# DeepNuc Instructions

June 19, 2017

## 0.1 *DeepNuc* Manual

DeepNuc v0.8. Manual v1.0. Date: 6/15/17. Author: Lawrence Du

## 0.2 API docs

API documentation for *DeepNuc* can be found under DeepNuc/docs/_build/py-modindex.html
Currently, (as of version 0.8) not all methods are documented.

## 0.3 Setting path

If *DeepNuc* was not installed as a pip package, the correct path needs to be set at the top of your script in order to use *DeepNuc*.

This can be done with

```
In [2]: import sys
        sys.path.append('DEEPNUC_PATH')
```

where DEEPNUC_PATH specifies the path to the DeepNuc directory.

All examples in this manual assume usage of this directory structure.

Although this manual was written using Jupyter notebook, I would not recommend running training within a notebook environment.

## 0.4 Example 1: Training and testing a model from a fasta file

For our first example, we will train and test a binary classifier using the commonly used fasta format. All entries of an input fasta file need to be the same length.

First import the appropriate classes.

```
In [ ]: import tensorflow as tf
        import sys
        import os.path
        sys.path.append('../../')
        import numpy as np

        from deepnuc.nucdata import *
        from deepnuc.nucbinaryclassifier import NucBinaryClassifier
        from deepnuc.databatcher import DataBatcher
```

```
from deepnuc.modelparams import *
from deepnuc.logger import Logger
import deepnuc.nucconvmodel as nucconvmodel
from deepnuc.crossvalidator import CrossValidator
```

Then specify model parameters. * seq_len: Sequence length. * num_epochs: Number of epochs. An epoch is a single pass through every item in a training set. * learning_rate: Learning rate for gradient descent * batch_size: Mini-batch size. * keep_prob: The keep probability for Dropout. A keep prob of .6 will randomly drop 40% of weights on each training cycle. * beta1: A parameter used by the AdamOptimizer * concat_revcom_input: If set to true, the reverse complemented sequence nucleotide sequence for a item will be concatenated to the input vector. * inference_method_key: A string value key for different methods found in nucconvmodel.py. Currently can be set to "inferenceA" or "inferenceD"

```
In [ ]: params = ModelParams(
                                    seq_len=600,
                                    num_epochs=35,
                                    learning_rate=1e-4,
                                    batch_size=24,
                                    keep_prob=0.5,
                                    beta1=0.9,
                                    concat_revcom_input=False,
                                    inference_method_key="inferenceA"
                                )
```

Now create a NucData type object from your fasta file. NucData type objects are described in the nucdata.py module. There are many different types of NucData objects for different filetypes. We will use NucDataBedMem which reads all nucleotide sequences into the computer's memory.

Note that since most bed files will have irregularly sized entries (end-start will differ for most lines), this the NucDataBedMem constructor has an option to only pull objects within a window around the start coordinate to make things nice and even.

This constructor also has an option to automatically generate a dinucleotide shuffled negative class fasta file from each bed file entry. More information can be found in the API docs.

```
In [ ]: bed_file = "kruesi_bed_0_4.bed"
        genome_file="WS230_genome.fa"
        chr_sizes_file="WS230_genome.chr.sizes"
        seq_len=600
        skip_first=False

        nuc_data = NucDataBedMem( bed_file,
                                  genome_file,
                                  chr_sizes_file,
                                  seq_len=600,
                                  skip_first= False, #skip header line if true
                                  gen_dinuc_shuffle=True,
                                  neg_data_reader=None,
                                  start_window=[-300,300]
                                 )
```

NucData objects encapsulate data from different file formats, but aren't used for training and testing directly. DataBatcher objects are useful for slicing NucData objects into training and testing sets by index. DataBatcher objects also shuffle records by index every epoch and pull randomized mini-batches for training without replacement.

```
In [ ]: #First scramble the indices you want to use
        seed = 12415
        perm_indices = np.random.RandomState(seed).\
                            permutation(range(nuc_data.num_records))

        #Slice out 20% of indices for testing and 80% for training
        test_frac = .2
        test_size = int(nuc_data.num_records*test_frac)
        train_size = nuc_data.num_records - test_size
        test_indices = perm_indices[0:int(nuc_data.num_records*test_frac)]
        train_indices = np.setdiff1d(perm_indices,test_indices)

        #Create training and testing batchers
        train_batcher = DataBatcher(nuc_data,train_indices)
        test_batcher = DataBatcher(nuc_data,test_indices)
```

With the training and testing batchers, we can now run training from within a Tensorflow session.

Note: Training parameters can be passed directly to NucBinaryClassifier, but in many situtations, using the ModelParams object can be useful for writing methods which train off different dataset using the same training parameters.

```
In [ ]: save_dir="test_train1"
        os.makedirs(save_dir)
        with tf.Session() as sess:
            nc_test = NucBinaryClassifier(sess,
                                          train_batcher,
                                          test_batcher,
                                          params.num_epochs,
                                          params.learning_rate,
                                          params.batch_size,
                                          params.seq_len,
                                          save_dir,
                                          params.keep_prob,
                                          params.beta1
                                          params.concat_revcom_input,
                                          params.inference_method_key)

            nc_test.build_model()
            nc_test.train()
```

nc_test.build_model() will also load trained models. For example. After running training, the following line can be used to evaluate metrics from a differerent test or evaluation batcher.

```
In [ ]: nc_test.build_model()
        nc_test.eval_model_metrics(different_batcher)
```

#### 0.4.1 Generating a dinucleotide shuffled file manually

Making dinucleotide shuffled files can also be done manually using the BedReader class

```
In [ ]:
```

### 0.5 Example 2: Training and testing a model from a fasta file

Training and testing a from a fasta file is identical to training and testing from a bed file. The only difference is changing the nuc_data object from a NucDataBedMem to a NucDataFastaMem.

NucDataFastaMem takes a list of fasta files as an argument. For binary classification this list should be length = 2. The first fasta file passed in the list is the negative class and the second file passed should be the positive class.

```
In [ ]: fasta_neg = "kruesi_tss_-300_300_dinuc_shuffle.fa"
        fasta_pos = "kruesi_tss_-300_300.fa"
        nuc_data = NucDataFastaMem([fasta_neg,fasta_pos],seq_len=600)
```

### 0.6 Example 3: Performing k-folds cross-validation using a bed file

Getting a model trained and evaluating a test set is nice, but in a real world situation, we want to have a way to know how well our model performed. This is where k-folds cross-validation is useful. Cross validation can be done in the following manner.

```
In [ ]: cv_save_dir = "cv_test1"
        cv_test = CrossValidator(params,
                                 nuc_data,
                                 cv_save_dir,
                                 seed=124155,
                                 k_folds=3,
                                 test_frac=0.15)
        cv.run()

        #Calculate average of k-fold metrics and display in terminal
        cv.calc_avg_k_metrics()

        #Plot curves
        cv.plot_auroc_auprc("cv test curves")
```

Note that cv.run() does not need to be called from within a tf.Session()

### 0.7 Example 4: Performing a grid search with cross-validation using a bed file

On top of k-folds cross-validation, we want to have cross-validation metrics for different hyper-parameter combinations. This is where a grid search is useful. Performing a grid search is one of the simplest and most common ways of using *DeepNuc*

A grid search requires initialization of a GridParams object. This object is similar to ModelParams except all arguments except seq_len should be passed as lists

```
In [ ]: gsave_dir="gridtest1"
        gparams = GridParams(
                                seq_len= seq_len,
                                num_epochs=[55],
                                learning_rate=[1e-4],
                                batch_size=[24],
                                keep_prob=[0.5],
                                beta1=[0.9],
                                concat_revcom_input = [True,False],
                                inference_method_key=["inferenceA","inferenceD"])
                               )

        nuc_data = NucDataBed(bed_file,
                                genome_file,
                                chr_sizes_file,
                                seq_len=600,
                                skip_first= False, #skip header line if true
                                gen_dinuc_shuffle=True,
                                start_window=[-300,300]
                                )

        gsearch = GridSearch(
                                  gparams,
                                  nuc_data,
                                  save_dir=gsave_dir,
                                  seed = 29517,
                                  k_folds=3,
                                  test_frac=.2,
                                  fig_title_prefix = "")
        gsearch.run() #Does not need to be run within tf.Session()
```

After a grid search is performed, the model can be loaded and final training performed on the best classifier:

```
In [ ]: with tf.Session() as sess:
            best_classifier,best_model_params = gsearch.best_binary_classifier(sess
            print "\n\n\nBest model params for {}".format(tss_bed)
            best_model_params.print_params()
```

The best_classifier can be treated as any other NucBinaryClassifier object.

```
In [ ]:     best_classifier.eval_model_metrics(different_batcher)
```

Note that if running back to back grid search or training runs from the same script, you should reset the default graph with the following command.

```
In [ ]: tf.reset_default_graph()
```

### 0.8 Example 5: Using a fasta or bed file as a negative class

To specify a specific bed or fasta file to use as a negative class, pass a BedReader or FastaReader to the neg_data_reader parameter of NucDataBedMem when making a NucData object.

```
In [ ]: seq_len=600
        specific_neg_file="my_specific_file.fa"
        my_neg_reader = FastaReader(specific_neg_file,seq_len)


        bed_file = "kruesi_bed_0_4.bed"
        genome_file="WS230_genome.fa"
        chr_sizes_file="WS230_genome.chr.sizes"

        skip_first=False

        nuc_data = NucDataBedMem( bed_file,
                                  genome_file,
                                  chr_sizes_file,
                                  seq_len=600,
                                  skip_first= False, #skip header line if true
                                  gen_dinuc_shuffle=False
                                  neg_data_reader= my_neg_reader
                                  start_window=[-300,300]
                                 )
```

Since training should be performed with balanced data, you may want to set a limit on the number of negative items you pull from a specified file. This is available as an argument to the FastaReader object.

```
In [ ]: seq_len=600
        specific_neg_file="my_specific_file.fa"

        my_neg_reader = FastaReader(specific_neg_file, seq_len, pull_limit=4121)
```

### 0.9 Example 6: Using DeepNuc for regression (untested)

Regression has yet to be fully implemented. Running models for regression problems (ie: protein binding microarray data) is available through the nucregressor.NucRegressor class. There is also a cross-validation class implemented as crossvalidator.RegressorCrossValidator

```
In [ ]:
```

### 0.10 Addenum 1: Adding new models using Tensorflow

*See nucconvmodels.py for examples*

```
In [ ]:
```