EBLearn: Open-Source Energy-Based Learning in C++

Pierre Sermanet Koray Kavukcuoglu Yann LeCun

Courant Institute of Mathematical Sciences
Computer Science Department
New York University
New York, NY, 10003, USA
{sermanet,koray,yann}@cs.nyu.edu

Abstract

*Energy-based learning (EBL) is a general framework to describe supervised and unsupervised training methods for probabilistic and non-probabilistic factor graphs. An energy-based model associates a scalar energy to configurations of inputs, outputs, and latent variables. Inference consists in finding configurations of output and latent variables that minimize the energy. Learning consists in finding parameters that minimize a suitable loss function so that the module produces lower energies for "correct" outputs than for all "incorrect" outputs. Learning machines can be constructed by assembling modules and loss functions. Gradient-based learning procedures are easily implemented through semi-automatic differentiation of complex models constructed by assembling predefined modules. We introduce an open-source and cross-platform C++ library called EBLearn[1] to enable the construction of energy-based learning models. EBLearn is composed of two major components, **libidx:** an efficient and very flexible multi-dimensional tensor library, and **libeblearn:** an object-oriented library of trainable modules and learning algorithms. The latter has facilities for such models as convolutional networks, as well as for image processing. It also provides graphical display functions.*

1. Introduction

Energy-based learning [11] *(EBL)* provides a unified framework for probabilistic and non-probabilistic machine learning methods. Energy based learning models have been successfully used in a number of applications such as object recognition [6, 16, 7], outdoor unstructured robotics vision [5], signal processing [13], time series modeling [12],

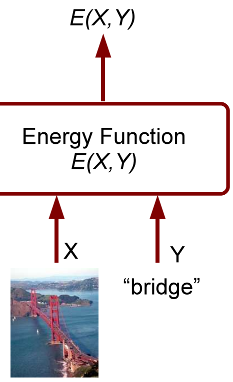

Figure 1. Energy Based Inference: Inference in *EBL* is done by finding Y that minimizes E(X,Y) for a given input X. Input can be a set of image pixels and the output to be predicted can be a class label as shown.

manifold learning [1, 4], financial prediction [2], document recognition [10], natural language processing [3], unsupervised learning of feature hierarchies [18, 15, 8] and text classification [17]. ***Inference*** in energy based models for a given input X is performed by finding a configuration of output Y and latent variables Z that minimize an ***energy function*** $E(X, Y)$. For ***Learning***, the energy function is parameterized by a parameter vector W. Learning is performed by minimizing a suitable ***loss functional*** with respect to W. For a given input X, the purpose of learning is to shape the energy surface so that corresponding desired output configuration(s) have lower energy than all other configurations.

A simplified *EBL* model is shown in Figure 1, X represents the inputs to the system (a sample from the data), Y represents the output variable of the model. the inference process is formulated as:

$$Y^* = \arg\min_{Y \in \mathcal{Y}} E(X, Y) \tag{1}$$

[1] http://eblearn.sf.net

where $\mathcal{Y}$ is a set of possible outputs. When $\mathcal{Y}$ is a discrete set with few elements, exhaustive search can be used, but when $\mathcal{Y}$ has high cardinality or is a continuous set, suitable minimization algorithms must be employed. In unsupervised scenarios, the energy function has no observation X, and the model $E(Y)$ simply indicates whether a particular Y is similar to training samples (low energies) or dissimilar (higher energies). Different types of problems can be formulated under this model, such as classification, detection, regression, ranking, density estimation, clustering, and others.

Probabilistic models are a special case of EBL in which the energy is integrable with respect to Y. The distribution can be obtained using the Gibbs formula:

$$P(Y|X) = \frac{e^{-\beta E(X,Y)}}{\int_y e^{-\beta E(X,y)}} \quad (2)$$

where β is a positive constant, and the denominator is called the *partition function* (variable X is simply dropped for unsupervised scenarios).

When latent variables Z are present, they can be minimized over or marginalized over. With minimization, the energy function is simply redefined as $E(X,Y) = \min_{Z \in \mathcal{Z}} E(X,Y,Z)$, and with marginalization as $E(X,Y) = -\frac{1}{\beta}\log\int_z e^{-\frac{1}{\beta}E(X,Y,Z)}$. If necessary. this integral can be approximated through sampling or using variational methods.

Given a training set $\{(X^1,Y^1),\dots(X^P,Y^P)\}$, training a model consists in shaping the energy surface $E(W,X,.)$ (expressed as a function of Y) parametrized by $W \in \mathcal{W}$ by minimizing a suitable loss functional with respect to W, averaged over the training set:

$$W^* = \min_{W \in \mathcal{W}} \frac{1}{P}\sum_i \mathcal{L}(E(W,X^i,.),Y^i) \quad (3)$$

The loss can more simply be expressed as a function of W : $\frac{1}{P}\sum_i \mathcal{L}(W,X^i,Y^i)$. The purpose of loss is to measure whether the "correct" output Y^i for a given X^i has lower energy than all other outputs. As a result, the system produces lower energy values for regions around observed Y values, as shown in Figure 2. Building an *EBL* model can therefore be achieved by designing:

1. the *architecture* of the $E(W,X,Y)$,
2. the *inference algorithm* that will be used to infer outputs Y that minimize $E(W,X,Y)$ for a given X and fixed W,
3. the *loss function* $\mathcal{L}(E(W,X,.),Y)$ and
4. the *learning algorithm* that will be used to find the best W that minimizes the loss averaged over a training set.

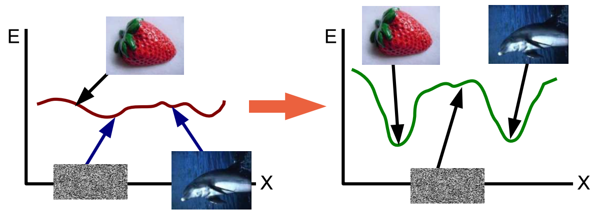

Figure 2. Energy Based Training: Before training, the energy surface produced by an *EBL* model is not distinctive around training data. After training, the energy surface is shaped lower around training data.

1.1 Loss Functions for Energy-Based Learning

In energy based learning designing proper loss functions for different types of architectures is required to avoid trivial solutions where the energy surface becomes flat. Many different loss functions have been proposed in machine learning literature and in this section we formulate several popular loss functions in energy based learning framework.

Energy Loss: simply means that the loss function per each sample is equal to the energy function:

$$\mathcal{L}_{en}(W,X^i,Y^i) = E(W,X^i,Y^i) \quad (4)$$

Energy loss is the simplest possible loss function that can be used in wide variety of cases. The energy loss will ensure that the energy surface is *pulled down* around desired data, but it does not guarantee that the energy surface is *pulled up* at all other locations. This might lead to situations where the energy for all samples becomes constant. Linear regression using mean squared error (MSE) can be formulated using energy loss:

$$\mathcal{L}(W,X^i,Y^i) = E(W,X^i,Y^i) = ||Y^i - WX^i||^2 \quad (5)$$

where W is the matrix of parameters with respect to which $\mathcal{L}$ is minimized.

Perceptron Loss: defined per each sample and over all possible configurations of outputs takes the following form

$$\mathcal{L}_{perceptron}(W,X^i,\mathcal{Y}) = E(W,X^i,Y^i) - \min_{y \in \mathcal{Y}} E(W,X^i,y) \quad (6)$$

Perceptron loss pulls down the energy of the correct configuration (first term) and it pulls up the energy of current prediction. When the machine prediction is correct, the loss is equal to zero and positive otherwise. One can see that, this loss function does not enforce a margin between correct and incorrect configurations, thus might lead to almost flat surfaces.

A simple binary classification problem can be formulated using perceptron loss as follows.

$$E(W,X^i,Y^i) = -YG_W(X), \quad Y \in \{-1,+1\} \quad (7)$$
$$\mathcal{L}(W,X^i,Y^i) = (sign(G_W(X)) - Y^i)G_W(X^i) \quad (8)$$

where $sign(G_w(X))$ is the result of inference process.

Negative Log-Likelihood Loss: is suitable to train a model to produce probability estimates for $P(Y|X)$:

$$\mathcal{L}_{\text{nll}} = E(W, X^i, Y^i) + \frac{1}{\beta} log \int_y e^{-\beta E(W, X^i, y)} \quad (9)$$

where $\beta \in \mathcal{R}^+$. As in maximum likelihood solutions for probabilistic models, the integral in the second term of equation 9 might be intractable to compute or might not have an analytical solution in most cases. Approximate solutions to this integral can be obtained by approximate analytical solutions, sampling methods and variational methods.

Contrastive Loss: enforces a gap between the energy of correct answer Y^i and the energy of the *most offending incorrect answer* $\bar{Y}^i$, defined as the *wrong* answer with the smallest energy:

$$\mathcal{L}_{\text{contrast}} = M(E(W, X^i, Y^i), E(W, X^i, \bar{Y}^i)) \quad (10)$$

where M is an increasing function of its first argument and a decreasing function of its second argument. The most commonly used contrastive loss is the hinge loss:

$$\mathcal{L}_{\text{hinge}} = max(0, m + E(W, X^i, Y^i) - E(W, X^i, \bar{Y}^i)) \quad (11)$$

where m is the positive margin. The model is updated whenever the energy of incorrect answer is less than m larger than energy of correct answer. Another type of contrastive loss (with infinite margin) is the log loss:

$$\mathcal{L}_{\text{log}} = log(1 + e^{E(W, X^i, Y^i) - E(W, X^i, \bar{Y}^i)}) \quad (12)$$

1.2 Modules in Energy-Based Learning

Until now, we have stated that the energy function is parametrized by W. In most supervised scenarios, this is conveniently achieved by using a functional module $G_W(X)$ which maps samples from input space to output space, and by capping it with a "distance" module that measure the discrepancy between $G_W(X)$ and the output Y. A very simple example of a functional module is a single matrix multiplication that projects the input along the space defined by its columns. In more complicated cases it can be combination of linear and nonlinear functions. Common modules include linear modules as explained, simple nonlinear modules that are applied on each element of the input state independently, like sigmoid functions, and convolutional modules that are very similar to linear modules, but are applied as convolution operations on input image maps. The loss function $\mathcal{L}$ is minimized with respect to the parameters W of the functional modules.

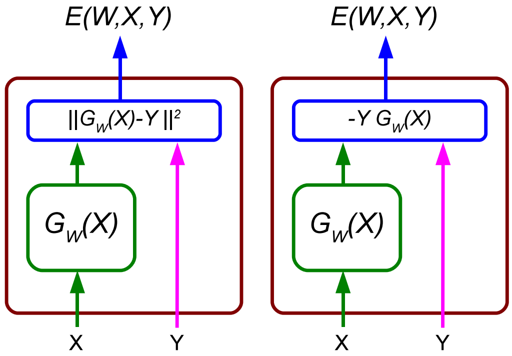

Figure 3. Architectures for Energy Based Model: Left: Regression can be formulated by using a squared distance energy function combined with energy loss and a module $G_w(X)$. Right: Two class classification can be formulated similarly using perceptron loss.

1.3 Architectures for Energy-Based Learning

In this section we provide *EBL* models for some widely used learning algorithms.

Regression: is one of the most common algorithms used in machine learning. A regressor model (Fig. 3) can be obtained by using a squared error energy function

$$E(W, X^i, Y^i) = \frac{1}{2}||G_W(X^i) - Y^i||^2 \quad (13)$$

together with energy loss. When G_W is a linear operator, this model becomes equivalent to solving the least squares problem.

Two-Class Classification: can be formulated using a simple energy function as shown in Figure 3.

$$E(W, X^i, Y^i) = -Y^i G_W(X^i) \quad (14)$$

Any of the perceptron loss, hinge loss or negative log likelihood loss can be used with this energy function to solve two class classification problems.

Multi-Class Classification: can be done by replacing the energy function in Figure 3 with

$$G_W = [g_1 \; g_2 \; \ldots \; g_c] \quad (15)$$

$$E(W, X^i, Y^i) = \sum_{k=1}^{c} \delta(Y^i - k) g_k \quad (16)$$

where $\delta(u)$ is Kronecker delta function. As with the two-class classification problem, perceptron, hinge and negative log likelihood loss functions can be used.

One can imagine that complicated architectures can be built by combining several functional modules and energy functions as long as the combined energy function can be minimized with respect to the desired outputs Y and the

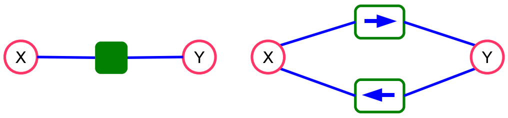

Figure 4. **Modeling Factor Graphs:** *Left*: A simple factor graph showing an observed variable X and a latent variable Y. The factor node models the compatibility constraint between the two states. *Right:* The factor node can be divided into two directional nodes representing compatibility between one node and transformation of the other.

final loss function can be minimized with respect to the parameters W.

It has to be noted that, any factor graph can also be modeled using energy based learning. In a simple factor graph an observed state is connected to a latent state through a factor node which models the constraint in between two states as shown in left of Figure 4. The combination of two states are assigned high likelihood under the compatibility constraint defined by the factor node. One can also separate the dependency constraints between two states into two directional factor nodes representing the compatibility between one node and transformation of the other.

Most unsupervised learning algorithms can be modeled in this framework. In Figure 5, we show several common unsupervised learning algorithms. *PCA* is a linear model, where the transformation from observed input X to latent representation Y is a linear projection. Accordingly, transformation from Y to input space X' is also a linear projection. The model has to be trained under the reconstruction compatibility constraint such that transformed reconstruction X' has to minimize the squared reconstruction error between original input X and projected input X'. *Auto-encoder neural networks* are very similar to PCA, except the projection from input X to latent variable Y is non-linear. *Sparse Decomposition* [14] is a uni-directional model where, there is no direct projection form input X to latent variable Y. Instead, for each input X, the system has to carry out an optimization process to infer latent representation Y. In addition to reconstruction compatibility constraint, the latent representation has to minimize the L_1 norm constraint. *Predictive Sparse Decomposition* [9] is an extension to sparse coding models, where a nonlinear predictor function is also trained to infer latent variable Y from input X without requiring any optimization process.

In the next sections, the *EBLearn* open source machine learning library will be introduced by demonstrating simple coding examples. First, an overview of the underlying tensor library will be provided. Then, the details for the functional modules will be introduced. Finally, several complementary modules will be explained.

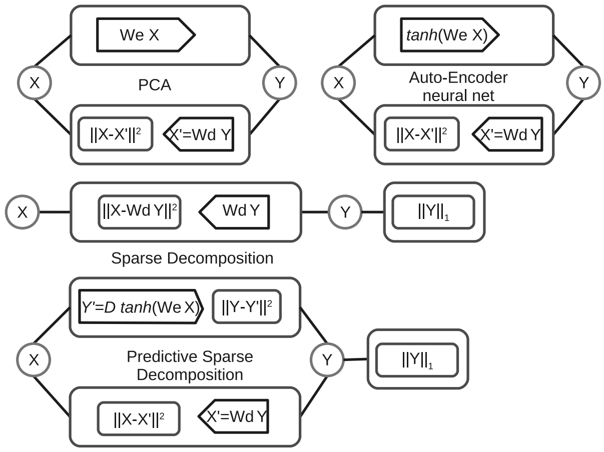

Figure 5. **Modeling Unsupervised Learning Algorithms:** Many unsupervised learning algorithms can be represented in the factor graph model and trained with energy based learning.

2. *libidx*: Tensor Descriptors and Operators

The idx library (or $libidx$) provides convenient and efficient tensor (multi-dimensional arrays) manipulations, used as a basis for the eblearn library. There are three main components to the library: tensor descriptors and iterators, content operators and image-specific operators.

2.1 *idx*: Tensor Descriptors

The idx class can be thought as a tensor pointer to a chunk of memory (or an srg class, srg standing for storage). One idx could describe a tensor held by that entire memory storage, and another could describe a tensor that is a subset of it.

Thus an idx describe a tensor via the storage (srg) it points to, its offset from the beginning of the storage, its number of dimensions (or order), the size of each dimension, and the memory stride of each dimension. Declaring a new idx will allocate and initialize to zero a new storage of size and type specified to the constructor and class template. Here for example, we create a 3-dimensional tensor with double precision of size 32x32x3, which could also be interpreted as a 32x32 RGB image:

```
idx<double> t(32, 32, 3);
```

Being relatively cheap memory and computationaly wise, an idx can be manipulated efficiently like a tensor pointer without affecting the actual tensor memory. For example, the user can select at no cost the entire slice at position p of the d^{th} dimension of a tensor:

```
idx<double> slice = t.select(d, p);
```

Note that the tensor $slice$ is now a 2-dimensional tensor (or matrix). Similarly, the user can narrow a dimension d to size s starting at position p to create a 3-dimensional subset of the 3d tensor t:

```
idx<double> subset3d = t.narrow(d, s, p);
```

One can also change the order of the dimensions, e.g. 3x32x32, via the transpose method. The elements of the tensor are accessed via the get and set methods.

The srg class is self garbage-collected by keeping a reference counter of the idx pointing to it. This means that a storage will survive as long as an idx is pointing to it and self destruct otherwise. Requiring no memory management, idx manipulations are hence facilitated and flexible.

2.2 idx loops: Tensor Iterators

While tensor elements can be accessed individually via set and get methods, one will mostly need to loop over entire dimensions or entire tensors. For that effect, iterator classes can be used for each tensor to iterate on. However looping macros make it even easier allowing to loop over multiple tensors of any type but of the same size at the same time.

For instance, the idx_aloop2 macro loops over all elements of 2 tensors, the idx_bloop3 macro loops over the first dimensions of 3 tensors while the idx_eloop1 loops over the last dimension of 1 tensor. For each tensor to be iterated, one must specify in order the name of the new lower-order tensor, the name of the original tensor, and its type. Thus to compute the sum of multiple tensors we could write:

```
idx<double> td3d(32, 32, 3);
idx<int>    ti2d(32, 32);
int total = 0;
idx_bloop2(td2d, td3d, double, ti1d, ti2d, int) {
  idx_bloop2(td1d, td2d, double, ti0d, ti1d, int) {
    total += ti0d.get();
    idx_bloop1(td0d, td1d, double)
      total += (int) td0d.get(); }}
```

Or simply:

```
idx_aloop1(td0d, td3d, double) total += td0d.get();
idx_aloop1(ti0d, ti2d,    int) total += ti0d.get();
```

2.3 Tensor Operators: Content Manipulations

While idx descriptors are inexpensive pointers, the idx content operators work with the actual tensor data (also allowed to modify it) yielding more expensive operations. We now describe a few important operators among others.

- **Copy operator:** copy the content of idx $d1$ to $f2$ (they must have the same dimensions), automatically casting the source type into the destination type:

```
idx<double> d1(32, 32, 3);
idx<float>  f1(32, 32, 3);
idx_copy(d1, f1);
```

- **I/O operators:** save or load tensors :

```
save_matrix(f1, "im.mat");
idx<float> f2 = load_matrix<float>("im.mat");
```

- **Additions, summations operators:** add two tensors into another or compute the sum of all elements:

```
idx_add(f1, f2, f1);
float sum = idx_sum(f1);
```

- **Product operators:** the dot product between two tensors or the matrix-vector multiplication:

```
float dot = idx_dot(f1, f2);
idx<float> f3(32, 16), f4(16), f5(32);
idx_m2dotm1(f3, f4, f5);
```

- **Non-linearity operators:** apply the hyperbolic tangent function to all elements of $t1$ and put the results in $t2$:

```
idx_tanh(f1, f2);
```

2.4 Tensor-based Image Operators

In libidx, images are seen and can be manipulated like tensors. We present here some key operators specific to image tensors.

- **Image I/O operators:** load or save images:

```
idx<float> im = load_image<float>("im.jpg");
save_image(im, "im.png");
```

- **Image Resizing:** with bilinear interpolation:

```
im = image_resize(im, 16, 16);
```

- **Image filtering:** local, global or Mexican-hat normalization:

```
idx<float> im2 = idx_copy<float>(im);
image_global_normalization(im);
image_local_normalization(im, im2, 9);
image_mexican_filter(im2, im, 5, 9);
```

3. *libeblearn*: Energy-Based Learning

The $libeblearn$ library is mainly constituted of modules of two types: $module\_1\_1$ which takes 1 input and produces 1 output and $module\_2\_1$ with 2 inputs and 1 output (Fig. 6). In particular for an *EBL* model, we derive the ebm_2 module from $module\_2\_1$ to output an energy from its 2 inputs. Those two models are the basis for all modules in the library. Consequently, elementary blocks are easily assembled in any possible combination to form elaborate blocks that can again be combined to reach any level of complexity. This modularity provides a great flexibility allowing users to combine their own modules with preexisting modules.

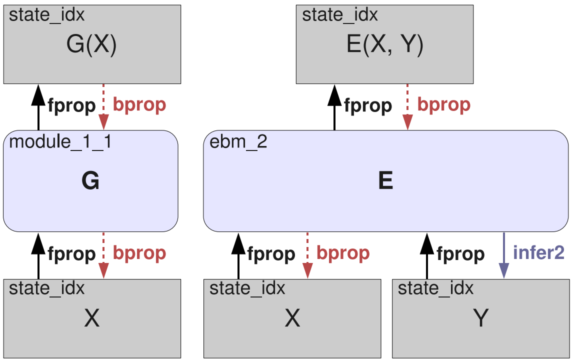

Figure 6. **The two basic types of modules.** $module\_1\_1$ (left) has 1 input and 1 output and ebm_2 has 2 inputs and 1 energy output. The dashed *bprop* method is used during training only while the $infer2$ method is used during inference only ($fprop$ is used for both training and inference). The intermediate results between modules are contained by $state\_idx$ objects (top and bottom), which store the results of calls to $fprop$, *bprop* and *bbprop* methods.

Each module implements the $fprop$ (forward propagation), *bprop* (backward propagation) and *bbprop* (usually back propagation of second derivatives) methods. $module\_2\_1$ also implements the $infer2$ method (infer second input). While *bprop* and *bbprop* methods are only used during training and $infer2$ during inference, the $fprop$ method is used during both phases. Intermediate results of $fprop$, *bprop* and *bbprop* calls are held between modules in $state\_idx$ objects.

We now demonstrate some examples starting with pre-existing $libeblearn$ modules. In the following sections we first show how to build a simple linear regression architecture, then we construct a full vision system in a few lines using *libeblearn* and the *NORB* dataset [2].

3.1 Example: Linear Regression

To build a linear regressor modeled by a linear least square function, one can simply stack a linear module with a bias module, encapsulate them with an euclidean energy module and train this machine with the energy loss module (see resulting architecture in Fig. 7):

```
parameter<double> w;
layers<double> linear(true);
linear.add(new linear_module(w, n_in, n_out);
linear.add(new bias_module(w, n_out));
euclidean_energy<double, int> eenergy;
machine<double, float> E(linear, eenergy);
energy_loss eloss;
supervised_trainer<double, float> trainer(E, eloss);
```

where w are the trainable weights of the machine, n_in and n_out are the number of inputs and outputs respectively,

[2]http://www.cs.nyu.edu/~ylclab/data/norb-v1.0

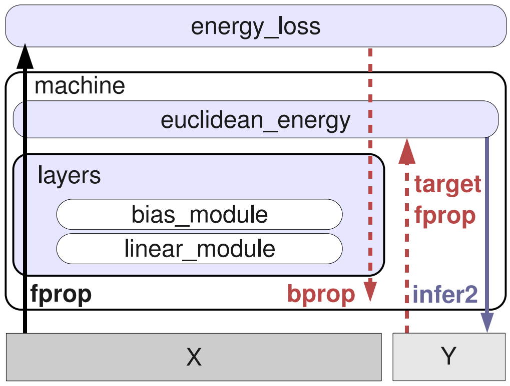

Figure 7. **Linear regression architecture.** A linear and bias modules are stacked together in the $layers$ class, encapsulated in a $machine$ with an euclidean energy module. During training, the $energy\_loss$ module is added to obtain least square minimization with the euclidean energy.

$double$ and $float$ are the precision of respectively the machine and the targets Y.

3.2 Example: a Vision System

In this example, we build, train and execute in a few lines of code a convolutional neural network capable of object recognition as in [6]. The machine is a stack of convolution, subsampling and fully-connected modules (Fig. 8) that takes an image as input and classifies it as belonging to one of five categories. We now describe the construction of that system:

1. Build $E(W, X, Y)$, using a lenet7 neural network as $G_W(X)$ and an euclidean energy module as energy function (see Fig. 8 for corresponding architecture):

```
parameter<double> W;
layers<double> l7(true);
l7.add(new convolution_layer(W, 5, 5, 1, 1,
                             full_table(1, 8)));
l7.add(new subsampling_layer(W, 4, 4, 4, 4, 8));
l7.add(new convolution_layer(W, 6, 6, 1, 1,
                             random_table(8, 24, 4)));
l7.add(new subsampling_layer(W, 3, 3, 3, 3, 24));
l7.add(new convolution_layer(W, ki2, kj2, 1, 1,
                             full_table(24, 100)));
l7.add(new full_layer(W, 100, 5));
euclidean_energy<double, int> eenergy;
machine<double, int> E(l7, eenergy);
```

where the numbers are kernels sizes, strides and output sizes. Those depend on the type of input and output and the complexity of the task to learn. The *full_table* and *random_table* functions provide full or sparse random connections between layers. One can use a shorter equivalent to the previous code:

```
parameter<double> W;
```

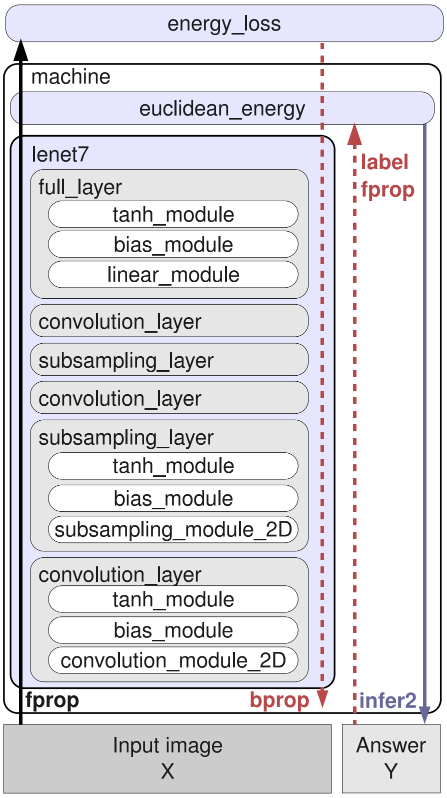

Figure 8. A vision architecture (lenet7). This *machine* combines a euclidean energy with a convolutional neural network of 6 layers called *lenet7*. During both training and inference, the machine is first evaluated with *fprop*. Then for training only (dashed lines), an energy loss module comes on top of the machine and uses the training label to back-propagate (*bprop*) the gradient of the loss through the entire machine. During inference however, the loss module is not used and the answer is inferred via *infer2* following an *fprop*.

```
lenet7<double> l7(W);
euclidean_machine<double, int> E(l7);
```

2. Build the loss and the trainer:

```
energy_loss eloss;
supervised_trainer<double, int> trainer(E, eloss);
```

3. Train the system with the NORB dataset and a learning rate of 0.0001:

```
norb_datasource ds("/datasets/norb");
gd_param p(0.0001);
trainer.train(ds, p);
```

4. Execute the system:

```
idx<double> image = load_image<double>("im.jpg");
state_idx<double> input(image);
int answer = E.infer2(input);
```

When the input image produces a multi-dimensional output (instead of a single output), use the multi-scale and multi-dimensional detector (see Fig. 9) to obtain a vector of bounding boxes around detected objects:

```
detector d(E, scales_number);
vector<bbox> answers = d.fprop(input);
```

Figure 9. Object detection example, using the *detector* class with the *lenet7* architecture build in section 3.2 and trained on the NORB dataset to classify objects between 5 categories: animal, car, human, plane, truck. Einstein is correctly classified by the bounding box as human (left). The internal states and kernels of the machine are shown on the right, from the inputs on top to the 5 category outputs (zoomed) at the bottom.

4 Complementary Tools

The EBLearn project contains in addition to the core libraries *libeblearn* and *libidx* a complete set of tools facilitating development around them:

- *libidxgui*: a cross-platform General User Interface (GUI) library providing tensor-based display functions.
- *libeblearngui*: a GUI library containing display classes for each learning module (See Fig. 9).
- *libeblearntools*: a set of tools to create formatted datasets from image directories, visualize datasets, and automatically train learning machines with different training configurations.
- *tester*: a unit-tester of EBLearn functionalities allows developers to contribute safely to the project.
- demos: EBLearn contains a set of small demonstration projects that users can take inspiration from.

5. Conclusion

Energy based learning has been used in many different contexts of machine learning and provide a very efficient and flexible framework. We have showed that many supervised and unsupervised learning algorithms and factor graphs can be modeled using energy based learning framework. Inference and learning processes are formulated for many popular problems. More importantly, in this work we have presented an open source machine learning library (*EBLearn*) that can be used to built energy based learning models. *EBLearn* is developed using *C++* programing language for maximum portability and flexibility. We have also shown several code examples on how to use *EBLearn* and several additional graphical display methods and image processing methods that are also included.

With this work we introduce the availability of an open source machine learning library that can be used to train supervised, semi-supervised and unsupervised models. We believe this library contains one of the most extensive collection of machine learning algorithms.

References

[1] S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Proc. of Computer Vision and Pattern Recognition Conference*. IEEE Press, 2005.

[2] S. Chopra, T. Thampy, J. Leahy, A. Caplin, and Y. LeCun. Discovering the hidden structure of house prices with non-parametric latent manifold model. In *Proc. Knowledge Discovery in Databases (KDD'07)*, 2007.

[3] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *International Conference on Machine Learning, ICML*, 2008.

[4] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'06)*. IEEE Press, 2006.

[5] R. Hadsell, P. Sermanet, M. Scoffier, A. Erkan, K. Kavukcuoglu, U. Muller, and Y. LeCun. Learning long-range vision for autonomous off-road driving. *Journal of Field Robotics*, 26(2):120–144, February 2009.

[6] F.-J. Huang and Y. LeCun. Large-scale learning with svm and convolutional nets for generic object categorization. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'06)*. IEEE Press, 2006.

[7] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*. IEEE, 2009.

[8] K. Kavukcuoglu, M. Ranzato, R. Fergus, and Y. LeCun. Learning invariant features through topographic filter maps. In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR'09)*. IEEE, 2009.

[9] K. Kavukcuoglu, M. Ranzato, and Y. LeCun. Fast inference in sparse coding algorithms with applications to object recognition. Technical report, Computational and Biological Learning Lab, Courant Institute, NYU, 2008. Tech Report CBLL-TR-2008-12-01.

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[11] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang. A tutorial on energy-based learning. In G. Bakir, T. Hofman, B. Schölkopf, A. Smola, and B. Taskar, editors, *Predicting Structured Data*. MIT Press, 2006.

[12] P. Mirowski and Y. LeCun. Dynamic factor graphs for time series modeling. In *Proc. European Conference on Machine Learning (ECML'09)*, 2009.

[13] P. Mirowski, Y. LeCun, D. Madhavan, and R. Kuzniecky. Comparing svm and convolutional networks for epileptic seizure prediction from intracranial eeg. In *Proc. Machine Learning and Signal Processing (MLSP'08)*. IEEE, 2008.

[14] B. A. Olshausen and D. J. Field. Sparse coding with an overcomplete basis set: a strategy employed by v1? *Vision Research*, 37:3311–3325, 1997.

[15] M. Ranzato, Y. Boureau, and Y. LeCun. Sparse feature learning for deep belief networks. In *Advances in Neural Information Processing Systems (NIPS 2007)*, 2007.

[16] M. Ranzato, F. Huang, Y. Boureau, and Y. LeCun. Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'07)*. IEEE Press, 2007.

[17] M. A. Ranzato and M. Szummer. Semi-supervised learning of compact document representations with deep networks. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 792–799, New York, NY, USA, 2008. ACM.

[18] Y. W. Teh, M. Welling, S. Osindero, G. E. Hinton, T. won Lee, J. francois Cardoso, E. Oja, and S. ichi Amari. Energy-based models for sparse overcomplete representations. *Journal of Machine Learning Research*, 4:2003, 2003.