Deep Learning on GPU with Theano

Presenting: James Bergstra

Olivier Breuleux
Pascal Lamblin
Guillaume Desjardins
Joseph Turian

Frédéric Bastien
Razvan Pascanu
Dumitru Erhan
Olivier Delalleau

Yoshua Bengio
Design Goals and Motivation

• Combine best of interpreted and compiled:
  - Fast prototyping of interpreted language
  - Fast execution of compiled language (or better?)

• Symbolic computation
  - Automatic differentiation

• Division of labour:
  - Scientist gives generic formulae, puts pieces together
  - Theano optimizes particular inference/computation

• Make use of rich development environment
  - Python, NumPy, SciPy, MatPlotLib

• High-level symbolic program makes GPU use transparent
x = matrix()
y = vector(dtype='int32')
weights = shared(numpy.random.randn(32, 10))
bias = shared(numpy.zeros(10))

p_y = softmax(dot(x,w)+b)
loss = -log(p_y)[arange(y.shape[0]), y]
prediction = argmax(p_y)
gw, gb = grad(loss.mean(), [w, b])

train = function(inputs=[x,y],
                  outputs=[prediction,loss],
                  updates={w:w-0.1*gw, b:b-0.3*gb})

for xdata, ydata in training_set:
    pred, err = train(xdata, ydata)
Theano as Language

Types

• Strided N-D tensors in all C datatypes on CPU
• Strided N-D tensors of float on GPU
• Sparse matrices of all C datatypes on CPU
• Generic Python objects (strings, lists, files, etc.)

Operations

• Arithmetic operators
• Reshaping and dimension manipulations
• Linear algebra
• Elementwise math.h special functions
• Elementwise logical functions
• Convolutions, FFT, downsampling
• Random numbers
• Looping: scan
Theano as Compiler

Graph Optimizations:
• Full use of level-3 BLAS (dgemm, sgemm)
• Judicious use of GPU
• Memory reuse
• Arithmetic special cases, contant folding, loop fusion...
• Numeric stability - e.g. log(1-sigm(x)), x(y/x+z)

Code Optimizations:
• custom code for specific
  - number of dimensions
  - shape
  - stride pattern
Welcome

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Theano features:

- **Light integration with numpy** – Use numpy.ndarray in Theano-compiled functions.
- **Transparent use of a GPU** – Perform data-intensive calculations up to 140x faster than with CPU.
- **Symbolic differentiation** – Let Theano do your derivatives.
- **Speed and stability optimizations** – Get the right answer for \( \log(1+x) \) even when \( x \) is really tiny.
- **Dynamic code generation** – Evaluate expressions faster.
- **Extensive unit-testing and self-verification** – Detect and diagnose many types of mistakes.

Theano has been powering large-scale computationally intensive scientific investigations since 2007. But it is also approachable enough to be used in the classroom (IFT6256 at the University of Montreal).

Download

We recommend the latest development version, available via:

```
hg clone http://hg.assemble.com/theano Theano
```

The theano subfolder should be on your `PYTHONPATH`. For more information about installation and configuration, see [installing Theano](http://www.deeplearning.net/software/theano/).  

Documentation

Roughly in order of what you’ll want to check out:

- [Installing Theano](http://www.deeplearning.net/software/theano/) – How to install Theano.
- [Theano at a Glance](http://www.deeplearning.net/software/theano/) – What is Theano?
- [Tutorial](http://www.deeplearning.net/software/theano/) – Learn the basics.
- [Library Documentation](http://www.deeplearning.net/software/theano/) – Theano’s functionality, module by module.
- [Optimizations](http://www.deeplearning.net/software/theano/) – Guide to Theano’s graph optimizations.
- [Extending Theano](http://www.deeplearning.net/software/theano/) – Learn to add a Type, Op, or graph optimization.
- [Internal Documentation](http://www.deeplearning.net/software/theano/) – How to maintain Theano, LISA-specific tips, and more...
- [API](http://www.deeplearning.net/software/theano/) – The automatically-generated API

You can download the latest [PDF documentation](http://www.deeplearning.net/software/theano/) rather than reading it online.

Check out how Theano can be used for Machine Learning: [Deep Learning Tutorials](http://www.deeplearning.net/software/theano/).

Community

- Register and post to theano-users if you want to talk to all Theano users.
- Register and post to theano-dev if you want to talk to the developers.
- We try to stay organized with [Theano’s Trac](http://www.deeplearning.net/software/theano/).
- Come visit us in Montreal! Most of the developers are students in the LISA group at the University of Montreal.

Batches of 60, random MNIST-sized examples, 3 layers of 1000 units, 10-class classification.
Batches of up to 60, LeNet-style architecture with 96x96 images, 10-way classification.
Batches of up to 60, 1024 x 1024 Restricted Boltzmann Machine, 1-step Contrastive Divergence.
Theano in Summary

- Symbolic complement to NumPy, SciPy
  - very expressive functional language

- Transparent CPU/GPU switch
  - single-precision float (and more to come)

- Graph and Code optimizations
  - fast function evaluation

- Lots of documentation

- Good language for Deep Architectures
Deep Learning Tutorials

Deep Learning is a new area of Machine Learning research, which has been introduced with the objective of moving Machine Learning closer to one of its original goals: Artificial Intelligence. See these course notes for a brief introduction to Machine Learning for AI and an introduction to Deep Learning algorithms.

Deep Learning is about learning multiple levels of representation and abstraction that help to make sense of data such as images, sound, and text. For more about deep learning algorithms, see for example:

- The monograph or review paper Learning Deep Architectures for AI (Foundations & Trends in Machine Learning, 2009).
- The ICML 2009 Workshop on Learning Feature Hierarchies webpage has a list of references.
- The LISA public wiki has a reading list and a bibliography.
- Geoff Hinton has readings from last year’s NIPS tutorial.

The tutorials presented here will introduce you to some of the most important deep learning algorithms and will also show you how to run them using Theano. Theano is a python library that makes writing deep learning models easy, and gives the option of training them on a GPU.

The algorithm tutorials have some prerequisites. You should know some python, and be familiar with numpy. Since this tutorial is about using Theano, you should read over the Theano basic tutorial first. Once you’ve done that, read through our Getting Started chapter – it introduces the notation, and [downloadable] datasets used in the algorithm tutorials, and the way we do optimization by stochastic gradient descent.

The purely supervised learning algorithms are meant to be read in order:

1. Logistic Regression - using Theano for something simple
2. Multilayer perceptron - Introduction to layers
3. Deep Convolutional Network - a simplified version of LeNet5

The unsupervised and semi-supervised learning algorithms are here (the auto-encoders can be read independently of the RBM/DBN thread):

- Auto Encoders, Denoising Autoencoders - description of autoencoders
- Stacked Denoising Auto-Encoders - easy steps into unsupervised pre-training for deep nets
- Restricted Boltzmann Machines - single layer generative RBM model
- Deep Belief Networks - unsupervised generative pre-training of stacked RBMs followed by supervised fine-tuning
Deep Learning is a new area of Machine Learning research, which has been introduced with the objective of moving Machine Learning closer to one of its original goals: Artificial Intelligence. See these course notes for a brief introduction.

1. **Logistic Regression** - using Theano for something simple
2. **Multilayer perceptron** - introduction to layers
3. **Deep Convolutional Network** - a simplified version of LeNet

The purely supervised learning algorithms are meant to be read in order:

- **Auto Encoders, Denoising Autoencoders** - description of auto-encoders
- **Stacked Denoising Auto-Encoders** - easy steps into unsupervised learning
- **Restricted Boltzmann Machines** - single layer generative model
- **Deep Belief Networks** - unsupervised generative pre-training
- **Unsupervised fine-tuning**
Denoising Autoencoders (dA)

Note
This section assumes the reader has already read through Classifying MNIST digits using Logistic Regression and Multilayer Perceptron. Additionally it uses the following Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, Random numbers, floatX. If you intend to run the code on GPU also read GPU.

Note
The code for this section is available for download [here](#).

The Denoising Autoencoder (dA) is an extension of a classical autoencoder and it was introduced as a building block for deep networks in [Vincent09]. We will start the tutorial with a short discussion on Autoencoders.

Autoencoders

See section 4.6 of [Bengio09] for an overview of auto-encoders. An autoencoder takes an input \( x \in [0, 1]^d \) and first maps it (with an encoder) to a hidden representation \( y \in [0, 1]^d \) through a deterministic mapping, e.g.:

\[
y = s(Wx + b)
\]

Where \( s \) is a non-linearity such as the sigmoid. The latent representation \( y \), or code is then mapped back (with a decoder) into a reconstruction \( z \) of same shape as \( x \) through a similar transformation, e.g.:

\[
z = s(W'y + b')
\]

where \( \cdot \) does not indicate transpose, and \( z \) should be seen as a prediction of \( x \), given the code \( y \). The weight matrix \( W' \) of the reverse mapping may be optionally constrained by \( W' = W^T \), which is an instance of tied weights. The parameters of this model (namely \( W, b, b' \) and, if one doesn’t use tied weights, also \( W' \)) are optimized such that the average reconstruction error is minimized. The reconstruction error can be measured in many ways, depending on the appropriate distributional assumptions on the input given the code, e.g., using the traditional squared error \( L(x, z) = ||x - z||^2 \), or if the input is interpreted as either bit vectors or vectors of bit probabilities by the reconstruction cross-entropy defined as:

\[
L_H(x, z) = -\sum_{k=1}^{d} [x_k \log z_k + (1 - x_k) \log (1 - z_k)]
\]

The hope is that the code \( y \) is a distributed representation that captures the coordinates along the main factors of variation in the data (similarly to how the projection on principal components captures the main factors of variation in the data). Because \( y \) is viewed as a lossy compression of \( x \), it cannot be a good compression (with small loss) for all \( x \), so learning drives it to be one that is a good compression in particular for training examples, and hopefully for others as well, but not for arbitrary inputs. That is the sense in which an auto-encoder generalizes: it gives low reconstruction error to test examples from the same distribution as the training examples, but generally high reconstruction error to uniformly chosen configurations of the input vector.

If there is one linear hidden layer (the code) and the mean squared error criterion is used to train the network, the autoencoder is a network that learns a linear ridge regression with weight decay.
```python
self.W_prime = self.W.T
self.theano_rng = theano_rng
# if no input is given, generate a variable representing the input
if input is None:
    # we use a matrix because we expect a minibatch of several examples,
    # each example being a row
    self.x = T.dmatrix(name='input')
else:
    self.x = input

self.params = [self.W, self.b, self.b_prime]

def get_corrupted_input(self, input, corruption_level):
    """This function keeps `1-corruption_level` entries of the inputs the same
    and zero-out randomly selected subset of size `corruption level`
    Note: first argument of theano_rng.binomial is the shape(size) of
    random numbers that it should produce
    second argument is the number of trials
    third argument is the probability of success of any trial
    this will produce an array of 0s and 1s where 1 has a probability of
    1 - `corruption level` and 0 with `corruption level`.
    """
    return self.theano_rng.binomial(size=input.shape, n=1, prob=1 - corruption_level)

def get_hidden_values(self, input):
    """Computes the values of the hidden layer"
    return T.nnet.sigmoid(T.dot(input, self.W) + self.b)

def get_reconstructed_input(self, hidden):
    """Computes the reconstructed input given the values of the hidden layer"
    return T.nnet.sigmoid(T.dot(hidden, self.W_prime) + self.b_prime)

def get_cost_updates(self, corruption_level, learning_rate):
    """This function computes the cost and the updates for one training step of the DA"

    tilde_x = self.get_corrupted_input(self.x, corruption_level)
    y = self.get_hidden_values(tilde_x)
    z = self.get_reconstructed_input(y)
    # note: we sum over the size of a datapoint; if we are using minibatches,
    # L will be a vector, with one entry per example in minibatch
    L = T.sum(self.x*T.log(z) + (1-self.x)*T.log(1-z), axis=1)
    # note: L is now a vector, where each element is the cross-entropy cost
    # of the reconstruction of the corresponding example of the
    # minibatch. We need to compute the average of all these to get
    # the cost of the minibatch
    cost = T.mean(L)

    # compute the gradients of the cost of the `DA` with respect
    # to its parameters
    gparams = T.grad(cost, self.params)
    # generate the list of updates
    updates = {}
    for param, gparam in zip(self.params, gparams):
        updates[param] = param - learning_rate*gparam
    return (cost, updates)
```
Running the Code

To run the code:

```
python da.py
```

The resulted filters when we do not use any noise are:

![Filter example without noise](image)

The filters for 30 percent noise:

![Filter example with 30% noise](image)

- Expected output
- Expected running time
- Tips, Tricks, Caveats
Thank You!

Theano:
http://www.deeplearning.net/software/theano

Deep Learning Tutorials
http://www.deeplearning.net/tutorial/