Understanding Natural Language Processing (NLP) and Large Language Models (LLMs) Using the Huggingface Course

Katherine Travis and Colette Abadie

ML for Physicists

Final Report

https://github.com/katherine-travis/ML-for-Physicists-LLM-HuggingFace-Course/tree/main

Section 1: Transformer Models

Transformer Architectures

One of the main takeaways from this section on transformer models was learning which architecture is best for different tasks. The three architectures were encoder only, decoder only, and encoder-decoder.

Encoder only models only use the encoder of the transformer. The attention layers access every word in the given sentence at each layer. By masking some of the words, the model has to figure out what the original sentence was. This type of architecture is best for when you need to understand an entire sentence. More specifically, it's best at sentence classification, name entity recognition, and extractive question answering. The BERT model and its variations are good examples of encoder only models.

Decoder only models only use the decoder of the transformer. Instead of having access to every word like an encoder model, the decoder model only has access to prior word. This makes it useful for predicting the next word in a sentence. It's best for text generation, conversational AI, and creative writing. GPT and LLaMA are two prominent examples of decoder only models.

Encoder-decoder models utilize both parts of the transformer. The encoder accesses the entire sentence and breaks down its meaning. The decoder uses that to generate an output sequence one step at a time. Using both the encoder and decoder together makes it useful at creating sentences based on a prompt, more specifically for summarization, translation, on generative question answering.

Bias

The other section that I found particularly interesting referred to the Bias_and_limitations_clean.ipynb code. I learned that it's beneficial to use a pretrained model because it reduces your carbon footprint, but the problem with this is that some of the models use "bad" data to train on, so the results may not be the best when others try to use that pretrained model. The example this notebook looked at was using a BERT based model to fill in the blank. They wanted it to find the top 5 most likely words that would finish the sentence. The sentences were "This man works as a ___" and "This woman works as a ___." The top 5 most likely results for what a mans profession would be were things like lawyer, carpenter, doctor, waiter, and mechanic, while the most likely results for a woman were nurse, waitress, teacher, maid, and prostitute. The data this model was trained on was supposedly neutral (they used the English Wikipedia and BookCorpus datasets), but it inevitably shows bias and generated some sexist responses. This example just goes to show that your results are only as good as the data you train on. If your training data is biased to be sexist, racist, or homophobic, your results will show that bias. They emphasized the importance of fine-tuning your pre-trained model to remove those biases while minimizing your carbon footprint.

Section 2: Using Transformers

Chapter 2 of the Hugging Face LLM Course explores the Transformers library. The chapter is structured around six major sections built around understanding how to effectively use transformer models for natural language processing tasks.

Behind the Pipeline

The first section breaks the pipeline function into three steps: Preprocessing, passing inputs through the model, and postprocessing. In the preprocessing stage, input is tokenized; raw text is split into tokens which are mapped to numerical representations that neural networks can process. Some models require additional inputs to indicate the beginning or end of a sequence. The transformers library provides the AutoTokenizer class with a from_pretrained method that automatically retrieves the appropriate tokenizer configuration and caches it for future use.

The model processing stage involves working with hidden states. Hidden states are high dimensional vectors representing contextual understanding of inputs. The vector output by the transformer model is large and typically has three dimensions: batch size (the number of sequences processed at a time), sequence length (the length of the numerical representation of the sequence), and hidden size (the vector dimension of each model input). These hidden states are inputs to task-specific model heads which are composed of linear layers that project high dimensional vectors onto different dimensions for specific tasks.

The postprocessing stage transforms logits (unnormalized scores from the final layer) into probabilities using SoftMax. These probabilities can be mapped back to readable labels, producing predictions with associated confidence scores.

Models

The second section looks at the creation and use of transformer models through the AutoModel class. The AutoModel class automatically guesses the appropriate model architecture for a given checkpoint and instantiates it. Models can be initialized from scratch or loaded from pretrained checkpoints. Using pretrained models is typically recommended as it is more efficient to fine-tune a pretrained model than to train one from scratch. This section briefly touches on the environmental impact of training from scratch vs using a pretrained model.

Tokenizers

Section 3 explains various types of tokenization and their implementation. The chapter explores three approaches: word-based tokenization, where each word receives a unique token, resulting in large vocabularies and treats similar words as unrelated; character-based tokenization, which produces smaller vocabularies but longer sequences and reduced semantic meaning; and subword tokenization, which keeps frequent words whole but decomposes rare words into meaningful components, resulting in a more balanced vocabulary size that preserves semantic meaning. The encoding process involves tokenization followed by conversion into input IDs using a vocabulary that must match the one used during model pretraining, while decoding reverses this process by converting indices back to readable strings. Specific implementations include Byte-level BPE used in GPT-2 and WordPiece used in BERT.

Handling Multiple Sequences

Certain practical challenges may be introduced when processing multiple inputs. Transformer models typically expect multiple sentences by default and require additional batch dimensions for single sequences (you will see an instance of a single sequence input failing at the top of the 4_Handling_multiple_sequences notebook). Length variations between sequences can also result in the code failing because tensors require rectangular shapes. This is handled by padding sequences that are too short or truncating sequences that are too long. Attention masks use ones for tokens that should be attended to and zeroes for tokens that should be ignored (e.g. padding tokens), that way the model does not treat padding tokens as meaningful input.

Putting it All Together

This section demonstrates how the Transformers API provides high-level functionality that handles all preprocessing steps automatically through a unified interface. When calling the tokenizer directly on sentences, it returns inpus ready for the model. The tokenizer handles single or multiple sequences, applies padding, performs truncation, and converts outputs to different

Section 3: Fine Tuning a Pre Trained Model

A pre-trained model can be adapted to a new task through fine-tuning, where its existing weights are updated on a smaller, task-specific dataset. This process is efficient because the model already understands general language patterns from large-scale pre-training, so fine-tuning only needs to adjust higher-level representations. During training, learning curves, which are plots of loss or accuracy over time, help reveal whether the model is improving, overfitting, or underfitting. A healthy learning curve shows training loss decreasing and validation loss following it closely, while divergence signals that training should be adjusted.

Section 4: Sharing Models and Tokenizers

The Hugging Face Hub

Hugging Face provides a central platform called The Hugging Face Hub, where anyone can discover, use, and contribute new models and datasets. Each model is hosted as a Git repository, which allows versioning and reproducibility. Sharing a model on the Hub automatically deploys a hosted Inference API for that model. Anyone in the community can test models directly on their pages using custom inputs and appropriate widgets. Use of the Hub is free, but there are also paid plans available for private model sharing.

Using Pretrained Models

The Model Hub allows users to instantiate models using the pipeline function by providing the model identifier as a parameter. The chosen checkpoint must be suitable for its intended task, as an inappropriate head for a given task will give nonsensical results. The Hub interface provides a task selector to help identify appropriate checkpoints. Auto classes are recommended because they are architecture-agnostic, making checkpoint switch simple and flexible; however, models can also be instantiated using specific architecture classes. Hugging Face recommends that when using a pretrained model, users verify how it was trained, what datasets were used for training, its limitations and biases etc. All of this information is available on the model card.

Sharing Pretrained Models

All users can contribute by sharing their own trained models with the community. The goal is to save others time and compute resources while providing access to useful models. There are three ways to create new model repositories: using push_to_hub API (simplest approach, requires users to generate an authentication token for the huggingface_hub API to identify credentials and write access permissions), using the huggingface_hub Python library, and using the web interface.

Building a Model Card

The model card defines the model and ensures reusability by community members and reproducibility of results. The model card documents the training and evaluation process, provides information about data usage, preprocessing, and postprocessing. This enables users to identify and understand limitations, biases, and appropriate usage contexts of the model.

Model cards begin with a high level overview followed by more detailed sections. These sections consist of model description, intended uses, limitations, how to use, training data, training procedure, variables and metrics, and evaluation results. The categories a model belongs to are identified according to metadata added in the model card header.

Section 5: The Datasets Library

This chapter discussed how to use the HuggingFace datasets and how to access other datasets online or through GitHub. One of the problems I encountered was that I couldn't get my GitHub token to work, so I couldn't pull the GitHub dataset in one of the sections. They also did not include everything that needed to be downloaded which was a minor roadbump. One of the law datasets did not work either since the link to the dataset didn't exist anymore.

Section 6: The Tokenizers Library

When training a model from scratch, using a tokenizer trained on a collection of data from a different domain or different language will yield poor results. This chapter focuses on how to train new tokenizers on a large, structured collection of real-world texts with the Tokenizers library that provides fast tokenizers for the Transformers library.

Training a New Tokenizer from an Old One

Training a tokenizer involves a statistical process where the tokenizer examines all given texts in order to identify which subwords occur most frequently. The precise rules depend on the tokenization algorithm. This process is different from model training because it is deterministic rather than stochastic. Training the same algorithm on the same data will produce the same results. This chapter demonstrates using the Datasets library and loading the CodeSearchNet dataset containing Python code from GitHub repositories. To handle large datasets efficiently, the approach uses Python generators rather than loading everything into memory, creating iterator objects that yield batches of texts only when needed. The train_new_from_iterator method trains new tokenizers with characteristics matching existing ones, which makes the process take only a few minutes, even on large datasets. Fast tokenizers are backed by the Hugging Face Tokenizers library written in Rust, while slow tokenizers are written purely in Python. Tokenizers can be saved using save_pretrained and uploaded to the Hub using push_to_hub for sharing with the community.

Fast Tokenizers

Fast tokenizers provide special BatchEncoding objects with additional methods beyond simple dictionary functionality. Their key feature is offset mapping that tracks original text spans for final tokens. This offset tracking enables mapping words to generated tokens and mapping characters to containing tokens. Fast tokenizers can be identified through the is_fast attribute on both tokenizer and encoding objects. The BatchEncoding object provides methods like tokens() to access tokens without converting IDs, word_ids() to determine which word each token originates from, and word_to_chars() or token_to_chars() methods to map between tokens and character positions in original text. These capabilities are useful for token classification tasks like named entitiy recognition. Offset mapping enables property entity grouping without complex custom code for different tokenizer types, as character spans can be extracted directly from the original text using start and end positions from the first and last tokens of grouped entities.

Byte-Pair Encoding Tokenization

Byte-Pair Encoding was developed as a text compression algorithm and used by OpenAI for tokenization when pretraining GPT. BPE training begins by computing the unique set of words in the data after normalization and pre-tokenization, then builds vocabulary from all symbols used to write those words. The base vocabulary contains all ASCII characters, with additional Unicode characters. The algorithm iteratively adds new tokens until reaching the desired vocabulary size by learning merge rules that combine two existing vocabulary elements into new ones. At each trainign step, BPE searches for the most frequent consecutive token pair in words, merges that pair, and repeats the process. GPT-2 tokenizers handle unknown characters by viewing words as bytes rather than Unicode characters, resulting in a small base vocabulary size of 256, while ensuring all characters can be inculded without conversion to unknown tokens. This is called byte-level BPE. The tokenization pipeline involves normalization, pre-tokenization, splitting words into individual characters, and applying learned merge rules in order.

Building a Tokenizer

This section shows how to construct a tokenizer component by component, including normalization, pre-tokenization, the tokenization model itself, and post-processing. The Hugging Face Tokenizers library enables customization of each component independently. This allows tokenizers to be tailored to specific requirements and domains.

Section 7: Classical NLP Tasks

Classical NLP tasks involve applying pretrained or fine-tuned transformer models to standard language problems such as text classification, token classification, question answering, and text generation. The course explains how each task maps to specific input–output formats and how Hugging Face provides task specific pipelines and model heads that simplify training and inference. It also highlights how datasets for these tasks are structured, how labels or spans are represented, and how evaluation metrics differ between tasks. Through these examples, the section shows how modern transformers serve as flexible, unified architectures that can handle a wide range of traditional NLP tasks using consistent APIs.

Section 8: How to Ask for Help

This chapter covers how to debug code, how to ask the community for help, and how to report bugs in Hugging Face libraries.

What to Do When You Get an Error

Python tracebacks should be read from bottom to top, with the last line indicating the final error message and exception name. The debugging process involves examining error messages and searching for similar issues on Stack Overflow or Google, verifying model identifiers on the Hub, and using tools like list_repo_files to inspect repository contents. The huggingface_hub library provides tools for debugging repositories on the Hub. When debugging the forward pass, common issues include passing incorrect data types to models expecting specific tensor formats. The solution often involves adding parameters like return_tensors='pt' to tokenizer calls to ensure proper tensor conversion.Common issues encountered in training include data not being processed correctly, problems with batch formation, and label mismatches.

The Hugging Face forums have categories for beginners, intermediate, and research advanced questions as well as a section for questions specifically related to Hugging Face courses. To ask for help on the forums, titles should be reasonably descriptive, code snippets should be formatted properly using Markdown with three backticks for code blocks and single backticks for inline variables. Complete traceback is useful, as information higher in the traceback may be essential for debugging.

Section 9: Building and Sharing Demos

Section 9 teaches you how to turn machine learning models into interactive demos using Gradio, and how to share those demos with others or host them on Hugging Face Spaces. You learn that Gradio is model-agnostic, so it works with any ML framework, not just PyTorch, and can be launched from standard Python scripts, Jupyter notebooks, or Google Colab. The chapter focuses on the core Interface class, where you define your model function and connect it to pre-built UI components like textboxes, image uploaders, audio inputs, and more. You also learn how to support multiple inputs/outputs and how to add state, which lets your demo remember conversation history or intermediate values across interactions. The course then introduces Blocks, a more flexible system for building multi-step, multi-component apps with tabs, event triggers, conditional logic, and reusable components. Finally, you learn how to share demos using share=True or deploy them publicly through Hugging Face Spaces.

Section 10: Curate High Quality Datasets

Chapter 10 teaches how to use Argilla to annotate and curate datasets for training and evaluating models. Existing Hub datasets may not align with specific applications or use cases, necessitating custom dataset creation and curation. Argilla allows users to turn unstructured data into structured data for NLP tasks.

Set Up Your Argilla Instance

You can create an Argilla Space through Hugging Face by following a configuration form, with the option to enable persistent storage. After deployment, users log in with credentials and install the Argilla Python SDK using pip install argilla. Connecting to the Argilla instance requires the API URL, the API key, and optionally a Hugging Face token with write permissions for private spaces. To verify connection, call client.me, which should return user information if properly configured.

Load Your Dataset to Argilla

Define settings that represent annotation tasks. This chapter shows how to use a news dataset to complete two tasks: text classification on topics and token classification for named entity identification. Settings include fields and questions. For text classification, LabelQuestion uses unique values from existing label columbs as options, ensuring compatibility with pre-existing labels. For token classification, SpanQuestion defines entity labels like person, organization, location, and event. After defining settings, datasets are created using rg.Dataset with the specified configuration and settings. Pre-existing labels can be mapped as pre-annotations.

Annotate Your Dataset

Before beginning annotation, teams should establish alignment through annotation guidelines written in the dataset settings page. Guidelines help align teams on tasks and label usage. For tasks without suggestions like token classification, annotators manually add all labels by selecting text spans and assigning appropriate entity types. As annotators progress through records, they can submit responses when complete, save as drafts, or discard records that shouldn't be included in the dataset or won't receive responses.

Section 11: Fine Tune Large Language Models

Section 11 focuses on Supervised Fine-Tuning (SFT) of pretrained LMs using labeled instruction-response data, while using LoRA (Low-Rank Adapters) to make fine-tuning efficient and parameter-sparse. In practice you prepare a cleaned, tokenized dataset of prompts and targets, freeze most of the base model weights, and inject small low-rank adapter matrices into attention / feed-forward layers; only those adapter parameters are trained, which dramatically reduces GPU memory and storage needs. This workflow preserves the base model's general knowledge, speeds up training, and makes sharing updates cheaper (you can share just the LoRA weights), but it still requires careful hyperparameter tuning, proper batching/prompt formatting, and evaluation on held-out instruction examples to avoid overfitting. Monitoring training/validation loss and task metrics shows how well SFT+LoRA is improving task performance; if validation loss plateaus or diverges from training loss you should reduce learning rate, add regularization, or check data quality. Overall, SFT+LoRA is a practical compromise for adapting large models to supervised tasks while keeping compute and storage costs manageable.

Section 12: Build Reasoning Models

This chapter covers the basics of reinforcement learning and its role in LLMs through exploration of Open R1. LLMs are good at many generative tasks, but, until recently, they exhibited poor performance on reasoning problems.

Introduction to Reinforcement Learning

Reinforcement learning involves training an agent that learns from environment feedback. While LLMs are excellent at predicting the next word through pretraining on large amounts of text, this doesn't necessarily make them good at providing useful information, avoiding biased content, or responding naturally and engagingly. There are 5 key components to reinforcement learning:

- Agent: This is the learner. For LLMs, the LLM is the agent we want to train

- Environment: The environment provides feedback to the agent. For LLMs, this could be the user it interacts with, or a simulated scenario.

- Action: These are the choices the agent can make in the environment.

- Reward: This is the feedback the agent receives after taking an action. Rewards are usually numbers.
    - Positive rewards reinforce an action
    - Negative rewards are penalties that discourage an action

- Policy: This is an agent's strategy for choosing an action. It sets a function that tells the agent what action to take in different situations. As the agent learns, the policy becomes better at choosing actions that lead to higher rewards.

Reinforcement learning happens through trial and error. First, the agent observes the environment, then it takes action, receives feedback about the action, and finally updates policy based on the reward. This process is iteratively repeated.

Reinforcement learning from human feedback uses human feedback as a proxy for the reward signal. Human preference data is used to train a reward model that learns to predict what kinds of responses humans will prefer. The reward model is then used to fine-tune the LLM.

Understanding DeepSeek R1

Deepseek represents a significant advancement in developing reasoning capabilities through reinforcement learning. A new reinforcement algorithm called group relative policy optimization (GRPO) is introduced. The goal of this paper was to explore whether pure reinforcement learning could develop reasoning capabilities without supervised fine-tuning, departing from the standard approach requiring supervised fine-tuning. The R1-Zero model made initial attempts, recognize errors or inconsistencies, and adjust approaches based on recognition and explain why new approaches are better. This emerged naturally from RL training without explicit programming, demonstrating learning rather than memorization. Deepseek R1 expanded upon this by prioritizing both reasoning performance and usability through multiple training phases. The reasoning RL focuses on developing core reasoning capabilities across mathematics, coding, science, and logic domains.

GRPO

Reinforcement learning involves training an agent that learns from environmental feedback. Reinforcement learning from human feedback (RLHF) requires collecting human preference data comparing different generated responses. Group Relative Policy Optimization directly evaluates model generated responses by comparing them within groups of generation to optimize the policy of the model, rather than training a separate value model. A reward score is assigned to each generated response. Outputs within the same group are compared. Using standardization allows the model to assess each response's performance. Advantage values greater than 0 is better than average while advantage values less than 0 are worse than the average. The advantage values are fed to a target function for policy update. The policy update uses a clipped objective function with a KL divergence penalty to ensure stable learning. The Transformer Reinforcement Learning library provides practical GRPO implementation through the GRPO Trainer class.