# keras_dnn_babyweight

December 18, 2020

# 1 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

## 1.1 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 the student lab notebook – try to complete that notebook first before reviewing this solution notebook.

## 1.2 Set up environment variables and load necessary libraries

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

[ ]:
```
!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 4.7 MB/s eta 0:00:01
Requirement already satisfied:  six in /home/jupyter/.local/lib/python3.7/site-
packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: google-auth in /usr/local/lib/python3.7/site-
packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: google-resumable-media in
/usr/local/lib/python3.7/dist-packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: google-cloud-core in
```

```
/usr/local/lib/python3.7/dist-packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: protobuf in /usr/local/lib/python3.7/dist-
packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: google-api-core in /usr/local/lib/python3.7/dist-
packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: cachetools in /usr/local/lib/python3.7/dist-
packages(from google-cloud-bigquery==1.25.0)
Requirement already satisfied: rsa in /usr/local/lib/python3.7/dist-packages
(from google-cloud-bigquery==1.25.0)
Requirement already satisfied: pyasn1-modules in /usr/local/lib/python3.7/dist-
packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: googleapis-common-protos in
/usr/local/lib/python3.7/dist-packages (from google-cloud-bigquery==1.25.0)
Requirement already satisfied: pyasn1 in /usr/local/lib/python3.7/dist-packages
(from google-cloud-bigquery==1.25.0)
Requirement already satisfied: certifi in /usr/local/lib/python3.7/dist-packages
(from google-cloud-bigquery==1.25.0)
Installing collected packages: google-resumable-media, google-cloud-bigquery
WARNING: You are using pip version 20.1; however, version 20.2.3 is
available.
```

**Note**: Restart your kernel to use updated packages.

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

Import necessary libraries.

[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__)
```

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

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

[3]:
```
PROJECT = "cloud-training-demos"  # Replace with your PROJECT
```

### 1.3 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.

[4]:
```
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.

[5]:
```
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.

[6]:
```
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.

[7]:
```
# 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)
```

[7]:

```
   weight_pounds  is_male  mother_age  plurality  gestation_weeks  year  \
0       7.568469     True          22          1               46  2001
1       8.807467     True          39          1               42  2001
2       8.313632     True          23          1               35  2001
3       8.000575    False          27          1               40  2001
4       6.563162    False          29          1               39  2001
5       7.125340    False          34          1               40  2001
6       7.438397    False          31          1               38  2001
7       7.352416     True          30          1               37  2001
8       8.062305     True          16          1               40  2001
9       7.251004     True          17          1               39  2001

   month  date state mother_birth_state
0      7     5    CA                 CA
1      8     3    CA            Foreign
2     10     7    IL                 IL
3      6     7    IL                 IL
4     11     7    KY                 IN
5     12     7    MD                 MD
6      4     3    MA            Foreign
```

```
7       5       7     MI                    MI
8      10       5     MN                    MN
9       2       5     MS                    MS
```

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.

[8]:
```
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)
```

[8]:
```
   weight_pounds  is_male  mother_age  plurality  gestation_weeks  \
0       7.109908    False          25          1               38
1       7.588311    False          19          1               40
2       4.812691     True          35          1               33
3       7.251004     True          30          2               38
4       6.206013    False          21          1               36
5       6.062712    False          33          1               40
6       7.500126    False          19          1               39
7       7.687519     True          23          1               41
8       8.875811     True          24          1               40
9       7.387690    False          28          1               38

           hash_values
0   5635612483318840029
1  3487851893553562338
```

```
2  2669304657201106008
3  7076342771382320241
4  8828960867056723893
5  4280252324912833683
6  6090508671071281093
7  8708360030053768340
8  8530116731648975419
9  1776323475383399588
```

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`.

[9]:
```
# 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)
```

[9]:
```
           hash_values  num_records
0  6001926139587584124           19
1  6064126287360941757          758
2  6824828135709159935           72
3  3363240092080644183          631
4  2666158614438147859          964
5  2958542686973584093          363
6  8332670353336108110           47
7  1459116430691530322           52
8  8084544908979932787            7
9  2610866487448411172           23
```

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

[10]:
```
# 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)
```

[10]:

| | bucket_index | num_records |
|---|---|---|
| 0 | 17 | 222562 |
| 1 | 46 | 281627 |
| 2 | 7 | 270933 |
| 3 | 85 | 368045 |
| 4 | 40 | 333712 |
| 5 | 19 | 384793 |
| 6 | 77 | 401941 |
| 7 | 95 | 313544 |
| 8 | 81 | 233538 |
| 9 | 24 | 352559 |

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.

[11]:

```
# 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)
```

[11]:

| | bucket_index | num_records | percent_records |
|---|---|---|---|
| 0 | 4 | 398118 | 0.012060 |
| 1 | 92 | 336735 | 0.010201 |
| 2 | 70 | 285539 | 0.008650 |
| 3 | 78 | 326758 | 0.009898 |
| 4 | 16 | 172145 | 0.005215 |
| 5 | 94 | 431001 | 0.013056 |
| 6 | 5 | 449280 | 0.013610 |
| 7 | 62 | 426834 | 0.012930 |
| 8 | 30 | 333513 | 0.010103 |

```
9             34       379000          0.011481
```

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

[12]:
```
# 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)
```

[12]:

| | bucket_index | num_records | percent_records | dataset_name |
|---|---|---|---|---|
| 0 | 52 | 204972 | 0.006209 | train |
| 1 | 33 | 410226 | 0.012427 | train |
| 2 | 23 | 559019 | 0.016934 | train |
| 3 | 28 | 449682 | 0.013622 | train |
| 4 | 62 | 426834 | 0.012930 | train |
| 5 | 73 | 411771 | 0.012474 | train |
| 6 | 38 | 338150 | 0.010243 | train |
| 7 | 35 | 250505 | 0.007588 | train |
| 8 | 65 | 289303 | 0.008764 | train |
| 9 | 61 | 453904 | 0.013750 | train |

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

[13]:
```
# 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)
```

[13]:

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

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

[14]:

```
# 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)
```

[14]:

| | bucket_index | num_records | percent_records | dataset_name |
|---|---|---|---|---|
| 0 | 92 | 336735 | 0.010201 | test |
| 1 | 91 | 333267 | 0.010096 | test |
| 2 | 90 | 286465 | 0.008678 | test |
| 3 | 94 | 431001 | 0.013056 | test |
| 4 | 93 | 215710 | 0.006534 | test |
| 5 | 99 | 223334 | 0.006765 | test |
| 6 | 95 | 313544 | 0.009498 | test |
| 7 | 97 | 480790 | 0.014564 | test |
| 8 | 96 | 529357 | 0.016036 | test |
| 9 | 98 | 374697 | 0.011351 | 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.

[15]:
```python
# 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)
```

[15]:

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

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.

[16]:
```python
# 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)
```

[16]:

```
   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.

[33]:

```
# 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_divisor)

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)
```

```python
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.
```

### 1.4 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.

[34]:
```python
train_df.head()
```

[34]:
```
   weight_pounds  is_male  mother_age  plurality  gestation_weeks  \
0       9.499719     True          30          1               40
1       6.027438     True          26          1               36
2       6.124442     True          34          2               37
3       9.001474     True          28          1               35
4       7.070225    False          23          1               40

           hash_values
0   5057322745617000014
1  14093484355091000014
2  26208601650938000008
3  14093484355091000014
4  46593541140388000077
```

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)

[35]:
```python
train_df.describe()
```

[35]:
```
       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 | 4.983286e+18 |
| std | 1.303220 | 6.134232 | 0.201568 | 2.531921 | 2.551244e+18 |
| min | 0.562179 | 13.000000 | 1.000000 | 18.000000 | 5.826385e+15 |
| 25% | 6.624891 | 23.000000 | 1.000000 | 38.000000 | 3.153609e+18 |
| 50% | 7.345803 | 28.000000 | 1.000000 | 39.000000 | 4.896699e+18 |
| 75% | 8.062305 | 33.000000 | 1.000000 | 40.000000 | 6.784884e+18 |
| max | 11.563246 | 48.000000 | 4.000000 | 47.000000 | 9.210618e+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.

[36]:
```
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:

[37]:
```
train_df = preprocess(train_df)
eval_df = preprocess(eval_df)
test_df = preprocess(test_df)
```

[38]:
```
train_df.head()
```

[38]:
```
   weight_pounds   is_male  mother_age  plurality  gestation_weeks  \
0       7.874912  Unknown          38  Single(1)               38
1       8.999270  Unknown          31  Single(1)               45
2       7.251004     True          24  Single(1)               40
3       8.562754     True          43  Single(1)               39
4       6.194990     True          23  Single(1)               41

           hash_values
0  8717259940738900003
1  6781866293108400060
2  1696737464106800060
3  4614303140002600076
4   780565305641800050
```

[39]:
```
train_df.tail()
```

[39]:
```
       weight_pounds   is_male  mother_age  plurality  gestation_weeks  \
15461       7.251004     True          32  Single(1)               39
15462       8.811877     True          30  Single(1)               39
15463       7.248799     True          26  Single(1)               40
15464       7.625790  Unknown          22  Single(1)               40
15465       6.499227  Unknown          22  Single(1)               38

               hash_values
15461  8655151740159000017
15462   845203792559000058
15463  1409348435509100014
15464  2875790318525700041
15465  8720767384765100051
```

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

[40]:
```
train_df.describe()
```

[40]:

| | 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 | 4.983286e+18 |
| std | 1.303178 | 6.134034 | 2.531839 | 2.551162e+18 |
| min | 0.562179 | 13.000000 | 18.000000 | 5.826385e+15 |
| 25% | 6.624891 | 23.000000 | 38.000000 | 3.153609e+18 |
| 50% | 7.345803 | 28.000000 | 39.000000 | 4.896699e+18 |
| 75% | 8.062305 | 33.000000 | 40.000000 | 6.784884e+18 |
| max | 11.563246 | 48.000000 | 47.000000 | 9.210618e+18 |

### 1.5 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.

[41]:
```
# 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)
```

[42]:
```
%%bash
wc -l *.csv
```

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

[43]:
```
%%bash
head *.csv
```

```
==> eval.csv <==
8.62448368944,Unknown,31,Single(1),42
6.9996768185,Unknown,32,Single(1),39
```

```
6.6248909731,False,30,Single(1),38
8.3114272774,False,19,Single(1),41
8.313631900019999,True,32,Single(1),37
7.06140625186,Unknown,34,Single(1),41
7.62578964258,Unknown,34,Single(1),39
7.3744626639,Unknown,20,Single(1),39
1.93786328298,False,32,Triplets(3),28
8.99926953484,True,34,Single(1),39

==> test.csv <==
7.3744626639,Unknown,25,Single(1),44
6.93794738514,Unknown,24,Single(1),40
6.87621795178,True,30,Single(1),39
6.87621795178,Unknown,29,Single(1),39
7.0327461578,Unknown,36,Single(1),38
9.31232594688,False,25,Single(1),39
7.936641432,True,23,Single(1),37
4.7840310854,Unknown,34,Multiple(2+),38
7.31273323054,True,23,Single(1),39
8.24969784404,False,32,Single(1),39

==> train.csv <==
7.87491199864,Unknown,38,Single(1),38
8.99926953484,Unknown,31,Single(1),45
7.25100379718,True,24,Single(1),40
8.56275425608,True,43,Single(1),39
6.1949895622,True,23,Single(1),41
9.0609989682,Unknown,24,Single(1),38
7.5618555866,True,26,Single(1),41
7.30611936268,False,31,Single(1),41
9.6672701887,True,29,Single(1),40
6.4992274837599995,True,22,Single(1),39
```

[44]:
```
%%bash
tail *.csv
```

```
==> eval.csv <==
7.43839671988,False,25,Single(1),37
7.06140625186,True,34,Single(1),41
7.43619209726,True,36,Single(1),40
3.56267015392,True,35,Twins(2),31
8.811876612139999,False,27,Single(1),36
8.0689187892,Unknown,36,Single(1),40
8.7633749145,Unknown,34,Single(1),39
7.43839671988,True,43,Single(1),40
4.62529825676,Unknown,38,Multiple(2+),35
6.1839664491,Unknown,20,Single(1),38
```

```
==> test.csv <==
6.37576861704,Unknown,21,Single(1),39
7.5618555866,True,22,Single(1),39
8.99926953484,Unknown,28,Single(1),42
7.82420567838,Unknown,24,Single(1),39
9.25059651352,True,26,Single(1),40
8.62448368944,Unknown,28,Single(1),39
5.2580249487,False,18,Single(1),38
7.87491199864,True,25,Single(1),37
5.81138522632,Unknown,41,Single(1),36
6.93794738514,True,24,Single(1),40

==> train.csv <==
7.81318256528,True,18,Single(1),43
7.31273323054,False,35,Single(1),34
6.75055446244,Unknown,37,Single(1),39
7.43839671988,True,32,Single(1),39
6.9666074791999995,True,20,Single(1),38
7.25100379718,True,32,Single(1),39
8.811876612139999,True,30,Single(1),39
7.24879917456,True,26,Single(1),40
7.62578964258,Unknown,22,Single(1),40
6.4992274837599995,Unknown,22,Single(1),38
```

[2]:
```
%%bash
ls *.csv
```

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

[3]:
```
%%bash
head -5 *.csv
```

```
==> eval.csv <==
6.87621795178,False,33,Single(1),40
7.7492485093,Unknown,21,Single(1),38
8.86699217764,False,22,Single(1),38
6.60504936952,False,32,Single(1),40
8.313631900019999,True,36,Single(1),39

==> test.csv <==
7.5618555866,True,40,Twins(2),43
9.3586230219,Unknown,22,Single(1),40
8.5539357656,True,26,Single(1),37
5.81138522632,Unknown,36,Multiple(2+),36
7.06140625186,Unknown,23,Single(1),40
```

```
==> train.csv <==
10.18756112702,Unknown,23,Single(1),33
8.93754010148,True,40,Single(1),41
6.9996768185,Unknown,23,Single(1),38
8.65975765136,Unknown,19,Single(1),42
4.2549216566,True,20,Single(1),33
```

## 1.6 Create Keras model

### 1.6.1 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.

[4]:
```python
# 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]]
```

### 1.6.2 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.

[5]:
```python
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)

    # 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
```

### 1.6.3 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…)

[6]:
```
# TODO 1
def create_input_layers():
```

```
    """Creates dictionary of input layers for each feature.

    Returns:
        Dictionary of `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
```

#### 1.6.4 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.

[7]:
```
# TODO 2
def categorical_fc(name, values):
    """Helper function to wrap categorical feature by indicator column.

    Args:
        name: str, name of feature.
        values: list, list of strings of categorical values.
    Returns:
        Indicator column of categorical feature.
    """
    cat_column = tf.feature_column.categorical_column_with_vocabulary_list(
            key=name, vocabulary_list=values)

    return tf.feature_column.indicator_column(categorical_column=cat_column)


def create_feature_columns():
    """Creates dictionary of feature columns from inputs.

    Returns:
        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
```

### 1.6.5 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.

[8]:
```
# 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
```

### 1.6.6 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.

[9]:
```
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))
```

### 1.6.7 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.

[10]:
```
# TODO 4
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.values())(inputs)

    # Get output of model given inputs
    output = get_model_outputs(dnn_inputs)

    # Build model and compile it all together
    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:

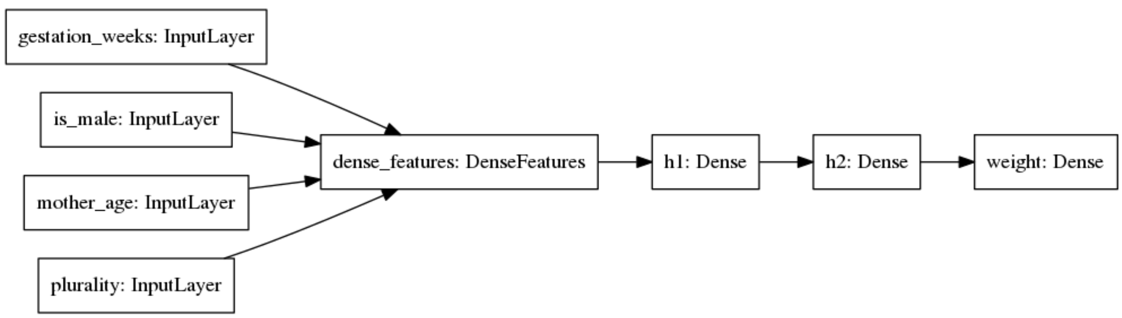
```
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-
packages/tensorflow_core/python/feature_column/feature_column_v2.py:4276:
IndicatorColumn._variable_shape (from
tensorflow.python.feature_column.feature_column_v2) is deprecated and will be
removed in a future version.
Instructions for updating:
The old _FeatureColumn APIs are being deprecated. Please use the new
FeatureColumn APIs instead.
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-
packages/tensorflow_core/python/feature_column/feature_column_v2.py:4331:
VocabularyListCategoricalColumn._num_buckets (from
tensorflow.python.feature_column.feature_column_v2) is deprecated and will be
removed in a future version.
Instructions for updating:
The old _FeatureColumn APIs are being deprecated. Please use the new
FeatureColumn APIs instead.
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 (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[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.

[11]:
```python
tf.keras.utils.plot_model(
    model=model, to_file="dnn_model.png", show_shapes=False, rankdir="LR")
```

[11]:




## 1.7 Run and evaluate model

### 1.7.1 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.

[12]:
```python
# 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

trainds = load_dataset(
    pattern="train*",
```

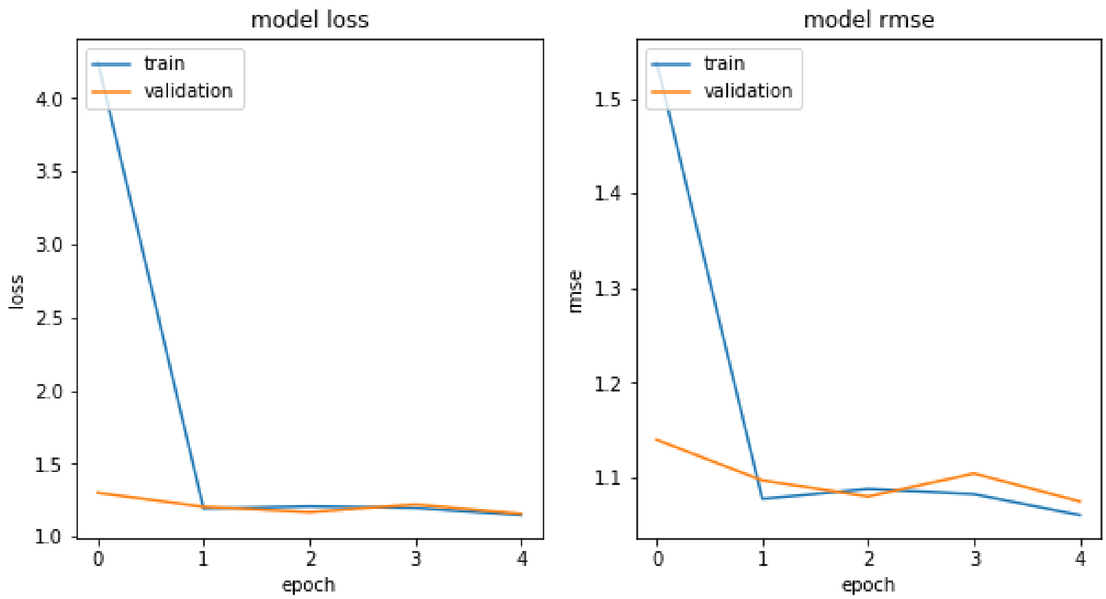
```
    batch_size=TRAIN_BATCH_SIZE,
    mode='train')

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])
```

```
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-
packages/tensorflow_core/python/data/experimental/ops/readers.py:521:
parallel_interleave (from
tensorflow.python.data.experimental.ops.interleave_ops) is deprecated and will
be removed in a future version.
Instructions for updating:
Use `tf.data.Dataset.interleave(map_func, cycle_length, block_length,
num_parallel_calls=tf.data.experimental.AUTOTUNE)` instead. If sloppy execution
is desired, use `tf.data.Options.experimental_determinstic`.
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-
packages/tensorflow_core/python/data/experimental/ops/readers.py:215:
shuffle_and_repeat (from tensorflow.python.data.experimental.ops.shuffle_ops) is
deprecated and will be removed in a future version.
Instructions for updating:
Use `tf.data.Dataset.shuffle(buffer_size, seed)` followed by
`tf.data.Dataset.repeat(count)`. Static tf.data optimizations will take care of
using the fused implementation.
Train for 312 steps, validate for 10 steps
Epoch 1/5
312/312 [==============================] - 5s 16ms/step - loss: 4.2510 - rmse:
1.5391 - mse: 4.2510 - val_loss: 1.3007 - val_rmse: 1.1402 - val_mse: 1.3007
Epoch 2/5
312/312 [==============================] - 3s 9ms/step - loss: 1.1924 - rmse:
1.0779 - mse: 1.1924 - val_loss: 1.2050 - val_rmse: 1.0974 - val_mse: 1.2050
Epoch 3/5
```

```
312/312 [==============================] - 4s 12ms/step - loss: 1.2077 - rmse:
1.0884 - mse: 1.2077 - val_loss: 1.1679 - val_rmse: 1.0804 - val_mse: 1.1679
Epoch 4/5
312/312 [==============================] - 4s 11ms/step - loss: 1.1964 - rmse:
1.0829 - mse: 1.1964 - val_loss: 1.2209 - val_rmse: 1.1047 - val_mse: 1.2209
Epoch 5/5
312/312 [==============================] - 3s 9ms/step - loss: 1.1474 - rmse:
1.0607 - mse: 1.1474 - val_loss: 1.1566 - val_rmse: 1.0752 - val_mse: 1.1566
```

### 1.7.2 Visualize loss curve

[13]:
```
# 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");
```



### 1.7.3 Save the model

[14]:
```python
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:From /usr/local/lib/python3.5/dist-
packages/tensorflow_core/python/ops/resource_variable_ops.py:1781: calling
BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops)
with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
INFO:tensorflow:Assets written to: babyweight_trained/20191119050541/assets
Exported trained model to babyweight_trained/20191119050541
```

[15]:
```
!ls $EXPORT_PATH
```

```
assets  saved_model.pb  variables
```