BPReveal documentation

Charles McAnany, Melanie Weilert

October 12, 2023

Contents

1 Workflow

BPReveal is an expansion of the chrombpnet-lite repository, designed to allow for more flexibility. It also incorporates a new interpretation tool, called PISA (Pairwise Interaction Shap Analysis) that lets you view a two-dimensional map of cause and effect over a region. The two main changes are the addition of multi-task models, and a generalization of the regression step. For cases where you don't have a bias track you want to regress out, here are the steps:

1. Prepare bigwig tracks and select the regions you are interested in. There are some utilities for this in BPReveal, but you are mostly on your own for this stage.
2. Prepare training data files. These hdf5 files contain the sequences and experimental profiles for all the heads in your model.
3. Train a solo model.
4. Measure the performance of your model.
5. Make predictions from the model.
6. Generate importance scores, either one-dimensionally, as is usual, or by making two-dimensional PISA plots.
7. Run MoDISco to extract motifs (using the external `tfmodisco-lite` package)
8. Use the motif mapping tools to map the discovered motifs back to the genome.

If you do have strong experimental biases, you will need to regress them out. In that case, the workflow is the following:

1. Prepare bigwig tracks and select the regions you are interested in.
2. Prepare a data file containing bias. The bias regions may be uninteresting regions in the genome, or you may train on your regions of interest but use an experimental control for the data.
3. Train a bias (AKA solo) model.
4. Train a transformation model to match the bias model to the experimental data.
5. Train a residual model to explain non-bias parts of the experimental data.
6. Measure the performance of the full model.
7. Make predictions from the full model and residual model.
8. Generate importance scores from the residual model.
9. Run MoDISco.
10. Map the discovered motifs back to the genome.

2 Philosophy

These are the guidelines for code design in BPReveal:

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
- Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter the output with extraneous information. Don't insist on interactive input.
- But do include logging messages that can be enabled for debugging.
- Explicit is better than implicit. Wherever possible, do not allow for defaults when a value is not specified. (Looking at you, MoDISco!)
- For substantial programs, prefer configuration files over a sea of command-line arguments. Use JSON for all data that aren't (1.) bed files, (2.) bigwig files, (3.) fasta files, or (4.) potentially enormous (for large datasets, prefer hdf5.).
- Flat is better than nested. Write code that has minimal architectural overhead.
- Don't be too clever. The code should use the standard idioms of the language, even if an operation could be completed in fewer characters or slightly more efficiently some other way.
- But performance matters. Write code that works fast enough that people can use it to ask new questions. Use parallelism with wild abandon.
- Errors should never pass silently.
- One line of code is ten lines of documentation. The documentation consists of in-code comments, clear specifications (this document), tutorials, and the reference publication.
- If the implementation is hard to explain, it's a bad idea.
- Use only well-established and stable dependencies. Don't require specific versions of libraries, and only require packages that are truly essential.

3 Programs

BPReveal includes the following core programs:

trainSoloModel.py Takes in a training input configuration and trains up a model to predict the the given data, with no bias correction. Saves the model to disk, along with information from the training phase.

trainTransformationModel.py Takes in a bias (i.e., solo) model and the actual experimental (i.e., biology + bias) data. Derives a relation to best fit the bias profile onto the experimental data. Saves a new model to disk, adding a simple layer or two to do the regression.

trainCombinedModel.py Takes a transformation model and experimental data and builds a model to explain the residuals. Saves both a combined model and the residual model alone to disk.

prepareTrainingData.py Takes in bed and bigwig files and a genome, and generates an hdf5-format file containing the samples used for training.

makePredictionsBed.py Takes a trained model (solo, combined, residual, or even transformation models work) and predicts over the given regions.

makePredictionsFasta.py Takes a trained model and predicts on sequences given as a fasta file.

interpretFlat.py Generates shap scores of the same type as BPNet. Hypothetical contributions for each base are written to a modisco-compatible h5.

interpretPisaBed.py Runs an all-to-all shap analysis on the given regions.

interpretPisaFasta.py Runs an all-to-all shap analysis on the given sequences.

motifAddQuantiles.py Loads the output from modiscolite and calculates cutoff values to use during motif scanning.

motifScan.py Scan the genome for patterns of contribution scores that match motifs identified by modiscolite.

And the following utilities:

metrics.py Calculates a suite of metrics about how good a model's predictions are.

predictToBigwig.py Takes the hdf5 file generated by the predict step and converts one track from it into a bigwig file.

shapToBigwig.py Converts a shap hdf5 file (from `interpretFlat.py`) into a bigwig track for visualization.

shapToNumpy.py Takes the interpretations from `interpretFlat.py` and converts them to numpy arrays that can be read in by modiscolite.

lengthCalc.py Given the parameters of a network, like input filter width, number of layers &c., determine the input width or output width.

makeLossPlots.py Once you've trained a model, you can run this on the history file to get plots of all of the components of the loss.

prepareBed.py Given a set of regions and data tracks, reject regions that have too few (or too many) reads, or that have unmappped bases in the genome.

showModel.py Make a pretty picture of your model.

motifAddQuantiles.py Takes the output from `motifScan.py` and `motifAddQuantiles.py` and adds quantile information for determining how good your motif matches were.

And the following libraries:

gaOptimize.py contains tools for evolving sequences that lead to desired profiles. It implements a genetic algorithm that supports insertions and deletions.

utils.py Contains general-use utilities and a high-performance tool to generate predictions for many sequences.

4 Setup

BPReveal requires just a few libraries, and I use conda to manage my environment. You can also install these with pip.

- `python` version 3.10 or later. Note that this codebase uses features that were introduced in version 3.10, so it will not work with 3.9 or earlier.
- `tfmodisco-lite`, only needed if you want to do motif identification.
- `pyBigWig`, for reading and writing bigwig files.
- `pysam`, to read in fasta files.
- `tensorflow` version 2.8.0 or later (This also installs numpy, scipy, and h5py).
- `tensorflow-probability`, which contains functions needed to calculate the loss.
- `pybedtools`, for reading and writing bed files.
- `matplotlib`, for rendering the loss plots by `makeLossPlots.py`.
- `tqdm`, for progress bars.

You can, of course, install any other packages you like. I'm fond of `jupyter lab` for doing analysis. I found that I also needed to install a `cudatoolkit` package to get everything compatible, but this may not be necessary on your system. You should check the TensorFlow website for instructions on how to install TensorFlow on your system.

5 Programs

In this section, I will provide detailed specifications for all of the input files, with particular emphasis on configuration JSON files.

5.1 Prepare training data: `prepareTrainingData.py`

This program reads in a genome file, a list of regions in bed format, and a set of bigwig files containing profiles that the model will predict. It generates an hdf5-format file that is used during training. If you want to train on a custom genome, or you don't have a meaningful genome for your experiment, you can still provide sequences and profiles by creating an hdf5 file in the same format as this tool generates.

5.1.1 Input specification

```
<prepare-data-configuration> ::=
   {
    "genome" : <file-name>,
    "input-length" : <integer>,
    "output-length" : <integer>,
    "max-jitter" : <integer>,
    "regions" : <file-name>,
    "output-h5" : <file-name>,
    "reverse-complement" : <boolean>,
    "heads" : [<prepare-data-heads-list>],
    <verbosity-section>
}

<boolean> ::=
    true
  | false

<prepare-data-heads-list> ::=
   <prepare-data-individual-head>
 | <prepare-data-individual-head>, <prepare-data-heads-list>


<prepare-data-individual-head> ::=
  { <revcomp-head-section>
    "bigwig-files" : [<list-of-bigwig-file-names>] }

<revcomp-head-section> ::=
    <empty>
  | "revcomp-task-order" : "auto",
  | "revcomp-task-order" : [<list-of-integers>],

<list-of-bigwig-file-names> ::=
   <file-name>
 | <file-name>, <file-name>

<verbosity-section> ::=
    "verbosity" : <verbosity-level>

<verbosity-level> ::= "DEBUG"
  | "INFO"
  | "WARN"
```

5.1.2 Parameter notes

- `genome` is the name of the fasta-format file for your organism.
- `regions` is the name of the bed file of regions you will train on. These regions must be `output-width` in length.
- The `reverse-complement` flag sets whether the data files will include reverse-complement augmentation. If this is set to `true` then you must include `revcomp-task-order` in every head section.

- The `revcomp-task-order` is a list specifying which tasks in the forward sample should map to the tasks in the reverse sample. For example, if the two tasks represent reads on the plus and minus strand, then when you create a reverse-complemented training example, the minus strand becomes the plus strand, and vice versa. So you'd set this parameter to `[1,0]` to indicate that the data for the two tasks should be swapped (in addition to reversed 5' to 3', of course). If you only have one task in a head, you should set this to `[0]`, to indicate that there is no swapping. If you have multiple tasks, say, a task for the left end of a read, one for the right end, and one for the middle, then the left and right should be swapped and the middle left alone. In this case, you'd set `revcomp-task-order` to `[1,0,2]`. If this parameter is set to `"auto"`, then it will choose `[1,0]` if there are two strands, `[0]` if there is only one strand, and it will issue an error if there are more strands than that. `"auto"` is appropriate for data like ChIP-nexus.

5.1.3 Output specification

It will generate a file that is organized as follows:

- `head_N`, where `N` is an integer from 0 to however many heads you have. It will have shape (num-regions x (output-length + 2*jitter) x num-tasks).
- (more `head_N` entries)...
- `sequence`, which is the one-hot encoded sequence for each corresponding region. It will have shape (num-regions x (input-width + 2*jitter) x 4).

5.2 Training a solo model: `trainSoloModel.py`

This program generates a neural network that maps genomic sequence to experimental readout. In cases where you don't need to regress out a bias track, this is the only training program you'll use. By and large, you can refer to the BPNet paper for details on how this program works, the only difference is in the input padding. In the original BPNet, for an output window of 1 kb, the input sequence was 1 kb long, and if a neuron needed to know about bases outside that window, it got all zeros. For image processing, this makes sense, because you can't un-crop an image. However, for DNA in a genome, you can just expand out the windows and get as much DNA sequence as you like. Therefore, BPReveal models require an input length that is larger than output length, so that the model can use DNA sequence information that is outside of its output window.

The required input files are two hdf5-format files that contain sequences and profiles. One of these is used for training, one for validation during training. See `prepareTrainingData.py` for the specification for this file.

The program will produce two outputs, one being a Keras model. This model will be used later for prediction and interpretation. The other output records the progress of the model during training, and it is read in by `makeLossPlots.py`.

5.2.1 Input specification

A solo training input configuration file is a JSON file that specifies what data should be used to train up the solo model. It shall contain exactly one settings section, one data section, one heads section, and one verbosity section.

```
<solo-training-input-configuration> ::=
   {<solo-settings-section> , <data-section>,
     <head-section>, <verbosity-section>}

<solo-settings-section> ::=
    "settings" : {<solo-settings-contents>}

<data-section> ::=
    "train-data" : <file-name>,
    "val-data" : <file-name>

<head-section> ::=
  "heads" : [<head-list>]

<head-list> ::=
    <individual-head>
  | <individual-head>, <head-list>

<individual-head> ::=
    { "num-tasks" : <integer>,
      "profile-loss-weight" : <number>,
      "counts-loss-weight" : <number>,
      "head-name" : <string>
    }


<solo-settings-contents> ::=
    {"output-prefix" : <string>,
     "epochs" : <integer>,
     "max-jitter" : <integer>,
     "early-stopping-patience" : <integer>,
     "batch-size" : <integer>,
     "learning-rate" : <number>,
     "learning-rate-plateau-patience" : <integer>,
     "architecture" : <solo-architecture-specification> }
```

```
<solo-architecture-specification> ::=
    <solo-bpnet-architecture-specification>

<solo-bpnet-architecture-specification> ::=
    {"architecture-name" : "bpnet",
    "input-length" : <integer>,
    "output-length" : <integer>,
    "model-name" : <string>,
    "model-args" : <string>,
    "filters" : <integer>,
    "layers" : <integer>,
    "input-filter-width" : <integer>,
    "output-filter-width" : <integer>}
```

5.2.2 Parameter notes

- The `profile-loss-weight` is simply a scalar that the profile loss is multiplied by. This comes in handy when you're training on two datasets with disparate coverage. Since the MNLL loss is proportional to the number of reads, a track with higher coverage will dominate the loss. Instead of calculating this a priori, I find it easiest to start training the model, look at the loss values, and then pick multipliers that will make them about even.
- The `counts-loss-weight` is similar to `profile-loss-weight`, but do keep in mind that you need to set it even when you're training a single output head, since the mean squared error value of the counts prediction tends to be minuscule compared to the MNLL loss of the profile. Again, instead of calculating it a priori, I start training with an initial guess and then refine the value later.
- `output-prefix` is the file name where you want your model saved. For example, if you are saving models in a directory called `models`, and you want the model to be called `solo`, then you'd write `"output-prefix" : "models/solo"`. In this case, you'll find the files `models/solo.model`, which is the Keras model, and `models/solo.history.json`, containing the training history.
- `early-stopping-patience` controls how long the network should wait for an improvement in the loss before quitting. I recommend a bit more than double the `learning-rate-plateau-patience`, on the order of 11.
- `batch-size` determines how many regions the network will look at simultaneously during training. It doesn't really matter, but if you make it too big your data won't fit on the GPU and if you make it too small your network will take an eternity to train. I like 64 or so.
- `learning-rate` determines how aggressive the optimizer will be as the network trains. 0.004 is a good bet. (Note that the LR will decrease during training because of the plateau patience.)
- `learning-rate-plateau-patience` controls how many epochs must pass without improvement before the optimizer decreases the learning rate. I recommend 5 or so.
- The `architecture-name` is a future-proofing argument that will determine what type of network you want. Currently, only the basic bpnet-style architecture is supported.
- `input-length` is the width of the input sequence that will be fed into the network. You can use the `lengthCalc.py` script to calculate this based on a desired profile width and architecture.
- `output-length` is the width of the predicted profile. This is usually on the order of 1000.
- `model-name` is just a string that is stored along with the model. BPReveal does not use it internally.
- `model-args` is a future-proofing argument. If there is a new feature added to a particular architecture, the `model-args` string will be passed to the architecture and the architecture may do with that string as it pleases. Currently, this serves no purpose.

- `filters` is the number of convolutional filters at each layer. The more filters you add, the more patterns the model will try to learn. Typically this is between 32 and 128, smaller for simpler tasks.
- `input-filter-width` is the size of the very first motif-scanning layer in BPNet. Lately, there's been a trend of making this small, on the order of 7.
- `output-filter-width` is the width of the very last convolution, the one that actually results in the predicted profile. This layer is placed at the very bottom of the dilated layers. I use a width of 75, but many people use smaller output widths, on the order of 25.
- `max-jitter` is the maximum allowed shifting of the regions. This random shifting is applied during training, and helps to create some variety in the counts values to prevent over-fitting. Note that you must use the same jitter you used when you created your training data file - if you want to try a different jitter, you need to re-generate your data hdf5 files.

5.3 Training a transformation model: `trainTransformationModel.py`

The transformation input file is a JSON file that names a solo model and gives the experimental data that it should be fit to. Note that it may occasionally be appropriate to chain several transformation models together. Currently, the easiest way to do this is to feed the first transformation model in as the solo model for the second transformation. A better way to do it would be to write your own custom transformation Model.

5.3.1 Input specification

```
<transformation-input-configuration> ::=
    {"settings" : <transformation-settings-section>,
        <data-section>,
        <head-section>, <verbosity-section>}

<transformation-settings-section> ::=
     {"output-prefix" : "<string>",
      "epochs" : <integer>,
      "early-stopping-patience" : <integer>,
      "batch-size" : <integer>,
      "learning-rate" : <number>,
      "learning-rate-plateau-patience" : <integer>
      "solo-model-file" : <file-name>,
      "input-length" : <integer>,
      "output-length" : <integer>,
      "max-jitter" : <integer>,
      "profile-architecture" : <transformation-architecture-specification>,
      "counts-architecture" : <transformation-architecture-specification> }

<transformation-architecture-specification> ::=
    <simple-transformation-architecture-specification>
  | "name" : "passthrough"

<simple-transformation-architecture-specification> ::=
    "name" : "simple",
    "types" : [<list-of-simple-transformation-types>]

<list-of-simple-transformation-types> ::=
    <simple-transformation-type>
  | <simple-transformation-type>, <list-of-simple-transformation-types>

<simple-transformation-type> ::=
    "linear"
  | "sigmoid"
  | "relu"
```

This code does not support using an experimental bias track as the input to the transformation - it must be a solo model that uses sequence as input.

5.3.2 Parameter notes

1. `solo-model-file` is the name of the file (or directory, since that's how keras likes to save models) that contains the solo model.

2. A `passthrough` transformation does nothing to the solo model, it doesn't regress anything.

3. A `simple` transformation applies the specified functions to the output of the solo model, and adjusts the parameters to best fit the experimental data. A linear model applies $y = mx + b$ to the solo predictions (which, remember, are in log-space), a sigmoid applies $y = m_1 * sigmoid(m_2x + b_2) + b_1$, and a relu applies $y = m_1 * relu(m_2x + b_2) + b_1$. In other words, there's a linear model both before and

after the sigmoid or relu activation. Generally, you need to use these more complex functions when the solo model is not a great fit for the experimental bias.

5.4 Combined training: `trainCombinedModel.py`

The combined model is specified by a JSON file, much like the solo configuration file.

5.4.1 Input specification

```
<combined-training-input-configuration> ::=
   {<combined-settings-section> , <data-section>,
     <combined-head-section>, <verbosity-section>}

<combined-head-section> ::=
  "heads" : [<combined-head-list>]

<combined-head-list> ::=
    <combined-individual-head>
  | <combined-individual-head>, <combined-head-list>

<combined-individual-head> ::=
    {
      "num-tasks" : <integer>,
      "profile-loss-weight" : <number>,
      "counts-loss-weight" : <number>,
      "head-name" : <string>,
      "use-bias-counts" : <boolean>
}

<combined-settings-section> ::=
    "settings" : {<combined-settings-contents>}

<combined-settings-contents> ::=
    {"output-prefix" : "<string>",
     "epochs" : <integer>,
     "early-stopping-patience" : <integer>,
     "batch-size" : <integer>,
     "learning-rate" : <number>,
     "learning-rate-plateau-patience" : <integer>,
     "transformation-model" : <transformation-combined-settings>,
     "max-jitter" : <integer>,
     "architecture" : <combined-architecture-specification> }

<transformation-combined-settings> ::=
   { "transformation-model-file" : <file-name> }

<combined-architecture-specification> ::=
    <combined-bpnet-architecture-specification>

<combined-bpnet-architecture-specification> ::=
    {"architecture-name" : "bpnet",
    "input-length" : <number>,
    "output-length" : <number>,
    "model-name" : <string>,
    "model-args" : <string>,
    "filters" : <number>,
    "layers" : <number>,
    "input-filter-width" : <number>,
    "output-filter-width" : <number>}
```

5.4.2 Parameter notes

1. `use-bias-counts` selects if you want to add the counts prediction from the transformation model, and the appropriateness of this flag will depend on the nature of your bias. If the bias is a constant

background signal, then it makes sense to subtract the bias contribution to the counts. However, if your bias is multiplied by the underlying biology, then you probably shouldn't add in the bias counts since they won't affect the actual experiment.

2. `input-length` refers to the input size of the *residual* model, not the *solo* model. The solo model, having already been created, knows its own input length. If the solo model's input length is smaller than the `input-length` setting in this config file, the sequence input to the solo model will automatically be cropped down to match.

5.5 Prediction from bed regions: `makePredictionsBed.py`

This program takes in a list of regions in bed format, extracts the corresponding sequences from a given genome file, and runs those sequences through your model.

5.5.1 Input specification

```
<prediction-input-configuration>::=
    {<prediction-settings-section>, <prediction-bed-section>,
    <verbosity-section>}

<prediction-settings-section> ::= "settings" : {
    "output-h5" : <file-name>,
    "genome" : <file-name>,
    "batch-size" : <integer>,
    "heads" : <integer>,
    "architecture" : <prediction-model-settings> }

<prediction-bed-section> ::=
    "bed-file" : <file-name>

<prediction-model-settings> ::={
    "model-file" : <file-name>,
    "input-length" : <integer>,
    "output-length" : <integer> }
```

5.5.2 Parameter notes

1. `heads` just gives the number of output heads for your model. You don't need to tell this program how many tasks there are for each head, since it just blindly sticks whatever the model outputs into the hdf5 file.

5.5.3 Output specification

This program will produce an hdf5-format file containing the predicted values. It is organized as follows:

- `chrom_names` is a list of strings that give you the meaning of each index in the `coords_chrom` dataset. This is particularly handy when you want to make a bigwig file, since you can extract a header from this data.
- `chrom_size`, the size of each chromosome in the same order as `chrom_names`. Mostly used to create bigwig headers.
- `coords_chrom` a list of integers, one for each region predicted, that gives the chromosome index (see `chrom_names`) for that region.
- `coords_start`, the start base of each predicted region.
- `coords_stop`, the end point of each predicted region.
- A subgroup for each output head of the model. The subgroups are named `head_N`, where N is 0, 1, 2, etc.
  - `logcounts`, a vector of shape (numRegions,) that gives the logcounts value for each region.
  - `logits`, the array of logit values for each track for each region. The shape is (numRegions x outputWidth x numTasks). Don't forget that you must calculate the softmax on the whole set of logits, not on each task's logits independently. (There is a `logitsToProfile` function inside `utils.py` for doing this conversion.)

5.6 Prediction from fasta: `makePredictionsFasta.py`

This is the JSON file given to the prediction from fasta script. It names the model to use and the file that contains the sequences to run predictions on. This program streams sequences in and writes them out as it makes predictions, so it can be used on very large numbers of sequences without requiring much memory.

5.6.1 Input specification

```
<prediction-fasta-input-configuration> ::=
    { <prediction-fasta-settings-section>, <prediction-fasta-input-section>,
        <verbosity-section> }

<prediction-fasta-settings-section> ::= "settings" : {
    "output-h5" : <file-name>,
    "batch-size" : <integer>,
    "heads" : <integer>,
    "architecture" : <prediction-model-settings> }

<prediction-fasta-input-section> ::=
    "fasta-file" : <file-name>
```

5.6.2 Output specification

This program will produce an hdf5-format file containing the description lines from the original fasta, as well as the predicted logcounts and logits. It is structured so:

- `descriptions`, a list of strings of length (numRegions,). Each string corresponds to one description line (i.e., a line starting with `>`).
- A subgroup for each output head of the model. The subgroups are named `head_N`, where N is 0, 1, 2, etc.
  - `logcounts`, a vector of shape (numRegions,) that gives the logcounts value for each region.
  - `logits`, the array of logit values for each track for each region. The shape is (numRegions x outputWidth x numTasks). Don't forget that you must calculate the softmax on the whole set of logits, not on each task's logits independently. (There is a `logitsToProfile` function inside `utils.py` that does this.)

5.7 Interpret flat: `interpretFlat.py`

This is the configuration file that is passed to interpretFlat.py, that lists the regions of the genome where classical importance scores should be calculated. It is structured as follows:

5.7.1 Input specification

```
<flat-interpretation-configuration> ::=
{
    <bed-or-fasta>,
    "model-file" : <file-name>,
    "input-length" : <integer>,
    "output-length" : <integer>,
    "heads" : <integer>,
    "head-id" : <integer>
    "profile-task-ids" : [<list-of-integers>],
    "profile-h5" : <file-name>,
    "counts-h5" : <file-name>,
    "num-shuffles" : <integer>,
    <verbosity-section>
}

<bed-or-fasta> ::=
    "genome" : <file-name>,
    "bed-file" : <file-name>
 |  "fasta-file" : <file-name>


<list-of-integers> ::=
    <integer>
  | <integer>, <list-of-integers>
```

5.7.2 Parameter notes

- If you specify a `genome` and `bed-file` in the configuration, then this program will read coordinates from the bed file, extract the sequences from the provided fasta, and run interpretations on those sequences. In this case, the output file will include `chrom_names`, `chrom_sizes`, `coords_start`, `coords_end` and `coords_chrom`. If, however, you specify a fasta file, then the sequences are taken directly from that file. In this case, the output hdf5 file will not contain those fields, and instead it will contain a `descriptions` dataset, which holds the description lines from the fasta.

- The `bed-file` that you give should have regions matching the model's *output* length. The regions will be automatically inflated in order to extract the input sequence from the genome. Somewhat confusingly, this means that the contribution scores will include contributions from bases that are not in your bed file. This is because the contribution scores explain how all of the input bases contribute to the output observed at the region in the bed file. However, if you specify `fasta-file`, then the sequences in that fasta must be as long as the model's *input* length. (Since we need the whole sequence that will be explained.) In this case, the contribution scores in the output will match one-to-one with the input bases.

- The `heads` parameter is the *total* number of heads that the model has, and the `head-id` parameter gives which head you want importance values calculated for.

- The `profile-task-ids` parameter lists which of the profile predictions (i.e., tasks) from the specified head you want considered. Almost always, you should include all of the profiles. For a single-task head, this would be `[0]`, and for a two-task head this would be `[0,1]`.

- The `profile-h5` and `counts-h5` file names are the output files from the algorithm.

5.7.3 Output specification

If you provide `bed-file` and `genome`, the output will be structured as follows:

- `chrom_names`, a list of strings giving the name of each chromosome. Unlike the predictions from fasta, the `coords_chrom` columns are *strings*, so the `chrom_names` field is mostly for generating bigwig headers.
- `chrom_sizes`, a list of integers giving the size of each chromosome. This is mostly here as a handy reference when you want to make a bigwig file.
- `coords_start`, the start point for each of the regions in the input bed file. This will have shape (num-regions,).
- `coords_end`, the end point for each of the regions in the bed file. This will have shape (num-regions,).
- `coords_chrom`, the chromosome on which each region is found. These are strings, and this dataset has shape (num-regions,)
- `input_seqs`, a one-hot encoded array representing the input sequences. It will have shape (num-regions x input-length x 4)
- `hyp_scores`, a table of the shap scores. It will have shape (num-regions x input-length x 4) If you want the actual contribution scores, not the hypothetical ones, multiply `hyp_scores` by `input_seqs` to zero out all purely hypothetical contribution scores.

However, if you just provide a fasta of sequences to analyze, then it will be structured so:

- `descriptions`, a list of strings that are the description lines from the input fasta file (with the leading `>` removed). This list will have shape (num-regions,)
- `input_seqs` and `hyp_scores`, with the same meaning as in the bed-and-genome based output files.

Note that while you can use `shapToNumpy.py` on either format of `interpretFlat.py` output, you cannot convert a fasta-based interpretation h5 to a bigwig, since it doesn't contain coordinate information.

5.8 Interpret PISA from bed regions: `interpretPisaBed.py`

This is the JSON file given to the PISA interpretation tool. This tool wants a bed file, but this file represents the *individual bases* that should be shapped. There is no restriction on the number of regions, nor on their length. If you are interested in the effects of a particular motif, then you'd put the region surrounding that motif in the bed file, making it as large as you want to see the interactions you're interested in.

5.8.1 Input specification

```
<pisa-configuration> ::= {
    "genome" : <file-name>,
    "bed-file" : <file-name>,
    "model-file" : <file-name>,
    "input-length" : <integer>,
    "output-length" : <integer>,
    "heads" : <integer>,
    "head-id" : <integer>,
    "task-id" : <integer>,
    "output-h5" : <integer>,
    "num-shuffles" : <integer>,
    <verbosity-section>
}
```

5.8.2 Output specification

It produces an hdf5 format which is organized as follows:

- `head-id`, an integer representing which head of the model was used to generate the data.
- `task-id`, an integer giving the task number within the specified head.
- `chrom_names`, a list of strings giving the name of each chromosome. This is used to figure out which chromosome each number in `coords_chrom` corresponds to.
- `chrom_sizes`, a list of integers giving the size of each chromosome. This is mostly here as a handy reference when you want to make a bigwig file.
- `coords_base`, the center point for each of the regions in the table of PISA values.
- `coords_chrom`, the chromosome on which each PISA vector is found. This is a list of integers. The width of the integer data type may vary from run to run, and is calculated based on the number of chromosomes in the genome file.
- `input_predictions`, a (numSamples,) array of the logit value of the target base when that sequence is run through the network.
- `shuffle_predictions`, a (numSamples, numShuffles) array of the logits of the target base in the shuffled reference sequences.
- `sequence`, a one-hot encoded array representing the sequence under each PISA value. The shape is (num regions * receptive-field * 4). Note that this is receptive field, not input width, since each base being shapped will only be affected by bases in its receptive field, and there's no reason to store the noise.
- `shap`, a table of the shap scores. The shape is the same as the sequence table, and each position in the shap table represents the corresponding base in the sequence table. These values are contribution scores to the difference-from-reference of the logit at this base.

5.9 Interpret PISA from fasta: `interpetPisaFasta.py`

This is the JSON file given to the PISA fasta tool, instead of fetching sequences based on a bed file and a genome, you supply the sequences directly. Since PISA calculates shap scores for a single base, this tool always calculates the pisa scores for the *leftmost* base in the output window.

Suppose the input length is 3090 bp, and the output is 1000 bp. In this case, the receptive field is 2091 bp, and there are 1045 bp of overhang on each end of the input sequence. So, for each input sequence, this program will assign shap scores to the 1046th base (one-indexed) of the input sequence.

5.9.1 Input specification

```
<pisa-fasta-configuration> ::= {
    "model-file" : <file-name>,
    "sequence-fasta" : <file-name>,
    "num-shuffles" : <integer>,
    "head-id" : <integer>,
    "task-id" : <integer>,
    "output-h5" : <file-name>,
    "output-length" : <integer>,
    "input-length" : <integer>,
    <verbosity-section>
}
```

5.9.2 Output specification

It produces an hdf5 format file which is organized as follows:

- `head-id`, an integer representing which head of the model was used to generate the data.
- `task-id`, an integer giving the task number within the specified head
- `descriptions`, a string taken from the fasta file. These are the comment (`>`) lines, and are stored without the leading `>`. This will have shape (numSamples,)
- `sequence`, the one-hot encoded input sequences. Note that these only cover the receptive field of the model, so they are not as wide as the input. This will have shape (numSamples x receptiveField x 4).
- `input_predictions`, the logit predicted for the base being shapped. This will have shape (numSamples,).
- `shap`, the shap scores. This will have shape (numSamples x receptiveField x 4).
- `shuffle_predictions`, the predicted logit of the target base on the shuffled input sequences. This will have shape (numSamples, numShuffles).

5.10 Preparing bed files: `prepareBed.py`

This is the JSON file given to the input preparation script. It splits your regions into test, train, and validation regions, and optionally applies some filtering. N.B. This program does not validate the name, score, or strand columns of the bed file. They are retained exactly as passed in.

5.10.1 Input specification

```
<prepare-bed-configuration> ::={
    <bigwig-section>,
    "splits" : {<split-settings>},
    "genome" : <file-name>,
    "output-length" : <integer>,
    "input-length" : <integer>,
    "max-jitter" : <integer>,
    <output-file-name-section>,
    "resize-mode" : <resize-mode>,
    <overlaps-section>
    <verbosity-section>
    }
<bigwig-section> ::=
    "heads" : [<head-preparation-list>]
  | (DEPRECATED) "bigwigs" : [<bigwig-preparation-list>]

<overlap-section> ::=
    "remove-overlaps" : true,
    "overlap-max-distance" : <integer>,
  | "remove-overlaps" : false,

(DEPRECATED) <bigwig-preparation-list> ::=
(DEPRECATED)    <individual-preparation-bigwig>
(DEPRECATED)  | <individual-preparation-bigwig>, <bigwig-preparation-list>

<head-preparation-list> ::=
    <individual-preparation-head>
  | <individual-preparation-head>, <head-preparation-list>

<resize-mode> ::=
    "none"
  | "center"
  | "start"

<output-file-name-section> ::=
    "output-prefix" : "<string>"
  | "output-train" : <file-name>,
    "output-val" : <file-name>,
    "output-test" : <file-name>

<individual-preparation-head> ::=
 { "bigwig-names" : [<list-of-bigwig-files>],
   <max-cutoff-section>,
   <min-cutoff-section>
   }

(DEPRECATED) <individual-preparation-bigwig> ::=
(DEPRECATED)  { "file-name" : <file-name>,
(DEPRECATED)    <max-cutoff-section>,
(DEPRECATED)    <min-cutoff-section>
(DEPRECATED)    }

<max-cutoff-section> ::=
   "max-quantile" : <number>
 | "max-counts" : <integer>
```

```
<min-cutoff-section> ::=
   "min-quantile" : <number>
 | "min-counts" : <integer>

<split-settings> ::=
    <split-by-chromosome-settings>
  | <split-by-name-settings>
  | <split-by-bed-settings>

<split-by-chromosome-settings> ::=
    "train-chroms" : [<list-of-strings>],
    "val-chroms" : [<list-of-strings> ],
    "test-chroms" : [<list-of-strings> ],
    "regions" : [<list-of-bed-files>]

<split-by-bed-settings> ::=
    "train-regions" : [<list-of-bed-files>],
    "val-regions" : [<list-of-bed-files>],
    "test-regions" : [<list-of-bed-files>]

<split-by-name-settings> ::=
    "regions" : [<list-of-bed-files>],
    "test-regex" : "<string>",
    "train-regex" : "<string>",
    "val-regex" : "<string>"

<list-of-bigwig-files> ::=
    <file-name>, <list-of-bigwig-files>
  | <file-name>

<list-of-bed-files> ::=
    <file-name>, <list-of-bed-files>
  | <file-name>
```

5.10.2 Parameter notes

- The `resize-mode` specifies where in the regions in the bed file the output regions should be centered. Note that this program assumes your bed files are in bed3 format, that is, (chrom, start, stop). If you have additional columns with information like peak offset, those data will be ignored.
- The max and min quantile values, if provided, will be used to threshold which regions are included in the output. First, all of the counts in the given regions are computed (which takes a while!), and then the given quantile is computed. All regions exceeding that value are not included in the output files.
- Similarly, if max and min counts are given, all regions having more (or fewer) reads than the given number will be excluded.
- If you use regexes to make your splits, then the `name` entry of the bed files will be compared against each regex. On a match, that region will be included in that split.
- The `remove-overlaps` flag can be set to `true` if you'd like to exclude overlapping regions. This is done by resizing all regions down to `overlap-max-distance`, and then, if multiple regions have an overlap, one is deleted at random. If `remove-overlaps` is `false`, then `overlap-max-distance` does not need to be set.

I should mention that the maximum and minimum counts are not compared across the same window. When comparing a region against the maximum counts value, all counts within the $\textit{input-length} + \textit{jitter}$ are added up. This way, if you have a crazy-huge spike just outside your region, that region will be rejected. Conversely, for minimum counts, the counts within the $\textit{output-length} - \textit{jitter}$ will be considered. This way, no matter what jitter value is selected, there will be at least the given number of counts in the region.

5.11 Modisco seqlet analysis: `motifSeqletCutoffs.py`

In order to see where seqlets are found on the genome, we need to scan the cwms derived from modiscolite. The first step of this process is to look at the seqlets that MoDISco called for each pattern it identified, and establish cutoff values. This program does that. It reads in a modiscolite hdf5 file and calculates cutoff values of seqlet similarity for what constitutes a hit. It requires one JSON-format input, and generates two outputs: First, it generates a tsv file containing all of the seqlets in the modisco h5 with some helpful metadata, like how well they match the pattern they are identified as being a part of. Second, it produces a JSON-format file that will be needed by `motifScan.py`.

5.11.1 Input specification

```
<motif-seqlet-cutoffs-configuration> ::= {
    <seqlet-scanning-settings>,
    <verbosity-section>
}

<seqlet-scanning-settings> ::=
    <seqlet-tsv-section>
    "modisco-h5" : <file-name>,
    <seqlet-contrib-section>
    <pattern-spec-section>,
    "seq-match-quantile" : <float-or-null>,
    "contrib-match-quantile" : <float-or-null>,
    "contrib-magnitude-quantile" : <float-or-null>,
    "trim-threshold" : <float>,
    "trim-padding" : <integer>,
    "background-probs" : [<float>, <float>, <float>, <float>],
    <quantile-json-section>

<float-or-null> ::=
    <float>
  | null

<pattern-spec-section> ::=
    "patterns" : "all"
  | "patterns" : [<list-of-pattern-specs>]

<list-of-pattern-specs> ::=
    <pattern-spec>
  | <pattern-spec>, <list-of-pattern-specs>

<pattern-spec> ::=
    {"metacluster-name" : <string>,
     "pattern-name" : <string> }
  | {"metacluster-name" : <string>,
     "pattern-names" : [<list-of-string>]
  | {"metacluster-name" : <string>,
     "pattern-name" : <string>,
     "short-name" : <string>}
  | {"metacluster-name" : <string>,
     "pattern-names" : [<list-of-string>],
     "short-names" : [<list-of-string>]}


<quantile-json-section> ::=
    <empty>
  | "quantile-json" : <file-name>,

<seqlet-contrib-section> ::=
    <empty>
  | "modisco-contrib-h5" : <file-name>,
```

```
<seqlet-tsv-section> ::=
    <empty>
  | "seqlets-tsv" : <file-name>,
```

5.11.2 Parameter notes

- `seqlets-tsv`, if provided, is the name of the file that should be writteni containing the scanned seqlets. See `motifAddQuantiles.py` for the structure of this file.
- `modisco-contrib-h5`, if provided, gives the contribution score file generated by `interpretFlat.py`, which is necessary to recover the genomic coordinates of the seqlets, since the Modisco hdf5 doesn't contain that info. The contribution scores are *not* extracted from this file, just coordinates. *THIS DOES NOT CURRENTLY WORK, SINCE SEQLET INDEXES ARE RESET BY MODISCO*
- There are two ways of specifying patterns, either by giving each pattern and metacluster pair individually, or by listing multiple patterns under a single metacluster. The short-names, if provided, will be used to populate the name field in the generated tsv. You could use this to give a particular pattern the name of its binding protein.
- TODO ADD DOCUMENTATION FOR QUANTILE AND TRIM PARAMETERS.
- The background-probs field gives the genetic content of your genome. For example, if you had a genome with 60 percent GC content, this would be [0.2, 0.3, 0.3, 0.2]. The order of the bases is A, C, G, and T.

5.11.3 Output specification

See `motifAddQuantiles.py` for a description of the tsv file, and `motifScan.py` for a description of the generated JSON.

5.12 Scanning for motifs: `motifScan.py`

This program scans over the contribution scores you calculated with `interpretFlat.py` and looks for matches to motifs called by modiscolite. It can be run with a quantile JSON from `motifSeqletCutoffs.py`, or you can include the settings for that program inside the JSON for this one, in which case it will perform the seqlet analysis first and save those results for you. If you include a `seqlet-cutoff-settings` block in the config, it will run the `motifSeqletCutoffs.py` tools, and if you don't include that, you must include a `seqlet-cutoff-json` file with the appropriate cutoffs. It is an error to specify both `seqlet-cutoff-settings` and `seqlet-cutoff-json`.

5.12.1 Input specification

```
<motif-scan-config> ::= {
    <scan-quantile-settings>
    "scan-settings" : {
        "scan-contrib-h5" : <string>,
        "hits-tsv" : <string>,
        "num-threads" : <integer>,
    }
    <verbosity-section>}

<scan-quantile-settings> ::=
    "seqlet-cutoff-json" : <string>,
  | "seqlet-cutoff-settings" : <seqlet-scanning-settings>,
```

You also need a JSON file containing the information for each pattern. This file is generated by `motifSeqletCutoffs.py` and saved to the name `quantile-json` in the configuration to that script.

```
<quantile-json> ::= [
    <list-of-scan-patterns> ]

<list-of-scan-patterns> ::=
    <scan-pattern>
  | <scan-pattern>, <list-of-scan-pattern>

<scan-pattern> ::= {
    "metacluster-name" : <string>,
    "pattern-name" : <string>,
    "short-name" : <string>
    "cwm" : <motif-array>,
    "pssm" : <motif-array>,
    "seq-match-cutoff" : <float-or-null>,
    "contrib-match-cutoff" : <float-or-null>,
    "contrib-magnitude-cutoff" : <float-or-null>}

<motif-array> ::= [
    <list-of-base-arrays> ]

<list-of-base-arrays> ::=
    <base-array>
  | <base-array>, <list-of-base-arrays>

base-array ::=
    [<float>, <float>, <float>, <float>]
```

5.12.2 Parameter notes

- The `scan-contrib-h5` file is the output of `interpretFlat.py` and contains contribution scores. All of the regions in this file will be scanned.

- `num-threads` is the number of parallel workers to use. Due to the streaming architecture of this program, `num-threads` must be at least 3. I have found that this program scales very well up to 70 cores, and haven't tested it beyond that.

In the quantile JSON, we find the actual numerical cutoffs for scanning.

- `metacluster-name` and `pattern-name` are from the modisco hdf5 file.
- `short-name` is a convenient name for this motif, and is entirely up to you. The short name will be used to populate the name column in the generated bed and csv files.
- `cwm` is an array of shape (length, 4) that contains the cwm of the motif. It is used to calculate the Jaccard similarity and the L1 score.
- `pssm` is the sequence-based information content at each position, and is used to calculate sequence match scores.
- The three cutoff values are the actual scores, *not quantile values*. These are calculated by `motifSeqletCutoffs.py`. You could set these manually, but why would you?

5.12.3 Output specification

For the generated tsv file, see `motifAddQuantiles.py`. If you include a `quantile-json` in your `seqlet-cutoff-settings`, then running `motifScan.py` will save out the cutoff JSON.

5.13 Annotating seqlets with quantile information: cwmAddQuantiles.py

This little helper program calculates quantile values for seqlets and called motif instances. For each pattern (patterns in different metaclusters are distinct), it looks at the seqlets and determines where that seqlet's importance magnitude, contribution match, and sequence match scores fall among other seqlets in that pattern. Then, for motif hits, it sees where each hit falls, in terms of quantile, among the seqlets in that pattern.

The quantile is based on a very simple definition. A particular seqlet's quantile is calculated by sorting all of the seqlets for one pattern. The lowest-scoring seqlet gets a quantile of 0.0, the highest-scoring gets 1.0, and the seqlets in between get quantile values based on their order in the sorted metric.

For scanned hits, we take the sorted array of seqlet statistics (for the same pattern as the matched hit fell into) and ask, 'where would the score of this hit rank among the sorted array of seqlet scores?' The hit's rank is then its quantile score. If a hit falls between the scores of two seqlets, then a linear interpolation is performed to assign a quantile value.

The input and ouput to this program are tsv files, and the only difference in format is that the outputs have three additional columns: `contrib_magnitude_quantile`,
`seq_match_quantile`, and `contrib_match_quantile`. The remaining columns are described below:

- `chrom` gives the chromosome where the seqlet or hit was found.
- `start` gives the start position (inclusive, zero-based) of the seqlet or motif hit.
- `end` gives the end position (exclusive, zero-based) of the seqlet or motif hit.
- `short_name` is the user-provided name for this motif. If you didn't provide one in the configuration for `motifSeqletCutoffs.py`, then it will be something like `pos_0` for the positive metacluster, pattern zero.
- `contrib_magnitude` is the total contribution across this motif instance. A higher value means more motif contribution.
- `strand` is a single character indicating if the motif was on the positive or negative strand.
- `metacluster_name` is straigh from the modisco hdf5 file. It will be something like `pos_patterns`.
- `pattern_name` is also from the modisco hdf5. It will be something like `pattern_5`.
- `sequence` is the DNA sequence of that motif instance.
- `index` gives either the region index in the contribution hdf5 (from `motifScan.py`), or the seqlet index in the modisco hdf5 (from `motifSeqletCutoffs.py`).
- `seq_match` gives the information content of the sequence match to the motif's pwm.
- `contrib_match` gives the continuous Jaccard similarity between the motif's cwm and the contribution scores of this seqlet.

If a contribution hdf5 file was not provided to `cwmPrepare.py`, the chrom, start, and end columns are meaningless. Note that the first six columns define a bed file, and a simple `cut` command can generate a viewable bed file from these tsvs:

```
cat scan.tsv | cut -f 1-6 | tail -n +2 > scan.bed
```

Note that the hits from scanning can contain duplicates. This can happen if the same bases appear in multiple regions (i.e., there is overlap in the region set). In this case, it makes sense to only keep the best instance (highest importance magnitude) of that motif hit. This can be done with a little Unix-fu:

```
cat scan.tsv | \
    cut -f 1-6 | \
    tail -n +2 | \
    sort -k1,1 -k2,2n -k3,3n -k4,4 -k5,5nr | \
    awk '!_[$1,$2,$3,$4,$6]++' > scan.bed
```

6 Model architectures

The precise details of the model architectures can be found in models.py, but they share some common themes. Every model that ever gets saved to disk accepts a one-hot encoded sequence as input, and produces outputs that are grouped into "heads". A model may generate any number of heads, and the heads may have different sizes. In general, each head should represent one set of DNA fragments. For example, an experiment that produces cut sites on the + and - strand of DNA produces two tracks, but the tracks represent two ends of the same fragments. So these two tracks would be in the same head. However, if you have an experiment where it's appropriate to split fragments into "short" (100-500 bp) and "long" (1 kb to 10 kb), then those tracks do not represent the same fragments, so they should be in different heads.

If you have done ChIP-nexus on three different factors, then you'd have three heads, each one corresponding to a different factor, and each head would predict both the + and - strand data for that factor.

If you're not sure if you can combine your data under one output head, it's much safer to split the data into multiple heads.

A head contains a profile prediction and a counts prediction. The profile prediction is a tensor of shape (batch-size x) number-of-tracks x output-width, and each value in this tensor is a logit. Note that the *whole* profile prediction should be considered when taking the softmax. That is to say, the profile of the first track is NOT $e^{logcounts} * softmax(profile_{0,:})$, but rather you have to take the softmax first and then slice out the profile: $e^{logcounts} * softmax(profile)_{0,:}$. There is a function, `logitsToProfile`, in `utils.py` that does this automatically.

Of course, if the profile only has one track, this distinction is vacuous. The counts output is a scalar that represents the natural logarithm of the number of reads predicted for the current region.

It is possible to add more model architectures, but currently the program only supports a BPNet-style architecture. You can take a look at soloModel in `models.py` for details on how it works.

7 PISA

The traditional method of interpretation of BPNet models asks the question, "How is it that this base here affects some readout over this window?", where the readout is a single scalar value calculated over an entire prediction window. If our readout is counts, then we are calculating how much each input base contributed to the total counts over our prediction window. For profile, we calculate a weighted mean-normalized logit value (as was done in the original BPNet paper). In this way, each base gets a single scalar value for how it contributed to the overall readout.

PISA asks a subtly different question: "How is it that this base here affects the readout at that base over there?" In this case, instead of looking at how much one base contributes to a global readout, we're asking about its effect on a single one of its neighbors. With PISA, each base gets a vector of its contributions to the readout at each one of its neighbors. This vector has shape (receptive-field,) Accordingly, if we perform PISA on a region of the genome, we would get a PISA array of shape (region-length, receptive-field).

We use code derived from `deepshap` to perform this calculation. With deepShap, we create an explainer that looks at one output from the model. For the sake of simplicity, all of the PISA implementation looks at the leftmost base in the output window, but this implementation detail has no effect on the actual calculated values. Once the explainer has been created, we provide it with genomic sequences to explain, and it assigns, to each base in the input, its contribution to the observed output.

The outputs of the model are logits, and the contribution scores have the same units, so the explainer is effectively assigning a (base e) fold-change value to each input.

There are several properties of Shapley values that are important here. In these formulae, ϕ_i is the contribution score of base i, drawn from the sequence S. The readout that we're measuring, the logit at the leftmost base, is $v(S)$. I'll also use K to refer to a subset of the input sequence S. I'll use R to refer to an ensemble of reference sequences, and the average prediction from those reference sequences is $\bar{v}(R)$

The first, and arguably most important, property of Shapley values is *efficiency*:

$$\sum_{i \in S} \phi_i(v) = v(S) - \bar{v}(R) \tag{1}$$

This means that if we add up all the shap values that were assigned for a particular logit, we recover the difference-from-reference of that logit.

One possible weakness of a method like this has to do with cooperation. Suppose that some readout is observed if at least one of the bases in a region is A. If two bases are A, we observe that readout. But how should we assign ϕ values to those two bases? With shapley values, we are guaranteed that they will get the same score:

$$\Big(\forall (K \subseteq S \backslash \{i,j\}) \; \big(v(K \cup \{i\}) = v(K \cup \{j\})\big)\Big) \implies \phi_i(v) = \phi_j(v) \tag{2}$$

The third property turns out to be very important in performing PISA on bias-corrected models. The combined model uses a simple sum to combine the logits from the solo model and the residual model, so $v_{combined}(S) = v_{solo}(S) + v_{residual}(S)$. Shapley values preserve this linearity, ensuring that it's meaningful to look at PISA plots of a residual model:

$$\phi_i(u + v) = \phi_i(u) + \phi_i(v) \tag{3}$$

Finally, an almost-obvious property: If a particular base has no effect on the readout, its ϕ values should be zero. So it is:

$$\Big(\forall (K \subseteq S \backslash \{i\}) \big(v(K \cup \{i\}) = v(K)\big)\Big) \implies phi_i(v) = 0 \tag{4}$$