# rnn : Recurrent Library for Torch7

**Nicholas Léonard**
Element Inc.
New York, NY
`nick@nikopia.org`

**Sagar Wagmar**
Element Inc.
New York, NY
`sw@discoverelement.com`

**Yang Wang**
Element Inc.
New York, NY
`yw@discoverelement.com`

## Abstract

The **rnn** package provides components for implementing a wide range of Recurrent Neural Networks. It is built withing the framework of the Torch distribution for use with the **nn** package. The components have evolved from 3 iterations, each adding to the flexibility and capability of the package. All component modules inherit either the `AbstractRecurrent` or `AbstractSequencer` classes. Strong unit testing, continued backwards compatibility and access to supporting material are the principles followed during its development. The package is compared against existing implementations of two published papers.

## 1 Introduction

In recent years, deep learning research has seen a resurgence of interest in Recurrent Neural Networks (RNN). In the scope of our own research, we have developed a package that makes it easy to implement a wide range of RNNs using the Torch distribution. The **rnn** package[1] can be used to implement recurrent neural networks like simple RNNs and Long Short Term Memory (LSTM) networks. The package is very general and makes heavy use of object-oriented programming to keep it as simple to use and extend as possible. The sections are divided into an overview of the **Torch 7** distribution, package components organized historically and principles during its development.

## 2 Torch

Torch[2] is a scientific computing distribution with a focus on deep learning research and applications [2]. The main interface is accessible through the Lua programming language [7], which uses functions and structures implemented using the C and CUDA programming languages. Lua is simple enough to make it easy to implement code for fast execution in C/CUDA. Torch 7 has fast and efficient support for Graphical Processing Unit (GPU) via the **cutorch** and **cunn** packages. The distribution is used by Facebook, Google DeepMind, Twitter, New York University and many other organizations. Through GitHub[3], one can access documentation, tutorials and a wide variety of examples. The project is quite mature as it has been under active development since October 2012. The distribution is divided into different packages which we will overview in the next sections.

### 2.1 torch7

This package is the core of the distribution[4]. It provides a `Tensor` class for manipulating multi-dimensional arrays. This is the main class of objects used in Torch 7. The `Tensor` supports

[1]https://github.com/Element-Research/rnn
[2]http://torch.ch/
[3]https://github.com/
[4]https://github.com/torch/torch7

common operations like Basic Linear Algebra Sub-routines (BLAS), random initialization, indexing, slicing, transposition, etc. Most operations for`FloatTensor` and `DoubleTensor` are also implemented for `CudaTensors` (via the **cutorch**).

While Lua can be used to implement class hierarchies, or more generally, object-oriented programming (OOP), the torch package provides utilities such as `torch.class` for implementing inheritance and `torch.serialize` for serialization. The **torch** package also provides utilities for saving objects to disk, unit testing, etc.

### 2.2 nn

This package implements feed-forward neural networks[5]. These form a computational flow-graph of transformation. They typically learn through backpropagation, which is gradient descent using the chain rule [14].

The **nn** package is very simple as all classes inherit one of either two abstract classes :

- Module : differentiable transformations of `input` to `output` ;
- Criterion : cost function to minimize. Outputs a scalar loss;

The **nn** is used by first building a graph of modules using composite (`Container` subclasses) and component modules, and then training a the resulting neural network on some data.

As an example, a Multi-Layer Perceptron (MLP) with 2 layers of hidden units can be assembled as such:

```
mlp = nn.Sequential()
mlp:add(nn.Convert('bchw', 'bf')) -- collapse 3D to 1D
mlp:add(nn.Linear(1*28*28, 200))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(200, 200))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(200, 10))
mlp:add(nn.LogSoftMax()) -- for classification problems
```

In the above example, the `Sequential` is a `Container` subclass. A call to `output = mlp:forward(input)` will iteratively transform the `input` one module at a time, in order that these were added to the composite.

To fit the `mlp` module to a dataset, the Negative Log-Likelihood (NLL) criterion could be used:

```
nll = nn.ClassNLLCriterion()
```

The actual training loop would look somewhat like this :

```
function trainEpoch(module, criterion, inputs, targets)
   for i=1,inputs:size(1) do
      local idx = math.random(1,inputs:size(1))
      local input, target = inputs[idx], targets:narrow(1,idx,1)
      -- forward
      local output = module:forward(input)
      local loss = criterion:forward(output, target)
      -- backward
      local gradOutput = criterion:backward(output, target)
      module:zeroGradParameters()
      local gradInput = module:backward(input, gradOutput)
      -- update
      module:updateParameters(0.1) -- W = W - 0.1*dL/dW
   end
end
```

[5]https://github.com/torch/nn

The above `trainEpoch` function could be used to train the `mlp` module using the `nll` criterion to fit a classification dataset defined by the `inputs` and `targets` tensors.

The **rnn** package was designed to be used in the scope of the **nn** package. This means that its components must conform to the `Module` and `Criterion` interfaces such that these can be used in for training with functions like `trainEpoch`.

## 3 Package Components

This section is a kind of analysis of the package, exploring its historical development and the components that evolved from it. While it would be nice to come up with the finished product in the first iteration, often times we only get to such a state as time progresses. And in our necessity to maintain a certain level of backwards compatibility, the final product can only really be understood through its historical development. As such, we have divided the analysis of its components into the 3 major iterations in which they appeared.

Before this package, the only way to implement RNNs for variable length sequences was to manually clone the recurrent modules for each time-step, have these share parameters and write code to apply these clones over a sequence. This was against the base philosophy of the **nn** package where every transformation of `input` to `output` should implemented as a `Module` (or composite thereof). So the **rnn** package started out as a single `Recurrent` module that internally implemented a general interface for implementing variations of Simple RNNs as described in [15, section 2.5-2.8], [10, section 3.2-3.3] and [1]. More generally, a recurrent module is responsible for managing the cloning, parameter sharing of the and sequentially applying these internal modules to elements of a sequence.

### 3.1 First Iteration : Recurrent module

As a first iteration, we wanted to be able to forward a sequence through a `Recurrent` module by making successive calls to its `forward` method[6] :

```
-- backpropagate every 5 time steps
rho = 5
batchSize = 3
inputSize = 10
outputSize = 10

-- generate some dummy inputs and gradOutputs sequences
inputs, gradOutputs = {}, {}
for step=1,rho do
   inputs[step] = torch.randn(batchSize,inputSize)
   gradOutputs[step] = torch.randn(batchSize,inputSize)
end

-- an AbstractRecurrent instance
rnn = nn.Recurrent(
   hiddenSize, -- size of the input layer
   nn.Linear(inputSize,outputSize), -- input layer
   nn.Linear(outputSize, outputSize), -- recurrent layer
   nn.Sigmoid(), -- transfer function
   rho -- maximum number of time-steps for BPTT
)

-- feed-forward and backpropagate through time like this :
for step=1,rho do
   rnn:forward(inputs[step])
   rnn:backward(inputs[step], gradOutputs[step])
end
```

[6] For reasons of backwards compatibility this use case is still supported

```
rnn:backwardThroughTime() -- call backward on the internal modules
gradInputs = rnn.gradInputs
rnn:updateParameters(0.1)
rnn:forget() -- resets the time-step counter
```

As can be seen by the above example, the original design allowed for the call to `forward` of each element in the sequence to be immediately followed by a commensurate call to `backward`. Since backpropagation through time (BPTT)[14] can only occur after the entire sequence had been forwarded through the RNN, the above calls to `backward` cannot perform BPTT. Instead they only keep a copy of the provided `gradOutput` for each time-step. The actual BPTT required an explicit call to the `backwardThroughTime` of all `AbstractRecurrent` instances.

This design also prevented calls to `backward` from returning a valid `gradInput`, as these are only made available after BPTT. This is also what necessitated the second argument of the `Recurrent` constructor, which offers a means for handling previous layers internally.

## 3.2 Second Iteration : Sequencer and LSTM

When the `LSTM` module was being implemented during out second iteration, it quickly became apparent that constraints resulting from our design of the `AbstractRecurrent` were too limiting. For one, the first iteration made it impossible to stack `AbstractRecurrent` instances. However, as is often the case with the **nn** package, the problem could be resolved by abstracting these intricacies away into another module. Hence the `Sequencer` was born.

### 3.2.1 Sequencer

The `Sequencer` module is a decorator used to abstract away the intricacies of `AbstractRecurrent` modules like `Recurrence`, `Recurrent` and `LSTM`.

```
seq = nn.Sequencer(module)
```

While an `AbstractRecurrent` instance requires a sequence to be presented one element at a time, each with its own call to `forward` (and `backward`), the `Sequencer` forwards an entire `input` sequence (a table) to yield the resulting `output` sequence (a table of the same length). It also takes care of calling `forget`, `backwardOnline` and other such `AbstractRecurrent`-specific methods.

For example, `rnn`, an `AbstractRecurrent` instance, can forward an input sequence one `forward` call at a time:

```
input = {torch.randn(3,4), torch.randn(3,4), torch.randn(3,4)}
rnn:forward(input[1])
rnn:forward(input[2])
rnn:forward(input[3])
```

Equivalently, we can use a `Sequencer` to forward the entire `input` sequence at once:

```
seq = nn.Sequencer(rnn)
seq:forward(input)
```

Furthermore, the `Sequencer` manages the `backward` and `backwardThroughTime` calls to the decorated module internally. This means that a call to `Sequencer:backward` will return the appropriate `gradInput` table.

The `Sequencer` can also take a *non-recurrent module* [7] and apply it to each element of the `input` sequence to produce an `output` table of the same length. However, in this second iteration of the package, each `Sequencer` instance could only either decorate a recurrent instance [8], or a

[7]By non-recurrent module, we mean a module that isn't an instance of `AbstractRecurrent`, and that neither contains such instances.

[8]Any `AbstractRecurrent` instance is a recurrent instance.

non-recurrent instance. This was not an imposing constraint as it can be subverted by stacking `Sequencer` instances:

```
srnn = nn.Sequential()
   :add(nn.Sequencer(nn.Linear(inputSize, hiddenSize)))
   :add(nn.Sequencer(nn.LSTM(hiddenSize, hiddenSize)))
   :add(nn.Sequencer(nn.LSTM(hiddenSize, hiddenSize)))
   :add(nn.Sequencer(nn.Linear(hiddenSize, outputSize)))
   :add(nn.Sequencer(nn.LogSoftMax()))
```

The above was actually the use-case that brought us to this second iteration of the code base. The objective was to build the stacked networks of LSTM layers outlined in [17].

#### 3.2.2 LSTM

The `LSTM` module is an implementation of a layer of Long-Short Term Memory units[6]. We used the LSTM in [3] as a blueprint for this module as it was the most concise. Yet it is also the vanilla LSTM described in [4].

```
module = nn.LSTM(inputSize, outputSize, [rho])
```

The implementation of the `forward` method corresponds to the following algorithm:

**Algorithm 1** Long Short Term Memory feed forward

```
1: $i_t = \sigma(W_{x\to i}x_t + W_{h\to i}h_{t-1} + W_{c\to i}c_{t-1} + b_{1\to i})$
2: $f_t = \sigma(W_{x\to f}x_t + W_{h\to f}h_{t-1} + W_{c\to f}c_{t-1} + b_{1\to f})$
3: $z_t = \tanh(W_{x\to c}x_t + W_{h\to c}h_{t1} + b_{1\to c})$
4: $c_t = f_t c_{t-1} + i_t z_t$
5: $o_t = \sigma(W_{x\to o}x_t + W_{h\to o}h_{t1} + W_{c\to o}c_t + b_{1\to o})$
6: $h_t = o_t \tanh(c_t)$
```

where $W_{s\to q}$ is the weight matrix from $s$ to $q$, $t$ indexes the time-step, $b_{1\to q}$ are the biases leading into $q$, $\sigma()$ is the logistic function, $x_t$ is the input, $i_t$ is the input gate (line 1), $f_t$ is the forget gate (line 2), $z_t$ is the input to the cell (which we call the hidden) (line 3), $c_t$ is the cell (line 4), $o_t$ is the output gate (line 5), and $h_t$ is the output of this module (line 6). Also note that the weight matrices from cell to gate vectors are diagonal $W_{c\to s}$, where $s$ is gate $i$, $f$, or $o$.

The `LSTM` module is implemented internally as a composite of existing modules. As in the case of the `Recurrent` class, a different clone sharing parameters with the internal module is applied to each time-step. Each clone manages its own copy of intermediate representations, which consists mostly of `output` and `gradInput` attributes.

#### 3.2.3 Repeater

```
r = nn.Repeater(module, nStep)
```

While the `Sequencer` applied a decorated module to an `input` sequence (a table), the `Repeater` repeatedly applies a module to a single unchanging `input`. Both decorators produce an `output` sequence (a table). The `Repeater` was designed to implement things that are recursively applied to the same input, like Recurrent Convolutional Neural Networks (RCNN)[13].

The second iteration arose out of the necessity to allow for stacking of recurrent instances, specifically `LSTM` modules.

### 3.3 Third Iteration

The current iteration arose from the reproduction of the Recurrent Attention Model (RAM) described in [11]. The only lacking component to reproduce the RAM was the `RecurrentAttention` module.

### 3.3.1 RecurrentAttention

This module is similar to the `Repeater` module in that it recursively applies an `rnn` module to a fixed `input`, which in this case is an image.

```
ram = nn.RecurrentAttention(rnn, action, nStep, hiddenSize)
```

The `rnn` argument is an AbstractRecurrent instance which expects a table {`x, z`} as `input` where `x` is the `ram input` and `z` is an action sampled from the `action` module.

The `action` is a Module that learns using the REINFORCE learning rule [16]. It samples actions given the previous time-step's `rnn output`. The `action` module's outputs are only used internally to guide the attention of the `RecurrentAttention` module.

The implementation of `RecurrentAttention` module was a kind of validation of the separation of functionality between the `AbstractRecurrent` and `AbstractSequencer` classes. The first defines general components that handles the recursion from `forward` to the next, i.e. one element at a time. It is an abstract class inherited by `LSTM` and `Recurrent`. The second defines how the recurrent component is used for specific tasks involving sequences, i.e. one sequence of elements at a time. It is an abstract class inherited by `Sequencer`, `Repeater` and `RecurrentAttention` This division of labor happens to be modular enough to allow for implementing most tasks without requiring the writing of new code for both types of modules. What we mean by this is that research topics will generally explore modifications of either abstract classes, but not both at the same time.

Nevertheless, the **rnn** library was still lacking the flexibility to allow for more complex configurations of non-recurrent instances with recurrent instances. The solution to this problem arose from the observation that `RecurrentAttention` expected the `action` constructor argument to be a non-recurrent instance. However, to make the `RecurrentAttention` module generalize to the later DRAM implementation in (citation), it would need to allow composites of both recurrent and non-recurrent instances for the `action` argument. Again, the easiest way to make this happen, was to implement a new module, in this case the `Recursor`.

### 3.3.2 Recursor

This module decorates another `module` to allow it to be used within an `AbstractSequencer` instance. It does this by making the decorated `module` conform to the `AbstractRecurrent` interface, which like the `LSTM` and `Recurrent` classes, this class inherits.

```
rec = nn.Recursor(module[, rho])
```

For each successive call to `updateOutput` (i.e. `forward`), this decorator will call `stepClone` on the decorated `module`. So for each time-step, it will forward the commensurate input through a commensurate clone of the `module`. As usual, both the clone and original share parameters and gradients w.r.t. parameters. [9]

So in the second iteration, to stack `LSTMs`, two `Sequencers` were required :

```
lstm = nn.Sequential()
   :add(nn.Sequencer(nn.LSTM(100,100)))
   :add(nn.Sequencer(nn.LSTM(100,100)))
```

Using a Recursor, the same model can be assembled with a single `Sequencer` :

```
lstm = nn.Sequencer(
   nn.Recursor(
      nn.Sequential()
         :add(nn.LSTM(100,100))
         :add(nn.LSTM(100,100))
      )
   )
```

[9]For recurrent modules, the clones and original module are one and the same (i.e. no cloning occurs)

Actually, the `Sequencer` will wrap any non-recurrent module automatically. So the above model can be further simplified :

```
lstm = nn.Sequencer(
   nn.Sequential()
      :add(nn.LSTM(100,100))
      :add(nn.LSTM(100,100))
   )
```

A non-recurrent instance like `Linear` can also be added between both `LSTMs`. In this case, a `Linear` will be cloned (and have its parameters shared) for each time-step, while the `LSTMs` will handle cloning internally :

```
lstm = nn.Sequencer(
   nn.Sequential()
      :add(nn.LSTM(100,100))
      :add(nn.Linear(100,100))
      :add(nn.LSTM(100,100))
   )
```

To recapitulate, recurrent instances are expected to manage time-steps internally. Non-recurrent instances can be wrapped by a `Recursor` to yield the same behavior.

So the final version of the `AbstractSequencer` subclasses automatically decorate all non-recurrent instances with a `Recursor`. This allows the `RecurrentAttention` module to accept any type of `action` module, thus providing the required flexibility to use it to implement the DRAM model without any modifications to existing modules.

### 3.3.3 Recurrence

The last module introduced in this third iteration is the `Recurrence` module. Another `AbstractRecurrent` subclass, this module is an extremely general container for implementing recurrences that feedback the previous `output` alongside the current input to the `Recurrence`.

```
rnn = nn.Recurrence(module, outputSize, nInputDim, [rho])
```

Unlike the older `Recurrent` module, `Recurrence` only requires a single `module` which implements the actual recurrence internally. This `module` should forward an `output` a tensor (or table) for the current time-step (`output(t)`) given an `input` table : {`input(t), output(t-1)`}. Using a mix of `Recursor` (say, via `Sequencer`) and `Recurrence`, it is possible to implement any a very general set of recurrent neural networks, including LSTMs and Simple RNNs.

For the first step, the `Recurrence` forwards a Tensor (or table thereof) of zeros through the recurrent layer (like `LSTM`, unlike `Recurrent`).

As an example, let us combine `Sequencer` and `Recurrence` to build a Simple RNN for language modeling :

```
rho = 5
hiddenSize = 10
nIndex = 10000

-- recurrent module
rm = nn.Sequential()
   :add(nn.ParallelTable()
      :add(nn.LookupTable(nIndex, hiddenSize))
      :add(nn.Linear(hiddenSize, hiddenSize)))
   :add(nn.CAddTable())
   :add(nn.Sigmoid())

rnn = nn.Sequencer(
   nn.Sequential()
      :add(nn.Recurrence(rm, hiddenSize, 1))
      :add(nn.Linear(hiddenSize, nIndex))
      :add(nn.LogSoftMax())
)
```

Both the `input` and `output` of the `rnn` module will be a table of tensors. For example :

```
batchSize = 32
input = {}
for i=1,rho do
   table.insert(input, torch.Tensor(batchSize):random(1,nIndex))
end
output = rnn:forward(input)
assert(#output == #input)
```

RNNs require sequential data. In the above example, the `input` is a sequence of `LookupTable` indices. If the task is to predict the next word given the previous word(s) (i.e. language modeling), then the `target` would also be a sequence of indices.

If however we only wanted to use the `rho` previous time-steps (words) to predict a single target word, we could do so by having the output layer depend only on the most recent `output(t)` of the `rnn`.

For example if we want to do sentiment analysis [12], we could use something like the following :

```
rho = 5
hiddenSize = 100
nIndex = 10000
nSentiment = 10

-- recurrent module
rm = nn.Sequential()
   :add(nn.ParallelTable()
      :add(nn.LookupTable(nIndex, hiddenSize))
      :add(nn.Linear(hiddenSize, hiddenSize)))
   :add(nn.CAddTable())
   :add(nn.Sigmoid())

rnn = nn.Sequential()
   :add(nn.Sequencer(nn.Recurrence(rm, hiddenSize, 1)))
   :add(nn.SelectTable(-1)) --select last element
   :add(nn.Linear(hiddenSize, nSentiment))
   :add(nn.LogSoftMax())
)
```

## 4 Development Principles

The previous section discussed the main components used in the **rnn** package, and how they evolved from the need for additional functionality or new use cases. In all these cases, we didn't go into too much details regarding the internal workings of each model. For example, we did not discuss the ability of `Sequencers` to remember previously presented sequences, the ability of recurrent instances to evaluate very long sequences without requiring any additional memory, the ability of all modules to deal with nested tables of tensors, the ability to handle variable length inputs, or how RNNs can BPTT for less time-steps than the number of forwarded time-steps.

### 4.1 Unit Testing

In any cases, each of these features potentially introduce bugs. The only way to make sure that these are weeded out and not introduced in later revisions is by emphasizing the requirement for broad unit tests. The **rnn** has unit tests for each of its component modules. It also includes unit tests for different combinations of modules. These unit tests are almost always designed the same way. Functionality of modules introduced by the package is compared to a baseline which is known to work. For example, when implementing the `Recurrent` module unit tests, it was compared to an equivalent composite structure built using modules taken directly from the **nn** package (which are already unit tested).

However, unit tests can only go so far. The ultimate test is to reproduce the results of existing papers. For the `LSTM` module, we were initially unable to reproduce the LSTM paper (citation). The implementation of the paper was available on GitHub [10], and used a combination of the `nngraph` package and custom code to implement an stack of LSTMs. So we ended up extracting the code from the original repository that we wanted to reproduce. We included it in a unit test that tried to have our `LSTM` module match the behavior of their own implementation. It was only in doing so that we were able to resolve hidden discrepancies (bugs). Noteworthy among them was the fact that their LSTM implementation used the last hidden states of the previous sequence to influence the current sequence. This was not obvious for us as doing so for a Simple RNN introduced instability which often led to divergence during training. In any case, this massive unit test now ensures that our `LSTM` implementation matches a published open-source state of the art implementation.

### 4.2 Backward Compatibility

Since November 2014, the **rnn** package has been available for use on GitHub as an BSD-licensed open-source repository. As can be seen by the above overview of its major iterations, the design has evolved over time. From the start, we have tried to maintain its backward compatibility so that users can continue to benefit from updates without requiring major changes to existing code or serialized objects that depends on the **rnn** package.

However, maintaining backwards compatibility has its drawbacks. For example, the `Recurrent` module is convoluted compared to the newer `Recurrence` module. Users will continue to use the former even though the latter is more general and easier to use.

As for the `LSTM` code, it is basically made redundant by the new `Recurrence` module. A compromise worthy of consideration is to make the `LSTM` module a `Recurrence` subclass. But this would break backwards compatibility for users loading an serialized instance of the older `LSTM` in the scope of a version of the **rnn** including the newer `LSTM` instance. This issue is caused by the way Torch handles serialization and deserialization of objects. The class definition (i.e. the Lua *metatable*) is not serialized but is required prior to serialization. Therefore to implement this compromise, we would still be breaking backwards compatibility for serialized modules. But such a change, if implemented correctly (by preserving the same interface), would not break existing scripts making use of the `LSTM` module. We opted to preserve the `LSTM` code in its current state.

The constraint for backwards compatibility is an important one as it minimizes the hassle for users. But at the same time, it does result in redundant code (multiple ways of doing the same thing) and support for deprecated use cases.

### 4.3 Supporting Material

From its inception, our focus has been on providing supporting material. We can divide these into the following categories :

- Documentation;
- Examples; and
- Tutorials.

Documentation is provided for all modules and criterions provided in the package. It also includes references to related scientific articles, examples and tutorials. Documentation is used as a kind of reference manual for specific components. Examples are concrete demonstrations of the capabilities of the package with respect to implementing a particular use case. These also demonstrate how

[10]https://github.com/wojzaremba/lstm

the package can be used with other packages, or more generally, within the scope of the Torch distribution. The package references example scripts for training language models and a recurrent attention models on different datasets. Tutorials include videos, articles or blog posts explaining how to use the package, often with respect to a concrete example.

All the supporting material is important as it brings the package to life, allowing the user to learn how to use it. It also has the side-effect of making it seem more legitimate, thereby encouraging new users to dive in.

### 4.4 Core Extensions

Submitting a GitHub Pull Request (PR) to get some specific code merged into the core packages can be daunting. Delays can range from days to weeks. After which the PR is sometimes refused. Lua has a certain advantage here over other programming languages as its heavy reliance on tables makes it very easy to overwrite or extend core package functionality from within an non-core package.

For example, the core implementation of the `Module:type()` would decouple share parameters. But because the `Module` class definition is just another table, it was easy to overwrite the method to preserve sharing semantics when type-casting. Many more such core extensions were necessary to make the **rnn** package.

## 5 Results

The package was used to reproduce two papers : Recurrent Neural Network Regularization [17] and Recurrent Models for Visual Attention [11].

The first paper implements a stack of LSTM layers [5] and benchmarks various sizes of the model on different datasets. The results presented in the paper are better than those that can be attained using their commensurate GitHub repository [11]. The provided code allows one to train a stack of LSTM layers, with and without dropout, on the Penn Tree Bank dataset [9]. Using their script, test set perplexity with and without dropout is 82 and 115, respectively. Using the **rnn** package, our script was able to reach commensurate perplexities of 83 and 115[12].

The second paper implements recurrent attention model (RAM) that learns using a combination of backpropagation and REINFORCE[16] learning. The authors do not provide code, but their paper includes a detailed description of the model. The RAM was implemented using the `Recurrent` and `RecurrentAttention` modules of the **rnn** package [13] As specified in the original paper, the RAM is trained on the MNIST [8] and Translated MNIST datasets. While they respectively reach 1.07% and 1.22% error on both datasets, our implementation was able to surpass these results by reaching 0.85% and 1.14% error.

## 6 Conclusion

In this paper, we discussed the evolution of the **rnn** package, its different component modules, the various principles underlying its development, and its performance compared to empirical results of published RNN models.

Unlike other RNN implementations using Torch, the **rnn** package doesn't depend on the **nngraph** library. Like the **nn** package, this one is designed with the assumption that all transformations and loss functions can be refactored into either a `Module` or a `Criterion`, respectively. It can also be used with the official **optim** or the unofficial **dp** numeric optimization packages.

## References

[1] M. Boden. A guide to recurrent neural networks and backpropagation. 2001.

[11]https://github.com/wojzaremba/lstm

[12]https://github.com/nicholas-leonard/dp/blob/master/examples/recurrentlanguagemodel.lua

[13]https://github.com/Element-Research/rnn/blob/master/examples/recurrent-visual-attention.lua

[2] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[3] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[4] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[5] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[6] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[8] Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits, 1998.

[9] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

[10] T. Mikolov. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*, 2012.

[11] V. Mnih, N. Heess, A. Graves, et al. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems*, pages 2204–2212, 2014.

[12] B. Pang and L. Lee. Opinion mining and sentiment analysis. *Foundations and trends in information retrieval*, 2(1-2):1–135, 2008.

[13] P. H. Pinheiro and R. Collobert. Recurrent convolutional neural networks for scene parsing. *arXiv preprint arXiv:1306.2795*, 2013.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 1:213, 2002.

[15] I. Sutskever. *Training recurrent neural networks*. PhD thesis, University of Toronto, 2013.

[16] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[17] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.