ELIX

Federated Learning

Handbook v1.0

November 13, 2020

ELIX

Handbook v1.0

November 13, 2020

Federated Learning

In this joined project between Elix Inc. and Kyoto University, we implement a small communication module called "Mila" for federated learning, following the implementations of NVIDIA Clara (as close as possible). To validate the results, we built 3 models; a plain Graph Convolutional Model and a Graph Isomorphism model trained on Tox21 (multi-label multi-task classification), and a multi-modal network applied on a subset of the ChEMBL dataset, which combines molecule (ligand) and sequence (protein) data.

Contents

Theoretical Background

1.1 Federated Learning

Federated learning is a machine learning paradigm focused on distributed training across decentralized machines. As opposed to classic approaches where all the data is in one central location, during federated learning, the data is spread across multiple servers or edge devices. Each device (hereafter called "client") runs training for a number of epochs and sends the weights/checkpoints to a central location (hereafter called "server") which aggregates the information from all clients. The new aggregate model would then be sent to all participating clients, which then repeat the process using the latest model.

This way, each client's data remains private, and is not shared with any of the other clients, nor the server. For companies where data is a sensitive matter (as is the case for big pharmaceutical companies), this is a very attractive approach.

Additional details can be found in the original paper: Communication-Efficient Learning of Deep Networks from Decentralized Data (arxiv: 1602.05629)

1.2 Models

We validate the efficiency of federated learning approaches using 3 models:

- *Graph Convolutional Model* - Semi-supervised Classification with Graph Convolutional Networks
- *Graph Isomorphism Model* - How Powerful are Graph Neural Networks?
- *a Multi-Modal Network* - which mainly uses linear layers

1.2.1 Graph Convolutional Model

Our implementation follows the paper very closely, with 2 graph convolutional layers. For the implementation of the graph convolutions, we used the Pytorch Geometric library, which has very efficient modules for sparse operations (in PyTorch).

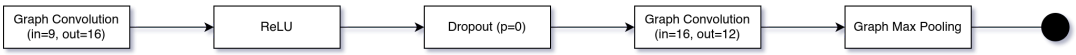

Figure 1.1: Architecture of the Graph Convolutional Network.

1.2.2 Graph Isomorphism Model

Similar to the Graph Convolutional Model, we used Pytorch Geometric for the implementation of the GIN Convolution. The model chains 5 GIN convolutions with 2 final fully connected layers.

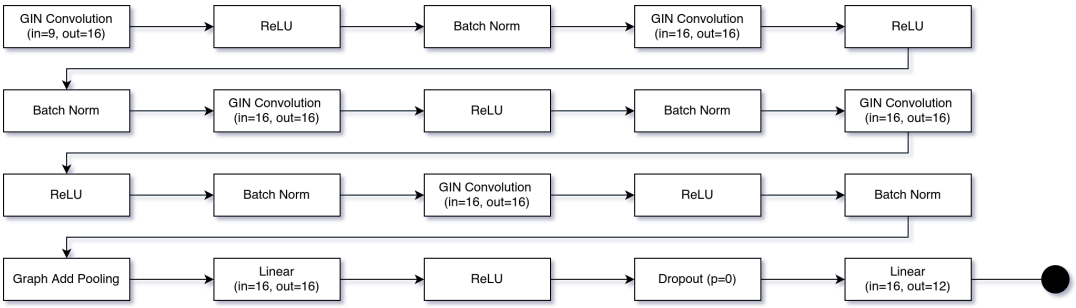

Figure 1.2: Architecture of the Graph Isomorphism Network.

1.2.3 Multi-Modal Network

The multi-modal network contains a protein and a ligand head. For the ligand head, the expected input is a fingerprint of 2048 bits, although the size is adjustable.

The protein head receives as input a one-hot encoded value of the protein ID. In the original requirements, 1-dimensional convolutions are suggested for handling the sequences, however, these don't work well because of the small sample size (4 unique sequences). As the focus of the project is to verify the usefulness of federated learning, we went with an approach which seemed to work decently, namely, the one-hot encoded values.

Both modules pass through a series of fully-connected layers; the outputs are then concatenated and passed through another set of fully-connected layers.

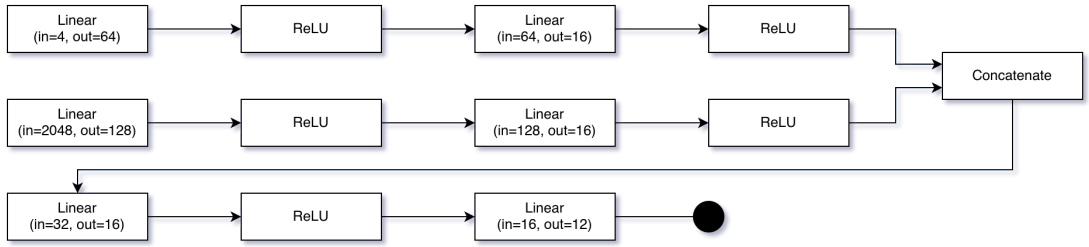

Figure 1.3: Architecture of the Multi-Modal Network.

1.3 Datasets

We make use of 2 datasets in our experiments. For the Graph Convolutional Model and the Graph Isomorphism Model, we use the Tox21 dataset; for the multi-modal network we use a subset of the ChEMBL dataset.

For all experiments, we used an 80/20 train/test split.

Tox21

The Tox21 dataset contains 7831 entries for 12 different toxicological experiments with binary labels which represent activity (active/inactive). It should be noted that the dataset contains missing values, which we ignore in the loss function and during back-propagation. Also, the dataset is very imbalanced; the number of positive samples ranging from 186 to 942, depending on the target.

ChEMBL Subset

Similar to the Tox21, the ChEMBL dataset contains binarity activity values. As features, we have ligands, represented as SMILES strings and proteins, represented as amino-acid sequences. The dataset contains 15250 equally distributed entries (7625 positive and 7625 negative samples), with 4 unique protein sequences and 5095 unique ligand SMILES.

1.4 Communication (Mila)

For the communication part of federated learning, we were planning to use NVIDIA's Clara, but it turned out that Clara only works with Tensorflow models (version 1.14). We, as well as Kyoto University would prefer working with PyTorch or at least Tensorflow 2.x, so we decided to rebuild the communication module, following in the steps of the NVIDIA Clara implementation.

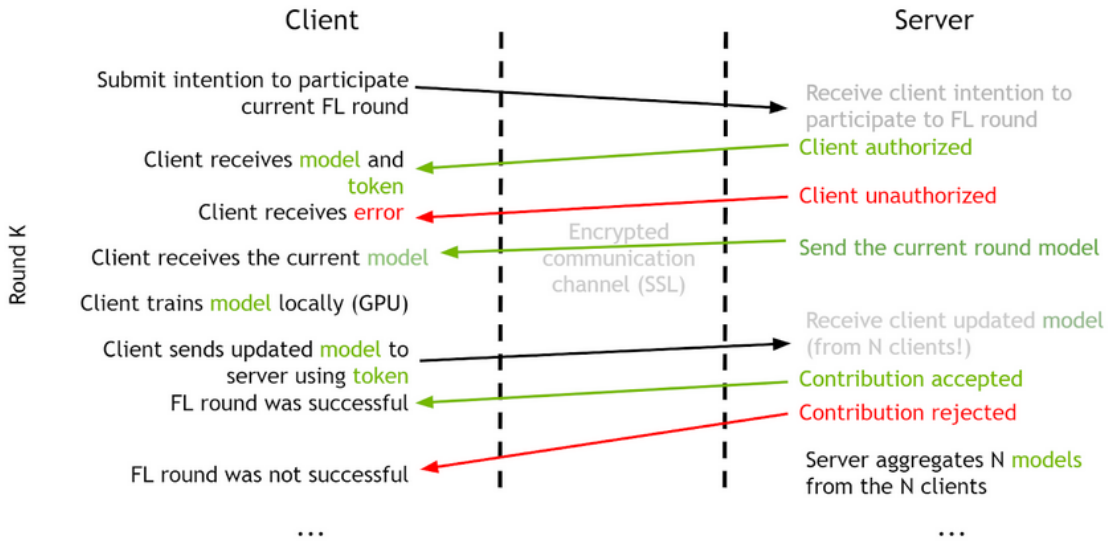

Figure 1.4: Federated Learning workflow followed by NVIDIA Clara and modeled by us (source: NVIDIA Blog)

We named the module "Mila", the greek word for speech. Similar to NVIDIA Clara, we use gRPC and protocol buffers to transmit information between the client and the servers.

Once the server is online, we wait for a number of clients to join before the training starts. The server then sends the latest model to each participant, which then train the model on their local data for a fixed number of epochs. After training finished, the latest checkpoint is sent back to the server, which will wait for all checkpoints before they are averaged. In our experiments we perform a plain averaging operation, but as we will show, other strategies could be more suitable. The new aggregate model is then used for the next round, and the process is repeated.

Each client sends a keepalive signal every minute, and in case one client goes offline or we stop receiving the heartbeat signal, the server halts the process.

1.4.1 Security

Communication is done through SSL/TLS with both server and client certificate chains. For development purposes, insecure connections are also allowed (no certs required).

When a client authenticates, it receives a token, which will subsequently be used for all operations. Machines not authenticates (ie: without a valid token), or dead clients (with an expired keeplive signal) will not be able to join the training process.

Furthermore, the server has the option to whitelist only a few trusted IP addresses which can join the process; blacklisting suspicious IPs is also possible.

1.4.2 Differences from NVIDIA Clara

The main advantage of our implementation is that it is framework independent. All models and checkpoints are transmitted as bytes and stored the same way they are on the server. NVIDIA Clara transports directly transports (Tensorflow) Tensor objects, and so, only supports Tensorflow models and aggregators.

Another key difference is that our implementation does not require GPUs and can run with all participants (clients and servers) using CPU only, should they wish to do so.

In addition, we added support for IP whitelisting and blacklisting.

On the negative side, our implementation is new and didn't go through extensive testing as (presumably) NVIDIA Clara did. The implementation of a custom client-server module was not planned for this phase and had time only for limited testing. While this is good enough to validate our ideas and evaluate the usefulness of federated learning, a solid test suite is a **MUST** before Mila can be used in production.

Experiment Results

2.1 Introduction

As the purpose of the project is to evaluate the usefulness of federated learning, we don't spend a lot of time on model improvements and hyper-parameter tuning.

We first take a look at the performance of each model trained in the classic way, on a single node, which we will use as baseline. We then run experiments using federated learning and analyze how the performance changes when we alter:

- the number of clients (from 1 to 20);
- the number of epochs trained per round (from 1 to 50);
- and data balances (from evenly distributed to highly skewed splits).

All experiments are evaluated on 3 metrics:

- the accuracy;
- the area under the receiver operating characteristic curve (AUROC, ROC-AUC);
- and the mean average precision (mAP).

2.2 Graph Convolutional Network

2.2.1 Baseline

Caution: The Graph Convolutional Network is trained on the Tox21 dataset; a highly unbalanced dataset. The accuracy metric therefore cannot be trusted.

We start with the Graph Convolutional Network. Our best model achieves an accuracy of 92.38%, a ROC-AUC of 69.93%, and an mAP of 21.25%; an average performance. It converges around epoch 200. The accuracy doesn't really change after the first few epochs, at least not for a balanced threshold of 0.5 which was used to get the predictions.

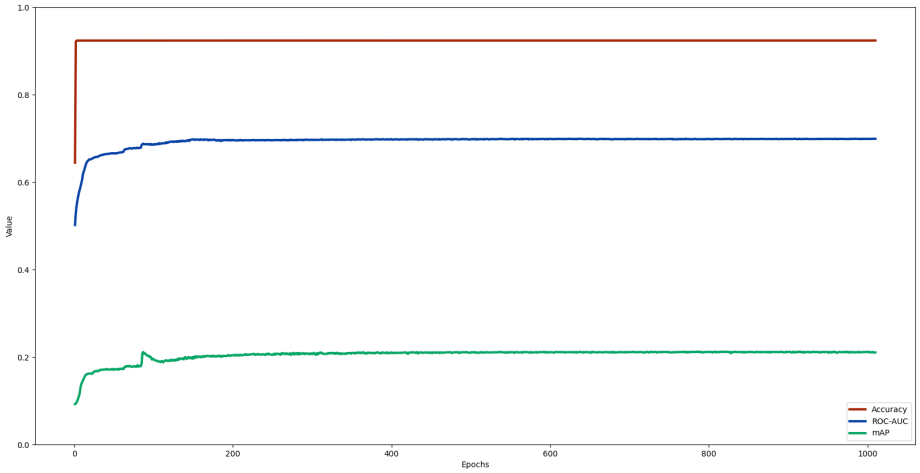

Figure 2.5: Baseline Graph Convolutional Network, trained independently on a single machine

As an additional validation, we trained the graph convolutional network using https://github.com/deepchem/torchchemTorchChem as well, which also uses Pytorch Geomteric, and we were able to get a ROC-AUC score of 0.78. The main difference between our implementation and theirs is that TorchChem uses a custom atom featurizer, instead of the Pytorch Geometric default one, which contains less features. If our goal changes and we ever wish to improve the performance of the model, this would be the area we should look at first.

2.2.2 Client Number Comparison

Note: Each round was trained for 10 epochs in these experiments. For a fair comparison, the baseline was adjusted to contain only every 10th epoch as well.

Note: All experiments were performed on evenly distributed data. The training set was split based on the number of clients, and each client received an even subset to train on.

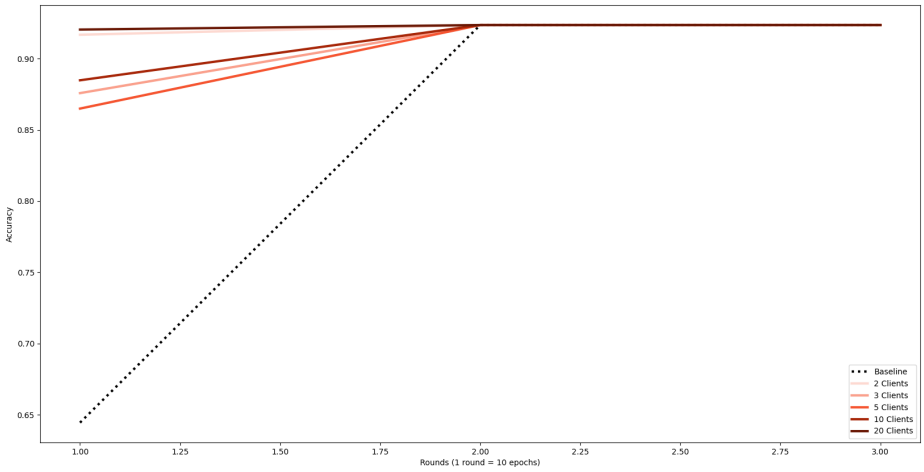

Figure 2.6: Graph Convolutional Network - Client Number Comparison - Accuracy

The accuracy metric is not very helpful for these experiments, but we include it for completeness. What really stands out is that it does not change after a few epochs.

For the ROC-AUC value, we can see that the number of clients matters. For 2 and 3 clients, the performance gets very close to the baseline, which is quite impressive, given that we train only on half or one third of the dataset at a time. If we go a bit too far, we see that 5, 10 and 20 clients (corresponding to splits of fifths, tenths, and twentieths) are affected to some extent, but even for 20 clients, the difference is just 4%.

The performance of the mAP score is very similar to the ROC-AUC score, although the effect seems to be of a larger proportion. Comparing the baseline to the 20 client experiment, the difference was close to 8%, which is 35% smaller.

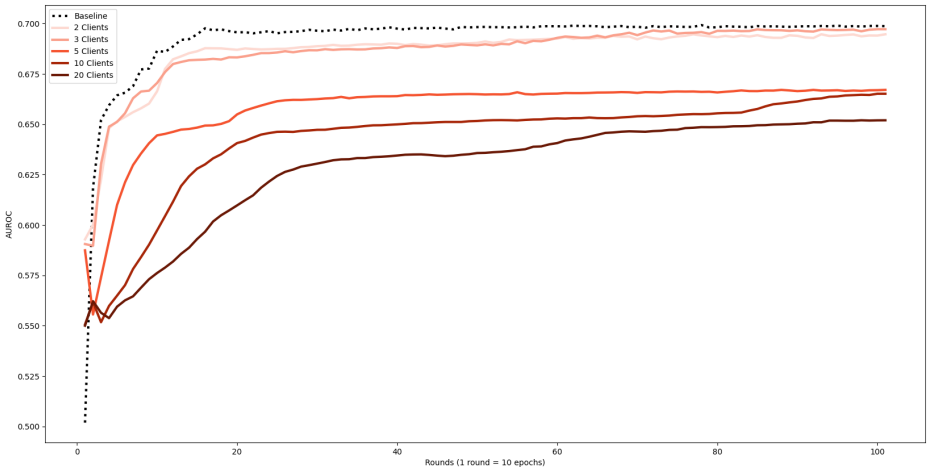

Figure 2.7: Graph Convolutional Network - Client Number Comparison - ROC-AUC

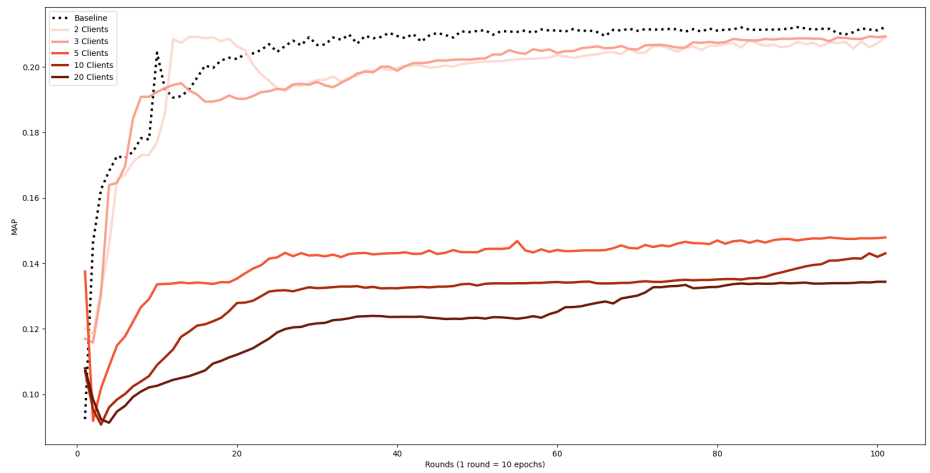

Figure 2.8: Graph Convolutional Network - Client Number Comparison - mAP

	Accuracy	ROC-AUC	mAP
Baseline	92.38%	69.93%	21.26%
2 Clients	92.37%	69.47%	20.92%
3 Clients	92.37%	69.72%	20.93%
5 Clients	92.36%	66.71%	14.79%
10 Clients	92.36%	66.52%	14.30%
20 Clients	92.36%	65.20%	13.44%

Table 2.2.1: Graph Convolutional Model - Client Number Comparison - Best Scores

2.2.3 Epoch Count Comparison

Note: Each experiment was performed using 2 clients with an even 50/50 data split.

In the previous section, clients trained for 10 epochs per round in experiments. Here, we take a look how the number of epochs affects the performance and inference time.

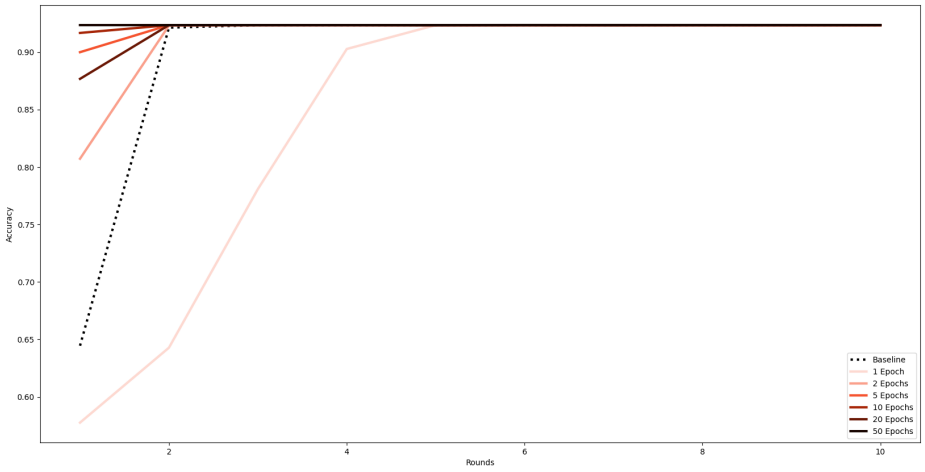

Figure 2.9: Graph Convolutional Network - Epoch Count Comparison - Accuracy

For the accuracy metric, we see that all experiments converge after 2 rounds, except when we aggregate the model epoch by epoch. In this case, it takes 5 rounds for the model to converge.

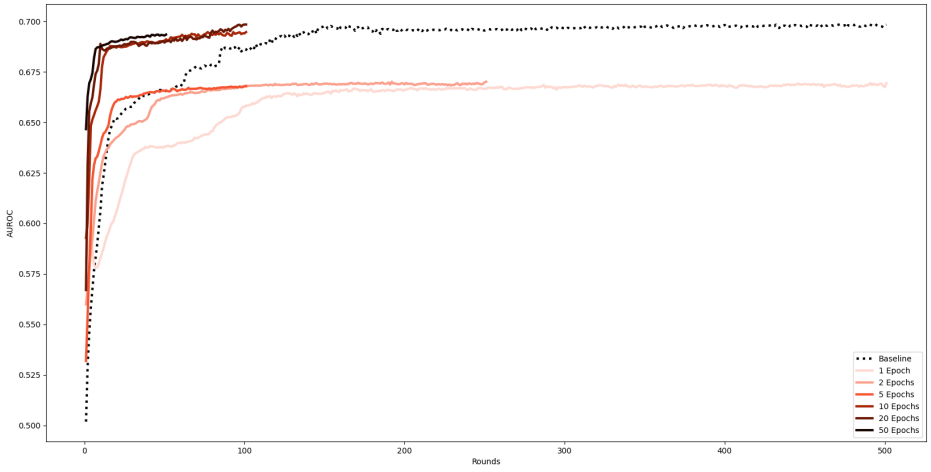

Figure 2.10: Graph Convolutional Network - Epoch Count Comparison - ROC-AUC

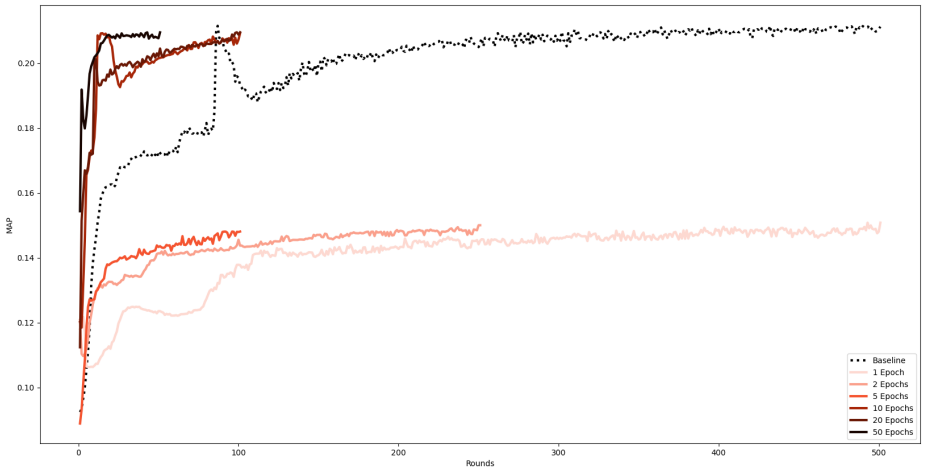

Figure 2.11: Graph Convolutional Network - Epoch Count Comparison - mAP

For the ROC-AUC and mAP scores, 10 epochs per round seems to be the best choice. If we use too few rounds (from 1 to 5), the model underperformed by up to 3% on the ROC-AUC metric and by up to 6% on the mAP metric. If we use too many rounds (20 to 50), we gain very little but clients spend more time training.

The aggregate model doesn't seem to overfit on a particular client's contribution when training for longer epochs.

	Accuracy	ROC-AUC	mAP
Baseline	92.38%	69.93%	21.26%
1 Epoch	92.36%	66.94%	15.08%
2 Epoch	92.36%	67.01%	15.00%
5 Epoch	92.36%	66.78%	14.81%
10 Epoch	92.37%	69.47%	20.92%
20 Epoch	92.37%	69.84%	20.95%
50 Epoch	92.37%	69.35%	20.95%

Table 2.2.2: Graph Convolutional Model - Epoch Count Comparison - Best Scores

2.2.4 Uneven Data Distributions

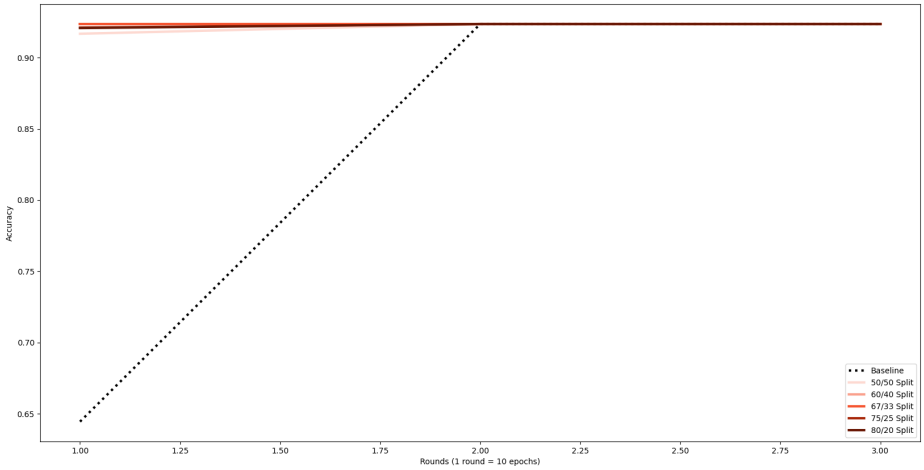

Figure 2.12: Graph Convolutional Network - 2 Unbalanced Clients - Accuracy

Note: Each round was trained for 10 epochs in these experiments. For a fair comparison, the baseline was adjusted to contain only every 10th epoch as well.

All experiments contained equal data splits, but what happens if the proportions are skewed? We take a look, starting with 2 client experiments.

There is not much to see when it comes to accuracy, and most of the experiments behave similarly for ROC-AUC and mAP as well. All except for the 66% - 33% split, which was 3% worse off in terms of ROC-AUC and 6% in terms of mAP.

The cause of this phenomenon can be debated. It is possible that even distributions contain large enough samples for all clients to train on, while heavily balanced splits contain most of the information in one split and make the smaller samples less relevant. But this effect could also be chaotic, and depend largely on where/how the data is split.

There are ways to combat these sort of anomalies which we discuss in the Aggregators section.

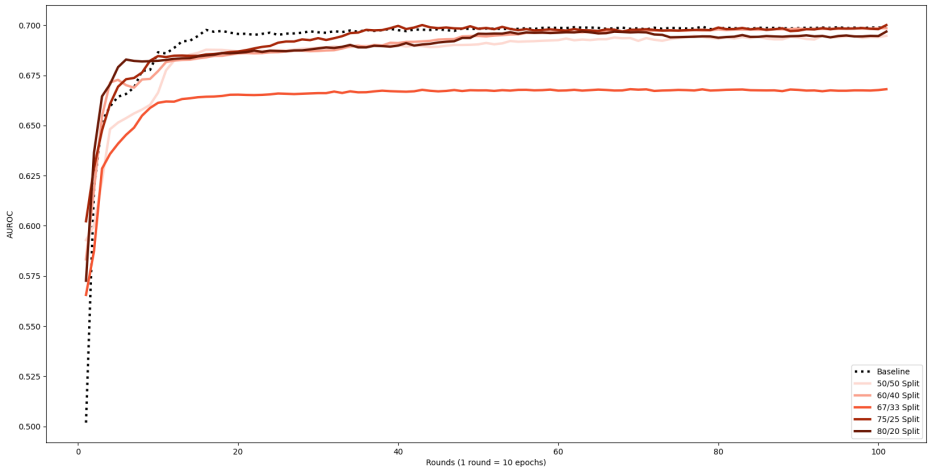

Figure 2.13: Graph Convolutional Network - 2 Unbalanced Clients - ROC-AUC

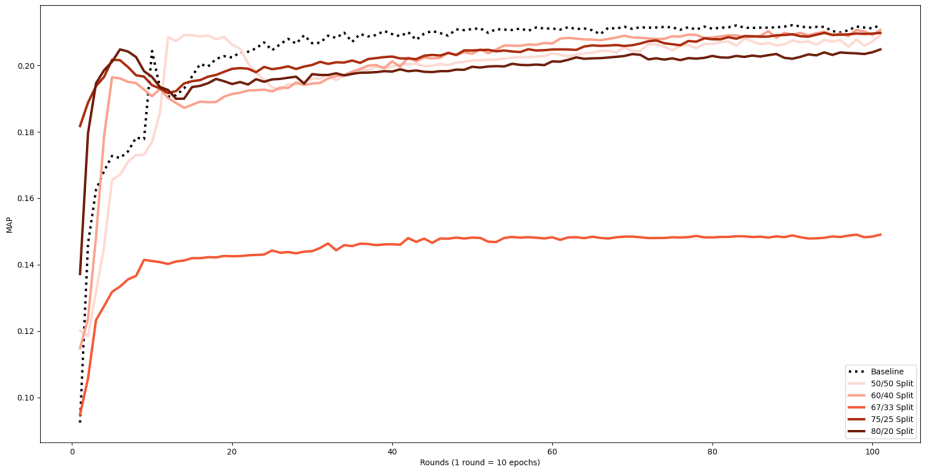

Figure 2.14: Graph Convolutional Network - 2 Unbalanced Clients - mAP

	Accuracy	ROC-AUC	mAP
Baseline	92.38%	69.93%	21.26%
50/50 Split	92.37%	69.47%	20.92%
60/40 Split	92.37%	69.85%	21.07%
67/33 Split	92.36%	66.81%	14.90%
75/25 Split	92.37%	70.01%	20.98%
80/20 Split	92.37%	69.68%	20.48%

Table 2.2.3: Graph Convolutional Model - 2 Unbalanced Clients - Best Scores

When looking at 3 client uneven splits, the difference seems to matter more, and highly imbalanced distributions also seem to be affected. In our experiments, 34/33/33 and 40/30/30 seem to achieve values close to the baseline, but 60/20/20, 80/10/10, and 60/30/10 splits score 3% less on ROC-AUC and, again, 6% lower on mAP.

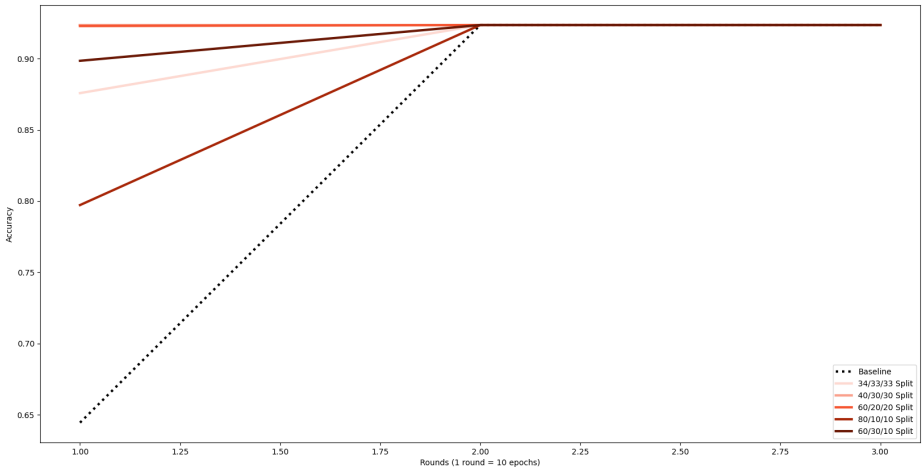

Figure 2.15: Graph Convolutional Network - 3 Unbalanced Clients - Accuracy

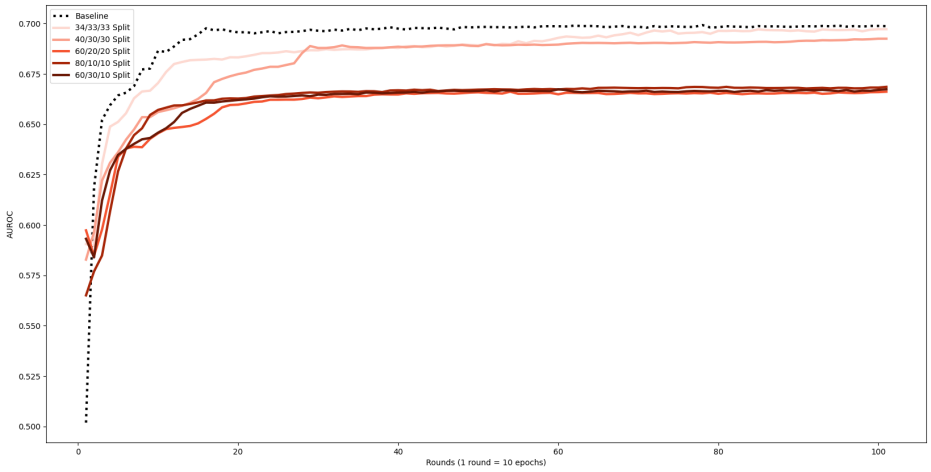

Figure 2.16: Graph Convolutional Network - 3 Unbalanced Clients - ROC-AUC

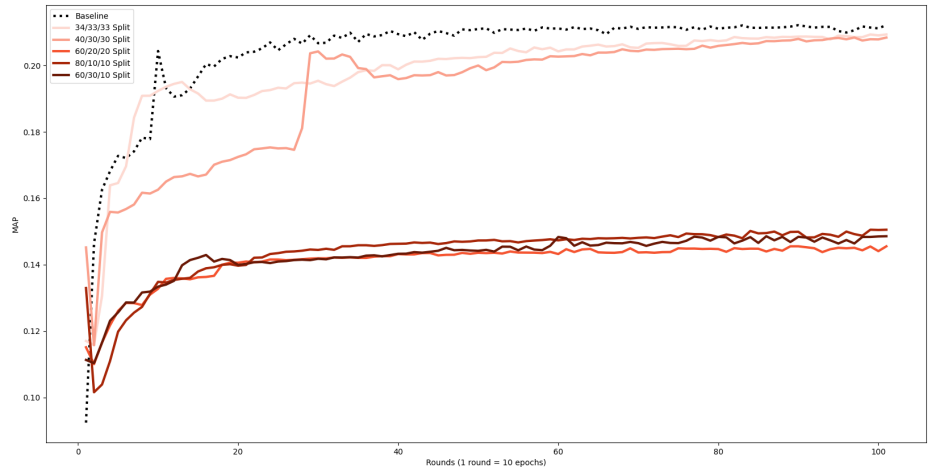

Figure 2.17: Graph Convolutional Network - 3 Unbalanced Clients - mAP

	Accuracy	ROC-AUC	mAP
Baseline	92.38%	69.93%	21.26%
34/33/33 Split	92.37%	69.72%	20.93%
40/30/30 Split	92.37%	69.25%	20.84%
60/20/20 Split	92.36%	66.62%	14.55%
80/10/10 Split	92.36%	66.86%	15.05%
60/30/10 Split	92.36%	66.74%	14.86%

Table 2.2.4: Graph Convolutional Model - 3 Unbalanced Clients - Best Scores

For 5 client splits, all models were weaker, including the balanced split (3% for ROC-AUC, 6% for mAP).

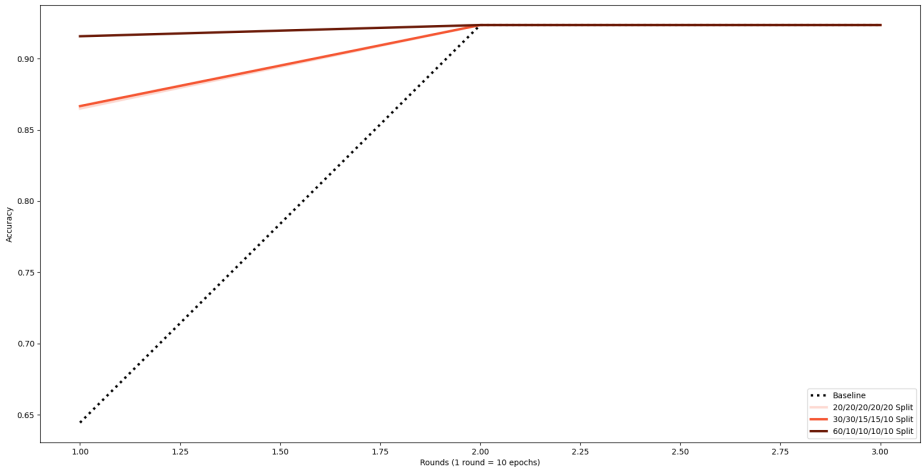

Figure 2.18: Graph Convolutional Network - 5 Unbalanced Clients - Accuracy

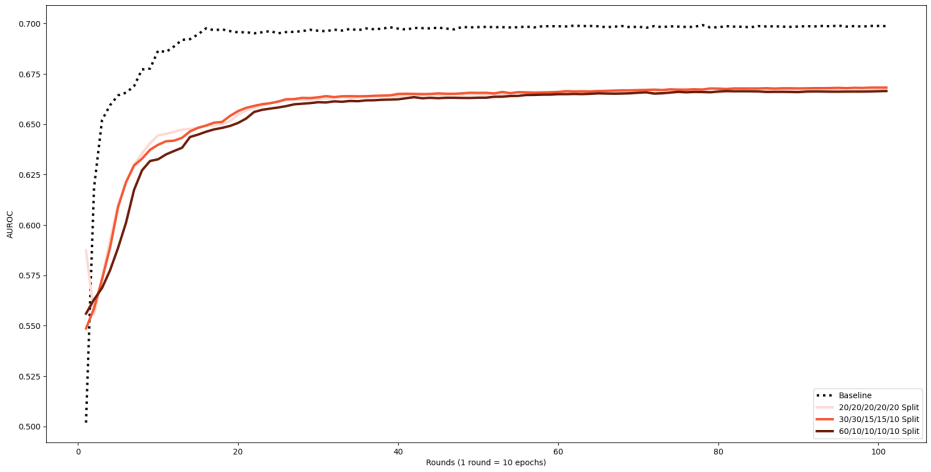

Figure 2.19: Graph Convolutional Network - 5 Unbalanced Clients - ROC-AUC

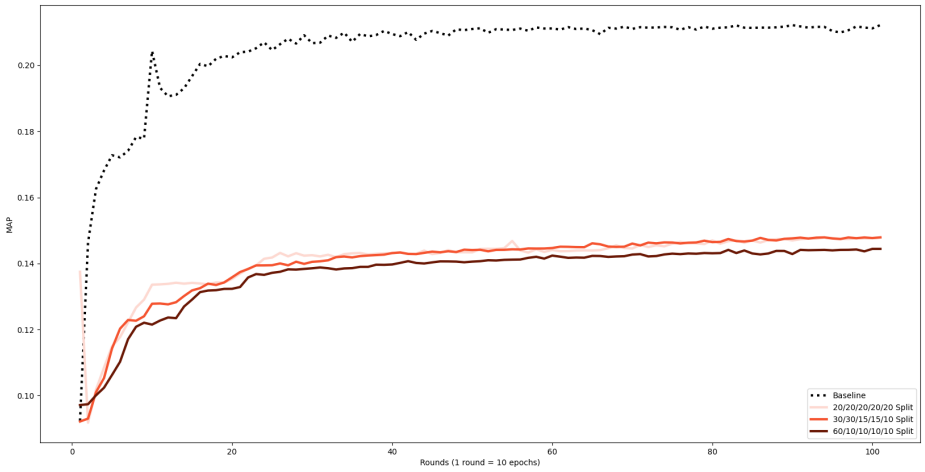

Figure 2.20: Graph Convolutional Network - 5 Unbalanced Clients - mAP

	Accuracy	ROC-AUC	mAP
Baseline	92.38%	69.93%	21.26%
20/20/20/20/20 Split	92.36%	66.71%	14.79%
30/30/15/15/10 Split	92.36%	66.82%	14.79%
60/10/10/10/10 Split	92.36%	66.65%	14.44%

Table 2.2.5: Graph Convolutional Model - 5 Unbalanced Clients - Best Scores

2.3 Graph Isomorphism Network

2.3.1 Baseline

Caution: The Graph Isomorphism Network is trained on the Tox21 dataset; a highly unbalanced dataset. The accuracy metric therefore cannot be trusted.

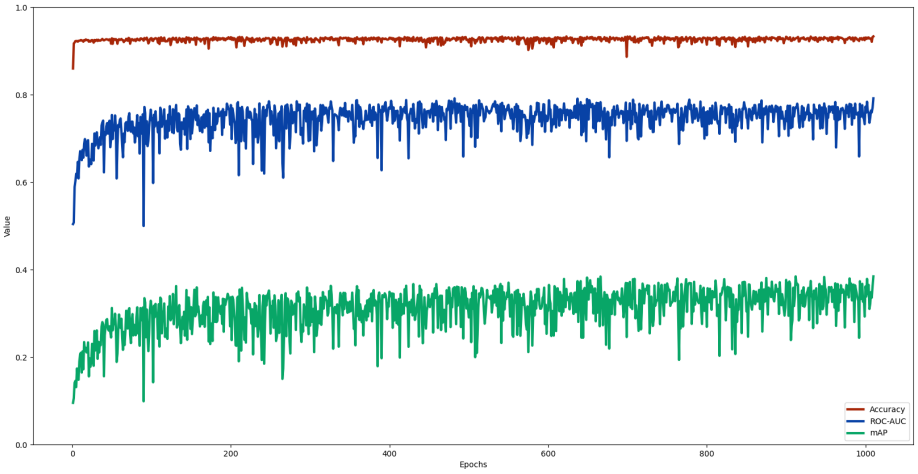

Figure 2.21: Baseline Graph Isomorphism Network, trained independently on a single machine

Our best model trained on GIN achieves an accuracy of 93.32%, a ROC-AUC of 79.16%, and an mAP of 38.44%; a better performance. It converges in roughly 100 to 150 epochs (although, we should note, it takes longer to train 1 epoch

of GIN than GCN). The accuracy is a bit more volatile compared to the Graph Convolutional Network, but it fiddles around a very small interval.

Similar to the Graph Convolutional Network, this model uses atom features computed by PyTorch Geometric, and performance would probably improve with a custom atom featurizer.

The Graph Isomorphism Network easily outperforms the base Graph Convolutional Network, although the scores jump around a lot more. This might be a sign that the Graph Convolutional Network is over-fitting.

2.3.2 Client Number Comparison

Note: Each round was trained for 10 epochs in these experiments. For a fair comparison, the baseline was adjusted to contain only every 10th epoch as well.

Note: All experiments were performed on evenly distributed data. The training set was split based on the number of clients, and each client received an even subset to train on.

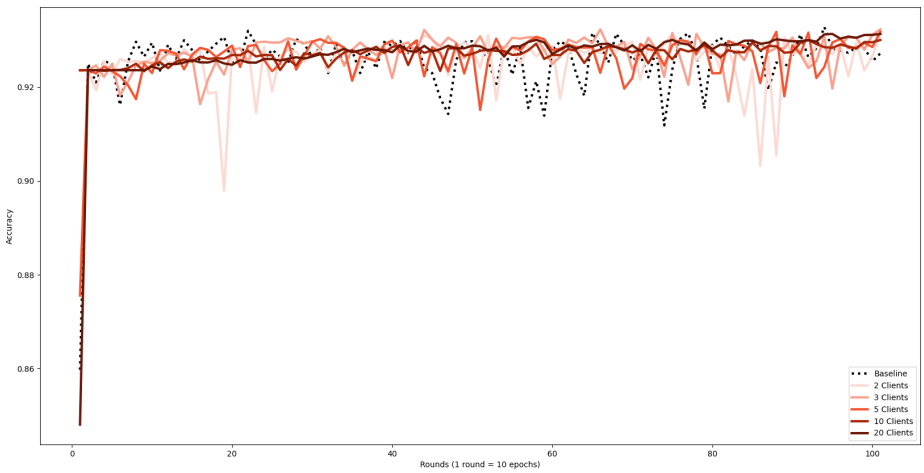

Figure 2.22: Graph Isomorphism Network - Client Number Comparison - Accuracy

Again, the accuracy metric should not be trusted very much, but we include it

for completeness. The chart looks quite noisy, but the message is actually quite clear; accuracy is not affected by the number of clients.

Even more so, the data distribution seems to have a regularizing effect, similar to a batch size.

For the ROC-AUC and the mAP scores, a very high number of clients seems to affect the score a bit (10-20 clients), but the effect is much smaller when compared to the Graph Convolutional Network. For 20 clients, the difference is similar (3% difference in ROC-AUC and 6-7% mAP), but for 10 clients the difference is smaller (2% ROC-AUC; 4% mAP). This effect also seems to be much smaller in terms of proportions, since the performance of the Graph Isomorphism Network is higher.

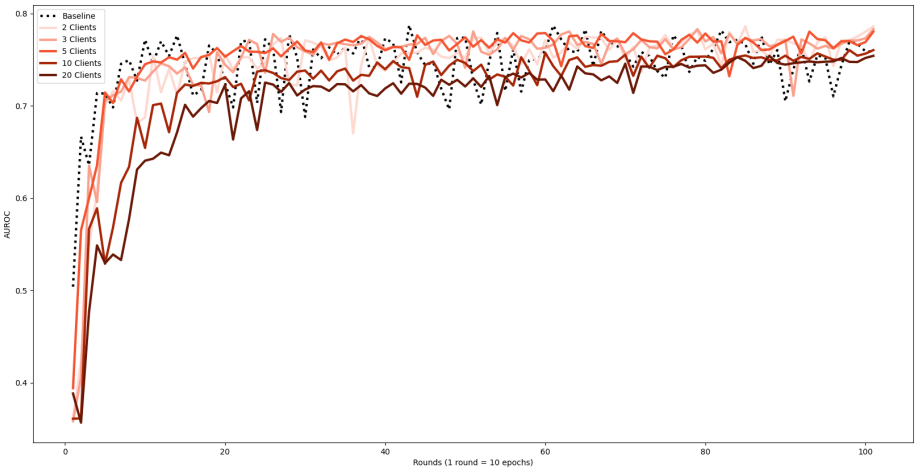

Figure 2.23: Graph Isomorphism Network - Client Number Comparison - ROC-AUC

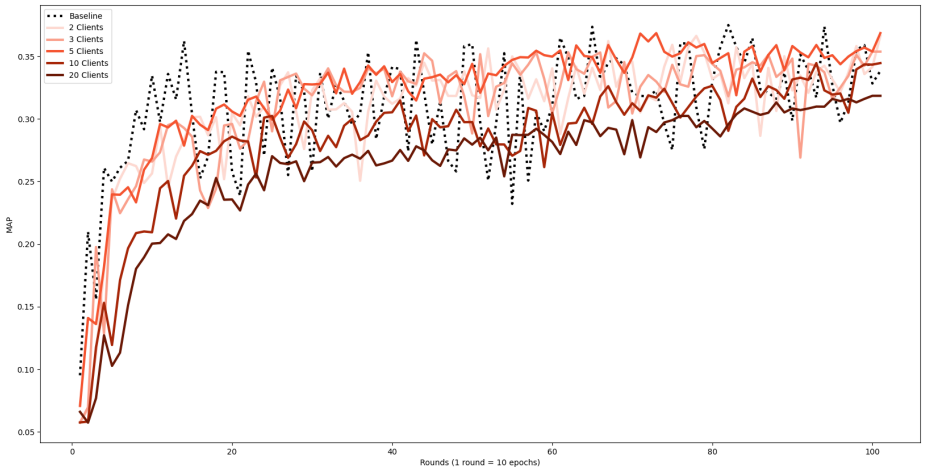

Figure 2.24: Graph Isomorphism Network - Client Number Comparison - mAP

	Accuracy	ROC-AUC	mAP
Baseline	93.32%	79.16%	38.44%
2 Clients	93.13%	78.62%	36.63%
3 Clients	93.23%	78.32%	35.37%
5 Clients	93.18%	78.05%	36.85%
10 Clients	93.00%	76.05%	34.46%
20 Clients	93.13%	75.43%	31.84%

Table 2.3.1: Graph Isomorphism Model - Client Number Comparison - Best Scores

2.3.3 Epoch Count Comparison

Note: Each experiment was performed using 2 clients with an even 50/50 data split.

The effect of the epoch count appears to be mostly unanimous across all metrics; the number of epochs, does not affect the performance much. When trained for 50 epochs, the maximum scores drop by 3% in ROC-AUC and 6% in mAP, but the overall distribution seems very close to the other experiments.

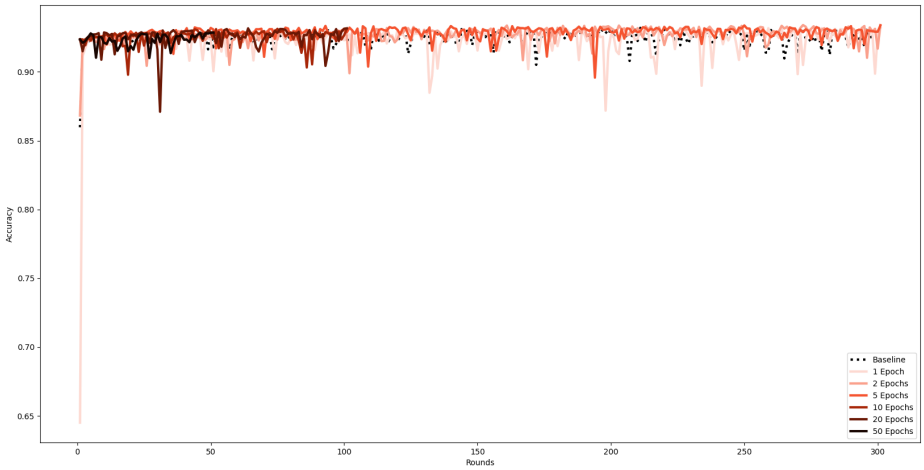

Figure 2.25: Graph Isomorphism Network - Epoch Count Comparison - Accuracy

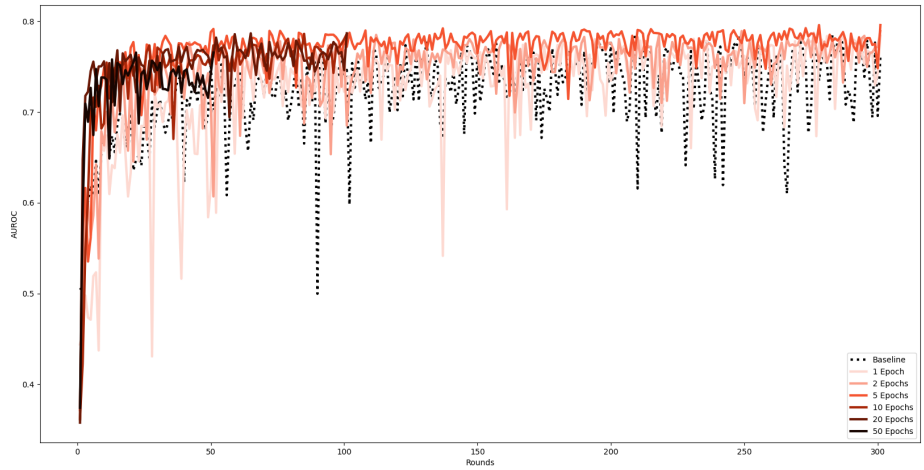

Figure 2.26: Graph Isomorphism Network - Epoch Count Comparison - ROC-AUC

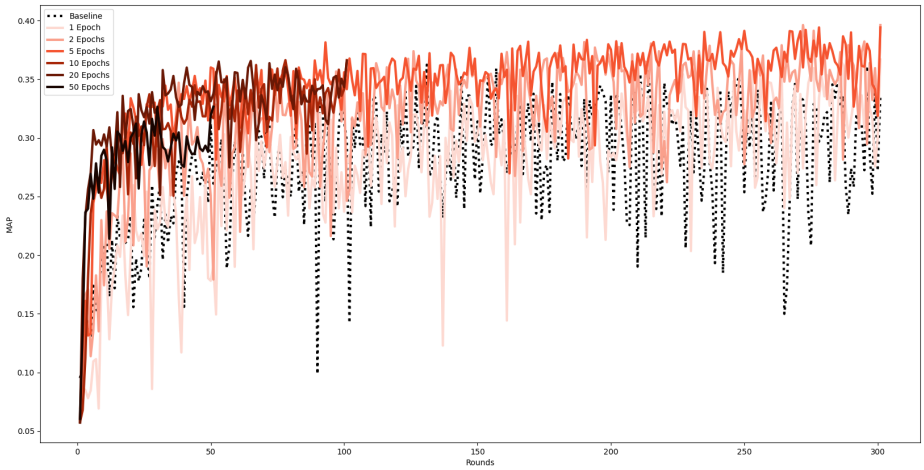

Figure 2.27: Graph Isomorphism Network - Epoch Count Comparison - mAP

But similar to the client count comparison; when trained for fewer epochs the plots are more noisy, and the epoch count also seems to have a regularizing effect.

	Accuracy	ROC-AUC	mAP
Baseline	93.32%	79.16%	38.44%
1 Epochs	93.19%	78.40%	36.68%
2 Epochs	93.40%	78.92%	39.62%
5 Epochs	93.38%	79.59%	39.41%
10 Epochs	93.13%	78.62%	36.63%
20 Epochs	93.17%	78.68%	36.56%
50 Epochs	92.85%	76.55%	32.65%

Table 2.3.2: Graph Isomorphism Model - Epoch Count Comparison - Best Scores

2.3.4 Uneven Data Distributions

Note: Each round was trained for 10 epochs in these experiments. For a fair comparison, the baseline was adjusted to contain only every 10th epoch as well.

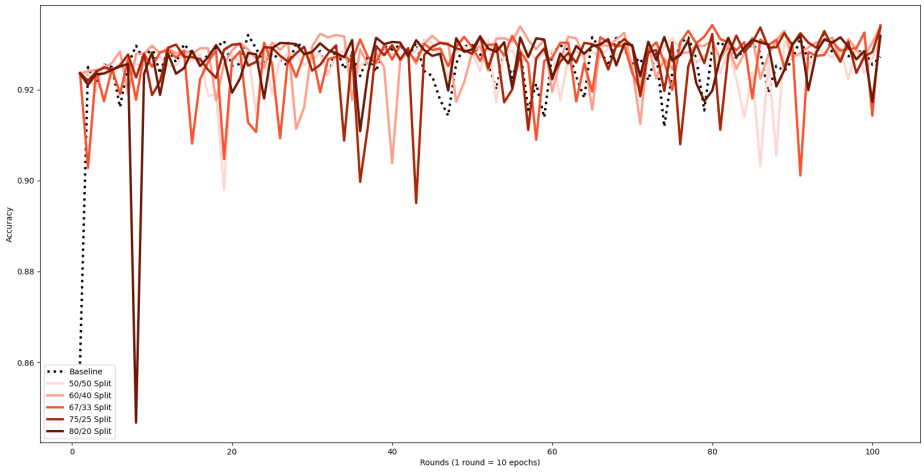

Figure 2.28: Graph Isomorphism Network - 2 Unbalanced Clients - Accuracy

Similar to the epoch count, the data distribution among clients also seems to be much less relevant for the Graph Isomorphism Network. The accuracy is unaffected, and we see a difference of up to 2% in ROC-AUC and up to 3% in mAP, however, some experiments also outperform the baseline (the 60/40 split). This applies to 2 client, 3 client, and 5 client experiments as well.

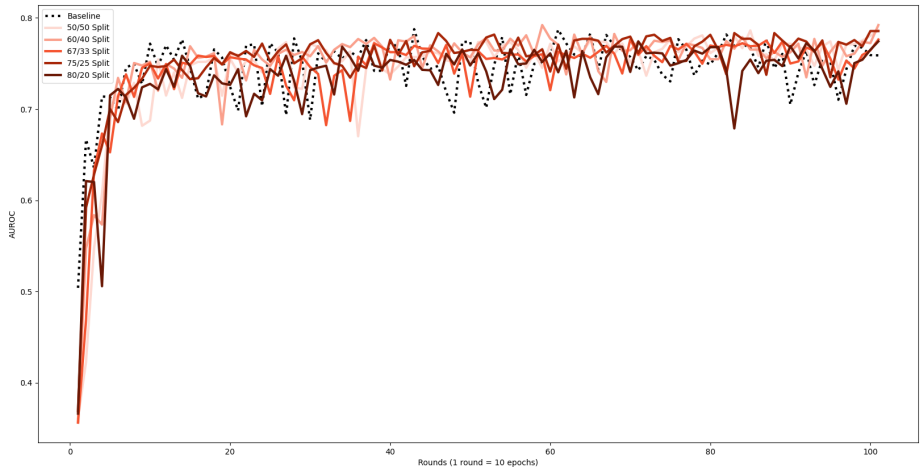

Figure 2.29: Graph Isomorphism Network - 2 Unbalanced Clients - ROC-AUC

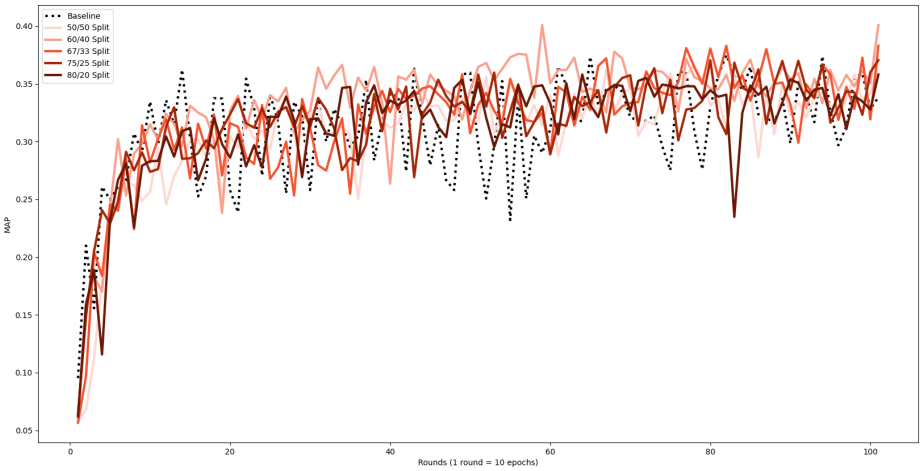

Figure 2.30: Graph Isomorphism Network - 2 Unbalanced Clients - mAP

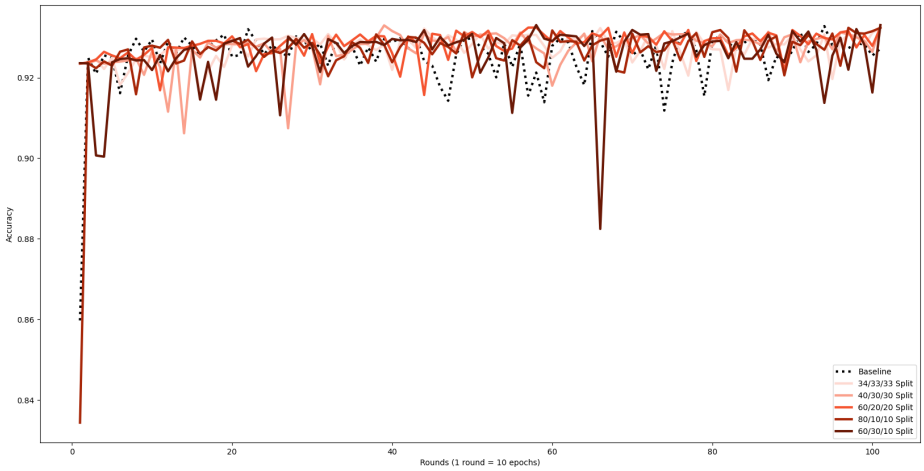

Figure 2.31: Graph Isomorphism Network - 3 Unbalanced Clients - Accuracy

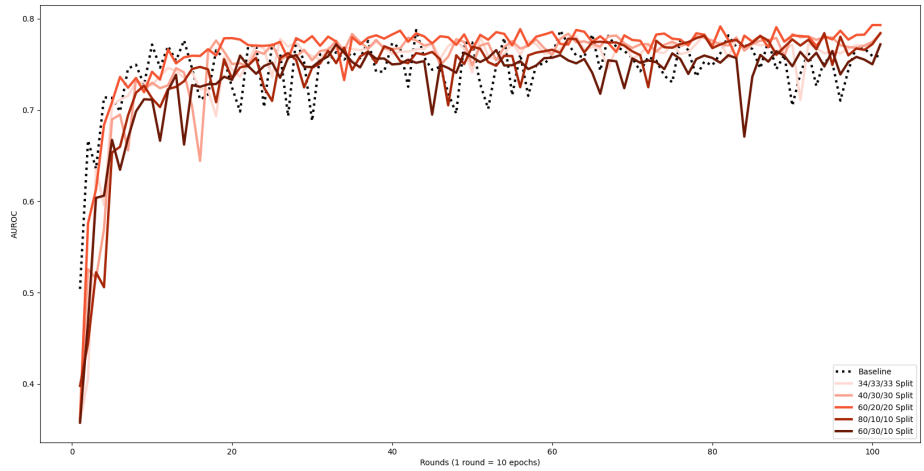

Figure 2.32: Graph Isomorphism Network - 3 Unbalanced Clients - ROC-AUC

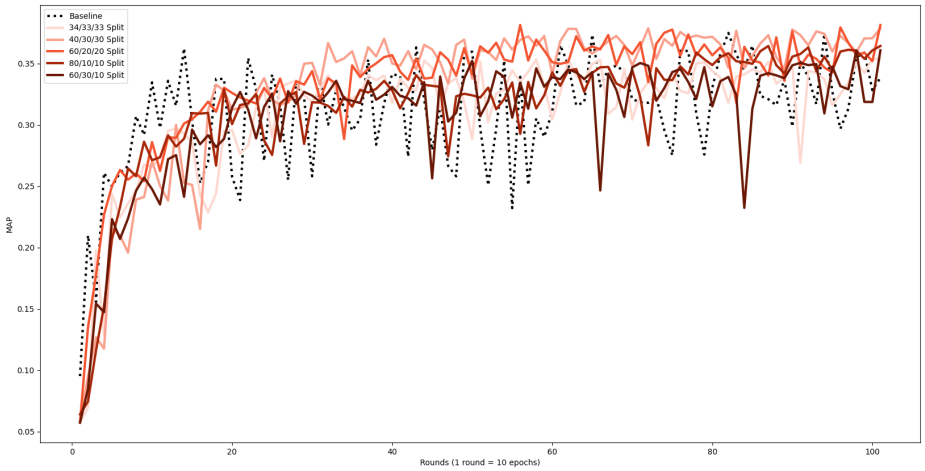

Figure 2.33: Graph Isomorphism Network - 3 Unbalanced Clients - mAP

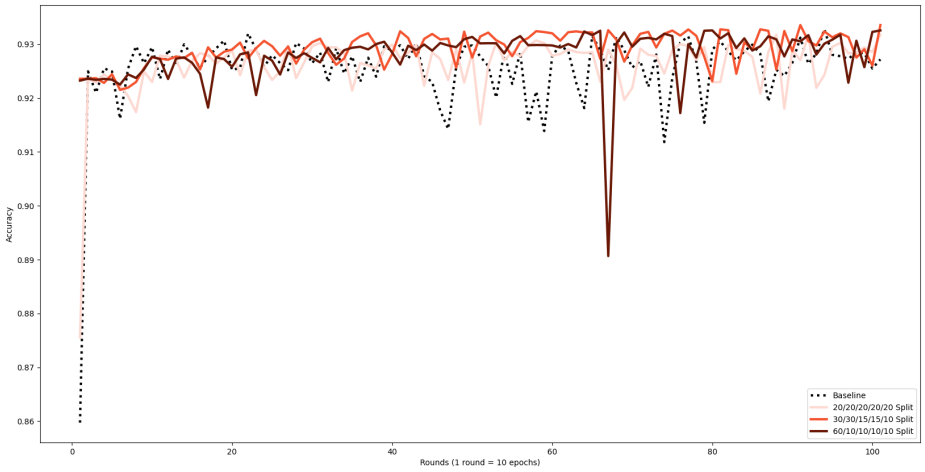

Figure 2.34: Graph Isomorphism Network - 5 Unbalanced Clients - Accuracy

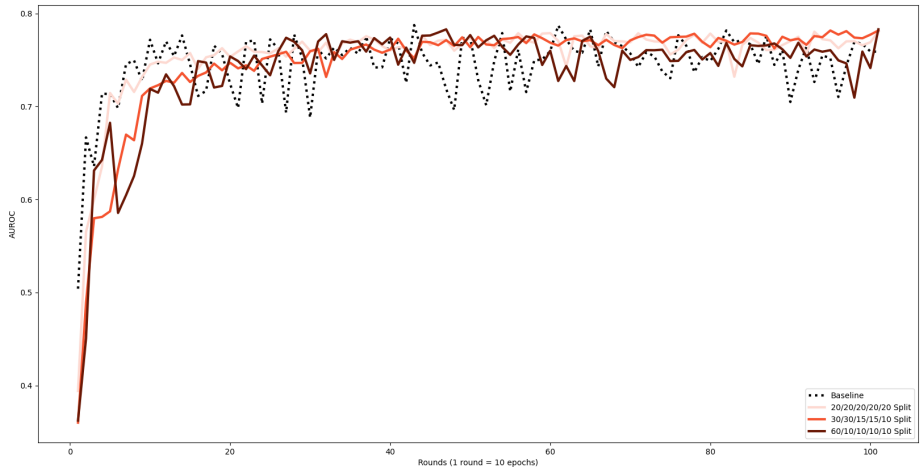

Figure 2.35: Graph Isomorphism Network - 5 Unbalanced Clients - ROC-AUC

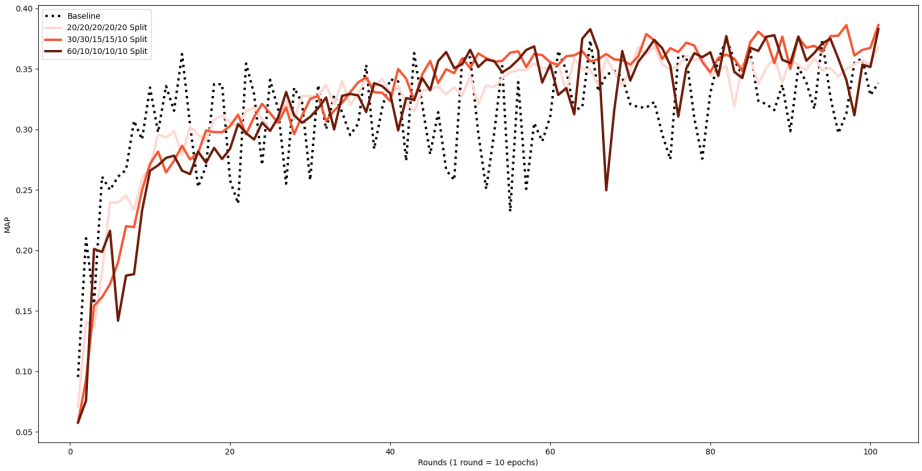

Figure 2.36: Graph Isomorphism Network - 5 Unbalanced Clients - mAP

	Accuracy	ROC-AUC	mAP
Baseline	93.32%	79.16%	38.44%
50/50 Split	93.13%	78.62%	36.63%
60/40 Split	93.39%	79.22%	40.09%
67/33 Split	93.42%	77.63%	38.30%
75/25 Split	93.37%	78.56%	37.04%
80/20 Split	93.18%	77.43%	35.81%

Table 2.3.3: Graph Isomorphism Model - 2 Unbalanced Clients - Best Scores

	Accuracy	ROC-AUC	mAP
Baseline	93.32%	79.16%	38.44%
34/33/33 Split	93.23%	78.32%	35.37%
40/30/30 Split	93.30%	78.37%	37.87%
60/20/20 Split	93.24%	79.31%	38.13%
80/10/10 Split	93.24%	78.43%	36.43%
60/30/10 Split	93.31%	77.19%	36.07%

Table 2.3.4: Graph Isomorphism Model - 3 Unbalanced Clients - Best Scores

	Accuracy	ROC-AUC	mAP
Baseline	93.32%	79.16%	38.44%
20/20/20/20/20 Split	93.18%	78.05%	36.85%
30/30/15/15/10 Split	93.36%	78.17%	38.63%
60/10/10/10/10 Split	93.26%	78.30%	38.29%

Table 2.3.5: Graph Isomorphism Model - 5 Unbalanced Clients - Best Scores

2.4 Multi-Modal Network

2.4.1 Baseline

The best scores achieved on the Multi-Modal Network are as follows: accuracy of 82.10%, a ROC-AUC of 88.06%, and an mAP of 85.28%; a decent model. It converges in 10–15 epochs and is not using any graph convolutions (and thus,

does not leverage PyTorch Geometric). All scores are very stable throughout the training process, and the accuracy is of more use due to the balanced nature of the dataset.

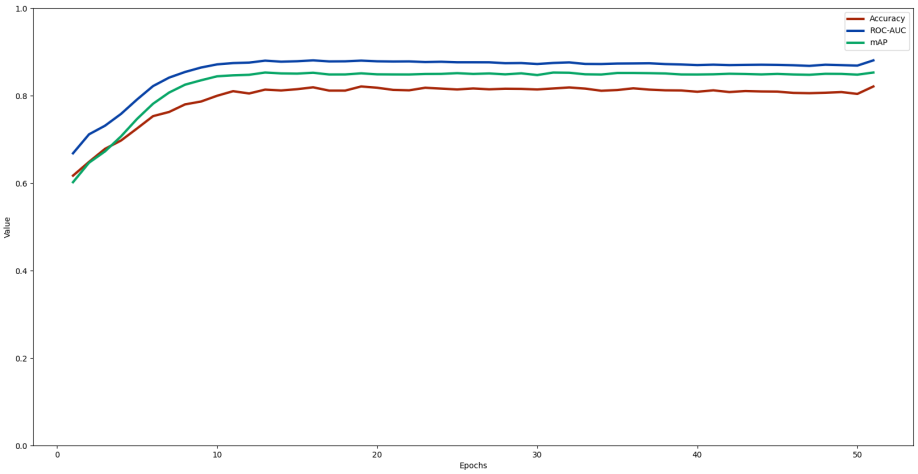

Figure 2.37: Baseline Multi-Modal Network, trained independently on a single machine

2.4.2 Client Number Comparison

Note: Each round was trained for 5 epochs in these experiments. For a fair comparison, the baseline was adjusted to contain only every 5th epoch as well.

Note: All experiments were performed on evenly distributed data. The training set was split based on the number of clients, and each client received an even subset to train on.

The number of clients clearly affects the Multi-Modal Network. The difference is small, when the number of clients is small (2–3 clients), with a 3% difference across all metrics. For 5 clients or more, the difference becomes 12% on average.

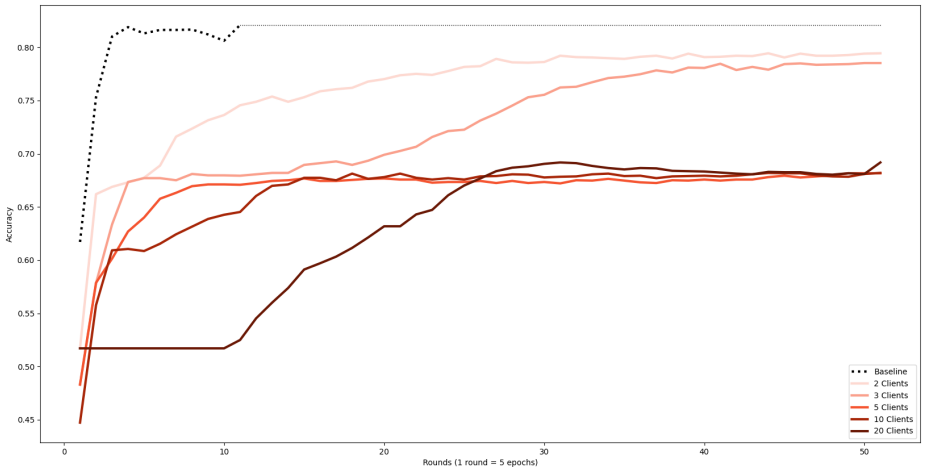

Figure 2.38: Multi-Modal Network - Client Number Comparison - Accuracy

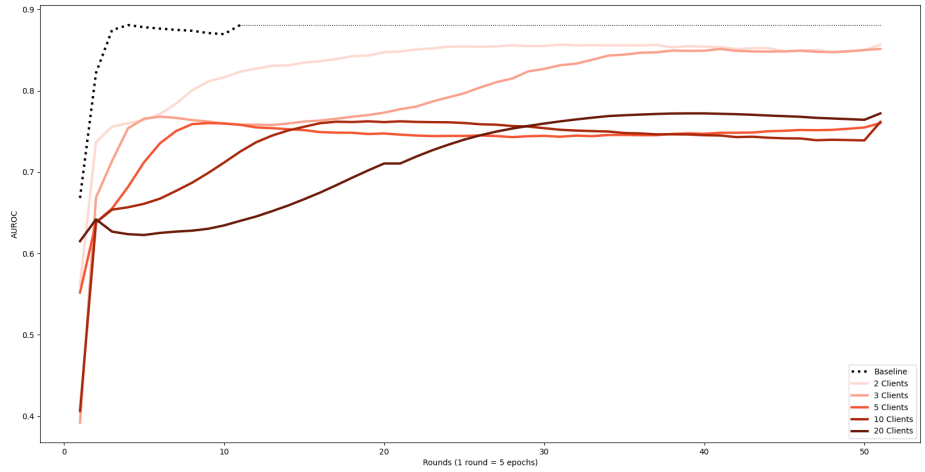

Figure 2.39: Multi-Modal Network - Client Number Comparison - ROC-AUC

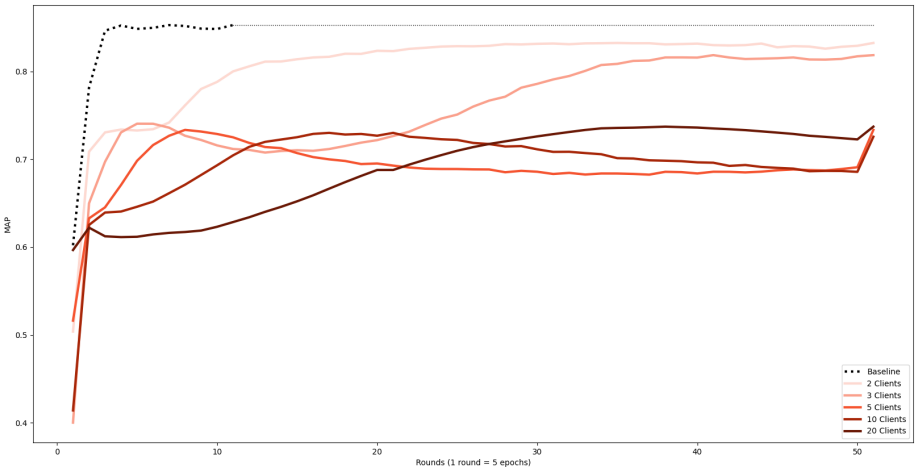

Figure 2.40: Multi-Modal Network - Client Number Comparison - mAP

	Accuracy	ROC-AUC	mAP
Baseline	82.10%	88.06%	85.28%
2 Clients	79.44%	85.66%	83.24%
3 Clients	78.52%	85.12%	81.86%
5 Clients	68.16%	76.02%	73.34%
10 Clients	68.20%	76.23%	73.00%
20 Clients	69.18%	77.21%	73.71%

Table 2.4.1: Multi-Modal Model - Client Number Comparison - Best Scores

2.4.3 Epoch Count Comparison

Note: Each experiment was performed using 2 clients with an even 50/50 data split.

When we train the models for 2 or more epochs per round, the effect seems to be minimal; with a 2–3% difference across all metrics. If we train for 1 epoch only, there is a noticeable effect of 12% in accuracy, 8% in ROC-AUC, and 7% in mAP.

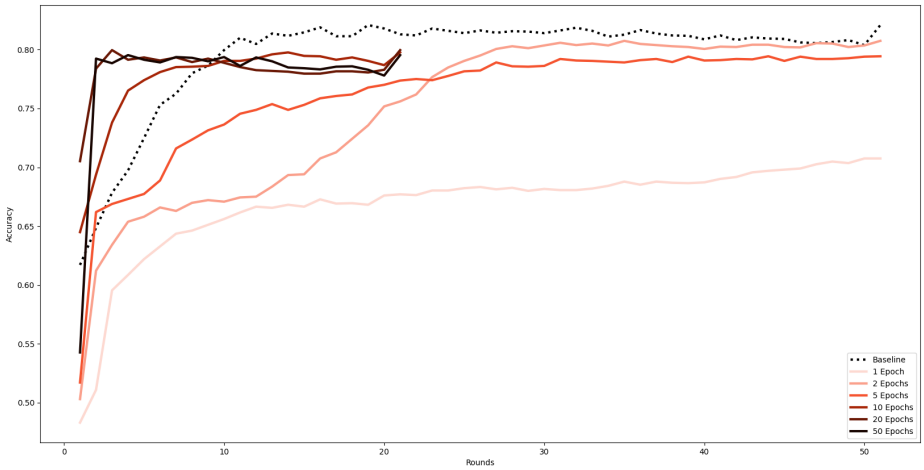

Figure 2.41: Multi-Modal Network - Epoch Count Comparison - Accuracy

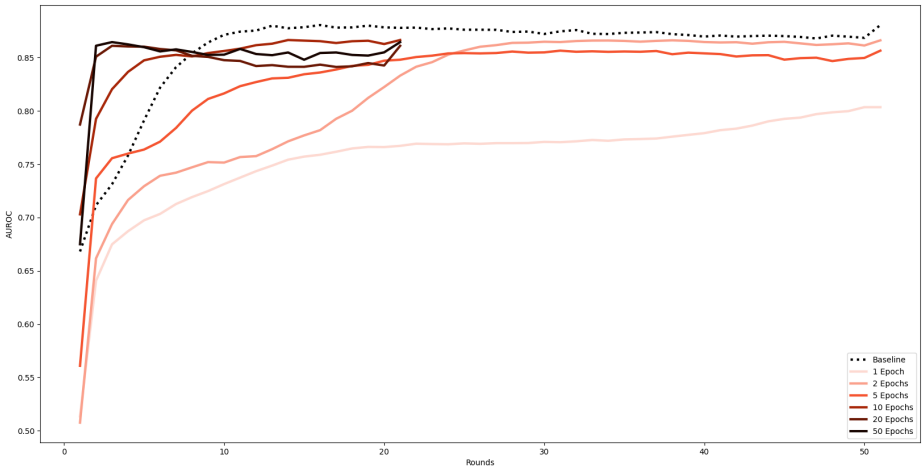

Figure 2.42: Multi-Modal Network - Epoch Count Comparison - ROC-AUC

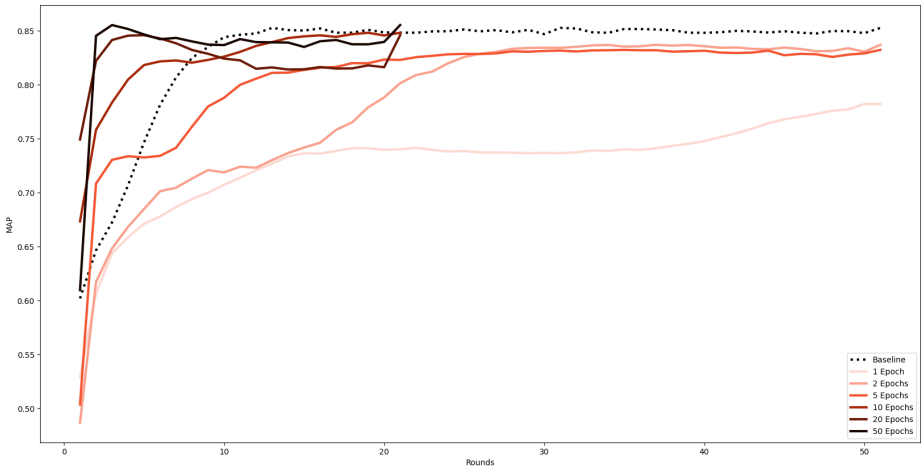

Figure 2.43: Multi-Modal Network - Epoch Count Comparison - mAP

	Accuracy	ROC-AUC	mAP
Baseline	82.10%	88.06%	85.28%
1 Epoch	70.75%	80.36%	78.22%
2 Epochs	80.75%	86.63%	83.71%
5 Epochs	79.44%	85.66%	83.24%
10 Epochs	79.77%	86.66%	84.83%
20 Epochs	79.97%	86.12%	84.60%
50 Epochs	79.54%	86.46%	85.55%

Table 2.4.2: Multi-Modal Model - Epoch Count Comparison - Best Scores

2.4.4 Uneven Data Distributions

Note: Each round was trained for 5 epochs in these experiments. For a fair comparison, the baseline was adjusted to contain only every 5th epoch as well.

Uneven data splits seem to have little effect across all metrics, with a difference of less than 2% among experiments. This applies to 2 client, 3 client, and 5 client experiments as well.

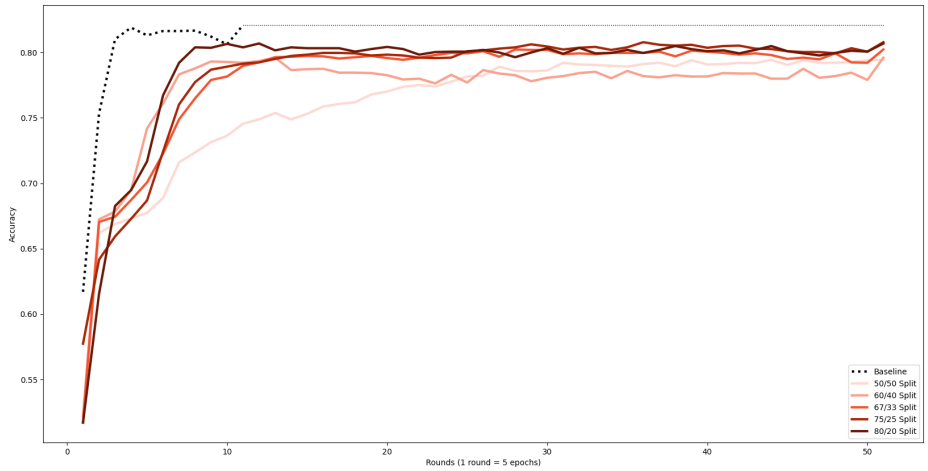

Figure 2.44: Multi-Modal Network - 2 Unbalanced Clients - Accuracy

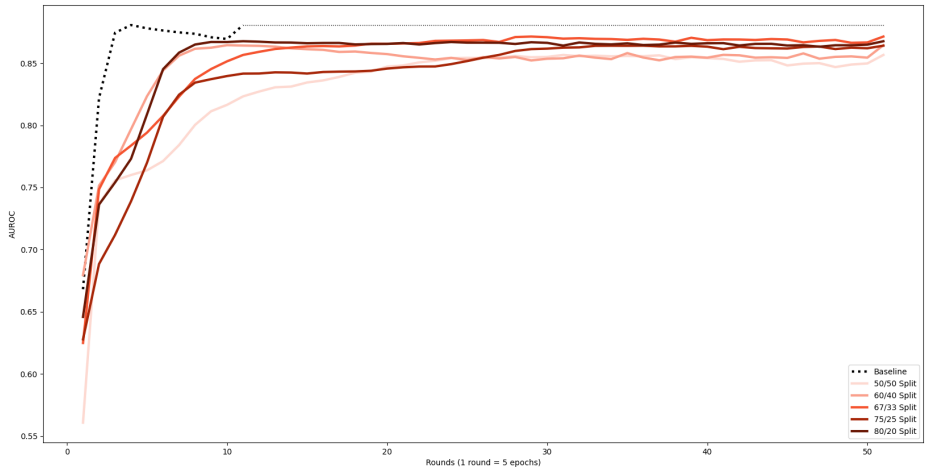

Figure 2.45: Multi-Modal Network - 2 Unbalanced Clients - ROC-AUC

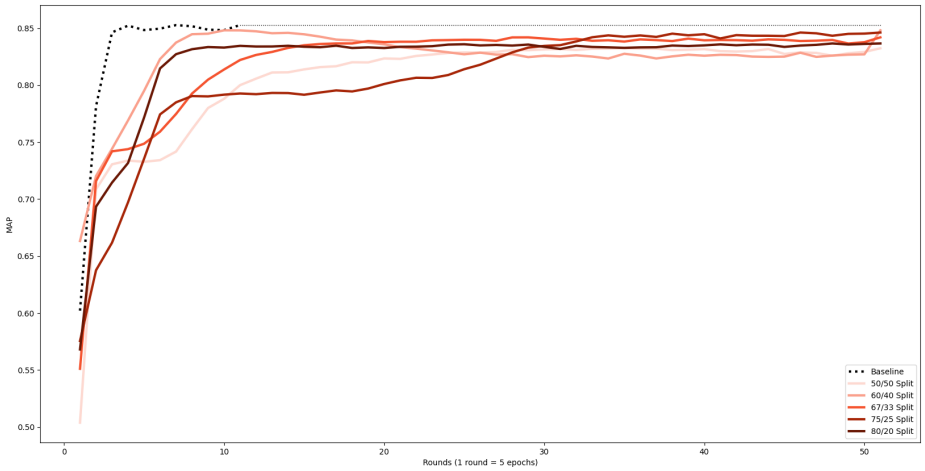

Figure 2.46: Multi-Modal Network - 2 Unbalanced Clients - mAP

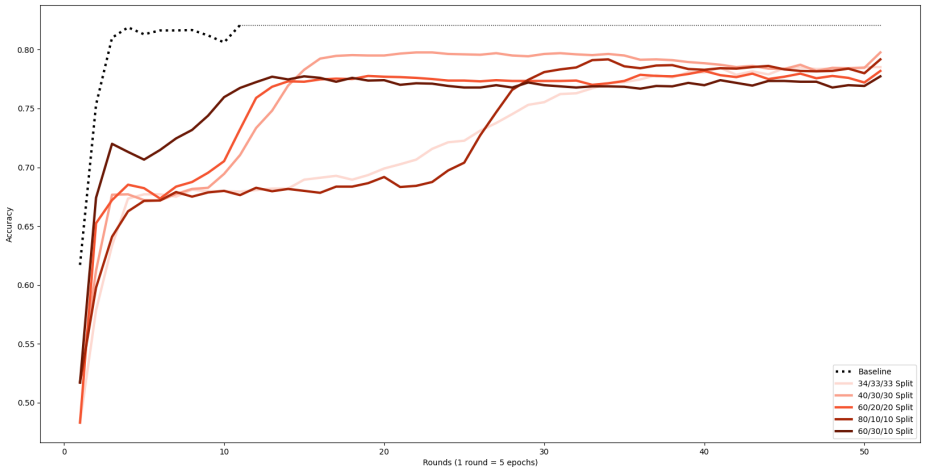

Figure 2.47: Multi-Modal Network - 3 Unbalanced Clients - Accuracy

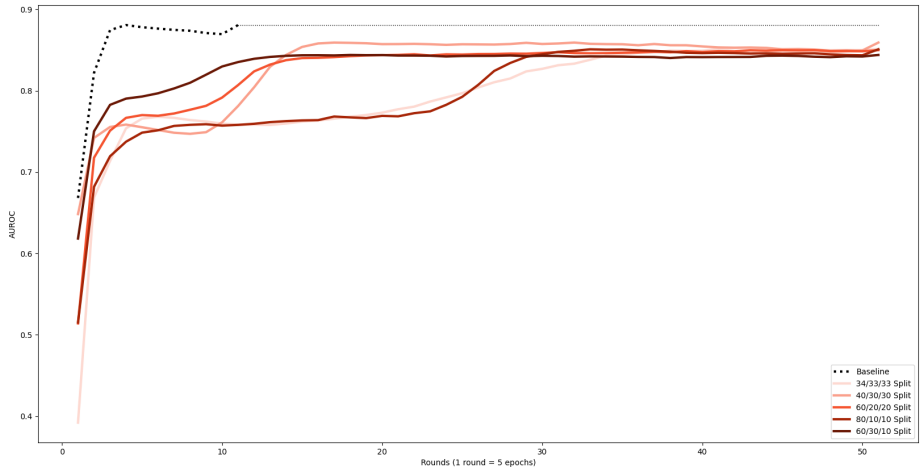

Figure 2.48: Multi-Modal Network - 3 Unbalanced Clients - ROC-AUC

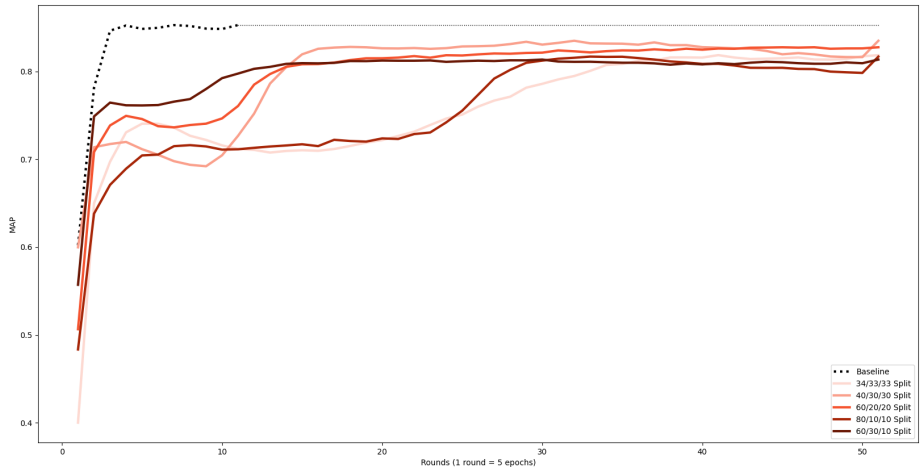

Figure 2.49: Multi-Modal Network - 3 Unbalanced Clients - mAP

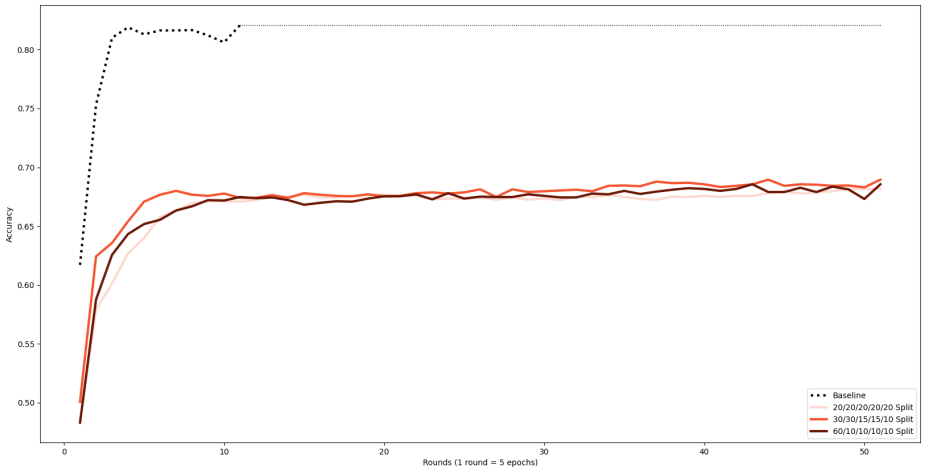

Figure 2.50: Multi-Modal Network - 5 Unbalanced Clients - Accuracy

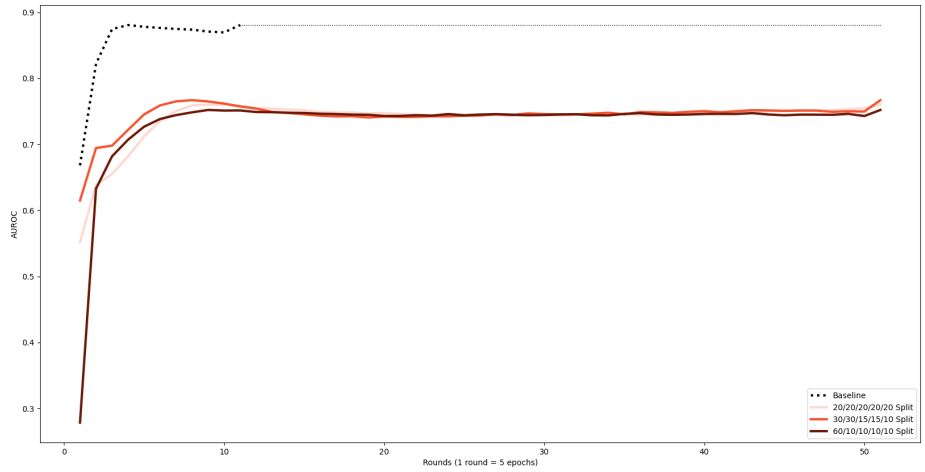

Figure 2.51: Multi-Modal Network - 5 Unbalanced Clients - ROC-AUC

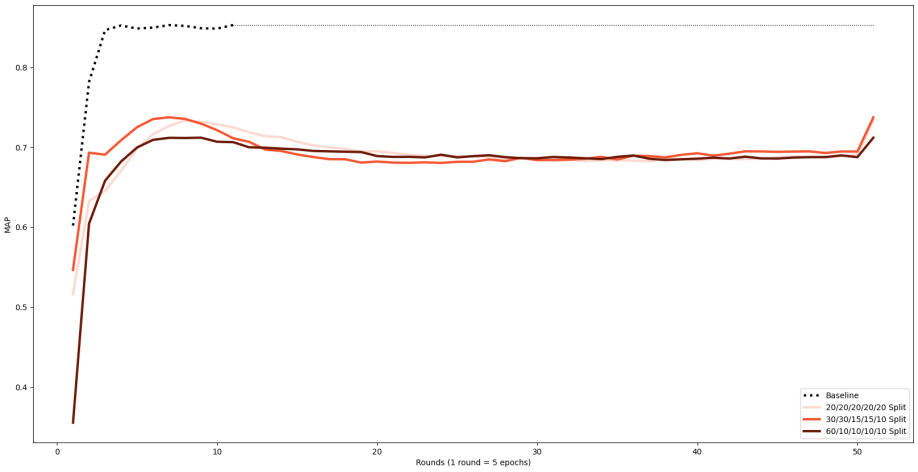

Figure 2.52: Multi-Modal Network - 5 Unbalanced Clients - mAP

	Accuracy	ROC-AUC	mAP
Baseline	82.10%	88.06%	85.28%
50/50 Spit	79.44%	85.66%	83.24%
60/40 Spit	79.61%	86.44%	84.82%
67/33 Spit	80.23%	87.14%	84.20%
75/25 Spit	80.79%	86.40%	84.62%
80/20 Spit	80.69%	86.76%	83.67%

Table 2.4.3: Multi-Modal Model - 2 Unbalanced Clients - Best Scores

	Accuracy	ROC-AUC	mAP
Baseline	82.10%	88.06%	85.28%
34/33/33 Split	78.52%	85.12%	81.86%
40/30/30 Split	79.77%	85.91%	83.49%
60/20/20 Split	78.20%	84.98%	82.75%
80/10/10 Split	79.18%	85.08%	81.69%
60/30/10 Split	77.74%	84.39%	81.35%

Table 2.4.4: Multi-Modal Model - 3 Unbalanced Clients - Best Scores

	Accuracy	ROC-AUC	mAP
Baseline	82.10%	88.06%	85.28%
20/20/20/20/20 Split	68.16%	76.02%	73.34%
30/30/15/15/10 Split	68.95%	76.69%	73.75%
60/10/10/10/10 Split	68.56%	75.21%	71.20%

Table 2.4.5: Multi-Modal Model - 5 Unbalanced Clients - Best Scores

2.5 Aggregators

The aggregators are an important part of the federated learning process. In all of the experiments described above, we used a plain weight average across all checkpoints, however, this is the simplest approach one might take. Even with this simple approach, most experiments performed well on unbalanced data splits (except in some cases for the Graph Convolutional Model).

However, this effect should not be taken for granted. If one of the clients was to contain wrong labels or highly biased data, it might affect the overall performance.

2.5.1 Weighted Averaging

One easy alternative is to weigh each client's contribution by the fraction of their dataset. We implemented such an aggregator, but didn't perform extensive experimentation with it. But even from a few quick experiments we can see that it is able to fix issues encountered on imbalanced datasets, such as the 67/33 split of the Graph Convolutional Network.

This is still a very basic aggregator, and other more advanced approaches are likely to easily outperform our results.

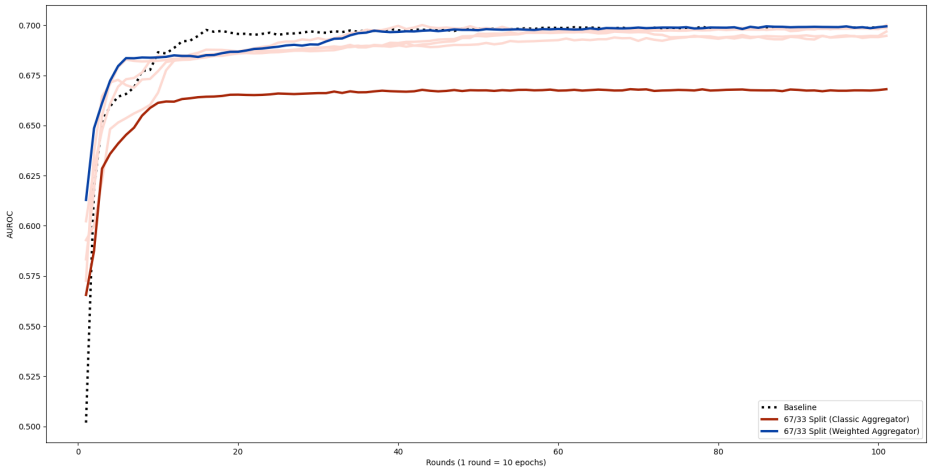

Figure 2.53: Weighted Average Aggregator on unbalanced Graph Convolutional Networks (ROC-AUC)

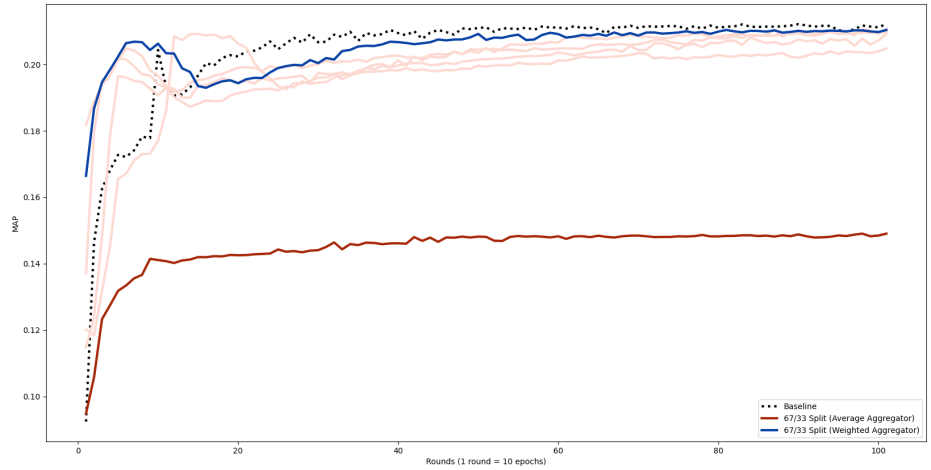

Figure 2.54: Weighted Aggregator on unbalanced Graph Convolutional Networks (mAP)

2.6 Discussion

The results look promising.

On the graph based models, we noticed that some parameters like the number of clients and the epochs trained per round can affect the results, but differences seem to be negligible, with recurring values of 3% ROC-AUC and 6% mAP noticed throughout the experiments. On the multi-modal network, the differences were steeper occasionally; especially when the number of clients was very large. But we should keep in mind; if models were trained on the small subsets used by the large client experiments (of one twentieth of the whole dataset), the performance would be very poor.

Also, we experimented with a very basic weight aggregator, and other options are likely to perform much better. The potential benefits of federated learning certainly outweigh the mostly small percentages lost throughout our experiments.

2.7 Next Steps

We came a long way, especially considering that we had to build the communication part from scratch, but there is still a long way to go.

Some areas remain untested. Question like, what happens when some clients train on GPU while others use CPU is yet to be answered. Different network architectures have not been considered. Furthermore, a common data format with pharmaceutical companies has not been agreed upon yet, and the model to be used is yet to be decided.

We recommend at least the following point to be considered before pushing the solution to production:

- development of a solid test suite;
- research on additional aggregators;
- agreement on a common data format, and implementation to handle that format;
- deciding and implementing the desired models to be used.

Additional features can also be added, depending on the status and needs of the project, like implementing aggregators for Tensorflow, Theano and other frameworks.

Technical Documentation

3.1 Installing Dependencies

In order to train the model some preliminary steps must be taken, namely, we must install all dependencies, which can be done with conda. If conda is not installed locally, one may follow the conda installation guide.

As a first step, we create a new environment using the provided snapshot. Then, we install some additional dependencies for PyTorch Geometric.

```
conda env create -f environment.yml
conda activate federated
bash install_additional_dependencies.sh
```

Note, this step has to be performed only one time. Once the environment is created and the depencies installed, one may use the new environment at any time by activating it.

```
conda activate federated
```

3.2 Project Structure

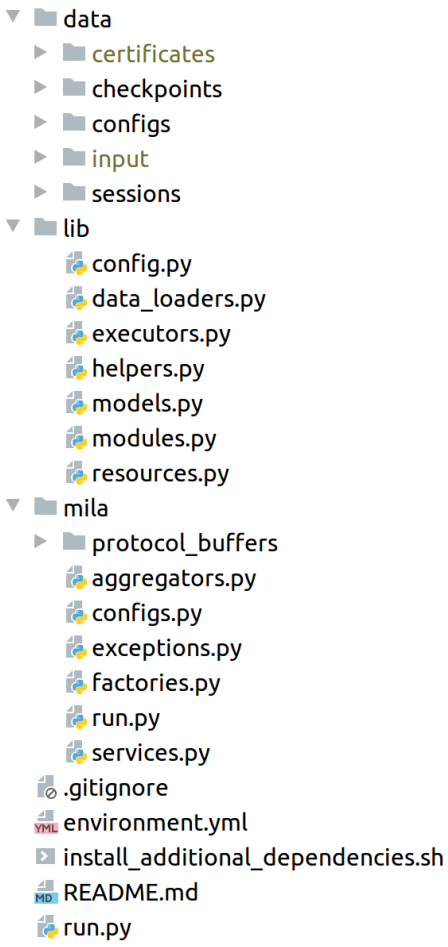

Figure 3.55: Project Structure

data: datasets, checkpoints, certs, logs

lib: model codebase;

lib/config.py: config management;

lib/data_loaders.py: data loading tools;

lib/executors.py: training / inference pipelines;

lib/helpers.py: helper classes;

lib/models.py: model architectures;

lib/modules.py: custom layers, cost functions;

lib/resources.py: data skeletons;

mila: communication module;

mila/protocol_buffers: gRPC messages

mila/aggregators.py: aggregators' code;

mila/configs.py: server and client configs;

mila/exceptions.py: exception handling tools;

mila/factories.py: abstractions models need to extend;

mila/run.py: server and client entry point

services.py: server and client implementation;

environment.yml, install_additional_dependencies.sh: Environment setup tools

run.py: Model entry point.

3.3 Models

Models live under the `lib/` folder.

Each experiment requires a configuration file to run, which we describe in the next section. Furthermore, for the models to be compatible with Mila, it has to extend 2 abstract classes from the Mila library.

- the **AbstractConfiguration** class - responsible for configuration handling
- and the **AbstractExecutor** class - responsible for starting the train, validation, and inference processes.

3.3.1 Configuration Files

Configuration files are stored in JSON format and parsed by a class extending `AbstractConfiguration` .

All 3 models make use of the same configuration file. In what follows, we list its available options:

model_name (no default)

Specifies which model should be used. There are 3 options available:

- GraphConvolutionalNetwork
- GraphIsomorphismNetwork
- BimodalProteinLigandNetwork

model_options (no default)

A list of arguments the model receives when it is instantiated (this setting should be a dictionary)

The options for `GraphConvolutionalNetwork` and `GraphIsomorphismNetwork` are:

- *in_features*: input layer size (number of atomic features) (type: int)

- *hidden_features*: hidden layer size (type: int)
- *out_features*: output layer size (number of classes) (type: int)
- *dropout*: dropout rate (between 0 and 1) (type: float)

The options for `BimodalProteinLigandNetwork` are:

- *in_features_protein*: input protein features (number of unique proteins) (type: int)
- *in_features_ligand*: input ligand features (fingerprint size) (type: int)
- *out_features*: output layer size (number of classes) (type: int)

data_loader (no default)

Which data loader to use.

`GraphConvolutionalNetwork` and `GraphIsomorphismNetwork` only supports `MoleculeNetLoader` right now, and the `BimodalProteinLigandNetwork` supports only `ProteinLigandLoader` .

dataset (no default)

If the "data_loader" is MoleculeNetLoader, specifies which dataset will be used. Possible options include: "tox21", "pcba", "muv", "hiv", "bbbp", "toxcast", "sider", and "clintox".

Note: the dataset is automatically downloaded

input_path (no default)

Path to the input dataset. This is the path to a CSV file for `ProteinLigandLoader` , and the path to a folder where the dataset should be downloaded to for `MoleculeNetLoader` .

output_path (no default)

Path where checkpoints will be saved to

checkpoint_path (no default)

Load a particular checkpoint. When training, this is used as a backbone. When running evaluation, or inference, this field is mandatory.

subset_distributions (default=[1])

This field, along with "subset_id" is used to run experiments on a subset of the whole dataset. "subset_distributions" specifies a list of fractions the dataset should be split into. The expected value is a list of floating point values which sum up to 1.

subset_id (default=0)

This field, along with "subset_distributions" is used to run experiments on a subset of the whole dataset. "subset_id" specified which subset should be used from the list of "subset_distributions". This is a 0-based index.

epochs (default=100)

Specifies how many epochs to train for.

batch_size (default=32)

Specifies the batch size.

learning_rate (default=0.01)

Specifies the learning rate.

weight_decay (default=0)

Customizes weight decay.

threshold (default=0.5)

Specifies the threshold.

train_ratio (default=0.8)

Specifies the size of the training set (a train/test split is performed by all data loaders)

split_method (default="index")

How to perform the data split? 2 options are available right now:

- "index"
- "random"

seed (default=42)

Used when "split_method" is set to "random".

use_cuda (default=true)

If true, and if an NVIDIA graphics card is available, the model will use GPU acceleration.

enabled_gpus (default=[0, 1, 2, 3])

A list of GPU ids to use when CUDA is used.

log_frequency (default=20)

After how many iterations should an update be printed when training the model.

log_level (default="info")

How many logs should be printed. The available options are: "debug", "info", "warn", "error", and "critical".

log_format (default="")

Can be used to include additional details for logged messages, like timestamps.

3.3.2 Commands

There are 4 commands available.

3.3.3 Training

To train a model, the following command can be used:

```
python run.py train {config_path}
```

```
(federated) elix@bash:/var/www/federated$ python run.py train data/configs/model/gcn.json
epoch: 1 - iteration: 5 - examples: 640 - loss: 1.6510812520980835 - time elapsed: 0:00:00 - progress: 0.1022
epoch: 1 - iteration: 10 - examples: 1280 - loss: 1.5520516395568849 - time elapsed: 0:00:00 - progress: 0.2043
epoch: 1 - iteration: 15 - examples: 1920 - loss: 1.3994948625564576 - time elapsed: 0:00:00 - progress: 0.3065
epoch: 1 - iteration: 20 - examples: 2560 - loss: 1.3421561241149902 - time elapsed: 0:00:00 - progress: 0.4086
epoch: 1 - iteration: 25 - examples: 3200 - loss: 1.2676585912704468 - time elapsed: 0:00:01 - progress: 0.5108
epoch: 1 - iteration: 30 - examples: 3840 - loss: 1.1720728397369384 - time elapsed: 0:00:01 - progress: 0.6129
epoch: 1 - iteration: 35 - examples: 4480 - loss: 1.076970887184143 - time elapsed: 0:00:01 - progress: 0.7151
epoch: 1 - iteration: 40 - examples: 5120 - loss: 1.0261453866958619 - time elapsed: 0:00:01 - progress: 0.8172
epoch: 1 - iteration: 45 - examples: 5760 - loss: 0.939975917339325 - time elapsed: 0:00:01 - progress: 0.9194
Saving checkpoint: data/checkpoints/gcn/checkpoint.1
epoch: 2 - iteration: 5 - examples: 640 - loss: 0.8188358187675476 - time elapsed: 0:00:01 - progress: 0.1022
epoch: 2 - iteration: 10 - examples: 1280 - loss: 0.7449125289916992 - time elapsed: 0:00:01 - progress: 0.2043
epoch: 2 - iteration: 15 - examples: 1920 - loss: 0.6891981363296509 - time elapsed: 0:00:01 - progress: 0.3065
epoch: 2 - iteration: 20 - examples: 2560 - loss: 0.6271933436393737 - time elapsed: 0:00:01 - progress: 0.4086
epoch: 2 - iteration: 25 - examples: 3200 - loss: 0.5882496953010559 - time elapsed: 0:00:01 - progress: 0.5108
epoch: 2 - iteration: 30 - examples: 3840 - loss: 0.5321418404579162 - time elapsed: 0:00:02 - progress: 0.6129
epoch: 2 - iteration: 35 - examples: 4480 - loss: 0.4965818762779236 - time elapsed: 0:00:02 - progress: 0.7151
epoch: 2 - iteration: 40 - examples: 5120 - loss: 0.4600786030292511 - time elapsed: 0:00:02 - progress: 0.8172
epoch: 2 - iteration: 45 - examples: 5760 - loss: 0.4279358506202698 - time elapsed: 0:00:02 - progress: 0.9194
Saving checkpoint: data/checkpoints/gcn/checkpoint.2
```

Figure 3.56: Sample output of the training command.

3.3.4 Evaluating a Single Checkpoint

To evaluate the performance of the ‘checkpoint_path‘ specified in the configuration file, use the following command

```
python run.py eval {config_path}
```

```
(federated) elix@bash:/var/www/federated$ python run.py eval data/configs/model/gcn.json
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.20
Results(accuracy=0.9235937064057557, roc_auc_score=0.6017352604054721, average_precision=0.14202833727883182)
(federated) elix@bash:/var/www/federated$
```

Figure 3.57: Sample output of the evaluation command.

3.3.5 Evaluating Multiple Checkpoints

To evaluate all checkpoints found in ‘save_path‘ option specified in the configuration file, use the following command

```
python run.py eval {config_path}
```

```
(federated) elix@bash:/var/www/federated$ python run.py analyze data/configs/model/gcn.json
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.1
Results(accuracy=0.5897798921815177, roc_auc_score=0.5669079475088123, average_precision=0.12370861192977232)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.2
Results(accuracy=0.6011243486573233, roc_auc_score=0.572564618300161, average_precision=0.13025362354429615)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.3
Results(accuracy=0.8670450949299089, roc_auc_score=0.575121470605737, average_precision=0.13004910924219812)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.4
Results(accuracy=0.9233982294234778, roc_auc_score=0.5731728565335796, average_precision=0.12985773488412947)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.5
Results(accuracy=0.923528905991033, roc_auc_score=0.5765950451278467, average_precision=0.128679341055074)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.6
Results(accuracy=0.9235937064057557, roc_auc_score=0.5754187506868594, average_precision=0.12734277424048054)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.7
Results(accuracy=0.9235937064057557, roc_auc_score=0.5781369663655589, average_precision=0.12703691719006027)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.8
Results(accuracy=0.9235937064057557, roc_auc_score=0.5784101249120567, average_precision=0.1263590986389785)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.9
Results(accuracy=0.9235937064057557, roc_auc_score=0.5792559582738116, average_precision=0.12618832092848062)
Restoring from Checkpoint: data/checkpoints/gcn/checkpoint.10
Results(accuracy=0.9235937064057557, roc_auc_score=0.5802817438346853, average_precision=0.1270240960920648)
```

Figure 3.58: Sample output of the analyze command.

3.3.6 Inference

To run inference using the ‘checkpoint_path‘ specified in the configuration file, the following command can be used:

```
python run.py predict {config_path}
```

```
(federated) elix@bash:/var/www/federated$ python run.py predict data/configs/model/multimodal.json
Restoring from Checkpoint: data/memos/last/mm_2cl_1ep_50-50sp.50
1
0
1
1
1
0
0
0
1
0
1
1
1
1
0
```

Figure 3.59: Sample output of the inference command.

3.4 Federated Learning (Mila)

Server and client processes are started similar to how models work. Each servicer and consumer requires a configuration file, which is fed to the entry point. In what follows, we describe the options for servers and clients separately.

3.4.1 Server Configurations

Server configuration files are expected to be in JSON format. The available options include:

task_configuration_file (no default)

Path to the model configuration file. This will be sent to each client along with the checkpoint.

config_type: (no default)

Module path to the configuration class (ie: "lib.config.Config")

executor_type: (no default)

Module path to the executor class (ie: "run.Executor")

aggregator_type (no default)

Which aggregator to use? Options include:

- PlainTorchAggregator
- WeightedTorchAggregator

aggregator_options (default={})

Input parameters for the aggregator (the expected value is a dictionary) "PlainTorchAggregator" does not expect anything; the "WeightedTorchAggregator" expects a "weights" options which is mapping

between the client's "name" parameter and the expected wight.

For example, if we have 2 clients named "tester1" and "tester2", this option could be:

```
"aggregator_options": {
    "weights": {
        "tester1": 0.67,
        "tester2": 0.33
    }
}
```

target (default="localhost:8024")

The gRPC service location URL (that is, the server address).

rounds_count (default=10)

How many rounds to perform

save_path (default="data/logs/server/")

Where to checkpoints received from clients and the aggregate models?

start_point (default=null)

Optionally, specify a checkpoint for the first round. If nothing is specified, clients will start training from scratch.

workers (default=2)

Maximum number of processes handling client requests

minimum_clients (default=2)

Minimum number of clients required to start federated learning. The server won't start the first round until this number is reached.

maximum_clients (default=100)

Maximum number of clients allowed to join the federated learning process.

client_wait_time (default=10)

Once the "minimum_clients" number is reach, the server will wait this many seconds for additional clients before the process starts. After this time expires, no new members will be allowed to join.

heartbeat_timeout (default=300)

How long to wait for a keepalive signal from clients before declaring them "dead"?

use_secure_connection (default=false)

When true, the communication will be performed through HTTPS protocol. The 3 SSL files specified below must be valid for this to work.

ssl_private_key (default="data/certificates/server.key")

gRPC secure communication private key

ssl_cert (default="data/certificates/server.crt")

gRPC secure communication SSL certificate

ssl_root_cert (default="data/certificates/rootCA.pem")

gRPC secure communication trusted root certificate

options

Additional gRPC options. "grpc.max_send_message_length" represents the maximum length of a sent message, and "grpc.max_receive_message_length" the maximum length of a received message.

The default value for this option is:

```
[
    ("grpc.max_send_message_length", 1000000000),
    ("grpc.max_receive_message_length", 1000000000)
]
```

blacklist (default=[])

A list of IP addresses which will be declined upon authentication.

whitelist (default=[])

A list of IP addresses which will be allowed to join the federated learning process. If "use_whitelist" is True, these will be the only IP addresses allowed to join.

use_whitelist (default=false)

Enables whitelist filtering.

3.4.2 Client Configurations

Client configuration files are also expected to be in JSON format. The available options include:

name (no default)

A unique identifier for this client (could be a company name for example)

save_path (default="data/logs/client/")

Where to save checkpoints received from the client?

heartbeat_frequency (default=60)

How often to send keepalive signals to the client?

retry_timeout (default=1)

If a request failes because the server is under heavy load, we retry the connection after this many seconds.

model_overwrites

Used to override model configuration options. Generally, clients might want to change the path where local (model) checkpoints are stored. The default value for this option is:

```
{
    "output_path": "data/logs/local/",
    "epochs": 5
}
```

config_type: (no default)

Module path to the configuration class (ie: "lib.config.Config")

executor_type: (no default)

Module path to the executor class (ie: "run.Executor")

target (default="localhost:8024")

The gRPC service location URL (that is, the server address).

use_secure_connection (default=false)

When true, the communication will be performed through HTTPS protocol. The 3 SSL files specified below must be valid for this to work.

ssl_private_key (default="data/certificates/client.key")

gRPC secure communication private key

ssl_cert (default="data/certificates/client.crt")

gRPC secure communication SSL certificate

ssl_root_cert (default="data/certificates/rootCA.pem")

gRPC secure communication trusted root certificate

3.4.3 Creating SSL Certificates

We first need to create the root private key and a self-signed root certificate

```
openssl genrsa -des3 -out rootCA.key 2048

openssl req -x509 -new -nodes -key rootCA.key
-sha256 -days 1024 -out rootCA.pem
```

Next, we create a server private key, a server certificate signing request (CSR), and we sign the CSR using the root certificate:

```
openssl genrsa -out server.key 2048

openssl req -new -key server.key -out server.csr

openssl x509 -req -in server.csr -CA rootCA.pem
-CAkey rootCA.key -CAcreateserial -out server.crt
-days 500 -sha256
```

The client creates the client private key, a client certificate signing request (CSR). A common name should be specified when creating the CSR.

```
openssl genrsa -out client.key 2048
openssl req -new -key client.key -out client3.csr
```

The client sends the CSR to the server which wigns it using the root certificate. The server sends "client.crt" back to the client.

```
openssl x509 -req -in client.csr -CA rootCA.pem
-CAkey rootCA.key -CAcreateserial -out client3.crt
-days 500 -sha256
```

3.4.4 Commands

Starting the server can be done using following command:

```
python -m mila.run server {config_path}
```

Starting a client process is very similar:

```
python -m mila.run client {config_path}
```

```
(federated) elix@bash:/var/www/federated$ python -m mila.run server data/configs/mila/2/server.json
E1113 18:12:03.679359766   21507 socket_utils_common_posix.cc:223] check for SO_REUSEPORT: {"created":"@1605258723.679350852","description":"SO_REUSEPORT unavailable o
n compiling system","file":"src/core/lib/iomgr/socket_utils_common_posix.cc","file_line":192}
[CAUTION] Connection is insecure!
Starting server at: [localhost:8024]
[tester1|127.0.0.1] Successfully authenticated (clients=1)
[tester2|127.0.0.1] Successfully authenticated (clients=2)
[tester1|127.0.0.1] Sending Model (round=1)
[tester2|127.0.0.1] Sending Model (round=1)
[tester1|127.0.0.1] Checkpoint Received
[tester2|127.0.0.1] Checkpoint Received
Start aggregation (round=1)
Aggregate model saved: [data/logs/server//1.aggregate]
Starting round [2]
[tester1|127.0.0.1] Sending Model (round=2)
[tester2|127.0.0.1] Sending Model (round=2)
[tester1|127.0.0.1] Checkpoint Received
[tester2|127.0.0.1] Checkpoint Received
Start aggregation (round=2)
Aggregate model saved: [data/logs/server//2.aggregate]
Starting round [3]
[tester2|127.0.0.1] Sending Model (round=3)
[tester1|127.0.0.1] Sending Model (round=3)
[tester2|127.0.0.1] Checkpoint Received
[tester1|127.0.0.1] Checkpoint Received
Start aggregation (round=3)
Aggregate model saved: [data/logs/server//3.aggregate]
Starting round [4]
```

Figure 3.60: Sample output of from the server.

```
(federated) elix@bash:/var/www/federated$ python -m mila.run client data/configs/mila/2/client1.json
[CAUTION] Connection is insecure!
[CAUTION] Connection is insecure!
[CAUTION] Connection is insecure!
epoch: 1 - iteration: 5 - examples: 640 - loss: 1.2937450647354125 - time elapsed: 0:00:00 - progress: 0.1525
epoch: 1 - iteration: 10 - examples: 1280 - loss: 1.1932451248168945 - time elapsed: 0:00:00 - progress: 0.305
epoch: 1 - iteration: 15 - examples: 1920 - loss: 1.1230454444885254 - time elapsed: 0:00:00 - progress: 0.4575
epoch: 1 - iteration: 20 - examples: 2560 - loss: 1.0506272077560426 - time elapsed: 0:00:00 - progress: 0.61
epoch: 1 - iteration: 25 - examples: 3200 - loss: 1.0024004578590393 - time elapsed: 0:00:00 - progress: 0.7624
epoch: 1 - iteration: 30 - examples: 3840 - loss: 0.9705534219741822 - time elapsed: 0:00:01 - progress: 0.9149
Saving checkpoint: data/logs/local/tester1/checkpoint.1
epoch: 2 - iteration: 5 - examples: 640 - loss: 0.9004879713058471 - time elapsed: 0:00:01 - progress: 0.1525
epoch: 2 - iteration: 10 - examples: 1280 - loss: 0.8540361762046814 - time elapsed: 0:00:01 - progress: 0.305
epoch: 2 - iteration: 15 - examples: 1920 - loss: 0.8188570618629456 - time elapsed: 0:00:01 - progress: 0.4575
epoch: 2 - iteration: 20 - examples: 2560 - loss: 0.8083093166351318 - time elapsed: 0:00:01 - progress: 0.61
epoch: 2 - iteration: 25 - examples: 3200 - loss: 0.7534210562705994 - time elapsed: 0:00:01 - progress: 0.7624
epoch: 2 - iteration: 30 - examples: 3840 - loss: 0.7510416865348816 - time elapsed: 0:00:01 - progress: 0.9149
Saving checkpoint: data/logs/local/tester1/checkpoint.2
epoch: 3 - iteration: 5 - examples: 640 - loss: 0.6974666357040405 - time elapsed: 0:00:01 - progress: 0.1525
epoch: 3 - iteration: 10 - examples: 1280 - loss: 0.6492686152458191 - time elapsed: 0:00:01 - progress: 0.305
epoch: 3 - iteration: 15 - examples: 1920 - loss: 0.6089567184448242 - time elapsed: 0:00:01 - progress: 0.4575
epoch: 3 - iteration: 20 - examples: 2560 - loss: 0.5941434621810913 - time elapsed: 0:00:01 - progress: 0.61
epoch: 3 - iteration: 25 - examples: 3200 - loss: 0.542203176021576 - time elapsed: 0:00:02 - progress: 0.7624
epoch: 3 - iteration: 30 - examples: 3840 - loss: 0.5095760762691498 - time elapsed: 0:00:02 - progress: 0.9149
Saving checkpoint: data/logs/local/tester1/checkpoint.3
epoch: 4 - iteration: 5 - examples: 640 - loss: 0.4610184669494629 - time elapsed: 0:00:02 - progress: 0.1525
epoch: 4 - iteration: 10 - examples: 1280 - loss: 0.4187736213207245 - time elapsed: 0:00:02 - progress: 0.305
epoch: 4 - iteration: 15 - examples: 1920 - loss: 0.4015202224254608 - time elapsed: 0:00:02 - progress: 0.4575
```

Figure 3.61: Sample output of from the client.

3.5 Environment Information

Note: All experiments were performed using only CPU resources

OS: Ubuntu 16.04.6 LTS

Python Version: 3.8.3

PyTorch Version: 1.6.0

CUDA Toolkit Version: 10.2.89

cuDNN Version: 7.6.5