Llama2

Overview

The Llama2 model was proposed in LLaMA: Open Foundation and Fine-Tuned Chat Models by Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushka rMishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing EllenTan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, Thomas Scialom. It is a collection of foundation language models ranging from 7B to 70B parameters, with checkpoints finetuned for chat application!

The abstract from the paper is the following:

In this work, we develop and release Llama 2, a collection of pretrained and fine-tuned large language models (LLMs) ranging in scale from 7 billion to 70 billion parameters. Our fine-tuned LLMs, called Llama 2-Chat, are optimized for dialogue use cases. Our models outperform open-source chat models on most benchmarks we tested, and based on our human evaluations for helpfulness and safety, may be a suitable substitute for closed-source models. We provide a detailed description of our approach to fine-tuning and safety improvements of Llama 2-Chat in order to enable the community to build on our work and contribute to the responsible development of LLMs.

Checkout all Llama2 model checkpoints here. This model was contributed by Arthur Zucker with contributions from Lysandre Debut. The code of the implementation in Hugging Face is based on GPT-NeoX here. The original code of the authors can be found here.

Usage tips

> The `Llama2` models were trained using `bfloat16`, but the original inference uses `float16`. The checkpoints uploaded on the Hub use `torch_dtype = 'float16'`, which will be used by the `AutoModel` API to cast the checkpoints from `torch.float32` to `torch.float16`.
> The `dtype` of the online weights is mostly irrelevant unless you are using `torch_dtype="auto"` when initializing a model using `model = AutoModelForCausalLM.from_pretrained("path", torch_dtype = "auto")`. The reason is that the model will first be downloaded (using the `dtype` of the checkpoints online), then it will be casted to the default `dtype` of `torch` (becomes `torch.float32`), and finally, if there is a `torch_dtype` provided in the config, it will be used.
> Training the model in `float16` is not recommended and is known to produce `nan`; as such, the model should be trained in `bfloat16`.

Tips:

- Weights for the Llama2 models can be obtained by filling out this form
- The architecture is very similar to the first Llama, with the addition of Grouped Query Attention (GQA) following this paper

- Setting `config.pretraining_tp` to a value different than 1 will activate the more accurate but slower computation of the linear layers, which should better match the original logits.
- The original model uses `pad_id = -1` which means that there is no padding token. We can't have the same logic, make sure to add a padding token using `tokenizer.add_special_tokens({"pad_token":"<pad>"})` and resize the token embedding accordingly. You should also set the `model.config.pad_token_id`. The `embed_tokens` layer of the model is initialized with `self.embed_tokens = nn.Embedding(config.vocab_size, config.hidden_size, self.config.padding_idx)`, which makes sure that encoding the padding token will output zeros, so passing it when initializing is recommended.
- After filling out the form and gaining access to the model checkpoints, you should be able to use the already converted checkpoints. Otherwise, if you are converting your own model, feel free to use the conversion script. The script can be called with the following (example) command:

```
python src/transformers/models/llama/convert_llama_weights_to_hf.py \
    --input_dir /path/to/downloaded/llama/weights --model_size 7B --output_dir /output/pa
```

- After conversion, the model and tokenizer can be loaded via:

```
from transformers import LlamaForCausalLM, LlamaTokenizer

tokenizer = LlamaTokenizer.from_pretrained("/output/path")
model = LlamaForCausalLM.from_pretrained("/output/path")
```

Note that executing the script requires enough CPU RAM to host the whole model in float16 precision (even if the biggest versions come in several checkpoints they each contain a part of each weight of the model, so we need to load them all in RAM). For the 75B model, it's thus 145GB of RAM needed.

- The LLaMA tokenizer is a BPE model based on sentencepiece. One quirk of sentencepiece is that when decoding a sequence, if the first token is the start of the word (e.g. "Banana"), the tokenizer does not prepend the prefix space to the string.

- When using Flash Attention 2 via `attn_implementation="flash_attention_2"`, don't pass `torch_dtype` to the `from_pretrained` class method and use Automatic Mixed-Precision training. When using `Trainer`, it is simply specifying either `fp16` or `bf16` to `True`. Otherwise, make sure you are using `torch.autocast`. This is required because the Flash Attention only support `fp16` and `bf16` data type.

Resources

A list of official Hugging Face and community (indicated by 🌎) resources to help you get started with LLaMA2. If you're interested in submitting a resource to be included here, please feel free to open a Pull Request and we'll review it! The resource should ideally demonstrate something new instead of duplicating an existing resource.

- Llama 2 is here - get it on Hugging Face, a blog post about Llama 2 and how to use it with 🤗 Transformers and 🤗 PEFT.
- LLaMA 2 - Every Resource you need, a compilation of relevant resources to learn about LLaMA 2 and how to get started quickly.

Text Generation

- A notebook on how to fine-tune Llama 2 in Google Colab using QLoRA and 4-bit precision. 🌎
- A notebook on how to fine-tune the "Llama-v2-7b-guanaco" model with 4-bit QLoRA and generate Q&A datasets from PDFs. 🌎

Text Classification

- A notebook on how to fine-tune the Llama 2 model with QLoRa, TRL, and Korean text classification dataset. 🌎 🇰🇷

⚗️ Optimization

- Fine-tune Llama 2 with DPO, a guide to using the TRL library's DPO method to fine tune Llama 2 on a specific dataset.

- Extended Guide: Instruction-tune Llama 2, a guide to training Llama 2 to generate instructions from inputs, transforming the model from instruction-following to instruction-giving.
- A notebook on how to fine-tune the Llama 2 model on a personal computer using QLoRa and TRL. 🌎

⚡️ Inference

- A notebook on how to quantize the Llama 2 model using GPTQ from the AutoGPTQ library. 🌎
- A notebook on how to run the Llama 2 Chat Model with 4-bit quantization on a local computer or Google Colab. 🌎

🚀 Deploy

- Fine-tune LLaMA 2 (7-70B) on Amazon SageMaker, a complete guide from setup to QLoRA fine-tuning and deployment on Amazon SageMaker.
- Deploy Llama 2 7B/13B/70B on Amazon SageMaker, a guide on using Hugging Face's LLM DLC container for secure and scalable deployment.

LlamaConfig

class transformers.LlamaConfig <>

```
( vocab_size = 32000, hidden_size = 4096, intermediate_size = 11008, num_hidden_layers = 32,
num_attention_heads = 32, num_key_value_heads = None, hidden_act = 'silu',
max_position_embeddings = 2048, initializer_range = 0.02, rms_norm_eps = 1e-06, use_cache =
True, pad_token_id = None, bos_token_id = 1, eos_token_id = 2, pretraining_tp = 1,
tie_word_embeddings = False, rope_theta = 10000.0, rope_scaling = None, attention_bias =
False, attention_dropout = 0.0, mlp_bias = False, head_dim = None, **kwargs )
```

Expand 22 parameters

This is the configuration class to store the configuration of a LlamaModel. It is used to instantiate an LLaMA model according to the specified arguments, defining the model architecture. Instantiating a configuration with the defaults will yield a similar configuration to that of the LLaMA-7B.

Configuration objects inherit from PretrainedConfig and can be used to control the model outputs. Read the documentation from PretrainedConfig for more information.

```
>>> from transformers import LlamaModel, LlamaConfig

>>> # Initializing a LLaMA llama-7b style configuration
>>> configuration = LlamaConfig()

>>> # Initializing a model from the llama-7b style configuration
>>> model = LlamaModel(configuration)

>>> # Accessing the model configuration
>>> configuration = model.config
```

LlamaTokenizer

class transformers.**LlamaTokenizer** <>

```
( vocab_file, unk_token = '<unk>', bos_token = '<s>', eos_token = '</s>', pad_token = None,
sp_model_kwargs: Optional = None, add_bos_token = True, add_eos_token = False,
clean_up_tokenization_spaces = False, use_default_system_prompt = False,
spaces_between_special_tokens = False, legacy = None, add_prefix_space = True, **kwargs )
```

Expand 12 parameters

Construct a Llama tokenizer. Based on byte-level Byte-Pair-Encoding. The default padding token is unset as there is no padding token in the original model.

build_inputs_with_special_tokens <>

```
( token_ids_0, token_ids_1 = None )
```

get_special_tokens_mask <>

```
( token_ids_0: List, token_ids_1: Optional = None, already_has_special_tokens: bool = False
) → List[int]
```

Parameters

- **token_ids_0** (`List[int]`) — List of IDs.
- **token_ids_1** (`List[int]`, *optional*) — Optional second list of IDs for sequence pairs.
- **already_has_special_tokens** (`bool`, *optional*, defaults to `False`) — Whether or not the token list is already formatted with special tokens for the model.

Returns `List[int]`

A list of integers in the range [0, 1]: 1 for a special token, 0 for a sequence token.

Retrieve sequence ids from a token list that has no special tokens added. This method is called when adding special tokens using the tokenizer `prepare_for_model` method.

create_token_type_ids_from_sequences <>

`( token_ids_0: List, token_ids_1: Optional = None ) → List[int]`

Parameters

- **token_ids_0** (`List[int]`) — List of ids.
- **token_ids_1** (`List[int]`, *optional*) — Optional second list of IDs for sequence pairs.

Returns `List[int]`

List of token type IDs according to the given sequence(s).

Creates a mask from the two sequences passed to be used in a sequence-pair classification task. An ALBERT

sequence pair mask has the following format:

```
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
| first sequence    | second sequence |
```

if token_ids_1 is None, only returns the first portion of the mask (0s).

save_vocabulary <>

`( save_directory, filename_prefix: Optional = None ) → Tuple(str)`

Parameters

- **save_directory** (`str`) — The directory in which to save the vocabulary.

Returns `Tuple(str)`

Paths to the files saved.

Save the vocabulary and special tokens file to a directory.

LlamaTokenizerFast

class transformers.**LlamaTokenizerFast** <>

```
( vocab_file = None, tokenizer_file = None, clean_up_tokenization_spaces = False, unk_token =
'<unk>', bos_token = '<s>', eos_token = '</s>', add_bos_token = True, add_eos_token = False,
use_default_system_prompt = False, legacy = None, add_prefix_space = None, **kwargs )
```

Expand 10 parameters

Construct a Llama tokenizer. Based on byte-level Byte-Pair-Encoding.

This uses notably ByteFallback and no normalization.

```
>>> from transformers import LlamaTokenizerFast

>>> tokenizer = LlamaTokenizerFast.from_pretrained("hf-internal-testing/llama-tokenizer
>>> tokenizer.encode("Hello this is a test")
[1, 15043, 445, 338, 263, 1243]
```

If you want to change the `bos_token` or the `eos_token`, make sure to specify them when initializing the model, or call `tokenizer.update_post_processor()` to make sure that the post-processing is correctly done (otherwise the values of the first token and final token of an encoded sequence will not be correct). For more details, checkout [post-processors] (https://huggingface.co/docs/tokenizers/api/post-processors) documentation.

This tokenizer inherits from PreTrainedTokenizerFast which contains most of the main methods. Users should refer to this superclass for more information regarding those methods.

build_inputs_with_special_tokens

<>

```
( token_ids_0, token_ids_1 = None )
```

get_special_tokens_mask

<>

```
( token_ids_0: List, token_ids_1: Optional = None, already_has_special_tokens: bool = False
) → A list of integers in the range [0, 1]
```

Parameters

- **token_ids_0** (`List[int]`) — List of ids of the first sequence.
- **token_ids_1** (`List[int]`, *optional*) — List of ids of the second sequence.
- **already_has_special_tokens** (`bool`, *optional*, defaults to `False`) — Whether or not the token list is already formatted with special tokens for the model.

Returns A list of integers in the range [0, 1]

1 for a special token, 0 for a sequence token.

Retrieves sequence ids from a token list that has no special tokens added. This method is called when adding special tokens using the tokenizer `prepare_for_model` or `encode_plus` methods.

create_token_type_ids_from_sequences

`( token_ids_0: List, token_ids_1: Optional = None ) → List[int]`

Parameters

- **token_ids_0** (`List[int]`) — The first tokenized sequence.
- **token_ids_1** (`List[int]`, *optional*) — The second tokenized sequence.

Returns `List[int]`

The token type ids.

Create the token type IDs corresponding to the sequences passed. What are token type IDs?

Should be overridden in a subclass if the model has a special way of building those.

update_post_processor

`( )`

Updates the underlying post processor with the current `bos_token` and `eos_token`.

save_vocabulary

`( save_directory: str, filename_prefix: Optional = None )`

LlamaModel

class transformers.LlamaModel

<>

```
( config: LlamaConfig )
```

Parameters

- **config** (LlamaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights. config — LlamaConfig

The bare LLaMA Model outputting raw hidden-states without any specific head on top. This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

Transformer decoder consisting of *config.num_hidden_layers* layers. Each layer is a `LlamaDecoderLayer`

forward

<>

```
( input_ids: LongTensor = None, attention_mask: Optional = None, position_ids: Optional =
None, past_key_values: Union = None, inputs_embeds: Optional = None, use_cache: Optional =
None, output_attentions: Optional = None, output_hidden_states: Optional = None,
return_dict: Optional = None, cache_position: Optional = None )
```

Expand 10 parameters

The LlamaModel forward method, overrides the `__call__` special method.

Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

LlamaForCausalLM

class transformers.**LlamaForCausalLM** <>

```
( config )
```

forward <>

```
( input_ids: LongTensor = None, attention_mask: Optional = None, position_ids: Optional =
None, past_key_values: Union = None, inputs_embeds: Optional = None, labels: Optional =
None, use_cache: Optional = None, output_attentions: Optional = None, output_hidden_states:
Optional = None, return_dict: Optional = None, cache_position: Optional = None,
num_logits_to_keep: int = 0 ) → transformers.modeling_outputs.CausalLMOutputWithPast or
tuple(torch.FloatTensor)
```

Expand 10 parameters

The LlamaForCausalLM forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

Example:

```
>>> from transformers import AutoTokenizer, LlamaForCausalLM

>>> model = LlamaForCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf")
>>> tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")

>>> prompt = "Hey, are you conscious? Can you talk to me?"
>>> inputs = tokenizer(prompt, return_tensors="pt")

>>> # Generate
>>> generate_ids = model.generate(inputs.input_ids, max_length=30)
>>> tokenizer.batch_decode(generate_ids, skip_special_tokens=True, clean_up_tokeniza
"Hey, are you conscious? Can you talk to me?\nI'm not conscious, but I can talk to y
```

LlamaForSequenceClassification

class transformers.**LlamaForSequenceClassification** <>

(config)

Parameters

- **config** (LlamaConfig) — Model configuration class with all the parameters of the model. Initializing with a config file does not load the weights associated with the model, only the configuration. Check out the from_pretrained() method to load the model weights.

The LLaMa Model transformer with a sequence classification head on top (linear layer).

LlamaForSequenceClassification uses the last token in order to do the classification, as other causal models (e.g. GPT-2) do.

Since it does classification on the last token, it requires to know the position of the last token. If a `pad_token_id` is defined in the configuration, it finds the last token that is not a padding token in each row. If no `pad_token_id` is defined, it simply takes the last value in each row of the batch. Since it cannot guess the padding tokens when `inputs_embeds` are passed instead of `input_ids`, it does the same (take the last value in each row of the batch).

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads etc.)

This model is also a PyTorch torch.nn.Module subclass. Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage and behavior.

forward <>

```
( input_ids: Optional = None, attention_mask: Optional = None, position_ids: Optional =
None, past_key_values: Union = None, inputs_embeds: Optional = None, labels: Optional =
None, use_cache: Optional = None, output_attentions: Optional = None, output_hidden_states:
Optional = None, return_dict: Optional = None )
```

Expand 11 parameters

The LlamaForSequenceClassification forward method, overrides the `__call__` special method.

> Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the pre and post processing steps while the latter silently ignores them.

<> Update on GitHub

← LLaMA Llama3 →