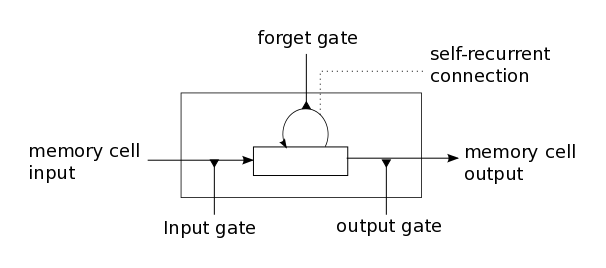
Long-Short Term Memory

Simple recurrent neural network computes output by taking previous output h and current input x. To solve vanishing gradient problem, usual LSTM cell along with input x and h previous hidden states, also takes m memory data from previous time step. Intuitively, LSTM learns when to remember and forget information.

LSTM model contains an input gate, a neuron with self-recurrent connection, a forget gate and an output gate.[[1]](#footnote-1) Self-recurrent connection is responsible for storing the information of previous time step. Input Gate alters the memory state of the current cell and output gate affects future cells. Forget Gate decides when to remember or forget the previous information. The computation is done as follows.

Figure 1: LSTM cell

gu = σ(Wu H)

gf =σ(Wf H)

go = σ(Wo H)

gc = tanh(Wc H)

m′ = gf ⊙ m + gu ⊙ gc

h′ = tanh(go ⊙ m′)

(1)

Where H = [Ix, h], h is the hidden output of previous cell, m is the memory output and W =[Wu, Wf, Wo, Wc] is the weight matrix to train. (σ - sigmoid function, tanh - tangent function and ⊙ component wise multiplication)

Variation of LSTM called Gated Recurrent Unit (GRU) was proposed that achieves slightly better performance. Using same fashion of connecting gates or introducing new inner cells, genetic algorithm generated three mutations (JZ1, JZ2 and JZ3), which outperform LSTM and GRU. For simplicity we will concentrate the scope of our project only on LSTM cells.

GridLSTM

Grid Long Short-Term Memory is a recurrent neural network of LSTM cells arranged in multidimensional grid that processes input such as vectors, images videos or MRI scans. Like multidimensional recurrent neural networks, model takes an input a data from the input dimension and hidden outputs of previous cells per each sequential dimension. At each step, each cell produces an output along each dimension for next steps including the final output.

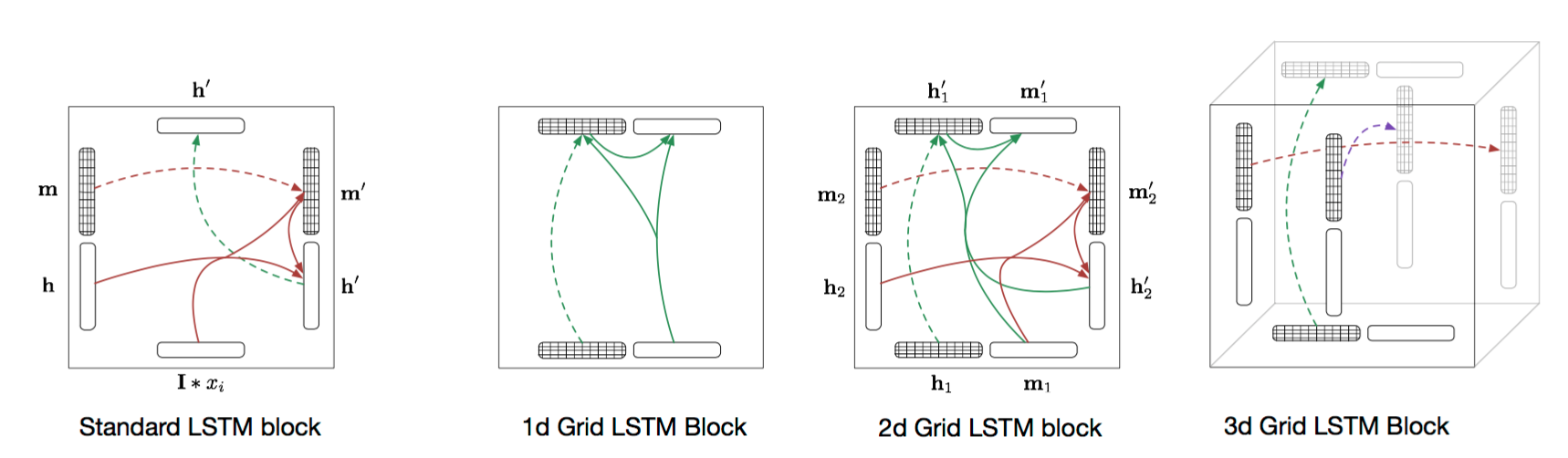


Figure 2: GridLSTM block

Unlike RNN or LSTM, GridLSTM cell called block takes from each dimension (h'k, m'k) (including input) and correspondingly processes outputs. For each k of n dimensions, block computes.

(h'k, m'k) = LSTM(H, mk, Wk)

(2)

Where H = [h1, h2, ... , hn], h is the hidden output, m is the memory output and W =[Wu, Wf, Wo, Wc] is the weight matrix to train. LSTM function is just the concatenation of formulae described in (1).

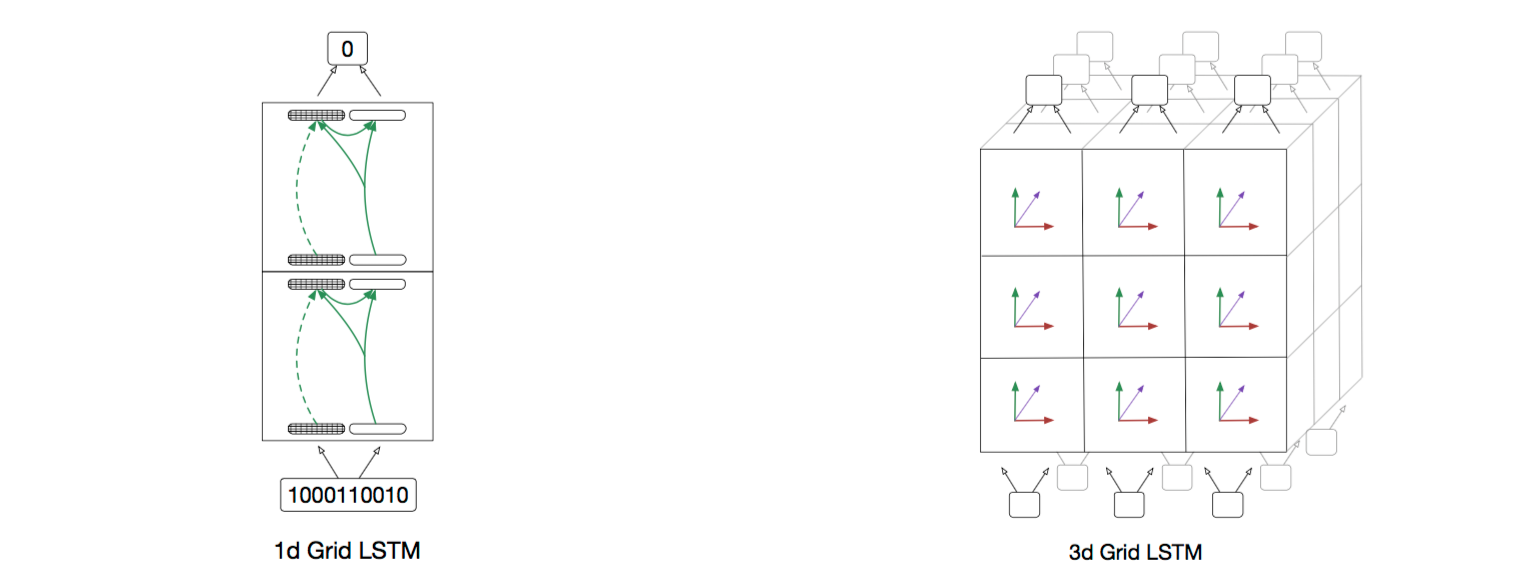


Figure 3: GridLSTM

Nal Kalchbrenner also introduced several additions to GridLSTM: priority dimensions and weight sharing. In order to take processed information outputs from other dimensions to include into priority dimension, the cell computes initially outputs of secondary dimensions and only then concatenates with input of primary dimension and forwards the data. The weights of LSTM along one dimension could be shared across different layers.

Implementation

Torch Framework

Torch deep learning framework is used to build and train the model. Any Neural Network can be implemented using Torch modules with nn and nngraph packages by creating computational graph. Custom modules extend nn.module and can be easily plugged into any architecture. For developing your own module, a computational node should be created by defining forward and backward passes including calculation of gradient of the input.

Recurrent neural networks are unfolded at each time step of input sequence. Thus, a clone of RNN cell is created in order to suit into nn.graph abstraction. In other words, RNN is abstracted into usual Neural Network that takes a sequence at once and outputs another sequence. Furthermore, there exists RNN package that takes the responsibility of cloning by introducing Sequencer decorator. Any RNN cell (including simple sigmoid cell, LSTM or GRU) extends AbstractRecurrent class and is decorated by Sequencer to fit into any NN architecture.

To extend AbstractRecurrent class in order to define custom module mainly three functions should defined

*buildModel*() - builds the computational graph/model

*updateOutput*(input) - forwards the input and returns the output

*updateGradOutput*(input, gradOutput) - proceeds the backward pass and returns the input gradient

In the following part our modular implementation of 3D Grid LSTM is discussed. It is based on Corey Lynch's 2D GridLSTM code of character language model that is in turn based on Andrej Karpathy's Char-RNN.

Build

As RNN model should be unfolded in Torch abstraction, 3D GridLSTM takes 6 inputs (mI, hI, mx, hx, my, hy) where first 2 vectors represent the input and the last 4 vectors represent hidden previous timesteps along x and y dimensions. Input sums of hs are correspondingly calculated and passed through three LSTM function to process outputs.

(houtput, m output) = LSTM([hI , hx , hy ], mI, WI)

(h'x, m'x) = LSTM([hI , hx , hy ],, mx, Wx)

(h'y, m'y) = LSTM([hI , hx , hy ],, my, Wy)

And in case of enabling priority dimensions the final output of the block is

(houtput, m output) = LSTM([hI , h'x , h'y ], mI, WI)

Tying the weights includes creating shared global variable and reusing it.

Mac:Users:david:Git:DepthRNN:docs:logs:architecture.pdf

Figure 4: Implementation architecture of GridLSTM build. where described the computational graph of nn package in torch.

Forward

In order to comply with RNN Torch abstraction, any input is treated as one-dimensional sequence with N dimensions reshaped to one. So, at any time step only one input is given with an order based on reshaping policy (For example for 2D case: row-wise or column-wise). Hidden previous states are zeroed at the boundaries of the image (or any other input) when there is no prior information available. During the forward pass, along with the input, the algorithm concatenates previous outputs of block along each dimension and feeds into the network. Using computational graph defined in Build section, Torch computes the output of the given input at the time step and stores all outputs while passing only main dimension to the next layer as the output of neural network. Stored outputs are retrieved for next time steps. Hence network remembers the previous input.

Backward

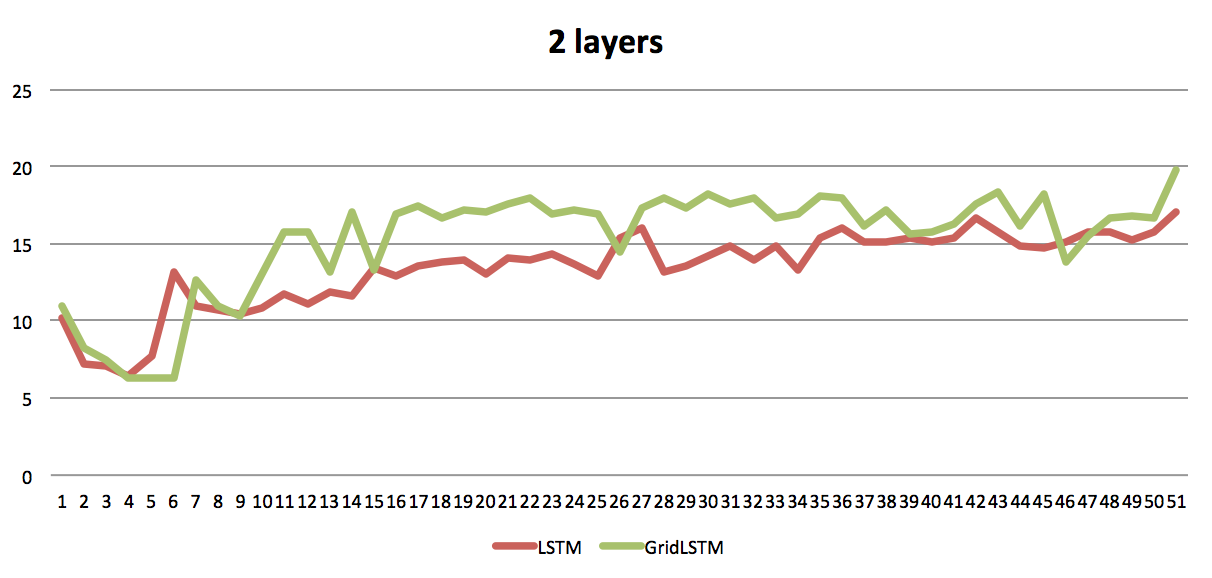
To train the network, Backpropogation algorithm is used. The *updateGradInput*() function is defined in order to estimate the gradient of the module with the respect to its input. We parse the sequence in reverse order, being output aware. In other words, the layer knows all its outputs for the given input as it assumes that forward pass has been already processed. The gradient of the output is already estimated by loss criterion, however gradients along inner dimensions are retrieved from previous gradient calculations (which are stored in *drnn-states*)*.* So, backpropogating the computational graph by providing the input and gradients with the respect to its output, we recieve the gradient with the respect to its input. The gradient of the module with the respect to its own parameters is also processed and finally the weights of GridLSTM are updated. And thus the network 'learns'.

Training

To train the network, each node of the network is back propagated and correspondingly the weights are updated. User defines the criterion of loss, which estimates how good is the prediction of the network based on the ground truth. There are built-in criterions such as binary cross-entropy, negative log-likelihood, absolute criterion, however one can define their own criterion as a module under Torch NN abstraction. For GridLSTM, as described in the paper, instead of using Stochastic Gradient Descent (SGD) learning algorithm, the module is usually trained using a stochastic optimization called Adam by taking advantage[[2]](#footnote-2) of effectiveness on sparse gradients and non-stationary objectives based on adaptive estimates of low-order moments[[3]](#footnote-3). The method of training and criterion varies for each type of problem.

Evaluation

In order to provide initial evidence that the model is actually working, initially prototype of 2D GridLSTM has been built for parsing 1D dimension data. We replicated an example in the paper that sums two numbers. The input of the network is a sequence of two 15-digits separated by GridLSTM. It outputs another sequence of 15 or 16 digits. For testing dataset random 100 numbers are generated and digit error is calculated. The mini-batch size 15 and the algorithm uses Adam optimization method with 0.001 learning rate. In the figure 5 and 6 below, a comparison of digit-wise accuracies is provided, where one can see that GridLSTM (green) slightly outperforms usual LSTM in stacked in 2 layers.



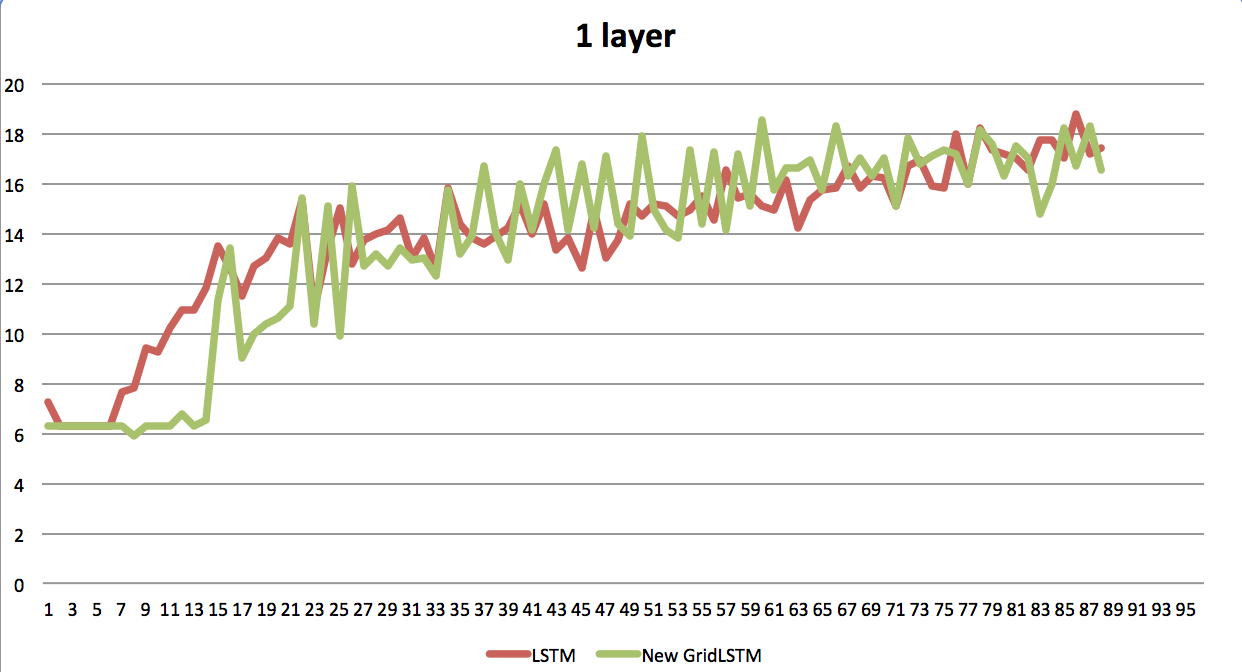


Figure 5,6: Comparison of 2D GridLSTM and LSTM on Sums digits.

Due to our computational limitations it is not sufficient to validate the model, as in order to reach 100% accuracy it is required to train the model for up to 5 million iterations.

Also we tried to compare the Lynch's available implementation on character level language modeling, however have been unable because of architecture differences of environments. Their model is solely based on nngraph and it will be very time consuming for reintegration of our model into their environment.

Another 'unit-test' for neural networks is considered to be over fitting the model as soon as possible and feed the same training set back to reach as good as possible. The given technique was used during the development of the model.

In conclusion, even though the lack of significant results that would describe evidence that the model is correct, we have been able to use validation techniques to provide sufficient evidence that the given network is actually learning and is better than simple LSTM cell in the given example. In order to validate the model, we will fully replicate digit classification for images.

Directional layers

As Recurrent Neural Networks parse a sequence iteratively, the hidden input and memory input from previous time steps contain only past information. In other words the RNN remembers only what it has seen. However, it is often the case when presented information has no actual beginning. It either was captured at once or contains information that is based on not only past context but also future context. We will describe directional layers, which overcome this limitation of RNN.

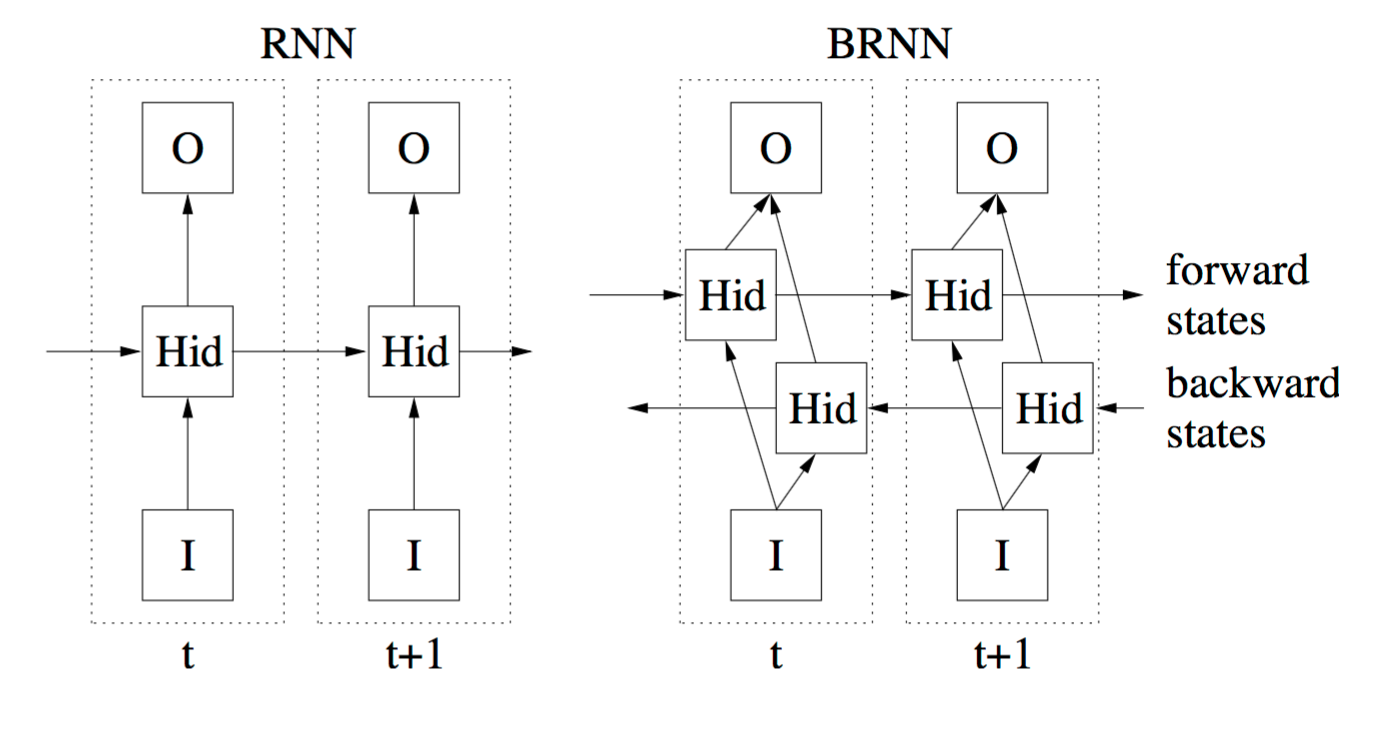
Bi-Directional

Figure 7: Comparison of simple Recurrent Neural Network and Bidirectional Recurrent Neural Network

The main motivation behind Bi-Directional RNN is that while making prediction at each time step, the output should be based not only on past, but also future context. Extending simple one directional recurrent neural network by introducing parallel layer that parses the input from negative direction solves the issue. As depicted in Figure 7, forward layer processes the information from left to right and backward layer processes the original sequence from right to left. Final composition happens on output layer by concatenating forward and backward outputs at each state.

Backward:add(nn.ReverseTable())

Backward:add(nn.RNN(...)))

Backward :add(nn.ReverseTable())

On implementation side, reversing the input sequence, applying usual RNN layer and then reversing back the output to the original order implements backward layer, which is processed simultaneously with original RNN layer. The final layer usually uses either sum or multiplication concatenation function to process the output.

Multi Directional

When the input is not one dimensional, a question arise how to treat the past/future context at any coordinate in space. Having the same intuition behind, Alex Grave at al introduced multidirectional RNN where multidimensional input is parsed from each corner (vertex) of the input.

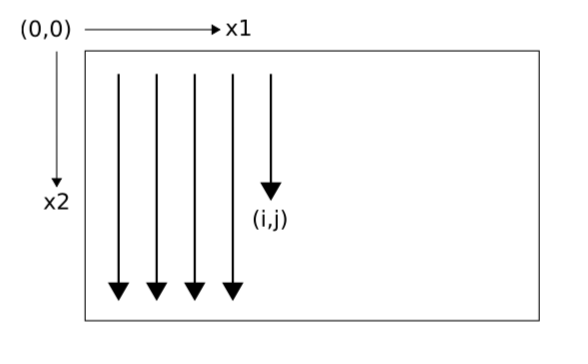
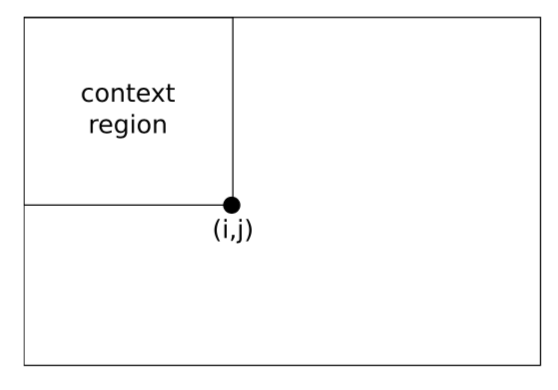


Figure 8: One directional parsing of an image (left) and available context at each timestep (right)

For N dimensional input, parallel 2N layers are added to the network each parsing the input sequence from each vertex. In Figure 9, a 2d image is shown where all contexts at each time step is available for producing the output.

As we simplify N dimensional input in our implementation of GridLSTM by reshaping into 1D sequence under AbstractRecurrent class, more complicated reversing algorithm should be proposed. For 2d images, simple layer is developed that outputs the symmetric image represented in one dimensions. This simulates top-right corner

Layer\_3:add(nn.SymmetricTable(width, height))

Layer\_3:add(nn.rnn(...))

Layer\_3:add(nn. SymmetricTable (width, height))

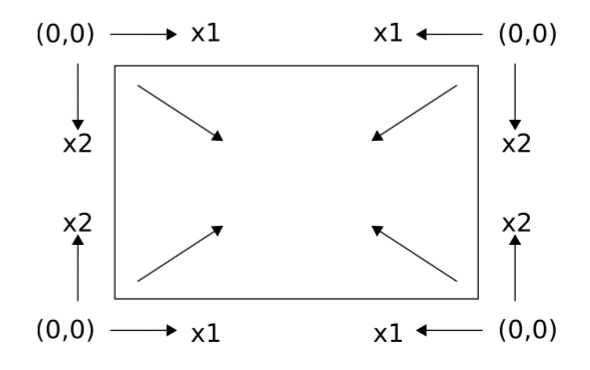
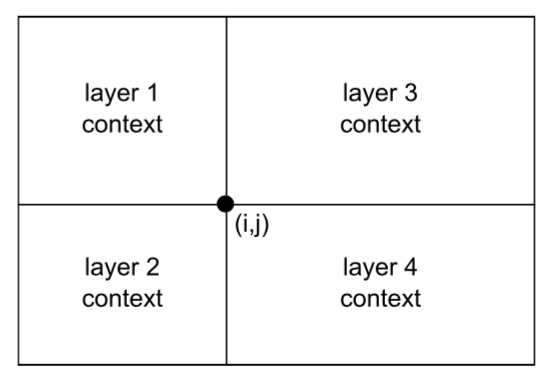


Figure 9: Multi directional parsing from each corner (left) and the available context at each timestep

Using the same fashion, we add ReverseTable layer, which will simulate the opposite left-bottom corner and in the result we will have all four layers for providing context-wise predication at each pixel. Later, it will be seen that actually in digit classification problem described in GridLSTM paper, instead of simultaneously using the layers, they are used in sequential order, which achieves better results in the given problem.

Digit Classifier

Dataset

// Make 100 words and add a picture

Model and Variants

// Make 300 words and add a picture

Results

// Make 100 words and add a picture

Evaluation

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

// Make 300 words and add a picture

Unlike Lynch's implementation, our implementation is modular. So, it could be plugged into any deep neural network by defining a GridLSTM within a single line.

nn.GridLSTM([parameters])

It is more optimized and 4x faster. Unlike directly processing depth dimension, our implementation only handles one layer such that user can define the number of sequential layers by adding nn.GridLSTM() into the Sequential module. This adds more flexibility, which will be seen in architecture section. In addition to that adding several loops could easily result in N dimensional grid.

Overall Goal: 2500 words

Figure 1,2: Simple recurrent neural network with one cell (left), Long Short-term memory (LSTM) cell (right). (Those figures will changed in future)

Figure 3: Variants of GridLSTM blocks

1. http://deeplearning.net/tutorial/lstm.html [↑](#footnote-ref-1)
2. https://moodle2.cs.huji.ac.il/nu15/pluginfile.php/316969/mod\_resource/content/1/adam\_pres.pdf [↑](#footnote-ref-2)
3. http://arxiv.org/pdf/1412.6980v8.pdf [↑](#footnote-ref-3)