Configuration

Configure a dataset with shuffling, repetitions, and batch size.

Chapter Goals:

- Learn how to configure a dataset
- Implement a function that shuffles, repeats, and batches an input dataset

A. Shuffling

When using a dataset to train a machine learning model, there are certain things we need to do to properly configure the dataset. When we first create a dataset from NumPy arrays or files, the observations may be ordered in a particular way. For example, many data files will sort the data observations by some particular feature, like a person's name or year.

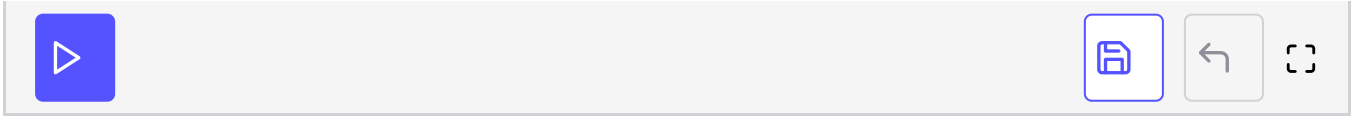
While systematic ordering of data files makes it easier for humans to look over the data, it actually hinders the training of a machine learning model. The model will learn to make predictions based on the ordering of the observations rather than the observations themselves, which is not what we want our model to do. To avoid this, we need to randomly shuffle our dataset prior to training a model, which is done with the `shuffle` function.

```
import numpy as np
import tensorflow as tf

data = np.random.uniform(-100, 100, (1000, 5))
original = tf.data.Dataset.from_tensor_slices(data)

shuffled1 = original.shuffle(100)
print(shuffled1)

shuffled2 = original.shuffle(len(data))
print(shuffled2)
```



Using the shuffle function to randomly shuffle a dataset.

In the example, `original` contains 1000 observations, with each observation consisting of 5 floating point numbers. The datasets `shuffled1` and `shuffled2` are the results of randomly shuffling `original` using different *buffer sizes*.

The buffer size (specified by the required argument of `shuffle`) essentially dictates how random our shuffling is. The larger the buffer size, the more uniformly random the shuffling will be.

If the buffer size is 1, then no shuffling occurs. If the buffer size is $\geq$ the number of total observations (the case for `shuffled2`), then the shuffling is uniformly random across the entire dataset. When the buffer size is somewhere in between (the case for `shuffled1`), shuffling will still occur but it will not be uniformly random for all the observations.

In most cases, uniform shuffling is the best option, since that's the safest bet to avoiding any systematic ordering of the initial data observations. However, for datasets that take up a lot of memory, it may be useful to initially shuffle the entire dataset once, then use a smaller buffer size for faster training.

B. Epochs

Aside from randomly shuffling, we also need to configure a dataset so that training can be done for multiple *epochs*. An epoch refers to a single training run over the entire dataset. We normally need to go through several epochs of a dataset before the model is finished training.

Without any configuration, a dataset can only be used to train a model for one epoch. If we try to train for more than one epoch, we get back an `OutOfRangeError`. To fix this, we use the `repeat` function to specify the number of epochs we can run a dataset.

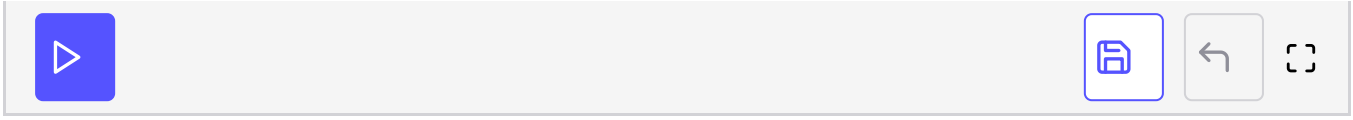
```
import numpy as np
import tensorflow as tf

data = np.random.uniform(-100, 100, (1000, 5))
original = tf.data.Dataset.from_tensor_slices(data)

repeat1 = original.repeat(1)
print(repeat1)

repeat2 = original.repeat(100)
print(repeat2)
```

```
repeat3 = original.repeat()
print(repeat3)
```



Using the repeat function to configure a dataset for multiple epochs.

The `repeat` function takes in a single (optional) argument, which specifies how many epochs we can run the dataset for.

The `repeat1` dataset is explicitly configured for one epoch, while `repeat2` is configured for 100 epochs. This means we can use `repeat1` for one full run through of `original` and `repeat2` for 100 full run throughs of `original`, before getting back an `OutOfRangeError`.

For `repeat3`, we didn't pass in any argument. This is the same as passing in `None`, which is the default value for the optional argument. It means that `repeat3` can be run indefinitely, and will never raise an `OutOfRangeError`.

C. Batching

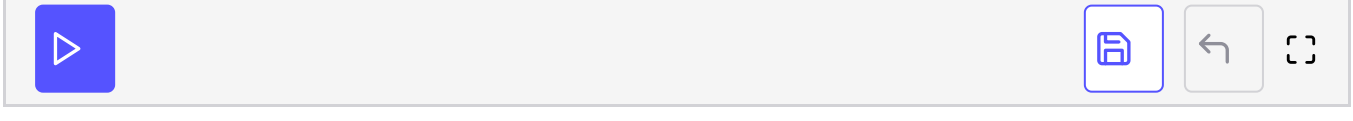
The default number of data observations we use for each dataset iteration is one. This can be incredibly slow when training or evaluating very large datasets, which can have millions of observations. Luckily, we can explicitly set the number of observations per iteration (i.e. the *batch size*) using the `batch` function.

```
import numpy as np
import tensorflow as tf

data = np.random.uniform(-100, 100, (1000, 5))
original = tf.data.Dataset.from_tensor_slices(data)

batch1 = original.batch(1)
print(batch1)

batch2 = original.batch(100)
print(batch2)
```



Using the batch function to set explicit batch sizes for each iteration of observations. The ? in the shape of the batched datasets is just a placeholder for the batch size.

The `batch` function takes in a required argument, representing the batch size per iteration. In the example, `batch1` will use one data observation per iteration while `batch2` will use 100. So at each iteration, the shape of the data for `batch1` will be `(1, 5)`, while the shape of the data for `batch2` will be `(100, 5)`.

In general, it is a good idea to explicitly set the batch size using `batch`, even if the batch size is just 1. Without setting the batch size, the shape of the data at each iteration is 1-D (in the example, it is `(5,)`). This can potentially cause issues, since a single observation might be interpreted as a batch (e.g. a shape of `(5,)` could be viewed as a batch of 5 observations).

The optimal batch size will differ depending on the size of the dataset and the problem itself. The batch size can also change at different points of the training. Initially, you may want a larger batch size so the model can train faster in its early stages, then gradually decrease the batch size as the model gets closer to converging so as to not overshoot the convergence point.