Advanced Model Training

In the tutorials so far we have followed a simple procedure for training models: load a dataset, create a model, call `fit()`, evaluate it, and call ourselves done. That's fine for an example, but in real machine learning projects the process is usually more complicated. In this tutorial we will look at a more realistic workflow for training a model.

Colab

This tutorial and the rest in this sequence can be done in Google colab. If you'd like to open this notebook in colab, you can use the following link.

Setup

To run DeepChem within Colab, you'll need to run the following installation commands. You can of course run this tutorial locally if you prefer. In that case, don't run these cells since they will download and install DeepChem in your local machine again.

In [ ]:
```
!pip install --pre deepchem
import deepchem
deepchem.__version__
```

Hyperparameter Optimization

Let's start by loading the HIV dataset. It classifies over 40,000 molecules based on whether they inhibit HIV replication.

In [1]:
```
import deepchem as dc

tasks, datasets, transformers = dc.molnet.load_hiv(featurizer='ECFP', split='scaffold')
train_dataset, valid_dataset, test_dataset = datasets
```

Now let's train a model on it. We will use a `MultitaskClassifier`, which is just a stack of dense layers. But that still leaves a lot of options. How many layers should there be, and how wide should each one be? What dropout rate should we use? What learning rate?

These are called hyperparameters. The standard way to select them is to try lots of values, train each model on the training set, and evaluate it on the validation set. This lets us see which ones work best.

You could do that by hand, but usually it's easier to let the computer do it for you. DeepChem provides a selection of hyperparameter optimization algorithms, which are found in the `dc.hyper` package. For this example we'll use `GridHyperparamOpt`, which is the most basic method. We just give it a list of options for each hyperparameter and it exhaustively tries all combinations of them.

The lists of options are defined by a `dict` that we provide. For each of the model's arguments, we provide a list of values to try. In this example we consider three possible sets of hidden layers: a single layer of width 500, a single layer of width 1000, or two layers each of width 1000. We also consider two dropout rates (20% and 50%) and two learning rates (0.001 and 0.0001).

In [2]:
```
params_dict = {
    'n_tasks': [len(tasks)],
    'n_features': [1024],
    'layer_sizes': [[500], [1000], [1000, 1000]],
    'dropouts': [0.2, 0.5],
    'learning_rate': [0.001, 0.0001]
}
optimizer = dc.hyper.GridHyperparamOpt(dc.models.MultitaskClassifier)
metric = dc.metrics.Metric(dc.metrics.roc_auc_score)
best_model, best_hyperparams, all_results = optimizer.hyperparam_search(
        params_dict, train_dataset, valid_dataset, metric, transformers)
```

`hyperparam_search()` returns three arguments: the best model it found, the hyperparameters for that model, and a full listing of the validation score for every model. Let's take a look at the last one.

In [3]:
```
all_results
```

Out[3]:

```
{'_dropouts_0.200000_layer_sizes[500]_learning_rate_0.001000_n_features_1024_n_tasks_1': 0.759624393738977,
 '_dropouts_0.200000_layer_sizes[500]_learning_rate_0.000100_n_features_1024_n_tasks_1': 0.7680791323731138,
 '_dropouts_0.500000_layer_sizes[500]_learning_rate_0.001000_n_features_1024_n_tasks_1': 0.7623870149911817,
 '_dropouts_0.500000_layer_sizes[500]_learning_rate_0.000100_n_features_1024_n_tasks_1': 0.7552282358416618,
 '_dropouts_0.200000_layer_sizes[1000]_learning_rate_0.001000_n_features_1024_n_tasks_1': 0.7689915858318636,
 '_dropouts_0.200000_layer_sizes[1000]_learning_rate_0.000100_n_features_1024_n_tasks_1': 0.7619292572996277,
 '_dropouts_0.500000_layer_sizes[1000]_learning_rate_0.001000_n_features_1024_n_tasks_1': 0.7641491524593376,
 '_dropouts_0.500000_layer_sizes[1000]_learning_rate_0.000100_n_features_1024_n_tasks_1': 0.7609877155594749,
 '_dropouts_0.200000_layer_sizes[1000, 1000]_learning_rate_0.001000_n_features_1024_n_tasks_1': 0.770716980207721,
 '_dropouts_0.200000_layer_sizes[1000, 1000]_learning_rate_0.000100_n_features_1024_n_tasks_1': 0.7750327625906329,
 '_dropouts_0.500000_layer_sizes[1000, 1000]_learning_rate_0.001000_n_features_1024_n_tasks_1': 0.725972314079953,
 '_dropouts_0.500000_layer_sizes[1000, 1000]_learning_rate_0.000100_n_features_1024_n_tasks_1': 0.7546280986674505}
```


We can see a few general patterns. Using two layers with the larger learning rate doesn't work very well. It seems the deeper model requires a smaller learning rate. We also see that 20% dropout usually works better than 50%. Once we narrow down the list of models based on these observations, all the validation scores are very close to each other, probably close enough that the remaining variation is mainly noise. It doesn't seem to make much difference which of the remaining hyperparameter sets we use, so let's arbitrarily pick a single layer of width 1000 and learning rate of 0.0001.

Early Stopping

There is one other important hyperparameter we haven't considered yet: how long we train the model for. `GridHyperparamOpt` trains each for a fixed, fairly small number of epochs. That isn't necessarily the best number.

You might expect that the longer you train, the better your model will get, but that isn't usually true. If you train too long, the model will usually start overfitting to irrelevant details of the training set. You can tell when this happens because the validation set score stops increasing and may even decrease, while the score on the training set continues to improve.

Fortunately, we don't need to train lots of different models for different numbers of steps to identify the optimal number. We just train it once, monitor the validation score, and keep whichever parameters maximize it. This is called "early stopping". DeepChem's `ValidationCallback` class can do this for us automatically. In the example below, we have it compute the validation set's ROC AUC every 1000 training steps. If you add the `save_dir` argument, it will also save a copy of the best model parameters to disk.

In [4]:
```
model = dc.models.MultitaskClassifier(n_tasks=len(tasks),
                                      n_features=1024,
                                      layer_sizes=[1000],
                                      dropouts=0.2,
                                      learning_rate=0.0001)
callback = dc.models.ValidationCallback(valid_dataset, 1000, metric)
model.fit(train_dataset, nb_epoch=50, callbacks=callback)
```

```
Step 1000 validation: roc_auc_score=0.759757
Step 2000 validation: roc_auc_score=0.770685
Step 3000 validation: roc_auc_score=0.771588
Step 4000 validation: roc_auc_score=0.777862
Step 5000 validation: roc_auc_score=0.773894
Step 6000 validation: roc_auc_score=0.763762
Step 7000 validation: roc_auc_score=0.766361
Step 8000 validation: roc_auc_score=0.767026
Step 9000 validation: roc_auc_score=0.761239
Step 10000 validation: roc_auc_score=0.761279
Step 11000 validation: roc_auc_score=0.765363
Step 12000 validation: roc_auc_score=0.769481
Step 13000 validation: roc_auc_score=0.768523
Step 14000 validation: roc_auc_score=0.761306
Step 15000 validation: roc_auc_score=0.77397
Step 16000 validation: roc_auc_score=0.764848
```

Out[4]:
```
0.8040038299560547
```

Learning Rate Schedules

In the examples above we use a fixed learning rate throughout training. In some cases it works better to vary the learning rate during training. To do this in DeepChem, we simply specify a `LearningRateSchedule` object instead of a number for the `learning_rate` argument. In the following example we use a learning rate that decreases exponentially. It starts at 0.0002, then gets multiplied by 0.9 after every 1000 steps.

In [5]:
```
learning_rate = dc.models.optimizers.ExponentialDecay(0.0002, 0.9, 1000)
```

```
model = dc.models.MultitaskClassifier(n_tasks=len(tasks),
                                      n_features=1024,
                                      layer_sizes=[1000],
                                      dropouts=0.2,
                                      learning_rate=learning_rate)
model.fit(train_dataset, nb_epoch=50, callbacks=callback)
```

```
Step 1000 validation: roc_auc_score=0.736547
Step 2000 validation: roc_auc_score=0.758979
Step 3000 validation: roc_auc_score=0.768361
Step 4000 validation: roc_auc_score=0.764898
Step 5000 validation: roc_auc_score=0.775253
Step 6000 validation: roc_auc_score=0.779898
Step 7000 validation: roc_auc_score=0.76991
Step 8000 validation: roc_auc_score=0.771515
Step 9000 validation: roc_auc_score=0.773796
Step 10000 validation: roc_auc_score=0.776977
Step 11000 validation: roc_auc_score=0.778866
Step 12000 validation: roc_auc_score=0.777066
Step 13000 validation: roc_auc_score=0.77616
Step 14000 validation: roc_auc_score=0.775646
Step 15000 validation: roc_auc_score=0.772785
Step 16000 validation: roc_auc_score=0.769975
```

Out[5]: 0.22854619979858398

Congratulations! Time to join the Community!

Congratulations on completing this tutorial notebook! If you enjoyed working through the tutorial, and want to continue working with DeepChem, we encourage you to finish the rest of the tutorials in this series. You can also help the DeepChem community in the following ways:

Star DeepChem on GitHub

This helps build awareness of the DeepChem project and the tools for open source drug discovery that we're trying to build.

Join the DeepChem Discord

The DeepChem Discord hosts a number of scientists, developers, and enthusiasts interested in deep learning for the life sciences. Join the conversation!

Citing This Tutorial

If you found this tutorial useful please consider citing it using the provided BibTeX.

In [ ]:
```
@manual{Intro9,
 title={Advanced Model Training},
 organization={DeepChem},
 author={Eastman, Peter and Ramsundar, Bharath},
 howpublished = {\url{https://github.com/deepchem/deepchem/blob/master/examples/tutorials/Advanced_Model_Traini
 year={2021},
}
```