tf-encrypted Documentation

Release 0.4.0

The tf-encrypted Contributors

Dec 06, 2018

Getting Started

tf-encrypted is a Python library built on top of TensorFlow for researchers and practitioners to experiment with privacy-preserving machine learning. It provides an interface similar to that of TensorFlow, and aims at making the technology readily available without first becoming an expert in machine learning, cryptography, distributed systems, and high performance computing.

In particular, the library focuses on:

- **Usability**: The API and its underlying design philosophy make it easy to get started, use, and integrate privacy-preserving technology into pre-existing machine learning processes.
- **Extensibility**: The architecture supports and encourages experimentation and benchmarking of new cryptographic protocols and machine learning algorithms.
- **Performance**: Optimizing for tensor-based applications and relying on TensorFlow's backend means runtime performance comparable to that of specialized stand-alone frameworks.
- **Community**: With a primary goal of pushing the technology forward the project encourages collaboration and open source over proprietary and closed solutions.
- **Security**: Cryptographic protocols are evaluated against strong notions of security and [known limitations](#known-limitations) are highlighted.

Checkout the Getting Started guide to learn how to get up and running with private machine learning.

You can view the project source, contribute, and asks questions on GitHub.

CHAPTER 1

License

1.1 Installation

tf-encrypted is available as a package on PyPA and supports Python 3.5+. You can install the package using pip:

```
pip install tf-encrypted
```

A change log is kept for each version of tf-encrypted released which can be found on GitHub.

Once installed, you can import the package into your code as shown below:

```
import tf_encrypted as tfe
```

To learn how to use tf-encrypted to perform encrypted machine learning checkout our Getting Started guide.

1.2 Getting Started

This walkthrough assumes that you have installed *tf-encrypted* by following the installation instructions.

tf-encrypted is a secure multiparty computation library where multiple people (or "*parties*") work together to compute results in a secure fashion without any one party having access to the underlying data. This is achieved by splitting up the input data into shares that are perfectly secure.

1.2.1 Introduction to tf-encrypted's API

tf-encrypted provides an API similar to TensorFlow that data scientists and researchers can use to train models and predict upon them in privacy-preserving fashion.

One of the goals of tf-encrypted is to make experimenting with secure private machine learning accessible to anyone. To do this, we've implemented an API that is very similar to TensorFlow while abstracting away the complexity of securely managing public and private data. The *PondTensor* is the primary abstraction provided for managing public and private data.

The following example demonstrates constructing a public value (known to all parties) using *tfe.define_public_variable*.

```
import numpy as np
import tf_encrypted as tfe

variable = tfe.define_public_variable(np.array([1,2,3]))
print(variable) # PondPublicVariable(shape=(3,))
```

We can then perform operations on these Tensors which define an underlying computation graph which can be executed inside a Session which manages figuring out which nodes run which parts of the computation. This is demonstrated in the following example:

```
variable = tfe.define_public_variable(np.array([1,2,3]))
answer = variable * 2

sess = tfe.Session()
sess.run(tfe.global_variables_initializer(), tag='init') # ignore this for now :)
sess.run(answer)

# => array([2., 4., 6.])
```

Similar to public variables we can define private variables as demonstrated below:

```
variable = tfe.define_private_variable(np.array([1,2,3]))

sess = tfe.Session()
sess.run(tfe.global_variables_initializer(), tag='init')
sess.run([variable.share0, variable.share1])

# => [array([ 1601115100, -2072569751,  -600438257], dtype=int32),
#     array([-1601049564,  2072700823,   600634865], dtype=int32)]
```

Unlike with public tensors, each node involved in a computation will get a different share of the encrypted (private) data. This sharing mechanism is the backbone of multiparty computation.

For more indepth examples of how to use *tf-encrypted* to train and predict upon machine learning models please check out our MNIST or Logistic Regression guies.

If you have any questions, please don't hesitate to reach out via a GitHub Issue.

1.3 MNIST

This tutorial is also available on Google Collab, feel free to follow along there!

In this tutorial, we will train our model in plaintext with Tensorflow, then make private predictions with *tf-encrypted*. we will use the MNIST dataset.

```
from __future__ import absolute_import
import os
import sys
```

(continues on next page)

(continued from previous page)

```
import math
from typing import List, Tuple

import tensorflow as tf
import tf_encrypted as tfe

from tensorflow.keras.datasets import mnist
```

We save the MNIST data in TFRecord format which is the recommended format for TensorFlow. Below are just helper functions to encode and decode the images and the labels in the right format. To build the input pipeline, we use *tf.data.TFRecordDataset*. This object is very handy if we want to chain operations such as normalizing the inputs, generating batches etc.

```
def encode_image(value):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value.tostring()]))


def decode_image(value):
    image = tf.decode_raw(value, tf.uint8)
    image.set_shape((28 * 28))
    return image


def encode_label(value):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))


def decode_label(value):
    return tf.cast(value, tf.int32)


def encode(image, label):
    return tf.train.Example(features=tf.train.Features(feature={
        'image': encode_image(image),
        'label': encode_label(label)
    }))


def decode(serialized_example):
    features = tf.parse_single_example(serialized_example, features={
        'image': tf.FixedLenFeature([], tf.string),
        'label': tf.FixedLenFeature([], tf.int64)
    })
    image = decode_image(features['image'])
    label = decode_label(features['label'])
    return image, label


def normalize(image, label):
    x = tf.cast(image, tf.float32) / 255.
    image = (x - 0.1307) / 0.3081  # image = (x - mean) / std
    return image, label


def get_data_from_tfrecord(filename: str, bs: int) -> Tuple[tf.Tensor, tf.Tensor]:
    return tf.data.TFRecordDataset([filename]) \
```

(continues on next page)

(continued from previous page)

```
                    .map(decode) \
                    .map(normalize) \
                    .repeat() \
                    .batch(bs) \
                    .make_one_shot_iterator()

def save_training_data(images, labels, filename):
    assert images.shape[0] == labels.shape[0]
    num_examples = images.shape[0]

    with tf.python_io.TFRecordWriter(filename) as writer:

        for index in range(num_examples):

            image = images[index]
            label = labels[index]
            example = encode(image, label)
            writer.write(example.SerializeToString())


(x_train, y_train), (x_test, y_test) = mnist.load_data()

data_dir = os.path.expanduser("./data/")
if not os.path.exists(data_dir):
    os.makedirs(data_dir)

save_training_data(x_train, y_train, os.path.join(data_dir, "train.tfrecord"))
save_training_data(x_test, y_test, os.path.join(data_dir, "test.tfrecord"))
```

Below is just an helper function to print tensors in a notebook.

```
# Source: https://stackoverflow.com/questions/37898478/is-there-a-way-to-get-
↪tensorflow-tf-print-output-to-appear-in-jupyter-notebook-o
def tf_print(tensor, transform=None):

   def print_tensor(x):
       print(x if transform is None else transform(x))
       return x

   log_op = tf.py_func(print_tensor, [tensor], [tensor.dtype])[0]
   with tf.control_dependencies([log_op]):
       res = tf.identity(tensor)

   return res
```

1.3.1 Select your cryptography protocol

In this example we use the SecureNN protocol. As for the different parties involved, we here assume a setting with two server, a crypto producer, a weights provider (model-trainer), and a private input provider (prediction-client). Note that we could have selected very easily the Pond protocol by running instead: *tfe.set_protocol(tfe.protocol.Pond(*tfe.get_config().get_players(['server0', 'server1', 'crypto-producer'])))*

```
config = tfe.LocalConfig([
        'server0',
        'server1',
```

(continues on next page)

(continued from previous page)

```
        'crypto-producer',
        'model-trainer',
        'prediction-client'
    ])


tfe.set_config(config)
tfe.set_protocol(tfe.protocol.SecureNN(*tfe.get_config().get_players(['server0',
↪'server1', 'crypto-producer'])))
```

1.3.2 Plaintext Training

Then we create a *ModelTrainer* object which is responsible for training the model in plaintext then provides the weights to perform private predictions.

```
class ModelTrainer():

    BATCH_SIZE = 256
    ITERATIONS = 60000 // BATCH_SIZE
    EPOCHS = 3
    LEARNING_RATE = 3e-3
    IN_N = 28 * 28
    HIDDEN_N = 128
    OUT_N = 10

    def cond(self, i: tf.Tensor, max_iter: tf.Tensor, nb_epochs: tf.Tensor, avg_loss:␣
↪tf.Tensor) -> tf.Tensor:
        is_end_epoch = tf.equal(i % max_iter, 0)
        to_continue = tf.cast(i < max_iter * nb_epochs, tf.bool)

        def true_fn() -> tf.Tensor:
            #tf_print(tensor, transform=None)
            #res = tf_print(avg_loss)
            #return res
            return tf.Print(to_continue, data=[avg_loss], message="avg_loss: ")

        def false_fn() -> tf.Tensor:
            return to_continue

        return tf.cond(is_end_epoch, true_fn, false_fn)

    def build_training_graph(self, training_data) -> List[tf.Tensor]:
        j = self.IN_N
        k = self.HIDDEN_N
        m = self.OUT_N
        r_in = math.sqrt(12 / (j + k))
        r_hid = math.sqrt(12 / (2 * k))
        r_out = math.sqrt(12 / (k + m))

        # model parameters and initial values
        w0 = tf.Variable(tf.random_uniform([j, k], minval=-r_in, maxval=r_in))
        b0 = tf.Variable(tf.zeros([k]))
        w1 = tf.Variable(tf.random_uniform([k, k], minval=-r_hid, maxval=r_hid))
        b1 = tf.Variable(tf.zeros([k]))
        w2 = tf.Variable(tf.random_uniform([k, m], minval=-r_out, maxval=r_out))
```

(continues on next page)

(continued from previous page)

```
        b2 = tf.Variable(tf.zeros([m]))
        params = [w0, b0, w1, b1, w2, b2]

        # optimizer and data pipeline
        optimizer = tf.train.AdamOptimizer(learning_rate=self.LEARNING_RATE)

        # training loop
        def loop_body(i: tf.Tensor, max_iter: tf.Tensor, nb_epochs: tf.Tensor, avg_
↪loss: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
            # get next batch
            x, y = training_data.get_next()

            # model construction
            layer0 = x
            layer1 = tf.nn.relu(tf.matmul(layer0, w0) + b0)
            layer2 = tf.nn.relu(tf.matmul(layer1, w1) + b1)
            predictions = tf.matmul(layer2, w2) + b2

            loss = tf.reduce_mean(tf.losses.sparse_softmax_cross_
↪entropy(logits=predictions, labels=y))

            is_end_epoch = tf.equal(i % max_iter, 0)

            def true_fn() -> tf.Tensor:
                return loss

            def false_fn() -> tf.Tensor:
                return (tf.cast(i - 1, tf.float32) * avg_loss + loss) / tf.cast(i, tf.
↪float32)

            with tf.control_dependencies([optimizer.minimize(loss)]):
                return i + 1, max_iter, nb_epochs, tf.cond(is_end_epoch, true_fn,␣
↪false_fn)

        loop, _, _, _ = tf.while_loop(self.cond, loop_body, [0, self.ITERATIONS, self.
↪EPOCHS, 0.])

        # return model parameters after training
        loop = tf.Print(loop, [], message="Training complete")
        with tf.control_dependencies([loop]):
            return [param.read_value() for param in params]

    def provide_input(self) -> List[tf.Tensor]:
        with tf.name_scope('loading'):
            training_data = get_data_from_tfrecord("./data/train.tfrecord", self.
↪BATCH_SIZE)

        with tf.name_scope('training'):
            parameters = self.build_training_graph(training_data)

        return parameters
```

1.3.3 Private Predictions

The *PredictionClient* object will provide the private input that will be used to make a private prediction.

```
class PredictionClient():

    BATCH_SIZE = 20

    def provide_input(self) -> List[tf.Tensor]:
        with tf.name_scope('loading'):
            prediction_input, expected_result = get_data_from_tfrecord("./data/test.
↪tfrecord", self.BATCH_SIZE).get_next()

        with tf.name_scope('pre-processing'):
            prediction_input = tf.reshape(prediction_input, shape=(self.BATCH_SIZE, 28␣
↪* 28))
            expected_result = tf.reshape(expected_result, shape=(self.BATCH_SIZE,))

        return [prediction_input, expected_result]

    def receive_output(self, likelihoods: tf.Tensor, y_true: tf.Tensor) -> tf.Tensor:
        with tf.name_scope('post-processing'):
            prediction = tf.argmax(likelihoods, axis=1)
            eq_values = tf.equal(prediction, tf.cast(y_true, tf.int64))
            acc = tf.reduce_mean(tf.cast(eq_values, tf.float32))
            op = tf.Print([], [y_true], summarize=self.BATCH_SIZE, message="EXPECT: ")
            op = tf.Print(op, [prediction], summarize=self.BATCH_SIZE, message=
↪"ACTUAL: ")
            op = tf_print(prediction)
            op = tf.Print([op], [acc], summarize=self.BATCH_SIZE, message="Acuraccy: ")
            return op
```

Once you instantiate the *ModelTrainer* and *PredictionClient* objects, you can very easily get the weights trained in plaintext, get the private input from the client and finally make private predictions. As you can see, to create a model, *tf-encrypted* and TensorFlow follow a very similar API

```
layer0 = x
layer1 = tfe.relu((tfe.matmul(layer0, w0) + b0))
layer2 = tfe.relu((tfe.matmul(layer1, w1) + b1))
logits = tfe.matmul(layer2, w2) + b2
```

```
model_trainer = ModelTrainer()
prediction_client = PredictionClient()

# get model parameters as private tensors from model owner
params = tfe.define_private_input('model-trainer', model_trainer.provide_input,␣
↪masked=True)

# we'll use the same parameters for each prediction so we cache them to avoid re-
↪training each time
params = tfe.cache(params)

# get prediction input from client
x, y = tfe.define_private_input('prediction-client', prediction_client.provide_input,␣
↪masked=True)

# compute prediction
w0, b0, w1, b1, w2, b2 = params
layer0 = x
layer1 = tfe.relu((tfe.matmul(layer0, w0) + b0))
layer2 = tfe.relu((tfe.matmul(layer1, w1) + b1))
```

(continues on next page)

(continued from previous page)

```
logits = tfe.matmul(layer2, w2) + b2

# send prediction output back to client
prediction_op = tfe.define_output('prediction-client', [logits, y], prediction_client.
↪receive_output)

with tfe.Session() as sess:
    print("Init")
    sess.run(tf.global_variables_initializer(), tag='init')

    print("Training")
    sess.run(tfe.global_caches_updator(), tag='training')

    for _ in range(5):
        print("Private Predictions:")
        sess.run(prediction_op, tag='prediction')
```

And voila! you have just trained a model in plaintext then made private predictions without revealing anything about the input!

1.4 Logistic Regression

This tutorial is also available on Google Collab, feel free to follow along there!

In this section we will see how to do an easy task, but in secret: Logistic Regression.

Let's go through piece by piece. This section assumes some familiarity with machine learning and TensorFlow.

```
import numpy as np
import tensorflow as tf
import tf_encrypted as tfe

from data import gen_training_input, gen_test_input

tf.set_random_seed(1)

# Parameters
learning_rate = 0.01
training_set_size = 2000
test_set_size = 100
training_epochs = 10
batch_size = 100
nb_feats = 10

xp, yp = tfe.define_private_input('input-provider', lambda: gen_training_
↪input(training_set_size, nb_feats, batch_size))
xp_test, yp_test = tfe.define_private_input('input-provider', lambda: gen_test_
↪input(training_set_size, nb_feats, batch_size))

W = tfe.define_private_variable(tf.random_uniform([nb_feats, 1], -0.01, 0.01))
b = tfe.define_private_variable(tf.zeros([1]))
```

There is nothing here that should be too unfamiliar except the last four lines.

```
xp, yp = tfe.define_private_input('input-provider', lambda: gen_training_
↪input(training_set_size, nb_feats, batch_size))
xp_test, yp_test = tfe.define_private_input('input-provider', lambda: gen_test_
↪input(training_set_size, nb_feats, batch_size))
```

This code creates two nodes in the tf graph that represent where private data & labels will enter the computation. See full code below of the *gen* methods.

```
W = tfe.define_private_variable(tf.random_uniform([nb_feats, 1], -0.01, 0.01))
b = tfe.define_private_variable(tf.zeros([1]))
```

W and *b* represent the *weights* and *bias* of a classical neural network. This network will train the *weight* and *bias* to learn how to predict the generated sample data.

Next, we will declare how the model learns

```
out = tfe.matmul(xp, W) + b
pred = tfe.sigmoid(out)
```

and the backprop

```
dc_dout = pred - yp
dW = tfe.matmul(tfe.transpose(xp), dc_dout) * (1 / batch_size)
db = tfe.reduce_sum(1. * dc_dout, axis=0) * (1 / batch_size)
ops = [
    tfe.assign(W, W - dW * learning_rate),
    tfe.assign(b, b - db * learning_rate)
]
```

To test the model

```
pred_test = tfe.sigmoid(tfe.matmul(xp_test, W) + b)
```

Finally, we can run our training loop

```
def print_accuracy(pred_test_tf, y_test_tf: tf.Tensor) -> tf.Operation:
    correct_prediction = tf.equal(tf.round(pred_test_tf), y_test_tf)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    return tf.Print(accuracy, data=[accuracy], message="Accuracy: ")


print_acc_op = tfe.define_output('input-provider', [pred_test, yp_test], print_
↪accuracy)

total_batch = training_set_size // batch_size
with tfe.Session() as sess:
    sess.run(tfe.global_variables_initializer(), tag='init')

    for epoch in range(training_epochs):
        avg_cost = 0.

        for i in range(total_batch):
            _, y_out, p_out = sess.run([ops, yp.reveal(), pred.reveal()], tag=
↪'optimize')
```

(continues on next page)

(continued from previous page)

```
            # Our sigmoid function is an approximation
            # it can have values outside of the range [0, 1], we remove them and add/
↪substract an epsilon to compute the cost
            p_out = p_out * (p_out > 0) + 0.001
            p_out = p_out * (p_out < 1) + (p_out >= 1) * 0.999
            c = -np.mean(y_out * np.log(p_out) + (1 - y_out) * np.log(1 - p_out))
            avg_cost += c / total_batch

        print("Epoch:", '%04d' % (epoch + 1), "cost=", "{:.9f}".format(avg_cost))

    print("Optimization Finished!")

    sess.run(print_acc_op)
```

You have just done private training without revealing anything about the input!

1.5 *protocol*

In *tf-encrypted*, a protocol represents a certain type of cryptographic protocol to achieve security.

The goal is to allow you to easily play with or use different cryptographic methods by simply changing the protocol.

```
import tf_encrypted as tfe

tfe.set_protocol(tfe.protocol.SecureNN())
tfe.set_protocol(tfe.protocol.Pond())
```

1.5.1 *Pond*

Pond is an implementation of the SPDZ algorithm for MPC.

PondTensor

PondTensor is the tensor type you will interact with most.

Generally, you will never instantiate a tensor directly, rather you create them through *protocol*.

```
pondPrivateTensor = prot.define_private_variable(np.array([1,2,3,4]))
pondPublicTensor = prot.define_private_variable(np.array([1,2,3,4]))
```

PondPrivateTensor

PondPublicTensor

1.5.2 *SecureNN*

SecureNN is an implementation from the SecureNN paper. SecureNN is an extension of the *Pond* protocol. ie *SecureNN* is a superset of the *SPDZ* protocol. The main difference between *SecureNN* and *SPDZ* is exact `Relu` and *Maxpooling* layers. In SPDZ, *Maxpooling* is simply not supported, and `Relu` will be approximated.

Approximation can be quicker in some cases but it will break down when inputs are sufficiently large. This requires users to implement workaround techniques such as adding a `Batchnorm` layer before a `Relu`.

1.6 *Session*

Session is an extension of *tf.Session* that lets the graph run in a secure manner.

The aim of *tf-encrypted* is to look as close to *TensorFlow* as possible. With this goal in mind, you get and use a session the same way that you're used to with Tensorflow:

```
import tf_encrypted as tfe

with tfe.Session() as sess:
  # sess.run like normal
```

See also the official TensorFlow docs on Session.

1.7 *Config*

Config determines how a session should run in tf-encrypted.
There are two primary ways in which config can be used:

class tf_encrypted.config.**LocalConfig**

and

class tf_encrypted.config.**RemoteConfig**

As the name implies, *LocalConfig* is used to create a local session. This is useful for quick debugging and prototyping.

RemoteConfig is more robust and is used to specify how a graph will run in a production environment. What machines are on which host, etc.

See class definitions for usage examples and more.

1.7.1 *LocalConfig*

1.7.2 *RemoteConfig*

1.8 *layers*

Layers implements ready to use *neural network* layers that come with crypto for free.

This is similar to TensorFlow's tf.nn module.

1.8.1 *Dense*

1.8.2 *Convolution*

1.8.3 *AveragePooling2D*

1.8.4 *MaxPooling2D*

1.8.5 *Batchnorm*

1.8.6 *Sigmoid*

1.8.7 *Relu*

Classicly, *Relu* computes the following on input:

```
Relu(x) = max(0, x)
```

In *tf-encrypted*, how *Relu* behaves will depend on the underlying protocol you are using.

With `Pond`, *Relu* will be approximated using Chebyshev Polynomial Approximation
With `SecureNN`, *Relu* will behave as you expect (*Relu(x) = max(0, x)*)

1.8.8 *Tanh*

1.8.9 *Reshape*