UNITED STATES PATENT APPLICATION

For

A FEDERATED LEARNING APPLIANCE FOR CONSTRAINED DEVICES

INVENTORS:

THOMAS SANDHOLM

BERNARDO HUBERMAN

SAYANDEV MUKHERJEE

A Federated Learning Appliance for Constrained Devices

Abstract

We propose a novel federated learning appliance that can plug into any embedded device with a USB serial port to perform custom federated machine learning with minimal coding and configuration required.

Background

Federated learning promises improved privacy and performance by training models closer to the location where data are generated. Many network devices are single-purpose and constrained in their computing capacity, making it hard to even execute trained models. Installing software on embedded devices in general is also a cumbersome process inhibiting early exploration. Many of these devices are also limited in disk space and memory available.

Many state-of-the art machine learning models have been implemented in Python, but installing even the base Python runtime could use up 100MB of disk. Many ML models require further libraries both native and Python based. Even if you manage to port all the required libraries and run them on the device, it will take up CPU and memory from other functions running on the device.

We solve all these problems by running the ML model training on a USB dongle with a microprocessor that runs an embedded version of Python [1].

A gateway is also deployed in the dongle that is run on the host system and is designed to require a minimal common Linux coreutils runtime. It interacts with the Python runtime as well as a controller service deployed in the network.

We propose the dongle software, the controller, the gateway as well as a model to customize the federated learning process to enable implementation of many network operations optimization tasks.

The only assumptions here are that the network device has Linux-kernel-based embedded firmware, and a serial port available (e.g. USB). We note that many routers and Wi-Fi Access Points provide open USB ports to plug in shared storage and printers and make them available to the network. Our proposed ML/FL dongle could be enabled in a similar way on a local network through some AP control panel. An example dongle microcontroller could be [2], which currently retails for ~$4, which would make our solution affordable for both big and small deployments.

Description

Our goal is to support federated learning use cases where network operations tasks can be automated and self-optimization.

In the general model some state (network load, network configuration) is observed, and an action is taken (change of network configuration), then in a subsequent time period a reward (performance, QoS, QoE) for taking the action is measured. The process repeats until the system has learned the optimal action to take for each possible state. At that point the model is used to make predictions of which action will give the best performance based on the observed state.

Now a second level model is built to find a linear combination between state parameters observed, actions taken and outcomes (performance). The linear combination parameters are then shared across all learners via a centralized controller that is responsible for combining all the local models into a global model and then it communicates the global model back to the individual learners that then make a linear combination of the global and local model in the next step of learning.

Collection of state vectors and action reward samples as well as creating the linear models can be done regardless of the specific state, action, and reward parameters that are chosen to implement a specific use case. Hence our appliance only requires the deployer to specify the state collection, action enforcement and reward measurement to implement a full end-to-end federated learning system.

Data collection, Calibration, Training and Predictions proceed as follows:

In each epoch (fixed time interval) the following steps are performed.

1. Current state is collected (an arbitrary feature vector)
2. If there are enough state records collected a sample record is created,
   If there are enough sample records created a linear model is trained over the sample records (se below how training is done)
3. If a model exists communicate it to the controller which returns a global model that is then merged with the local model using a linear combination of the coefficients. Higher weight on the global model leads to faster convergence but may suppress local differences.
4. If a model exists use it to predict the next action. If not pick the next calibration action randomly.
5. Enforce the action picked

A state record contains the triple:
<state vector> <action> <reward>

Note that the <state vector> and <action> are collected in epoch t and <reward> in t+1.

The sample records are created from the state records when there are enough state records available. The max number of records may also be set to create a moving window of history maintained. Sample record construction proceeds as follows:

1. For each feature in the feature vector, quantize the feature in a configured fixed set of levels
2. For each unique quantized feature vector combination pick the action with the highest reward and record the quantized feature vector, and the action in the sample record
3. Like with state records the sample records can also having a moving window of history configured
4. When enough sample records are available separate the state vector portions into a feature matrix and the optimal actions into an output vector
5. Train a MLR model with e.g. OLS over the features and outputs to obtain coefficients, one for each feature plus an optional intercept.
6. The coefficient vector obtained through local training can now be combined with a global model (see above)

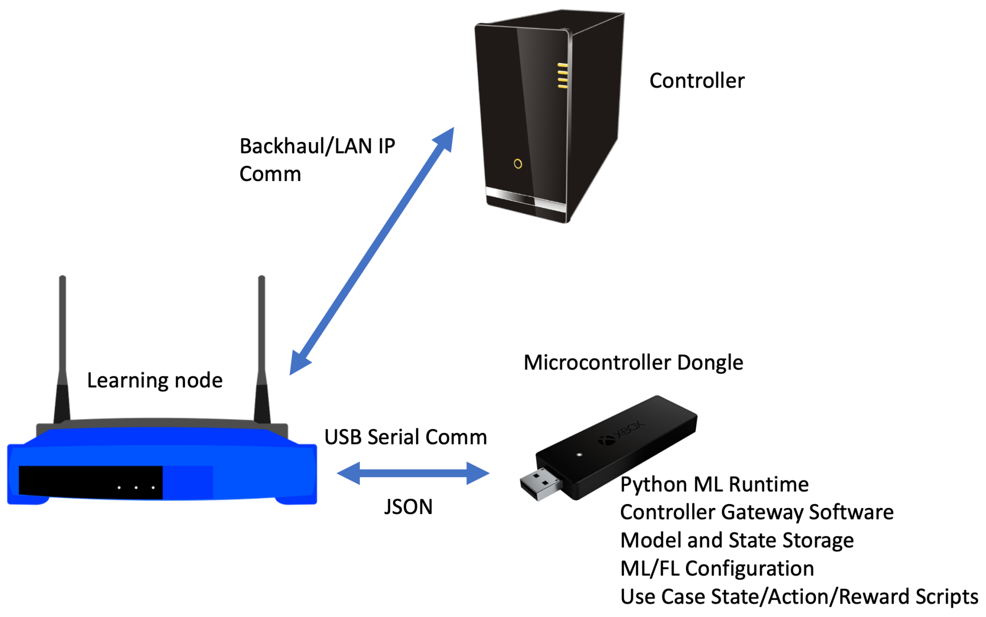
Note that the coefficient vector could be obtained by other means than via OLS assuming simple linear relationships. Other candidates would be Bayesian inference and deep neural networks (DNN). Any model that allows a feature vector to be mapped to an output vector based on labelled samples may be used. Although the dongle runs on a dedicated microprocessor and a potentially smaller data set, the storage, memory and processing capacity is still very limited and simpler models are hence preferred. The dongle can also support simple OTA updates as the microprocessor file system is used by the controller gateway and it can also be programmatically rebooted to pick up new changes.

The controller gateway performs the following functions

1. Runs the training and inference loops and calls out to the custom use case state, next, reward, action scripts and invokes the ML functions in the dongle ML runtime.
2. Stores temporary and model and sample data on the dongle
3. Shares and updates local models by interacting with the remote controller over protocols such as HTTP(S)
4. Manages configuration of the training and inference pipeline and controller mapping
5. Provides all scripts on the dongle but operates in (gets invoked from) the host OS to use its backhaul connectivity and access its state and enforce controls.

The controller performs the following functions

1. When a coefficient vector is received from a local learner record it in a record for that learner
2. Combine all coefficient vectors from all learners recorded potentially using moving averages and sample based weighting across learners.
3. Communicate the combined coefficient vector back to the learner



Demo

https://web.microsoftstream.com/video/ad2c917d-4eec-4841-9ba1-267dba66a70e

Source Code

https://code.cablelabs.com/tsandholm/bashml/-/tree/master/pico#federated-learning

References

[1] https://circuitpython.org/
[2] https://www.raspberrypi.org/products/raspberry-pi-pico/