Chrome OS ML Service

THIS IS A PUBLIC DOCUMENT

go/chromeos-ml-service

Authors: kennetht, martis, renjieliu, claudiomagni, amoylan

2018-Q4 status - Completed. See up-to-update docs in platform2/ml.
See below for original design.

See also subcomponent design docs:

- ***Chrome OS ML Service: IPC Implementation***
- ***Chrome OS ML Service: Model Publishing***
- ***Chrome OS ML Service: Mojo API***

One-page overview

Summary

The goal is to create a new Machine Learning Component/Service which uses TFLite (and later TensorFlow) as a platform for client-side ML model inferencing. This will provide a common ML runtime for personalization features for Chrome OS and lay the foundation for supporting on-device training in the future (Personalized and Federated Learning).

Platforms

Chrome OS.

Team

kennetht@google.com (emeritus), omrilio@google.com (PM), martis@google.com, renjieliu@google.com, claudiomagni@google.com, chromeos-ml@google.com, amoylan@google.com (TL)

PRD

Pending.

Launch bug

crbug.com/892495

Code affected

See this doc for all code & model source locations, and installed paths on CrOS devices.
The main source locations are:

- Client library for accessing the learning service, in //chromeos/services/machine_learning in the Chromium codebase.
- Daemon implementing the service, in /src/platform2/ml/ in the chromiumos repo.

Background

A number of future features/use-cases were outlined at the Chrome OS Product Strategy review, which illustrate how using ML on the client/device can be used to power smart features and better personalize the experience for users. Some features which are in active development are listed below. They will initially use existing ML facilities in Chrome (e.g. Assist Ranker) with the option to migrate to TensorFlow when it is available.

- Chrome OS power management (design)
- Chrome OS tab discarder (design)
- Use-cases also being considered include palm rejection and adaptive screen brightness

Design

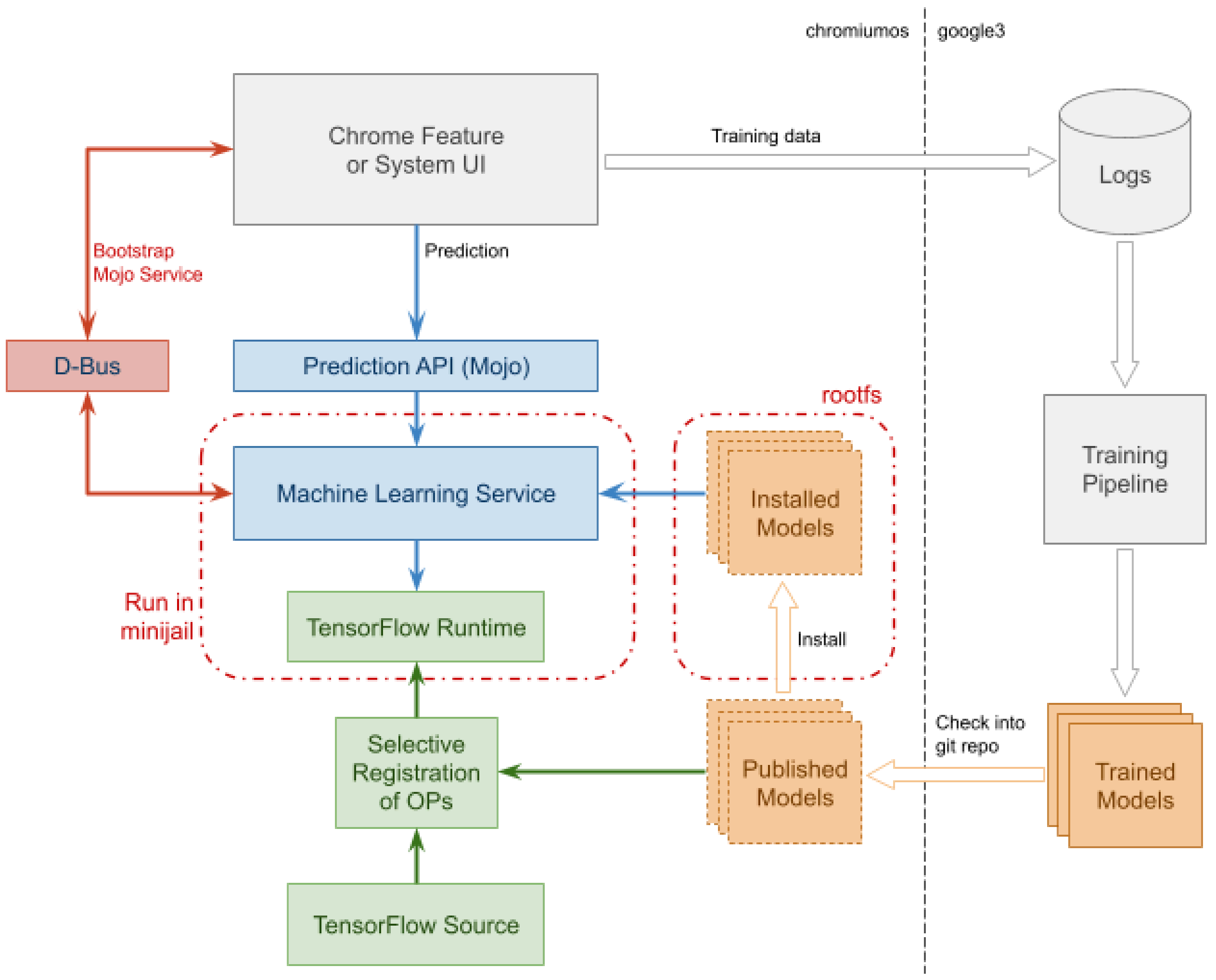
The design outlined in this document introduces a new Machine Learning Service, which wraps TensorFlow / TFLite and provides an API for performing prediction on pre-trained machine learning models. Where possible, the approach will be to follow well-worn paths for Chrome OS and Tensorflow with regards to building, packaging and deploying the TensorFlow runtime and pre-trained models. This will allow the engineering effort to be focused on the challenges of bringing a large third-party library like TensorFlow into Chrome and working through the architecture, security and system health requirements.

In the future, the Machine Learning Service will be extended to support on-device training (Personalized and Federated Learning) by integrating the client library for the new Federated Computation Platform (FCP), which is the next major iteration of Brella and is being built from the beginning to have cross-platform support and be open sourced.

Architecture

The proposed developer flow is as follows (and illustrated below):

- Client feature teams log training data which is used to train ML models using TensorFlow (server-side), which are ~~checked into the chromium git repo~~ uploaded to pantheon and subsequently installed on rootfs.
- A Machine Learning Service which wraps a version of the TensorFlow Lite runtime which supports all of the published models is built using the chromiumos build system (ebuild packages).
- The Machine Learning Service runs in a minijail sandbox.
- Client features can perform inference on the pre-trained models through a new Prediction Mojo API.
- Clients in Chromium can access the Mojo API (with automatic D-Bus-bootstrapped connection to the daemon process) using a small provided client library.



The details for each part are discussed in the following sections. Options and alternatives are also given for each, along with lines of thinking which led to each design decision. See the Architectural Choices doc and ML Service Discussion notes for a summary of the discussions.

Build and Infrastructure

As our initial goal is to release only on Chrome OS for v1, the ML Service can be directly deployed via custom ebuild packages.

- The initial plan is to build the TensorFlow/Lite static or dynamic libraries and ML Service binary as Portage packages in Chrome OS. See Chrome OS TF build notes for further details.
    - ebuild is flexible and allows TensorFlow to be built using its Makefile or Bazel build targets, avoiding the complication of handcrafting BUILD.gn rules.

  - Building the ML Service also as an ebuild means that it can have full access to the TensorFlow headers, as well as give the greatest control over compiler and link-time optimizations in order to minimize binary size.
  - The Mojo API for the ML Service is the bridge between the chromiumos and chromium repositories and has the benefit of being narrow, stable and maintainable.
- *Alternative(1):* ebuild for TensorFlow, but ML Service as a Chrome Service.
  - TensorFlow would need to be wrapped by a narrower C++ API which does not directly include any TensorFlow headers (which contain inlined implementations). This wrapped API implementation would be packaged with TensorFlow into a shared library and the headers would need to be mirrored between the chromiumos and chromium repos.
  - The ML Service in Chrome would dynamically load the wrapped TensorFlow shared library when it is initialized.
  - May be an easier developer workflow for client feature teams who could otherwise use the Simple Chrome build. After building the TensorFlow .so once, it won't likely be needed to be rebuilt until TensorFlow is up-rev'd.
  - Opens the opportunity for the ML Service to be tested with Browser tests.
  - **Decision:** Will keep this open as a viable Plan B.
- *Alternative(2):* Manually create the BUILD.gn targets for TensorFlow and its deps, and build into Chrome (and not Chrome OS). This appears difficult; our initial attempts to resolve the impedance mismatch between Bazel and GN has led to expanding dimensions of complexity which would require significant effort to resolve at this stage (see Building TensorFlow in Chrome and Building TF Lite in Chrome for details).
  - Paves the way for the future launch of the ML Service in Chrome.
  - Requires that the build systems mismatch (GN vs Bazel) is resolved up-front, including looking into opportunities to automate this conversion process (or in part) to reduce the future maintenance burdens.
  - **Decision:** Will re-open investigations when considering v2 plans for making the ML Service available on Chrome platforms.

The key consideration is the impact of binary size of building TensorFlow/Lite into Chrome, along with other system health requirements (see the Speed section for more discussion).

- Experiment with Selective Registration of Ops when building TensorFlow to reduce binary size. This can be run for each release over the set of published models.
- For TensorFlow, we could follow process used by Brella for automatically determining the set of OPs used by the ML models shipped with the build. The tensorflow.so and JNI bindings build for the Brella GMSCore module (Android) is less than 3MB, and is a good target to set for the ML Service.
- TensorFlow Lite is smaller still, around 1.2 MB in most recent test.

Note, the Bazel build targets and Makefiles for TensorFlow (which are optimized for client-side runtimes) are assumed to used as part of either Android or iOS builds. It is likely that there will need to be some upstream changes to the TensorFlow build rules to either modify the mobile targets to make them more flexible, or add new light-weight targets for client runtimes. These will need:

- No RTTI
- Disabling exceptions
- Stripping debug strings and symbols
- Compiling with protobuf lite-runtime
- Using the tensorflow/core:framework_lite (instead of :framework)
- Removing GPU/CUDA support (i.e. CPU only)
- Supporting CPU instruction sets for math acceleration (e.g. SSE4)

ML Model Publishing

See: Chrome OS ML Service Model Publishing

Summary of the ML model publishing flow:

- Training and publishing ML models
    - [Client team responsibility] Log training samples
    - [Client team responsibility] Pipeline for training ML Models with TensorFlow in google3 on Borg
    - [ML Service responsible for upload instructions] Snapshot models and upload them to the designated location
    - [ML Service responsibility] Need to limit ML model check-ins to Google engineers
        - If installing into rootfs, models are uploaded to the ChromeOS file mirror, which can be accessed by Google engineers only. Furthermore, actual installation of models is gated by changes to the ML Service ebuild file, which is protected by the usual code review process.
        - If using DLC, models become ChromeOS components that need to be authorized by the DLC team beforehand. Details TBD.
- Installing published ML models
    - ebuild package for copying the library of published ML models into rootfs in the board image (/opt/google/chrome/ml_models/)
    - Since rootfs is verified we can assume that the ML models have not been modified or tampered with on disk.
- Experiments and model variants are managed by their respective feature teams.
    - Feature teams can ship multiple models for which the ML Service has no understanding of how they are related to one another.
    - The feature code chooses which model to load for predictions based upon experiment flags that it has configured.

    - In the future we could support automatically switching to the right model file based on fieldtrials.
    - *Alternative:* When requesting a model to be loaded, an optional variant/version could be supplied and if it is not available it could fallback to the official version (and log an error).

Prediction API (Mojo)

See: Chrome OS ML Service Mojo API.

Summary of key design decisions for the Mojo API:

- An async API to avoid features accidentally blocking the main/UI thread (e.g. I/O strict mode violations on Android).
    - Will make servicification easier (if/when needed).
    - Allows the Session Scheduler to queue requests to regulate resource usage.
- ModelProvider API to load the given model (as specified by ModelSpec).
- Model API to create predictor (the same model can be shared by different clients).
    - The loaded model life cycle is controlled by the Model mojo interface pipe.
- Predictor takes client feature inputs (doing their own batching if necessary) and make predictions (In Tensor format).

Client library (Chromium)

See: ML Service Client Library

A thin client library for feature teams to access the Mojo API from within Chromium.

- Effectively just provides a wrapped version of the top-level MachineLearningServicePtr Mojo handle.
- Takes care of performing Mojo-over-D-Bus bootstrapping as needed.

Machine Learning Service daemon

See: Chrome ML Service IPC Implementation

Key considerations for the Machine Learning Service implementation and execution.

- ebuild package to build a binary and deploy to /usr/bin/
- Statically links the TensorFlow library (from its ebuild) and applies compiler and linker optimizations to reduce binary size.
- Start/Stop for each user session through Upstart script (Boot documentation)
    - Initialize the ML Service on-demand when first requested via D-Bus by a client (Chrome or System UI) (“D-Bus service activation”).
    - *Alternative(1)*: Wait until after boot-complete and initialize with the System Services since the current use-cases (e.g. Tab Discarding, Screen dimming) are not critical during startup.

    - *Alternative(2):* Need to start as part of the System Application phase, which starts Chrome and also the System Power Management service
- Run in a minijail sandbox
    - Limit the capabilities, namespaces and system API calls (seccomp-bpf)

TensorFlow Lite vs TensorFlow

Update 2018-09: We decided to switch to TF Lite simply because several potential early clients of ML Service are training TF Lite models. Full TensorFlow will likely still be required to support future training (e.g. federated learning by integrating FCP).
Original discussion below.

With the release of the TF Lite developer preview (v1.5.0) there is also an option of using this for ML inference in the client and should come with smaller binary size impact (< 300 KB), as well as support more compact models for faster evaluation and smaller memory/storage requirements.

At this stage TF Lite does not fully support sparse feature vectors, which for example would be useful for URL strings as input to the Tab Discarder, and does not support all of the TensorFlow Ops. Also, focusing initially on releasing only for Chrome OS there is less pressure on binary size and appears it will be acceptable to ship a full version TensorFlow (with Selective Registration of Ops).

The TF Lite source has a much smaller overall footprint and has a potentially more manageable set of external deps.
Shipping TF Lite will be kept as an option though and reconsidered if issues are encountered with building and packaging full TensorFlow in Bazel (or cmake).

TensorFlow Versioning

Need to be mindful of potential Issues with forward and backward compatibility:

- Ops required by the model were not compiled into the runtime (Selective Registration set of Ops changes over releases)
- It is also not uncommon that the (undocumented) behavior for a particular TensorFlow OP may change, causing a breakage for existing models.
- Server-side training vs client-side runtime skew of TensorFlow versions (built at different CLs/revisions)

The plan is to check the ML models (or source URIs to fetch them from static storage) into the chromiumos repository, which avoids issues of forward compatibility (i.e. evaluating new ML models on old runtimes) and verifies backward compatibility through integration tests for each of the published ML models.

- ebuild package for installing the ML models into a folder in rootfs of the system image.
- All Ops required by the published ML models will be kept by the Selective Registration process.
- Avoid forward compatibility issues by ensuring that the TensorFlow revision of the runtime in Chrome OS is up-rev'd to be equal to or ahead of the TensorFlow revision used for training on the server (google3).
- Define a test framework for verifying that:
    - the published models can be loaded (without error) by the TensorFlow runtime library in Chrome OS.
    - The models predict correctly using golden inputs/outputs.
    - This will catch any cases of Ops erroneously dropped during Selective Registration and adds some protection against undocumented changes in behavior for existing Ops.

Alternative: Design a versioning scheme which creates a digest of supported Ops for each binary (Chrome OS release)

- Which is checked against the set of required Ops for each model prior to loading. This would guard against potential crashes due to missing Ops.
- Integration tests for forward and backward compatibility could be added to also check for subtle/undocumented behavior changes for existing Ops.
- This would open the opportunity to ship the models independent of the runtime binary
    - Which could allow for faster experiment cycles for assessing new models.
    - Would need to also need to design the model download mechanism, including signing/verification of models which are loaded from disk.
- **Decision:** This option can be reopened in the future if the flexibility of evaluating models over the broadest set of runtimes is considered worth the engineering effort.

Metrics

Success metrics

Events: (by model/version)

- Model loaded
- Inference started
- Inference completed (or failed)

Performance Timers: (by model/version, bucketed)

- Model load time and session initialization time
    - Time to first inference (from cold start)
- Inference time (per eval)

System health impact:
- Process stats for the ml_service
- Memory used by model/version (model + session size) (When there are multiple models)

Regression metrics

At launch, no features will be using the Learning Component, so it will be mainly:
- The overall system health/stability metrics that will need to be monitored for anomalies (crash reports, app start latency).
- It will also be worth keeping an eye on the system update rates (and failures) for Chrome OS to see if they are affected by the increase in binary size.

In the future, features which use the Learning Component will need to watch for any regressions:
- for their specific performance metrics (baseline vs new ML model),
- and consider any system health impacts.

Experiments

N/A - each client feature will have its own experiments setup to choose from the set of models available on the device.

Rollout plan

Waterfall.

Core principle considerations

Speed

- Assess binary size impact
    - Estimated <3 MB for TF runtime using Selective Registration of Ops
- Measure any Chrome OS startup cost
    - May be impacted by when the ML Service is initialized
- Assess memory-use for a trivial session
    - ML Service initialized
    - Load a toy model (e.g. add 2 scalars), or model from a feature team and eval

        - Possibly use early versions of the power management or tab discarder models if available.
- Battery use, CPU/GPU execution time
    - Interrupt/Cancel model evals which overuse of CPU (most likely bugs) and log an error event.
    - E.g. session running for longer than *X* minutes, or continuous spamming of small requests (over max_count per minute/hour/day).

Out of scope:
- May need to support in-process/synchronous inferencing for use-cases in the future requiring very low-latency evals (e.g. stylus on ChromeOS) and future high-bandwidth use-cases for images, video and audio streams. Not considering this for v1. Feature teams may be able to use either the Chrome Assist Ranker or code-generation frameworks (Neurosis, tf.native) in the interim if latency is a concern.

Security

Main security issues to consider are:
- Size and complexity of TensorFlow as a third-party library, especially since it has a parser for model graphs/snapshots.
    - Sandbox the Machine Learning Service by running it under minijail (kerrnel@).
    - See Chrome OS Sandboxing documentation
- Executing of TensorFlow models which have been downloaded (either as part of rootfs or via DLC or component updater) and stored on the client would present a potential attack surface.
    - Sophisticated ML models are essentially programs which we load and run with the TensorFlow runtime.
    - By installing models in rootfs when building the system image, they will be protected by the system integrity checks on startup.
    - The policy will be that only Google engineers can be authors of ML models which are checked into the chromiumos repo and subsequently installed on rootfs. At a minimum, this can be enforced through the normal code-review process.
        - Update: The steps (only possible for Google engineers) for adding a model that will be deployed to rootfs for use by ML Service is documented here.

Privacy considerations

Teams which use the Machine Learning Component/Service will be responsible for the privacy reviews for their individual feature launches.

For the Machine Learning Service v1:

- All metrics regarding the performance and stability of the system will be aggregated through UMA.
- No user data passed to the Machine Learning Service API will be logged, sent to a server or stored on device.

Testing plan

Integration tests:

- Model integration/regression tests: Need to verify that the correct set of OPs has been included in the TensorFlow build (via Selective Registration), as trying to load a model which uses a missing OP will cause an error.
  - Each model author will need to add an integration/golden test for the ML Service which verifies that their model loads and is successfully evaluated by the TensorFlow runtime it is bundled with. Run during the test phase when building releases.
- Test that the D-Bus -> Mojo IPC bootstrap succeeds.
- Tests to verify graceful failure (i.e. appropriate Mojo error) when:
  - A requested model is not available (not present in the archive)
  - Errors during model evaluation

Unit tests:

- Coverage at >70% for the ML Service.

Follow up work

Which of these is the first priority is TBD depending on prospective clients:

- Federated Learning in Chrome using a cross-platform implementation of go/brella
  - Supports on-device learning which has a stronger privacy story and would allow training ML models using a broader spectrum of user data.
- Expand support to TensorFlow
  - Assess the supported Ops (Sparse tensors, gradient Ops)
- Take advantage of any dedicated ML hardware that may be present on devices
- Look at making the Machine Learning Service available more generally in Chrome and across its other platforms.

Appendix

Working/discussion docs [Google-internal links]:

- Chrome Learning Component - Working Doc (early thoughts)
- Chrome On-device Learning (proposal)
- Privacy: On-device Learning in Chrome
- Tracker: ML Service Tasks tab in go/ck-ml-tracker

Document history

Date	Author	Description
2017-11	kenneth t	Initial draft design (original plans for Chrome-wide service)
2018-01	kenneth t, renjieliu	First draft for review (redesigned as Chrome OS service)
2018-03	kenneth t, martis	First approved design (crbug.com/811014)
2018-08	amoylan	Add client library
2018-09	amoylan	TFLite not TF for V1. Update testing outline.