OPEN SOURCE TECHNOLOGIES

PROJECT REPORT

TITLE:

DL-TRACKER

Submitted By:

Varun Sharma

COE-2, NSIT

2017UCO1573

Abstract

In the advent of growing advances in the field of Deep Learing, more and more people have started employing these techniques to solve complex problems in the areas of Artificial Intelligence, Data Science, Computer Vision, Natural Language Processing, etc. Solving such problems, often, requires large datasets and involves a crucial step known as 'training'. '*Training*' or '*Fitting*' a Deep Neural Network, sometimes also referred to as a Deep Learning model, usually refers to the act of optimization of a cost function, chosen by the user, using certain strategies like gradient descent, hill climbing, etc.

This training step often takes a large amount of time to complete and, might require some *monitoring* and *manual stopping* upon obtaining somewhat desired result. But, one just cannot sit in front of the screen all the time to monitor the progress, so, we usually specify a particular number of epochs, and leave it to run on a local machine or a cloud environment in the background for hours. Though, to facilitate early stopping, one can use automatic callback packages available in certain deep learning packages (like the EarlyStopping and ModelCheckpoint callbacks in Tensorflow and Keras), but these tricks don't always yield the desired results.

This software has been developed **to solve this problem of realtime monitoring and manual early stopping of Deep Learning model training, remotely.** This report describes the design, implementation and motivation behind the same.

Contents

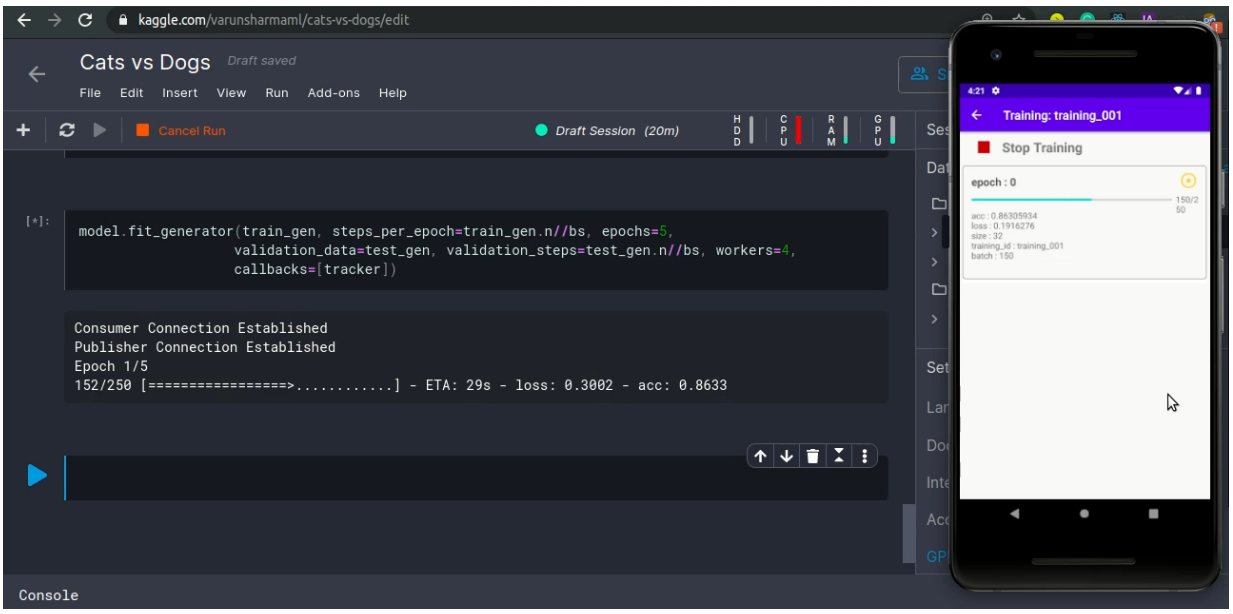
Introduction

DL-Tracker (short for *Deep Learning Tracker*) is *a software that allows one to track (monitor), manage and terminate (stop) training of deep learning models written in Tensorflow or Keras, remotely.*

The software mainly consists two parts:

1. **DL-Tracker Python Package**: A custom keras callback class that sends model training updates to the database server and the AMQP broker. Along with this, it also keeps listening to a 'signal' AMQP message queue for a 'STOP' signal from a remote user. The user needs to supply an instance of this class along the list of callbacks while executing the .fit (or .fit_generator) method on the model instance.
2. **Tracker Client (Android App and/or CLI)**: The client that is responsible for recieving the updates from the model by reading from the AMQP message queue and the database. The client can also send a 'STOP' signal to terminate training by writing to the 'signal' message queue, which causes the DL-Tracker callback package to stop model training. The client is available as an *Android Mobile Application* and a *Python command line tool*.

These the DL-Tracker python-package when used in combination with the Android Client App or the CLI client, can be used for remote monitoring and management of deep learning model trainings.



```
model.fit_generator(train_gen, steps_per_epoch=train_gen.n//bs, epochs=5,
                    validation_data=test_gen, validation_steps=test_gen.n//bs, workers=4,
                    callbacks=[tracker])
```

```
Consumer Connection Established
Publisher Connection Established
Epoch 1/5
152/250 [=================>...........] - ETA: 29s - loss: 0.3002 - acc: 0.8633
```

Training: training_001

epoch : 0

150/250

acc : 0.86305934
loss : 0.1916276
size : 32
training_id : training_001
batch : 150

Figure. DL-Tracker in action

Background Research

Motivation and Currently Available Solutions

The Background Reasearch for this project involved exploring already existing technologies and their viablity and extensiveness for remote tracking and controlling of deep learning model trainings. In fact, tensorflow and keras already ship with a loose solution for remote tracking in the form of **RemoteMonitor** which is a part of their callbacks utility. This RemoteMonitor callback can be used to stream events to a server, but these are only epoch end events, and do not allow one to Monitor training of batches in real-time. Also, this commmunication is unidirectional i.e. the *user cannot send a '**STOP**' signal* to the callback in order to stop it. In addition to this, there exists *no standard implementation for the event streaming server and no standard client application for user to read those updates*.

This project aims to create a standard, efficient and easy-to-use *interface which allows users to remotely track and control their deep learning model trainings* in Tensorflow or Keras using their Android Devices (Mobiles, Tablets, etc) or a Command Line Interface.

Choosing the Right Technologies

Several technologies and techniques were explored for fast transfer of updates between the callback and the client. These include:

- **Using a websocket connection**: Although websockets prove to be a fast method for transfer of data between two connected clients, there needs to be maintained a constant and synchronous connection between the two. Thus, **due to the need for asynchronicity, websockets do not render the desired solution.**

- **Reading and Writing all progress updates to and from the database**: This might seem to be a clever solution, because the database can provide an intermediate store which can help facilitate asynchronous exchange of updates between the client and the callback. But, here **the problem is frequent and excessive number of writes**. Sure, the database can be used for storing and exchanging infrequent updates like training updates and epoch updates, but not the frequent batch progress updates. As we are aware that high and frequent number of writes make the database transactions inefficient, we would be using database for storing just the aforementioned infrequent updates. Plus, as batch updates are not something a user would want to see in the future, as opposed to training and epoch updates which one might want to keep for future, **database provides persistent storage** for the same (infrequent updates). Also, the only way of updating the database might be HTTP requests from the callback side, which further add an overhead to the exchange of updates.

- **Using a *'Message Queuing'* based Protocol**: Message Queuing allows applications to communicate by sending messages to each other. The message queue provides temporary message storage when the destination program is busy or not connected. This **fulfills the requirement of asynchronicity**. The queue-like natures comes to us as an advantage, because here, we will **able to implicitly receive sequential updates** (which was not the case while using a database, where whole table had to be read every time). In addtion, Message-Queuing systems are **fast** (due to the eliminated request overhead) and **reliable** (messages are acknowledged upon reception). This makes it very much **suitable for exchanging the frequent updates** viz. the batch-progress updates and 'STOP' signal updates. One such widely used protocol is the **Advanced Message Queuing Protocol (AMQP)**. Multiple implementations are available for this protocol; one popular choice is RabbitMQ, which is open-source, lightweight and easy-to-use.

Based on the aforementioned analysis, ***Messsage Queuing (for frequent updates) along with a Database (for infrequent and persistent updates)*** has been chosen as the suitable method for an efficient implementation of this application.

Technology Stack

Tensorflow/Keras Callback Interface

01

Callbacks in tensorflow and keras offer a great way of performing certain actions like model checkpointing, learning rate decay/scheduling, early stopping, logging and creating metric visualizations. One can also define custom callbacks using the **tensorflow.keras.callbacks.Callback** and the **keras.callbacks.Callback** base class.

The user needs to inherit this Callback base class and override certain methods in order to define desired custom operations to be performed at specific events of model training such as training begin, training end, epoch begin, epoch end, batch begin, batch end, etc.

DL-Tracker Python package inherits this Base class and overrides its methods to send updates to the Database webhook and RabbitMQ broker at specific events, as well as, to listen to the 'STOP' signal queue while training and invoke termination when required.

RabbitMQ (Advanced Message Queuing Protocol)

02

RabbitMQ is an open-source message-broker software that supports multiple messaging protocols, including *AMQP (Advanced Message Queueing Protocol)*. 'MQ' here, stands for Message Queue; Messaging protocols often involve a queue which resides in a Message broker. A 'Publisher' can 'publish' messages to a queue and 'Subscribers' can subscribe to a queue to read/consume messages from. This publish-and-subscribe system involving an intermediate queue and some middleware like message brokers, constitutes a Messaging Queuing Protocol.

RabbitMQ provides a lightweight and easy to deploy interface, implementing AMQP. It supports features such as delivery acknowledgement, flexible routing and multiple exchange types. Several queue properties such as durabilty, auto-deletion, exclusiveness, queue length limit, etc. can be specified to provide extra customization to behavior. Due to the queue-order based delivery of messages and higher efficiency than HTTP (due to request overhead), AQMP is a suitable application layer protocol to be used for real time tracking of updates of model training and delivery of signals on a live-wire like channel.

DL-Tracker uses RabbitMQ client libraries for Android: Java (**rabbitmq.amqp.client**) and Callback Package: Python (**pika**).

03 Android Studio

Android Studio is the official integrated development environment for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development. It is available for download on Windows, macOS and Linux based operating systems. Over 2.5 Billion devices on Earth run Android. It makes it a suitable platform for running a Remote Controlling and Tracking device, as users can access it anytime and anywhere they want.

04 Firebase Cloud Firestore

Cloud Firestore is a flexible, scalable NoSQL database for mobile, web, and server development from Firebase and Google Cloud Platform. It provides features such as keeping data synced across client apps through realtime listeners and offline support for mobile and web so you can build responsive apps that work regardless of network latency or Internet connectivity.

Cloud Firestore

DL-Tracker android app uses *Firebase Firestore Android client API* to manage models and training related information and metrics. Database Webhook uses *Firebase Admin SDK* for Python, which is accessed by the DL-Tracker Callback Package through HTTP requests.

Flask Web Framework 05

Flask is a micro web framework written in *Python*. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. Thus, is provides an easy and lightweight way of writing a Database webhook server for the DL-Tracker callback to interact with, for accessing and updating Firestore database using the Firebase admin SDK.

System Architecture and Components

Components of the System

The system as a whole consists of five components, including the middleware:

1. **Device(s) training Tensorflow/Keras model(s) and using DL-Tracker Callback Package**

   This in some remote device or cloud environment executing the code for training a Tensorflow or Keras deep learning model, while using the DL-Tracker Callback instance. This environment needs to have an active internet connection in order to send and receive updates to and from the Database Webhook and the RabbitMQ Broker. The DL-Tracker callback inherits from ad overrides the following methods of the keras.callbacks.Callbacks package:

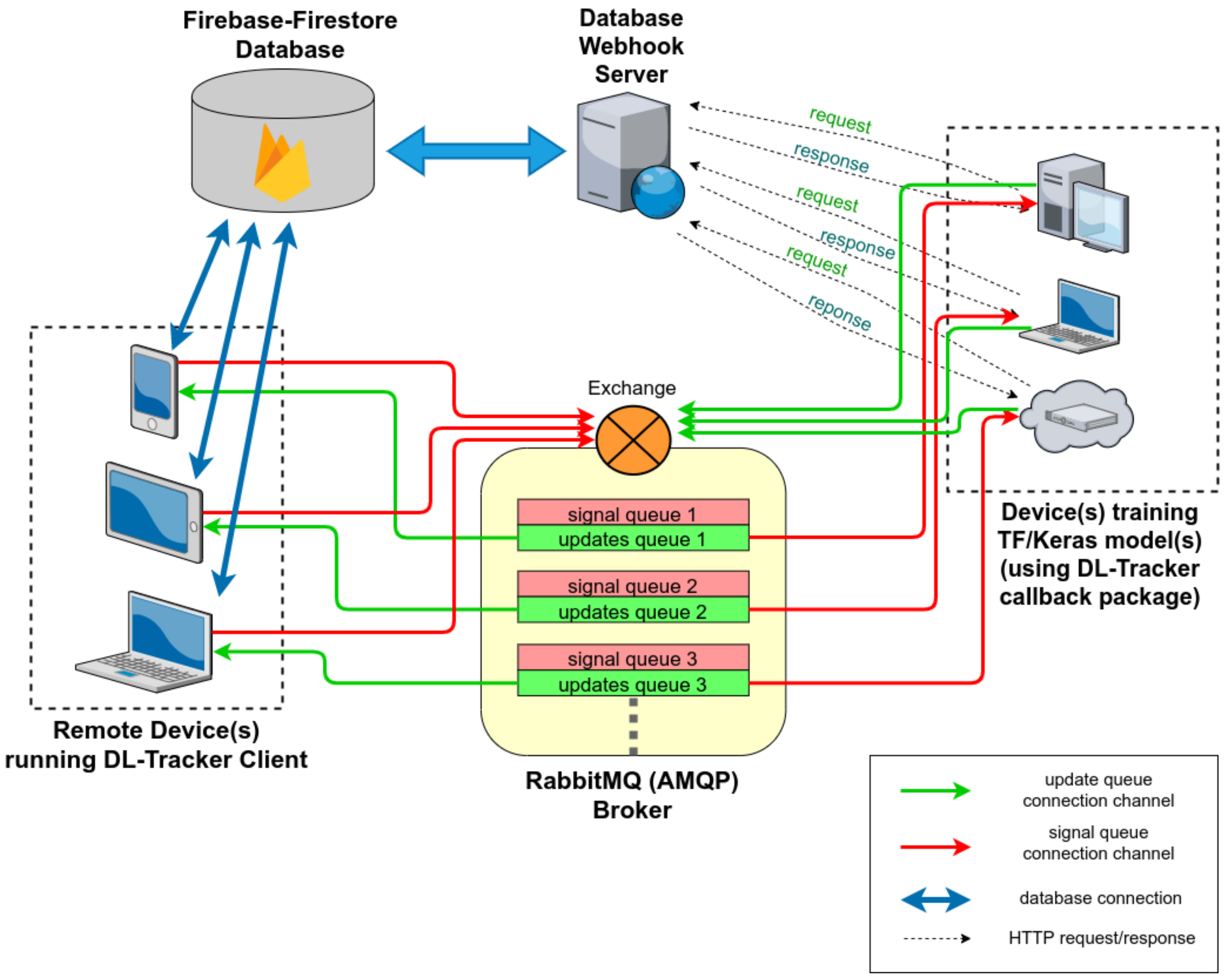
   - ***on_train_begin*** : sends a 'train_begin' update to the Database webhook and to the update queue, containing the details of the training.
   - ***on_train_end*** : sends a 'train_end' update to the Database webhook and to the update queue, containing the details of the training.
   - ***on_epoch_begin*** : sends an 'epoch_begin' update to the Database webhook and to the update queue, containing the details of the newly started epoch.
   - ***on_epoch_end*** : sends an 'epoch_end' update to the Database webhook and to the update queue, containing the details of the last fininshed epoch.
   - ***on_train_batch_end*** : sends a 'batch' type update to the update queue, containing the details of last processed batch. Note that batch updates are not written to the database, only to the update queue.

   Simultaneously, the callback maintains a connection channel with the signal queue, in order to listen to the 'STOP' signal from the remote device so that it can stop the training upon its reception.

2. **Database Webhook Server**

   The Database webhook server is responsible for authentication of DL-Tracker callback user and for recieving the training updates from the DL-Tracker callback and reflecting them in the Database. Instead of directly communicating with the Firestore Database the callback communicates using HTTP requests with this webhook. It provides an additional layer of security as it does doesn't leave the Database key and AQMP key open for use by the world.

The Database webhook server has been written using Flask Web Framework in Python and deployed on *pythonanywhere.com* (free web-hosting).



3. RabbitMQ (AMQP) Broker

The AMQP protocol is the heart of the DL-Tracker software as it is responsible for ensuring fast realtime delivery of updates between the callback and the tracker client. For each Model or 'ModelKey' the broker maintains two queues:

- an '*update*' queue, which queues the updates *published by the callback*, regarding the training progress. The TTL (Time-To-Live) for queue messages has been set to 5 seconds, so that older updates (those that were published when client wasn't connected to channel) are not read by tracker client and do not waste the bandwidth.

- a '*signal*' queue, which is responsible for storing the 'STOP' signal if published by the tracker-client. The callback constantly maintains a connection channel for consumption from this queue, listening for the 'STOP' signal. Again, the TTL (Time-to-Live) for queue messages has been set to 5 seconds so that older unread STOP signals sent to older trainings aren't recieved by the current training.

An AMQP 'Exchange' is responsible for routing messages recieved from the Publishers to the appropriate queues based on their routing key. The Subscribers of a queue consume directly from the queue without mediation from the Exchange. The RabbitMQ broker has been setup on *cloudamqp.com* (little lemur free plan).

4. Firebase Firestore Database

The Firestore Database is *responsible for storing the non-frequent updates* from the callback as well as the user information. Non-Frequent updates include *Training Begin* update, *Training End* update, *Epoch Begin* update and *Epoch End* update. Frequent updates like *Batch End* updates are not written to the database as they might increase the number of writes and worsen the efficiency of the database. Plus, in order to recieve batch type updates as a queue we would be needing to add a timestamp field, and then access it using the timestamp, slowing the database access even further.

Firestore Database is accessed by the Database Webhook from the callback side and by tracker client-device directly to read and write model, training and user data. It, thus, provides a flexible, scalable and easy to use database hosted in the cloud for our system.

5. DL-Tracker Client(s) (Remote Device using the DL-Tracker App or CLI)

The DL-Tracker Client is a tool for tracking the progress of the model training remotely. It is the client interface that reads the training updates from the Database and the update queue as well as for sending the 'STOP' signal to the signal queue. Other tasks such as Creating a new model or 'ModelKey', viewing models, deleting models, viewing tranings, deleting trainings and viewing epochs are also supported by the client. The Tracker-client is available as an *Android Mobile Application* and as a *Python Command Line* Tool.

System Workflow

The system workflow broadly consists three steps:

1. User Authetication and Establishment of Connection

- While creating an instance of DL-Tracker callback, the user needs to supply a ModelKey (generated through the DL-Tracker App) and his/her account password, which are authenticated by sending an HTTP request to the Database Webhook.

- Once, the user is authenticated (s)he can go ahead and run the model.fit or model.fit_generator method, with this instance passed along with the list of callbacks.

- Upon **train_begin** event, connection with AMQP broker is established and channels are created for 'update' and 'signal' queues. A **train_begin** type signal is sent to the 'update' queue and an HTTP request for adding a new training to the database is sent which returns the **training_id** in the response.

2. Connection by the Client Application

- As explained earlier, because the application is a message queuing system, it doesn't need to maintain a connection to the callback package at all times. It can just connect to the update and signal queue and publish/consume messages whenever it wants.

- To manage the connection of the Android Client, a **Service** is started upon start of application. Whenever a user goes to the epoch progress tracking screen (namely, **EpochsActivity**), the current activity binds to the service. Then, if there is already a connection established for the corresponding ModelKey, then the connection will just serve those updates, else, it will establish a new connection and serve from it. When the current activity is destroyed, it unbinds from the service, but the connection remains intact.

- The service, and hence, the started connections are closed along with the application.

3. **Tracking and Controlling Training using the Remote Client**

- While the connection is established and the user is on the **EpochsActivity** screen, the updates from the queue reflect on the progress bar. Upon reception of the **epoch_end** command a new epoch progress bar is added.

- At the same time, snapshot event listeners on the training info collection in make sure the **TrainingsActivity** screen is updated along with the database.

- When the user clicks the 'Stop Training' button on the **EpochsActivity** screen, a 'STOP' message is written to the signal queue.

- This 'STOP' message, when read by the connection at callback end, sets the **model.stop_training** attribute to True, which in turn, halts the model training gracefully. After this, **epoch_end** and **train_end** messages are written to the update queue and database, and and connection is closed.

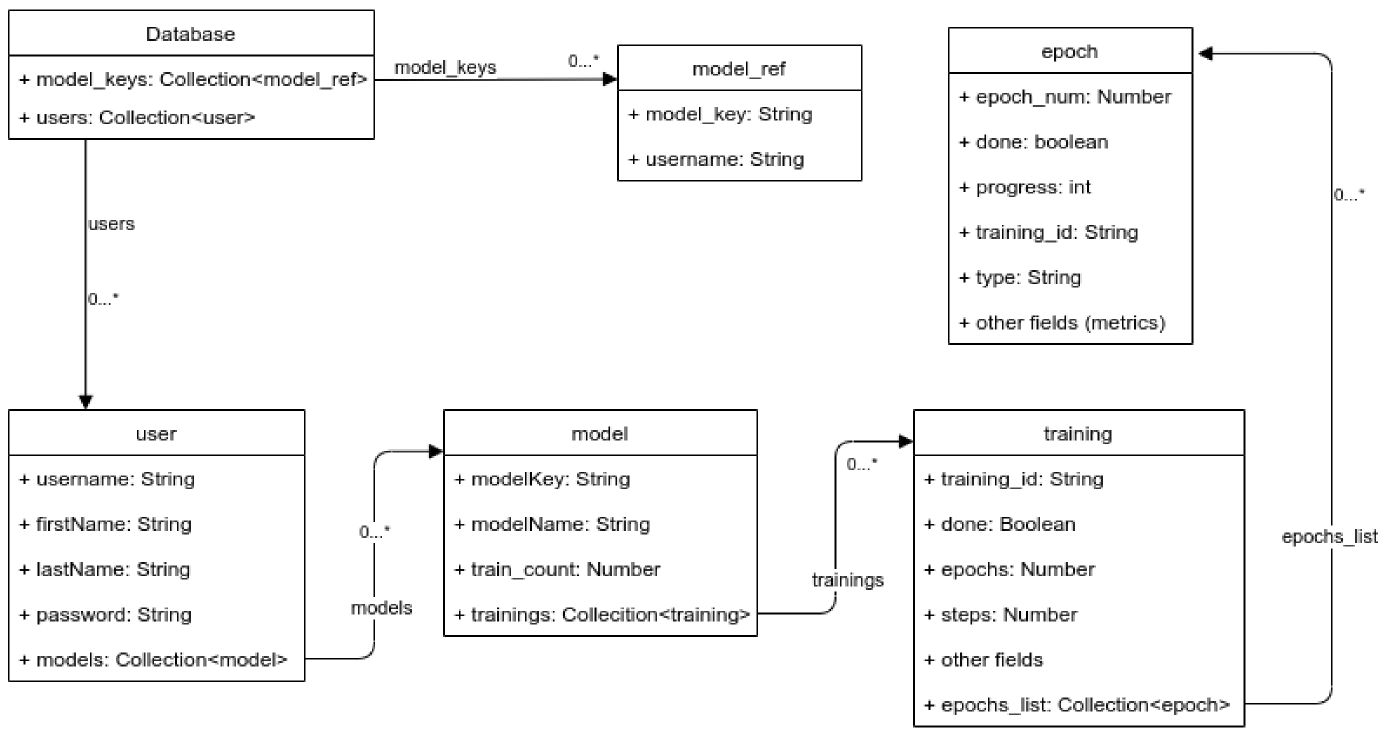
- For the *CLI client*, the user supplies the ModelKey as a command line argument, which diaplays all the previous and current trainings, and the ongoing epochs (if any). The user can send a 'STOP' signal by invoking an interrupt signal, which ultimately stops the training as well as the program.

Database Design

This application uses Firebase Cloud Firestore, which is a scalable and easy to use NoSQL database. Although, in NoSQL databases, the schema is not very concrete, but the data model can still be represented using the UML shown in figure.

- **users** collection stores the users of the application and their details. username is used as the document Id for a user document.

- **models** collection is a part of each user document. It contains the documents holding the details of all models created and registered by the user on the app. The document Id makes up the ModelKey for a model.



- **trainings** collection is a part of each model document. It contains the documents holding the details of all trainings of a model. The document Id for a training is generated automatically using the *train_count* field.

- **epochs** collection is a part of each training document. It contains the documents holding the details of each executed epoch of the training. The *epoch_num* field (which denotes the serial index of the epoch) is used as the document Id for an epoch document.

- **model_keys** collection stores a mapping of ModelKeys to their corresponding owner's usernames. Used during authentication at the callback side.

Usage

Requirements

In order to use the software, user needs to make sure the following requirements are met:

1. **For DL-Tracker Python Callback Package**

    - Active Internet Connection
    - Following Python Packages, pre-installed:
        - pika (RabbitMQ Python Client Library)
        - requests (Python HTTP Requests Library)
        - keras or tensorflow.keras

2. **For DL-Tracker Client**

    - ***For Android Application***:
        - Android Device
        - Active Internet Connection

    - ***For Python CLI tool***:
        - pika (RabbitMQ Python Client Library)
        - Firebase-Admin Python SDK
        - Active Internet Connection

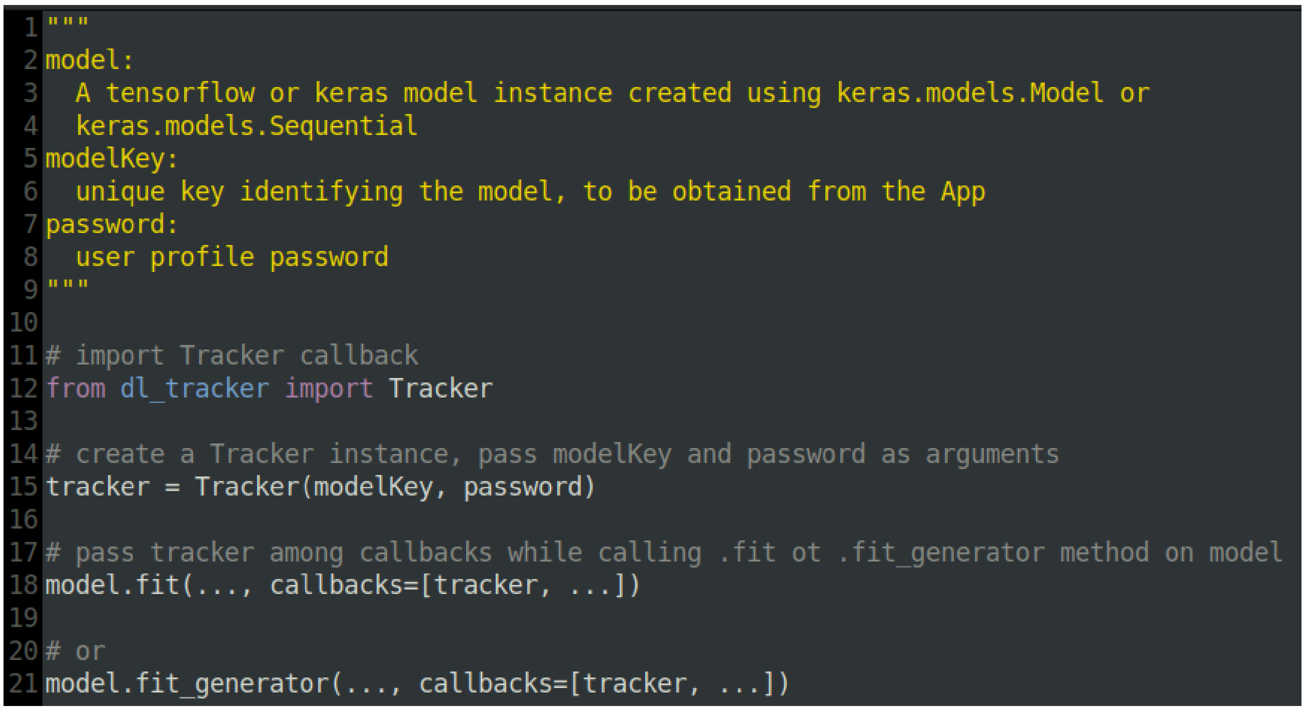
Instructions

1. **Creating a New Model Instance in App**

    - Sign In to the app and head to "My Model" tab.
    - Create a new Model Instance by clickng the "+" button.
    - Copy the corresponding Model-Key (to be supplied to Tracker instance)

2. **Adding DL-Tracker to Model Callbacks**

    The following code snippet demonstrates this step:



```
"""
model:
  A tensorflow or keras model instance created using keras.models.Model or
  keras.models.Sequential
modelKey:
  unique key identifying the model, to be obtained from the App
password:
  user profile password
"""

# import Tracker callback
from dl_tracker import Tracker

# create a Tracker instance, pass modelKey and password as arguments
tracker = Tracker(modelKey, password)

# pass tracker among callbacks while calling .fit ot .fit_generator method on model
model.fit(..., callbacks=[tracker, ...])

# or
model.fit_generator(..., callbacks=[tracker, ...])
```

3. Tracking and Controlling Training

- ***From the App***:
  - Head to the trainings screen of corresponding model by clicking "View Trainings".
  - The current unfinished training will be having a yellow dot on it.
  - Head to epochs screen of that training by clicking "View Epochs".
  - Now, the training progress of can be seen as Progress Bars.
  - To stop the training, press the Stop Training Button.

- ***From the Python CLI***
  - Run the following command in terminal with proper arguments:

    $ python3 tracker.py MODELKEY

    where, *MODELKEY* is the model-key obtained while creating the model instance in app.
  - After this, all previous and current trainings of the model will show up, along with progress bars showing current epoch's progress, if any.
  - To stop the training, send an interrupt signal using *Ctrl+C*.

CAUTION: *One ModelKey should be used to track only one model, at a time.*

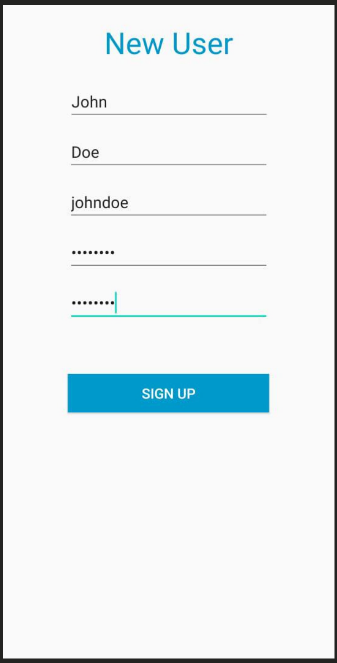
Miscellaneous Features of the App

- Deleting a Model Instance, if not needed.
- Deleting a Training instance, if not needed.
- Viewing Training Metrics of previous trainings & models.
- Signing Up as a New User.
- Editing user profile firstname, lastname, password, etc.

Demonstration

Android Application Client

1. Sign Up Screen


New User

John

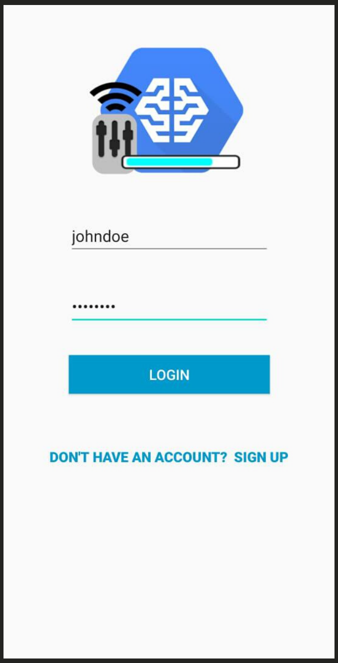
Doe

johndoe

••••••••

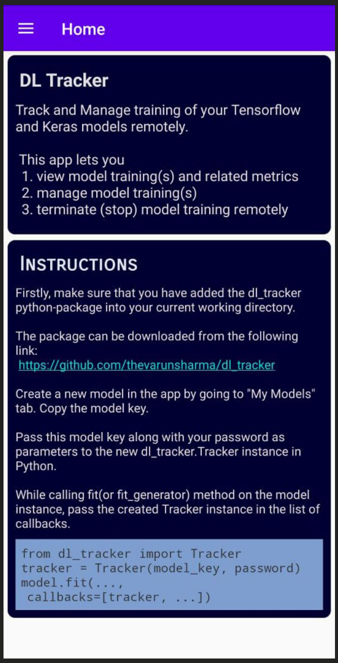
••••••••

2. Login Screen


johndoe

••••••••

3. Home Screen



DL Tracker

Track and Manage training of your Tensorflow and Keras models remotely.

This app lets you
1. view model training(s) and related metrics
2. manage model training(s)
3. terminate (stop) model training remotely

Instructions

Firstly, make sure that you have added the dl_tracker python-package into your current working directory.

The package can be downloaded from the following link:
https://github.com/thevarunsharma/dl_tracker

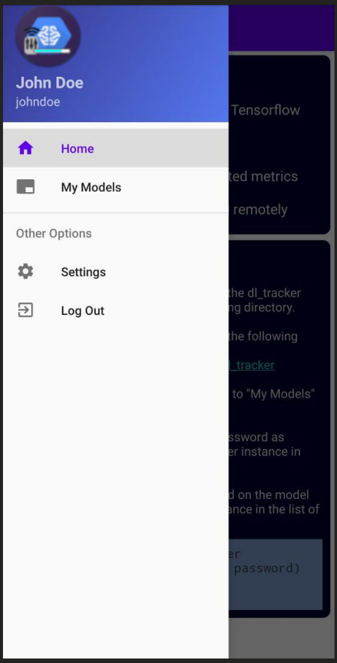
Create a new model in the app by going to "My Models" tab. Copy the model key.

Pass this model key along with your password as parameters to the new dl_tracker.Tracker instance in Python.

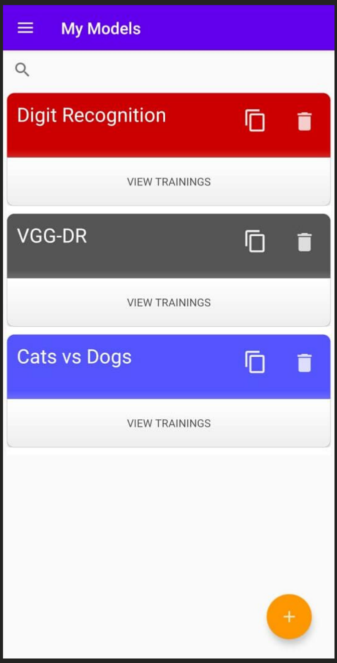
While calling fit(or fit_generator) method on the model instance, pass the created Tracker instance in the list of callbacks.

```
from dl_tracker import Tracker
tracker = Tracker(model_key, password)
model.fit(...,
 callbacks=[tracker, ...])
```

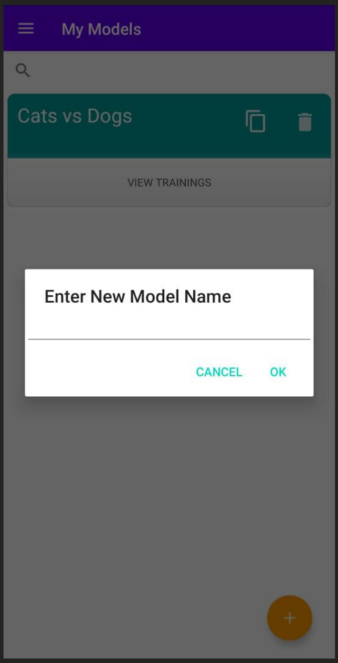
4. Navigation



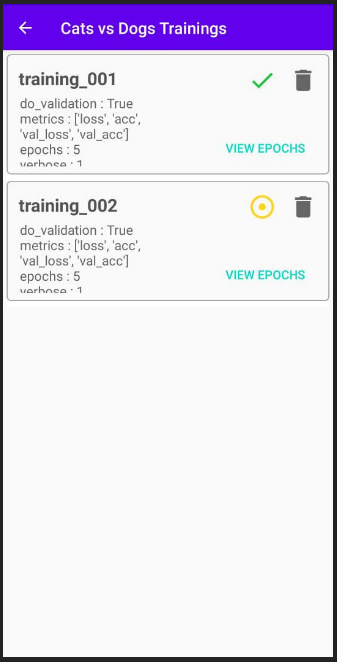
5. 'My Models' Screen



6. 'Add Model' Dialog



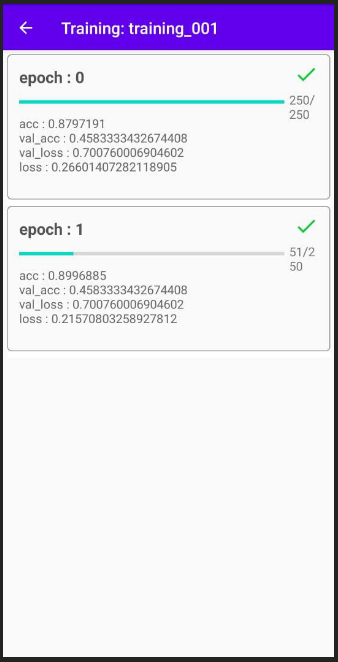
7. Model Trainings Screen with an ongoing training


Cats vs Dogs Trainings

training_001
do_validation : True
metrics : ['loss', 'acc', 'val_loss', 'val_acc']
epochs : 5
verbose : 1
VIEW EPOCHS

training_002
do_validation : True
metrics : ['loss', 'acc', 'val_loss', 'val_acc']
epochs : 5
verbose : 1
VIEW EPOCHS

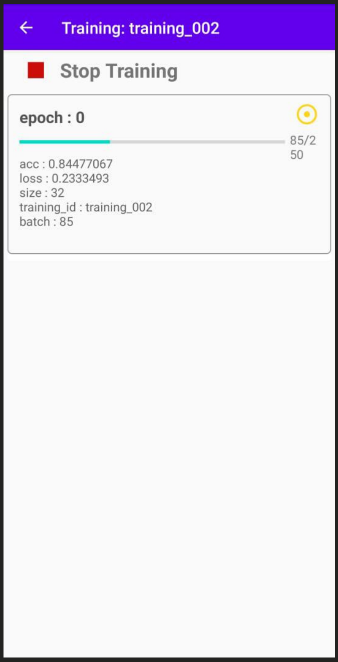
8. Training Epoch Screen


Training: training_001

epoch : 0
250/250
acc : 0.8797191
val_acc : 0.4583333432674408
val_loss : 0.700760006904602
loss : 0.26601407282118905

epoch : 1
51/250
acc : 0.8996885
val_acc : 0.4583333432674408
val_loss : 0.700760006904602
loss : 0.21570803258927812

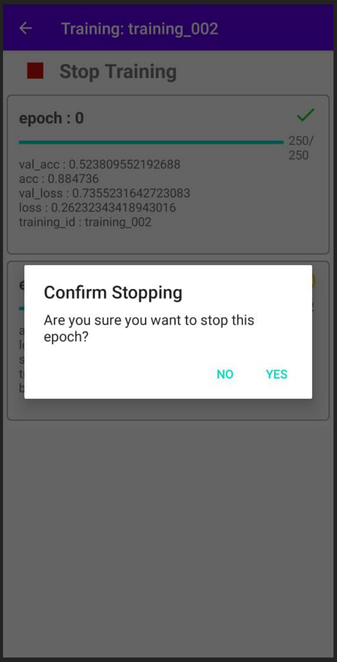
9. Ongoing Epoch Progress


Training: training_002

Stop Training

epoch : 0
85/250
acc : 0.84477067
loss : 0.2333493
size : 32
training_id : training_002
batch : 85

10. Stop training Dialog

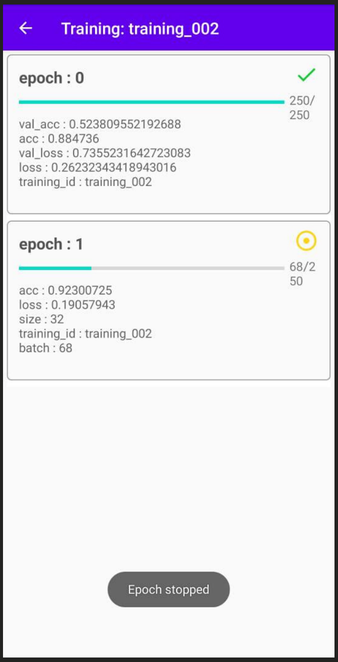

Training: training_002

Stop Training

epoch : 0
250/250
val_acc : 0.523809552192688
acc : 0.884736
val_loss : 0.7355231642723083
loss : 0.26232343418943016
training_id : training_002

Confirm Stopping
Are you sure you want to stop this epoch?
NO YES

11. After Stopping Epoch

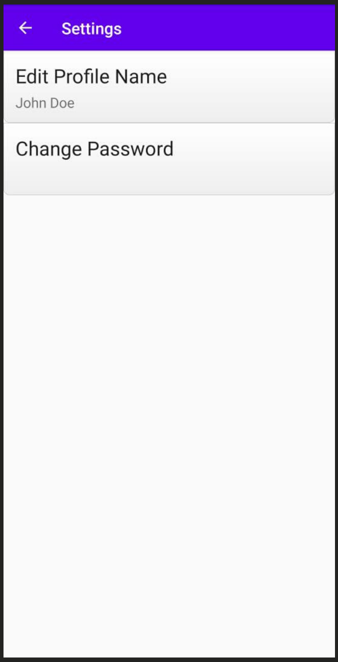

Training: training_002

epoch : 0
250/250
val_acc : 0.523809552192688
acc : 0.884736
val_loss : 0.7355231642723083
loss : 0.26232343418943016
training_id : training_002

epoch : 1
68/250
acc : 0.92300725
loss : 0.19057943
size : 32
training_id : training_002
batch : 68

Epoch stopped

12. Settings Screen

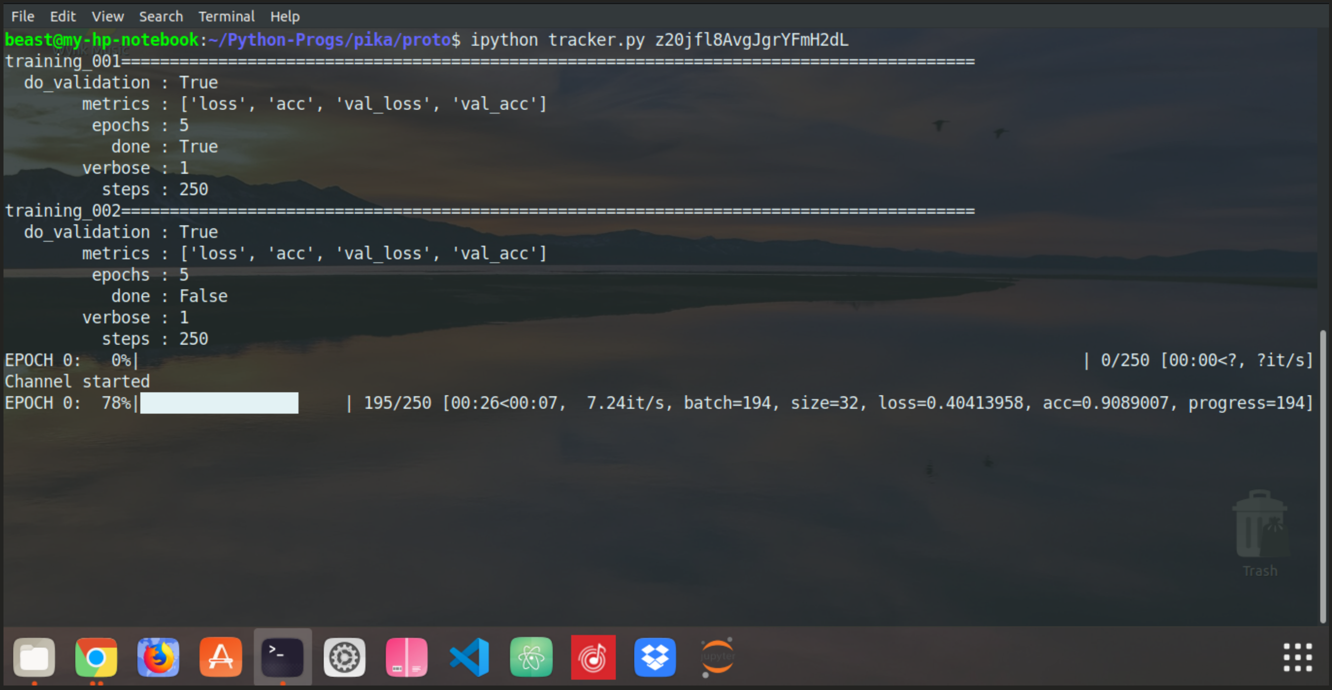

Settings

Edit Profile Name
John Doe

Change Password

Python Tracker CLI client

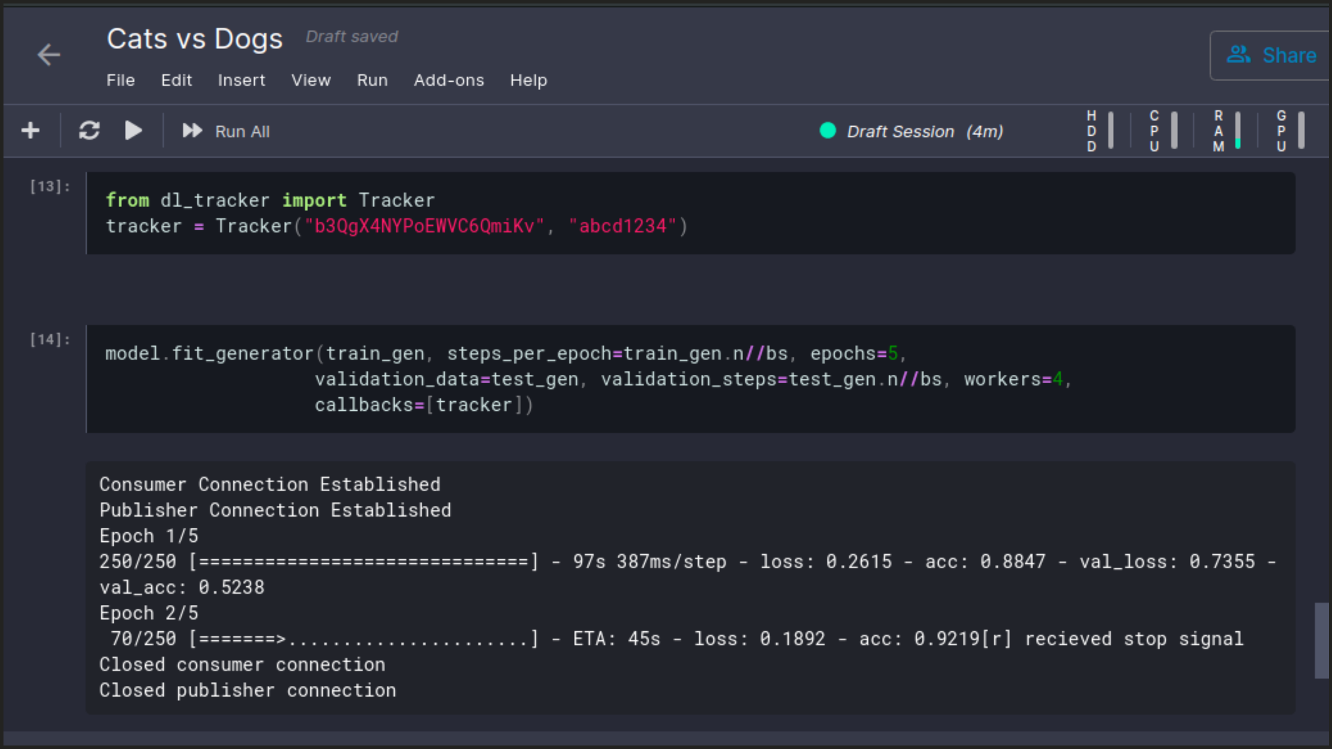
Monitoring an ongoing training using CLI



```
beast@my-hp-notebook:~/Python-Progs/pika/proto$ ipython tracker.py z20jfl8AvgJgrYFmH2dL
training_001=====================================================================================
  do_validation : True
        metrics : ['loss', 'acc', 'val_loss', 'val_acc']
         epochs : 5
           done : True
        verbose : 1
          steps : 250
training_002=====================================================================================
  do_validation : True
        metrics : ['loss', 'acc', 'val_loss', 'val_acc']
         epochs : 5
           done : False
        verbose : 1
          steps : 250
EPOCH 0:   0%|                                                                                | 0/250 [00:00<?, ?it/s]
Channel started
EPOCH 0:  78%|████████████████     | 195/250 [00:26<00:07,  7.24it/s, batch=194, size=32, loss=0.40413958, acc=0.9089007, progress=194]
```

DL-Tracker Callback Package

After receiving 'STOP' signal



```
[13]: from dl_tracker import Tracker
      tracker = Tracker("b3QgX4NYPoEWVC6QmiKv", "abcd1234")
```

```
[14]: model.fit_generator(train_gen, steps_per_epoch=train_gen.n//bs, epochs=5,
                          validation_data=test_gen, validation_steps=test_gen.n//bs, workers=4,
                          callbacks=[tracker])
```

```
Consumer Connection Established
Publisher Connection Established
Epoch 1/5
250/250 [==============================] - 97s 387ms/step - loss: 0.2615 - acc: 0.8847 - val_loss: 0.7355 -
val_acc: 0.5238
Epoch 2/5
 70/250 [=======>......................] - ETA: 45s - loss: 0.1892 - acc: 0.9219[r] recieved stop signal
Closed consumer connection
Closed publisher connection
```

Conclusion and Future Scope

Thus, *a software for real-time, remote tracking and controlling training progress of Deep-Learning models in Tensorflow and Keras has been created*. So far, while testing the application only minor bugs were encountered, which were fixed right away. In the future, I plan to add some more features like manually controlling the learning rate of optimization. Although, the implementational details and viability of this feature are still being explored.

Links to Implementation

The implementation code for all the components of *this project has been Open-Sourced*, while only hiding some minor confidential security details.

- **DL-Tracker Android Client**
  https://github.com/thevarunsharma/DL-Tracker-App

- **DL-Tracker Python Callback Package**
  https://github.com/thevarunsharma/dl_tracker

- **Database Webhook**
  https://github.com/thevarunsharma/Tracker-DB-webhook

*The Python CLI for the tracker client is still very crude, and hence, hasn't yet been open-sourced.