17a1d71 on 26 Mar

gforge All files updated to Jupyter nb v 4. Fixed markdown formatting issues...

5 contributors y

1292 lines (1291 sloc) 33.2 KB Raw Blame History

Deep Learning with Torch: the 60-minute blitz

Goal of this talk

- Understand torch and the neural networks package at a high-level.
- Train a small neural network on CPU and GPU

What is Torch?

Torch is a scientific computing framework based on Lua[JIT] with strong CPU and CUDA backends.

Strong points of Torch:

- Efficient Tensor library (like NumPy) with an efficient CUDA backend
- Neural Networks package -- build arbitrary acyclic computation graphs with automatic differentiation
  - also with fast CUDA and CPU backends
- Good community and industry support - several hundred community-built and maintained packages.
- Easy to use Multi-GPU support and parallelizing neural networks

http://torch.ch (http://torch.ch)
https://github.com/torch/torch7/wiki/Cheatsheet (https://github.com/torch/torch7/wiki/Cheatsheet)

Before getting started

- Based on Lua and runs on Lua-JIT (Just-in-time compiler) which is fast
- Lua is pretty close to javascript.
  - variables are global by default, unless `local` keyword is used
- Only has one data structure built-in, a table: `{}`. Doubles as a hash-table and an array.
- 1-based indexing.
- `foo:bar()` is the same as `foo.bar(foo)`

Getting Started

Strings, numbers, tables - a tiny introduction

In [ ]:
```
a = 'hello'
```

In [ ]:
```
print(a)
```

In [ ]:
```
b = {}
```

In [ ]:
```
b[1] = a
```

In [ ]:
```
print(b)
```

In [ ]:
```
b[2] = 30
```

In [ ]:
```
for i=1,#b do -- the # operator is the length operator in Lua
    print(b[i])
end
```

Tensors

In [ ]:
```
a = torch.Tensor(5,3) -- construct a 5x3 matrix, uninitialized
```

In [ ]:
```
a = torch.rand(5,3)
print(a)
```

In [ ]:
```
b=torch.rand(3,4)
```

In [ ]:
```
-- matrix-matrix multiplication: syntax 1
a*b
```

In [ ]:
```
-- matrix-matrix multiplication: syntax 2
torch.mm(a,b)
```

In [ ]:
```
-- matrix-matrix multiplication: syntax 3
c=torch.Tensor(5,4)
c:mm(a,b) -- store the result of a*b in c
```

CUDA Tensors

Tensors can be moved onto GPU using the :cuda function

In [ ]:
```
require 'cutorch';
a = a:cuda()
b = b:cuda()
c = c:cuda()
c:mm(a,b) -- done on GPU
```

Exercise: Add two tensors

https://github.com/torch/torch7/blob/master/doc/maths.md#res-torchaddres-tensor1-tensor2 (https://github.com/torch/torch7/blob/master/doc/maths.md#res-torchaddres-tensor1-tensor2)

In [ ]:
```
function addTensors(a,b)
    return a -- FIX ME
end
```

In [ ]:
```
a = torch.ones(5,2)
b = torch.Tensor(2,5):fill(4)
print(addTensors(a,b))
```

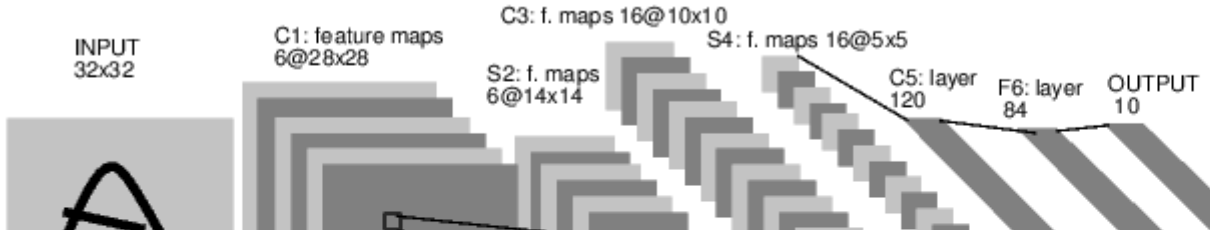
Neural Networks

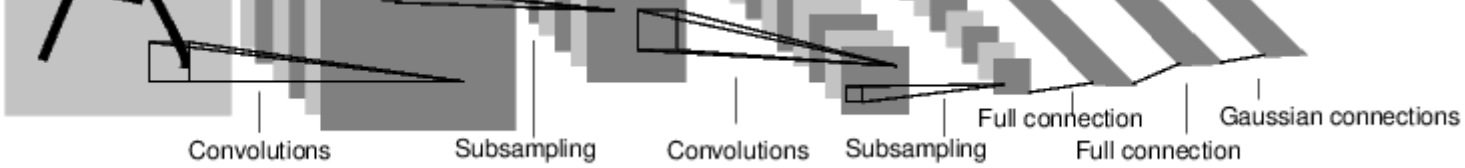
Neural networks in Torch can be constructed using the `nn` package.

In [ ]:
```
require 'nn';
```

`Modules` are the bricks used to build neural networks. Each are themselves neural networks, but can be combined with other networks using `containers` to create complex neural networks

For example, look at this network that classfies digit images:





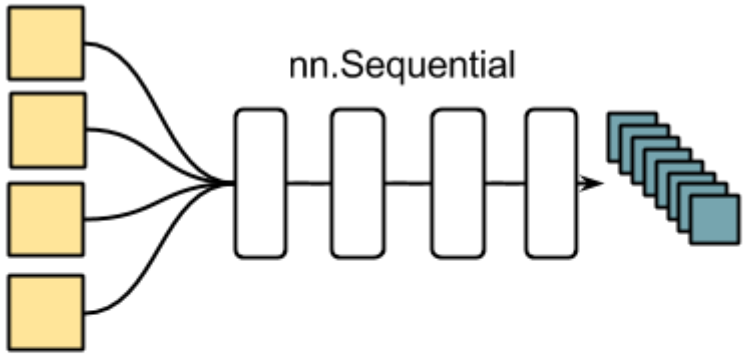
It is a simple feed-forward network.
It takes the input, feeds it through several layers one after the other, and then finally gives the output.

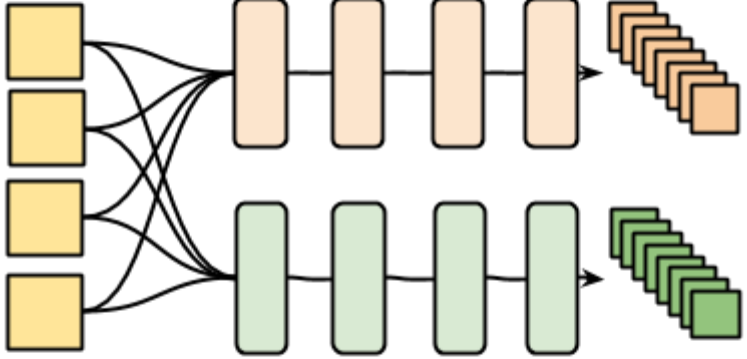
Such a network container is `nn.Sequential` which feeds the input through several layers.

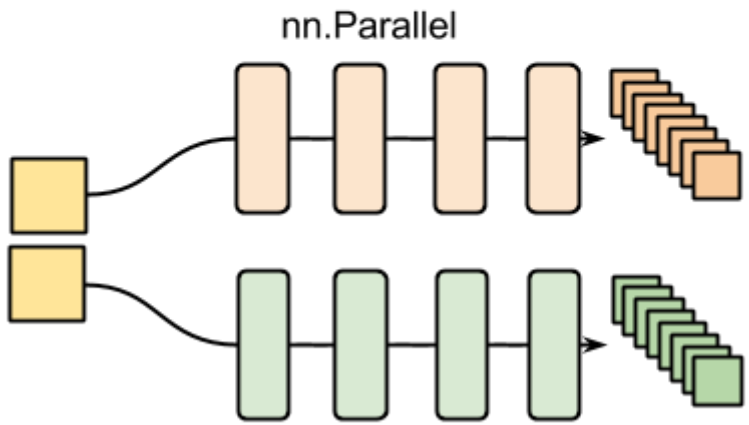
In [ ]:
```
net = nn.Sequential()
net:add(nn.SpatialConvolution(1, 6, 5, 5)) -- 1 input image channel, 6 output channels, 5x5 convolution kernel
net:add(nn.ReLU())                         -- non-linearity
net:add(nn.SpatialMaxPooling(2,2,2,2))     -- A max-pooling operation that looks at 2x2 windows and finds the max.
net:add(nn.SpatialConvolution(6, 16, 5, 5))
net:add(nn.ReLU())                         -- non-linearity
net:add(nn.SpatialMaxPooling(2,2,2,2))
net:add(nn.View(16*5*5))                    -- reshapes from a 3D tensor of 16x5x5 into 1D tensor of 16*5*5
net:add(nn.Linear(16*5*5, 120))             -- fully connected layer (matrix multiplication between input and weights)
net:add(nn.ReLU())                         -- non-linearity
net:add(nn.Linear(120, 84))
net:add(nn.ReLU())                         -- non-linearity
net:add(nn.Linear(84, 10))                   -- 10 is the number of outputs of the network (in this case, 10 digits)
net:add(nn.LogSoftMax())                     -- converts the output to a log-probability. Useful for classification pr
oblems

print('Lenet5\n' .. net:__tostring());
```

Other examples of nn containers are shown in the figure below:







Every neural network module in torch has automatic differentiation. It has a `:forward(input)` function that computes the output for a given input, flowing the input through the network. and it has a `:backward(input, gradient)` function that will differentiate each neuron in the network w.r.t. the gradient that is passed in. This is done via the chain rule.

In [ ]:
```
input = torch.rand(1,32,32) -- pass a random tensor as input to the network
```

In [ ]:
```
output = net:forward(input)
```

In [ ]:
```
print(output)
```

In [ ]:
```
net:zeroGradParameters() -- zero the internal gradient buffers of the network (will come to this later)
```

In [ ]:
```
gradInput = net:backward(input, torch.rand(10))
```

In [ ]:
```
print(#gradInput)
```

Criterion: Defining a loss function

When you want a model to learn to do something, you give it feedback on how well it is doing. This function that computes an objective measure of the model's performance is called a **loss function**.

A typical loss function takes in the model's output and the groundtruth and computes a value that quantifies the model's performance.

The model then corrects itself to have a smaller loss.

In torch, loss functions are implemented just like neural network modules, and have automatic differentiation.
They have two functions - `forward(input, target)`, `backward(input, target)`

For example:

In [ ]:
```
criterion = nn.ClassNLLCriterion() -- a negative log-likelihood criterion for multi-class classification
criterion:forward(output, 3) -- let's say the groundtruth was class number: 3
gradients = criterion:backward(output, 3)
```

In [ ]:
```
gradInput = net:backward(input, gradients)
```

Review of what you learnt so far

- Network can have many layers of computation
- Network takes an input and produces an output in the :forward pass
- Criterion computes the loss of the network, and it's gradients w.r.t. the output of the network.
- Network takes an (input, gradients) pair in it's backward pass and calculates the gradients w.r.t. each layer (and neuron) in the network.

Missing details

> A neural network layer can have learnable parameters or not.

A convolution layer learns it's convolution kernels to adapt to the input data and the problem being solved.
A max-pooling layer has no learnable parameters. It only finds the max of local windows.

A layer in torch which has learnable weights, will typically have fields .weight (and optionally, .bias)

In [ ]:
```
m = nn.SpatialConvolution(1,3,2,2) -- learn 3 2x2 kernels
print(m.weight) -- initially, the weights are randomly initialized
```

In [ ]:
```
print(m.bias) -- The operation in a convolution layer is: output = convolution(input,weight) + bias
```

There are also two other important fields in a learnable layer. The gradWeight and gradBias. The gradWeight accumulates the gradients w.r.t. each weight in the layer, and the gradBias, w.r.t. each bias in the layer.

Training the network

For the network to adjust itself, it typically does this operation (if you do Stochastic Gradient Descent):

```
weight = weight + learningRate * gradWeight [equation 1]
```

This update over time will adjust the network weights such that the output loss is decreasing.

Okay, now it is time to discuss one missing piece. Who visits each layer in your neural network and updates the weight according to Equation 1?

There are multiple answers, but we will use the simplest answer.
We shall use the simple SGD trainer shipped with the neural network module: **nn.StochasticGradient** (https://github.com/torch/nn/blob/master/doc/training.md#stochasticgradientmodule-criterion).

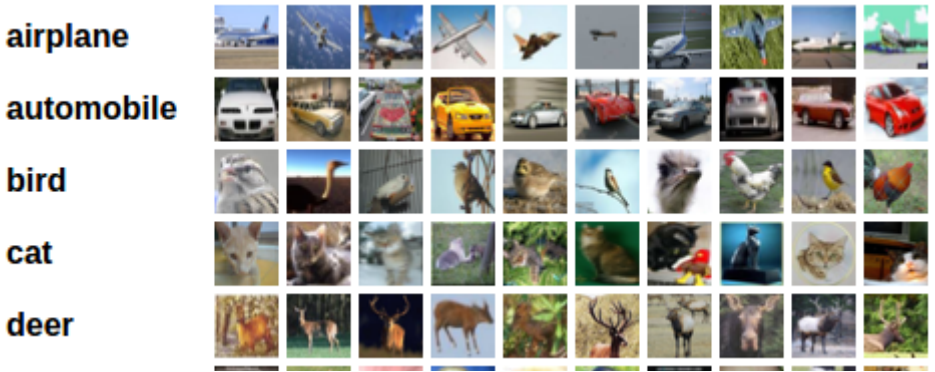
It has a function :train(dataset) that takes a given dataset and simply trains your network by showing different samples from your dataset to the network.

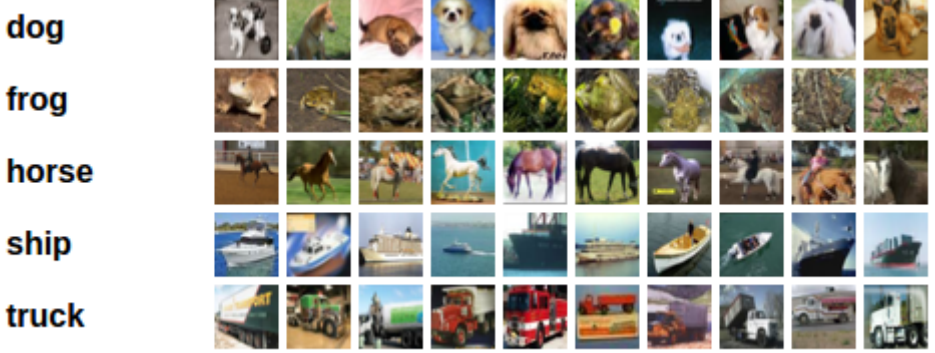
What about data?

Generally, when you have to deal with image, text, audio or video data, you can use standard functions like: **image.load** (https://github.com/torch/image#res-imageloadfilename-depth-tensortype) or **audio.load** (https://github.com/soumith/lua---audio#usage) to load your data into a *torch.Tensor* or a Lua table, as convenient.

Let us now use some simple data to train our network.

We shall use the CIFAR-10 dataset, which has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'.
The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.





The dataset has 50,000 training images and 10,000 test images in total.

We now have 5 steps left to do in training our first torch neural network

1. Load and normalize data
2. Define Neural Network
3. Define Loss function
4. Train network on training data
5. Test network on test data.

1. Load and normalize data

Today, in the interest of time, we prepared the data before-hand into a 4D torch ByteTensor of size 50000x3x32x32 (training) and 10000x3x32x32 (testing) Let us load the data and inspect it.

In [ ]:
```
require 'paths'
if (not paths.filep("cifar10torchsmall.zip")) then
    os.execute('wget -c https://s3.amazonaws.com/torch7/data/cifar10torchsmall.zip')
    os.execute('unzip cifar10torchsmall.zip')
end
trainset = torch.load('cifar10-train.t7')
testset = torch.load('cifar10-test.t7')
classes = {'airplane', 'automobile', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck'}
```

In [ ]:
```
print(trainset)
```

In [ ]:
```
print(#trainset.data)
```

For fun, let us display an image:

In [ ]:
```
itorch.image(trainset.data[100]) -- display the 100-th image in dataset
```

```
itorch.image(trainset.data[100]) -- display the 100-th image in dataset
print(classes[trainset.label[100]])
```

Now, to prepare the dataset to be used with **nn.StochasticGradient**, a couple of things have to be done according to it's documentation (https://github.com/torch/nn/blob/master/doc/training.md#traindataset).

1. The dataset has to have a :size() function.
2. The dataset has to have a [i] index operator, so that dataset[i] returns the ith sample in the datset.

Both can be done quickly:

In [ ]:
```
-- ignore setmetatable for now, it is a feature beyond the scope of this tutorial. It sets the index operator.
setmetatable(trainset,
    {__index = function(t, i)
                    return {t.data[i], t.label[i]}
                end}
);
trainset.data = trainset.data:double() -- convert the data from a ByteTensor to a DoubleTensor.

function trainset:size()
    return self.data:size(1)
end
```

In [ ]:
```
print(trainset:size()) -- just to test
```

In [ ]:
```
print(trainset[33]) -- load sample number 33.
itorch.image(trainset[33][1])
```

One of the most important things you can do in conditioning your data (in general in data-science or machine learning) is to make your data to have a mean of 0.0 and standard-deviation of 1.0.

Let us do that as a final step of our data processing.

To do this, we introduce you to the tensor indexing operator. It is shown by example:

In [ ]:
```
redChannel = trainset.data[{ {}, {1}, {}, {}  }] -- this picks {all images, 1st channel, all vertical pixels, all hori
zontal pixels}
```

In [ ]:
```
print(#redChannel)
```

In this indexing operator, you initally start with ***[{}]***. You can pick all elements in a dimension using ***{}*** or pick a particular element using ***{i}*** where

i is the element index. You can also pick a range of elements using ***{i1, i2}***, for example ***{3,5}*** gives us the 3,4,5 elements.

Exercise: Select the 150th to 300th data elements of the data

In [ ]:
```
-- TODO: fill
```

Moving back to mean-subtraction and standard-deviation based scaling, doing this operation is simple, using the indexing operator that we learnt above:

In [ ]:
```
mean = {} -- store the mean, to normalize the test set in the future
stdv  = {} -- store the standard-deviation for the future
for i=1,3 do -- over each image channel
    mean[i] = trainset.data[{ {}, {i}, {}, {}  }]:mean() -- mean estimation
    print('Channel ' .. i .. ', Mean: ' .. mean[i])
    trainset.data[{ {}, {i}, {}, {}  }]:add(-mean[i]) -- mean subtraction

    stdv[i] = trainset.data[{ {}, {i}, {}, {}  }]:std() -- std estimation
    print('Channel ' .. i .. ', Standard Deviation: ' .. stdv[i])
    trainset.data[{ {}, {i}, {}, {}  }]:div(stdv[i]) -- std scaling
end
```

As you notice, our training data is now normalized and ready to be used.

2. Time to define our neural network

Exercise: Copy the neural network from the **Neural Networks** section above and modify it to take 3-channel images (instead of 1-channel images as it was defined).
Hint: You only have to change the first layer, change the number 1 to be 3.

In [ ]:
```

```

Solution:

In [ ]:
```
net = nn.Sequential()
net:add(nn.SpatialConvolution(3, 6, 5, 5)) -- 3 input image channels, 6 output channels, 5x5 convolution kernel
net:add(nn.ReLU())                       -- non-linearity
net:add(nn.SpatialMaxPooling(2,2,2,2))     -- A max-pooling operation that looks at 2x2 windows and finds the max.
net:add(nn.SpatialConvolution(6, 16, 5, 5))
net:add(nn.ReLU())                       -- non-linearity
net:add(nn.SpatialMaxPooling(2,2,2,2))
net:add(nn.View(16*5*5))                    -- reshapes from a 3D tensor of 16x5x5 into 1D tensor of 16*5*5
```

```
net:add(nn.Linear(16*5*5, 120))              -- fully connected layer (matrix multiplication between input and weights)
net:add(nn.ReLU())                           -- non-linearity
net:add(nn.Linear(120, 84))
net:add(nn.ReLU())                           -- non-linearity
net:add(nn.Linear(84, 10))                   -- 10 is the number of outputs of the network (in this case, 10 digits)
net:add(nn.LogSoftMax())                     -- converts the output to a log-probability. Useful for classification pr
oblems
```

3. Let us define the Loss function

Let us use a Log-likelihood classification loss. It is well suited for most classification problems.

In [ ]:
```
criterion = nn.ClassNLLCriterion()
```

4. Train the neural network

This is when things start to get interesting.
Let us first define an **nn.StochasticGradient** object. Then we will give our dataset to this object's ***:train*** function, and that will get the ball rolling.

In [ ]:
```
trainer = nn.StochasticGradient(net, criterion)
trainer.learningRate = 0.001
trainer.maxIteration = 5 -- just do 5 epochs of training.
```

In [ ]:
```
trainer:train(trainset)
```

5. Test the network, print accuracy

We have trained the network for 2 passes over the training dataset.
But we need to check if the network has learnt anything at all.
We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

Okay, first step. Let us display an image from the test set to get familiar.

In [ ]:
```
print(classes[testset.label[100]])
itorch.image(testset.data[100])
```

Now that we are done with that, let us normalize the test data with the mean and standard-deviation from the training data.

In [ ]:
```
testset.data = testset.data:double()   -- convert from Byte tensor to Double tensor
```

In [ ]:
```
testset.data = testset.data:double()   -- convert from Byte tensor to Double tensor
for i=1,3 do -- over each image channel
    testset.data[{ {}, {i}, {}, {}  }]:add(-mean[i]) -- mean subtraction
    testset.data[{ {}, {i}, {}, {}  }]:div(stdv[i]) -- std scaling
end
```

In [ ]:
```
-- for fun, print the mean and standard-deviation of example-100
horse = testset.data[100]
print(horse:mean(), horse:std())
```

Okay, now let us see what the neural network thinks these examples above are:

In [ ]:
```
print(classes[testset.label[100]])
itorch.image(testset.data[100])
predicted = net:forward(testset.data[100])
```

In [ ]:
```
-- the output of the network is Log-Probabilities. To convert them to probabilities, you have to take e^x
print(predicted:exp())
```

You can see the network predictions. The network assigned a probability to each classes, given the image.

To make it clearer, let us tag each probability with it's class-name:

In [ ]:
```
for i=1,predicted:size(1) do
    print(classes[i], predicted[i])
end
```

Alright, fine. One single example sucked, but how many in total seem to be correct over the test set?

In [ ]:
```
correct = 0
for i=1,10000 do
    local groundtruth = testset.label[i]
    local prediction = net:forward(testset.data[i])
    local confidences, indices = torch.sort(prediction, true)  -- true means sort in descending order
    if groundtruth == indices[1] then
        correct = correct + 1
    end
end
```

In [ ]:
```
print(correct, 100*correct/10000 .. ' %')
```

That looks waaay better than chance, which is 10% accuracy (randomly picking a class out of 10 classes). Seems like the network learnt something.

Hmmm, what are the classes that performed well, and the classes that did not perform well:

In [ ]:
```
class_performance = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
for i=1,10000 do
    local groundtruth = testset.label[i]
    local prediction = net:forward(testset.data[i])
    local confidences, indices = torch.sort(prediction, true)  -- true means sort in descending order
    if groundtruth == indices[1] then
        class_performance[groundtruth] = class_performance[groundtruth] + 1
    end
end
```

In [ ]:
```
for i=1,#classes do
    print(classes[i], 100*class_performance[i]/1000 .. ' %')
end
```

Okay, so what next? How do we run this neural network on GPUs?

cunn: neural networks on GPUs using CUDA

In [ ]:
```
require 'cunn';
```

The idea is pretty simple. Take a neural network, and transfer it over to GPU:

In [ ]:
```
net = net:cuda()
```

Also, transfer the criterion to GPU:

In [ ]:
```
criterion = criterion:cuda()
```

Ok, now the data:

In [ ]:
```
trainset.data = trainset.data:cuda()
trainset.label = trainset.label:cuda()
```

Okay, let's train on GPU :) #sosimple

Okay, let's train on GPU :) #soSimple

In [ ]:
```
trainer = nn.StochasticGradient(net, criterion)
trainer.learningRate = 0.001
trainer.maxIteration = 5 -- just do 5 epochs of training.
```