federated_learning_basic_concepts_aggregation_operators

June 18, 2020

1 Federated learning: Aggregation operators

In this notebook we provide an explanation of the implementation of the different federated aggregation operators provided in the platform. Before discussing the different aggregation operators, we establish the federated configuration (for more information see Basic Concepts Notebook).

[1]:
```python
import matplotlib.pyplot as plt
import shfl
import tensorflow as tf
import numpy as np


database = shfl.data_base.Emnist()
train_data, train_labels, test_data, test_labels = database.load_data()

iid_distribution = shfl.data_distribution.IidDataDistribution(database)
federated_data, test_data, test_labels = iid_distribution.
 ↪get_federated_data(num_nodes=5, percent=10)

def model_builder():
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3), padding='same',␣
 ↪activation='relu', strides=1, input_shape=(28, 28, 1)))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=2, strides=2,␣
 ↪padding='valid'))
    model.add(tf.keras.layers.Dropout(0.4))
    model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3), padding='same',␣
 ↪activation='relu', strides=1))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=2, strides=2,␣
 ↪padding='valid'))
    model.add(tf.keras.layers.Dropout(0.3))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(128, activation='relu'))
    model.add(tf.keras.layers.Dropout(0.1))
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

```
    model.compile(optimizer="rmsprop", loss="categorical_crossentropy",␣
 ↪metrics=["accuracy"])

    return shfl.model.DeepLearningModel(model)


class Reshape(shfl.private.FederatedTransformation):

    def apply(self, labeled_data):
        labeled_data.data = np.reshape(labeled_data.data, (labeled_data.data.
 ↪shape[0], labeled_data.data.shape[1], labeled_data.data.shape[2],1))

shfl.private.federated_operation.apply_federated_transformation(federated_data,␣
 ↪Reshape())

class Normalize(shfl.private.FederatedTransformation):

    def __init__(self, mean, std):
        self.__mean = mean
        self.__std = std

    def apply(self, labeled_data):
        labeled_data.data = (labeled_data.data - self.__mean)/self.__std


mean = np.mean(train_data.data)
std = np.std(train_data.data)
shfl.private.federated_operation.apply_federated_transformation(federated_data,␣
 ↪Normalize(mean, std))

test_data = np.reshape(test_data, (test_data.shape[0], test_data.shape[1],␣
 ↪test_data.shape[2],1))
```

Once we have loaded and federated the data and established the learning model, it only remains to establish the aggregation operator. At the moment, we have implemented: FedAvg and WeightedFedAvg. The implementation of the federated aggregation operators are as follows.

1.1 Federated Averaging (`FedAvg`) Operator

In this section, we detail the implementation of FedAvg (see FedAVg) proposed by Google in this paper.

It is based on the arithmetic mean of the local weights W_i trained in each of the local clients C_i. That is, the weights W of the global model after each round of training are

$$W = \frac{1}{n_{\text{C}}} \sum_{i=1}^{n_{\text{C}}} W_i$$

For its implementation, we create a class that implements FederatedAggregator interface. The method aggregate_weights is overwritten calculating the mean of the local weights of each client.

[2]:
```
import numpy as np

from shfl.federated_aggregator.federated_aggregator import FederatedAggregator


class FedAvgAggregator(FederatedAggregator):
    """
    Implementation of Federated Averaging Aggregator. It only uses a simple␣
↪average of the parameters of all the models
    """

    def aggregate_weights(self, clients_params):
        clients_params_array = np.array(clients_params)

        num_clients = clients_params_array.shape[0]
        num_layers = clients_params_array.shape[1]

        aggregated_weights = np.array([np.mean(clients_params_array[:, layer],␣
↪axis=0) for layer in range(num_layers)])

        return aggregated_weights


fedavg_aggregator = FedAvgAggregator()
```

1.2 Weighted Federated Averaging (`WeightedFedAvg`) Operator

In this section, we detail the implementation of `WeightedFedAvg` (see WeightedFedAVg). It is the weighted version of `FedAvg`. The weight of each client C_i is determined by the amount of client's data n_i with respect to total training data n. That is, the parameters W of the global model after each round of training are

$$W = \sum_{i=1}^{n} \frac{n_i}{n} W_i$$

When all clients have the same amount of data it is equivalent to FedAvg.

For its implementation, we create a class that implements `FederatedAggregator` interface. The method `aggregate_weights` is overwritten calculating the weighted mean of the local parameters of each client. For that purpose, we first weight the local parameters by the percentage and, after that, we sum the weighted parameters.

[3]:
```
import numpy as np

from shfl.federated_aggregator.federated_aggregator import FederatedAggregator
```

```python
class WeightedFedAvgAggregator(FederatedAggregator):
    """
    Implementation of Weighted Federated Averaging Aggregator. The aggregation␣
↪of the parameters is based in the number of data \
    in every node.
    """

    def aggregate_weights(self, clients_params):
        clients_params_array = np.array(clients_params)

        num_clients = clients_params_array.shape[0]
        num_layers = clients_params_array.shape[1]

        ponderated_weights = np.array([self._percentage[client] *␣
↪clients_params_array[client, :] for client in range(num_clients)])
        aggregated_weights = np.array([np.sum(ponderated_weights[:, layer],␣
↪axis=0) for layer in range(num_layers)])

        return aggregated_weights

weighted_fedavg_aggregator = WeightedFedAvgAggregator()
```

Finally, we are ready to establish the federated government with any of the implemented aggregation operators and start the federated learning process.

[4]:
```python
federated_government = shfl.federated_government.
↪FederatedGovernment(model_builder, federated_data, fedavg_aggregator)
```

[5]:
```python
federated_government.run_rounds(1, test_data, test_labels)
```

```
Accuracy round 0
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x13f9a2d90>: [14.62197494506836, 0.8900499939918518]
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x13fdca110>: [17.196195602416992, 0.8769749999046326]
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x13fdca290>: [15.791318893432617, 0.8892250061035156]
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x13fdca3d0>: [27.159305572509766, 0.8375999927520752]
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x13fdca510>: [25.262977600097656, 0.8450750112533569]
Global model test performance : [12.045165061950684, 0.8735250234603882]
```

1.3 Cluster Federated Averaging (`ClusterFedAvg`) Operator

In this section, we detail the implementation of `ClusterFedAvg` (see ClusterFedAVg).

It consists on the aggregation operator used for k-means clustering. When aggregating the centroids of a federated k-means clustering we face with the problem of grouping the clusters for subsequent aggregation. Based on the hypothesis that the closest centroids will belong to the same cluster, we apply K-Means over the centroids in order to group the centroids which belong to the same cluster and to obtain the representation (aggregation) of each group. We choose as the aggregation the new centroids obtained.

For its implementation, we create a class that implements `FederatedAggregator` interface. The method `aggregate_weights` is overwritten applying K-Means to the clients' centroids.

[6]:
```
from shfl.federated_aggregator.federated_aggregator import FederatedAggregator
import numpy as np
from sklearn.cluster import KMeans


class ClusterFedAvgAggregator(FederatedAggregator):
    """
    Implementation of Cluster Average Federated Aggregator.
    It adds another k-means to find the minimum distance of cluster centroids␣
 ↪coming from each node.
    """

    def aggregate_weights(self, clients_params):
        clients_params_array = np.concatenate((clients_params))

        n_clusters = clients_params[0].shape[0]
        model_aggregator = KMeans(n_clusters=n_clusters, init='k-means++')
        model_aggregator.fit(clients_params_array)
        aggregated_weights = np.array(model_aggregator.cluster_centers_)
        return aggregated_weights
```

We create a federated government of clustering in order to apply this aggregation operator.

[7]:
```
c_database = shfl.data_base.Iris()
c_train_data, c_train_labels, c_test_data, c_test_labels = c_database.
 ↪load_data()

c_iid_distribution = shfl.data_distribution.IidDataDistribution(c_database)
c_federated_data, c_test_data, c_test_labels = c_iid_distribution.
 ↪get_federated_data(num_nodes=3, percent=50)

n_clusters = 3 # Set number of clusters
n_features = train_data.shape[1]
def clustering_model_builder():
```

```
    model = shfl.model.KMeansModel(n_clusters=n_clusters, n_features =␣
↪n_features)
    return model

clustering_aggregator = ClusterFedAvgAggregator()

clustering_federated_government = shfl.federated_government.
↪FederatedGovernment(clustering_model_builder, c_federated_data,␣
↪clustering_aggregator)
```

[8]:
```
clustering_federated_government.run_rounds(1, c_test_data, c_test_labels)
```

```
Accuracy round 0
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x14151bc10>: (0.7236313820969188, 0.8094668170538626,
0.7641462130876469, 0.6593673965936739)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x1415dab50>: (0.7653811841780584, 0.7653811841780584,
0.7653811841780586, 0.6338273757628596)
Test performance client <shfl.private.federated_operation.FederatedDataNode
object at 0x1415daed0>: (0.7236313820969188, 0.8094668170538626,
0.7641462130876469, 0.6593673965936739)
Global model test performance : (0.7236313820969188, 0.8094668170538626,
0.7641462130876469, 0.6593673965936739)
```