

A Gradio web UI for Large Language Models. Supports transformers, GPTQ, AWQ, EXL2, llama.cpp (GGUF), Llama models.

Features

- 3 interface modes: default (two columns), notebook, and chat.
- Multiple model backends: [Transformers](#), [llama.cpp](#) (through [llama-cpp-python](#)), [ExLlamaV2](#), [AutoGPTQ](#), [AutoAWQ](#), [GPTQ-for-LLaMa](#), [QuIP#](#).
- Dropdown menu for quickly switching between different models.
- Large number of extensions (built-in and user-contributed), including Coqui TTS for realistic voice outputs, Whisper STT for voice inputs, translation, multimodal pipelines, vector databases, Stable Diffusion integration, and a lot more.
- Chat with custom characters.
- Precise chat templates for instruction-following models, including Llama-2-chat, Alpaca, Vicuna, Mistral.
- LoRA: train new LoRAs with your own data, load/unload LoRAs on the fly for generation.
- Transformers library integration: load models in 4-bit or 8-bit precision through bitsandbytes, use llama.cpp with transformers samplers (`llamacpp_HF` loader), CPU inference in 32-bit precision using PyTorch.
- OpenAI-compatible API server with Chat and Completions endpoints

What Works

Loader	Loadin g 1 LoRA	Loadin g 2 or more LoRAs	Trainin g LoRAs	Multimoda l extension	Perplexity evaluation
Transformers	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> ***	<input checked="" type="checkbox"/> *	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Loader	Loadin g 1 LoRA	Loadin g 2 or more LoRAs	Trainin g LoRAs	Multimoda l extension	Perplexity evaluation
ExLlamav2_H F	☑	☑	✗	✗	☑
ExLlamav2	☑	☑	✗	✗	use ExLlamav2_H F
AutoGPTQ	☑	✗	✗	☑	☑
GPTQ-for- LLaMa	☑ ^{**}	☑ ^{***}	☑	☑	☑
llama.cpp	✗	✗	✗	✗	use llamacpp_HF
llamacpp_HF	✗	✗	✗	✗	☑
ctransformers	✗	✗	✗	✗	✗
AutoAWQ	?	✗	?	?	☑

✗ = not implemented

☑ = implemented

* Training LoRAs with GPTQ models also works with the Transformers loader. Make sure to check "auto-devices" and "disable_exllama" before loading the model.

** Requires the monkey-patch.*** Multi-LoRA in PEFT is tricky and the current implementation does not work reliably in all cases.

Training Your Own LoRAs

The WebUI seeks to make training your own LoRAs as easy as possible. It comes down to just a few simple steps:

Step 1: Make a plan.

- What base model do you want to use? The LoRA you make has to be matched up to a single architecture (eg LLaMA-13B) and cannot be transferred to others (eg LLaMA-7B, StableLM, etc. would all be different). Derivatives of the same model (eg Alpaca finetune of LLaMA-13B) might be transferrable, but even then it's best to train exactly on what you plan to use.
- What are you training it on? Do you want it to learn real information, a simple format, ...?

Step 2: Gather a dataset.

- If you use a dataset similar to the [Alpaca](#) format, that is natively supported by the `Formatted Dataset` input in the WebUI, with premade formatter options.
- If you use a dataset that isn't matched to Alpaca's format, but uses the same basic JSON structure, you can make your own format file by copying `training/formats/alpaca-format.json` to a new file and editing its content.
- If you can get the dataset into a simple text file, that works too! You can train using the `Raw text file` input option.
 - This means you can for example just copy/paste a chatlog/documentation page/whatever you want, shove it in a plain text file, and train on it.
- If you use a structured dataset not in this format, you may have to find an external way to convert it - or open an issue to request native support.

Step 3: Do the training.

- **3.1:** Load the WebUI, and your model.
 - Make sure you don't have any LoRAs already loaded (unless you want to train for multi-LoRA usage).
- **3.2:** Open the `Training` tab at the top, `Train LoRA` sub-tab.
- **3.3:** Fill in the name of the LoRA, select your dataset in the dataset options.
- **3.4:** Select other parameters to your preference.

- **3.5:** click `Start LoRA Training`, and wait.
 - It can take a few hours for a large dataset, or just a few minute if doing a small run.
 - You may want to monitor your loss value while it goes.

Step 4: Evaluate your results.

- Load the LoRA under the Models Tab.
- You can go test-drive it on the `Text generation` tab, or you can use the `Perplexity evaluation` sub-tab of the `Training` tab.
- If you used the `Save every n steps` option, you can grab prior copies of the model from sub-folders within the LoRA model's folder and try them instead.

Step 5: Re-run if you're unhappy.

- Make sure to unload the LoRA before training it.
- You can simply resume a prior run - use `Copy parameters from` to select your LoRA, and edit parameters. Note that you cannot change the `Rank` of an already created LoRA.
 - If you want to resume from a checkpoint saved along the way, simply copy the contents of the checkpoint folder into the LoRA's folder.
 - (Note: `adapter_model.bin` is the important file that holds the actual LoRA content).
 - This will start Learning Rate and Steps back to the start. If you want to resume as if you were midway through, you can adjust your Learning Rate to the last reported LR in logs and reduce your epochs.
- Or, you can start over entirely if you prefer.
- If your model is producing corrupted outputs, you probably need to start over and use a lower Learning Rate.
- If your model isn't learning detailed information but you want it to, you might need to just run more epochs, or you might need a higher Rank.
- If your model is enforcing a format you didn't want, you may need to tweak your dataset, or start over and not train as far.

Format Files

If using JSON formatted datasets, they are presumed to be in the following approximate format:

```
[
  {
    "somekey": "somevalue",
    "key2": "value2"
  },
  {
    // etc
  }
]
```

Where the keys (eg `somekey`, `key2` above) are standardized, and relatively consistent across the dataset, and the values (eg `somevalue`, `value2`) contain the content actually intended to be trained.

For Alpaca, the keys are `instruction`, `input`, and `output`, wherein `input` is sometimes blank.

A simple format file for Alpaca to be used as a chat bot is:

```
{
  "instruction,output": "User: %instruction%\nAssistant: %output%",
  "instruction,input,output": "User: %instruction%: %input%\nAssistant: %output%"
}
```

Note that the keys (eg `instruction,output`) are a comma-separated list of dataset keys, and the values are a simple string that use those keys with `%%`.

So for example if a dataset has `"instruction": "answer my question"`, then the format file's `User: %instruction%\n` will be automatically filled in as `User: answer my question\n`.

If you have different sets of key inputs, you can make your own format file to match it. This format-file is designed to be as simple as possible to enable easy editing to match your needs.

Raw Text File Settings

When using raw text files as your dataset, the text is automatically split into chunks based on your `Cutoff Length` you get a few basic options to configure them.

- `Overlap Length` is how much to overlap chunks by. Overlapping chunks helps prevent the model from learning strange mid-sentence cuts, and instead learn continual sentences that flow from earlier text.
- `Prefer Newline Cut Length` sets a maximum distance in characters to shift the chunk cut towards newlines. Doing this helps prevent lines from starting or ending mid-sentence, preventing the model from learning to cut off sentences randomly.
- `Hard Cut String` sets a string that indicates there must be a hard cut without overlap. This defaults to `\n\n\n`, meaning 3 newlines. No trained chunk will ever

contain this string. This allows you to insert unrelated sections of text in the same text file, but still ensure the model won't be taught to randomly change the subject.

Parameters

The basic purpose and function of each parameter is documented on-page in the WebUI, so read through them in the UI to understand your options.

That said, here's a guide to the most important parameter choices you should consider:

VRAM

- First, you must consider your VRAM availability.
 - Generally, under default settings, VRAM usage for training with default parameters is very close to when generating text (with 1000+ tokens of context) (ie, if you can generate text, you can train LoRAs).
 - Note: worse by default in the 4-bit monkeypatch currently.
Reduce `Micro Batch Size` to `1` to restore this to expectations.
 - If you have VRAM to spare, setting higher batch sizes will use more VRAM and get you better quality training in exchange.
 - If you have large data, setting a higher cutoff length may be beneficial, but will cost significant VRAM. If you can spare some, set your batch size to `1` and see how high you can push your cutoff length.
 - If you're low on VRAM, reducing batch size or cutoff length will of course improve that.
 - Don't be afraid to just try it and see what happens. If it's too much, it will just error out, and you can lower settings and try again.

Rank

- Second, you want to consider the amount of learning you want.
 - For example, you may wish to just learn a dialogue format (as in the case of Alpaca) in which case setting a low `Rank` value (32 or lower) works great.
 - Or, you might be training on project documentation you want the bot to understand and be able to understand questions about, in which case the higher the rank, the better.

- Generally, higher Rank = more precise learning = more total content learned = more VRAM usage while training.

Learning Rate and Epochs

- Third, how carefully you want it to be learned.
 - In other words, how okay or not you are with the model losing unrelated understandings.
 - You can control this with 3 key settings: the Learning Rate, its scheduler, and your total epochs.
 - The learning rate controls how much change is made to the model by each token it sees.
 - It's in scientific notation normally, so for example $3e-4$ means 3×10^{-4} which is 0.0003. The number after $e-$ controls how many 0s are in the number.
 - Higher values let training run faster, but also are more likely to corrupt prior data in the model.
 - You essentially have two variables to balance: the LR, and Epochs.
 - If you make LR higher, you can set Epochs equally lower to match. High LR + low epochs = very fast, low quality training.
 - If you make LR low, set epochs high. Low LR + high epochs = slow but high-quality training.
 - The scheduler controls change-over-time as you train - it starts high, and then goes low. This helps balance getting data in, and having decent quality, at the same time.

Loss

When you're running training, the WebUI's console window will log reports that include, among other things, a numeric value named `LOSS`. It will start as a high number, and gradually get lower and lower as it goes.

"Loss" in the world of AI training theoretically means "how close is the model to perfect", with 0 meaning "absolutely perfect". This is calculated by measuring the difference between the model outputting exactly the text you're training it to output, and what it actually outputs.

In practice, a good LLM should have a very complex variable range of ideas running in its artificial head, so a loss of 0 would indicate that the model has broken and forgotten to how think about anything other than what you trained it.

So, in effect, Loss is a balancing game: you want to get it low enough that it understands your data, but high enough that it isn't forgetting everything else. Generally, if it goes below 1.0, it's going to start forgetting its prior memories, and you should stop training. In some cases you may prefer to take it as low as 0.5 (if you want it to be very very predictable). Different goals have different needs, so don't be afraid to experiment and see what works best for you.

Note: if you see Loss start at or suddenly jump to exactly 0, it is likely something has gone wrong in your training process (eg model corruption).

Note: 4-Bit Monkeypatch

The 4-bit LoRA monkeypatch works for training, but has side effects:

- VRAM usage is higher currently. You can reduce the `Micro Batch Size` to 1 to compensate.
- Models do funky things. LoRAs apply themselves, or refuse to apply, or spontaneously error out, or etc. It can be helpful to reload base model or restart the WebUI between training/usage to minimize chances of anything going haywire.
- Loading or working with multiple LoRAs at the same time doesn't currently work.
- Generally, recognize and treat the monkeypatch as the dirty temporary hack it is - it works, but isn't very stable. It will get better in time when everything is merged upstream for full official support.

Model loaders

Transformers

Loads: full precision (16-bit or 32-bit) models. The repository usually has a clean name without GGUF, EXL2, GPTQ, or AWQ in its name, and the model files are named `pytorch_model.bin` or `model.safetensors`.

Example: <https://huggingface.co/lmsys/vicuna-7b-v1.5>.

Full precision models use a ton of VRAM, so you will usually want to select the "load_in_4bit" and "use_double_quant" options to load the model in 4-bit precision using bitsandbytes.

This loader can also load GPTQ models and train LoRAs with them. For that, make sure to check the "auto-devices" and "disable_exllama" options before loading the model.

Options:

- **gpu-memory:** When set to greater than 0, activates CPU offloading using the accelerate library, where part of the layers go to the CPU. The performance is very bad. Note that accelerate doesn't treat this parameter very literally, so if you want the VRAM usage to be at most 10 GiB, you may need to set this parameter to 9 GiB or 8 GiB. It can be used in conjunction with "load_in_8bit" but not with "load-in-4bit" as far as I'm aware.
- **cpu-memory:** Similarly to the parameter above, you can also set a limit on the amount of CPU memory used. Whatever doesn't fit either in the GPU or the CPU will go to a disk cache, so to use this option you should also check the "disk" checkbox.
- **compute_dtype:** Used when "load-in-4bit" is checked. I recommend leaving the default value.
- **quant_type:** Used when "load-in-4bit" is checked. I recommend leaving the default value.
- **alpha_value:** Used to extend the context length of a model with a minor loss in quality. I have measured 1.75 to be optimal for 1.5x context, and 2.5 for 2x context. That is, with $\alpha = 2.5$ you can make a model with 4096 context length go to 8192 context length.
- **rope_freq_base:** Originally another way to write "alpha_value", it ended up becoming a necessary parameter for some models like CodeLlama, which was fine-tuned with this set to 1000000 and hence needs to be loaded with it set to 1000000 as well.
- **compress_pos_emb:** The first and original context-length extension method, discovered by [kaiokendev](#). When set to 2, the context length is doubled, 3 and it's tripled, etc. It should only be used for models that have been fine-tuned with this parameter set to different than 1. For models that have not been tuned to have greater context length, alpha_value will lead to a smaller accuracy loss.
- **cpu:** Loads the model in CPU mode using Pytorch. The model will be loaded in 32-bit precision, so a lot of RAM will be used. CPU inference with transformers is older than llama.cpp and it works, but it's a lot slower. Note: this parameter has a different interpretation in the llama.cpp loader (see below).
- **load-in-8bit:** Load the model in 8-bit precision using bitsandbytes. The 8-bit kernel in that library has been optimized for training and not inference, so load-in-8bit is slower than load-in-4bit (but more accurate).
- **bf16:** Use bfloat16 precision instead of float16 (the default). Only applies when quantization is not used.

- **auto-devices:** When checked, the backend will try to guess a reasonable value for "gpu-memory" to allow you to load a model with CPU offloading. I recommend just setting "gpu-memory" manually instead. This parameter is also needed for loading GPTQ models, in which case it needs to be checked before loading the model.
- **disk:** Enable disk offloading for layers that don't fit into the GPU and CPU combined.
- **load-in-4bit:** Load the model in 4-bit precision using bitsandbytes.
- **trust-remote-code:** Some models use custom Python code to load the model or the tokenizer. For such models, this option needs to be set. It doesn't download any remote content: all it does is execute the .py files that get downloaded with the model. Those files can potentially include malicious code; I have never seen it happen, but it is in principle possible.
- **no_use_fast:** Do not use the "fast" version of the tokenizer. Can usually be ignored; only check this if you can't load the tokenizer for your model otherwise.
- **use_flash_attention_2:** Set use_flash_attention_2=True while loading the model. Possibly useful for training.
- **disable_exllama:** Only applies when you are loading a GPTQ model through the transformers loader. It needs to be checked if you intend to train LoRAs with the model.

ExLlamav2_HF

Loads: GPTQ and EXL2 models. EXL2 models usually have "EXL2" in the model name, while GPTQ models usually have GPTQ in the model name, or alternatively something like "-4bit-128g" in the name.

Examples:

- <https://huggingface.co/turboderp/Llama2-70B-exl2>
- <https://huggingface.co/TheBloke/Llama-2-13B-chat-GPTQ>
- **gpu-split:** If you have multiple GPUs, the amount of memory to allocate per GPU should be set in this field. Make sure to set a lower value for the first GPU, as that's where the cache is allocated.
- **max_seq_len:** The maximum sequence length for the model. In ExLlamaV2, the cache is preallocated, so the higher this value, the higher the VRAM. It is automatically set to the maximum sequence length for the model based on its

metadata, but you may need to lower this value to be able to fit the model into your GPU. After loading the model, the "Truncate the prompt up to this length" parameter under "Parameters" > "Generation" is automatically set to your chosen "max_seq_len" so that you don't have to set the same thing twice.

- **cfg-cache:** Creates a second cache to hold the CFG negative prompts. You need to set this if and only if you intend to use CFG in the "Parameters" > "Generation" tab. Checking this parameter doubles the cache VRAM usage.
- **no_flash_attn:** Disables flash attention. Otherwise, it is automatically used as long as the library is installed.
- **cache_8bit:** Create a 8-bit precision cache instead of a 16-bit one. This saves VRAM but increases perplexity (I don't know by how much).
- **cache_4bit:** Creates a Q4 cache using grouped quantization.

ExLlamav2

The same as ExLlamav2_HF but using the internal samplers of ExLlamav2 instead of the ones in the Transformers library.

AutoGPTQ

Loads: GPTQ models.

- **wbits:** For ancient models without proper metadata, sets the model precision in bits manually. Can usually be ignored.
- **groupsize:** For ancient models without proper metadata, sets the model group size manually. Can usually be ignored.
- **triton:** Only available on Linux. Necessary to use models with both act-order and groupsize simultaneously. Note that ExLlamaV2 can load these same models on Windows without triton.
- **no_inject_fused_attention:** Improves performance while increasing the VRAM usage.
- **no_inject_fused_mlp:** Similar to the previous parameter but for Triton only.
- **no_use_cuda_fp16:** On some systems, the performance can be very bad with this unset. Can usually be ignored.

- **desc_act**: For ancient models without proper metadata, sets the model "act-order" parameter manually. Can usually be ignored.

GPTQ-for-LLaMa

Loads: GPTQ models.

Ancient loader, the first one to implement 4-bit quantization. It works on older GPUs for which ExLlamaV2 and AutoGPTQ do not work, and it doesn't work with "act-order", so you should use it with simple 4-bit-128g models.

- **pre_layer**: Used for CPU offloading. The higher the number, the more layers will be sent to the GPU. GPTQ-for-LLaMa CPU offloading was faster than the one implemented in AutoGPTQ the last time I checked.

llama.cpp

Loads: GGUF models. Note: GGML models have been deprecated and do not work anymore.

Example: <https://huggingface.co/TheBloke/Llama-2-7b-Chat-GGUF>

- **n-gpu-layers**: The number of layers to allocate to the GPU. If set to 0, only the CPU will be used. If you want to offload all layers, you can simply set this to the maximum value.
- **n_ctx**: Context length of the model. In llama.cpp, the cache is preallocated, so the higher this value, the higher the VRAM. It is automatically set to the maximum sequence length for the model based on the metadata inside the GGUF file, but you may need to lower this value to be able to fit the model into your GPU. After loading the model, the "Truncate the prompt up to this length" parameter under "Parameters" > "Generation" is automatically set to your chosen "n_ctx" so that you don't have to set the same thing twice.
- **tensor_split**: For multi-gpu only. Sets the amount of memory to allocate per GPU as proportions. Not to be confused with other loaders where this is set in GB; here you can set something like `30, 70` for 30%/70%.
- **n_batch**: Batch size for prompt processing. Higher values are supposed to make generation faster, but I have never obtained any benefit from changing this value.
- **threads**: Number of threads. Recommended value: your number of physical cores.

- **threads_batch**: Number of threads for batch processing. Recommended value: your total number of cores (physical + virtual).
- **tensorcores**: Use llama.cpp compiled with "tensor cores" support, which improves performance on NVIDIA RTX cards in most cases.
- **streamingllm**: Experimental feature to avoid re-evaluating the entire prompt when part of it is removed, for instance, when you hit the context length for the model in chat mode and an old message is removed.
- **cpu**: Force a version of llama.cpp compiled without GPU acceleration to be used. Can usually be ignored. Only set this if you want to use CPU only and llama.cpp doesn't work otherwise.
- **no_mul_mat_q**: Disable the mul_mat_q kernel. This kernel usually improves generation speed significantly. This option to disable it is included in case it doesn't work on some system.
- **no-mmap**: Loads the model into memory at once, possibly preventing I/O operations later on at the cost of a longer load time.
- **mlock**: Force the system to keep the model in RAM rather than swapping or compressing (no idea what this means, never used it).
- **numa**: May improve performance on certain multi-cpu systems.

llamacpp_HF

The same as llama.cpp but with transformers samplers, and using the transformers tokenizer instead of the internal llama.cpp tokenizer.

To use it, you need to download a tokenizer. There are two options:

1. Download `oobabooga/llama-tokenizer` under "Download model or LoRA". That's a default Llama tokenizer.
2. Place your .gguf in a subfolder of `models/` along with these 3 files: `tokenizer.model`, `tokenizer_config.json`, and `special_tokens_map.json`. This takes precedence over Option 1.

It has an additional parameter:

- **logits_all**: Needs to be checked if you want to evaluate the perplexity of the llama.cpp model using the "Training" > "Perplexity evaluation" tab. Otherwise, leave it unchecked, as it makes prompt processing slower.


AutoAWQ

Loads: AWQ models.

Example: <https://huggingface.co/TheBloke/Phind-CodeLlama-34B-v2-AWQ>

The parameters are overall similar to AutoGPTQ.

Model dropdown

Here you can select a model to be loaded, refresh the list of available models () , load/unload/reload the selected model, and save the settings for the model. The "settings" are the values in the input fields (checkboxes, sliders, dropdowns) below this dropdown.

After saving, those settings will get restored whenever you select that model again in the dropdown menu.

If the **Autoload the model** checkbox is selected, the model will be loaded as soon as it is selected in this menu. Otherwise, you will have to click on the "Load" button.

LoRA dropdown

Used to apply LoRAs to the model. Note that LoRA support is not implemented for all loaders.

Download model or LoRA

Here you can download a model or LoRA directly from the <https://huggingface.co/> website.

- Models will be saved to `text-generation-webui/models`.
- LoRAs will be saved to `text-generation-webui/loras`.

In the input field, you can enter either the Hugging Face username/model path (like `facebook/galactica-125m`) or the full model URL (like `https://huggingface.co/facebook/galactica-125m`). To specify a branch, add it at the end after a ":" character like this: `facebook/galactica-125m:main`.

To download a single file, as necessary for models in GGUF format, you can click on "Get file list" after entering the model path in the input field, and then copy and paste the desired file name in the "File name" field before clicking on "Download".