Homework 1

Instructions

First, ensure you have cloned the course repository.

Then, open the interactive notebook version of this homework from your local copy.

For part A, fill in the code and answers within the notebook and save your changes.

For part B, create and archive the necessary Python/shell scripts together.

Finally, upload the notebook and the archive to the assignment in ILIAS.

Part A (12 points)

Note

When not otherwise specified, use the following parameter values in experiment runs:

- `nr_clients` (N): 100
- `lr` : 0.01
- `client_fraction` (C): 0.1
- `nr_local_epochs` (E): 1
- `batch_size` (B): 100
- `nr_rounds` : 10
- `iid` : True

For all exercises, pass `seed = 10` to calls for splitting data, server initialization, or plotting.

In [ ]:
```
import pandas as pd
import seaborn as sns
from tutorial_1a.hfl_complete import *

n = 100
lr = 0.01
c = 0.1
e = 1
b = 100
nr_rounds = 10
iid = True
seed = 10
```

Exercise A1: FedSGD with weights (3 points)

Question

(2 points) Implement a version of FedSGD that uses weights in its updates, like FedAvg, instead of the gradients from the version of the tutorials. The two FedSGD versions should have the same test accuracy after each round (with a tolerance of at most 0.02%). To show this, compare their output for the following two scenarios over *5 rounds*:

- `lr = 0.01, client_subsets = split(100, True, ...), client_fraction = 0.5`
- `lr = 0.1, client_subsets = split(50, False, ...), client_fraction = 0.2`

Tip: You can use the existing FedAvg implementation to minimize the amount of code writing required.

(1 point) Explain in which cases (about the different parameters for decentralized learning) the two are equivalent.

Answer

In [ ]:
```
# TODO
```

TODO

Exercise A2: Client number & fraction (4 points)

Question

(2 points) Run the necessary experiments to fill in the following table showing the final message count and test accuracy of FedSGD and FedAvg for different total client numbers:

Algorithm	N	C	Message count	Test accuracy
FedSGD	10	0.1		
FedAvg	10	0.1		
FedSGD	50	0.1		
FedAvg	50	0.1		
FedSGD	100	0.1		
FedAvg	100	0.1		

Is the relationship between the metrics and client numbers monotonous?

(2 points) Run the experiments to fill in the table when varying the fraction of clients used in every round:

Algorithm	N	C	Message count	Test accuracy

Algorithm	N	C	Message count	Test accuracy
FedSGD	100	0.01		
FedAvg	100	0.01		
FedSGD	100	0.1		
FedAvg	100	0.1		
FedSGD	100	0.2		
FedAvg	100	0.2		

How does the observed pattern differ?

Answer

In [ ]:
```
# TODO
```

TODO

In [ ]:
```
# TODO
```

TODO

Exercise A3: Local epoch count & (non-)IID data (5 points)

Question

(1 point) Create a line plot of the accuracy after each round for the following algorithm variants:

- FedSGD
- FedAvg (E=1)
- FedAvg (E=2)
- FedAvg (E=4)

How does FedAvg compare to FedSGD? What is the effect of increasing the work clients perform locally for each update in FedAvg?

(2 points) Make one line plot of FedSGD and FedAvg under an IID and non-IID split for 15 rounds (leaving all other parameter values as they previously mentioned default). How does the non-IID setting affect the accuracy achieved by the two algorithms? What is the difference in terms of the smoothness of learning?

(2 points) Make another plot for only non-IID splits, including the FedSGD and FedAvg configs from before, and add a version for each with a learning rate of 0.001 and client fraction of 0.5. How does the stability of the new variants compare to the old ones? Why do the changes in parameters have the observed effect?

Answer

In [ ]:
```
# TODO
```

TODO

In [ ]:
```
# TODO
```

TODO

In [ ]:
```
# TODO
```

TODO

Part B (12 points)

Exercise B1: Microbatch Pipeline Model Parallelism (7 points)

Implement pipeline parallelism with microbatches, as discussed during the lab.

As with the other data/model parallelism examples, you will need a Python script for the nodes and a shell script to orchestrate execution.

Be aware of the possibility of deadlocks: due to how `good` operates, it is possible to deadlock by having device 1 send B_2 to device 2 in the forward pass, and simultaneously, device 2 send B_1 in the backward pass. Since both operations will await a corresponding receive the training will stop indefinitely.

Use `isend` & `irecv`, the asynchronous (non-blocking) versions of `send` & `recv` in `torch.distributed`. Add comments or text explaining how you expect your implementation to work and test that it runs for the same number of steps and model architecture as in class.

Note that `torch.distributed`'s implementation of `gloo` does not currently support properly asynchronous communication even when using the corresponding primitives. Thus, you will not see the same improvements in speed as with a backend like `nccl`.

You may also take advantage of the fact that `torch` gradients naturally accumulate if zeroed out. Also, scaling the loss by a constant is equivalent to scaling the resulting gradients by the same constant.

Exercise B2: Joint Data & Model Parallelism (5 points)

Implement a training setup that uses data and model parallelism together.

Create 2 pipelines of 3 stages running sequentially, where each stage works with 3 sequential micro-batches.

Once again, add comments or text explaining your implementation and test it on the

setting that mimics those from the class.

You can use groups from `torch.distributed` to handle operations that require interaction between a subset of more than two but less than all workers.