HPC term project - Diabetic Retinopathy

Ricco Ferraro

Goal:

Train a computationally expensive predictive model which is capable of predicting whether or not an image of a retina has diabetic retinopathy. Profile and/or time running this locally and on an m2 cluseter with slurm. Varying the degrees of optimization to compare the time to train. Optimize for the lowest copute time.

The Dataset

A ~8Gb training set was available in kaggle here: from https://www.kaggle.com/c/diabetic-retinopathy-detection

This dataset contains a large collection of over 45,000 retinal images with binary classifications of retinopathy or not retinopathy for each image in a csv alongside the images.

It was chosen because of it's size and complexity.

Pytorch Lightning In a Nutshell

Pytorch Lightning is provides a nice API for orchestrating machine learning tasks via slurm. It is syntatically similar to pytorch, with a few differences. Lightning takes advantage of lazy loading with test dataloaders and validation dataloaders. It also makes use of embarassingly parallel tasks (like batched gradient descent) to parallelize gradient descent work across multiple nodes via slurm and GPU's with accelerators. It is a python orchestrator which sits on top of nickel and slurm. It is capable of running on on-prem slurm clusters which makes it desirable for m2.

The Model

The model used in this study was inspired and largely a port of a pytorch implementation submitted to kaggle here: https://www.kaggle.com/code/fanbyprinciple/pytorch-diabetic-retinopathy The goal was to port and or wrap this solution to pytorch lightning.

The solution leverages google's inception v3 transfer learning deep Convolutional Neural Network model as implemented in a pytorch neural network model. Training this model in vanilla pytorch locally (on an 8 core Intel i9 Mac with 32 gb of RAM and NO NVIDEA GPU) takes about 24 hours to train on 5000 images as a baseline for 100 epochs and a batch size of 32.

In this work, I ported the model from something like this:

```
from retina_dataset import RetinaDataset

ssl._create_default_https_context = ssl._create_unverified_context

def get_cropped_data(relative_path = "../input/diabetic-retinopathy-
resized/trainLabels_cropped.csv", count=5000):
```

```
    return pd.read_csv(relative_path)[:count]


def train_model(train_dataloader):
    learning_rate = 1e-4
    num_epochs = 1
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = inception_v3(pretrained=True)

    for param in model.parameters():
        param.requires_grad = False

    model.fc = torch.nn.Linear(in_features=2048, out_features=5,
bias=True)
    model.aux_logits = False
    model = model.to(device=device)

    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)
    loss_criterion = torch.nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        for data, target in tqdm(train_dataloader):
            data = data.to(device=device)
            target = target.to(device=device)

            score = model(data)
            optimizer.zero_grad()

            loss = loss_criterion(score, target)
            loss.backward()

            optimizer.step()

        print(f"for epoch {epoch}, loss : {loss}")

    return model

def check_accuracy(model, loader):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.eval()

    correct_output = 0
    total_output = 0

    with torch.no_grad():
        for x, y in tqdm(loader):
            x = x.to(device=device)
            y = y.to(device=device)

            score = model(x)
            _,predictions = score.max(1)

            correct_output += (y==predictions).sum()
            total_output += predictions.shape[0]
```

```
    model.train()
    print(f"out of {total_output} , total correct: {correct_output} with
an accuracy of {float(correct_output/total_output)*100}")
```

To Something like this

```
class InceptionV3LightningModel(pl.LightningModule):
    def __init__(
        self,
        num_epochs = 1,
        in_features = 2048,
        out_features=5,
        bias=True,
        aux_logits = False,
    ):
        super().__init__()
        self._model = inception_v3(pretrained=True)
        for param in self._model.parameters():
            param.requires_grad = False
        # Be careful not to overwrite `pl.LightningModule` attributes such
as `self.model`.
        self._model.fc = torch.nn.Linear(in_features=in_features,
out_features=out_features, bias=bias)
        self._num_epochs = num_epochs
        self._in_features = in_features
        self._out_features = out_features
        self._bias = bias
        self._aux_logits = aux_logits
        self._loss_criterion = torch.nn.CrossEntropyLoss()

    def forward(self, x):
        return self._model(x)

    def get(self, item):
        return self.__dict__[item]

    def training_step(self, train_batch, batch_idx):
        X, y = train_batch
        (y_hat, _) = self._model(X)
        loss = self._loss_criterion(y_hat, y)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, batch, batch_idx):
        X, y = batch
        y_hat = self._model(X)
        loss = self._loss_criterion(y_hat, y)
        self.log("valid_loss", loss, prog_bar=True)
        # self.log("val_acc", self._model.accuracy(y_hat, y),
prog_bar=True)
        return batch_idx
```

```
    def configure_optimizers(self):
        return torch.optim.Adam(self._model.parameters(), lr = 1e-4)
```

Optimzation Targets

Optimization targets evaluated in this study included:

- Level of parallelization across nodes
- Parallelization via GPU or no GPU
- File Type serialization for Tensor objects representing images

Evaluation Metric

Total Run Time

Experiments

Experiment 1: Number of Nodes

Goal: determine the affect of adding more nodes to running our model in a slurm cluster. **Constant Configuration:**

- 5000 images per training set. 500 validation set
- 1000 Epochs
- CrossEntropyLoss
- Batch size of 32
- LEarning Rate of
- 10G memory per node
- 16 CPU Tasks per node
- gpgpu-1 node type

Varried Configurations:

- 2 Nodes WITH GPU
- 8 Nodes WITH GPU

Redundancy:

- 2-3 runs per configuration

Test: If the number speed increased significantly from 1 node without GPU to 8 nodes with GPU. The percentage of this speed increase will inform our inference on the Experiment 2.

Experiment 2: The affect of NVIDEA GPUs

Goal: determine the affect of adding GPU's to our model **Constant Configuration:**

- 10G memory per node
- 16 CPU Tasks per node
- gpgpu-1 node type

Varried Configurations: -1 Node NO GPU -2 Nodes WITH GPU

Redundancy:

- 2-4 runs per configuration

Test: If the speed increased exceeds the percentage increase in `Experiment 1` when comparing the run with 1 node without GPU to 2 nodes each with a GPU, we can infer that the addition of a GPU contributed to a faster run time. For a CNN we expect this.

Experiment 3 Tensor Serialization: To determine the affect of file extension type and serialization. In this case, we use pytorch to pre-serialize the transformed images **Configuration**:

- smaller dataset 500 total files
- smaller training size 180, validation 20. Max 15 Epochs for brevity.
- ran locally (m2 was VERY busy with 1000's of jobs and this shouald behave similar on my MAC)

Varried Configurations: -1 Pre-Serialize Transformed Image Tensors, load these instead of the images and transfomring them. -2 Do NOT serialize image tensors ahead of time. Load the image and transform during training period.

Test: If the speed increased when pre-serializeing transformed image tensors when compared to doing that as part of training.

Results:

Experiment 1:

2 Nodes With 1 GPU for each node (Decently Fast)

Run 1: completed training at time --- 21016.801413297653 seconds --- Run 2: completed training at time --- 21281.795115232468 seconds --- Run 3: completed training at time --- 21172.826137065887 seconds --- Run 4: completed training at time --- 21067.387937784195 seconds --- MEAN: 21134.70265084505 seconds

8 Nodes With 1 GPU for each node (Even FASTER)

Run 1: completed training at time --- 7304.014988899231 seconds --- Run 2: completed training at time --- 7339.133647680283 seconds --- MEAN: 7321.574318289757 seconds

Experiment 2:

Node NO GPU (VERY SLOW)

- Run 1: completed training at time --- 197146.97274017334 seconds ---
- Run 2: completed training at time --- 199454.64568400383 seconds --- MEAN: 198300.80921208858 seconds

Node With 1 GPU for each node (MUCH FASTER)

- Run 1: completed training at time --- 21016.801413297653 seconds ---

- Run 2: completed training at time --- 21281.795115232468 seconds ---
- Run 3: completed training at time --- 21172.826137065887 seconds ---
- Run 4: completed training at time --- 21067.387937784195 seconds --- MEAN: 21134.70265084505 seconds

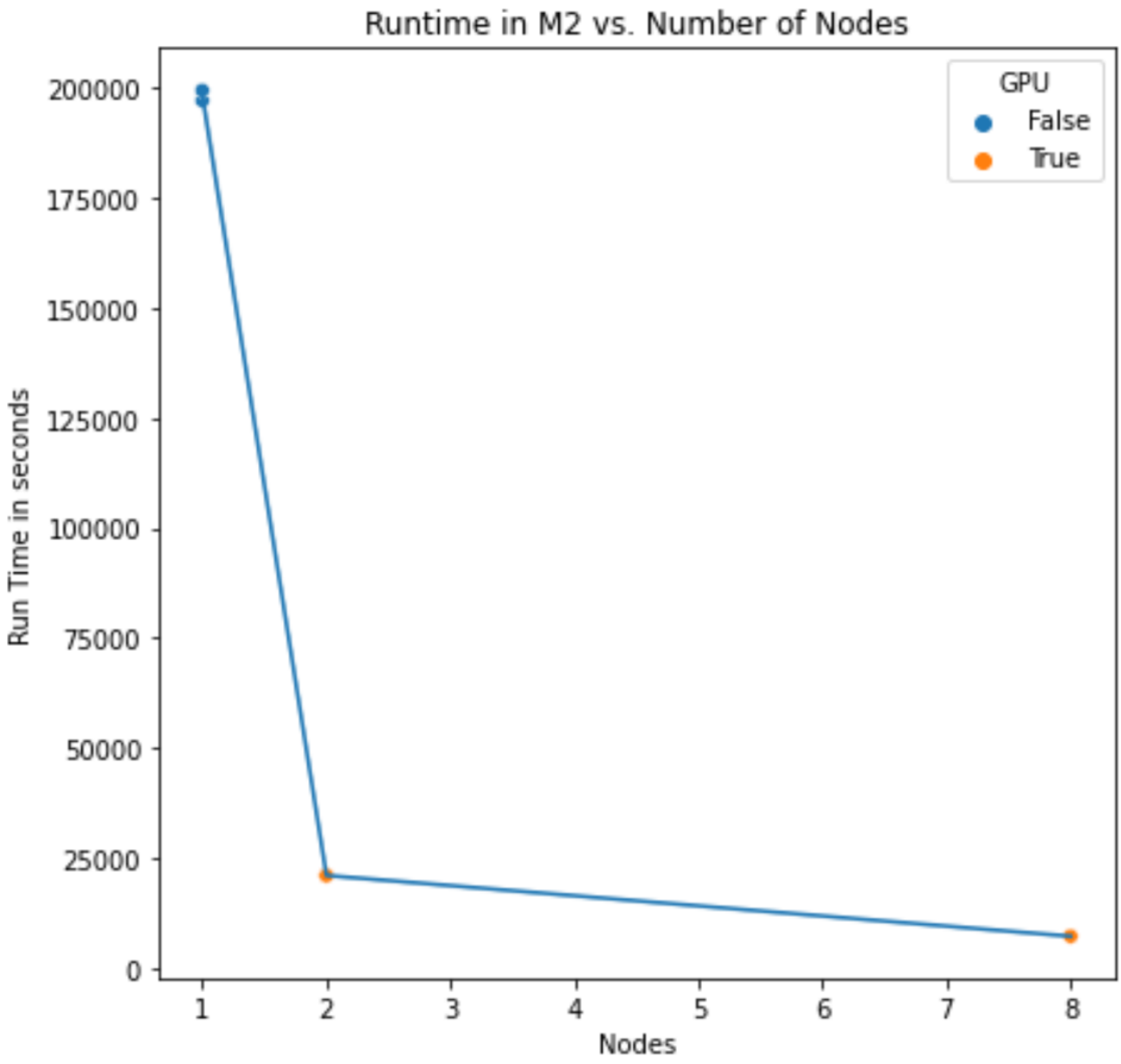
Experiment 3:

Run Without Pre-Serialization

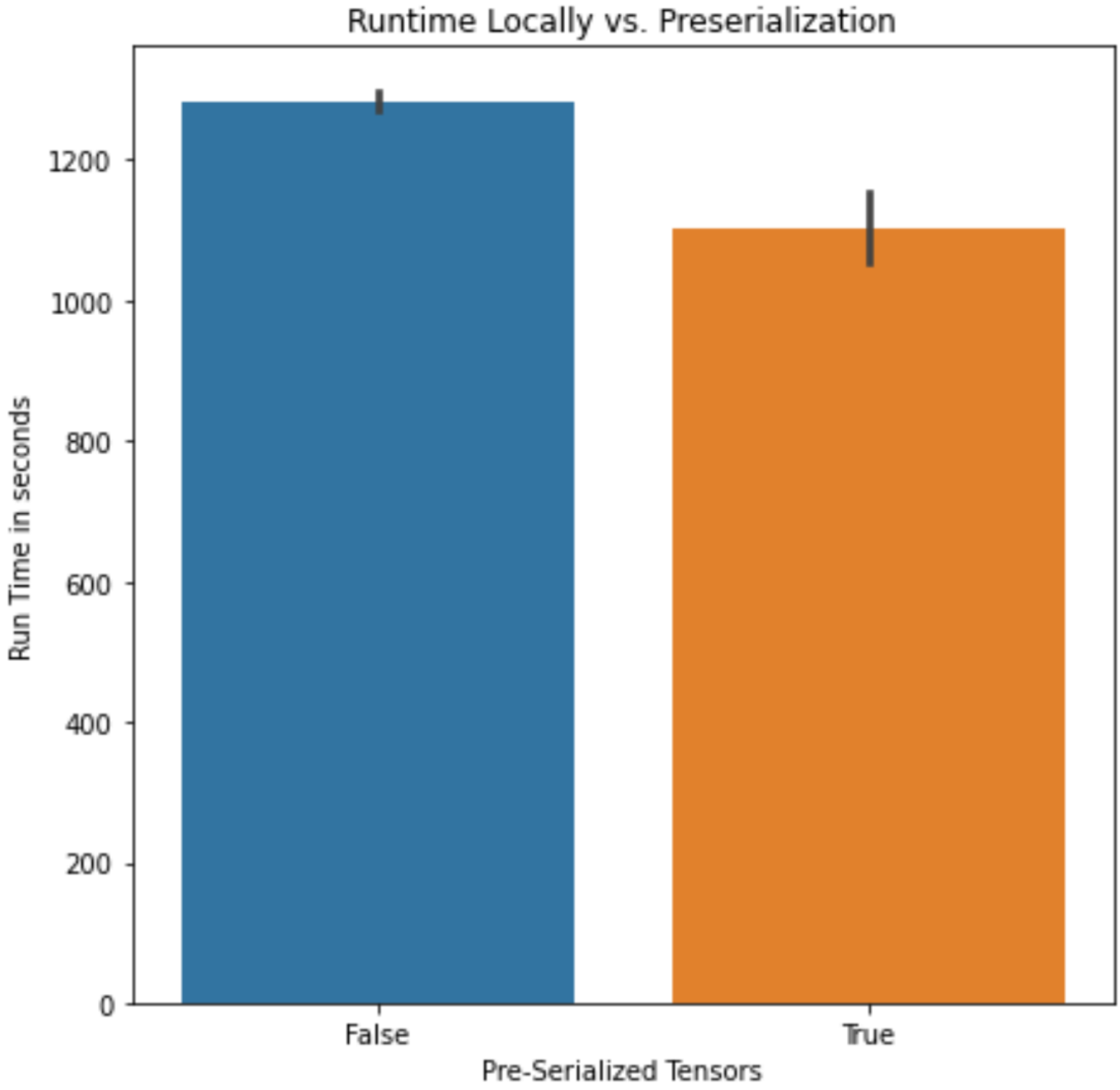
- Run 1: completed training at time --- 1153.6431467533112 seconds ---
- Run 2: completed training at time --- 1053.66729369484 seconds --- MEAN: 1103.6552202240755 seconds

Run WITH Pre-Serialization

- Run 1: completed training at time --- 1271.0400450229645 seconds ---
- Run 2: completed training at time --- 1296.9230398457849 seconds --- MEAN: 1283.9815424343747 seconds



For experiment 1 and 2: We can see that the drop in runtime is much more pronounced when comparing Using a GPU vs NOT using a GPU and increasing the number of nodes from 1 to 2, than it was using gpus but increasing from 2 to 8 nodes. This likely means that adding a GPU has a strong, impact, where adding more nodes has a lesser impact on training run time.



For experiment 3, we can see that there is a small performance increase to be had from pre-serializing image tensors. This affect is less than increasing nodes or using a GPU.

Steps to Reproduce Each Experiment:

Repro Experiments 1 and 2

1. Download the dataset into M2 using a Virtual Desktop
2. cloine this project repo
3. create a virtual python environment and activate it with `python3 -m venv /path/to/new/virtual/environment && source`

`/path/to/new/virtual/environment/bin/activate`

4. run `pip install -r Requirements.txt`
5. run sbatch file with different configurations above by running `sbatch /src/project_lightning.sbatch`
6. use a SFTP client to download the .err and .out files that are generated from these runs. Note the time at the bottom of the .out files and the run configuration in the .err file.

   Repro Experiment 3

7. Download the dataset locally
8. cloining this project repo
9. create a virtual python environment and activate it with `python3 -m venv /path/to/new/virtual/environment && source /path/to/new/virtual/environment/bin/activate`
10. run `pip install -r Requirements.txt`
11. Run the `python serialize_tensor.py` module to serialize the tensors
12. modify the `retina_dataset.py` to use the first 400 or so records in the `trainLabels_cropped.csv`.
13. Run `python project_lightning.py --num_nodes=1 --num_devices=0` to record a run WITH preserialization.
14. Note the time to train in the terminal log
15. modify the `retina_dataset.py` file's `__getitem__` function to load from the NON-transformed tensors.
16. 7. Run `python project_lightning.py --num_nodes=1 --num_devices=0` to record a run without preserialization
17. Note the time to train in the terminal log

Takeaways

Relatively easy to port pytorch models to pytorch lightning. For computer vision problems like inceptionv3 and diabetic retinopathy detection, GPU usage is most important, followed by node parallelization, and lastly followed by pre-serialization of input data.

Difficulties

printing synchronously and gathering logs in multinode environment. You need to use `file = sys.stdout, flush=True` in a print statement.

pytorch lightning trainer spinning up 8 different training runs instead of using 8 nodes. This was fixed by setting the `ntasks-per-node=1` in the sbatch file which sets an environment variable for

SLURM_NTASKS. This wasn't well documented.