tf-encrypted Documentation

Release 0.4.0

The tf-encrypted Contributors

Mar 01, 2019

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

This project is licensed under the Apache License, Version 2.0 (see License). Copyright as specified in the NOTICE contained in the code base.

1.1 Installation

tf-encrypted is available as a package on PyPA and supports Python 3.5+ and tensorflow 1.12.0+. 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
            tf.print(to_continue, data=[avg_loss], message="avg_loss: ")
            return to_continue

        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]))
```

(continues on next page)

(continued from previous page)

```
        w2 = tf.Variable(tf.random_uniform([k, m], minval=-r_out, maxval=r_out))
        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
        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))
            tf.print([], [y_true], summarize=self.BATCH_SIZE, message="EXPECT: ")
            op=[]
            tf.print(op, [prediction], summarize=self.BATCH_SIZE, message="ACTUAL: ")
            op=prediction
            tf_print(prediction)
            op = [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
```

(continues on next page)

(continued from previous page)

```
layer1 = tfe.relu((tfe.matmul(layer0, w0) + b0))
layer2 = tfe.relu((tfe.matmul(layer1, w1) + b1))
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))
    tf.print(accuracy, data=[accuracy], message="Accuracy: ")
    return 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())
```

class tf_encrypted.protocol.protocol.**Protocol**
Protocol is the base class that other protocols in TF Encrypted will extend from.

Do not directly instantiate this class. You should use a subclass instead, such as `SecureNN` or `Pond`

tf_encrypted.protocol.protocol.**get_protocol**() → Protocol or None
Returns the current global protocol.

tf_encrypted.protocol.protocol.**global_caches_updater**() → tensorflow.Operation
Groups all ops that have been instantiated with a memoize decoratored function into a single tf.Operation.

tf_encrypted.protocol.protocol.**memoize**(*func*) → Callable
Decorates a function for memoization, which explicitly caches the function's output.

Parameters **func** (*Callable*) – The function to memoize

tf_encrypted.protocol.protocol.**set_protocol**(*prot*)
Sets the global protocol. E.g. `SecureNN` or `Pond`.

Parameters **prot** (*Protocol*) – An instance of a tfe protocol.

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]))
```

class tf_encrypted.protocol.pond.**PondTensor**(*prot*, *is_scaled*)

This class functions mostly as a convenient way of exposing operations directly on the various tensor objects, ie allowing one to write *x + y* instead of *prot.add(x, y)*. Since this functionality is shared among all tensors we put it in this superclass.

This class should never be instantiated on its own. Instead you should use your chosen protocols factory methods:

```
x = prot.define_private_input(tf.constant(np.array([1,2,3,4])))
y = prot.define_public_input(tf.constant(np.array([4,5,6,7])))

z = x + y

with config.Session() as sess:
    answer = z.reveal().eval(sess)

    print(answer) # => [5, 7, 9, 11]
```

add(*other*)

Add *other* to this PondTensor. This can be another tensor with the same backing or a primitive.

This function returns a new PondTensor and does not modify this one.

Parameters **other** (PondTensor) – a or primitive (e.g. a float)

Returns A new PondTensor with *other* added.

Return type *PondTensor*

dot(*other*)

Alias for matmul()

Returns A new PondTensor

Return type *PondTensor*

expand_dims()

See tf.expand_dims

Returns A new PondTensor

Return type *PondTensor*

matmul(*other*)

MatMul this tensor with *other*. This will perform matrix multiplication, rather than elementwise like mul()

Parameters **other** (PondTensor) – to subtract

Returns A new PondTensor

Return type *PondTensor*

mul(*other*)

Multiply this tensor with *other*

Parameters **other** (PondTensor) – to multiply

Returns A new PondTensor

Return type *PondTensor*

reduce_max(*axis: int*) → tf_encrypted.protocol.pond.PondTensor

See tf.reduce_max

Parameters **axis** (*int*) – The axis to take the max along

Return type *PondTensor*

Returns A new pond tensor with the max value from each axis.

reduce_sum(*axis=None*, *keepdims=None*)
Like `tensorflow.reduce_sum()`

Parameters

- **axis** (*int*) – The axis to reduce along
- **keepdims** (*bool*) – If true, retains reduced dimensions with length 1.

Returns A new PondTensor

Return type *PondTensor*

reshape(*shape: List[int]*) → tf_encrypted.protocol.pond.PondTensor

See tf.reshape

Parameters **shape** (*List[int]*) – The new shape of the tensor.

Return type *PondTensor*

Returns A new tensor with the contents of this tensor, but with the new specified shape.

shape

Return type List[int]

Returns The shape of this tensor.

square()
Square this tensor.

Returns A new PondTensor

Return type *PondTensor*

sub(*other*)
Subtract *other* from this tensor.

Parameters **other** (PondTensor) – to subtract

Returns A new PondTensor

Return type *PondTensor*

sum(*axis=None*, *keepdims=None*)
See *PondTensor.reduce_sum()*

transpose(*perm=None*)
Transpose this tensor.

See `tensorflow.transpose()`

Parameters **List[int]** – A permutation of the dimensions of this tensor.

Returns A new PondTensor

Return type *PondTensor*

truncate()
Truncate this tensor.

TODO

Returns A new PondTensor

Return type *PondTensor*

PondPrivateTensor

class tf_encrypted.protocol.pond.**PondPrivateTensor**(*prot: tf_encrypted.protocol.pond.Pond*, *share0: tf_encrypted.tensor.factory.AbstractTensor*, *share1: tf_encrypted.tensor.factory.AbstractTensor*, *is_scaled: bool*)

This class represents a private value that may be unknown to everyone.

shape

Return type List[int]

Returns The shape of this tensor.

unwrapped
Unwrap the tensor.

This will return the shares for each of the parties that collectively own the tensor.

```
x_0, y_0 = tensor.unwrapped
# x_0 == private shares of the value pinned to player_0's device.
# y_0 == private shares of the value pinned to player_1's device.
```

In most cases you will not need to use this method. All funtions will hide this functionality for you (e.g. *add*, *mul*, etc).

PondPublicTensor

class tf_encrypted.protocol.pond.**PondPublicTensor**(*prot: tf_encrypted.protocol.pond.Pond*, *value_on_0: tf_encrypted.tensor.factory.AbstractTensor*, *value_on_1: tf_encrypted.tensor.factory.AbstractTensor*, *is_scaled: bool*)

This class represents a public tensor, known by at least the two servers but potentially known by more. Although there is only a single value we replicate it on both servers to avoid sending it from one to the other in the operations where it's needed by both (eg multiplication).

shape

Return type List[int]

Returns The shape of this tensor.

unwrapped
Unwrap the tensor.

This will return the value for each of the parties that collectively own the tensor.

In most cases, this will be the same value on each device.

```
x_0, y_0 = tensor.unwrapped
# x_0 == 10 with the value pinned to player_0's device.
# y_0 == 10 with the value pinned to player_1's device.
```

In most cases you will want to work on this data on the specified device.

```
x_0, y_0 = tensor.unwrapped

with tf.device(prot.player_0.device_name):
    # act on x_0

with tf.device(prot.player_1.device_name):
    # act on y_0
```

In most cases you will not need to use this method. All funtions will hide this functionality for you (e.g. *add*, *mul*, etc).

class tf_encrypted.protocol.pond.**Pond**(*server_0*, *server_1*, *crypto_producer*, *tensor_factory*, *fixedpoint_config*)

Pond is similar to SPDZ except it has been vectorized plus a few more optimizations.

Pond works with 2 parties for computation and one crypto producer for triples.

Parameters

- **server_0** (*Player*) – The "alice" of MPC.
- **server_1** (*Player*) – The "bob" of MPC.
- **crypto_producer** (*Player*) – The host to act as the crypto producer. In *Pond* this party is responsible for producing triples to aid in computation.
- **tensor_factory** (*AbstractFactory*) – Which backing type of tensor you would like to use, e.g. *int100* or *int64*

add(*x*, *y*) → PondTensor

Adds two tensors *x* and *y*.

Parameters

- **x** (PondTensor) – The first operand.
- **y** (PondTensor) – The second operand.

define_constant(*value*, *apply_scaling*, *name*, *factory*) → PondConstant

Define a constant to use in computation.

```
x = prot.define_constant(np.array([1,2,3,4]), apply_scaling=False)
```

See tf.constant

Parameters

- **value** (*np.ndarray*) – The value to define as a constant.
- **apply_scaling** (*bool*) – Whether or not to scale the value.
- **name** (*str*) – What name to give to this node in the graph.
- **factory** (*AbstractFactory*) – Which tensor type to represent this value with.

define_output(*player*, *xs*, *outputter_fn*, *name*) → tensorflow.Operation
Define an output for this graph.

Parameters **player** (*Union[str,Player]*) – Which player/device this output will be sent to.

define_private_input(*player*, *inputter_fn*, *apply_scaling*, *name*, *masked*, *factory) -> PondPrivateTensor(s*)
Define a private input.

This represents a *private* input owned by the specified player into the graph.

Parameters

- **player** (*Union[str,Player]*) – Which player owns this input.
- **apply_scaling** (*bool*) – Whether or not to scale the value.
- **name** (*str*) – What name to give to this node in the graph.
- **masked** (*bool*) – Whether or not to mask the input.
- **factory** (*AbstractFactory*) – Which backing type to use for this input (e.g. *int100* or *int64*).

define_private_placeholder(*shape*, *apply_scaling*, *name*, *factory*) → PondPrivatePlaceholder
Define a *private* placeholder to use in computation. This will only be known by the party that defines it.

```
x = prot.define_private_placeholder(shape=(1024, 1024))
```

See tf.placeholder

Parameters

- **shape** (*List[int]*) – The shape of the placeholder.
- **apply_scaling** (*bool*) – Whether or not to scale the value.
- **name** (*str*) – What name to give to this node in the graph.
- **factory** (*AbstractFactory*) – Which tensor type to represent this value with.

define_private_variable(*initial_value*, *apply_scaling*, *name*, *factory*) → PondPrivateVariable
Define a private variable.

This will take the passed value and construct shares that will be split up between those involved in the computationself.

For example, in a two party architecture, this will split the value into two sets of shares and transfer them between each party in a secure manner.

:see tf.Variable

Parameters

- **initial_value** (*Union[np.ndarray,tf.Tensor,PondPublicTensor]*) – The initial value.
- **apply_scaling** (*bool*) – Whether or not to scale the value.
- **name** (*str*) – What name to give to this node in the graph.
- **factory** (*AbstractFactory*) – Which tensor type to represent this value with.

define_public_input (*player*, *inputter_fn*, *apply_scaling*, *name) -> PondPublicTensor(s*)
Define a public input.

This represents a *public* input owned by the specified player into the graph.

Parameters

- **player** (*Union[str,Player]*) – Which player owns this input.
- **apply_scaling** (*bool*) – Whether or not to scale the value.
- **name** (*str*) – What name to give to this node in the graph.

define_public_placeholder (*shape*, *apply_scaling*, *name*, *factory*) → PondPublicPlaceholder
Define a *public* placeholder to use in computation. This will be known to both parties.

```
x = prot.define_public_placeholder(shape=(1024, 1024))
```

See tf.placeholder

Parameters

- **shape** (*List[int]*) – The shape of the placeholder.
- **apply_scaling** (*bool*) – Whether or not to scale the value.
- **name** (*str*) – What name to give to this node in the graph.
- **factory** (*AbstractFactory*) – Which tensor type to represent this value with.

define_public_variable (*initial_value*, *apply_scaling*, *name*, *factory*) → PondPublicVariable
Define a public variable.

This is like defining a variable in tensorflow except it creates one that can be used by the protocol.

For most cases, you can think of this as the same as the one from tensorflow and you don't generally need to consider the difference.

For those curious, under the hood, the major difference is that this function will pin your data to a specific device which will be used to optimize the graph later on.

See tf.Variable

Parameters

- **initial_value** (*Union[np.ndarray,tf.Tensor,PondPublicTensor]*) – The initial value.
- **apply_scaling** (*bool*) – Whether or not to scale the value.
- **name** (*str*) – What name to give to this node in the graph.
- **factory** (*AbstractFactory*) – Which tensor type to represent this value with.

div (*x*, *y*)
Performs a true division of *x* by *y* where *y* is public.

No flooring is performing if *y* is an integer type as it is implicitly treated as a float.

lift (*x*, *y=None*, *apply_scaling=None) -> PondTensor(s*)
Convenience method for working with mixed typed tensors in programs: combining any of the Pond objects together with e.g. ints and floats will automatically lift the latter into Pond objects.

Parameters

- **x** (*int,float,PondTensor*) – Python object to lift.

- **y** (*int*, *float*, PondTensor) – Second Python object to lift, optional.
- **apply_scaling** (*bool*) – Whether to apply scaling to the input object(s).

reshape(*x*, *shape*) → PondTensor
Reshape *x* into a tensor with a new *shape*.

Parameters

- **x** (PondTensor) – Input tensor.
- **shape** ((*int*, ..)) – Shape of output tensor.

strided_slice(*x*, **args*, ***kwargs*) → PondTensor
See https://www.tensorflow.org/api_docs/python/tf/strided_slice for further documentation.

transpose(*x*, *perm=None*) → PondTensor
Transposes the input *x*, or permutes the axes of *x* if *perm* is given.

Parameters

- **x** (PondTensor) – The tensor to transpose or permute.
- **perm** (*List*) – A permutation of axis indices.

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*.

class tf_encrypted.protocol.securenn.**SecureNN**(*server_0*, *server_1*, *server_2*, *prime_factory*, *odd_factory*, ***kwargs*)
Implementation of SecureNN from Wagh et al.

argmax(*x*, *axis*) → PondTensor
Find the index of the max value along an axis.

```
>>> argmax([[10, 20, 30], [11, 13, 12], [15, 16, 17]], axis=0)
[[2], [1], [2]]
```

See tf.argmax

Parameters

- **x** (PondTensor) – Input tensor.
- **axis** (*int*) – The tensor axis to reduce along.

Return type *PondTensor*

Returns A new tensor with the indices of the max values along specified axis.

bits(*x*, *factory*) → PondPublicTensor
Convert a fixed-point precision tensor into its bitwise representation.

Parameters **x** (PondPublicTensor) – A fixed-point tensor to extract into a bitwise representation.

bitwise_and(*x*, *y*) → PondTensor
Computes the bitwise *AND* of the given inputs, .

Parameters

- **x** (PondTensor) – Input tensor.
- **y** (PondTensor) – Input tensor.

bitwise_not(*x*) → PondTensor
Computes the bitwise *NOT* of the input, i.e. .

Parameters **x** (PondTensor) – Input tensor.

bitwise_or(*x*, *y*) → PondTensor
Computes the bitwise *OR* of the given inputs, .

Parameters

- **x** (PondTensor) – Input tensor.
- **y** (PondTensor) – Input tensor.

bitwise_xor(*x*, *y*) → PondTensor
Compute the bitwise *XOR* of the given inputs,

Parameters

- **x** (PondTensor) – Input tensor.
- **y** (PondTensor) – Input tensor.

equal_zero(*x*, *dtype*) → PondTensor
Evaluates the Boolean expression .

```
>>> equal_zero([1,0,1])
[0, 1, 0]
```

Parameters

- **x** (PondTensor) – The tensor to evaluate.
- **dtype** (*AbstractFactory*) – An optional tensor factory, defaults to dtype of *x*.

greater(*x*, *y*) → PondTensor
Returns .

```
>>> greater([1,2,3], [0,1,5])
[1, 1, 0]
```

Parameters

- **x** (PondTensor) – The tensor to check.
- **y** (PondTensor) – The tensor to check against.

greater_equal(*x*, *y*) → PondTensor
Returns .

```
>>> greater_equal([1,2,3], [0,1,3])
[1, 1, 1]
```

Parameters

- **x** (PondTensor) – The tensor to check.
- **y** (PondTensor) – The tensor to check against.

less (*x*, *y*) → PondTensor
Returns .

```
>>> less([1,2,3], [0,1,5])
[0, 0, 1]
```

Parameters

- **x** (PondTensor) – The tensor to check.
- **y** (PondTensor) – The tensor to check against.

less_equal (*x*, *y*) → PondTensor
Returns .

```
>>> less_equal([1,2,3], [0,1,3])
[0, 0, 1]
```

Parameters

- **x** (PondTensor) – The tensor to check.
- **y** (PondTensor) – The tensor to check against.

lsb (*x*) → PondTensor
Computes the least significant bit of the provided tensor.

Parameters **x** (PondTensor) – The tensor to take the least significant bit of.

maximum (*x*, *y*) → PondTensor
Computes .

Returns the greater value of each tensor per index.

```
>>> maximum([10, 20, 30], [11, 19, 31])
[11, 20, 31]
```

Parameters

- **x** (PondTensor) – Input tensor.
- **y** (PondTensor) – Input tensor.

maxpool2d (*x*, *pool_size*, *strides*, *padding*) → PondTensor
Performs a *MaxPooling2d* operation on *x*.

Parameters

- **x** (PondTensor) – Input tensor.
- **pool_size** (*List[int]*) – The size of the pool.
- **strides** (*List[int]*) – A list describing how to stride over the convolution.
- **padding** (*str*) – Which type of padding to use (“SAME” or “VALID”).

msb (*x*) → PondTensor
Computes the most significant bit of the provided tensor.

Parameters **x** (PondTensor) – The tensor to take the most significant bit of

negative (*x: tf_encrypted.protocol.pond.PondTensor*) → tf_encrypted.protocol.pond.PondTensor
Returns .

```
>>> negative([-1, 0, 1])
[1, 0, 0]
```

Parameters **x** (PondTensor) – The tensor to check.

non_negative (*x*) → PondTensor
Returns .

```
>>> non_negative([-1, 0, 1])
[0, 1, 1]
```

Note this is the derivative of the ReLU function.

Parameters **x** (PondTensor) – The tensor to check.

reduce_max (*x*, *axis*) → PondTensor
Find the max value along an axis.

```
>>> reduce_max([[10, 20, 30], [11, 13, 12], [15, 16, 17]], axis=0)
[[30], [13], [17]]
```

See tf.reduce_max

Parameters

- **x** (PondTensor) – Input tensor.
- **axis** (*int*) – The tensor axis to reduce along.

Return type *PondTensor*

Returns A new tensor with the specified axis reduced to the max value in that axis.

relu (*x*) → PondTensor
Returns the exact *ReLU* by computing *ReLU(x) = x * nonnegative(x)*.

```
>>> relu([-12, -3, 1, 3, 3])
[0, 0, 1, 3, 3]
```

Parameters **x** (PondTensor) – Input tensor.

select (*choice_bit*, *x*, *y*) → PondTensor
The *select* protocol from Wagh et al. Secretly selects and returns elements from two candidate tensors.

```
>>> option_x = [10, 20, 30, 40]
>>> option_y = [1, 2, 3, 4]
>>> select(choice_bit=1, x=option_x, y=option_y)
[1, 2, 3, 4]
>>> select(choice_bit=[0,1,0,1], x=option_x, y=option_y)
[10, 2, 30, 4]
```

NOTE: Inputs to this function in real use will not look like above. In practice these will be secret shares.

Parameters

- **choice_bit** (PondTensor) – The bits representing which tensor to choose. If *choice_bit = 0* then choose elements from *x*, otherwise choose from *y*.
- **x** (PondTensor) – Candidate tensor 0.
- **y** (PondTensor) – Candidate tensor 1.

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.

class tf_encrypted.session.**Session**(*graph=None*, *config=None*, *target=None*)

Wrap a Tensorflow Session.

See the documentation of tf.Session for more details.

Parameters

- **graph** (*Optional[tf.Graph]*) – A `tf.Graph`. Used in the same as in tensorflow. This is the graph to be launched. If nothing is specified then the default session graph will be used.
- **config** (*Optional[Config]*) – A `Local` or `Remote` config to be used to execute the graph.

run(*fetches*, *feed_dict*, *tag*, *write_trace*) → Any

See the documentation for tf.Session.run for more details.

This method functions just as the one from tensorflow.

The value returned by run() has the same shape as the fetches argument, where the leaves are replaced by the corresponding values returned by TensorFlow.

Parameters

- **fetches** (*Any*) – A single graph element, a list of graph elements, or a dictionary whose values are graph elements or lists of graph elements (described in tf.Session.run docs).
- **feed_dict** (*str->np.ndarray*) – A dictionary that maps graph elements to values (described in tf.Session.run docs).
- **tag** (*str*) – An optional namespace to run the session under.
- **write_trace** (*bool*) – If true, the session logs will be dumped for use in Tensorboard.

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*

class tf_encrypted.config.**LocalConfig**(*player_names=[]*, *job_name='localhost'*, *auto_add_unknown_players=True*)

Configure TF Encrypted to use threads on the local CPU to simulate the different players.

Intended mostly for development/debugging use.

By default new players will be added when looked up for the first time; this is useful for instance to get a complete list of players involved in a particular computation (see *auto_add_unknown_players*).

Parameters

- **player_names** (*(str)*) – List of players to be used in the session.
- **job_name** (*str*) – The name of the job.
- **auto_add_unknown_players** (*bool*) – Automatically add player on first lookup.

get_player(*name*)
Retrieve a specific `Player` object by name.

get_tf_config() → tf.ConfigProto, or str
Extract the underlying `tf.ConfigProto`.

players
Returns the config's list of `Player` objects.

1.7.2 *RemoteConfig*

class tf_encrypted.config.**RemoteConfig**(*hostmap*, *job_name='tfe'*)

Configure TF Encrypted to use network hosts for the different players.

Parameters

- **hostmap** (*(str,str),str->str*) – A mapping of hostnames to their IP / domain.
- **job_name** (*str*) – The name of the job.

get_player(*name*)
Retrieve a specific `Player` object by name.

get_tf_config() → tf.ConfigProto, or str
Extract the underlying `tf.ConfigProto`.

static load(*filename*)
Constructs a RemoteConfig object from a JSON hostmap file.

Parameters **filename** (*str*) – Name of file to load from.

players
Returns the config's list of `Player` objects.

save (*filename*)
Saves the configuration as a JSON hostmap file.

Parameters **filename** (*str*) – Name of file to save to.

server (*name*, *start=True*)
Construct a `tf.train.Server` object for the corresponding `Player`.

Parameters **name** (*str*) – Name of player.

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*

class tf_encrypted.layers.dense.**Dense**(*input_shape: List[int], out_features: int, transpose_input=False, transpose_weight=False*)
Standard dense linear layer including bias.

Parameters

- **in_features** (*int*) – number of input features
- **out_features** (*int*) – number of output neurons for the layer

backward(*d_y*, *learning_rate*)
The backward pass for training.

forward(*x*)
Forward pass for inference

get_output_shape()
Returns the layer's output shape

1.8.2 *Convolution*

class tf_encrypted.layers.convolution.**Conv2D**(*input_shape: List[int], filter_shape: List[int], strides: int = 1, padding: str = 'SAME', filter_init=<function Conv2D.<lambda>>, l2reg_lambda: float = 0.0, channels_first: bool = True*)
2 Dimensional convolutional layer, expects NCHW data format

Parameters

- **input_shape** (*List[int]*) – The shape of the data flowing into the convolution.
- **filter_shape** (*List[int]*) – The shape of the convolutional filter. Expected to be rank 4.
- **strides** (*int*) – The size of the stride
- **str** (*padding*) – The type of padding ("SAAME" or "VALID")

- **filter_init** (*lambda*) – lambda function with shape parameter

  Example

```
Conv2D((4, 4, 1, 20), strides=2, filter_init=lambda shp:
        np.random.normal(scale=0.01, size=shp))
```

backward(*d_y*, *learning_rate*)
The backward pass for training.

forward(*x*)
Forward pass for inference

get_output_shape() → List[int]
Returns the layer's output shape

1.8.3 *AveragePooling2D*

class tf_encrypted.layers.pooling.**AveragePooling2D**(*input_shape: List[int], pool_size: Union[int, Tuple[int, int], List[int]], strides: Union[int, Tuple[int, int], List[int], None] = None, padding: str = 'SAME', channels_first: bool = True*)

See tf.nn.avg_pool

1.8.4 *MaxPooling2D*

class tf_encrypted.layers.pooling.**MaxPooling2D**(*input_shape: List[int], pool_size: Union[int, Tuple[int, int], List[int]], strides: Union[int, Tuple[int, int], List[int], None] = None, padding: str = 'SAME', channels_first: bool = True*)

See tf.nn.max_pool

1.8.5 *Batchnorm*

class tf_encrypted.layers.batchnorm.**Batchnorm**(*input_shape: List[int], mean: numpy.ndarray, variance: numpy.ndarray, scale: numpy.ndarray, offset: numpy.ndarray, variance_epsilon: float = 1e-08*)

Batch Normalization Layer

Parameters

- **input_shape** (*List[int]*) – input shape of the data flowing into the layer
- **mean** (*np.ndarray*) – ...
- **variance** (*np.ndarray*) – ...
- **scale** (*np.ndarray*) – ...
- **offset** (*np.ndarray*) – ...

- **variance_epsilon** (*float*) – ...

backward() → None
backward is not implemented for *batchnorm*

Raises NotImplementedError

forward(*x: tf_encrypted.protocol.pond.PondPrivateTensor*) → tf_encrypted.protocol.pond.PondPrivateTensor
Forward pass for inference

get_output_shape() → List[int]
Returns the layer's output shape

1.8.6 *Sigmoid*

class tf_encrypted.layers.activation.**Sigmoid**(*input_shape: List[int]*)
Sigmoid Layer

See tf.nn.Sigmoid

backward(*d_y*, **args*)
The backward pass for training.

forward(*x*)
Forward pass for inference

get_output_shape() → List[int]
Returns the layer's output shape

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)*)

class tf_encrypted.layers.activation.**Relu**(*input_shape: List[int]*)
Relu Layer

See tf.nn.relu

backward(*d_y*, **args*)
backward is not implemented for *Relu*

Raises NotImplementedError

forward(*x*)

Parameters **x** (*PondTensor*) – The input tensor

Return type PondTensor

Returns A pond tensor with the same backing type as the input tensor.

get_output_shape() → List[int]
Returns the layer's output shape

1.8.8 *Tanh*

class tf_encrypted.layers.activation.**Tanh**(*input_shape: List[int]*)
Tanh Layer

See tf.nn.tanh

backward(*d_y*, **args*)
backward is not implemented for *Tanh*

Raises NotImplementedError

forward(*x*)
Forward pass for inference

get_output_shape() → List[int]
Returns the layer's output shape

1.8.9 *Reshape*

class tf_encrypted.layers.reshape.**Reshape**(*input_shape: List[int], output_shape: List[int] = [-1]*)

backward(**args*, ***kwargs*)
The backward pass for training.

forward(*x*)
Forward pass for inference

get_output_shape() → List[int]
Returns the layer's output shape

Python Module Index

t

Index

T

U