Deep Learning (hands on)

-bilingual, due to copy-paste- ;-D

Angel Navia Vázquez

MLG/8-07-2013

Librerías

- Theano (tensores, GPU)
- Deeplearning tutorial (sobre Theano): demos para empezar…
- Scikit-learn (no incluye DL actualmente):
  - *Issam issamou01@gm...: "Eventually, scikit-learn would need easy-to-use … Deep learning …"*
  - *Andreas Mueller amueller@ai...: "sklearn will not add any GPU code in the near future. Also, we will probably not use numba in quite a while. … So you want to add a theano-dependency to sklearn? No, sorry, that won't happen either."*
- Cuda-convnet (http://code.google.com/p/cuda-convnet/): fast C++/CUDA implementation of convolutional neural networks
- Pylearn2 (sobre Theano, +yaml?):
  - Autoencoders
  - RBMs
  - Partial implementation of DBMs
- Torch7 (NN package, autoencoders): uses Lua instead of Python (…)

Contenido

- Preparativos y descargas previas
- Técnicas Deep Learning evaluadas:

  (sólo clasificación, no denoising ni reducc. Dim.)

  – Regresión Logística (LR)

  – Multilayer perceptron (MLP), usa LR

  – Convolutional NN (usa MLP y LR)

  – Deep Belief Networks (DBN)

Preparativos

- **Instalar librería de tensores "theano":**
  http://deeplearning.net/software/theano/

- Disponible en Pypi: sudo pip install Theano

- Theano is a Python library that allows you to define, optimize, and efficiently evaluate mathematical expressions involving multi-dimensional arrays. It is built on top of NumPy. Theano features:
  - tight integration with NumPy: a similar interface to NumPy's. numpy.ndarrays are also used internally in Theano-compiled functions.
  - transparent use of a GPU: perform data-intensive computations up to 140x faster than on a CPU (support for float32 only).
  - efficient symbolic differentiation: Theano can compute derivatives for functions of one or many inputs.
  - speed and stability optimizations: avoid nasty bugs when computing expressions such as log(1 + exp(x)) for large values of x.
  - dynamic C code generation: evaluate expressions faster.
  - extensive unit-testing and self-verification: includes tools for detecting and diagnosing bugs and/or potential problems.

Descargar datos y código

- Seguiremos el tutorial sobre DL:

  http://deeplearning.net/tutorial/

- Bajar código fuente y datos (bajar todo como zip):

  https://github.com/lisa-lab/DeepLearningTutorials

También accesible via git...

```
git clone git://github.com/lisa-lab/DeepLearningTutorials.git
```

- Usaremos cualquier entorno python (e.g. Spyder)
- Working directory: “DeepLearningTutorials/code”

Theano

Is as simple as following these simple steps:

1. Declare *variables*, specifying the types
2. Write down the expression in terms of the variables
3. *Compile the function* that will compute the expression
4. Call the function on the intended data
   - Theano provides symbolic differentiation
   - The computational graph representing the gradients is automatically optimized
   - And all this at a single call away: `TT.grad(cost, wrt)`

- Shared variables
  - Think of them as global variables
  - They have a state that stays consistent in between calls
  - Especially useful to keep memory on the same device
  - Can be updated via `set_value` and `updates` provided to `theano.function`

Ejemplo desc. Grad.

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random
N = 400
feats = 784
D = (rng.randn(N, feats), rng.randint(size=N,low=0,
      high=2))
training_steps = 10000
# Declare Theano symbolic variables
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(feats), name="w")
b = theano.shared(0., name="b")
print "Initial model:"
print w.get_value(), b.get_value()
# Construct Theano expression graph
# Probability that target = 1
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))
# The prediction thresholded
prediction = p_1 > 0.5
```

```
# Cross-entropy loss function
xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1)
# The cost to minimize
cost = xent.mean() + 0.01 * (w ** 2).sum()
# Compute the gradient of the cost
gw,gb = T.grad(cost, [w, b])
# Compile
train = theano.function(
inputs=[x,y],
outputs=[prediction, xent],
updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))
predict = theano.function(inputs=[x],
      outputs=prediction)
# Train
for i in range(training_steps):
      pred, err = train(D[0], D[1])
print "Final model:"
print w.get_value(), b.get_value()
print "target values for D:", D[1]
print "prediction on D:", predict(D[0])
```

Si os da problemas el copy-paste, el código está al final de esta página:
http://deeplearning.net/software/theano/tutorial/examples.html

Theano en GPUs

- En theano config (.theanorc) definir [global] device=gpu, floatX = float32

http://deeplearning.net/software/theano/library/config.html

- Código de prueba (http://deeplearning.net/software/theano/tutorial/using_gpu.html):

```
from theano import function, config, shared, sandbox
import theano.tensor as T
import numpy
import time
vlen = 10 * 30 * 768 # 10 x #cores x # threads per core
iters = 1000
rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], sandbox.cuda.basic_ops.gpu_from_host(T.exp(x)))
#f = function([], T.exp(x))
print f.maker.fgraph.toposort()
t0 = time.time()
for i in xrange(iters):
r = f()
t1 = time.time()
print 'Looping %d times took' % iters, t1 - t0, 'seconds'
print 'Result is', r
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.toposort()]):
print 'Used the cpu'
else:
print 'Used the gpu’
```

Regresión logística

$$P(Y = i|x, W, b) = softmax_i(Wx + b) = \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}}$$

$$y_{pred} = \text{argmax}_i P(Y = i|x, W, b)$$

- Abrimos el fichero logistic_sgd.py, en la carpeta "code". Carga por defecto la base de datos **MNIST**. El modelo tiene 28x28=784 entradas y 10 salidas.
- Ejecutar: finaliza en 74 epochs y se para con E=7.48%, 28 segs. (GPU=7.2 segs)

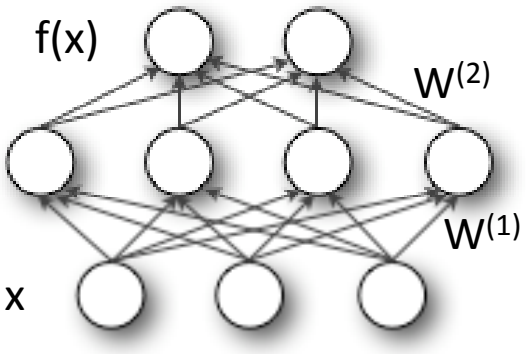
- Optimization complete with best validation score of 7.5%,with test performance **7.48%** The code run for 74 epochs, with 1.936983 epochs/sec
- On an Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00 Ghz the code runs with approximately 1.936 epochs/sec and it took 75 epochs to reach a test error of 7.489%. On the GPU the code does almost 10.0 epochs/sec. For this instance we used a batch size of 600.
- GPU speedup = 10/1.93=5.2

- BEST : committee of 35 conv. net, 1-20-P-40-P-150-10 [elastic distortions] width normalization 0.23% Ciresan et al. CVPR 2012
- SVM, Gaussian Kernel none 1.4%
- Virtual SVM, deg-9 poly, 2-pixel jittered deskewing 0.56% DeCoste and Scholkopf, MLJ 2002)

Regresión logística (II)

```
import theano.tensor as T
x = T.fmatrix('x')
y = T.lvector('y')
# allocate shared variables model params
b = theano.shared(numpy.zeros((10,)), name='b')
W = theano.shared(numpy.zeros((784, 10)), name='W')

# symbolic expression for computing the vector of class-membership probabilities
p_y_given_x = T.nnet.softmax(T.dot(x, W) + b)

#compiled Theano function that returns the vector of class-membership  probabilities
get_p_y_given_x = theano.function(inputs=[x], outputs=p_y_given_x)

# print the probability of some example represented by x_value x_value is not a
    symbolic variable but a numpy array describing the datapoint
print 'Probability that x is of class %i is %f' % (i, get_p_y_given_x(x_value)[i])

# symbolic description of how to compute prediction as class whose probability is
    maximal
y_pred = T.argmax(p_y_given_x, axis=1)

# compiled theano function that returns this value
classify = theano.function(inputs=[x], outputs=y_pred)
```

MLP



$$f(x) = G(b^{(2)} + W^{(2)}(s(b^{(1)} + W^{(1)}x))).$$

$$h(x) = \Phi(x) = s(b^{(1)} + W^{(1)}x)$$

s(x)= $sigmoid(a) = 1/(1+e^{-a})$

G(x)= “softmax”

- Abrimos el fichero mlp.py, en la carpeta “code”. Por defecto n_hidden=500: n_epochs=1000, batch_size=20;
- L 176: modificar parámetros y ejecutar (según CPU):
  - n_hidden=10: n_epochs=100, 6.8% 1.4 minutos (GPU=3.5 mins)
  - n_hidden=20: n_epochs=100, 3.96% 2 minutos (GPU= 4.65 mins)
  - n_hidden=30: n_epochs=100, 3.3% 2.5 minutos (GPU= 3.6 mins)
  - n_hidden=40: n_epochs=100, 2.9% 3.6 minutos (GPU= 3.63 mins)
  - n_hidden=60: n_epochs=100, 2.7% 4.88 minutos (GPU= 3.67 mins)
  - n_hidden=100:n_epochs=100, 2.26% 7 minutos (GPU= 4.1 mins), nh=500, 2.2%, 8.13 mins
  - n_hidden=100:n_epochs=200, 2.13% 14 minutos (GPU= mins)
- Optimization complete. Best validation score of 1.69 % obtained at iteration 2070000, with test performance 1.65% The code for file mlp.py ran for 97.34m
- On an Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz the code runs with approximately 10.3 epoch/minute and it took 828 epochs to reach a test error of 1.65%.
- GPU Speedup= 97.34/828*10.3=1.21

MLP (II)

```
class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, activation=T.tanh):
…
self.output = activation(T.dot(input, self.W) + self.b)
# parameters of the model
self.params = [self.W, self.b]


class MLP(object):
def __init__(self, rng, input, n_in, n_hidden, n_out):

self.hiddenLayer = HiddenLayer(rng = rng, input = input,
                               n_in = n_in, n_out = n_hidden,
                               activation = T.tanh)

self.logRegressionLayer = LogisticRegression(
                                  input=self.hiddenLayer.output,
                                  n_in=n_hidden,
                                  n_out=n_out)
```

MLP (III)

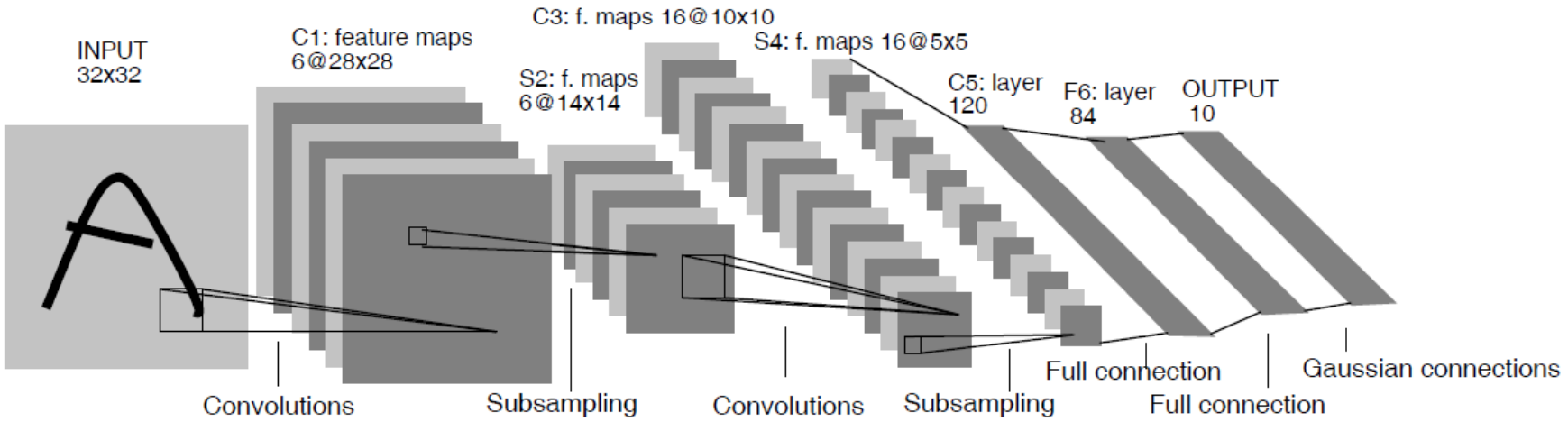
```
Costes:
self.L1 = abs(self.hiddenLayer.W).sum() \
        + abs(self.logRegressionLayer.W).sum()

self.L2_sqr = (self.hiddenLayer.W ** 2).sum() \
            + (self.logRegressionLayer.W ** 2).sum()

# negative log likelihood of the MLP is given by the negative
# log likelihood of the output of the model, computed in the
# logistic regression layer

self.negative_log_likelihood =
   self.logRegressionLayer.negative_log_likelihood

# same holds for the function computing the number of errors
self.errors = self.logRegressionLayer.errors

cost = classifier.negative_log_likelihood(y) \
     + L1_reg * L1 \
     + L2_reg * L2_sqr
```

MLP (IV)

Optimización:

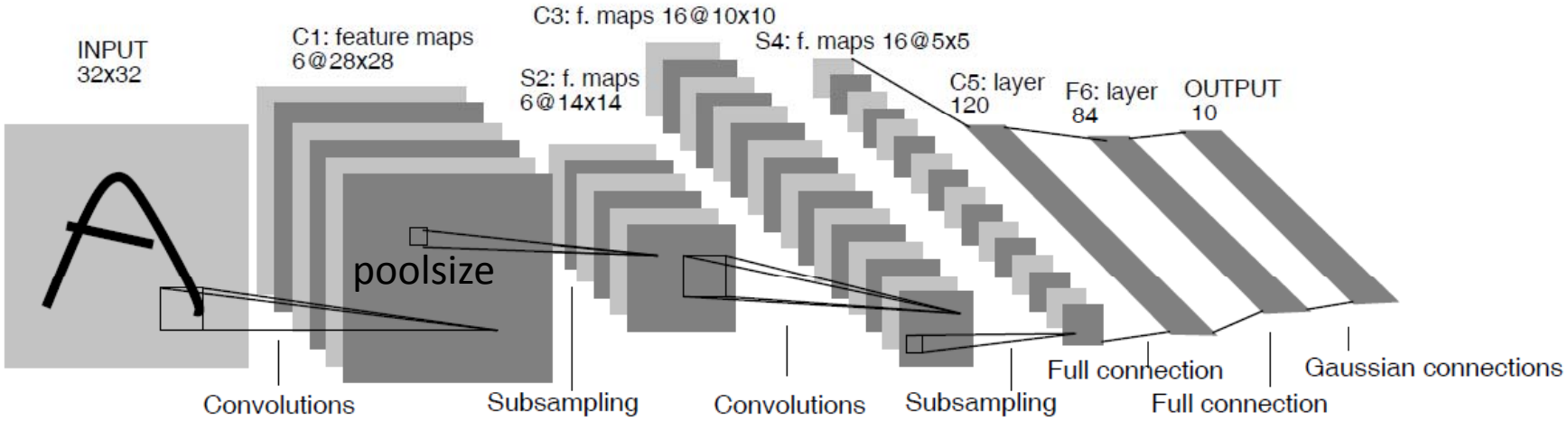
```
# compute the gradient of cost with respect to theta (stored in params)
gparams = []
for param in classifier.params:
    gparam  = T.grad(cost, param)
    gparams.append(gparam)


# compiling a Theano function `train_model` that returns the cost, but
# in the same time updates the parameter of the model based on the rules
# defined in `updates`
train_model = theano.function(inputs=[index], outputs=cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size:(index + 1) * batch_size],
            y: train_set_y[index * batch_size:(index + 1) * batch_size]})
```

Convolutional NN (LeNet5)



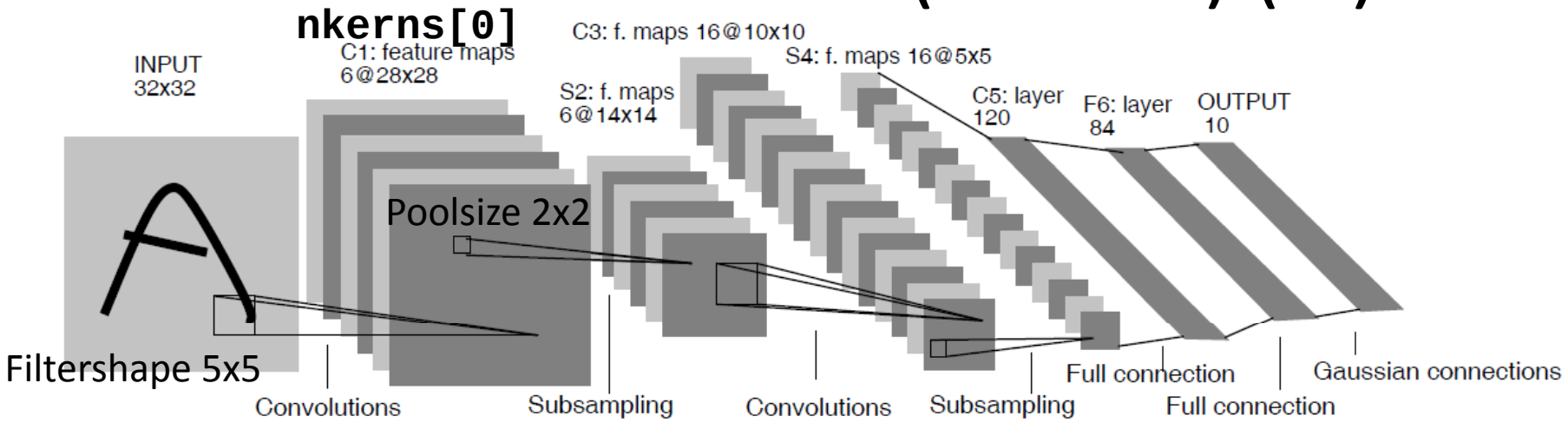
- Abrimos el fichero convolutional_mlp.py, en la carpeta "code". Por defecto n_epochs=200 nkerns=[20,50], n_hidden=500
- L106, 181 (n_out=n_hidden), 184 (n_in=n_hidden) : modificar parámetros y ejecutar (según CPU):
  - n_epochs=10, nkerns=[2,5], n_hidden=10: 6.54% , 5 minutos (GPU=0.13 mins) x38
  - n_epochs=20 nkerns=[2,5], n_hidden=10: 4.26% , 10 minutos (GPU=0.26 mins) x38
  - n_epochs=30 nkerns=[4,10], n_hidden=20: 1.88% , 71 minutos (GPU=0.64 mins) x110
  - n_epochs=30 nkerns=[8,20], n_hidden=40: xxx% , xxx minutos (1.56% GPU= 1.48 mins, %)
  - n_epochs=200 nkerns=[20,50], n_hidden=500: xxx% , xxx minutos (0.93% GPU= 40 mins, %)
- Optimization complete. Best validation score of 0.91 % obtained at iteration 17800,with test performance **0.92%** The code for file convolutional_mlp.py ran for 380.28m
- (GeForce GTX 480): Best validation score of 0.91% obtained at iteration 16400,with test performance **0.93%** The code for file convolutional_mlp.py ran for 32.52m
- GPU speedup = 380.28/32.52=11.69

Convolutional NN (LeNet5) (II)



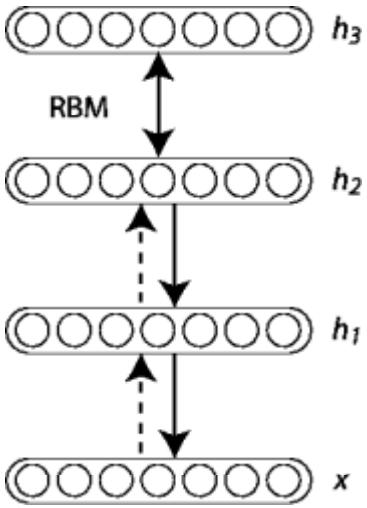

```
class LeNetConvPoolLayer(object):
    def __init__(self, rng, input, filter_shape, image_shape, poolsize=(2,
   2)):

# convolve input feature maps with filters
conv_out = conv.conv2d(input, self.W,
               filter_shape=filter_shape, image_shape=image_shape)
# downsample each feature map individually, using maxpooling
pooled_out = downsample.max_pool_2d(conv_out, poolsize, ignore_border=True)

self.output = T.tanh(pooled_out + self.b.dimshuffle('x', 0, 'x', 'x'))
```

Convolutional NN (LeNet5) (III)




```
layer0_input = x.reshape((batch_size,1,28,28))
layer0 = LeNetConvPoolLayer(rng, input=layer0_input,
            image_shape=(batch_size, 1, 28, 28),
            filter_shape=(nkerns[0], 1, 5, 5), poolsize=(2, 2))
layer1 = LeNetConvPoolLayer(rng, input=layer0.output,
            image_shape=(batch_size, nkerns[0], 12, 12), ##12=(28-5+1)/2###
            filter_shape=(nkerns[1], nkerns[0], 5, 5), poolsize=(2, 2))
layer2_input = layer1.output.flatten(2)
layer2 = HiddenLayer(rng, input=layer2_input, n_in=nkerns[1] * 4 * 4,
                         n_out=500, activation=T.tanh)
layer3 = LogisticRegression(input=layer2.output, n_in=500, n_out=10)
```

Deep Belief Networks



- Abrimos el fichero DBN.py, en la carpeta “code”. Por defecto hidden_layers_sizes=[500, 500], pretraining_epochs=100, training_epochs=1000, hidden_layers_sizes=[1000, 1000, 1000]
- L32, 258, 296 : modificar parámetros y ejecutar (según CPU):
  - pre+training_epochs=10, hidden_layers_sizes=[10, 10, 10] : **21.88%** , 4.7+0.3 (pretraining+finetunig) minutos (GPU=3.77+ 0.92 mins), 14.82%
  - training_epochs=100, hidden_layers_sizes=[100, 100, 100] : **2.28%** , 76+11minutos (GPU=44.2+ 9.2 mins)

- With early-stopping, this configuration achieved a minimal validation error of 1.27 with corresponding test error of **1.34** after 46 supervised epochs.
- On an Intel(R) Xeon(R) CPU X5560 running at 2.80GHz, using a multi-threaded MKL library (running on 4 cores), pretraining took 615 minutes with an average of 2.05 mins/(layer * epoch). Fine-tuning took only 101 minutes or approximately 2.20 mins/epoch.

Conclusiones

- Sobre el problema MNIST:
  - Mejor solución: committee of 35 conv. net, 1-20-P-40-P-150-10 [elastic distortions] width normalization **0.23**% Ciresan et al. CVPR 2012
  - SVM, Gaussian Kernel, **1.4**%
  - Virtual SVM, deg-9 poly, 2-pixel jittered deskewing **0.56**% DeCoste and Scholkopf, MLJ 2002) (SVM+preprocesado "ad hoc")
  - Regresión logística 7.48%
  - MLP: 1.65%
  - Convolutional 0.92%
  - Deep Belief 1.34%
- Las mejores soluciones a MNIST incorporan mucho preprocesado "ad hoc"
- GPUs aceleran mucho conv-NNs, pero no los otros modelos
- DL parece ir bien en este problema (y en otros específicos… Netflix…speech recognition…)
- No he conseguido hacer funcionar el código del tutorial para resolver otros problemas de clasificación… voluntarios?
- Explorar pylearn2…