Lab: CudaVision – Learning Vision Systems on Graphics Cards (MA-INF 4308)

Assignment 7

18.7.2016

Prof. Sven Behnke
Dr. Seongyong Koo

Recurrent Neural Networks

- **Goal: Understanding RNN and implementing MNIST sequential training model with LSTM**

- **RNN**

- **LSTN**

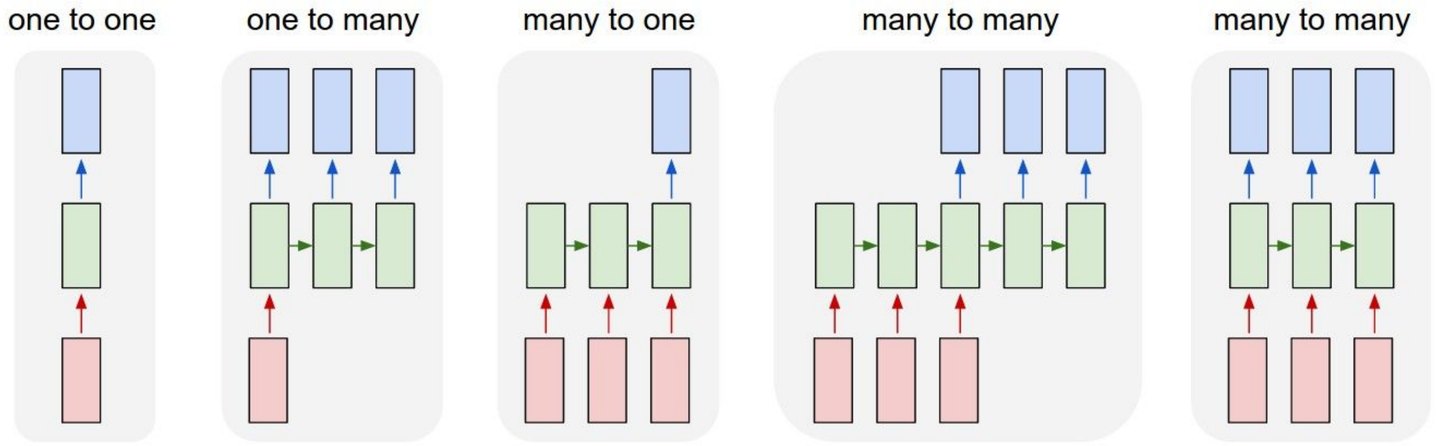
- **Sequentialization of a static MNIST image data**

- **Theoretical reference**
  - RNN in general: http://www.deeplearningbook.org/contents/rnn.html
  - LSTM: http://colah.github.io/posts/2015-08-Understanding-LSTMs/
  - Effectiveness of RNN: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Recurrent Neural Networks (RNNs)

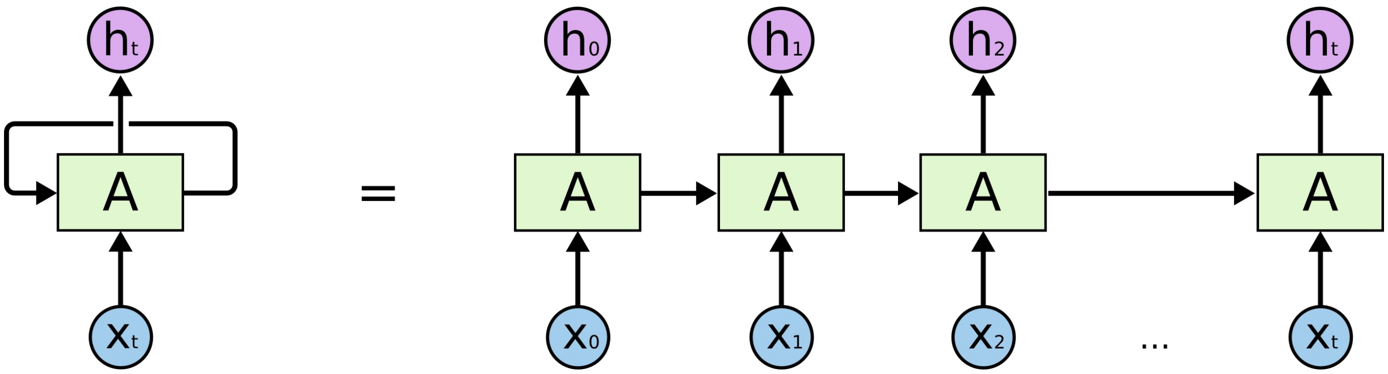
- Limitations of static NNs, e.g. CNN
  - Too much constrained: they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes).
  - These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model).

- Recurrent Neural Networks (RNN)s are a family of neural network for processing sequential data: Sequences in the input, the output, or in the most general case both.



Recurrent Neural Network (RNN)

- Advantages of sequential model
  - Much more powerful compared to fixed networks that are doomed from the get-go by a fixed number of computational steps,
  - Be able to connect previous information to the present task
  - RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector. This can in programming terms be interpreted as running a fixed program with certain inputs and some internal variables.

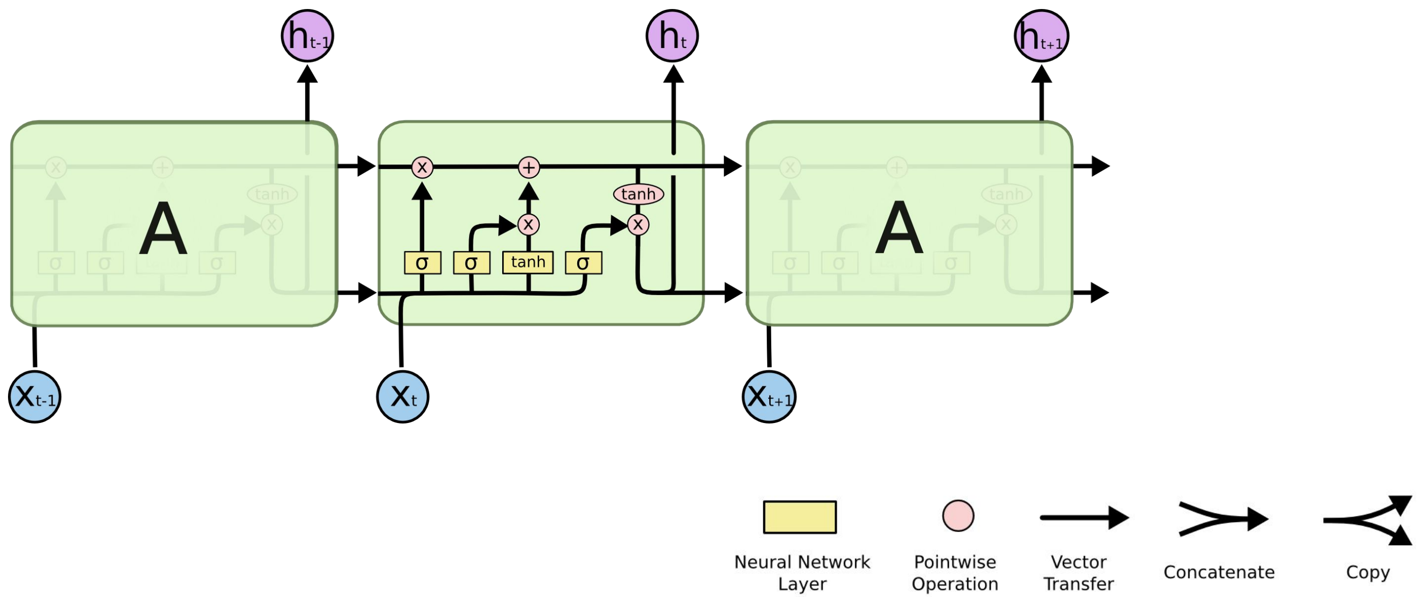


universitätbonn ais

Long Short Term Memory Networks (LSTM)

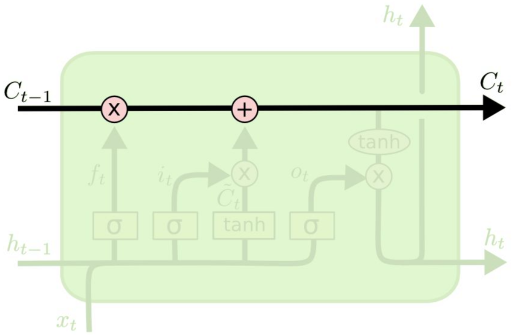
- Difficulties to train RNNs
  - As the sequence needed to train grows, RNNs become unable to learn to connect the information in the long term dependencies

- Long Short Term Memory Networks (LSTMs) don't have this problem
  - Introduced by Hochreiter & Schmidhuber (1997)
  - LSTMs are explicitly designed to avoid the long-term dependency problem by *remembering* information for long periods of time

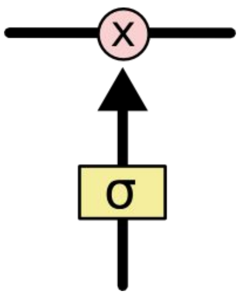


Core idea behind LSTMs

- Cell state
    - It runs straight down the entire chain, with only some minor linear interactions
    - The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates



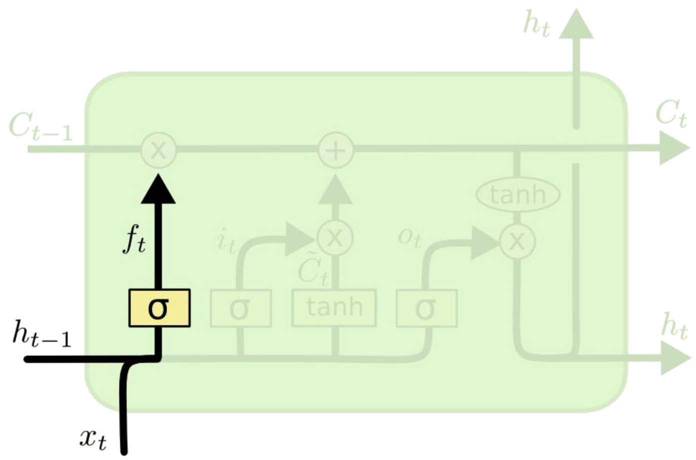
- Gates
    - They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.
    - The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.
    - An LSTM has three of these gates, to protect and control the cell state.



universitätbonn ais

Step-by-Step LSTM Walk Through

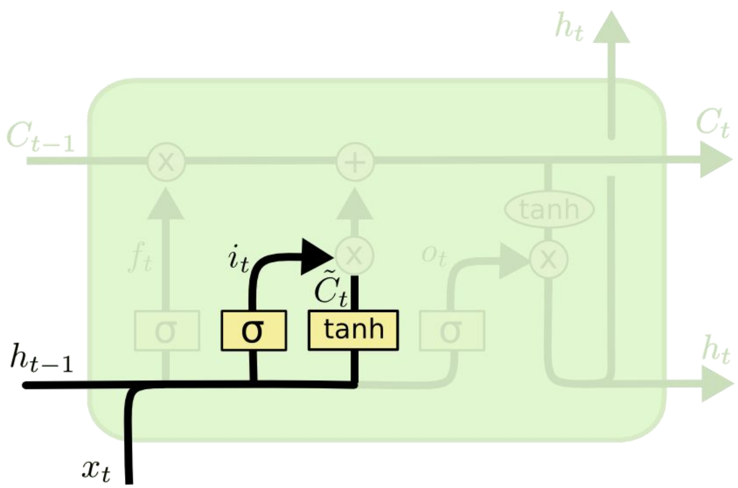
- Forget gate layer
  - To decide what information we're going to throw away from the cell state
  - It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Step-by-Step LSTM Walk Through

- Input gate layer
  - To decide what new information we're going to store in the cell state
  - First, a sigmoid layer decides which values we'll update.
  - Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state.
  - In the next step, we'll combine these two to create an update to the state.

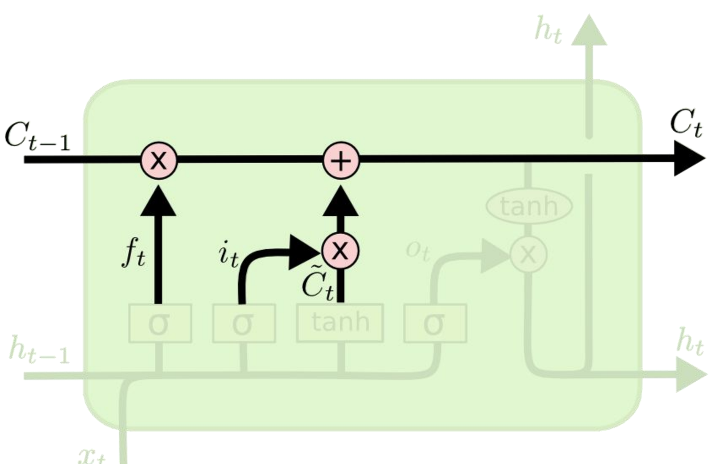


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Step-by-Step LSTM Walk Through

- Update cell state
  - Multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add it $i_t * \tilde{C}_t$
  - This is the new candidate values, scaled by how much we decided to update each state value.

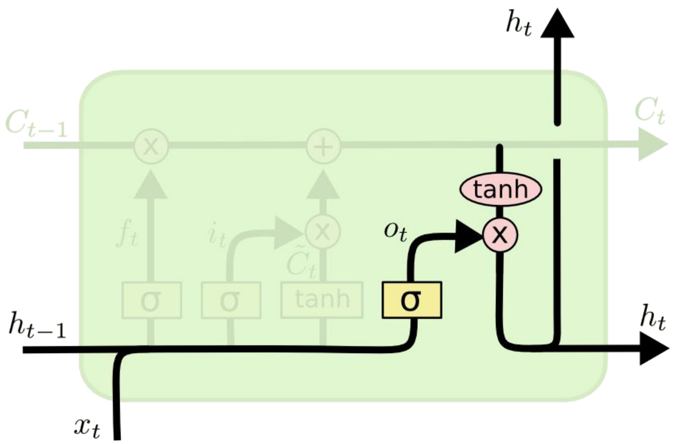


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

universitätbonn ais

Step-by-Step LSTM Walk Through

- Output
  - First, we run a sigmoid layer which decides what parts of the cell state we're going to output.
  - We put the cell state through tanh (to push the values to be between −1 and 1)
  - Multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



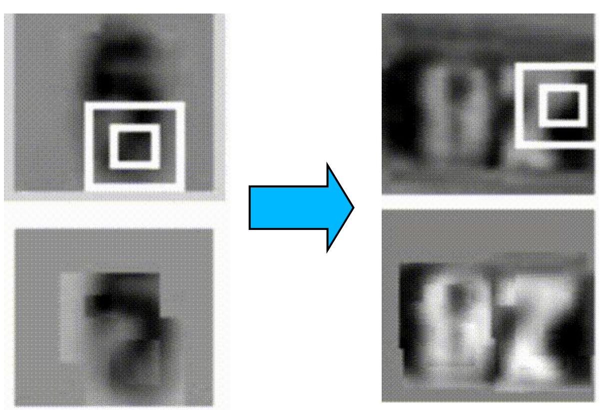
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

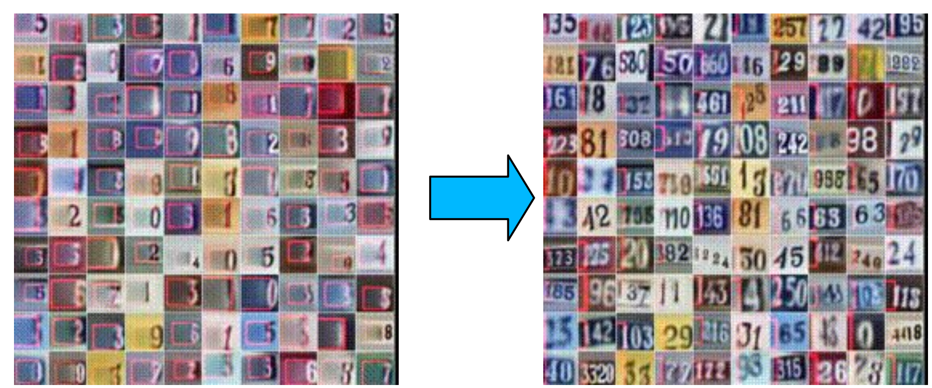
$$h_t = o_t * \tanh\left(C_t\right)$$

Sequential processing in absence of sequences

- Even if your inputs/outputs are fixed vectors, it is still possible to use this powerful formalism to process them in a sequential manner.

- For example,
    - An algorithm learns a recurrent network policy that steers its attention around an image; In particular, it learns to read out house numbers from left to right (Ba et al.).
    - A recurrent network generates images of digits by learning to sequentially add color to a canvas (Gregor et al.)

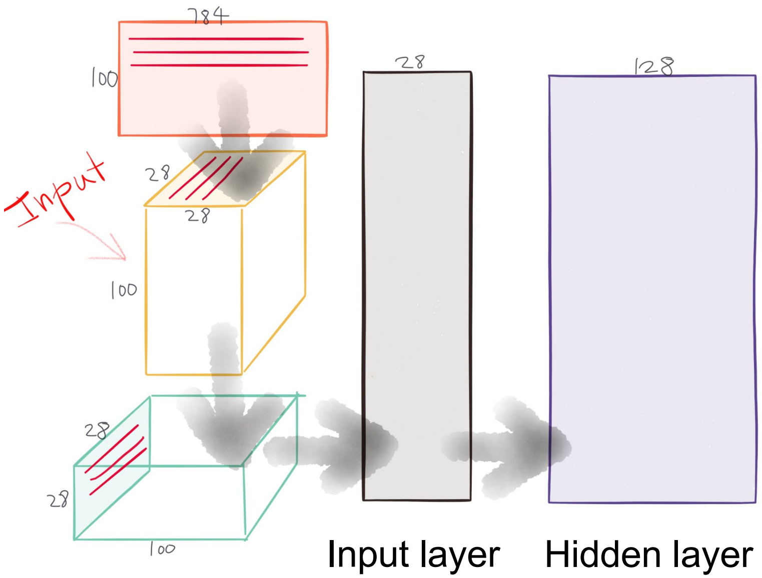


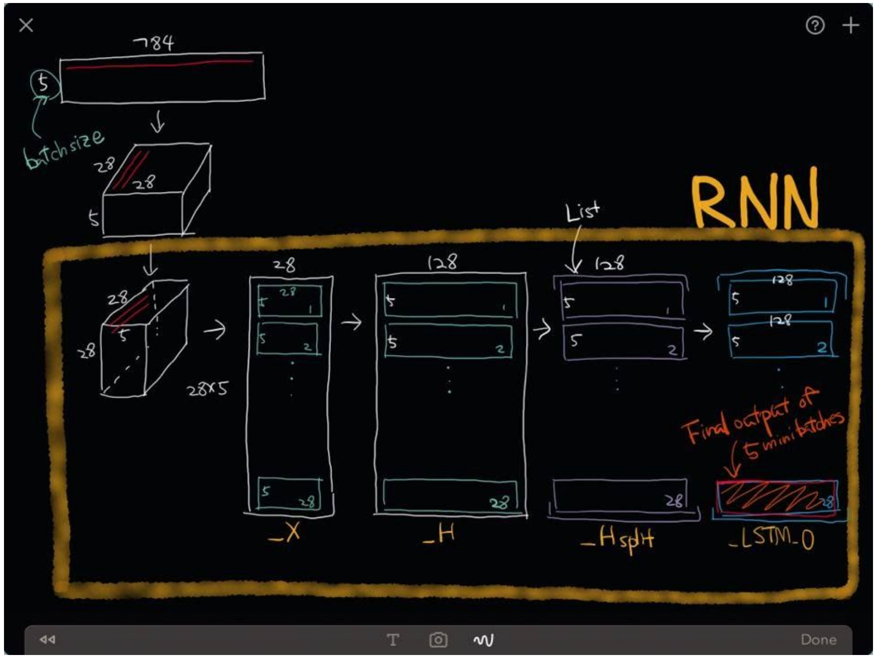


universitätbonn ais

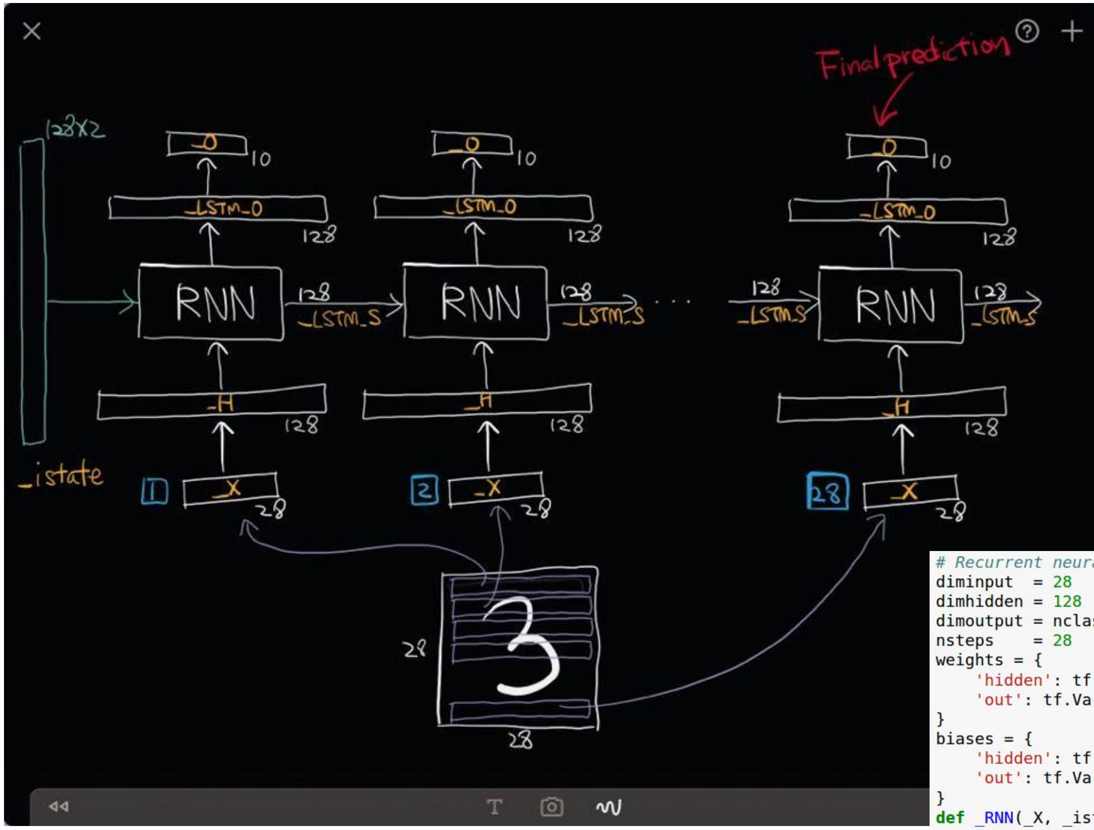
MNIST sequencial vector

- We will treat the 28x28 MNIST image as sequences of a 28x1 vector
- Our simple RNN consists of
  - one input layer which converts a 28-dimensional input to an 128-dimensional hidden layer,
  - one intermediate recurrent neural network (LSTM),
  - and one output layer which converts an 128-dimensional output of the LSTM to 10-dimensional output indicating a class label.





LSTM model




```python
# Recurrent neural network
diminput  = 28
dimhidden = 128
dimoutput = nclasses
nsteps    = 28
weights = {
    'hidden': tf.Variable(tf.random_normal([diminput, dimhidden])),
    'out': tf.Variable(tf.random_normal([dimhidden, dimoutput]))
}
biases = {
    'hidden': tf.Variable(tf.random_normal([dimhidden])),
    'out': tf.Variable(tf.random_normal([dimoutput]))
}
def _RNN(_X, _istate, _W, _b, _nsteps, _name):
    # 1. Permute input from [batchsize, nsteps, diminput] => [nsteps, batchsize, diminput]
    _X = tf.transpose(_X, [1, 0, 2])
    # 2. Reshape input to [nsteps*batchsize, diminput]
    _X = tf.reshape(_X, [-1, diminput])
    # 3. Input layer => Hidden layer
    _H = tf.matmul(_X, _W['hidden']) + _b['hidden']
    # 4. Splite data to 'nsteps' chunks. An i-th chunck indicates i-th batch data
    _Hsplit = tf.split(0, _nsteps, _H)
    # 5. Get LSTM's final output (_O) and state (_S)
    #    Both _O and _S consist of 'batchsize' elements
    with tf.variable_scope(_name):
        lstm_cell = rnn_cell.BasicLSTMCell(dimhidden, forget_bias=1.0)
        _LSTM_O, _LSTM_S = rnn.rnn(lstm_cell, _Hsplit, initial_state=_istate)
    # 6. Output
    _O = tf.matmul(_LSTM_O[-1], _W['out']) + _b['out']
    # Return!
    return {
        'X': _X, 'H': _H, 'Hsplit': _Hsplit,
        'LSTM_O': _LSTM_O, 'LSTM_S': _LSTM_S, 'O': _O
    }
```

Assignment 7

- **Complete implementation of MNIST training with LSTM model**
  - Use the provided RNN model
  - Training with initialized weights and zero 2x128 initial states (state & cell)

- **Test accuracy with smaller number of steps**
  - 28 steps mean using full image (28 x 28)
  - You can use smaller steps to predict test image label. It means the trained LSTM model works even the input image is truncated with a certain level of accuracy.
  - Show the accuracy drop-out according to the number of steps.