.