Convolutional Neural Networks (LeNet)

Note

This section assumes the reader has already read through *Classifying MNIST digits using Logistic Regression* and *Multilayer Perceptron*. Additionally, it uses the following new Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, floatX, downsample , conv2d, dimshuffle. If you intend to run the code on GPU also read GPU.

To run this example on a GPU, you need a good GPU. It needs at least 1GB of GPU RAM. More may be required if your monitor is connected to the GPU.

When the GPU is connected to the monitor, there is a limit of a few seconds for each GPU function call. This is needed as current GPUs can't be used for the monitor while doing computation. Without this limit, the screen would freeze for too long and make it look as if the computer froze. This example hits this limit with medium-quality GPUs. When the GPU isn't connected to a monitor, there is no time limit. You can lower the batch size to fix the time out problem.

Note

The code for this section is available for download here and the 3wolfmoon image

Motivation

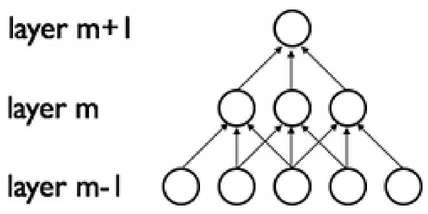
Convolutional Neural Networks (CNN) are biologically-inspired variants of MLPs. From Hubel and Wiesel's early work on the cat's visual cortex [Hubel68], we know the visual cortex contains a complex arrangement of cells. These cells are sensitive to small sub-regions of the visual field, called a *receptive field*. The sub-regions are tiled to cover the entire visual field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images.

Additionally, two basic cell types have been identified: Simple cells respond maximally to specific edge-like patterns within their receptive field. Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

The animal visual cortex being the most powerful visual processing system in existence, it seems natural to emulate its behavior. Hence, many neurally-inspired models can be found in the literature. To name a few: the NeoCognitron [Fukushima], HMAX [Serre07] and LeNet-5 [LeCun98], which will be the focus of this tutorial.

Sparse Connectivity

CNNs exploit spatially-local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. In other words, the inputs of hidden units in layer **m** are from a subset of units in layer **m-1**, units that have spatially contiguous receptive fields. We can illustrate this graphically as follows:

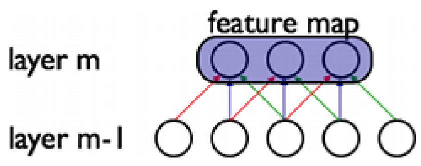


Imagine that layer **m-1** is the input retina. In the above figure, units in layer **m** have receptive fields of width 3 in the input retina and are thus only connected to 3 adjacent neurons in the retina layer. Units in layer **m+1** have a similar connectivity with the layer below. We say that their receptive field with respect to the layer below is also 3, but their receptive field with respect to the input is larger (5). Each unit is unresponsive to variations outside of its receptive field with respect to the retina. The architecture thus ensures that the learnt "filters" produce the strongest response to a spatially local input pattern.

However, as shown above, stacking many such layers leads to (non-linear) "filters" that become increasingly "global" (i.e. responsive to a larger region of pixel space). For example, the unit in hidden layer **m+1** can encode a non-linear feature of width 5 (in terms of pixel space).

Shared Weights

In addition, in CNNs, each filter h_i is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a *feature map*.



In the above figure, we show 3 hidden units belonging to the same feature map. Weights of the same color are shared—constrained to be identical. Gradient descent can still be used to learn such shared parameters, with only a small change to the original algorithm. The gradient of a shared weight is simply the sum of the gradients of the parameters being shared.

Replicating units in this way allows for features to be detected *regardless of their position in the visual field.* Additionally, weight sharing increases learning efficiency by greatly reducing the number of free parameters being learnt. The constraints on the model enable CNNs to achieve better generalization on vision problems.

Details and Notation

A feature map is obtained by repeated application of a function across sub-regions of the entire image, in other words, by *convolution* of the input image with a linear filter, adding a bias term and then applying a non-linear function. If we denote the k-th feature map at a given layer as h^k, whose filters are determined by the weights W^k and bias b_k, then the feature map h^k is obtained as follows (for $tanh$ non-linearities):

$$h^k_{ij} = \tanh((W^k * x)_{ij} + b_k).$$

Note

Recall the following definition of convolution for a 1D signal.
$o[n] = f[n] * g[n] = \sum_{u=-\infty}^{\infty} f[u]g[n-u] = \sum_{u=-\infty}^{\infty} f[n-u]g[u]$.

This can be extended to 2D as follows:
$o[m,n] = f[m,n] * g[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v]g[m-u,n-v]$.

To form a richer representation of the data, each hidden layer is composed of *multiple* feature maps, $\{h^{(k)}, k=0..K\}$. The weights W of a hidden layer can be represented in a 4D tensor containing elements for every combination of destination feature map, source feature map, source vertical position, and source horizontal position. The biases b can be represented as a vector containing one element for every destination feature map. We illustrate this graphically as follows:

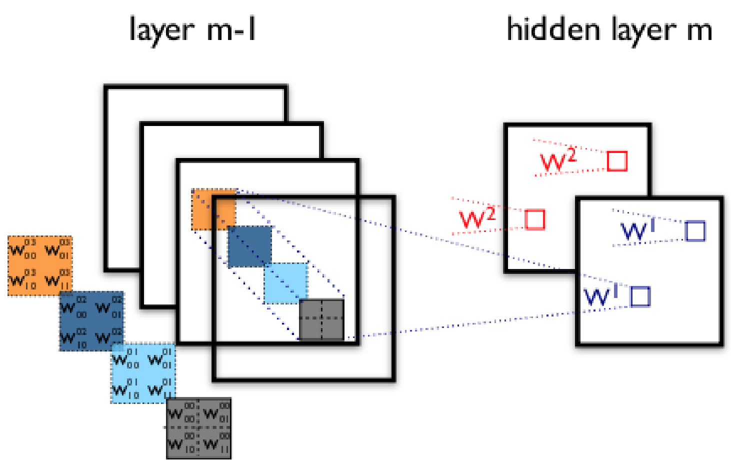

layer m-1
hidden layer m
W^2
W^2
W^1
W^1
W^{03}_{00} W^{03}_{01} W^{03}_{10} W^{03}_{11}
W^{02}_{00} W^{02}_{01} W^{02}_{10} W^{02}_{11}
W^{01}_{00} W^{01}_{01} W^{01}_{10} W^{01}_{11}
W^{00}_{00} W^{00}_{01} W^{00}_{10} W^{00}_{11}

Figure 1: example of a convolutional layer

The figure shows two layers of a CNN. **Layer m-1** contains four feature maps. **Hidden layer m** contains two feature maps (h^0 and h^1). Pixels (neuron outputs) in h^0 and h^1 (outlined as blue and red squares) are computed from pixels of layer (m-1) which fall within their 2x2 receptive field in the layer below (shown as colored rectangles). Notice how the receptive field spans all four input feature maps. The weights W^0 and W^1 of h^0 and h^1 are thus 3D weight tensors. The leading dimension indexes the input feature maps, while the other two refer to the pixel coordinates.

Putting it all together, W^{kl}_{ij} denotes the weight connecting each pixel of the k-th feature map at layer m, with the pixel at coordinates (i,j) of the l-th feature map of layer (m-1).

The Convolution Operator

ConvOp is the main workhorse for implementing a convolutional layer in Theano. ConvOp is used by `theano.tensor.signal.conv2d`, which takes two symbolic inputs:

- a 4D tensor corresponding to a mini-batch of input images. The shape of the tensor is as follows: [mini-batch size, number of input feature maps, image height, image width].
- a 4D tensor corresponding to the weight matrix W. The shape of the tensor is: [number of feature maps at layer m, number of feature maps at layer m-1, filter height, filter width]

Below is the Theano code for implementing a convolutional layer similar to the one of Figure 1. The input consists of 3 features maps (an RGB color image) of size 120x160. We use two convolutional filters with 9x9 receptive fields.

```
import theano
from theano import tensor as T
from theano.tensor.nnet import conv

import numpy

rng = numpy.random.RandomState(23455)

# instantiate 4D tensor for input
input = T.tensor4(name='input')

# initialize shared variable for weights.
w_shp = (2, 3, 9, 9)
w_bound = numpy.sqrt(3 * 9 * 9)
W = theano.shared( numpy.asarray(
            rng.uniform(
                low=-1.0 / w_bound,
                high=1.0 / w_bound,
                size=w_shp),
            dtype=input.dtype), name ='W')

# initialize shared variable for bias (1D tensor) with random values
# IMPORTANT: biases are usually initialized to zero. However in this
# particular application, we simply apply the convolutional layer to
# an image without learning the parameters. We therefore initialize
# them to random values to "simulate" learning.
b_shp = (2,)
b = theano.shared(numpy.asarray(
            rng.uniform(low=-.5, high=.5, size=b_shp),
            dtype=input.dtype), name ='b')

# build symbolic expression that computes the convolution of input with filters in w
conv_out = conv.conv2d(input, W)

# build symbolic expression to add bias and apply activation function, i.e. produce neural n
# A few words on ``dimshuffle`` :
#   ``dimshuffle`` is a powerful tool in reshaping a tensor;
#   what it allows you to do is to shuffle dimension around
#   but also to insert new ones along which the tensor will be
#   broadcastable;
#   dimshuffle('x', 2, 'x', 0, 1)
#   This will work on 3d tensors with no broadcastable
#   dimensions. The first dimension will be broadcastable,
#   then we will have the third dimension of the input tensor as
#   the second of the resulting tensor, etc. If the tensor has
#   shape (20, 30, 40), the resulting tensor will have dimensions
#   (1, 40, 1, 20, 30). (AxBxC tensor is mapped to 1xCx1xAxB tensor)
#   More examples:
#    dimshuffle('x') -> make a 0d (scalar) into a 1d vector
#    dimshuffle(0, 1) -> identity
#    dimshuffle(1, 0) -> inverts the first and second dimensions
#    dimshuffle('x', 0) -> make a row out of a 1d vector (N to 1xN)
#    dimshuffle(0, 'x') -> make a column out of a 1d vector (N to Nx1)
#    dimshuffle(2, 0, 1) -> AxBxC to CxAxB
#    dimshuffle(0, 'x', 1) -> AxB to Ax1xB
#    dimshuffle(1, 'x', 0) -> AxB to Bx1xA
output = T.nnet.sigmoid(conv_out + b.dimshuffle('x', 0, 'x', 'x'))

# create theano function to compute filtered images
f = theano.function([input], output)
```

Let’s have a little bit of fun with this...

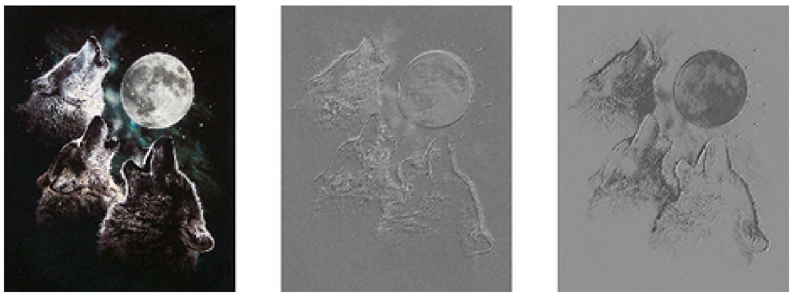
```
import numpy
import pylab
from PIL import Image

# open random image of dimensions 639x516
img = Image.open(open('doc/images/3wolfmoon.jpg'))
# dimensions are (height, width, channel)
img = numpy.asarray(img, dtype='float64') / 256.

# put image in 4D tensor of shape (1, 3, height, width)
img_ = img.transpose(2, 0, 1).reshape(1, 3, 639, 516)
filtered_img = f(img_)

# plot original image and first and second components of output
pylab.subplot(1, 3, 1); pylab.axis('off'); pylab.imshow(img)
pylab.gray();
# recall that the convOp output (filtered image) is actually a "minibatch",
# of size 1 here, so we take index 0 in the first dimension:
pylab.subplot(1, 3, 2); pylab.axis('off'); pylab.imshow(filtered_img[0, 0, :, :])
pylab.subplot(1, 3, 3); pylab.axis('off'); pylab.imshow(filtered_img[0, 1, :, :])
pylab.show()
```

This should generate the following output.



Notice that a randomly initialized filter acts very much like an edge detector!

Note that we use the same weight initialization formula as with the MLP. Weights are sampled randomly from a uniform distribution in the range [-1/fan-in, 1/fan-in], where fan-in is the number of inputs to a hidden unit. For MLPs, this was the number of units in the layer below. For CNNs however, we have to take into account the number of input feature maps and the size of the receptive fields.

MaxPooling

Another important concept of CNNs is *max-pooling,* which is a form of non-linear down-sampling. Max-pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value.

Max-pooling is useful in vision for two reasons:

1. By eliminating non-maximal values, it reduces computation for upper layers.
2. It provides a form of translation invariance. Imagine cascading a max-pooling layer with a convolutional layer. There are 8 directions in which one can translate the input image by a single pixel. If max-pooling is done over a 2x2 region, 3 out of these 8 possible configurations will produce exactly the same output at the convolutional layer. For max-pooling over a 3x3 window, this jumps to 5/8.

   Since it provides additional robustness to position, max-pooling is a “smart” way of reducing the dimensionality of intermediate representations.

Max-pooling is done in Theano by way of `theano.tensor.signal.downsample.max_pool_2d`. This function takes as input an N dimensional tensor (where N >= 2) and a downscaling factor and performs max-pooling over the 2 trailing dimensions of the tensor.

An example is worth a thousand words:

```
from theano.tensor.signal import downsample

input = T.dtensor4('input')
maxpool_shape = (2, 2)
pool_out = downsample.max_pool_2d(input, maxpool_shape, ignore_border=True)
f = theano.function([input],pool_out)

invals = numpy.random.RandomState(1).rand(3, 2, 5, 5)
print 'With ignore_border set to True:'
print 'invals[0, 0, :, :] =\n', invals[0, 0, :, :]
print 'output[0, 0, :, :] =\n', f(invals)[0, 0, :, :]

pool_out = downsample.max_pool_2d(input, maxpool_shape, ignore_border=False)
f = theano.function([input],pool_out)
print 'With ignore_border set to False:'
print 'invals[1, 0, :, :] =\n ', invals[1, 0, :, :]
print 'output[1, 0, :, :] =\n ', f(invals)[1, 0, :, :]
```

This should generate the following output:

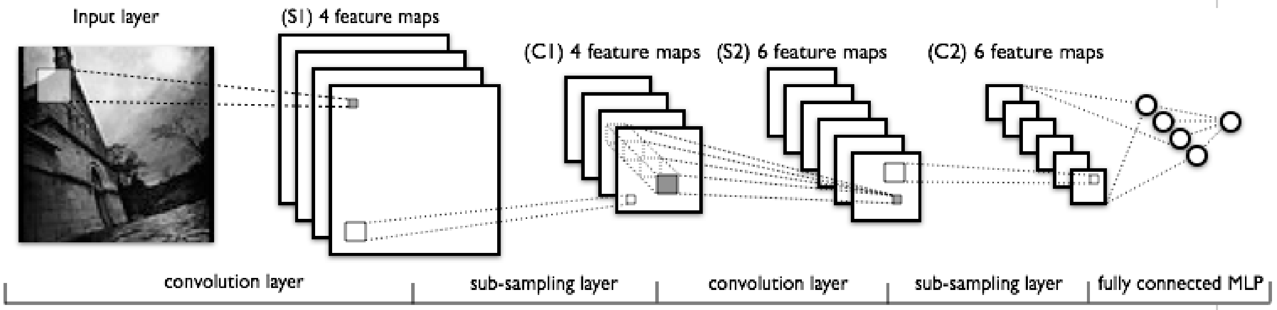
```
With ignore_border set to True:
    invals[0, 0, :, :] =
    [[  4.17022005e-01   7.20324493e-01   1.14374817e-04   3.02332573e-01 1.46755891e-01]
     [  9.23385948e-02   1.86260211e-01   3.45560727e-01   3.96767474e-01 5.38816734e-01]
     [  4.19194514e-01   6.85219500e-01   2.04452250e-01   8.78117436e-01 2.73875932e-02]
     [  6.70467510e-01   4.17304802e-01   5.58689828e-01   1.40386939e-01 1.98101489e-01]
     [  8.00744569e-01   9.68261576e-01   3.13424178e-01   6.92322616e-01 8.76389152e-01]]
    output[0, 0, :, :] =
    [[ 0.72032449  0.39676747]
     [ 0.6852195   0.87811744]]

With ignore_border set to False:
    invals[1, 0, :, :] =
    [[ 0.01936696  0.67883553  0.21162812  0.26554666  0.49157316]
     [ 0.05336255  0.57411761  0.14672857  0.58930554  0.69975836]
     [ 0.10233443  0.41405599  0.69440016  0.41417927  0.04995346]
     [ 0.53589641  0.66379465  0.51488911  0.94459476  0.58655504]
     [ 0.90340192  0.1374747   0.13927635  0.80739129  0.39767684]]
    output[1, 0, :, :] =
    [[ 0.67883553  0.58930554  0.69975836]
     [ 0.66379465  0.94459476  0.58655504]
     [ 0.90340192  0.80739129  0.39767684]]
```

Note that compared to most Theano code, the `max_pool_2d` operation is a little *special*. It requires the downscaling factor `ds` (tuple of length 2 containing downscaling factors for image width and height) to be known at graph build time. This may change in the near future.

The Full Model: LeNet

Sparse, convolutional layers and max-pooling are at the heart of the LeNet family of models. While the exact details of the model will vary greatly, the figure below shows a graphical depiction of a LeNet model.



The lower-layers are composed to alternating convolution and max-pooling layers. The upper-layers however are fully-connected and correspond to a traditional MLP (hidden layer + logistic regression). The input to the first fully-connected layer is the set of all features maps at the layer below.

From an implementation point of view, this means lower-layers operate on 4D tensors. These are then flattened to a 2D matrix of rasterized feature maps, to be compatible with our previous MLP implementation.

Note

Note that the term “convolution” could corresponds to different mathematical operations:

1. theano.tensor.nnet.conv2d, which is the most common one in almost all of the recent published convolutional models. In this operation, each output feature map is connected to each input feature map by a different 2D filter, and its value is the sum of the individual convolution of all inputs through the corresponding filter.
2. The convolution used in the original LeNet model: In this work, each output feature map is only connected to a subset of input feature maps.
3. The convolution used in signal processing: theano.tensor.signal.conv.conv2d, which works only on single channel inputs.

Here, we use the first operation, so this models differ slightly from the original LeNet paper. One reason to use 2. would be to reduce the amount of computation needed, but modern hardware makes it as fast to have the full connection pattern. Another reason would be to slightly reduce the number of free parameters, but we have other regularization techniques at our disposal.

Putting it All Together

We now have all we need to implement a LeNet model in Theano. We start with the LeNetConvPoolLayer class, which implements a {convolution + max-pooling} layer.

```
class LeNetConvPoolLayer(object):
    """Pool Layer of a convolutional network """

    def __init__(self, rng, input, filter_shape, image_shape, poolsize=(2, 2)):
        """
        Allocate a LeNetConvPoolLayer with shared variable internal parameters.

        :type rng: numpy.random.RandomState
        :param rng: a random number generator used to initialize weights

        :type input: theano.tensor.dtensor4
        :param input: symbolic image tensor, of shape image_shape

        :type filter_shape: tuple or list of length 4
        :param filter_shape: (number of filters, num input feature maps,
                              filter height, filter width)

        :type image_shape: tuple or list of length 4
        :param image_shape: (batch size, num input feature maps,
                             image height, image width)

        :type poolsize: tuple or list of length 2
        :param poolsize: the downsampling (pooling) factor (#rows, #cols)
        """

        assert image_shape[1] == filter_shape[1]
        self.input = input

        # there are "num input feature maps * filter height * filter width"
        # inputs to each hidden unit
        fan_in = numpy.prod(filter_shape[1:])
        # each unit in the lower layer receives a gradient from:
        # "num output feature maps * filter height * filter width" /
        #   pooling size
        fan_out = (filter_shape[0] * numpy.prod(filter_shape[2:]) /
                   numpy.prod(poolsize))
        # initialize weights with random weights
        W_bound = numpy.sqrt(6. / (fan_in + fan_out))
        self.W = theano.shared(
            numpy.asarray(
                rng.uniform(low=-W_bound, high=W_bound, size=filter_shape),
                dtype=theano.config.floatX
            ),
            borrow=True
        )

        # the bias is a 1D tensor -- one bias per output feature map
        b_values = numpy.zeros((filter_shape[0],), dtype=theano.config.floatX)
        self.b = theano.shared(value=b_values, borrow=True)

        # convolve input feature maps with filters
        conv_out = conv.conv2d(
            input=input,
            filters=self.W,
            filter_shape=filter_shape,
            image_shape=image_shape
        )

        # downsample each feature map individually, using maxpooling
        pooled_out = downsample.max_pool_2d(
            input=conv_out,
            ds=poolsize,
            ignore_border=True
        )

        # add the bias term. Since the bias is a vector (1D array), we first
        # reshape it to a tensor of shape (1, n_filters, 1, 1). Each bias will
        # thus be broadcasted across mini-batches and feature map
        # width & height
        self.output = T.tanh(pooled_out + self.b.dimshuffle('x', 0, 'x', 'x'))

        # store parameters of this layer
        self.params = [self.W, self.b]

        # keep track of model input
        self.input = input
```

Notice that when initializing the weight values, the fan-in is determined by the size of the receptive fields and the number of input feature maps.

Finally, using the LogisticRegression class defined in *Classifying MNIST digits using Logistic Regression* and the HiddenLayer class defined in *Multilayer Perceptron* , we can instantiate the network as follows.

```
    x = T.matrix('x')   # the data is presented as rasterized images
    y = T.ivector('y')  # the labels are presented as 1D vector of
                        # [int] labels

    ######################
    # BUILD ACTUAL MODEL #
    ######################
    print '... building the model'

    # Reshape matrix of rasterized images of shape (batch_size, 28 * 28)
    # to a 4D tensor, compatible with our LeNetConvPoolLayer
    # (28, 28) is the size of MNIST images.
    layer0_input = x.reshape((batch_size, 1, 28, 28))

    # Construct the first convolutional pooling layer:
    # filtering reduces the image size to (28-5+1 , 28-5+1) = (24, 24)
    # maxpooling reduces this further to (24/2, 24/2) = (12, 12)
    # 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
    layer0 = LeNetConvPoolLayer(
        rng,
        input=layer0_input,
        image_shape=(batch_size, 1, 28, 28),
        filter_shape=(nkerns[0], 1, 5, 5),
        poolsize=(2, 2)
    )

    # Construct the second convolutional pooling layer
    # filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
    # maxpooling reduces this further to (8/2, 8/2) = (4, 4)
    # 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
    layer1 = LeNetConvPoolLayer(
        rng,
        input=layer0.output,
        image_shape=(batch_size, nkerns[0], 12, 12),
        filter_shape=(nkerns[1], nkerns[0], 5, 5),
        poolsize=(2, 2)
    )

    # the HiddenLayer being fully-connected, it operates on 2D matrices of
    # shape (batch_size, num_pixels) (i.e matrix of rasterized images).
    # This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
    # or (500, 50 * 4 * 4) = (500, 800) with the default values.
    layer2_input = layer1.output.flatten(2)

    # construct a fully-connected sigmoidal layer
    layer2 = HiddenLayer(
        rng,
        input=layer2_input,
        n_in=nkerns[1] * 4 * 4,
        n_out=500,
        activation=T.tanh
    )

    # classify the values of the fully-connected sigmoidal layer
    layer3 = LogisticRegression(input=layer2.output, n_in=500, n_out=10)

    # the cost we minimize during training is the NLL of the model
    cost = layer3.negative_log_likelihood(y)

    # create a function to compute the mistakes that are made by the model
    test_model = theano.function(
        [index],
        layer3.errors(y),
        givens={
            x: test_set_x[index * batch_size: (index + 1) * batch_size],
            y: test_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    validate_model = theano.function(
        [index],
        layer3.errors(y),
        givens={
            x: valid_set_x[index * batch_size: (index + 1) * batch_size],
            y: valid_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    # create a list of all model parameters to be fit by gradient descent
    params = layer3.params + layer2.params + layer1.params + layer0.params

    # create a list of gradients for all model parameters
    grads = T.grad(cost, params)

    # train_model is a function that updates the model parameters by
    # SGD Since this model has many parameters, it would be tedious to
    # manually create an update rule for each model parameter. We thus
    # create the updates list by automatically looping over all
    # (params[i], grads[i]) pairs.
    updates = [
        (param_i, param_i - learning_rate * grad_i)
        for param_i, grad_i in zip(params, grads)
    ]

    train_model = theano.function(
        [index],
        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]
        }
    )
```

We leave out the code that performs the actual training and early-stopping, since it is exactly the same as with an MLP. The interested reader can nevertheless access the code in the 'code' folder of DeepLearningTutorials.

Running the Code

The user can then run the code by calling:

```
python code/convolutional_mlp.py
```

The following output was obtained with the default parameters on a Core i7-2600K CPU clocked at 3.40GHz and using flags 'floatX=float32':

```
Optimization complete.
Best validation score of 0.910000 % obtained at iteration 17800,with test
performance 0.920000 %
The code for file convolutional_mlp.py ran for 380.28m
```

Using a GeForce GTX 285, we obtained the following:

```
Optimization complete.
Best validation score of 0.910000 % obtained at iteration 15500,with test
```

```
performance 0.930000 %
The code for file convolutional_mlp.py ran for 46.76m
```

And similarly on a GeForce GTX 480:

```
Optimization complete.
Best validation score of 0.910000 % obtained at iteration 16400,with test
performance 0.930000 %
The code for file convolutional_mlp.py ran for 32.52m
```

Note that the discrepancies in validation and test error (as well as iteration count) are due to different implementations of the rounding mechanism in hardware. They can be safely ignored.

Tips and Tricks

Choosing Hyperparameters

CNNs are especially tricky to train, as they add even more hyper-parameters than a standard MLP. While the usual rules of thumb for learning rates and regularization constants still apply, the following should be kept in mind when optimizing CNNs.

Number of filters

When choosing the number of filters per layer, keep in mind that computing the activations of a single convolutional filter is much more expensive than with traditional MLPs !

Assume layer $(l-1)$ contains K^{l-1} feature maps and $M \times N$ pixel positions (i.e., number of positions times number of feature maps), and there are K^l filters at layer l of shape $m \times n$. Then computing a feature map (applying an $m \times n$ filter at all $(M-m) \times (N-n)$ pixel positions where the filter can be applied) costs $(M-m) \times (N-n) \times m \times n \times K^{l-1}$. The total cost is K^l times that. Things may be more complicated if not all features at one level are connected to all features at the previous one.

For a standard MLP, the cost would only be $K^l \times K^{l-1}$ where there are K^l different neurons at level l. As such, the number of filters used in CNNs is typically much smaller than the number of hidden units in MLPs and depends on the size of the feature maps (itself a function of input image size and filter shapes).

Since feature map size decreases with depth, layers near the input layer will tend to have fewer filters while layers higher up can have much more. In fact, to equalize computation at each layer, the product of the number of features and the number of pixel positions is typically picked to be roughly constant across layers. To preserve the information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) to be non-decreasing from one layer to the next (of course we could hope to get away with less when we are doing supervised learning). The number of feature maps directly controls capacity and so that depends on the number of available examples and the complexity of the task.

Filter Shape

Common filter shapes found in the litterature vary greatly, usually based on the dataset. Best results on MNIST-sized images (28x28) are usually in the 5x5 range on the first layer, while natural image datasets (often with hundreds of pixels in each dimension) tend to use larger first-layer filters of shape 12x12 or 15x15.

The trick is thus to find the right level of “granularity” (i.e. filter shapes) in order to create abstractions at the proper scale, given a particular dataset.

Max Pooling Shape

Typical values are 2x2 or no max-pooling. Very large input images may warrant 4x4 pooling in the lower-layers. Keep in mind however, that this will reduce the dimension of the signal by a factor of 16, and may result in throwing away too much information.

Footnotes

[1] For clarity, we use the word “unit” or “neuron” to refer to the artificial neuron and “cell” to refer to the biological neuron.

Tips

If you want to try this model on a new dataset, here are a few tips that can help you get better results:

- Whitening the data (e.g. with PCA)
- Decay the learning rate in each epoch