# OpenDelta

***Release 0.3.1***

**THUNLP OpenDelta Team**

**Oct 17, 2022**

# GETTING STARTED

OpenDelta is a **Plug-and-play** Library of the parameter-efficient fine-tuning (*delta-tuning*) technology for pre-trained models.

CHAPTER

# ONE

# ESSENTIAL ADVANTAGES:

- Clean: No need to edit the backbone PTM's codes.
- Simple: Migrating from full-model tuning to delta-tuning needs as little as 3 lines of codes.
- Sustainable: Most evolution in external library doesn't require a new OpenDelta.
- Extendable: Various PTMs can share the same delta-tuning codes.
- Flexible: Able to apply delta-tuning to (almost) any position of the PTMs.

## 1.1 What is Delta-tuning and Why OpenDelta?

**What is Delta?**

As Pre-trained language models (PLMs) have become the fundamental infrastructure on many NLP tasks and benchmarks, it is becoming increasingly clear from recent research that **larger models tend to lead to better performance**. However, large-scale PLMs also bring prohibitive adaptation costs when fine-tuning all the parameters of a model and retaining separate instances for different tasks.

**Parameter-efficient model stimulation methods** thus have attracted researchers' eyes, which only tune a small fraction of model parameter while achieving comparable or even better performance than full-model fine-tuning, dubbed as "Delta-tuning".

**Delta** thus means a small fraction $\Delta\Theta$ of parameters besides the pretrained models $\Theta_0$.

$$\Theta \sim \Theta_0(\text{frozen}) + \Delta\Theta(\text{tunable})$$

This open-source project implement several delta-tuning methods, which allows researchers and engineers to quickly migrate their codes from full-model tuning to delta-tuning without replace the backend (the implementation of the backbone PLM).

### 1.1.1 Why OpenDelta?


- Clean: No need to edit the backbone PTM's codes.
- Simple: Migrating from full-model tuning to delta-tuning needs as little as 3 lines of codes.
- Sustainable: Most evolution in external library doesn't require a new OpenDelta.
- Extendable: Various PTMs can share the same delta-tuning codes.
- Flexible: Able to apply delta-tuning to (almost) any position of the PTMs.

### 1.1.2 Delta-tuning papers

## 1.2 Installation

OpenDelta is tested on on Python 3.8 and Pytorch 1.9.

```
pip install opendelta
```

or from the source

```
git clone
cd OpenDelta
python setup.py install
```

If you want to do some modifications on the code for your research, run

```
git clone
cd OpenDelta
python setup.py develop
```

## 1.3 Basic Usage

Now we introduce the general pipeline to migrate your full-model tuning scripts to a delta tuning one.

### 1.3.1 STEP 1: Load the pretrained models

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained("facebook/bart-base") #␣
↪suppose we load BART
```

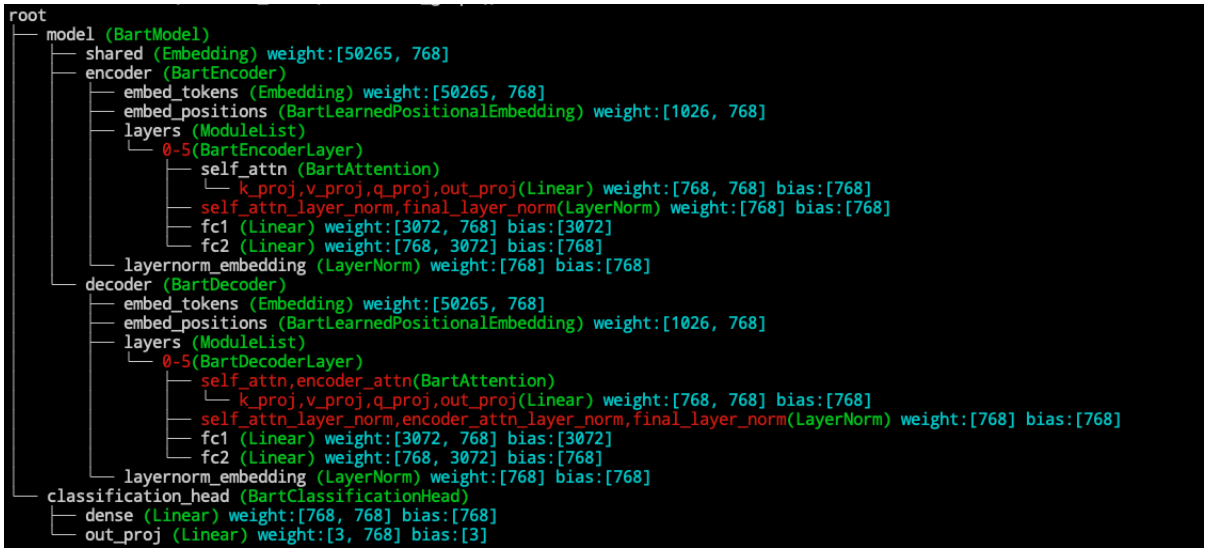
### 1.3.2 STEP 2: Add delta modules

We provide two alternatives to add the delta modules.

#### 2.1 Modification based on visualization

Suppose we want to make the feedforward layer of each block as our *modification target module*, We should first know what is the name of the feedforward layer in the BART model by visualization.  *For more about visualization, see* Visualization.

```
from opendelta import Visualization
Visualization(model).structure_graph()
```




We can see from the structure graph that the feed forward layer in Bart is called `model.encoder.layers.$.fc1` and `model.encoder.layers.$.fc2`, where `$` represent a number from 0-5. Since we want to apply adapter after *all* the feed forward layers, we specify the `modified_modules=['fc2']`, which is the common suffix for feed forward layers.

 *For details about the name based addressing, see* Name-based submodule addressing

Other configurations, such as the `bottleneck_dim` in Adapter, can be passed as key word arguments.

```
from opendelta import AdapterModel
delta_model = AdapterModel(backbone_model=model, modified_modules=['fc2'], bottleneck_
↪dim=12)
delta_model.log() # This will visualize the backbone after modification and other␣
↪information.
```

### 2.2 Use the default modification.

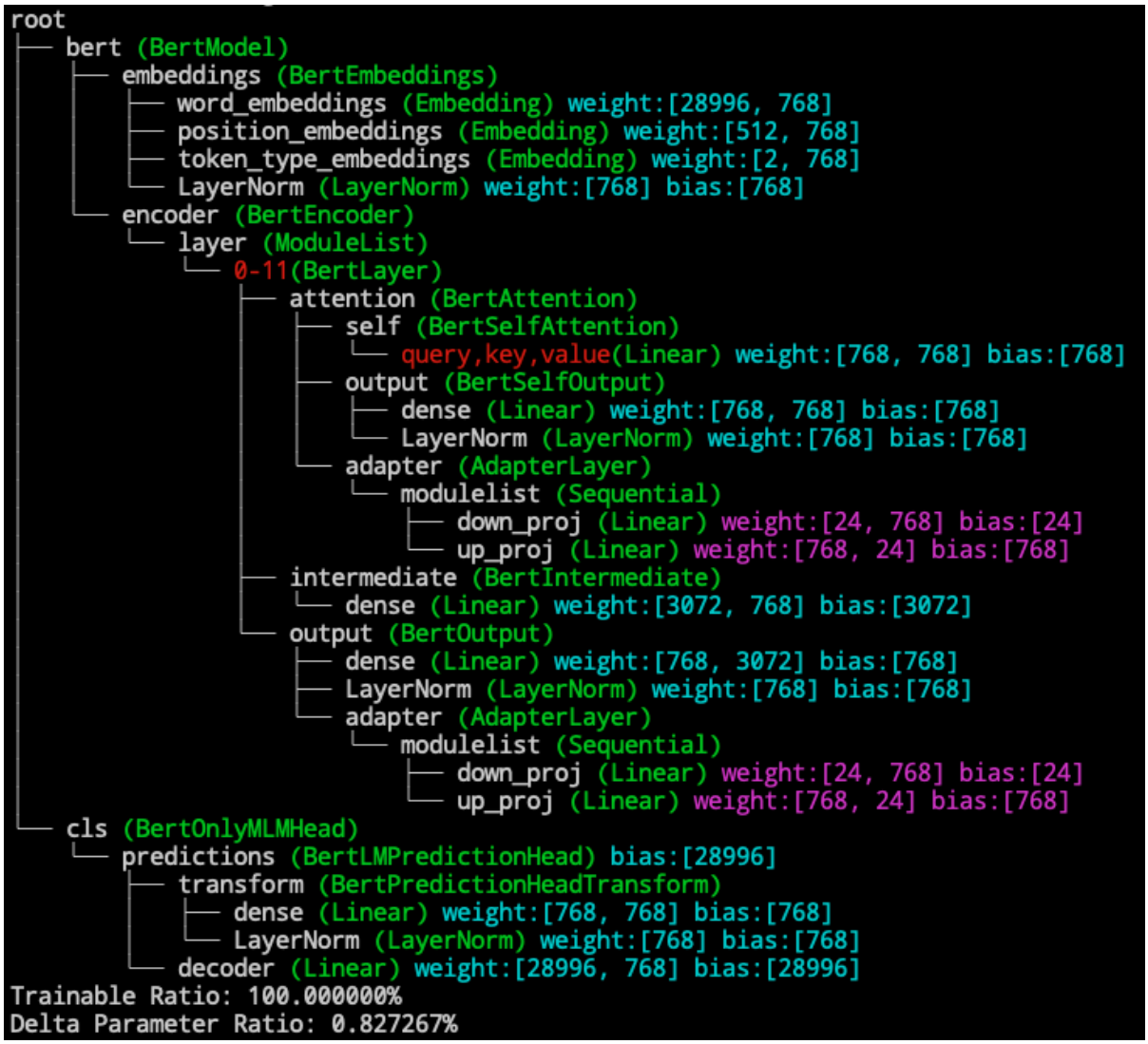
We also provide the default modifications of each delta methods for some commonly used PTMs (e.g., BERT, RoBERTA, DistilBERT, T5, GPT2), so the users don't need to specify the submodules to modify.

The default modifications is achieved by mapping a name of a submodule to it's name on a common transformer structure.  *For details about the common structure mapping, see* Common Structure Mapping

```
# a seperate example using BERT.
from transformers import BertForMaskedLM
from opendelta import AdapterModel
model = BertForMaskedLM.from_pretrained("bert-base-cased")
delta_model = AdapterModel(model) # This will apply adapter to the self-attn and feed-
↪forward layer.
delta_model.log()
```

**Delta model vs Backbone model**

The delta_model **CAN NOT** be used alone, and its *forward* is canceled. The training pipeline should be conducted on


root
bert (BertModel)
embeddings (BertEmbeddings)
word_embeddings (Embedding) weight:[28996, 768]
position_embeddings (Embedding) weight:[512, 768]
token_type_embeddings (Embedding) weight:[2, 768]
LayerNorm (LayerNorm) weight:[768] bias:[768]
encoder (BertEncoder)
layer (ModuleList)
0-11(BertLayer)
attention (BertAttention)
self (BertSelfAttention)
query,key,value(Linear) weight:[768, 768] bias:[768]
output (BertSelfOutput)
dense (Linear) weight:[768, 768] bias:[768]
LayerNorm (LayerNorm) weight:[768] bias:[768]
adapter (AdapterLayer)
modulelist (Sequential)
down_proj (Linear) weight:[24, 768] bias:[24]
up_proj (Linear) weight:[768, 24] bias:[768]
intermediate (BertIntermediate)
dense (Linear) weight:[3072, 768] bias:[3072]
output (BertOutput)
dense (Linear) weight:[768, 3072] bias:[768]
LayerNorm (LayerNorm) weight:[768] bias:[768]
adapter (AdapterLayer)
modulelist (Sequential)
down_proj (Linear) weight:[24, 768] bias:[24]
up_proj (Linear) weight:[768, 24] bias:[768]
cls (BertOnlyMLMHead)
predictions (BertLMPredictionHead) bias:[28996]
transform (BertPredictionHeadTransform)
dense (Linear) weight:[768, 768] bias:[768]
LayerNorm (LayerNorm) weight:[768] bias:[768]
decoder (Linear) weight:[28996, 768] bias:[28996]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 0.827267%

the backbone model (In the above example, its the `model`).

**Try different positions**

OpenDelta provide the flexibility to add delta to different positions on the backbone model. For example, If you want to move the adapter in the above example after the layer norm of the feed forward layer. The code should be changed into

```
# continue with the BART example, but not used later.
delta_model = AdapterModel(backbone_model=model, modified_modules=['final_layer_norm'],␣
↪bottleneck_dim=12)
```

The performance may vary due to positional differences, but there is no academic guarantee that one will outperform the other.

**Favored Configurations**

Feel confused about the flexibility that OpenDelta brings? Currently you can refer to the papers for their configuration. And We will add *Favored Configurations* soon.

### 1.3.3 STEP 3: Freezing parameters

The main part of the backbone model is not automatically frozen (We may add the option in future). To freeze the main part of the backbone model except the trainable parts (usually the delta paramters), use *freeze_module* method. The `exclude` field obeys the same name-based addressing rules as the `modified_modules` field.

```
# continue with the BART example
delta_model.freeze_module(exclude=["deltas", "layernorm_embedding"], set_state_dict=True)
delta_model.log()
```

The `set_state_dict=True` will tell the method to change the `state_dict` of the `backbone_model` to maintaining only the trainable parts.

### 1.3.4 STEP 4: Normal training pipeline

The **model** then can be trained in traditional training scripts. Two things should be noticed:

**Note**

1. No need to change the optimizer, since the optimizer will only calculated and store gradient for those parameters with `requires_grad=True`, and the `requires_grad` attribute has been changed during the call to *freeze_module* method.
2. `model.eval()` or `model.train()` should be used when needed to set dropout, etc. Delta model doesn't touch those configuration.

```
root
├── model (BartModel)
│   ├── shared (Embedding) weight:[50265, 768]
│   ├── encoder (BartEncoder)
│   │   ├── embed_tokens (Embedding) weight:[50265, 768]
│   │   ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│   │   ├── layers (ModuleList)
│   │   │   └── 0-5(BartEncoderLayer)
│   │   │       ├── self_attn (BartAttention)
│   │   │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│   │   │       ├── self_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│   │   │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│   │   │       └── fc2 (Linear) weight:[768, 3072] bias:[768]
│   │   │           └── adapter (AdapterLayer)
│   │   │               └── modulelist (Sequential)
│   │   │                   ├── down_proj (Linear) weight:[12, 768] bias:[12]
│   │   │                   └── up_proj (Linear) weight:[768, 12] bias:[768]
│   │   └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
│   └── decoder (BartDecoder)
│       ├── embed_tokens (Embedding) weight:[50265, 768]
│       ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│       ├── layers (ModuleList)
│       │   └── 0-5(BartDecoderLayer)
│       │       ├── self_attn,encoder_attn(BartAttention)
│       │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│       │       ├── self_attn_layer_norm,encoder_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│       │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│       │       └── fc2 (Linear) weight:[768, 3072] bias:[768]
│       │           └── adapter (AdapterLayer)
│       │               └── modulelist (Sequential)
│       │                   ├── down_proj (Linear) weight:[12, 768] bias:[12]
│       │                   └── up_proj (Linear) weight:[768, 12] bias:[768]
│       └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
└── classification_head (BartClassificationHead)
    ├── dense (Linear) weight:[768, 768] bias:[768]
    └── out_proj (Linear) weight:[3, 768] bias:[3]
Trainable Ratio: 0.166578%
```

### 1.3.5 STEP 5: Saved/Share the Delta Model

*see* Save a delta model to local, or share with the community.

## 1.4 Name-based Addressing

Named based addressing is what set OpenDelta apart from other packages and provide the possibility to be used to a broader range of models (even emerging ones).

### 1.4.1 Name of a submodule.

We locate the submodules that we want to apply a delta layer via name-based addressing.

In pytorch fashion, a submodule can be accessed from a root model via 'dot' addressing. For example, we define a toy language model

```
import torch.nn as nn
class MyNet1(nn.Module):
    def __init__(self,):
        super().__init__()
        self.name_a = nn.Linear(5,5)
    def forward(self, hiddens):
        return self.name_a(hiddens)
```

(continues on next page)

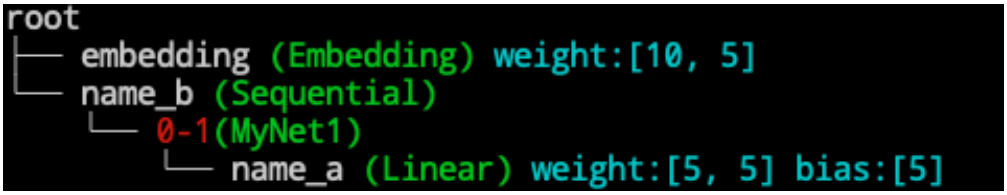
(continued from previous page)

```
class MyNet2(nn.Module):
    def __init__(self,):
        super().__init__()
        self.embedding = nn.Embedding(10,5)
        self.name_b = nn.Sequential(MyNet1(), MyNet1())
    def forward(self, input_ids):
        hiddens = self.embedding(input_ids)
        return self.name_b(hiddens)

root = MyNet2()
print(root.name_b[0].name_a)
# Linear(in_features=5, out_features=5, bias=True)
```

We can visualize the model (For details, see *visualization*)

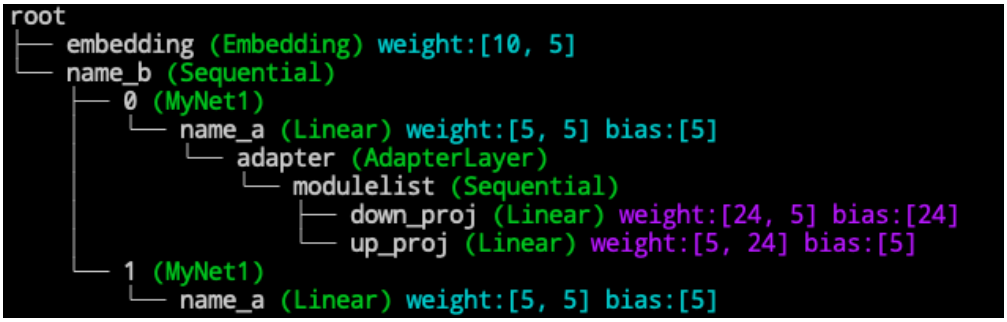
```
from opendelta import Visualization
Visualization(root).structure_graph()
```




In this case, string "name_b.0.name_a" will be the name to address the submodule from the root model.

Thus when applying a delta model to this toy net.

```
from opendelta import AdapterModel
AdapterModel(backbone_model=root, modified_modules=['name_b.0.name_a'])
Visualization(root).structure_graph()
```



### 1.4.2 Target modules.

For different delta methods, the operation for the modification target is different.

- Adapter based method: Insert at the target module's forward function.
- BitFit: Add bias to all allowed position of the target module.
- Lora: Substitute the all the linear layers of the target module with Lora.Linear.
- Prefix Tuning: the target module must be an attention module.

---

**Auto Searching**

We are working on unifying operations to automatically search within a given module for its submodules that can be applied using a specific delta method.

---

### 1.4.3 Makes addressing easier.

Handcrafting the full names of submodules can be frustrating. We made some simplifications

1. **End-matching** Rules.

   OpenDelta will take every modules that **ends with** the provided name suffix as the modification *target module*.

   ---

   **Example**

   Taking DistilBert with an classifier on top as an example:

   - set to `["0.attention.out_lin"]` will add delta modules to the attention output of distilbert's ayer 0, i.e., `distilbert.transformer.layer.0.attention.out_lin`.
   - set to `["attention.out_lin"]` will add the delta modules in every layer's `attention.out_lin`.

   ---

2. Regular Expression.

   We also support regex end-matching rules. We use a beginning `[r]` followed by a regular expression to represent this rule, where `[r]` is used to distinguish it from normal string matching rules and has no other meanings.

   Taking RoBERTa with an classifier on top as an example: It has two modules named `roberta.encoder.layer.0.attention.output.dense` and `roberta.encoder.layer.0.output.dense`, which both end up with `output.dense`. To distinguish them:

   - set `'[r](\d)+\.output.dense'` using regex rules, where `(\d)+` match any layer numbers. This rule will match all `roberta.encoder.layer.$.output.dense`. where `$` represents all integer numbers, here in a 12-layer RoBERTa, it's 0-11.
   - set `'[r][0-5]\.attention'` will match only the 0-5 layers' attention submodule.
   - set `'attention.output.dense'` using ordinary rules, which only match `roberta.encoder.layer.0.attention.output.dense`.

   ---

   **Regex in Json Configs**

   In json, you should write `"\\."` instead of `"\."` for a real dot due to json parsing rules. That is

```
{
    ...
    "modified_moduls": ['[r][0-5]\\.attention'],
    ...
}
```

3. Interactive Selection.

   We provide a way to interact visually to select modules needed.

```
from transformers import BertForMaskedLM
model = BertForMaskedLM.from_pretrained("bert-base-cased")
# suppose we load BERT

from opendelta import LoraModel # use lora as an example, others are same
delta_model = LoraModel(backbone_model=model, interactive_modify=True)
```

   by setting `interactive_modify`, a web server will be opened on local host, and the link will be print in the terminal, e.g.,

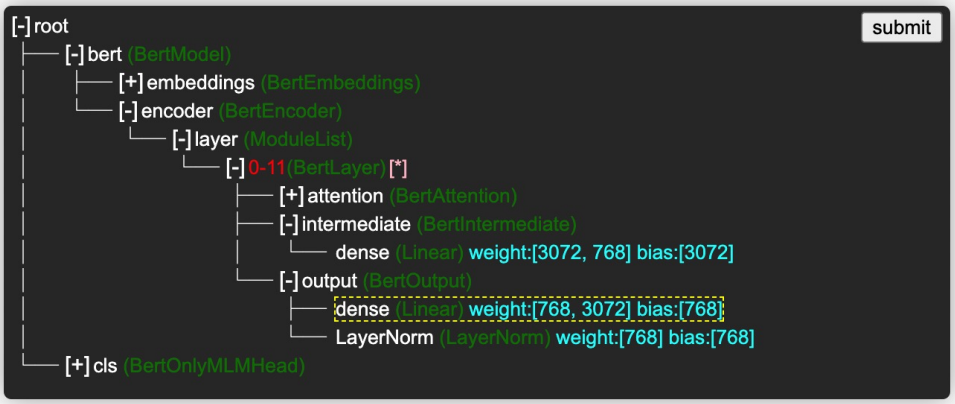
```
http://0.0.0.0:8888/
```

   If on your local machine, click to open the link for interactive modification.

   If on remote host, you could use port mapping. For example, vscode terminal will automatically do port mapping for you, you can simply use `control/command + click` to open the link.

   You can change the port number in case the default port number is occupied by other program by setting `interactive_modify=port_number`, in which port_number is an integer.

   The web page looks like the following figure.




   - By clicking on `[+]`/`[-]` to expand / collapse tree nodes.
   - By clicking on text to select tree nodes, **yellow dotted** box indicates the selection.
   - **Double** click on the pink `[*]` is an advanced option to unfold the repeated nodes. By default, modules with the same architecture are folded into one node and are marked in red, for example, the `BertLayer` of layers 0~11 in the above figure are in the same structure. Regular model changes will make the same changes to each layers.

- If you want to change only a few of them, first double-click on `[*]`, then select the parts you want in the unfolded structure.
- If you want to make the same change to all but a few of them, first select the common parts you want in the folded structure, then double-click on `[*]` to remove the few positions you don't need to change in the expanded structure.

Click `submit` button on the top-right corner, then go back to your terminal, you can get a list of name-based addresses printed in the terminal in the following format, and these modules are being "delta".

```
modified_modules:
[bert.encoder.layer.0.output.dense, ..., bert.encoder.layer.11.output.dense]
```

### 1.4.4 Examples

Nothing works better than a few lively examples. Comming Soon...

## 1.5 Visualize the Parameters

When OpenDelta makes modifications to a pretrained model (PTM), it is beneficial to know what your PTM looks like, especially the location of the parameters.

- **Before** applying opendelta, you can know **how to specify your modifications in terms of key addressing**.
- **After** the modification is done, you can know **if your modification is what you expected**, for example, whether the position of the delta modules are desired, or whether you froze the correct parameters.

Now let's begin to try the visualization utility.

### 1.5.1 Visualization is NOT easy using pytorch native function.

```
from transformers import BertForMaskedLM
backbone_model = BertForMaskedLM.from_pretrained("bert-base-uncased")
print(backbone_model)
```

The original presentation of models is **not tailored for repeated structures, big models, or parameters-centric tasks**.

### 1.5.2 Using visualization from opendelta.

First let's visualize all the parameters in the bert model. As we can see, structure inside a bert model, and the all the paramters location of the model are neatly represented in tree structure. (See *color scheme* for the colors)

```
from opendelta import Visualization
model_vis = Visualization(backbone_model)
model_vis.structure_graph()
```

**Suggestion**

We can reference a module according to the graph easily:

```
BertForMaskedLM(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
```

```
root
├── bert (BertModel)
│   ├── embeddings (BertEmbeddings)
│   │   ├── word_embeddings (Embedding) weight:[30522, 768]
│   │   ├── position_embeddings (Embedding) weight:[512, 768]
│   │   ├── token_type_embeddings (Embedding) weight:[2, 768]
│   │   └── LayerNorm (LayerNorm) weight:[768] bias:[768]
│   └── encoder (BertEncoder)
│       └── layer (ModuleList)
│           └── 0-11(BertLayer)
│               ├── attention (BertAttention)
│               │   ├── self (BertSelfAttention)
│               │   │   └── query,key,value(Linear) weight:[768, 768] bias:[768]
│               │   └── output (BertSelfOutput)
│               │       ├── dense (Linear) weight:[768, 768] bias:[768]
│               │       └── LayerNorm (LayerNorm) weight:[768] bias:[768]
│               ├── intermediate (BertIntermediate)
│               │   └── dense (Linear) weight:[3072, 768] bias:[3072]
│               └── output (BertOutput)
│                   ├── dense (Linear) weight:[768, 3072] bias:[768]
│                   └── LayerNorm (LayerNorm) weight:[768] bias:[768]
└── cls (BertOnlyMLMHead)
    └── predictions (BertLMPredictionHead) bias:[30522]
        ├── transform (BertPredictionHeadTransform)
        │   ├── dense (Linear) weight:[768, 768] bias:[768]
        │   └── LayerNorm (LayerNorm) weight:[768] bias:[768]
        └── decoder (Linear) weight:[30522, 768] bias:[30522]
```

```
print(backbone_model.bert.encoder.layer[0].intermdiate)
```

When using opendelta on a new backbone model, it's better to first visualize the child module names (shown in white), and then designating the `modified_modules`.

### 1.5.3 Now add a delta model and visualize the change.

```
from opendelta import LowRankAdapterModel
delta_model = LowRankAdapterModel(backbone_model)
delta_model.freeze_module(exclude=["cls", "intermediate", "LayerNorm"])
Visualization(backbone_model).structure_graph()
```


**Color Schema**

**PlatForm Sentivity**

Depending on the platform the code is running on, the colors may vary slightly.

### 1.5.4 We also provide the option to visualize the nodes without parameters.

```
Visualization(backbone_model).structure_graph(keep_non_params=True)
```

Thus, the modules like dropout and activations are kept.


**Order of the submodule**

Currently, OpenDelta's Visualization visualize the model based on pytorch's named_modules method. That means the order of the presented submodule is the order they are add to the parent module, not necessarily the order that tensors flows through.

## 1.6 Save and Share the Delta

### 1.6.1 Space efficient saving without changing the code.

After a modified backbone model is trained, you can save only trained part without change to any code, because **the state dict of the backbone model has been changed to the trainable parts**

```
from opendelta import CompacterModel
from transformers import BertForMaskedLM
backbone_model = BertForMaskedLM.from_pretrained("bert-base-uncased")
delta_model = CompacterModel(backbone_model) # modify the default modules.

# freeze module
delta_model.freeze_module(exclude=["deltas"], set_state_dict=True)
# or
delta_model.freeze_module(exclude=["deltas"])
```

#### save the checkpoint.

now save the backbone_model in normal way, and the checkpoint is **very space efficient**.

```
# ...
# After some training pipeline
# ...
torch.save(backbone_model.state_dict(), "delta.ckpt")

# the checkpoint size
import os
print("checkpoint size: {:.2f}M".format(os.path.getsize("delta.ckpt")/1024**2))
# checkpoint size: 0.32M
```

#### load the checkpoint.

In order to load the checkpoint, you should make sure the backbone model is a modified ones (so that it can take in the delta parameters). Then load the checkpoint with `strict=False`.

```
backbone_model.load_state_dict(torch.load("delta.ckpt"), strict=False)
# this will return long string of warning about the 'missing key'.
# if you want to supress it, use
# _ = backbone_model.load_state_dict(torch.load("delta.ckpt"), strict=False)
```

### 1.6.2 Save/Load the entire model after training.

#### save a delta model.

```
delta_model.save_finetuned("delta_model")
# Configuration saved in delta_model/config.json
# Model weights saved in delta_model/pytorch_model.bin
```

This will save all the trained parameters and the configuration of the delta model to path `delta_model/`

#### load a delta model.

```
backbone_model = BertForMaskedLM.from_pretrained("bert-base-uncased")
delta_model.from_finetuned("delta_model", backbone_model, local_files_only=True)
# passing local_files_only=True will save the time of checking in the web.
```

### 1.6.3 Share or download a model to/from the community.

#### Share.

```
delta_model.save_finetuned("test_delta_model", push_to_hub = True)
```

#### Download from community.

```
from transformers import AutoModelForSeq2SeqLM
t5 = AutoModelForSeq2SeqLM.from_pretrained("t5-base")
from opendelta import AutoDeltaModel
delta = AutoDeltaModel.from_finetuned("DeltaHub/lora_t5-base_mrpc", backbone_model=t5)
delta.log()
```

---

**Push to Hub**

Currently we only provide the option to push to huggingface model hub.

Before push to hub, you may need to register an account on Huggingface. You can refer to this tutorial about model sharing and uploading

In some cases, your checkpoint is still large for git, please install `git-lfs`.

---

**Sharing with the Community**

If you are satisfied with your checkpoint, do not forget to share your model to DeltaHub:

1. Add yourself to DeltaHub with the public link
2. Be sure to edit your model card to clearly illustrate the delta model before you share.
3. Click `setting` on the model
4. Transfer the model in `rename or transfer this model` section.

---

### 1.6.4 Save & Load for Composition of Delta


Currently save & load method is not suitable for *composition* of delta model. Please wait for future releases.

## 1.7 Philosophy and Key Features

---

**Plug-and-play Design.**

Existing open-source project to propogate this **''delta-tuning''** paradigm includes AdapterHub, which copies the transformers code base and modify on it, which makes it unintuitive to transfer from a normal code base to a delta-tuning ones.

OpenDelta approaches this problem via a **true plug-and-play** fashion to the PLMs. To migrate from a full-model finetuning training scripts to a delta tuning training scripts, you **DO NOT** need to change the backbone bone model code base to an adapted code base.

---

Here is how we achieve it.


**Read through it will also help you to implement your own delta models in a sustainable way.**

### 1.7.1 1. Name-based submodule addressing.

See *name based addressing*

### 1.7.2 2. Three basic submodule-level delta operations.

We use three key functions to achieve the modifications to the backbone model outside the backbone model's code.

1. **unfreeze some paramters**

   Some delta models will unfreeze a part of the model parameters and freeze other parts of the model, e.g. BitFit. For these methods, just use *freeze_module* method and pass the delta parts into `exclude`.

2. **replace an module**

   Some delta models will replace a part of the model with a delta model, i.e., the hidden states will no longer go through the original submodules. This includes Lora. For these methods, we have an *update_module* interface.

3. **insertion to the backbone**

   - **sequential insertion**

   Most adapter model insert a new adapter layer after/before the original transformers blocks. For these methods, insert the adapter's forward function after/before the original layer's forward function using *insert_sequential_module* interface.

   - **parallel insertion**

   Adapters can also be used in a parallel fashion (see Paper). For these methods, use *insert_parallel_module* interface.

**Doc-preserving Insertion**

In the insertion operations, the replaced forward function will inherit the doc strings of the original functions.

### 1.7.3 3. Pseudo input to initialize.

Some delta models, especially the ones that is newly introduced into the backbone, will need to determine the parameters' shape. To get the shape, we pass a pseudo input to the backbone model and determine the shape of each delta layer according to the need of smooth tensor flow.

**Pseudo Input**

Most models in Huggingface Transformers have an attribute dummy_inputs. This will create a nonsensical input with the correct format to pass into the model's forward function.

For the models that doesn't inherit/implement this attributes, we assume the pseudo input to the model is something like `input_id`, i.e., an integer tensor.

```
pseudo_input = torch.tensor([[0,0,0]])
# or
pseudo_input = torch.tensor([0,0,0])
```

 We will add interface to allow more pseudo input in the future.

## 1.8 Common Structure Mapping

Although different PTMs often share similar Transformers structures, the codebases, and most importantly, the variable names for each submodule, are quite different.

On the one hand, we **encourage the users to first *visualize* the PTMs' structure and then determine the name of submoduels.**

On the other hand, we designed a unified name convention of Transformer Structure, and provided several structure mapping from the original name to the unified name convention.

In this section, we will illustrate the unified name convention and structure mapping.

### 1.8.1 Common blocks in Transformers structure.

- embeddings (word embedding)
- encoder
  - block
    - $ (layer_id)
      - attn
      - q, k, v
      - proj

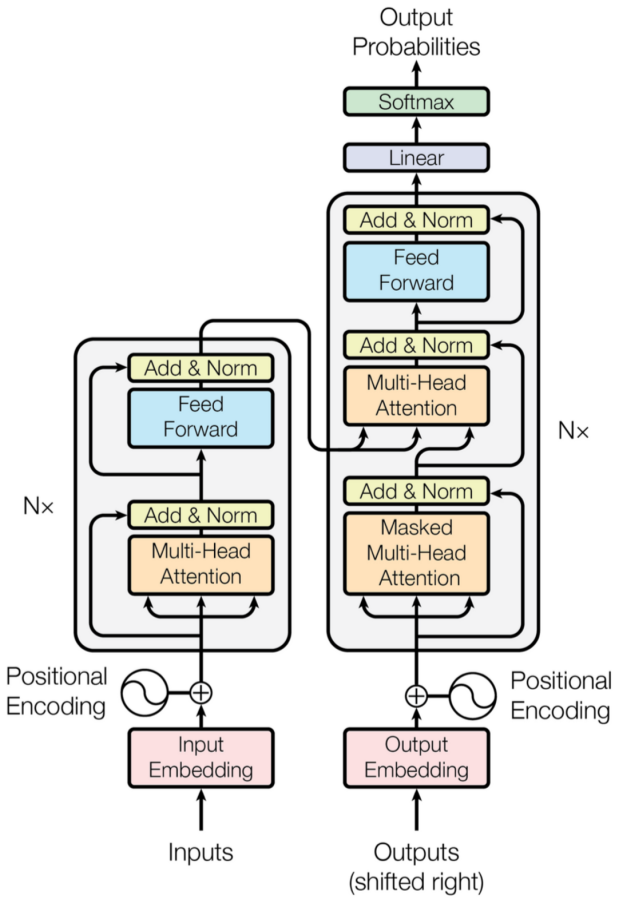
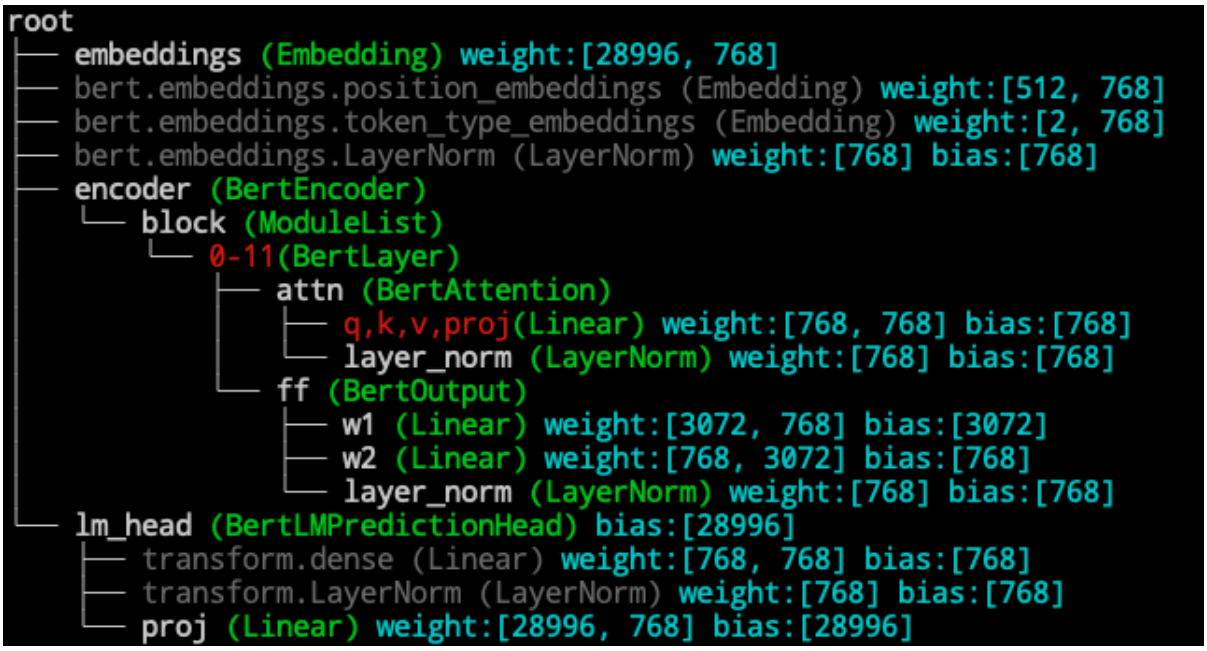



Figure 1: The Transformer - model architecture.

        - layer_norm
        - ff
        - w1
        - w2
        - layer_norm
- decoder (similar to encoder)
- lm_head
    - proj

Visualize bert-base using a common structure name: The submodules that are not common are grey.




### 1.8.2 Example

Example of bert mapping: a tree with node names specified by "__name__"

```
{
    "bert.embeddings.word_embeddings": {"__name__":"embeddings"},
    "bert.embeddings.position_embeddings": {"__name__":""},
    "bert.embeddings.token_type_embeddings": {"__name__":""},
    "bert.embeddings.LayerNorm": {"__name__":""},
    "bert.encoder": {"__name__":"encoder",
        "layer": {"__name__":"block",
            "$": {"__name__":"$",
                "attention": {"__name__":"attn",
                    "self.query": {"__name__":"q"},
                    "self.key": {"__name__":"k"},
                    "self.value": {"__name__":"v"},
                    "output.dense": {"__name__":"proj"},
```

(continues on next page)

(continued from previous page)

```
                    "output.LayerNorm": {"__name__":"layer_norm"},
                },
                "output": {"__name__":"ff",
                           "dense": {"__name__":"w2"},
                           "LayerNorm": {"__name__":"layer_norm"}
                },
                "intermediate.dense": {"__name__":"ff.w1"},
            }
        }
    },
    "cls.predictions": {"__name__": "lm_head",
        "transform.dense": {"__name__":""},
        "transform.LayerNorm": {"__name__":""},
        "decoder": {"__name__":"proj"},
    }
}
```

## 1.9 AutoDelta Mechanism

Inspired by Huggingface transformers AutoClasses , we provide an AutoDelta features for the users to

1. Easily to experiment with different delta models
2. Fast deploy from configuration file, especially from the repos in DeltaHub.

### 1.9.1 Easily load from dict, so that subject to change the type of delta models.

```
from opendelta import AutoDeltaConfig, AutoDeltaModel
from transformers import T5ForConditionalGeneration

backbone_model = T5ForConditionalGeneration.from_pretrained("t5-base")
```

We can load a config from a dict

```
config_dict = {
    "delta_type":"lora",
    "modified_modules":[
        "SelfAttention.q",
        "SelfAttention.v",
        "SelfAttention.o"
    ],
    "lora_r":4}
delta_config = AutoDeltaConfig.from_dict(config_dict)
```

Then use the config to add a delta model to the backbone model

```
delta_model = AutoDeltaModel.from_config(delta_config, backbone_model=backbone_model)

# now visualize the modified backbone_model
```

(continues on next page)

(continued from previous page)

```
from opendelta import Visualization
Visualizaiton(backbone_model).structure_graph()
```

### 1.9.2 Fast deploy from a finetuned delta checkpoints from DeltaHub

```
delta_model = AutoDeltaModel.from_finetuned("DeltaHub/sst2-t5-base", backbone_
↪model=backbone_model)  # TODO: the link may change.
```

**Hash checking**

Since the delta model only works together with the backbone model. we will automatically check whether you load the delta model the same way it is trained.

We calculate the trained model's md5 and save it to the config. When finishing loading the delta model, we will re-calculate the md5 to see whether it changes.

## 1.10 Composition of delta models

With OpenDelta, you can perform compostion of different delta models.

### 1.10.1 Add different deltas to the backbone

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained("roberta-base")
from opendelta import LoraModel, AdapterModel
delta_model = LoraModel(backbone_model=model, modified_modules=['key'], lora_r=1)
delta_model2 = AdapterModel(backbone_model=model, modified_modules=['output'],␣
↪bottleneck_dim=12)
delta_model.log()
```

### 1.10.2 Even add multiple delta to the same layer

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained("facebook/bart-base")
from opendelta import AdapterModel, LowRankAdapterModel
delta_model = AdapterModel(backbone_model=model, modified_modules=['fc2'])
delta_model2 = AdapterModel(backbone_model=model, modified_modules=['fc2'], bottleneck_
↪dim=12)
delta_model3 = LowRankAdapterModel(backbone_model=model, modified_modules=['fc2'],␣
↪reduction_factor=12)
delta_model.log()
```

**Order of Insertion**

```
root
├── shared(Embedding),lm_head(Linear) weight:[32128, 768]
├── encoder (T5Stack)
│   ├── embed_tokens (Embedding) weight:[32128, 768]
│   ├── block (ModuleList)
│   │   ├── 0 (T5Block)
│   │   │   └── layer (ModuleList)
│   │   │       ├── 0 (T5LayerSelfAttention)
│   │   │       │   ├── SelfAttention (T5Attention)
│   │   │       │   │   ├── q,v,o(Linear) weight:[768, 768] lora_A:[4, 768] lora_B:[768, 4]
│   │   │       │   │   ├── k (Linear) weight:[768, 768]
│   │   │       │   │   └── relative_attention_bias (Embedding) weight:[32, 12]
│   │   │       │   └── layer_norm (T5LayerNorm) weight:[768]
│   │   │       └── 1 (T5LayerFF)
│   │   │           ├── DenseReluDense (T5DenseReluDense)
│   │   │           │   ├── wi (Linear) weight:[3072, 768]
│   │   │           │   └── wo (Linear) weight:[768, 3072]
│   │   │           └── layer_norm (T5LayerNorm) weight:[768]
│   │   └── 1-11(T5Block)
│   │       └── layer (ModuleList)
│   │           ├── 0 (T5LayerSelfAttention)
│   │           │   ├── SelfAttention (T5Attention)
│   │           │   │   ├── q,v,o(Linear) weight:[768, 768] lora_A:[4, 768] lora_B:[768, 4]
│   │           │   │   └── k (Linear) weight:[768, 768]
│   │           │   └── layer_norm (T5LayerNorm) weight:[768]
│   │           └── 1 (T5LayerFF)
│   │               ├── DenseReluDense (T5DenseReluDense)
│   │               │   ├── wi (Linear) weight:[3072, 768]
│   │               │   └── wo (Linear) weight:[768, 3072]
│   │               └── layer_norm (T5LayerNorm) weight:[768]
│   └── final_layer_norm (T5LayerNorm) weight:[768]
└── decoder (T5Stack)
    ├── embed_tokens (Embedding) weight:[32128, 768]
    ├── block (ModuleList)
    │   ├── 0 (T5Block)
    │   │   └── layer (ModuleList)
    │   │       ├── 0 (T5LayerSelfAttention)
    │   │       │   ├── SelfAttention (T5Attention)
    │   │       │   │   ├── q,v,o(Linear) weight:[768, 768] lora_A:[4, 768] lora_B:[768, 4]
    │   │       │   │   ├── k (Linear) weight:[768, 768]
    │   │       │   │   └── relative_attention_bias (Embedding) weight:[32, 12]
    │   │       │   └── layer_norm (T5LayerNorm) weight:[768]
    │   │       ├── 1 (T5LayerCrossAttention)
    │   │       │   ├── EncDecAttention (T5Attention)
    │   │       │   │   └── q,k,v,o(Linear) weight:[768, 768]
    │   │       │   └── layer_norm (T5LayerNorm) weight:[768]
    │   │       └── 2 (T5LayerFF)
    │   │           ├── DenseReluDense (T5DenseReluDense)
    │   │           │   ├── wi (Linear) weight:[3072, 768]
    │   │           │   └── wo (Linear) weight:[768, 3072]
    │   │           └── layer_norm (T5LayerNorm) weight:[768]
    │   └── 1-11(T5Block)
    │       └── layer (ModuleList)
    │           ├── 0 (T5LayerSelfAttention)
    │           │   ├── SelfAttention (T5Attention)
    │           │   │   ├── q,v,o(Linear) weight:[768, 768] lora_A:[4, 768] lora_B:[768, 4]
    │           │   │   └── k (Linear) weight:[768, 768]
    │           │   └── layer_norm (T5LayerNorm) weight:[768]
    │           ├── 1 (T5LayerCrossAttention)
    │           │   ├── EncDecAttention (T5Attention)
    │           │   │   └── q,k,v,o(Linear) weight:[768, 768]
    │           │   └── layer_norm (T5LayerNorm) weight:[768]
    │           └── 2 (T5LayerFF)
    │               ├── DenseReluDense (T5DenseReluDense)
    │               │   ├── wi (Linear) weight:[3072, 768]
    │               │   └── wo (Linear) weight:[768, 3072]
    │               └── layer_norm (T5LayerNorm) weight:[768]
    └── final_layer_norm (T5LayerNorm) weight:[768]
```

```
root
├── roberta (RobertaModel)
│    ├── embeddings (RobertaEmbeddings)
│    │    ├── word_embeddings (Embedding) weight:[50265, 768]
│    │    ├── position_embeddings (Embedding) weight:[514, 768]
│    │    ├── token_type_embeddings (Embedding) weight:[1, 768]
│    │    └── LayerNorm (LayerNorm) weight:[768] bias:[768]
│    └── encoder (RobertaEncoder)
│         └── layer (ModuleList)
│              └── 0-11(RobertaLayer)
│                   ├── attention (RobertaAttention)
│                   │    ├── self (RobertaSelfAttention)
│                   │    │    ├── query,value(Linear) weight:[768, 768] bias:[768]
│                   │    │    └── key (Linear) weight:[768, 768] bias:[768] lora_A:[1, 768] lora_B:[768, 1]
│                   │    └── output (RobertaSelfOutput)
│                   │         ├── dense (Linear) weight:[768, 768] bias:[768]
│                   │         ├── LayerNorm (LayerNorm) weight:[768] bias:[768]
│                   │         └── adapter (AdapterLayer)
│                   │              └── modulelist (Sequential)
│                   │                   ├── down_proj (Linear) weight:[12, 768] bias:[12]
│                   │                   └── up_proj (Linear) weight:[768, 12] bias:[768]
│                   ├── intermediate (RobertaIntermediate)
│                   │    └── dense (Linear) weight:[3072, 768] bias:[3072]
│                   └── output (RobertaOutput)
│                        ├── dense (Linear) weight:[768, 3072] bias:[768]
│                        ├── LayerNorm (LayerNorm) weight:[768] bias:[768]
│                        └── adapter (AdapterLayer)
│                             └── modulelist (Sequential)
│                                  ├── down_proj (Linear) weight:[12, 768] bias:[12]
│                                  └── up_proj (Linear) weight:[768, 12] bias:[768]
└── classifier (RobertaClassificationHead)
     ├── dense (Linear) weight:[768, 768] bias:[768]
     └── out_proj (Linear) weight:[2, 768] bias:[2]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 0.383228%
```

**When adding to the same layer, please pay attention to the order of adding delta.** As the above example, adapter is added after the `fc2`, the tensor will first go through `adapter` then go through `adapter_1`, at last `compacter`. If the delta is added before the backbone layer, then the last added delta will be the first to go through.

Also, pay attention to the detaching order. The delta that is first added should be the last to be detached.

## 1.11 Multitask Modeling using OpenDelta

**Multitask Serving with Delta-tuning**

A huge advange of Delta-tuning is that it can be used for multitask serving. Imagine we have a pretrained model trained on a mix of data coming from multiple languages, e.g.,English, Chinese, and French. Now you want to have seperate models that specialise in Chinese, French, English. We can thus delta-tune three deltas on each language with small amount of additional language-specific data. During serving, when a Chinese sentence comes, you attach the "Chinese Delta", and next a French sentence comes, you detach the "Chinese Delta", and attach a "French Delta".

**Here is how to achieve multitask serving using OpenDelta.**

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained("facebook/bart-base")
from opendelta import LoraModel
delta_model = LoraModel(backbone_model=model, modified_modules=['fc2'])
delta_model.log()
```

```
root
├── model (BartModel)
│   ├── shared (Embedding) weight:[50265, 768]
│   ├── encoder (BartEncoder)
│   │   ├── embed_tokens (Embedding) weight:[50265, 768]
│   │   ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│   │   ├── layers (ModuleList)
│   │   │   └── 0-5(BartEncoderLayer)
│   │   │       ├── self_attn (BartAttention)
│   │   │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│   │   │       ├── self_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│   │   │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│   │   │       └── fc2 (Linear) weight:[768, 3072] bias:[768]
│   │   │           ├── adapter (AdapterLayer)
│   │   │           │   └── modulelist (Sequential)
│   │   │           │       ├── down_proj (Linear) weight:[24, 768] bias:[24]
│   │   │           │       └── up_proj (Linear) weight:[768, 24] bias:[768]
│   │   │           ├── adapter_1 (AdapterLayer)
│   │   │           │   └── modulelist (Sequential)
│   │   │           │       ├── down_proj (Linear) weight:[12, 768] bias:[12]
│   │   │           │       └── up_proj (Linear) weight:[768, 12] bias:[768]
│   │   │           └── low_rank_adapter (LowRankAdapter)
│   │   │               ├── down_sampler (LowRankLinear) W_left:[768, 1] W_right:[1, 64] b:[64]
│   │   │               └── up_sampler (LowRankLinear) W_left:[64, 1] W_right:[1, 768] b:[768]
│   │   └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
│   └── decoder (BartDecoder)
│       ├── embed_tokens (Embedding) weight:[50265, 768]
│       ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│       ├── layers (ModuleList)
│       │   └── 0-5(BartDecoderLayer)
│       │       ├── self_attn,encoder_attn(BartAttention)
│       │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│       │       ├── self_attn_layer_norm,encoder_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│       │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│       │       └── fc2 (Linear) weight:[768, 3072] bias:[768]
│       │           ├── adapter (AdapterLayer)
│       │           │   └── modulelist (Sequential)
│       │           │       ├── down_proj (Linear) weight:[24, 768] bias:[24]
│       │           │       └── up_proj (Linear) weight:[768, 24] bias:[768]
│       │           ├── adapter_1 (AdapterLayer)
│       │           │   └── modulelist (Sequential)
│       │           │       ├── down_proj (Linear) weight:[12, 768] bias:[12]
│       │           │       └── up_proj (Linear) weight:[768, 12] bias:[768]
│       │           └── low_rank_adapter (LowRankAdapter)
│       │               ├── down_sampler (LowRankLinear) W_left:[768, 1] W_right:[1, 64] b:[64]
│       │               └── up_sampler (LowRankLinear) W_left:[64, 1] W_right:[1, 768] b:[768]
│       └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
└── classification_head (BartClassificationHead)
    ├── dense (Linear) weight:[768, 768] bias:[768]
    └── out_proj (Linear) weight:[3, 768] bias:[3]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 0.506210%
```

```
root
├── model (BartModel)
│   ├── shared (Embedding) weight:[50265, 768]
│   ├── encoder (BartEncoder)
│   │   ├── embed_tokens (Embedding) weight:[50265, 768]
│   │   ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│   │   ├── layers (ModuleList)
│   │   │   └── 0-5(BartEncoderLayer)
│   │   │       ├── self_attn (BartAttention)
│   │   │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│   │   │       ├── self_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│   │   │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│   │   │       └── fc2 (Linear) weight:[768, 3072] bias:[768] lora_A:[8, 3072] lora_B:[768, 8]
│   │   └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
│   └── decoder (BartDecoder)
│       ├── embed_tokens (Embedding) weight:[50265, 768]
│       ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│       ├── layers (ModuleList)
│       │   └── 0-5(BartDecoderLayer)
│       │       ├── self_attn,encoder_attn(BartAttention)
│       │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│       │       ├── self_attn_layer_norm,encoder_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│       │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│       │       └── fc2 (Linear) weight:[768, 3072] bias:[768] lora_A:[8, 3072] lora_B:[768, 8]
│       └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
└── classification_head (BartClassificationHead)
    ├── dense (Linear) weight:[768, 768] bias:[768]
    └── out_proj (Linear) weight:[3, 768] bias:[3]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 0.262598%
```

Now we detach the deltas from the backbone

```
delta_model.detach()
delta_model.log()
```

```
root
├── model (BartModel)
│    ├── shared (Embedding) weight:[50265, 768]
│    ├── encoder (BartEncoder)
│    │    ├── embed_tokens (Embedding) weight:[50265, 768]
│    │    ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│    │    ├── layers (ModuleList)
│    │    │    └── 0-5(BartEncoderLayer)
│    │    │         ├── self_attn (BartAttention)
│    │    │         │    └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│    │    │         ├── self_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│    │    │         ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│    │    │         └── fc2 (Linear) weight:[768, 3072] bias:[768]
│    │    └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
│    └── decoder (BartDecoder)
│         ├── embed_tokens (Embedding) weight:[50265, 768]
│         ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│         ├── layers (ModuleList)
│         │    └── 0-5(BartDecoderLayer)
│         │         ├── self_attn,encoder_attn(BartAttention)
│         │         │    └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│         │         ├── self_attn_layer_norm,encoder_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│         │         ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│         │         └── fc2 (Linear) weight:[768, 3072] bias:[768]
│         └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
└── classification_head (BartClassificationHead)
     ├── dense (Linear) weight:[768, 768] bias:[768]
     └── out_proj (Linear) weight:[3, 768] bias:[3]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 0.000000%
```

We can reattach the deltas to the backbone

```
delta_model.attach()
delta_model.log()
```

```
root
├── model (BartModel)
│    ├── shared (Embedding) weight:[50265, 768]
│    ├── encoder (BartEncoder)
│    │    ├── embed_tokens (Embedding) weight:[50265, 768]
│    │    ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│    │    ├── layers (ModuleList)
│    │    │    └── 0-5(BartEncoderLayer)
│    │    │         ├── self_attn (BartAttention)
│    │    │         │    └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│    │    │         ├── self_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│    │    │         ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│    │    │         └── fc2 (Linear) weight:[768, 3072] bias:[768] lora_A:[8, 3072] lora_B:[768, 8]
│    │    └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
│    └── decoder (BartDecoder)
│         ├── embed_tokens (Embedding) weight:[50265, 768]
│         ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│         ├── layers (ModuleList)
│         │    └── 0-5(BartDecoderLayer)
│         │         ├── self_attn,encoder_attn(BartAttention)
│         │         │    └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│         │         ├── self_attn_layer_norm,encoder_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│         │         ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│         │         └── fc2 (Linear) weight:[768, 3072] bias:[768] lora_A:[8, 3072] lora_B:[768, 8]
│         └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
└── classification_head (BartClassificationHead)
     ├── dense (Linear) weight:[768, 768] bias:[768]
     └── out_proj (Linear) weight:[3, 768] bias:[3]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 0.262598%
```

**Independence of Different Delta Models**

Different delta models will be independent in detaching and attaching. (But the visualization will not show all deltas in the backbone model.)

```
# continue from the above example
from opendelta import AdapterModel
delta_model2 = AdapterModel(backbone_model=model, modified_modules=['fc1'])
delta_model2.log()
```

```
root
├── model (BartModel)
│    ├── shared (Embedding) weight:[50265, 768]
│    ├── encoder (BartEncoder)
│    │    ├── embed_tokens (Embedding) weight:[50265, 768]
│    │    ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│    │    ├── layers (ModuleList)
│    │    │    └── 0-5(BartEncoderLayer)
│    │    │         ├── self_attn (BartAttention)
│    │    │         │    └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│    │    │         ├── self_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│    │    │         ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│    │    │         │    └── adapter (AdapterLayer)
│    │    │         │         └── modulelist (Sequential)
│    │    │         │              ├── down_proj (Linear) weight:[24, 3072] bias:[24]
│    │    │         │              └── up_proj (Linear) weight:[3072, 24] bias:[3072]
│    │    │         └── fc2 (Linear) weight:[768, 3072] bias:[768] lora_A:[8, 3072] lora_B:[768, 8]
│    │    └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
│    └── decoder (BartDecoder)
│         ├── embed_tokens (Embedding) weight:[50265, 768]
│         ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│         ├── layers (ModuleList)
│         │    └── 0-5(BartDecoderLayer)
│         │         ├── self_attn,encoder_attn(BartAttention)
│         │         │    └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│         │         ├── self_attn_layer_norm,encoder_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│         │         ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│         │         │    └── adapter (AdapterLayer)
│         │         │         └── modulelist (Sequential)
│         │         │              ├── down_proj (Linear) weight:[24, 3072] bias:[24]
│         │         │              └── up_proj (Linear) weight:[3072, 24] bias:[3072]
│         │         └── fc2 (Linear) weight:[768, 3072] bias:[768] lora_A:[8, 3072] lora_B:[768, 8]
│         └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
└── classification_head (BartClassificationHead)
     ├── dense (Linear) weight:[768, 768] bias:[768]
     └── out_proj (Linear) weight:[3, 768] bias:[3]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 1.529844%
```

detach the lora delta

```
delta_model.detach() # detach the lora delta
delta_model.log()
```

```
root
├── model (BartModel)
│   ├── shared (Embedding) weight:[50265, 768]
│   ├── encoder (BartEncoder)
│   │   ├── embed_tokens (Embedding) weight:[50265, 768]
│   │   ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│   │   ├── layers (ModuleList)
│   │   │   └── 0-5(BartEncoderLayer)
│   │   │       ├── self_attn (BartAttention)
│   │   │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│   │   │       ├── self_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│   │   │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│   │   │       │   └── adapter (AdapterLayer)
│   │   │       │       └── modulelist (Sequential)
│   │   │       │           ├── down_proj (Linear) weight:[24, 3072] bias:[24]
│   │   │       │           └── up_proj (Linear) weight:[3072, 24] bias:[3072]
│   │   │       └── fc2 (Linear) weight:[768, 3072] bias:[768]
│   │   └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
│   └── decoder (BartDecoder)
│       ├── embed_tokens (Embedding) weight:[50265, 768]
│       ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│       ├── layers (ModuleList)
│       │   └── 0-5(BartDecoderLayer)
│       │       ├── self_attn,encoder_attn(BartAttention)
│       │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│       │       ├── self_attn_layer_norm,encoder_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│       │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│       │       │   └── adapter (AdapterLayer)
│       │       │       └── modulelist (Sequential)
│       │       │           ├── down_proj (Linear) weight:[24, 3072] bias:[24]
│       │       │           └── up_proj (Linear) weight:[3072, 24] bias:[3072]
│       │       └── fc2 (Linear) weight:[768, 3072] bias:[768]
│       └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
└── classification_head (BartClassificationHead)
    ├── dense (Linear) weight:[768, 768] bias:[768]
    └── out_proj (Linear) weight:[3, 768] bias:[3]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 1.273886%
```

detach the adapter delta and reattach the lora delta

```
delta_model2.detach() # detach the adapter delta
delta_model.attach() # reattach the lora delta
delta_model.log()
```

```
root
├── model (BartModel)
│   ├── shared (Embedding) weight:[50265, 768]
│   ├── encoder (BartEncoder)
│   │   ├── embed_tokens (Embedding) weight:[50265, 768]
│   │   ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│   │   ├── layers (ModuleList)
│   │   │   └── 0-5(BartEncoderLayer)
│   │   │       ├── self_attn (BartAttention)
│   │   │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│   │   │       ├── self_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│   │   │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│   │   │       └── fc2 (Linear) weight:[768, 3072] bias:[768] lora_A:[8, 3072] lora_B:[768, 8]
│   │   └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
│   └── decoder (BartDecoder)
│       ├── embed_tokens (Embedding) weight:[50265, 768]
│       ├── embed_positions (BartLearnedPositionalEmbedding) weight:[1026, 768]
│       ├── layers (ModuleList)
│       │   └── 0-5(BartDecoderLayer)
│       │       ├── self_attn,encoder_attn(BartAttention)
│       │       │   └── k_proj,v_proj,q_proj,out_proj(Linear) weight:[768, 768] bias:[768]
│       │       ├── self_attn_layer_norm,encoder_attn_layer_norm,final_layer_norm(LayerNorm) weight:[768] bias:[768]
│       │       ├── fc1 (Linear) weight:[3072, 768] bias:[3072]
│       │       └── fc2 (Linear) weight:[768, 3072] bias:[768] lora_A:[8, 3072] lora_B:[768, 8]
│       └── layernorm_embedding (LayerNorm) weight:[768] bias:[768]
└── classification_head (BartClassificationHead)
    ├── dense (Linear) weight:[768, 768] bias:[768]
    └── out_proj (Linear) weight:[3, 768] bias:[3]
Trainable Ratio: 100.000000%
Delta Parameter Ratio: 0.262598%
```

**BitFit not supported**

 Currently detach is not suitable for BitFit, which modify the requires_grad property. Please wait for future releases.

## 1.12 OpenDelta+

 We are working on testing and improving the functionality with work with other acceleration packages for model training and inference. For example, deepspeed, BMInf.

Feel free to contact us via email (shengdinghu@gmail.com) if you have any suggestion.

## 1.13 Favored Configuration

Generally, the default configurations are already good enough. If you want squeeze the size of delta models further, you can refer to the following papers.

- AdapterDrop: On the Efficiency of Adapters in Transformers
- Sparse Structure Search for Parameter-Efficient Tuning(Delta Tuning)

## 1.14 Citation

If you find our repo useful, please cite the following paper.

```
@article{ding2022delta,
  title={Delta tuning: A comprehensive study of parameter efficient methods for pre-
↪trained language models},
  author={Ding, Ning and Qin, Yujia and Yang, Guang and Wei, Fuchao and Yang, Zonghan␣
↪and Su, Yusheng and Hu, Shengding and Chen, Yulin and Chan, Chi-Min and Chen, Weize␣
↪and others},
  journal={arXiv preprint arXiv:2203.06904},
  year={2022}
}
```

## 1.15 Update Logs and Known Issues

### 1.15.1 Version 0.3.1

- We update must_try.py for a simple introduction of the core functionality of OpenDelta.
- Thanks to Weilin Zhao We merge a long-developed branch parallel_adapter into the main branch.

### 1.15.2 Version 0.3.0

#### Updates:

- Add this changelog for a granular record of updates.
- The default configuration of delta models can be applied to more wrapped models.
  - There is less need to configure 'modified_modules' for wrapped models like BertForSequenceClassification or even OpenMatch.DRModel, as long as it has a model we support default configuration inside. **Note that if you customize `modified_modules` by yourself, most pytorch models are supported.**
- LoRA and BitFit models now does not need pseudo data to instantiate the model.
- BitFit models can now support Conv1D using default configuration.
- Improve type hint for AutoDeltaModel.
- Fix bugs in documentation.
- Fix small bugs when saving a model without a config attributes.
- Make the default modified modules of adapter-like methods more accurate: attach the adapter-like modules after the output of attention layer and second feed-forward layer, both before the layernorm layers.
- A simple unit test folder containing development-time tests has been added for interested users.

#### Known Issues

- SoftPrompt is still not supported for wrapped model if the model has no attribute `get_input_embeddings`.
- Prefix Tuning is still limited to T5, GPT2, Bart, Bert, Roberta.

### 1.15.3 Version 0.2.4

#### Updates

- examples/examples_seq2seq and examples/examples_text-classification is depreciated and moved to legacy
- Thanks to Zhen Zhang, we provide examples_prompt, as a cleaner and more general framework, which unifies the delta tuning paradigm and the prompt-tuning paradigm. It is still based on Huggingface Trainers. In this example framework, the running pipeline is a unified script, the differences in tasks, models, delta tuning models, and even prompt-tuning paradigms are more modular and be more independent . Please try it out!

## 1.16 FAQs

1. **Why I encounder NotImplementedError in Prefix Tuning?**

   This is because we find no easy way to get a unified Prefix Tuning implementation for different attention classes. If you really want to use Prefix Tuning for the models we have not supported, you can implement the `PrefixLayerYOURMODEL` on your own or raise a issue to request the feature for your model.

2. **Available Models with default configurations are ..., Please manually add the delta models by speicifying 'modified_modules' based on the visualization of your model structure**

   Although most pre-trained models (PTMs) use the transformers archtecture, they are implemented differently. For example, the attention module in GPT2 and BERT is not only named differently, but also implemented in

different ways. Common structure mapping mapps the different name conventions of different PTMs into a unified name convention. But there are many PTMs that we do not currently cover. But don't worry! For these models, you can figure out which modules should you modify by simply *visualizing the PTMs*, and then specify the `modified modules` manually (See *name-based addressing*).

3. **Requires a dummy_inputs to be passed through the model to understand the dimensionality of each tensor in the computation graph. The {module.class.name} Class has no dummy_inputs, and automatically created dummy_inputs failed.**

   The `dummy_inputs` can be any data that make `backbone_model.forward(**dummy_inputs)` succeed. Only the form and shape of the `dummy_inputs` matter. To set dummy_inputs for your model, please use: `setattr(backbone_model, 'dummy_inputs', some_dummy_inputs)` before initializing `{self.__class__.__name__}`.

## 1.17 Base Classes

### 1.17.1 BaseDeltaConfig

**class BaseDeltaConfig**(*modified_modules=None*, *exclude_modules=None*, *unfrozen_modules=['deltas']*, *common_structure=False*, *backbone_class=None*, *backbone_checkpoint_name=None*, *backbone_hash=None*)

Base class for all configuration classes. Handles a few parameters common to all delta models' configurations as well as methods for loading/downloading/saving configurations.

Class attributes (overridden by derived classes):

- **delta_type** (`str`) – the name of the delta modules, used to create the correct `AutoConfig`.

**Parameters**

- **modified_modules** (`List[str]`, *optional*, defaults to `None`) – The list of keys to determine which modules you want to modify. OpenDelta will take every modulees that **ends with** the one of the provided keys as the modification target. When not given any value, i.e. `modified_modules=None`, the delta module will use the it corresponding default modification modules. Taking DistilBertModel with an classifier on top as an example:

  ---

  **Note: Examples**: When adding delta to DistilBertModel,

  1. set to `["0.attention.out_lin"]` will add delta modules to the attention output of distilbert's layer 0, i.e., `distilbert.transformer.layer.0.attention.out_lin`.
  2. set to `["attention.out_lin"]` will add the delta modules in every layer's `attention.out_lin`.

  ---

- **unfrozen_modules** (`List[str]`, *optional*, defaults to `["deltas"]`) – The modules that are unfrozen during training in *`freeze_module()`*, which includes the ones that are newly introduced as delta modules, and the ones that are originally a part of the model but set to trainable (`requires_grad=True`) to train together with the delta modules. Opendelta will take every modules that **ends with** the one of the provided keys and all its sub-modules and paramters as trainable.
- **exclude_modules** (`str`, *optional*, default to `None`) – The modules starts with these strings will be excluded in modification. Note that currently only plain text (no regular expression) is supported.

**Note:** **Examples**: When adding delta to DistilBertModel,

1. set this argument to `["bias"]` will make all bias terms tunable.
2. set this argument to `["attention"]` will make all parameters in all attention modules tunable.
3. set this argument to `["deltas"]` will make all the parameters in the newly introduced delta modules tunable.
4. set this argument to `["classifier"]` will make all parameters in the classifier tunable.
5. set this argument to `["3.ffn.lin2", "deltas", "classifier"]`, will make all parameters in the third layer's feed forward layer's send linear layer, the detla modules, and the classifiers modules tunable.

- **common_structure** (`bool`, *optional*, default to `None`) – Whether using the common structure mapping of the transformer model when designating `modified_modules` and ``unfrozen_modules`.
- **backbone_class** (`str`, *optional*, default to `None`) – The name of backbone model's class, e.g. `RobertaForMaskedLM`. Saving this infomation let the users explicitly know on which backbone the delta model is trained.
- **backbone_checkpoint_name** (`str`, *optional*, default to `None`) – The specific checkpoint of the model. In ideal case, it should be the url to download the checkpoint. However, we do not force the user to specify a downloadable url here.
- **backbone_hash** (`str`, *optional*, default to `None`) – The md5-hash of the backbone model. It is calculated using the string representation of the model and the sequential expansion of all the parameters in the model. When loading a delta checkpoint in strict mode, the hash of the backbone model will be compared to the hash in this config.

**classmethod from_finetuned**(*finetuned_delta_path: Union[str, PathLike]*, ***kwargs*) → *BaseDeltaConfig*

Instantiate a *BaseDeltaConfig* (or a derived class) from a finetined delta module configuration.

**Parameters**

- **finetuned_model_name_or_path** (`str` or `os.PathLike`) – This can be either:
  - a string, the *model id* of a finetuned delta model configuration hosted inside a model repo on deltahub.co. Valid model ids can be located at the root-level, like `bert-base-uncased`, or namespaced under a user or organization name, like `dbmdz/bert-base-german-cased`.
  - a path to a *directory* containing a configuration file saved using the *BaseDeltaConfig.save_finetuned()* method, e.g., `./my_model_directory/`.
  - a path or url to a saved configuration JSON *file*, e.g., `./my_model_directory/configuration.json`.
- **cache_dir** (`str` or `os.PathLike`, *optional*) – Path to a directory in which a downloaded pretrained delta model configuration should be cached if the standard cache should not be used.

```
delta_config = AdapterConfig.from_finetuned("thunlp/FactQA_T5-large_Adapter",␣
↪backbone_model=t5)
```

**save_finetuned**(*save_directory: Union[str, PathLike]*, ***kwargs*)

Save a configuration object to the directory `save_directory`, so that it can be re-loaded using the *BaseDeltaConfig.from_finetuned()* class method.

**Parameters**

- **save_directory** (`str` or `os.PathLike`) – Directory where the configuration JSON file will be saved (will be created if it does not exist).
- **push_to_hub** (`bool`, *optional*, defaults to `False`) – Whether or not to push your model to the Hugging Face model hub after saving it.

  **Warning:**

  1. Will raise error if you haven't config a Huggingface Model Hub.
  2. Using `push_to_hub=True` will synchronize the repository you are pushing to with `save_directory`, which requires `save_directory` to be a local clone of the repo you are pushing to if it's an existing folder. Pass along `temp_dir=True` to use a temporary directory instead.

- **kwargs** – Additional key word arguments.

**classmethod from_dict**(*config_dict: Dict[str, Any]*, ***kwargs*) → *BaseDeltaConfig*

Instantiate a *BaseDeltaConfig* from a python dictionary of parameters.

**Parameters**

- **config_dict** (`Dict[str, Any]`) – Dictionary that will be used to instantiate the configuration object. Such a dictionary can be retrieved from a pretrained checkpoint by leveraging the `get_config_dict()` method.
- **kwargs** (`Dict[str, Any]`) – Additional parameters from which to initialize the configuration object.

**Returns**

The configuration object instantiated from those parameters.

**Return type**

*BaseDeltaConfig*

**to_dict**() → Dict[str, Any]

Serializes this instance to a Python dictionary.

### 1.17.2 DeltaBase

**class DeltaBase**(*backbone_model: Module*, *modified_modules: Optional[List[str]] = None*, *exclude_modules: Optional[List[str]] = None*, *unfrozen_modules: Optional[List[str]] = None*, *interactive_modify: Optional[Union[bool, int]] = False*, *common_structure: Optional[bool] = False*)

This is the base class for all delta models. It provides four simple but effective functionalities for building the delta model:

1. addressing a module inside the backbone model using a minimal description key.
2. provide the interface for modifying and inserting model which keeps the docs/IO the same as the module before modification.
3. pass a pseudo input to determine the inter dimension of the delta models.
4. freeze a part of model parameters according to key.

It also provides unified interface for model loading and saving.

Class attributes (overridden by derived classes):

- delta_type (`str`): the name of the delta modules, used to create the correct `opendelta.AutoDeltaModel`.
- config_class (`BaseDeltaConfig`): The corresponding config model

**Parameters**

- **backbone_model** (`nn.Module`, *required*) – backbone model that the delta models are build opon. The modification to the backbone model are in place.
- **modified_modules** (`List[str]`, *optional*, default to `None`) – The modules are subjected to update.

  ---

  **Note:** leave this argument `None` will make the delta model return to the default setting, which add the delta models to the position experimented the paper. In this setting, the common structure mapping is loaded to addressing the corresponding modules.

  ---

- **exclude_modules** (`str`, *optional*, default to `None`) – The modules starts with these strings will be excluded in modification. Note that currently only plain text (no regular expression) is supported.
- **unfrozen_modules** (`str`, *optional*, default to `None`) – The modules that are **not** frozen when freezing the main part of the model.
- **registraction_name** (`str`, *optional*, default to `"deltas"`) – The root name of the delta models when attached to the backbone model.
- **common_structure** (`bool`, *optional*, default to `None`) – Whether use the common structure mapping to specify the modified_modules. i.e., if common_structure=True, then we use a common ["attn"] for attention module in different models. We DO NOT recommend manually set `common_structure` to `true` by yourself unless you are using delta among multiple backbones and don't want to modify the code.
- **interactive_modify** (`bool` or `int`, *optional*, default to `None`) – Whether to use interactive modification. By setting to `int` can specify the port of web server.

**config_class**

alias of *BaseDeltaConfig*

**forward**(**args*, ***kwargs*) → RuntimeError

**Warning:** Removed method. As the model is a delta model, which should be attached to a backbone model and can't forward any data by itself. Please using the backbone model's forward function after attach the delta model to the backbone.

**classmethod from_config**(*config: Union[*BaseDeltaConfig*, dict]*, *backbone_model: Module*, *check_hash=True*, ***kwargs*)

Initialize a delta model from a config object or a dict containing the configs. To temperarily change a value in the config, pass it through kwargs. If the config has a backbone model's hash, which means it is a finetuned delta model's config, then we will compare the hash in the config and the newly caculated to ensure the finedtuned delta model is trained on the passed backbone_model. Pass `check_hash=False` to disable the checking.

**Parameters**

- **config** (`BaseDeltaConfig` or `dict`) – initialize the delta model.
- **backbone_model** (`nn.Module`) – model. modifications will be made in place in the backbone model.
- **check_hash** (`bool`, default to `True`) – backbone hash.
- **kwargs** – Any configurations that are passed to update the config object. #TODO unit test needed.

**add_all_delta_to_backbone**(*backbone: Module*, *modified_modules: List[str]*) → Module

The main function to add delta models to the backbone model based on the `modified_modules`.

**Parameters**

- **backbone_model** (`nn.Module`, *required*) – modification to the backbone model are in place.
- **modified_modules** (`List[str]`, *optional*, default to `None`) – leave this argument `None` will make the delta model return to the default setting, which add the delta models to the position experimented the paper. In this setting, the common structure mapping is loaded to addressing the corresponding modules.

**Returns**

`nn.Module` The modified backbone model.

**update_module**(*module: Module*, *key: str*)

Update a module specified by `key`. The method is reimplemented in each specific delta model.

**freeze_module**(*module: Optional[Module] = None*, *exclude: Optional[List[str]] = None*, *set_state_dict: Optional[bool] = True*)

Freeze the parameters of plm. Leave the parameters in exclude untouched. deltas module is filtered with `_is_delta` attributes because it may have parameter sharing to the main model, (e.g., bias term)

**Parameters**

- **module** (`nn.Module`, *optional*, default to `None`) – The module of which some parts are frozen. If left with `None`, the function will the self.backbone_model as the module to be frozen.
- **exclude** (`List[str]`, *optional*, default to `["deltas"]`) – The parameters that don't need to be freezed. Default to all the delta parameters.
- **set_state_dict** (`bool`, *optional*, default to `True`) – Whether setting the backbone model's state dict to all the parameters that still need grad.

- **prefix** (str, *optional*, default to "") – A parameters that are used for recursive frozen. Should not be changed by passing argument other than "".

**find_key**(*key: str*, *target_list: List[str]*)

Check whether any target string is in the key or in the tail of the key, i.e.,

**Parameters**

- **key** (str) – The key (name) of a submodule in a ancestor module. E.g., model.encoder.layer.0.attention
- **target_list** (List[Union[str, re.Pattern]]) – The target list that we try to match key with. E.g., ["attention"]

**Returns**

bool True if the key matchs the target list.

**find_module**(*root_module: Module*, *key: str*)

Find the module using a key and the root module. Return both the parent reference, the child name and reference.

**Parameters**

- **root_module** (root_module) – The root_module to find the sub module in
- **key** (str) – The relative key to the root module.

**Returns**

- A reference to the parent module of the target module, mainly for substuting the target module.
- The key of the target module relevant to its parent module
- Target module.

**Return type**

(nn.Module, str, nn.Module)

**replace_module**(*parent_module: Module*, *child_name: str*, *child_module: Module*, *new_module: Module*, *delta_name: Optional[str] = 'delta'*)

Replace a module's child module with the new_module(a delta module). Used by delta method based on direct replacement, such as opendelta.delta_modules.lora.LoraModel.

**Parameters**

- **parent_module** (nn.Module) – The parent module of the replacement.
- **child_name** (str) – The chird module's name, i.e., parent_module.child_name give us child_module
- **child_module** (nn.Module) – The original child module.
- **new_module** (nn.Module) – The delta module.
- **delta_name** (str, *optional*, default ot delta) – The name of the delta module, used for recording. parent_module.delta_name WILL NOT give you the delta module.

**modify_module**(*module: Module*)

Modify the inside parameteres of a module. This method will be reimplemented in different derived class if needed.

**insert_module**(*module*, *method='sequential'*, *delta_module=None*, *delta_name='delta'*, *strict=False*, *_delta_info=None*)

insert a module (previous not exists in the code base) before/after a module. Specifically, it modifies the forward function of the original module to firstly pass the arguments into the new module's forward function and then pass it into the original ones. The new module can also be inserted after the original module with similar mechanism.

When implementing the new module , researchers should be aware of the components of arguments of the original module's forward function.

**Parameters**

- **module** – (nn.Module): The (sub)module to inserted a delta module.
- **delta_module** – (*DeltaBase*): The delta module to be inserted.
- **name** – (str, *optional*): The name of the delta in the backbone module.
- **strict** – (bool, *optional*): Whether to prohibit modify a modified module.
- **_delta_info** (Dict, *optional*) – Used in attach(), reattach a delta module to backbone. The info of original delta is passed through _delta_info.

**insert_sequential_module**(*module*, *delta_module=None*, *delta_name='delta'*, *strict=False*, *_delta_info=None*)

insert a module (previous not exists in the code base) before/after a module. Specifically, it modifies the forward function of the original module to firstly pass the arguments into the new module's forward function and then pass it into the original ones. The new module can also be inserted after the original module with similar mechanism.

When implementing the new module , researchers should be aware of the components of arguments of the original module's forward function.

**Parameters**

- **module** – (nn.Module): The (sub)module to inserted a delta module.
- **delta_module** – (*DeltaBase*): The delta module to be inserted.
- **name** – (str, *optional*): The name of the delta in the backbone module.
- **strict** – (bool, *optional*): Whether to prohibit modify a modified module.
- **_delta_info** (Dict, *optional*) – Used in attach(), reattach a delta module to backbone. The info of original delta is passed through _delta_info.

**insert_parallel_module**(*module*, *delta_module=None*, *delta_name='delta'*, *strict=False*, *_delta_info=None*)

insert a module (previous not exists in the code base) across a module. Specifically, it modifies the forward function of the original module to firstly pass the arguments into the delta model's forward function and set aside the calculation result. Then combine it with the calculation result output from the backbone module.

When implementing the new module , researchers should be aware of the arguments and keywards of the original module's forward function.

**Parameters**

- **module** – (nn.Module): The (sub)module to inserted a delta module.
- **delta_module** – (*DeltaBase*): The delta module to be inserted.
- **name** – (str, *optional*): The name of the delta in the backbone module.
- **strict** – (bool, *optional*): Whether to prohibit modify a modified module.

- **_delta_info** (Dict, *optional*) – Used in attach(), reattach a delta module to backbone. The info of original delta is passed through _delta_info.

**set_active_state_dict**(*module: Module*)

modify the state_dict function of the model (by default, the backbone model) to return only the tunable part.

**Parameters**
**module** (nn.Module) – The module modified. The modification is in-place.

**log**(*module=None*, *delta_ratio=True*, *trainable_ratio=True*, *visualization=True*, *cuda_memory=True*)

Log and visualize the result of applying delta. Possible Options are trainable_ratio, visualization, delta_ratio.

**Parameters**

- **delta_ratio** (bool, *optional*) – Whether computing the ratio of parameters in the delta modules.
- **trainable_ratio** (bool, *optional*) – Whether computing the ratio of trainable parameters.
- **visualization** (bool, *optional*) – Whether visualize the parameter information of the modified backbone.

**get_statistics**(*module=None*)

Get the statistics of the parameters in the delta modules.

**Parameters**
**module** (nn.Module, *optional*) – The module to compute the statistics.

**Returns**
The statistics of the parameters in the delta modules.

**Return type**
dict

**attach**(*module: Optional[Module] = None*, *reset_state_dict=True*)

Reattach the delta modules to the backbone. Note that this method can not be used to create new delta modules. Instead, a *DeltaBase.detach()* should precede this method.

**Parameters**
**module** (object, *optional*, default to None) – The backbone module that we reattach the deltas to.

**detach**(*module: Optional[Module] = None*, *reset_state_dict=True*)

Detach the delta module from the backbone. The delta module is not deleted, but temporarily turned off. Use *DeltaBase.attach()* to reattach the delta model to the backbone.

**Parameters**
**module** (object, *optional*, default to None) – The backbone module that we detached the deltas from.

## 1.18 Delta Models

### 1.18.1 Lora

**class LoraModel**(*backbone_model: Module*, *lora_r=8*, *lora_alpha=16*, *lora_dropout=0.0*, *modified_modules: Optional[List[str]] = None*, *unfrozen_modules: Optional[List[str]] = None*, *exclude_modules: Optional[List[str]] = None*, *common_structure: Optional[bool] = None*, *interactive_modify: Optional[Union[bool, int]] = False*)

The implementation of LoRA: Low-Rank Adaptation of Large Language Models . Thanks for their loralib.

---

**Note:** In our implementation, we did not use loralib.linear to replace the linear layer of the backbone model. Instead, we insert a parallel module into the backbone. In other words, we treat $(W + A^T B)X$ as $WX + A^T BX$, and insert the $A^T BX$ as a parallel insertion module. If you want to use the original implementation, please refer to *lora_old.py*

---

class attributes:

- default_modified_modules = [‘attn.q’, ‘attn.v’] According to the paper, they modify q and v matrix in the attention layer. However, other linears can also be modified, and may lead to better performance.

---

**Note:** modified_modules should point to linear layer. We currently don’t support broadcast to all linears in a module’s child modules.

---

- delta_type = “lora”

**Parameters**

- **backbone_model** (`transformers.PretrainedModels`) – The backbone model to be modified.
- **lora_r** (`int`, *optional*) – the rank of the lora parameters. The smaller lora_r is , the fewer parameters lora has.
- **lora_alpha** (`int`, *optional*) – A hyper-parameter to control the init scale of loralib.linear .
- **lora_dropout** (`float`, *optional*) – The dropout rate in lora.linear.
- **modified_modules** (`List[str]`) – For prefix tuning, the it must refer to an attention layer (Currently, only the implemented ones)
- **unfrozen_modules** (`List[str]`, *optional*, default to `None`) – The modules that should be unfrozen together with the prefix parameters.
- **common_structure** (`bool`) – whether using name-based addressing with a common structure mapping.

**config_class**

alias of `LoraConfig`

### 1.18.2 BitFit

**class BitFitModel**(*backbone_model: Module*, *modified_modules: Optional[List[str]] = None*, *exclude_modules: Optional[List[str]] = None*, *unfrozen_modules: Optional[List[str]] = None*, *common_structure: Optional[bool] = None*, *interactive_modify: Optional[Union[bool, int]] = False*)

The implementation of BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models . Unfreeze bias term (or add bias term if bias term is absent in the backbone, e.g. T5) to the modules of a transformer block.

---

**Note: Broadcast to Submodule**: We modify all potential positions of the specified `modified_modules`. That is to say, if we specify `attn` in the modified_modules, then all position including the q, k, v and out linear layer of the attention layer are added bias layer (or unfreezing). The potential position is determined according to equation (1)-(5) and the previous three equations.

---

**class attributes:**

- default_modified_modules = ["attn", "ff", "layer_norm","lm_head.proj"] According to the paper and the implementation in Compacter's baseline , we modify the bias term in the above modules.
- delta_type = "bitfit"

**Parameters**

- **backbone_model** (`transformers.PretrainedModels`) – The backbone model to be modified.
- **modified_modules** (`List[str]`) – For prefix tuning, the it must refer to an attention layer (Currently, only the implemented ones)
- **unfrozen_modules** (`List[str]`, *optional*, default to `None`) – The modules that should be unfrozen together with the prefix parameters.
- **common_structure** (`bool`) – whether using name-based addressing with a common structure mapping.

**config_class**

alias of `BitFitConfig`

**add_bias_to_modules_have_bias_or_known_type**(*c*)

If it has bias, unfreeze it. If it doesn't have bias: if it is Linear of LN, add to it, else pass.

**detach**(*module*)

Not implemented for BitFit yet. Please wait for the next version.

**attach**(*module*)

Not implemented for BitFit yet. Please wait for the next version.

### 1.18.3 Adapter

**class AdapterModel**(*backbone_model: Module*, *bottleneck_dim: Optional[int] = 24*, *non_linearity: Optional[str] = 'gelu_new'*, *modified_modules: Optional[bool] = None*, *unfrozen_modules: Optional[bool] = None*, *common_structure: Optional[bool] = None*, *interactive_modify: Optional[Union[bool, int]] = False*)

The implementation of Adapter(Parameter-Efficient Transfer Learning for NLP ) . Add adapter to the designated `modified_modules`. In sequential paradigm, The modules' output is then passed into the adapter's post_forward.

---

**Note:** We **assume** the output of the modified module is the hidden state or a tuple where hidden state is the first element. This is true for most PLMs. However, we admit that currently it's not rigorous, We will improve it in the next version. Currently, if you encount an error here for you backbone, you can modify the code to get the hidden state.

---

**class attributes:**

- default_modified_modules = ["attn", "ff"] According to the Adapter paper, we add adapter to the attention layer and feed forward layer.
- delta_type = "adapter"

**Parameters**

- **backbone_model** (`transformers.PretrainedModels`) – The backbone model to be modified.
- **bottleneck_dim** (`int`) – The dimension of the adapter's bottleneck.
- **non_linearity** (`str`) – The non linearity of the adapter.
- **modified_modules** (`List[str]`) – modules to add adapter after them.
- **unfrozen_modules** (`List[str]`, *optional*, default to `None`) – The modules that should be unfrozen together with the adapter parameters.
- **common_structure** (`bool`) – whether using name-based addressing witha common structure mapping.

**config_class**

alias of `AdapterConfig`

### 1.18.4 LowRankAdapter

**class LowRankAdapterModel**(*backbone_model: Module*, *reduction_factor=32*, *non_linearity='gelu_new'*, *low_rank_w_init='glorot-uniform'*, *low_rank_rank=1*, *modified_modules: Optional[List[str]] = None*, *exclude_modules: Optional[List[str]] = None*, *unfrozen_modules: Optional[List[str]] = None*, *common_structure: Optional[bool] = None*, *interactive_modify: Optional[Union[bool, int]] = False*)

The implementation of LowRankAdapter, proposed as a baseline in Compacter: Efficient Low-Rank Hypercomplex Adapter Layers . We found that it enjoys very few parameters but competitive performance, thus add it into OpenDelta. Low Rank Adapter parameterize each adapter's weight as a product of two rank-one(low) weights.

Add lowrank adapter layer to the designated `modified_modules`. In sequential paradigm, The modules' output is then passed into the low rank adapter's post_forward.

**Note:** We **assume** the output of the modified module is the hidden state or a tuple where hidden state is the first element. This is true for most PLMs. However, we admit that currently it's not rigorous, We will improve it in the next version. Currently, if you encount an error here for you backbone, you can modify the code to get the hidden state.

All the hyperparameter is adopted from the compacter code base .
**class attributes:**

- default_modified_modules = ["attn", "ff"] According to the compacter paper, we add low rank adapter to the attention layer and feed forward layer.
- delta_type = "lowrankadapter"

**Parameters**

- **backbone_model** (`transformers.PretrainedModels`) – The backbone model to be modified.
- **reduction_factor** (`int`, *optional*, default to `16`) – bottleneck_dim = hidden_dim//reduction_factor
- **non_linearity** (`str`, *optional*, default to `"gelu_new"`) – The non linearity activation used in between the down projecter and the up projecter.
- **low_rank_w_init** (`str`, *optional*, default to `"glorot-uniform"`) – The weight init method of the factorized linear weight.
- **low_rank_rank** (`int`, *optional*, default to 1) – The rank of the low-rank decomposition.
- **modified_modules** (`List[str]`) – For prefix tuning, the it must refer to an attention layer (Currently, only the implemented ones)
- **unfrozen_modules** (`List[str]`, *optional*, default to `None`) – The modules that should be unfrozen together with the prefix parameters.
- **common_structure** (`bool`, *optional*, default to `None`) – whether using name-based addressing with a common structure mapping.

`config_class`

alias of `LowRankAdapterConfig`

### 1.18.5 Compacter

**class CompacterModel**(*backbone_model*, *modified_modules: Optional[List[str]] = None*, *exclude_modules: Optional[List[str]] = None*, *unfrozen_modules: Optional[List[str]] = None*, *common_structure: Optional[bool] = None*, *interactive_modify: Optional[Union[bool, int]] = False*, *reduction_factor=16*, *non_linearity='gelu_new'*, *phm_c_init='normal'*, *hypercomplex_division=4*, *learn_phm=True*, *hypercomplex_nonlinearity='glorot-uniform'*, *shared_phm_rule=False*, *factorized_phm=True*, *shared_W_phm=False*, *factorized_phm_rule=False*, *phm_rank=1*, *phm_init_range=0.0001*, *kronecker_prod=None*, *use_bias_up_sampler=True*, *use_bias_down_sampler=True*)

The implementation of Compacter: Efficient Low-Rank Hypercomplex Adapter Layers . Add compacter layer to the designated `modified_modules`. In sequential paradigm, The modules' output is then passed into the compacter's post_forward.

**Note:** We **assume** the output of the modified module is the hidden state or a tuple where hidden state is the first element. This is true for most PLMs. However, we admit that currently it's not rigorous, We will improve it in the next version. Currently, if you encount an error here for you backbone, you can modify the code to get the hidden state.

All the hyperparameter is adopted from the compacter code base .
**class attributes:**

- default_modified_modules = ["attn", "ff"] According to the compacter paper, we add compacter to the attention layer and feed forward layer.
- delta_type = "compacter"

**Parameters**

- **backbone_model** (`transformers.PretrainedModels`) – The backbone model to be modified.
- **modified_modules** (`List[str]`) – For prefix tuning, the it must refer to an attention layer (Currently, only the implemented ones)
- **unfrozen_modules** (`List[str]`, *optional*, default to `None`) – The modules that should be unfrozen together with the prefix parameters.
- **common_structure** (`bool`, *optional*, default to `None`) – whether using name-based addressing with a common structure mapping.
- **reduction_factor** (`int`, *optional*, default to `16`) – bottleneck_dim = hidden_dim//reduction_factor
- **non_linearity** (`str`, *optional*, default to `"gelu_new"`) – The non linearity activation used in between the down projecter and the up projecter.
- **phm_c_init** (`str`, *optional*, default to `"normal"`) – The initialize method of the C in compacter.
- **hypercomplex_division** (`str`, *optional*, default to 4) – The `n` in the paper. The number of division along a dimension in compector.
- **learn_phm** (`bool`, *optional*, default to `True`) – Whether the phm rule requires_grad. Note that we didn't check the performance of learn_phm=False.
- **hypercomplex_nonlinearity** (`str`, *optional*, default to `"glorot-uniform"`) – The initialize method of the W in compacter.
- **shared_phm_rule** (`str`, *optional* , default to `False`) – Whether the phm rule is shared accross layer.
- **factorized_phm** (`str`, *optional*, default to `True`) – Whether to factorize the phm into low rank product.
- **shared_W_phm** (`str`, *optional* , default to `False`) – Whether the W_phm is shared accross layer.
- **factorized_phm_rule** (`str`, *optional* , default to `False`) – Whether to factorize the phm rule into low rank product.
- **phm_rank=1** (`int`, *optional*, default to 1) – The rank of low rank decomposition of phm.
- **phm_init_range** (`float`, *optional*, default to 0.0001) – The range of phm initialization.

- **kronecker_prod** (bool, *optional*, default to False) – Whether to perform kronecker_prod in matvec_product, proposed by Parameterization of Hypercomplex Multiplications
- **use_bias_up_sampler** (float, *optional*, default to True) – Whether add bias to the up projector. Note that the bias for this is a hidden_dim vector.
- **use_bias_down_sampler** (float, *optional*, default to True) – Whether add bias to the down projector. Note that the bias for this is a bottleneck_dim vector.

**config_class**

alias of CompacterConfig

### 1.18.6 Prefix tuning

**class PrefixModel**(*backbone_model: Module*, *prefix_token_num=6*, *reparameterize=True*, *embed_dim: Optional[int] = 512*, *mid_dim: Optional[int] = 512*, *modified_modules: Optional[List[str]] = None*, *exclude_modules: Optional[List[str]] = None*, *unfrozen_modules: Optional[List[str]] = None*, *common_structure: Optional[bool] = None*, *interactive_modify: Optional[Union[bool, int]] = False*)

The implementation of Prefix-Tuning: Optimizing Continuous Prompts for Generation . However, as attention block of different PLM differs substantially, e.g., the input arguments, the name convention of past_key_value, we have to implement different prefixlayer for different PLM. Given the inconvenience in the code level, we only support several commonly used backbone models (Currently: T5, DistilBert,Bert, Roberta, GPT2, BART). If you are trying to apply delta tuning to other backbone models, we suggest you trying other delta models or implementing it and making a pull request.

Experimental Feature:

Support inserting prefix token before each layer. For example, layer 3 4 6 10 and other layer untouched.

---

**Note:** If using reparameterize, the parameters will be in a reparameterization network, not in the prefix, which we attach to the first prefix layer. We will add a function to save only the generated prefix parameters for saving in the next version.

---

**Parameters**

- **backbone_model** (transformers.PretrainedModels) – The backbone model to be modified.
- **prefix_token_num** (int) – the number of prefix token
- **reparameterize** (bool) – Whether use the reparameterization for prefix tuning.
- **embed_dim** (int) – The embeding dimension of prefix token when using the reparameterization.
- **mid_dim** (int) – The dimension of the hiddens of the reparameterization network.
- **modified_modules** (List[str]) – For prefix tuning, the it must refer to an attention layer (Currently, only the implemented ones)
- **unfrozen_modules** (List[str], *optional*, default to None) – The modules that should be unfrozen together with the prefix parameters.
- **common_structure** (bool) – whether using name-based addressing with a common structure mapping.

### 1.18.7 Soft Prompt Tuning

**class SoftPromptModel**(*backbone_model: Module*, *soft_token_num=100*, *init_range=0.5*, *token_init=True*, *other_expand_ids={'attention_mask': 1, 'token_type_ids': 0}*, *modified_modules: Optional[List[str]] = None*, *exclude_modules: Optional[List[str]] = None*, *unfrozen_modules: Optional[List[str]] = None*, *common_structure: Optional[bool] = None*, *interactive_modify: Optional[Union[bool, int]] = False*)

This is the implementation of The Power of Scale for Parameter-Efficient Prompt Tuning . Similar to `PrefixTuningTemplate`, This template also does not need any textual template. Addition tokens are directly concatenated into the input ids. There are two initializations of the new tokens. (1). random initialization. (2) initialize with the tokens of the plm (We simply take the first n_tokens similar to their implementation).

Note that this template can be simply achieved by `SoftManualTemplate`, in which you set `n_token` <soft> tokens template before the <text_a> will give the same result.

**Parameters**

- **backbone_model** (`transformers.PretrainedModels`) – The backbone model to be modified.
- **soft_token_num** (`int`, *optional*) – num of new tokens to add in the front of the input.
- **init_range** (`float`, *optional*) – If initialize new tokens randomly, the random range of uniform distribution.
- **token_init** (`bool`, *optional*, default to `True`) – Whether to initialize the new tokens with tokens of the PLM.
- **other_expand_ids** (`dict`, *optional*, default to `{'attention_mask':1, 'token_type_ids':0}`) – The name of other tokens and its default value that expand along with the input sequence. For example, when you prepend 100 tokens to the input_ids, the attention_mask should be extended, and the token_type_ids should be extended as well.
- **modified_modules** (`List[str]`) – For prefix tuning, the it must refer to an attention layer (Currently, only the implemented ones).
- **unfrozen_modules** (`List[str]`, *optional*, default to `None`) – The modules that should be unfrozen together with the prefix parameters.
- **common_structure** (`bool`) – whether using name-based addressing with a common structure mapping.

**config_class**

alias of `SoftPromptConfig`

## 1.19 Auto Classes

### 1.19.1 AutoDeltaConfig

**class AutoDeltaConfig**(**args*, ***kwargs*)

This is a generic configuration class that will be instantiated as one of the configuration classes of the library when created with the *from_finetuned()* or *from_dict()* class method. This class cannot be instantiated directly using `__init__()` (throws an error).

**classmethod from_dict**(*config_dict: Dict[str, Any]*, ***kwargs*)

Instantiate a DeltaConfig according to the dict. Automatically load the config specified by `delta_type`.

**Parameters**

- **config_dict** (dict) – The dict of configs of delta model.
- **kwargs** – Other keyword argument pass to initialize the config.

Examples:

```
config = AutoDeltaConfig.from_dict({"delta_type":"lora"}) # This will load the␣
↪dault lora config.
config = AutoDeltaConfig.from_dict({"delta_type":"lora", "lora_r":5}) # Will␣
↪load the default lora config, with lora_r = 5
```

**classmethod from_finetuned**(*finetuned_delta_path*, ***kwargs*)

Instantiate one of the configuration classes of the library from a finetuned delta model configuration. The configuration class to instantiate is selected based on the `delta_type` property of the config object that is loaded.

**Parameters**

- **finetuned_delta_path** (str or os.PathLike, *optional*) – Can be either:
  - A string, the model id of a finetuned delta model configuration hosted inside a model repo on huggingface.co. Valid model ids can be located at the root-level, like `Davin/lora`, or namespaced under a user or organization name, like `DeltaHub/lora_t5-base_mrpc`.
  - A path to a *directory* containing a configuration file saved using the `save_finetuned()` method, e.g., `./my_model_directory/`.
  - A path or url to a saved configuration JSON *file*, e.g.,``./my_model_directory/configuration.json``.
- **cache_dir** (str or os.PathLike, *optional*) – Path to a directory in which a downloaded pretrained model configuration should be cached if the standard cache should not be used.

Examples:

```
from transformers import AutoConfig
delta_config = AutoDeltaConfig.from_finetuned("thunlp/FactQA_T5-large_Adapter")
```

### 1.19.2 AutoDeltaModel

**class AutoDeltaModel**(**args*, ***kwargs*)

**classmethod from_config**(*config*, *backbone_model*, ***kwargs*) → *DeltaBase*

Automatically instantiates a delta model based on the `config`. The delta model correspond to the delta `config` will be loaded and initialized using the arguments in `config`.

---

**Note:** Only using *from_config()* method will not load the finetuned weight file (e.g., pytorch_model.bin). Please use from_finetuned directly.

---

**Parameters**

- **config** (BaseDeltaConfig) –

- **backbone_model** (nn.Module) –

Examples:

```
config = AutoDeltaConfig.from_finetuned("DeltaHub/lora_t5-base_mrpc")
delta_model = AutoDeltaModel.from_config(config, backbone_model)
```

**classmethod from_finetuned**(*finetuned_delta_path*, *backbone_model*, **model_args*, ***kwargs*) → *DeltaBase*

Automatically instantiated a delta model and load the finetuned checkpoints based on the `finetuned_delta_path`, which can either be a string pointing to a local path or a url pointint to the delta hub. It will check the hash after loading the delta model to see whether the correct backbone and delta checkpoint are used.

**Parameters**

- **finetuned_delta_path** (str or os.PathLike, *optional*) – Can be either:
  - A string, the model name of a finetuned delta model configuration hosted inside a model repo on Delta Center, like `thunlp/FactQA_T5-large_Adapter`.
  - A path to a directory containing a configuration file saved using the *save_finetuned()* method, e.g., `./my_model_directory/`.
  - A path or url to a saved configuration JSON *file*, e.g., `./my_model_directory/configuration.json`.The last two option are not tested but inherited from huggingface.
- **backbone_model** (nn.Module) – The backbone model to be modified.
- **model_args** – Other argument for initialize the model. See :*DeltaBase.from_finetuned* for details.
- **kwargs** – Other kwargs that will be passed into DeltaBase.from_finetuned. See *DeltaBase.from_finetuned* for details.

Example:

```
delta_model = AutoDeltaModel.from_finetuned("thunlp/FactQA_T5-large_Adapter",␣
↪backbone_model=5)
```

# 1.20 Utils

## 1.20.1 SaveLoadMixin

**class SaveLoadMixin**

**save_finetuned**(*finetuned_delta_path: ~typing.Optional[~typing.Union[str, ~os.PathLike]] = './delta_checkpoints/'*, *save_config: bool = True*, *state_dict: ~typing.Optional[dict] = None*, *save_function: ~typing.Callable = <function save>*, *push_to_dc: bool = False*, *center_args: ~typing.Optional[~typing.Union[~opendelta.utils.saving_loading_utils.DeltaCenterArguments, dict]] = {}*, *center_args_pool: ~typing.Optional[dict] = {}*, *list_tags: ~typing.Optional[~typing.List] = []*, *dict_tags: ~typing.Optional[~typing.Dict] = {}*, *delay_push: bool = False*, *test_result=None*, *usage: ~typing.Optional[str] = ''*)

Save a model and its configuration file to a directory, so that it can be re-loaded using the `save_finetuned()` class method.

**Parameters**

- **finetuned_delta_path** – (optional) path to the directory where the model and its configuration file will be saved. If not specified, the model will be saved in the directory `./delta_checkpoints/`, which is a subdirectory of the current working directory.
- **save_config** – (optional) if `True`, the configuration file will be saved in the same directory as the model file. if `False`, only the state dict will be saved.
- **state_dict** – (optional) a dictionary containing the model's state_dict. If not specified, the state_dict is loaded from the backbone model's trainable parameters.
- **save_function** – (optional) the function used to save the model. Defaults to `torch.save`.
- **state_dict_only** – (optional) if `True`, only the state_dict will be saved.
- **push_to_dc** – (optional) if `True`, the model will prepare things to pushed to the DeltaCenter. This includes: - creating a configuration file for the model - creating a directory for the model - saving the model's trainable parameters - pushing the model to the DeltaCenter
- **center_args** – (optional) the arguments that are used to distinguish between different delta models on the DeltaCenter
- **center_args_pool** – (optional) a dictionary containing the arguments that are used to distinguish between different delta models on the DeltaCenter
- **list_tags** – (optional) a list of tags that will be added to the model's configuration file
- **dict_tags** – (optional) a dictionary of tags that will be added to the model's configuration file
- **delay_push** – (optional) if `True`, the model will not be pushed to the DeltaCenter. This is useful if you want to push the model later.

**load_checkpoint**(*path*, *load_func=<function load>*, *backbone_model=None*)

Simple method for loading only the checkpoint

**save_checkpoint**(*path*, *save_func=<function save>*, *backbone_model=None*)

Simple method for saving only the checkpoint

**classmethod from_finetuned**(*finetuned_delta_path: Optional[Union[str, PathLike]]*, *backbone_model: Module*, *delta_config=None*, *cache_dir: Optional[Union[str, PathLike]] = None*, *state_dict: Optional[dict] = None*, **model_args*, *force_download: Optional[bool] = False*, *check_hash: Optional[bool] = True*, *local_files_only: Optional[bool] = False*, ***kwargs*)

Instantiate a finetuned delta model from a path. The backbone_model is set in evaluation mode by default using `model.eval()` (Dropout modules are deactivated). To further train the model, you can use the *freeze_module* method.

**Parameters**

- **finetuned_delta_path** – (optional) path to the directory where the model and its configuration file will be saved. If not specified, the model will be loaded from the directory cahce directory. (see `cache_dir`),

- **backbone_model** – the backbone model that will be used to instantiate the fine-tuned delta model.
- **delta_config** – (optional) the configuration file of the finetuned delta model. If not specified, the configuration file is loaded from the directory `finetuned_delta_path`.
- **cache_dir** – (optional) path to the directory where the model and its configuration file will be saved. If not specified, we will first look into current working directory, then the cache directory of your system, e.g., ~/.cache/delta_center/,
- **state_dict** – (optional) a dictionary containing the model's state_dict. If not specified, the state_dict is loaded from the `finetuned_delta_path`.
- **force_download** – (optional) if `True`, the model will be downloaded from the internet even if it is already present in the cache directory.
- **check_hash** – (optional) if `True`, check whether the hash of the model once it's trained differs from what we load now.
- **local_files_only** – (optional) if `True`, the model will be loaded from the local cache directory.

**create_delta_center_args**(*center_args*, *center_args_pool*)

Create the delta center args for the center model. center_args has higher priority than center_args_pool.

**create_default_name**(***kwargs*)

Currently, it's only a simple concatenation of the arguments.

### 1.20.2 Visualization

**class Visualization**(*plm: Module*)

Better visualization tool for *BIG* pretrained models.

- Better repeated block representation
- Clearer parameter position
- and Visible parameter state.

**Parameters**

**plm** (torch.nn.Module) – The pretrained model, actually can be any pytorch module.

**structure_graph**(*rootname='root'*, *expand_params=False*, *keep_non_params=False*, *common_structure=False*, *mapping=None*, *only_common=False*, *printTree=True*)

Draw the structure graph in command line.

**Parameters**

- **rootname** (str) –
- **keep_non_params** (bool) –
- **expand_params** (bool) Display parameter infomation (shape, etc) –
- **common_structure** (bool) –
- **only_common** (bool) –
- **mapping** (dict) –

**build_common_tree**(*module: Module*, *mapping*, *tree: Optional[ModuleTree] = None*, *query=''*, *key_to_root=''*)

(Unstable) build the common tree structure

### 1.20.3 Structure Map

**class CommonStructureMap**(*backbone_model*, *strict=True*, *warning=False*, *visualize=True*)

A loading structure map.

**transform**(*org_key*, *strict=True*, *warning=False*)

Transform a key in the original model to the name convention in common structure.

### 1.20.4 Utility Functions

#### Hashing

**gen_model_hash**(*model*, *with_parameters=True*)

Get model hash (structure and parameter)

**gen_parameter_hash**(*generator*, *md5=None*)

Get parameter hash. From https://zhuanlan.zhihu.com/p/392942816

#### Signature

**signature**(*f*)

Get the function f 's input arguments. A useful gadget when some function slot might be instantiated into multiple functions.

**Parameters**

**f** (function) – the function to get the input arguments.

**Returns**

of args, default, varargs, keywords, respectively.s

**Return type**

namedtuple

**get_arg_names**(*f*)

Get a functions argument name, remove the `self` argument

**get_arg_names_inside_func**(*func*)

Get the functions argument name inside the function itself. Remove `self` argument.

#### Named-based addressing

**superstring_in**(*str_a: str*, *list_b: List[str]*)

check whether there is any string in list b containing str_a.

Args: Returns:

**is_child_key**(*str_a: str*, *list_b: List[str]*)

check whether a string in `list_b` is the child key in `str_a`

Args: Returns:

**endswith_in_normal**(*str_a: str*, *list_b: List[str]*)

check whether `str_a` has a substring that is in list_b.

Args: Returns:

**endswith_in_regex**(*str_a: str*, *list_b: List[str]*)

check whether `str_a` has a substring that is in list_b.

Args: Returns:

## 1.21 Indices and tables

- genindex

# PYTHON MODULE INDEX

## O

# INDEX

## O



## P



## R



## S



## T



## U



## V