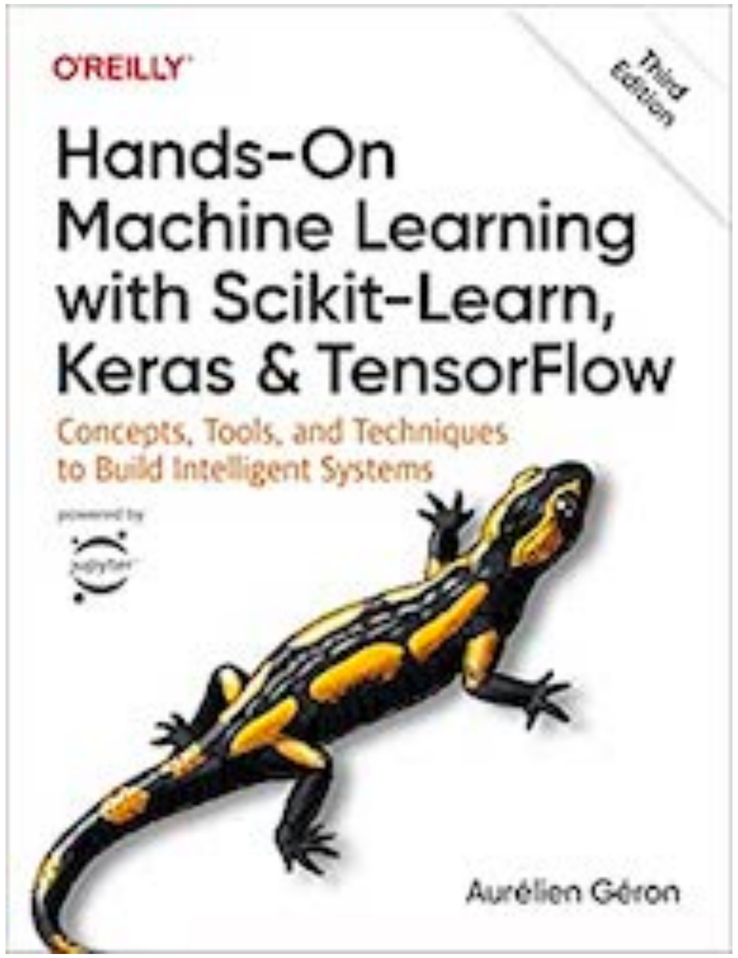
Machine Learning Security

13 Loading and Preprocessing Data with Tensorflow



Made Dec 8, 2023

Topics

- The tf.data API
- The TFRecord Format
- Keras Preprocessing Layers
- The TensorFlow Datasets Project

The tf.data API

Pandas and Scikit-Learn

- We used them in project ML 104 to import and preprocess the California housing data
- We used Pandas to read a CSV file of data

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_housing_data():
    tarball_path = Path("datasets/housing.tgz")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "https://github.com/ageron/data/raw/main/housing.tgz"
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing = load_housing_data()
housing.info()
```

Scikit-Learn

- We used Scikit-Learn to split the data into training and test sets

```
from sklearn.model_selection import train_test_split

strat_train_set, strat_test_set = train_test_split(
    housing, test_size=0.2, stratify=housing["income_cat"], random_state=42)
print("Training set:")
print(strat_train_set["income_cat"].value_counts() / len(strat_train_set))
print()
print("Test set:")
print(strat_test_set["income_cat"].value_counts() / len(strat_test_set))
```

tf.data

- A more powerful library for loading and preprocessing datasets
- Can handle large datasets efficiently
  - Can read from multiple files in parallel
  - With multithreading, queuing, shuffling, batching samples, and more
- Can load and process the next batch of data across multiple CPU cores
  - While the GPUs or TPUs are busy training the current batch of data
- Can handle datasets that don't fit in memory

TFRecord

- A flexible and efficient binary format
- Supports records of varying sizes
- tf.data API supports reading from SQL databases
  - And has many extensions
  - Including reading from Google's BigQuery service

tf.data.Dataset

- A sequence of data items
- Here, just numbers 0-9

```
import tensorflow as tf

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

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

```
import tensorflow as tf

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

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

```
<class 'tensorflow.python.data.ops.from_tensor_slices_op._TensorSliceDataset'>

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

Tuple/Dictionary Structure

```
X_nested = {"a": ([1, 2, 3], [4, 5, 6]), "b": [7, 8, 9]}
dataset = tf.data.Dataset.from_tensor_slices(X_nested)
for item in dataset:
    print(item)
```

```
>>> X_nested = {"a": ([1, 2, 3], [4, 5, 6]), "b": [7, 8, 9]}
>>> dataset = tf.data.Dataset.from_tensor_slices(X_nested)
>>> for item in dataset:
...     print(item)
...
{'a': (<tf.Tensor: [...]=1>, <tf.Tensor: [...]=4>), 'b': <tf.Tensor: [...]=7>}
{'a': (<tf.Tensor: [...]=2>, <tf.Tensor: [...]=5>), 'b': <tf.Tensor: [...]=8>}
{'a': (<tf.Tensor: [...]=3>, <tf.Tensor: [...]=6>), 'b': <tf.Tensor: [...]=9>}
```

Repeat and Batch

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

- Repeat() repeats the input data
- Batch() picks out a batch of instances
- 3x10 makes 4 batches of 7 and 2 left over

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

```
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)
```

Chaining Dataset Transformations

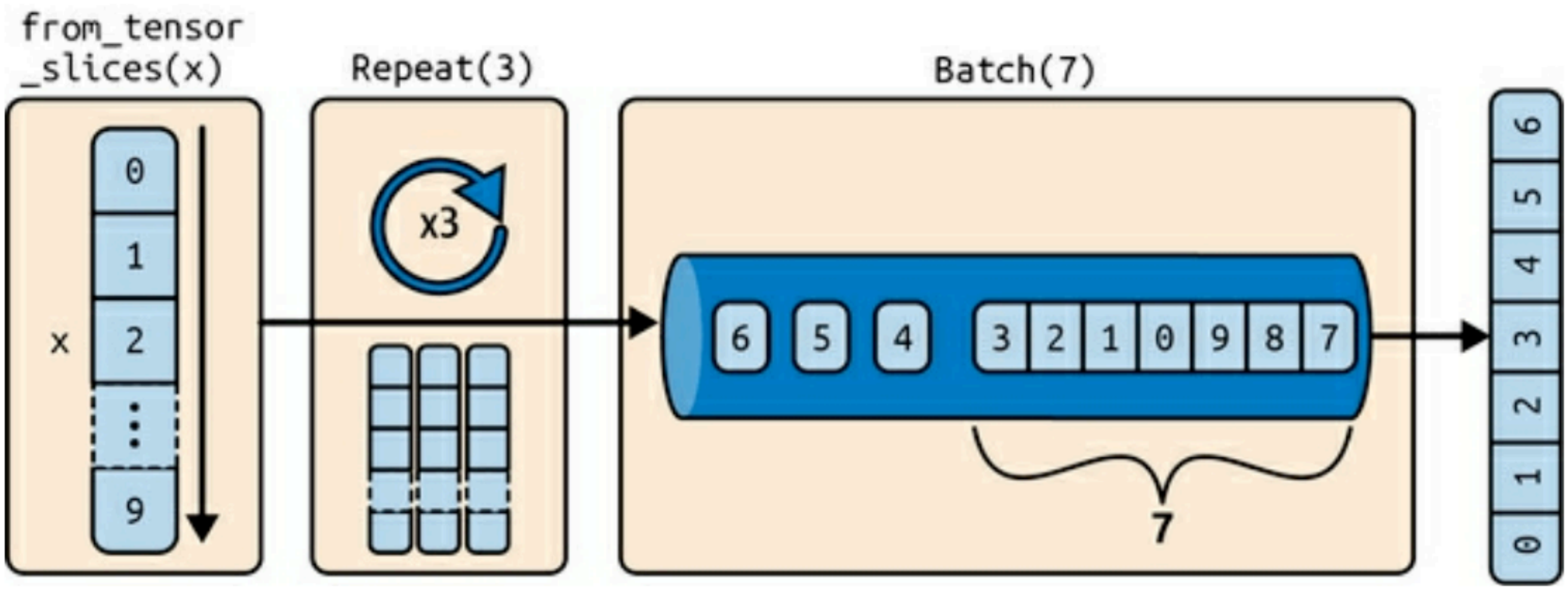


Figure 13-1. Chaining dataset transformations

Map

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

- Do computations on the data
- For preprocessing
- This example doubles the data values

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

```
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)
```

Filter

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

- Keeps only batches that satisfy the condition
  - Total > 50


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


```
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

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

- Pulls a few items from a dataset


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


```
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)
```

Shuffle

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

- Randomly mixes the instances
- Specify the seed to always see the same random choices


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


```
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)
```

Interleave

- Reads multiple files and interleaves the instances to form a new dataset

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

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

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

Preprocessing the Data

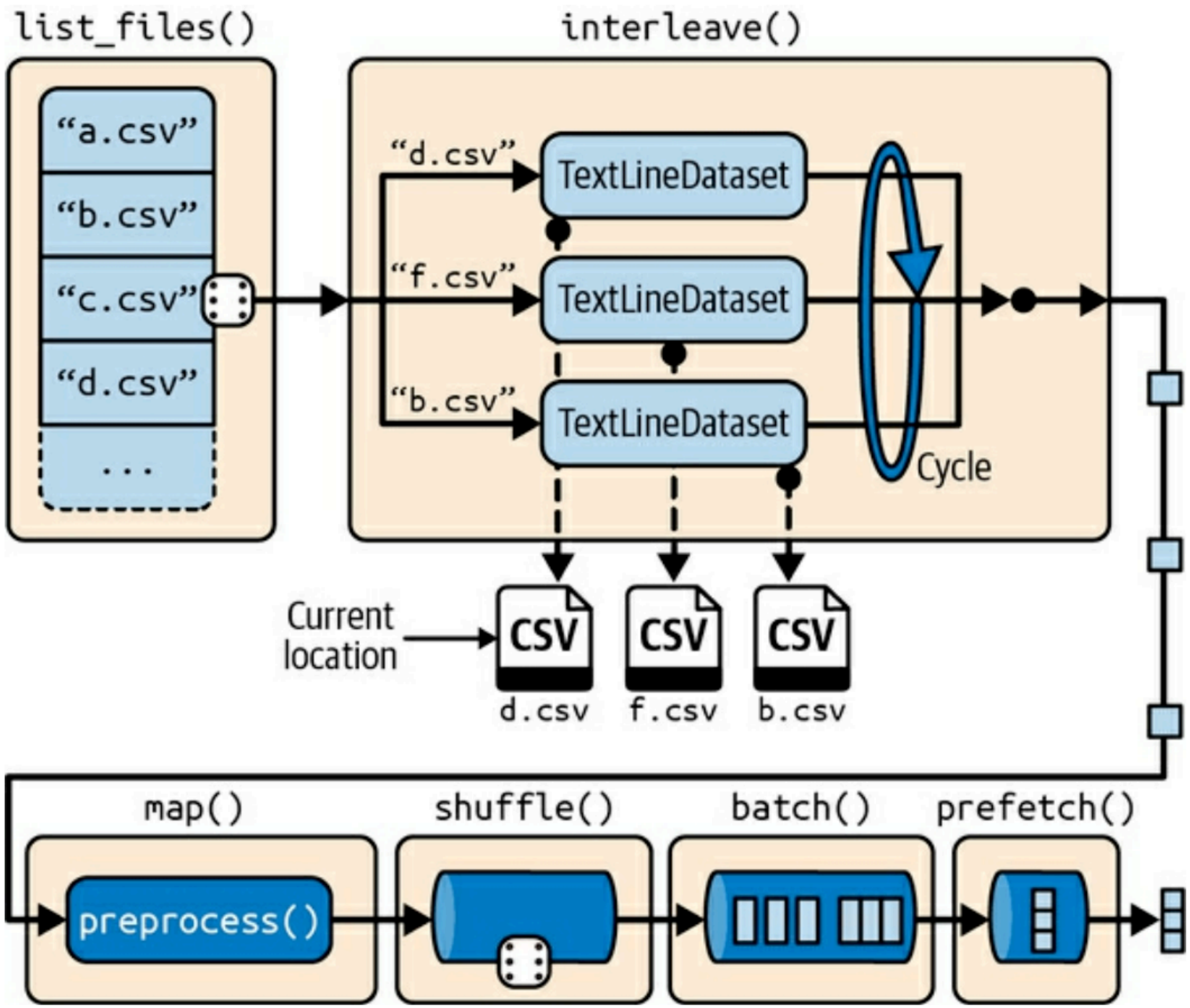
- Normalizes data using the mean and std deviation

```
X_mean, X_std = [...]  # mean and scale of each feature in the training set
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

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

Putting Everything Together

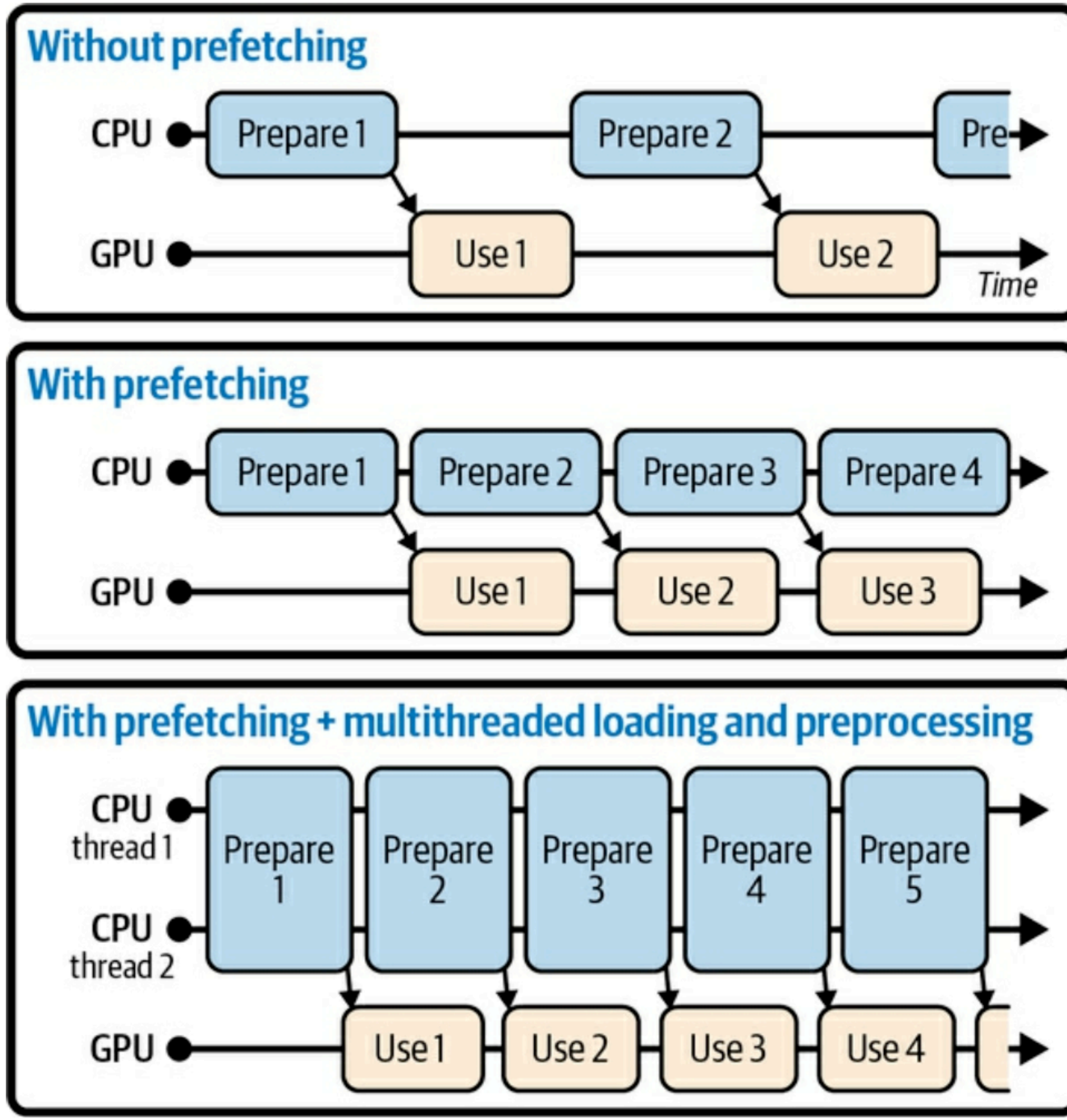
```
def csv_reader_dataset(filepaths, n_readers=5, n_read_threads=None,
                       n_parse_threads=5, shuffle_buffer_size=10_000, seed=42,
                       batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths, seed=seed)
    dataset = dataset.interleave(
        lambda 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)
```

Loading and Preprocessing Data from Multiple CSV Files



Prefetching

- Loads the next batch while the current batch is training
- Keeps CPU and GPU (or TPU) running at the same time
- Improves performance



Using the Dataset with Keras

- Read the CSV files in

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

- Train a model

```
model = tf.keras.Sequential([...])
model.compile(loss="mse", optimizer="sgd")
model.fit(train_set, validation_data=valid_set, epochs=5)
```

- Evaluate and predict

```
test_mse = model.evaluate(test_set)
new_set = test_set.take(3)  # pretend we have 3 new samples
y_pred = model.predict(new_set)  # or you could just pass a NumPy array
```

Custom Training Function

- This function can speed up training
- If you compile() it with steps_per_execution > 1, such as 50
- That will process multiple batches at once

```
@tf.function
def train_one_epoch(model, optimizer, loss_fn, train_set):
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
loss_fn = tf.keras.losses.mean_squared_error
for epoch in range(n_epochs):
    print("\rEpoch {}/{}".format(epoch + 1, n_epochs), end="")
    train_one_epoch(model, optimizer, loss_fn, train_set)
```

The TFRecord Format

The TFRecord Format

- TensorFlow’s preferred format for storing large amounts of data
    - And reading it efficiently
- A simple binary format
- A sequence of binary records of varying sizes
- Each record contains
    - Length, CRC checksum
    - Data, CRC checksum

Creating a TFRecord File

- Creating a file

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

- Reading a file
    - Can read many files in parallel

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

- Output

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

Compress TFRecord Files

- Helpful if your file is being loaded over a network
- Use the **options** argument
- Creating the file

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
    f.write(b"Compress, compress, compress!")
```

- Reading the file

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

Protocol Buffers (protobufs)

- TFRecord files usually contain serialized protocol buffers
- Developed by Google in 2001
- Protobuf definition
  - In a **.proto** file
- 1, 2, ,and 3 are field identifiers
- Compile with **protoc** compiler
- But the commonly used ones are already defined and compiled into TensorFlow

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

Using a Protobuf Class

- Creating a message

```
>>> from person_pb2 import Person  # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
>>> print(person)  # display the Person
name: "Al"
id: 123
email: "a@b.com"
```

- Read & modify a field

```
>>> person.name  # read a field
'Al'
>>> person.name = "Alice"  # modify a field
>>> person.email[0]  # repeated fields can be accessed like arrays
'a@b.com'
>>> person.email.append("c@d.com")  # add an email address
```

- Serialize a record

```
>>> serialized = person.SerializeToString()  # serialize person to a byte string
>>> serialized
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person()  # create a new Person
>>> person2.ParseFromString(serialized)  # parse the byte string (27 bytes long)
27
>>> person == person2  # now they are equal
True
```

TensorFlow Protobufs

- The main protobuf used in a TFRecord file is **Example**
- Represents one instance in a dataset
- An **Example** contains a list of named **Features**

```
message Example { Features features = 1; };
```

- A **Feature** is a list containing one of:
    - Byte strings
    - Floats, or
    - Integers

```
message Features { map<string, Feature> feature = 1; };
```

Message Protobuf

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

- packed=true encodes numerical data more efficiently

Creating an Example

- Contains a Person

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                                                          b"c@d.com"]))
        }))
```

- Serialize and write to a TFRecord file
- Five copies of the same Person

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    for _ in range(5):
        f.write(person_example.SerializeToString())
```

Loading and Parsing Examples

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

def parse(serialized_example):
    return tf.io.parse_single_example(serialized_example, feature_description)

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).map(parse)
for parsed_example in dataset:
    print(parsed_example)
```

```
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x7f829147c040>, 'id': <
tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape=(), dtype=string, numpy=b'Ali
ce'>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x7f82390756a0>, 'id': <
tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape=(), dtype=string, numpy=b'Ali
ce'>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x7f8239068a60>, 'id': <
tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape=(), dtype=string, numpy=b'Ali
ce'>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x7f829147b310>, 'id': <
tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape=(), dtype=string, numpy=b'Ali
ce'>}
{'emails': <tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x7f829155d850>, 'id': <
tf.Tensor: shape=(), dtype=int64, numpy=123>, 'name': <tf.Tensor: shape=(), dtype=string, numpy=b'Ali
ce'>}
```

Kahoot!

Ch 12a

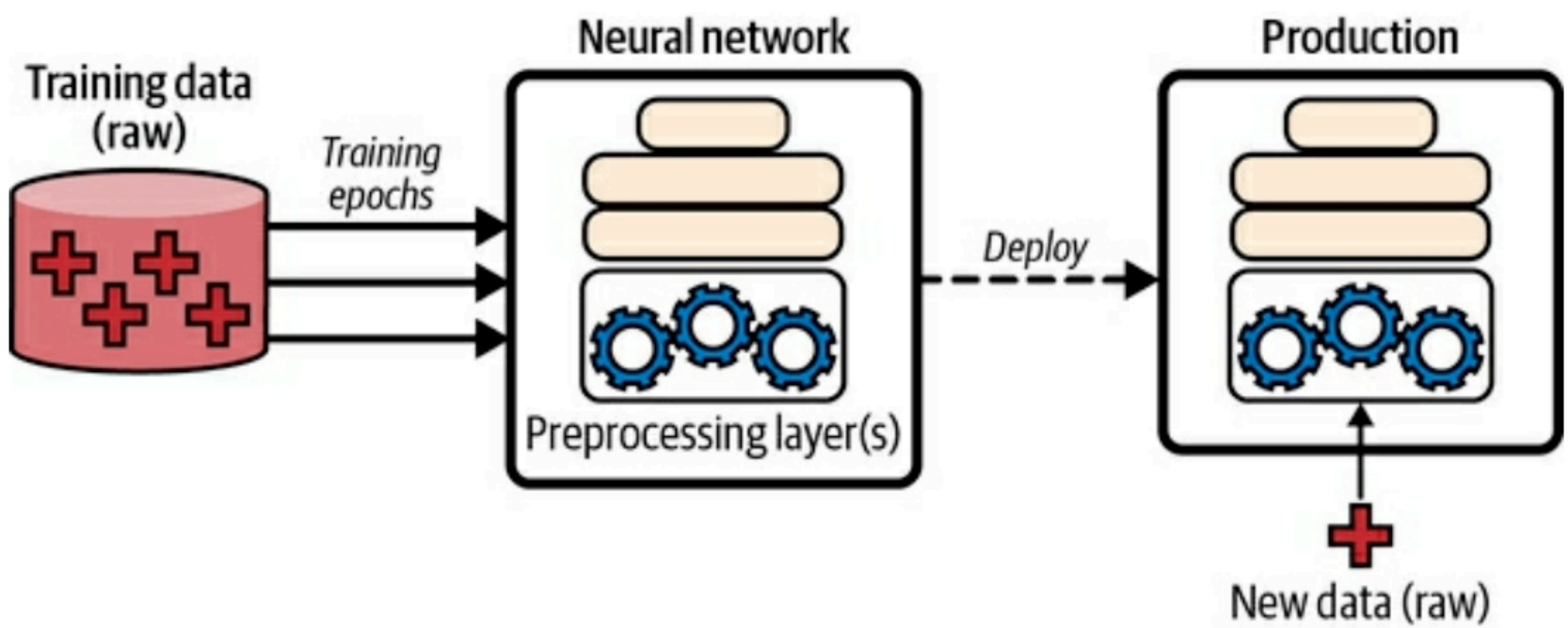
Keras Preprocessing Layers

Preparing Data

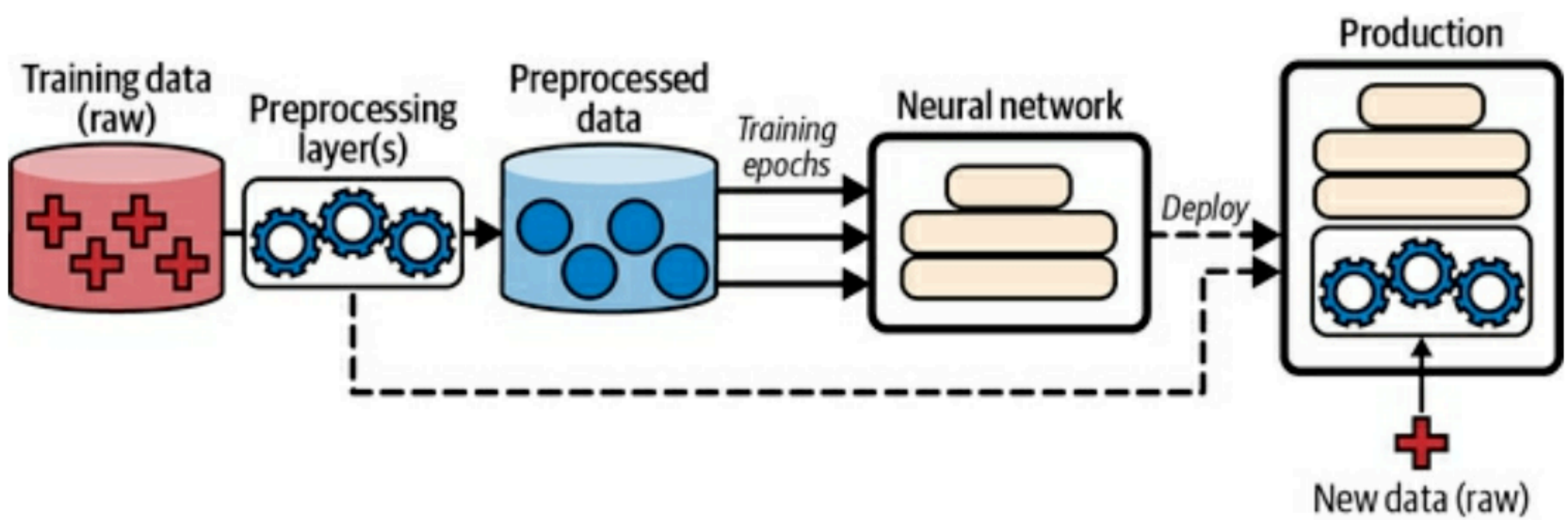
  - Normalizing numerical features
  - Encoding categorical features and text
  - Cropping and resizing images
  - etc.
- Could be done with original data files with Python
- Or on the fly while loading it with **tf.data**
- Or with preprocessing layers inside your model

Including Preprocessing Layers in a Model

- Advantage: no need to manually preprocess production data after training



- Preprocessing the data just once before training using preprocessing layers,
- then deploying these layers inside the final model



Discretization Layer

- Transforms a numerical feature into a categorical feature
  - By mapping value ranges (called bins) to categories
- Useful for features with multimodal distributions
  - Or with highly nonlinear relationship to the target

Discretization Example

- This code maps **age** into three bins
  - Less than 18
  - 18-50
  - 50 or more

```
>>> age = tf.constant([[10.], [93.], [57.], [18.], [37.], [5.]])
>>> discretize_layer = tf.keras.layers.Discretization(bin_boundaries=[18., 50.])
>>> age_categories = discretize_layer(age)
>>> age_categories
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=array([[0],[2],[2],[1],[1],[0]])>
```

CategoryEncoding Layer

- Category identifiers should not be passed directly to a neural network
  - Because their values cannot be meaningfully compared
- They should be encoded, using one-hot encoding or some other such system

```
>>> onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3)
>>> onehot_layer(age_categories)
<tf.Tensor: shape=(6, 3), dtype=float32, numpy=
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]], dtype=float32)>
```

Multi-Hot Encoding

- More than one categorical feature
  - Both using the same categories

```
>>> two_age_categories = np.array([[1, 0], [2, 2], [2, 0]])
>>> onehot_layer(two_age_categories)
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1., 1., 0.],
       [0., 0., 1.],
       [1., 0., 1.]], dtype=float32)>
```

StringLookup Layer

- By default, encodes strings as integers
- Unknown categories are mapped to 0

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

- Can also do one-hot encoding

```
>>> str_lookup_layer = tf.keras.layers.StringLookup(output_mode="one_hot")
>>> str_lookup_layer.adapt(cities)
>>> str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
<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)>
```

Hashing Layer

- Computes a hash, modulo the number of bins
  - Pseudorandom but stable across runs and platforms
  - The same category will always be mapped to the same integer
  - (As long as the number of bins is unchanged)

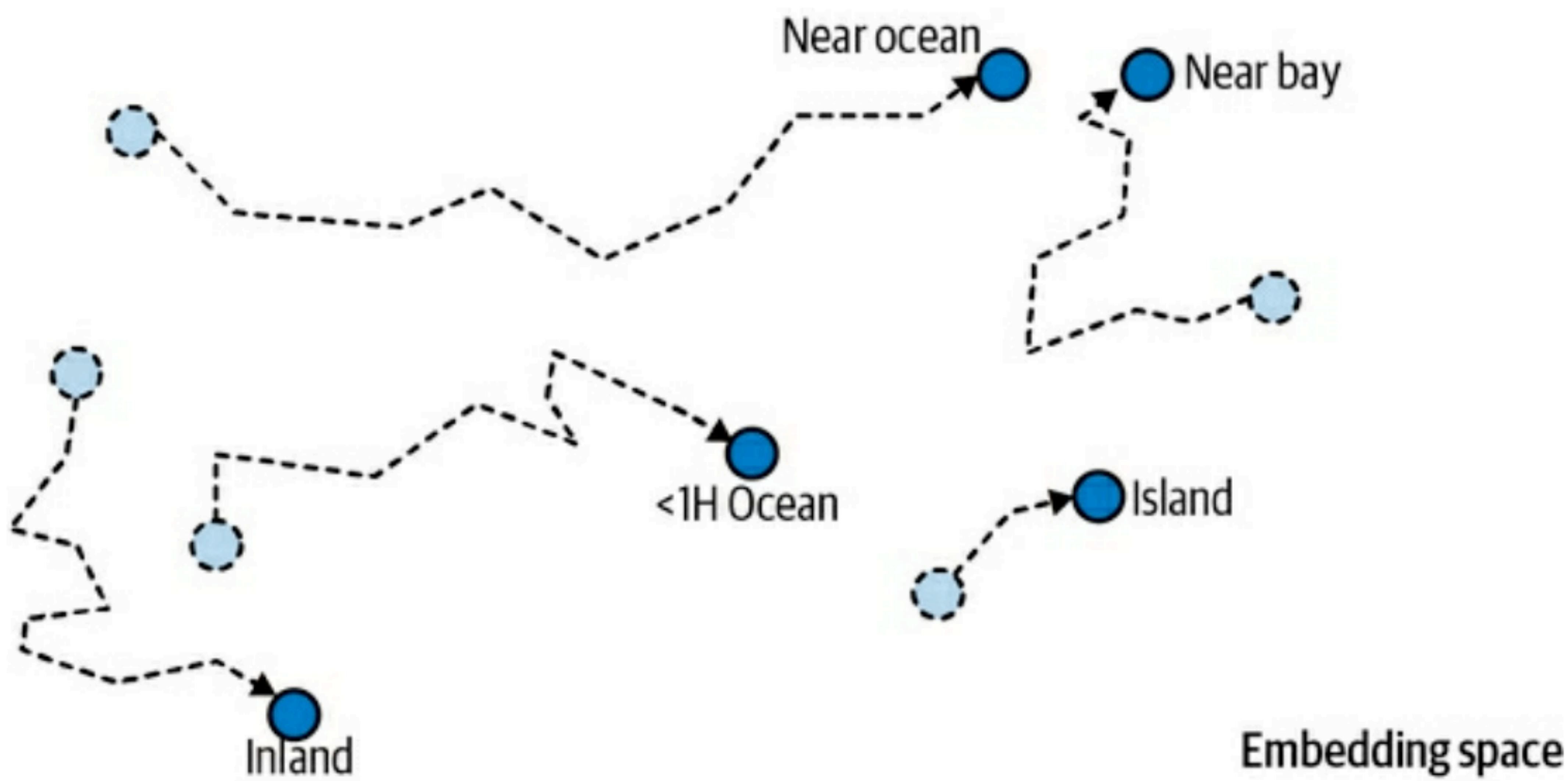
```
>>> hashing_layer = tf.keras.layers.Hashing(num_bins=10)
>>> hashing_layer([["Paris"], ["Tokyo"], ["Auckland"], ["Montreal"]])
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[0], [1], [9], [1]])>
```

  - There are hash collisions (e.g. 'Tokyo' and 'Montreal')
  - Usually a **StringLookup** layer is better

Encoding Categorical Features Using Embeddings

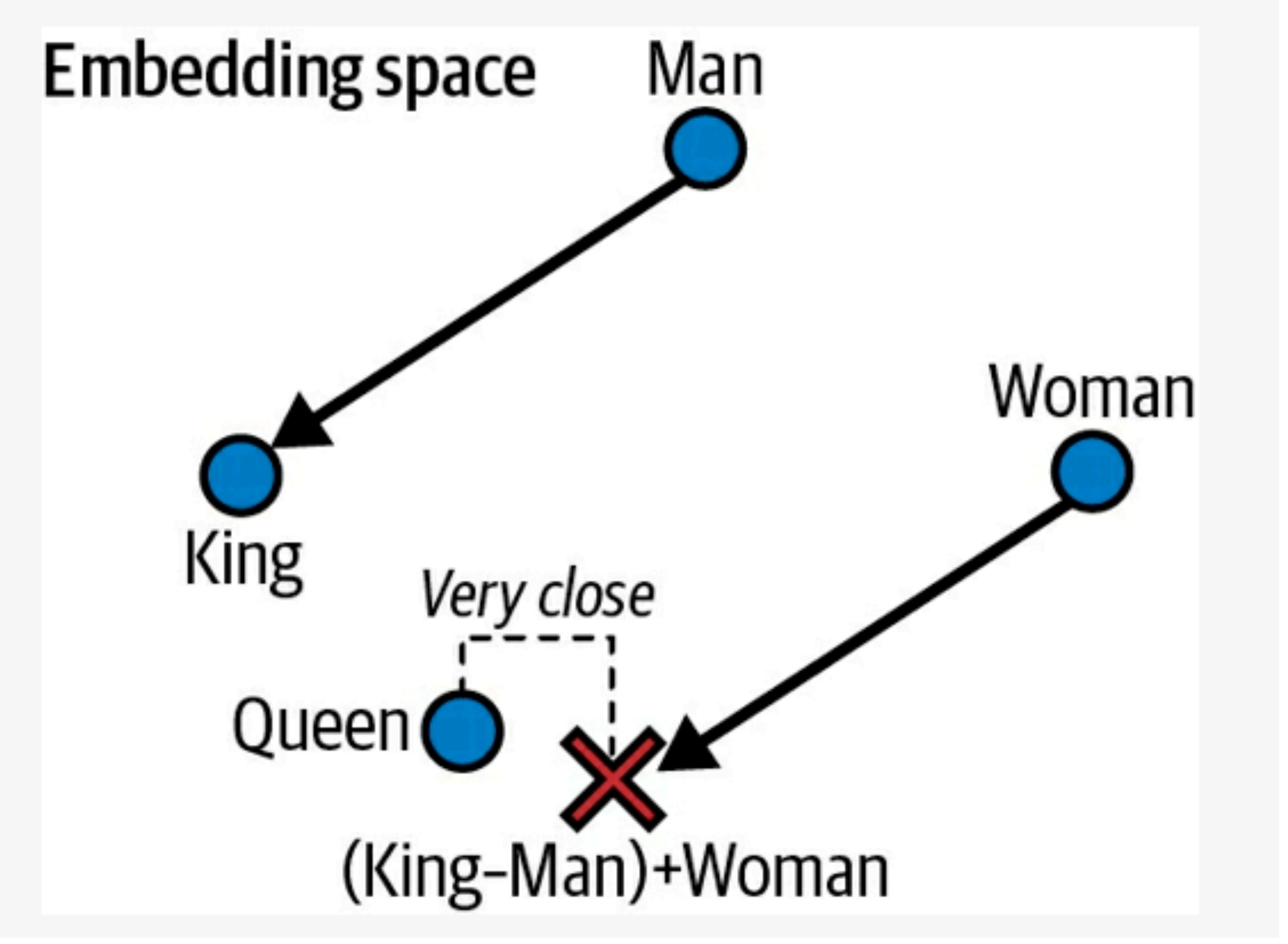
- Embedding: a dense representation of higher-dimensional data
- Consider a vocabulary of 50,000 words
- One-hot encoding would produce a 50,000 dimensional sparse vector
- An embedding would be a smaller dense vector
    - Such as 100 dimensions
- Embeddings are initialized randomly
    - Trained by gradient descent

Embeddings Will Gradually Improve During Training



Word Embeddings

- Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts



Embedding a Numerical Field

- Input of 2 is always embedded to the same value
- These values are random (before training)

```
>>> tf.random.set_seed(42)
>>> embedding_layer = tf.keras.layers.Embedding(input_dim=5, output_dim=2)
>>> embedding_layer(np.array([2, 4, 2]))
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.04663396,  0.01846724],
       [-0.02736737, -0.02768031],
       [-0.04663396,  0.01846724]], dtype=float32)>
```

Embedding a Categorical Text Attribute

- Chain two layers
    - **StringLookup** and **Embedding**

```
>>> tf.random.set_seed(42)
>>> 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([
...     str_lookup_layer,
...     tf.keras.layers.Embedding(input_dim=str_lookup_layer.vocabulary_size(),
...                               output_dim=2)
... ])
...
>>> lookup_and_embed(np.array([["<1H OCEAN"], ["ISLAND"], ["<1H OCEAN"]]))
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.01896119,  0.02223358],
       [ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358]], dtype=float32)>
```

Text Preprocessing

- Three ways
  - **TextVectorization** layer
  - TensorFlow Text library
  - Pretrained language model components

TextVectorization

- Splits sentences on whitespace
- Counts the words, sorts on descending frequency
- Numbers the words with Word IDs
- Unknown words are encoded as 1s
- Pads the first sentence below with zeros

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

Encoding WordIDs

- TextVectorization layer output can be **count** or **multi_hot**
- But usually the best option is **tf_idf**
  - ***Term frequency*** × ***inverse-document-frequency***
- Words that are rare in the document are upweighted

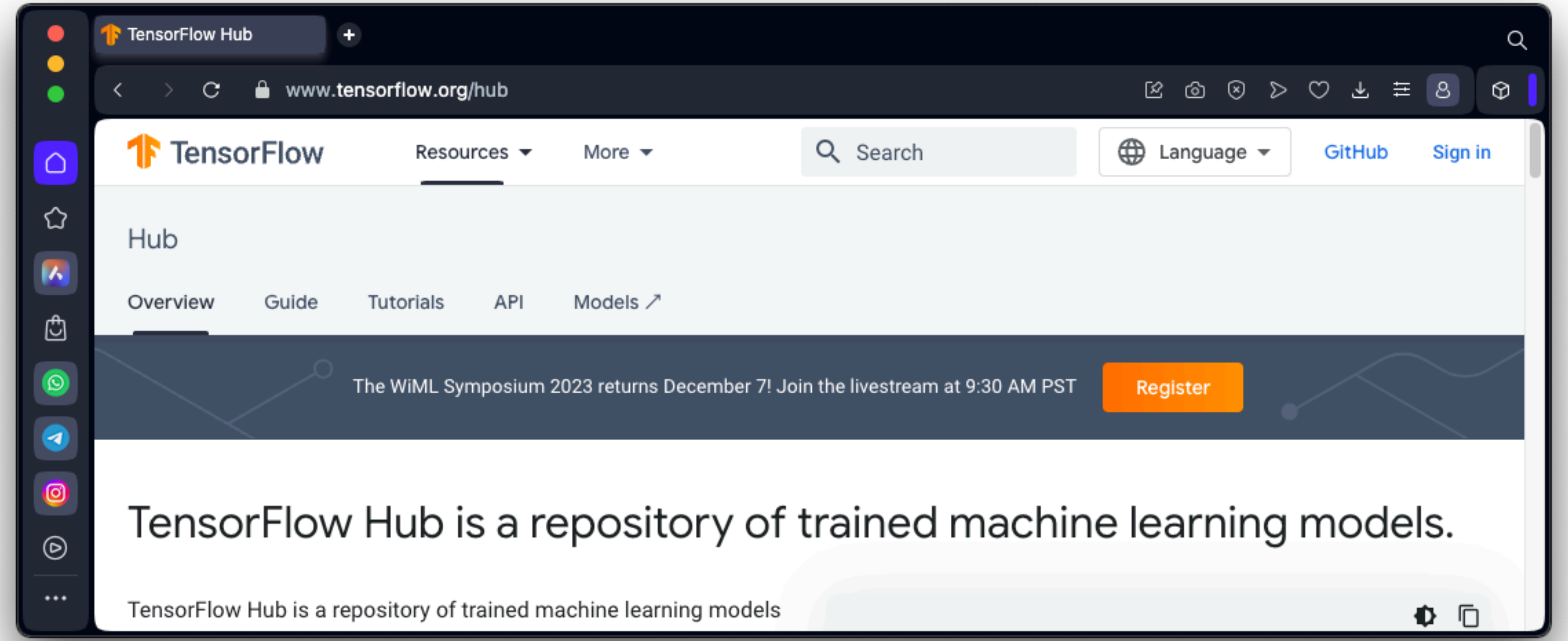
```
>>> text_vec_layer = tf.keras.layers.TextVectorization(output_mode="tf_idf")
>>> text_vec_layer.adapt(train_data)
>>> text_vec_layer(["Be good!", "Question: be or be?"])
<tf.Tensor: shape=(2, 6), dtype=float32, numpy=
array([[0.96725637, 0.6931472 , 0. , 0. , 0. , 0.        ],
       [0.96725637, 1.3862944 , 0. , 0. , 0. , 1.0986123 ]], dtype=float32)>
```

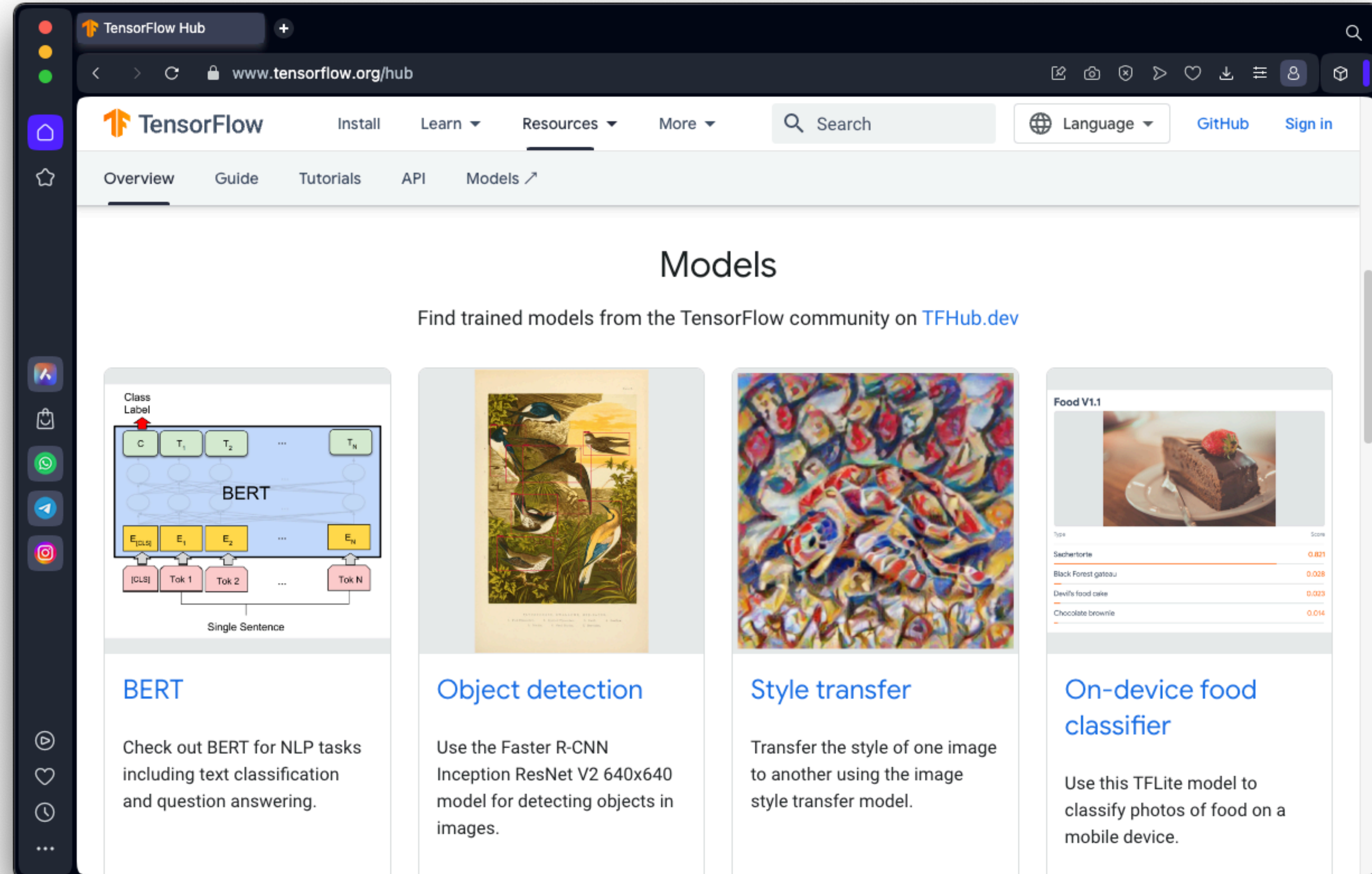
Using Pretrained Language Model Components

- Pretrained model components
    - For text, image, audio, and more
- The **nnlm-en-dim50** module encodes sentences as 50-dimensional vectors

```
>>> 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)
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




TensorFlow Hub
www.tensorflow.org/hub
TensorFlow
Install
Learn
Resources
More
Search
Language
GitHub
Sign in
Overview
Guide
Tutorials
API
Models ↗
Models
Find trained models from the TensorFlow community on TFHub.dev
Class Label
BERT
Single Sentence
Food V1.1
Sachertorte 0.821
Black Forest gateau 0.028
Devil's food cake 0.023
Chocolate brownie 0.014
BERT
Check out BERT for NLP tasks including text classification and question answering.
Object detection
Use the Faster R-CNN Inception ResNet V2 640x640 model for detecting objects in images.
Style transfer
Transfer the style of one image to another using the image style transfer model.
On-device food classifier
Use this TFLite model to classify photos of food on a mobile device.

HuggingFace Transformers

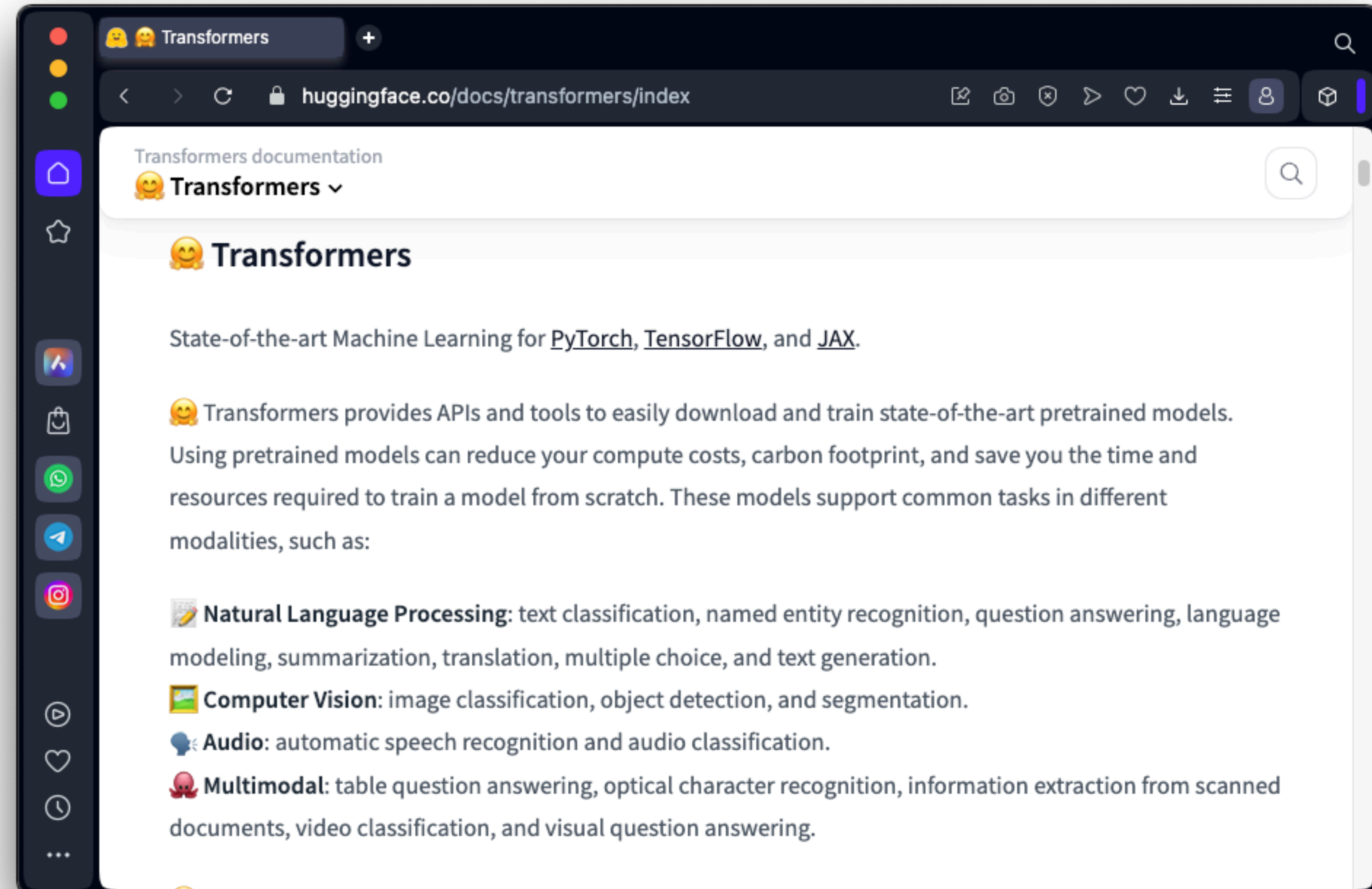


Image Preprocessing Layers

- **tf.keras.layers.Resizing**
  - Resizes the input images to the desired size
- **tf.keras.layers.Rescaling**
  - Rescales the pixel values to a range, such as -1 to 1
- **tf.keras.layers.CenterCrop**
  - Crops the image, keeping only a center patch of the desired height and width

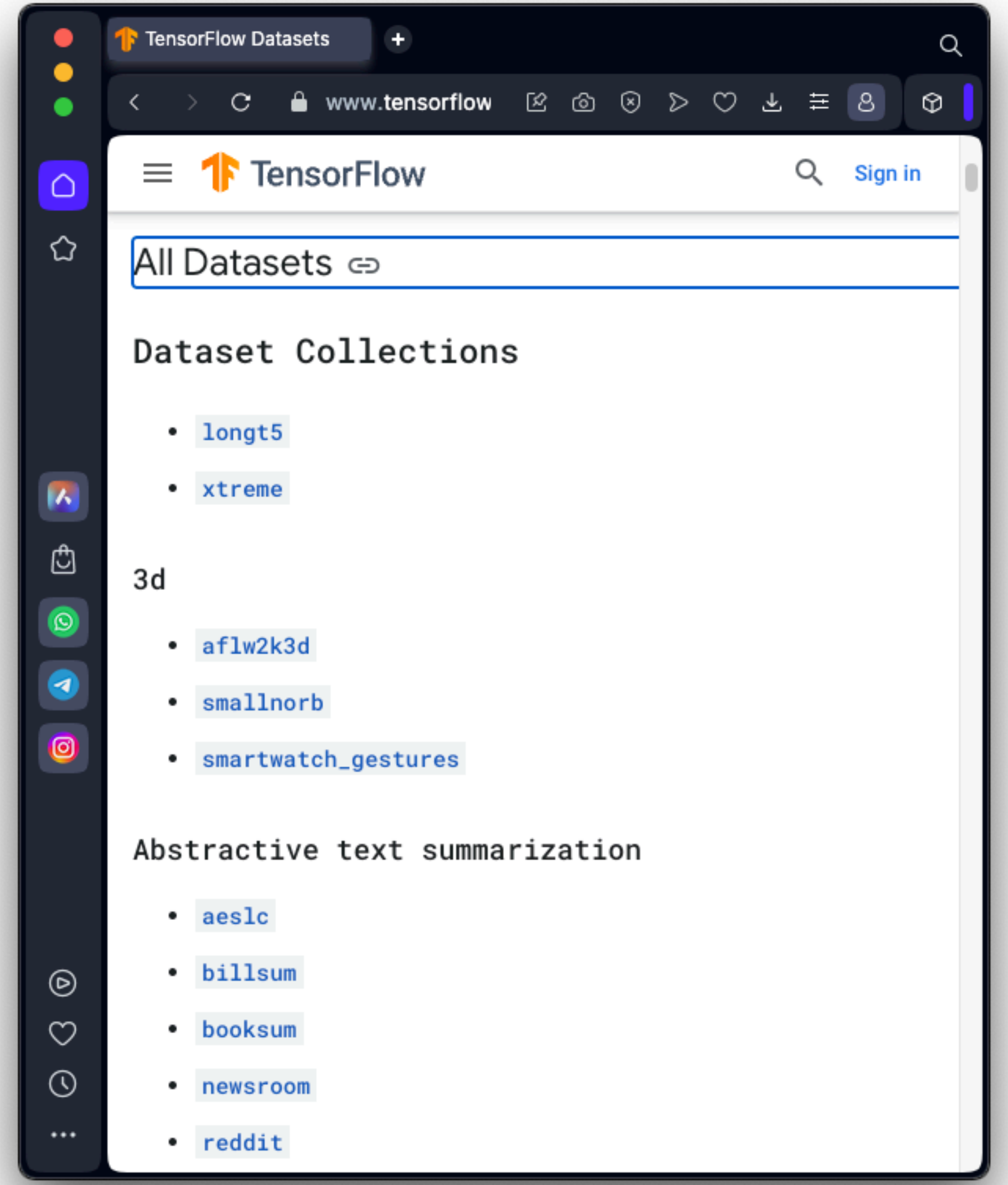
Data Augmentation Layers

- RandomCrop
- RandomFlip
- RandomTranslation
- RandomRotation
- RandomZoom
- RandomHeight
- RandomWidth
- RandomContrast

The TensorFlow Datasets Project

TensorFlow Datasets

- Many common datasets
- Images, text, video, audio, and much more



Kahoot!

Ch 13b