NaturalSelection: A Python package for user-friendly evolution of neural networks

Dan Saattrup Nielsen
dan.nielsen@bristol.ac.uk
University of Bristol
United Kingdom

ABSTRACT

Hyperparameter optimisation of neural networks is an incredibly time consuming process. A grid search is usually too time consuming, and a random search does base its search on prior results and is therefore still wasteful. Other approaches include Bayesian optimisation and genetic algorithms, both of which are "guided", meaning that they base the choice of hyperparameters on past performance. We propose a new Python package, NaturalSelection, which provides a simple Pythonic API for applying genetic algorithms to general optimisation problems, with built-in support for tuning both the hyperparameters and architecture of neural networks. Several other Python packages that optimise hyperparameters of neural networks exist, but these are utilising algorithms that tune the hyperparameters not including the architecture, such as learning rate, momentum and batch size. Our package "evolves" a neural network, starting from a shallow network and gradually making it more complex while optimising performance. As a side benefit this prioritises simpler neural architectures, resulting in faster training times.

KEYWORDS

Neural networks, hyperparameter optimization, hyperparameter tuning, genetic algorithm, python

ACM Reference Format:
Dan Saattrup Nielsen. 2019. NaturalSelection: A Python package for user-friendly evolution of neural networks. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 EXAMPLE

Here is an example of finding a multilayer perceptron to model the well-known dataset Fashion-MNIST. Here the fashion_mnist_train_val_sets function fetches the dataset and performs standard normalisation preprocessing.

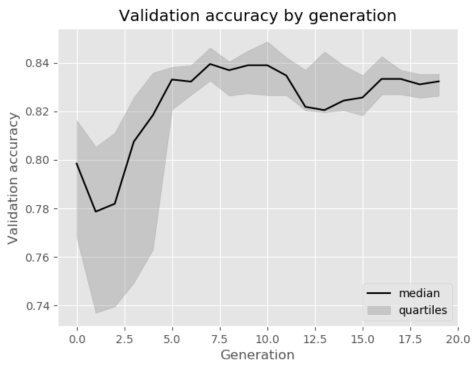
Listing 1: Evolution of a population of MLPs

```
>>> import naturalselection as ns
>>> nns = ns.NNs(
...     size = 20,
...     train_val_sets = train_val_sets(),
...     loss_fn = 'categorical_crossentropy',
...     score = 'accuracy',
...     output_activation = 'softmax',
...     max_epochs = 1,
...     max_training_time = 120,
...     )
...
>>> history = nns.evolve(generations = 20)
Evolving population: 100%|===| 20/20 [1:32:00<00:00]
Computing fitness: 100%|=======| 11/11 [05:37<00:00]
>>>
>>> history.fittest
{'genome': {'optimizer': 'adamax',
'activation': 'relu', 'batch_size': 32,
'initializer': 'glorot_uniform',
'input_dropout': 0.0,
'neurons0': 256, 'dropout0': 0.3,
'neurons1': 512, 'dropout1': 0.3,
'neurons2': 128, 'dropout2': 0.0,
'neurons3': 0, 'dropout3': 0.1,
'neurons4': 1024, 'dropout4': 0.2},
'fitness': 0.855400025844574}
```

From the evolved population we can then plot the evolution.

Listing 2: Plotting the evolution

```
>>> history.plot(
...     title = "Validation accuracy by generation",
...     ylabel = "Validation accuracy"
...     )
```



Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Lastly, we can train the best performing model and save it locally:

Listing 3: Saving the optimised model

```
>>> # Training the best model and saving
>>> # it to fashion_mnist_model.h5
>>> best_score = nns.train_best(
...     file_name = 'fashion_mnist_model'
...     )
Epoch 0: 100%|====| 60000/60000 [00:21<00:00]
(...)
Epoch 44: 100%|===| 60000/60000 [00:21<00:00]
>>>
>>> best_score
0.9029
```