Creating Keras DNN model

Learning Objectives

1. Create input layers for raw features
2. Create feature columns for inputs
3. Create DNN dense hidden layers and output layer
4. Build DNN model tying all of the pieces together
5. Train and evaluate

Introduction

In this notebook, we'll be using Keras to create a DNN model to predict the weight of a baby before it is born.

We'll start by defining the CSV column names, label column, and column defaults for our data inputs. Then, we'll construct a tf.data Dataset of features and the label from the CSV files and create inputs layers for the raw features. Next, we'll set up feature columns for the model inputs and build a deep neural network in Keras. We'll create a custom evaluation metric and build our DNN model. Finally, we'll train and evaluate our model.

Each learning objective will correspond to a **#TODO** in this student lab notebook -- try to complete this notebook first and then review the solution notebook.

Set up environment variables and load necessary libraries

In [1]:
```
!sudo chown -R jupyter:jupyter /home/jupyter/training-data-analyst
```

In [2]:
```
!pip install --user google-cloud-bigquery==1.25.0
```

```
Collecting google-cloud-bigquery==1.25.0
  Downloading google_cloud_bigquery-1.25.0-py2.py3-none-any.whl (169 kB)
     |████████████████████████████████| 169 kB 7.8 MB/s eta 0:00:01
Requirement already satisfied: six<2.0.0dev,>=1.13.0 in /opt/conda/lib/python3.
7/site-packages (from google-cloud-bigquery==1.25.0) (1.16.0)
Requirement already satisfied: protobuf>=3.6.0 in /opt/conda/lib/python3.7/site-
packages (from google-cloud-bigquery==1.25.0) (3.16.0)
Collecting google-resumable-media<0.6dev,>=0.5.0
  Downloading google_resumable_media-0.5.1-py2.py3-none-any.whl (38 kB)
Requirement already satisfied: google-api-core<2.0dev,>=1.15.0 in /opt/conda/li
b/python3.7/site-packages (from google-cloud-bigquery==1.25.0) (1.31.1)
Requirement already satisfied: google-auth<2.0dev,>=1.9.0 in /opt/conda/lib/pyth
on3.7/site-packages (from google-cloud-bigquery==1.25.0) (1.34.0)
Requirement already satisfied: google-cloud-core<2.0dev,>=1.1.0 in /opt/conda/li
b/python3.7/site-packages (from google-cloud-bigquery==1.25.0) (1.7.2)
Requirement already satisfied: googleapis-common-protos<2.0dev,>=1.6.0 in /opt/c
onda/lib/python3.7/site-packages (from google-api-core<2.0dev,>=1.15.0->google-c
loud-bigquery==1.25.0) (1.53.0)
```

```
Requirement already satisfied: packaging>=14.3 in /opt/conda/lib/python3.7/site-
packages (from google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==1.25.0)
(21.0)
Requirement already satisfied: pytz in /opt/conda/lib/python3.7/site-packages (f
rom google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==1.25.0) (2021.1)
Requirement already satisfied: requests<3.0.0dev,>=2.18.0 in /opt/conda/lib/pyth
on3.7/site-packages (from google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery
==1.25.0) (2.25.1)
Requirement already satisfied: setuptools>=40.3.0 in /opt/conda/lib/python3.7/si
te-packages (from google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==1.25.
0) (49.6.0.post20210108)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /opt/conda/lib/python3.
7/site-packages (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0)
(0.2.7)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /opt/conda/lib/python3.
7/site-packages (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0)
(4.2.2)
Requirement already satisfied: rsa<5,>=3.1.4 in /opt/conda/lib/python3.7/site-pa
ckages (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0) (4.7.2)
Requirement already satisfied: pyparsing>=2.0.2 in /opt/conda/lib/python3.7/site
-packages (from packaging>=14.3->google-api-core<2.0dev,>=1.15.0->google-cloud-b
igquery==1.25.0) (2.4.7)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /opt/conda/lib/python3.7/
site-packages (from pyasn1-modules>=0.2.1->google-auth<2.0dev,>=1.9.0->google-cl
oud-bigquery==1.25.0) (0.4.8)
Requirement already satisfied: chardet<5,>=3.0.2 in /opt/conda/lib/python3.7/sit
e-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->go
ogle-cloud-bigquery==1.25.0) (4.0.0)
Requirement already satisfied: idna<3,>=2.5 in /opt/conda/lib/python3.7/site-pac
kages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->google-
cloud-bigquery==1.25.0) (2.10)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /opt/conda/lib/python3.
7/site-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.
0->google-cloud-bigquery==1.25.0) (1.26.6)
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/lib/python3.7/si
te-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->g
oogle-cloud-bigquery==1.25.0) (2021.5.30)
Installing collected packages: google-resumable-media, google-cloud-bigquery
ERROR: pip's dependency resolver does not currently take into account all the pa
ckages that are installed. This behaviour is the source of the following depende
ncy conflicts.
tfx-bsl 1.2.0 requires absl-py<0.13,>=0.9, but you have absl-py 0.13.0 which is
 incompatible.
tfx-bsl 1.2.0 requires google-api-python-client<2,>=1.7.11, but you have google-
api-python-client 2.15.0 which is incompatible.
tfx-bsl 1.2.0 requires google-cloud-bigquery<2.21,>=1.28.0, but you have google-
cloud-bigquery 1.25.0 which is incompatible.
tfx-bsl 1.2.0 requires pyarrow<3,>=1, but you have pyarrow 5.0.0 which is incomp
atible.
tensorflow-transform 1.2.0 requires absl-py<0.13,>=0.9, but you have absl-py 0.1
3.0 which is incompatible.
tensorflow-transform 1.2.0 requires google-cloud-bigquery<2.21,>=1.28.0, but you
have google-cloud-bigquery 1.25.0 which is incompatible.
tensorflow-transform 1.2.0 requires pyarrow<3,>=1, but you have pyarrow 5.0.0 wh
ich is incompatible.
google-cloud-storage 1.41.1 requires google-resumable-media<3.0dev,>=1.3.0; pyth
on_version >= "3.6", but you have google-resumable-media 0.5.1 which is incompat
ible.
Successfully installed google-cloud-bigquery-1.25.0 google-resumable-media-0.5.1
```

Note: Restart your kernel to use updated packages.

Kindly ignore the deprecation warnings and incompatibility errors related to google-cloud-storage.

Import necessary libraries.

In [1]:
```
from google.cloud import bigquery
import pandas as pd
import datetime
import os
import shutil
import matplotlib.pyplot as plt
import tensorflow as tf
print(tf.__version__)
```

```
2.5.0
```

Set environment variables so that we can use them throughout the notebook.

In [2]:
```
%%bash
export PROJECT=$(gcloud config list project --format "value(core.project)")
echo "Your current GCP Project Name is: "$PROJECT
```

```
Your current GCP Project Name is: qwiklabs-gcp-04-8d7a9e25b038
```

In [8]:
```
PROJECT = "qwiklabs-gcp-04-8d7a9e25b038"  # Replace with your PROJECT
```

Create ML datasets by sampling using BigQuery

We'll begin by sampling the BigQuery data to create smaller datasets. Let's create a BigQuery client that we'll use throughout the lab.

In [9]:
```
bq = bigquery.Client(project = PROJECT)
```

We need to figure out the right way to divide our hash values to get our desired splits. To do that we need to define some values to hash within the module. Feel free to play around with these values to get the perfect combination.

In [10]:
```
modulo_divisor = 100
train_percent = 80.0
eval_percent = 10.0

train_buckets = int(modulo_divisor * train_percent / 100.0)
eval_buckets = int(modulo_divisor * eval_percent / 100.0)
```

We can make a series of queries to check if our bucketing values result in the correct sizes of each of our dataset splits and then adjust accordingly. Therefore, to make our code more compact and reusable, let's define a function to return the head of a dataframe produced from our queries up to a certain number of rows.

In [11]:
```
def display_dataframe_head_from_query(query, count=10):
    """Displays count rows from dataframe head from query.
```

```
    Args:
        query: str, query to be run on BigQuery, results stored in dataframe.
        count: int, number of results from head of dataframe to display.
    Returns:
        Dataframe head with count number of results.
    """
    df = bq.query(
        query + " LIMIT {limit}".format(
            limit=count)).to_dataframe()

    return df.head(count)
```

For our first query, we're going to use the original query above to get our label, features, and columns to combine into our hash which we will use to perform our repeatable splitting. There are only a limited number of years, months, days, and states in the dataset. Let's see what the hash values are. We will need to include all of these extra columns to hash on to get a fairly uniform spread of the data. Feel free to try less or more in the hash and see how it changes your results.

In [12]:

```
# Get label, features, and columns to hash and split into buckets
hash_cols_fixed_query = """
SELECT
    weight_pounds,
    is_male,
    mother_age,
    plurality,
    gestation_weeks,
    year,
    month,
    CASE
        WHEN day IS NULL THEN
            CASE
                WHEN wday IS NULL THEN 0
                ELSE wday
            END
        ELSE day
    END AS date,
    IFNULL(state, "Unknown") AS state,
    IFNULL(mother_birth_state, "Unknown") AS mother_birth_state
FROM
    publicdata.samples.natality
WHERE
    year > 2000
    AND weight_pounds > 0
    AND mother_age > 0
    AND plurality > 0
    AND gestation_weeks > 0
"""

display_dataframe_head_from_query(hash_cols_fixed_query)
```

Out[12]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	year	month	date	state	m
0	7.063611	True	32	1	37	2001	12	3	CO	
1	4.687028	True	30	3	33	2001	6	5	IN	

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	year	month	date	state	m
2	7.561856	True	20	1	39	2001	4	5	MN	
3	7.561856	True	31	1	37	2001	10	5	MS	
4	7.312733	True	32	1	40	2001	11	3	MO	
5	7.627994	False	30	1	40	2001	10	5	NY	
6	7.251004	True	33	1	37	2001	11	5	WA	
7	7.500126	False	23	1	39	2001	9	2	OK	
8	7.125340	False	33	1	39	2001	1	4	TX	
9	7.749249	True	31	1	39	2001	1	1	TX	

Using `COALESCE` would provide the same result as the nested `CASE WHEN`. This is preferable when all we want is the first non-null instance. To be precise the `CASE WHEN` would become `COALESCE(wday, day, 0) AS date`. You can read more about it here.

Next query will combine our hash columns and will leave us just with our label, features, and our hash values.

In [13]:

```
data_query = """
SELECT
    weight_pounds,
    is_male,
    mother_age,
    plurality,
    gestation_weeks,
    FARM_FINGERPRINT(
        CONCAT(
            CAST(year AS STRING),
            CAST(month AS STRING),
            CAST(date AS STRING),
            CAST(state AS STRING),
            CAST(mother_birth_state AS STRING)
        )
    ) AS hash_values
FROM
    ({CTE_hash_cols_fixed})
""".format(CTE_hash_cols_fixed=hash_cols_fixed_query)

display_dataframe_head_from_query(data_query)
```

Out[13]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
0	7.063611	True	32	1	37	4762325092919148672
1	4.687028	True	30	3	33	2341060194216507348
2	7.561856	True	20	1	39	-8842767231851202242
3	7.561856	True	31	1	37	7957807816914159435
4	7.312733	True	32	1	40	-5961624242430066305
5	7.627994	False	30	1	40	5493295634082918412
6	7.251004	True	33	1	37	-2988893757655690534

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
7	7.500126	False	23	1	39	-6735199252008114417
8	7.125340	False	33	1	39	-3514093303120687641
9	7.749249	True	31	1	39	2175328516857391398

The next query is going to find the counts of each of the unique 657484 `hash_values` . This will be our first step at making actual hash buckets for our split via the `GROUP BY` .

In [14]:
```
# Get the counts of each of the unique hash of our splitting column
first_bucketing_query = """
SELECT
    hash_values,
    COUNT(*) AS num_records
FROM
    ({CTE_data})
GROUP BY
    hash_values
""".format(CTE_data=data_query)

display_dataframe_head_from_query(first_bucketing_query)
```

Out[14]:

	hash_values	num_records
0	2341060194216507348	696
1	-8842767231851202242	515
2	7957807816914159435	369
3	-746421027886559730	167
4	-7566476151165360246	87
5	315818780995586851	400
6	8711610669332498583	7
7	-4170581330234584329	30
8	-4741381250325891292	1357
9	19168170974013054	503

The query below performs a second layer of bucketing where now for each of these bucket indices we count the number of records.

In [15]:
```
# Get the number of records in each of the hash buckets
second_bucketing_query = """
SELECT
    ABS(MOD(hash_values, {modulo_divisor})) AS bucket_index,
    SUM(num_records) AS num_records
FROM
    ({CTE_first_bucketing})
GROUP BY
    ABS(MOD(hash_values, {modulo_divisor}))
""".format(
    CTE_first_bucketing=first_bucketing_query, modulo_divisor=modulo_divisor)
```

```
display_dataframe_head_from_query(second_bucketing_query)
```

Out[15]:

	bucket_index	num_records
0	35	250505
1	74	480999
2	85	368045
3	77	401941
4	23	559019
5	36	246041
6	14	251675
7	21	247072
8	53	230298
9	54	256517

The number of records is hard for us to easily understand the split, so we will normalize the count into percentage of the data in each of the hash buckets in the next query.

In [16]:

```
# Calculate the overall percentages
percentages_query = """
SELECT
    bucket_index,
    num_records,
    CAST(num_records AS FLOAT64) / (
    SELECT
        SUM(num_records)
    FROM
        ({CTE_second_bucketing})) AS percent_records
FROM
    ({CTE_second_bucketing})
""".format(CTE_second_bucketing=second_bucketing_query)

display_dataframe_head_from_query(percentages_query)
```

Out[16]:

	bucket_index	num_records	percent_records
0	72	229541	0.006953
1	41	244850	0.007417
2	70	285539	0.008650
3	83	411258	0.012458
4	33	410226	0.012427
5	52	204972	0.006209
6	89	256482	0.007770
7	32	423507	0.012829
8	79	403701	0.012229

	bucket_index	num_records	percent_records
9	75	367455	0.011131

We'll now select the range of buckets to be used in training.

In [17]:

```
# Choose hash buckets for training and pull in their statistics
train_query = """
SELECT
    *,
    "train" AS dataset_name
FROM
    ({CTE_percentages})
WHERE
    bucket_index >= 0
    AND bucket_index < {train_buckets}
""".format(
    CTE_percentages=percentages_query,
    train_buckets=train_buckets)

display_dataframe_head_from_query(train_query)
```

Out[17]:

	bucket_index	num_records	percent_records	dataset_name
0	48	370308	0.011218	train
1	17	222562	0.006742	train
2	8	370758	0.011231	train
3	4	398118	0.012060	train
4	62	426834	0.012930	train
5	51	180001	0.005453	train
6	57	453019	0.013723	train
7	29	453175	0.013728	train
8	68	197797	0.005992	train
9	45	265930	0.008056	train

We'll do the same by selecting the range of buckets to be used evaluation.

In [18]:

```
# Choose hash buckets for validation and pull in their statistics
eval_query = """
SELECT
    *,
    "eval" AS dataset_name
FROM
    ({CTE_percentages})
WHERE
    bucket_index >= {train_buckets}
    AND bucket_index < {cum_eval_buckets}
""".format(
    CTE_percentages=percentages_query,
    train_buckets=train_buckets,
    cum_eval_buckets=train_buckets + eval_buckets)
```

```
display_dataframe_head_from_query(eval_query)
```

Out[18]:

	bucket_index	num_records	percent_records	dataset_name
0	83	411258	0.012458	eval
1	89	256482	0.007770	eval
2	80	312489	0.009466	eval
3	82	468179	0.014182	eval
4	88	423809	0.012838	eval
5	84	341155	0.010334	eval
6	86	274489	0.008315	eval
7	81	233538	0.007074	eval
8	85	368045	0.011149	eval
9	87	523881	0.015870	eval

Lastly, we'll select the hash buckets to be used for the test split.

In [19]:

```
# Choose hash buckets for testing and pull in their statistics
test_query = """
SELECT
    *,
    "test" AS dataset_name
FROM
    ({CTE_percentages})
WHERE
    bucket_index >= {cum_eval_buckets}
    AND bucket_index < {modulo_divisor}
""".format(
    CTE_percentages=percentages_query,
    cum_eval_buckets=train_buckets + eval_buckets,
    modulo_divisor=modulo_divisor)

display_dataframe_head_from_query(test_query)
```

Out[19]:

	bucket_index	num_records	percent_records	dataset_name
0	94	431001	0.013056	test
1	90	286465	0.008678	test
2	93	215710	0.006534	test
3	98	374697	0.011351	test
4	97	480790	0.014564	test
5	91	333267	0.010096	test
6	99	223334	0.006765	test
7	96	529357	0.016036	test
8	95	313544	0.009498	test

	bucket_index	num_records	percent_records	dataset_name
9	92	336735	0.010201	test

In the below query, we'll `UNION ALL` all of the datasets together so that all three sets of hash buckets will be within one table. We added `dataset_id` so that we can sort on it in the query after.

In [20]:

```
# Union the training, validation, and testing dataset statistics
union_query = """
SELECT
    0 AS dataset_id,
    *
FROM
    ({CTE_train})
UNION ALL
SELECT
    1 AS dataset_id,
    *
FROM
    ({CTE_eval})
UNION ALL
SELECT
    2 AS dataset_id,
    *
FROM
    ({CTE_test})
""".format(CTE_train=train_query, CTE_eval=eval_query, CTE_test=test_query)

display_dataframe_head_from_query(union_query)
```

Out[20]:

	dataset_id	bucket_index	num_records	percent_records	dataset_name
0	0	72	229541	0.006953	train
1	0	41	244850	0.007417	train
2	0	70	285539	0.008650	train
3	0	33	410226	0.012427	train
4	0	52	204972	0.006209	train
5	0	32	423507	0.012829	train
6	0	79	403701	0.012229	train
7	0	75	367455	0.011131	train
8	0	65	289303	0.008764	train
9	0	0	277395	0.008403	train

Lastly, we'll show the final split between train, eval, and test sets. We can see both the number of records and percent of the total data. It is really close to that we were hoping to get.

In [21]:

```
# Show final splitting and associated statistics
split_query = """
SELECT
```

```
    dataset_id,
    dataset_name,
    SUM(num_records) AS num_records,
    SUM(percent_records) AS percent_records
FROM
    ({CTE_union})
GROUP BY
    dataset_id,
    dataset_name
ORDER BY
    dataset_id
""".format(CTE_union=union_query)

display_dataframe_head_from_query(split_query)
```

Out[21]:

	dataset_id	dataset_name	num_records	percent_records
0	0	train	25873134	0.783765
1	1	eval	3613325	0.109457
2	2	test	3524900	0.106778

Now that we know that our splitting values produce a good global splitting on our data, here's a way to get a well-distributed portion of the data in such a way that the train, eval, test sets do not overlap and takes a subsample of our global splits.

In [22]:

```
# every_n allows us to subsample from each of the hash values
# This helps us get approximately the record counts we want
every_n = 1000

splitting_string = "ABS(MOD(hash_values, {0} * {1}))".format(every_n, modulo_div

def create_data_split_sample_df(query_string, splitting_string, lo, up):
    """Creates a dataframe with a sample of a data split.

    Args:
        query_string: str, query to run to generate splits.
        splitting_string: str, modulo string to split by.
        lo: float, lower bound for bucket filtering for split.
        up: float, upper bound for bucket filtering for split.
    Returns:
        Dataframe containing data split sample.
    """
    query = "SELECT * FROM ({0}) WHERE {1} >= {2} and {1} < {3}".format(
        query_string, splitting_string, int(lo), int(up))

    df = bq.query(query).to_dataframe()

    return df

train_df = create_data_split_sample_df(
    data_query, splitting_string,
    lo=0, up=train_percent)

eval_df = create_data_split_sample_df(
    data_query, splitting_string,
    lo=train_percent, up=train_percent + eval_percent)
```

```
test_df = create_data_split_sample_df(
    data_query, splitting_string,
    lo=train_percent + eval_percent, up=modulo_divisor)

print("There are {} examples in the train dataset.".format(len(train_df)))
print("There are {} examples in the validation dataset.".format(len(eval_df)))
print("There are {} examples in the test dataset.".format(len(test_df)))
```

```
There are 7733 examples in the train dataset.
There are 1037 examples in the validation dataset.
There are 561 examples in the test dataset.
```

Preprocess data using Pandas

We'll perform a few preprocessing steps to the data in our dataset. Let's add extra rows to simulate the lack of ultrasound. That is we'll duplicate some rows and make the `is_male` field be `Unknown` . Also, if there is more than child we'll change the `plurality` to `Multiple(2+)` . While we're at it, we'll also change the plurality column to be a string. We'll perform these operations below.

Let's start by examining the training dataset as is.

In [23]:

```
train_df.head()
```

Out[23]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
0	7.625790	True	38	1	38	968753748112600022
1	7.363440	True	38	1	40	-6784884401981100070
2	7.561856	False	35	1	39	-6373125659687700000
3	5.249206	False	33	2	38	4896699230184800022
4	4.687028	True	15	1	36	5815559452861000023

Also, notice that there are some very important numeric fields that are missing in some rows (the count in Pandas doesn't count missing data)

In [24]:

```
train_df.describe()
```

Out[24]:

	weight_pounds	mother_age	plurality	gestation_weeks	hash_values
count	7733.000000	7733.000000	7733.000000	7733.000000	7.733000e+03
mean	7.264415	28.213371	1.035691	38.691064	-2.984870e+17
std	1.303220	6.134232	0.201568	2.531921	5.590715e+18
min	0.562179	13.000000	1.000000	18.000000	-9.210618e+18
25%	6.624891	23.000000	1.000000	38.000000	-6.781866e+18
50%	7.345803	28.000000	1.000000	39.000000	5.057323e+17
75%	8.062305	33.000000	1.000000	40.000000	4.896699e+18
max	11.563246	48.000000	4.000000	47.000000	9.203641e+18

It is always crucial to clean raw data before using in machine learning, so we have a preprocessing step. We'll define a `preprocess` function below. Note that the mother's age is an input to our model so users will have to provide the mother's age; otherwise, our service won't work. The features we use for our model were chosen because they are such good predictors and because they are easy enough to collect.

In [25]:

```
def preprocess(df):
    """ Preprocess pandas dataframe for augmented babyweight data.

    Args:
        df: Dataframe containing raw babyweight data.
    Returns:
        Pandas dataframe containing preprocessed raw babyweight data as well
            as simulated no ultrasound data masking some of the original data.
    """
    # Clean up raw data
    # Filter out what we don"t want to use for training
    df = df[df.weight_pounds > 0]
    df = df[df.mother_age > 0]
    df = df[df.gestation_weeks > 0]
    df = df[df.plurality > 0]

    # Modify plurality field to be a string
    twins_etc = dict(zip([1,2,3,4,5],
                    ["Single(1)",
                     "Twins(2)",
                     "Triplets(3)",
                     "Quadruplets(4)",
                     "Quintuplets(5)"]))
    df["plurality"].replace(twins_etc, inplace=True)

    # Clone data and mask certain columns to simulate lack of ultrasound
    no_ultrasound = df.copy(deep=True)

    # Modify is_male
    no_ultrasound["is_male"] = "Unknown"

    # Modify plurality
    condition = no_ultrasound["plurality"] != "Single(1)"
    no_ultrasound.loc[condition, "plurality"] = "Multiple(2+)"

    # Concatenate both datasets together and shuffle
    return pd.concat(
        [df, no_ultrasound]).sample(frac=1).reset_index(drop=True)
```

Let's process the train, eval, test set and see a small sample of the training data after our preprocessing:

In [26]:

```
train_df = preprocess(train_df)
eval_df = preprocess(eval_df)
test_df = preprocess(test_df)
```

In [27]:

```
train_df.head()
```

Out[27]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
0	6.937947	False	34	Single(1)	37	4896699230184800022
1	7.936641	False	36	Single(1)	40	-8560578499498900025
2	6.999677	Unknown	26	Single(1)	41	-4795548143318100049
3	7.251004	False	33	Single(1)	39	-6784884401981100070
4	7.874912	Unknown	34	Single(1)	40	-6471666456918200072

In [28]:

```
train_df.tail()
```

Out[28]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
15461	7.345803	Unknown	19	Single(1)	39	-4614303140002600076
15462	7.063611	Unknown	30	Single(1)	38	6810056410456600040
15463	5.952481	False	34	Single(1)	36	780565305641800050
15464	7.125340	Unknown	30	Single(1)	43	505732274561700014
15465	0.914918	Unknown	22	Single(1)	21	780565305641800050

Let's look again at a summary of the dataset. Note that we only see numeric columns, so `plurality` does not show up.

In [29]:

```
train_df.describe()
```

Out[29]:

	weight_pounds	mother_age	gestation_weeks	hash_values
count	15466.000000	15466.000000	15466.000000	1.546600e+04
mean	7.264415	28.213371	38.691064	-2.984870e+17
std	1.303178	6.134034	2.531839	5.590534e+18
min	0.562179	13.000000	18.000000	-9.210618e+18
25%	6.624891	23.000000	38.000000	-6.781866e+18
50%	7.345803	28.000000	39.000000	5.057323e+17
75%	8.062305	33.000000	40.000000	4.896699e+18
max	11.563246	48.000000	47.000000	9.203641e+18

Write to .csv files

In the final versions, we want to read from files, not Pandas dataframes. So, we write the Pandas dataframes out as csv files. Using csv files gives us the advantage of shuffling during read. This is important for distributed training because some workers might be slower than others, and shuffling the data helps prevent the same data from being assigned to the slow workers.

In [30]:

```
# Define columns
```

```
columns = ["weight_pounds",
           "is_male",
           "mother_age",
           "plurality",
           "gestation_weeks"]

# Write out CSV files
train_df.to_csv(
    path_or_buf="train.csv", columns=columns, header=False, index=False)
eval_df.to_csv(
    path_or_buf="eval.csv", columns=columns, header=False, index=False)
test_df.to_csv(
    path_or_buf="test.csv", columns=columns, header=False, index=False)
```

In [31]:

```
%%bash
wc -l *.csv
```

```
  2074 eval.csv
  1122 test.csv
 15466 train.csv
 18662 total
```

In [32]:

```
%%bash
head *.csv
```

```
==> eval.csv <==
9.7775013197,Unknown,33,Single(1),38
7.06140625186,True,20,Single(1),38
6.87621795178,True,17,Single(1),40
7.936641432,False,28,Single(1),39
8.93754010148,True,31,Single(1),42
6.0009827716399995,False,27,Single(1),46
7.68751907594,Unknown,29,Single(1),35
7.41414587106,True,32,Single(1),38
9.56144830294,True,36,Single(1),41
6.3735639944199995,True,19,Single(1),36

==> test.csv <==
7.81318256528,True,24,Single(1),39
7.50012615324,Unknown,22,Single(1),39
4.62529825676,False,18,Single(1),38
8.811876612139999,True,39,Single(1),41
9.62538235892,True,29,Single(1),40
5.1257475915,Unknown,21,Single(1),24
5.56226287026,Unknown,35,Single(1),39
8.62448368944,True,28,Single(1),41
6.2501051276999995,False,22,Single(1),30
7.25100379718,True,25,Single(1),39

==> train.csv <==
6.93794738514,False,34,Single(1),37
7.936641432,False,36,Single(1),40
6.9996768185,Unknown,26,Single(1),41
7.25100379718,False,33,Single(1),39
7.87491199864,Unknown,34,Single(1),40
7.87491199864,Unknown,26,Single(1),39
6.6248909731,Unknown,30,Single(1),42
4.87442061282,True,26,Single(1),35
```

```
5.8135898489399995,Unknown,33,Multiple(2+),37
6.686620406459999,False,38,Single(1),37
```

In [33]:

```
%%bash
tail *.csv
```

```
==> eval.csv <==
6.75055446244,Unknown,32,Single(1),36
7.31273323054,Unknown,29,Single(1),38
8.062304921339999,Unknown,20,Single(1),39
8.12623897732,Unknown,23,Single(1),41
6.5367060683,False,21,Single(1),38
8.62448368944,False,28,Single(1),39
8.3114272774,Unknown,25,Single(1),41
8.12623897732,Unknown,23,Single(1),39
8.18796841068,True,26,Single(1),41
9.18666245754,Unknown,40,Single(1),40

==> test.csv <==
7.3744626639,Unknown,30,Single(1),39
6.87621795178,False,36,Single(1),38
7.83522879148,False,23,Single(1),40
6.9996768185,False,21,Single(1),39
7.25100379718,False,41,Twins(2),40
7.0437692708999995,Unknown,38,Single(1),40
7.5618555866,True,40,Twins(2),43
7.12534030784,True,40,Twins(2),43
7.31273323054,True,24,Single(1),40
7.5618555866,False,26,Single(1),37

==> train.csv <==
6.4992274837599995,Unknown,42,Single(1),36
8.1901730333,Unknown,38,Single(1),40
5.1257475915,False,25,Single(1),35
8.062304921339999,True,25,Single(1),39
7.62578964258,True,26,Single(1),37
7.34580256984,Unknown,19,Single(1),39
7.06361087448,Unknown,30,Single(1),38
5.952481074,False,34,Single(1),36
7.12534030784,Unknown,30,Single(1),43
0.9149183873,Unknown,22,Single(1),21
```

In [34]:

```
%%bash
ls *.csv
```

```
eval.csv
test.csv
train.csv
```

In [35]:

```
%%bash
head -5 *.csv
```

```
==> eval.csv <==
9.7775013197,Unknown,33,Single(1),38
7.06140625186,True,20,Single(1),38
6.87621795178,True,17,Single(1),40
7.936641432,False,28,Single(1),39
8.93754010148,True,31,Single(1),42
```

```
==> test.csv <==
7.81318256528,True,24,Single(1),39
7.50012615324,Unknown,22,Single(1),39
4.62529825676,False,18,Single(1),38
8.811876612139999,True,39,Single(1),41
9.62538235892,True,29,Single(1),40

==> train.csv <==
6.93794738514,False,34,Single(1),37
7.936641432,False,36,Single(1),40
6.9996768185,Unknown,26,Single(1),41
7.25100379718,False,33,Single(1),39
7.87491199864,Unknown,34,Single(1),40
```

Create Keras model

Set CSV Columns, label column, and column defaults.

Now that we have verified that our CSV files exist, we need to set a few things that we will be using in our input function.

- `CSV_COLUMNS` is going to be our header name of our column. Make sure that they are in the same order as in the CSV files
- `LABEL_COLUMN` is the header name of the column that is our label. We will need to know this to pop it from our features dictionary.
- `DEFAULTS` is a list with the same length as `CSV_COLUMNS` , i.e. there is a default for each column in our CSVs. Each element is a list itself with the default value for that CSV column.

In [36]:

```
# Determine CSV, label, and key columns
# Create list of string column headers, make sure order matches.
CSV_COLUMNS = ["weight_pounds",
               "is_male",
               "mother_age",
               "plurality",
               "gestation_weeks"]

# Add string name for label column
LABEL_COLUMN = "weight_pounds"

# Set default values for each CSV column as a list of lists.
# Treat is_male and plurality as strings.
DEFAULTS = [[0.0], ["null"], [0.0], ["null"], [0.0]]
```

Make dataset of features and label from CSV files.

Next, we will write an input_fn to read the data. Since we are reading from CSV files we can save ourselves from trying to recreate the wheel and can use `tf.data.experimental.make_csv_dataset` . This will create a CSV dataset object. However we will need to divide the columns up into features and a label. We can do this by applying the map method to our dataset and popping our label column off of our dictionary of feature tensors.

In [37]:
```
def features_and_labels(row_data):
    """Splits features and labels from feature dictionary.

    Args:
        row_data: Dictionary of CSV column names and tensor values.
    Returns:
        Dictionary of feature tensors and label tensor.
    """
    label = row_data.pop(LABEL_COLUMN)

    return row_data, label  # features, label


def load_dataset(pattern, batch_size=1, mode='eval'):
    """Loads dataset using the tf.data API from CSV files.

    Args:
        pattern: str, file pattern to glob into list of files.
        batch_size: int, the number of examples per batch.
        mode: 'train' | 'eval' to determine if training or evaluating.
    Returns:
        `Dataset` object.
    """
    # Make a CSV dataset
    dataset = tf.data.experimental.make_csv_dataset(
        file_pattern=pattern,
        batch_size=batch_size,
        column_names=CSV_COLUMNS,
        column_defaults=DEFAULTS,
        ignore_errors=True)

    # Map dataset to features and label
    dataset = dataset.map(map_func=features_and_labels)  # features, label

    # Shuffle and repeat for training
    if mode == 'train':
        dataset = dataset.shuffle(buffer_size=1000).repeat()

    # Take advantage of multi-threading; 1=AUTOTUNE
    dataset = dataset.prefetch(buffer_size=1)

    return dataset
```

Create input layers for raw features.

We'll need to get the data to read in by our input function to our model function, but just how do we go about connecting the dots? We can use Keras input layers (tf.Keras.layers.Input) by defining:

- shape: A shape tuple (integers), not including the batch size. For instance, shape=(32,) indicates that the expected input will be batches of 32-dimensional vectors. Elements of this tuple can be None; 'None' elements represent dimensions where the shape is not known.
- name: An optional name string for the layer. Should be unique in a model (do not reuse the same name twice). It will be autogenerated if it isn't provided.
- dtype: The data type expected by the input, as a string (float32, float64, int32...)

Lab Task #1: Creating input layers for raw features.

In [42]:

```
# TODO 1
def create_input_layers():
    """
    Args: create an input layer of each features,
    return: Dictionary for `tf.Keras.layers.Input` layers for each feature.
    """
    inputs = {
        colname: tf.keras.layers.Input(name=colname,
                                       shape=(),dtype="float32")
        for colname in ["mother_age", "gestation_weeks"]}
    inputs.update({
        colname: tf.keras.layers.Input(name=colname, shape=(), dtype="string")
        for colname in ["is_male", "plurality"]})

    return inputs
```

Create feature columns for inputs.

Next, define the feature columns. `mother_age` and `gestation_weeks` should be numeric. The others, `is_male` and `plurality`, should be categorical. Remember, only dense feature columns can be inputs to a DNN.

Lab Task #2: Creating feature columns.

In [43]:

```
# TODO 2
def categorical_fc(name, values):
    """
    Args:
    name: name of features
    values: list of strings of categorical values
    output:
    feature indicator column categorical features
    """
    cat_column=tf.feature_column.categorical_column_with_vocabulary_list(key=nam
    return tf.feature_column.indicator_column(categorical_column=cat_column)

def create_feature_columns():
    """
    Desc: create dictionary of feature columns from inputs
    Args: Null
    Output: dictionary of Feature columns
    """
    feature_columns = {
        colname : tf.feature_column.numeric_column(key = colname)
        for colname in ["mother_age", "gestation_weeks"]
    }
    feature_columns["is_male"] = categorical_fc(
        "is_male", ["True", "False", "Unknown"])
    feature_columns["plurality"] = categorical_fc(
        "plurality", ["Single(1)", "Twins(2)", "Triplets(3)",
                      "Quadruplets(4)", "Quintuplets(5)", "Multiple(2+)"])

    return feature_columns
```

Create DNN dense hidden layers and output layer.

So we've figured out how to get our inputs ready for machine learning but now we need to connect them to our desired output. Our model architecture is what links the two together. Let's create some hidden dense layers beginning with our inputs and end with a dense output layer. This is regression so make sure the output layer activation is correct and that the shape is right.

Lab Task #3: Creating DNN dense hidden layers.

In [54]:

```
# TODO 3
def get_model_outputs(inputs):
    """Creates model architecture and returns outputs.

    Args:
        inputs: Dense tensor used as inputs to model.
    Returns:
        Dense tensor output from the model.
    """
    # Create two hidden layers of [64, 32] just in like the BQML DNN
    h1 = tf.keras.layers.Dense(64, activation="relu", name="h1")(inputs)
    h2 = tf.keras.layers.Dense(32, activation="relu", name="h2")(h1)



    # Final output is a linear activation because this is regression
    output = tf.keras.layers.Dense(
        units=1, activation="linear", name="weight")(h2)

    return output
```

Create custom evaluation metric.

We want to make sure that we have some useful way to measure model performance for us. Since this is regression, we would like to know the RMSE of the model on our evaluation dataset, however, this does not exist as a standard evaluation metric, so we'll have to create our own by using the true and predicted labels.

In [55]:

```
def rmse(y_true, y_pred):
    """Calculates RMSE evaluation metric.

    Args:
        y_true: tensor, true labels.
        y_pred: tensor, predicted labels.
    Returns:
        Tensor with value of RMSE between true and predicted labels.
    """
    return tf.sqrt(tf.reduce_mean((y_pred - y_true) ** 2))
```

Build DNN model tying all of the pieces together.

Excellent! We've assembled all of the pieces, now we just need to tie them all together into a Keras Model. This is a simple feedforward model with no branching, side inputs, etc. so we could have used Keras' Sequential Model API but just for fun we're going to use Keras'

Functional Model API. Here we will build the model using tf.keras.models.Model giving our inputs and outputs and then compile our model with an optimizer, a loss function, and evaluation metrics.

Lab Task #4: Building DNN model.

In [56]:

```
# TODO 4
# create model
def build_dnn_model():
    """Builds simple DNN using Keras Functional API.

    Returns:
        `tf.keras.models.Model` object.
    """
    # Create input layer
    inputs = create_input_layers()

    #create feature columns
    feature_columns = create_feature_columns()
    # The constructor for DenseFeatures takes a list of numeric columns
    # The Functional API in Keras requires: LayerConstructor()(inputs)
    dnn_inputs = tf.keras.layers.DenseFeatures(feature_columns=feature_columns.v

    #Get model outputs
    output = get_model_outputs(dnn_inputs)

    #create model
    model = tf.keras.models.Model(inputs = inputs, outputs = output)
    model.compile(optimizer="adam", loss="mse", metrics=[rmse, "mse"])

    return model

print("Here is our DNN architecture so far:\n")
model = build_dnn_model()
print(model.summary())
```

```
Here is our DNN architecture so far:

Model: "model"
________________________________________________________________________________
__________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================
==================
gestation_weeks (InputLayer)    [(None,)]            0
________________________________________________________________________________
__________________
is_male (InputLayer)            [(None,)]            0
________________________________________________________________________________
__________________
mother_age (InputLayer)         [(None,)]            0
________________________________________________________________________________
__________________
plurality (InputLayer)          [(None,)]            0
________________________________________________________________________________
__________________
dense_features_4 (DenseFeatures (None, 11)           0           gestation_weeks
[0][0]
                                                                 is_male[0][0]
```

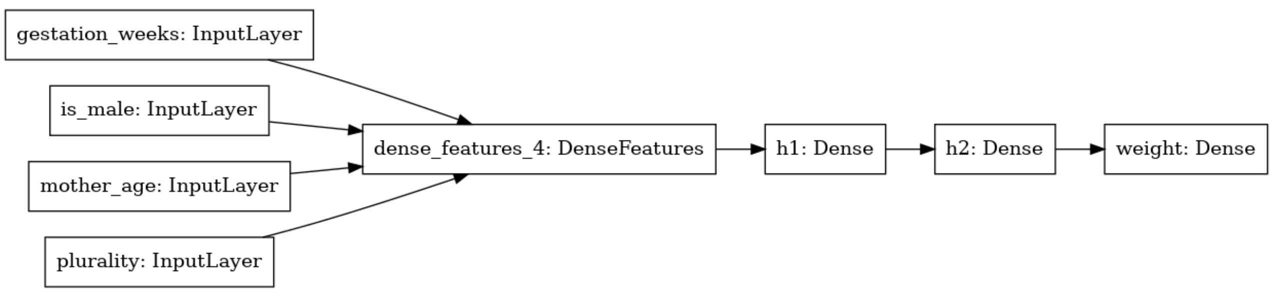
```
                                                                 mother_age[0]
[0]
                                                                 plurality[0][0]
________________________________________________________________________________
__________________
h1 (Dense)                      (None, 64)           768         dense_features_
4[0][0]
________________________________________________________________________________
__________________
h2 (Dense)                      (None, 32)           2080        h1[0][0]
________________________________________________________________________________
__________________
weight (Dense)                  (None, 1)            33          h2[0][0]
================================================================================
==================
Total params: 2,881
Trainable params: 2,881
Non-trainable params: 0
________________________________________________________________________________
__________________
None
```

We can visualize the DNN using the Keras plot_model utility.

In [57]:

```
tf.keras.utils.plot_model(
    model=model, to_file="dnn_model.png", show_shapes=False, rankdir="LR")
```

Out[57]:



Run and evaluate model

Train and evaluate.

We've built our Keras model using our inputs from our CSV files and the architecture we designed. Let's now run our model by training our model parameters and periodically running an evaluation to track how well we are doing on outside data as training goes on. We'll need to load both our train and eval datasets and send those to our model through the fit method. Make sure you have the right pattern, batch size, and mode when loading the data.

Lab Task #5: Training and evaluating the model.

In [58]:

```
# TODO 5
TRAIN_BATCH_SIZE = 32
NUM_TRAIN_EXAMPLES = 10000 * 5  # training dataset repeats, it'll wrap around
NUM_EVALS = 5  # how many times to evaluate
# Enough to get a reasonable sample, but not so much that it slows down
NUM_EVAL_EXAMPLES = 10000
```

```
# TODO -- Your code here.
trainds = load_dataset(
    pattern = "train*",
    batch_size=TRAIN_BATCH_SIZE,
    mode = 'train')
evalds = load_dataset(
    pattern = "eval*",
    batch_size=1000,
    mode = 'eval').take(count = NUM_EVAL_EXAMPLES // 1000)

steps_per_epoch = NUM_TRAIN_EXAMPLES // (TRAIN_BATCH_SIZE * NUM_EVALS)

logdir = os.path.join(
    "logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir=logdir, histogram_freq=1)

history = model.fit(
    trainds,
    validation_data=evalds,
    epochs=NUM_EVALS,
    steps_per_epoch=steps_per_epoch,
    callbacks=[tensorboard_callback])
```

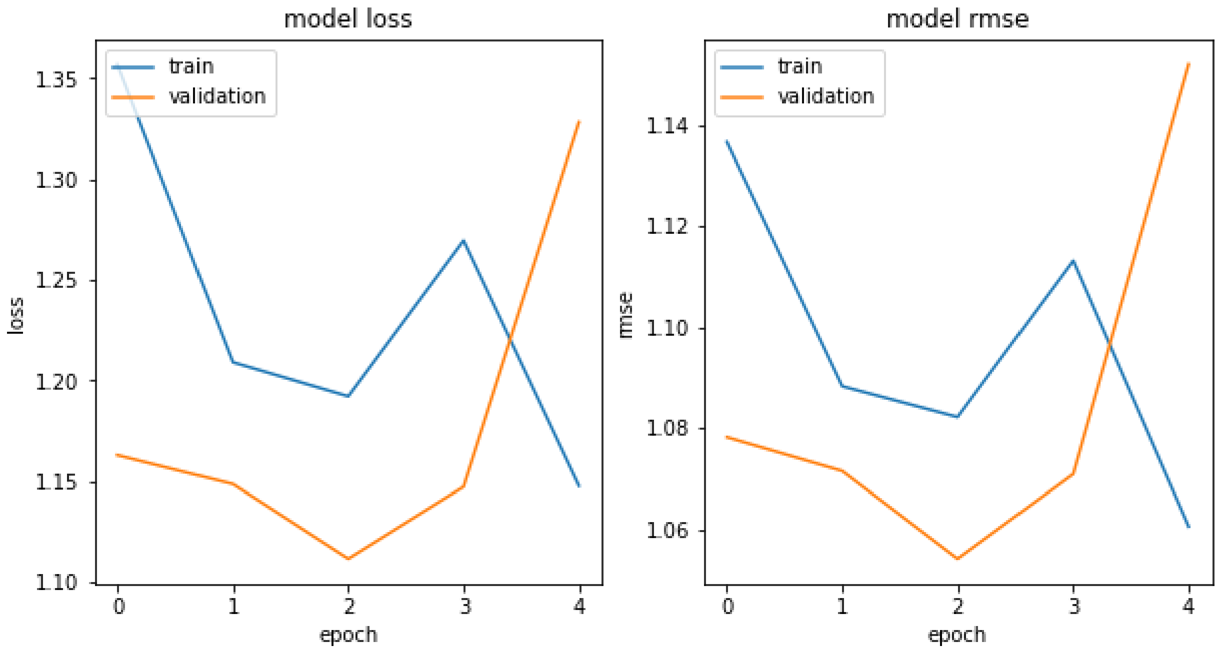
```
Epoch 1/5
312/312 [==============================] - 5s 11ms/step - loss: 1.3571 - rmse: 1.1367 - mse: 1.3571 - val_loss: 1.1629 - val_rmse: 1.0782 - val_mse: 1.1629
Epoch 2/5
312/312 [==============================] - 2s 8ms/step - loss: 1.2091 - rmse: 1.0883 - mse: 1.2091 - val_loss: 1.1486 - val_rmse: 1.0715 - val_mse: 1.1486
Epoch 3/5
312/312 [==============================] - 2s 8ms/step - loss: 1.1922 - rmse: 1.0822 - mse: 1.1922 - val_loss: 1.1115 - val_rmse: 1.0541 - val_mse: 1.1115
Epoch 4/5
312/312 [==============================] - 3s 8ms/step - loss: 1.2696 - rmse: 1.1132 - mse: 1.2696 - val_loss: 1.1473 - val_rmse: 1.0709 - val_mse: 1.1473
Epoch 5/5
312/312 [==============================] - 3s 9ms/step - loss: 1.1478 - rmse: 1.0605 - mse: 1.1478 - val_loss: 1.3282 - val_rmse: 1.1521 - val_mse: 1.3282
```

Visualize loss curve

In [59]:

```
# Plot
import matplotlib.pyplot as plt
nrows = 1
ncols = 2
fig = plt.figure(figsize=(10, 5))

for idx, key in enumerate(["loss", "rmse"]):
    ax = fig.add_subplot(nrows, ncols, idx+1)
    plt.plot(history.history[key])
    plt.plot(history.history["val_{}".format(key)])
    plt.title("model {}".format(key))
    plt.ylabel(key)
    plt.xlabel("epoch")
    plt.legend(["train", "validation"], loc="upper left");
```



Save the model

In [60]:

```
OUTPUT_DIR = "babyweight_trained"
shutil.rmtree(OUTPUT_DIR, ignore_errors=True)
EXPORT_PATH = os.path.join(
    OUTPUT_DIR, datetime.datetime.now().strftime("%Y%m%d%H%M%S"))
tf.saved_model.save(
    obj=model, export_dir=EXPORT_PATH)  # with default serving function
print("Exported trained model to {}".format(EXPORT_PATH))
```

```
WARNING:tensorflow:FOR KERAS USERS: The object that you are saving contains one
or more Keras models or layers. If you are loading the SavedModel with `tf.kera
s.models.load_model`, continue reading (otherwise, you may ignore the following
instructions). Please change your code to save with `tf.keras.models.save_model`
or `model.save`, and confirm that the file "keras.metadata" exists in the export
directory. In the future, Keras will only load the SavedModels that have this fi
le. In other words, `tf.saved_model.save` will no longer write SavedModels that
can be recovered as Keras models (this will apply in TF 2.5).

FOR DEVS: If you are overwriting _tracking_metadata in your class, this property
has been used to save metadata in the SavedModel. The metadta field will be depr
ecated soon, so please move the metadata to a different file.
INFO:tensorflow:Assets written to: babyweight_trained/20210823205958/assets
Exported trained model to babyweight_trained/20210823205958
```

In [61]:

```
!ls $EXPORT_PATH
```

```
assets  saved_model.pb  variables
```