Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Ottimizzazione e Scalabilità dei Modelli con DeepSpeed: Implementazione di Vision Transformers su minGPT

Francesco Gargiulo, Antonio Francesco Gentile, Emilio Greco

RT- ICAR-NA-25-02 **Gennaio 2025**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 8-9C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.icar.cnr.it*

Premessa

Questo rapporto tecnico analizza l'ottimizzazione e la scalabilità dei modelli di deep learning mediante l'implementazione delle tecniche avanzate offerte dalla libreria **DeepSpeed**[1]. Attraverso un caso pratico basato su **minGPT**[2], un'implementazione semplificata dei modelli **Transformer**[3], vengono illustrate strategie per affrontare le sfide legate alla crescente complessità dei modelli e all'incremento del fabbisogno di risorse computazionali. In particolare, vengono esplorati metodi per ottimizzare l'uso della memoria durante l'addestramento, accelerare i calcoli e ridurre l'overhead computazionale, migliorando l'efficienza dell'addestramento distribuito su infrastrutture multi-GPU.

L'analisi si concentra sull'applicazione di tecniche come il ricalcolo selettivo delle attivazioni, l'**addestramento a precisione mista**[4] e l'ottimizzazione con **ZeRO**[5], dimostrando la loro utilità per l'addestramento di modelli **Vision Transformers**[6] su larga scala. Questo lavoro fornisce un approccio pratico e replicabile per sfruttare le infrastrutture computazionali avanzate, garantendo al contempo prestazioni elevate e un'efficienza ottimale.

Questo rapporto tecnico fa parte di una collana di documenti tecnici volti a fornire utili informazioni per l'uso efficiente e sostenibile di infrastrutture di calcolo per l'IA basate su architetture multi-GPU. In particolare, questo lavoro è stato reso possibile grazie alle attività realizzate nell'ambito del progetto "Humanities and Cultural Heritage Italian Open Science Cloud – H2IOSC," finanziato dall'Unione europea - NextGenerationEU nell'ambito del PNRR Missione 4, "Istruzione e Ricerca" - Componente 2, "Dalla ricerca all'impresa" - Linea di investimento 3.1, "Fondo per la realizzazione di un sistema integrato di infrastrutture di ricerca e innovazione", decreto di concessione del finanziamento prot. MUR n. 112 del 20-06-2022 (CUP B63C22000730005).
I corsi di formazione forniti da NVIDIA Academy, le competenze interne all'ICAR e la documentazione acquisita attraverso vari canali, hanno permesso di approfondire e applicare tecniche all'avanguardia nel calcolo distribuito consentendo di esplorare e formalizzare modalità operative e casi d'uso delle risorse di calcolo recentemente acquistate dall'ICAR CNR, per il supporto alle ricerche avanzate condotte nell'ambito di infrastrutture per l'AI basate sul paradigma multi-GPU.

---

[1] **DeepSpeed**: https://www.deepspeed.ai
[2] **minGPT**: https://github.com/karpathy/minGPT
[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. https://arxiv.org/abs/1706.03762
[4] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., & Wu, H. (2018). **"Mixed Precision Training"**. *Proceedings of the International Conference on Learning Representations (ICLR).* https://arxiv.org/abs/1710.03740
[5] Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). **"ZeRO: Memory Optimization Towards Training A Trillion Parameter Models"**. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC).*https://arxiv.org/abs/1910.02054
[6] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). **An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale**. *International Conference on Learning Representations (ICLR).* https://arxiv.org/abs/2010.11929

Introduzione

L'obiettivo di questa relazione tecnica è illustrare il processo di ottimizzazione e scalabilità dei modelli di deep learning su larga scala utilizzando la libreria **DeepSpeed**. In particolare, il lavoro si concentra sull'implementazione di una pipeline per i **Vision Transformers** basata su **minGPT**, un'implementazione minimale dei Transformer proposta da Karpathy.

Il progetto mira a dimostrare come utilizzare DeepSpeed per integrare tecniche avanzate di ottimizzazione della memoria e del calcolo, fondamentali per affrontare le sfide poste dall'addestramento di modelli su larga scala. Tra le funzionalità principali implementate vi sono:

- **Activation Checkpointing**: una tecnica per ottimizzare l'uso della memoria attraverso il salvataggio e il ricalcolo selettivo delle attivazioni durante il backpropagation, riducendo il consumo di memoria senza compromettere significativamente i tempi di calcolo.
- **Mixed Precision Training**: un approccio che combina precisioni FP16 e FP32 per accelerare il calcolo e ottimizzare l'utilizzo della memoria delle GPU.
- **ZeRO Redundancy Optimizer (ZeRO)**: un ottimizzatore che riduce drasticamente il consumo di memoria durante l'addestramento distribuito, permettendo di addestrare modelli di dimensioni molto superiori rispetto a quelli gestibili con approcci convenzionali.

Per semplificare l'implementazione e focalizzarsi sulle tecniche di ottimizzazione, è stato scelto **minGPT**, un framework leggero che, pur non massimizzando le prestazioni, rappresenta un punto di partenza ideale per sperimentare con i modelli Transformer e il loro addestramento.

Gli obiettivi di questo lavoro si possono riassumere in tre punti:

1. **Migrazione a DeepSpeed**. Il nostro primo obiettivo consiste nel convertire una pipeline di addestramento standalone in PyTorch per sfruttare le ottimizzazioni offerte da DeepSpeed, consentendo l'addestramento distribuito su un cluster composto da due server.
2. **Ottimizzazione della Memoria**. Il secondo obbiettivo che ci proponiamo è quello di applicare ed integrare funzionalità avanzate come Mixed Precision Training, Activation Checkpointing e ZeRO per gestire in modo efficiente l'uso delle risorse.
3. **Scalabilità**. In ultimo dimostrare l'efficacia delle tecniche implementate nel supportare modelli di dimensioni significativamente superiori nella dimensione del modello addestrato.

Il processo di implementazione è stato strutturato per guidare il lettore attraverso una serie di modifiche incrementali ai file chiave, accompagnate da test e verifiche per garantire il corretto funzionamento del sistema.

Introduzione a DeepSpeed

DeepSpeed è una libreria avanzata di ottimizzazione del deep learning sviluppata da Microsoft Research, progettata per affrontare le sfide legate all'addestramento di modelli di intelligenza artificiale su larga scala. Questa libreria si propone di migliorare sia le prestazioni che la scalabilità dei modelli, consentendo agli sviluppatori di sfruttare appieno le risorse computazionali disponibili, anche in presenza di hardware limitato.

DeepSpeed si distingue per la capacità di ottimizzare l'uso della memoria, permettendo l'addestramento di modelli significativamente più grandi rispetto ai limiti convenzionali. Inoltre, offre una scalabilità elevata grazie al supporto per l'addestramento distribuito, consentendo di distribuire il carico computazionale su più GPU o interi cluster di server. Le API intuitive di DeepSpeed rendono semplice integrare queste ottimizzazioni nei flussi di lavoro esistenti, accelerando lo sviluppo di soluzioni basate sull'intelligenza artificiale.

In questo lavoro tecnico verranno implementati e analizzati tre metodi principali offerti da DeepSpeed. L'activation checkpointing è una tecnica progettata per ridurre il consumo di memoria durante l'addestramento, attraverso un sistema che salva selettivamente alcune attivazioni e le ricalcola solo quando necessario, minimizzando il sovraccarico computazionale. Il mixed precision training, che combina calcoli in precisione FP16 e FP32, permette di accelerare l'addestramento riducendo al contempo l'uso della memoria, mantenendo però la precisione del modello. ZeRO (Zero Redundancy Optimizer), un'altra innovazione chiave di DeepSpeed, è un ottimizzatore che distribuisce i parametri del modello, gli stati dell'ottimizzatore e i gradienti su più dispositivi, riducendo drasticamente i requisiti di memoria e migliorando l'efficienza dell'addestramento distribuito.

L'uso di DeepSpeed in questo lavoro è finalizzato non solo a migliorare l'efficienza dell'addestramento, ma anche a esplorare le tecniche avanzate per ottimizzare e scalare modelli complessi, con particolare attenzione all'implementazione di Vision Transformers attraverso l'adattamento della libreria minGPT. Questo approccio permette di coniugare innovazione tecnologica e praticità, dimostrando come sfruttare le potenzialità di DeepSpeed per affrontare le sfide dell'addestramento su larga scala.

I file su cui lavoreremo per migliorare il nostro processo di addestramento sono: **runStartingPoint.py** che rappresenta lo script principale, **trainer.py** gestisce il ciclo di addestramento ed **ds_config_basic.json** file di configurazione della libreria DeepSpeed.

Implementazione script di addestramento del modello GPT

Il codice di sempio presentato di seguito, **runStartingPoint.py**, utilizza PyTorch per implementare una pipeline di addestramento per un modello GPT applicato al dataset CIFAR-10. Lo script funziona nel seguente modo: dopo aver configurato gli argomenti per la distribuzione e aver impostato i parametri iniziali, il dataset CIFAR-10 viene scaricato e pre-processato. Le immagini vengono trasformate in rappresentazioni numeriche tramite un clustering K-means, che crea un vocabolario compatto di dimensioni finite. Successivamente, il modello GPT viene configurato con 12 layer, 8 head e un embedding di dimensione 256, adattandolo al task di classificazione.
Il trainer viene configurato per eseguire l'addestramento con un tasso di apprendimento di 3e-3, utilizzando un warm-up iniziale e un decadimento del learning rate. L'addestramento viene eseguito per 2 epoche su batch di dimensione 1, i pesi finali del modello vengono salvati per un utilizzo futuro. Questo processo rappresenta il punto di partenza per applicare tecniche avanzate di parallelizzazione e ottimizzazione con DeepSpeed, come il checkpointing delle attivazioni, il training a precisione mista e ZeRO.

runStartingPoint.py

```
import numpy as np
import argparse
import torchvision
import torch
from torch.utils.data import Dataset
import logging
from mingpt.utils import set_seed
from mingpt.utils import ImageDataset, TrainerConfig, kmeans
from mingpt.model import GPT, GPTConfig, GPT1Config
from mingpt.trainer import Trainer

def add_argument():
    parser = argparse.ArgumentParser(description='CIFAR')
    parser.add_argument('--local_rank', type=int, default=-1,
                        help='local rank passed from distributed launcher')

    args = parser.parse_args()
    return args

args = add_argument()

logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s -   %(message)s",
        datefmt="%m/%d/%Y %H:%M:%S",
        level=logging.INFO,
)
set_seed(42)

# get the data CIFAR10
root = './'
train_data = torchvision.datasets.CIFAR10(root, train=True, transform=None, target_transform=None, download=True)
test_data  = torchvision.datasets.CIFAR10(root, train=False, transform=None, target_transform=None, download=True)
print(len(train_data), len(test_data))

# get random 5 pixels per image and stack them all up as rgb values to get half a million random pixels
pluck_rgb = lambda x: torch.from_numpy(np.array(x)).view(32*32, 3)[torch.randperm(32*32)[:5], :]
px = torch.cat([pluck_rgb(x) for x, y in train_data], dim=0).float()
```

```
# run kmeans to get our codebook
ncluster = 512
with torch.no_grad():
    C = kmeans(px, ncluster, niter=8)

train_dataset = ImageDataset(train_data, C)
test_dataset = ImageDataset(test_data, C)

# we'll do something a bit smaller
mconf = GPTConfig(train_dataset.vocab_size, train_dataset.block_size,
                  embd_pdrop=0.0, resid_pdrop=0.0, attn_pdrop=0.0,
                  n_layer=12, n_head=8, n_embd=256)
model = GPT(mconf)

tokens_per_epoch = len(train_data) * train_dataset.block_size
train_epochs = 2 # todo run a bigger model and longer, this is tiny

# initialize a trainer instance and kick off training
tconf = TrainerConfig(max_epochs=train_epochs, batch_size=1, learning_rate=3e-3,
                      betas = (0.9, 0.95), weight_decay=0,
                      lr_decay=True, warmup_tokens=tokens_per_epoch, final_tokens=train_epochs*tokens_per_epoch,
                      ckpt_path='cifar10_model.pt',
                      num_workers=4,
                     cmd_args=args)
trainer = Trainer(model, train_dataset, test_dataset, tconf)
trainer.train()
```

La classe **trainer.py** gestisce il ciclo di addestramento utilizzando PyTorch, includendo forward pass, backward pass e aggiornamenti dei parametri, con supporto al parallelismo multi-GPU tramite **torch.nn.DataParallel**. Questo file viene richiamato da script come **runStartingPoint.py** e altri file presenti nel progetto, che orchestrano l'addestramento del modello utilizzando una pipeline predefinita. La sua importanza risiede nella gestione completa del ciclo di addestramento, incluso il caricamento dei dati, la definizione delle strategie di ottimizzazione. È fondamentale per i nostri test, poiché permette di verificare il funzionamento del modello sia in configurazioni di base che in scenari distribuiti, misurando l'efficienza, l'accuratezza e la scalabilità del processo.

trainer.py

```
import math
import logging
from tqdm import tqdm
import numpy as np
import torch
import torch.optim as optim
from torch.optim.lr_scheduler import LambdaLR
from torch.utils.data.dataloader import DataLoader

logger = logging.getLogger(__name__)

class Trainer:

    def __init__(self, model, train_dataset, test_dataset, config):
        self.model = model
        self.train_dataset = train_dataset
        self.test_dataset = test_dataset
        self.config = config

        if torch.cuda.is_available():
```

```
        self.device = torch.cuda.current_device()
        self.model = torch.nn.DataParallel(self.model).to(self.device)

    def train(self):
        model, config = self.model, self.config
        raw_model = model.module if hasattr(self.model, "module") else model
        optimizer = raw_model.configure_optimizers(config)
                    train_loader = DataLoader(self.train_dataset, shuffle=True, pin_memory=True, batch_size=config.batch_size,
num_workers=config.num_workers)

        def run_epoch(loader, is_train):
            model.train(is_train)

            losses = []
            pbar = tqdm(enumerate(loader), total=len(loader)) if is_train else enumerate(loader)
            for it, (x, y) in pbar:

                x = x.to(self.device)
                y = y.to(self.device)

                # forward the model
                with torch.set_grad_enabled(is_train):
                    logits, loss = model(x, y)
                    loss = loss.mean() # collapse all losses if they are scattered on multiple gpus
                    losses.append(loss.item())

                if is_train:

                    model.zero_grad()
                    loss.backward()

                    torch.nn.utils.clip_grad_norm_(model.parameters(), config.grad_norm_clip)
                    optimizer.step()

                    # decay the learning rate based on our progress
                    if config.lr_decay:
                        self.tokens += (y >= 0).sum() # number of tokens processed this step (i.e. label is not -100)
                        if self.tokens < config.warmup_tokens:
                            # linear warmup
                            lr_mult = float(self.tokens) / float(max(1, config.warmup_tokens))
                        else:
                            # cosine learning rate decay
                            progress = float(self.tokens - config.warmup_tokens) / float(max(1, config.final_tokens - config.warmup_tokens))
                            lr_mult = max(0.1, 0.5 * (1.0 + math.cos(math.pi * progress)))
                        lr = config.learning_rate * lr_mult
                        for param_group in optimizer.param_groups:
                            param_group['lr'] = lr
                    else:
                        lr = config.learning_rate

                    # report progress
                    pbar.set_description(f"epoch {epoch+1} iter {it}: train loss {loss.item():.5f}. lr {lr:e}")

            if not is_train:
                test_loss = float(np.mean(losses))
                logger.info("test loss: %f", test_loss)
                return test_loss

        best_loss = float('inf')
        self.tokens = 0 # counter used for learning rate decay

        for epoch in range(config.max_epochs):
            run_epoch(train_loader, is_train=True)
```

E’ possible quindi eseguire lo script e verificare che non vi siano errori nell’esecuzione. In questa fase non è importante far completare il training del modello pertanto lo script può essere terminato dopo avene verificato la corretta esecuzione.

1. Migrazione a DeepSpeed

Il primo step che ci proponiamo di eseguire consiste nel modificare il file **trainer.py** che gestisce il ciclo di addestramento. La modifica sostenziale consiste nell’aggiunta al file della classe DeepSpeedTrainer.

Il file `trainer.py` conterrà quindi al suo interno due classi per l'addestramento di un modello GPT, una implementazione standard (`Trainer`) e una progettata per integrare DeepSpeed (`DeepSpeedTrainer`). La classe standard gestisce il ciclo di addestramento utilizzando PyTorch, includendo forward pass, backward pass e aggiornamenti dei parametri, con supporto al parallelismo multi-GPU tramite `torch.nn.DataParallel`. La versione DeepSpeed invece utilizza funzionalità avanzate come l'ottimizzazione della memoria e il parallelismo distribuito per scalare l'addestramento su più GPU. Questa modularità consente di passare facilmente da un'implementazione base a una distribuita ottimizzata.

Questo file viene richiamato da script come `runStartingPoint.py` e altri file successivi nel progetto, che orchestrano l'addestramento del modello utilizzando una pipeline predefinita. La sua importanza risiede nella gestione completa del ciclo di addestramento, incluso il caricamento dei dati, la definizione delle strategie di ottimizzazione, e il supporto per tecniche avanzate come DeepSpeed.

trainer.py

```
import math
import logging
from tqdm import tqdm
import numpy as np
import torch
import deepspeed  # Importare DeepSpeed
from torch.utils.data.dataloader import DataLoader

logger = logging.getLogger(__name__)

class Trainer:
……..

class DeepSpeedTrainer:
    """
    Trainer ottimizzato con DeepSpeed.
    """
    def __init__(self, model, train_dataset, test_dataset, config):
        self.model = model
        self.train_dataset = train_dataset
        self.test_dataset = test_dataset
        self.config = config
```

```
def train(self):
    model, config = self.model, self.config
    raw_model = model.module if hasattr(self.model, "module") else model

    # Inizializza DeepSpeed
    parameters = filter(lambda p: p.requires_grad, model.parameters())
    model_engine, optimizer, train_loader, _ = deepspeed.initialize(
        args=config.cmd_args,
        model=model,
        model_parameters=parameters,
        training_data=self.train_dataset
    )

    def run_epoch(loader, is_train):
        model_engine.train(is_train)

        losses = []
        pbar = tqdm(enumerate(loader), total=len(loader)) if is_train else enumerate(loader)
        for it, (x, y) in pbar:

            # Assegna i dati al dispositivo corretto
            device = torch.device(f"cuda:{config.cmd_args.local_rank}")
            x = x.to(device)
            y = y.to(device)

            # Forward pass con model_engine
            with torch.set_grad_enabled(is_train):
                logits, loss = model_engine(x, y)
                losses.append(loss.item())

            if is_train:
                # Backward pass
                model_engine.backward(loss)
                # Ottimizzazione
                model_engine.step()

        if not is_train:
            test_loss = float(np.mean(losses))
            logger.info("test loss: %f", test_loss)
            return test_loss

    best_loss = float('inf')
    self.tokens = 0

    for epoch in range(config.max_epochs):
        run_epoch(train_loader, is_train=True)            # Assegna i dati al dispositivo corretto
            device = torch.device(f"cuda:{config.cmd_args.local_rank}")
            x = x.to(device)
            y = y.to(device)

            # Forward pass con model_engine
            with torch.set_grad_enabled(is_train):
                logits, loss = model_engine(x, y)
                losses.append(loss.item())
```

```
            if is_train:
                # Backward pass
                model_engine.backward(loss)

                # Ottimizzazione
                model_engine.step()

        if not is_train:
            test_loss = float(np.mean(losses))
            logger.info("test loss: %f", test_loss)
            return test_loss

    best_loss = float('inf')
    self.tokens = 0  # Contatore per il learning rate decay

    for epoch in range(config.max_epochs):
        run_epoch(train_loader, is_train=True)
```

La classe `DeepSpeedTrainer` rappresenta un'evoluzione rispetto alla classe `Trainer` standard, poiché integra le ottimizzazioni avanzate offerte dalla libreria DeepSpeed. Entrambe condividono l'obiettivo comune di addestrare modelli di deep learning, ma il modo in cui realizzano questa funzione è profondamente diverso, riflettendo la differenza tra un approccio tradizionale e uno scalabile ottimizzato per grandi modelli.

La classe Trainer si basa sulle capacità native di PyTorch, fornendo un ciclo di addestramento che include passaggi ben definiti come la propagazione in avanti, il calcolo della perdita, la retropropagazione e l'aggiornamento dei parametri del modello. È una soluzione diretta ed efficace, ideale per ambienti meno complessi e per modelli con una dimensione gestibile. Tuttavia, mancano le sofisticate ottimizzazioni necessarie per gestire modelli su larga scala o sfruttare al meglio un cluster di GPU. La gestione delle risorse, come la memoria delle GPU, è lasciata alla configurazione predefinita di PyTorch, il che può limitare l'addestramento di modelli di grandi dimensioni.

La classe DeepSpeed offre strumenti avanzati per la gestione delle risorse e l'ottimizzazione delle prestazioni, rendendo possibile l'addestramento di modelli con miliardi di parametri. Ad esempio, utilizza tecniche come l'Activation Checkpointing, che riduce il consumo di memoria durante l'addestramento salvando e ricalcolando dinamicamente le attivazioni necessarie, e il Mixed Precision Training, che sfrutta la combinazione di precisione FP16 e FP32 per accelerare i calcoli e ridurre ulteriormente l'uso della memoria. Un'altra caratteristica distintiva è l'uso dell'ottimizzatore ZeRO, che distribuisce i parametri del modello e gli stati dell'ottimizzatore su più GPU, eliminando la ridondanza e consentendo una scalabilità efficiente su più nodi.

Queste funzionalità non sono direttamente implementate all'interno della classe DeepSpeedTrainer, ma vengono abilitate tramite la configurazione di DeepSpeed. Una configurazione di DeepSpeed standard, ovvero senza la configurazione di particolari ottimizzatori, viene presentata di seguito all'interno del file **ds_config_basic.json**. A questa prima versione verranno aggiunti gli ottimizzatori man mano che andremo avanti nella descrizione del lavoro.

ds_config_basic.json

```
{
  "train_micro_batch_size_per_gpu": 8,
  "optimizer": {
  "type": "Adam",
              "params": {
               "lr": 3e-3
              }
   },
    "gradient_clipping": 1.0
}
```

La configurazione di DeepSpeed in questo semplice esempio:

- Imposta il valore di **micro-batch size per GPU** a **8**. Questo parametro indica quante istanze del batch (micro-batch) vengono elaborate da ciascuna GPU per ogni passaggio di addestramento.
- Abilita l'ottimizzatore **Adam** e setta il valore del **learning rate** ad 3e-3. Adam è un ottimizzatore comune per il training di modelli di deep learning. Il learning rate specificato (3e-3) determina quanto velocemente i pesi del modello vengono aggiornati.
- Imposta il valore del **gradient clipping** ad 1. Limita i valori massimi dei gradienti calcolati durante il backpropagation per evitare instabilità numeriche o esplosioni dei gradienti.

Questa prima configurazione ci serve per eseguire un primo test, serve per garantire che il codice funzioni correttamente con la libreria DeepSpeed configurato con parametri di base. Successivamente, verranno inserito man mano gli ottimizzatori e verificato il funzionamento.

Un altro aspetto fondamentale da considerara per la classe DeepSpeed è **la gestione automatizzata del ciclo di addestramento**. Mentre nella classe Trainer ogni passaggio, dal forward alla retropropagazione, deve essere gestito manualmente, in DeepSpeedTrainer questa responsabilità è delegata all'oggetto **DeepSpeedEngine**, che automatizza molti aspetti complessi, come il caricamento distribuito dei dati, l'ottimizzazione della memoria e la sincronizzazione dei gradienti tra GPU. Questa automazione non solo riduce la complessità del codice, ma assicura anche che le migliori pratiche per l'addestramento distribuito vengano seguite automaticamente.

In sintesi, mentre Trainer rappresenta un approccio tradizionale e diretto all'addestramento dei modelli, DeepSpeedTrainer si distingue come una soluzione moderna e scalabile, progettata per affrontare le sfide dell'addestramento di modelli sempre più grandi e complessi. La scelta tra le due dipende dall'ambito del problema: per applicazioni di piccola o media scala, Trainer può essere sufficiente; ma per scenari che richiedono scalabilità e ottimizzazioni avanzate, DeepSpeedTrainer offre strumenti indispensabili per sfruttare appieno le risorse hardware disponibili.

runFirstDeepSpeed.py

```
import numpy as np
import argparse
import torchvision
import torch
from torch.utils.data import Dataset
import logging
from mingpt.utils import set_seed
from mingpt.utils import ImageDataset, TrainerConfig, kmeans
from mingpt.model import GPT, GPTConfig, GPT1Config
from mingpt.trainer import Trainer, DeepSpeedTrainer

# Step 1: Importa la libreria necessaria
import deepspeed

def add_argument():
    parser = argparse.ArgumentParser(description='CIFAR')
    parser.add_argument('--local_rank', type=int, default=-1,
                        help='local rank passed from distributed launcher')

    # Step 1: Includi gli argomenti per DeepSpeed
    parser = deepspeed.add_config_arguments(parser)

    args = parser.parse_args()
    return args

args = add_argument()

# Step 1: Inizializza il backend distribuito
deepspeed.init_distributed()

logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s -   %(message)s",
        datefmt="%m/%d/%Y %H:%M:%S",
        level=logging.INFO,
)
set_seed(42)

# Ottieni i dati CIFAR10
root = './'
train_data = torchvision.datasets.CIFAR10(root, train=True, transform=None, target_transform=None, download=True)
test_data  = torchvision.datasets.CIFAR10(root, train=False, transform=None, target_transform=None, download=True)

# Estrai 5 pixel casuali per immagine e creane un vocabolario con k-means
pluck_rgb = lambda x: torch.from_numpy(np.array(x)).view(32*32, 3)[torch.randperm(32*32)[:5], :]
px = torch.cat([pluck_rgb(x) for x, y in train_data], dim=0).float()

ncluster = 512
with torch.no_grad():
    C = kmeans(px, ncluster, niter=8)

train_dataset = ImageDataset(train_data, C)
```

```
test_dataset = ImageDataset(test_data, C)

# Configura il modello GPT
mconf = GPTConfig(train_dataset.vocab_size, train_dataset.block_size,
                  embd_pdrop=0.0, resid_pdrop=0.0, attn_pdrop=0.0,
                  n_layer=12, n_head=8, n_embd=256)
model = GPT(mconf)

tokens_per_epoch = len(train_data) * train_dataset.block_size
train_epochs = 2

# Configura il Trainer
tconf = TrainerConfig(max_epochs=train_epochs, batch_size=1, learning_rate=3e-3,
                      betas=(0.9, 0.95), weight_decay=0,
                      lr_decay=True, warmup_tokens=tokens_per_epoch, final_tokens=train_epochs*tokens_per_epoch,
                      ckpt_path='cifar10_model.pt',
                      num_workers=4,
                      cmd_args=args)

# Step 1: Usa il DeepSpeedTrainer
trainer = DeepSpeedTrainer(model, train_dataset, test_dataset, tconf)
trainer.train()
```

In questo primo step abbiamo visto come inserire la libreria DeepSpeed configurata con parametri di base all'interno del nostro codice.
Per testare che tutto funzioni correttamente lanciamo il seguente comando ed attendiamo il tempo necessario all'avvio della fase di traing. Se non si riscontrano errori possiamo interrompere l'esecuzione e passare alle fasi successive.

```
!deepspeed minGPT/minGPT/runFirstDeepSpeed.py --deepspeed --deepspeed_config minGPT/minGPT/ds_config_basic.json
```

Questo comando avvia l'addestramento di minGPT su un nodo con 4 GPU utilizzando una configurazioni di base.

Addestramento Multi-Node

Il secondo step ha come obiettivo l'addestramento del modello su due nodi. Visto che lo scenario applicativo coinvolge più nodi di un cluster di calcolo, dovremmo far uso in questo caso di un **sistema di gestione dei job come SLURM**. Il seguente script utilizza un gestore di job per avviare un'operazione di addestramento distribuito su due nodi. Lo script è strutturato per essere sottomesso come job a un cluster gestito da Slurm e utilizza i comandi e le opzioni specifiche di Slurm. Di seguito, viene fornita una spiegazione dettagliata dei comandi utilizzati nello script.

runSlurmStep3.sh

```
%%writefile ./minGPT/minGPT/runSlurmStep3.sh
```

```
#!/bin/bash
#SBATCH --job-name=dli_assessment_step3
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32  # Numero di thread per task (OMP threads)
#SBATCH -o /dli/megatron/logs/%j.out
#SBATCH -e /dli/megatron/logs/%j.err

# Numero di nodi
NUM_NODES=2
# Numero di GPU per nodo
NUM_GPUS=4

deepspeed --num_nodes=${NUM_NODES} --hostfile /dli/minGPT/minGPT/hostfile --num_gpus=${NUM_GPUS} /dli/minGPT/minGPT/runFirstDeepSpeed.py \
  --deepspeed \
  --deepspeed_config /dli/minGPT/minGPT/ds_config_basic.json
```

Nella prima parte dello script sono inserite le righe che iniziano con **#SBATCH** che sono direttive per Slurm. Queste serviranno per specificare i requisiti del job che deve gestire. In particolare il parametro:

--job-name=dli_assessment_step3: Assegna un nome al job, in questo caso dli_assessment_step3.

--nodes=2: Specifica che il job richiede 2 nodi per l'esecuzione.

--ntasks-per-node=1: Ogni nodo eseguirà un singolo task.

--cpus-per-task=32: Ogni task utilizzerà 32 CPU (thread).

-o /dli/megatron/logs/%j.out: Specifica il percorso del file di log per l'output standard del job. %j sarà sostituito con l'ID del job.

-e /dli/megatron/logs/%j.err: Specifica il percorso del file di log per l'errore standard.

Successivamente troveremo la definizione di due variabili (NUM_NODES e NUM_GPUS) per configurare il numero di nodi e le GPU per nodo, nonchè il comando che avvia il training attraverso la libreria DeepSpeed.

In questo esempio lo script è configurato per **eseguire un job distribuito con DeepSpeed su 2 nodi e 4 GPU per nodo** (quindi un totale di 8 GPU). Il backend distribuito NCCL sarà inizializzato da DeepSpeed, in coordinamento con Slurm.

Per sottomettere lo script a Slurm, si utilizza il comando:

```
!sbatch ./minGPT/minGPT/runSlurmStep3.sh

!squeue
```

Questo comando invia il job alla coda di Slurm, che lo pianificherà sui nodi disponibili in base alle specifiche richieste. Una volta avviato, i file di log verranno creati nei percorsi definiti con -o e -e.

Infine il comando `!squeue` mostrerà lo stato del job lanciato:

```
Submitted batch job 3
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
                 3  slurmpar dli_asse    admin PD       0:00      2 (None)
```

Se il job è stato inviato correttamente si troverà inizialmente nello stato Pending (PD). Tuttavia, potrebbe accadere che la colonna NODELIST(REASON) indica (None), il che significa che non ci sono problemi evidenti con il job. Questo di solito accade quando il sistema sta cercando di allocare le risorse necessarie.
Se lanciamo dopo qualche minuto di nuovo il comando `!squeue` potremmo verificare che il job è passato allo di stato Running (R), il che significa che il training è in corso su entrambi i nodi (slurmnode1 e slurmnode2).

```
Submitted batch job 3
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
                 3  slurmpar dli_asse    admin  R       0:10      2 slurmnode[1-2]
```

Per monitorare i log di output e verificare che il training stia procedendo correttamente si può usare il seguente comando che leggere il file di log di output:

```
job_id = 3  # Sostituisci 3 con il tuo Job ID

!cat /dli/megatron/logs/{job_id}.out
```

Questo comando mostrerà in tempo reale i progressi del job, inclusi eventuali messaggi relativi alla perdita (`loss`), agli aggiornamenti dell'ottimizzatore e ad altre informazioni utili.

```
slurmnode2: 11/20/2024 11:02:15 - INFO - mingpt.model -   number of parameters: 1.000166e+07
slurmnode1: 11/20/2024 11:02:16 - INFO - mingpt.model -   number of parameters: 1.000166e+07
slurmnode1: 11/20/2024 11:02:19 - INFO - torch.distributed.distributed_c10d -   Added key: store_based_barrier_key:2 to store for rank: 0
slurmnode2: 11/20/2024 11:02:20 - INFO - torch.distributed.distributed_c10d -   Added key: store_based_barrier_key:2 to store for rank: 4
slurmnode1: 11/20/2024 11:02:29 - INFO - torch.distributed.distributed_c10d -   Waiting in store based barrier to initialize process group for rank: 0, key: store_based_barrier_key:2 (world_size=8, worker_count=2, timeout=0:30:00)
slurmnode2: 11/20/2024 11:02:30 - INFO - mingpt.model -   number of parameters: 1.000166e+07
slurmnode2: 11/20/2024 11:02:30 - INFO - torch.distributed.distributed_c10d -   Waiting in store based barrier to initialize process group for rank: 4, key: store_based_barrier_key:2 (world_size=8, worker_count=2, timeout=0:30:00)
slurmnode2: 11/20/2024 11:02:32 - INFO - torch.distributed.distributed_c10d -   Added key: store_based_barrier_key:2 to store for rank: 5
slurmnode2: 11/20/2024 11:02:32 - INFO - mingpt.model -   number of parameters: 1.000166e+07
slurmnode2: 11/20/2024 11:02:33 - INFO - mingpt.model -   number of parameters: 1.000166e+07
slurmnode1: 11/20/2024 11:02:33 - INFO - mingpt.model -   number of parameters: 1.000166e+07
slurmnode1: 11/20/2024 11:02:33 - INFO - mingpt.model -   number of parameters: 1.000166e+07
slurmnode1: 11/20/2024 11:02:33 - INFO - mingpt.model -   number of parameters: 1.000166e+07
slurmnode2: 11/20/2024 11:02:34 - INFO - torch.distributed.distributed_c10d -   Added key: store_based_barrier_key:2 to store for rank: 6
slurmnode2: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Added key: store_based_barrier_key:2 to store for rank: 7
```

```
slurmnode1: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Added key: store_based_barrier_key:2 to store
for rank: 1
slurmnode1: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Added key: store_based_barrier_key:2 to store
for rank: 2
slurmnode1: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Added key: store_based_barrier_key:2 to store
for rank: 3
slurmnode1: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Rank 3: Completed store-based barrier for
key:store_based_barrier_key:2 with 8 nodes.
slurmnode1: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Rank 1: Completed store-based barrier for
key:store_based_barrier_key:2 with 8 nodes.
slurmnode1: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Rank 2: Completed store-based barrier for
key:store_based_barrier_key:2 with 8 nodes.
slurmnode1: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Rank 0: Completed store-based barrier for
key:store_based_barrier_key:2 with 8 nodes.
slurmnode2: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Rank 7: Completed store-based barrier for
key:store_based_barrier_key:2 with 8 nodes.
slurmnode2: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Rank 5: Completed store-based barrier for
key:store_based_barrier_key:2 with 8 nodes.
slurmnode2: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Rank 4: Completed store-based barrier for
key:store_based_barrier_key:2 with 8 nodes.
slurmnode2: 11/20/2024 11:02:35 - INFO - torch.distributed.distributed_c10d -   Rank 6: Completed store-based barrier for
key:store_based_barrier_key:2 with 8 nodes.
 13%|▉         | 782/6250 [04:14<29:41,  3.07it/s]
 13%|▉         | 782/6250 [04:14<29:41,  3.07it/s]
 13%|▉         | 782/6250 [04:14<29:40,  3.07it/s]
 13%|▉         | 782/6250 [04:14<29:41,  3.07it/s]
slurmnode2:
 13%|▉         | 782/6250 [04:14<29:41,  3.07it/s]
 13%|▉         | 782/6250 [04:14<29:42,  3.07it/s]
 13%|▉         | 782/6250 [04:15<29:44,  3.06it/s]
```

Dall'analisi del log possiamo trarre le seguenti informazioni:

- Informazioni sui parametri del modello:

  slurmnode2: INFO - mingpt.model - number of parameters: 1.000166e+07

Queste righe mostrano che **il modello è stato caricato correttamente** su ciascun nodo e che entrambi i nodi (slurmnode1 e slurmnode2) elaborano il modello con circa 10 milioni di parametri.

- Barriera di sincronizzazione:

  INFO - torch.distributed.distributed_c10d - Waiting in store based barrier to initialize process group for rank: X
  INFO - torch.distributed.distributed_c10d - Rank X: Completed store-based barrier for key:store_based_barrier_key:2 with 8 nodes.

Questi messaggi indicano che i processi distribuiti (ognuno con un rank unico) stanno sincronizzandosi. Rank 0-7: Sono i processi distribuiti su 8 GPU (4 per nodo). La sincronizzazione avviene senza timeout, il che indica che **i nodi stanno comunicando correttamente**.

- Progresso dell'addestramento:

  13%|▉ | 782/6250 [04:14<29:41, 3.07it/s]

Questo messaggio indica che l'addestramento è in corso. Che siamo al 13% delle iterazioni, con una velocità di 3.07 iterazioni al secondo. La barra di progresso è presente per ogni GPU, confermando che **tutte le GPU stanno lavorando correttamente**.

Verificato che la parallelizzazione del processo sia avvenuta correttamente possiamo anche terminare questa fase e procedere ad eliminare i job in esecuzione. Per eliminare i job in esecuzione possiamo utilizzare il seguente comando:

```
!scancel 3      # 3 è l'ID del mio job
```

2. Ottimizzazione della Memoria

Il progetto mira a dimostrare l'utilizzo di DeepSpeed per integrare tecniche avanzate di ottimizzazione della memoria e del calcolo, fondamentali per affrontare le sfide poste dall'addestramento di modelli su larga scala. Nello step precedente abbiamo avuto modo di vedere come convertire una pipeline di addestramento standalone in PyTorch in una Multi-nodo per sfruttare le ottimizzazioni offerte da DeepSpeed, consentendo l'addestramento distribuito su un cluster composto da due server. In questo secondo step l'obbiettivo sarà quello di integrare funzionalità avanzate come Mixed Precision Training, Activation Checkpointing e ZeRO per gestire in modo efficiente l'uso delle risorse.

Tra le funzionalità principali implementate vi sono:

- **Activation Checkpointing**: una tecnica per ottimizzare l'uso della memoria attraverso il salvataggio e il ricalcolo selettivo delle attivazioni durante il backpropagation, riducendo il consumo di memoria senza compromettere significativamente i tempi di calcolo.
- **Mixed Precision Training**: un approccio che combina precisioni FP16 e FP32 per accelerare il calcolo e ottimizzare l'utilizzo della memoria delle GPU.
- **ZeRO Redundancy Optimizer (ZeRO)**: un ottimizzatore che riduce drasticamente il consumo di memoria durante l'addestramento distribuito, permettendo di addestrare modelli di dimensioni molto superiori rispetto a quelli gestibili con approcci convenzionali.

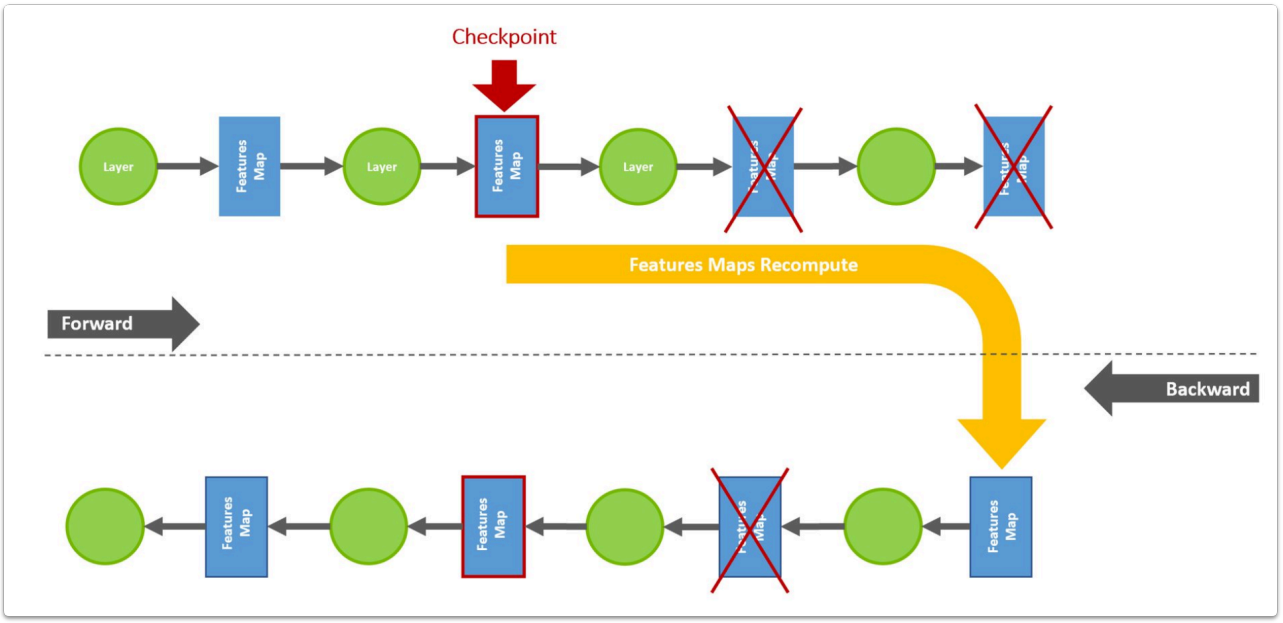
In **questo terzo step**, esploreremo in dettaglio come implementare il **checkpointing delle attivazioni** e la precisione mista nel contesto del nostro modello **Vision Transformer**, che è definito nella classe GPT. Tratteremo ZeRO nel prossimo capitolo dove affronteremo anche la problematica della scalabilità dei modelli su sistemi con risorse limitate.

Implementazione del Checkpointing delle Attivazioni

Il checkpointing delle attivazioni è una tecnica che consente di risparmiare memoria durante l'addestramento. L'idea centrale è quella di non mantenere tutte le attivazioni in memoria durante il passaggio forward, ma di calcolarle nuovamente durante il backpropagation solo quando necessario. Questo si ottiene, attraverso l'utilizzo di DeepSpeed, racchiudendo i blocchi del modello con una funzione fornita dalla libreria, chiamata `deepspeed.checkpointing.checkpoint()`.

L'activation checkpointing rappresenta una tecnica efficace per ottimizzare l'uso della memoria durante l'addestramento dei modelli. Normalmente, nel passaggio forward di un modello, le attivazioni generate da ogni livello vengono mantenute in memoria, in quanto sono necessarie successivamente per calcolare i gradienti durante il backpropagation. Tuttavia, questo processo tradizionale comporta un consumo significativo di memoria GPU, specialmente nei modelli di grandi dimensioni.

Con l'activation checkpointing, si introduce un approccio più parsimonioso nella gestione della memoria. Invece di mantenere tutte le attivazioni in memoria per l'intera durata del passaggio forward e backward, solo una parte di esse viene conservata. Le attivazioni mancanti vengono calcolate nuovamente in modo dinamico durante il backpropagation, utilizzando i dati iniziali e i parametri del modello. Questo meccanismo, pur introducendo un moderato costo computazionale, consente un risparmio di memoria significativo, rendendo possibile l'addestramento di modelli di grandi dimensioni anche su hardware con risorse limitate.



Per abilitare l'activation checkpointing di un modello (o di una parte di esso) con DeepSpeed, durante la definizione del passaggio forward, dobbiamo inserire ogni blocco del modello con la funzione `deepspeed.checkpointing.checkpoint()`.

Se consideriamo a titolo di esempio un modello di rete CNN, semplicemente quello che dovremmo fare è ridefinire la funzione di forward richiamando i blocchi come parametro della funzione `checkpoint()` come mostrato di seguito:

```
class CNN(nn.Module):
   def __init__(self):
      super().__init__()
      self.cnn_block_1 = ...
      self.cnn_block_2 = ...
      self.linearize = ...
      self.out = ...

   def forward(self, X):
      X = deepspeed.checkpointing.checkpoint(self.cnn_block_1, X)
      X = deepspeed.checkpointing.checkpoint(self.cnn_block_2, X)
      ...
      return X
```

Nel nostro caso specifico, la classe **GPT** definisce il **Vision Transformer**. I blocchi Transformer sono rappresentati come una sequenza di livelli, definiti come `self.blocks`. Per abilitare il checkpointing delle attivazioni, modificheremo il passaggio forward del modello in modo da inglobare ciascun blocco con la funzione `deepspeed.checkpointing.checkpoint()`.

Per facilitare la lettura, nel file `./minGPT/minGPT/mingpt/model.py`, identifichiamo la sezione che andremo a modificare con **# Step 2**.

`model.py`

```
"""GPT model"""
import math
import logging
import torch
import torch.nn as nn
from torch.nn import functional as F
import deepspeed

 ..........................

class GPT(nn.Module):
   """  the full GPT language model, with a context size of block_size """

     ....................

   def forward(self, idx, targets=None):
      b, t = idx.size()
      assert t <= self.block_size, "Cannot forward, model block size is exhausted."

      # forward the GPT model
      token_embeddings = self.tok_emb(idx) # each index maps to a (learnable) vector
      position_embeddings = self.pos_emb[:, :t, :] # each position maps to a (learnable) vector
      x = self.drop(token_embeddings + position_embeddings)

      # ****** Step 2: Enable here activation checkpointing of all tranformer bloks ******
      # transformer bloks
      # x = self.blocks(x)
      for block in self.blocks:
         x = deepspeed.checkpointing.checkpoint(block, x)

      # decoder head
```

```
        x = self.ln_f(x)
        logits = self.head(x)

        # if we are given some desired targets also calculate the loss
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1))

        return logits, loss
```

Questa semplice modifica al modello, applica il checkpointing a ogni singolo blocco del Transformer, consentendo una gestione efficiente della memoria durante l'addestramento.

Implementazione del Mixed Precision Training

Parallelamente al checkpointing delle attivazioni, configureremo anche il Mixed Precision Training. Questo approccio sfrutta rappresentazioni numeriche a precisione ridotta (FP16) per i calcoli intensivi, riservando la precisione completa (FP32) alle operazioni critiche, come l'accumulo dei gradienti. Ciò consente di ridurre l'utilizzo della memoria GPU, accelerare i calcoli grazie al supporto hardware ottimizzato per FP16 su GPU moderne.
Nel contesto di DeepSpeed, l'abilitazione del Mixed Precision Training avviene attraverso la configurazione del file JSON di DeepSpeed, aggiungendo la sezione:

```
"fp16": {
        "enabled": true
}
```

Questa impostazione consente di eseguire il training in modalità Mixed Precision senza ulteriori modifiche al codice del modello.
Procediamo per step alla modifica del nostro codice di esempio assicurandoci di salvare una copia prima di ogni modifica.

Per inserire le ottimizzazioni alla libreria DeepSpeed dovremmo modificare il file di configurazione JSON della libreria che chiameremo ***ds_config_step4.json***. In questo file, oltre alle configurazioni di base viste in precedenza, andremo ad inserire le seguenti direttive:

- Abilitazione dell'activation checkpointing
- Definire la dimensione del micro batch per GPU a 128
- Definire il numero di checkpoint a 12
- Abilitare il training con una precisione FP16

Il file con le modifiche richieste è il seguente:

ds_config_step4.json

```
{
  "train_micro_batch_size_per_gpu": 128,
  "optimizer": {
    "type": "Adam",
    "params": {
      "lr": 3e-4
```

```
    }
  },
  "gradient_clipping": 1.0,
  "activation_checkpointing": {
    "partition_activations": true,
    "cpu_checkpointing": false,
    "contiguous_memory_optimization": true,
    "number_checkpoints": 12,
    "synchronize_checkpoint_boundary": false,
    "profile": false
  },
  "fp16": {
    "enabled": true
  }
}
```

Possiamo ora testare la corretta esecuzione dell'addestramento sul nostro sistema con SLURM seguendo gli stessi passi visti in precedenza:

- creiamo una copia dello script `runFirstDeepSpeed.py`:

```
!cp /dli/minGPT/minGPT/runFirstDeepSpeed.py /dli/minGPT/minGPT/runStep4.py
```

- creiamo lo script batch per configurare un Job SLURM `runSlurmStep4.sh`:

```
%%writefile ./minGPT/minGPT/runSlurmStep4.sh
#!/bin/bash
#SBATCH --job-name=dli_assessment_step4
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32
#SBATCH -o /dli/megatron/logs/%j.out
#SBATCH -e /dli/megatron/logs/%j.err

# Numero di nodi
NUM_NODES=2
# Numero di GPU per nodo
NUM_GPUS=2

deepspeed --num_nodes=${NUM_NODES} --hostfile /dli/minGPT/minGPT/hostfile --num_gpus=${NUM_GPUS}
/dli/minGPT/minGPT/runStep4.py \
   --deepspeed \
   --deepspeed_config /dli/minGPT/minGPT/ds_config_step4.json
```

- Lanciamo il job con SLURM:

```
!sbatch /dli/minGPT/minGPT/runSlurmStep4.sh
!squeue
```

NOTA: se esaminiamo i log possiamo osservare il seguente messaggio:

```
slurmnode1: [2024-11-20 13:47:25,227] [INFO] [stage3.py:2281:_overflow_clean_up] [deepspeed]
OVERFLOW! Rank 0 Skipping step. Attempted loss scale: 134217728.0, reducing to 67108864.0
```

Il log mostra che l'addestramento con DeepSpeed è stato configurato e avviato correttamente, ma si stanno verificando degli **overflow numerici** durante l'addestramento. Questi errori sono comuni

quando si usa FP16 (precisione a 16 bit) per accelerare il calcolo e ridurre il consumo di memoria, ma il modello o i gradienti diventano troppo piccoli o instabili. DeepSpeed sta tentando di ridurre la scala della perdita (loss scale) per gestire l'overflow.

Ulteriori ottimizzazioni

Nel processo di addestramento di minGPT, il calcolo del k-means, che avviene durante la fase iniziale di preparazione dei dati per l'addestramento del modello, è un passaggio fondamentale per costruire un vocabolario di rappresentazioni visive a partire dai dati di input, nel caso specifico immagini del dataset CIFAR-10. L'obiettivo è creare cluster di rappresentazioni simili per ridurre la dimensionalità del problema e ottenere una rappresentazione compatta dei dati che il modello utilizzerà per apprendere.

Nel contesto dell'addestramento distribuito, come quello realizzato con DeepSpeed, il k-means viene calcolato utilizzando un sottoinsieme di pixel estratti casualmente dalle immagini del dataset di addestramento. Questa operazione genera i centri dei cluster, che vengono poi utilizzati per rappresentare ogni immagine come una sequenza di token che corrispondono ai cluster identificati. Questo approccio permette di **trattare il problema come un compito di modellazione sequenziale**, rendendo possibile l'uso di **modelli basati su Transformer** come il Vision Transformer.

In una configurazione distribuita, ogni nodo o worker calcola inizialmente il k-means in modo indipendente. Tuttavia, come già discusso in RT- ICAR-NA-24-06, questo comportamento risulta inefficiente, perché tutti i worker generano lo stesso risultato, **duplicando il lavoro computazionale** senza apportare benefici. Per ottimizzare questa fase, si adotta una strategia in cui il calcolo del k-means viene eseguito una sola volta su un worker selezionato (ad esempio, il worker con rank 0). Una volta completato, i risultati, ovvero i centri dei cluster, vengono distribuiti agli altri worker tramite una sincronizzazione, utilizzando strumenti di comunicazione come quelli forniti da **torch.distributed**. Questo processo garantisce che tutti i worker abbiano accesso agli stessi risultati, senza la necessità di ricalcolare il k-means localmente.

Per realizzare questa ottimizzazione, cerchiamo all'interno del file **`runFirstDeepSpeed.py`** la funzione : **`C = kmeans(px, ncluster, niter=8)`**

`runFirstDeepSpeed.py`

```
# Ottieni 5 pixel casuali per immagine e aggrega i valori RGB
pluck_rgb = lambda x: torch.from_numpy(np.array(x)).view(32*32, 3)[torch.randperm(32*32)[:5], :]
px = torch.cat([pluck_rgb(x) for x, y in train_data], dim=0).float()

# Calcola i cluster usando k-means
ncluster = 512
with torch.no_grad():
    C = kmeans(px, ncluster, niter=8)
```

Andremo pertanto a sostituire questa funzione con una nuova funzione che chiameremo **run_kmeans** la quale ci dovrà garantire:

- **Calcolo del k-means sul worker con rank 0**: Solo il processo con rank 0 esegue effettivamente il calcolo del k-means, riducendo il carico computazionale sugli altri worker.

- **Sincronizzazione tra worker**: Una barriera (dist.barrier()) viene introdotta per assicurare che tutti i worker siano pronti a ricevere i risultati. Successivamente, dist.broadcast() trasmette i cluster calcolati dal worker 0 agli altri worker nel cluster.
- **Uso delle GPU**: I centri dei cluster vengono gestiti sulla GPU specificata dal rank (local_rank) per massimizzare le prestazioni nei calcoli distribuiti.
- **Efficienza**: Con questa strategia, il calcolo del k-means viene eseguito una sola volta, e i risultati vengono riutilizzati da tutti i worker. Ciò riduce significativamente il tempo di elaborazione complessivo e migliora l'efficienza del processo di addestramento distribuito.

La funzione **run_kmeans** viene realizzata e descritta nel listato seguente:

runFirstDeepSpeed.py

```
……..
import torch.distributed as dist
……..
def run_kmeans(x, ncluster, niter=8, rank, size):
   print('KMeans executed on rank ', rank, ' World size ', size)
   N, D = x.size()
   c = x[torch.randperm(N)[:ncluster]]  # Inizializza i cluster in modo casuale
   c = c.cuda()  # Sposta i cluster su GPU

   if rank == 0:
      # Esegui il calcolo del k-means solo sul worker con rank 0
      with torch.no_grad():
         c = kmeans(x, ncluster, niter)

   # Sincronizza i cluster tra tutti i worker
   dist.barrier()
   dist.broadcast(c, src=0)

   print('Rank ', rank, ' has data ', c.size())
   return c)
…….

# Ottieni 5 pixel casuali per immagine e aggrega i valori RGB
pluck_rgb = lambda x: torch.from_numpy(np.array(x)).view(32*32, 3)[torch.randperm(32*32)[:5], :]
px = torch.cat([pluck_rgb(x) for x, y in train_data], dim=0).float()

# Calcola i cluster usando la versione distribuita del k-means
ncluster = 512
with torch.no_grad():
   C = run_kmeans(px, ncluster, niter=8, rank=torch.distributed.get_rank(),
size=torch.distributed.get_world_size())
```

3. Scalabiltà con ZeRO Redundancy Optimizer

L'**ultimo step** di questo lavoro consiste nell'**inserire l'ottimizzatore ZeRO** all'interno del nostro processo di training al fine di consentire la scalabilità del modello e l'addestramento dello stesso su hardware con risorse limitate.

ZeRO, acronimo di Zero Redundancy Optimizer, è stato introdotto nel nostro processo di addestramento per affrontare una delle problematiche più comuni e significative nell'addestramento di modelli di grandi dimensioni: il consumo di memoria. Con l'evoluzione dei modelli di machine learning, i parametri da gestire sono cresciuti a dismisura, rendendo spesso difficile o impossibile sfruttare completamente le risorse hardware disponibili, soprattutto quando si utilizzano GPU con memoria limitata.

ZeRO è un ottimizzatore sviluppato da Microsoft come parte della libreria DeepSpeed e consente di superare questi limiti distribuendo in modo intelligente i carichi di memoria su più GPU. Normalmente, durante l'addestramento di un modello, ogni GPU deve conservare una copia completa dei parametri del modello, dei gradienti (utilizzati per aggiornare i parametri) e degli stati dell'ottimizzatore (come i valori dei momenti usati, ad esempio, nell'algoritmo Adam). Questa ridondanza porta a un rapido esaurimento della memoria disponibile, limitando le dimensioni del modello addestrabile o costringendo a ridurre drasticamente la batch size.

ZeRO risolve questa sfida eliminando la duplicazione di questi dati tra le GPU. Suddivide i parametri del modello, i gradienti e gli stati dell'ottimizzatore in modo che ogni GPU ne gestisca solo una porzione. Questo approccio permette di ridurre significativamente l'uso della memoria da parte di ciascuna GPU, consentendo così di addestrare modelli di dimensioni molto più grandi rispetto a quanto sarebbe possibile con metodi tradizionali.

L'ottimizzatore è implementato in diverse "fasi" di ottimizzazione. Nella prima fase, ZeRO distribuisce i gradienti su tutte le GPU, riducendo la memoria necessaria per conservarli. Nella seconda fase, distribuisce anche gli stati dell'ottimizzatore, permettendo ulteriori risparmi di memoria. La terza fase, che è quella più avanzata, distribuisce infine anche i parametri del modello, riducendo al minimo la memoria richiesta per GPU e abilitando l'addestramento di modelli di dimensioni enormi.

Nel nostro progetto, l'integrazione di ZeRO ci permette di sfruttare al massimo l'hardware disponibile, scalando il processo di addestramento su più GPU e nodi in un cluster. Questo si traduce in una maggiore efficienza e nella possibilità di gestire modelli con miliardi di parametri, rendendo il processo di addestramento distribuito non solo più veloce, ma anche più pratico ed economico.

Seguendo l'approccio visto in precedenza, creiamo delle copie dei nostri file ed apportimo le modifiche agli script:

- creiamo una copia dello script `runFirstDeepSpeed.py`:

```
!cp /dli/minGPT/minGPT/runFirstDeepSpeed.py /dli/minGPT/minGPT/runStep5.py
```

- creiamo il file di configurazione JSON di DeepSpeed `ds_config_step5.json`: inserendo i parametri dell'ottimizzatore ZeRO:

ds_config_step5.json

```
{
  "train_micro_batch_size_per_gpu": 128,
  "gradient_accumulation_steps": 4,
  "optimizer": {
    "type": "Adam",
    "params": {
      "lr": 3e-4
    }
  },
  "gradient_clipping": 1.0,
  "activation_checkpointing": {
    "partition_activations": true,
    "cpu_checkpointing": false,
    "contiguous_memory_optimization": true,
    "number_checkpoints": 24,
    "synchronize_checkpoint_boundary": false,
    "profile": false
  },
  "fp16": {
    "enabled": true
  },
  "zero_optimization": {
    "stage": 3,
    "stage3_max_live_parameters": 1e9,
    "stage3_max_reuse_distance": 1e7,
    "stage3_prefetch_bucket_size": 5e7,
    "stage3_param_persitence_threshold": 1e6,
    "reduce_bucket_size": 5e7,
    "contiguous_gradients": true,
    "offload_optimizer": {
      "device": "cpu"
    },
    "offload_param": {
      "device": "cpu"
    }
  }
}
```

Andiamo a questo punto a **scalare il modello aumentando significativamente le dimensioni**.

Per fare ciò andiamo a modifica il numero di strati del VisionTransformer portandolo da 12 a 24. Per farlo, aggiorniamo la sezione "GPTConfig" in runStep5.py, dove sono definite le dimensioni della rete neurale.

Sostituiamo quindi:

runStep5.py

```
mconf = GPTConfig(train_dataset.vocab_size, train_dataset.block_size, embd_pdrop=0.0, resid_pdrop=0.0, attn_pdrop=0.0, n_layer=12, n_head=8, n_embd=256)
```

con :

```
mconf = GPTConfig(train_dataset.vocab_size, train_dataset.block_size, embd_pdrop=0.0, resid_pdrop=0.0, attn_pdrop=0.0, n_layer=24, n_head=8, n_embd=256)
```

- Creiamo uno script SLURM per eseguire il training con la configurazione aggiornata:

runSlurmStep5.sh

```
%%writefile ./minGPT/minGPT/runSlurmStep5.sh
#!/bin/bash
#SBATCH --job-name=dli_assessment_step5
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32 ### Numero di thread per task (thread OMP)
#SBATCH -o /dli/megatron/logs/%j.out
#SBATCH -e /dli/megatron/logs/%j.err

# Numero di nodi
NUM_NODES=2
# Numero di GPU per nodo
NUM_GPUS=2

deepspeed --num_nodes=${NUM_NODES} --hostfile /dli/minGPT/minGPT/hostfile --num_gpus=${NUM_GPUS} /dli/minGPT/minGPT/runStep5.py \
  --deepspeed \
  --deepspeed_config /dli/minGPT/minGPT/ds_config_step5.json
```

- Lanciamo il job con SLURM:

```
!sbatch /dli/minGPT/minGPT/runSlurmStep5.sh
!squeue
```

Bibliografia

F. Gargiulo, A. Francesco Gentile, E. Greco. (2024, December). *Data Parallelism: Deep Learning con GPU Multiple, in modo efficiente e sostenibile*, RT-ICAR-NA-2024-04 https://intranet.icar.cnr.it/wp-content/uploads/2024/11/RT-ICAR-NA-2024-04.pdf

F. Gargiulo, A. Francesco Gentile, E. Greco. (2024, December). *Data Parallelism: Tecniche Avanzate di Stabilità e Convergenza di modelli su Larga Scala*, RT-ICAR-NA-2024-06
https://intranet.icar.cnr.it/wp-content/uploads/2024/11/RT-ICAR-NA-2024-06.pdf

F. Gargiulo, A. Francesco Gentile, E. Greco. (2025, Gennaio). *Model Parallelism: Deep Learning con GPU Multiple in modo efficiente e sostenibile*, RT-ICAR-NA-2025-01 https://intranet.icar.cnr.it/wp-content/uploads/2024/11/RT-ICAR-NA-2025-01.pdf

NVIDIA. *NCCL Operations – Usage Guide*. NVIDIA Documentation, https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/operations.html. Accessed 20 June 2024.

PyTorch. *Getting Started with Distributed Data Parallel*. PyTorch Tutorials, https://pytorch.org/tutorials/intermediate/dist_tuto.html. Accessed 20 June 2024.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., & He, Y. (2021). ZeRO-Offload: Democratizing Billion-Scale Model Training.

PyTorch. *Distributed Data Parallel*. PyTorch Documentation, https://pytorch.org/docs/stable/nn.html#torch.nn.parallel.DistributedDataParallel. Accessed 20 June 2024.

Microsoft and NVIDIA. *Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model*. 2021, https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b/.

Li, Dong, et al. *PyTorch Distributed: Experiences on Accelerating*. Proceedings of the 2020 USENIX Conference on Operational Machine Learning (OpML '20), USENIX Association, 2020, https://www.usenix.org/conference/opml20/presentation/li.

PyTorch. *Distributed Communication Package* – torch.distributed. PyTorch Documentation, https://pytorch.org/docs/stable/distributed.html. Accessed 20 June 2024.

NVIDIA. Parallelism Strategies in NeMo Megatron. NVIDIA Documentation, https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/main/nlp/nemo_megatron/parallelisms.html. Accessed 20 June 2024.

NVIDIA. *Scaling Language Model Training to a Trillion Parameters Using Megatron*. NVIDIA Developer Blog, 21 Apr. 2021, https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/. Accessed 20 June 2024.

FairScale. *Pipeline Parallelism*. FairScale Documentation, https://fairscale.readthedocs.io/en/latest/deep_dive/pipeline_parallelism.html. Accessed 20 June 2024.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., & Zaharia, M.

(2021). *Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM*. arXiv. https://arxiv.org/pdf/2104.04473.

Microsoft and NVIDIA. *Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model*. NVIDIA Developer Blog, 2021, https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b/. Accessed 20 June 2024.

NVIDIA. *NVIDIA NeMo Framework Documentation*. NVIDIA, https://docs.nvidia.com/nemo-framework/. Accessed 20 June 2024.

Li, S., Xue, F., Baranwal, C., Li, Y., & You, Y. (2022). *Sequence Parallelism: Long Sequence Training from System Perspective*. arXiv preprint arXiv:2205.02620. https://arxiv.org/abs/2205.02620.

Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., & Catanzaro, B. (2022). *Reducing Activation Recomputation in Large Transformer Models*. arXiv preprint arXiv:2205.05198. https://arxiv.org/abs/2205.05198.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., ... El Sayed, W. (2024). *Mixtral of Experts*. arXiv preprint arXiv:2401.04088. https://arxiv.org/pdf/2401.04088.pdf

Fedus, W., Zoph, B., & Shazeer, N. (2021). *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. arXiv preprint arXiv:2101.03961. https://arxiv.org/abs/2101.03961.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In Proceedings of the 5th International Conference on Learning Representations* (ICLR 2017). https://arxiv.org/abs/1701.06538

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991). *Adaptive mixtures of local experts*. Neural Computation, 3(1), 79–87. https://doi.org/10.1162/neco.1991.3.1.79

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). *FlashAttention: Fast and memory-efficient exact attention with IO-awareness*. In Advances in Neural Information Processing Systems (NeurIPS 2022). https://arxiv.org/abs/2205.14135

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., & Wu, H. (2018). *Mixed precision training*. International Conference on Learning Representations (ICLR). Disponibile al link: https://arxiv.org/abs/1710.03740

Microsoft. DeepSpeed. https://www.deepspeed.ai. Accessed 20 June 2024

Karpathy, Andrej. minGPT. GitHub, https://github.com/karpathy/minGPT. Accessed 20 June 2024

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention is all you need*. In *Advances in Neural Information Processing Systems (NeurIPS 2017)*. https://arxiv.org/abs/1706.03762

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., & Wu, H. (2018). "*Mixed Precision Training*". Proceedings of the International Conference on Learning Representations (ICLR). https://arxiv.org/abs/1710.03740

Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). "*ZeRO: Memory Optimization Towards Training A Trillion Parameter Models*". Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC).https://arxiv.org/abs/1910.02054

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. International Conference on Learning Representations (ICLR). https://arxiv.org/abs/2010.11929