تمرین کامپیوتری سوم (بخش دوم) درس یادگیری ماشین توزیع شده

پدرام رستمی – ۸۱۰۱۰۰۳۵۳

توضیحات کد

برای این تمرین، ۶ فایل پایتون به همراه یک خروجی html از فایل Jupyter Notebook مربوط به Quantization تهیه شده است. در این بخش به توضیح وظیفه‌ی توابع و کلاس‌ها خواهیم پرداخت.

- *تابع* ***get_data_loader*** : این تابع data loader های آموزش و تست را از مجموعه داده‌ی MNIST آماده می‌کند. تابع get_data_loader با گرفتن rank و ورودی‌های دیگر، data loader ها را بر اساس ورودی‌های کاربر تولید کرده و خروجی می‌دهد. در این تابع، داده‌ها shuffle نمی‌شوند تا بتوان نتایج یکسانی را با اجراهای متفاوت تولید کرد.
- *کلاس* ***ConvNet*** : این فایل مدل را تولید می‌کند. کلاس ConvNet با ورودی گرفت تعداد کلاس‌های نهایی (برای مجموعه داده‌ی MNIST برابر ۱۰ است) و ورودی‌های دیگر مدل را می‌سازد. مدل ساخته شده شامل لایه‌های کانولوشنی، پولینگ و fully connected است. برای ساخت مدل‌های quantized هم از همین کلاس استفاده می‌شود.
- *تابع* ***train*** : این تابع وظیفه‌ی آموزش مدل را دارد. تابع train مدل توزیع شده (DDP)، data loader های آموزش و تست، بهینه ساز، تابع خطا، رنک و ورودی‌های دیگر کاربر را گرفته و بر اساس آن‌ها مدل را آموزش می‌دهد. مقادیر خطای پیشبینی و نتایج واقعی و دقت مدل در طول آموزش در انتهای هر ۱۰۰ iteration برای داده‌های آموزش و در انتهای هر ایپاک برای داده‌های تست به دست آمده و در انتهای آموزش، در پوشه‌ی مربوط با رنک آن، در فایل‌های train.acc، train.loss، test.acc و test.loss ذخیره می‌شوند.
- *توابع* ***init*** : به کمک این توابع، می‌توان instance های اجزا تشکیل دهنده را ساخت. تابع init_model برای ساخت مدل است. این تابع با گرفتن ورودی‌های کاربر، مدل کوانتیزه یا نرمال را ایجاد کرده و آن را به مدل توزیع شده (DDP) تبدیل کرده و خروجی می‌دهد. در این تابع، قبل از ساخت مدل مقدار seed برای PyTorch تنظیم می‌شود تا مدل‌های یکسانی برای آزمایش‌های مختلف تولید شوند. از آنجایی که هم وزن‌های مدل با seed های یکسانی تولید می‌شوند و هم داده‌ها shuffle نمی‌شوند، در آزمایش‌های مختلف توابع نهایی یکسانی تولید می‌شوند. تابع init_criterion تابع خطای CrossEntropy را به عنوان تابع خطا خروجی می‌دهد. تابع init_optimizer بهینه ساز Adam را بر اساس ورودی‌های کاربر و مدل توزیع شده تولید می‌کند. تابع init_data_loader وظیفه‌ی تولید

data loader های آموزش و تست را بر اساس ورودی‌های کاربر و رنک فرایند دارد. در نهایت تابع init_log_dir با گرفتن رنک و ورودی‌های کاربر که شامل آدرس پوشه‌ی لاگ می‌شود، برای هر رنک پوشه‌ی مخصوص به خود را تولید می‌کند.

- *تابع main* : این تابع وظیفه‌ی تولید اجزای مختلف (مانند مدل، تابع خطا و ...) به وسیله‌ی توابع init را داشته و مدل را آموزش می‌دهد. همچنین در این فایل ورودی‌هایی که کاربر می‌تواند مقادیر آن‌ها را تغییر دهد نیز وجود دارند. در شکل ۱ ورودی‌های کاربر هنگام اجرای این تابع به همراه توضیحات آن‌ها و مقادیر پیشفرض‌شان آورده شده است.

```
## parsing args phase
parser = argparse.ArgumentParser(description='Distributed MNIST Classification')
parser.add_argument('--train-batch-size', type=int, default=32, metavar='N',
                    help='data batch size for training - default: 32')
parser.add_argument('--test-batch-size', type=int, default=128, metavar='N',
                    help='data batch size for testing - default: 128')
parser.add_argument('--data-root', type=str, default='/home/rostami/pytorch_ddp/dataset/data', metavar='/path/to/mnist',
                    help='MINST dataset path - default: /home/rostami/pytorch_ddp/dataset/data')
parser.add_argument('--shuffle', default=False, action='store_true',
                    help='flag to shuffle - default: not flag')
parser.add_argument('--num-workers', type=int, default=0, metavar='N',
                    help='data loader number of workers - default: 0')
parser.add_argument('--world-size', type=int, default=1, metavar='N',
                    help='number of processes - default: 1')
parser.add_argument('--epochs', type=int, default=3, metavar='N',
                    help='number of epochs - default: 3')
parser.add_argument('--optimizer-lr', type=float, default=5e-4, metavar='a.aaaa',
                    help='Adam init learning rate - default: 5e-4')
parser.add_argument('--optimizer-lr-decay', type=float, default=5e-4, metavar='a.aaaa',
                    help='Adam learning rate decay - default: 5e-4')
parser.add_argument('--num-classes', type=int, default=10, metavar='N',
                    help='number of MNIST classes - default: 10')
parser.add_argument('--log-path', type=str, default='/home/rostami/pytorch_ddp/tb_logs', metavar='/path/to/log/path',
                    help='Tensorboard log path - default: /home/rostami/pytorch_ddp/tb_logs')
parser.add_argument('--seed', type=int, default=123, metavar='N',
                    help='torch seed for instantiating model weights - default: 123')
parser.add_argument('--quantized', default=False, action='store_true',
                    help='flag to train quantized model - default: not flag')
parser.add_argument('--backend', type=str, default='gloo', metavar='gloo/mpi',
                    help='distributed communication backend - default: gloo')
args = parser.parse_args()
```

شکل ۱- متغیرهای ورودی کاربر برای اجرای فایل main.py

- *فایل quantization.html* : به دلیل محدودیت انجام تست‌های مختلف در قالب ساخته شده توسط فایل‌های پیشین برای مدل‌های کوانتیزه، فایل Jupyter Notebook ای برای این کار ساخته شده و خروجی html آن گرفته شده است. هر چند در همان قالب سابق، این مدل‌ها آموزش داده و خطایشان هم محاسبه شده‌اند.

در طول اجراهای مختلف، بعضی از مقادیر متغیرهای ورودی کاربر همیشه یکسان باقی می‌مانند که در این بخش به آن‌ها اشاره می‌کنیم.

- سایز batch برای داده‌های آموزش برابر ۳۲ در نظر گرفته شده است.

- سایز batch برای داده‌های تست برابر ۱۲۸ در نظر گرفته شده است.
- مجموعه داده‌های آموزش و تست MNIST در هیچ آزمایشی shuffle نمی‌شوند.
- تعداد worker ها هنگام ساخت data loader ها صفر در نظر گرفته شده است که از بیشترین تعداد worker های ممکن استفاده شود.
- تعداد ایپاک‌های آموزش ۳ در نظر گرفته شده است.
- نرخ یادگیری بهینه ساز در همه‌ی آزمایش‌ها 5e-4 در نظر گرفته شده است.
- مقدار کاهش نرخ یادگیری بهینه ساز در همه‌ی آزمایش‌ها 5e-4 در نظر گرفته شده است.
- مقدار seed برای کتابخانه‌ی پایتورچ در همه‌ی آزمایش‌ها ۱۲۳ در نظر گرفته شده است. تنظیم این مقدار کمک می‌کند تا تمام مدل‌های ساخته شده در ابتدا وزن‌های مشابهی داشته باشند.

با توجه به توابع گفته شده، کد اجرایی تمام بخش‌ها یکسان می‌شود و برای اجرای آزمایش‌های مختلف تنها با تغییر پارامترهای ورودی کاربر می‌توان آن‌ها را انجام داد.

سوال ۱

الف)

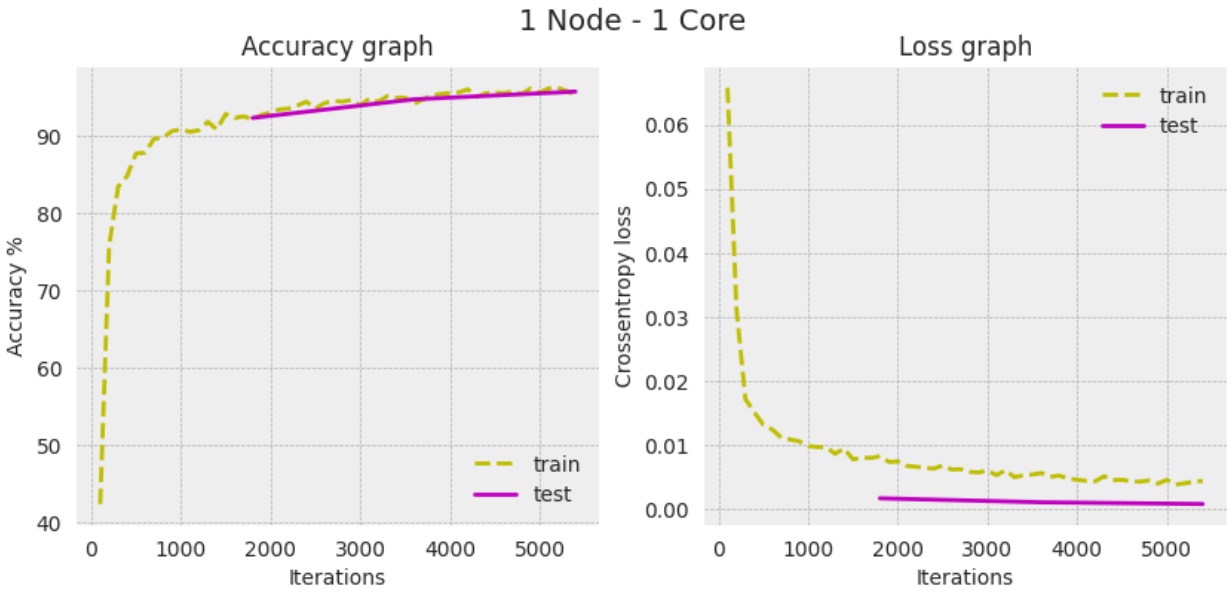
در ابتدا به بررسی ساختار شبکه‌ی عصبی استفاده می‌پردازیم. لایه‌ی اول این شبکه لایه‌ی کانولوشنی با سایز کرنل ۳ و padding ۱ است و تعداد کانال‌های ورودی آن ۱ و خروجی آن ۴ است. پس از آن تابع فعالساز ReLU قرار می‌گیرد. لایه‌ی بعدی MaxPooling ۲ بعدی با سایز کرنل ۲ و stride ۲ است. پس از آن لایه‌های کانولوشنی، ReLU و MaxPooling دوباره تکرار می‌شوند. با این تفاوت که تعداد کانال‌های ورودی و خروجی کانولوشنی این بار ۴ و ۸ در نظر گرفته می‌شود. پس از این لایه‌ها، داده‌ها به کمک دستور reshape تک بُعدی شده و به لایه‌ی fully connected ای با تعداد ورودی‌های ۳۹۲ و خروجی‌های ۱۰ (تعداد کلاس‌های مجموعه داده‌ی MNIST) داده شده و نتیجه‌ی نهایی به دست می‌آید. در شکل ۲ این مدل قابل مشاهده است.

```
ConvNet(
  (conv1): Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu1): ReLU()
  (max1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(4, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu2): ReLU()
  (max2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (linear): Linear(in_features=392, out_features=10, bias=True)
)
```

شکل ۲- ساختار مدل استفاده شده

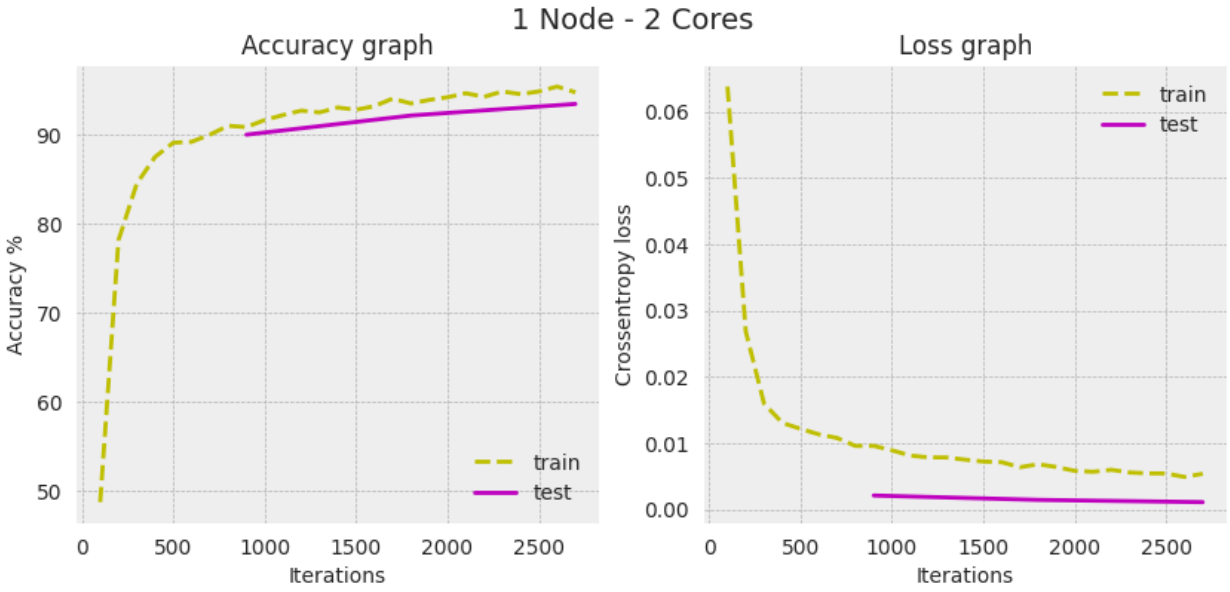
این مدل در مجموع دارای ۴۲۶۶ پارامتر است و مدل بسیار کوچکی است زیرا اجرای مدل‌های بزرگتر بر روی CPU می‌تواند بسیار زمانبر باشد. همچنین مجموعه داده‌ی MNIST به راحتی دسته بندی می‌شود و حتی مدل‌هایی در این سایز هم می‌توانند دقت خوبی داشته باشند.

a) در حالتی که از یک ماشین و یک هسته برای آموزش مدل استفاده شد، دقت نهایی مدل برای داده‌های آموزشی و تست به ترتیب ۹۵.۲۸٪ و ۹۷.۷۷٪ به دست آمد. همچنین زمان آموزش مدل حدودا ۱۸۶ ثانیه طول کشید. در شکل ۳ نمودار خطا و دقت برای داده‌های آموزش و تست کشیده شده است.



شکل ۳- نمودار خطا و دقت آموزش مدل با یک ماشین و یک هسته

b) در حالتی که از یک ماشین و دو هسته برای آموزش مدل استفاده شد، دقت نهایی مدل برای داده‌های آموزشی و تست به ترتیب ۹۴.۵ و ۹۳.۶ برای رنک ۰ و ۹۵ و ۹۳.۳۲ برای رنک ۱ به دست آمد. همچنین زمان آموزش حدودا ۱۲۰ ثانیه طول کشید. در شکل ۴ نمودار میانگین دقت و خطای رنک‌های مختلف برای داده‌های آموزش و تست رسم شده است.

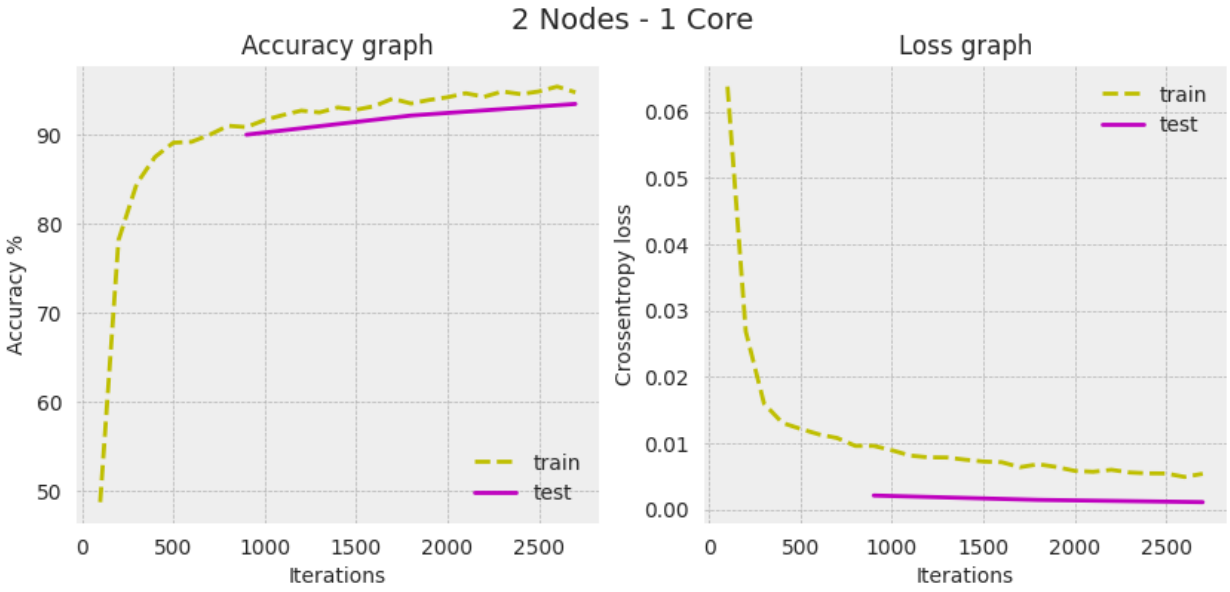


شکل ۴ - نمودار خطا و دقت آموزش مدل با یک ماشین و دو هسته

ب)

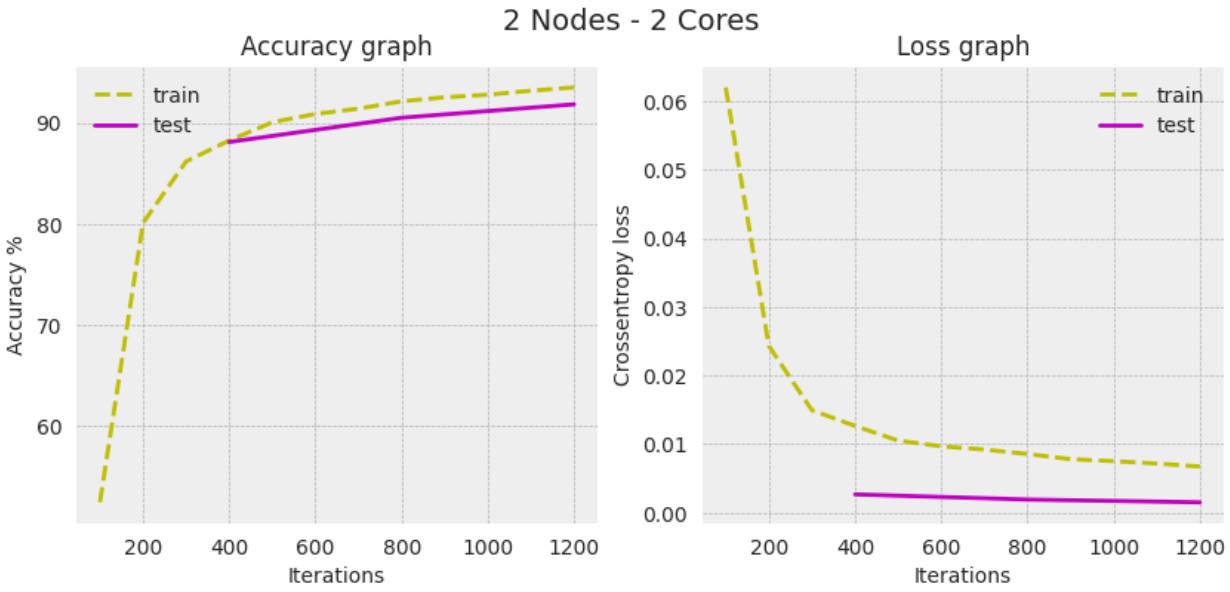
در این بخش، مدل مدنظر بر روی دو ماشین و یک هسته و دو ماشین و دو هسته آموزش داده شده و نتایج آن ذکر می‌شوند.

a) در حالتی که از ۲ ماشین و یک هسته برای آموزش مدل استفاده شد، دقت نهایی مدل برای داده‌های آموزش و تست به ترتیب ۹۴.۵ و ۹۳.۶۷ برای رنک ۰ و ۹۵ و ۹۳.۳۲ برای رنک ۱ به دست آمد. همچنین زمان آموزش حدودا ۹۹ ثانیه طول کشید. در شکل ۵ نمودار میانگین دقت و خطا رنک‌های مختلف برای داده‌های آموزش و تست رسم شده است.



شکل ۵- نمودار خطا و دقت آموزش مدل با دو ماشین و یک هسته

b) در حالتی که از دو ماشین و ۲ هسته برای آموزش مدل استفاده شد، دقت نهایی مدل برای داده‌های آموزش و تست به ترتیب ۹۳.۶۸٪ و ۹۲.۲۶٪ برای رنک ۰، ۹۳.۱۸٪ و ۹۱.۲۱٪ برای رنک ۱، ۹۳.۵۶٪ و ۹۲.۰۷٪ برای رنک ۲ و ۹۳.۸۱٪ و ۹۱.۹۹٪ برای رنک ۳ به دست آمدند. همچنین زمان آموزش حدودا ۶۷ ثانیه طول کشید. در شکل ۶ نمودار میانگین دقت و خطای رنک‌های مختلف برای داده‌های آموزش و تست رسم شده است.



شکل ۶- نمودار خطا و دقت آموزش مدل با دو ماشین و دو هسته

ج)

با مقایسه‌ی نتایج به دست آمده توسط بخش‌های الف و ب به این نتیجه می‌رسیم که با افزایش تعداد نودها و هسته‌ها، دقت مدل‌ها کمتر می‌شود. زیرا هر مدل با داده‌های کمتری آموزش می‌بیند و اگر داده‌ها زیاد باشند یا تعداد iteration ها افزایش یابد، احتمالا افت دقتی اتفاق نمی‌افتد. در جدول ۱ میانگین دقت مدل‌های نهایی رنک‌ها بر روی داده‌ی آموزش در حالت‌های مختلف قابل مشاهده است.

جدول ۱- میانگین دقت مدل‌های نهایی رنک‌ها بر روی مجموعه داده‌ی تست در حالت‌های مختلف

۱ ماشین، ۱ هسته	۱ ماشین، ۲ هسته	۲ ماشین، ۱ هسته	۲ ماشین، ۲ هسته	
۹۷.۷۷	۹۳.۴۶	۹۳.۴۹	۹۱.۸۸	دقت (داده‌ی تست)

همچنین با افزایش هسته‌ها و نودها زمان آموزش کاهش می‌یابد. زیرا آموزش مدل بین نودها و هسته‌ها تقسیم می‌شود. هر چند این کاهش کاملا خطی نیست. زیرا سربار ارتباطات مانع از افزایش سرعت خطی می‌شود. در جدول ۲ مدت زمان آموزش مدل در حالت‌های مختلف قابل مشاهده است.

جدول ۲ – مدت زمان آموزش مدل در حالت‌های مختلف

۱ ماشین، ۱ هسته	۱ ماشین، ۲ هسته	۲ ماشین، ۱ هسته	۲ ماشین، ۲ هسته	
۱۸۶	۱۲۰	۹۹	**۶۷**	زمان آموزش (ثانیه)

در جدول ۲، مدت زمان آموزش مدل بر روی ۲ ماشین و ۱ هسته کمتر از ۱ ماشین و ۲ هسته شده است. زیرا در این تمرین، مدل‌ها بر روی CPU آموزش می‌بینند و احتمالا استفاده از ۱ هسته‌ی ۲ CPU ی مجزا بهتر از استفاده از ۲ هسته‌ی ۱ CPU است.

سوال ۲

الف)

کتابخانه‌ی پایتورچ از ۳ روش quantization پشتیبانی می‌کند. در روش **dynamic quantization** تنها وزن‌ها کوانتیزه می‌شوند و مقادیر activation ها با همان حجم قبلی بر روی حافظه نوشته یا از روی آن خوانده می‌شود و تنها برای محاسبات کوانتیزه می‌شود. در روش **static quantization** هم وزن‌ها و هم activation ها کوانتیزه می‌شود. سومین روش هم **static quantization aware training** است که وزن‌ها و activation ها در حین آموزش مدل کوانتیزه می‌شوند.

روش‌های **dynamic quantization** و **static quantization** هر دو روش‌های post training هستند و در حین آموزش مدل هیچ کاری نمی‌کنند و تنها بعد از آموزش کامل مدل، مدل نهایی را کوانتیزه می‌کنند. در حالی که در روش **static quantization aware training** وزن‌ها و activation ها در طول آموزش کوانتیزه می‌شوند. در نتیجه از نظر زمان آموزش، روش‌های **dynamic quantization** و **static quantization** تفاوتی با حالت‌های قبل ندارند. در حالی که روش **static quantization aware training** بیشتر از حالت‌های قبلی طول می‌کشد. زیرا محاسبات مربوط به کوانتیزه هم در طول آموزش انجام می‌شود.

از نظر دقت هم، مطابق جدول شکل ۷، روش **static quantization aware training** بیشترین دقت را حفظ می‌کند. به علاوه‌ی اینکه روش **dynamic quantization** تنها برای مدل‌های LSTM، MLP و ترنسفورمرها مناسب است و تنها روش‌های **static quantization** و **static quantization aware training** برای مدل‌های کانولوشنی مناسبند. در نتیجه برای این بخش تنها از روش **static quantization aware training** استفاده می‌شود.

کتابخانه‌ی پایتورچ برای استفاده از روش‌های کوانتیزه کردن، ۲ روش Eager و FX graph را ارائه داده است که برای پیاده سازی این بخش، از روش Eager استفاده شده است.

WORKFLOWS

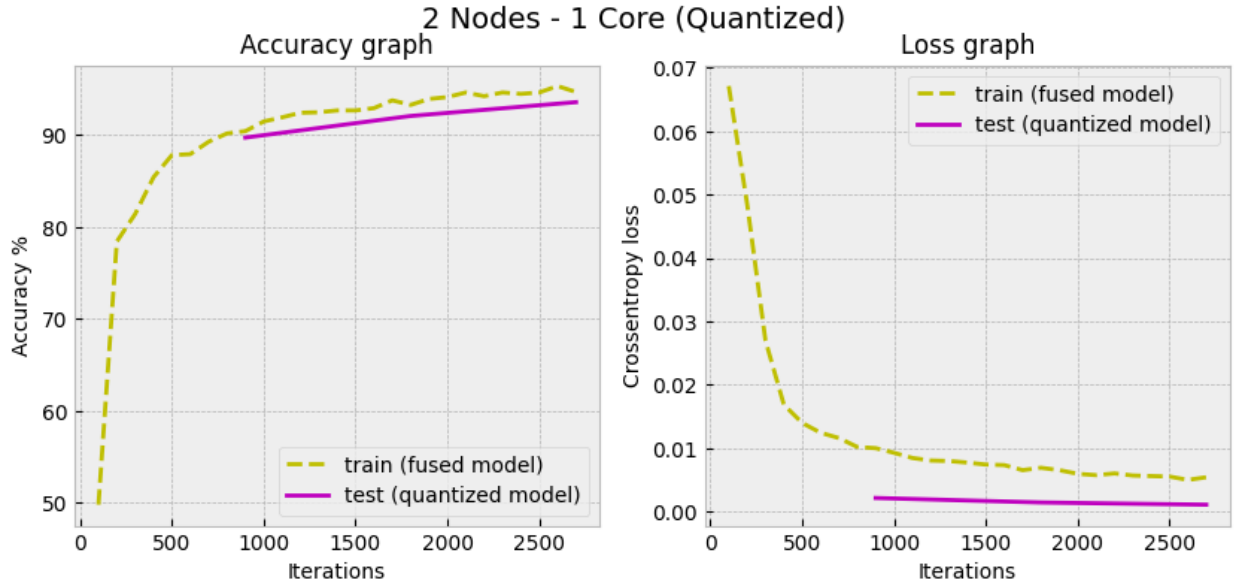
	Quantization	Dataset Requirements	Works Best For	Accuracy	Notes
Dynamic Quantization	weights only (both fp16 and int8)	None	LSTMs, MLPs, Transformers	good	Suitable for dynamic models (LSTMs), Close to static post training quant when performance is compute bound or memory bound due to weights.
Static Post Training Quantization	weights and activations (8 bit)	calibration	CNNs	good	Suitable for static models, provides best perf
Static Quantization-Aware Training	weights and activations (8 bit)	fine-tuning	CNNs	best	Requires fine tuning of model, currently supported only for static quantization.

شکل ۷- مقایسه‌ی روش‌های مختلف کوانتیزه کردن مدل‌ها در پایتورچ[1]

[1] Deep Dive on PyTorch Quantization – Chris Gottbrath (from PyTorch YouTube channel: link)

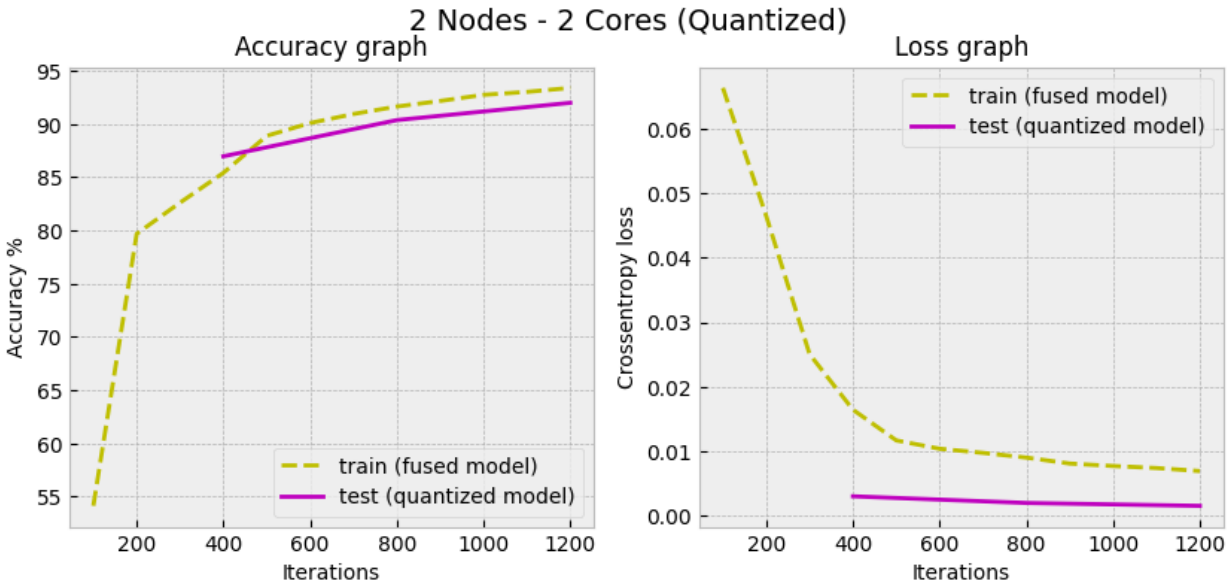
برای پیاده سازی این بخش، ابتدا backend کوانتیزه کردن مدل qnnpack در نظر گرفته شده و سپس ماژول‌های [[conv1, relu1], [conv2, relu2]] فیوز می‌شوند. بقیه‌ی لایه‌ها (لایه‌های fully connected و MaxPooling) نیازی به فیوز کردن ندارند زیرا به راحتی به نسخه‌ی کوانتیزه شده تبدیل می‌شوند. سپس مدل فیوز شده آموزش می‌بیند و در انتهای هر ایپاک، مدل کوانتیزه شده (مدل int8) از مدل فیوز شده به دست آمده و دقت و خطای آن بر روی داده‌ی تست سنجیده می‌شود.

در حالتی که از دو ماشین و یک هسته برای آموزش مدل با روش **static quantization aware training** استفاده شد، دقت مدل فیوز شده بر روی داده‌های آموزش به ترتیب ۹۴.۶۲٪ و ۹۴.۷۱٪ برای رنک‌های ۰ و ۱ و دقت مدل کوانتیزه شده (int8) بر روی داده‌های تست به ترتیب ۹۳.۶۵٪ و ۹۳.۵۳٪ برای رنک‌های ۰ و ۱ به دست آمد. همچنین زمان آموزش حدودا ۱۳۵ ثانیه طول کشید. در شکل ۸، نمودار خطا و دقت در طول آموزش برای داده‌های آموزش و تست رشم شده است.



شکل ۸- نمودار دقت و خطای آموزش مدل کوانتیزه بر روی دو ماشین و یک هسته

در حالتی که از دو ماشین و دو هسته برای آموزش مدل با روش **static quantization aware training** استفاده شد، دقت مدل فیوز شده بر روی داده‌های آموزش به ترتیب ۹۳.۴۳٪ ، ۹۲.۹۶٪ ، ۹۳.۶۵٪ و ۹۳.۶۲٪ برای رنک‌های ۰، ۱، ۲ و ۳ و دقت مدل کوانتیزه شده (int8) بر روی داده‌های تست به ترتیب ۹۲.۰۳٪ ، ۹۱.۶٪ ، ۹۲.۳۸٪ و ۹۲.۰۳٪ برای رنک‌های ۰، ۱، ۲ و ۳ به دست آمد. همچنین زمان آموزش حدودا ۹۱ ثانیه طول کشید. در شکل ۹، نمودار خطا و دقت در طول آموزش برای داده‌های آموزش و تست رشم شده است.



شکل ۹- نمودار دقت و خطای آموزش مدل کوانتیزه بر روی ۲ ماشین و ۲ هسته

همچنین در جدول ۳ مقایسه‌ی دقت حالت استفاده از **static quantization aware training** با حالت عدم استفاده از آن وجود دارد. طبق مقایسه‌ای که در این جدول قابل مشاهده است، مدل‌های کوانتیزه شده توانسته‌اند حتی به دقت بهتری نسبت به مدل‌های اصلی برسند.

جدول ۳- مقایسه‌ی دقت مدل‌های کوانتیزه با مدل‌های اصلی

	۲ ماشین، ۲ هسته	۲ ماشین، ۱ هسته
دقت (داده‌ی تست) در حالت عادی	۹۱.۸۸	۹۳.۴۹
دقت (داده‌ی تست) در کوانتیزه	**۹۲.۰۱**	**۹۳.۵۹**

در جدول ۴ هم مقایسه‌ی مدت زمان آموزش مدل‌های فیوز شده با مدل‌های عادی آمده است. همانطور که مشخص است، زمان آموزش در حالت عادی نزدیک به ۳۵٪ زمانبرتر است. همانطور که گفته شد، دلیل این اتفاق این است که در روش **static quantization aware training** محاسبات quantization در طول آموزش انجام می‌شود و مدل کوانتیزه همزمان با آموزش مدل فیوز شده مدام ساخته می‌شود تا دقت بهتری داشته باشد. به همین علت، حجم محاسبات آن نسبت به آموزش مدل عادی بیشتر است.

جدول ۴- مقایسه‌ی زمان آموزش مدل‌های کوانتیزه با مدل‌های اصلی

۲ ماشین، ۱ هسته	۲ ماشین، ۲ هسته	
۹۹	**۶۷**	زمان آموزش (ثانیه) در حالت عادی
۱۳۵	۹۱	زمان آموزش (ثانیه) در کوانتیزه

مقایسه‌ی بیشتر مدل کوانتیزه با مدل عادی

اصلی‌ترین مزیت مدل‌های کوانتیزه نسبت به مدل‌های عادی، حجم کوچکتر آن‌ها و پیشبینی (inference) های سریع‌ترشان است. در این بخش به طور مختصر به این موارد پرداخته می‌شود.

در ابتدا به مقایسه‌ی سایز checkpoint مدل اصلی با مدل کوانتیزه (int8) با بک‌اِند qnnpack پرداخته می‌شود. برای اینکار یک بار مدل عادی و بار دیگر مدل کوانتیزه بر روی حافظه ذخیره می‌شوند و حجم آن‌ها به دست محاسبه می‌شوند. در این آزمایش حجم مدل عادی ۱۹ کیلوبایت و مدل کوانتیزه ۸.۷ کیلوبایت به دست آمد که نزدیک به 2.2x کمتر است.

برای مقایسه‌ی حافظه‌ی مصرفی و زمان اجرای هر دو مدل، از ماژول profiler پایتورچ استفاده شده است. در جدول ۵ خروجی‌های این ماژول برای مدل عادی و در جدول ۶ همان خروجی‌ها در شرایطی مشابه برای مدل کوانتیزه آمده است. همانطور که در جدول ۵ مشخص است، برای مدل اصلی، استنتاج (inference) مقداری داده‌ی مشخص، حدودا ۷۵ میلی ثانیه طول کشیده است. هر چند این مقدار در اجراهای متفاوت ممکن است کمی زیاد یا کم شود ولی به طور میانگین، همین مقدار طول می‌کشد. در حالی که در جدول ۶، زمان استنتاج برای مدل کوانتیزه تنها ۲۰ میلی ثانیه است که نزدیک به 3.75x سریع‌تر است. همچنین ستون Self CPU Mem نشان دهنده‌ی حافظه‌ی RAM مصرفی دو مدل است و همانطور که مشخص است، حافظه‌ی مصرفی مدل عادی در مجموع 4.74MB و مدل کوانتیزه 4.61KB است که ۱۰۰۰ برابر کمتر است.

جدول 5 - نتایج profiler مدل عادی

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	CPU Mem	Self CPU Mem	# of Calls
aten::empty	0.27%	201.000us	0.27%	201.000us	11.167us	1.72 Mb	1.72 Mb	18
aten::conv2d	0.12%	90.000us	48.55%	36.585ms	6.098ms	1.72 Mb	0 b	6
aten::convolution	0.38%	289.000us	48.43%	36.495ms	6.082ms	1.72 Mb	0 b	6
aten::_convolution	0.28%	212.000us	48.04%	36.206ms	6.034ms	1.72 Mb	0 b	6
aten::_nnpack_spatial_convolution	47.62%	35.887ms	47.75%	35.987ms	5.998ms	1.72 Mb	0 b	6
aten::relu	0.35%	266.000us	43.17%	32.535ms	5.423ms	1.72 Mb	0 b	6
aten::clamp_min	42.82%	32.269ms	42.82%	32.269ms	5.378ms	1.72 Mb	1.72 Mb	6
aten::max_pool2d	0.10%	79.000us	3.01%	2.270ms	378.333us	1.29 Mb	0 b	6
aten::max_pool2d_with_indices	2.91%	2.191ms	2.91%	2.191ms	365.167us	1.29 Mb	1.29 Mb	6
aten::linear	0.06%	46.000us	1.03%	778.000us	259.333us	3.75 Kb	0 b	3
aten::addmm	0.61%	462.000us	0.80%	604.000us	201.333us	3.75 Kb	3.75 Kb	3
aten::zeros	0.10%	77.000us	0.21%	155.000us	25.833us	24 b	0 b	6
aten::zero_	0.01%	9.000us	0.01%	9.000us	1.500us	0 b	0 b	6
aten::_nnpack_available	0.01%	7.000us	0.01%	7.000us	1.167us	0 b	0 b	6
aten::reshape	0.05%	35.000us	0.15%	112.000us	37.333us	0 b	0 b	3
aten::_reshape_alias	0.10%	77.000us	0.10%	77.000us	25.667us	0 b	0 b	3
aten::t	0.08%	64.000us	0.17%	128.000us	42.667us	0 b	0 b	3
aten::transpose	0.05%	35.000us	0.08%	64.000us	21.333us	0 b	0 b	3
aten::as_strided	0.05%	36.000us	0.05%	36.000us	6.000us	0 b	0 b	6
aten::expand	0.04%	29.000us	0.05%	36.000us	12.000us	0 b	0 b	3
aten::copy_	0.14%	102.000us	0.14%	102.000us	34.000us	0 b	0 b	3
aten::resolve_conj	0.01%	4.000us	0.01%	4.000us	0.667us	0 b	0 b	6
ProfilerStep*	0.34%	258.000us	99.84%	75.241ms	25.080ms	-12 b	-876 b	3
model_inference	3.50%	2.634ms	99.43%	74.929ms	24.976ms	-12 b	-4.74 Mb	3

Self CPU time total: 75.359ms

جدول 6 - نتایج profiler مدل کوانتیزه

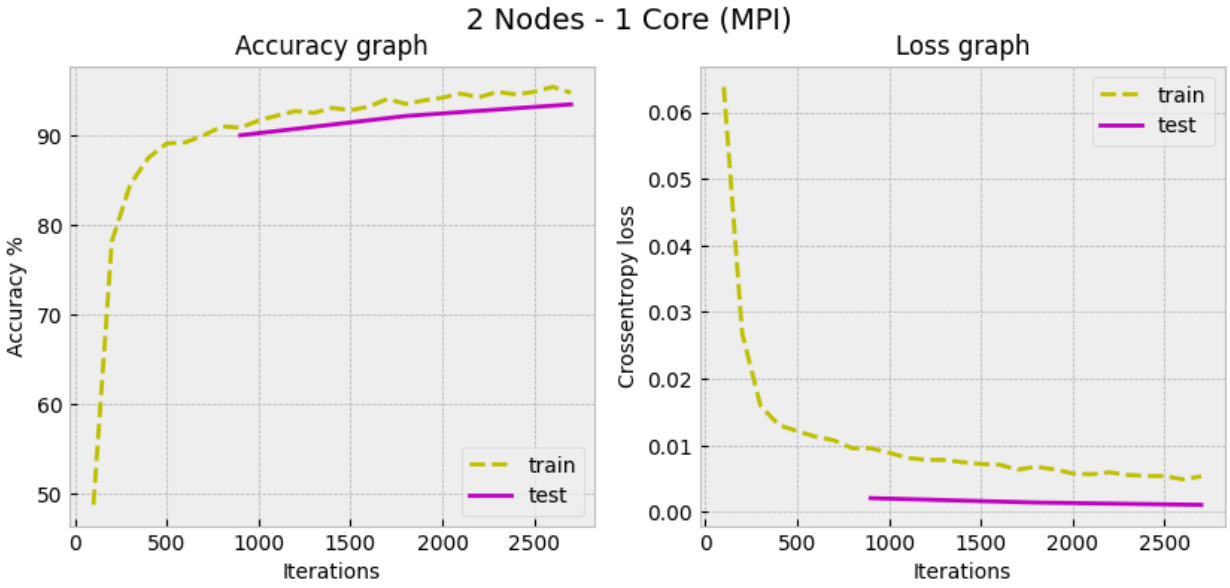
Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	CPU Mem	Self CPU Mem	# of Calls
aten::empty	0.97%	188.000us	0.97%	188.000us	12.533us	5.46 Kb	5.46 Kb	15
aten::dequantize	0.34%	65.000us	0.87%	168.000us	56.000us	3.75 Kb	0 b	3
aten::zeros	0.34%	65.000us	0.65%	125.000us	20.833us	24 b	0 b	6
aten::zero_	0.05%	9.000us	0.05%	9.000us	1.500us	0 b	0 b	6
aten::item	0.16%	30.000us	0.24%	47.000us	7.833us	0 b	0 b	6
aten::_local_scalar_dense	0.09%	17.000us	0.09%	17.000us	2.833us	0 b	0 b	6
aten::quantize_per_tensor	1.34%	259.000us	1.34%	259.000us	86.333us	0 b	0 b	3
quantized::conv2d_relu	56.01%	10.818ms	56.25%	10.864ms	1.811ms	0 b	0 b	6
aten::q_scale	0.28%	55.000us	0.28%	55.000us	3.056us	0 b	0 b	18
aten::q_zero_point	0.19%	36.000us	0.19%	36.000us	1.500us	0 b	0 b	24
aten::max_pool2d	0.30%	58.000us	7.53%	1.455ms	242.500us	0 b	0 b	6
aten::quantized_max_pool2d	6.38%	1.233ms	7.23%	1.397ms	232.833us	0 b	0 b	6
aten::_empty_affine_quantized	0.97%	187.000us	0.97%	187.000us	15.583us	0 b	0 b	12
aten::reshape	0.27%	53.000us	2.13%	412.000us	137.333us	0 b	0 b	3
aten::clone	1.51%	292.000us	1.68%	324.000us	108.000us	0 b	0 b	3
aten::qscheme	0.04%	7.000us	0.04%	7.000us	0.778us	0 b	0 b	9
aten::_unsafe_view	0.18%	35.000us	0.18%	35.000us	11.667us	0 b	0 b	3
quantized::linear	0.71%	138.000us	0.94%	181.000us	60.333us	0 b	0 b	3
ProfilerStep*	1.30%	251.000us	99.54%	19.226ms	6.409ms	-12 b	-876 b	3
model_inference	28.57%	5.519ms	97.97%	18.923ms	6.308ms	-12 b	-4.61 Kb	3

Self CPU time total: 19.315ms

ب)

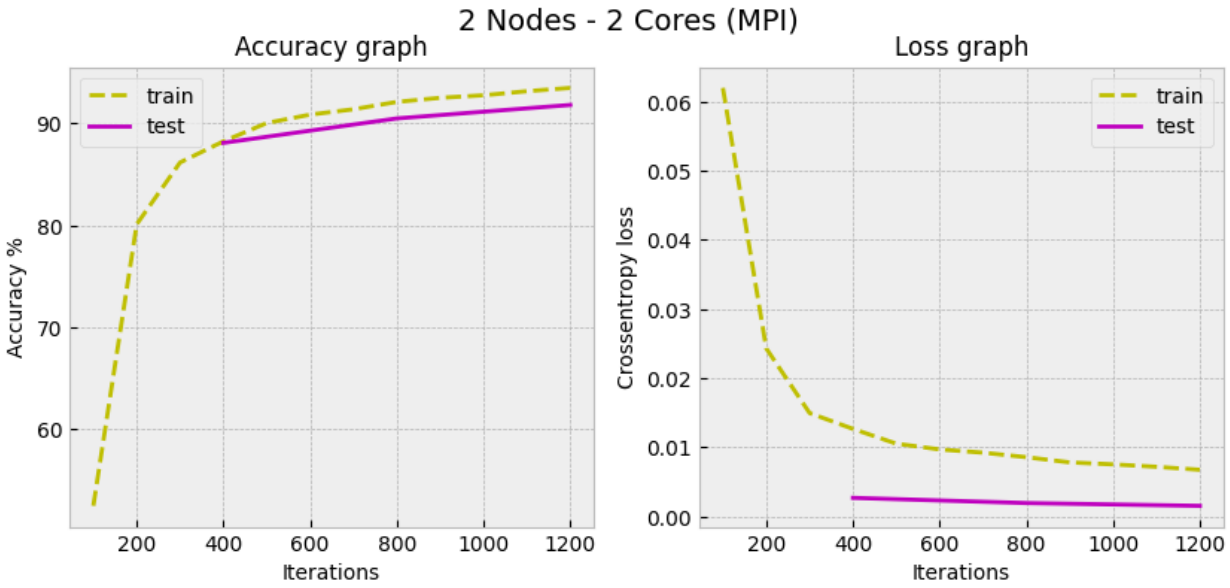
در این بخش از بک‌اِند mpi به جای gloo استفاده کرده و نتایج آن‌ها را با همدیگر مقایسه می‌کنیم. در این سوال، در بخش استفاده از ۲ نود و هر نود ۲ هسته، به دلیل خطای srun و torchrun هنگام استفاده از بک‌اِند mpi، مجبور به استفاده از دستور mpirun به جای srun و عدم استفاده از torchrun شدیم. همچنین در فایل sbatch، تعداد نودها و تعداد تسک‌های هر نود ۲ در نظر گرفته شد. در دستور mpirun هم تعداد فرایندها ۴ و با تنظیم پارامتر --bind-to با core ، فرایندها را بین هسته‌ها پخش کردیم. این مشکل فقط برای حالت اجرا بر روی ۲ نود و ۲ هسته وجود داشت و هنگام استفاده از بک‌اِند mpi بر روی ۲ نود و ۱ هسته، این مشکل پیش نیامد.

در حالتی که از ۲ ماشین و ۱ هسته برای آموزش مدل با بک‌اِند mpi استفاده شد، دقت مدل بر روی داده‌های آموزش به ترتیب ۹۴.۵٪ و ۹۵٪ برای رنک‌های ۰ و ۱ و بر روی داده‌های تست ۹۳.۶۷٪ و ۹۳.۳۲٪ برای رنک‌های ۰ و ۱ بود. همچنین آموزش مدل حدودا ۹۵ ثانیه طول کشید. در شکل ۱۰ نمودار خطا و دقت آموزش مدل بر روی داده‌های آموزشی و تست قابل مشاهده است.



شکل ۱۰- نمودار خطا و دقت آموزش مدل با بک‌اِند mpi بر روی ۲ ماشین و ۱ هسته

در حالتی که از ۲ ماشین و ۲ هسته برای آموزش مدل با بک‌اِند mpi استفاده شد، دقت مدل بر روی داده‌های آموزش به ترتیب ۹۳.۶۸٪ ، ۹۳.۱۸٪ ، ۹۳.۵۶٪ و ۹۳.۸۱٪ برای رنک‌های ۰، ۱، ۲ و ۳ و بر روی داده‌های تست ۹۲.۲۶٪ ، ۹۱.۲۱٪ ، ۹۲.۰۷٪ و ۹۱.۹۹٪ برای رنک‌های ۰، ۱، ۲ و ۳ بود. همچنین آموزش مدل حدودا ۵۹.۵ ثانیه طول کشید. در شکل ۱۱ نمودار خطا و دقت آموزش مدل بر روی داده‌های آموزشی و تست قابل مشاهده است.



شکل ۱۱- نمودار خطا و دقت آموزش مدل با بک‌اِند mpi بر روی ۲ ماشین و ۲ هسته

در جدول ۷، دقت مدل‌های آموزش دیده با بک‌اِند mpi و بک‌اِند gloo با همدیگر مقایسه شده‌اند. همانطور که انتظار می‌رفت، تغییر بک‌اِند هیچ تاثیری بر دقت مدل‌ها ندارد. زیرا بک‌اِندهای gloo و mpi تنها در نوع ارتباطاتشان با یکدیگر فرق دارند.

جدول ۷- مقایسه‌ی دقت مدل‌های آموزش داده شده با بک‌اِندهای mpi و gloo

	۲ ماشین، ۲ هسته	۲ ماشین، ۱ هسته
دقت (داده‌ی تست) با بک‌اِند mpi	۹۱.۸۸	۹۳.۴۹
دقت (داده‌ی تست) با بک‌اِند gloo	۹۱.۸۸	۹۳.۴۹

همچنین در جدول ۸، مدت زمان آموزش مدل‌ها با بک‌اِندهای mpi و gloo مقایسه شده‌اند. همانطور که مشخص است، آموزش مدل با بک‌اِند mpi کمی کمتر از آموزش با بک‌اِند gloo زمان می‌برد.

جدول ۸- مقایسه‌ی زمان آموزش مدل‌ها با بک‌اِند mpi و gloo

	۲ ماشین، ۲ هسته	۲ ماشین، ۱ هسته
زمان آموزش (ثانیه) با بک‌اِند mpi	**۵۹.۵**	**۹۵**
زمان آموزش (ثانیه) با بک‌اِند gloo	۶۷	۹۹