DBNsim

Giorgio Giuffrè

Contents

0 Abstract

DBNsim is a web application for training and analysing **Deep Belief Networks** ("DBNs", for short). A Deep Belief Network (`http://www.scholarpedia.org/article/Deep_belief_networks`) is a particular type of artificial neural network; it is a stack of Restricted Boltzmann Machines ("RBMs", for short) that can be trained with an algorithm known as **contrastive divergence**. RBMs learn to reconstruct a given input and, if properly analysed, they can reveal latent features in the distribution of the input.

DBNsim offers a simple interface for defining the architecture of a DBN, train it on a remote web server (optionally equipped with a GPU), analyse it, and then download the DBN in various formats for performing further analyses.

0.1 How to run it on your machine

As said, DBNsim is a web app. That means you can run it on one computer and access it on another. According to your needs, there are several ways in which you can use DBNsim:

1. If you want to use it locally (i.e. as if it was a normal desktop application on your PC), you just have to launch the app from the command line with Python; then, you will be able to access the user interface on your browser at the local address `http://127.0.0.1:8000`.

2. You may want your students to use it in a classroom. For this, you just have to launch the app from the command line and ask your audience to connect their devices to the University network; they will find the user interface at `http://your.public.ip.address:8000`.

3. Finally, if you don't have a local intranet like above, you can still publish DBNsim on a remote web server: it will then be available from any computer connected to the internet.

In all three cases, you can follow the step-by-step guide in the next section.

Launching DBNsim from a computer equipped with a CUDA-enabled GPU[1] lets you train a DBN directly on the GPU; DBNsim should automatically detect whether or not your computer has a GPU. The main advantage of using it is speed: you will be able to train much larger networks in fewer time.

0.2 How to contribute

If you would like to improve this project and/or report a bug, please feel free to e-mail Giorgio Giuffrè at `[first_name]giuffre23@gmail.com`, replacing `[first_name]` with my first name. The source code is freely available on GitHub at `https://github.com/ggiuffre/DBNsim`.

1 Installing DBNsim

1.1 Requirements

For running DBNsim on you own server you will need:

[1] See `https://en.wikipedia.org/wiki/CUDA`

- Some knowledge of the **command line interface** of your machine;
- An intepreter for **Python 2.7** (not Python 3);
- the **pip** package manager for Python 2, which should be shipped with a standard Python installation;
- optionally (for GPU support) the NVIDIA drivers for CUDA.

Please note that you have to use Python 2 (preferably 2.7), not Python 3! This is beacause, unfortunately, DBNsim currently depends on two libraries that are only available for Python 2.

If you have installed Python under Windows but your command line says it doesn't recognize the commands `pip` and `python`, then you just have to add Python to your path. This is generally done by going to *System Properties / Advanced / Environment Variables*.

1.2 Basic installation

For installing DBNsim (both for CPU and GPU usage) follow these steps:

```
git clone https://github.com/ggiuffre/DBNsim.git
cd DBNsim
pip install --user -r requirements.txt
```

In addition, it is recommended to install *scipy*, with the command `pip install -user scipy`. Windows user may find it difficult but this is currently an optional dependency; the only thing is that without it you won't be able to handle MATLAB files as an administrator (see here).

If you don't have Git installed, you can still go to the GitHub repo, click "Clone or Download", then "Download ZIP"; now extract the archive, and then `cd` into the extracted directory.

1.3 Adding GPU support

If you have a CUDA-enabled GPU, you can install CUDAMat. To install it, first exit the `DBNsim` directory (this is very important) — a simple way to exit the directory is to `cd` into your home directory; then execute the following commands:

```
git clone https://github.com/cudamat/cudamat.git
cd cudamat
pip install --user .
```

As before, if you don't have Git installed you can download a zipped version of CUDAMat from GitHub.

If you get errors while installing with `pip`, make sure you have the CUDA drivers installed, along with the NVIDIA CUDA compiler (nvcc). If you're using Ubuntu, then you can install them with:

```
sudo apt-get install nvidia-cuda-toolkit
```

After you manage to install CUDAMat, remember to test it. To do this, exit the `cudamat` directory (this is very important), enter a Python interpreter and type:

```
import numpy as np
import cudamat as cm

cm.cublas\_init()

a = cm.CUDAMatrix(np.random.rand(32, 256))
b = cm.CUDAMatrix(np.random.rand(256, 32))

c = cm.dot(a, b)
d = c.sum(axis = 0)

print(d.asarray())
```

1.4 Launching DBNsim

Finally, launch DBNsim with:

```
cd path/to/DBNsim
python DBNsite/manage.py runserver
```

... this actually launches DBNsim only *locally*. Instead, if you want other users to access it, type:

```
cd path/to/DBNsim
python DBNsite/manage.py runserver your.public.ip.address:8000
```

You should see something like:

```
Performing system checks...

loading data from pickle...
loading data from pickle...
loading data from pickle...
loading data from pickle...
System check identified no issues (0 silenced).
June 22, 2017 - 16:02:15
Django version 1.11.2, using settings 'DBNsite.settings'
Starting development server at http://your.public.ip.address:8000/
Quit the server with CONTROL-C.
```

This means the server is up and running. Now you should find the user interface at `http://127.0.0.1:8000` if you're running the app locally. Or, if you're publishing it to a network, you should find it at `http://your.public.ip.address:8000`. 8000 is just an example — you can run DBNsim on any port you have access to.

Once you're on the page, you will be asked for a **password**. If the server administrator didn't change the default configuration, the password is *user*.

When you want to shut down the server, return to the command line interface and press CONTROL-C.

1.5 Changing the default configuration

As the server administrator, you have more power than normal users. It is possible to configure DBNsim under the following aspects:

- you can edit the available training datasets, removing them and/or adding new ones;
- you can set a password that the users have to know in order to use the app;
- you can alter the default values of the hyper-parameters (e.g. you can set a default value of 0.5 for the learning rate, if the user doesn't change it on the form);
- you can change some graphical parameters in the user interface.

For more about how to change the configuration, please read here.

2 Building a network

2.1 What do we need for building a DBN

Creating a DBN means specifying two things:

- Its **architecture** — how many layers of neurons the network will have, and how many neurons each of these layers will have.
- The weights of its **connections** — the initial values of the weighted connections between neurons.

2.2 Architecture

For defining the architecture, look at the form on the upper-left corner. The "DBN architecture" fieldset allows you to specify the number of (hidden) layers in the DBN, and then to specify how many neurons each layer must have. *h1*, *h2* etcetera are the hidden layers, with number 1 being the hidden layer of the first RBM and so on.

You will notice that you cannot change directly the number of neurons in the visible layer: instead, when you want to change it, you must operate on the "Dataset info" fieldset (above). This is because the number of visible units can be deduced from the length of each example in the training dataset: if a

DBN wants to learn from a dataset where each example is 900 pixels (a 30x30 flattened image), then the visible layer must have 900 units.

While you build your DBN, a symbolic representation of its architecture is painted automatically on the upper-right corner of the screen. This is a graph showing the layers of the DBN, with the lowest one being the visible layer.

2.3 Connections

Now let's specify the initial values of the connections. We initialize the weights of the DBN to a random value picked from a normal distribution; the mean of this distribution is always zero, but you can choose the standard deviation by editing the "std. dev." field in the "Hyper-parameters" fieldset.

For seeing the result, click on the "init" button at the right of the standard deviation field — a new DBN will be created on the server. Now try to click on the connections between two layers: a histogram should appear on the bottom-center part of the screen, representing the distribution of the weights that you have just clicked. For example, if you click on the connections between the visible layer and the first hidden layer, you will get a histogram of the weights of the first RBM.

3 Training a network

3.1 Setting the hyper-parameters

When you have defined the architecture and the weights of a DBN, you are ready to train it. You can train it manually stepping one epoch at a time, or automatically if you just want a fully trained network. In both cases, the first thing to do is to set the training **hyper-parameters**.

Hyper-parameters are those numbers and coefficients with which you can "customize" the training algorithm. They are called *hyper* to distinguish them from the weights of the DBN, that are sometimes called "parameters". For instance, Contrastive Divergence depends on the following hyper-parameters:

- **Maximum number of epochs**. One epoch is the time in which a network sees all the examples in a training set; the number of epochs is the number of opportunities that a DBN has of observing each example.
- **Size of a mini-batch**. The training set can be divided in little subsets, known as mini-batches; the DBN will update its weights only after having seen *all* the examples in a mini-batch; this allows for a faster and more precise learning. The size of a mini-batch **must** divide the number of total examples. Note that a mini-batch size of 1 is equivalent to online training; in contrast, setting the size to the exact number of examples is equivalent to batch training, or "one shot" training.
- **Learning rate**. The learning rate is a coefficient in the range [0, 1] that models the plasticity of the network, i.e. how easy it is to update the weights of the DBN.
- **Momentum**. If a DBN learning from a dataset was a ball rolling down a mountain to reach the valley (the minimum) the momentum would simulate

gravity, i.e. the acceleration that speeds the ball proportionally to the steepness of the descent. The momentum ranges from 0 to 1.

- **Weight decay factor**. The weight decay factor (in the range [0, 1]) makes the increase of big weights more difficult than the increase of little weights. This avoids increasing the weights indefinitely.
- **Sparsity target**. The sparsity target (in the range (0, 1]) is how sparse we would like the network to be; a network is more sparse when it has fewer active units — when the hidden representations are more localistic.
- **Standard deviation of the weights distribution**. We have already seen this hyper-parameter while building a DBN; this is more related to the DBN than to the training algorithm, but it's nonetheless a hyper-parameter setting the standard deviation of the probability distribution within which the weights are initialized.

At the right of the "std. dev." field you will find a "init" button. It is **not** necessary to initialize the weights before training the network, because they will be initialized automatically anyway. This button is meant to analyse the network before the training.

3.2 Launching the training

When you have chosen the hyper-parameters (or if you want to accept the default ones), you can start training the DBN with one of the two buttons after the "Hyper-parameters" fieldset:

- "**1 epoch**" trains the network for just one epoch: use this button if you want to train the DBN in a step-by-step fashion.
- "**all epochs**" trains the network for the number of epochs that you have specified in the "max. epochs" field.

While training the network, you will see that a chart is updated plotting the reconstruction error against the number of epochs.

4 Analysing a network

4.1 Analysing the DBN architecture

You can analyse a DBN in several ways. The simplest and the first thing to do when you set up a DBN to learn a dataset is analysing its architecture. The architecture of a network is the way in which its neurons are connected; in a DBN, the architecture is defined by:

- the **number of hidden layers**, that is equal to the number of RBMs forming the DBN;
- the **number of units in each layer**.

For a straightforward representation of the architecture, you can look at the graph in the upper-right corner.

4.2 Plotting the reconstruction error

In the bottom-left corner, you can see a plot reporting the reconstruction error for each RBM and for each training epoch. For each DBN that you train, the plot will add one line for each RBM in the DBN. This is meant to compare the reconstruction error of two different RBMs inside one DBN and the plots of two different DBNs; if the chart gets too crowded, you can clean it with the "clean plot" button near the chart.

4.3 Analysing the receptive fields

You can view the receptive fields of some of the neurons in one hidden layer. For this, just choose a layer of the graph in the top-right corner and click it: DBNsim will pick some neurons from the layer (max. 24 neurons) and display their receptive fields.

If you want to see a larger version of a receptive field, simply click on it. A new tab will open, and from there you'll even be able to download the receptive field as an image.

Note that, for consistency, the neurons picked by DBNsim are the same **every time you click on the same DBN**. This allows you to see the changes in one receptive field while the network is learning; for this purpose, the receptive fields are updated automatically during the training, every five epochs.

4.4 Viewing a histogram of the weights in a RBM

To analyse the weights of a RBM in your DBN, you can view a histogram of the weights by clicking between two layers on the graph in the upper-right corner. Clicking between layer n and layer $n+1$ will display a histogram of the weights in the $n+1$th RBM.

As for the receptive fields, the histogram is updated automatically during the training, every five epochs.

4.5 Viewing a training example

Finally, you can visualize a random training example by clicking on the visible layer of the graph in the upper-right corner. Every time you click on the visible layer, a random training example will be picked from the current dataset and displayed under the visible layer.

5 Changing the default configuration

5.1 What you can do as an admin

DBNsim has two different kinds of user: normal users of the web interface and *administrators* with access to the configuration of the server. If you are an admin, here's how you can customize your instance of DBNsim for your users:

- you can edit the available **training datasets**, removing them and/or adding new ones;

- you can set a **password** that the users have to know in order to use the app;
- you can alter the default values of the **hyper-parameters** (e.g. you can set a default value of 0.1 for the learning rate, if the user doesn't change it on the form);
- you can change the number of epochs between each automatic update of the receptive fields and histogram;
- you can change the colors of the graph edges (two different colors: one for training and the other for resting).

Keep reading for more details.

5.2 Datasets

Adding a new dataset is as easy as dragging a file in a special directory in DBNsim. From inside the `DBNsim/` root, this directory is `./DBNsite/DBNlogic/data/`.

Here you can put a dataset in one or more of these 3 formats:

- **CSV**: a CSV file (with extension `.csv`) where each example is a comma-separated row, and each row is separated from the following by a newline.
- **Pickle**: a Python "pickle" file (with extension `.pkl` and encoded with **version 2** of the Pickle protocol) that is the serialization of a Numpy 2D array (matrix) containing each example as a row of the matrix.
- **MATLAB**: a MATLAB file (with extension `.mat`) containing a matrix variable called "data", where each row is a training example.

N.B. if DBNsim doesn't recognize your MATLAB files, try to install *scipy*. This is because *scipy* is currently an optional dependency, for compatibility with Windows. Under UNIX, you can install it with `pip install -user scipy`.

Note that the training sets are (currently) meant for *unsupervised* learning, so they cannot contain labels — unless you want the labels to be interpreted as features. Note that an exception to this behaviour is the MATLAB file, where you can put as many variables as you like as long as there is one matrix named "data"; that is, you *can* actually have the labels in the MATLAB file, but you have to store them in a different variable.

Of course, removing a dataset can be performed by just deleting the corresponding file(s) whose name(s) match the name of the dataset. For example, if you want to remove the Numerosity dataset, make sure to remove `Numerosity.csv`, `Numerosity.pkl` and `Numerosity.mat`.

DBNsim searches for new datasets only at startup: if you want to add a dataset, you must restart the server. The order in which DBNsim searches the `data/` directory for an instance of each dataset is:

1. Pickle file;
2. CSV file;
3. MATLAB file.

This is beacause the Pickle file is much faster to load (for the Python interpreter) than the other two. However, the downside is that its size can be much bigger.

5.3 Password and hyper-parameters

By editing some Python files, you can change the password and the default values of the hyper-parameters.

In order to change the password, open `./DBNsite/DBNtrain/views.py` and change the value of the variable `PASSWORD` to a string of your choice. The default is "user".

For the hyper-parameters, open `./DBNsite/DBNlogic/util.py` and search for the constructor of the `Configuration` class. There you will find the default values for the constructor arguments. These values are the default values of the hyper-parameters that the user sees when he load the main page of the app: feel free to change them. Read the comments in the body of the constructor, if you want to know the meaning of each hyper-parameter.

5.4 graphical parameters

If you want the receptive fields and the weights histogram to be updated every n epochs, open `./DBNsite/DBNtrain/static/dbntrain.js` and look for the variable `chartsUpdateRate`. Set it to a positive value that will be the number of epochs that DBNsim will wait before automatically updating the receptive fields and the histogram during training.

Finally, the edges in the upper-right graph change color according to what they are "doing": if an RBM is learning, its weights will be colored in red; else, they will be blue. If you want to change these colors, open `./DBNsite/DBNtrain/static/dbntrain.js` and change the values of the variables `edgesColor` and `trainingEdgesColor`.

6 Contributing to the source code

6.1 Which files do I have to focus on?

Modifying the source code can seem daunting at first: this repo contains a lot of directories, subdirectories and files. Don't fear: this section will clarify which are the "important" files, i.e. the few files that actually *do* something — most of the files in here exist just because they're required by Django, or because they are static files needed by the HTML interface.

So here are the files that actually count:

- `DBNsim/DBNtrain/views.py` is the Python **interface** between a client and the server; this file, along with `dbntrain.js`, is the one you will be working on most of the time.
- `DBNsim/DBNtrain/templates/DBNtrain/index.html` is the Django HTML template that forms the main interface.
- `DBNsim/DBNtrain/static/dbntrain.js` is the JavaScript source code that brings to life the HTML interface; it contains several JavaScript functions that are event-oriented (but not object-oriented at all).

- `DBNsim/DBNtrain/static/main.css` is (currently) the only CSS stylesheet that affects the HTML interface.
- `DBNsim/DBNlogic/nets.py` is the Python module that manages the **network classes** (currently DBN and RBM).
- `DBNsim/DBNlogic/sets.py` is the Python module that manages the available training datasets.
- `DBNsim/DBNlogic/train.py` is the Python module where strategy objects for training the networks reside.
- `DBNsim/DBNlogic/util.py` is a Python module that stores useful functions needed by the other modules.

6.2 Structure of the repository

To have an idea of the big picture, the repo is structured like this. The `DBNsim` directory contains the source code, while the `docs` directory contains all the documentation needed by the **users**, **administrators** and **developers**. (By "users" we mean the end users of the web app, while the administrators are users of the server (those serving DBNsim from a machine) and developers maintain the source code.)

The `DBNsim` directory contains two main subdirectories, which are actually Python packages (they have a `__init__.py` file that marks them as packages). They are:

- `DBNlogic`, the business logic of the application;
- `DBNtrain`, the application logic that makes DBNsim a web app.

More specifically, `DBNlogic` is a Python package that is totally independent from the rest of the application: you can use it as a command line library for handling and training DBNs, if you like. `DBNtrain` uses `DBNlogic` for satisfying the clients' requests and is itself a Python package.

`DBNtrain` is centered around the `views` module, where the developer has to define a function for each request (URL) that the server can accept; adding a function to `views.py` requires adding a line in `urls.py`, so that Django knows that a particular URL has to be mapped to this new Python function.

6.3 Things to do

If you want some ideas about how to improve DBNsim, here are some:

- Configure DBNsim to run on a **production-stable** WSGI server. The default Django server is not really secure for production.
- Find a more convenient way to manage the jobs run by clients on the server. Currently, DBNsim requires that each client be assigned a random 10-character key, so that the server can update the right training job when the client asks (for example) to perform one training epoch of a job. A much more professional way to do this would be to use **WebSockets**, for example.

- Tidy the `JavaScript` source code, maybe distributing it across two or three distinct JS modules.
- Where possible, replace **jQuery** with vanilla JavaScript, as jQuery was used only to quickly prototype the application but tends to clutter the UI, slowing down the app on older browsers.
- Improve the use of **Cytoscape.js**, maybe adding some calls to `cy.batch` instead of manually iterating through a large number of nodes and edges. Currently, the graph drawing efficiency is reasonable but can definitely be improved.
- Add some more consistency checks to the JS code that handles the HTML forms and the buttons. I set Travis CI to systematically test the Python back end, but there are no automatic tests for the front end, so it isn't possible to systematically catch hidden bugs.
- Repair the "upload DBN" button. Currently, it is only possible to download a DBN, not uploading it.
- Remove **Gnumpy** and try to use another library for GPU computing, as Gnumpy only works for Python 2. After this, upgrading DBNsim to Python 3 should be easy.