Tox21 Dataset

For our modeling case study, we will use a chemical dataset. Toxicologists are very interested in the task of using machine learning to predict whether a given compound will be toxic or not. This task is extremely complicated, since today's science has only a limited understanding of the metabolic processes that happen in a human body. However, biologists and chemists have worked out a limited set of experiments that provide indications of toxicity. If a compound is a "hit" in one of these experiments, it will likely be toxic for a human to ingest. However, these experiments are often costly to run, so data scientists aim to build machine learning models that can predict the outcomes of these experiments on new molecules.

One of the most important toxicological dataset collections is called Tox21. It was released by the NIH and EPA as part of a data science initiative and was used as the dataset in a model building challenge. The winner of this challenge used multitask fully connected networks (a variant of fully connected networks where each network predicts multiple quantities for each datapoint). We will analyze one of the datasets from the Tox21 collection. This dataset consists of a set of 10,000 molecules tested for interaction with the androgen receptor. The data science challenge is to predict whether new molecules will interact with the androgen receptor.

Processing this dataset can be tricky, so we will make use of the MoleculeNet dataset collection curated as part of DeepChem. Each molecule in Tox21 is processed into a bit-vector of length 1024 by DeepChem. Loading the dataset is then a few simple calls into DeepChem (Example 4-1).

Example 4-1. Load the Tox21 dataset

```
import deepchem as dc

_, (train, valid, test), _ = dc.molnet.load_tox21()
train_X, train_y, train_w = train.X, train.y, train.w
valid_X, valid_y, valid_w = valid.X, valid.y, valid.w
test_X, test_y, test_w = test.X, test.y, test.w
```

Here the `X` variables hold processed feature vectors, `y` holds labels, and `w` holds example weights. The labels are binary 1/0 for compounds that interact or don't interact with the androgen receptor. Tox21 holds *imbalanced* datasets, where there are far fewer positive examples than negative examples. `w` holds recommended per-example weights that give more emphasis to positive examples (increasing the importance of rare examples is a common technique for handling imbalanced datasets). We won't use these weights during training for simplicity. All of these variables are NumPy arrays.

Tox21 has more datasets than we will analyze here, so we need to remove the labels associated with these extra datasets (Example 4-2).

Example 4-2. Remove extra datasets from Tox21

```
# Remove extra tasks
train_y = train_y[:, 0]
valid_y = valid_y[:, 0]
test_y = test_y[:, 0]
train_w = train_w[:, 0]
valid_w = valid_w[:, 0]
test_w = test_w[:, 0]
```

Accepting Minibatches of Placeholders

In the previous chapters, we created placeholders that accepted arguments of fixed size. When dealing with minibatched data, it is often convenient to be able to feed batches of variable size. Suppose that a dataset has 947 elements. Then with a minibatch size of 50, the last batch will have 47 elements. This would cause the code in Chapter 3 to crash. Luckily, TensorFlow has a simple fix to the situation: using `None` as a dimensional argument to a placeholder allows the placeholder to accept tensors with arbitrary size in that dimension (Example 4-3).

Example 4-3. Defining placeholders that accept minibatches of different sizes

```
d = 1024
with tf.name_scope("placeholders"):
  x = tf.placeholder(tf.float32, (None, d))
  y = tf.placeholder(tf.float32, (None,))
```

Note `d` is 1024, the dimensionality of our feature vectors.

Implementing a Hidden Layer

The code to implement a hidden layer is very similar to code we've seen in the last chapter for implementing logistic regression, as shown in Example 4-4.

Example 4-4. Defining a hidden layer

```
with tf.name_scope("hidden-layer"):
  W = tf.Variable(tf.random_normal((d, n_hidden)))
  b = tf.Variable(tf.random_normal((n_hidden,)))
  x_hidden = tf.nn.relu(tf.matmul(x, W) + b)
```

We use a `tf.name_scope` to group together introduced variables. Note that we use the matricial form of the fully connected layer. We use the form xW instead of Wx in