Machine Learning with PLearn

How some standard (and non-standard) ML algorithms are implemented in PLearn, and how to play with them

August 3, 2022

Contents

Introduction

The purpose of this document is to document the way some particular learning algorithms (like Deep Belief Networks) are implemented using PLearn's base classes. It is not to detail how those base classes are working.

You should read *PLearn programmer's guide* first (or at least have it reachable), you will need it for information about PLearn's generic classes, especially `Object` and `PLearner`, but also `Var` and `OnlineLearningModule`, and for the general coding philosophy.

Chapter 1

A Var-based PLearner: NNet

Chapter 2

Boltzmann Machines and Deep Belief Networks

The equations can be seen on `http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DBNEquations`.

All the code files are located in `$PLEARNDIR/plearn_learners/online`.

2.1 Architecture

2.1.1 Restricted Boltzmann Machines

A Restricted Boltzmann Machine (RBM) is composed of two different layers of units, with weighted connection between them.

The layers are modelled by the `RBMLayer` class, while the connections are represented by `RBMConnection`. Different sub-classes implement the multiple types of layers and connections. `RBMLayer` and `RBMConnection` both inherit from `OnlineLearningModule`.

An RBM can therefore be considered as a structure containing two instances of `RBMLayer` and one of `RBMConnection`, but there is no class modelling an RBM for the moment.

2.1.2 Deep Belief Networks

A Deep Belief Network (DBN) is a learning algorithm, therefore contained in a `PLearner`, namely `DeepBeliefNet`.

It is composed of stacked RBMs. The units of a layer are shared between two RBMs, hence the need of dissociating layers and connections. A `DeepBeliefNet` containing n unit layers (including input and output layers) will typically contain n instances of `RBMLayer` and $n-1$ instances of `RBMConnection`.

The training is usually done one layer at a time, each layer being trained as an RBM. See part 2.4 for the detailed explanation.

There are no functions for sampling from the learned probability distribution yet, they might be added at some point in time.

2.2 Code Components

Both classes inherit from `OnlineLearningModule`, so they have deterministic `fprop(...)` and `bpropUpdate(...)` functions, that can be chained.

2.2.1 `RBMLayer`

This class models a set of (usually independant) units, some of their intrinsic parameters, and their current state.

`RBMLayer` stores:

- `size`: number of units
- `bias`: vector of the units' biases
- `activation`: the value of the weighting sum of the inputs, plus the bias
- `expectation`: the expected value of each unit's distribution
- `sample`: a sample from the distribution
- some flags to know what is up-to-date
- `bias_pos_stats` and `bias_neg_stats`: accumulate positive phase and negative phase contributions to the CD gradient wrt the `bias`
- `pos_count` and `neg_count`: keep track of the number of accumulated contributions
- `learning_rate` and `momentum`: control the update (more hyper-parameters might be added)

The methods are:

- `getUnitActivation( int i, RBMConnection rbmc )`: get the result of the linear transformation from `rbmc`, and add the corresponding bias for unit `i`. It calls `rbmc->computeProduct`.
- `getAllActivations( RBMConnection rbmc )`: same as above, but for all units in the layer
- `computeExpectation()`: compute the value of `expectation`, given `activation` (with a caching system, to avoid computing twice if `activation` didn't change)
- `generateSample()`: generates a sample, given the value of `activation`, and places it in `sample`

- `accumulatePosStats( Vec pos_values )`: accumulate statistics from the positive phase

```
bias_pos_stats += pos_values;
pos_count++;
```

- `accumulateNegStats( Vec neg_values )`: idem with the negative phase

```
bias_neg_stats += neg_values;
neg_count++;
```

- `update()`: update the bias (and other parameters if some) from accumulated statistics

```
bias -= learning_rate * (bias_pos_stats/pos_count - bias_neg_stats/neg_count)

# reset
bias_pos_stats.clear();
bias_neg_stats.clear();
pos_count = 0;
neg_count = 0;
```

And from the `OnlineLearningModule` interface:

- `fprop( Vec input, Vec output )`: `input` represents the output of the `RBMConnection`, and `output` the expectation (mean-field approximation) of the layer. For an `RBMBinomialLayer`:

```
output = sigmoid( -(input + bias) );
```

- `bpropUpdate( Vec input, Vec output, Vec input_grad, Vec output_grad )`: back-propagate a gradient through the layer, and update the parameters (`bias`,...) accordingly, given the `learning_rate`, `momentum`, etc.

Different types of units (binomial, Gaussian, even groups of units representing a multinomial distribution, etc.), so this class has several derived sub-classes, which may store more information (like a quadratic parameter, and the standard deviation for a Gaussian unit) and use them in the `accumulate{Pos,Neg}Stats(...)` and `update()` methods.

List of known sub-classes:

- `RBMBinomialLayer`: stores binomial (0 or 1) units (the simplest implementation)
- `RBMMultinomialLayer`: stores a group of 0/1 units, so that exactly one of them is 1 at any time
- `RBMGaussianLayer`: stores real-valued units with Gaussian distributions
- `RBMTruncExpLayer`: stores real-valued units in a [0, 1] range, with a truncated exponential distribution
- `RBMMixedLayer`: concatenation of several `RBMLayer`

2.2.2 RBMParameters

This class represents a linear transformation (not affine! the bias is in the `RBMLayer`), used to compute one layer's activation given the other layer's value.

`RBMConnection` stores (and has to update):

- `up_size` and `down_size`: the number of units in the layers above and below (respectively)
- `input_vec`: a pointer to its current input vector (sample or expectation), and a flag to know if it is up or down
- Something that contains the weights of the connections (can be a matrix, a set of convolution filters...), let's call it `weights`
- `weights_pos_stats`, `weight_neg_stats`: statistics accumulated during positive and negative (respectively) phases
- `pos_count` and `neg_count`
- `learning_rate` and `momentum`

The different sub-classes will store differently the parameters allowing to compute the linear transformation, and the statistics used to update those parameters (usually named `[paramname]_pos_stats` and `[paramname]_neg_stats`).

The methods are:

- `setAsUpInput( Vec input )`: set the input vector, and flag to 'up'
- `setAsDownInput( Vec input )`: same, but 'down'
- `computeProduct( int start, int length, Vec activations, bool accumulate )`: compute the output activation of `length` units, starting from `start`. These units belong to the *above* layer if the `input` was *down*, and to the layer *below* if the `input` was *up*. The output is put in `activations` (or added if `accumulate`, not shown in the code below).

  ```
  if( up ):
      for i=start to start+length:
          activations[i-start] += sum_j weights(i,j) input_vec[j]
  else:
      for j=start to start+length:
          activations[j-start] += sum_i weights(i,j) input_vec[i]
  ```

- `accumulatePosStats( Vec down_values, Vec up_values )`: in the basic case of an `RBMMatrixConnection`

  ```
  weights_pos_stats += up_values * down_values';
  pos_count++;
  ```

- `accumulateNegStats( Vec down_values, Vec up_values )`: in the basic case of an `RBMMatrixConnection`

```
weights_neg_stats += up_values * down_values';
neg_count++;
```

- `update()`: update from accumulated statistics

```
weights -= learning_rate * (weights_pos_stats/pos_count - weight_neg_stats/neg_count);

# reset
weights_pos_stats.clear();
weights_neg_stats.clear();
pos_count = 0;
neg_count = 0;
```

- `fprop( input, output )`: performs the linear transformation on `input`, and put the result in `output`; typically

```
output = weights * input;
```

And from the `OnlineLearningModule` interface:

- `bpropUpdate( input, output, input_grad, output_grad )`: backpropagates the output gradient, and update the parameters (weights, ...) accordingly, given the `learning_rate`, `momentum`, etc.

```
input_grad = weights' * output_grad;
weights -= learning_rate * output_grad * input';
```

List of known subclasses, and their parameters:

- `RBMMatrixConnection`: `Mat weights` (simple matrix multiplication)
- `RBMConv2DConnection`: `Mat kernel`, along with `int down_image_length, down_image_width, up_image_length, up_image_width, kernel_step1, kernel_step2, kernel_length, kernel_width` (2 dimensional convolution filters)
- `RBMMixedConnection`: `TMat<RBMConnection> sub_connections` (block-matrix containing other `RBMConnection`, which specify a part of the global linear transformation)

2.3 Code Samples

2.3.1 Propagation in an RBM

In the simple case of a Restricted Boltzmann Machine, we have two instances of `RBMLayer` (`input` and `hidden`) and one of `RBMConnection` (`rbmc`) linking both of them.

Getting in `hidden_exp` the expected value of the hidden layer, given one input sample `input_sample`, is easy:

```
input.sample << input_sample;
rbmc.setAsDownInput( input.sample );
hidden.getAllActivations( rbmc );
hidden.computeExpectation();
hidden_exp << hidden.expectation;
```

If we want a sample `hidden_sample` instead, it is:

```
input.sample << input_sample;
rbmc.setAsDownInput( input.sample );
hidden.getAllActivations( rbmc );
hidden.generateSample();
hidden_sample << hidden.sample;
```

2.3.2 Step of Contrastive Divergence in an RBM

One step of contrastive divergence learning (with only one example, `input_sample`) in the same RBM would be:

```
// positive phase
input.sample << input_sample;
rbmc.setAsDownInput( input.sample );
hidden.getAllActivations( rbmc );
hidden.computeExpectation();
hidden.generateSample();
input.accumulatePosStats( input.sample );
rbmc.accumulatePosStats( input.sample, hidden.expectation );
hidden.accumulatePosStats( hidden.expectation );

// down propagation
rbmc.setAsUpInput( hidden.sample );
input.getAllActivations( rbmc );
input.generateSample();

// negative phase
rbmc.setAsDownInput( input.sample );
```

```
hidden.getAllActivations( rbmc );
hidden.computeExpectation();
input.accumulateNegStats( input.sample );
rbmc.accumulateNegStats( input.sample, hidden.expectation );
hidden.accumulateNegStats( hidden.expectation );

// update
input.update();
rbmc.update();
hidden.update();
```

Note: it was empirically shown that the convergence is better if we use `hidden.expectation` instead of `hidden.sample` in the statistics.

Or `update(..., ...)`

2.3.3 Learning in a DBN

Instead of having only one RBM, let's consider three sequential layers (`input`, `hidden`, `output`) and two connections:

- `rbmc_ih` between `input` and `hidden`;
- `rbmp_ho` between `hidden` and `output`.

They form a (small) DBN.

We first train the first RBM formed by (`input`, `rbmc_ih`, `hidden`) as shown previously, ignoring the other elements. Then, we freeze the parameters of `input` and `rbmc_ih`, and train the second RBM, formed by (`hidden`, `rbmc_ho`, `output`) taking the output of the first one as inputs.

One step of this second phase (with only one example, `input_sample`) will look like:

```
// propagation to hidden
input.sample << input_sample;
rbmc_ih.setAsDownInput( input.sample );
hidden.getAllActivations( rbmc_ih );
hidden.computeExpectation(); // we use mean-field approximation

// positive phase
rbmc_ho.setAsDownInput( hidden.expectation );
output.getAllActivations( rbmc_ho );
output.computeExpectation();
output.generateSample();
hidden.accumulatePosStats( hidden.expectation );
rbmc_ho.accumulatePosStats( hidden.expectation, output.expectation );
output.accumulatePosStats( output.expectation );
```

```
// down propagation
rbmc_ho.setAsUpInput( output.sample );
hidden.getAllActivations( rbmc_ho );
hidden.generateSample();

// negative phase
rbmc_ho.setAsDownInput( hidden.sample );
output.getUnitActivations( rbmc_ho );
output.computeExpectation();
hidden.accumulateNegStats( hidden.sample );
rbmc_ho.accumulateNegStats( hidden.sample, output.expectation );
output.accumulateNegStats( output.expectation );

// update
hidden.update();
rbmc_ho.update();
output.update();
```

2.4 The `DeepBeliefNet` Class

To be continued…

License

This document is covered by the license appearing after the title page.

The PLearn software library and tools described in this document are distributed under the following BSD-type license: