```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False
```

Large Structured Data

When dealing with large amounts of structured text data we can also do some stuff to speed things up, though there are some key differences that lessen our toolkit:

- Data manipulation (filtering out features, customized data cleanup, etc...) is far easier to do in a tabular format like a dataframe. If there is going to be a lot of that, and the data is really large, we can do the 'manual' prep separately, write the data to a file, and then read it in again as ready-to-use data.
- As the data gets really large, most real life scenarios will either use big data approaches like Spark, or store structured data in a DB. That's the 'real' way to deal with large amounts of strucutred data, so there are not as many easy to use tools for this as we find with images.
- Further to the two ponts above, if the dataset is a CSV, we can likely load it into memory in its entirety as "too many rows to fit in memory" and "this data is stored in a CSV file" tend not to come around together all that often in a situation where there is actual infrastructure.
- Really large amounts of text can be broken into multiple smaller files, then we load a file at a time, similar to how we deal with images. This is common with NLP text, much more so than structured data.
- There is often an assumption that when needing to deal with large amounts of structured CSV data that we have the data already split into training and validation sets. This makes sense, as

On the whole, dealing with large amounts of structured data tends to not be as large of an issue to be solved as dealing with large amounts of unstrucutured data in a non-big data environment. This is because huge data goes to big data strategies, or at least a DB, less huge data can just fit in memory and be dealt with how we have dealt with all other CSV based data.

TensorFlow Datasets

Tensorflow Datasets are something that we used when loading image files from disk, as loading all of the data at once can be impossible for larger datasets. These datasets serve the same function as a regular dataframe for modle training purposes, but they are designed more to be able to efficiently load large amounts of data from disk than to allow easy viewing and manipulation of the data.

Tensorflow datasets allow us to set several options on how the data is loaded, that we can use to make a dataset that is more efficient for our purposes.

Dataset for Structured CSV

The function below reads CSV data from disk and generates training and validation datasets that we can feed to our model. We also add batching, shuffle the training data, and use prefetch to make the data loading process more efficient.

```
import tensorflow as tf
import numpy as np
import csv
import time
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import Sequence
import os
import zipfile

# Define a function to load the CSV data and create a tf.data.Dataset object
def create_dataset(csv_path, batch_size=32, buffer_size=1024, validation_split=0.2, shuffle=True, start=None):
    # Load the CSV data
    with open(csv_path) as f:
        csv_reader = csv.reader(f)
        header = next(csv_reader)
        feature_names = header[start:-1]
        label_name = header[-1]
        features = []
        labels = []
        for row in csv_reader:
            features.append([float(x) for x in row[start:-1]])
            labels.append(float(row[-1]))
    features = np.array(features)
    labels = np.array(labels)

    # Split the data into training and validation sets
    split_idx = int(len(features) * (1 - validation_split))
    train_features, train_labels = features[:split_idx], labels[:split_idx]
    val_features, val_labels = features[split_idx:], labels[split_idx:]

    # Create a tf.data.Dataset object for the training data
    train_ds = tf.data.Dataset.from_tensor_slices((train_features, train_labels))
    train_ds = train_ds.cache()
    if shuffle:
        train_ds = train_ds.shuffle(buffer_size=buffer_size)
```

Defaulting to standard GPU accelerator. Select premium accelerator in Runtime > Change runtime type Open notebook settings

Resources

You are subscribed to Colab Pro+. Learn more.
Available: 1343.07 compute units
Usage rate: approximately 15.04 per hour
You have 2 active sessions.
Manage sessions

Python 3 Google Compute Engine backend (GPU)
Showing resources from 2:31 PM to 2:47 PM

System RAM
5.5 / 12.7 GB

GPU RAM
1.1 / 15.0 GB

Disk
26.5 / 166.8 GB

```
    # Create a tf.data.Dataset object for the validation data
    val_ds = tf.data.Dataset.from_tensor_slices((val_features, val_labels)).prefetch(tf.data.experimental.AUTOTUNE)
    val_ds = val_ds.cache()
    val_ds = val_ds.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)

    return train_ds, val_ds

BASE_EPOCHS  = 20
VAL_SPLIT = 0.2
DIABETES_CSV_PATH = 'diabetes.csv'
BATCH_SIZE = 1024


if not os.path.exists(DIABETES_CSV_PATH):
    url = 'https://jrssbcrsefilesnait.blob.core.windows.net/3950data1/diabetes.csv'
    d_path = tf.keras.utils.get_file(origin=url, extract=True, archive_format='auto')
    print(d_path)

    /root/.keras/datasets/diabetes.csv
```

Simple and Small Example

We can test the generator on a small file.

Note: with small examples, we won't really see any advantage in terms of speed as we can probably just load the data into memory without concern, no matter what. This starts to matter more when dealing with large files, where the disk access time can actually add up.

```
# Load the CSV data and create the tf.data.Dataset objects
train_ds, val_ds = create_dataset(d_path)

# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics="accuracy")

# Fit the model to the data
start = time.time()
model.fit(train_ds, epochs=BASE_EPOCHS, validation_data=val_ds)
end = time.time()
print("DS Training time: {} seconds".format(end - start))

    Epoch 1/20
    20/20 [==============================] - 1s 9ms/step - loss: 4.6845 - accuracy: 0.5065 - val_loss: 3.3233 - val_a
    Epoch 2/20
    20/20 [==============================] - 0s 4ms/step - loss: 1.6055 - accuracy: 0.5505 - val_loss: 0.9220 - val_a
    Epoch 3/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.7322 - accuracy: 0.6743 - val_loss: 0.7417 - val_a
    Epoch 4/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.7397 - accuracy: 0.6645 - val_loss: 1.0319 - val_a
    Epoch 5/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.7465 - accuracy: 0.6694 - val_loss: 0.9542 - val_a
    Epoch 6/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.7441 - accuracy: 0.6482 - val_loss: 0.6946 - val_a
    Epoch 7/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.7060 - accuracy: 0.6629 - val_loss: 0.6600 - val_a
    Epoch 8/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.8146 - accuracy: 0.6401 - val_loss: 0.8072 - val_a
    Epoch 9/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.6266 - accuracy: 0.6792 - val_loss: 0.6577 - val_a
    Epoch 10/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.6490 - accuracy: 0.6678 - val_loss: 0.6683 - val_a
    Epoch 11/20
    20/20 [==============================] - 0s 5ms/step - loss: 0.6163 - accuracy: 0.6840 - val_loss: 0.7606 - val_a
    Epoch 12/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.5952 - accuracy: 0.7020 - val_loss: 0.6546 - val_a
    Epoch 13/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.5601 - accuracy: 0.7166 - val_loss: 0.7556 - val_a
    Epoch 14/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.5686 - accuracy: 0.7231 - val_loss: 0.7452 - val_a
    Epoch 15/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.6055 - accuracy: 0.7101 - val_loss: 0.6371 - val_a
    Epoch 16/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.6270 - accuracy: 0.6987 - val_loss: 0.7760 - val_a
    Epoch 17/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.5549 - accuracy: 0.7215 - val_loss: 0.6378 - val_a
    Epoch 18/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.5576 - accuracy: 0.7199 - val_loss: 0.6952 - val_a
    Epoch 19/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.5732 - accuracy: 0.7231 - val_loss: 0.7769 - val_a
    Epoch 20/20
    20/20 [==============================] - 0s 4ms/step - loss: 0.5732 - accuracy: 0.7231 - val_loss: 0.7024 - val_a
    DS Training time: 3.777135133743286 seconds


# Time dataframe for comparison
df_small = pd.read_csv(d_path)
df_small_y = df_small["Outcome"]
df_small_X = df_small.drop(columns=["Outcome"])
```

```
df_small_X = df_small.drop(columns=["Outcome"])
width = df_small_X.shape[1]
# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(width,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model

model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics="accuracy")
# Fit the model to the data
start = time.time()
model.fit(x=df_small_X, y=df_small_y, epochs=BASE_EPOCHS, validation_split=0.2)
end = time.time()
print("DS Training time: {} seconds".format(end - start))
```

```
Epoch 1/20
20/20 [==============================] - 1s 12ms/step - loss: 1.8232 - accuracy: 0.5440 - val_loss: 1.1896 - val_
Epoch 2/20
20/20 [==============================] - 0s 5ms/step - loss: 1.0007 - accuracy: 0.6091 - val_loss: 1.0485 - val_a
Epoch 3/20
20/20 [==============================] - 0s 5ms/step - loss: 0.7562 - accuracy: 0.6384 - val_loss: 0.7388 - val_a
Epoch 4/20
20/20 [==============================] - 0s 5ms/step - loss: 0.7148 - accuracy: 0.6482 - val_loss: 0.7454 - val_a
Epoch 5/20
20/20 [==============================] - 0s 4ms/step - loss: 0.7249 - accuracy: 0.6580 - val_loss: 0.8177 - val_a
Epoch 6/20
20/20 [==============================] - 0s 5ms/step - loss: 0.8352 - accuracy: 0.6189 - val_loss: 0.7443 - val_a
Epoch 7/20
20/20 [==============================] - 0s 5ms/step - loss: 0.7298 - accuracy: 0.6580 - val_loss: 0.6577 - val_a
Epoch 8/20
20/20 [==============================] - 0s 5ms/step - loss: 0.5976 - accuracy: 0.6889 - val_loss: 0.8131 - val_a
Epoch 9/20
20/20 [==============================] - 0s 5ms/step - loss: 0.6700 - accuracy: 0.6743 - val_loss: 0.7494 - val_a
Epoch 10/20
20/20 [==============================] - 0s 5ms/step - loss: 0.6799 - accuracy: 0.6678 - val_loss: 1.0498 - val_a
Epoch 11/20
20/20 [==============================] - 0s 5ms/step - loss: 0.8166 - accuracy: 0.6726 - val_loss: 1.0711 - val_a
Epoch 12/20
20/20 [==============================] - 0s 5ms/step - loss: 0.7315 - accuracy: 0.6678 - val_loss: 0.8238 - val_a
Epoch 13/20
20/20 [==============================] - 0s 6ms/step - loss: 0.6274 - accuracy: 0.6954 - val_loss: 0.6840 - val_a
Epoch 14/20
20/20 [==============================] - 0s 5ms/step - loss: 0.5602 - accuracy: 0.7134 - val_loss: 0.7466 - val_a
Epoch 15/20
20/20 [==============================] - 0s 5ms/step - loss: 0.5528 - accuracy: 0.7280 - val_loss: 0.6249 - val_a
Epoch 16/20
20/20 [==============================] - 0s 5ms/step - loss: 0.6404 - accuracy: 0.7003 - val_loss: 0.6527 - val_a
Epoch 17/20
20/20 [==============================] - 0s 5ms/step - loss: 0.6060 - accuracy: 0.6873 - val_loss: 0.7331 - val_a
Epoch 18/20
20/20 [==============================] - 0s 5ms/step - loss: 0.5598 - accuracy: 0.7134 - val_loss: 0.8630 - val_a
Epoch 19/20
20/20 [==============================] - 0s 4ms/step - loss: 0.7093 - accuracy: 0.6629 - val_loss: 0.9069 - val_a
Epoch 20/20
20/20 [==============================] - 0s 6ms/step - loss: 0.6189 - accuracy: 0.7036 - val_loss: 0.6423 - val_a
DS Training time: 3.4213695526123047 seconds
```

Larger Example

We can download a larger file, and try it out. We ill also use the .cache() method to cache the data in memory, so that we don't have to reload it every time we run the code. This CSV file is roughly 150mb in size, so it is large enough to be noticable when we need to load the entire thing, but small enough to fit in memory. For most CSV data that we might encounter, this is probably a good approach - most systems can handle the memory demands of the CSV file size we might see.

```
# Download the file

zip_name = 'fraud.zip'
if not os.path.exists(zip_name):
    url = 'https://jrssbcrsefilesnait.blob.core.windows.net/3950data1/creditcard.csv'
    zip_path = tf.keras.utils.get_file(origin=url, extract=True, archive_format='auto')
    print(zip_path)
```

```
/root/.keras/datasets/creditcard.csv
```

```
#big_file = "/Users/akeems/.keras/datasets/creditcard.csv"
big_file = zip_path
# Load the CSV data and create the tf.data.Dataset objects
train_ds, val_ds = create_dataset(big_file, start=1, batch_size=BATCH_SIZE)
```



```
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(),
```

```
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics="accuracy")

# Fit the model to the data
# time the fit

start = time.time()
model.fit(train_ds, epochs=BASE_EPOCHS, validation_data=val_ds)
end = time.time()
print("Time to fit: ", end - start)
```

```
Epoch 1/20
223/223 [==============================] - 3s 7ms/step - loss: 0.5424 - accuracy: 0.9608 - val_loss: 0.0294 - val
Epoch 2/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0241 - accuracy: 0.9990 - val_loss: 0.0048 - val
Epoch 3/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0110 - accuracy: 0.9991 - val_loss: 0.0039 - val
Epoch 4/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0080 - accuracy: 0.9993 - val_loss: 0.0037 - val
Epoch 5/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0049 - accuracy: 0.9993 - val_loss: 0.0041 - val
Epoch 6/20
223/223 [==============================] - 1s 5ms/step - loss: 0.0039 - accuracy: 0.9993 - val_loss: 0.0037 - val
Epoch 7/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0040 - accuracy: 0.9993 - val_loss: 0.0034 - val
Epoch 8/20
223/223 [==============================] - 2s 7ms/step - loss: 0.0047 - accuracy: 0.9994 - val_loss: 0.0040 - val
Epoch 9/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0041 - accuracy: 0.9993 - val_loss: 0.0044 - val
Epoch 10/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0036 - accuracy: 0.9994 - val_loss: 0.0031 - val
Epoch 11/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0036 - accuracy: 0.9994 - val_loss: 0.0033 - val
Epoch 12/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0048 - accuracy: 0.9994 - val_loss: 0.0039 - val
Epoch 13/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0044 - accuracy: 0.9994 - val_loss: 0.0039 - val
Epoch 14/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0035 - accuracy: 0.9994 - val_loss: 0.0037 - val
Epoch 15/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0043 - accuracy: 0.9995 - val_loss: 0.0043 - val
Epoch 16/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0040 - accuracy: 0.9994 - val_loss: 0.0039 - val
Epoch 17/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0061 - accuracy: 0.9994 - val_loss: 0.0077 - val
Epoch 18/20
223/223 [==============================] - 2s 8ms/step - loss: 0.0039 - accuracy: 0.9994 - val_loss: 0.0029 - val
Epoch 19/20
223/223 [==============================] - 1s 5ms/step - loss: 0.0033 - accuracy: 0.9995 - val_loss: 0.0037 - val
Epoch 20/20
223/223 [==============================] - 1s 4ms/step - loss: 0.0034 - accuracy: 0.9995 - val_loss: 0.0037 - val
Time to fit:  26.499851942062378
```

Dataframe for Comparison

```
df_large = pd.read_csv(big_file)
df_large_y = df_large["Class"]
df_large_X = df_large.drop(columns={"Class"})
width = df_large_X.shape[1]
# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(width,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics="accuracy")

# Fit the model to the data
# time the fit
start = time.time()
model.fit(x=df_large_X, y=df_large_y, epochs=BASE_EPOCHS, validation_split=VAL_SPLIT, batch_size=BATCH_SIZE)
end = time.time()
print("Time to fit: ", end - start)
```

```
Epoch 1/20
223/223 [==============================] - 2s 5ms/step - loss: 17.7116 - accuracy: 0.9893 - val_loss: 50.3043 - v
Epoch 2/20
223/223 [==============================] - 1s 4ms/step - loss: 22.1066 - accuracy: 0.9928 - val_loss: 66.0235 - v
Epoch 3/20
223/223 [==============================] - 1s 4ms/step - loss: 29.6614 - accuracy: 0.9982 - val_loss: 32.5662 - v
Epoch 4/20
                                                                                                  oss: 1.3205 - val
                                                                                                  oss: 0.0329 - val
Epoch 6/20
223/223 [==============================] - 1s 5ms/step - loss: 0.3102 - accuracy: 0.9954 - val_loss: 7.1010 - val
Epoch 7/20
223/223 [==============================] - 1s 5ms/step - loss: 36.9468 - accuracy: 0.9938 - val_loss: 64.6632 - v
Epoch 8/20
223/223 [==============================] - 1s 5ms/step - loss: 27.4913 - accuracy: 0.9982 - val_loss: 28.7538 - v
Epoch 9/20
223/223 [==============================] - 1s 4ms/step - loss: 30.1140 - accuracy: 0.9937 - val_loss: 54.5705 - v
```

```
Epoch 10/20
223/223 [==============================] - 1s 4ms/step - loss: 23.2461 - accuracy: 0.9981 - val_loss: 24.0134 - v
Epoch 11/20
223/223 [==============================] - 1s 4ms/step - loss: 5.7816 - accuracy: 0.9974 - val_loss: 0.3292 - val
Epoch 12/20
223/223 [==============================] - 1s 4ms/step - loss: 0.1567 - accuracy: 0.9950 - val_loss: 0.0462 - val
Epoch 13/20
223/223 [==============================] - 1s 4ms/step - loss: 0.9297 - accuracy: 0.9963 - val_loss: 13.0957 - va
Epoch 14/20
223/223 [==============================] - 1s 4ms/step - loss: 4.1531 - accuracy: 0.9968 - val_loss: 0.4223 - val
Epoch 15/20
223/223 [==============================] - 1s 4ms/step - loss: 37.5602 - accuracy: 0.9940 - val_loss: 46.8260 - v
Epoch 16/20
223/223 [==============================] - 1s 4ms/step - loss: 17.9299 - accuracy: 0.9982 - val_loss: 13.2215 - v
Epoch 17/20
223/223 [==============================] - 1s 4ms/step - loss: 8.5467 - accuracy: 0.9921 - val_loss: 66.1687 - va
Epoch 18/20
223/223 [==============================] - 1s 4ms/step - loss: 28.8926 - accuracy: 0.9982 - val_loss: 32.5429 - v
Epoch 19/20
223/223 [==============================] - 1s 4ms/step - loss: 10.6996 - accuracy: 0.9981 - val_loss: 4.1515 - va
Epoch 20/20
223/223 [==============================] - 1s 5ms/step - loss: 20.3139 - accuracy: 0.9935 - val_loss: 34.5448 - v
Time to fit:  21.23183250427246
```

Dataframe to Dataset

If we have a dataframe we can convert it to a dataset using the from_tensor_slices() method. Manipulating the data in a dataframe is far easier, so we can prep in a df then convert to a dataset. The function below creates a dataset from a dataframe, long with a few of the other things we commonly want to do in our data prep.

```
def get_keras_dataset(df, target="target", val_split=0.2, batch_size=32):
    # Splitting the dataframe into training and validation sets
    train_df, val_df = train_test_split(df, test_size=val_split, random_state=42)

    # Extracting the target variable from the dataframes
    train_y = train_df.pop(target)
    val_y = val_df.pop(target)

    # Converting the target variable to categorical if necessary
    num_classes = len(train_y.unique())
    if num_classes > 2:
        train_y = to_categorical(train_y, num_classes)
        val_y = to_categorical(val_y, num_classes)

    # Creating a tf.data.Dataset for training and validation sets
    train_ds = tf.data.Dataset.from_tensor_slices((train_df.values, train_y))
    val_ds = tf.data.Dataset.from_tensor_slices((val_df.values, val_y))

    train_ds =train_ds.cache()
    val_ds = val_ds.cache()

    # Shuffling and batching the datasets
    train_ds = train_ds.shuffle(len(train_df)).batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)
    val_ds = val_ds.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)

    return train_ds, val_ds


train_ds_df, val_ds_df = get_keras_dataset(df_large, target="Class", val_split=VAL_SPLIT, batch_size=BATCH_SIZE)

# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics="accuracy")

# Fit the model to the data
# time the fit

start = time.time()
model.fit(train_ds_df, epochs=BASE_EPOCHS, validation_data=val_ds_df)
end = time.time()
print("Time to fit: ", end - start)
```

```
Epoch 1/20
223/223 [==============================] - 4s 6ms/step - loss: 20.2613 - accuracy: 0.9893 - val_loss: 12.6636 - v
Epoch 2/20
223/223 [==============================] - 1s 5ms/step - loss: 7.1570 - accuracy: 0.9982 - val_loss: 1.7405 - val
223/223 [==============================] - 1s 5ms/step - loss: 11.3642 - accuracy: 0.9981 - val_loss: 2.0811 - va
Epoch 5/20
223/223 [==============================] - 1s 5ms/step - loss: 6.2208 - accuracy: 0.9929 - val_loss: 32.9861 - va
Epoch 6/20
223/223 [==============================] - 1s 5ms/step - loss: 22.5364 - accuracy: 0.9983 - val_loss: 15.5577 - v
Epoch 7/20
223/223 [==============================] - 1s 5ms/step - loss: 10.0594 - accuracy: 0.9935 - val_loss: 40.4826 - v
```

```
Epoch 8/20
223/223 [==============================] - 2s 8ms/step - loss: 29.3185 - accuracy: 0.9934 - val_loss: 39.4090 - v
Epoch 9/20
223/223 [==============================] - 2s 6ms/step - loss: 27.8363 - accuracy: 0.9983 - val_loss: 20.7998 - v
Epoch 10/20
223/223 [==============================] - 1s 5ms/step - loss: 11.9904 - accuracy: 0.9982 - val_loss: 4.9587 - va
Epoch 11/20
223/223 [==============================] - 1s 5ms/step - loss: 3.4529 - accuracy: 0.9964 - val_loss: 0.3103 - val
Epoch 12/20
223/223 [==============================] - 1s 5ms/step - loss: 10.2066 - accuracy: 0.9953 - val_loss: 7.3768 - va
Epoch 13/20
223/223 [==============================] - 1s 5ms/step - loss: 1.8395 - accuracy: 0.9958 - val_loss: 0.1140 - val
Epoch 14/20
223/223 [==============================] - 1s 5ms/step - loss: 0.4728 - accuracy: 0.9949 - val_loss: 0.1197 - val
Epoch 15/20
223/223 [==============================] - 1s 5ms/step - loss: 0.1062 - accuracy: 0.9953 - val_loss: 0.1201 - val
Epoch 16/20
223/223 [==============================] - 1s 5ms/step - loss: 8.5181 - accuracy: 0.9938 - val_loss: 33.3056 - va
Epoch 17/20
223/223 [==============================] - 2s 8ms/step - loss: 20.5965 - accuracy: 0.9983 - val_loss: 11.6870 - v
Epoch 18/20
223/223 [==============================] - 1s 5ms/step - loss: 3.7815 - accuracy: 0.9972 - val_loss: 0.1211 - val
Epoch 19/20
223/223 [==============================] - 1s 5ms/step - loss: 0.1108 - accuracy: 0.9959 - val_loss: 0.0994 - val
Epoch 20/20
223/223 [==============================] - 1s 5ms/step - loss: 0.1045 - accuracy: 0.9968 - val_loss: 0.0961 - val
Time to fit:  32.16196894645691
```

Polars DataFrame

We can also use the faster and more efficient Polars DataFrame to load the data. This is a DataFrame that is written in Rust, and is much faster than Pandas. Polars dataframes aren't promised to be a one-to-one replacement for Pandas, but they are very similar, and can be used in most cases where Pandas is used with few, if any, changes.

Polars Specifics

Polars offers a fair bit of stuff for performance, as that is it's main selling point. Among them:

- Low memory parameter - this will try to load the data in a way that uses less memory, but may be slower.
- Lazy execution - Polars has options to work lazily, which means that it won't actually do work like loading data until it is needed.
- Parallel execution - Polars can use multiple threads to do work, which can speed things up.

```
if IN_COLAB:
  !pip install polars
import polars as pl
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting polars
  Downloading polars-0.16.18-cp37-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (16.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 16.4/16.4 MB 57.0 MB/s eta 0:00:00
Requirement already satisfied: typing_extensions>=4.0.1 in /usr/local/lib/python3.9/dist-packages (from polars) (
Installing collected packages: polars
Successfully installed polars-0.16.18
```

```
# Read file at zip path into a polars dataframe
df_polar = pl.read_csv(zip_path, ignore_errors=True, low_memory=True)
df_polar.head()
```

shape: (5, 31)

Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V1
i64	f64	f64	f64	f64	f64	f64	f64	f64	f64	f64	f64	f6
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	-0.5516	-0.6178(
0	1.191857	0.266151	0.16648	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	1.612727	1.06523
1	-1.358354	-1.340163	1.773209	0.37978	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	0.624501	0.06608
1	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	-0.226487	0.17822
2	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	-0.822843	0.53819

Using Polars

Polars doesn't have the same native support in TensorFlow as Pandas does, so we need to convert it to a Pandas dataframe or an array to feed it into any models. One thing that may be useful with Polars would be to split a very large csv into multiple smallers ones, that could then be loaded one at a time. Something like the function below could be adapted to load a csv into a Polars dataframe, do whatever data manipulation is needed, then write it out to several smaller csv files. The make_csv_dataset is able to natively read in multiple csv files.

Note: If we were actually doing something like this, it is likely easier to do a train-validation-test split as we write the output into different subfolders. Manipulating data is easier in a dataframe than a dataset.


```
if IN_COLAB:
  !mkdir polar_out
  !mkdir polar_out/train
  !mkdir polar_out/val


i = 0
split = 0.2
```

```
#Shuffle data. Takes a peak of 2x memory to do so.
df_polar = df_polar.sample(frac=1.0)
for frame in df_polar.iter_slices(n_rows=BATCH_SIZE):
    record_batch = frame
    if i % 5 == 0:
        fname = "polar_out/val/data_{}.csv".format(i)
    else:
        fname = "polar_out/train/data_{}.csv".format(i)
    record_batch.write_csv(fname)
    i += 1
```

Read Folder

We can create datasets from a folder of csv files. This is useful if we have a large csv file that we have split into multiple smaller ones. We can utilize any of the tuning things like cache and batch size to control the memory usage.

```
split_ds = tf.data.experimental.make_csv_dataset(
            file_pattern = "polar_out/train/*.csv",
            batch_size=64,
            num_epochs=BASE_EPOCHS,
            num_parallel_reads=20,
            shuffle_buffer_size=10000)
```

Generators

We can also make a generator for the above files. This one loads each file as one batch, so it fits with the idea above on embedding all the processing into the step that writes the files.

```
class CSVGenerator(Sequence):
    def __init__(self, folder_path, shuffle=True):
        self.folder_path = folder_path
        self.shuffle = shuffle
        self.files = sorted(os.listdir(self.folder_path))
        self.indexes = np.arange(len(self.files))
        if self.shuffle:
            np.random.shuffle(self.indexes)

    def __len__(self):
        return len(self.files)

    def __getitem__(self, index):
        file_path = os.path.join(self.folder_path, self.files[index])
        data = pd.read_csv(file_path)
        X = data.iloc[:, :-1].values
        y = data.iloc[:, -1].values
        return X, y



train_generator = CSVGenerator("polar_out/train")
val_generator = CSVGenerator("polar_out/val")


# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics="accuracy")

# Fit the model to the data
# time the fit

start = time.time()
model.fit(train_generator, epochs=BASE_EPOCHS, validation_data=val_generator, steps_per_epoch=len(train_generator))
end = time.time()
print("Time to fit: ", end - start)
```

```
Epoch 1/20
223/223 [==============================] - 4s 13ms/step - loss: 22.6675 - accuracy: 0.9893 - val_loss: nan - val_
Epoch 2/20
223/223 [==============================] - 3s 14ms/step - loss: 17.5551 - accuracy: 0.9937 - val_loss: nan - val_
Epoch 3/20
223/223 [==============================] - 3s 12ms/step - loss: 11.7901 - accuracy: 0.9983 - val_loss: nan - val_
                                                                                                  loss: nan - val_a
223/223 [==============================] - 3s 13ms/step - loss: 21.3341 - accuracy: 0.9932 - val_loss: nan - val_
Epoch 6/20
223/223 [==============================] - 3s 13ms/step - loss: 33.2795 - accuracy: 0.9983 - val_loss: nan - val_
Epoch 7/20
223/223 [==============================] - 3s 13ms/step - loss: 17.3493 - accuracy: 0.9983 - val_loss: nan - val_
Epoch 8/20
223/223 [==============================] - 4s 17ms/step - loss: 10.2571 - accuracy: 0.9936 - val_loss: nan - val_
Epoch 9/20
```

```
223/223 [==============================] - 3s 12ms/step - loss: 15.3892 - accuracy: 0.9938 - val_loss: nan - val_
Epoch 10/20
223/223 [==============================] - 3s 12ms/step - loss: 34.7191 - accuracy: 0.9983 - val_loss: nan - val_
Epoch 11/20
223/223 [==============================] - 5s 21ms/step - loss: 12.6141 - accuracy: 0.9975 - val_loss: nan - val_
Epoch 12/20
223/223 [==============================] - 3s 12ms/step - loss: 5.8039 - accuracy: 0.9924 - val_loss: nan - val_a
Epoch 13/20
223/223 [==============================] - 3s 15ms/step - loss: 34.7634 - accuracy: 0.9983 - val_loss: nan - val_
Epoch 14/20
223/223 [==============================] - 3s 12ms/step - loss: 14.0800 - accuracy: 0.9983 - val_loss: nan - val_
Epoch 15/20
223/223 [==============================] - 3s 12ms/step - loss: 0.8834 - accuracy: 0.9946 - val_loss: nan - val_a
Epoch 16/20
223/223 [==============================] - 3s 12ms/step - loss: 1.7343 - accuracy: 0.9937 - val_loss: nan - val_a
Epoch 17/20
223/223 [==============================] - 3s 14ms/step - loss: 18.2130 - accuracy: 0.9926 - val_loss: nan - val_
Epoch 18/20
223/223 [==============================] - 3s 12ms/step - loss: 13.2676 - accuracy: 0.9983 - val_loss: nan - val_
Epoch 19/20
223/223 [==============================] - 4s 20ms/step - loss: 27.5450 - accuracy: 0.9939 - val_loss: nan - val_
Epoch 20/20
223/223 [==============================] - 3s 12ms/step - loss: 25.6958 - accuracy: 0.9983 - val_loss: nan - val_
Time to fit:  88.34010601043701
```