Chapter 13

> Loading and Preprocessing Data With TensorFlow

GitHub Page: LINK
Google Drive: LINK

- All the datasets so far could all fit in RAM but often data are much larger
- It's easy with TensorFlow to ingest data thanks to Data API
- to use it just create a dataset object and tell it where to get the data and how to transform it
- Keras takes care of the rest
- Data API can read from:
    - text file (CSV, TXT, etc.)
    - binary files with fixed size record
    - binary files using TFRecord format
- Also support reading SQL databases
- Reading huge datasets efficiently isn't the only difficulty:
    - data needs to be preprocessed (usually normalized)
    - not always composed strictly of convenient numerical fields
- A solution might be to write custom preprocessing layers
- Another solution is to use the standard preprocessing layer provided by Keras
- This Chapter will cover:
    - data API
    - TFRecord format
    - create custom preprocessing layers

- Using Keras preprocessing layers
- TF Transform (tf.Transform)
- TF Datasets (TFDS)

The Data API

- The whole Data API revolves around the concept of a dataset
- Usually data is read from disk but the for simplicity the code below will be running on RAM

```
X = tf.range(10)
dataset = tf.data.Dataset.from_tensor_slices(X)
dataset
```

- from_tensor_slices() takes a tensor and creates a tf.data.Datset whose elements are the slices of X
- This is the same from:

```
dataset = tf.dataDataset.range(10)
```

- To iterate through a dataset

```
for item in dataset:
  print(item)
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/f2c57070-5fc2-4ae6-8844-684780b9e1ed/Data_API.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/1c90c660-e898-4f6d-a2e4-52f19c350aa2/data_api.py

Chaining Transformation

- Once you have the dataset, you can apply all sorts of transformations by calling its transformation methods

```
dataset = dataset.repeat(3).batch(7)
for item in dataset:
  print(item)
```

👉 Chaining dataset transformation diagram on page 415 fig 13-1

- Repeat method returns a dataset that repeats the dataset 3 times
- batch separates the the data in 7 batches in the example
- drop_remainder=True will drop the remainder so all the batches have the same size
- can also transform the items by calling map() method

```
dataset = dataset.map(lambda x: x * 2)
```

- Function passed to map must be TF function
- map() applies transformation to each item, apply method applies a transformation to the whole dataset

```
dataset = dataset.apply(tf.data.experimental.unbatch())
```

- unbatch() changes batches of 7 to single tensors
- Also possible to filter tensors

```
dataset = dataset.filter(lambda x: x < 10)
```

- To look at a few items use take() method

```
for item in dataset.take(3):
  print(item)
```

Shuffling the Data

- Gradient descent works best when data is independent and shuffled
- to shuffle the instance use method shuffle()
- Shuffle will create a new dataset from 1 and whenever data is requested it will return one from the buffer randomly
- Have to specifically mention the buffer size (As long as RAM can fit the data)

```
dataset = tf.data.Dataset.range(10).repeat(3)
dataset = dataset.shuffle(buffer_size=5, seed=41).batch(7)
for item in dataset:
  print(item)
```

- For large dataset that doesn't fit in memory this approach isn't sufficient
- A solution is to shuffle the source data itself
- May still be biased after shuffling source data
- To solve bias, shuffle the instance more by splitting the source data into multiple files and read them in random order

Interleaving lines from Multiple Files

REDO WHEN HAVE ACCESS TO GOOGLE COLAB OR LOCAL TENSORFLOW MACHINE

Preprocessing

- Implementing a function that perform preprocessing

```
X_mean, X_std = [...] # Mean and Scale of each feature in the training set
n_inputs = 8

def preprocessing(line):
  defs = [0] * n_inputs + [tf.constant([], type=tf.float32)]
  fields = tf.io.decode_csv(line, record_defaults=defs)
  x = tf.stack(fields[:-1)
  y = tf.stack(fields[-1:])
  return (x - X_mean) / X_std, y
```

- First code assumes there is precomputed mean and standard deviation

- X_mean and X_std are 1D tensors containing 8 floats
- preprocessing takes a line one CSV line to parse it
- it uses tf.io.decode_csv() function, which takes 2 arguments:
    - first argument is the variable to parse
    - second is an array containing the default values for each column
- decode_csv() returns a list of scalar tensors
- call stack() to return 1D tensors except the last value which is the target
- lastly, scale the input features by subtracting the feature mean and then divide by feature by standard deviation and return a tuple containing scaled feature and the target
- To preprocesses:

```
preprocess(b'4.382, 44.5, 5.23, 0.984, 758.3, 2.3892, 37.38, -128.38, 2.748')
```

Putting Everything Together

- put everything in the previous chapter together to make it reusable

```
def csv_redaer_dataset(filepaths, repeat=1, n_reader=5, n_read_threads=None,
                      shuffle_buffer_size=10000, n_parse_threads=5, batch_size=32):
  dataset = tf.data.Dataset.list_files(filepaths)
  dataset = dataset.interleave(lambda filepath :
          tf.data.TextLineDataset(filepath).skip(1), cycle_length = n_reader,
          num_parallel_calls = n_read_threads)
  dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
  dataset = dataset.shuffle(shuffle_buffer_size).repeat(repeat)
  return dataset.batch(batch_size).prefetch(1)
```

- Everything makes sense in the code except the last line

Prefetch

- calling prefetch(1) creates a dataset that is a batch ahead of the current batch
- Can exploit multiple CPU cores with hopefully reduce running time

👉 Diagram on Prefetch on page 422

- If training fits in memory it could significantly increase training time

Using Dataset with tf.keras

- use csv_reader_dataset() function to create a dataset for training

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepath)
test_set = csv_reader_dataset(test_filepath)
# create model and compile [...]
model.fit(train_set, epochs=10, validation_data=valid_set)
```

- Pass a dataset to evaluate and predict

```
model.evaluate(test_set)
new_set = test_set.take(3).map(lambda X, y: X)
model.predict(new_set)
```

The TFRecord Format

- TFRecord format is TensorFlow's preferred format to store large amount of data
- To use TFRecord:

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
  f.write(b"This is the first record")
  f.write(b"This is the second record")
```

- reading it:

```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
  print(item)
```

Compressed TFRecord Files

- useful to compress TFRecord files can be useful when loading on network

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter*"my_compressed.tfrecord", options) as f:
  [...]
  # Optional implementations
```

- Reading compressed file:

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"], compression_type="GZIP")
```

A Brief Introduction to Protocol Buffers

- TFRecord files usually contains serialized protocol bufferes (protobufs)
- This is a portable, extensible and efficient binary format developed at Google in 2001
- Open source in 2008
- Defined in simple language that looks like this:

```
syntax = "proto3";
message Person {
  string name = 1;
  int32 id = 2;
  repeated string email = 3;
}
```

- simple example that uses generated class Person

```
from person_pb2 import Person
person = Person(name="Al", id=123, email=["a@b.com"]
print(person)
print(person.name)
person.name = "Alice"
print(person.email)
person.email.append("c@d.com")
s = person.SerializeToString()
print(s)
person2 = Person()
person2.ParseFromString(s)
person == person2
```

- Imported Person class generated by protoc

- Create an instance and play with it
- serialized it using SerializeToString() method
- Ready to transmit over the network
- ParseFromStrong copy the object that was serialized
- Could save the serialized Person object to a TFRecord file and parse it
- Those are not options for TensorFlow

TensorFlow Protobufs

- Protobufs typically used in a TFRecord file
- Protobuf definition:

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message BytesList { repeated float value = 1 [packed = true]; }
message BytesList { repeated int64 value = 1 [packed = true]; }
message Feature {
  oneof kind{
      BytesList bytes_list = 1;
      FloatList float_list = 2;
      Int64List int64_list = 3;
  }
}
```

- [packed = true] is used for repeated numerical fields for more efficient encoding
- Code for protobusfs:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2022acd5-b0b7-4512-944b-71a142715227/protobufs.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/23802508-116d-4b00-abd8-c460056cd5ff/protobufs.py

Loading and Parsing Examples

- To load the serialized Example protobufs, use a tf.data.TFRecordDataset
- requires at least 2 arguments, string scalar tensor containing the serialized data and a description of each feature
- Following code defines a description dictionary and then iterates over the TFRecord Dataset and parse the serialized example

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/918d3ff2-2bb3-4857-a79f-7f5e282b2e46/loading_and_parsing.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e7d74cd8-2fad-477d-9f17-7e256a905ddc/loading_and_parsing.ipynb

Handling Lists of Lists Using the SequenceExample Protobuf

- A sequence example using protobuf:

```
message FeatureList { repeated Feature feature = 1; }
message FeatureLists { map<string, FeatureList> feature_list = 1; }
message SequenceExample {
  Features context = 1;
  FeatureLists feature_lists = 2;
}
```

- Each feature list contains a list of Feature objects
- If the feature lists contain sequences of varying sizes, may be better to convert them to ragged tensors using tf.RaggedTensor.from_sparse()

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
  serialized_sequence_example, context_feature_descriptions,
  sequence_feature_descriptions)
  parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Preprocessing the Input Features

- Preparing data for neural network requires converting all features into numerical features
- If data feature text, it also have to converted to numbers
- For example, here is an implementation for a standardization layer using a Lambda layer

```
means = np.mean(train_data, axis=0, keepdims=True)
stds = np.std(train_data, axis=0, keepdims=True)
eps = keras.backend.epsilon()
model = keras.models.Sequential([
  keras.layers.Lambda(lambda inputs: (inputs - mean) / (stds + eps)),
  # Other Layers [...]
])
```

- A custom layer may look better

```
class Standardization(keras.layers.Layer):
  def adapt(self, data_sample):
    self.means_ = np.mean(data_sample, axis=0, keepdims=True)
    self.stds_ = np.mean(data_sample, axis=0, keepdims=True)
  def call(self, inputs):
    return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
```

- before using layer, it's required to adapt it into the dataset by calling adapt() method

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

- Next, use this preprocessing layer like a normal layer

```
model = keras.Sequential()
model.add(std_layer)
```

Encoding Categorical Features Using One-Hot Vectors

- Consider Ocean proximity feature in California housing dataset
- There's five possible values:
  - 1h ocean

    - Inland
    - Near Ocean
    - Near Bay
    - Island

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num-oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

- first defines vocabulary
- Create tensors corresponding indices
- Create an initializer for lookup table, passing categories and their corresponding indices
- If categories are listed in a text file then use TextFileInitializer Instead of KeyValueTensorInitializer
- Last 2 lines create a lookup table and specify the number of oov buckets
- If unknown vocabulary is looked up, it will assign it to unknown category
- The reason to use oov buckets:
    - If number of categories are too large or keeps changing, listing categories may not be convenient
    - A solution would be to add some oov buckets for the other categories that were not in the data sample
    - the more unknown data the more oov buckets should be used
- use the lookup table to encode small batch of categorical features

```
categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
cat_indices = table.lookup(categories)
print(cat_indices)
cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
print(cat_one_hot)
```

- This may not be the best solution since the size of each one-hot vector is the vocabular length plus the number of oov buckets

- It's more efficient in large dataset by using **embeddings**

Encoding Categorical Features Using Embedding

- Am embedding is a trainable dense vector that represents a category
- embedding are generated randomly
- Embedding will gradually improve during training with gradient descent

👉 Two diagrams on page 334 and 335 regarding embedding

- Here is how to implement embeddings manually to understand how it works

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

- Using 2D embedding in the example
- embedding matrix is random 6 * 2 matrix stored in variable so can be tweaked during training

```
print(embedding_matrix)
categories = tf.constant(["NEAR BAY", "DESER", "INLAND", "INLAND"])
cat_indices = table.lookup(categories)
print(cat_indices)
tf.nn.embedding_lookup(embedding_matrix, cat_indices)
```

- tf.nn.embedding_lookup() function looks up the rows in the embedding matrix
- Keras provides a embedding layer that handles the embedding matrix

```
embedding = keras.layers.Embedding(input_dim=len(vocab) + num_oov_buckets,
  output_dim=embedding_dim)
print(embedding(cat_indices)
```

- To put them all together, it is possible to create a Keras model that can process categorical

```
regular_input = keras.layers.Input(shape[8])
categories = keras.layers.Input(shape=[], dtype=tf.string)
cat_indices = keras.layer.Lambda(lambda cats: table.lookup(cats))(categories)
cat_embed = leras.layers.Embedding(input_dim=6 output_dim=2)(cat_indices)
encode_inputs = keras.layers.concatenate([regular_inputs, cat_embed])
outputs = keras.layers.concatenate([regular_inputs, cat_embed])
outputs = keras.layers.Dense(1)(encoded_inputs_
model = keras.models.Model(inputs[regular_inputs, categories], outputs=[outputs])
```

- Model takes 2 inputs containing 8 numerical features
- Uses Lambda layer to look up each categorical input
- It uses a Lambda layer to look up each category's index
- then it looks up the embedding for these indices
- Next it concatenates the embedding and the regular inputs
- then add a dense output layer

Keras Preprocessing Layers

- The API will also include a keras.layers.Discretization layer that will chop continuous data into different bins and encode each bin as one hot vector
- Discretization is not differentiable and should only be used at the start of the model
- Also possible to chain multiple preprocessing layer

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...]) # TODO
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

- TextVectorization layer will also have an option to output word-count vectors instead of word indices
- This is called bags of words
- Common words will have a large value in most texts
- A common way to normalize in a way that reduces the importance of frequency words is by a technique called TF-IDF
    - Term-Frequency $\times$ Inverse-Document-Frequency
- Keras preprocessing layers will make preprocessing much easier

TF Transform

- If preprocessing is computationally expensive then handling it before training rather than on the fly is a better option
- The data will preprocess once before training rather per instance every epoch
- If data fits on ram than use cache() method if not then tools like Apache Beam or Spark will help
- If you deploy the model on android and web page, when you want to update any code:
    - You would have to change the JS code, preprocessing logic, Apache Beam code
    - It's very time consuming
- A improvement would be before deploying, add preprocessing layer for on the fly processing on top it
- Another solution to just define preprocessing operations just once is TF Transform
- TF Transform is a part of TensorFlow Extended (TFX)
- Here's what a preprocessing layer could look like

```
import tensorflow_transform as tft

def preprocess(input):
  median_age = inputs["house_median_age"]
  ocean_proximity = inputs["ocean_proximity"]
  standardized_age = tft.scale_to_z_score(median_age)
  ocean-proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
  return {
    "standardized_median_age" : standardized_age,
    "ocean_proximity_id" : ocean_proximity_id
  }
```

- TF Transformation lets you apply preprocess() function to the whole training set using Apache Beam
- It would also compute all the necessary statistics over the whole training set
- TF Transform will also generate an equivalent TensorFlow Function that you can plug into the model you deploy

- If just require a standard dataset, using TFDS is adequate

The TensorFlow Datasets (TFDS) Project

- TensorFlow includes a lot of datasets from small ones like MNIST to large ones like ImageNet
- TFDS is not bundled therefore it's required to do a pip install before using
- "tensorflow-datasets"
- An example of downloading mnist

```
import tensorflow_datasets as tfds

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

- Then apply transformation

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
  images = item["image"]
  labels = item["label"]
  #[...] more code
```

- A simpler way:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].prefetch(1)
model = keras.model.Sequential([...]) # TODO
model.compile([...]) # TODO
model.fit([...]) #TODO
```

Exercises

1. Processing Data could be difficult but Data API makes it simple, it's possible to read text files (CSV, TXT, etc...), binary files and TFRecord Files
2. It solves bias when shuffling the data around and it also helps to compute on different machines at the same time
3. It's possible to visualize GPU usage with TensorBoard and if the utilization isn't fully used, then it's likely there's a bottleneck, a way to solve this issue

is by using prefetch to get one or multiple steps ahead of what is about to be trained

4. You can save any binary data on a TFRecord file and not just serialized protocol. but most of the time TFRecord file contains serialized buffers
5. A reason to use protobufs instead of defining own protobufs is because in Example Protobuf format it contains lots of TensorFlow defined operations which could be really useful and it's hard to implement manually
6. You would want active compression when you are transferring data from any models that is online since that would save a lot of space. It does not do it automatically because it's not always required to compress it (For example using it on a local machine) and it requires extra computational power
7.
   - When data is preprocessed directly when writing datafile, it's faster for training the model but it's more difficult to change the code later on if you are deploying online

   - If preprocessing is built in the pipeline then it's easier to tune and easier to change the values if needed but the down side to this solution is that it's more computationally expensive

   - This is probably the worst solution out of all since you would have to write all the files and it takes lots of time to change just a single value in the model and it's computationally expensive since it's preprocessing during training

   - TF Transform gives most of the benefits from the previous processing techniques, it only preprocess once so it doesn't waste computational power and it only requires to change the code once, the only drawback is the fact that you would have to learn it before you can use it which isn't significant since you are required to learn other preprocessing techniques too
8. For encoding text that has natural order like "bad", "okay", "good", it's good to just use ordinary encoding. For more complex text and sentences it's good to use bags of words
9. Code:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9a40668d-3a98-4e5b-9dca-b527a7aa0f76/excerciese_9.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/d3c958c1-a171-4ab6-b4a5-f5ab83e799fb/excerciese_9.py

10. Code:

    https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3cd7d5bb-541b-48df-95de-62123892156e/excercise_10.ipynb

    https://s3-us-west-2.amazonaws.com/secure.notion-static.com/90e818b2-5e7b-441f-994a-c6a227dc2b34/excercise_10.py