
HetSeq

Release 0.0.1

Yifan Ding

Mar 17, 2021

GETTING STARTED:

1	Installation	3
1.1	Create Conda Virtual Environment	3
1.2	Git Clone and Install Packages	3
1.3	Download BERT Processed File	3
1.4	Download MNIST Dataset (not required)	3
2	Distributed Setting	5
2.1	Control Parameters	5
2.2	Different Distributed Settings	5
2.3	Main Logic	6
3	Parameters	7
3.1	Overview	7
3.2	Extendable Components Parameters	7
3.3	Distributed Parameters	8
3.4	Training Parameters	9
4	Running Script	11
4.1	BERT Task	11
4.2	MNIST Task	12
4.3	Evaluate MNIST Task	13
5	Overview	15
5.1	Task	15
5.2	Dataset	15
5.3	Model	16
5.4	Optimizer	16
5.5	Learning Rate Scheduler	17
6	MNIST example	19
6.1	Task	19
6.2	Dataset	20
6.3	Model	21
6.4	Running Script	21
7	Task	23
8	Dataset	25
9	Model	27

10 Optimizer	29
11 Learning Rate Scheduler	31
12 Meters	33
13 Progress_bar	35
14 Copyright	37
15 Contact	39
16 Indices and tables	41
Index	43

HetSeq is a distributed neural network platform designed to run on Heterogeneous Infrastructure with common scientific shared file system. It can be run directly on command line with SSH or task queue submission system without privilege or any extra packages. It takes care of the data index randomization and assignment to different GPUs in the multi-node and multi-GPU setting. Users can easily extend HetSeq to many other models with minimum effort.

Note: HetSeq requires installation of [PyTorch](#) with GPU support and [NCCL](#).

INSTALLATION

1.1 Create Conda Virtual Environment

Recommend to create and activate conda virtual environment with Python 3.7.4

```
$ conda create --name hetseq  
$ conda activate hetseq  
$ conda install python=3.7.4
```

1.2 Git Clone and Install Packages

```
$ git clone https://github.com/yifding/hetseq.git  
$ cd /path/to/hetseq  
$ pip install -r requirements.txt  
$ pip install --editable .
```

1.3 Download BERT Processed File

(Required to run bert model) Download data files including training corpus, model configuration, and BPE dictionary. Test corpus from [here](#), full data from [this link](#). Download `test_DATA.zip` for test or `DATA.zip` for full run, unzip it and place the `preprocessing/` directory inside the package directory.

1.4 Download MNIST Dataset (not required)

(For mnist model) Download MNIST dataset from `torchvision`, see example [here](#).

```
from torchvision import datasets  
dataset1 = datasets.MNIST('../data', train=True, download=True)
```


DISTRIBUTED SETTING

HetSeq can be executed on single GPU on a single node, multiple GPUs on a single node, or multiple GPUs across multiple nodes. Main logic is defined at [train.py](#).

2.1 Control Parameters

`--distributed-init-method`: defines an initialization.

- `tcp://10.32.82.207:11111` (IP address:port. TCP example for multiple nodes) or
- `file:///hetseq/communicate.txt` (shared file example for multiple nodes).

`--distributed-world-size`: total number of GPUs used in the training.

`--distributed-gpus`: the number of GPUs on the current node.

`--distributed-rank`: represents the rank/index of the first GPU used on current node.

2.2 Different Distributed Settings

1. Single GPU:

```
$ --distributed-world-size 1 --device-id 1
```

2. Four GPUs on a single node:

```
$ --distributed-world-size 4
```

3. Four nodes with four GPUs each (16 GPUs in total) 10.00.123.456 is the IP address of first node and 11111 is the port number:

- 1st node

```
$ --distributed-init-method tcp://10.00.123.456:11111 --distributed-world-size 16 --  
→distributed-gpus 4 --distributed-rank 0
```

- 2nd node

```
$ --distributed-init-method tcp://10.00.123.456:11111 --distributed-world-size 16 --  
→distributed-gpus 4 --distributed-rank 4
```

- 3rd node

```
$ --distributed-init-method tcp://10.00.123.456:11111 --distributed-world-size 16 --  
↪distributed-gpus 4 --distributed-rank 8
```

- 4th node

```
$ --distributed-init-method tcp://10.00.123.456:11111 --distributed-world-size 16 --  
↪distributed-gpus 4 --distributed-rank 12
```

2.3 Main Logic

```
if args.distributed_init_method is not None:  
    assert args.distributed_gpus <= torch.cuda.device_count()  
  
    if args.distributed_gpus > 1 and not args.distributed_no_spawn:  
        start_rank = args.distributed_rank  
        args.distributed_rank = None # assign automatically  
        torch.multiprocessing.spawn(  
            fn=distributed_main,  
            args=(args, start_rank),  
            nprocs=args.distributed_gpus,  
        )  
    else:  
        distributed_main(args.device_id, args)  
  
elif args.distributed_world_size > 1:  
    assert args.distributed_world_size <= torch.cuda.device_count()  
    port = random.randint(10000, 20000)  
    args.distributed_init_method = 'tcp://localhost:{port}'.format(port=port)  
    args.distributed_rank = None # set based on device id  
    torch.multiprocessing.spawn(  
        fn=distributed_main,  
        args=(args, ),  
        nprocs=args.distributed_world_size,  
    )  
else:  
    main(args)
```

PARAMETERS

3.1 Overview

To run HetSeq, almost all the parameters are passed through command line processed by `argparse`. Those parameters can be grouped into several clusters:

- **Extendable Components Parameters:** including `task`, `optimizer`, and `lr_scheduler`;
- **Distributed Parameters:** to set up distributed training environments;
- **Training Parameters:** other important parameters to control stop criteria, logging information, checkpoints and etc.

Here we are going to explain most parameters in details.

3.2 Extendable Components Parameters

- **Task:** `--task`: Application name, its corresponding class defines major parts of the application. Currently support `bert` and `mnist`, can be extended to other models.

- **`--task bert`:**

- Extra parameters for `bert` task:**

- * `--data`: Dataset directory or file to be loaded in the corresponding task.
 - * `--config_file`: Configuration file of BERT model, example can be found [here](#)
 - * `--dict`: PATH of BPE dictionary for BERT model. Typically it has ~30,000 tokens.
 - * `--max_pred_length`: max number of tokens in a sentence, 512 by default.
 - * `--num_file`: number of input files for training, used with `--data` to debug. 0 by default to use all the data files.

- **`--task mnist`:**

- Extra parameters for `mnist` task:**

- * `--data`: Dataset directory or file to be loaded in the corresponding task, compatible with `torchvision.datasets.MNIST(path, train=True, download=True)`.

- **Optimizer:** `--optimizer`: Optimizer defined in HetSeq is based on `torch.optim.Optimizer` with extra gradient and learning rate manipulation function. Currently support `adam` and `adadelat` which can be extended to many other optimizers.

- **`--optimizer adam`:**

Extra parameters for adam optimizer: Fixed Weight Decay Regularization in Adam.

- * `--adam-betas`: betas to control momentum and velocity. Default='(0.9, 0.999)'.
- * `--adam-eps`: epsilon for avoiding deviding by 0. Default=1e-8.
- * `--weight-decay`: weight decay. 0 by default.

- --optimizer adadelta:**Extra parameters for adadelta optimizer:**

- * `--adadelta_rho`: Default=0.9.
- * `--adadelta_eps`: epsilon for avoiding deviding by 0. Default=1e-6.
- * `--adadelta_weight_decay`: 0 by default.

- **Lr_scheduler**: `--lr_scheduler`: Learning rate scheduler defined in HetSeq customized to consider stop criteria end-learning-rate, total-num-update and warmup-updates. Currently support Polynomial Decay Scheduler.

- --optimizer PolynomialDecayScheduler:**Extra parameters for PolynomialDecayScheduler:**

- * `--force-anneal`: force annealing at specified epoch, by default not existed.
- * `--power`: decay power. 1.0 by default.
- * `--warmup-updates`: warmup the learning rate linearly for the first N updates, 0 by default.
- * `--total-num-update`: total number of update steps until learning rate decay to `--end-learning-rate`, 10000 by default.
- * `--end-learning-rate`: learning rate when traing stops. 0 by default.

3.3 Distributed Parameters

Distributed parameters play a key role in HetSeq to set up the distributed training environments, it defines the number of nodes, number of GPUs, communication methods and etc.

- `--fast-stat-sync`: Enable fast sync of stats between nodes, this hardcodes to sync only some default stats from `logging_output`.
- `--device-id`: index of single GPU used in the training. 0 by default.

`torch.nn.parallel.distributed` import `DistributedDataParallel` related parameters, see [document](#) for more informaiton. Our implementation consider input and put tensors on the same device.

- `--bucket-cap-mb`: 25 by default
- `--find-unused-parameters`: False by default

`torch.distributed.init_process_group` related parameters, control the main environment of distributed training. See [document](#) for more information.

- `--ddp-backend`: distributed data parallel backend, currently only support `c10d` with NCCL to communicate between GPUs. Default: `'c10d'`.
- `--distributed-init-method`: initial methods to communicate between GPUs. Default `None`.
- `--distributed-world-size`: total number of GPUs/processes in the distributed setting. Defalut: `max(1, torch.cuda.device_count())`.

- `--distributed-rank`: rank of the current GPU, 0 by default.

3.4 Training Parameters

- `--max-epoch`: maximum epoches allowed in the training. 0 by default.
- `--max-update`: maximum number of updates allowed in the training. 0 by default.
- `--required-batch-size-multiple`: check the batch size is the multiple times of the given number. 1 by default.
- `--update-freq`: update parameters every N_i batches, when in epoch i . 1 by default.
- `--max-tokens`: maximum number of tokens of a batch, not assigned.
- `--max-sentences`: maximum number of sentences/images/instances of a batch (batch size), not assigned.

Note: `--max-tokens` or `--max-sentences` must be assigned in the parameter settings.

- `--train-subset`: string to store training subset, `train` by default.
- `--num-workers`: number of threads used in the data loading process.
- `--save-interval-updates`: save a checkpoint (and validate) every N updates, 0 by default.
- `--seed`: onlu seed in the training process to control all the possible random steps (e.g. in `torch`, `numpy` and `random`). 19940802 by default.
- `--log-interval`: log progress every N batches (when progress bar is disabled), 1 by default.
- `--log-format`: log format to use, choices=['none', 'simple'], `simple` by default.

RUNNING SCRIPT

4.1 BERT Task

1. Single GPU

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/train.py \
    --task bert --data ${DIST}/preprocessing/test_128/ \
    --dict ${DIST}/preprocessing/uncased_L-12_H-768_A-12/vocab.txt \
    --config_file ${DIST}/preprocessing/uncased_L-12_H-768_A-12/bert_config.json \
    --max-sentences 32 --fast-stat-sync --max-update 900000 --update-freq 4 \
    --valid-subset test --num-workers 4 \
    --warmup-updates 10000 --total-num-update 1000000 --lr 0.0001 \
    --weight-decay 0.01 --distributed-world-size 1 \
    --device-id 0 --save-dir bert_single_gpu
```

2. Multiple GPU on a single Node, examples are all four GPUs on one Node.

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/train.py \
    --task bert --data ${DIST}/preprocessing/test_128/ \
    --dict ${DIST}/preprocessing/uncased_L-12_H-768_A-12/vocab.txt \
    --config_file ${DIST}/preprocessing/uncased_L-12_H-768_A-12/bert_config.json \
    --max-sentences 32 --fast-stat-sync --max-update 900000 --update-freq 4 \
    --valid-subset test --num-workers 4 \
    --warmup-updates 10000 --total-num-update 1000000 --lr 0.0001 \
    --weight-decay 0.01 \
    --save-dir bert_node1gpu4
```

3. Multiple GPUs on multiple nodes, examples are two nodes with four GPUs each.

- on the main node

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/train.py \
    --task bert --data ${DIST}/preprocessing/test_128/ \
    --dict ${DIST}/preprocessing/uncased_L-12_H-768_A-12/vocab.txt \
    --config_file ${DIST}/preprocessing/uncased_L-12_H-768_A-12/bert_config.json \
    --max-sentences 32 --fast-stat-sync --max-update 900000 --update-freq 4 \
    --valid-subset test --num-workers 4 \
    --warmup-updates 10000 --total-num-update 1000000 --lr 0.0001 \
```

(continues on next page)

(continued from previous page)

```
--weight-decay 0.01 --save-dir bert_node2gpu4 \
--distributed-init-method tcp://10.00.123.456:11111 \
--distributed-world-size 8 --distributed-gpus 4 --distributed-rank 0
```

- on the other second node

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/train.py \
  --task bert --data ${DIST}/preprocessing/test_128/ \
  --dict ${DIST}/preprocessing/uncased_L-12_H-768_A-12/vocab.txt \
  --config_file ${DIST}/preprocessing/uncased_L-12_H-768_A-12/bert_config.json \
  --max-sentences 32 --fast-stat-sync --max-update 900000 --update-freq 4 \
  --valid-subset test --num-workers 4 \
  --warmup-updates 10000 --total-num-update 1000000 --lr 0.0001 \
  --weight-decay 0.01 \
  --distributed-init-method tcp://10.00.123.456:11111 \
  --distributed-world-size 8 --distributed-gpus 4 --distributed-rank 4
```

4.2 MNIST Task

1. Single GPU

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/train.py \
  --task mnist --optimizer adadelta --lr-scheduler PolynomialDecayScheduler \
  --data ${DIST} --clip-norm 100 \
  --max-sentences 64 --fast-stat-sync --max-epoch 20 --update-freq 1 \
  --valid-subset test --num-workers 4 \
  --warmup-updates 0 --total-num-update 50000 --lr 1.01 \
  --distributed-world-size 1 --device-id 0 --save-dir mnist_single_node
```

2. Multiple GPU on a single Node, examples are all four GPUs on one Node.

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/train.py \
  --task mnist --optimizer adadelta --lr-scheduler PolynomialDecayScheduler \
  --data ${DIST} --clip-norm 100 \
  --max-sentences 64 --fast-stat-sync --max-epoch 20 --update-freq 1 \
  --valid-subset test --num-workers 4 \
  --warmup-updates 0 --total-num-update 50000 --lr 1.01 \
  --save-dir mnist_node1gpu4
```

3. Multiple GPUs on multiple nodes, examples are two nodes with four GPUs each.

- on the main node

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/train.py \
  --task mnist --optimizer adadelta --lr-scheduler PolynomialDecayScheduler \
  --data ${DIST} --clip-norm 100 \
  --max-sentences 64 --fast-stat-sync --max-epoch 20 --update-freq 1 \
  --valid-subset test --num-workers 4 \
  --warmup-updates 0 --total-num-update 50000 --lr 1.01 \
```

(continues on next page)

(continued from previous page)

```
--save-dir mnist_node2gpu4 \
--distributed-init-method tcp://10.00.123.456:11111 \
--distributed-world-size 8 --distributed-gpus 4 --distributed-rank 0
```

- on the other second node

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/train.py \
  --task mnist --optimizer adadelata --lr-scheduler PolynomialDecayScheduler \
  --data ${DIST} --clip-norm 100 \
  --max-sentences 64 --fast-stat-sync --max-epoch 20 --update-freq 1 \
  --valid-subset test --num-workers 4 \
  --warmup-updates 0 --total-num-update 50000 --lr 1.01 \
  --distributed-init-method tcp://10.00.123.456:11111 \
  --distributed-world-size 8 --distributed-gpus 4 --distributed-rank 4
```

4.3 Evaluate MNIST Task

```
$ DIST=~ /hetseq
$ python3 ${DIST}/hetseq/eval_mnist.py --model_ckpt /path/to/check/point --mnist_dir $
↪ ${DIST}
```


OVERVIEW

To extend HetSeq to another model, one needs to define a new *Task* with corresponding *Model*, *Dataset*, *Optimizer* and *Learning Rate Scheduler*. A MNIST example is given with all the extended classes. Pre-defined optimizers, Learning Rate Scheduler, datasets and models can be reused in other applications.

5.1 Task

For each individual application, task is the basic unit. Defined by class `Task` in `Task.py`. datasets is stored and load in a dictionary manner. Define a child class of `Task` to define a new task, necessary function is to define `Model` (in `def build_model`), `Dataset` (in `def load_dataset`).

```
class Task(object):
    def __init__(self, args):
        self.args = args
        self.datasets = {}
        self.dataset_to_epoch_iter = {}

    def build_model(self, args):
        raise NotImplementedError

    def load_dataset(self, split, **kwargs):
        """Load a given dataset split.
        Args:
            split (str): name of the split (e.g., train, valid, test)
        """
        raise NotImplementedError
```

5.2 Dataset

Dataset should be defined as a child class of `torch.utils.data.Dataset` to be compatible with `torch.utils.data.DataLoader`.

- `__getitem__` (get item),
- `__len__` (total length of the dataset),
- `ordered_indices` (index used to split and assignment to different GPUs,
- `np.arange(len(self))`),
- `num_tokens` (total tokens in a instance, 1 for image model),

- `collater` (collater function to combined the output of `__getitem__`, typically use `torch.utils.data.dataloader.default_collate`)
- `set_epoch` (pass) function in the class.

See following MNIST example for more information.

Note: In our implementation, each process/GPU has its own dataset and dataloader. When dataset is small (like MNIST example), the dataset can be put into `__init__` function. However, if the dataset is large (like BERT example or ImageNet), the dataset can not be loaded into memory at once, then the loading process should be defined inside `__getitem__` function.

5.3 Model

Model should be defined as a child class of `torch.nn.Module`. By default, the model should output a loss function. This is compatible with the `def train_step(self, sample, model, optimizer, ignore_grad=False)` function inside `class Task`. One can change the logic but need to fit the `train_step`.

5.4 Optimizer

Optimizer in distributed data parallel (DDP) has to consider manipulate gradients and learning rates. In our implementation, `optimizer` (`class _Optimizer(object):`) is defined as a higher level class than `torch.optim.Optimizer` to include other parameters to be recorded. For example, the Adam optimizer provided in HetSeq, has initial learning rate: `lr`, `beta1` and `beta2`: `betas`, `epsilon` `eps` to avoid normalize by 0 and `weight_decay`.

```
class _Adam(_Optimizer):
    def __init__(self, args, params):
        super().__init__(args)

        self._optimizer = Adam(params, **self.optimizer_config)

    @property
    def optimizer_config(self):
        """
        Return a kwarg dictionary that will be used to override optimizer
        args stored in checkpoints. This allows us to load a checkpoint and
        resume training using a different set of optimizer args, e.g., with a
        different learning rate.
        """
        return {
            'lr': self.args.lr[0],
            'betas': eval(self.args.adam_betas),
            'eps': self.args.adam_eps,
            'weight_decay': self.args.weight_decay,
        }
```

5.5 Learning Rate Scheduler

In HetSeq, common `PolynomialDecayScheduler` is provided and compatible to BERT model and MNIST model. Other learning rate scheduler can be easily extended by providing `step_update` and `step` function.

MNIST EXAMPLE

MNIST example is adapted from [PyTorch mnist example](#). It is convolutional neural network model for image classification. We adapt the original datasets, model and data loader to be compatible to HetSeq.

6.1 Task

```
class MNISTTask(Task):
    def __init__(self, args):
        super(MNISTTask, self).__init__(args)

    @classmethod
    def setup_task(cls, args, **kwargs):
        """Setup the task (e.g., load dictionaries).
        Args:
            args (argparse.Namespace): parsed command-line arguments
        """
        return cls(args)

    def build_model(self, args):
        model = MNISTNet()
        return model

    def load_dataset(self, split, **kwargs):
        """Load a given dataset split.
        Args:
            split (str): name of the split (e.g., train, valid, test)
        """
        path = self.args.data

        if not os.path.exists(path):
            raise FileNotFoundError(
                "Dataset not found: {}".format(path)
            )

        if os.path.isdir(path):
            if os.path.exists(os.path.join(path, 'MNIST/processed/')):
                path = os.path.join(path, 'MNIST/processed/')
            elif os.path.basename(os.path.normpath(path)) != 'processed':
                datasets.MNIST(path, train=True, download=True)
                path = os.path.join(path, 'MNIST/processed/')

        files = [os.path.join(path, f) for f in os.listdir(path)] if os.path.
        isdir(path) else [path]
```

(continues on next page)

(continued from previous page)

```

files = sorted([f for f in files if split in f])

assert len(files) == 1, "no suitable file in split {}".format(split)

dataset = MNISTDataset(files[0])

print('| loaded {} sentences from: {}'.format(len(dataset), path), flush=True)

self.datasets[split] = dataset
print('| loading finished')

```

6.2 Dataset

```

class MNISTDataset(torch.utils.data.Dataset):
    def __init__(self, path):
        self.data = None
        self.path = path
        self.read_data(self.path)
        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])

    def read_data(self, path):
        self.data = torch.load(path)
        self._len = len(self.data[0])
        self.image = self.data[0]
        self.label = self.data[1]

    def __getitem__(self, index):
        img, target = self.image[index], int(self.label[index])
        img = Image.fromarray(img.numpy(), mode='L')
        img = self.transform(img)
        return img, target

    def __len__(self):
        return self._len

    def ordered_indices(self):
        """Return an ordered list of indices. Batches will be constructed_
↪based
        on this order."""
        return np.arange(len(self))

    def num_tokens(self, index: int):
        return 1

    def collater(self, samples):
        if len(samples) == 0:
            return None
        else:
            return default_collate(samples)

```

(continues on next page)

(continued from previous page)

```
def set_epoch(self, epoch):  
    pass
```

6.3 Model

```
class MNISTNet(nn.Module):  
    def __init__(self):  
        super(MNISTNet, self).__init__()  
        self.conv1 = nn.Conv2d(1, 32, 3, 1)  
        self.conv2 = nn.Conv2d(32, 64, 3, 1)  
        self.dropout1 = nn.Dropout2d(0.25)  
        self.dropout2 = nn.Dropout2d(0.5)  
        self.fc1 = nn.Linear(9216, 128)  
        self.fc2 = nn.Linear(128, 10)  
  
    def forward(self, x, target, eval=False):  
        x = self.conv1(x)  
        x = F.relu(x)  
        x = self.conv2(x)  
        x = F.relu(x)  
        x = F.max_pool2d(x, 2)  
        x = self.dropout1(x)  
        x = torch.flatten(x, 1)  
        x = self.fc1(x)  
        x = F.relu(x)  
        x = self.dropout2(x)  
        x = self.fc2(x)  
        output = F.log_softmax(x, dim=1)  
        loss = F.nll_loss(output, target)  
        return loss
```

6.4 Running Script

See *running script* for details.

TASK

class `hetseq.tasks.Task` (*args*)

Tasks store dictionaries and provide helpers for loading/iterating over Datasets, initializing the Model/Criterion and calculating the loss.

class `hetseq.tasks.LanguageModelingTask` (*args*, *dictionary*)

Train a language model, currently support BERT. Args:

args: parsed from command line dictionary: the BPE dictionary for the input of the language model

DATASET

```
class hetseq.data.h5pyDataset.Berth5pyData (*args: Any, **kwargs: Any)
class hetseq.data.h5pyDataset.ConBerth5pyData (*args: Any, **kwargs: Any)
class hetseq.data.mnist_dataset.MNISTDataset (*args: Any, **kwargs: Any)
```


MODEL

```
class hetseq.bert_modeling.BertConfig(vocab_size_or_config_json_file,          hid-
                                     den_size=768,          num_hidden_layers=12,
                                     num_attention_heads=12,  intermediate_size=3072,
                                     hidden_act='gelu',      hidden_dropout_prob=0.1,
                                     attention_probs_dropout_prob=0.1,
                                     max_position_embeddings=512, type_vocab_size=2,
                                     initializer_range=0.02)
```

Configuration class to store the configuration of a *BertModel*.

```
class hetseq.bert_modeling.BertForPreTraining(*args: Any, **kwargs: Any)
```

BERT model with pre-training heads. This module comprises the BERT model followed by the two pre-training heads:

- the masked language modeling head, and
- the next sentence classification head.

Params: config: a BertConfig class instance with the configuration to build a new model.

Inputs:

input_ids: a torch.LongTensor of shape [batch_size, sequence_length] with the word token indices in the vocabulary(see the tokens preprocessing logic in the scripts *extract_features.py*, *run_classifier.py* and *run_squad.py*)

token_type_ids: an optional torch.LongTensor of shape [batch_size, sequence_length] with the token types indices selected in [0, 1]. Type 0 corresponds to a *sentence A* and type 1 corresponds to a *sentence B* token (see BERT paper for more details).

attention_mask: an optional torch.LongTensor of shape [batch_size, sequence_length] with indices selected in [0, 1]. It's a mask to be used if the input sequence length is smaller than the max input sequence length in the current batch. It's the mask that we typically use for attention when a batch has varying length sentences.

masked_lm_labels: optional masked language modeling labels: torch.LongTensor of shape [batch_size, sequence_length] with indices selected in [-1, 0, ..., vocab_size]. All labels set to -1 are ignored (masked), the loss is only computed for the labels set in [0, ..., vocab_size]

next_sentence_label: optional next sentence classification loss: torch.LongTensor of shape [batch_size] with indices selected in [0, 1]. 0 => next sentence is the continuation, 1 => next sentence is a random sentence.

Outputs:

if masked_lm_labels and next_sentence_label are not None: Outputs the total_loss which is the sum of the masked language modeling loss and the next sentence classification loss.

if *masked_lm_labels* or *next_sentence_label* is *None*: Outputs a tuple comprising - the masked language modeling logits of shape [batch_size, sequence_length, vocab_size], and - the next sentence classification logits of shape [batch_size, 2].

Example usage: `python # Already been converted into WordPiece token ids input_ids = torch.LongTensor([[31, 51, 99], [15, 5, 0]]) input_mask = torch.LongTensor([[1, 1, 1], [1, 1, 0]]) token_type_ids = torch.LongTensor([[0, 0, 1], [0, 1, 0]])`

`config = BertConfig(vocab_size_or_config_json_file=32000, hidden_size=768, num_hidden_layers=12, num_attention_heads=12, intermediate_size=3072)`

`model = BertForPreTraining(config) masked_lm_logits_scores, seq_relationship_logits = model(input_ids, token_type_ids, input_mask)`

OPTIMIZER

class hetseq.optim.**Adam** (*args: Any, **kwargs: Any)

Implements Adam algorithm. This implementation is modified from torch.optim.Adam based on: *Fixed Weight Decay Regularization in Adam* (see <https://arxiv.org/abs/1711.05101>) It has been proposed in [Adam: A Method for Stochastic Optimization](#). Arguments:

params (iterable): iterable of parameters to optimize or dicts defining parameter groups

lr (float, optional): learning rate (default: 1e-3) **betas (Tuple[float, float], optional):** coefficients used for computing

running averages of gradient and its square (default: (0.9, 0.999))

eps (float, optional): term added to the denominator to improve numerical stability (default: 1e-8)

weight_decay (float, optional): weight decay (L2 penalty) (default: 0) **amsgrad (boolean, optional):** whether to use the AMSGrad variant of this

algorithm from the paper [On the Convergence of Adam and Beyond](#)

class hetseq.optim.**Adadelta** (*args: Any, **kwargs: Any)

Implements Adadelta algorithm. It has been proposed in [ADADELTA: An Adaptive Learning Rate Method](#). Arguments:

params (iterable): iterable of parameters to optimize or dicts defining parameter groups

rho (float, optional): coefficient used for computing a running average of squared gradients (default: 0.9)

eps (float, optional): term added to the denominator to improve numerical stability (default: 1e-6)

lr (float, optional): coefficient that scale delta before it is applied to the parameters (default: 1.0)

weight_decay (float, optional): weight decay (L2 penalty) (default: 0)

LEARNING RATE SCHEDULER

```
class hetseq.lr_scheduler.PolynomialDecayScheduler (args, optimizer)  
    Decay the LR on a fixed schedule.
```


METERS

```
class hetseq.meters.AverageMeter  
    Computes and stores the average and current value  
  
class hetseq.meters.TimeMeter (init=0)  
    Computes the average occurrence of some event per second  
  
class hetseq.meters.StopwatchMeter  
    Computes the sum/avg duration of some event in seconds
```


PROGRESS_BAR

```
class hetseq.progress_bar.progress_bar (iterable, epoch=None, prefix=None)  
    Abstract class for progress bars.  
  
class hetseq.progress_bar.noop_progress_bar (iterable, epoch=None, prefix=None)  
    No logging.  
  
class hetseq.progress_bar.simple_progress_bar (iterable, epoch=None, prefix=None,  
                                                log_interval=1000)  
    A minimal logger for non-TTY environments.
```


COPYRIGHT

HetSeq: Distributed GPU Training on Heterogeneous Infrastructure.

Yifan Ding, Nicholas Botzer, Tim Weninger (2020).

[Project Web](#) and [Project GitHub](#)

[@Weninger Lab](#), Department of Computer Science & Engineering, University of Notre Dame.

CHAPTER
FIFTEEN

CONTACT

Welcome to contact us if any question.

Yifan Ding: yding4@nd.edu

Tim Weninger: tweninge@nd.edu

INDICES AND TABLES

- `genindex`
- `search`

A

Adadelta (*class in hetseq.optim*), 29

Adam (*class in hetseq.optim*), 29

AverageMeter (*class in hetseq.meters*), 33

B

BertConfig (*class in hetseq.bert_modeling*), 27

BertForPreTraining (*class in hetseq.bert_modeling*), 27

BertH5pyData (*class in hetseq.data.h5pyDataset*), 25

C

ConBertH5pyData (*class in hetseq.data.h5pyDataset*), 25

L

LanguageModelingTask (*class in hetseq.tasks*), 23

M

MNISTDataset (*class in hetseq.data.mnist_dataset*), 25

N

noop_progress_bar (*class in hetseq.progress_bar*), 35

P

PolynomialDecayScheduler (*class in hetseq.lr_scheduler*), 31

progress_bar (*class in hetseq.progress_bar*), 35

S

simple_progress_bar (*class in hetseq.progress_bar*), 35

StopwatchMeter (*class in hetseq.meters*), 33

T

Task (*class in hetseq.tasks*), 23

TimeMeter (*class in hetseq.meters*), 33