Training a neural network

Training a neural network is easy with a simple `for` loop. Typically however we would use the `optim` optimizer, which implements some cool functionalities, like Nesterov momentum,
adagrad and
adam.

We will demonstrate using a for-loop first, to show the low-level view of what happens in training. StochasticGradient, a simple class
which does the job for you, is provided as standard. Finally, `optim` is a powerful module, that provides multiple optimization algorithms.

Example of manual training of a neural network

We show an example here on a classical XOR problem.

Neural Network

We create a simple neural network with one hidden layer.

```
require "nn"
mlp = nn.Sequential();  -- make a multi-layer perceptron
inputs = 2; outputs = 1; HUs = 20; -- parameters
mlp:add(nn.Linear(inputs, HUs))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(HUs, outputs))
```

Loss function

We choose the Mean Squared Error criterion:

```
criterion = nn.MSECriterion()
```

Training

We create data *on the fly* and feed it to the neural network.

```
for i = 1,2500 do
  -- random sample
  local input= torch.randn(2);     -- normally distributed example
in 2d
  local output= torch.Tensor(1);
  if input[1]*input[2] > 0 then  -- calculate label for XOR
function
    output[1] = -1
  else
    output[1] = 1
  end

  -- feed it to the neural network and the criterion
  criterion:forward(mlp:forward(input), output)

  -- train over this example in 3 steps
  -- (1) zero the accumulation of the gradients
  mlp:zeroGradParameters()
  -- (2) accumulate gradients
  mlp:backward(input, criterion:backward(mlp.output, output))
  -- (3) update parameters with a 0.01 learning rate
  mlp:updateParameters(0.01)
end
```

Test the network

```
x = torch.Tensor(2)
x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x))
x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x))
x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x))
x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x))
```

You should see something like:

```
> x = torch.Tensor(2)
> x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x))

-0.6140
[torch.Tensor of dimension 1]
```

```
> x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x))

 0.8878
[torch.Tensor of dimension 1]

> x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x))

 0.8548
[torch.Tensor of dimension 1]

> x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x))

-0.5498
[torch.Tensor of dimension 1]
```

StochasticGradient

`StochasticGradient` is a high-level class for training neural networks, using a stochastic gradient
algorithm. This class is serializable.

StochasticGradient(module, criterion)

Create a `StochasticGradient` class, using the given Module and Criterion.
The class contains several parameters you might want to set after initialization.

train(dataset)

Train the module and criterion given in the
constructor over `dataset`, using the
internal parameters.

StochasticGradient expect as a `dataset` an object which implements the operator `dataset[index]` and implements the method `dataset:size()`. The `size()` methods returns the number of examples and `dataset[i]` has to return the i-th example.

An `example` has to be an object which implements the operator `example[field]`, where `field` might take the value `1` (input features) or `2` (corresponding label which will be given to the criterion).
The input is usually a Tensor (except if you use special kind of gradient modules, like table layers). The label type depends of the criterion.
For example, the MSECriterion expects a Tensor, but the ClassNLLCriterion except a integer number (the class).

Such a dataset is easily constructed by using Lua tables, but it could any `C` object for example, as long as required operators/methods are implemented.
See an example.

Parameters

`StochasticGradient` has several field which have an impact on a call to train().

- `learningRate` : This is the learning rate used during training. The update of the parameters will be `parameters = parameters - learningRate * parameters_gradient`. Default value is `0.01`.
- `learningRateDecay` : The learning rate decay. If non-zero, the learning rate (note: the field learningRate will not change value) will be computed after each iteration (pass over the dataset) with: `current_learning_rate =learningRate / (1 + iteration * learningRateDecay)`
- `maxIteration` : The maximum number of iteration (passes over the dataset). Default is `25`.
- `shuffleIndices` : Boolean which says if the examples will be randomly sampled or not. Default is `true`. If `false`, the examples will be taken in the order of the dataset.
- `hookExample` : A possible hook function which will be called (if non-nil) during training after each example forwarded and backwarded through the network. The function takes `(self, example)` as parameters. Default is `nil`.
- `hookIteration` : A possible hook function which will be called (if non-nil) during training after a complete pass over the dataset. The function takes `(self, iteration, currentError)` as parameters. Default is `nil`.

Example of training using StochasticGradient

We show an example here on a classical XOR problem.

Dataset

We first need to create a dataset, following the conventions described in StochasticGradient.

```
dataset={};
function dataset:size() return 100 end -- 100 examples
for i=1,dataset:size() do
  local input = torch.randn(2);     -- normally distributed example
in 2d
  local output = torch.Tensor(1);
  if input[1]*input[2]>0 then     -- calculate label for XOR
function
    output[1] = -1;
  else
    output[1] = 1
  end
  dataset[i] = {input, output}
end
```

Neural Network

We create a simple neural network with one hidden layer.

```
require "nn"
mlp = nn.Sequential();  -- make a multi-layer perceptron
inputs = 2; outputs = 1; HUs = 20; -- parameters
mlp:add(nn.Linear(inputs, HUs))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(HUs, outputs))
```

Training

We choose the Mean Squared Error criterion and train the dataset.

```
criterion = nn.MSECriterion()
trainer = nn.StochasticGradient(mlp, criterion)
trainer.learningRate = 0.01
trainer:train(dataset)
```

Test the network

```
x = torch.Tensor(2)
x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x))
x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x))
x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x))
x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x))
```

You should see something like:

```
> x = torch.Tensor(2)
> x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x))

-0.3490
[torch.Tensor of dimension 1]

> x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x))

 1.0561
[torch.Tensor of dimension 1]

> x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x))

 0.8640
[torch.Tensor of dimension 1]

> x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x))

-0.2941
[torch.Tensor of dimension 1]
```

Using optim to train a network

`optim` is a powerful module, that provides multiple optimization algorithms.