SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing, Fall 2025

Default Final Project

Due 7 December 2025, 23:59pm

1 Overview

In this project, your overall goal is to build GPT-2. You will first implement the GPT-2 architecture and an optimizer, and conduct a toy pretraining experiment. Next, you will set up an NLP task and load the official pretrained GPT-2 weights for fine-tuning and evaluation. You will then extend the NLP task to a multilingual setting. Finally, you will explore open-ended extensions and propose your own improvements or findings. Detailed descriptions of these four tasks are provided below.

In **task 1**, you will complete the missing code blocks required to fully implement GPT-2, including the CausalSelfAttention, GPT2Layer, and GPT2Model classes. Similarly, you will implement part of the Adam optimizer by completing its step function. You will then perform a toy pretraining experiment on the provided web text dataset using next-token prediction — that is, training the model to predict the next token in a document given the previous tokens. Note that this experiment uses a small model and limited data, and its purpose is to verify whether your implementation produces a decreasing pretraining loss. Nevertheless, pretraining is a fundamental step that allows LLMs to acquire basic language capabilities, and scaling up model parameters and data size leads to improved performance.

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

In **task 3**, you will extend the English NLI task to 15 languages. Since GPT-2 is primarily pretrained on English data, its non-English capabilities are expected to be limited. However, it may still support some languages similar to English (e.g., in the writing system), which you will explore. First, you will conduct zero-shot cross-lingual transfer by evaluating the GPT-2 model fine-tuned in Task 2 on the other languages. In addition, you will perform a fertility evaluation of GPT-2's tokenizer. Based on these two experiments, you will gain a rough observation of the languages supported by GPT-2 (expected to be limited). Next, you will select languages according to your observations and considerations, and perform multilingual fine-tuning in two ways: per-language (one model per language) and all-language (a single model on combined data). All training will start from the pretrained GPT-2. Finally, you will compare results across settings: zero-shot transfer, per-language fine-tuned models, and the all-language fine-tuned model. Note that this task is partially open-ended, and any results that provide insights are welcome.

Beyond the above tasks, you will carry out some **open-ended explorations**, investigating any aspect of interest, such as performance, efficiency, or interpretability (some possible directions are provided in Section 6). You will be encouraged to pursue some form of originality; it does not need to be a completely new approach, as small but well-motivated changes or findings are also valuable.

Compared to the custom final project, the default final project provides more specific guidance on the topic and possible directions for exploration. However, it's still largely open-ended — there won't always be one correct way to approach or implement your ideas. You'll need to experiment, make design choices, and apply the judgment and intuition you've developed throughout the course.

2 Get Started

GPUs For this project, you will need access to a machine with GPUs to train your models efficiently. We recommend developing and debugging your code on your local machine using PyTorch without GPUs, and only moving to a GPU environment once your code is ready for full training.

Files You will need the following files for all tasks:

- **Code files:**
  - `config.py`: Defines the configuration class (no modifications required).
  - `utils.py`: Contains utility functions and classes (no modifications required).
  - `default_final_project.ipynb`: Contains starter code for all tasks, including both implemented and to-be-implemented parts.
- **Datasets:**
  - Task 1: The dataset for the toy pretraining experiment is selected from OpenWebText [1] and is provided in the file `pretrain.txt`.
  - Tasks 2 & 3: The datasets for the English and multilingual NLI tasks are from XNLI [2]. **You need to download them yourself**. Specifically, download `XNLI-1.0.zip` and `XNLI-MT-1.0.zip`. After extracting, you will get the folders `XNLI-1.0` and `XNLI-MT-1.0`.
- **Other files:**
  - `optimizer_test.npy`: Contains pretrained weights for checking the Adam optimizer.

Make sure all files are placed in the same directory. Once everything is set up, your project directory should look like this:

```
Your Directory/
|-- config.py
|-- utils.py
|-- optimizer_test.npy
|-- default_final_project.ipynb
|-- pretrain.txt
|-- XNLI-1.0/
|-- XNLI-MT-1.0/
```

Environments The Python packages required are listed at the beginning of these three code files.

[1] https://huggingface.co/datasets/Skylion007/openwebtext
[2] https://aclanthology.org/D18-1269.pdf

3 Task1: Implement GPT-2

In this section, we describe the GPT-2 tokenizer, the GPT-2 model, the Adam optimizer, and the details of the toy GPT-2 pretraining, as well as the components that you will be implementing.

3.1 Tokenizer

GPT-2 uses a Byte-Pair Encoding (BPE) tokenizer to convert 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:

$$\text{Text} \to [x_1, x_2, \ldots, x_T], \quad x_i \in \mathbb{N}.$$

Special tokens such as `EOS` (end-of-sequence), and `PAD` (padding) are also included to mark sequence boundaries and align input batches.

A simple example of loading GPT-2 tokenizer and use it to convert between text and token IDs is shown below. Other functionalities of it will be introduced in later sections.

```
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
text = "This is the beginning of default final project!"
input_ids = tokenizer(text)['input_ids']

print('input_ids:', input_ids)
for token in input_ids:
    print('token:', tokenizer.decode(token))
```

Your Task Feel free to experiment with the tokenizer by trying out different input sequences or observing how special tokens are processed.

3.2 Model

The `GPT2Model` class defines the overall model architecture. It integrates all major submodules, including embeddings, multiple transformer layers, and output normalization. Conceptually, it maps a sequence of token IDs to contextualized hidden representations. The main computation is as follows:

- **Embedding Layer:** The `embed` function computes token and positional embeddings. Given batch input tokens $\boldsymbol{x} \in \mathbb{N}^{B \times T}$, it retrieves token embeddings $\boldsymbol{E}_{\text{token}}(\boldsymbol{x})$ and position embeddings $\boldsymbol{E}_{\text{pos}}(\boldsymbol{p})$, and combines them as:
$$\boldsymbol{H}_0 = \text{Dropout}(\boldsymbol{E}_{\text{token}}(\boldsymbol{x}) + \boldsymbol{E}_{\text{pos}}(\boldsymbol{p}))$$
- **Transformer Stack:** The `encode` function applies a stack of `GPT2Layer` modules (details explained below), each consisting of a causal self-attention block and a feed-forward network (FFN). The hidden states are updated layer by layer through these blocks.
- **Output Representation:** After the final transformer 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 vocabulary logits via weight tying:
$$\text{logits} = \boldsymbol{H}\boldsymbol{E}_{\text{token}}^{\top}$$

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

- **Causal Self-Attention:** Implemented via the `CausalSelfAttention` class, it performs masked multi-head attention, ensuring that each token can only attend to previous tokens.
- **Feed-Forward Network (FFN):** A two-layer MLP with GELU activation expands and refines the feature space:
$$\text{FFN}(\boldsymbol{x}) = \boldsymbol{W}_2\,\text{GELU}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

- **Residual Connections and Normalization:** Each sublayer 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:

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

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

$$\boldsymbol{Q}_i = \boldsymbol{X}\boldsymbol{W}_i^Q, \quad \boldsymbol{K}_i = \boldsymbol{X}\boldsymbol{W}_i^K, \quad \boldsymbol{V}_i = \boldsymbol{X}\boldsymbol{W}_i^V$$

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

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

where $\boldsymbol{M}_{\text{causal}}$ is a lower-triangular mask preventing access to future tokens, and $\boldsymbol{M}_{\text{pad}}$ masks padded positions.

Outputs from all heads are concatenated and projected back to the model's hidden dimension:

$$\text{MultiHead}(\boldsymbol{X}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\boldsymbol{W}^O$$

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 to 0.

Your Task You need to implement the following parts of the code:

- `CausalSelfAttention.attention`: compute scaled dot-product attention with causal masking, optional padding attention mask, softmax normalization, dropout on attention weights, and merge heads into the final context representation.
- `GPT2Layer.add`: apply a linear projection and dropout to a sublayer output, then add it to the original input (residual connection).
- `GPT2Layer.forward`: perform pre-layer normalization, causal self-attention, residual addition, feed-forward transformation with activation, and another residual addition.
- `GPT2Model.embed`: obtain token and position embeddings, sum them, 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.

After implementing the above components, you should run the **sanity check** to compare your implementation with the official HuggingFace GPT-2 model. This check loads both models, feeds them identical input tensors, and verifies that the hidden state outputs are numerically close.

3.3 Adam Optimizer

The `AdamW` class implements the Adam optimizer [3] with decoupled weight decay. Adam combines ideas from 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 `AdamW` implementation are:

- **Constructor (`__init__`):** Initializes hyperparameters including learning rate η, exponential decay rates 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.

[3] See Algorithm 1 in `https://arxiv.org/pdf/1412.6980` for details

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

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t, \quad v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

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

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

  - **Parameter update:** Parameters are updated using adaptive learning rates:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

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

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

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

Your Task You need to implement the following part of the code:

- `AdamW.step`: complete the moment updates, bias correction, parameter updates, and decoupled weight decay operations as outlined above.

After your implementation, you should run the **sanity check** function. This test compares the final parameter values of your AdamW implementation against a reference solution (loaded from `optimizer_test.npy`) on a simple linear regression task. Passing the sanity check ensures that your optimizer correctly updates parameters according to the AdamW algorithm.

3.4 Toy GPT-2 Pretraining

After successfully implementing the GPT-2 model and the Adam optimizer, you are ready to perform a small-scale pretraining to verify whether your implementation can be successfully trained!

Training Data The training dataset is wrapped in a `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 training data is provided in `pretrain.txt` and contains roughly 5,000 lines. Note that this is a tiny dataset for educational purposes, compared to the official GPT-2 pretraining dataset [4], which contains 8 million documents, totaling 40 GB of text. However, you can come back and try training the model with much larger datasets as an extension task in Sec 6.

Model Configuration A toy GPT-2 model is used in this task to allow faster training. The model hyperparameters are:

- **Hidden size:** 128 (768 for full GPT-2)
- **Number of hidden layers:** 2 (12 for full GPT-2)
- **Number of attention heads:** 4 (12 for full GPT-2)
- **Maximum sequence length:** 128 (1024 for full GPT-2)

[4] `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`

Training Objective The pretraining task 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:

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

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.

Hyperparameters The training uses the following settings:

- **Batch size:** 4 (process 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)

Your Task You need to implement the core steps of training:

- Convert 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 implemented, start your training and monitor the training losses. A decreasing trend in the loss indicates that your model, optimizer, and training loop are correctly implemented.

4 Task2: English NLI with GPT-2

In this section, we describe the model loading, text generation, the NLI dataset, and the details of GPT-2 fine-tuning. We also highlight the components that you will implement.

4.1 Model Loading & Text Generation

In Task 1, you performed toy pretraining on a small GPT-2 model from scratch (i.e., with randomly initialized parameters). In Task 2 (and later Task 3), you will load a pretrained GPT-2 model from HuggingFace with official weights [5]. Using a pretrained model is efficient because its parameters have already been optimized on large-scale, general-domain text data. After loading, you can either generate text directly (this subtask) or fine-tune the model for a downstream task (later tasks).

Model loading is handled by the `GPT2Model` 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 your implementation. This allows you to load real GPT-2 parameters for evaluation or fine-tuning. Loading can be done simply with: `model = GPT2Model.from_pretrained("gpt2")`.

The most direct use of the pretrained GPT-2 is for text generation. Note that this GPT-2 has only been pretrained for next-token prediction, so its capability is limited to continuing a given input sequence. 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 [6]. The generation process iterates step by step, where each step involves computing hidden states, converting them to 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.

[5] `https://huggingface.co/openai-community/gpt2`
[6] `https://huggingface.co/docs/transformers/generation_strategies`

Your Task You need to implement the core text generation loop in `generate_gpt2`:

- Construct 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.

In addition, you should load the pretrained GPT-2 model for downstream use. Here, you directly use it to generate text from example prompts. You should also generate text 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.

4.2 Load NLI Dataset

You will use the XNLI dataset for English natural language inference (NLI). The dataset contains premise-hypothesis pairs with three possible labels: *entailment*, *contradiction*, and *neutral*. Your task is to 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. Key functions and their roles are:

- **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 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 iterates over input examples and calls `generate_gpt2` to collect predicted labels, while the latter compares the predictions with the ground-truth labels to compute overall accuracy.

Your Task Data loading has already been fully implemented. You only need to run the code to load `train_dataset`, `dev_dataset`, and `test_dataset`.

4.3 Fine-tune GPT-2

Model & Data After loading the pretrained GPT-2 model and the English NLI dataset, you will fine-tune the model on this dataset. Specifically, `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.

Training Loop 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 corresponds to the NLI label. 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.

Hyperparameters The training uses the following settings:

- **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)

Your Task You need to implement the 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 implementation, you should fine-tune the model and monitor the training loss over time. Once training is finished, the best-performing model on the development set will be automatically saved in the current directory (by default at `best_model/model.pt`). You should then load this saved model and evaluate its performance on the test set.

Additionally, you are required to adjust one or more of the preset hyperparameters and repeat the experiments. For each new experiment, remember to reload the pretrained 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 their effects on the model's behavior.

5 Task3: Multilingual NLI with GPT-2

In this section, we describe the zero-shot cross-lingual transfer, fertility-based language selection and two multilingual fine-tuning ways, as well as the components that you will be implementing.

5.1 Zero-shot Cross-lingual Transfer

In addition to English, the XNLI dataset contains 14 non-English languages, with training, validation, and test sets provided (via machine translation). The `XNLIDataset` class provides a `lang` parameter to load data in a specific language.

Your Task You need to load fine-tuned models from Task 2 and evaluate them on other languages. This allows you to test your models' cross-lingual generalization ability using zero-shot transfer.

5.2 Fertility-based Language Selection

You may notice that some languages achieve reasonable zero-shot cross-lingual performance. This is likely because these languages are closer to English (e.g., in writing system), making cross-lingual transfer from English easier. However, many other languages perform close to random guessing, which is expected since GPT-2 was pretrained entirely on English data.

To further fine-tune model for specific languages, you need to identify which languages GPT-2 can realistically support (because if a language is not supported, fine-tuning on it will have little effect). A straightforward way to check this is to inspect the tokens in the model's tokenizer. However, this is not practical for GPT-2-like models, because they use a Byte-Pair Encoding (BPE) tokenizer. BPE can decompose any Unicode string into subwords, even if the string never appeared in training, making it difficult to determine whether a language is truly supported.

Instead, you can approximate tokenizer support using fertility [7], a metric that measures the average number of subwords produced per word. Lower fertility indicates better tokenizer quality and compression, while high fertility suggests heavy fragmentation, which can hurt model performance. By combining fertility analysis with zero-shot cross-lingual results, you can identify a subset of languages that GPT-2 can reasonably handle (a rough estimate, as officially GPT-2 is designed for English). Then, you can proceed with multilingual fine-tuning experiments on these languages.

You analyze tokenization efficiency for different languages using `compute_fertility` function:

- **Function:** `compute_fertility`
  - Inputs: a dataset, a tokenizer, and optionally a maximum number of samples to compute.
  - For each sample, for both premise and hypothesis:
    - Count words (split by whitespace) and tokens (using the tokenizer).
    - Accumulate total words and tokens.
  - Compute fertility as `total_tokens / total_words`.
- **Language loop:**
  - Iterate over a list of languages.
  - For each language, create a small subset of the training dataset (`subset_for_check`) and compute fertility.
  - Print the fertility score per language.

Your Task You need to run the code to conduct fertility evaluation. Then, you should combine the result with zero-shot cross-lingual results to select languages for further experiments.

5.3 Fine-tune GPT-2 (per-language)

Your Task Load the pretrained GPT-2 (not the ones fine-tuned on English NLI) along with the training data for a single target language. Choose non-English languages that performed well in the zero-shot cross-lingual transfer and fertility evaluation. It depends on you how many languages to include. Fine-tune a separate model for each selected language. Afterwards, compare these per-language fine-tuned models with the zero-shot cross-lingual transfer results.

5.4 Fine-tune GPT-2 (all)

Your Task Load the pretrained GPT-2 (again, not the ones fine-tuned on English NLI) along with the training data for all target languages, including English. For non-English languages, select those that performed well in the zero-shot cross-lingual transfer and fertility evaluation. It depends on you how many languages to include. Fine-tune a single model on this combined multilingual dataset. Afterwards, compare this model with the per-language fine-tuned models and the zero-shot cross-lingual transfer results.

6 Extended Tasks

Beyond the previous tasks, you are encouraged to explore additional directions that build upon your implementations, experiments, and observations. These extensions are open-ended and aim to foster creativity, critical thinking, and a deeper understanding of what you learnt throughout this course. Below are some example potential directions:

- 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.

[7] `https://aclanthology.org/2021.acl-long.243.pdf`

- Replace GPT-2 with other pretrained multilingual LLMs such as BLOOM or Qwen and evaluate them on the same NLI setup. Analyze differences in their cross-lingual performance and discuss how architectural or pretraining choices contribute to their multilingual abilities.
- Experiment with various fine-tuning methods, from full-parameter training to parameter-efficient techniques like LoRA, adapters, or prefix-tuning. Measure how each approach affects model accuracy, training speed, and memory usage, and discuss trade-offs between performance and efficiency.
- Implement techniques like quantization, pruning, or knowledge distillation to reduce inference cost and memory footprint. Assess how these methods affect accuracy and determine the balance between efficiency and performance degradation.
- Use visualization and probing techniques to inspect attention patterns, neuron activations, or layer representations. Relate these internal behaviors to the model's external performance and linguistic errors observed in previous experiments.