parrot Documentation

parrot

May 02, 2023

Contents:

CHAPTER 1

Getting Started with PARROT

1.1 Installation

PARROT is available through GitHub or the Python Package Index (PyPI). To install through PyPI, run

```
$ pip install idptools-parrot
```

To clone the GitHub repository and gain the ability to modify a local copy of the code, run

```
$ git clone https://github.com/idptools/parrot.git
$ cd parrot
$ pip install .
```

This will install PARROT locally. If you modify the source code in the local repository, be sure to reinstall with pip.

JULY 2022 UPDATE

To mitigate package version dependency issues involving Python, GPy, and PyTorch, new releases of PARROT have separated *parrot-optimize* as an optional add-on installation. All of the documentation for using *parrot-optimize* is unchanged. However, if you wish to use the hyperparameter optimization, there are now slight differences in how you install PARROT.

To install the PARROT that is compatible with *parrot-optimize* install via pip using:

```
$ pip install idptools-parrot[optimize]
```

or

```
$ pip install "idptools-parrot[optimize]"
```

Alternatively if you have the PARROT repository cloned locally you can install using

```
$ pip install ".[optimize]"
```

1.2 Testing

To see if your local installation of PARROT is working properly, first install the "pytest" package:

```
$ pip install pytest
```

Then, you can run the unit test included in the package by navigating to the /tests folder within the installation directory and running:

```
$ pytest -v
```

Note that this only works if the package is installed as a repository via GitHub. Installation through PyPI does not include the necessary datafiles to run the tests.

1.3 Example datasets

Example data that can be used with PARROT can be found in the **/data** folder on GitHub. Examples of usage with these datasets can be found in the documentation.

CHAPTER 2

parrot-train

`parrot-train` is the primary command for training a PARROT network. In the most basic usage, the user specifies their data and an output location and a trained bi-directional LSTM network will be created along with an estimate of network predictive performance on randomly chosen test samples. There are many optional arguments to `parrot-train` that allow users to specify network parameters, create helpful output images, and otherwise modify PARROT to meet their needs.

Once PARROT is installed, the user can run `parrot-train` from the command line:

```
$ parrot-train data_file output_network <flags>
```

Where *data_file* specifies the path to the whitespace-separated datafile and *output_network* is the path to where the final trained network will be saved. It is recommended, but not required, that networks are saved using a ".pt" file extension, following the PyTorch convention.

Required flags:

- `--datatype` / `-d` : Describes how values are formatted in *datafile*. Should be 'sequence' if there is a single value per sequence, or 'residues' if there are values for every residue in each sequence. See the example datasets in the **/data** folder for more information.
- `--classes` / `-c` : The number of classes for the machine learning task. If the task is regression, then specify '1' (without the quote marks).

Optional flags:

- `--help` / `-h` : Display a help message.
- `--learning-rate` / `-lr` : Learning rate of the network (default is 0.001). Must be a float between 0 and 1.
- `--num-layers` / `-nl` : Number of hidden layers in the network (default is 1). Must be a positive integer.
- `--hidden-size` / `-hs` : Size of hidden vectors within the network (default is 10). Must be a positive integer.
- `--batch` / `-b` : Training minibatch size (default is 32). Must be a positive integer, and for most datasets should be in the range of 8-256. Powers of 2 (64, 128, 256, etc.) are optimized for slightly faster performance, but are not explicitly required.

- `--epochs` / `-e` : Number of training epochs (default is 100). Has different behavior depending on what is specified by the `--stop` flag.
- `--stop` : Stop condition to terminate training. Must be either 'auto' or 'iter' (default is 'iter'). If 'iter', then train for exactly `-e` epochs and stop. If 'auto', then train until performance has plateaued for `-e` epochs. If using 'auto', be careful not to indicate a large number of epochs, as this can take much longer than is necessary.
- `--split` : Path to a "split-file" for manually dividing dataset into training, validation and test sets. The file should contain three lines, corresponding to training, validation and test sets respectively. Each line should have integers separated by whitespace, with the integers specify which sequences/lines in the *datafile* (0-indexed) will belong to which dataset. See **/data** folder for examples. If a split-file is not provided, default behavior is for PARROT to randomly divide data into training, validation and test sets.
- `--set-fractions` : Include this flag to manually set the proportions of the data belonging to the training, validation and test sets. This option must be followed by three floats (representing train, validation, and test) between 0 and 1 that cumulatively sum to 1. By default, PARROT uses splits of 70:15:15. Note that the `--split` flag overrides these values.
- `--save-splits` : Include this flag if you would like PARROT to produce a split-file based on its random partitioning of data into training/validation/test sets, which can be useful for replication and/or testing multiple networks on the same data. Output split-file will be saved in the same folder as *output_network* using the same name followed by "_split_file.txt". This flag is overridden if a split-file is manually provided. (NOTE: This is a new feature, let us know if you run into any issues!)
- `--encode` : Include this flag to specify the numeric encoding scheme for each amino acid. Available options are 'onehot' (default), 'biophysics' or user-specified. If you wish to manually specify an encoding scheme, provide a path to a text file describing the amino acid to vector mapping. Example encoding files are provided in the **/data** folder.
- `--exclude-seq-id` : Include this flag if the *data_file* is formatted without sequence IDs as the first column in each row.
- `--probabilistic-classification` : Include this flag to output class predictions as continuous values [0-1], based on the probability that the input sample belongs to each class. Currently only implemented for sequence classification. This flag also modifies the output figures and output performance stats. (NOTE: This is a new feature, let us know if you run into any issues!)
- `--include-figs` : Include this flag to generate images based on network training and performance on test set. Figures will be saved to same directory as specified by *output_network* using same naming convention.
- `--no-stats` : Include this flag to prevent a "_performance_stats.txt" file from being output.
- `--ignore-warnings` : By default, PARROT checks your data for a few criteria and prints warnings if it doesn't meet some basic heuristics. Use this flag to silence these warnings (network training occurs unimpeded in either case).
- `--force-cpu` : Include this flag to force network training on the CPU, even if a GPU is available.
- `--verbose` / `-v` : Include this flag to produce more descriptive output to terminal.
- `--silent` : Include this flag to produce no output to terminal.

Output:

After running `parrot-train`, at least three files will be saved to the directory specified by *output_network*. One contains the saved network weights from the training process which can be used with `parrot-predict`. The other file with the suffix "_predictions.tsv" contains the true values and predicted values for all of the sequences in the test set. The final "_performance_stats.txt" file quantifies a variety of performance metrics on the test set. Output text detailing network performance across training is also printed to terminal by default.

If `--include-figs` is provided, there will be additional PNG images saved in this directory. The first, with suffix '_train_val_loss.png' displays the network's performance on the training and validation sets over the course of training.

The other image(s) describes the network performance on the held out test set, and will vary depending on the data format and machine learning task. If training a network for a classification task, the image will be a confusion matrix. If training for a regression task, the image will be a scatterplot comparing the predicted and true values of the test set sequences. If using probabilistic-classification mode, then there will be two output figures: one plotting receiver operator characteristic (ROC) curves and the other plotting precision-recall curves.

CHAPTER 3

parrot-optimize

`parrot-optimize` is a command for performing hyperparameter optimization on a given dataset and then training a PARROT network with the optimal hyperparameters. The dataset is divided 5-fold and iteratively retrained and validated using cross-validation to estimate the performance of the network using a given set of hyperparameters. Bayesian optimization ensures that in each iteration, hyperparameters are selected that better sample hyperparameter space and/or are expected to improve performance. After a specified number of iterations, the algorithm returns the best-performing hyperparameters and a new network will be trained on the cumulative cross-validation data, then tested on held out test data. Because of its iterative, cross-validation-based procedure, using `parrot-optimize` takes **significantly** longer to train a network than `parrot-train`, and is not recommended for basic usage and for users running PARROT on computers lacking GPUs. The final trained network will be saved to to memory along with a file specifying the optimal hyperparameters and several estimates of network performance on a held-out test set (identically to `parrot-train`).

NOTE: there are slightly different instructions for installing optimize-compatible PARROT. See the "Getting Started with PARROT" page for more information. Once PARROT is installed, the user can run `parrot-optimize` from the command line:

```
$ parrot-optimize data_file output_network <flags>
```

Where *data_file* specifies the path to the whitespace-separated datafile and *output_network* is the path to where the final trained network will be saved. It is recommended, but not required, that networks are saved using a ".pt" file extension, following the PyTorch convention.

Required flags:

- `--datatype` / `-d` : Describes how values are formatted in *datafile*. Should be 'sequence' if there is a single value per sequence, or 'residues' if there are values for every residue in each sequence. See the example datasets in the **data** folder for more information.
- `--classes` / `-c` : The number of classes for the machine learning task. If the task is regression, then specify '1'.

Optional flags:

- `--help` / `-h` : Display a help message.

- `--batch` / `-b` : Training minibatch size (default is 32). Must be a positive integer, and for most datasets should be in the range of 8-256. Powers of 2 (64, 128, 256, etc.) are optimized for slightly faster performance, but are not explicitly required.
- `--epochs` / `-e` : Number of epochs to train for during each iteration of optimization on each data fold (default is 100).
- `--max-iter` : Maximum number of iterations the optimization algorithm should run for. Default is 75.
- `--split` : Path to a "split-file" for manually dividing dataset into training, validation and test sets. The file should contain three lines, corresponding to training, validation and test sets respectively. Each line should have integers separated by whitespace, with the integers specify which sequences/lines in the *datafile* (0-indexed) will belong to which dataset. See **/data** folder for examples. If a split-file is not provided, default behavior is for PARROT to randomly divide data into training, validation and test sets. The cross-validation folds will be divided from the training and validation data.
- `--set-fractions` : Include this flag to manually set the proportions of the data belonging to the training, validation and test sets. This option must be followed by three floats (representing train, validation, and test) between 0 and 1 that cumulatively sum to 1. By default, PARROT uses splits of 70:15:15. Note that the `--split` flag overrides these values.
- `--save-splits` : Include this flag if you would like PARROT to produce a split-file based on its random partitioning of data into training/validation/test sets, which can be useful for replication and/or testing multiple networks on the same data. Output split-file will be saved in the same folder as *output_network* using the same name followed by "_split_file.txt". This flag is overridden if a split-file is manually provided. (NOTE: This is a new feature, let us know if you run into any issues!)
- `--encode` : Include this flag to specify the numeric encoding scheme for each amino acid. Available options are 'onehot' (default), 'biophysics' or user-specified. If you wish to manually specify an encoding scheme, provide a path to a text file describing the amino acid to vector mapping. Example encoding files are provided in the **/data** folder.
- `--exclude-seq-id` : Include this flag if the *data_file* is formatted without sequence IDs as the first column in each row.
- `--probabilistic-classification` : Include this flag to output class predictions as continuous values [0-1], based on the probability that the input sample belongs to each class. Currently only implemented for sequence classification. This flag also modifies the output figures and output performance stats. (NOTE: This is a new feature, let us know if you run into any issues!)
- `--include-figs` : Include this flag to generate images based on network training and performance on test set. Figures will be saved to same directory as specified by *output_network* using same naming convention.
- `--no-stats` : Include this flag to prevent a "_performance_stats.txt" file from being output.
- `--ignore-warnings` : By default, PARROT checks your data for a few criteria and prints warnings if it doesn't meet some basic heuristics. Use this flag to silence these warnings (network training occurs unimpeded in either case).
- `--force-cpu` : Include this flag to force network training on the CPU, even if a GPU is available.
- `--verbose` / `-v` : Include this flag to produce more descriptive output to terminal.
- `--silent` : Include this flag to produce no output to terminal.

Output:

`parrot-optimize` will produce similar output as `parrot-train`, with one exception. `parrot-optimize` also produces a file "optimal_hyperparams.txt" which specifies which hyperparameters were ultimately chosen for the final network.

CHAPTER 4

parrot-predict

`parrot-predict` is a command for making predictions using a trained PARROT network. The `parrot-train` and `parrot-optimize` commands both output a file with trained network weights, and this trained network can be used by `parrot-predict` to make new predictions on unlabeled sequences. The prediction will be output as a text file saved to a specified location. Note that this command will only make predictions for *non-redundant* sequences in the provided file. Currently, users must input the hyperparameters (–num-layers and –hidden-size) they used to train their network originally, but in future versions of PARROT, `parrot-predict` will be able to dynamically read in your saved network and automatically detect these hyperparameters.

Once PARROT is installed, the user can run `parrot-predict` from the command line:

```
$ parrot-predict seq_file saved_network output_file <flags>
```

Where *seq_file* specifies a file containing a list of sequences. Each line of *seq_file* should have two whitespace-separated columns: a sequence ID and the amino acid sequence. Optionally, the file may also be formatted without the sequence IDs. Two example *seq_file* can be found in the **/data** folder. *saved_network* is the path to where the trained network is saved in memory. *output_file* is the path to where the predictions will be saved as a text file.

Required flags:

- `--datatype` / `-d` : Describes how values are formatted in *datafile*. Should be 'sequence' if there is a single value per sequence, or 'residues' if there are values for every residue in each sequence. See the example datasets in the **data** folder for more information.
- `--classes` / `-c` : The number of classes for the machine learning task. If the task is regression, then specify '1'.

Optional flags:

- `--help` / `-h` : Display a help message.
- `--num-layers` / `-nl` : Number of hidden layers in the network (default is 1). Must be a positive integer and must be identical to the number of layers used when the network was trained.
- `--hidden-size` / `-hs` : Size of hidden vectors within the network (default is 10). Must be a positive integer and must be identical to the hidden size used when the network was trained.

- `--encode` : Include this flag to specify the numeric encoding scheme for each amino acid. Available options are 'onehot' (default), 'biophysics' or user-specified. If you wish to manually specify an encoding scheme, provide a path to a text file describing the amino acid to vector mapping. The encoding scheme used for sequence prediction must be identical to that used for network training.
- `--exclude-seq-id` : Include this flag if the *seq_file* is formatted without sequence IDs as the first column in each row.
- `--probabilistic-classification` : Include this flag to output class predictions as continuous values [0-1], based on the probability that the input sample belongs to each class. Currently only implemented for sequence classification. (NOTE: This is a new feature, let us know if you run into any issues!)
- `--silent` : Flag which, if provided, ensures no output is generated to the terminal.
- `--print-frequency` : Value that defines how often status updates should be printed (in number of sequences predicted). Default=1000

Output:

`parrot-predict` will produce a single text file as output, as well as status updates to the console (if `--silent` is not specified). This file will be formatted similarly to the original datafiles used for network training: each row contains a sequence ID (exluded if the flag `--exclude-seq-id` is given), an amino acid sequence, and the prediction values for that sequence.

CHAPTER 5

parrot-cvsplit

`parrot-cvsplit` is a command for generating PARROT-usable *split-files* for conducting K-fold cross validation on a dataset (where K is specified by the user). This function does not extensively validate the data it's passed and assumes that it a PARROT-usable datafile.

Once PARROT is installed, the user can run `parrot-cvsplit` from the command line:

```
$ parrot-cvsplit data_file output_splitfiles <flags>
```

Where *data_file* specifies the path to the whitespace-separated datafile and *output_splitfiles* denotes where the output split-files will be saved from this command. Files are output as <output_splitfiles>_cv#.txt for # = 0 . . . K-1.

Optional flags:

- `--help` / `-h` : Display a help message.
- `--k-folds` / `-k` : Number of split-files to generate for K-fold cross-validation (default is 10). This term determines the train+validation vs test set split, as for each fold the test set will contain ~1/K of the data, while the training and validation sets combined will contain (K-1)/K. Must be a positive integer.
- `--training-fraction` / `-t` : Percent of the non-test data that should be partioned into the training set for each fold (default is 0.8). This term determines the train vs validation set split, as for each fold the training set fraction will equal ~((K-1)/K) * *training-fraction* of the data, and the validation set fraction will equal ~((K-1)/K) * (1-*training-fraction*). Must be a float between 0 and 1.

Output:

`parrot-cvsplit` will generate K files to the specified location by *output_splitfiles*. Each file is a split-file designed to be used with `parrot-train` along with the `--split` flag.

CHAPTER 6

Basic Examples:

Below are a handful of examples outlining how to apply PARROT to various machine learning tasks. Provided in the PARROT distribution on GitHub is a **/data** folder which contains several example datasets (If you installed via pip you will need to download these files from GitHub). Among these, there are datasets with different data types (sequence-mapped values and residue-mapped values) and for different machine learning task (classification and regression). Read the README within this folder for more details on how to format PARROT datasets and on the particulars of these datasets. This folder also contains an example list of sequences for `parrot-predict`, and the other files that are used in these examples.

6.1 parrot-train

Sequence classification:

In our first example, each of the 300 sequences in *seq_class_dataset.tsv* belongs to one of three classes:

```
Frag0 WKHNPKHLRP 0
Frag1 DLFQDEDDAEEEDFMDDIWDPDS 1
Frag2 YHFAFTHMPALISQTSKYHYYSASMRG 2
Frag3 CNRNRNHKLKKFKHKKMGVPRKKRKHWK 0
...
Frag296 PDPLAMEDEVESHMEWCNRTHNRKG 2
Frag297 IWKYTHRSKACMHPH 0
Frag298 EDDEDVDENEEDDEDEEDNEEDPIE 1
Frag299 GEPCWVPYDIAQSADRMFFDKAMR 2
```

Let's train a network with `parrot-train` that learns how to classify sequences into these three data classes. Details on running the `parrot-train` command can be be found on its specific documentation page. For starters, we won't worry about the network hyperparameters and we'll just use the default values. In the most basic use case, all we need to provide is the datafile, the location where we want to save the trained network, and some basic information about what kind of network we are training. Here, since we are predicting only one value per sequence, we will indicate that the datatype is "sequence". We also will indicate how many data classes there are, which is '3' in this case.

```
parrot-train data/seq_class_dataset.tsv seq_class_network.pt --datatype sequence --
↪classes 3
```

Training has a stochastic component, so running this multiple times will yield slightly different results. The output to console should look something like:

```
#############################################
WARNING: Batch size is large relative to the number of samples in the training set.


This may decrease training efficiency.

#############################################



PARROT with user-specified parameters
-------------------------------------
Validation set loss per epoch:
Epoch 0 Loss 1.0972
Epoch 5 Loss 1.0831
Epoch 10    Loss 1.0446
Epoch 15    Loss 0.8525
Epoch 20    Loss 0.6279
Epoch 25    Loss 0.3620
Epoch 30    Loss 0.2419
Epoch 35    Loss 0.2805
Epoch 40    Loss 0.2273
Epoch 45    Loss 0.1774
Epoch 50    Loss 0.2103
Epoch 55    Loss 0.1682
Epoch 60    Loss 0.1524
Epoch 65    Loss 0.1488
Epoch 70    Loss 0.1498
Epoch 75    Loss 0.1464
Epoch 80    Loss 0.1448
Epoch 85    Loss 0.1564
Epoch 90    Loss 0.1593
Epoch 95    Loss 0.1581

Test Loss: 0.1507
```

First we notice that there is a message warning us that our batch size is too large. This isn't super problematic and we can ignore it for now. In future runs, we will decrease our batch size using the `--batch` flag (by default it's set to 32, which is pretty large relative to our dataset of only 300 sequences). Also note that you can explicitly hide warning messages with the `--ignore-warnings` flag.

Turning to the actual training results, we can see that our validation set loss decreases for a while, then plateaus around 0.14-0.15. This is pretty typical, generally this loss will decrease up to a certain point, then start to increase as the network begins to overfit on the training data. Don't worry about this overfitting, since the final network that PARROT returns will be from the iteration that produced the lowest validation set loss.

If you look in the current directory, you should also see three files: our trained network "seq_class_network.pt", a predictions file "seq_class_network_predictions.tsv", and a performance stats summary file "seq_class_network_performace_stats.txt". The network file can be used to make predictions on new sequences with `parrot-predict` but is not readable by eye. The second file is a bit more interesting to look at:

```
Frag1_TRUE DLFQDEDDAEEEDFMDDIWDPDS 1
```

(continues on next page)

(continued from previous page)

```
Frag1_PRED DLFQDEDDAEEEDFMDDIWDPDS 1
Frag20_TRUE SWQIHMPQWQCKHDMIQWLGDDAQ 2
Frag20_PRED SWQIHMPQWQCKHDMIQWLGDDAQ 2
Frag21_TRUE HQPKRKHHHYQHARHHHHKRVH 0
Frag21_PRED HQPKRKHHHYQHARHHHHKRVH 0
...
Frag273_TRUE LLHRHRFQRSTKRHLLK 0
Frag273_PRED LLHRHRFQRSTKRHLLK 0
Frag286_TRUE DDEDEDYWNEWEETEEIQESE 1
Frag286_PRED DDEDEDYWNEWEETEEIQESE 1
Frag299_TRUE GEPCWVPYDIAQSADRMFFDKAMR 2
Frag299_PRED GEPCWVPYDIAQSADRMFFDKAMR 2
```

NOTE: Your file will have the same general format, but with different sequences. These sequences are the ones that were randomly held out as a test set during the training of our network. After the network concluded training, the best-perfoming network (on the validation set) was applied to these test set sequences. **By analyzing this file, we can get an approximation of how well our network would perform on sequences it has not seen before.** This approximation may not hold in every case, but sometimes, it's the best we can do (see “Machine Learning Resources” for more info). In our case, it seems as if our network did a good job at predicting these test set sequences.

The performance stats file is an extension of these test set predictions:

```
Matthews Correlation Coef : 1.000
F1 Score : 1.000
Accuracy : 1.000
```

This file quantifies performance on the test set using a variety of different metrics, which vary between classification and regression tasks. For classification, as shown here, this file reports on the accuracy, F1 score and MCC of our predictions. You can always prevent this file from being output by providing the `--no-stats` flag. See “Machine Learning Resources” (or Google!) for more information on how to interpret these metrics.

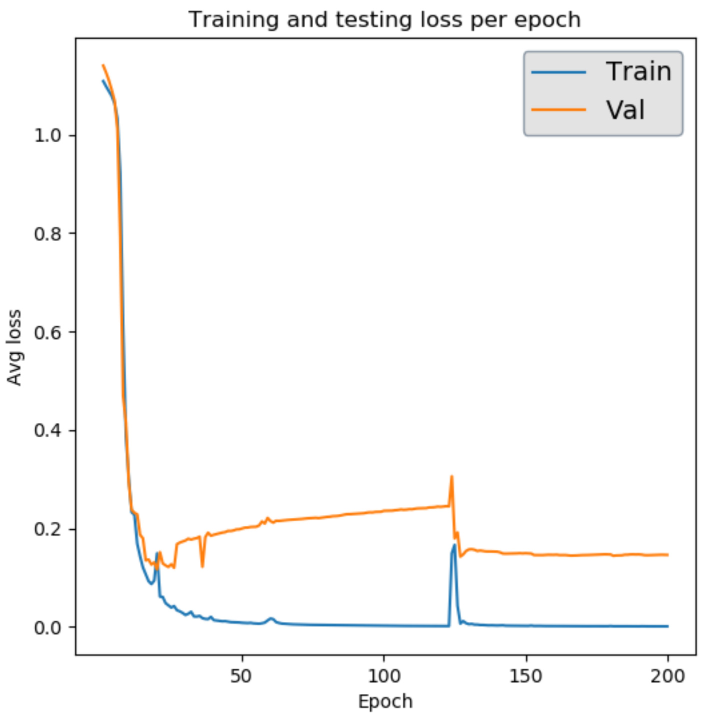
Let's demonstrate a few more features of PARROT by training another network. In this run, we'll decrease the `--batch` parameter to ‘8’ to get rid of the warning. A smaller batch size will cause the network to update more often during training, which means that training will take longer overall, but the network will learn more each epoch.

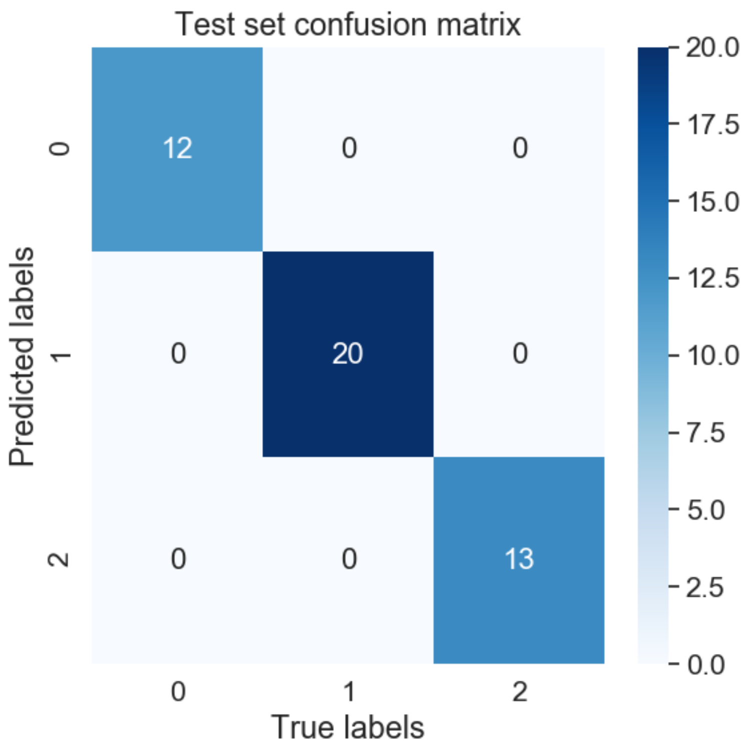
Additionally, we will also modify the training time with the `--epochs` flag. In the context of machine learning, an epoch is one “round” through the entire training set. By default, PARROT trains for 100 epochs, which means that a network will be exposed to every sequence in the training set 100 times. It's often necessary to increase this parameter to ensure that the network learns the data to its maximum potential.

The remaining two flags we will add are `--verbose` and `--include-figs`. “Verbose” simply causes the output to terminal to be more descriptive, printing the training results after every epoch instead of every 5. As the name suggests, “include-figs” will cause PNG images to be output into the same directory that we are saving the network.

```
parrot-train data/seq_class_dataset.tsv seq_class_network.pt --datatype sequence --
↪classes 3 --batch 8 --epochs 200 --include-figs --verbose
```

Let's look at the figures that we generated: “seq_class_network_train_val_loss.png” and “seq_class_network_seq_CM.png”





The first is a plot of the performance achieved by the network on the training and validation sets over the course of training. The validation loss here is the same as what is being output to terminal. This particular plot looks a little funny, but that's due to the fact that this classification task is not very difficult, so our network learns what it needs too by around epoch 20 and the rest of the time is just overfitting and noise.

The second figure is provides some insight on how well our network will generalize onto unseen data. After training completes, PARROT networks are applied to a test set of randomly held-out sequences. For a classification task, PARROT displays a confusion matrix detailing the true vs predicted classes for each sequence in this test set. As you can see, our network is perfect (also confirmed by our performance stats file)!

Sequence regression:

Training a PARROT network on a regression task is very similar to classification in terms of syntax. For this example we will use *seq_regress_dataset.tsv*:

```
Frag0 EHCWTYIFQMYRIDQTQRVKRGEKPIIYLEPMAR 3.8235294117647056
Frag1 SDAWVMKFLWDKCGDHFIQYQKPANRWEWVD 3.870967741935484
Frag2 IYPEQSPDNAWAW 3.076923076923077
...
Frag296 VWIMYFIA 8.75
Frag297 WICEWRVP 5.0
Frag298 YMYWTDDWEA 5.0
Frag299 PCHSWSMEGILCNHMH 3.125
```

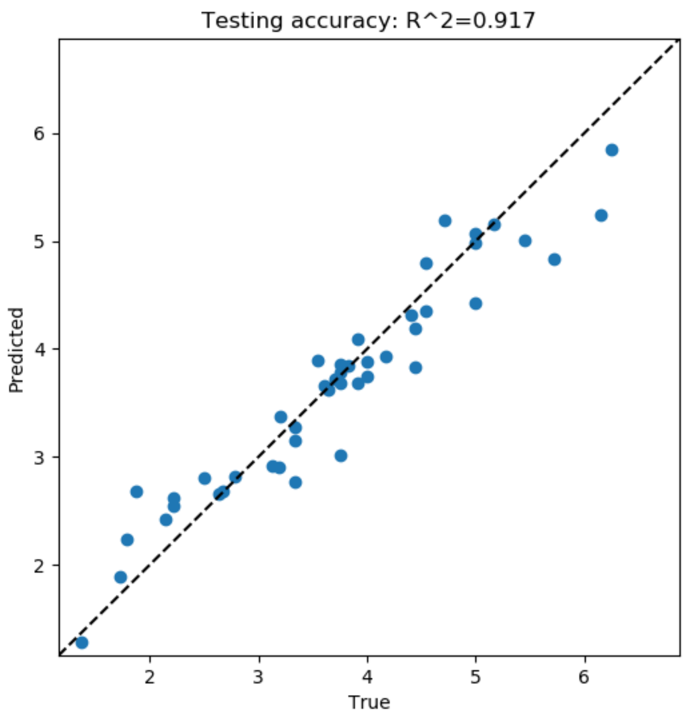
The key difference between regression datasets and classification datasets is that each value is a continuous number rather than an integer class label.

In terms of command-line syntax, the only difference in the `parrot-train` command for this regression case (other than the datafile path) is the `--classes` argument. Since we are doing regression, we will put '1' here. For the purposes of demonstration, we will also modify a few of the network hyperparameters in this run. Instead of the default network architecture with one hidden layer (`-nl 1`) and a hidden vector size of 10 (`-hs 10`), we will train a network with 2 layers and a vector size of 20. These two hyperparameters, along with learning rate (`-lr`), are the main ways to tune PARROT networks.

```
parrot-train data/seq_regress_dataset.tsv seq_regress_network.pt --datatype sequence -
↪-classes 1 -nl 2 -hs 20 -b 8 --epochs 200 --include-figs
```

You might notice that this network seems to train a bit slower than the previous example. This is because our network has an additional layer. Increasing the `-nl` hyperparameter increases training time, but creates a more complex network that may be better at discerning patterns from data.

Like before, this command outputs a network file, a prediction file, a performance stats file, a training results PNG and a test set performance PNG into the current directory. In this case, the performance image is a scatterplot that compares the true values of the test set sequences to what was predicted by the PARROT network.



The performance stats file provides the Pearson and Spearman correlations for this true vs predicted value scatterplot:

```
Pearson R : 0.958
Spearman R : 0.963
```

Not bad!

Residue classification:

Now let's try a task where the objective is to classify each residue in a sequence. Unlike before where every sequence had one class label, in *res_class_dataset.tsv* there are labels for every residue in each sequence.

```
Frag0 DEDGTEDDMATTK 1 1 1 1 1 1 1 1 1 1 1 1 1
Frag1 CGSAPSRFVKTCDPDEEDEDDEDE 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
Frag2 EWYEDDKPFPCPERVPHHKKGHRGGWRAKKNWKV 1 1 1 1 1 1 1 0 2 2 2 2 2 2 2 2 2 0 0 0 0 0␣
↪0 0 0 0 0 0 0 0 0 0 0 0
...
Frag297 HHWHRWDYERHKNCPIAGRIRR 0 0 0 0 0 0 0 1 1 1 0 0 0 0 2 2 2 2 0 0 0 0
Frag298 CEDEEEDEDHHQGPHHRT 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
Frag299 DPATGETHHDEDIEDSVEEDEDDDQDS 1 1 2 2 2 2 2 2 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1␣
↪1 1
```

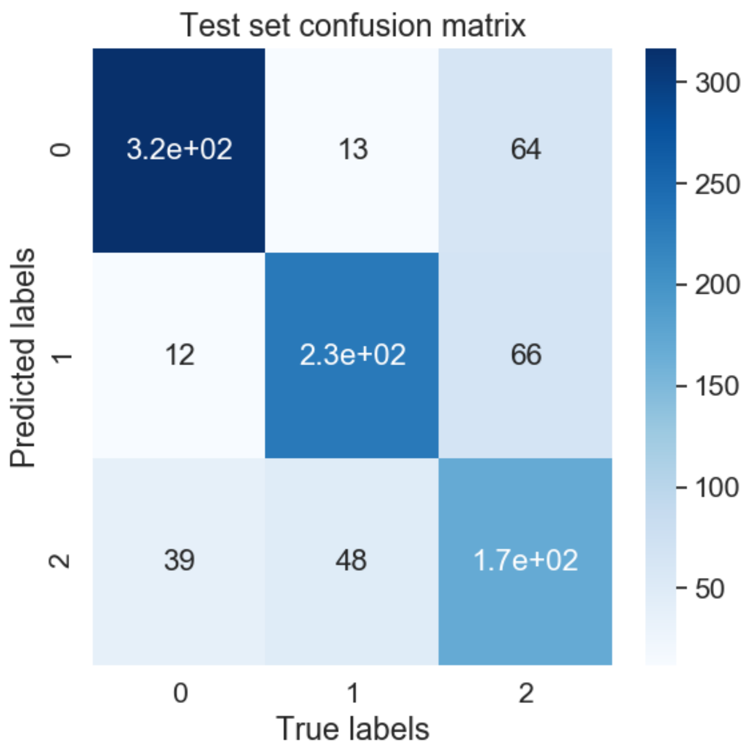
Despite this major difference, the `parrot-train` command is similar to the above examples. The only difference will be the value we input after the `--datatype` flag. Before we put 'sequence', and here we will put 'residues'. Just for demonstration, we will also decrease our learning rate (`-lr`) by an order of magnitude for training this network.

```
parrot-train data/res_class_dataset.tsv res_class_network.pt --datatype residues --
↪classes 3 -lr 0.0001 -e 200 -b 8 --include-figs
```

This produces more files to the output directory. If we look at the performance stats file, we can see this network is not perfectly accurate.

```
Matthews Correlation Coef : 0.621
F1 Score : 0.744
Accuracy : 0.748
```

In this case, the confusion matrix is for every single residue in all of the sequences in the test set. Looking at the confusion matrix can shed some light on which classes our network has trouble with.



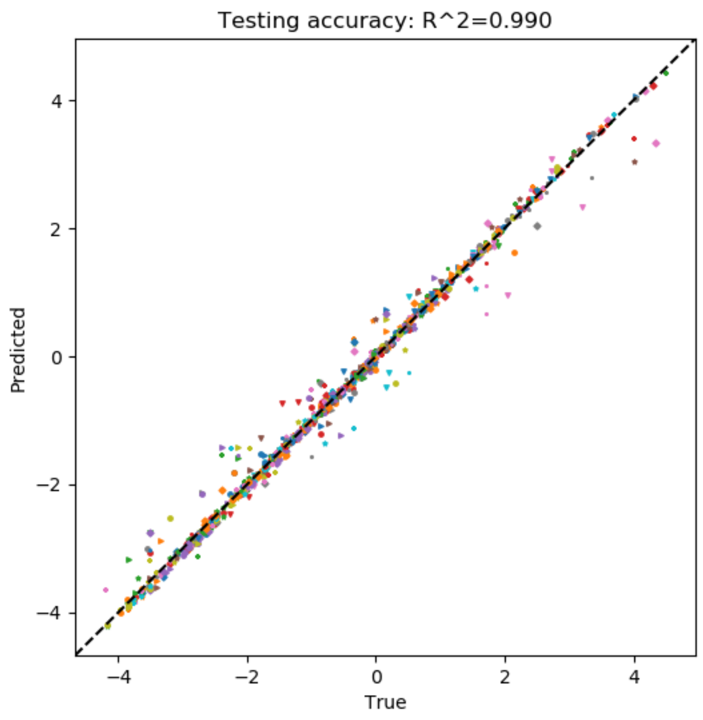
Evidently class ‘2’ is the tricky one in this example problem.

Residue regression:

The final kind of machine learning task that PARROT can handle is regression on every residue in a sequence. For this command `--datatype` should be set to ‘residues’ and `--classes` should be ‘1’. Notice that for convenience, we can use `-d` and `-c` for these flags. For this network, we’ll use all of the default hyperparameters and train for 300 epochs.

```
parrot-train data/res_regress_dataset.tsv res_regress_network.pt -d residues -c 1 -e␣
↪300 -b 8 --include-figs
```

The output from this command is analogous to the other examples. Like the sequence regression task, specifying `--include-figs` with a residue regression task will produce a scatter plot that shows the network’s performance on the test set.



Here, each point represents a single residue in the test set. Each combination of marker shape and color in this scatterplot belongs to a single sequence, which may provide some insight on whether the network systematically mis-predicts all sequences, or if there are only a few specific sequences that are outliers.

6.2 parrot-predict

You can use a trained network from `parrot-optimize` or `parrot-train` to predict the values of new, unseen sequences. An example file is provided in **/data** folder:

```
a1 EADDGLYWQQN
b2 RRLKHEEDSTSTSTSTSTQ
c3 YYYGGAFAFAGRM
d4 GGIL
e5 GREPCCMLLYILILAAAQRDESSSSST
f6 PGDEADLGHRSLVWADD
```

To run `parrot-predict`, we need to provide the path to this sequence file, the path to our trained network file, the location where we want to output our predictions to, and information on network type and architecture. The most important thing to keep in mind when using `parrot-predict` is that your `-nl` and `-hs` hyperparameters (and encoding scheme) must exactly match those used for network training, or else you will get an error.

Let's run our trained sequence regression network on this sequence file. Note the `-nl` and `-hs` flags are same as we used above.

```
parrot-predict data/seqfile.txt seq_regress_network.pt seq_regress_newPredictions.txt␣
↪--datatype sequence --classes 1 -nl 2 -hs 20
```

We can see these predictions in "seq_regress_newPredictions.txt":

```
a1 EADDGLYWQQN 2.8656542
b2 RRLKHEEDSTSTSTSTSTQ 0.7592569
c3 YYYGGAFAFAGRM 4.2728763
d4 GGIL 3.238177
e5 GREPCCMLLYILILAAAQRDESSSSST 3.377026
f6 PGDEADLGHRSLVWADD 2.486051
```

Remember: results will vary since networks train with stochasticity.

Now let's make predictions on the same sequences with our residue classification network. We don't need to provide hyperparameters here because we used the default values above.

```
parrot-predict data/seqfile.txt res_class_network.pt res_class_newPredictions.txt --
↪datatype residues --classes 3
```

```
a1 EADDGLYWQQN 1 1 1 1 1 1 1 1 1 1 2
b2 RRLKHEEDSTSTSTSTSTQ 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2
c3 YYYGGAFAFAGRM 2 2 2 2 2 2 2 2 2 2 2 0 0
d4 GGIL 2 2 2 2
e5 GREPCCMLLYILILAAAQRDESSSSST 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
f6 PGDEADLGHRSLVWADD 1 1 1 1 1 1 2 2 0 0 2 2 2 2 2 1 1
```

CHAPTER 7

Advanced Examples:

The core usage of PARROT is designed to be as simple as possible so that anyone, regardless of computational expertise, can train a network on their dataset with minimal investment. However, on top of this basic implementation, PARROT has a number of options intended to let experienced users tailor their networks to their particular needs and to facilitate more sophisticated computational workflows.

7.1 Advanced `parrot-train` options:

Automatic determination of number of training epochs with –stop:

This flag determines the stop condition for network training. Currently, there are two options implemented: either 'iter' or 'auto'. In all of the previous examples we used the default behavior, 'iter', which means that the number we specify for the `-e` flag will be the number of iterations that we train the network. Alternatively, using 'auto' means that training will stop automatically once performance on the validation set has plateaued for `-e` epochs. Thus, with 'auto' it is recommended to use a smaller number of epochs (10-20) for `-e` so training does not extend for a significantly long period of time.

```
parrot-train data/seq_regress_dataset.tsv stop_example.pt --datatype sequence -c 1 -
↪nl 2 -hs 5 -lr 0.001 -e 10 -b 32 -v --stop auto
```

```
PARROT with user-specified parameters
-------------------------------------
Train on:   cpu
Datatype:   sequence
ML Task:    regression
Learning rate:  0.001000
Number of layers:   2
Hidden vector size: 5
Batch size: 32

Validation set loss per epoch:
```

(continues on next page)

(continued from previous page)

```
Epoch 0 Loss 0.1779
Epoch 1 Loss 0.1752
Epoch 2 Loss 0.1727
...
Epoch 98    Loss 0.0456
Epoch 99    Loss 0.0456
Epoch 100   Loss 0.0456
Epoch 101   Loss 0.0456
Epoch 102   Loss 0.0456
Epoch 103   Loss 0.0456
Epoch 104   Loss 0.0456
Epoch 105   Loss 0.0456
Epoch 106   Loss 0.0456
Epoch 107   Loss 0.0456
Epoch 108   Loss 0.0456
Epoch 109   Loss 0.0456
Epoch 110   Loss 0.0455
Epoch 111   Loss 0.0455
Epoch 112   Loss 0.0455
```

Training stops here because performance has stopped improving. Worth mentioning: in some cases such as this dataset, 'auto' can actually get stuck in a local minimum well before the network is fully trained. Be mindful of this when using 'auto' stop condition.

You might also notice that in this example, the validation loss is listed for every single epoch instead of every 5. This is simply because the verbose `-v` flag was provided.

Splitting data into train/validation/test sets:

`--set-fractions`: This flag allows the user to set the proportions of data that will be a part of the training set, validation set, and test set. By default, the split is 70:15:15. This flag takes three input arguments, between 0 and 1, that must sum to 1.

```
parrot-train data/seq_regress_dataset.tsv setfractions_network.pt --datatype sequence␣
↪-c 1 -e 200 --set-fractions 0.5 0.45 0.05
```

Notice that the output predictions file from this command has fewer datapoints because of the reduced test set. Most likely, the accuracy will be a little worse then the default proportions because the training set is also smaller.

`--split`: In some cases, users might want precise control over over the training, validation and test set splits of their input data. This flag allows the user to manually specify which subset each sample in their dataset will be assigned. This flag requires an argument that is a path to a *split_file*, which specifically allocates sequences in *datafile* to the different datasets. An example *split_file* is provided in the **/data** folder for reference.

```
parrot-train data/seq_regress_dataset.tsv manualsplit_network.pt --datatype sequence -
↪c 1 -e 200 --split data/split_file.tsv
```

This can especially be useful if you wish to perform k-fold cross-validation on your dataset, as you can prepare k different split_files that each specify a particular 1/kth of your dataset into the test set.

`--save-splits`: Sometimes, a random partition into training/val/test sets is acceptable, but it is helpful to know for replicability where each sample was assigned. For example, if you are comparing multiple types of machine learning networks, it is best practice to use the same training set for each network. Including this flag causes an additional text file (suffix: "_split_file.txt") to be saved to the output directory. This file is formatted in the same way as a *split_file* for using with the `--split` flag.

Amino acid -> vector encoding:

"Encoding" in the context of PARROT refers to the process of converting a sequence of amino acids into computer-readable numeric vectors. By default, PARROT utilizes *one-hot* encoding, which represents each amino acid as a vector with 19 zeros and a single 1, where the position of the 1 determines its identity. However, users can change how amino acids are encoded using the `--encode` flag.

In addition to one-hot encoding, encoding using biophysical scales (vector of properties like charge, hydrophobicity, molecular weight, etc.) is also hard-coded into PARROT. Machine learning using biophysical encoding and can be carried out by providing 'biophysics' after this flag.

```
parrot-train data/seq_regress_dataset.tsv biophysics_network.pt -d sequence -c 1 -nl␣
↪2 -hs 10 -e 200 --encode biophysics
```

More powerfully, PARROT also allows the user to manually specify their own encoding scheme, if they desire. An example encoding file can be found in the **/data** folder. In this case, provide the path to this encoding file following the flag.

```
parrot-train data/seq_regress_dataset.tsv userencode_network.pt -d sequence -c 1 -nl␣
↪2 -hs 10 -e 200 --encode data/encoding_example.txt
```

With the `--encode` flag and a user-provided file, PARROT is even flexible enough to work on nucleotide sequences! To illustrate this, we've included the file "nucleotide_encoding.txt" which can be passed in via this flag to one-hot encode nucleotide sequences. We've also included an example sequence regression dataset (melting temperature prediction) with nucleotide sequences: "nucleotide_dataset.tsv".

```
parrot-train data/nucleotide_dataset.txt nucleotide_network.pt -d sequence -c 1 -nl 2␣
↪-hs 10 -e 200 --encode data/nucleotide_encoding.txt
```

Probabilistic classification with –probabilistic-classification:

The standard behavior of "classification" tasks in PARROT is to make predictions of discrete class labels. In reality though, this sort of behavior does not provide any information on the certainty behind these prediction. For example, in a two class problem (classes 0 and 1), if sequence A is deemed to be class 0 with 98% confidence, and sequence B is deemed class 0 with 51% confidence, both of these sequences will appear in the output prediction file as class 0. In some instances, it is useful to provide users a measure of confidence for each of the class predictions that PARROT makes. This can be accomplished with the `--probabilistic-classification` flag.

Using this flag is easy and can be used with `parrot-train`, `parrot-optimize` and `parrot-predict`. For the first two commands, this flag changes how predictions on the test set are output in the "_predictions.tsv" file and changes the figures and performance stats that are output (if specified). For the predict command, it changes how the predictions are outputed. If this flag is combined with `--include-figs`, it also changes the figure and metrics that are produced for evaluating performance on the test set (see `parrot-train` documentation page for more details). Conveniently, this flag can be used in `parrot-predict` even if it was not specified during training. As an example, here is the same sequence 3-class classification network making predictions with and without the `--probabilistic-classification` flag (default layers and hidden vector size):

```
parrot-predict data/seqfile.txt prob_example.pt discrete.txt -d sequence -c 3
```

Output:

```
a1 EADDGLYWQQN 2
b2 RRLKHEEDSTSTSTSTSTQ 0
c3 YYYGGAFAFAGRM 2
d4 GGIL 2
e5 GREPCCMLLYILILAAAQRDESSSSST 2
f6 PGDEADLGHRSLVWADD 2
```

```
parrot-predict data/seqfile.txt prob_example.pt probabilistic.txt -d sequence -c 3 --
↪probabilistic-classification
```

Output:

```
a1 EADDGLYWQQN 0.0527 0.1081 0.8392
b2 RRLKHEEDSTSTSTSTSTQ 0.9819 0.0034 0.0148
c3 YYYGGAFAFAGRM 0.0742 0.0098 0.916
d4 GGIL 0.1509 0.0596 0.7894
e5 GREPCCMLLYILILAAAQRDESSSSST 0.0465 0.0118 0.9418
f6 PGDEADLGHRSLVWADD 0.0645 0.2576 0.678
```

The three numbers following each sequence represent the probability that the sequence belongs to each of the three classes. Notice the numbers in each row sum to 1.

Currently, probabilistic classification is only implemented for *sequence classification* problems. The same principles would work for *residue classification*, however, we have not thought of a convenient way of representing the information in the output files (each sequence has num_classes x seq_len values).

7.2 Hyperparameter tuning with `parrot-optimize`:

`parrot-optimize` will train a network like `parrot-train`, however this command does not require the user to specify hyperparameters. Instead, it relies upon Bayesian Optimization to automatically select hyperparameters. Although Bayesian Optimization is much more efficient than grid search optimization, it still requires many iterations to converge upon the best hyperparameters. Additionally, this command relies upon 5-fold cross validation for each set of hyperparameters to achieve an accurate estimate of network performance. All together, this means that `parrot-optimize` can take over 100x longer to run than `parrot-train`. It is strongly recommended to only run this command on a machine with a GPU.

Nonetheless, usage for `parrot-optimize` is remarkably similar to `parrot-train`, since many of the flags are identical. As an example, let's run the command on a residue regression dataset:

```
parrot-optimize data/res_class_dataset.tsv optimize_example.pt -d residues -c 3 -e␣
↪200 --max-iter 20 -b 32 --verbose
```

Notice how we do not need to specify number of layers, hidden vector size, or learning rate as these are the parameters we are optimizing. Perhaps the most important consideration is the number of epochs. Running the optimization procedure with a large number of epochs is more likely to identify the best performing hyperparameters, however more epochs also means significantly longer run time. **IMPORTANT: I only used 20 iterations and 150 epochs here to speed up the example but it is HIGHLY recommended to use at least the default iterations for normal usage.** It is recommended to play around with your data using `parrot-train` with a few different parameters and visualizing the training and validation loss per epoch in order to pick the optimal number of epochs for training. Ideally, you should set the number of epochs to be around the point where validation accuracy tends to plateau during training.

Let's break down what is output to console during the optimization procedure:

```
PARROT with hyperparameter optimization
---------------------------------------
Train on:   cuda
Datatype:   residues
ML Task:    classification
Batch size: 32
Number of epochs:   200
```

(continues on next page)

(continued from previous page)

```
Number of optimization iterations:  20


Initial search results:
lr  nl  hs  output
0.00100  1  20  11.6680
0.00100  2  20  11.2927
0.00100  3  20  11.0651
0.00100  4  20  10.9217
0.00100  5  20  11.2689
0.01000  2  20  10.7816
0.00050  2  20  11.6328
0.00010  2  20  13.6755
0.00001  2  20  32.7119
0.00100  2   5  11.2988
0.00100  2  15  11.1669
0.00100  2  35  11.2267
0.00100  2  50  11.0833
Noise estimate: 0.7594081234203327
```

The first chunk of text details the network performance (average of 5 data folds) during the initial stage of hyperparameter optimization. This stage is used to gather an estimate of the noise (standard deviation across cross-val folds) for future optimization. The hyperparameters used in the initial search stage are hard-coded into the optimization procedure.

```
Primary optimization:
--------------------

Learning rate   |   n_layers   |   hidden vector size |  avg CV loss
======================================================================
  0.010000  |      3       |         20           |    10.593
  0.005001  |      3       |         19           |    10.820
  0.010000  |      4       |         21           |    10.715
  0.005513  |      3       |         21           |    10.852
  0.000744  |      4       |         21           |    11.113
  0.004678  |      5       |         22           |    10.847
  0.008415  |      4       |         22           |    10.550
  0.000954  |      4       |         23           |    11.024
  0.010000  |      3       |         23           |    10.597
  0.010000  |      4       |         24           |    10.559
  0.002181  |      3       |         24           |    10.757
  0.000709  |      4       |         25           |    11.065
  0.001744  |      5       |         24           |    11.281
  0.010000  |      3       |         25           |    10.707
  0.010000  |      2       |         22           |    10.869
  0.010000  |      2       |         24           |    10.758
  0.000822  |      2       |         25           |    11.275
  0.000859  |      2       |         23           |    11.100
  0.010000  |      5       |         26           |    10.817
  0.010000  |      4       |         30           |    10.774

The optimal hyperparameters are:
lr = 0.00841
nl = 4
hs = 22
```

This long block of text is the main process of optimization. The algorithm automatically selects the learning rate, number of layers and hidden vector size for each iteration. Finally, after the algorithm runs for 20 iterations (default: 50 iterations), the optimal hyperparameters are determined. These hyperparameters are also saved to a text file called 'optimal_hyperparams.txt' in the output directory. You might notice that the optimization procedure doesn't appear to sample the entire hyperparameter space, but this is due to the fact that we specified to use fewer iterations than normally recommended.

```
Training with optimal hyperparams:
Epoch 0 Loss 31.7953
Epoch 1 Loss 30.4627
Epoch 2 Loss 22.8318
Epoch 3 Loss 26.4293
Epoch 4 Loss 17.9814
Epoch 5 Loss 15.7970
Epoch 6 Loss 15.0506
Epoch 7 Loss 13.6761
Epoch 8 Loss 13.8338
Epoch 9 Loss 14.3309
Epoch 10    Loss 13.1378
...
Epoch 396   Loss 40.0893
Epoch 397   Loss 40.9645
Epoch 398   Loss 41.5348
Epoch 399   Loss 41.8932

Test Loss: 11.1555
```

Lastly, a network is trained on all the training data using the optimal hyperparameters and tested on the held-out test set. The output produced is analogous to `parrot-train`.

7.3 Integrating trained PARROT networks into Python workflows:

We added the option for users to create a predictor object in Python using their trained PARROT network. This option is built-in to the file "py_predictor.py" that is installed with PARROT. Importing PARROT within Python is simple:

```
>>> from parrot import py_predictor as ppp
```

To use a saved network, you need to create a Predictor() object. Initializing this object only requires the path to the saved network weights and specification of whether this network is for sequence or residue prediction.

```
>>> my_predictor = ppp.Predictor('/path/to/network.pt', dtype='sequence')
```

Now we're ready to make predictions! Once a network is loaded, the time to make predictions is negligible, so your predictor can be applied to as many sequences as you want. Just feed in amino acid sequences to the predict() function one at a time and predicted values will be output.

```
>>> value = my_predictor.predict('MYTESTAMINACIDSEQ')
```

Currently, this Python usage is only implemented for networks that were created using standard, one-hot amino acid encoding. In the future, we may add the option to feed in a particular encoding file so that all trained networks can be used in this manner. If this is a feature you'd be interested in, let us know and we can prioritize adding it!

CHAPTER 8

Evaluating a Network with Cross-Validation:

This page walks-through a full example of using 10-fold cross-validation to validate that PARROT is training accurate and generalizeable networks. For the purposes of this example, we will use the "seq_regress_dataset.tsv" dataset found in the **/data** folder. For the purposes of this example, I'm going to be saving the networks and other output file to a folder named "/output".

parrot-cvsplit

First, let's generate the 10 different split-files using `parrot-cvsplit`. Each of these split-files will specify a different 1/10th of the dataset to be the held-out test set. The remaining 9/10ths will be partitioned randomly into training and validation sets, 80:20.

```
parrot-cvsplit data/seq_regress_dataset.tsv output/cv_example_splits -k 10 -t 0.8
```

This should generate 10 files in /output named cv_example_splits_cv[0-9].txt

10-fold CV training

Next, we want to iteratively train PARROT networks for each of these cross-validation folds using `parrot-train`. We will manually specify which samples should belong in the training/val/test sets by using the `--split` flag in conjunction with the split-files we just generated. For this example, we are going to use the hyperparameters: `-nl 2 -hs 15 -lr 0.001 -b 16` and train for 250 epochs (these are relatively arbitrary decisions). It's important here to save the output prediction files under different names, so that we can go back and analyze all of them combined at the end of network training.

If you like, you can also use the `--include-figs` flag to assess how each of these networks perform individually. For this example, I did not include this flag because I will do analysis at the end using some external code.

```
parrot-train datasets/seq_regress_dataset.tsv cv_test/network0.pt -d sequence -c 1 -
↪nl 2 -hs 15 -lr 0.001 -b 16 -e 250 --split cv_test/cv_example_splits_cv0.txt

PARROT with user-specified parameters
-------------------------------------
Validation set loss per epoch:
Epoch 0 Loss 3.4118
Epoch 5 Loss 0.7524
```

(continues on next page)

(continued from previous page)

```
Epoch 10    Loss 0.7530
.
.
.
Epoch 235   Loss 0.1220
Epoch 240   Loss 0.1229
Epoch 245   Loss 0.1248

Test Loss: 0.0335
```

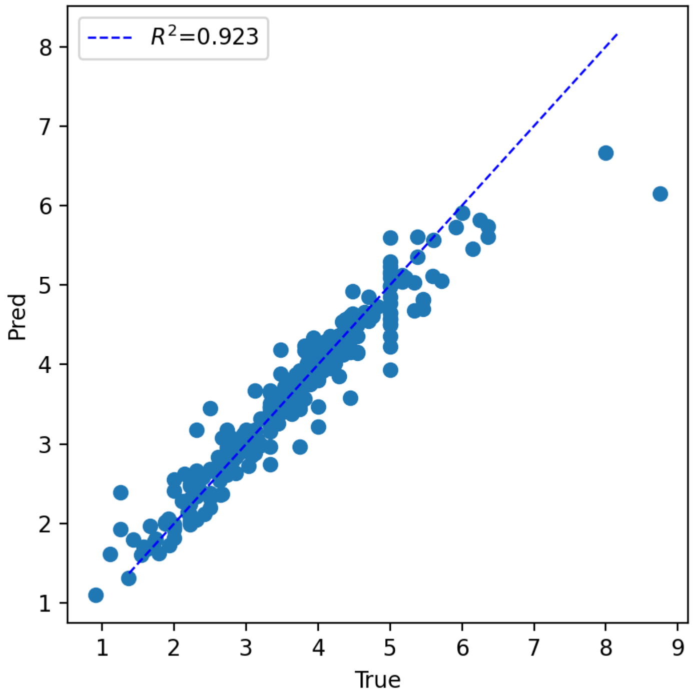
Now repeat this command 9 more times, for each of the cross-validation folds. I used the `--silent` flag to prevent additional output to terminal, but this is totally optional.

```
parrot-train datasets/seq_regress_dataset.tsv cv_test/network1.pt -d sequence -c 1 -
↪nl 2 -hs 15 -lr 0.001 -b 16 -e 250 --split cv_test/cv_example_splits_cv1.txt --
↪silent
parrot-train datasets/seq_regress_dataset.tsv cv_test/network2.pt -d sequence -c 1 -
↪nl 2 -hs 15 -lr 0.001 -b 16 -e 250 --split cv_test/cv_example_splits_cv2.txt --
↪silent
.
.
.
parrot-train datasets/seq_regress_dataset.tsv cv_test/network9.pt -d sequence -c 1 -
↪nl 2 -hs 15 -lr 0.001 -b 16 -e 250 --split cv_test/cv_example_splits_cv9.txt --
↪silent
```

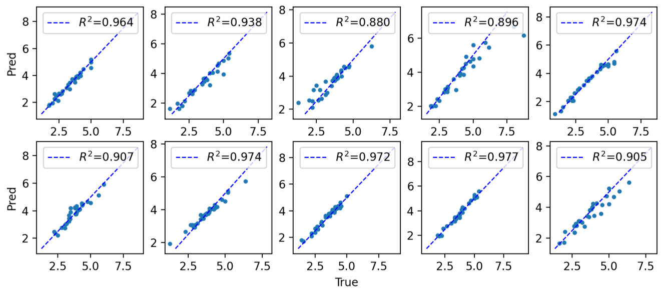
All of this could also be accomplished using a wrapper shell script. In fact, for larger datasets this is recommended so that you don't have sit around and wait for the network to train each fold.

Analyze CV predictions

After running all of this, you should have 10 files with test set predictions: "network[0-9]_predictions.tsv". First, we will evaluate (using Pearson's R square)each set of predictions separately and see how the networks perform on average and how much variance there is between different networks. In published work that uses cross-validation, it's typical to report this average performance (by some metric) across CV folds and the variance between folds.



Since all of these networks perform well, and there is low variance between predictions, we can also combine each of these prediction files and see how the cumulative predictions fare:



There's a couple of outlier sequences, but overall it looks like our networks did great!

Train final network

Finally, now that we have a good estimate of how reliable our networks' predictions are, we can create a new network using *all* of our data (and training for longer for good measure!). To do this, we simply run `parrot-train` again while specifying a test set size of ~0 using the `--set-fractions` flag (we will use 0.01, since using 0 throws an error). Importantly, validation set CANNOT be zero, as this subset is critical for making sure we do not overfit our data.

```
parrot-train datasets/seq_regress_dataset.tsv cv_test/final_cvnetwork.pt -d sequence -
↪c 1 -nl 2 -hs 15 -lr 0.001 -b 16 -e 350 --set-fractions 0.79 0.2 0.01
```

Now if we want, we can use final_cvnetwork.pt to predict unlabeled sequences with `parrot-predict`.

CHAPTER 9

Machine Learning Resources:

To many of those within the biological sciences that lack a computational or mathematical background, "machine learning" exists as a confusing and daunting topic. While some of the specific concepts in machine learning can certainly be complex and involved, on a practical level, the underlying principles of machine learning are rather intuitive and approachable. One of the motivating forces behind PARROT is to bridge this divide between complexity and accessibility. With PARROT, our lab aims to provide users with a robust machine learning framework that is easily-useable, even for those with little-to-no exposure to machine learning.

This document provides a high-level background on some of the pertinent machine learning concepts present in PARROT. Additionally, I hope to point out common pitfalls to avoid when using PARROT, or any other machine learning tool. Wherever possible, relevant papers, articles, and blog posts will be linked in order to provide a more thorough description of these concepts.

9.1 First things first, what is 'Machine Learning'?

Broadly speaking, machine learning describes a class of computer algorithms that improve automatically through experience. While the idea of machine learning has been around for over 50 years, modern advances in computing hardware and machine learning algorithms, along with the universal growth of 'big data', has caused the field to gain widespread popularity. Today, ML has been adopted by a whole host of applications, ranging from computer vision, language processing, advertising, banking and many many more. In particular, ML is becoming increasingly common within the biological research community in coordination with the massive increase of raw data from high throughput sequencing, proteomics, metabolomics, and other data generating techniques. Although their effectiveness is not unlimited, ML approaches are capable of identifying patterns in data that would otherwise go unnoticed.

Helpful resources:

- What is machine learning
- A gentle introduction to machine learning concepts
- Deep Learning for Biology

9.2 When can I use machine learning for my research?

Machine learning approaches like PARROT are great for **data exploration** and **hypothesis generation**. When facing a large set of biological data, ML can help you identify patterns in your data and allow you to design testable predictions based on those patterns. However, it is important to note that ML generally cannot validate a result on its own, and is most effective when combined with follow-up experimental validation.

There are two primary types of problems that machine learning is designed to address: **classification** and **regression**. As the name implies, classification is the process of assigning new datapoints to particular, discrete classes. For protein data, questions involving things like cellular localization, presence of PTMs, presence of a molecular interaction, etc., can be framed as classification problems. In contrast, regression problems involve assigning each data point a continuous real number value. For example, proteins can be assigned values corresponding to their expression levels, disorder, binding affinities, etc.

Helpful resources:

- Regression vs Classification in machine learning

9.3 What is a recurrent neural network (RNN)?

A recurrent neural network is a particular machine learning framework specifically designed for answering questions involving **sequential data**. RNNs were originally designed for tasks in the field of language processing (since language is essentially just a sequence of words), but in recent years they have been applied more broadly in the biological sciences. In particular, RNNs have seen promising use in analyzing protein sequences.

Part of the rationale for using RNNs for PARROT is that they are capable of taking in variable length sequences as input. Traditional neural network architectures require a fixed-length input, so for this proteins have to be split into fixed-sized fragments by running a sliding window across the entire sequence. From a practical standpoint, this kind of approach can be quite effective. However, it introduces extra hyperparameters (window size and stride length) that need to be tuned, and is not always optimal for identifying longer-range effects. In contrast, RNNs do not require sequences to be split or padded before being run through the network. More details on the architecture of RNNs can be found at the links below.

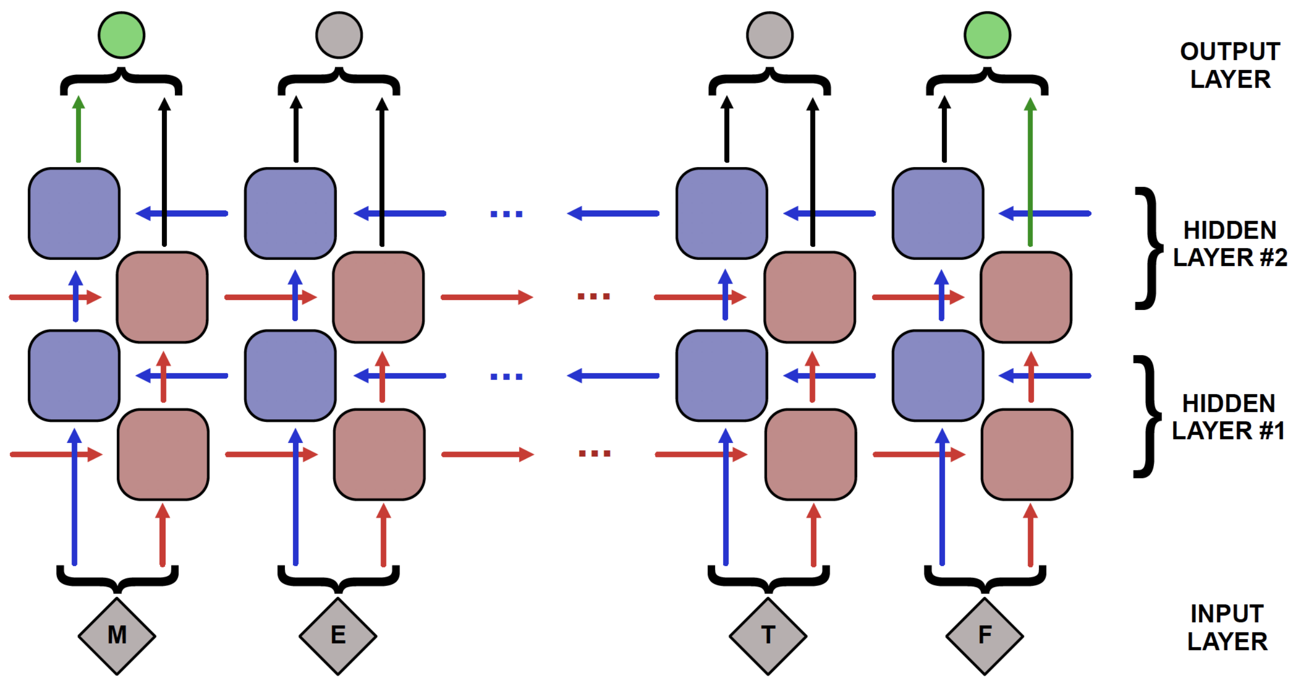
There are a few variations of RNN architecture that PARROT utilizes to achieve optimal performance. The first is the *bidirectional recurrent neural network*, or BRNN. Standard RNNs process sequence information one input at a time in the forward direction. So for protein sequences, this would mean looking at the sequence on amino acid at a time from the N-terminus to the C-terminus. BRNNs build on this architecture by also having neural network layers that process information in the reverse direction, then aggregate the information from the forward and reverse layers. PARROT implements a BRNN because the convention of writing proteins N-to-C is relatively arbitrary, and a bidirectional approach enables the network to capture more relevant information from the primary sequence. The second variation on RNNs used by PARROT is a Long Short Term Memory (LSTM) architecture. LSTMs fix some of the issues present in standard RNNs and improve the network's ability to learn long-range effects present in the data.

Helpful resources:

- Recurrent Neural Networks cheatsheet
- Understanding LSTM Networks

9.4 What are the hyperparameters of the networks used by PARROT? What should I set them as?

'Hyperparameters' in machine learning are parameters governing the architecture or learning ability of a neural network that can be tuned to impact overall performance. The networks used in PARROT have a few hyperparameters that the user can specify, or can be optimized automatically. These hyperparameters are:



Number of hidden layers ('–num-layers' / '-nl'): In a recurrent neural network, each layer constitutes a set of nodes to propogate information from one end of the sequence to the other. The most simple kind of RNN has only a single layer that processes input and produces output. However, an RNN can consist of multiple layers of sequence-spanning nodes. In this case, the first layer still processes input as before, except now this layer's output is passed as input for the next layer. For bidirectional RNNs, each layer specified by this parameter consists of both a forward and backwards layer, so specifying '–num-layers 2' will create a network with 2 forward and 2 backwards layers.

An RNN with many hidden layers is said to be "deep". In general, a deeper network is capable of identifying more complex patterns since there are more degrees of freedom within the network. However a deeper network also has the drawbacks of taking significantly longer to train and having a greater likelihood of overfitting the training data. Choosing the optimal depth of a network is highly dataset-dependent and a non-trivial problem in the field of machine learning. Ideally, a network will be as simple ("shallow") as possible while also performing well on the task at hand. In practice, networks with 1-4 layers tend to perform well on most tasks and improvements for using deeper networks is small, but PARROT provides the user with the option. PARROT also has a built-in optimization procedure for determining the depth of the network given the data, which can be useful for extended analyses of a dataset.

Hidden vector size ('–hidden-size' / '-hs'): A hidden vector in an RNN refers to the "packet" of information that is transmitted from one node to the next. If the hidden vector has a size of one, then only a single number is transmitted, if it has a size of 5 then 5 numbers are transmitted. Vectors of this size pass information both "laterally" across a layer as well as to the next deeper layer in the network.

The pros and cons of using a larger hidden vector size are similar to the number of layers hyperparameter, though the effects are less dramatic. Using larger vectors can be useful for more complex tasks, at the expense of computational speed and overfitting. As with number of layers, hidden vector size is also a hyperparameter that can be optimized for a dataset by PARROT.

Learning rate ('–learning-rate' / '-lr'): In machine learning, learning rate is a proportionality constant between 0 and 1 that affects the extent to which the weights in the network are updated after every round of training. A learning rate of 1 would cause very large updates of the weights, and a learning rate of 0 would cause the weights to remain constant throughout training. Large learning rates also tend to cause the network to train more quickly. A typical learning rate for machine learning is either 0.001 or 0.0001 as these tend to produce steady and gradual learning, though the optimal learning rate for a task is dataset-dependent. If the learning rate is too large, the network weights will experience large fluctuations and may not find the optimal values. In contrast, if the learning rate is too low, the network weights can get stuck in local minima and fail to find the optimal values. In practice, many machine learning applications rely upon algorithms called optimizers (PARROT uses the Adam optimizer) that adjust learning rate over the course of training in order to achieve optimal network performance. So for PARROT, the user-specified '–learning-rate' hyperparameter sets the *initial* learning rate, though this can also be automatically selected by running parrot-optimize.

Batch size ('–batch' / '-b'): Although most depictions of machine learning show a network processing one piece of data at a time, this is not very computationally efficient nor is it the most effective way of training a network. Rather, data is often grouped together into "batches" that are run through the network simulataneously. It is faster for computers to process data in these larger groups, especially when the computer has GPUs available for training. Running data together in batches also leads to more "averaging out" of errors during training, which can make training occur more smoothly.

Selecting the right batch size largely depends on the overall size of the dataset. It's not typically recommended to use a batch size larger than 256. Typically, a batch size that is around 1-10% of your overall dataset works well. Batch sizes slightly smaller or larger than this do not really effect overall performance, so it is not the most crucial hyperparameter to tune. A good default value for batch size is 32.

Number of epochs ('–epochs' / '-e'): In machine learning, an epoch is a round of training. Each epoch the network will "see" each item in the training set once. As one might expect, a larger number of epochs means that the network will train for longer and will lead to better performance up to a point. Too many training epochs will eventually cause the network to overfit on the training data, which will hurt network performance on non-training data. However, PARROT has features implemented that prevent this kind overfitting, so specifying a large number of training epochs should not hurt overall performance. For more details, read the over- and under-fitting section below.

Helpful resources:

- Reducing Loss and Learning Rate
- Adam optimizer
- Batch size
- Using learning curves to determine number of epochs

9.5 What is 'encoding'?

Encoding is the process of converting input data into a format that neural networks can read. For PARROT, that means converting protein sequences into a numerical representation. There are several possible ways that a protein sequence can be encoded. The most basic is one-hot encoding, the default for PARROT, which represents each amino acid as a length 20 vector, where 19 positions contain the value '0', and 1 position contains '1' depending on the identity of the amino acid. Other encoding schemes aim to represent more similar amino acids, like Glu and Asp, as having more similar encoding vectors than other unrelated amino acids. For example, one can encode each amino acid on the basis of their biophysical properties like charge, molecular weight, hydrophobicity, etc. While naturally one might assume that this kind of biologically relevant encoding scheme would be more effective for machine learning tasks, that is actually not the case. The paper referenced below showed that for many tasks, one-hot encoding actually performs as good as or better than biophysical scale encoding. The reasons for this are not entirely clear, but it illustrates the potent ability of ML to identify patterns in data.

Helpful resources:

- Raimondi et al

9.6 What are over-fitting and under-fitting?

Even when working with a large dataset, your dataset will never be completely representative of the entire space of possible data. The whole purpose of machine learning is to learn patterns from your labeled dataset (i.e. data where the underlying values are known) and *extend* it onto new, unlabeled data (where the underlying values are not known). As one might imagine, the ability of a machine learning network to extract these patterns heavily depends on the characteristics of this labeled data. One of the most common issues encountered in machine learning is over-fitting, which is when your network is trained such that it can perform well on the dataset it learned from, but its performance is not generalizable to outside data. Over-fitting can be the result of overtraining, or due to how the dataset is structured. The following section describes in more detail how one should go about setting up their dataset to avoid overfitting, and here I will briefly describe the techniques that PARROT employs to prevent overtraining.

As ML networks train on a dataset, they become better and better at predicting the data they are seeing. For a sufficiently complex network, after infinite training epochs the network will achieve 100% accuracy on the training dataset. However, for practical purposes this is not particularly helpful, since the "ground truth" of the training dataset is already known. Rather, we are interested in the performance of the network on unseen data. Many machine learninig approaches, including PARROT, approximate this unseen data by partitioning and setting aside a small chunk (often 10-20%) of the data and designating this as the **validation set**. This set of data is not used at all for the actual training of the network. Instead, after every epoch, the performance of the network on the validation set is evaluated. If training for an infinite number of epochs, the performance on the training set (AKA "training loss") will continue to improve, but after a while the performance of the validation set (AKA "validation loss") will plateau, then eventually decline as the network begins to overfit the training set. The state of the network when the max validation set performance is achieved is ultimately what is conserved after training.

In PARROT along with other ML applications, this partitioning of the data is actually taken a step further as well. The original data is actually divided into three groups: the training set (~70%), the validation set (~15%), and the **test set** (~15%). Training procedes as described in the previous paragraph. After training, the network is evaluated one final time on the test set, which it has never seen up to this point. The purpose of this step is because since the the test set is totally unseen, it can provide a good estimate of how well the final network will perform on new, unlabeled data.

So far I have described over-fitting, but under-fitting can also be a problem in machine learning. Underfitting occurs when your network is not complex enough to capture the patterns in the data. The mathematical representation of this would be trying to fit a line to a set of points generated from a parabola–it doesn't matter how long you train, there is just a finite limit of how accurate your model will be. The solution to underfitting is to use a more complex machine learning model. For PARROT this would entail specifying a larger number of layers or larger hidden vectors within the network.

Helpful resources:

- Overfitting and underfitting

9.7 How should I set up my dataset?

Even with all of the bells and whistles associated with modern neural network architectures, **by far the most important step in any ML pipeline is the initial data processing**. As such, this should be carried out diligently, with checks along the way to make sure everything is in order. Errors in your dataset can result in networks that fail to learn the patterns in the data entirely, or more insidiously, networks that *appear* to learn, when in reality they are not. Ultimately, being careful when setting up your dataset can save a lot of time troubleshooting down the road.

There are two important considerations in ML regarding your data:

1. **Make sure you have balanced classes or an even distribution of regression values**. What this means is that if you are addressing a classification task, each of the classes should have a roughly even number of datapoints in your dataset. For example, if you have three classes in a 1000-item dataset, ideally each class would compose ~300-400 of those datapoints. Likewise, for regression tasks, ideally your datapoints will have a roughly uniform

distribution across the range of regression values. The reason why a balanced dataset is important is because inbalanced data will bias your network towards predicting the over-represented class. To illustrate this point, take the following case:

You are working with a dataset in which you have 1000 labeled protein sequences that all belong to either 'Class A' or 'Class B', as well as another group of proteins for which you would like to predict what class they belong to using ML. In the extreme case, if every sequence in your dataset belong to Class A, then your ML approach will "learn" that every single sequence should be assigned Class A. After all, the strategy of "predict everything as A" has worked well for your network during training, so this will not be a very generalizable predictor. Even if the dataset is skewed 80%-20% in favor of Class A, the network is likely to predict Class A when facing any uncertainty, since statistically this is a good strategy to minimize loss.

By default, PARROT will output warnings in cases where a dataset is obviously imbalance (both for classification and regression).

2. **Limit similarity between samples in your dataset**. The issue with having similar samples arises when the similar samples are split between the training set and the validation set or test set. If this happens, your network will have artificially inflated performance and a tendency to overfit your data.

For example, imagine you have a protein dataset where half of the proteins come from the human proteome and the other half are orthologous proteins from the mouse proteome. If the network is trained on human protein A, if it encounters mouse protein A in the validation set, it will probably have an accurate prediction since these proteins (in most cases) will have highly similar sequences and will be functional orthologs. Thus the network will be incentivized to overfit the training data, rather than develop a generalizable predictive model. Likewise, if the data is split such that human protein B is in the training set and mouse protein B is in the test set, the network will perform better on the test data and give an inflated view of how accurate the network truly is on unseen data.

Fortunately, this problem is fairly easy to correct for with protein data by removing samples that display similarity above a certain threshold. If for some reason highly similar sequences can't be removed, then care should be taken so that similar sequences are grouped together in the training set, validation set, or test set. As a basic check, PARROT will warn users if there are duplicate sequences in their dataset.

Helpful resources:

- Imbalanced classification

9.8 How should I tackle a huge dataset?

Although larger datasets tend to yield more accurate ML networks, it also makes training a network much more time consuming. With PARROT, trying to optimize hyperparameters an a large dataset can take an unreasonably long time. There are a few possible ways to speed up this process.

1. **Train on a computer with GPUs**. PARROT is optimized to train in parallel on machines with GPUs, so if available, this can speed up training up to 10- or 20-fold.

2. **Optimize hyperparameters on smaller, representative subset of the data**. Although ideally you would optimize hyperparameters on the entirety of the data, this is not feasible on sufficiently large dataset. Instead, you extract a subset of the data (still considering the points about dataset structure from the preceding section) on which you can tune the hyperparameters. Once the best hyperparameters are determined, you can train on the entire network.

9.9 How can I validate that my trained network is performing well?

Since machine learning approaches tend to function as "black boxes" that cryptically make predictions given a set of input data, a critical challenge for users is being able to validate that their network is making **generalizeable** and

accurate predictions. In general, there are two ways one can validate their trained networks:

1. **Validate on external, independent data**

This is the best-case scenario. In an ideal world, you should evaluate your network on data that was collected in a manner completely independent from the dataset you used to train the network, either via different techniques or different underlying sources. By comparing the predictions made by a network to this independent "ground truth", you can ensure that your network is not overfitting to particular features/biases inherent in your training dataset.

2. **Intra-dataset validation using cross-validation**

Unfortunately, in many cases there aren't any independent datasets that you can use to validate your network! The solution developed by the machine learning community in these cases is to approximate an independent dataset by "holding-out" a subset of your data during training and evaluating how your network performs on this data. As described above, PARROT employs this sort of a test set by default, and even produces an output "_predictions.tsv" file that compare the test set predictions to the true scores. Careful analysis of this file can reveal potential biases in your network.

Extending this idea of a held-out test set, a commonly-used practice called *cross-validation* enables using an entire dataset as test data for validation. The first step of the cross-validation procedure entails dividing a dataset into K (typically K=5 or K=10) even sized chunks (termed "folds"). The network is then trained K different times, with a different fold being used as the test set in each iteration while the remainder are used to train the network. At the end, the predictions for each test set (i.e., the full dataset) are combined or averaged to produce a more robust estimate of network performance. Finally, the entire undivided dataset can be used to train a final network which should have similar (or better) performance than what is estimated by the combined cross-validation test sets.

PARROT includes several tools which facilitate cross-validation training, and a comprehensive example is provided on the "Evaluating a Network with Cross-Validation" page.

NOTE: Cross-validation is also often used to select the best hyperparameters for machine learning networks (it's used by `parrot-optimize`). This is a nearly identical procedure to what's described above, but hyperparameter tuning and network validation are NOT the same thing. Estimates produced by hyperparameter tuning can be overly optimistic compared to how your network would actually perform on data it's never encountered before.

Helpful resources:

- Why to use CV in your projects
- Nested cross validation for hyperparameter tuning

9.10 What are the different performance metrics for evaluating ML networks?

PARROT outputs a number of figures and performance stats to let user's know how accurate their trained networks are on held-out test data. These performance metrics can be useful snapshots for comparing different network architectures and training strategies, and as a result, many are widely used within the field of ML. While our intention in designing PARROT was to make this output information convienent and helpful, users should still carry our more comprehensive analysis directly on the predictions file ("_predictions.tsv").

The performance metrics used for assessing classification performance and regression performance are quite different. In classification tasks, it's common to make a confusion matrix for comparing predictions to ground truth class labels. The classification metrics used in PARROT are all derived from these matrices.

The most simple of these metrics is Accuracy, which simply denotes the fraction of correctly classified samples. Accuracy is practical and easy-to-understand, but is not as useful of a metric for imbalanced datasets. For example, if your dataset is 80% Class A and 20% Class B, a network that predicts all samples to be Class A would have an accuracy of 80%, even though it completely fails to discriminate between classes. This sort of network is obviously less useful than a different network that is 80% accurate, but with a more even distribution of incorrect classifications.

F1 score and Matthews Correlation Coefficient (MCC) are two metrics that take into account the balance between false positives and false negatives. F1 score is more widely used, but MCC is generally viewed as a more comprehensive metric (see link below). Like accuracy, the ideal MCC and F1 score are '1'. Together these three metrics effectively describe how well a classifier is performing.

Training a PARROT network in probabilistic-classification mode will also produce a Receiver Operating Characteristic (ROC) curve and a Precision-Recall curve, with their corresponding areas. For these metrics, an area under the curve of '1' denotes a perfect predictor. These curves are generated by varying the prediction threshold and seeing how the relative false positive and true positive rates vary (ROC) or how precision and recall vary at these different prediction thresholds. ROC curves are more common, but for imbalanced datasets, area under the PR curve is a more useful metric.

The standard way of evaluating regression networks is to make a scatterplot comparing the correlation of predictions to ground truth values. Two ways of measuring correlation are with Pearsons correlation and Spearmans correlation. These metrics are similar, but have a few subtle differences. Pearsons correlation, or Pearsons R, directly measures linear correlation (i.e., how well does a line fit the data). Spearmans correlation measures *rank order correlation*. As a nonparametric method, Spearmans correlation can detect correlations that are monotonic, but nonlinear. Spearmans correlation is more robust to outliers, and does not make assumptions that the underlying data is normally or uniformly distributed. Like the classificaiton metrics, these two metrics are quite similar in most realistic cases.

Helpful resources:

- Advantages of MCC over F1 score and accuracy
- ROC vs PR curves
- Pearson vs Spearman correlation

9.11 How does PARROT choose the optimal hyperparameters?

As described above, there are several different RNN hyperparameters that affect network architecture and training. In general, there is not hard and fast rule for selecting what these hyperparameters should be set at, since it varies from dataset to dataset. Since using different hyperparameters can have a noticeable impact on performance, people have developed algorithms for selecting the optimal hyperparameters for a given dataset. All of these algorithms take the general form of: 1. iteratively select hyperparameters by some criteria; 2. train a network on the data using these hyperparameters; 3. evaluate the performance of this network; and 4. pick the hyperparameters that yielded the best-performing network.

The most simple optimization algorithms are *grid search* and *random search*. These are iterative approaches that sample many points in hyperparameter space either systematically or randomly, respectively. As you can imagine, searching many combinations of hyperparameters is likely to find the best-performing set. Unfortunately, these approaches can be **very** time consuming and are not often used for more complex machine learning problems.

Instead, PARROT implements a technique called *Bayesian Optimization* to select the optimal hyperparameters for a given dataset. The details of this method are more involved than I will describe here, but below are several resources that do a good job explaining the algorithm. Briefly, instead of performing an iterative search over the hyperparameter search-space, Bayesian Optimization relies upon the mathematical concept of a Gaussian process (GP). GPs can estimate the loss function across all of hyperparameter search-space. Initially, this estimate is not very accurate, but as you train and test more sets of hyperparameters, the estimate becomes more accurate. The upshot is that Bayesian Optimization can generally identify the optimal hyperparameters *in much fewer iterations than grid or random search*.

Helpful resources:

- Hyperparameter optimization
- Gaussian processes
- Bayesian optimization

CHAPTER 10

Module Documentation

10.1 brnn_architecture.py

The underlying architecture of the bidirectional LSTM network used in PARROT

Question/comments/concerns? Raise an issue on github: https://github.com/idptools/parrot

Licensed under the MIT license.

class `parrot.brnn_architecture.`**`BRNN_MtM`**(*input_size*, *hidden_size*, *num_layers*, *num_classes*, *device*)

A PyTorch many-to-many bidirectional recurrent neural network

A class containing the PyTorch implementation of a BRNN. The network consists of repeating LSTM units in the hidden layers that propogate sequence information in both the foward and reverse directions. A final fully connected layer aggregates the deepest hidden layers of both directions and produces the outputs.

"Many-to-many" refers to the fact that the network will produce outputs corresponding to every item of the input sequence. For example, an input sequence of length 10 will produce 10 sequential outputs.

Variables

- **device** (*str*) – String describing where the network is physically stored on the computer. Should be either 'cpu' or 'cuda' (GPU).
- **hidden_size** (*int*) – Size of hidden vectors in the network
- **num_layers** (*int*) – Number of hidden layers (for each direction) in the network
- **num_classes** (*int*) – Number of classes for the machine learning task. If it is a regression problem, *num_classes* should be 1. If it is a classification problem, it should be the number of classes.
- **lstm** (*PyTorch LSTM object*) – The bidirectional LSTM layer(s) of the recurrent neural network.
- **fc** (*PyTorch Linear object*) – The fully connected linear layer of the recurrent neural network. Across the length of the input sequence, this layer aggregates the output of the

LSTM nodes from the deepest forward layer and deepest reverse layer and returns the output for that residue in the sequence.

forward(*x*)

Propogate input sequences through the network to produce outputs

Parameters **x** (*3-dimensional PyTorch IntTensor*) – Input sequence to the network. Should be in the format: [batch_dim X sequence_length X input_size]

Returns Output after propogating the sequences through the network. Will be in the format: [batch_dim X sequence_length X num_classes]

Return type 3-dimensional PyTorch FloatTensor

class parrot.brnn_architecture.**BRNN_MtO**(*input_size*, *hidden_size*, *num_layers*, *num_classes*, *device*)

A PyTorch many-to-one bidirectional recurrent neural network

A class containing the PyTorch implementation of a BRNN. The network consists of repeating LSTM units in the hidden layers that propogate sequence information in both the foward and reverse directions. A final fully connected layer aggregates the deepest hidden layers of both directions and produces the output.

"Many-to-one" refers to the fact that the network will produce a single output for an entire input sequence. For example, an input sequence of length 10 will produce only one output.

Variables

- **device** (*str*) – String describing where the network is physically stored on the computer. Should be either 'cpu' or 'cuda' (GPU).
- **hidden_size** (*int*) – Size of hidden vectors in the network
- **num_layers** (*int*) – Number of hidden layers (for each direction) in the network
- **num_classes** (*int*) – Number of classes for the machine learning task. If it is a regression problem, *num_classes* should be 1. If it is a classification problem, it should be the number of classes.
- **lstm** (*PyTorch LSTM object*) – The bidirectional LSTM layer(s) of the recurrent neural network.
- **fc** (*PyTorch Linear object*) – The fully connected linear layer of the recurrent neural network. Across the length of the input sequence, this layer aggregates the output of the LSTM nodes from the deepest forward layer and deepest reverse layer and returns the output for that residue in the sequence.

forward(*x*)

Propogate input sequences through the network to produce outputs

Parameters **x** (*3-dimensional PyTorch IntTensor*) – Input sequence to the network. Should be in the format: [batch_dim X sequence_length X input_size]

Returns Output after propogating the sequences through the network. Will be in the format: [batch_dim X 1 X num_classes]

Return type 3-dimensional PyTorch FloatTensor

10.2 encode_sequence.py

File containing functions for encoding a string of amino acids into a numeric vector.

Question/comments/concerns? Raise an issue on github: https://github.com/idptools/parrot

Licensed under the MIT license.

class parrot.encode_sequence.**UserEncoder**(*encode_file*)
User-specified amino acid-to-vector encoding scheme object

Variables

- **encode_file** (*str*) – A path to a file that describes the encoding scheme
- **encode_dict** (*dict*) – A dictionary that maps each amino acid to a numeric vector
- **input_size** (*int*) – The length of the encoding vector used for each amino acid

decode(*seq_vectors*)
Converts a list of sequence vectors back to a list of protein sequences

Parameters **seq_vectors** (*list of numpy arrays*) – A list containing sequence vectors

Returns Strings of amino acid sequences

Return type list

encode(*seq*)
Convert an amino acid sequence into this encoding scheme

Parameters **seq** (*str*) – An uppercase sequence of amino acids (single letter code)

Returns a PyTorch tensor representing the encoded sequence

Return type torch.FloatTensor

parrot.encode_sequence.**biophysics**(*seq*)
Convert an amino acid sequence to a PyTorch tensor with biophysical encoding

Each amino acid is represented by a length 9 vector with each value representing a biophysical property. The nine encoded biophysical scales are Kyte-Doolittle hydrophobicity, charge, isoelectric point, molecular weight, aromaticity, h-bonding ability, side chain solvent accessible surface area, backbone SASA, and free energy of solvation. Inputing a sequence with a nono-canonical amino acid letter will cause the program to exit.

E.g. Glutamic acid (E) is: [-3.5, -1, 3.2, 147.1, 0, 1, 161.8, 68.1, -107.3]

Parameters **seq** (*str*) – An uppercase sequence of amino acids (single letter code)

Returns a PyTorch tensor representing the encoded sequence

Return type torch.FloatTensor

parrot.encode_sequence.**one_hot**(*seq*)
Convert an amino acid sequence to a PyTorch tensor of one-hot vectors

Each amino acid is represented by a length 20 vector with a single 1 and 19 0's Inputing a sequence with a nono-canonical amino acid letter will cause the program to exit.

E.g. Glutamic acid (E) is encoded: [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Parameters **seq** (*str*) – An uppercase sequence of amino acids (single letter code)

Returns a PyTorch tensor representing the encoded sequence

Return type torch.IntTensor

parrot.encode_sequence.**parse_encode_file**(*file*)
Helper function to convert an encoding file into key:value dictionary

parrot.encode_sequence.**rev_biophysics**(*seq_vectors*)
Decode a list of biophysically-encoded sequence vectors into amino acid sequences

Parameters **seq_vectors** (*list of numpy arrays*) – A list containing sequence vectors

Returns Strings of amino acid sequences

Return type list

`parrot.encode_sequence.`**`rev_one_hot`**(*seq_vectors*)

Decode a list of one-hot sequence vectors into amino acid sequences

Parameters **seq_vectors** (*list of numpy arrays*) – A list containing sequence vectors

Returns Strings of amino acid sequences

Return type list

10.3 process_input_data.py

Module with functions for processing an input datafile into a PyTorch-compatible format.

Question/comments/concerns? Raise an issue on github: https://github.com/idptools/parrot

Licensed under the MIT license.

`class` `parrot.process_input_data.`**`SequenceDataset`**(*data*, *subset=array([], dtype=float64)*, *encoding_scheme='onehot'*, *encoder=None*)

A PyTorch-compatible dataset containing sequences and values

Stores a collection of sequences as tensors along with their corresponding target values. This class is designed to be provided to PyTorch Dataloaders.

Variables

- **`data`** (*`list of lists`*) – Each inner list represents a single sequence in the dataset and should have the format: [seqID, sequence, value(s)]
- **`encoding_scheme`** (*`str`*) – Description of how an amino acid sequence should be encoded as a numeric vector. Providing a string other than 'onehot', 'biophysics', or 'user' will produce unintended consequences.
- **`encoder`** (*`UserEncoder object, optional`*) – If encoding_scheme is 'user', encoder should be a UserEncoder object that can convert amino acid sequences to numeric vectors. If encoding_scheme is not 'user', use None.

`parrot.process_input_data.`**`parse_file`**(*tsvfile*, *datatype*, *problem_type*, *num_classes*, *excludeSeqID=False*, *ignoreWarnings=False*)

Parse a datafile containing sequences and values.

Each line of of the input tsv file contains a sequence of amino acids, a value (or values) corresponding to that sequence, and an optional sequence ID. This file will be parsed into a more convenient list of lists.

If excludeSeqID is False, then the format of each line in the file should be: <seqID> <sequence> <value(s)>

If excludeSeqID is True, then the format of each line in the file should be: <sequence> <value(s)>

value(s) will either be a single number if *datatype* is 'sequence' or a len(sequence) series of whitespace-separated numbers if it is 'residues'.

If *problem_type* is 'regression', then each value can be any real number. But if it is 'classification' then each value should be an integer in the range [0-N] where N is the number of classes.

Parameters

- **tsvfile** (*str*) – Path to a whitespace-separated datafile

- **datatype** (*str*) – Description of the format of the values in *tsvfile*. Providing a string other than 'sequence' or 'residues' will produce unintended behavior.
- **problem_type** (*str*) – Description of the machine-learning task. Providing a string other than 'regression' or 'classification' will produce unintended behavior.
- **excludeSeqID** (*bool, optional*) – Boolean indicating whether or not each line in *tsvfile* has a sequence ID (default is False)
- **ignoreWarnings** (*bool, optional*) – If False, assess the structure and balance of the provided dataset with basic heuristics and display warnings for common issues.

Returns A list representing the entire *tsvfile*. Each inner list corresponds to a single line in the file and has the format [seqID, sequence, values].

Return type list of lists

`parrot.process_input_data.`**`read_split_file`**(*split_file*)

Read in a split_file

Parameters **split_file** (*str*) – Path to a whitespace-separated splitfile

Returns

- *numpy int array* – an array of the indices for the training set samples
- *numpy int array* – an array of the indices for the validation set samples
- *numpy int array* – an array of the indices for the testing set samples

`parrot.process_input_data.`**`read_tsv_raw`**(*tsvfile*, *delimiter=None*)

Internal function for parsing a tsv file. Ignores empty lines and allows for comment lines (lines that start with a # symbol). Does not do any other sanity checking, however. :Parameters: * **tsvfile** (*str*) – Path to a whitespace-separated datafile

- **delimiter** (*str or None*) – Allows you to define the string to split columns in the file. Default is any whitespace character. Default = None (any whitespace).

Returns Returns a list of strings where each element in the list

Return type list

`parrot.process_input_data.`**`res_class_collate`**(*batch*)

Collates sequences and their values into a batch

Transforms a collection of tuples of sequence vectors and values into a single tuple by stacking along a newly-created batch dimension. This function is specifically designed for classification problems with residue-mapped data. To account for sequences with different lengths, all sequence vectors are zero- padded to the length of the longest sequence in the batch

Parameters **batch** (*list*) – A list of tuples of the form (sequence_vector, target_value(s))

Returns a tuple with concatenated names, sequence_vectors and target_values

Return type tuple

`parrot.process_input_data.`**`res_regress_collate`**(*batch*)

Collates sequences and their values into a batch

Transforms a collection of tuples of sequence vectors and values into a single tuple by stacking along a newly-created batch dimension. This function is specifically designed for regression problems with residue-mapped data. To account for sequences with different lengths, all sequence vectors are zero- padded to the length of the longest sequence in the batch

Parameters **batch** (*list*) – A list of tuples of the form (sequence_vector, target_value(s))

Returns a tuple with concatenated names, sequence_vectors and target_values

Return type tuple

`parrot.process_input_data.`**`seq_class_collate`**(*batch*)

Collates sequences and their values into a batch

Transforms a collection of tuples of sequence vectors and values into a single tuple by stacking along a newly-created batch dimension. This function is specifically designed for classification problems with sequence-mapped data.

Parameters **batch** (*list*) – A list of tuples of the form (sequence_vector, target_value(s))

Returns a tuple with concatenated names, sequence_vectors and target_values

Return type tuple

`parrot.process_input_data.`**`seq_regress_collate`**(*batch*)

Collates sequences and their values into a batch

Transforms a collection of tuples of sequence vectors and values into a single tuple by stacking along a newly-created batch dimension. This function is specifically designed for regression problems with sequence-mapped data.

Parameters **batch** (*list*) – A list of tuples of the form (sequence_vector, target_value(s))

Returns a tuple with concatenated names, sequence_vectors and target_value

Return type tuple

`parrot.process_input_data.`**`split_data`**(*data_file*, *datatype*, *problem_type*, *num_classes*, *excludeSeqID=False*, *split_file=None*, *encoding_scheme='onehot'*, *encoder=None*, *percent_val=0.15*, *percent_test=0.15*, *ignoreWarnings=False*, *save_splits_output=None*)

Divide a datafile into training, validation, and test datasets

Takes in a datafile and specification of the data format and the machine learning problem, and returns PyTorch-compatible Dataset objects for the training, validation and test sets of the data. The user may optionally specify how the dataset should be split into these subsets, as well as how protein sequences should be encoded as numeric vectors.

Parameters

- **data_file** (*str*) – Path to the datafile containing sequences and corresponding values
- **datatype** (*str*) – Format of the values within *data_file*. Should be 'sequence' if the *data_file* contains a single value per sequence, or 'residues' if it contains a value for each residue per sequence.
- **problem_type** (*str*) – The machine learning task to be addressed. Should be either 'regression' or 'classification'.
- **excludeSeqID** (*bool, optional*) – Flag that indicates how *data_file* is formatted. If False (default), then each line in the file should begin with a column containing a sequence ID. If True, then the datafile will not have this ID column, and will begin with the protein sequence.
- **split_file** (*str, optional*) – Path to a file containing information on how to divide the data into training, validation and test datasets. Default is None, which will cause the data to be divided randomly, with proportions based on *percent_val* and *percent_test*. If *split_file* is provided it must contain 3 lines in the file, corresponding to the training, validation and test

sets. Each line should have whitespace-separated integer indices which correspond to lines in *data_file*.

- **encoding_scheme** (*str, optional*) – The method to be used for encoding protein sequences as numeric vectors. Currently 'onehot' and 'biophysics' are implemented (default is 'onehot').
- **encoder** (*UserEncoder object, optional*) – If encoding_scheme is 'user', encoder should be a UserEncoder object that can convert amino acid sequences to numeric vectors. If encoding_scheme is not 'user', use None.
- **percent_val** (*float, optional*) – If *split_file* is not provided, the fraction of the data that should be randomly assigned to the validation set. Should be in the range [0-1] (default is 0.15).
- **percent_test** (*float, optional*) – If *split_file* is not provided, the fraction of the data that should be randomly assigned to the test set. Should be in the range [0-1] (default is 0.15). The proportion of the training set will be calculated by the difference between 1 and the sum of *percent_val* and *percent_train*, so these should not sum to be greater than 1.
- **ignoreWarnings** (*bool, optional*) – If False, assess the structure and balance of the provided dataset with basic heuristics and display warnings for common issues.
- **save_splits_output** (*str, optional*) – Location where the train / val / test splits for this run should be saved

Returns

- *SequenceDataset object* – a dataset containing the training set sequences and values
- *SequenceDataset object* – a dataset containing the validation set sequences and values
- *SequenceDataset object* – a dataset containing the test set sequences and values

`parrot.process_input_data.`**`split_data_cv`**(*data_file*, *datatype*, *problem_type*, *num_classes*, *excludeSeqID=False*, *split_file=None*, *encoding_scheme='onehot'*, *encoder=None*, *percent_val=0.15*, *percent_test=0.15*, *n_folds=5*, *ignoreWarnings=False*, *save_splits_output=None*)

Divide a datafile into training, val, test and 5 cross-val datasets.

Takes in a datafile and specification of the data format and the machine learning problem, and returns PyTorch-compatible Dataset objects for the training, validation, test and cross-validation sets of the data. The user may optionally specify how the dataset should be split into these subsets, as well as how protein sequences should be encoded as numeric vectors.

Parameters

- **data_file** (*str*) – Path to the datafile containing sequences and corresponding values
- **datatype** (*str*) – Format of the values within *data_file*. Should be 'sequence' if the *data_file* contains a single value per sequence, or 'residues' if it contains a value for each residue per sequence.
- **problem_type** (*str*) – The machine learning task to be addressed. Should be either 'regression' or 'classification'.
- **excludeSeqID** (*bool, optional*) – Flag that indicates how *data_file* is formatted. If False (default), then each line in the file should begin with a column containing a sequence ID. If True, then the datafile will not have this ID column, and will begin with the protein sequence.
- **split_file** (*str, optional*) – Path to a file containing information on how to divide the data into training, validation and test datasets. Default is None, which will cause the data to be divided randomly, with proportions based on *percent_val* and *percent_test*. If *split_file* is provided it must contain 3 lines in the file, corresponding to the training, validation and test

sets. Each line should have whitespace-separated integer indices which correspond to lines in *data_file*.

- **encoding_scheme** (*str, optional*) – The method to be used for encoding protein sequences as numeric vectors. Currently 'onehot' and 'biophysics' are implemented (default is 'onehot').
- **encoder** (*UserEncoder object, optional*) – If encoding_scheme is 'user', encoder should be a UserEncoder object that can convert amino acid sequences to numeric vectors. If encoding_scheme is not 'user', use None.
- **percent_val** (*float, optional*) – If *split_file* is not provided, the fraction of the data that should be randomly assigned to the validation set. Should be in the range [0-1] (default is 0.15).
- **percent_test** (*float, optional*) – If *split_file* is not provided, the fraction of the data that should be randomly assigned to the test set. Should be in the range [0-1] (default is 0.15). The proportion of the training set will be calculated by the difference between 1 and the sum of *percent_val* and *percent_train*, so these should not sum to be greater than 1.
- **n_folds** (*int, optional*) – Number of folds for cross-validation (default is 5).
- **ignoreWarnings** (*bool, optional*) – If False, assess the structure and balance of the provided dataset with basic heuristics and display warnings for common issues.
- **save_splits_output** (*str, optional*) – Location where the train / val / test splits for this run should be saved

Returns

- *list of tuples of SequenceDataset objects* – a list of tuples of length *n_folds*. Each tuple contains the training and validation datasets for one of the cross-val folds.
- *SequenceDataset object* – a dataset containing the training set sequences and values
- *SequenceDataset object* – a dataset containing the validation set sequences and values
- *SequenceDataset object* – a dataset containing the test set sequences and values

`parrot.process_input_data.`**`vector_split`**(*v*, *fraction*)

Split a vector randomly by a specified proportion

Randomly divide the values of a vector into two, non-overlapping smaller vectors. The proportions of the two vectors will be *fraction* and (1 - *fraction*).

Parameters

- **v** (*numpy array*) – The vector to divide
- **fraction** (*float*) – Size proportion for the returned vectors. Should be in the range [0-1].

Returns

- *numpy array* – a subset of *v* of length *fraction* * len(v) (rounding up)
- *numpy array* – a subset of *v* of length (1-*fraction*) * len(v).

10.4 train_network.py

Core training module of PARROT

Question/comments/concerns? Raise an issue on github: https://github.com/idptools/parrot

Licensed under the MIT license.

`parrot.train_network.`**`test_labeled_data`**(*network*, *test_loader*, *datatype*, *problem_type*, *weights_file*, *num_classes*, *probabilistic_classification*, *include_figs*, *device*, *output_file_prefix=''*)

Test a trained BRNN on labeled sequences

Using the saved weights of a trained network, run a set of sequences through the network and evaluate the performancd. Return the average loss per sequence and plot the results. Testing a network on previously-unseen data provides a useful estimate of how generalizeable the network's performance is.

Parameters

- **network** (*PyTorch network object*) – A BRNN network with the desired architecture
- **test_loader** (*PyTorch DataLoader object*) – A DataLoader containing the sequences and targets of the test set
- **datatype** (*str*) – The format of values in the dataset. Should be 'sequence' for datasets with a single value (or class label) per sequence, or 'residues' for datasets with values (or class labels) for every residue in a sequence.
- **problem_type** (*str*) – The machine learning task–should be either 'regression' or 'classification'.
- **weights_file** (*str*) – A path to the location of the best_performing network weights
- **num_classes** (*int*) – Number of data classes. If regression task, put 1.
- **probabilistic_classification** (*bool*) – Whether output should be binary labels, or "weights" of each label type. This field is only implemented for binary, sequence classification tasks.
- **include_figs** (*bool*) – Whether or not matplotlib figures should be generated.
- **device** (*str*) – Location of where testing will take place–should be either 'cpu' or 'cuda' (GPU). If available, training on GPU is typically much faster.
- **output_file_prefix** (*str*) – Path and filename prefix to which the test set predictions and plots will be saved.

Returns

- *float* – The average loss across the entire test set
- *list of lists* – Details of the output predictions for each of the sequences in the test set. Each inner list represents a sample in the test set, with the format: [sequence_vector, true_value, predicted_value, sequence_ID]

`parrot.train_network.`**`test_unlabeled_data`**(*network*, *sequences*, *device*, *encoding_scheme='onehot'*, *encoder=None*, *print_frequency=None*)

Test a trained BRNN on unlabeled sequences

Use a trained network to make predictions on previously-unseen data.

** Note: Unlike the previous functions, *network* here must have pre-loaded weights. **

Parameters

- **network** (*PyTorch network object*) – A BRNN network with the desired architecture and pre-loaded weights
- **sequences** (*list*) – A list of amino acid sequences to test using the network
- **device** (*str*) – Location of where testing will take place–should be either 'cpu' or 'cuda' (GPU). If available, training on GPU is typically much faster.

- **encoding_scheme** (*str, optional*) – How amino acid sequences are to be encoded as numeric vectors. Currently, 'onehot','biophysics' and 'user' are the implemented options.
- **encoder** (*UserEncoder object, optional*) – If encoding_scheme is 'user', encoder should be a UserEncoder object that can convert amino acid sequences to numeric vectors. If encoding_scheme is not 'user', use None.
- **print_frequency** (*int*) – If provided defines at what sequence interval an update is printed. Default = None.

Returns A dictionary containing predictions mapped to sequences

Return type dict

`parrot.train_network.`**`train`**(*network*, *train_loader*, *val_loader*, *datatype*, *problem_type*, *weights_file*, *stop_condition*, *device*, *learn_rate*, *n_epochs*, *verbose=False*, *silent=False*)

Train a BRNN and save the best performing network weights

Train the network on a training set, and every epoch evaluate its performance on a validation set. Save the network weights that acheive the best performance on the validation set.

User must specify the machine learning tast (*problem_type*) and the format of the data (*datatype*). Additionally, this function requires the learning rate hyperparameter and the number of epochs of training. The other hyperparameters, number of hidden layers and hidden vector size, are implictly included on the the provided network.

The user may specify if they want to train the network for a set number of epochs or until an automatic stopping condition is reached with the argument *stop_condition*. Depending on the stopping condition used, the *n_epochs* argument will have a different role.

Parameters

- **network** (*PyTorch network object*) – A BRNN network with the desired architecture
- **train_loader** (*PyTorch DataLoader object*) – A DataLoader containing the sequences and targets of the training set
- **val_loader** (*PyTorch DataLoader object*) – A DataLoader containing the sequences and targets of the validation set
- **datatype** (*str*) – The format of values in the dataset. Should be 'sequence' for datasets with a single value (or class label) per sequence, or 'residues' for datasets with values (or class labels) for every residue in a sequence.
- **problem_type** (*str*) – The machine learning task–should be either 'regression' or 'classification'.
- **weights_file** (*str*) – A path to the location where the best_performing network weights will be saved
- **stop_condition** (*str*) – Determines when to conclude network training. If 'iter', then the network will train for *n_epochs* epochs, then stop. If 'auto' then the network will train for at least *n_epochs* epochs, then begin assessing whether performance has sufficiently stagnated. If the performance plateaus for *n_epochs* consecutive epochs, then training will stop.
- **device** (*str*) – Location of where training will take place–should be either 'cpu' or 'cuda' (GPU). If available, training on GPU is typically much faster.
- **learn_rate** (*float*) – Initial learning rate of network training. The training process is controlled by the Adam optimization algorithm, so this learning rate will tend to decrease as training progresses.

- **n_epochs** (*int*) – Number of epochs to train for, or required to have stagnated performance for, depending on *stop_condition*.
- **verbose** (*bool, optional*) – If true, causes training updates to be written every epoch, rather than every 5 epochs.
- **silent** (*bool, optional*) – If true, causes not training updates to be written to standard out.

Returns

- *list* – A list of the average training set losses achieved at each epoch
- *list* – A list of the average validation set losses achieved at each epoch

10.5 bayesian_optimization.py

This file contains code for conducting Bayesian optimization.

Question/comments/concerns? Raise an issue on github: https://github.com/idptools/parrot

Licensed under the MIT license.

class parrot.bayesian_optimization.**BayesianOptimizer**(*cv_dataloaders*, *input_size*, *n_epochs*, *n_classes*, *dtype*, *weights_file*, *max_iterations*, *device*, *silent*)

A class for conducting Bayesian Optimization on a PyTorch RNN

Sets up and runs GPy Bayesian Optimization in order to choose the best- performing hyperparameters for a RNN for a given machine learning task. Iteratively change learning rate, hidden vector size, and the number of layers in the network, then train and validating using 5-fold cross validation.

Variables

- **cv_dataloaders** (*list of tuples of PyTorch DataLoader objects*) – For each of the cross-val folds, a tuple containing a training set DataLoader and a validation set DataLoader.
- **input_size** (*int*) – Length of the amino acid encoding vectors
- **n_epochs** (*int*) – Number of epochs to train for each iteration of the algorithm
- **n_classes** (*int*) – Number of classes
- **n_folds** (*int*) – Number of cross-validation folds
- **problem_type** (*str*) – 'classification' or 'regression'
- **dtype** (*str*) – 'sequence' or 'residues'
- **weights_file** (*str*) – Path to which the network weights will be saved during training
- **device** (*str*) – 'cpu' or 'cuda' depending on system hardware
- **max_iterations** (*int*) – Maximum number of iterations to perform the optimization procedure
- **silent** (*bool*) – If true, do not print updates to console
- **bds** (*list of dicts*) – GPy-compatible bounds for each of the hyperparameters to be optimized

compute_cv_loss(*hyperparameters*)
Compute the average cross-val loss for a given set of hyperparameters

Given N sets of hyperparameters, determine the average cross-validation loss for BRNNs trained with these parameters.

Parameters **hyperparameters** (*numpy float array*) – Each row corresponds to a set of hyperparameters, in the order: [log_learining_rate, n_layers, hidden_size]

Returns a Nx1 numpy array of the average cross-val loss per set of input hyperparameters

Return type numpy float array

eval_cv_brnns(*lr*, *nl*, *hs*)
Train and test a network with given parameters across all cross-val folds

Parameters

- **lr** (*float*) – Learning rate of the network
- **nl** (*int*) – Number of hidden layers (for each direction) in the network
- **hs** (*int*) – Size of hidden vectors in the network

Returns the best validation loss from each fold of cross validation

Return type numpy float array

initial_search(*x*)
Calculate loss and estimate noise for an initial set of hyperparameters

Parameters **x** (*numpy array*) – Array containing initial hyperparameters to test

Returns

- *numpy array* – Array containing the average losses of the input hyperparameters
- *float* – The standard deviation of loss across cross-val folds for the input hyperparameters; an estimation of the training noise

optimize()
Set up and run Bayesian Optimization on the BRNN using GPy

Returns the best hyperparameters are chosen by Bayesian Optimization. Returned in the order: [lr, nl, hs]

Return type list

10.6 brnn_plot.py

Plot training results for regression and classification tasks on both sequence-mapped and residue-mapped data.

Question/comments/concerns? Raise an issue on github: https://github.com/idptools/parrot

Licensed under the MIT license.

parrot.brnn_plot.**confusion_matrix**(*true_classes*, *predicted_classes*, *num_classes*, *output_file_prefix=''*)
Create a confusion matrix for a sequence classification problem

Figure is saved to file at "<output_file_prefix>_seq_CM.png".

Parameters

- **true_classes** (*list of PyTorch IntTensors*) – A list where each item is a [1 x 1] tensor with the true class label of a particular sequence
- **predicted_classes** (*list of PyTorch FloatTensors*) – A list where each item is a [1 x num_classes] tensor prediction of the class label for a particular sequence
- **num_classes** (*int*) – Number of distinct data classes
- **output_file_prefix** (*str, optional*) – File to which the plot will be saved as "<output_file_prefix>_seq_CM.png"

`parrot.brnn_plot.`**`output_predictions_to_file`**(*sequence_data*, *excludeSeqID*, *encoding_scheme*, *probabilistic_class*, *encoder=None*, *output_file_prefix=''*)

Output sequences, their true values, and their predicted values to a file

Used on the output of the test_unlabeled_data() function in the train_network module in order to detail the performance of the trained network on the test set. Produces the file "test_set_predictions.tsv" in output_dir. Each pair of lines in this tsvfile corresponds to a particular test set sequence, with the first containing the true data values, and the second line having the predicted data values.

Parameters

- **sequence_data** (*list of lists*) – Details of the output predictions for each of the sequences in the test set. Each inner list represents a sample in the test set, with the format: [sequence_vector, true_value, predicted_value, sequence_ID]
- **excludeSeqID** (*bool*) – Boolean indicating whether or not each line in *tsvfile* has a sequence ID (default is False)
- **encoding_scheme** (*str*) – Description of how an amino acid sequence should be encoded as a numeric vector. Providing a string other than 'onehot', 'biophysics', or 'user' will produce unintended consequences.
- **probabilistic_class** (*bool*) – Flag indicating if probabilistic classification was specified by the user. If True, instead of class labels, predictions will be output as probabilities of each class.
- **encoder** (*UserEncoder object, optional*) – If encoding_scheme is 'user', encoder should be a UserEncoder object that can convert amino acid sequences to numeric vectors. If encoding_scheme is not 'user', use None.
- **output_file_prefix** (*str*) – Path and filename prefix to which the test set predictions will be saved. Final file path is "<output_file_prefix>_predictions.tsv"

`parrot.brnn_plot.`**`plot_precision_recall_curve`**(*true_classes*, *predicted_class_probs*, *num_classes*, *output_file_prefix=''*)

Create an PR curve for a sequence classification problem

Figure is saved to file at "<output_file_prefix>_PR_curve.png".

Parameters

- **true_classes** (*list of PyTorch IntTensors*) – A list where each item is a [1 x 1] tensor with the true class label of a particular sequence
- **predicted_class_probs** (*list of PyTorch FloatTensors*) – A list where each item is a [1 x num_classes] tensor of the probabilities of assignment to each class
- **num_classes** (*int*) – Number of distinct data classes
- **output_file_prefix** (*str, optional*) – File to which the plot will be saved as "<output_file_prefix>_PR_curve.png"

`parrot.brnn_plot.`**`plot_roc_curve`**(*true_classes*, *predicted_class_probs*, *num_classes*, *output_file_prefix=''*)

Create an ROC curve for a sequence classification problem

Figure is saved to file at "<output_file_prefix>_ROC_curve.png".

Parameters

- **true_classes** (*list of PyTorch IntTensors*) – A list where each item is a [1 x 1] tensor with the true class label of a particular sequence
- **predicted_class_probs** (*list of PyTorch FloatTensors*) – A list where each item is a [1 x num_classes] tensor of the probabilities of assignment to each class
- **num_classes** (*int*) – Number of distinct data classes
- **output_file_prefix** (*str, optional*) – File to which the plot will be saved as "<output_file_prefix>_ROC_curve.png"

`parrot.brnn_plot.`**`res_confusion_matrix`**(*true_classes*, *predicted_classes*, *num_classes*, *output_file_prefix=''*)

Create a confusion matrix for a residue classification problem

Figure is saved to file at "<output_file_prefix>_res_CM.png".

Parameters

- **true_classes** (*list of PyTorch IntTensors*) – A list where each item is a [1 x len(sequence)] tensor with the true class label of the residues in a particular sequence
- **predicted_classes** (*list of PyTorch FloatTensors*) – A list where each item is a [1 x num_classes x len(sequence)] tensor with predictions of the class label for each residue in a particular sequence
- **num_classes** (*int*) – Number of distinct data classes
- **output_file_prefix** (*str, optional*) – File to which the plot will be saved as "<output_file_prefix>_res_CM.png"

`parrot.brnn_plot.`**`residue_regression_scatterplot`**(*true*, *predicted*, *output_file_prefix=''*)

Create a scatterplot for a residue-mapped values regression problem

Each sequence is plotted with a unique marker-color combination, up to 70 different sequences.

Figure is saved to file at "<output_file_prefix>_res_scatterplot.png".

Parameters

- **true** (*list of PyTorch FloatTensors*) – A list where each item is a [1 x len(sequence)] tensor with the true regression values of each residue in a sequence
- **predicted** (*list of PyTorch FloatTensors*) – A list where each item is a [1 x len(sequence)] tensor with the regression predictions for each residue in a sequence
- **output_file_prefix** (*str, optional*) – File to which the plot will be saved as "<output_file_prefix>_res_scatterplot.png"

`parrot.brnn_plot.`**`sequence_regression_scatterplot`**(*true*, *predicted*, *output_file_prefix=''*)

Create a scatterplot for a sequence-mapped values regression problem

Figure is saved to file at "<output_file_prefix>_seq_scatterplot.png".

Parameters

- **true** (*list of PyTorch FloatTensors*) – A list where each item is a [1 x 1] tensor with the true regression value of a particular sequence
- **predicted** (*list of PyTorch FloatTensors*) – A list where each item is a [1 x 1] tensor with the regression prediction for a particular sequence
- **output_file_prefix** (*str, optional*) – File to which the plot will be saved as "<output_file_prefix>_seq_scatterplot.png"

`parrot.brnn_plot.`**`training_loss`**`(`*train_loss*, *val_loss*, *output_file_prefix=''*`)`

Plot training and validation loss per epoch

Figure is saved to file at "<output_file_prefix>_train_val_loss.png".

Parameters

- **train_loss** (*list*) – training loss across each epoch
- **val_loss** (*list*) – validation loss across each epoch
- **output_file_prefix** (*str, optional*) – File to which the plot will be saved as "<output_file_prefix>_train_val_loss.png"

`parrot.brnn_plot.`**`write_performance_metrics`**`(`*sequence_data*, *dtype*, *problem_type*, *prob_class*, *output_file_prefix=''*`)`

Writes a short text file describing performance on a variety of metrics

Writes different output depending on whether a classification or regression task is specified. Also produces unique output if in probabilistic classification mode. File is saved to "<output_file_prefix>_performance_stats.txt".

Parameters

- **sequence_data** (*list of lists*) – Details of the output predictions for each of the sequences in the test set. Each inner list represents a sample in the test set, with the format: [sequence_vector, true_value, predicted_value, sequence_ID]
- **dtype** (*str*) – The format of values in the dataset. Should be 'sequence' for datasets with a single value (or class label) per sequence, or 'residues' for datasets with values (or class labels) for every residue in a sequence.
- **problem_type** (*str*) – The machine learning task–should be either 'regression' or 'classification'.
- **prob_class** (*bool*) – Flag indicating if probabilistic classification was specified by the user.
- **output_file_prefix** (*str*) – Path and filename prefix to which the test set predictions will be saved. Final file path is "<output_file_prefix>_performance_stats.txt"

10.7 py_predictor.py

Python module for integrating a trained network directly into a Python workflow.

Question/comments/concerns? Raise an issue on github: https://github.com/idptools/parrot

Licensed under the MIT license.

`class` `parrot.py_predictor.`**`Predictor`**`(`*saved_weights*, *dtype*`)`

Class that for integrating a trained PARROT network into a Python workflow

Usage: >>> from parrot import py_predictor >>> my_predictor = py_predictor.Predictor(</path/to/saved_network.pt>, dtype={“sequence” or “residues”})

```
>>> value = my_predictor.predict(AA_sequence)
```

*** NOTE: Assumes all sequences are composed of canonical amino acids and

that all networks were implemented using one-hot encoding.

***

Variables

- **dtype** (*str*) – Data format that the network was trained for. Either “sequence” or “residues”.
- **num_layers** (*int*) – Number of hidden layers in the trained network.
- **hidden_vector_size** (*int*) – Size of hidden vectoer in the trained network.
- **n_classes** (*int*) – Number of data classes that the network was trained for. If 1, then network is designed for regression task. If >1, then classification task with n_classes.
- **task** (*str*) – Designates if network is designed for “classification” or “regression”.
- **network** (*PyTorch object*) – Initialized PARROT network with loaded weights.

predict (*seq*)

Use the network to predict values for a single sequence of valid amino acids

Parameters **seq** (*str*) – Valid amino acid sequence

Returns Returns a 1D np.ndarray the length of the sequence where each position is the prediction at that position.

Return type np.ndarray

Python Module Index

Index