Loading and Preprocessing Data with TensorFlow

Deep Learning
Stijn Lievens and Sabine Devreese
2023-2024

Loading and Preprocessing Data with TensorFlow

Overview

- The data loading and preprocessing API is available in the `tf.data` package.
  - It is capable of loading and preprocessing data very efficiently.
  - It lets you handle datasets that are too large to fit in memory.
- `TFRecord` is a flexible and efficient binary format.
  - It usually contains protocol buffers.
- Keras also provides a few preprocessing layers.
  - They can be embedded in a Keras model.

13.1 The `tf.data` API

`tf.data.Dataset`

- A `tf.data.Dataset` represents a sequence of data items.
  - Usually datasets will gradually read the data from disk.
- One the next slide, we create a simple dataset using `tf.data.Dataset.from_tensor_slices`.
  - It takes a tensor and creates a `tf.data.Dataset` whose elements are all the slices of `X` (along the first dimension).

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

You can then iterate over this dataset and process it like this:

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

which yields:

```
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

13.1.1 Chaining Transformations

Chaining Transformations

- You can apply all sorts of **transformations** on a dataset by calling its transformation methods.
  - Each method returns a new dataset.
  - Transformations can be chained as demonstrated on the next slide.

Batching and Repeating

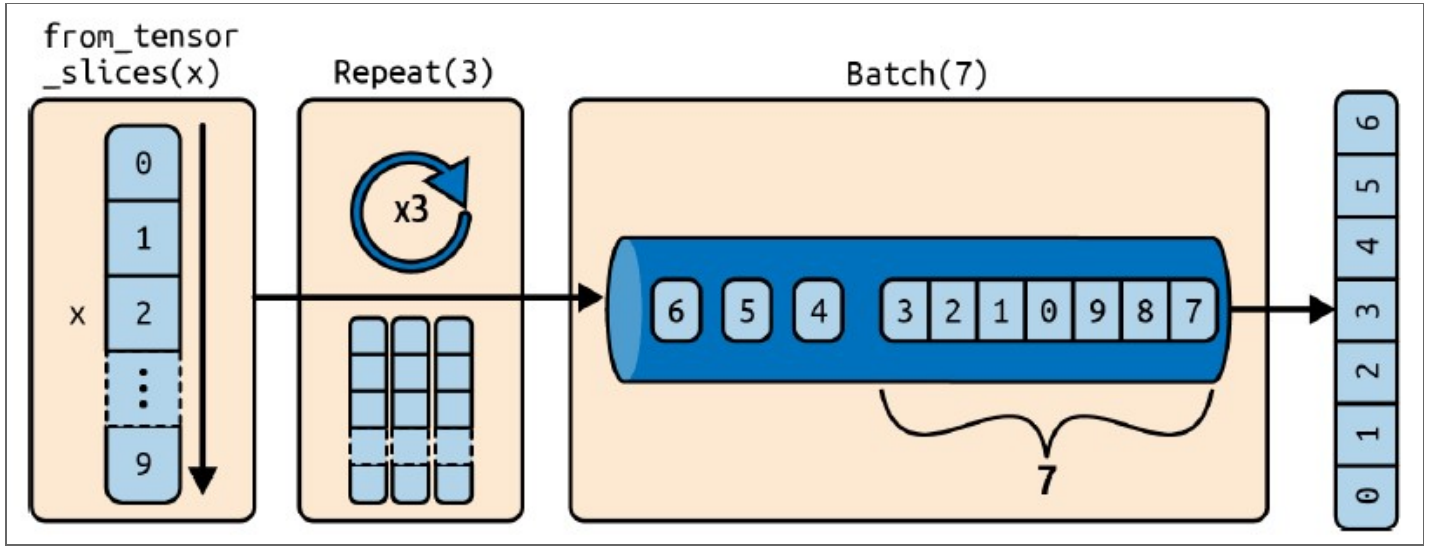
```
dataset = tf.data.Dataset.from_tensor_slices(tf.range(10))
dataset = dataset.repeat(3).batch(7)
for item in dataset:
    print(item)
```

yields

```
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

- If you set `drop_remainder=True` on the `batch()` method, then the last batch will be dropped if it is not full.

Batching and Repeating



Batching and Repeating

Caution!

The dataset methods do not modify datasets, they create new ones! Make sure to create a reference to these new datasets or else nothing will happen.

`map` over a Dataset

```
dataset = dataset.map(lambda x: x * 2)  # x is a batch
for item in dataset:
    print(item)
```

yields

```
tf.Tensor([ 0  2  4  6  8 10 12], shape=(7,), dtype=int32)
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
tf.Tensor([ 2  4  6  8 10 12 14], shape=(7,), dtype=int32)
tf.Tensor([16 18], shape=(2,), dtype=int32)
```

You can use multiple CPU cores by specifying `num_parallel_calls`. Use `tf.data.AUTOTUNE` to let TensorFlow decide how many threads to run.

`filter` on a Dataset

```
dataset = dataset.filter(lambda x: tf.reduce_sum(x) > 50)
for item in dataset:
    print(item)
```

only keeps those items (batches) whose sum is greater than 50:

```
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
tf.Tensor([ 2  4  6  8 10 12 14], shape=(7,), dtype=int32)
```

`take` on a Dataset

- With `take` you can easily look at the first few elements of a dataset.

```
for item in dataset.take(2):
    print(item)
```

yields

```
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
```

13.1.2 Shuffling the Data

Shuffling the Data

- Gradient descent works best when the training samples are independent and identically distributed (IID).
  - This is why we usually shuffle the training set.

The `shuffle` Method

- The `shuffle` method creates a new dataset that will start by filling up a buffer with the first items of the source dataset.
  - Then, whenever it is asked for an item, it will pull one out randomly from the buffer and replace it with the first element not in the buffer from the source dataset, until it has iterated through the source dataset once.
  - The `buffer_size` argument controls the size of the buffer.
- The value for `buffer_size` must be large enough or shuffling will not be very effective!

The `shuffle` Method

```
dataset = tf.data.Dataset.range(10).repeat(2)
dataset = dataset.shuffle(buffer_size=4, seed=42).batch(7)
for item in dataset:
    print(item)
```

yields

```
tf.Tensor([1 4 2 3 5 0 6], shape=(7,), dtype=int64)
tf.Tensor([9 8 2 0 3 1 4], shape=(7,), dtype=int64)
tf.Tensor([5 7 9 6 7 8], shape=(6,), dtype=int64)
```

- Note how the use of `seed` ensures the same 'random' order at each run.

Repeating a Shuffled Dataset

- If you call `repeat()` after `shuffle()`, you will get a different order at each iteration.
  - This is generally a good idea.
  - However, if you set `reshuffle_each_iteration=False`, then the order will be the same at each iteration.

13.1.3 Interleaving Lines from Multiple Files

Overview

We will show how to:

- Pick multiple files randomly.
- Read these simultaneously, interleaving their lines.
- Add some more shuffling.

Setup

- We assume that
  - The California housing dataset was split into a training, validation and test set.
  - Each set was split into many CSV files.
- Let's suppose `train_filepaths` contains the list of training file paths.

```
train_filepaths # ['datasets/housing/my_train_00.csv',
                #  'datasets/housing/my_train_01.csv', ...]
```

Step 1: Create a Dataset of File Paths

Use the `list_files` static method on `Dataset` to create a dataset containing these file paths.

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

Note: the filepaths are shuffled by default! Set `shuffle=False` if you don't want this.

```
for filepath in filepath_dataset.take(2):
    print(filepath)
# tf.Tensor(b'datasets/housing/my_train_05.csv', shape=(), dtype=string)
# tf.Tensor(b'datasets/housing/my_train_16.csv', shape=(), dtype=string)
```

Step 2: Read from Multiple Files

- The `interleave` can be called to read from multiple files simultaneously.
  - For `interleave` we need a `map_func` that takes a dataset item (in this case, a file path) and returns a Dataset.
  - In this case we will create a `TextLineDataset` for each file path and skip the first line (header row).
  - `cycle_length` controls the number of files to read in parallel.

Step 2: Read from Multiple Files

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

- Iterating over this dataset will result in lines from 5 different files being read and interleaved.
- Then, another five (random) files will be read, and so on until all files are read.
- Note: interleaving works best if the files have identical length; the end of longest file will not be interleaved

Step 2: Read from Multiple Files

```
for line in dataset.take(5):
    print(line)
```

yields

```
tf.Tensor(b'4.5909,[...],33.63,-117.71,2.418', shape=(), dtype=string)
tf.Tensor(b'2.4792,[...],34.18,-118.38,2.0', shape=(), dtype=string)
tf.Tensor(b'4.2708,[...],37.48,-122.19,2.67', shape=(), dtype=string)
tf.Tensor(b'2.1856,[...],32.76,-117.12,1.205', shape=(), dtype=string)
tf.Tensor(b'4.1812,[...],33.73,-118.31,3.215', shape=(), dtype=string)
```

These are the first (data) rows of five different CSV files.

13.1.4 Preprocessing the Data

Preprocessing the Data

- At this point the dataset is returned as byte strings.
  - We want to convert these into numerical tensors, and also scale the inputs.

```
# mean and standard deviation for each feature in the training set
X_mean, X_std = [...]
n_inputs = 8

def parse_csv_line(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    return tf.stack(fields[:-1]), tf.stack(fields[-1:])

def preprocess(line):
    x, y = parse_csv_line(line)
    return (x - X_mean) / X_std, y
```

```
# Repeated from previous slide
n_inputs = 8

def parse_csv_line(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    return tf.stack(fields[:-1]), tf.stack(fields[-1:])
```

- `decode_csv` takes a line (a tensor of type string) and a list of default values for each column (a list of tensors, one per column)
  - In this case the default values are all 0.0, except for the last one which doesn't have a default value.
- `decode_csv` returns a list of Tensor objects.
  - We want two (array) tensors: one containing the features, and one containing the labels: `stack` does this.

```
# Repeated from previous slide
def preprocess(line):
    x, y = parse_csv_line(line)
    return (x - X_mean) / X_std, y
```

- The `preprocess` function is easy: it simply takes a line (a String Tensor), parses it, and scales the features.
  - A tuple containing the scaled features and the label is returned.

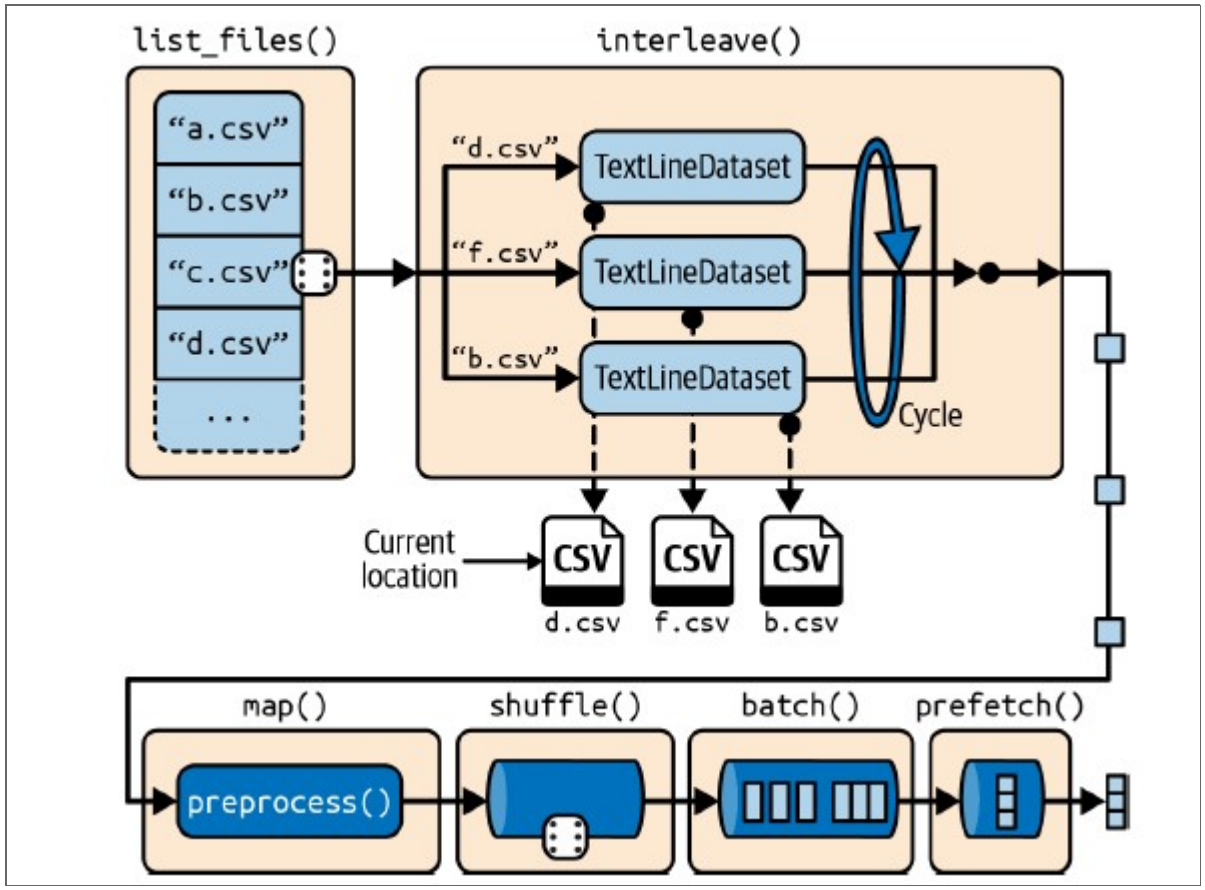
Test the `preprocess` function

```
preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
```

gives

```
(<tf.Tensor: shape=(8,), dtype=float32, numpy=
 array([ 0.16579159,  1.216324  , -0.05204564, -0.39215982, -0.5277444 ,
        -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32,
  numpy=array([2.782], dtype=float32)>)
```

13.1.5 Putting Everything Together



Loading and Preprocessing the dataset

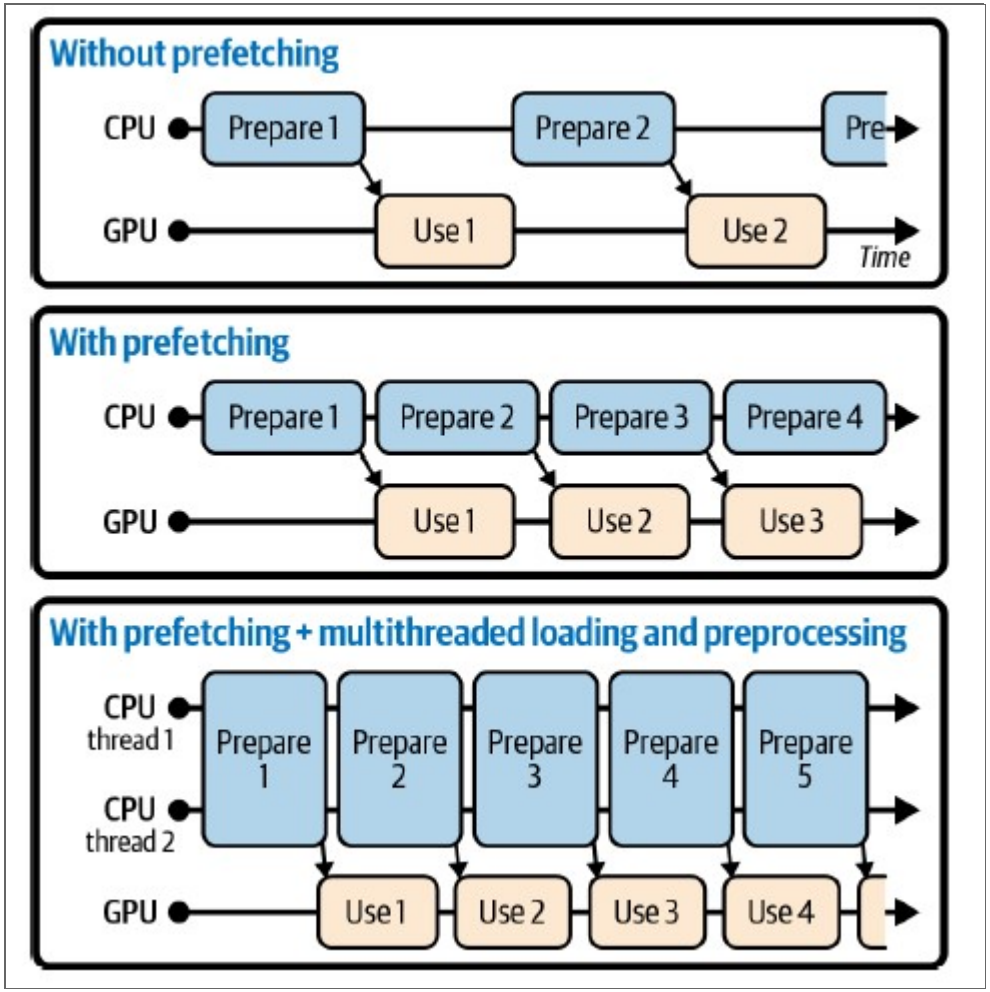
Putting Everything Together

```
def csv_reader_dataset(filepaths, n_readers=5, n_read_threads=None,
                       n_parse_threads=5, shuffle_buffer_size=10_000,
                       seed=42, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths, seed=seed)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess,
          num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size, seed=seed)
    return dataset.batch(batch_size).prefetch(1)
```

13.1.6 Prefetching

Prefetching

- By calling `prefetch(1)` at the end of the `csv_reader_dataset` function, we are creating a dataset that will do its best to always be one batch ahead.
  - While the training algorithm is working on one batch, the dataset will already be working in parallel on getting the next batch ready.
- If preparing the next batch is shorter than running a training step on the GPU, then the GPU will be almost 100% utilized.
  - “Almost”: because data needs to be transferred from the CPU to the GPU.



Effect of prefetching

Caching a Dataset

- If the Dataset is small enough to fit into memory, calling `cache()` on it will cache its content in RAM.
  - This will speed up the training by a lot.
- Typically you call `cache` after loading and preprocessing, but before shuffling and batching.
  - This way the data will only be loaded and preprocessed once (instead of once per epoch).

13.1.7 Using the Dataset with Keras

Creating Datasets

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

Training the Model

- We can now pass `train_set` instead of `X_train` and `y_train`.
- Similarly, we can pass `validation_data=valid_set` instead of `validation_data=(X_valid, y_valid)`.
- `fit` will repeat the training set once per epoch.
  - If will use a different random order at each epoch.

Training the Model

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(30, activation="relu",
                          kernel_initializer="he_normal",
                          input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(1),
])
model.compile(loss="mse", optimizer="sgd")
# Note use of train_set and valid_set !
model.fit(train_set, validation_data=valid_set, epochs=5)
```

Evaluating and Making Predictions

- Of course, you can also use `Dataset`s for evaluation.
  - You can pass a `Dataset` to `evaluate` instead of `X_test` and `y_test`.

```
test_mse = model.evaluate(test_set)
new_set = test_set.take(3)  # pretend we have 3 new samples
y_pred = model.predict(new_set)
```

- Note, normally the `Dataset` passed to `predict` doesn't contain labels.
  - Here, the labels *are* present but Keras ignores them in this case.

13.2 The `TFRecord` Format

13.3 Keras Preprocessing Layers

Preprocessing Options

Preparing data for a neural network typically requires a lot of preprocessing: normalizing, scaling, encoding categorical features, cropping and resizing images and so on.

Preprocessing Options

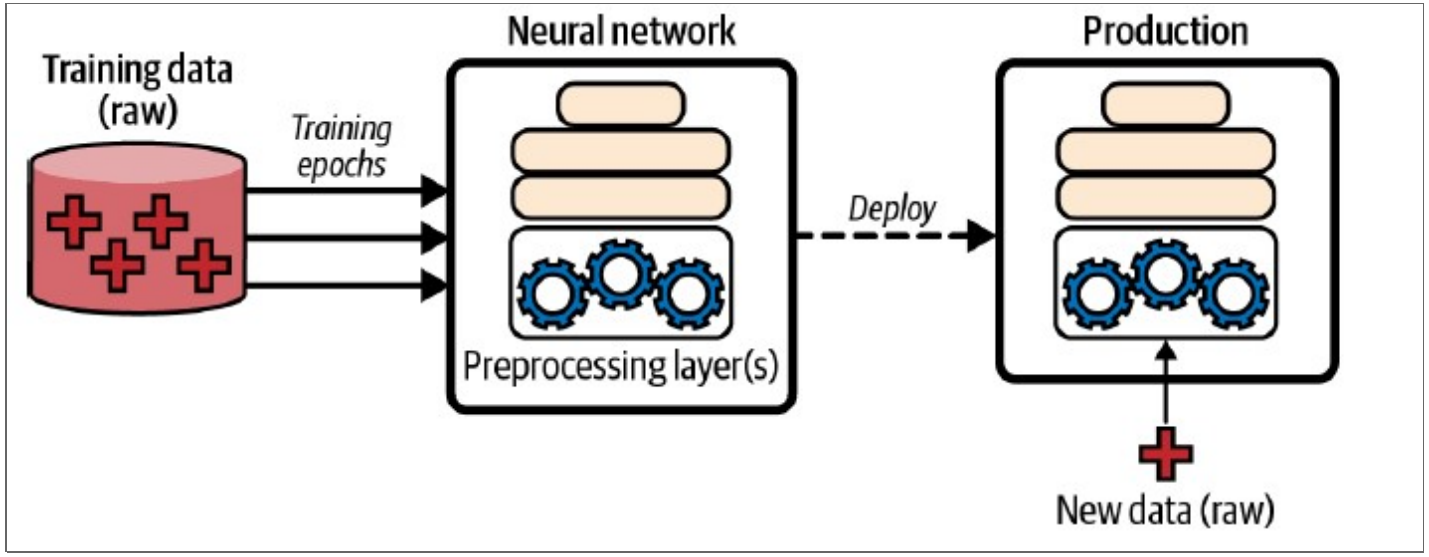
- Do the preprocessing outside of the model, using NumPy arrays or pandas DataFrames, Scikit-Learn, etc.
  - You will need the exact same steps in production!
- Preprocess on the fly while loading the data, e.g. using `tf.data` and applying a preprocessing function using the `map` method.
  - Again, you will need the exact same preprocessing steps in production!
- Include the preprocessing layers inside your model and preprocess the data on the fly during training.
  - You can use the exact same preprocessing layers in production.

13.3.1 The `Normalization` Layer

Including the `Normalization` Layer in the Model

```
norm_layer = tf.keras.layers.Normalization()
model = tf.keras.models.Sequential([
    norm_layer,  # normalization layer *inside* the model
    tf.keras.layers.Dense(1)
])
model.compile(loss="mse",
              optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))
norm_layer.adapt(X_train)  # compute mean & variance of every feature
model.fit(X_train, y_train, validation_data=(X_valid, y_valid),
          epochs=5)
```

Preprocessing inside the Model

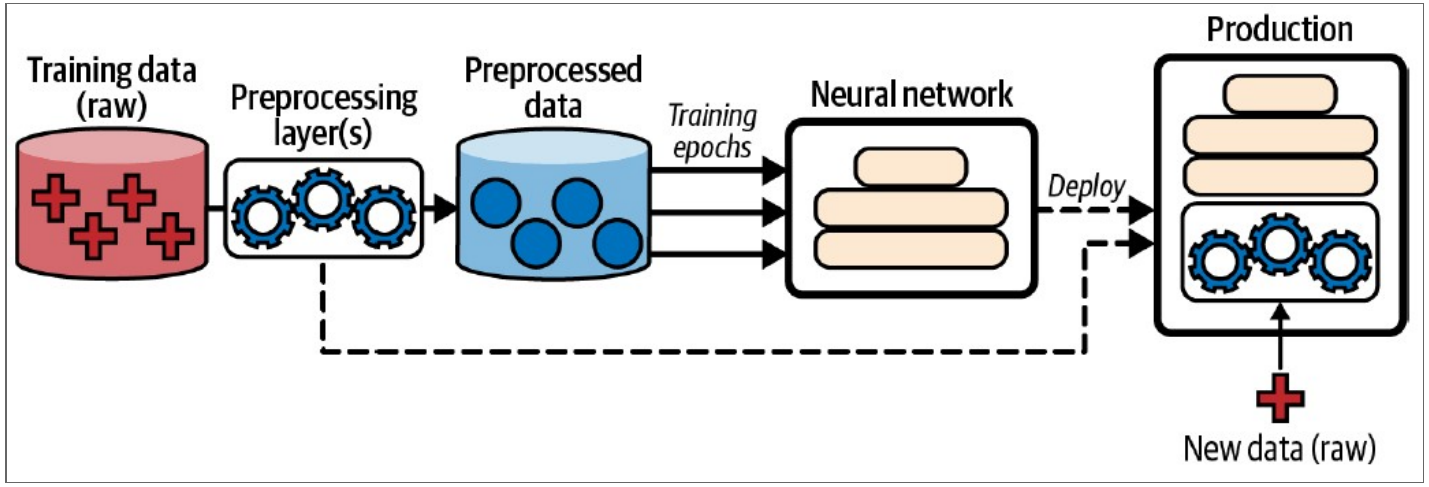


Using a preprocessing layer inside the model

Preprocessing before Training

- Performing the normalization inside the model will slow down training.
  - Data is normalized once per epoch!
- It is more efficient to normalize all the data before training.
  - Model can be trained *without* the normalization layer.
  - This will be faster.
- Finally, create a new model with the normalization layer and use it to make predictions on new instances.
  - This avoids preprocessing mismatch at prediction time.

Including the Preprocessing Layers after Training



Include preprocessing layer into production model

Train on Preprocessed Data

```
# Step 1: preprocess the data
norm_layer = tf.keras.layers.Normalization()
norm_layer.adapt(X_train)
X_train_scaled = norm_layer(X_train)
X_valid_scaled = norm_layer(X_valid)
```

```
# Step 2: train the model
model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
model.compile(loss="mse",
              optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))
model.fit(X_train_scaled, y_train, epochs=5,
          validation_data=(X_valid_scaled, y_valid))
```

```
# Step 3: Create a model for production.
final_model = tf.keras.Sequential([norm_layer, model])
```

Interaction with `tf.data.Dataset`

Preprocessing layers work seamlessly with `tf.data.Dataset` objects.

```
dataset = dataset.map(lambda X, y: (norm_layer(X), y))
```

Note: if this dataset fits into RAM you should probably `cache()` it after the `map()` call.

13.3.2 The `Discretization` Layer

The `Discretization` Layer

- The goal of this layer is to transform a numerical feature into a categorical feature.
  - Can be useful with multimodal distributions or features with a highly non-linear relationship with the target.
- You can either provide the bin boundaries yourself, or you can let `adapt()` find the boundaries based on the value percentiles.

Example Usage

```
# Create a constant tensor for demonstration purposes.
age = tf.constant([[10.], [93.], [57.], [18.], [37.], [5.]])
discretize_layer = tf.keras.layers.Discretization(
  bin_boundaries=[18., 50.])
# Note! NO adapt needed.
age_categories = discretize_layer(age)
age_categories
```

gives

```
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=
array([[0],
       [2],
       [2],
       [1],
       [1],
       [0]])>
```

Example Usage

```
age = tf.constant([[10.], [93.], [57.], [18.], [37.], [5.]])
discretize_layer = tf.keras.layers.Discretization(num_bins=3)
discretize_layer.adapt(age) # finds the boundaries based on the data
age_categories = discretize_layer(age)
age_categories
```

gives

```
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=
array([[1],
       [2],
       [2],
       [1],
       [2],
       [0]])>
```

13.3.3 The CategoryEncoding Layer

One-Hot Encoding

- When the number of categories is limited then often one-hot encoding is a good option.
- keras provides a `CategoryEncoding` layer for this purpose.

```
age_categories = tf.constant([[0], [2], [2], [1], [2], [0]])
onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3)
onehot_layer(age_categories)
```

gives

```
<tf.Tensor: shape=(6, 3), dtype=float32, numpy=
array([[1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]], dtype=float32)>
```

13.3.4 The StringLookup Layer

The `StringLookup` Layer

- `StringLookup` is used to (one-hot) encode categorical String features.
- You need to `adapt` it to the data.
- You then use the layer to encode Strings:
  - Unknown categories are mapped to 0.
  - Known categories are numbered from 1, from the most frequent to the least frequent.

Example

```
cities = ["Auckland", "Paris", "Paris", "San Francisco"]
str_lookup_layer = tf.keras.layers.StringLookup()
str_lookup_layer.adapt(cities)
str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
```

gives

```
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=
array([[1],
       [3],
       [3],
       [0]])>
```

`StringLookup` with One-Hot Encoding

If you set `output_mode="one_hot"` then the layer will output a one-hot vector for each instance.

```
cities = ["Auckland", "Paris", "Paris", "San Francisco"]
# Set output_mode="one_hot" to get one-hot vectors.
str_lookup_layer = tf.keras.layers.StringLookup(output_mode="one_hot")
str_lookup_layer.adapt(cities)
str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
```

gives

```
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [1., 0., 0., 0.]], dtype=float32)>
```

Number of OOV-Indices

- By default, the `StringLookup` layer will only use index zero for the out-of-vocabulary (OOV) category.
  - Thus, each unknown word gets mapped to zero.
  - This makes all these words indistinguishable.
- You can use more OOV-indices by setting `num_oov_indices` to a value greater than one.

```
str_lookup_layer = tf.keras.layers.StringLookup(num_oov_indices=5)
```

13.3.5 The Hashing Layer

The `Hashing` Layer

- The `Hashing` layer assigns categories to buckets or bins using a hash function.
  - The mapping is pseudo-random, but stable across runs and platforms. I.e. the same category will always be assigned to the same bucket.
- This layer does *not* need to be adapted to the data.

Example of `Hashing` layer

```
hashing_layer = tf.keras.layers.Hashing(num_bins=10)
hashing_layer([["Paris"], ["Tokyo"], ["Auckland"], ["Montreal"]])
```

gives

```
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=
array([[0],
       [1],
       [9],
       [1]])>
```

Note how “Tokyo” and “Montreal” are mapped to the same bucket. This is a *hashing collision*.

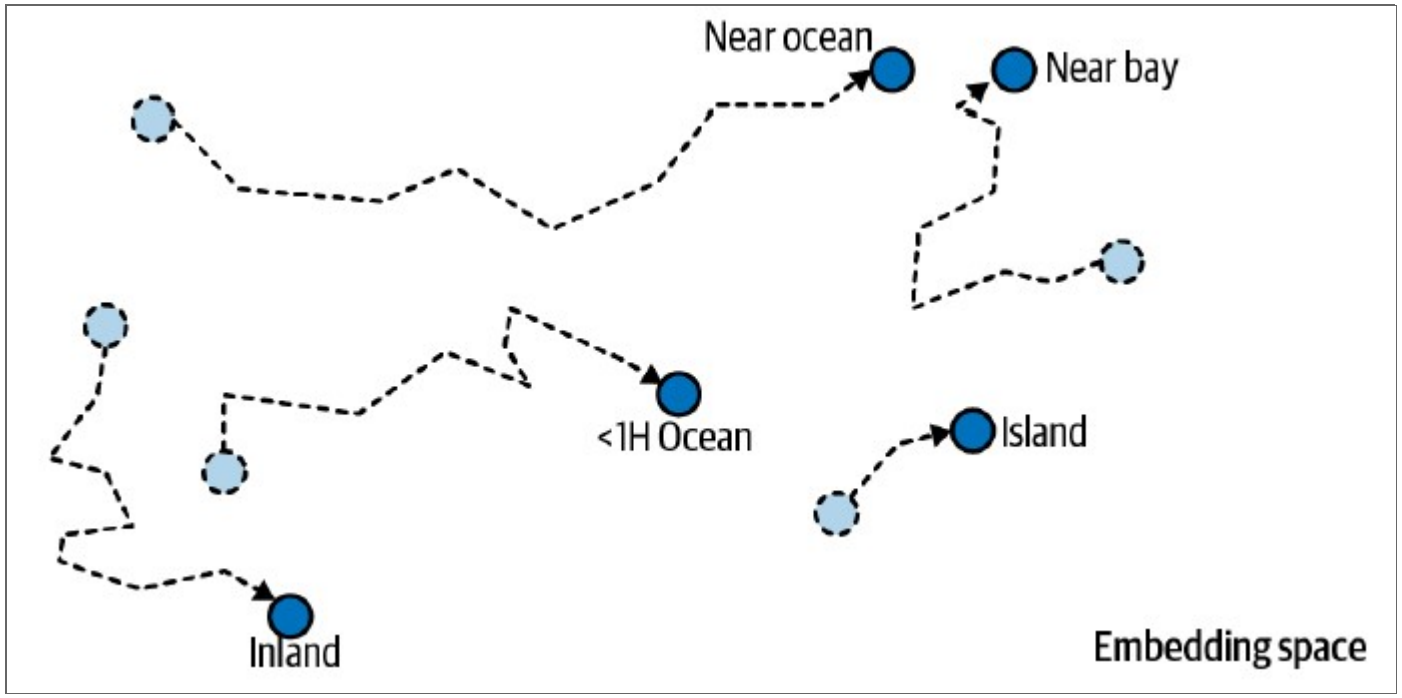
13.3.6 Encoding Categorical Features Using Embeddings

Embeddings

- An **embedding** is a dense representation of some high(er)-dimensional data, such as a category or a word (in a vocabulary).
- If the number of categories is very large, e.g. 50,000 words in a vocabulary, then it is not feasible to use one-hot encoding.
  - The one-hot vector would have 50,000 dimensions, and all but one of the dimensions would be zero.
  - This is very inefficient.
- An embedding by contrast is a much smaller (e.g. 100 dimensions) but dense vector.

Trainable Embeddings

- In deep learning, embeddings are typically initialized randomly and then trained to fit the data.
  - Thus, the embedding is **trainable**.
  - Typically, categories with similar meaning will have similar embeddings.
  - E.g. in the California housing dataset, the categories “NEAR OCEAN” and “NEAR BAY” will (probably) end up with similar embeddings.
- This is called **representation learning**.
  - The model learns a representation of the data that is useful for the task at hand.



Embeddings improving during training

The `Embedding` Layer

- Keras provides an `Embedding` layer, which wraps the **embedding matrix**.
  - This matrix has one row per category (e.g. per word in the vocabulary), and one column per embedding dimension.
  - Thus, if we have 50000 words and the embedding dimension is 100, then the embedding matrix will have shape `(50000, 100)`.
- Converting a category ID to its embedding is simply looking up the corresponding row in the embedding matrix.

Example of `Embedding` layer

In the code below, we initialize an embedding layer to embed 5 categories into 2 dimensions, and use it to embed the category IDs 2, 4, and 2.

```
tf.random.set_seed(42)
embedding_layer = tf.keras.layers.Embedding(input_dim=5, output_dim=2)
embedding_layer(np.array([2, 4, 2]))
```

gives

```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.04663396,  0.01846724],
       [-0.02736737, -0.02768031],
       [-0.04663396,  0.01846724]], dtype=float32)>
```

Embedding Categorical Text

- To embed a categorical text feature, first encode it using a `StringLookup` layer, then embed it using an `Embedding` layer.

```
ocean_prox = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
str_lookup_layer = tf.keras.layers.StringLookup()
str_lookup_layer.adapt(ocean_prox)
lookup_and_embed = tf.keras.Sequential([
  # InputLayer is WORKAROUND for Keras bug.
  tf.keras.layers.InputLayer(input_shape=[], dtype=tf.string),
  str_lookup_layer,
  tf.keras.layers.Embedding(input_dim=str_lookup_layer.vocabulary_size(),
    output_dim=2)
])
```

- Note the use of `vocabulary_size()` to get the number of categories and use it as the input dimension of the `Embedding` layer.

Embedding Categorical Text

```
# Continued from previous slide.
lookup_and_embed = ....
lookup_and_embed(np.array(["<1H OCEAN", "ISLAND", "<1H OCEAN"]))
```

could give something like

```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.01896119,  0.02223358],
       [ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358]], dtype=float32)>
```

Embedding Size

The embedding dimension is a hyperparameter that you can tune. Embeddings typically have 10 to 300 dimensions, depending on the task, the vocabulary size, and the amount of training data available.

Larger Example

- Let's create a model that can process a categorical text feature along with regular numerical features.
  - The model will learn an embedding for each category.
- Start by loading the training and validation data sets.

```
X_train_num, X_train_cat, y_train = [...] # load the training set
X_valid_num, X_valid_cat, y_valid = [...] # and the validation set
```

Larger Example (Ctd.)

- We build the model using the functional API.

```
num_input = tf.keras.layers.Input(shape=[8], name="num")
cat_input = tf.keras.layers.Input(shape=[], dtype=tf.string, name="cat")
# Embed the categorical input using lookup_and_embed layer defined
          above.
cat_embeddings = lookup_and_embed(cat_input)
# Concatenate the numerical and categorical inputs.
encoded_inputs = tf.keras.layers.concatenate([num_input,
          cat_embeddings])
# Add a dense output layer.
outputs = tf.keras.layers.Dense(1)(encoded_inputs)
# Create the model by specifying the inputs and outputs.
model = tf.keras.models.Model(
  inputs=[num_input, cat_input], outputs=[outputs])
```

- This model can now be compiled and trained as usual.

Larger Example (Ctd.)

- Let's compile and train the model.

```
model.compile(loss="mse", optimizer="sgd")
history = model.fit((X_train_num, X_train_cat), y_train, epochs=5,
                    validation_data=((X_valid_num, X_valid_cat),
          y_valid))
```

13.3.7 Text Preprocessing

The `TextVectorization` Layer

- Keras provides the `TextVectorization` layer for basic text preprocessing.
- You must either pass a vocabulary upon creation, or use the `adapt()` method to let it learn the vocabulary from some training data.

Example of `TextVectorization` layer

```
train_data = ["To be", "!(to be)", "That's the question", "Be, be, be."]
text_vec_layer = tf.keras.layers.TextVectorization()
text_vec_layer.adapt(train_data)
text_vec_layer(["Be good!", "Question: be or be?"])
```

gives

```
<tf.Tensor: shape=(2, 4), dtype=int64, numpy=
array([[2, 1, 0, 0],
       [6, 2, 1, 2]])>
```

Example (Ctd.)

- To construct the vocabulary, the sentences are converted to lowercase and punctuation is removed by default.
- Next, the sentences are split on whitespace.
- The resulting words are sorted by descending frequency.
  - The most common word gets encoded as 2.
  - 1 is reserved for out-of-vocabulary words.
  - 0 is reserved for padding.

TF-IDF Encoding

- The `TextVectorization` layer can also perform TF-IDF encoding.
- This is an alternative to simply counting the words.
  - Counting words is not ideal: words like “to” and “the” are so frequent that they don’t matter at all (for the meaning of the sentence or document).
  - Rarer words, like “basketball” are much more informative.

TF-IDF Encoding (Ctd.)

- **TF-IDF** stands for **term frequency-inverse document frequency**.
  - TF-IDF encoding gives a score to each word in each sentence or document.
  - The more often it appears in the sentence, the higher the score; the more often it appears in the corpus, the lower the score.

TF-IDF Example

```
train_data = ["To be", "!(to be)", "That's the question", "Be, be, be."]
text_vec_layer = tf.keras.layers.TextVectorization(output_mode="tf_idf")
text_vec_layer.adapt(train_data)
text_vec_layer(["Be good!", "Question: be or be?"])
```

gives

```
<tf.Tensor: shape=(2, 6), dtype=float32, numpy=
array([[0.96725637, 0.6931472 , 0.        , 0.        , 0.        , 0.
],
       [0.96725637, 1.3862944 , 0.        , 0.        , 0.
,1.0986123 ]],
       dtype=float32)>
```

TF-IDF Computation

In the example above,

- we trained the `TextVectorization` layer on 4 sentences (or documents), so $d = 4$.
- The weight for each word is computed as follows:

$$\text{idf} = \log(1 + d/(f + 1))$$

where d is the number of documents and f is the number of documents that contain the word.

- E.g. for the word “be” this weight is

$$\log(1 + 4/(3 + 1)) = 0.693$$

TF-IDF Computation (Ctd.)

- The word “be” appears twice in the sentence “Question: be or be?”, so its term frequency is 2.
- The word “be” hence gets encoded as $2 \times 0.693 \approx 1.386$.
- The word “question” only appears once in the sentence, but its weight is $\log(1 + 4/(1 + 1)) \approx 1.099$.
- The vocabulary (including the “[UNK]” word) contains 6 words.
  - This is way the result has 6 columns.
- For unknown words, the average idf-weight is used.

13.3.8 Using Pretrained Language Model Components

TensorFlow Hub

- The TensorFlow Hub library makes it easy to reuse pretrained components, called *modules*.
- Browse the **TensorFlow Hub** website to find components that you can use in your projects.
  - Find the component you need and copy the code example.
  - The module will be automatically downloaded and bundled into a Keras layer.
- Modules typically include both preprocessing code and pretrained weights.
  - Generally, no extra training is required.

TensorFlow Hub Example

```
import tensorflow_hub as hub

hub_layer = hub.KerasLayer("https://tfhub.dev/google/nnlm-en-dim50/2")
sentence_embeddings = hub_layer(tf.constant(["To be", "Not to be"]))
sentence_embeddings.numpy().round(2)
```

gives

```
array([[-0.25, 0.28, 0.01, 0.1 , [...] , 0.05, 0.31],
       [-0.2 , 0.2 , -0.08, 0.02, [...] , -0.04, 0.15]], dtype=float32)
```

TensorFlow Hub Example (Ctd.)

- The module downloaded from TensorFlow Hub is a pretrained sentence encoder.
- Each sentence is embedded as a vector of size 50.
  - This vector is the mean of the word embeddings in the sentence.
- The embedding was pretrained on a large corpus of English text.

13.3.9 Image Preprocessing

Image Preprocessing Layers

The Keras preprocessing API includes three image preprocessing layers:

- `tf.keras.layers.Resizing` resizes an image to the desired size.
- `tf.keras.layers.Rescaling` rescales the pixel values.
- `tf.keras.layers.CenterCrop` crops the image, keeping only the central part.

13.4 The TensorFlow Datasets Projects

TensorFlow Datasets

- The **TensorFlow Datasets** (TFDS) project makes it very easy to load common datasets.
  - Small ones likes (Fashion) MNIST, but also huge ones like ImageNet.
- TFDS is *not* bundled with TensorFlow.
  - You must install it separately.

MNIST Example with TFDS

```
import tensorflow_datasets as tfds # Standard import for tfds

datasets = tfds.load(name="mnist") # datasets is a dict
mnist_train, mnist_test = datasets["train"], datasets["test"]
```

Note:

- `datasets` is a Python dictionary with keys `"train"` and `"test"`.
- `datasets["train"]` is a `tf.data.Dataset` object.

MNIST Example with TFDS (Ctd.)

```
# Python code not in book
for item in mnist_train.take(1):
  print(f"{type(item)}, {item.keys()}")
```

gives

```
<class 'dict'>, dict_keys(['image', 'label'])
```

This shows that each item in the dataset is a dictionary with two keys: `"image"` and `"label"`.

MNIST Example with TFDS (Ctd.)

However, Keras expects features and targets as tuples. You can fix this by calling the `map()` method on the dataset:

```
# Also shuffle and batch the dataset
mnist_train = mnist_train.shuffle(10_000, seed=42).batch(32)
# Convert to tuples
mnist_train = mnist_train.map(
  lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

Note: setting `as_supervised=True` when loading the dataset would have done the same thing!

Splitting the Data

- The `load()` function lets you specify the splits you want to load.
  - E.g., `split=["train[:90%]", "train[90%:]", "test"]` will load the first 90% of the training set as the training set, the remaining 10% as the validation set, and the test set as is.

```
train_set, valid_set, test_set = tfds.load(
    name="mnist",
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True # Note use of as_supervised=True
)
# Use the sets here .....
```