Lightning Transformers

PyTorch Lightning

Nov 21, 2022

GET STARTED

CHAPTER

ONE

QUICK START

1.1 Installation

```
pip install lightning-transformers
```

Install bleeding-edge:

```
pip install git+https://github.com/PytorchLightning/lightning-transformers.git@master␣
↪--upgrade
```

Install all optional dependencies as well:

```
pip install lightning-transformers["extra"]
```

1.2 Using Lightning-Transformers

Lightning Transformers provides LightningModules, LightningDataModules and Strategies to use Transformers with the PyTorch Lightning Trainer, supporting tasks such as:ref:*language_modeling*, *Translation* and more. To use, simply:

1. Pick a task to train (*LightningModule*)
2. Pick a dataset (*LightningDataModule*)
3. Use any PyTorch Lightning parameters and optimizations

Here is an example of training bert-base-cased on the CARER emotion dataset using the Text Classification task.

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.text_classification import (
    TextClassificationDataModule,
    TextClassificationTransformer,
)

tokenizer = AutoTokenizer.from_pretrained(
    pretrained_model_name_or_path="bert-base-cased"
)
dm = TextClassificationDataModule(
    batch_size=1,
    dataset_name="emotion",
```

(continues on next page)

(continued from previous page)

```
    max_length=512,
    tokenizer=tokenizer,
)
model = TextClassificationTransformer(pretrained_model_name_or_path="bert-base-cased",
↪ num_labels=dm.num_classes)

trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1)

trainer.fit(model, dm)
```

1.2.1 Changing the Optimizer

Swapping to the RMSProp optimizer:

```
import pytorch_lightning as pl
import torch
import transformers
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.text_classification import (
    TextClassificationDataModule,
    TextClassificationTransformer,
)


class RMSPropTransformer(TextClassificationTransformer):
    def configure_optimizers(self):
        optimizer = torch.optim.RMSprop(self.parameters(), lr=1e-5)
        # automatically find the total number of steps we need!
        num_training_steps, num_warmup_steps = self.compute_warmup(self.num_training_
↪steps, num_warmup_steps=0.1)
        scheduler = transformers.get_linear_schedule_with_warmup(
            self.optimizer, num_warmup_steps=num_warmup_steps, num_training_steps=num_
↪training_steps
        )
        return {
            "optimizer": optimizer,
            "lr_scheduler": {"scheduler": scheduler, "interval": "step", "frequency":␣
↪1},
        }


tokenizer = AutoTokenizer.from_pretrained(
    pretrained_model_name_or_path="bert-base-cased"
)
dm = TextClassificationDataModule(
    batch_size=1,
    dataset_name="emotion",
    max_length=512,
    tokenizer=tokenizer,
)
model = RMSPropTransformer(pretrained_model_name_or_path="bert-base-cased", num_
↪labels=dm.num_classes)

trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1)
```

(continues on next page)

(continued from previous page)

```
trainer.fit(model, dm)
```

Enabling DeepSpeed/Sharded/Mixed Precision and more is super simple through the Lightning Trainer.

```
# enable DeepSpeed with 16bit precision
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1, strategy=
↪'deepspeed', precision=16)

# enable DeepSpeed ZeRO Stage 3 with BFLOAT16 precision
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1, strategy=
↪'deepspeed_stage_3_offload', precision="bf16")

# enable Sharded Training with 16bit precision
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1, strategy='ddp_
↪sharded', precision=16)
```

1.2.2 Inference

```
from transformers import AutoTokenizer
from lightning_transformers.task.nlp.text_classification import␣
↪TextClassificationTransformer

model = TextClassificationTransformer(
    pretrained_model_name_or_path="bert-base-uncased",
    tokenizer=AutoTokenizer.from_pretrained(pretrained_model_name_or_path="bert-base-
↪uncased"),
)
model.hf_predict("Lightning rocks!")
# Returns [{'label': 'LABEL_0', 'score': 0.545...}]
```

CHAPTER

TWO

LANGUAGE MODELING

2.1 The Task

Causal Language Modeling is the vanilla autoregressive pre-training method common to most language models such as GPT-3 or CTRL (Excluding BERT-like models, which were pre-trained using the Masked Language Modeling training method).

During training, we minimize the maximum likelihood during training across spans of text data (usually in some context window/block size). The model is able to attend to the left context (left of the mask). When trained on large quantities of text data, this gives us strong language models such as GPT-3 to use for downstream tasks.

2.2 Datasets

Currently supports the wikitext2 dataset, or custom input files. Since this task is usually the pre-training task for Transformers, it can be used to train new language models from scratch or to fine-tune a language model onto your own unlabeled text data.

2.3 Usage

Language Models pre-trained or fine-tuned to the Causal Language Modeling task can then be used in generative predictions.

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.language_modeling import (
    LanguageModelingDataModule,
    LanguageModelingTransformer,
)

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="gpt2")
model = LanguageModelingTransformer(pretrained_model_name_or_path="gpt2")
dm = LanguageModelingDataModule(
    batch_size=1,
    dataset_name="wikitext",
    dataset_config_name="wikitext-2-raw-v1",
    tokenizer=tokenizer,
)
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1)
```

(continues on next page)

(continued from previous page)

```
trainer.fit(model, dm)
```

We report the Cross Entropy Loss for validation.

2.4 Language Modeling Using Your Own Files

To use custom text files, the files should contain the raw data you want to train and validate on.

During data pre-processing the text is flattened, and the model is trained and validated on context windows (block size) made from the input text. We override the dataset files, allowing us to still use the data transforms defined with the base datamodule.

Below we have defined a csv file to use as our input data.

```
text,
this is the first sentence,
this is the second sentence,
```

```
from lightning_transformers.task.nlp.language_modeling import (
    LanguageModelingDataModule,
)

dm = LanguageModelingDataModule(
    batch_size=1,
    train_file="path/train.csv",
    validation_file="/path/valid.csv"
    tokenizer=tokenizer,
)
```

2.5 Language Modeling Inference Pipeline

By default we use the text generation pipeline, which requires a conditional input string and generates an output string.

```
from transformers import AutoTokenizer
from lightning_transformers.task.nlp.language_modeling import␣
↪LanguageModelingTransformer

model = LanguageModelingTransformer(
        pretrained_model_name_or_path="prajjwal1/bert-tiny",
        tokenizer=AutoTokenizer.from_pretrained(pretrained_model_name_or_path=
↪"prajjwal1/bert-tiny"),
    )
model.hf_predict("The house:")
```

CHAPTER

THREE

MULTIPLE CHOICE

3.1 The Task

The Multiple Choice task requires the model to decide on a set of options, given a question with optional context.

Similar to the text classification task, the model is fine-tuned on multi-class classification to provide probabilities across all possible answers. This is useful if the data you'd like the model to predict on requires selecting from a set of answers based on context or questions, where the answers can be variable. In contrast, use the text classification task if the answers remain static and are not needed to be included during training.

3.2 Datasets

Currently supports the RACE and SWAG datasets, or custom input files.

```
Question: What color is the sky?
Answers:
    A: Blue
    B: Green
    C: Red

Model answer: A
```

3.3 Training

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.multiple_choice import (
    MultipleChoiceTransformer,
    SwagMultipleChoiceDataModule,
)

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="bert-base-
↪uncased")
model = MultipleChoiceTransformer(pretrained_model_name_or_path="bert-base-uncased")
dm = SwagMultipleChoiceDataModule(
    batch_size=1,
    dataset_config_name="regular",
    padding=False,
```

(continues on next page)

(continued from previous page)

```
    tokenizer=tokenizer,
)
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1)

trainer.fit(model, dm)
```

We report Cross Entropy Loss, Precision, Recall and Accuracy for validation.

3.4 Multiple Choice Using Your Own Files

To use custom text files, the files should contain the data you want to train and validate on and be in CSV or JSON format as described below.

The format varies from dataset to dataset as input columns may differ, as well as pre-processing. To make our life easier, we use the RACE dataset format and override the files that are loaded.

Below we have defined a json file to use as our input data.

```
{
    "article": "The man walked into the red house but couldn't see where the light␣
↪was.",
    "question": "What colour is the house?",
    "options": ["White", "Red", "Blue"]
    "answer": "Red"
}
```

We override the dataset files, allowing us to still use the data transforms defined with the RACE dataset.

```
from lightning_transformers.task.nlp.multiple_choice import (
    RaceMultipleChoiceDataModule,
)

dm = RaceMultipleChoiceDataModule(
    batch_size=1,
    dataset_config_name="all",
    padding=False,
    train_file="path/train.json",
    validation_file="/path/valid.json"
    tokenizer=tokenizer,
)
```

3.5 Multiple Choice Inference

Currently there is no HF pipeline available for this model. Feel free to make an issue or PR if you require this functionality.

CHAPTER

FOUR

QUESTION ANSWERING

4.1 The Task

The Question Answering task requires the model to determine the start and end of a span within the given context, that answers a given question. This allows the model to pre-condition on contextual information to determine an answer.

Use this task when you would like to fine-tune onto data where an answer can be extracted from context information. Since this is an extraction task, you can rely on most Transformer models as your backbone.

4.2 Datasets

Currently supports the SQuAD dataset or custom input text files.

```
Context: The ground is black, the sky is blue and the car is red.
Question: What color is the sky?

Model answer: {"answer": "the sky is blue", "start": 21, "end": 35}
```

4.3 Training

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.question_answering import (
    QuestionAnsweringTransformer,
    SquadDataModule,
)

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="bert-base-
↪uncased")
model = QuestionAnsweringTransformer(pretrained_model_name_or_path="bert-base-uncased
↪")
dm = SquadDataModule(
    batch_size=1,
    dataset_config_name="plain_text",
    max_length=384,
    version_2_with_negative=False,
    null_score_diff_threshold=0.0,
    doc_stride=128,
```

(continues on next page)

(continued from previous page)

```
    n_best_size=20,
    max_answer_length=30,
    tokenizer=tokenizer,
)
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1)

trainer.fit(model, dm)
```

4.4 Question Answering Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files.

The format varies from dataset to dataset as input columns may differ, as well as pre-processing. To make our life easier, we use the SWAG dataset format and override the files that are loaded.

```
{
    "answers": {
        "answer_start": [1],
        "text": ["This is a test text"]
    },
    "context": "This is a test context.",
    "id": "1",
    "question": "Is this a test?",
    "title": "train test"
}
```

We override the dataset files, allowing us to still use the data transforms defined with this dataset.

```
from lightning_transformers.task.nlp.question_answering import (
    SquadDataModule,
)

dm = SquadDataModule(
    batch_size=1,
    dataset_config_name="plain_text",
    max_length=384,
    version_2_with_negative=False,
    null_score_diff_threshold=0.0,
    doc_stride=128,
    n_best_size=20,
    max_answer_length=30,
    train_file="path/train.csv",
    validation_file="/path/valid.csv"
    tokenizer=tokenizer,
)
```

4.5 Question Answering Inference Pipeline

By default we use the question answering pipeline, which requires a context and a question as input.

```
from transformers import AutoTokenizer
from lightning_transformers.task.nlp.question_answering import␣
↪QuestionAnsweringTransformer

model = QuestionAnsweringTransformer(
    pretrained_model_name_or_path="sshleifer/tiny-gpt2",
    tokenizer=AutoTokenizer.from_pretrained(pretrained_model_name_or_path="sshleifer/
↪tiny-gpt2"),
)
model.hf_predict(dict(context="Lightning is great", question="What is great?"))
```

CHAPTER

FIVE

SUMMARIZATION

5.1 The Task

The Summarization task requires the model to summarize a document into a shorter sentence.

5.2 Datasets

Currently supports the CNN/DailyMail and XSUM dataset or custom input text files.

In the CNN/Daily Mail dataset, this involves taking long articles and summarizing them.

```
document: "The car was racing towards the tunnel, whilst blue lights were flashing␣
↪behind it. The car entered the tunnel and vanished..."

Model answer: "Police are chasing a car entering a tunnel."
```

5.3 Training

To use this task, we must select a Seq2Seq Encoder/Decoder based model, such as T5 or BART. Encoder only models like GPT/BERT will not work as they are encoder only.

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.summarization import (
    SummarizationTransformer,
    XsumSummarizationDataModule,
)

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="t5-base")
model = SummarizationTransformer(
    pretrained_model_name_or_path="t5-base",
    use_stemmer=True,
    val_target_max_length=142,
    num_beams=None,
    compute_generate_metrics=True,
)
dm = XsumSummarizationDataModule(
    batch_size=1,
```

(continues on next page)

(continued from previous page)

```
    max_source_length=128,
    max_target_length=128,
    tokenizer=tokenizer,
)
trainer = pl.Trainer(accelerator="auto", devices=1, max_epochs=1)

trainer.fit(model, dm)
```

5.4 Summarization Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files.

```
{
    "source": "some-body",
    "target": "some-sentence"
}
```

We override the dataset files, allowing us to still use the data transforms defined with this dataset.

```
from lightning_transformers.task.nlp.summarization import (
    XsumSummarizationDataModule,
)

dm = XsumSummarizationDataModule(
    batch_size=1,
    max_source_length=128,
    max_target_length=128,
    train_file="path/train.csv",
    validation_file="/path/valid.csv"
    tokenizer=tokenizer,
)
```

5.5 Summarization Inference Pipeline

By default we use the summarization pipeline, which requires an input document as text.

```
from transformers import AutoTokenizer
from lightning_transformers.task.nlp.summarization import SummarizationTransformer

model = SummarizationTransformer(
    pretrained_model_name_or_path="patrickvonplaten/t5-tiny-random",
    tokenizer=AutoTokenizer.from_pretrained(pretrained_model_name_or_path=
↪"patrickvonplaten/t5-tiny-random"),
)

model.hf_predict(
    "The results found significant improvements over all tasks evaluated",
    min_length=2,
    max_length=12,
)
```

CHAPTER

SIX

TEXT CLASSIFICATION

6.1 The Task

The Text Classification Task fine-tunes the model to predict probabilities across a set of labels given input text. The task supports both binary and multi-class/multi-label classification.

6.2 Datasets

Currently supports the XLNI, GLUE and emotion datasets, or custom input files.

```
Input: I don't like this at all!

Model answer: {"label": "angry", "score": 0.8}
```

6.3 Training

Use this task when you would like to fine-tune Transformers on a labeled text classification task. For this task, you can rely on most Transformer models as your backbone.

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.text_classification import (
    TextClassificationDataModule,
    TextClassificationTransformer,
)

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="bert-base-
↪uncased")
dm = TextClassificationDataModule(
    batch_size=1,
    dataset_name="glue",
    dataset_config_name="sst2",
    max_length=512,
    tokenizer=tokenizer,
)
model = TextClassificationTransformer(pretrained_model_name_or_path="bert-base-uncased
↪", num_labels=dm.num_classes)
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1)
```

(continues on next page)

(continued from previous page)

```
trainer.fit(model, dm)
```

We report the Precision, Recall, Accuracy and Cross Entropy Loss for validation.

6.4 Text Classification Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files.

The label mapping is automatically generated from the training dataset labels if no mapping is given.

```
{
    "label": "sad",
    "text": "I'm feeling quite sad and sorry for myself but I'll snap out of it soon."
}
```

```
from lightning_transformers.task.nlp.text_classification import (
    TextClassificationDataModule,
    TextClassificationTransformer,
)

dm = TextClassificationDataModule(
    batch_size=1,
    max_length=512,
    train_file="path/train.json",
    validation_file="/path/valid.json"
    tokenizer=tokenizer,
)
```

6.5 Text Classification Inference Pipeline

By default we use the sentiment-analysis pipeline, which requires an input string.

```
from transformers import AutoTokenizer
from lightning_transformers.task.nlp.text_classification import␣
↪TextClassificationTransformer

model = TextClassificationTransformer(
    pretrained_model_name_or_path="prajjwal1/bert-tiny",
    tokenizer=AutoTokenizer.from_pretrained(pretrained_model_name_or_path="prajjwal1/
↪bert-tiny"),
)
model.hf_predict("Lightning rocks!")
```

CHAPTER

SEVEN

TOKEN CLASSIFICATION

7.1 The Task

The Token classification Task is similar to text classification, except each token within the text receives a prediction. A common use of this task is Named Entity Recognition (NER). Use this task if you require your data to be classified at the token level.

7.2 Datasets

Currently supports the conll dataset, or custom input files.

7.3 Training

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.token_classification import (
    TokenClassificationDataModule,
    TokenClassificationTransformer,
)

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="bert-base-
↪uncased")
dm = TokenClassificationDataModule(
    batch_size=1,
    task_name="ner",
    dataset_name="conll2003",
    preprocessing_num_workers=1,
    label_all_tokens=False,
    revision="master",
    tokenizer=tokenizer,
)
model = TokenClassificationTransformer(pretrained_model_name_or_path="bert-base-
↪uncased", labels=dm.num_classes)
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1)

trainer.fit(model, dm)
```

We report the Precision, Recall, Accuracy and Cross Entropy Loss for validation.

7.4 Token Classification Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files. For each token, there should be an associated label.

```
{
    "label_tags": [11, 12, 12, 21, 13, 11, 11, 21, 13, 11, 12, 13, 11, 21, 22, 11, 12,
↪ 17, 11, 21, 17, 11, 12, 12, 21, 22, 22, 13, 11, 0],
    "tokens": ["The", "European", "Commission", "said", "on", "Thursday", "it",
↪"disagreed", "with", "German", "advice", "to", "consumers"]
}
```

```
from lightning_transformers.task.nlp.token_classification import␣
↪TokenClassificationDataModule

dm = TokenClassificationDataModule(
    batch_size=1,
    task_name="ner",
    dataset_name="conll2003",
    preprocessing_num_workers=1,
    label_all_tokens=False,
    revision="master",
    train_file="path/train.json",
    validation_file="/path/valid.json"
    tokenizer=tokenizer,
)
```

7.5 Token Classification Inference Pipeline

By default we use the NER pipeline, which requires a an input sequence string and the number of labels.

```
from transformers import AutoTokenizer
from lightning_transformers.task.nlp.token_classification import␣
↪TokenClassificationTransformer

model = TokenClassificationTransformer(
    pretrained_model_name_or_path="prajjwal1/bert-tiny",
    tokenizer=AutoTokenizer.from_pretrained(pretrained_model_name_or_path="prajjwal1/
↪bert-tiny"),
    labels=2,
)
model.hf_predict("Have a good day!")
```

CHAPTER

EIGHT

TRANSLATION

8.1 The Task

The Translation task fine-tunes the model to translate text from one language to another.

8.2 Datasets

Currently supports the WMT16 dataset or custom input text files.

```
Input Text (English): "The ground is black, the sky is blue and the car is red."
Model Output (German): "Der Boden ist schwarz, der Himmel ist blau und das Auto ist␣
↪rot."
```

8.3 Training

To use this task, select a Seq2Seq Encoder/Decoder based model, such as multi-lingual T5 or BART. Conventional models like GPT/BERT will not work as they are encoder only. In addition, you also need a tokenizer that has been created on multi-lingual text. This is true for mt5 and mbart.

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.translation import (
    TranslationTransformer,
    WMT16TranslationDataModule,
)

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="t5-base")
model = TranslationTransformer(
    pretrained_model_name_or_path="t5-base",
    n_gram=4,
    smooth=False,
    val_target_max_length=142,
    num_beams=None,
    compute_generate_metrics=True,
)
dm = WMT16TranslationDataModule(
    # WMT translation datasets: ['cs-en', 'de-en', 'fi-en', 'ro-en', 'ru-en', 'tr-en']
    dataset_config_name="ro-en",
```

(continues on next page)

(continued from previous page)

```
    source_language="en",
    target_language="ro",
    max_source_length=128,
    max_target_length=128,
    tokenizer=tokenizer,
)
trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=1)

trainer.fit(model, dm)
```

8.4 Translation Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files.

```
{
    "source": "example source text",
    "target": "example target text"
}
```

We override the dataset files, allowing us to still use the data transforms defined with this dataset.

```
from lightning_transformers.task.nlp.translation import WMT16TranslationDataModule

dm = WMT16TranslationDataModule(
    # WMT translation datasets: ['cs-en', 'de-en', 'fi-en', 'ro-en', 'ru-en', 'tr-en']
    dataset_config_name="ro-en",
    source_language="en",
    target_language="ro",
    max_source_length=128,
    max_target_length=128,
    train_file="path/train.json",
    validation_file="/path/valid.json"
    tokenizer=tokenizer,
)
```

8.5 Translation Inference Pipeline

By default we use the translation pipeline, which requires a source text string.

```
from transformers import AutoTokenizer
from lightning_transformers.task.nlp.translation import TranslationTransformer

model = TranslationTransformer(
    pretrained_model_name_or_path="patrickvonplaten/t5-tiny-random",
    tokenizer=AutoTokenizer.from_pretrained(pretrained_model_name_or_path=
↪"patrickvonplaten/t5-tiny-random"),
)
model.hf_predict("¡Hola Sean!")
```

CHAPTER

NINE

IMAGE CLASSIFICATION

9.1 The Task

Image classification is the task of classifying an image to a label or task.

9.2 Datasets

Currently supports the beans dataset, or custom input files.

9.3 Usage

Language Models pre-trained or fine-tuned to the Causal Language Modeling task can then be used in generative predictions.

```
import pytorch_lightning as pl
from transformers import AutoFeatureExtractor

from lightning_transformers.task.vision.image_classification import (
    ImageClassificationDataModule,
    ImageClassificationTransformer,
)

feature_extractor = AutoFeatureExtractor.from_pretrained(pretrained_model_name_or_
↪path="nateraw/vit-base-beans")
dm = ImageClassificationDataModule(
    batch_size=8,
    dataset_name="beans",
    num_workers=8,
    feature_extractor=feature_extractor,
)
model = ImageClassificationTransformer(pretrained_model_name_or_path="nateraw/vit-
↪base-beans", num_labels=dm.num_classes)

trainer = pl.Trainer(accelerator="auto", devices="auto", max_epochs=5)
trainer.fit(model, dm)
```

We report the Cross Entropy Loss for validation.

9.4 Image Classification Inference Pipeline

```
from transformers import AutoTokenizer
from lightning_transformers.task.vision.image_classification import␣
↪ImageClassificationTransformer

model = ImageClassificationTransformer(
    pretrained_model_name_or_path="nateraw/vit-base-beans",
    tokenizer=AutoFeatureExtractor.from_pretrained(pretrained_model_name_or_path=
↪"nateraw/vit-base-beans"),
)
# predict on the logo
model.hf_predict(
    "https://github.com/PyTorchLightning/lightning-transformers/blob/master/"
    "docs/source/_static/images/logo.png?raw=true"
)
```

CHAPTER

TEN

SPARSEML

SparseML provides GPU-class performance on CPUs through sparsification, pruning, and quantization. For more details, see SparseML docs.

With multiple machines, the command has to be run on all machines either manually, or using an orchestration system such as SLURM or TorchElastic. More information can be seen in the Pytorch Lightning Computing Cluster.

We provide out of the box configs to use SparseML. Just pass the SparseML Callback when training.

```
import pytorch_lightning as pl
from lightning_transformers.callbacks import TransformerSparseMLCallback

pl.Trainer(
    callbacks=TransformerSparseMLCallback(
        output_dir="/content/MODELS",
        recipe_path="/content/recipe.yaml"
    )
)
```

These commands are only useful when a recipe has already been created. Example recipes can be found here.

After training, this will leave two ONNX models in the trainer.callbacks.output_dir folder: small_model.onnx and model.onnx. small_model.onnx is excellent for demos. For reliable inference, it is recommended to optimize model.onnx with your compression algorithm.

CHAPTER

ELEVEN

HUGGINGFACE HUB CHECKPOINTS

Lightning Transformers default behaviour means we save PyTorch based checkpoints.

HuggingFace Transformers provides a separate API for saving checkpoints. Below we describe two ways to save HuggingFace checkpoints manually or during training.

To manually save checkpoints from your model:

```
from lightning_transformers.task.nlp.text_classification import␣
↪TextClassificationTransformer

model = TextClassificationTransformer(pretrained_model_name_or_path="prajjwal1/bert-
↪tiny")

# saves a HF checkpoint to this path.
model.save_hf_checkpoint("checkpoint")
```

To save an additional HF Checkpoint everytime the checkpoint callback saves, pass in the `HFSaveCheckpoint` plugin:

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.plugins.checkpoint import HFSaveCheckpoint
from lightning_transformers.task.nlp.text_classification import (
    TextClassificationDataModule,
    TextClassificationTransformer,
)

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="prajjwal1/
↪bert-tiny")
dm = TextClassificationDataModule(
    batch_size=1,
    dataset_name="glue",
    dataset_config_name="sst2",
    max_length=512,
    tokenizer=tokenizer,
)
model = TextClassificationTransformer(pretrained_model_name_or_path="prajjwal1/bert-
↪tiny")
trainer = pl.Trainer(plugins=HFSaveCheckpoint(model=model))
trainer.fit(model, dm)
```

CHAPTER

TWELVE

BIG TRANSFORMERS MODEL INFERENCE

Lightning Transformers provides out of the box support for running inference with very large billion parameter models. Under-the-hood we use HF Transformer's large model support to auto-select devices for optimal throughput and memory usage.

Below is an example of how you can run generation with a large 6B parameter transformer model using Lightning Transformers. We've also managed to run `bigscience/bloom` which is 176B parameters using 8 A100s with the below code.

```
pip install accelerate
```

```
import torch
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.language_modeling import␣
↪LanguageModelingTransformer

model = LanguageModelingTransformer(
    pretrained_model_name_or_path="EleutherAI/gpt-j-6B",
    tokenizer=AutoTokenizer.from_pretrained("EleutherAI/gpt-j-6B"),
    low_cpu_mem_usage=True,
    device_map="auto",
)

output = model.generate("Hello, my name is", device=torch.device("cuda"))
print(model.tokenizer.decode(output[0].tolist()))
```

This will allow the model to be split onto GPUs/CPUs and even kept onto Disk to optimize memory space.

12.1 Inference with Manual Checkpoints

Download the sharded checkpoint weights that we'll be using:

```
git clone https://huggingface.co/sgugger/sharded-gpt-j-6B
cd sharded-gpt-j-6B
git-lfs install
git pull
```

```
import torch
from accelerate import init_empty_weights
from transformers import AutoTokenizer
```

(continues on next page)

(continued from previous page)

```
from lightning_transformers.task.nlp.language_modeling import␣
↪LanguageModelingTransformer

# initializes empty model for us to the load the checkpoint.
with init_empty_weights():
   model = LanguageModelingTransformer(
   pretrained_model_name_or_path="EleutherAI/gpt-j-6B",
   tokenizer=AutoTokenizer.from_pretrained("EleutherAI/gpt-j-6B")
 )

# automatically selects the best devices (cpu/gpu) to load model layers based on␣
↪available memory
model.load_checkpoint_and_dispatch("sharded-gpt-j-6B", device_map="auto", no_split_
↪module_classes=["GPTJBlock"])

output = model.generate("Hello, my name is", device=torch.device("cuda"))
print(model.tokenizer.decode(output[0].tolist()))
```

To see more details about the API, see here.

CHAPTER

THIRTEEN

DEEPSPEED TRAINING WITH BIG TRANSFORMER MODELS

Below is an example of how you can train a 6B parameter transformer model using Lightning Transformers and DeepSpeed.

The below script was tested on an 8 A100 machine.

```
import pytorch_lightning as pl
from transformers import AutoTokenizer

from lightning_transformers.task.nlp.language_modeling import␣
↪LanguageModelingDataModule, LanguageModelingTransformer

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path="gpt2")

model = LanguageModelingTransformer(
    pretrained_model_name_or_path="EleutherAI/gpt-j-6B",
    tokenizer=AutoTokenizer.from_pretrained("EleutherAI/gpt-j-6B"),
    deepspeed_sharding=True # defer initialization of the model to shard/load pre-
↪train weights
)

dm = LanguageModelingDataModule(
    batch_size=1,
    dataset_name="wikitext",
    dataset_config_name="wikitext-2-raw-v1",
    tokenizer=tokenizer,
)
trainer = pl.Trainer(accelerator="gpu", devices="auto", strategy="deepspeed_stage_3",␣
↪precision=16, max_epochs=1)
trainer.fit(model, dm)
```

If you have your own `pl.LightningModule` you can use DeepSpeed ZeRO Stage 3 parameter sharding & Transformers as well, just add this code:

```
import pytorch_lightning as pl
from transformers import T5ForConditionalGeneration
from lightning_transformers.utilities.deepspeed import enable_transformers_pretrained_
↪deepspeed_sharding


class MyModel(pl.LightningModule):

    def setup(self, stage: Optional[str] = None) -> None:
        if not hasattr(self, "ptlm"):
```

(continues on next page)

(continued from previous page)

```
            enable_transformers_pretrained_deepspeed_sharding(self)
            self.ptlm = T5ForConditionalGeneration.from_pretrained("t5-11b")
```

CHAPTER

FOURTEEN

CUSTOM DATA FILES

In most cases when training/validating/testing on custom files, you'll be able to do so without modifying any code, using the general data module classes directly.

Below we show per task how to fine-tune/validate/test on your own files per task or modify the logic within the data classes. Some tasks are more involved than others, as they may require more data processing.

14.1 Custom Subset Names (Edge Cases such as MNLI)

Some datasets, such as MNLI when loaded from the Huggingface *datasets* library, have special subset names that don't match the standard train/validation/test convention. Specifically, MNLI has two validation and two test sets, with flavors 'matched' and 'mismatched'. When using such datasets, you must manually indicate which subset names you want to use for each of train/validation/text.

An example for how to train and validate on MNLI would the the following:

```
from lightning_transformers.task.nlp.text_classification import (
    TextClassificationDataModule,
    TextClassificationTransformer,
)

dm = TextClassificationDataModule(
    batch_size=1,
    dataset_name="glue",
    dataset_config_name="mnli",
    max_length=512,
    validation_subset_name="validation_matched"
    tokenizer=tokenizer,
)
```

14.2 Language Modeling Using Your Own Files

To use custom text files, the files should contain the raw data you want to train and validate on.

During data pre-processing the text is flattened, and the model is trained and validated on context windows (block size) made from the input text. We override the dataset files, allowing us to still use the data transforms defined with the base datamodule.

Below we have defined a csv file to use as our input data.

```
text,
this is the first sentence,
this is the second sentence,
```

```
from lightning_transformers.task.nlp.language_modeling import (
    LanguageModelingDataModule,
)

dm = LanguageModelingDataModule(
    batch_size=1,
    train_file="path/train.csv",
    validation_file="/path/valid.csv"
    tokenizer=tokenizer,
)
```

14.3 Multiple Choice Using Your Own Files

To use custom text files, the files should contain the data you want to train and validate on and be in CSV or JSON format as described below.

The format varies from dataset to dataset as input columns may differ, as well as pre-processing. To make our life easier, we use the RACE dataset format and override the files that are loaded.

Below we have defined a json file to use as our input data.

```
{
    "article": "The man walked into the red house but couldn't see where the light␣
↪was.",
    "question": "What colour is the house?",
    "options": ["White", "Red", "Blue"]
    "answer": "Red"
}
```

We override the dataset files, allowing us to still use the data transforms defined with the RACE dataset.

```
from lightning_transformers.task.nlp.multiple_choice import (
    RaceMultipleChoiceDataModule,
)

dm = RaceMultipleChoiceDataModule(
    batch_size=1,
    dataset_config_name="all",
    padding=False,
    train_file="path/train.json",
    validation_file="/path/valid.json"
    tokenizer=tokenizer,
)
```

14.4 Question Answering Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files.

The format varies from dataset to dataset as input columns may differ, as well as pre-processing. To make our life easier, we use the SWAG dataset format and override the files that are loaded.

```
{
    "answers": {
        "answer_start": [1],
        "text": ["This is a test text"]
    },
    "context": "This is a test context.",
    "id": "1",
    "question": "Is this a test?",
    "title": "train test"
}
```

We override the dataset files, allowing us to still use the data transforms defined with this dataset.

```
from lightning_transformers.task.nlp.question_answering import (
    SquadDataModule,
)

dm = SquadDataModule(
    batch_size=1,
    dataset_config_name="plain_text",
    max_length=384,
    version_2_with_negative=False,
    null_score_diff_threshold=0.0,
    doc_stride=128,
    n_best_size=20,
    max_answer_length=30,
    train_file="path/train.csv",
    validation_file="/path/valid.csv"
    tokenizer=tokenizer,
)
```

14.5 Summarization Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files.

```
{
    "source": "some-body",
    "target": "some-sentence"
}
```

We override the dataset files, allowing us to still use the data transforms defined with this dataset.

```
from lightning_transformers.task.nlp.summarization import (
    XsumSummarizationDataModule,
)

dm = XsumSummarizationDataModule(
    batch_size=1,
```

(continues on next page)

(continued from previous page)

```
    max_source_length=128,
    max_target_length=128,
    train_file="path/train.csv",
    validation_file="/path/valid.csv"
    tokenizer=tokenizer,
)
```

14.6 Text Classification Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files.

The label mapping is automatically generated from the training dataset labels if no mapping is given.

```
{
    "label": "sad",
    "text": "I'm feeling quite sad and sorry for myself but I'll snap out of it soon."
}
```

```
from lightning_transformers.task.nlp.text_classification import (
    TextClassificationDataModule,
    TextClassificationTransformer,
)

dm = TextClassificationDataModule(
    batch_size=1,
    max_length=512,
    train_file="path/train.json",
    validation_file="/path/valid.json"
    tokenizer=tokenizer,
)
```

14.7 Token Classification Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files. For each token, there should be an associated label.

```
{
    "label_tags": [11, 12, 12, 21, 13, 11, 11, 21, 13, 11, 12, 13, 11, 21, 22, 11, 12,
↪ 17, 11, 21, 17, 11, 12, 12, 21, 22, 22, 13, 11, 0],
    "tokens": ["The", "European", "Commission", "said", "on", "Thursday", "it",
↪"disagreed", "with", "German", "advice", "to", "consumers"]
}
```

```
from lightning_transformers.task.nlp.token_classification import␣
↪TokenClassificationDataModule

dm = TokenClassificationDataModule(
    batch_size=1,
    task_name="ner",
    dataset_name="conll2003",
    preprocessing_num_workers=1,
```

(continues on next page)

(continued from previous page)

```
    label_all_tokens=False,
    revision="master",
    train_file="path/train.json",
    validation_file="/path/valid.json"
    tokenizer=tokenizer,
)
```

14.8 Translation Using Your Own Files

To use custom text files, the files should contain new line delimited json objects within the text files.

```
{
    "source": "example source text",
    "target": "example target text"
}
```

We override the dataset files, allowing us to still use the data transforms defined with this dataset.

```
from lightning_transformers.task.nlp.translation import WMT16TranslationDataModule

dm = WMT16TranslationDataModule(
    # WMT translation datasets: ['cs-en', 'de-en', 'fi-en', 'ro-en', 'ru-en', 'tr-en']
    dataset_config_name="ro-en",
    source_language="en",
    target_language="ro",
    max_source_length=128,
    max_target_length=128,
    train_file="path/train.json",
    validation_file="/path/valid.json"
    tokenizer=tokenizer,
)
```

CHAPTER

FIFTEEN

CUSTOMIZING DATASETS

You can use Lightning Transformers task on custom datasets by extending the base DataModule classes to implement your own data processing logic.

This is useful when you have specific data processing you'd like to apply when training/validating or testing using a task, or would like to modify how data is loaded/transformed for the model.

Currently we have examples for two tasks (one encoder, one encoder/decoder), more examples coming soon!

15.1 Language Modeling using Custom Data Processing

Below we show how to override data processing logic.

15.1.1 Extend the `LanguageModelingDataModule` base class

The base data module can be used to modify this code, and follows a simple pattern. Internally the dataset is loaded via HuggingFace Datasets, which returns an Apache Arrow Parquet Dataset. This data format is easy to transform and modify using map functions, which you'll see within the class.

```
class LanguageModelingDataModule(HFDataModule):

    def process_data(self, dataset: Dataset, stage: Optional[str] = None) -> Dataset:
        # `process_data` converting the dataset into features.
        # The dataset is pre-loaded using `load_dataset`.
        ...
        return dataset

    @staticmethod
    def tokenize_function(
        examples,
        tokenizer: Union[PreTrainedTokenizerBase],
        text_column_name: str = None,
    ):
        # tokenizes the data in a specific column using the AutoTokenizer,
        # called by `process_data`
        return tokenizer(examples[text_column_name])

    @staticmethod
    def convert_to_features(examples, block_size: int = None):
        # `process_data` calls this function to convert samples in the dataset into␣
↪features
        ...
```

(continues on next page)

(continued from previous page)

```
    @property
    def collate_fn(self) -> Callable:
        # `Describes how to collate the samples for the batch given to the model`
        return default_data_collator
```

Extend `LanguageModelingDataModule`, like this.

```
from lightning_transformers.task.nlp.language_modeling import␣
↪LanguageModelingDataModule

class MyLanguageModelingDataModule(LanguageModelingDataModule):
    ...
```

Make any changes you'd like to the dataset processing via the hooks.

Below we have the pseudo code version to show where most of the changes happened within the hooks:

```
from functools import partial

from datasets import Dataset, Optional
from transformers import PreTrainedTokenizerBase

from lightning_transformers.task.nlp.language_modeling import␣
↪LanguageModelingDataModule


class MyLanguageModelingDataModule(LanguageModelingDataModule):

    def __init__(self, tokenizer: PreTrainedTokenizerBase, *args, **kwargs):
        super().__init__(tokenizer, *args, **kwargs)
        self.tokenized_condition_term = tokenizer("This is a story: ")

    def process_data(self, dataset: Dataset, stage: Optional[str] = None) -> Dataset:
        ...
        # Pass in our additional condition term when converting to features
        convert_to_features = partial(
            self.convert_to_features,
            block_size=self.effective_block_size,
            tokenized_condition_term=self.tokenized_condition_term
        )
        ...
        return dataset

    @staticmethod
    def convert_to_features(examples, block_size: int, **kwargs):
        # Our argument is passed in via kwargs
        tokenized_condition_term = kwargs['tokenized_condition_term']

        ...
        # Add the term to the tokenized blocks of text
        result = {
            k: [tokenized_condition_term + t[i:i + block_size] for i in range(0,␣
↪total_length, block_size)]
            for k, t in concatenated_examples.items()
        }
        result["labels"] = result["input_ids"].copy()
```

(continues on next page)

(continued from previous page)

```
        return result
```

15.2 Translation using Custom Data Processing

Below we show how to override data processing logic.

15.2.1 Extend the `TranslationDataModule` base class

The base data module can be used to modify this code, and follows a simple pattern. Internally the dataset is loaded via HuggingFace Datasets, which returns an Apache Arrow Parquet Dataset. This data format is easy to transform and modify using map functions, which you'll see within the class.

```
class TranslationDataModule(Seq2SeqDataModule):

    @property
    def source_target_column_names(self) -> Tuple[str, str]:
        return self.cfg.source_language, self.cfg.target_language

...

class Seq2SeqDataModule(HFDataModule):

    def process_data(self, dataset: Dataset, stage: Optional[str] = None) -> Dataset:
        # `process_data` converting the dataset into features.
        # The dataset is pre-loaded using `load_dataset`.
        ...
        return dataset

    @property
    def source_target_column_names(self) -> Tuple[str, str]:
        return 'source', 'target'

    @staticmethod
    def convert_to_features(examples, block_size: int = None):
        # `process_data` calls this function to convert samples in the dataset into␣
↪features
        ...

    @property
    def collate_fn(self) -> Callable:
        # `Describes how to collate the samples for the batch given to the model`
        return default_data_collator
```

Extend `TranslationDataModule`, like this.

```
from lightning_transformers.task.nlp.translation import TranslationDataModule

class MyTranslationDataModule(TranslationDataModule):
    ...
```

Make any changes you'd like to the dataset processing via the hooks.

CHAPTER

SIXTEEN

CUSTOMIZING TASKS

Below we describe how you can customize the Language Modeling Task. In our example, we add weight noise when training and freeze the backbone. For the purpose of this example we freeze the backbone within the Task, however this is recommended to be done via a Callback as seen in the Freeze Embeddings Callback.

Tasks are based of a AutoModel Transformer, which handles all the internal logic when running the forward pass through the model, and the loss calculation for a specific task. Below are the steps to customize a task within the LightningModule.

1. Inherit from Lightning Transformers Base Class
2. Add custom task logic

16.1 1. Inherit from Lightning Transformers Base Class

For our example, we inherit from the Language Modeling base class.

```
from lightning_transformers.task.nlp.language_modeling import␣
↪LanguageModelingTransformer

class MyLanguageModelingTransformer(LanguageModelingTransformer):
    ...
```

Typically you'd store the file within the `lightning_transformers/task/` directory, in the appropriate task folder. In our example, we'd store our file in `lightning_transformers/task/language_modeling/custom_model.py`.

16.2 2. Add Custom Task Logic

The class follows a standard `pl.LightningModule`, thus all hooks and logic can be overridden easily. Below we override the training step to add our logic, as well as `on_fit_start` to freeze the model before training. The `LMHeadAutoModel` task provides separate keys for the backbone and the fully connected layer.

```
from lightning_transformers.task.nlp.language_modeling import␣
↪LanguageModelingTransformer

class MyLanguageModelingTransformer(LanguageModelingTransformer):

    def setup(self, stage):
        # Freeze BERT backbone
        for param in self.model.bert.parameters():
```

(continues on next page)

(continued from previous page)

```
            param.requires_grad = False

    def training_step(self, batch, batch_idx):
        loss = super().training_step(batch, batch_idx)

        # Add weight noise every training step
        with torch.no_grad():
            for param in self.model.parameters():
                param.add_(torch.randn(param.size()) * 0.1)
        return loss
```

CHAPTER

SEVENTEEN

TASK API REFERENCE

CHAPTER

EIGHTEEN

DATAMODULE API REFERENCE

CHAPTER

NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search