

 Search models, datasets, users...

Transformers documentation

## How to add a model to 😊 Transformers?

# How to add a model to 😊 Transformers?

The 😊 Transformers library is often able to offer new models thanks to community contributors. But this can be a challenging project and requires an in-depth knowledge of the 😊 Transformers library and the model to implement. At Hugging Face, we're trying to empower more of the community to actively add models and we've put together this guide to walk you through the process of adding a PyTorch model (make sure you have [PyTorch installed](#)).

If you're interested in implementing a TensorFlow model, take a look at the [How to convert a 😊 Transformers model to TensorFlow](#) guide!

Along the way, you'll:

- get insights into open-source best practices
- understand the design principles behind one of the most popular deep learning libraries
- learn how to efficiently test large models
- learn how to integrate Python utilities like `black`, `ruff`, and `make fix-copies` to ensure clean and readable code

A Hugging Face team member will be available to help you along the way so you'll never be alone.



To get started, open a [New model addition](#) issue for the model you want to see in 😊 Transformers. If you're not especially picky about contributing a specific model, you can filter by the [New model label](#) to see if there are any unclaimed model requests and work on it.

Once you've opened a new model request, the first step is to get familiar with 😊 Transformers if you aren't already!

## General overview of 😊 Transformers

First, you should get a general overview of 😊 Transformers. 😊 Transformers is a very opinionated library, so there is a chance that you don't agree with some of the library's philosophies or design choices. From our experience, however, we found that the fundamental design choices and philosophies of the library are crucial to efficiently scale 😊 Transformers while keeping maintenance costs at a reasonable level.

A good first starting point to better understand the library is to read the [documentation of our philosophy](#). As a result of our way of working, there are some choices that we try to apply to all models:

- Composition is generally favored over abstraction
- Duplicating code is not always bad if it strongly improves the readability or accessibility of a model
- Model files are as self-contained as possible so that when you read the code of a specific model, you ideally only have to look into the respective `modeling_....py` file.

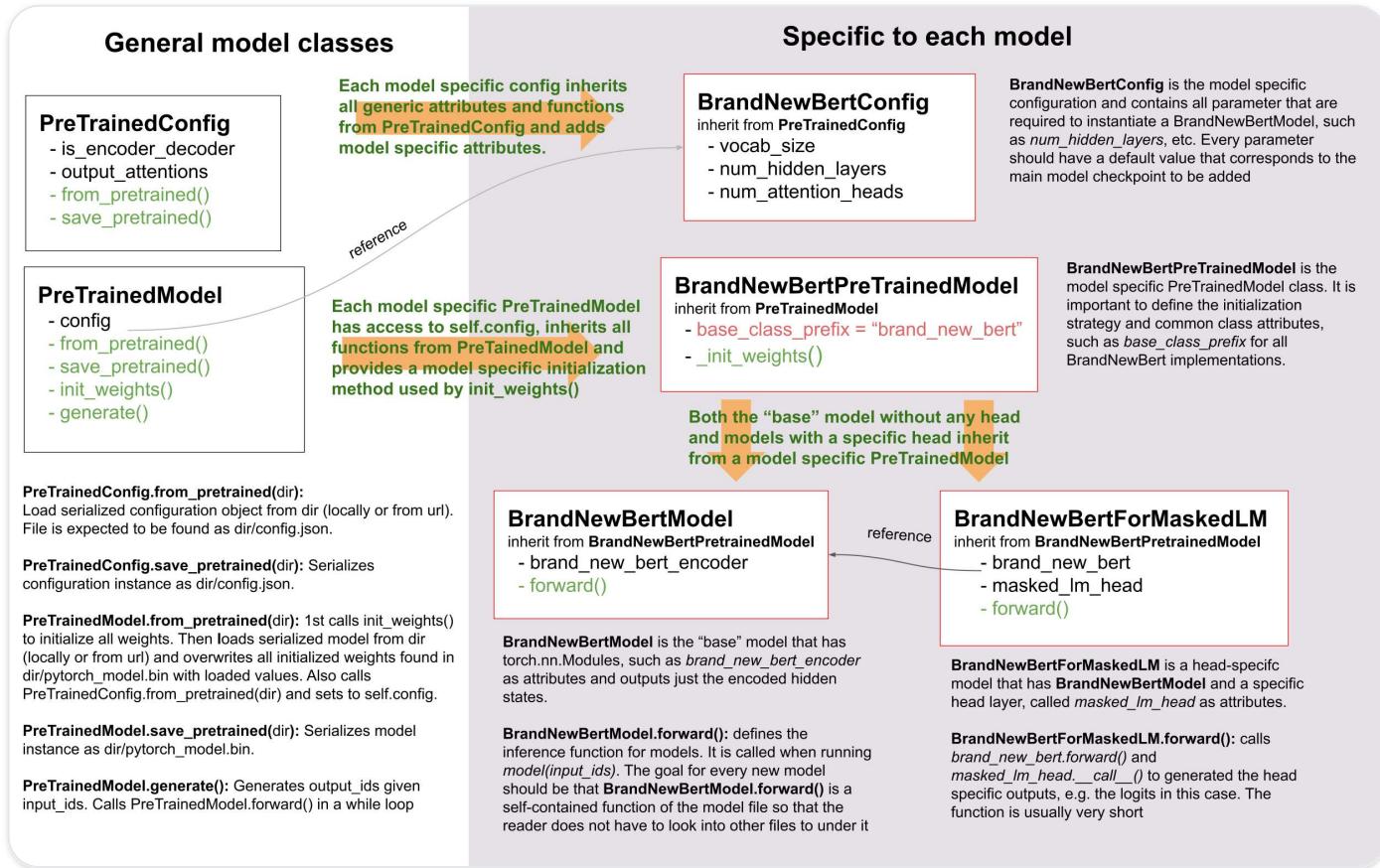
In our opinion, the library's code is not just a means to provide a product, e.g. the ability to use BERT for inference, but also as the very product that we want to improve. Hence, when adding a model, the user is not only the person who will use your model, but also everybody who will read, try to understand, and possibly tweak your code.

With this in mind, let's go a bit deeper into the general library design.

## Overview of models

To successfully add a model, it is important to understand the interaction between your model and its config, [PreTrainedModel](#), and [PretrainedConfig](#). For exemplary purposes, we will call the model to be added to 😊 Transformers `BrandNewBert`.

Let's take a look:



As you can see, we do make use of inheritance in 😊 Transformers, but we keep the level of abstraction to an absolute minimum. There are never more than two levels of abstraction for any model in the library. `BrandNewBertModel` inherits from `BrandNewBertPreTrainedModel` which in turn inherits from `PreTrainedModel` and that's it. As a general rule, we want to make sure that a new model only depends on `PreTrainedModel`. The important functionalities that are automatically provided to every new model are `from_pretrained()` and `save_pretrained()`, which are used for serialization and deserialization. All of the other important functionalities, such as `BrandNewBertModel.forward` should be completely defined in the new `modeling_brand_new_bert.py` script. Next, we want to make sure that a model with a specific head layer, such as `BrandNewBertForMaskedLM` does not inherit from `BrandNewBertModel`, but rather uses `BrandNewBertModel` as a component that can be called in its forward pass to keep the level of abstraction low. Every new model requires a configuration class, called `BrandNewBertConfig`. This configuration is always stored as an attribute in `PreTrainedModel`, and thus can be accessed via the `config` attribute for all classes inheriting from `BrandNewBertPreTrainedModel`:

```
model = BrandNewBertModel.from_pretrained("brandy/brand_new_bert")
model.config # model has access to its config
```

Similar to the model, the configuration inherits basic serialization and deserialization functionalities from [PretrainedConfig](#). Note that the configuration and the model are always serialized into two different formats - the model to a `pytorch_model.bin` file and the configuration to a `config.json` file. Calling `save_pretrained()` will automatically call `save_pretrained()`, so that both model and configuration are saved.

## Code style

When coding your new model, keep in mind that Transformers is an opinionated library and we have a few quirks of our own regarding how code should be written :-)

1. The forward pass of your model should be fully written in the modeling file while being fully independent of other models in the library. If you want to reuse a block from another model, copy the code and paste it with a `# Copied from` comment on top (see [here](#) for a good example and [there](#) for more documentation on Copied from).
2. The code should be fully understandable, even by a non-native English speaker. This means you should pick descriptive variable names and avoid abbreviations. As an example, `activation` is preferred to `act`. One-letter variable names are strongly discouraged unless it's an index in a for loop.
3. More generally we prefer longer explicit code to short magical one.
4. Avoid subclassing `nn.Sequential` in PyTorch but subclass `nn.Module` and write the forward pass, so that anyone using your code can quickly debug it by adding print statements or breaking points.
5. Your function signature should be type-annotated. For the rest, good variable names are way more readable and understandable than type annotations.

## Overview of tokenizers

Not quite ready yet :-( This section will be added soon!

## Step-by-step recipe to add a model to 😊 Transformers

Everyone has different preferences of how to port a model so it can be very helpful for you to take a look at summaries of how other contributors ported models to Hugging Face. Here is a list of community blog posts on how to port a model:

1. [Porting GPT2 Model](#) by [Thomas](#)
2. [Porting WMT19 MT Model](#) by [Stas](#)

From experience, we can tell you that the most important things to keep in mind when adding a model are:

- Don't reinvent the wheel! Most parts of the code you will add for the new 😊 Transformers model already exist somewhere in 😊 Transformers. Take some time to find similar, already existing models and tokenizers you can copy from. `grep` and `rg` are your friends. Note that it might very well happen that your model's tokenizer is based on one model implementation, and your model's modeling code on another one. *E.g.* FSMT's modeling code is based on BART, while FSMT's tokenizer code is based on XLM.
- It's more of an engineering challenge than a scientific challenge. You should spend more time creating an efficient debugging environment rather than trying to understand all theoretical aspects of the model in the paper.
- Ask for help, when you're stuck! Models are the core component of 😊 Transformers so we at Hugging Face are more than happy to help you at every step to add your model. Don't hesitate to ask if you notice you are not making progress.

In the following, we try to give you a general recipe that we found most useful when porting a model to 😊 Transformers.

The following list is a summary of everything that has to be done to add a model and can be used by you as a To-Do List:

- (Optional) Understood the model's theoretical aspects
- Prepared 😊 Transformers dev environment
- Set up debugging environment of the original repository

- Created script that successfully runs the `forward()` pass using the original repository and checkpoint
- Successfully added the model skeleton to 😊 Transformers
- Successfully converted original checkpoint to 😊 Transformers checkpoint
- Successfully ran `forward()` pass in 😊 Transformers that gives identical output to original checkpoint
- Finished model tests in 😊 Transformers
- Successfully added tokenizer in 😊 Transformers
- Run end-to-end integration tests
- Finished docs
- Uploaded model weights to the Hub
- Submitted the pull request
- (Optional) Added a demo notebook

To begin with, we usually recommend starting by getting a good theoretical understanding of `BrandNewBert`. However, if you prefer to understand the theoretical aspects of the model *on-the-job*, then it is totally fine to directly dive into the `BrandNewBert`'s code-base. This option might suit you better if your engineering skills are better than your theoretical skill, if you have trouble understanding `BrandNewBert`'s paper, or if you just enjoy programming much more than reading scientific papers.

## 1. (Optional) Theoretical aspects of `BrandNewBert`

You should take some time to read `BrandNewBert`'s paper, if such descriptive work exists. There might be large sections of the paper that are difficult to understand. If this is the case, this is fine - don't worry! The goal is not to get a deep theoretical understanding of the paper, but to extract the necessary information required to effectively re-implement the model in 😊 Transformers. That being said, you don't have to spend too much time on the theoretical aspects, but rather focus on the practical ones, namely:

- What type of model is `brand_new_bert`? BERT-like encoder-only model? GPT2-like decoder-only model? BART-like encoder-decoder model? Look at the [model summary](#) if you're not familiar with the differences between those.

- What are the applications of *brand\_new\_bert*? Text classification? Text generation? Seq2Seq tasks, e.g., summarization?
- What is the novel feature of the model that makes it different from BERT/GPT-2/BART?
- Which of the already existing 😊 Transformers models is most similar to *brand\_new\_bert*?
- What type of tokenizer is used? A sentencepiece tokenizer? Word piece tokenizer? Is it the same tokenizer as used for BERT or BART?

After you feel like you have gotten a good overview of the architecture of the model, you might want to write to the Hugging Face team with any questions you might have. This might include questions regarding the model's architecture, its attention layer, etc. We will be more than happy to help you.

## 2. Next prepare your environment

1. Fork the repository by clicking on the ‘Fork’ button on the repository’s page. This creates a copy of the code under your GitHub user account.
2. Clone your `transformers` fork to your local disk, and add the base repository as a remote:

```
git clone https://github.com/[your Github handle]/transformers.git
cd transformers
git remote add upstream https://github.com/huggingface/transformers.git
```

3. Set up a development environment, for instance by running the following command:

```
python -m venv .env
source .env/bin/activate
pip install -e ".[dev]"
```

Depending on your OS, and since the number of optional dependencies of Transformers is growing, you might get a failure with this command. If that’s the case make sure to install the Deep Learning framework you are working with (PyTorch, TensorFlow and/or Flax) then do:

```
pip install -e ".[quality]"
```

which should be enough for most use cases. You can then return to the parent directory

```
cd ..
```

4. We recommend adding the PyTorch version of *brand\_new\_bert* to Transformers. To install PyTorch, please follow the instructions on <https://pytorch.org/get-started/locally/>.

**Note:** You don't need to have CUDA installed. Making the new model work on CPU is sufficient.

5. To port *brand\_new\_bert*, you will also need access to its original repository:

```
git clone https://github.com/org_that_created_brand_new_bert_org/brand_new_bert.git
cd brand_new_bert
pip install -e .
```

Now you have set up a development environment to port *brand\_new\_bert* to 😊 Transformers.

### 3.-4. Run a pretrained checkpoint using the original repository

At first, you will work on the original *brand\_new\_bert* repository. Often, the original implementation is very “researchy”. Meaning that documentation might be lacking and the code can be difficult to understand. But this should be exactly your motivation to reimplement *brand\_new\_bert*. At Hugging Face, one of our main goals is to *make people stand on the shoulders of giants* which translates here very well into taking a working model and rewriting it to make it as **accessible, user-friendly, and beautiful** as possible. This is the number-one motivation to reimplement models into 😊 Transformers - trying to make complex new NLP technology accessible to **everybody**.

You should start thereby by diving into the original repository.

Successfully running the official pretrained model in the original repository is often **the most difficult** step. From our experience, it is very important to spend some time getting familiar with the

original code-base. You need to figure out the following:

- Where to find the pretrained weights?
- How to load the pretrained weights into the corresponding model?
- How to run the tokenizer independently from the model?
- Trace one forward pass so that you know which classes and functions are required for a simple forward pass. Usually, you only have to reimplement those functions.
- Be able to locate the important components of the model: Where is the model's class? Are there model sub-classes, e.g. EncoderModel, DecoderModel? Where is the self-attention layer? Are there multiple different attention layers, e.g. *self-attention*, *cross-attention*...?
- How can you debug the model in the original environment of the repo? Do you have to add *print* statements, can you work with an interactive debugger like *ipdb*, or should you use an efficient IDE to debug the model, like PyCharm?

It is very important that before you start the porting process, you can **efficiently** debug code in the original repository! Also, remember that you are working with an open-source library, so do not hesitate to open an issue, or even a pull request in the original repository. The maintainers of this repository are most likely very happy about someone looking into their code!

At this point, it is really up to you which debugging environment and strategy you prefer to use to debug the original model. We strongly advise against setting up a costly GPU environment, but simply work on a CPU both when starting to dive into the original repository and also when starting to write the 😊 Transformers implementation of the model. Only at the very end, when the model has already been successfully ported to 😊 Transformers, one should verify that the model also works as expected on GPU.

In general, there are two possible debugging environments for running the original model

- Jupyter notebooks / google colab
- Local python scripts.

Jupyter notebooks have the advantage that they allow for cell-by-cell execution which can be helpful to better split logical components from one another and to have faster debugging cycles as

intermediate results can be stored. Also, notebooks are often easier to share with other contributors, which might be very helpful if you want to ask the Hugging Face team for help. If you are familiar with Jupyter notebooks, we strongly recommend you work with them.

The obvious disadvantage of Jupyter notebooks is that if you are not used to working with them you will have to spend some time adjusting to the new programming environment and you might not be able to use your known debugging tools anymore, like ipdb.

For each code-base, a good first step is always to load a **small** pretrained checkpoint and to be able to reproduce a single forward pass using a dummy integer vector of input IDs as an input. Such a script could look like this (in pseudocode):

```
model = BrandNewBertModel.load_pretrained_checkpoint("/path/to/checkpoint/")
input_ids = [0, 4, 5, 2, 3, 7, 9] # vector of input ids
original_output = model.predict(input_ids)
```

Next, regarding the debugging strategy, there are generally a few from which to choose from:

- Decompose the original model into many small testable components and run a forward pass on each of those for verification
- Decompose the original model only into the original *tokenizer* and the original *model*, run a forward pass on those, and use intermediate print statements or breakpoints for verification

Again, it is up to you which strategy to choose. Often, one or the other is advantageous depending on the original code base.

If the original code-base allows you to decompose the model into smaller sub-components, e.g. if the original code-base can easily be run in eager mode, it is usually worth the effort to do so. There are some important advantages to taking the more difficult road in the beginning:

- at a later stage when comparing the original model to the Hugging Face implementation, you can verify automatically for each component individually that the corresponding component of the 😊 Transformers implementation matches instead of relying on visual comparison via print statements

- it can give you some rope to decompose the big problem of porting a model into smaller problems of just porting individual components and thus structure your work better
- separating the model into logical meaningful components will help you to get a better overview of the model's design and thus to better understand the model
- at a later stage those component-by-component tests help you to ensure that no regression occurs as you continue changing your code

Lysandre's integration checks for ELECTRA gives a nice example of how this can be done.

However, if the original code-base is very complex or only allows intermediate components to be run in a compiled mode, it might be too time-consuming or even impossible to separate the model into smaller testable sub-components. A good example is T5's MeshTensorFlow library which is very complex and does not offer a simple way to decompose the model into its sub-components. For such libraries, one often relies on verifying print statements.

No matter which strategy you choose, the recommended procedure is often the same that you should start to debug the starting layers first and the ending layers last.

It is recommended that you retrieve the output, either by print statements or sub-component functions, of the following layers in the following order:

1. Retrieve the input IDs passed to the model
2. Retrieve the word embeddings
3. Retrieve the input of the first Transformer layer
4. Retrieve the output of the first Transformer layer
5. Retrieve the output of the following n - 1 Transformer layers
6. Retrieve the output of the whole BrandNewBert Model

Input IDs should thereby consists of an array of integers, e.g. `input_ids = [0, 4, 4, 3, 2, 4, 1, 7, 19]`

The outputs of the following layers often consist of multi-dimensional float arrays and can look like this:

```
[[  
    [-0.1465, -0.6501, 0.1993, ..., 0.1451, 0.3430, 0.6024],  
    [-0.4417, -0.5920, 0.3450, ..., -0.3062, 0.6182, 0.7132],  
    [-0.5009, -0.7122, 0.4548, ..., -0.3662, 0.6091, 0.7648],  
    ...,  
    [-0.5613, -0.6332, 0.4324, ..., -0.3792, 0.7372, 0.9288],  
    [-0.5416, -0.6345, 0.4180, ..., -0.3564, 0.6992, 0.9191],  
    [-0.5334, -0.6403, 0.4271, ..., -0.3339, 0.6533, 0.8694]]],
```

We expect that every model added to 🤗 Transformers passes a couple of integration tests, meaning that the original model and the reimplemented version in 🤗 Transformers have to give the exact same output up to a precision of 0.001! Since it is normal that the exact same model written in different libraries can give a slightly different output depending on the library framework, we accept an error tolerance of 1e-3 (0.001). It is not enough if the model gives nearly the same output, they have to be almost identical. Therefore, you will certainly compare the intermediate outputs of the 🤗 Transformers version multiple times against the intermediate outputs of the original implementation of *brand\_new\_bert* in which case an **efficient** debugging environment of the original repository is absolutely important. Here is some advice to make your debugging environment as efficient as possible.

- Find the best way of debugging intermediate results. Is the original repository written in PyTorch? Then you should probably take the time to write a longer script that decomposes the original model into smaller sub-components to retrieve intermediate values. Is the original repository written in Tensorflow 1? Then you might have to rely on TensorFlow print operations like `tf.print` to output intermediate values. Is the original repository written in Jax? Then make sure that the model is **not jitted** when running the forward pass, e.g. check-out [this link](#).
- Use the smallest pretrained checkpoint you can find. The smaller the checkpoint, the faster your debug cycle becomes. It is not efficient if your pretrained model is so big that your forward pass takes more than 10 seconds. In case only very large checkpoints are available, it might make more sense to create a dummy model in the new environment with randomly initialized weights and save those weights for comparison with the 🤗 Transformers version of your model

- Make sure you are using the easiest way of calling a forward pass in the original repository. Ideally, you want to find the function in the original repository that **only** calls a single forward pass, *i.e.* that is often called `predict`, `evaluate`, `forward` or `__call__`. You don't want to debug a function that calls `forward` multiple times, *e.g.* to generate text, like `autoregressive_sample`, `generate`.
- Try to separate the tokenization from the model's *forward* pass. If the original repository shows examples where you have to input a string, then try to find out where in the forward call the string input is changed to input ids and start from this point. This might mean that you have to possibly write a small script yourself or change the original code so that you can directly input the ids instead of an input string.
- Make sure that the model in your debugging setup is **not** in training mode, which often causes the model to yield random outputs due to multiple dropout layers in the model. Make sure that the forward pass in your debugging environment is **deterministic** so that the dropout layers are not used. Or use `transformers.utils.set_seed` if the old and new implementations are in the same framework.

The following section gives you more specific details/tips on how you can do this for `brand_new_bert`.

### 5.-14. Port BrandNewBert to 😊 Transformers

Next, you can finally start adding new code to 😊 Transformers. Go into the clone of your 😊 Transformers' fork:

```
cd transformers
```

In the special case that you are adding a model whose architecture exactly matches the model architecture of an existing model you only have to add a conversion script as described in [this section](#). In this case, you can just re-use the whole model architecture of the already existing model.

Otherwise, let's start generating a new model. You have two choices here:

- `transformers-cli add-new-model-like` to add a new model like an existing one
- `transformers-cli add-new-model` to add a new model from our template (will look like BERT or Bart depending on the type of model you select)

In both cases, you will be prompted with a questionnaire to fill in the basic information of your model. The second command requires to install `cookiecutter`, you can find more information on it [here](#).

## Open a Pull Request on the main `huggingface/transformers` repo

Before starting to adapt the automatically generated code, now is the time to open a “Work in progress (WIP)” pull request, e.g. “[WIP] Add `brand_new_bert`”, in 😊 Transformers so that you and the Hugging Face team can work side-by-side on integrating the model into 😊 Transformers.

You should do the following:

1. Create a branch with a descriptive name from your main branch

```
git checkout -b add_brand_new_bert
```

2. Commit the automatically generated code:

```
git add .
git commit
```

3. Fetch and rebase to current main

```
git fetch upstream
git rebase upstream/main
```

4. Push the changes to your account using:

```
git push -u origin a-descriptive-name-for-my-changes
```

5. Once you are satisfied, go to the webpage of your fork on GitHub. Click on “Pull request”. Make sure to add the GitHub handle of some members of the Hugging Face team as reviewers, so that the Hugging Face team gets notified for future changes.
6. Change the PR into a draft by clicking on “Convert to draft” on the right of the GitHub pull request web page.

In the following, whenever you have made some progress, don’t forget to commit your work and push it to your account so that it shows in the pull request. Additionally, you should make sure to update your work with the current main from time to time by doing:

```
git fetch upstream  
git merge upstream/main
```

In general, all questions you might have regarding the model or your implementation should be asked in your PR and discussed/solved in the PR. This way, the Hugging Face team will always be notified when you are committing new code or if you have a question. It is often very helpful to point the Hugging Face team to your added code so that the Hugging Face team can efficiently understand your problem or question.

To do so, you can go to the “Files changed” tab where you see all of your changes, go to a line regarding which you want to ask a question, and click on the “+” symbol to add a comment. Whenever a question or problem has been solved, you can click on the “Resolve” button of the created comment.

In the same way, the Hugging Face team will open comments when reviewing your code. We recommend asking most questions on GitHub on your PR. For some very general questions that are not very useful for the public, feel free to ping the Hugging Face team by Slack or email.

## 5. Adapt the generated models code for brand\_new\_bert

At first, we will focus only on the model itself and not care about the tokenizer. All the relevant code should be found in the generated files

`src/transformers/models/brand_new_bert/modeling_brand_new_bert.py` and  
`src/transformers/models/brand_new_bert/configuration_brand_new_bert.py`.

Now you can finally start coding :). The generated code in `src/transformers/models/brand_new_bert/modeling_brand_new_bert.py` will either have the same architecture as BERT if it's an encoder-only model or BART if it's an encoder-decoder model. At this point, you should remind yourself what you've learned in the beginning about the theoretical aspects of the model: *How is the model different from BERT or BART?*. Implement those changes which often means changing the *self-attention* layer, the order of the normalization layer, etc... Again, it is often useful to look at the similar architecture of already existing models in Transformers to get a better feeling of how your model should be implemented.

**Note** that at this point, you don't have to be very sure that your code is fully correct or clean. Rather, it is advised to add a first *unclean*, copy-pasted version of the original code to `src/transformers/models/brand_new_bert/modeling_brand_new_bert.py` until you feel like all the necessary code is added. From our experience, it is much more efficient to quickly add a first version of the required code and improve/correct the code iteratively with the conversion script as described in the next section. The only thing that has to work at this point is that you can instantiate the 😊 Transformers implementation of *brand\_new\_bert*, i.e. the following command should work:

```
from transformers import BrandNewBertModel, BrandNewBertConfig  
  
model = BrandNewBertModel(BrandNewBertConfig())
```

The above command will create a model according to the default parameters as defined in `BrandNewBertConfig()` with random weights, thus making sure that the `init()` methods of all components works.

Note that all random initialization should happen in the `_init_weights` method of your `BrandnewBertPreTrainedModel` class. It should initialize all leaf modules depending on the variables of the config. Here is an example with the BERT `_init_weights` method:

```
def _init_weights(self, module):  
    """Initialize the weights"""  
    if isinstance(module, nn.Linear):  
        module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)  
        if module.bias is not None:
```

```

        module.bias.data.zero_()
    elif isinstance(module, nn.Embedding):
        module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if module.padding_idx is not None:
            module.weight.data[module.padding_idx].zero_()
    elif isinstance(module, nn.LayerNorm):
        module.bias.data.zero_()
        module.weight.data.fill_(1.0)

```

You can have some more custom schemes if you need a special initialization for some modules. For instance, in `Wav2Vec2ForPreTraining`, the last two linear layers need to have the initialization of the regular PyTorch `nn.Linear` but all the other ones should use an initialization as above. This is coded like this:

```

def _init_weights(self, module):
    """Initialize the weights"""
    if isinstance(module, Wav2Vec2ForPreTraining):
        module.project_hid.reset_parameters()
        module.project_q.reset_parameters()
        module.project_hid._is_hf_initialized = True
        module.project_q._is_hf_initialized = True
    elif isinstance(module, nn.Linear):
        module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if module.bias is not None:
            module.bias.data.zero_()

```

The `_is_hf_initialized` flag is internally used to make sure we only initialize a submodule once. By setting it to `True` for `module.project_q` and `module.project_hid`, we make sure the custom initialization we did is not overridden later on, the `_init_weights` function won't be applied to them.

## 6. Write a conversion script

Next, you should write a conversion script that lets you convert the checkpoint you used to debug `brand_new_bert` in the original repository to a checkpoint compatible with your just created 😊 Transformers implementation of `brand_new_bert`. It is not advised to write the conversion script from scratch, but rather to look through already existing conversion scripts in 😊 Transformers for

one that has been used to convert a similar model that was written in the same framework as `brand_new_bert`. Usually, it is enough to copy an already existing conversion script and slightly adapt it for your use case. Don't hesitate to ask the Hugging Face team to point you to a similar already existing conversion script for your model.

- If you are porting a model from TensorFlow to PyTorch, a good starting point might be BERT's conversion script [here](#)
- If you are porting a model from PyTorch to PyTorch, a good starting point might be BART's conversion script [here](#)

In the following, we'll quickly explain how PyTorch models store layer weights and define layer names. In PyTorch, the name of a layer is defined by the name of the class attribute you give the layer. Let's define a dummy model in PyTorch, called `SimpleModel` as follows:

```
from torch import nn

class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.dense = nn.Linear(10, 10)
        self.intermediate = nn.Linear(10, 10)
        self.layer_norm = nn.LayerNorm(10)
```

Now we can create an instance of this model definition which will fill all weights: `dense`, `intermediate`, `layer_norm` with random weights. We can print the model to see its architecture

```
model = SimpleModel()

print(model)
```

This will print out the following:

```
SimpleModel(
  (dense): Linear(in_features=10, out_features=10, bias=True)
  (intermediate): Linear(in_features=10, out_features=10, bias=True)
```

```
(layer_norm): LayerNorm((10,), eps=1e-05, elementwise_affine=True)
)
```

We can see that the layer names are defined by the name of the class attribute in PyTorch. You can print out the weight values of a specific layer:

```
print(model.dense.weight.data)
```

to see that the weights were randomly initialized

```
tensor([[-0.0818,  0.2207, -0.0749, -0.0030,  0.0045, -0.1569, -0.1598,  0.0212,
        -0.2077,  0.2157],
       [ 0.1044,  0.0201,  0.0990,  0.2482,  0.3116,  0.2509,  0.2866, -0.2190,
        0.2166, -0.0212],
      [-0.2000,  0.1107, -0.1999, -0.3119,  0.1559,  0.0993,  0.1776, -0.1950,
       -0.1023, -0.0447],
      [-0.0888, -0.1092,  0.2281,  0.0336,  0.1817, -0.0115,  0.2096,  0.1415,
       -0.1876, -0.2467],
      [ 0.2208, -0.2352, -0.1426, -0.2636, -0.2889, -0.2061, -0.2849, -0.0465,
        0.2577,  0.0402],
      [ 0.1502,  0.2465,  0.2566,  0.0693,  0.2352, -0.0530,  0.1859, -0.0604,
        0.2132,  0.1680],
      [ 0.1733, -0.2407, -0.1721,  0.1484,  0.0358, -0.0633, -0.0721, -0.0090,
        0.2707, -0.2509],
      [-0.1173,  0.1561,  0.2945,  0.0595, -0.1996,  0.2988, -0.0802,  0.0407,
        0.1829, -0.1568],
      [-0.1164, -0.2228, -0.0403,  0.0428,  0.1339,  0.0047,  0.1967,  0.2923,
        0.0333, -0.0536],
      [-0.1492, -0.1616,  0.1057,  0.1950, -0.2807, -0.2710, -0.1586,  0.0739,
        0.2220,  0.2358]]).
```

In the conversion script, you should fill those randomly initialized weights with the exact weights of the corresponding layer in the checkpoint. *E.g.*

```
# retrieve matching layer weights, e.g. by
# recursive algorithm
layer_name = "dense"
pretrained_weight = array_of_dense_layer
```

```
model_pointer = getattr(model, "dense")  
  
model_pointer.weight.data = torch.from_numpy(pretrained_weight)
```

While doing so, you must verify that each randomly initialized weight of your PyTorch model and its corresponding pretrained checkpoint weight exactly match in both **shape and name**. To do so, it is **necessary** to add assert statements for the shape and print out the names of the checkpoints weights. E.g. you should add statements like:

```
assert (  
    model_pointer.weight.shape == pretrained_weight.shape  
, f"Pointer shape of random weight {model_pointer.shape} and array shape of checkpoint w
```

Besides, you should also print out the names of both weights to make sure they match, e.g.

```
logger.info(f"Initialize PyTorch weight {layer_name} from {pretrained_weight.name}")
```

If either the shape or the name doesn't match, you probably assigned the wrong checkpoint weight to a randomly initialized layer of the 😊 Transformers implementation.

An incorrect shape is most likely due to an incorrect setting of the config parameters in `BrandNewBertConfig()` that do not exactly match those that were used for the checkpoint you want to convert. However, it could also be that PyTorch's implementation of a layer requires the weight to be transposed beforehand.

Finally, you should also check that **all** required weights are initialized and print out all checkpoint weights that were not used for initialization to make sure the model is correctly converted. It is completely normal, that the conversion trials fail with either a wrong shape statement or a wrong name assignment. This is most likely because either you used incorrect parameters in `BrandNewBertConfig()`, have a wrong architecture in the 😊 Transformers implementation, you have a bug in the `init()` functions of one of the components of the 😊 Transformers implementation or you need to transpose one of the checkpoint weights.

This step should be iterated with the previous step until all weights of the checkpoint are correctly loaded in the Transformers model. Having correctly loaded the checkpoint into the 😊 Transformers implementation, you can then save the model under a folder of your choice `/path/to/converted/checkpoint/folder` that should then contain both a `pytorch_model.bin` file and a `config.json` file:

```
model.save_pretrained("/path/to/converted/checkpoint/folder")
```

## 7. Implement the forward pass

Having managed to correctly load the pretrained weights into the 😊 Transformers implementation, you should now make sure that the forward pass is correctly implemented. In [Get familiar with the original repository](#), you have already created a script that runs a forward pass of the model using the original repository. Now you should write an analogous script using the 😊 Transformers implementation instead of the original one. It should look as follows:

```
model = BrandNewBertModel.from_pretrained("/path/to/converted/checkpoint/folder")
input_ids = [0, 4, 4, 3, 2, 4, 1, 7, 19]
output = model(input_ids).last_hidden_states
```

It is very likely that the 😊 Transformers implementation and the original model implementation don't give the exact same output the very first time or that the forward pass throws an error. Don't be disappointed - it's expected! First, you should make sure that the forward pass doesn't throw any errors. It often happens that the wrong dimensions are used leading to a *Dimensionality mismatch* error or that the wrong data type object is used, e.g. `torch.long` instead of `torch.float32`. Don't hesitate to ask the Hugging Face team for help, if you don't manage to solve certain errors.

The final part to make sure the 😊 Transformers implementation works correctly is to ensure that the outputs are equivalent to a precision of `1e-3`. First, you should ensure that the output shapes are identical, i.e. `outputs.shape` should yield the same value for the script of the 😊 Transformers implementation and the original implementation. Next, you should make sure that the output values are identical as well. This one of the most difficult parts of adding a new model. Common mistakes why the outputs are not identical are:

- Some layers were not added, i.e. an *activation* layer was not added, or the residual connection was forgotten
- The word embedding matrix was not tied
- The wrong positional embeddings are used because the original implementation uses an offset
- Dropout is applied during the forward pass. To fix this make sure `model.training` is `False` and that no dropout layer is falsely activated during the forward pass, i.e. pass `self.training` to [PyTorch's functional dropout](#)

The best way to fix the problem is usually to look at the forward pass of the original implementation and the 😊 Transformers implementation side-by-side and check if there are any differences. Ideally, you should debug/print out intermediate outputs of both implementations of the forward pass to find the exact position in the network where the 😊 Transformers implementation shows a different output than the original implementation. First, make sure that the hard-coded `input_ids` in both scripts are identical. Next, verify that the outputs of the first transformation of the `input_ids` (usually the word embeddings) are identical. And then work your way up to the very last layer of the network. At some point, you will notice a difference between the two implementations, which should point you to the bug in the 😊 Transformers implementation. From our experience, a simple and efficient way is to add many print statements in both the original implementation and 😊 Transformers implementation, at the same positions in the network respectively, and to successively remove print statements showing the same values for intermediate presentations.

When you're confident that both implementations yield the same output, verify the outputs with `torch.allclose(original_output, output, atol=1e-3)`, you're done with the most difficult part! Congratulations - the work left to be done should be a cakewalk 😊.

## 8. Adding all necessary model tests

At this point, you have successfully added a new model. However, it is very much possible that the model does not yet fully comply with the required design. To make sure, the implementation is fully compatible with 😊 Transformers, all common tests should pass. The Cookiecutter should have automatically added a test file for your model, probably under the same

`tests/models/brand_new_bert/test_modeling_brand_new_bert.py`. Run this test file to verify that all common tests pass:

```
pytest tests/models/brand_new_bert/test_modeling_brand_new_bert.py
```

Having fixed all common tests, it is now crucial to ensure that all the nice work you have done is well tested, so that

- a) The community can easily understand your work by looking at specific tests of *brand\_new\_bert*
- b) Future changes to your model will not break any important feature of the model.

At first, integration tests should be added. Those integration tests essentially do the same as the debugging scripts you used earlier to implement the model to 😊 Transformers. A template of those model tests has already been added by the Cookiecutter, called `BrandNewBertModelIntegrationTests` and only needs to be filled out by you. To ensure that those tests are passing, run

```
RUN_SLOW=1 pytest -sv tests/models/brand_new_bert/test_modeling_brand_new_bert.py::Brandn
```

In case you are using Windows, you should replace `RUN_SLOW=1` with `SET RUN_SLOW=1`

Second, all features that are special to *brand\_new\_bert* should be tested additionally in a separate test under `BrandNewBertModelTester`/`BrandNewBertModelTest`. This part is often forgotten but is extremely useful in two ways:

- It helps to transfer the knowledge you have acquired during the model addition to the community by showing how the special features of *brand\_new\_bert* should work.
- Future contributors can quickly test changes to the model by running those special tests.

## 9. Implement the tokenizer

Next, we should add the tokenizer of *brand\_new\_bert*. Usually, the tokenizer is equivalent to or very similar to an already existing tokenizer of 😊 Transformers.

It is very important to find/extract the original tokenizer file and to manage to load this file into the 😊 Transformers' implementation of the tokenizer.

To ensure that the tokenizer works correctly, it is recommended to first create a script in the original repository that inputs a string and returns the `input\_ids`. It could look similar to this (in pseudo-code):

```
input_str = "This is a long example input string containing special characters .$?-," numc  
model = BrandNewBertModel.load_pretrained_checkpoint("/path/to/checkpoint/")  
input_ids = model.tokenize(input_str)
```

You might have to take a deeper look again into the original repository to find the correct tokenizer function or you might even have to do changes to your clone of the original repository to only output the `input_ids`. Having written a functional tokenization script that uses the original repository, an analogous script for 😊 Transformers should be created. It should look similar to this:

```
from transformers import BrandNewBertTokenizer  
  
input_str = "This is a long example input string containing special characters .$?-," numb  
tokenizer = BrandNewBertTokenizer.from_pretrained("/path/to/tokenizer/folder/")  
  
input_ids = tokenizer(input_str).input_ids
```

When both `input_ids` yield the same values, as a final step a tokenizer test file should also be added.

Analogous to the modeling test files of *brand\_new\_bert*, the tokenization test files of *brand\_new\_bert* should contain a couple of hard-coded integration tests.

## 10. Run End-to-end integration tests

Having added the tokenizer, you should also add a couple of end-to-end integration tests using both the model and the tokenizer to

`tests/models/brand_new_bert/test_modeling_brand_new_bert.py` in  Transformers. Such a test should show on a meaningful text-to-text sample that the  Transformers implementation works as expected. A meaningful text-to-text sample can include e.g. a source-to-target-translation pair, an article-to-summary pair, a question-to-answer pair, etc... If none of the ported checkpoints has been fine-tuned on a downstream task it is enough to simply rely on the model tests. In a final step to ensure that the model is fully functional, it is advised that you also run all tests on GPU. It can happen that you forgot to add some `.to(self.device)` statements to internal tensors of the model, which in such a test would show in an error. In case you have no access to a GPU, the Hugging Face team can take care of running those tests for you.

## 11. Add Docstring

Now, all the necessary functionality for `brand_new_bert` is added - you're almost done! The only thing left to add is a nice docstring and a doc page. The Cookiecutter should have added a template file called `docs/source/model_doc/brand_new_bert.md` that you should fill out. Users of your model will usually first look at this page before using your model. Hence, the documentation must be understandable and concise. It is very useful for the community to add some *Tips* to show how the model should be used. Don't hesitate to ping the Hugging Face team regarding the docstrings.

Next, make sure that the docstring added to

`src/transformers/models/brand_new_bert/modeling_brand_new_bert.py` is correct and included all necessary inputs and outputs. We have a detailed guide about writing documentation and our docstring format [here](#). It is always good to remind oneself that documentation should be treated at least as carefully as the code in  Transformers since the documentation is usually the first contact point of the community with the model.

## Code refactor

Great, now you have added all the necessary code for `brand_new_bert`. At this point, you should correct some potential incorrect code style by running:

```
make style
```

and verify that your coding style passes the quality check:

```
make quality
```

There are a couple of other very strict design tests in 😊 Transformers that might still be failing, which shows up in the tests of your pull request. This is often because of some missing information in the docstring or some incorrect naming. The Hugging Face team will surely help you if you're stuck here.

Lastly, it is always a good idea to refactor one's code after having ensured that the code works correctly. With all tests passing, now it's a good time to go over the added code again and do some refactoring.

You have now finished the coding part, congratulations! 🎉 You are Awesome! 😊

## 12. Upload the models to the model hub

In this final part, you should convert and upload all checkpoints to the model hub and add a model card for each uploaded model checkpoint. You can get familiar with the hub functionalities by reading our [Model sharing and uploading Page](#). You should work alongside the Hugging Face team here to decide on a fitting name for each checkpoint and to get the required access rights to be able to upload the model under the author's organization of *brand\_new\_bert*. The `push_to_hub` method, present in all models in `transformers`, is a quick and efficient way to push your checkpoint to the hub. A little snippet is pasted below:

```
brand_new_bert.push_to_hub("brand_new_bert")
# Uncomment the following line to push to an organization.
# brand_new_bert.push_to_hub("<organization>/brand_new_bert")
```

It is worth spending some time to create fitting model cards for each checkpoint. The model cards should highlight the specific characteristics of this particular checkpoint, e.g. On which dataset was

the checkpoint pretrained/fine-tuned on? On what down-stream task should the model be used? And also include some code on how to correctly use the model.

### 13. (Optional) Add notebook

It is very helpful to add a notebook that showcases in-detail how *brand\_new\_bert* can be used for inference and/or fine-tuned on a downstream task. This is not mandatory to merge your PR, but very useful for the community.

### 14. Submit your finished PR

You're done programming now and can move to the last step, which is getting your PR merged into main. Usually, the Hugging Face team should have helped you already at this point, but it is worth taking some time to give your finished PR a nice description and eventually add comments to your code, if you want to point out certain design choices to your reviewer.

## Share your work!!

Now, it's time to get some credit from the community for your work! Having completed a model addition is a major contribution to Transformers and the whole NLP community. Your code and the ported pre-trained models will certainly be used by hundreds and possibly even thousands of developers and researchers. You should be proud of your work and share your achievements with the community.

You have made another model that is super easy to access for everyone in the community! 

← How to contribute to transformers?

How to convert a  Transformers model to TensorFlow? →