INSTITUTE OF LOGIC AND COMPUTATION

MACHINE LEARNING - EXERCISE 3

Group 8

Member:
Peter HOLZNER, 01426733
Alexander LEITNER, 01525882
Mario HITI, 01327428

Submission: February 21, 2021

Todo list

Contents

1 Introduction

1.1 Task description

We have chosen Topic 3.1.2.5 Secure Multi-party computation for Image Classification as our project for Exercise 3. The task is descried as follows:

"Utilising e.g. the library mpyc for Python (https://github.com/lschoe/mpyc) and the implementation of a secure computation for a binarised multi-layer perceptron, first try to recreate the results reported in Abspoel et al, "Fast Secure Comparison for Medium-Sized Integers and Its Application in Binarized Neural Networks", i.e. train a baseline CNN to estimate a potential upper limit of achievable results, and then train the binarized network, as a simplified but still rather performant version, in a secure way. If needed, you can use a subset of the MNIST dataset.

Then, try to perform a similar evaluation on another small dataset, either already available in grayscale, or converted to grayscale, e.g. using (a subset of) the AT&T faces dataset. Specifically, evaluate the final result in terms of effectiveness, but also consider efficiency aspects, i.e. primarily runtime, but also other resource consumption."

1.2 Outline

The first part of this documentation covers the different frameworks that we have used and our experiences while working with them. Next the models and datasets we used are introduced and briefly discussed. Afterwards, we provide our benchmarks that we used to evaluate the frameworks and models. Based on the results of our benchmarks, we look to summarize our experiences and results in the last chapter.

Installation instructions and requirements are given at the start of section 5 in subsection 5.1

1.3 Foreword

Since multi-party systems are primarily used to increase computational power and enable secure, private use of sensible data, trying to achieve a high accuracy is not our focus. Therefore, we are specifically evaluating memory usage and run time as our main indicator for effectiveness.

We would also like to preface our report by saying that most - if not all - Secure MPC and privacy preserving (machine learning) frameworks/libraries are still in their infancy and very much in development - especially compared to the at this point relatively stable conventional frameworks (Tensorflow, PyTorch, Keras, ...). This fact provided us with many challenges during our project that related more to programming/software development than to ML.

2 Frameworks

2.1 MPyC

MPyC is a framework for secure multi-party computation using cryptographic methods that are based on secret sharing techniques[7]. It offers secure replacements for most commonly used mathematical operations (vector addition, matrix multiplication, standard division, etc) as well as other functions such as standard deviation, datatypes or random number generation. Furthermore a new protocol for a fast secure integer comparison is featured.

Several demos for different applications are available including examples for evaluating a BNN and a CNN. The framework does not include model classes like the ones in PyTorch and have to be implemented "by hand" - e.g. a fully connected layer is specified a matrix-vector-product followed by a vector-addition in contrast to PyTorch- or Keras-style abstrations. Examples for securely training said networks are not included in the framework.

2.2 CrypTen

CrypTen[9] is a machine learning framework that aims to implement secure computations for PyTorch users and uses similar principles as MPyC, but looks to provide a more high-level API. The backed is based on the Secure Multiparty Communication paradigm[1] and assumes a semi-honest threat model which is based on the following assumptions[1]:

- every party faithfully carries out its' instructions and communicates its' results,
- secure communications channels are used,
- each party can only see the data communicated to them,
- private sources of randomness are available to each party,
- received ("seen") data can be used and processed to infer information.

The above functionality is intended to be accessible via an API that follows PyTorch conventions as closely as possible. For instance, PyTorch models can be converted either through a factory function or by simply replacing the word 'torch' with 'CrypTen' during module imports for existing torch models. Torch tensors can also be simply be converted to encrypted Cryptensors to access CrypTen's secure computation functions. The tensors are converted via fixed point encoding based on the owners ('source') secret key to an encrypted (integer) state. The owner then computes each parties share of the Cryptensor, as in the previously mentioned example, and communicates it to them (secret sharing). A shared (MPC)Tensor then looks like the following example output when each party looks at its' share:

Algorithm 1: Shared MPCTensor on each party, taken from here

```
Rank 0:
 MPCTensor(
    _tensor=tensor([-547737859053805 6738, -7414565399526290677,
        1067182702270173948])
    plain_text=HIDDEN
    ptype=ptype.arithmetic
)

Rank 1:
 MPCTensor(
    _tensor=tensor([ 5477378590538122274,  7414565399526421749,
        -1067182702269977340])
    plain_text=HIDDEN
    ptype=ptype.arithmetic
)
```

Only the original owner can decrypt to the original data/tensor. A similar scheme is employed to hide/encrypt a models weights from the other parties. The framework supports different use cases, such as different distributions of data (seperate parties can hold different parts of the feature set/the picture, while another holds the labels) and also supports model hiding - note that only the models weights (and biases) are hidden and not the model structure.

The concurrent execution is based on a communication model that is similar to MPI's and therefore naturally simplifies to allow for multiprocess execution on a single host as well (e.g. for development). Communication between parties is based on TCP connections that can be secured via ssh tunnels and revolve around a rendezvous point (IP address or a file, in the case of local multiprocess execution).

[1] see the earlier section

2.3 Restrictions

We used the latest release version available via pip, which is version 0.1. However, this release version is also restricted to use python 3.7 and torch 1.4.0 only. In other words, the framework is still very much in development.

Unfortunately, this is also reflected in the state of the documentation and the available examples and tutorials hosted on their GitHub [2]. Some of their examples (they provide on their GitHub) flat out do not work or do not work without tweaks, such as the encrypted training example here: Tutorial on Encrypted Training with CrypTen. They were either based on a previous or newer version of the framework API, since the autograd functionality had to be enabled differently than in the example.

Apart from that, we encountered three major problems:

1. s trained CrypTen models (weights) cannot be loaded from file (although they can be saved),
2. not all of torch's neural network module has been implemented securely yet or been is available in CrypTen models,
3. proper execution across multiple parties was not possible for us (substituted for local python based multiprocessing).

For number 1, it was still possible to load pre-trained torch models but specifically NOT CrypTen models. After struggling with this, I also checked the tutorials again - they specifically avoid that part. They only securely train a network and immediately evaluate it while it is still in memory. They only load from a standard PyTorch model that was trained in the clear. Due to the long time it takes to securely train a model with CrypTen and the issues we had to even get training running, we had to rely on models trained in the clear (the MPyC team [7] also trained their Binary-MLP for MNIST in the clear).

For number 2, we had to switch to a different activation function (ReLU instead of sign) because the sign function, although implemented and usable on Cryptensors, was not supported for use in a CrypTen model.

For number 3, this seems to be notoriously difficult for CrypTen users , outside of the provided AWS launcher, as can be seen in on their GitHub issue board. We also simply lacked time due to the other difficulties we faced. We therefore based our benchmarks on the easier to use local multiprocessing execution which of course means that our benchmarks only provide a baseline best-case performance estimation for execution times - accuracy and prediction quality is unaffected by this.

2.4 Other frameworks

MPyC and CrypTen use very similar assumptions and share many design goals, apart from the fact that CrypTen is based on and specifically designed for PyTorch. MPyC meanwhile is a more bare-bones/low-level type of framework in terms of its' API. They are also not the only 'available' (most of them still more or less in development/beta stages) frameworks, e.g. the Tensorflow pendant Tfencrypted [8].

Another interesting project is the open-source, community based project Syft (PySyft) [3] [11] which also simplifies and streamlines the connection procedure and allows for Google Colab sytle joint programming / data science sessions (Duet). It's also not tied to a single framework but looks to be accessible to users of Tensorflow, Torch and Keras. In fact, it integrated CrypTen as a usable backend for MPC. [4]. Note, that the above linked blog article [4] also mentions the difficulties they had when trying to benchmark CrypTen computations across multiple parties - they also had to fall back on multiprocessing for their benchmarks.

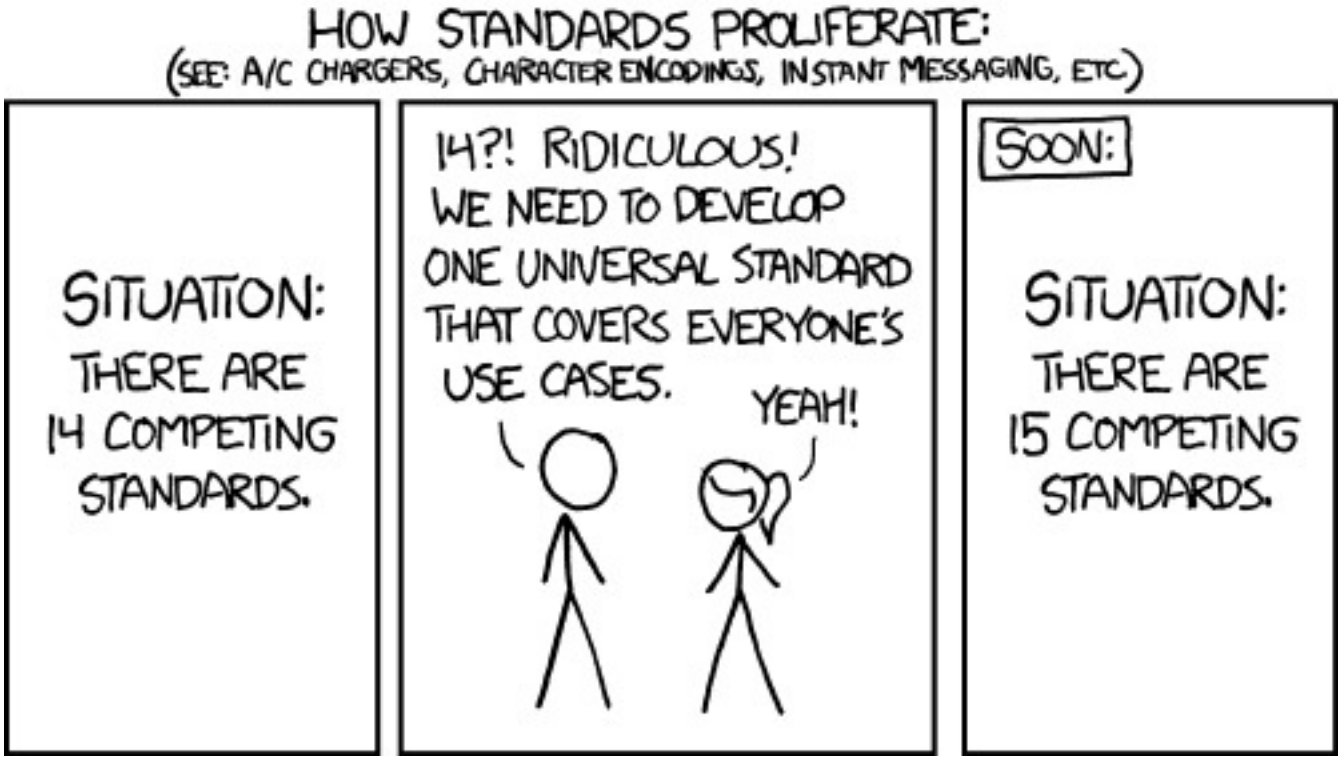
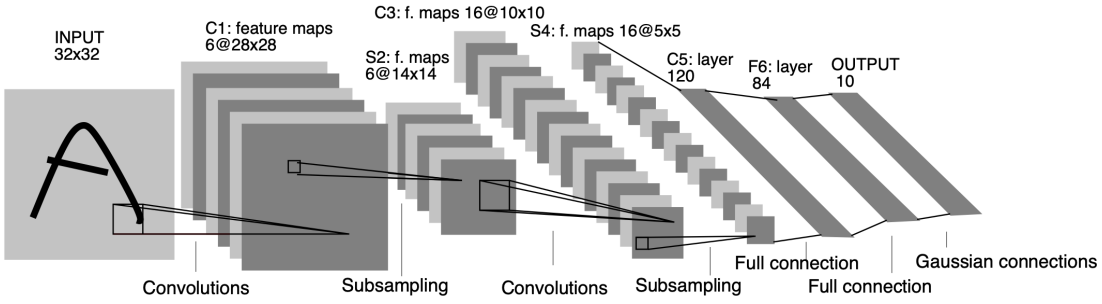


Figure 1: Credits to XKCD webcomics: `https://xkcd.com/927/`

In general, there are many different frameworks and projects out there right now that are working on these problems. When looking at this list[5] of different MPC frameworks, one particular XKCD comic came to mind:

3 Models

3.1 CNN Model



(a) Example of a CNN architecture

It is a simple feed-forward network, that takes the input, feeds it through several layers one after the other, and then finally gives the output. The upper illustration is only to have an idea how a CNN could look like. Our CNN has 2 convolution layers, 2 fully connected layers with a RELU activation function.

- Convolution Layers (Conv2d) get input_layer and output_layer as arguments which are the number of filtered (by 3x3 filter)default value tensors respectively from the previous layer and

on the current_layer. The kernel is the size of the filter we use on the current filter. Stride is the shifting step you take on the data point matrix when you do the entry multiplication of the data point and the filter. The padding is the number of columns you add when a filter is going over the original image. All values are chosen to preserve the layer size.

$$n_{out} = \frac{n_{in} + 2p - k}{s} + 1$$

- MaxPool Layers (MaxPool2d) get the kernel size as an argument, which is again the size of the filter.
- each activation function is a RELU
- Fully Convolutional Layer (Linear) gets the number of nodes as an argument from the previous layer and the number of nodes it currently has.
- We use an Adam optimizer in our case and an adapted learning rate

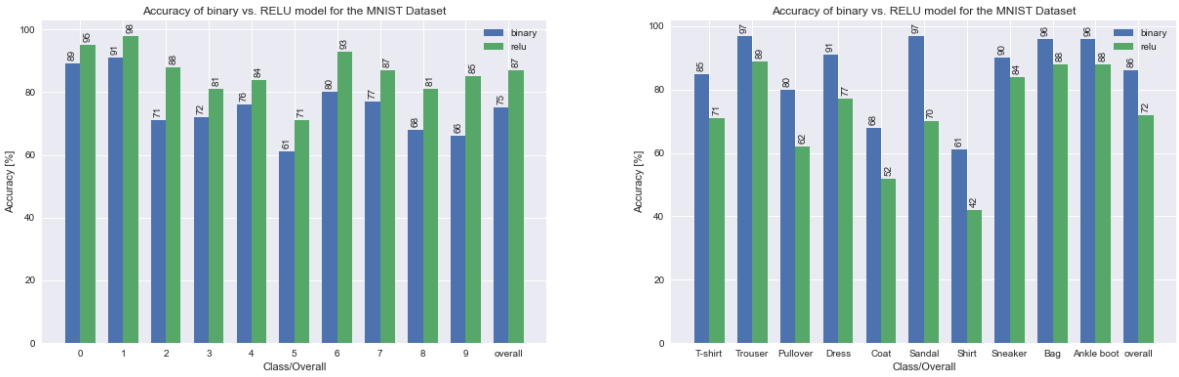
3.2 MLP

In their work Abspoel et. al. used a MLP network with binary sign functions as activators. The network consists of 4 fully connected hidden layers and an output layer of size 10. The result is represented by 10 integers of which the highest is interpreted as the best guess. Due to performance limitations described in Section 5.3 we decided to switch CrypTen as a different framework. Unfortunately the sign function is not fully supported by the framework with respect to multi party computation as it is still in development. After consultation with our supervisor we decided to switch to a network using RELU activators while keeping the rest of the architecture identical.

$$bsgn(z) = \begin{cases} 1 & \text{if } 0 \leq z \\ -1 & \text{otherwise} \end{cases} \quad (1) \qquad relu(z) = max(0, z) \quad (2)$$

3.3 Binary vs. RELU

In Fig. 3 we can see how the usage of different activators affects the models in terms of accuracy.



(a) MNIST

(b) Fashion-MNIST

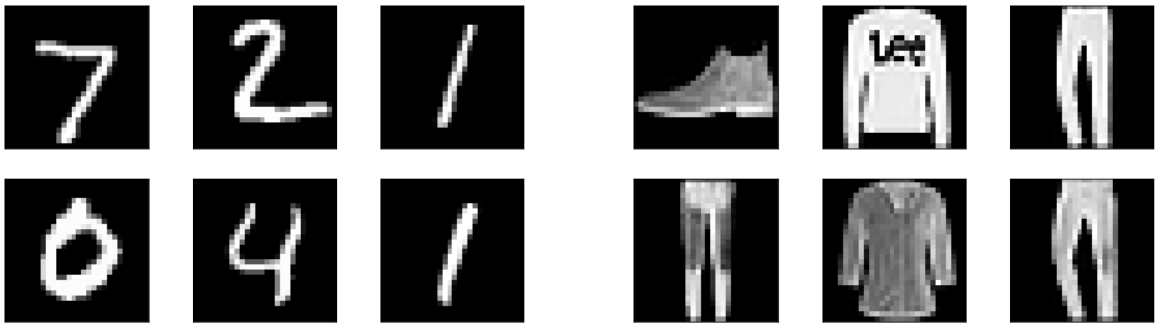
Figure 3

One can clearly see that the choice of activator drastically affects the performance of the model. While the RELU activator works better for the MNIST dataset it performs worse on the fashion dataset. Also, both models struggle with keeping coats and shirts apart. This is propably since

we are using binary thresholding as a transformer which reduces clothes down to their silhouettes. This makes it even for humans hard to distinguish between the two.

4 Datasets

We used two data-sets, first the classical MNIST data-set of handwritten digits[10] and secondly a similar data-set consisting of grayscale images of clothing articles called Fashion MNIST[12][2]. The Fashion MNIST data-set is suppossed to be a direct successor[12] to the classic MNIST dataset and intends to address some issues that researchers noticed with it over the years - in essence it is a more difficult set of images.



(a) MNIST example figures

(b) Fashion example figures

Figure 4: Example images of both datasets side-by-side.

Both data-sets consist of 70000 small square 28×28 pixel gray-scale images, in the case of MNIST of handwritten digits between 0 and 9. The data-set is split into test and training sets, which contain 10000 and 60000 images respectively. Both datasets have 10 different classes.

We used similar transformations on both datasets and normalized the 0...255 greyscale values. For the MNIST digits, we additionally employed a threshold filter to effectively turn them into (binary) black-and-white images. For Fashion dataset, we stuck to the normalized greyscale images.

5 Benchmarks)

The following section provides detailed benchmark results of the tested frameworks. All data concerning time and memory was collected on the same device to allow for a fair comparison. The experiments have been performed on an AMD Ryzen 7 3800X with 8 cores/16 logical processors and 32GB of RAM.

5.1 Environment and Installation

The software environment is based on Ubuntu 18.04 LTS and Python3.7. CrypTen should work with Python versions ¿= 3.7, but we run into some troubles during execution with Python3.8 - running models trained with python 3.7 were unable to be loaded on 3.8 and vice-versa, datasets were unable to be loaded under 3.8 and similar issues. Machine Learning frameworks for python are notoriously susceptible to such problem and this obviously holds true doubly so for frameworks that are still in development. Therefore we very much advise you to use Python 3.7 as well!

[2]The Fashion MNIST dataset is hosted by the fashion retailer Zalando's research division.

How to best install python 3.7 depends very much on your OS:

- MacOS: homebrew
- Linux Ubuntu: Via the deadsnakes-PPA (`https://launchpad.net/~deadsnakes/+archive/ubuntu/ppa`
- If you use pyenv: use that
- Or simply use a docker container

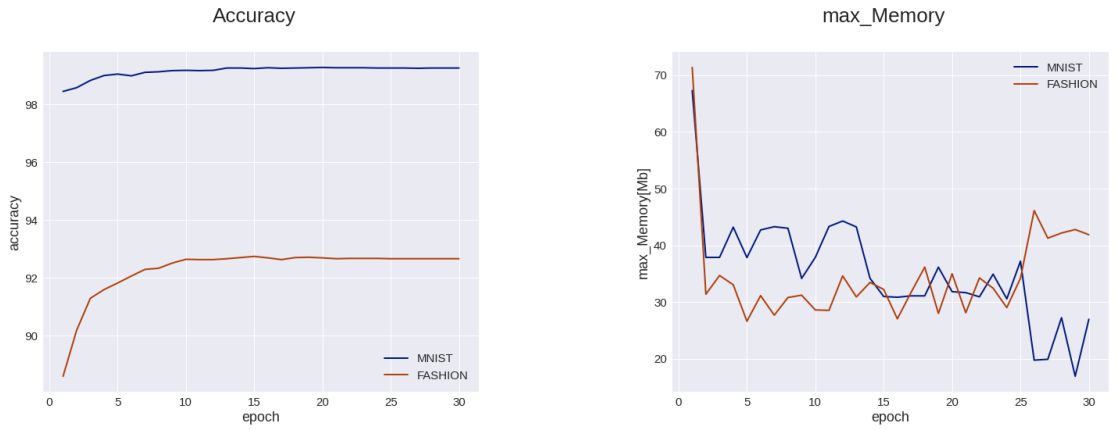
It is very much a matter of taste.

Installation of the required packages is easily done via the provided requirements.txt for pip installation or the Pipfile, if you are using pipenv. The packages are:

```
numpy==1.20.0
pandas
matplotlib
seaborn
jupyterlab
mpyc==0.7
crypten==0.1
torch==1.4.0
tqdm
psutil
ipywidgets
```

5.2 CNN

To have a comparison to the secure multi-party communication we also include a benchmark for the standard CNN model. We chose to train for 30 epochs for our tests and include the efficiency factor calculated $Eff = \frac{totalTime}{accuracy}$ where totalTime is the sum of inference_time and validation_time and accuracy is the amount of percentage of the right classification.



(a) accuracy plot of the CNN

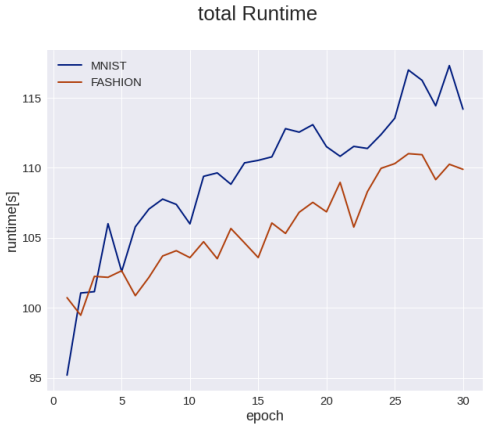
(b) max_Memory usage plot of the CNN

The accuracy is for both data-sets relatively high. For the MNIST data-set the accuracy score stays almost at the same level of around 98.5%. The Fashion data-set needs some epochs to reach a plateau of 92.6%. After one epoch the maximal RAM usage drops to a value of around 35±7 Megabyte for the two data-sets.

(a) inference_time of the CNN

(b) efficiency plot of the CNN

With each epoch the inference time slightly increases for both data-sets. That means for the efficiency plot it will decrease because the accuracy stays the same at a certain number of epoch.



(a) total runtime of the CNN

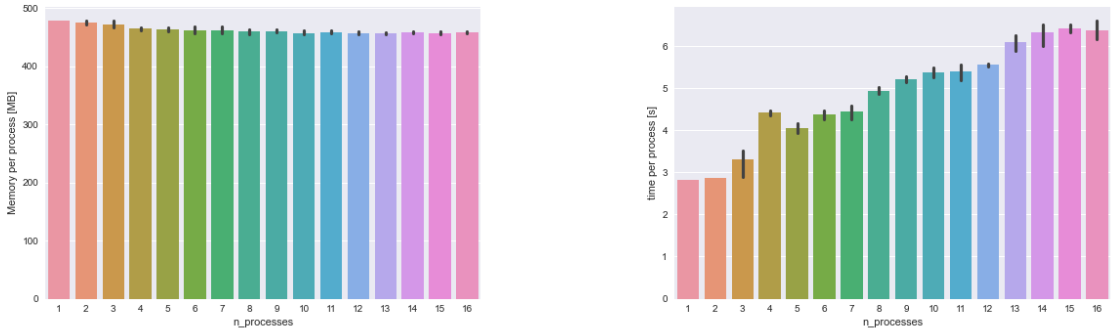
The MNIST-Fashon data-set is the more complex example but the total run-time is smaller than the run-time for the number MNIST data-set.

5.3 MpYC

First we tried to recreate the results of the MpYC benchmark. Since the code for training the model is not provided we can only analyse the evaluation routine.

While their framework can be scaled easily using command line arguments performance does not increase favourably as seen in Fig. 8. Note that the memory usage is reported for each core, i.e. the total memory required for 16 processes is $16 \cdot 450MB = 7.2GB$. Furthermore the runtime increases with the number of participants which is the opposite of what one would want to achieve using multi-party computation.

The benchmark only covers the analysis of just 1 (!) 28x28 pixel image. Training an MLP system would require the analysis of 1000s of images over several epochs making it very impractical for developing models. Furthermore the situation can not be alleviated by adding more processing power as more participants lead to worse performannce. This is also the main reason we decided to switch to another framework for securely training the a network.



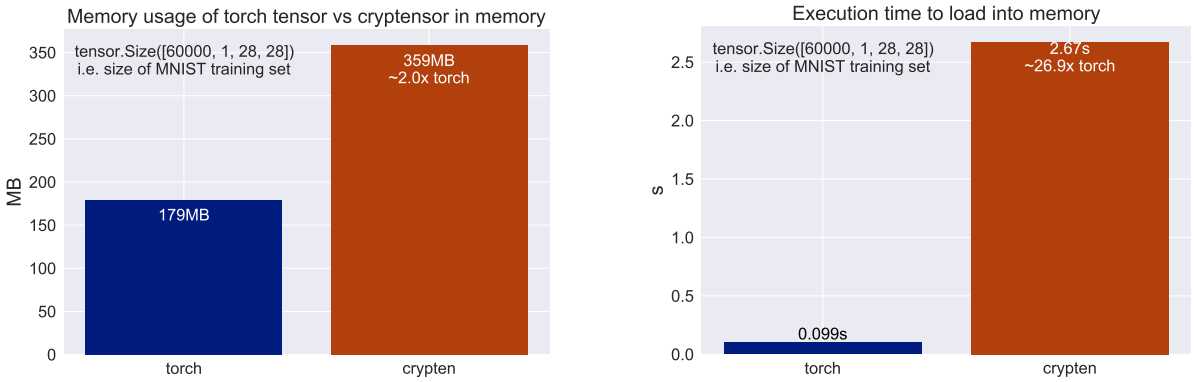
(a) Memory usage per process

(b) time spent evaluating

Figure 8: Benchmark results of the MpYC model

5.4 CrypTen

5.4.1 Torch vs CrypTen Tensors



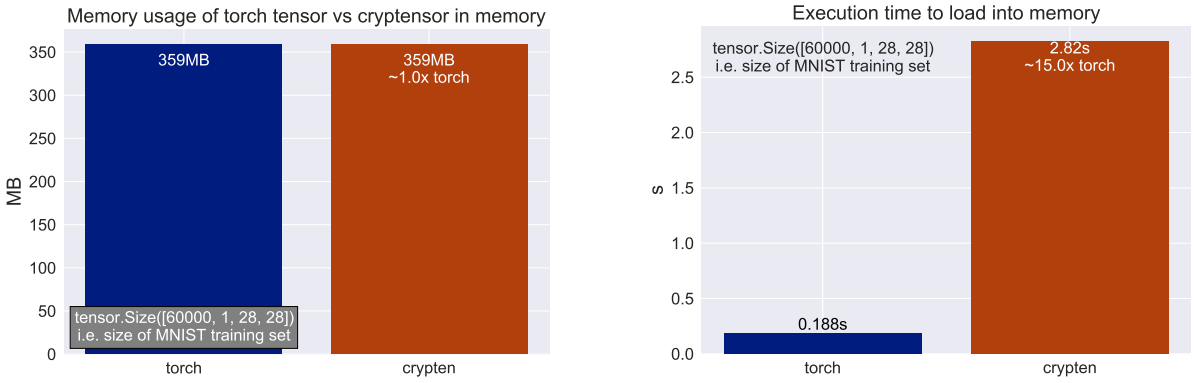
(a) Memory usage

(b) Execution time

Figure 9: Profile of loading a tensor of size [60000, 1, 28, 28] and dtype=float32 (single precision) to memory in Torch vs CrypTen.

We profiled the memory usage and execution time of loading a tensor to memory, the results can be seen in Fig. 9 (float32 tensor) and in Fig. 10 (float64 tensor). When comparing the memory usages it becomes immediatelly clear that CrypTen stores its encrypted tensors in a 64bit (fixed point/integer) format - we could not find any explicit information about this in either the CrypTen documentation[2] or white paper[9]. The encryption or conversion also takes a lot of time. Loading a crypten tensor takes rougly 15-30 times as long as loading a simple torch tensor of the same size. The precision of the torch tensor impacts this, since a torch tensor using less precision (32 vs 64bit) is loaded faster. See Fig. 11 for further reference.

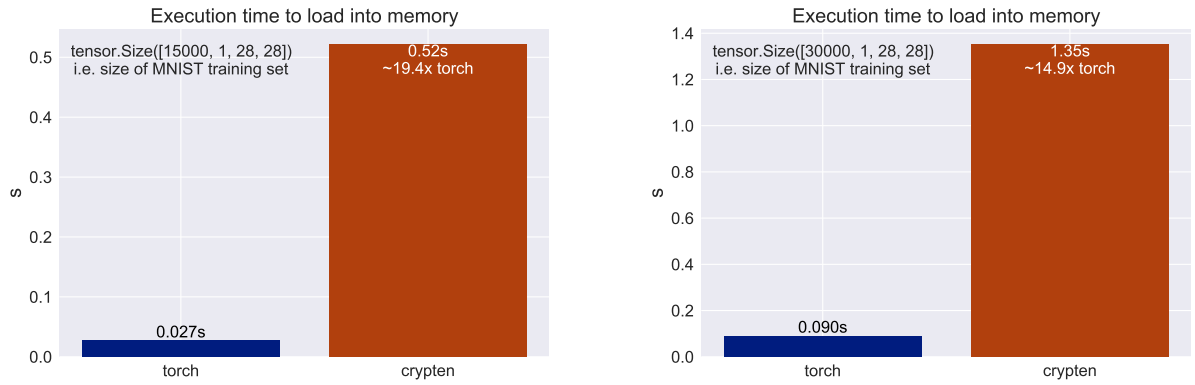
The encryption procedure is obviously costly and will be a major limiting factor in terms of execution speed. We do not expect the accuracy of an encrypted model or of a model that uses encrypted data to suffer (much) though, since the encrypted tensors and models also use 64bit data types. For contrast, MPyC[7] does encode data (and intermediate results) to fixed-point numbers as well, but they are of customizable length[6]. The example that our Binary-MLP benchmarks are based on, see `https://github.com/lschoe/mpyc/blob/master/demos/bnnmnist.py`, use 14bit and 10bit integers depending on the layer.



(a) Memory usage

(b) Execution time

Figure 10: Profile of loading a tensor of size [60000, 1, 28, 28] and dtype=float64 (double precision) to memory in Torch vs CrypTen.



(a) Execution time for a smaller tensor

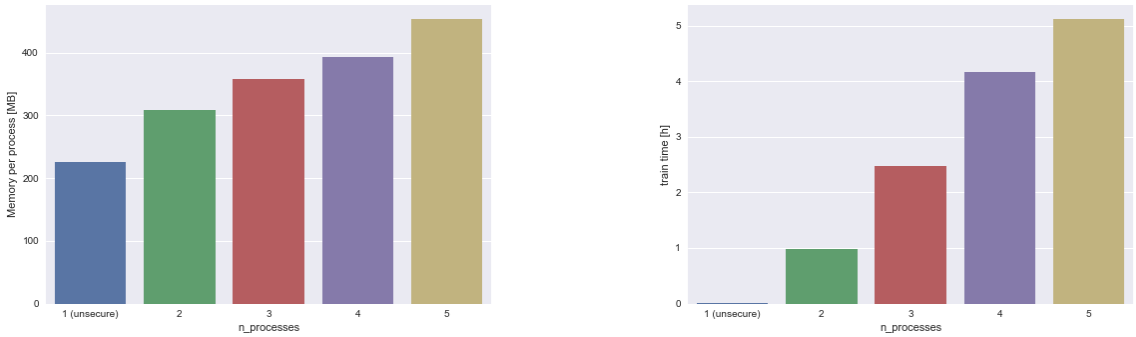
(b) Execution time for a larger tensor

Figure 11: Loading times for two tensors of different sizes and dtype=float64 (double precision)

5.4.2 Training

Fig. 12 shows the benchmark for our securely trained MLP network using CrypTen and the MNIST dataset. In their current version, CrypTen does not offer an increase in performance by adding more participants. On the contrary, every additional process roughly increases the total runtime by about 1 hour (linear time complexity). The increase in time seems to be a bit lower for higher n (number of parties), but more data needs to be collected to be sure. Therefore CrypTen is significantly slower than an unsecure network which takes about 30s to train.

The same applies to RAM usage per processor as well, but the increase is not as pronounced compared to the runtime. Of course, the total RAM usage across all processes/parties will be significantly higher - roughly linear to n once again.



(a) Memory usage per process

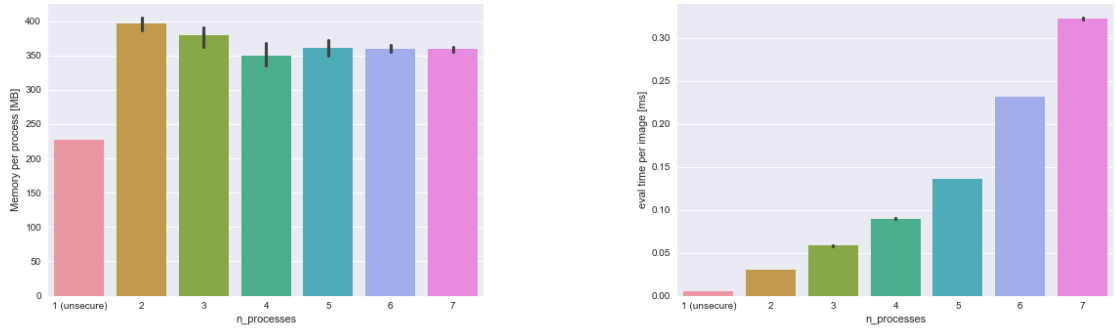
(b) time spent evaluating

Figure 12: Ressource consumption during training. The leftmost bar corresponds to an unsecure network using PyTorch

5.4.3 Predicting

Fig. 13 shows our benchmark during predictions. CrypTen is again significantly slower than an unsecure single processor model. This time the deficit in performance is less pronounced compared to the training times and also significantly faster than MPyC (see Fig. 8 which requires several seconds for one MNIST image. Best results are achieved using 2 processes where the evaluation time is approx. 30ms or 33 frames per second (FPS) which would make real time evaluation theoretically possible.

Our data hints to a quadratic or worse time complexity but more data needs to be collected to be sure. Unfortunately due to an unknown problem we are unable to start more than 7 processes although 8 physical cores are present. We do not know if this is caused by our software or CrypTen. Regardless, 7 (1 model + 6 data holders) seem sufficient for our benchmarks and for most image classification use cases as well.



(a) Memory usage per process

(b) time spent evaluating

Figure 13: Ressource consumption during evaluation (prediction). The leftmost bar corresponds to an unsecure network using PyTorch

6 Summary

6.1 Experiences with CrypTen

As noted before, we had some troubles with the framework. Right now, I would be hesitant to rely on CrypTen in a production setting - it is still considered to be a research project by Facebook after all - although that may change within the next release. One main deal breaker with CrypTen is the fact that you cannot yet save a securely trained network for later usage (to a file) Some other frameworks and projects seem to be further along as well in development at the moment, such as PySyft and Tfencrypted. Unfortunately we only realised that later on in our project after we had already invested a lot of time into our project.

CrypTen supports various forms of data aggregation, but some of these scenarios seem very unlikely to occur for image classification use cases - in theory, a single image could also be split across multiple parties, e.g. the bottom half belongs to A and the top half to B, or one party could hold the image while the other has the label. As they seemed unpractical to us, we focused on a scenario where different parties hold different image-label-pairs and another party holds ownership of the model.

6.2 Secure MPC for Image Classification in general

In general, if you plan to work with secure MPC in machine learning then consider the time investment. Orchestrating the communication and computation architecture behind it, but also simply the time needed for training. The simple and small models (5 layer MLP) we used for our relatively small and simple datasets (28x28 pixel images) already took upwards of 30-60mins for a single epoch of 60000 training images (20% for validation) in our two party MPC secure training. The inference time was relatively serviceable, as 34 images per second ('framerate') can be processed by the model in a two party setting - so a single image takes $0.03s$. As the number of parties increases, the frame rate decreasese, e.g. with 3 parties (two holding data) the framerate drops to about half with 17 images per second and with 7 parties it drops to 3 images per second. Always keep in mind, that these are best case estimations for our consumer grade PCs and running in the synthetic multiprocessing setting.

Due to the much higher runtime of the encrypted training, which also scales dramatically with the number of parties involved, we'd recommend to "pre-train" any model in the clear before moving on to secure MPC training which should work for most image classification use-cases (different parties hold image-label-pairs, usually).

When the model is already trained and purely used for prediction, but the data holder doesn't want to reveal its' data due to whatever reason, secure MPC prediction seems to already deliver usable performance. How well this scales to larger, state-of-the-art image classification (VGG, ResNet, Inception) or object detection models (YOLO, SSD, etc.) is a different questions - keep in mind that we only used single-channel (grayscale) images of resolution 28x28. Real-time performance isn't always needed of course, but usage of computing resources is always costly. This could be a big advantage when using the MPyC library, since it allows you to specify the length of the encoded integers by hand[6] - usually trading off runtime for accuracy - is a very interesting concept, that we really only came to appreciate towards the end of project and therefore did not time left to investigate further.

6.3 MPyC vs. CrypTen

Both frameworks offer powerful capabilities but are also still in (early) development which makes working with them sometimes frustrating. Currently CrypTen is clearly more powerful in terms of evaluating an MLP network, however this could change with future updates. We have also noticed that neither framework fully utilizes the CPU to its maximum potential. Training an unsecure

model using PyTorch leads to 100% CPU usage. This could mean that we are not computationally bound and/or there is still room for improvement.

CrypTen is also beating MPyC in terms of RAM usage but the difference is not as big as with the runtime. However, this difference is also negligible for evaluating MNIST images on a typical PC.

These differences may also be attributed to the fact that MPyC is created by a small team of researchers while CrypTen is developed by Facebook and relies on Torch as a back-end which is an industry-proven and mature framework - although CrypTen itself is still far from being mature itself.

Both frameworks offer different sets of functionality and either one has features that the other has not implemented (yet). However MPyC is better suited as a research and teaching tool as it is much simpler and has less dependencies on other packages. Setting up a virtual envrionment for CrypTen requires about 1.6GB of hardware space. MPyC needs just 60MB and only requires standard libraries - only some of their demos require common libraries such as Numpy or Jupyter.

Considering the potential market of distributed systems (IoT, Cloud Computing, etc.), the increasing use of machine learning in our daily lifes and strict regulations by Governments regarding personal data (i.e. GDPR - General Data Protection Regulation) an efficient framework for secure multi-party processing is needed. Currently neither solution is efficient enough for widespread use. Therefore, it is still anybodies game and time will tell who will come out ahead.

References

[1] URL: https://github.com/facebookresearch/CrypTen/blob/master/tutorials/Introduction.ipynb.

[2] In: URL: https://github.com/facebookresearch/CrypTen.

[3] URL: https://github.com/OpenMined/pysyft.

[4] URL: https://blog.openmined.org/crypten-integration-in-pysyft/.

[5] URL: https://awesomeopensource.com/project/rdragos/awesome-mpc.

[6] URL: https://lschoe.github.io/mpyc/mpyc.sectypes.html.

[7] Mark Abspoel et al. "Fast Secure Comparison for Medium-Sized Integers and Its Application in Binarized Neural Networks". In: *Topics in Cryptology – CT-RSA 2019*. Ed. by Mitsuru Matsui. Cham: Springer International Publishing, 2019, pp. 453–472. ISBN: 978-3-030-12612-4.

[8] Morten Dahl et al. *Private Machine Learning in TensorFlow using Secure Computation*. 2018. arXiv: 1810.08130 [cs.CR]. URL: https://github.com/tf-encrypted/tf-encrypted.

[9] B. Knott et al. "CrypTen: Secure Multi-Party Computation Meets Machine Learning". In: *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*. 2020. URL: https://lvdmaaten.github.io/publications/papers/crypten.pdf.

[10] Yann Lecun. *MNIST: hand-written digit database*. URL: http://yann.lecun.com/exdb/mnist/.

[11] Theo Ryffel et al. *A generic framework for privacy preserving deep learning*. 2018. arXiv: 1811.04017 [cs.LG].

[12] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

7 Appendix

7.1 Neural Net definition

7.1.1 PyTorch

```
import torch.nn as nn

PIXEL_CNT = 28

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(PIXEL_CNT, PIXEL_CNT)
        self.fc2 = nn.Linear(PIXEL_CNT, PIXEL_CNT)
        self.fc3 = nn.Linear(PIXEL_CNT, PIXEL_CNT)
        self.fc4 = nn.Linear(PIXEL_CNT, PIXEL_CNT)
        self.fc5 = nn.Linear(PIXEL_CNT, 10)

    def forward(self, x):
        x = x.view(-1, PIXEL_CNT)
        x = self.fc1(x)
        x = x.relu()
        x = self.fc2(x)
        x = x.relu()
        x = self.fc3(x)
        x = x.relu()
        x = self.fc4(x)
        x = x.relu()
        x = self.fc5(x)
        return x
```

7.1.2 CrypTen

```
import crypten.nn as nn

PIXEL_CNT = 28

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(PIXEL_CNT, PIXEL_CNT)
        self.fc2 = nn.Linear(PIXEL_CNT, PIXEL_CNT)
        self.fc3 = nn.Linear(PIXEL_CNT, PIXEL_CNT)
        self.fc4 = nn.Linear(PIXEL_CNT, PIXEL_CNT)
        self.fc5 = nn.Linear(PIXEL_CNT, 10)

    def forward(self, x):
        x = x.view(-1, PIXEL_CNT)
        x = self.fc1(x)
        x = x.relu()
        x = self.fc2(x)
        x = x.relu()
        x = self.fc3(x)
        x = x.relu()
        x = self.fc4(x)
        x = x.relu()
        x = self.fc5(x)
        return x
```

7.1.3 CNN in PyTorch

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class Net(nn.Module):
    def __init__(self,
                 first_layer_in,
                 first_layer_out):

        self.first_layer_in = first_layer_in
        self.first_layer_out = first_layer_out


        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, self.first_layer_in, 3, 1)
        self.conv2 = nn.Conv2d(self.first_layer_in, self.first_layer_out, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```